Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL-95/11 - Revision 2.1.5

PETSc Users Manual

by

Satish Balay
Kris Buschelman
William Gropp
Dinesh Kaushik
Matt Knepley
Lois Curfman McInnes
Barry Smith
Hong Zhang

Mathematics and Computer Science Division
http://www.mcs.anl.gov/petsc

This manual is intended for use with PETSc 2.1.5

January 27, 2003

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Abstract:

This manual describes the use of PETSc for the numerical solution of partial differential equations and related problems on high-performance computers. The Portable, Extensible Toolkit for Scientific Computation (PETSc) is a suite of data structures and routines that provide the building blocks for the implementation of large-scale application codes on parallel (and serial) computers. PETSc uses the MPI standard for all message-passing communication.

PETSc includes an expanding suite of parallel linear, nonlinear equation solvers and time integrators that may be used in application codes written in Fortran, C, and C++. PETSc provides many of the mechanisms needed within parallel application codes, such as parallel matrix and vector assembly routines. The library is organized hierarchically, enabling users to employ the level of abstraction that is most appropriate for a particular problem. By using techniques of object-oriented programming, PETSc provides enormous flexibility for users.

PETSc is a sophisticated set of software tools; as such, for some users it initially has a much steeper learning curve than a simple subroutine library. In particular, for individuals without some computer science background or experience programming in C or C++, it may require a significant amount of time to take full advantage of the features that enable efficient software use. However, the power of the PETSc design and the algorithms it incorporates may make the efficient implementation of many application codes simpler than "rolling them" yourself.

- For many simple (or even relatively complicated) tasks a package such as Matlab is often the best tool; PETSc is not intended for the classes of problems for which effective Matlab code can be written.

- PETSc should not be used to attempt to provide a "**parallel linear solver**" in an otherwise sequential code. Certainly all parts of a previously sequential code need not be parallelized but the matrix generation portion must be to expect any kind of reasonable performance. Do not expect to generate your matrix sequentially and then "use PETSc" to solve the linear system in parallel.

Since PETSc is under continued development, small changes in usage and calling sequences of routines may occur. PETSc is supported; see the web site `http://www.mcs.anl.gov/petsc` for information on contacting support.

A list of publications and web sites that feature work involving PETSc may be found at `http://www.mcs.anl.gov/petsc/publications`. We welcome any additions to these pages.

Getting Information on PETSc:

On-line:

- Manual pages for all routines, including example usage `docs/index.html` in the distribution or `http://www.mcs.anl.gov/petsc/docs/`
- Troubleshooting `docs/troubleshooting.html` in the distribution or `http://www.mcs.anl.gov/petsc/docs/`

In this manual:

- Basic introduction, page 14

Acknowledgments:

We thank all PETSc users for their many suggestions, bug reports, and encouragement. We especially thank Victor Eijkhout and David Keyes for their valuable comments on the source code, functionality, and documentation for PETSc.

Some of the source code and utilities in PETSc have been written by

- Mark Adams - scalability features of MPIBAIJ matrices;
- Allison Baker - the flexible GMRES code;
- Tony Caola - the SPARSEKIT2 ilutp() interface;
- Chad Carroll - Win32 graphics;
- Cameron Cooper - portions of the VecScatter routines;
- Victor Eijkhout;
- Paulo Goldfeld - balancing Neumann-Neumann preconditioner;
- Matt Hille;
- Domenico Lahaye - the interface to John Ruge and Klaus Stueben's AMG;
- Peter Mell - portions of the DA routines;
- Todd Munson - the LUSOL (sparse solver in MINOS) interface;
- Adam Powell - the PETSc Debian package,
- Robert Scheichl - the MINRES implementation,
- Liyang Xu - the interface to PVODE;

PETSc uses routines from

- BLAS;
- LAPACK;
- LINPACK - dense matrix factorization and solve; converted to C using `f2c` and then hand-optimized for small matrix sizes, for block matrix data structures;
- MINPACK - see page 95, sequential matrix coloring routines for finite difference Jacobian evaluations; converted to C using `f2c`;
- SPARSPAK - see page 70, matrix reordering routines, converted to C using `f2c`;
- SPARSEKIT2 - see page 68, written by Yousef Saad, iludtp(), converted to C using `f2c`; These routines are copyrighted by Saad under the GNU copyright, see `${PETSC_DIR}/src/mat/impls/aij/seq/ilut.c`.

- libtfs - the efficient, parallel direct solver developed by Henry Tufo and Paul Fischer for the direct solution of a coarse grid problem (a linear system with very few degrees of freedom per processor).

PETSc interfaces to the following external software:

- ADIC/ADIFOR - automatic differentiation for the computation of sparse Jacobians, `http://www.mcs.anl.gov/adic` , `http://www.mcs.anl.gov/adifor` ,
- AMG - the algebraic multigrid code of John Ruge and Klaus Stueben, `http://www.mgnet.org/mgnet-codes-gmd.html`
- BlockSolve95 - see page 68, for parallel ICC(0) and ILU(0) preconditioning, `http://www.mcs.anl.gov/blocksolve` ,
- DSCPACK - see page 76, Domain-Separator Codes for solving sparse symmetric positive-definite systems, developed by Padma Raghavan, `http://www.cse.psu.edu/~raghavan/Dscpack/` ,
- ESSL - IBM's math library for fast sparse direct LU factorization,
- Euclid - parallel ILU(k) developed by David Hysom, accessed through the Hypre interface,
- Hypre - the LLNL preconditioner library, `http://www.llnl.gov/CASC/hypre`
- LUSOL - sparse LU factorization code (part of MINOS) developed by Michael Saunders, Systems Optimization Laboratory, Stanford University, `http://www.sbsi-sol-optimize.com/` ,
- Mathematica - see page **??**,
- Matlab - see page 107,
- ParMeTiS - see page 58, parallel graph partitioner, `http://www-users.cs.umn.edu/~karypis/metis/` ,
- PVODE - see page 100, parallel ODE integrator, `http://www.llnl.gov/CASC/PVODE` ,
- SPAI - for parallel sparse approximate inverse preconditiong, `http://www.sam.math.ethz.ch/~grote/spai/` ,
- SPOOLES - see page 76, SParse Object Oriented Linear Equations Solver, developed by Cleve Ashcraft, `http://www.netlib.org/linalg/spooles/spooles.2.2.html` ,
- SuperLU and SuperLU_Dist - see page 76, the efficient sparse LU codes developed by Jim Demmel, Xiaoye S. Li, and John Gilbert, `http://www.nersc.gov/~xiaoye/SuperLU` .

These are all optional packages and do not need to be installed to use PETSc.

PETSc software is developed and maintained with

- Bitkeeper revision control system
- Emacs editor

PETSc documentation has been generated using

- the text processing tools developed by Bill Gropp

- c2html
- Microsoft Frontpage
- pdflatex
- python

Contents

Part I

Introduction to PETSc

Chapter 1

Getting Started

The Portable, Extensible Toolkit for Scientific Computation (PETSc) has successfully demonstrated that the use of modern programming paradigms can ease the development of large-scale scientific application codes in Fortran, C, and C++. Begun several years ago, the software has evolved into a powerful set of tools for the numerical solution of partial differential equations and related problems on high-performance computers. PETSc consists of a variety of libraries (similar to classes in C++), which are discussed in detail in Parts II and III of the users manual. Each library manipulates a particular family of objects (for instance, vectors) and the operations one would like to perform on the objects. The objects and operations in PETSc are derived from our long experiences with scientific computation. Some of the PETSc modules deal with

- index sets, including permutations, for indexing into vectors, renumbering, etc;
- vectors;
- matrices (generally sparse);
- distributed arrays (useful for parallelizing regular grid-based problems);
- Krylov subspace methods;
- preconditioners, including multigrid and sparse direct solvers;
- nonlinear solvers; and
- timesteppers for solving time-dependent (nonlinear) PDEs.

Each consists of an abstract interface (simply a set of calling sequences) and one or more implementations using particular data structures. Thus, PETSc provides clean and effective codes for the various phases of solving PDEs, with a uniform approach for each class of problems. This design enables easy comparison and use of different algorithms (for example, to experiment with different Krylov subspace methods, preconditioners, or truncated Newton methods). Hence, PETSc provides a rich environment for modeling scientific applications as well as for rapid algorithm design and prototyping. The libraries enable easy customization and extension of both algorithms and implementations. This approach promotes code reuse and flexibility, and separates the issues of parallelism from the choice of algorithms. The PETSc infrastructure creates a foundation for building large-scale applications. It is useful to consider the interrelationships among different pieces of PETSc. Figure 1 is a diagram of some of these pieces; Figure 2 presents several of the individual parts in more detail. These figures illustrate the library's hierarchical organization, which enables users to employ the level of abstraction that is most appropriate for a particular problem.

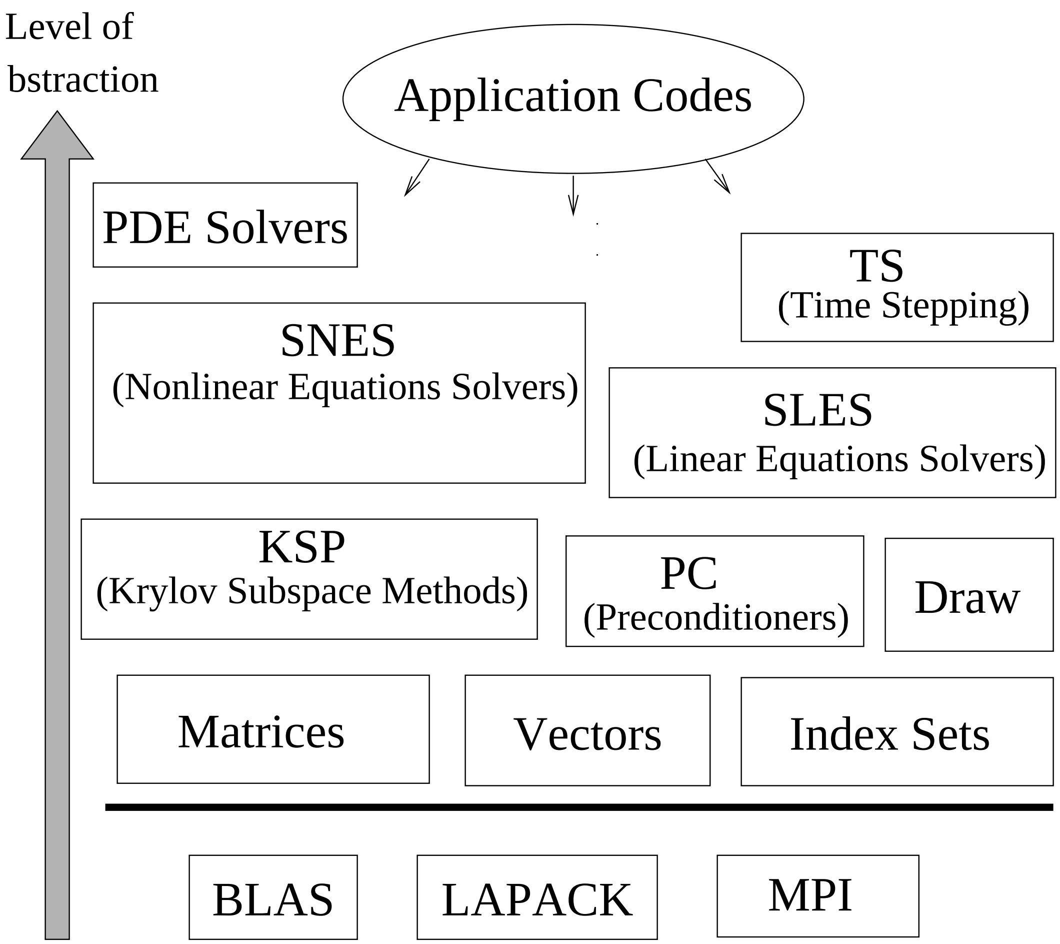

Figure 1: Organization of the PETSc Libraries

1.1 Suggested Reading

The manual is divided into three parts:

- Part I - Introduction to PETSc
- Part II - Programming with PETSc
- Part III - Additional Information

Part I describes the basic procedure for using the PETSc library and presents two simple examples of solving linear systems with PETSc. This section conveys the typical style used throughout the library and enables the application programmer to begin using the software immediately. Part I is also distributed separately for individuals interested in an overview of the PETSc software, excluding the details of library usage. Readers of this separate distribution of Part I should note that all references within the text to particular chapters and sections indicate locations in the complete users manual. Part II explains in detail the use of the various PETSc libraries, such as vectors, matrices, index sets, linear and nonlinear solvers, and graphics. Part III describes a variety of useful information, including profiling, the options database, viewers, error handling, makefiles, and some details of PETSc design. The *PETSc Users Manual* documents *all* of PETSc; thus, it can be rather intimidating for new users. We recommend that one initially read the entire document before proceeding with serious use of PETSc, but bear in mind that PETSc can be used efficiently before one understands all of the material presented here.

Within the PETSc distribution, the directory `${PETSC_DIR}/docs` contains all documentation. Manual pages for all PETSc functions can be accessed on line at

http://www.mcs.anl.gov/petsc/docs/

The manual pages provide hyperlinked indices (organized by both concepts and routine names) to the tutorial examples and enable easy movement among related topics.

Emacs users may find the *etags* option to be extremely useful for exploring the PETSc source code. Details of this feature are provided in Section 14.7.

The file `manual.pdf` contains the complete *PETSc Users Manual* in the portable document format (PDF), while `intro.pdf` includes only the introductory segment, Part I. The complete PETSc distribution, users manual, manual pages, and additional information are also available via the PETSc home page at `http://www.mcs.anl.gov/petsc` . The PETSc home page also contains details regarding installation, new features and changes in recent versions of PETSc, machines that we currently support, a troubleshooting guide, and a FAQ list for frequently asked questions.

Note to Fortran Programmers: In most of the manual, the examples and calling sequences are given for the

C/C++ family of programming languages. We follow this convention because we recommend that PETSc applications be coded in C or C++. However, pure Fortran programmers can use most of the functionality of PETSc from Fortran, with only minor differences in the user interface. Chapter 11 provides a discussion of the differences between using PETSc from Fortran and C, as well as several complete Fortran examples. This chapter also introduces some routines that support direct use of Fortran90 pointers.

1.2 Running PETSc Programs

Before using PETSc, the user must first set the environmental variable `PETSC_DIR`, indicating the full path of the PETSc home directory. For example, under the UNIX C shell a command of the form

setenv PETSC_DIR $HOME/petsc

can be placed in the user's `.cshrc` file. In addition, the user must set the environmental variable `PETSC_ARCH` to specify the architecture (e.g., rs6000, solaris, IRIX, etc.) on which PETSc is being used. The utility `${PETSC_DIR}/bin/petscarch` can be used for this purpose. For example,

setenv PETSC_ARCH ‘$PETSC_DIR/bin/petscarch‘

can be placed in a `.cshrc` file. Thus, even if several machines of different types share the same filesystem, `PETSC_ARCH` will be set correctly when logging into any of them.

All PETSc programs use the MPI (Message Passing Interface) standard for message-passing communication [15]. Thus, to execute PETSc programs, users must know the procedure for beginning MPI jobs on their selected computer system(s). For instance, when using the MPICH implementation of MPI [9] and many others, the following command initiates a program that uses eight processors:

`mpirun` -np 8 petsc_program_name petsc_options

PETSc also comes with a script

$PETSC_DIR/bin/petscmpirun -np 8 petsc_program_name petsc_options

that uses the information set in `${PETSC_DIR}/bmake/${PETSC_ARCH}/packages` to automatically use the correct `mpirun` for your configuration. All PETSc-compliant programs support the use of the `-h` or `-help` option as well as the `-v` or `-version` option.

Certain options are supported by all PETSc programs. We list a few particularly useful ones below; a complete list can be obtained by running any PETSc program with the option `-help`.

- `-log_summary` - summarize the program's performance
- `-fp_trap` - stop on floating-point exceptions; for example divide by zero
- `-trdump` - enable memory tracing; dump list of unfreed memory at conclusion of the run
- `-trmalloc` - enable memory tracing (by default this is activated for the debugging versions of PETSc)
- `-start_in_debugger [noxterm,gdb,dbx,xxgdb] [-display name]` - start all processes in debugger
- `-on_error_attach_debugger [noxterm,gdb,dbx,xxgdb] [-display name]` - start debugger only on encountering an error

See Section 14.3 for more information on debugging PETSc programs.

1.3 Writing PETSc Programs

Most PETSc programs begin with a call to

`PetscInitialize` (int *argc,char ***argv,char *file,char *help);

which initializes PETSc and MPI. The arguments `argc` and `argv` are the command line arguments delivered in all C and C++ programs. The argument `file` optionally indicates an alternative name for the PETSc options file, `.petscrc`, which resides by default in the user's home directory. Section 14.1 provides details regarding this file and the PETSc options database, which can be used for runtime customization. The final argument, `help`, is an optional character string that will be printed if the program is run with the `-help` option. In Fortran the initialization command has the form

call `PetscInitialize` (character file,integer ierr)

`PetscInitialize()` automatically calls `MPI_Init()` if MPI has not been not previously initialized. In certain circumstances in which MPI needs to be initialized directly (or is initialized by some other library), the user can first call `MPI_Init()` (or have the other library do it), and then call `PetscInitialize()`. By default, `PetscInitialize()` sets the PETSc "world" communicator, given by `PETSC_COMM_WORLD`, to `MPI_COMM_WORLD`. For those not familar with MPI, a *communicator* is a way of indicating a collection of processes that will be involved together in a calculation or communication. Communicators have the variable type `MPI_Comm`. In most cases users can employ the communicator `PETSC_COMM_WORLD` to indicate all processes in a given run and `PETSC_COMM_SELF` to indicate a single process. MPI provides routines for generating new communicators consisting of subsets of processors, though most users rarely need to use these. The book *Using MPI*, by Lusk, Gropp, and Skjellum [10] provides an excellent introduction to the concepts in MPI, see also the MPI homepage `http://www.mcs.anl.gov/mpi/`. Note that PETSc users need not program much message passing directly with MPI, but they must be familar with the basic concepts of message passing and distributed memory computing. Users who wish to employ PETSc routines on only a subset of processors within a larger parallel job, or who wish to use a "master" process to coordinate the work of "slave" PETSc processes, should specify an alternative communicator for `PETSC_COMM_WORLD` by calling

`PetscSetCommWorld` (MPI_Comm `comm`);

before calling `PetscInitialize()`, but, obviously, after calling `MPI_Init()`. `PetscSetCommWorld()` can be called at most once per process. Most users will never need to use the routine `PetscSetCommWorld()`. All PETSc routines return an integer indicating whether an error has occurred during the call. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For the C/C++ interface, the error variable is the routine's return value, while for the Fortran version, each PETSc routine has as its final argument an integer error variable. Error tracebacks are discussed in the following section. All PETSc programs should call `PetscFinalize()` as their final (or nearly final) statement, as given below in the C/C++ and Fortran formats, respectively:

`PetscFinalize` ();
call `PetscFinalize` (ierr)

This routine handles options to be called at the conclusion of the program, and calls `MPI_Finalize()` if `PetscInitialize()` began MPI. If MPI was initiated externally from PETSc (by either the user or another software package), the user is responsible for calling `MPI_Finalize()`.

1.4 Simple PETSc Examples

To help the user start using PETSc immediately, we begin with a simple uniprocessor example in Figure 3 that solves the one-dimensional Laplacian problem with finite differences. This sequential code, which can be found in `${PETSC_DIR}/src/sles/examples/tutorials/ex1.c`, illustrates the solution of a linear system with `SLES` , the interface to the preconditioners, Krylov subspace methods, and direct linear solvers of PETSc. Following the code we highlight a few of the most important parts of this example.

```
/*$Id: ex1.c,v 1.90 2001/08/07 21:30:54 bsmith Exp $*/

/* Program usage:  mpirun ex1 [-help] [all PETSc options] */

static char help[] = "Solves a tridiagonal linear system with SLES.\n\n";

/*T
   Concepts: SLES^solving a system of linear equations
   Processors: 1
T*/

/*
  Include "petscsles.h" so that we can use SLES solvers.  Note that this file
  automatically includes:
     petsc.h       - base PETSc routines   petscvec.h - vectors
     petscsys.h    - system routines       petscmat.h - matrices
     petscis.h     - index sets            petscksp.h - Krylov subspace methods
     petscviewer.h - viewers               petscpc.h  - preconditioners

  Note:  The corresponding parallel example is ex23.c
*/
#include "petscsles.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  Vec         x, b, u;      /* approx solution, RHS, exact solution */
  Mat         A;            /* linear system matrix */
  SLES        sles;         /* linear solver context */
  PC          pc;           /* preconditioner context */
```

```
KSP         ksp;          /* Krylov subspace method context */
PetscReal   norm;         /* norm of solution error */
int         ierr,i,n = 10,col[3],its,size;
PetscScalar neg_one = -1.0,one = 1.0,value[3];

PetscInitialize(&argc,&args,(char *)0,help);
ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");
ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       Compute the matrix and right-hand-side vector that define
       the linear system, Ax = b.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

/*
   Create vectors.  Note that we form 1 vector from scratch and
   then duplicate as needed.
*/
ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject) x, "Solution");CHKERRQ(ierr);
ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
ierr = VecSetFromOptions(x);CHKERRQ(ierr);
ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
ierr = VecDuplicate(x,&u);CHKERRQ(ierr);

/*
   Create matrix.  When using MatCreate(), the matrix format can
   be specified at runtime.

   Performance tuning note:  For problems of substantial size,
   preallocation of matrix memory is crucial for attaining good
   performance.  Since preallocation is not possible via the generic
   matrix creation routine MatCreate(), we recommend for practical
   problems instead to use the creation routine for a particular matrix
   format, e.g.,
       MatCreateSeqAIJ() - sequential AIJ (compressed sparse row)
       MatCreateSeqBAIJ() - block AIJ
   See the matrix chapter of the users manual for details.
*/
ierr = MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,n,n,&A);CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);

/*
   Assemble matrix
*/
value[0] = -1.0; value[1] = 2.0; value[2] = -1.0;
for (i=1; i<n-1; i++) {
  col[0] = i-1; col[1] = i; col[2] = i+1;
  ierr = MatSetValues(A,1,&i,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
}
i = n - 1; col[0] = n - 2; col[1] = n - 1;
ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
i = 0; col[0] = 0; col[1] = 1; value[0] = 2.0; value[1] = -1.0;
ierr = MatSetValues(A,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

/*
   Set exact solution; then compute right-hand-side vector.
```

```
*/
ierr = VecSet(&one,u);CHKERRQ(ierr);
ierr = MatMult(A,u,b);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
              Create the linear solver and set various options
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
   Create linear solver context
*/
ierr = SLESCreate(PETSC_COMM_WORLD,&sles);CHKERRQ(ierr);

/*
   Set operators. Here the matrix that defines the linear system
   also serves as the preconditioning matrix.
*/
ierr = SLESSetOperators(sles,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);

/*
   Set linear solver defaults for this problem (optional).
   - By extracting the KSP and PC contexts from the SLES context,
     we can then directly call any KSP and PC routines to set
     various options.
   - The following four statements are optional; all of these
     parameters could alternatively be specified at runtime via
     SLESSetFromOptions();
*/
ierr = SLESGetKSP(sles,&ksp);CHKERRQ(ierr);
ierr = SLESGetPC(sles,&pc);CHKERRQ(ierr);
ierr = PCSetType(pc,PCJACOBI);CHKERRQ(ierr);
ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,PETSC_DEFAULT,PETSC_DEFAULT);CHKERRQ(ierr);

/*
  Set runtime options, e.g.,
      -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <rtol>
  These options will override those specified above as long as
  SLESSetFromOptions() is called _after_ any other customization
  routines.
*/
ierr = SLESSetFromOptions(sles);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                    Solve the linear system
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
   Solve linear system
*/
ierr = SLESSolve(sles,b,x,&its);CHKERRQ(ierr);

/*
   View solver info; we could instead use the option -sles_view to
   print this info to the screen at the conclusion of SLESSolve().
*/
ierr = SLESView(sles,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                    Check solution and clean up
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
   Check the error
```

```
  */
  ierr = VecAXPY(&neg_one,u,x);CHKERRQ(ierr);
  ierr  = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %A, Iterations %d\n",norm,its);CHKERRQ(ierr);
  /*
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
  */
  ierr = VecDestroy(x);CHKERRQ(ierr); ierr = VecDestroy(u);CHKERRQ(ierr);
  ierr = VecDestroy(b);CHKERRQ(ierr); ierr = MatDestroy(A);CHKERRQ(ierr);
  ierr = SLESDestroy(sles);CHKERRQ(ierr);

  /*
     Always call PetscFinalize() before exiting a program.  This routine
       - finalizes the PETSc libraries as well as MPI
       - provides summary and diagnostic information if certain runtime
         options are chosen (e.g., -log_summary).
  */
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

Figure 3: Example of Uniprocessor PETSc Code

Include Files

The C/C++ include files for PETSc should be used via statements such as

#include ”petscsles.h”

where `petscsles.h` is the include file for the `SLES` library. Each PETSc program must specify an include file that corresponds to the highest level PETSc objects needed within the program; all of the required lower level include files are automatically included within the higher level files. For example, `petscsles.h` includes `petscmat.h` (matrices), `petscvec.h` (vectors), and `petsc.h` (base PETSc file). The PETSc include files are located in the directory `${PETSC_DIR}/include`. See Section 11.1.1 for a discussion of PETSc include files in Fortran programs.

The Options Database

As shown in Figure 3, the user can input control data at run time using the options database. In this example the command `OptionsGetInt(PETSC_NULL,"-n",&n,&flg);` checks whether the user has provided a command line option to set the value of `n`, the problem dimension. If so, the variable `n` is set accordingly; otherwise, `n` remains unchanged. A complete description of the options database may be found in Section 14.1.

Vectors

One creates a new parallel or sequential vector, `x`, of global dimension `M` with the commands

`VecCreate` (MPI_Comm `comm` ,`Vec` *x); `VecSetSizes` (`Vec` x, int m, int M);

where `comm` denotes the MPI communicator and `m` is the optional local `size` which may be `PETSC_DECIDE`. The type of storage for the vector may be set with either calls to `VecSetType()` or `VecSetFromOptions()`. Additional vectors of the same type can be formed with

`VecDuplicate` (`Vec` old,`Vec` *new);

The commands

VecSet(PetscScalar *value,Vec x);
VecSetValues(Vec x,int n,int *indices,PetscScalar *values,INSERT_VALUES);

respectively set all the components of a vector to a particular scalar value and assign a different value to each component. More detailed information about PETSc vectors, including their basic operations, scattering/gathering, index sets, and distributed arrays, is discussed in Chapter 2. Note the use of the PETSc variable type `PetscScalar` in this example. The `PetscScalar` is simply defined to be `double` in C/C++ (or correspondingly `double precision` in Fortran) for versions of PETSc that have *not* been compiled for use with complex numbers. The `PetscScalar` data type enables identical code to be used when the PETSc libraries have been compiled for use with complex numbers. Section 14.6 discusses the use of complex numbers in PETSc programs.

Matrices

Usage of PETSc matrices and vectors is similar. The user can create a new parallel or sequential matrix, `A`, which has `M` global rows and `N` global columns, with the routine

MatCreate(MPI_Comm comm,int m,int n,int M,int N,Mat *A);

where the matrix format can be specified at runtime. The user could alternatively specify each processes' number of local rows and columns using `m` and `n`. Values can then be set with the command

MatSetValues(Mat A,int m,int *im,int n,int *in,PetscScalar *values,INSERT_VALUES);

After all elements have been inserted into the matrix, it must be processed with the pair of commands

MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);

Chapter 3 discusses various matrix formats as well as the details of some basic matrix manipulation routines.

Linear Solvers

After creating the matrix and vectors that define a linear system, `Ax = b`, the user can then use `SLES` to solve the system with the following sequence of commands:

SLESCreate(MPI_Comm comm,SLES *sles);
SLESSetOperators(SLES sles,Mat A,Mat PrecA,MatStructure flag);
SLESSetFromOptions(SLES sles);
SLESSolve(SLES sles,Vec b,Vec x,int *its);
SLESDestroy(SLES sles);

The user first creates the `SLES` context and sets the operators associated with the system (linear system matrix and optionally different preconditioning matrix). The user then sets various options for customized solution, solves the linear system, and finally destroys the `SLES` context. We emphasize the command `SLESSetFromOptions()`, which enables the user to customize the linear solution method at runtime by using the options database, which is discussed in Section 14.1. Through this database, the user not only can select an iterative method and preconditioner, but also can prescribe the convergence tolerance, set various monitoring routines, etc. (see, e.g., Figure 7). Chapter 4 describes in detail the `SLES` package, including the `PC` and `KSP` packages for preconditioners and Krylov subspace methods.

Nonlinear Solvers

Most PDE problems of interest are inherently nonlinear. PETSc provides an interface to tackle the nonlinear problems directly called SNES . Chapter 5 describes the nonlinear solvers in detail. We recommend most PETSc users work directly with SNES , rather than using PETSc for the linear problem within a nonlinear solver.

Error Checking

All PETSc routines return an integer indicating whether an error has occurred during the call. The PETSc macro `CHKERRQ(ierr)` checks the value of `ierr` and calls the PETSc error handler upon error detection. `CHKERRQ(ierr)` should be used in all subroutines to enable a complete error traceback. In Figure 4 we indicate a traceback generated by error detection within a sample PETSc program. The error occurred on line 1673 of the file `${PETSC_DIR}/src/mat/impls/aij/seq/aij.c` and was caused by trying to allocate too large an array in memory. The routine was called in the program `ex3.c` on line 71. See Section 11.1.2 for details regarding error checking when using the PETSc Fortran interface.

```
eagle:mpirun -np 1 ex3 -m 10000
PETSC ERROR: MatCreateSeqAIJ () line 1673 in src/mat/impls/aij/seq/aij.c
PETSC ERROR: Out of memory. This could be due to allocating
PETSC ERROR: too large an object or bleeding by not properly
PETSC ERROR: destroying unneeded objects.
PETSC ERROR: Try running with -trdump for more information.
PETSC ERROR: MatCreate () line 99 in src/mat/utils/gcreate.c
PETSC ERROR: main() line 71 in src/sles/examples/tutorials/ex3.c
MPI Abort by user Aborting program !
Aborting program!
p0_28969: p4_error: : 1
```

Figure 4: Example of Error Traceback

When running the debug (BOPT=g compiled) version of the PETSc libraries, it does a great deal of checking for memory corruption (writing outside of array bounds etc). The macros `CHKMEMQ` can be called anywhere in the code to check the current status of the memory for corruption. By putting several (or many) of these macros into your code you can usually easily track down in what small segment of your code the corruption has occured.

Parallel Programming

Since PETSc uses the message-passing model for parallel programming and employs MPI for all interprocessor communication, the user is free to employ MPI routines as needed throughout an application code. However, by default the user is shielded from many of the details of message passing within PETSc, since these are hidden within parallel objects, such as vectors, matrices, and solvers. In addition, PETSc provides tools such as generalized vector scatters/gathers and distributed arrays to assist in the management of parallel data. Recall that the user must specify a communicator upon creation of any PETSc object (such as a vector, matrix, or solver) to indicate the processors over which the object is to be distributed. For example, as mentioned above, some commands for matrix, vector, and linear solver creation are:

```
MatCreate (MPI_Comm comm ,int M,int N,Mat *A);
VecCreate (MPI_Comm comm ,Vec *x);
```

`SLESCreate`(MPI_Comm `comm`,`SLES` *sles);

The creation routines are collective over all processors in the communicator; thus, all processors in the communicator *must* call the creation routine. In addition, if a sequence of collective routines is being used, they *must* be called in the same order on each processor. The next example, given in Figure 5, illustrates the solution of a linear system in parallel. This code, corresponding to `${PETSC_DIR}/src/sles/examples/tutorials/ex2.c`, handles the two-dimensional Laplacian discretized with finite differences, where the linear system is again solved with `SLES`. The code performs the same tasks as the sequential version within Figure 3. Note that the user interface for initiating the program, creating vectors and matrices, and solving the linear system is *exactly* the same for the uniprocessor and multiprocessor examples. The primary difference between the examples in Figures 3 and 5 is that each processor forms only its local part of the matrix and vectors in the parallel case.

```
/*$Id: ex2.c,v 1.94 2001/08/07 21:30:54 bsmith Exp $*/

/* Program usage:  mpirun -np <procs> ex2 [-help] [all PETSc options] */

static char help[] = "Solves a linear system in parallel with SLES.\n\
Input parameters include:\n\
  -random_exact_sol : use a random exact solution vector\n\
  -view_exact_sol   : write exact solution vector to stdout\n\
  -m <mesh_x>       : number of mesh points in x-direction\n\
  -n <mesh_n>       : number of mesh points in y-direction\n\n";

/*T
   Concepts: SLES^basic parallel example;
   Concepts: SLES^Laplacian, 2d
   Concepts: Laplacian, 2d
   Processors: n
T*/

/*
  Include "petscsles.h" so that we can use SLES solvers.  Note that this file
  automatically includes:
     petsc.h       - base PETSc routines   petscvec.h - vectors
     petscsys.h    - system routines       petscmat.h - matrices
     petscis.h     - index sets            petscksp.h - Krylov subspace methods
     petscviewer.h - viewers               petscpc.h  - preconditioners
*/
#include "petscsles.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  Vec         x,b,u;  /* approx solution, RHS, exact solution */
  Mat         A;        /* linear system matrix */
  SLES        sles;     /* linear solver context */
  PetscRandom rctx;     /* random number generator context */
  PetscReal   norm;     /* norm of solution error */
  int         i,j,I,J,Istart,Iend,ierr,m = 8,n = 7,its;
  PetscTruth  flg;
  PetscScalar v,one = 1.0,neg_one = -1.0;
  KSP         ksp;

  PetscInitialize(&argc,&args,(char *)0,help);
  ierr = PetscOptionsGetInt(PETSC_NULL,"-m",&m,PETSC_NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       Compute the matrix and right-hand-side vector that define
       the linear system, Ax = b.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
   Create parallel matrix, specifying only its global dimensions.
   When using MatCreate(), the matrix format can be specified at
   runtime. Also, the parallel partitioning of the matrix is
   determined by PETSc at runtime.

   Performance tuning note:  For problems of substantial size,
   preallocation of matrix memory is crucial for attaining good
   performance.  Since preallocation is not possible via the generic
   matrix creation routine MatCreate(), we recommend for practical
   problems instead to use the creation routine for a particular matrix
   format, e.g.,
       MatCreateMPIAIJ() - parallel AIJ (compressed sparse row)
       MatCreateMPIBAIJ() - parallel block AIJ
   See the matrix chapter of the users manual for details.
*/
ierr = MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,m*n,m*n,&A);CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);

/*
   Currently, all PETSc parallel matrix formats are partitioned by
   contiguous chunks of rows across the processors.  Determine which
   rows of the matrix are locally owned.
*/
ierr = MatGetOwnershipRange(A,&Istart,&Iend);CHKERRQ(ierr);

/*
   Set matrix elements for the 2-D, five-point stencil in parallel.
    - Each processor needs to insert only elements that it owns
      locally (but any non-local elements will be sent to the
      appropriate processor during matrix assembly).
    - Always specify global rows and columns of matrix entries.

   Note: this uses the less common natural ordering that orders first
   all the unknowns for x = h then for x = 2h etc; Hence you see J = I +- n
   instead of J = I +- m as you might expect. The more standard ordering
   would first do all variables for y = h, then y = 2h etc.

 */
for (I=Istart; I<Iend; I++) {
  v = -1.0; i = I/n; j = I - i*n;
  if (i>0)   {J = I - n; ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);}
  if (i<m-1) {J = I + n; ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);}
  if (j>0)   {J = I - 1; ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);}
  if (j<n-1) {J = I + 1; ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);}
  v = 4.0; ierr = MatSetValues(A,1,&I,1,&I,&v,INSERT_VALUES);CHKERRQ(ierr);
}

/*
   Assemble matrix, using the 2-step process:
     MatAssemblyBegin(), MatAssemblyEnd()
   Computations can be done while messages are in transition
   by placing code between these two statements.
*/
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
```

```
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

/*
   Create parallel vectors.
    - We form 1 vector from scratch and then duplicate as needed.
    - When using VecCreate(), VecSetSizes and VecSetFromOptions()
      in this example, we specify only the
      vector’s global dimension; the parallel partitioning is determined
      at runtime.
    - When solving a linear system, the vectors and matrices MUST
      be partitioned accordingly.  PETSc automatically generates
      appropriately partitioned matrices and vectors when MatCreate()
      and VecCreate() are used with the same communicator.
    - The user can alternatively specify the local vector and matrix
      dimensions when more sophisticated partitioning is needed
      (replacing the PETSC_DECIDE argument in the VecSetSizes() statement
      below).
*/
ierr = VecCreate(PETSC_COMM_WORLD,&u);CHKERRQ(ierr);
ierr = VecSetSizes(u,PETSC_DECIDE,m*n);CHKERRQ(ierr);
ierr = VecSetFromOptions(u);CHKERRQ(ierr);
ierr = VecDuplicate(u,&b);CHKERRQ(ierr);
ierr = VecDuplicate(b,&x);CHKERRQ(ierr);

/*
   Set exact solution; then compute right-hand-side vector.
   By default we use an exact solution of a vector with all
   elements of 1.0;  Alternatively, using the runtime option
   -random_sol forms a solution vector with random components.
*/
ierr = PetscOptionsHasName(PETSC_NULL,"-random_exact_sol",&flg);CHKERRQ(ierr);
if (flg) {
  ierr = PetscRandomCreate(PETSC_COMM_WORLD,RANDOM_DEFAULT,&rctx);CHKERRQ(ierr);
  ierr = VecSetRandom(rctx,u);CHKERRQ(ierr);
  ierr = PetscRandomDestroy(rctx);CHKERRQ(ierr);
} else {
  ierr = VecSet(&one,u);CHKERRQ(ierr);
}
ierr = MatMult(A,u,b);CHKERRQ(ierr);

/*
   View the exact solution vector if desired
*/
ierr = PetscOptionsHasName(PETSC_NULL,"-view_exact_sol",&flg);CHKERRQ(ierr);
if (flg) {ierr = VecView(u,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);}

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
              Create the linear solver and set various options
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

/*
   Create linear solver context
*/
ierr = SLESCreate(PETSC_COMM_WORLD,&sles);CHKERRQ(ierr);

/*
   Set operators. Here the matrix that defines the linear system
   also serves as the preconditioning matrix.
*/
ierr = SLESSetOperators(sles,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);
```

```
  /*
     Set linear solver defaults for this problem (optional).
     - By extracting the KSP and PC contexts from the SLES context,
       we can then directly call any KSP and PC routines to set
       various options.
     - The following two statements are optional; all of these
       parameters could alternatively be specified at runtime via
       SLESSetFromOptions().  All of these defaults can be
       overridden at runtime, as indicated below.
  */

  ierr = SLESGetKSP(sles,&ksp);CHKERRQ(ierr);
  ierr = KSPSetTolerances(ksp,1.e-2/((m+1)*(n+1)),1.e-50,PETSC_DEFAULT,PETSC_DEFAULT);CHKERRQ(ierr);

  /*
    Set runtime options, e.g.,
        -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <rtol>
    These options will override those specified above as long as
    SLESSetFromOptions() is called _after_ any other customization
    routines.
  */
  ierr = SLESSetFromOptions(sles);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                  Solve the linear system
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = SLESSolve(sles,b,x,&its);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                  Check solution and clean up
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /*
     Check the error
  */
  ierr = VecAXPY(&neg_one,u,x);CHKERRQ(ierr);
  ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);

  /* Scale the norm */
  /*  norm *= sqrt(1.0/((m+1)*(n+1))); */

  /*
     Print convergence information.  PetscPrintf() produces a single
     print statement from all processes that share a communicator.
     An alternative is PetscFPrintf(), which prints to a file.
  */
  ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %A iterations %d\n",norm,its);CHKERRQ(ierr);

  /*
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
  */
  ierr = SLESDestroy(sles);CHKERRQ(ierr);
  ierr = VecDestroy(u);CHKERRQ(ierr);  ierr = VecDestroy(x);CHKERRQ(ierr);
  ierr = VecDestroy(b);CHKERRQ(ierr);  ierr = MatDestroy(A);CHKERRQ(ierr);

  /*
     Always call PetscFinalize() before exiting a program.  This routine
```

```
       - finalizes the PETSc libraries as well as MPI
       - provides summary and diagnostic information if certain runtime
         options are chosen (e.g., -log_summary).
  */
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

Figure 5: Example of Multiprocessor PETSc Code

Compiling and Running Programs

Figure 6 illustrates compiling and running a PETSc program using MPICH. Note that different sites may have slightly different library and compiler names. See Chapter 15 for a discussion about compiling PETSc programs. Users who are experiencing difficulties linking PETSc programs should refer to the troubleshooting guide via the PETSc WWW home page `http://www.mcs.anl.gov/petsc` or given in the file `${PETSC_DIR}/docs/troubleshooting.html`.

```
eagle: make BOPT=g ex2
gcc -pipe -c -I../../../ -I../../..//include
-I/usr/local/mpi/include -I../../..//src -g
-DPETSC_USE_DEBUG -DPETSC_MALLOC -DPETSC_USE_LOG ex1.c
gcc -g -DPETSC_USE_DEBUG -DPETSC_MALLOC -DPETSC_USE_LOG -o ex1 ex1.o
/home/bsmith/petsc/lib/libg/sun4/libpetscsles.a
-L/home/bsmith/petsc/lib/libg/sun4 -lpetscstencil -lpetscgrid -lpetscsles
-lpetscmat -lpetscvec -lpetscsys -lpetscdraw
/usr/local/lapack/lib/lapack.a /usr/local/lapack/lib/blas.a
/usr/lang/SC1.0.1/libF77.a -lm /usr/lang/SC1.0.1/libm.a -lX11
/usr/local/mpi/lib/sun4/ch_p4/libmpi.a
/usr/lib/debug/malloc.o /usr/lib/debug/mallocmap.o
/usr/lang/SC1.0.1/libF77.a -lm /usr/lang/SC1.0.1/libm.a -lm
rm -f ex1.o
eagle: mpirun -np 1 ex2
Norm of error 3.6618e-05 iterations 7
eagle:
eagle: mpirun -np 2 ex2
Norm of error 5.34462e-05 iterations 9
```

Figure 6: Running a PETSc Program

As shown in Figure 7, the option `-log_summary` activates printing of a performance summary, including times, floating point operation (flop) rates, and message-passing activity. Chapter 12 provides details about profiling, including interpretation of the output data within Figure 7. This particular example involves the solution of a linear system on one processor using GMRES and ILU. The low floating point operation (flop) rates in this example are due to the fact that the code solved a tiny system. We include this example merely to demonstrate the ease of extracting performance information.

```
eagle> mpirun -np 1 ex1 -n 1000 -pc_type ilu -ksp_type gmres -ksp_rtol 1.e-7 -log_summary
-------------------------------- PETSc Performance Summary: ---------------------------
-----------
ex1 on a sun4 named merlin.mcs.anl.gov with 1 processor, by curfman Wed Aug  7 17:24:27 1996
Max         Min        Avg        Total
Time (sec):           1.150e-01      1.0   1.150e-01
```

```
Objects:              1.900e+01      1.0   1.900e+01
Flops:                3.998e+04      1.0   3.998e+04  3.998e+04
Flops/sec:            3.475e+05      1.0              3.475e+05
MPI Messages:         0.000e+00      0.0   0.000e+00  0.000e+00
MPI Messages:         0.000e+00      0.0   0.000e+00  0.000e+00 (lengths)
MPI Reductions:       0.000e+00      0.0
-------------------------------------------------------------------------------------
-----------
Phase               Count      Time (sec)       Flops/sec                            -- Global -
-
                            Max     Ratio    Max    Ratio   Mess  Avg len  Reduct  %T %F %M %L %R
-------------------------------------------------------------------------------------
-----------
MatMult                 2  2.553e-03    1.0  3.9e+06    1.0  0.0e+00 0.0e+00 0.0e+00  2 25  0  0  0
MatAssemblyBegin        1  2.193e-05    1.0  0.0e+00    0.0  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
MatAssemblyEnd          1  5.004e-03    1.0  0.0e+00    0.0  0.0e+00 0.0e+00 0.0e+00  4  0  0  0  0
MatGetReordering        1  3.004e-03    1.0  0.0e+00    0.0  0.0e+00 0.0e+00 0.0e+00  3  0  0  0  0
MatILUFctrSymbol        1  5.719e-03    1.0  0.0e+00    0.0  0.0e+00 0.0e+00 0.0e+00  5  0  0  0  0
MatLUFactorNumer        1  1.092e-02    1.0  2.7e+05    1.0  0.0e+00 0.0e+00 0.0e+00  9  7  0  0  0
MatSolve                2  4.193e-03    1.0  2.4e+06    1.0  0.0e+00 0.0e+00 0.0e+00  4 25  0  0  0
MatSetValues         1000  2.461e-02    1.0  0.0e+00    0.0  0.0e+00 0.0e+00 0.0e+00 21  0  0  0  0
VecDot                  1     60e-04    1.0  9.7e+06    1.0  0.0e+00 0.0e+00 0.0e+00  0  5  0  0  0
VecNorm                 3  5.870e-04    1.0  1.0e+07    1.0  0.0e+00 0.0e+00 0.0e+00  1 15  0  0  0
VecScale                1  1.640e-04    1.0  6.1e+06    1.0  0.0e+00 0.0e+00 0.0e+00  0  3  0  0  0
VecCopy                 1  3.101e-04    1.0  0.0e+00    0.0  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
VecSet                  3  5.029e-04    1.0  0.0e+00    0.0  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0
VecAXPY                 3  8.690e-04    1.0  6.9e+06    1.0  0.0e+00 0.0e+00 0.0e+00  1 15  0  0  0
VecMAXPY                1  2.550e-04    1.0  7.8e+06    1.0  0.0e+00 0.0e+00 0.0e+00  0  5  0  0  0
SLESSolve               1  1.288e-02    1.0  2.2e+06    1.0  0.0e+00 0.0e+00 0.0e+00 11 70  0  0  0
SLESSetUp               1  2.669e-02    1.0  1.1e+05    1.0  0.0e+00 0.0e+00 0.0e+00 23  7  0  0  0
KSPGMRESOrthog          1  1.151e-03    1.0  3.5e+06    1.0  0.0e+00 0.0e+00 0.0e+00  1 10  0  0  0
PCSetUp                 1 24e-02    1.0  1.5e+05    1.0  0.0e+00 0.0e+00 0.0e+00 18  7  0  0  0
PCApply                 2  4.474e-03    1.0  2.2e+06    1.0  0.0e+00 0.0e+00 0.0e+00  4 25  0  0  0
-------------------------------------------------------------------------------------
----------
Memory usage is given in bytes:
Object Type      Creations   Destructions   Memory  Descendants' Mem.
Index set            3              3       12420     0
Vector               8              8       65728     0
Matrix               2              2      184924     4140
Krylov Solver        1              1       16892     41080
Preconditioner       1              1           0     64872
SLES                 1              1           0     122844
```

Figure 7: Running a PETSc Program with Profiling

Writing Application Codes with PETSc

The examples throughout the library demonstrate the software usage and can serve as templates for developing custom applications. We suggest that new PETSc users examine programs in the directories

\${PETSC_DIR}/src/<library>/examples/tutorials,

where <library> denotes any of the PETSc libraries (listed in the following section), such as snes or sles. The manual pages located at

$PETSC_DIR/docs/index.html or
http://www.mcs.anl.gov/petsc/docs/

provide indices (organized by both routine names and concepts) to the tutorial examples. To write a new application program using PETSc, we suggest the following procedure:

1. Install and test PETSc according to the instructions at the PETSc web site.

2. Copy one of the many PETSc examples in the directory that corresponds to the class of problem of interest (e.g., for linear solvers, see `${PETSC_DIR}/src/sles/examples/tutorials`).

3. Copy the corresponding makefile within the example directory; compile and run the example program.

4. Use the example program as a starting point for developing a custom code.

1.5 Referencing PETSc

When referencing PETSc in a publication please cite the following:

@Unpublished{petsc-home-page,
Author = ”Satish Balay and William D. Gropp and Lois C. McInnes and Barry F. Smith”,
Title = ”PETSc home page”,
Note = ”http://www.mcs.anl.gov/petsc”,
Year = ”2001”}
@TechReport{petsc-manual,
Author = ”Satish Balay and William D. Gropp and Lois C. McInnes and Barry F. Smith”,
Title = ”PETSc Users Manual”,
Number = ”ANL-95/11 - Revision 2.1.5”,
Institution = ”Argonne National Laboratory”,
Year = ”2003”}
@InProceedings{petsc-efficient,
Author = ”Satish Balay and William D. Gropp and Lois C. McInnes and Barry F. Smith”,
Title = ”Efficienct Management of Parallelism in Object Oriented Numerical Software Libraries”,
Booktitle = ”Modern Software Tools in Scientific Computing”,
Editor = ”E. Arge and A. M. Bruaset and H. P. Langtangen”,
Pages = ”163–202”,
Publisher = ”Birkhauser Press”,
Year = ”1997”}

1.6 Directory Structure

We conclude this introduction with an overview of the organization of the PETSc software. The root directory of PETSc contains the following directories:

- `docs` - All documentation for PETSc. The files `manual.pdf` contains the hyperlinked users manual, suitable for printing or on-screen viewering. Includes the subdirectory

  - `manualpages` (on-line manual pages).

- `bin` - Utilities and short scripts for use with PETSc, including

  – `petsarch` (utility for setting `PETSC_ARCH` environmental variable),

- `bmake` - Base PETSc makefile directory. Includes subdirectories for various architectures.

- `include` - All include files for PETSc that are visible to the user.
- `include/finclude` - PETSc include files for Fortran programmers using the .F suffix (recommended).
- `include/pinclude` - Private PETSc include files that should *not* be used by application programmers.
- `src` - The source code for all PETSc libraries, which currently includes
  - `vec` - vectors,
    - `is` - index sets,
  - `mat` - matrices,
  - `dm`
    - `da` - distributed arrays,
    - `ao` - application orderings,
  - `sles` - complete linear equations solvers,
    - `ksp` - Krylov subspace accelerators,
    - `pc` - preconditioners,
  - `snes` - nonlinear solvers
  - `ts` - ODE solvers and timestepping,
  - `sys` - general system-related routines,
    - `plog` - PETSc logging and profiling routines,
    - `draw` - simple graphics,
  - `fortran` - Fortran interface stubs,
  - `contrib` - contributed modules that use PETSc but are not part of the official PETSc package. We encourage users who have developed such code that they wish to share with others to let us know by writing to petsc-maint@mcs.anl.gov.

Each PETSc source code library directory has the following subdirectories:

- `examples` - Example programs for the component, including
  - `tutorials` - Programs designed to teach users about PETSc. These codes can serve as templates for the design of custom applicatinos.
  - `tests` - Programs designed for thorough testing of PETSc. As such, these codes are not intended for examination by users.
- `interface` - The calling sequences for the abstract interface to the component. Code here does not know about particular implementations.
- `impls` - Source code for one or more implementations.
- `utils` - Utility routines. Source here may know about the implementations, but ideally will not know about implementations for other components.

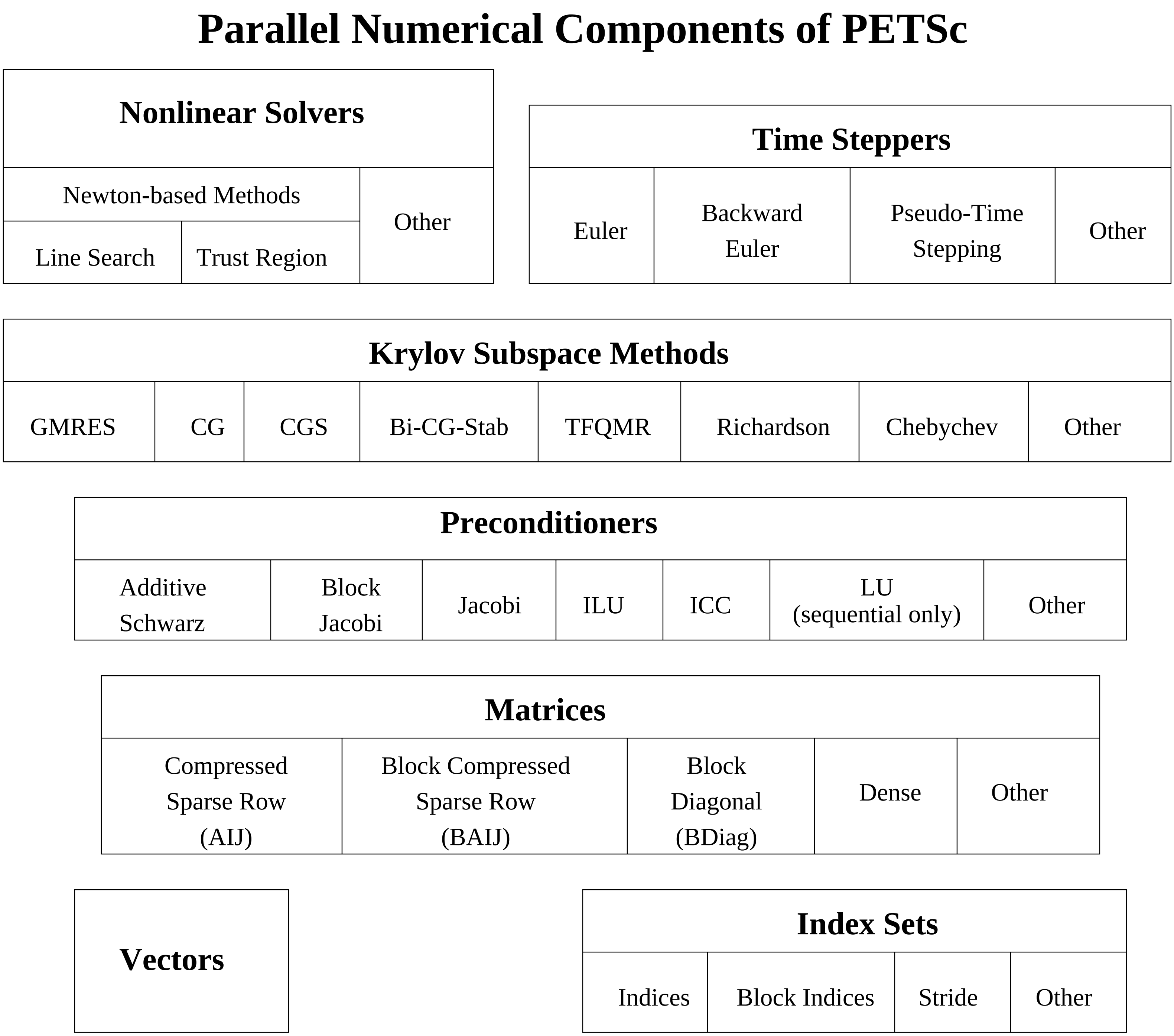

Figure 2: Numerical Libraries of PETSc

Part II

Programming with PETSc

Chapter 2

Vectors and Distributing Parallel Data

The vector (denoted by `Vec`) is one of the simplest PETSc objects. Vectors are used to store discrete PDE solutions, right-hand sides for linear systems, etc. This chapter is organized as follows:

- (`Vec`) Sections 2.1 and 2.2 - basic usage of vectors
- Section 2.3 - management of the various numberings of degrees of freedom, vertices, cells, etc.
  - (`AO`) Mapping between different global numberings
  - (`ISLocalToGlobalMapping`) Mapping between local and global numberings
- (`DA`) Section 2.4 - management of structured grids
- (`IS` , `VecScatter`) Section 2.5 - management of vectors related to unstructured grids

2.1 Creating and Assembling Vectors

PETSc currently provides two basic vector types: sequential and parallel (MPI based). To create a sequential vector with `m` components, one can use the command

`VecCreateSeq` (PETSC_COMM_SELF,int m,`Vec` *x);

To create a parallel vector one can either specify the number of components that will be stored on each processor or let PETSc decide. The command

`VecCreateMPI` (MPI_Comm `comm` ,int m,int M,`Vec` *x);

creates a vector that is distributed over all processors in the communicator, `comm` , where `m` indicates the number of components to store on the local processor, and `M` is the total number of vector components. Either the local or global dimension, but not both, can be set to PETSC_DECIDE to indicate that PETSc should determine it. More generally, one can use the routines

`VecCreate` (MPI_Comm `comm` ,`Vec` *v);
`VecSetSizes` (`Vec` v, int m, int M);
`VecSetFromOptions` (`Vec` v);

which automatically generates the appropriate vector type (sequential or parallel) over all processors in `comm`. The option `-vec_type mpi` can be used in conjunction with `VecCreate` () and `VecSetFromOptions` () to specify the use of MPI vectors even for the uniprocessor case. We emphasize that all processors in `comm` *must* call the vector creation routines, since these routines are collective over all processors in the communicator. If you are not familar with MPI communicators, see the discussion in Section 1.3 on page 17. In addition, if a sequence of `VecCreateXXX()` routines is used, they must be called in the same order on each processor in the communicator. One can assign a single value to all components of a vector with the command

`VecSet`(`PetscScalar` *value,`Vec` x);

Assigning values to individual components of the vector is more complicated, in order to make it possible to write efficient parallel code. Assigning a set of components is a two-step process: one first calls

`VecSetValues`(`Vec` x,int n,int *indices,`PetscScalar` *values,INSERT_VALUES);

any number of times on any or all of the processors. The argument `n` gives the number of components being set in this insertion. The integer array `indices` contains the *global component indices*, and `values` is the array of values to be inserted. Any processor can set any components of the vector; PETSc insures that they are automatically stored in the correct location. Once all of the values have been inserted with `VecSetValues`(), one must call

`VecAssemblyBegin`(`Vec` x);

followed by

`VecAssemblyEnd`(`Vec` x);

to perform any needed message passing of nonlocal components. In order to allow the overlap of communication and calculation, the user's code can perform any series of other actions between these two calls while the messages are in transition.

Example usage of `VecSetValues`() may be found in `${PETSC_DIR}/src/vec/examples/tutorials/ex2.c` or `ex2f.F` Often, rather than inserting elements in a vector, one may wish to add values. This process is also done with the command

`VecSetValues`(`Vec` x,int n,int *indices, `PetscScalar` *values,ADD_VALUES);

Again one must call the assembly routines `VecAssemblyBegin`() and `VecAssemblyEnd`() after all of the values have been added. Note that addition and insertion calls to `VecSetValues`() *cannot* be mixed. Instead, one must add and insert vector elements in phases, with intervening calls to the assembly routines. This phased assembly procedure overcomes the nondeterministic behavior that would occur if two different processors generated values for the same location, with one processor adding while the other is inserting its value. (In this case the addition and insertion actions could be performed in either order, thus resulting in different values at the particular location. Since PETSc does not allow the simultaneous use of INSERT_VALUES and ADD_VALUES this nondeterministic behavior will not occur in PETSc.) There is no routine called VecGetValues(), since we provide an alternative method for extracting some components of a vector using the vector scatter routines. See Section 2.5.2 for details; see also below for `VecGetArray`(). One can examine a vector with the command

`VecView`(`Vec` x,`PetscViewer` v);

To print the vector to the screen, one can use the viewer `PETSC_VIEWER_STDOUT_WORLD`, which ensures that parallel vectors are printed correctly to `stdout`. To display the vector in an X-window, one can use the default X-windows viewer `PETSC_VIEWER_DRAW_WORLD`, or one can create a viewer with the routine PetscViewerDrawOpenX(). A variety of viewers are discussed further in Section 14.2. To create a new vector of the same format as an existing vector, one uses the command

`VecDuplicate`(`Vec` old,`Vec` *new);

To create several new vectors of the same format as an existing vector, one uses the command

VecDuplicateVecs(Vec old,int n,Vec **new);

This routine creates an array of pointers to vectors. The two routines are very useful because they allow one to write library code that does not depend on the particular format of the vectors being used. Instead, the subroutines can automatically correctly create work vectors based on the specified existing vector. As discussed in Section 11.1.6, the Fortran interface for VecDuplicateVecs() differs slightly. When a vector is no longer needed, it should be destroyed with the command

VecDestroy(Vec x);

To destroy an array of vectors, one should use the command

VecDestroyVecs(Vec *vecs,int n);

Note that the Fortran interface for VecDestroyVecs() differs slightly, as described in Section 11.1.6. It is also possible to create vectors that use an array provided by the user, rather than having PETSc internally allocate the array space. Such vectors can be created with the routines

VecCreateSeqWithArray(PETSC_COMM_SELF,int m,PetscScalar *array,Vec *x);

and

VecCreateMPIWithArray(MPI_Comm comm,int m,int M,,PetscScalar *array,Vec *x);

Note that here one must provide the value `m`, it cannot be PETSC_DECIDE and the user is responsible for providing enough space in the array; `m*sizeof(PetscScalar)`.

2.2 Basic Vector Operations

As listed in Table 1, we have chosen certain basic vector operations to support within the PETSc vector library. These operations were selected because they often arise in application codes. The NormType argument to VecNorm() is one of `NORM_1`, `NORM_2`, or `NORM_INFINITY`. The 1-norm is $\sum_i |x_i|$, the 2-norm is $(\sum_i x_i^2)^{1/2}$ and the infinity norm is $\max_i |x_i|$.

For parallel vectors that are distributed across the processors by ranges, it is possible to determine a processor's local range with the routine

VecGetOwnershipRange(Vec vec,int *low,int *high);

The argument `low` indicates the first component owned by the local processor, while `high` specifies *one more than* the last owned by the local processor. This command is useful, for instance, in assembling parallel vectors. On occasion, the user needs to access the actual elements of the vector. The routine VecGetArray() returns a pointer to the elements local to the processor:

VecGetArray(Vec v,PetscScalar **array);

When access to the array is no longer needed, the user should call

VecRestoreArray(Vec v, PetscScalar **array);

Minor differences exist in the Fortran interface for VecGetArray() and VecRestoreArray(), as discussed in Section 11.1.3. It is important to note that VecGetArray() and VecRestoreArray() do *not* copy the vector elements; they merely give users direct access to the vector elements. Thus, these routines require essentially no time to call and can be used efficiently. The number of elements stored locally can be accessed with

Function Name	Operation
VecAXPY (PetscScalar *a,Vec x, Vec y);	$y = y + a * x$
VecAYPX (PetscScalar *a,Vec x, Vec y);	$y = x + a * y$
VecWAXPY (PetscScalar *a,Vec x,Vec y, Vec w);	$w = a * x + y$
VecAXPBY (PetscScalar *a,PetscScalar *,Vec x,Vec y);	$y = a * x + b * y$
VecScale (PetscScalar *a, Vec x);	$x = a * x$
VecDot (Vec x, Vec y, PetscScalar *r);	$r = \bar{x}' * y$
VecTDot (Vec x, Vec y, PetscScalar *r);	$r = x' * y$
VecNorm (Vec x,NormType type, double *r);	$r = \|\|x\|\|_{type}$
VecSum (Vec x, PetscScalar *r);	$r = \sum x_i$
VecCopy (Vec x, Vec y);	$y = x$
VecSwap (Vec x, Vec y);	$y = x$ while $x = y$
VecPointwiseMult (Vec x,Vec y, Vec w);	$w_i = x_i * y_i$
VecPointwiseDivide (Vec x,Vec y, Vec w);	$w_i = x_i / y_i$
VecMDot (int n,Vec x, Vec y[],PetscScalar *r);	$r[i] = \bar{x}' * y[i]$
VecMTDot (int n,Vec x, Vec y[],PetscScalar *r);	$r[i] = x' * y[i]$
VecMAXPY (int n, PetscScalar *a,Vec y, Vec x[]);	$y = y + \sum_i a_i * x[i]$
VecMax (Vec x, int *idx, double *r);	$r = \max x_i$
VecMin (Vec x, int *idx, double *r);	$r = \min x_i$
VecAbs (Vec x);	$x_i = \|x_i\|$
VecReciprocal (Vec x);	$x_i = 1/x_i$
VecShift (PetscScalar *s,Vec x);	$x_i = s + x_i$

Table 1: PETSc Vector Operations

```
VecGetLocalSize(Vec v,int *size);
```

The global vector length can be determined by

```
VecGetSize(Vec v,int *size);
```

In addition to VecDot () and VecMDot () and VecNorm () PETSc provides split phase versions of these that allow several independent inner products and/or norms to share the same communication (thus improving parallel efficiency). For example, one may have code such as

```
VecDot(Vec x,Vec y,PetscScalar *dot);
VecNorm(Vec x,NormType NORM_2,double *norm2);
VecNorm(Vec x,NormType NORM_1,double *norm1);
```

This code works fine, the problem is that it performs three seperate parallel communication operations. Instead one can write

```
VecDotBegin(Vec x,Vec y,PetscScalar *dot);
VecNormBegin(Vec x,NormType NORM_2,double *norm2);
VecNormBegin(Vec x,NormType NORM_1,double *norm1);
VecDotEnd(Vec x,Vec y,PetscScalar *dot);
VecNormEnd(Vec x,NormType NORM_2,double *norm2);
VecNormEnd(Vec x,NormType NORM_1,double *norm1);
```

With this code, the communication is delayed until the first call to `VecxxxEnd()` at which a single MPI reduction is used to communicate all the required values. It is required that the calls to the `VecxxxEnd()`

are performed in the same order as the calls to the VecxxxBegin(); however if you mistakenly make the calls in the wrong order PETSc will generate an error, informing you of this. There are two additional routines VecTDotBegin () and VecTDotEnd (). These routines where suggested by Victor Eijkhout. Note: these routines use only MPI 1 functionality; so they do not allow you to overlap computation and communication (assuming no threads are spawned within a MPI process). Once MPI 2 implementations are more common we'll improve these routines to allow overlap of inner product and norm calculations with other calculations. Also currently these routines only work for the PETSc built in vector types.

2.3 Indexing and Ordering

When writing parallel PDE codes there is extra complexity caused by having multiple ways of indexing (numbering) and ordering objects such as vertices and degrees of freedom. For example, a grid generator or partitioner may renumber the nodes, requiring adjustment of the other data structures that refer to these objects; see Figure 9. In addition, local numbering (on a single processor) of objects may be different than the global (cross-processor) numbering. PETSc provides a variety of tools that help to manage the mapping among the various numbering systems. The two most basic are the AO (application ordering), which enables mapping between different global (cross-processor) numbering schemes and the ISLocalToGlobalMapping, which allows mapping between local (on-processor) and global (cross-processor) numbering.

2.3.1 Application Orderings

In many applications it is desirable to work with one or more "orderings" (or numberings) of degrees of freedom, cells, nodes, etc. Doing so in a parallel environment is complicated by the fact that each processor cannot keep complete lists of the mappings between different orderings. In addition, the orderings used in the PETSc linear algebra routines (often contiguous ranges) may not correspond to the "natural" orderings for the application. PETSc provides certain utility routines that allow one to deal cleanly and efficiently with the various orderings. To define a new application ordering (called an AO in PETSc), one can call the routine

AOCreateBasic (MPI_Comm comm ,int n,int *apordering,int *petscordering,AO *ao);

The arrays apordering and petscordering, respectively, contain a list of integers in the application ordering and their corresponding mapped values in the PETSc ordering. Each processor can provide whatever subset of the ordering it chooses, but multiple processors should never contribute duplicate values. The argument n indicates the number of local contributed values. For example, consider a vector of length five, where node 0 in the application ordering corresponds to node 3 in the PETSc ordering. In addition, nodes 1, 2, 3, and 4 of the application ordering correspond, respectively, to nodes 2, 1, 4, and 0 of the PETSc ordering. We can write this correspondence as

$$0, 1, 2, 3, 4 \rightarrow 3, 2, 1, 4, 0.$$

The user can create the PETSc-AO mappings in a number of ways. For example, if using two processors, one could call

```
AOCreateBasic (PETSC_COMM_WORLD,2,{0,3},{3,4},&ao);
```

on the first processor and

```
AOCreateBasic (PETSC_COMM_WORLD,3,{1,2,4},{2,1,0},&ao);
```

on the other processor. Once the application ordering has been created, it can be used with either of the commands

AOPetscToApplication (AO ao,int n,int *indices);
AOApplicationToPetsc (AO ao,int n,int *indices);

Upon input, the n-dimensional array indices specifies the indices to be mapped, while upon output, indices contains the mapped values. Since we, in general, employ a parallel database for the AO mappings, it is crucial that all processors that called AOCreateBasic () also call these routines; these routines *cannot* be called by just a subset of processors in the MPI communicator that was used in the call to AOCreateBasic (). An alternative routine to create the application ordering, AO , is

AOCreateBasicIS (IS apordering,IS petscordering,AO *ao);

where index sets (see 2.5.1 are used instead of integer arrays.

The mapping routines

AOPetscToApplicationIS (AO ao,IS indices);
AOApplicationToPetscIS (AO ao,IS indices);

will map index sets (IS objects) between orderings. Both the AOXxxToYyy() and AOXxxToYyyIS() routines can be used regardless of whether the AO was created with a AOCreateBasic () or AOCreateBasicIS (). The AO context should be destroyed with AODestroy(AO ao) and viewed with AOView(AO ao,PetscViewer viewer). Although we refer to the two orderings as "PETSc" and "application" orderings, the user is free to use them both for application orderings and to maintain relationships among a variety of orderings by employing several AO contexts. The AOxxToxx() routines allow negative entries in the input integer array. These entries are not mapped; they simply remain unchanged. This functionality enables, for example, mapping neighbor lists that use negative numbers to indicate nonexistent neighbors due to boundary conditions, etc.

2.3.2 Local to Global Mappings

In many applications one works with a global representation of a vector (usually on a vector obtained with VecCreateMPI ()) and a local representation of the same vector that includes ghost points required for local computation. PETSc provides routines to help map indices from a local numbering scheme to the PETSc global numbering scheme. This is done via the following routines

ISLocalToGlobalMappingCreate (int N,int* globalnum,
ISLocalToGlobalMapping * ctx);
ISLocalToGlobalMappingApply (ISLocalToGlobalMapping ctx,int n,int *in,
int *out);
ISLocalToGlobalMappingApplyIS (ISLocalToGlobalMapping ctx,IS isin,
IS * isout);
ISLocalToGlobalMappingDestroy (ISLocalToGlobalMapping ctx);

Here N denotes the number of local indices, globalnum contains the global number of each local number, and ISLocalToGlobalMapping is the resulting PETSc object that contains the information needed to apply the mapping with either ISLocalToGlobalMappingApply () or ISLocalToGlobalMappingApplyIS (). Note that the ISLocalToGlobalMapping routines serve a different purpose than the AO routines. In the former case they provide a mapping from a local numbering scheme (including ghost points) to a global numbering scheme, while in the latter they provide a mapping between two global numbering schemes. In fact, many applications may use both AO and ISLocalToGlobalMapping routines. The AO routines are first used to map from an application global ordering (that has no relationship to parallel processing etc.) to the PETSc ordering scheme (where each processor has a contiguous set of indices in the numbering). Then in order to perform function or Jacobian evaluations locally on each processor, one works with a local numbering scheme that includes ghost points. The mapping from this local numbering scheme back to the global PETSc numbering can be handled with the `ISLocalToGlobalMapping` routines. If one is given a list of indices in a global numbering, the routine

`ISGlobalToLocalMappingApply` (`ISLocalToGlobalMapping` ctx,
`ISGlobalToLocalMappingType` type,int nin,int idxin[],int *nout,int idxout[]);

will provide a new list of indices in the local numbering. Again, negative values in `idxin` are left unmapped. But, in addition, if `type` is set to `IS_GTOLM_MASK` , then `nout` is set to `nin` and all global values in `idxin` that are not represented in the local to global mapping are replaced by -1. When `type` is set to `IS_GTOLM_DROP`, the values in `idxin` that are not represented locally in the mapping are not included in `idxout`, so that potentially `nout` is smaller than `nin`. One must pass in an array long enough to hold all the indices. One can call `ISGlobalToLocalMappingApply` () with `idxout` equal to `PETSC_NULL` to determine the required length (returned in `nout`) and then allocate the required space and call `ISGlobalToLocalMappingApply()` a second time to set the values. Often it is convenient to set elements into a vector using the local node numbering rather than the global node numbering (e.g., each processor may maintain its own sublist of vertices and elements and number them locally). To set values into a vector with the local numbering, one must first call

`VecSetLocalToGlobalMapping` (`Vec` v,`ISLocalToGlobalMapping` ctx);

and then call

`VecSetValuesLocal` (`Vec` x,int n,int *indices,`PetscScalar` *values,INSERT_VALUES);

Now the `indices` use the local numbering, rather than the global.

2.4 Structured Grids Using Distributed Arrays

Distributed arrays (DAs), which are used in conjunction with PETSc vectors, are intended for use with *logically regular rectangular grids* when communication of nonlocal data is needed before certain local computations can occur. PETSc distributed arrays are designed only for the case in which data can be thought of as being stored in a standard multidimensional array; thus, DAs are *not* intended for parallelizing unstructured grid problems, etc. DAs are intended for communicating vector (field) information; they are not intended for storing matrices. For example, a typical situation one encounters in solving PDEs in parallel is that, to evaluate a local function, `f(x)`, each processor requires its local portion of the vector `x` as well as its ghost points (the bordering portions of the vector that are owned by neighboring processors). Figure 8 illustrates the ghost points for the seventh processor of a two-dimensional, regular parallel grid. Each box represents a processor; the ghost points for the seventh processor's local part of a parallel array are shown in gray.

2.4.1 Creating Distributed Arrays

The PETSc `DA` object manages the parallel communication required while working with data stored in regular arrays. The actual data is stored in approriately sized vector objects; the `DA` object only contains the parallel data layout information and communication information.

One creates a distributed array communication data structure in two dimensions with the command

`DACreate2d` (MPI_Comm `comm` ,`DAPeriodicType` wrap,`DAStencilType` st,int M,
int N,int m,int n,int dof,int s,int *lx,int *ly,`DA` *da);

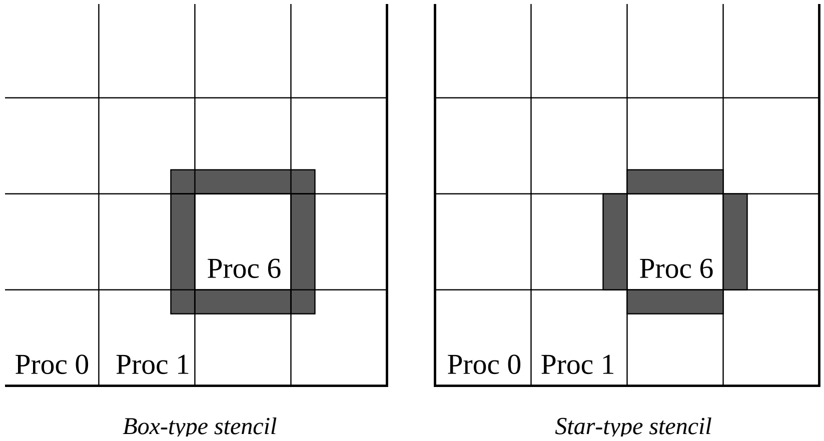

Figure 8: Ghost Points for Two Stencil Types on the Seventh Processor

The arguments M and N indicate the global numbers of grid points in each direction, while m and n denote the processor partition in each direction; m*n must equal the number of processes in the MPI communicator, comm. Instead of specifying the processor layout, one may use PETSC_DECIDE for m and n so that PETSc will determine the partition using MPI. The type of periodicity of the array is specified by wrap, which can be DA_NONPERIODIC (no periodicity), DA_XYPERIODIC (periodic in both x- and y-directions), DA_XPERIODIC , or DA_YPERIODIC. The argument dof indicates the number of degrees of freedom at each array point, and s is the stencil width (i.e., the width of the ghost point region). The optional arrays lx and ly may contain the number of nodes along the x and y axis for each cell, i.e. the dimension of lx is m and the dimension of ly is n; or PETSC_NULL may be passed in. Two types of distributed array communication data structures can be created, as specified by st. Star-type stencils that radiate outward only in the coordinate directions are indicated by DA_STENCIL_STAR, while box-type stencils are specified by DA_STENCIL_BOX. For example, for the two-dimensional case, DA_STENCIL_STAR with width 1 corresponds to the standard 5-point stencil, while DA_STENCIL_BOX with width 1 denotes the standard 9-point stencil. In both instances the ghost points are identical, the only difference being that with star-type stencils certain ghost points are ignored, decreasing substantially the number of messages sent. Note that the DA_STENCIL_STAR stencils can save interprocessor communication in two and three dimensions. These DA stencils have nothing directly to do with any finite difference stencils one might chose to use for a discretization; they only ensure that the correct values are in place for application of a user-defined finite difference stencil (or any other discretization technique). The commands for creating distributed array communication data structures in one and three dimensions are analogous:

DACreate1d (MPI_Comm comm ,DAPeriodicType wrap,int M,int w,int s,int *lc,DA *inra);
DACreate3d (MPI_Comm comm ,DAPeriodicType wrap,DAStencilType stencil_type,
int M,int N,int P,int m,int n,int p,int w,int s,int *lx,int *ly,int *lz,DA *inra);

DA_ZPERIODIC, DA_XZPERIODIC, DA_YZPERIODIC, and DA_XYZPERIODIC are additional options in three dimensions for DAPeriodicType . The routines to create distributed arrays are collective, so that all processors in the communicator comm must call DACreateXXX().

2.4.2 Local/Global Vectors and Scatters

Each DA object defines the layout of two vectors: a distributed global vector and a local vector that includes room for the appropriate ghost points. The DA object provides information about the size and layout of these vectors, but does not internally allocate any associated storage space for field values. Instead, the user can create vector objects that use the DA layout information with the routines

```
DACreateGlobalVector(DA da,Vec *g);
DACreateLocalVector(DA da,Vec *l);
```

These vectors will generally serve as the building blocks for local and global PDE solutions, etc. If additional vectors with such layout information are needed in a code, they can be obtained by duplicating `l` or `g` via `VecDuplicate`() or `VecDuplicateVecs`(). We emphasize that a distributed array provides the information needed to communicate the ghost value information between processes. In most cases, several different vectors can share the same communication information (or, in other words, can share a given `DA`). The design of the `DA` object makes this easy, as each `DA` operation may operate on vectors of the appropriate `size`, as obtained via `DACreateLocalVector`() and `DACreateGlobalVector`() or as produced by `VecDuplicate`(). As such, the `DA` scatter/gather operations (e.g., `DAGlobalToLocalBegin`()) require vector input/output arguments, as discussed below. PETSc currently provides no container for multiple arrays sharing the same distributed array communication; note, however, that the `dof` parameter handles many cases of interest. At certain stages of many applications, there is a need to work on a local portion of the vector, including the ghost points. This may be done by scattering a global vector into its local parts by using the two-stage commands

```
DAGlobalToLocalBegin(DA da,Vec g,InsertMode iora,Vec l);
DAGlobalToLocalEnd(DA da,Vec g,InsertMode iora,Vec l);
```

which allow the overlap of communication and computation. Since the global and local vectors, given by `g` and `l`, respectively, must be compatible with the distributed array, `da`, they should be generated by `DACreateGlobalVector`() and `DACreateLocalVector`() (or be duplicates of such a vector obtained via `VecDuplicate`()). The `InsertMode` can be either `ADD_VALUES` or `INSERT_VALUES`. One can scatter the local patches into the distributed vector with the command

```
DALocalToGlobal(DA da,Vec l,InsertMode mode,Vec g);
```

Note that this function is not subdivided into beginning and ending phases, since it is purely local. A third type of distributed array scatter is from a local vector (including ghost points that contain irrelevant values) to a local vector with correct ghost point values. This scatter may be done by commands

```
DALocalToLocalBegin(DA da,Vec l1,InsertMode iora,Vec l2);
DALocalToLocalEnd(DA da,Vec l1,InsertMode iora,Vec l2);
```

Since both local vectors, `l1` and `l2`, must be compatible with the distributed array, `da`, they should be generated by `DACreateLocalVector`() (or be duplicates of such vectors obtained via `VecDuplicate`()). The `InsertMode` can be either `ADD_VALUES` or `INSERT_VALUES`. It is possible to directly access the vector scatter contexts (see below) used in the local-to-global (`ltog`), global-to-local (`gtol`), and local-to-local (`ltol`) scatters with the command

```
DAGetScatter(DA da,VecScatter *ltog,VecScatter *gtol,VecScatter *ltol);
```

Most users should not need to use these contexts.

2.4.3 Local (Ghosted) Work Vectors

In most applications the local ghosted vectors are only needed during user "function evaluations". PETSc provides an easy light-weight (requiring essentially no CPU time) way to obtain these work vectors and return them when they are no longer needed. This is done with the routines

```
DAGetLocalVector(DA da,Vec *l);
.... use the local vector l
DARestoreLocalVector(DA da,Vec *l);
```

2.4.4 Accessing the Vector Entries for DA Vectors

PETSc provides an easy way to set values into the DA Vectors and access them using the natural grid indexing. This is done with the routines

```
DAVecGetArray(DA da,Vec l,(void**)array);
.... use the array indexing it with 1 or 2 or 3 dimensions
.... depending on the dimension of the DA
DAVecRestoreArray(DA da,Vec l,(void**)array);
```

The vector l can be either a global vector or a local vector. The `array` is accessed using the usual **global** indexing on the entire grid. For example for a scalar problem in two dimensions one could do

```
PetscScalar **f,**u;
...
DAVecGetArray(DA da,Vec local,(void**)u);
DAVecGetArray(DA da,Vec global,(void**)f);
...
f[i][j] = u[i][j] - ...
...
DAVecRestoreArray(DA da,Vec local,(void**)u);
DAVecRestoreArray(DA da,Vec global,(void**)f);
```

See `${PETSC_DIR}/src/snes/examples/tutorials/ex5.c` for a complete example and See `${PETSC_DIR}/src/snes/examples/tutorials/ex19.c` for an example for a multi-component PDE.

2.4.5 Grid Information

The global indices of the lower left corner of the local portion of the array as well as the local array `size` can be obtained with the commands

```
DAGetCorners(DA da,int *x,int *y,int *z,int *m,int *n,int *p);
DAGetGhostCorners(DA da,int *x,int *y,int *z,int *m,int *n,int *p);
```

The first version excludes any ghost points, while the second version includes them. The routine `DAGetGhostCorners()` deals with the fact that subarrays along boundaries of the problem domain have ghost points only on their interior edges, but not on their boundary edges. When either type of stencil is used, `DA_STENCIL_STAR` or `DA_STENCIL_BOX`, the local vectors (with the ghost points) represent rectangular arrays, including the extra corner elements in the `DA_STENCIL_STAR` case. This configuration provides simple access to the elements by employing two- (or three-) dimensional indexing. The only difference between the two cases is that when `DA_STENCIL_STAR` is used, the extra corner components are *not* scattered between the processors and thus contain undefined values that should *not* be used. To assemble global stiffness matrices, one needs either

- the global node number of each local node including the ghost nodes. This number may be determined by using the command

  ```
  DAGetGlobalIndices(DA da,int *n,int *idx[]);
  ```

  The output argument `n` contains the number of local nodes, including ghost nodes, while `idx` contains a list of the global indices that correspond to the local nodes. Note that the Fortran interface differs slightly; see Section 11.1.3 for details.

Processor 2 | Processor 3

26	27	28	29	30
21	22	23	24	25
16	17	18	19	20
11	12	13	14	15
6	7	8	9	10
1	2	3	4	5

Processor 0 | Processor 1

Natural Ordering

Processor 2 | Processor 3

22	23	24	29	30
19	20	21	27	28
16	17	18	25	26
7	8	9	14	15
4	5	6	12	13
1	2	3	10	11

Processor 0 | Processor 1

PETSc Ordering

Figure 9: Natural Ordering and PETSc Ordering for a 2D Distributed Array (Four Processors)

- or to set up the vectors and matrices so that their entries may be added using the local numbering. This is done by first calling

  `DAGetISLocalToGlobalMapping(DA da,ISLocalToGlobalMapping *map);`

  followed by

  ```
  VecSetLocalToGlobalMapping(Vec x,ISLocalToGlobalMapping map);
  MatSetLocalToGlobalMapping(Vec x,ISLocalToGlobalMapping map);
  ```

  Now entries may be added to the vector and matrix using the local numbering and `VecSetValuesLocal`() and `MatSetValuesLocal`().

Since the global ordering that PETSc uses to manage its parallel vectors (and matrices) does not usually correspond to the "natural" ordering of a two- or three-dimensional array, the `DA` structure provides an application ordering `AO` (see Section 2.3.1) that maps between the natural ordering on a rectangular grid and the ordering PETSc uses to parallize. This ordering context can be obtained with the command

`DAGetAO(DA da,AO *ao);`

In Figure 9 we indicate the orderings for a two-dimensional distributed array, divided among four processors. The example `${PETSC_DIR}/src/snes/examples/tutorials/ex5.c`, illustrates the use of a distributed array in the solution of a nonlinear problem. The analogous Fortran program is `${PETSC_DIR}/src/snes/examples/tutorials/ex5f.F`; see Chapter 5 for a discussion of the nonlinear solvers.

2.5 Software for Managing Vectors Related to Unstructured Grids

2.5.1 Index Sets

To facilitate general vector scatters and gathers used, for example, in updating ghost points for problems defined on unstructured grids, PETSc employs the concept of an index set. An index set, which is a generalization of a set of integer indices, is used to define scatters, gathers, and similar operations on vectors and matrices.

The following command creates a index set based on a list of integers:

```
ISCreateGeneral(MPI_Comm comm,int n,int *indices, IS *is);
```

This routine essentially copies the `n` indices passed to it by the integer array `indices`. Thus, the user should be sure to free the integer array `indices` when it is no longer needed, perhaps directly after the call to `ISCreateGeneral`(). The communicator, `comm`, should consist of all processors that will be using the `IS`. Another standard index set is defined by a starting point (`first`) and a stride (`step`), and can be created with the command

```
ISCreateStride(MPI_Comm comm,int n,int first,int step,IS *is);
```

Index sets can be destroyed with the command

```
ISDestroy(IS is);
```

On rare occasions the user may have to access information directly from an index set. Several commands assist in this process:

```
ISGetSize(IS is,int *size);
ISStrideGetInfo(IS is,int *first,int *stride);
ISGetIndices(IS is,int **indices);
```

The function `ISGetIndices`() returns a pointer to a list of the indices in the index set. For certain index sets, this may be a temporary array of indices created specifically for a given routine. Thus, once the user finishes using the array of indices, the routine

```
ISRestoreIndices(IS is, int **indices);
```

should be called to ensure that the system can free the space it may have used to generate the list of indices.
A blocked version of the index sets can be created with the command

```
ISCreateBlock(MPI_Comm comm,int bs,int n,int *indices, IS *is);
```

This version is used for defining operations in which each element of the index set refers to a block of `bs` vector entries. Related routines analogous to those described above exist as well, including `ISBlockGetIndices`(), `ISBlockGetSize`(), `ISBlockGetBlockSize`(), and `ISBlock`(). See the man pages for details.

2.5.2 Scatters and Gathers

PETSc vectors have full support for general scatters and gathers. One can select any subset of the components of a vector to insert or add to any subset of the components of another vector. We refer to these operations as generalized scatters, though they are actually a combination of scatters and gathers.

To copy selected components from one vector to another, one uses the following set of commands:

```
VecScatterCreate(Vec x,IS ix,Vec y,IS iy,VecScatter *ctx);
VecScatterBegin(Vec x,Vec y,INSERT_VALUES,SCATTER_FORWARD,VecScatter ctx);
VecScatterEnd(Vec x,Vec y,INSERT_VALUES,SCATTER_FORWARD,VecScatter ctx);
VecScatterDestroy(VecScatter ctx);
```

Here `ix` denotes the index set of the first vector, while `iy` indicates the index set of the destination vector. The vectors can be parallel or sequential. The only requirements are that the number of entries in the index set of the first vector, `ix`, equal the number in the destination index set, `iy`, and that the vectors be long enough to contain all the indices referred to in the index sets. The argument `INSERT_VALUES` specifies that the vector elements will be inserted into the specified locations of the destination vector, overwriting any existing values. To add the components, rather than insert them, the user should select the option `ADD_VALUES` instead of `INSERT_VALUES`. To perform a conventional gather operation, the user simply makes the destination index set, `iy`, be a stride index set with a stride of one. Similarly, a conventional scatter can be done with an initial (sending) index set consisting of a stride. The scatter rotines are collective operations (i.e. all processes that own a parallel vector *must* call the scatter routines). When scattering from a parallel vector to sequential vectors, each processor has its own sequential vector that receives values from locations as indicated in its own index set. Similarly, in scattering from sequential vectors to a parallel vector, each processor has its own sequential vector that makes contributions to the parallel vector. *Caution*: When `INSERT_VALUES` is used, if two different processors contribute different values to the same component in a parallel vector, either value may end up being inserted. When `ADD_VALUES` is used, the correct sum is added to the correct location. In some cases one may wish to "undo" a scatter, that is perform the scatter backwards switching the roles of the sender and receiver. This is done by using

```
VecScatterBegin(Vec y,Vec x,INSERT_VALUES,SCATTER_REVERSE,VecScatter ctx);
VecScatterEnd(Vec y,Vec x,INSERT_VALUES,SCATTER_REVERSE,VecScatter ctx);
```

Note that the roles of the first two arguments to these routines must be swapped whenever the `SCATTER_REVERSE` option is used. Once a `VecScatter` object has been created it may be used with any vectors that have the appropriate parallel data layout. That is, one can call `VecScatterBegin`() and `VecScatterEnd`() with different vectors than used in the call to `VecScatterCreate`() so long as they have the same parallel layout (number of elements on each processor are the same). Usually, these "different" vectors would have been obtained via calls to `VecDuplicate`() from the original vectors used in the call to `VecScatterCreate`(). There is no PETSc routine that is the opposite of `VecSetValues`() , that is, VecGetValues(). Instead, the user should create a new vector where the components are to be stored and perform the appropriate vector scatter. For example, if one desires to obtain the values of the 100th and 200th entries of a parallel vector, `p`, one could use a code such as that within Figure 10. In this example, the values of the 100th and 200th components are placed in the array values. In this example each processor now has the 100th and 200th component, but obviously each processor could gather any elements it needed, or none by creating an index set with no entries. The scatter comprises two stages, in order to allow overlap of communication and computation. The introduction of the `VecScatter` context allows the communication patterns for the scatter to be computed once and then reused repeatedly. Generally, even setting up the communication for a scatter requires communication; hence, it is best to reuse such information when possible.

2.5.3 Scattering Ghost Values

The scatters provide a very general method for managing the communication of required ghost values for unstructured grid computations. One scatters the global vector into a local "ghosted" work vector, performs the computation on the local work vectors, and then scatters back into the global solution vector. In the simplest case this may be written as

```
Function: (Input Vec globalin, Output Vec globalout)
VecScatterBegin(Vec globalin,Vec localin,InsertMode INSERT_VALUES,
ScatterMode SCATTER_FORWARD,VecScatter scatter);
VecScatterEnd(Vec globalin,Vec localin,InsertMode INSERT_VALUES,
```

```
Vec p, x; /* initial vector, destination vector */
VecScatter scatter; /* scatter context */
IS from, to; /* index sets that define the scatter */
PetscScalar *values;
int idx_from[] = {100,200}, idx_to[] = {0,1};
VecCreateSeq(PETSC_COMM_SELF,2,&x);
ISCreateGeneral(PETSC_COMM_SELF,2,idx_from,&from);
ISCreateGeneral(PETSC_COMM_SELF,2,idx_to,&to);
VecScatterCreate(p,from,x,to,&scatter);
VecScatterBegin(p,x,INSERT_VALUES,SCATTER_FORWARD,scatter);
VecScatterEnd(p,x,INSERT_VALUES,SCATTER_FORWARD,scatter);
VecGetArray(x,&values);
ISDestroy(from);
ISDestroy(to);
VecScatterDestroy(scatter);
```

Figure 10: Example Code for Vector Scatters

```
ScatterMode SCATTER_FORWARD,VecScatter scatter);
/* For example, do local calculations from localin to localout */
VecScatterBegin(Vec localout,Vec globalout,InsertMode ADD_VALUES,
ScatterMode SCATTER_REVERSE,VecScatter scatter);
VecScatterEnd(Vec localout,Vec globalout,InsertMode ADD_VALUES,
ScatterMode SCATTER_REVERSE,VecScatter scatter);
```

2.5.4 Vectors with Locations for Ghost Values

We recommend that application developers skip this section on a first reading. It contains information about more advanced use of PETSc vectors to improve efficiency slightly. Once an application code is fully debugged and optimized these techniques can be tried to slightly decrease memory use and improve computation speed. There are two minor drawbacks to the basic approach described above:

- the extra memory requirement for the local work vector, `localin`, which duplicates the memory in `globalin`, and
- the extra time required to copy the local values from `localin` to `globalin`.

An alternative approach is to allocate global vectors with space preallocated for the ghost values; this may be done with either

```
VecCreateGhost(MPI_Comm comm,int n,int N,int nghost,int *ghosts,Vec *vv)
```

or

```
VecCreateGhostWithArray(MPI_Comm comm,int n,int N,int nghost,int *ghosts,
PetscScalar *array,Vec *vv)
```

Here `n` is the number of local vector entries, `N` is the number of global entries (or `PETSC_NULL`) and `nghost` is the number of ghost entries. The array `ghosts` is of `size` `nghost` and contains the global vector location for each local ghost location. Using `VecDuplicate`() or `VecDuplicateVecs`() on a ghosted vector will generate additional ghosted vectors. In many ways a ghosted vector behaves just like any other MPI vector created by VecCreateMPI (), the difference is that the ghosted vector has an additional “local” representation that allows one to access the ghost locations. This is done through the call to

```
VecGhostGetLocalForm(Vec g,Vec *l);
```

The vector l is a sequential representation of the parallel vector g that shares the same array space (and hence numerical values); but allows one to access the “ghost” values past “the end of the” array. Note that one access the entries in l using the local numbering of elements and ghosts, while they are accessed in g using the global numbering. A common usage of a ghosted vector is given by

```
VecGhostUpdateBegin(Vec globalin,InsertMode INSERT_VALUES,
ScatterMode SCATTER_FORWARD);
VecGhostUpdateEnd(Vec globalin,InsertMode INSERT_VALUES,
ScatterMode SCATTER_FORWARD);
VecGhostGetLocalForm(Vec globalin,Vec *localin);
VecGhostGetLocalForm(Vec globalout,Vec *localout);
/*
Do local calculations from localin to localout
*/
VecGhostRestoreLocalForm(Vec globalin,Vec *localin);
VecGhostRestoreLocalForm(Vec globalout,Vec *localout);
VecGhostUpdateBegin(Vec globalout,InsertMode ADD_VALUES,
ScatterMode SCATTER_REVERSE);
VecGhostUpdateEnd(Vec globalout,InsertMode ADD_VALUES,
ScatterMode SCATTER_REVERSE);
```

The routines VecGhostUpdateBegin/End() are equivalent to the routines VecScatterBegin/End() above except that since they are scattering into the ghost locations, they do not need to copy the local vector values, which are already in place. In addition, the user does not have to allocate the local work vector, since the ghosted vector already has allocated slots to contain the ghost values. The input arguments INSERT_VALUES and SCATTER_FORWARD cause the ghost values to be correctly updated from the appropriate processor. The arguments ADD_VALUES and SCATTER_REVERSE update the “local” portions of the vector from all the other processors’ ghost values. This would be appropriate, for example, when performing a finite element assembly of a load vector. Section 3.5 discusses the important topic of partitioning an unstructured grid.

Chapter 3

Matrices

PETSc provides a variety of matrix implementations because no single matrix format is appropriate for all problems. Currently we support dense storage and compressed sparse row storage (both sequential and parallel versions), as well as several specialized formats. Additional formats can be added. This chapter describes the basics of using PETSc matrices in general (regardless of the particular format chosen) and discusses tips for efficient use of the several simple uniprocessor and parallel matrix types. The use of PETSc matrices involves the following actions: create a particular type of matrix, insert values into it, process the matrix, use the matrix for various computations, and finally destroy the matrix. The application code does not need to know or care about the particular storage formats of the matrices.

3.1 Creating and Assembling Matrices

The simplest routine for forming a PETSc matrix, `A`, is

`MatCreate` (MPI_Comm `comm` ,int m,int n,int M,int N,`Mat` *A)

This routine generates a sequential matrix when running one process and a parallel matrix for two or more processes; the particular matrix format is set by the user via options database commands. The user specifies either the global matrix dimensions, given by `M` and `N` or the local dimensions, given by `m` and `n` while PETSc completely controls memory allocation. This routine facilitates switching among various matrix types, for example, to determine the format that is most efficient for a certain application. By default, `MatCreate` () employs the sparse AIJ format, which is discussed in detail Section 3.1.1. See the manual pages for further information about available matrix formats. To insert or add entries to a matrix, one can call a variant of `MatSetValues`, either

`MatSetValues` (`Mat` A,int m,int im[],int n,int in[],`PetscScalar` *values,INSERT_VALUES);

or

`MatSetValues` (`Mat` A,int m,int im[],int n,int in[],`PetscScalar` *values,ADD_VALUES);

This routine inserts or adds a logically dense subblock of dimension `m*n` into the matrix. The integer indices `im` and `in`, respectively, indicate the global row and column numbers to be inserted. `MatSetValues` () uses the standard C convention, where the row and column matrix indices begin with zero *regardless of the storage format employed*. The array `values` is logically two-dimensional, containing the values that are to be inserted. By default the values are given in row major order, which is the opposite of the Fortran convention. To allow the insertion of values in column major order, one can call the command

`MatSetOption` (`Mat` A,MAT_COLUMN_ORIENTED);

Warning: Several of the sparse implementations do *not* currently support the column-oriented option. This notation should not be a mystery to anyone. For example, to insert one matrix into another when using Matlab, one uses the command `A(im,in) = B;` where `im` and `in` contain the indices for the rows and columns. This action is identical to the calls above to `MatSetValues` (). When using the block compressed sparse row matrix format (MATSEQBAIJ or MATMPIBAIJ), one can insert elements more efficiently using the block variant, `MatSetValuesBlocked` (). The function `MatSetOption` () accepts several other inputs; see the manual page for details. We discuss two of these options, which are related to the efficiency of the assembly process. To indicate to PETSc that the row (`im`) or column (`in`) indices set with `MatSetValues` () are sorted (in increasing order), one uses the command

```
MatSetOption(Mat A,MAT_ROWS_SORTED);
```

or

```
MatSetOption(Mat A,MAT_COLUMNS_SORTED);
```

Note that these flags indicate the format of the data passed in with `MatSetValues` (); they do not have anything to do with how the sparse matrix data is stored internally in PETSc. After the matrix elements have been inserted or added into the matrix, they must be processed (also called assembled) before they can be used. The routines for matrix processing are

```
MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);
```

By placing other code between these two calls, the user can perform computations while messages are in transit. Calls to `MatSetValues` () with the `INSERT_VALUES` and `ADD_VALUES` options *cannot* be mixed without intervening calls to the assembly routines. For such intermediate assembly calls the second routine argument typically should be `MAT_FLUSH_ASSEMBLY`, which omits some of the work of the full assembly process. `MAT_FINAL_ASSEMBLY` is required only in the last matrix assembly before a matrix is used. Even though one may insert values into PETSc matrices without regard to which processor eventually stores them, for efficiency reasons we usually recommend generating most entries on the processor where they are destined to be stored. To help the application programmer with this task for matrices that are distributed across the processors by ranges, the routine

```
MatGetOwnershipRange(Mat A,int *first_row,int *last_row);
```

informs the user that all rows from `first_row` to `last_row-1` (since the value returned in `last_row` is one more than the global index of the last local row) will be stored on the local processor. In the sparse matrix implementations, once the assembly routines have been called, the matrices are compressed and can be used for matrix-vector multiplication, etc. Inserting new values into the matrix at this point will be expensive, since it requires copies and possible memory allocation. Thus, whenever possible one should completely set the values in the matrices before calling the final assembly routines.

If one wishes to repeatedly assemble matrices that retain the same nonzero pattern (such as within a nonlinear or time-dependent problem), the option

```
MatSetOption(Mat mat,MAT_NO_NEW_NONZERO_LOCATIONS);
```

should be specified after the first matrix has been fully assembled. This option ensures that certain data structures and communication information will be reused (instead of regenerated) during successive steps, thereby increasing efficiency. See `${PETSC_DIR}/src/sles/examples/tutorials/ex5.c` for a simple example of solving two linear systems that use the same matrix data structure.

3.1.1 Sparse Matrices

The default matrix representation within PETSc is the general sparse AIJ format (also called the Yale sparse matrix format or compressed sparse row format, CSR). This section discusses tips for *efficiently* using this matrix format for large-scale applications. Additional formats (such as block compressed row and block diagonal storage, which are generally much more efficient for problems with multiple degrees of freedom per node) are discussed below. Beginning users need not concern themselves initially with such details and may wish to proceed directly to Section 3.2. However, when an application code progresses to the point of tuning for efficiency and/or generating timing results, it is *crucial* to read this information.

Sequential AIJ Sparse Matrices

In the PETSc AIJ matrix formats, we store the nonzero elements by rows, along with an array of corresponding column numbers and an array of pointers to the beginning of each row. Note that the diagonal matrix entries are stored with the rest of the nonzeros (not separately).

To create a sequential AIJ sparse matrix, `A`, with `m` rows and `n` columns, one uses the command

`MatCreateSeqAIJ` (PETSC_COMM_SELF,int m,int n,int nz,int *nnz,`Mat` *A);

where `nz` or `nnz` can be used to preallocate matrix memory, as discussed below. The user can set `nz=0` and `nnz=PETSC_NULL` for PETSc to control all matrix memory allocation. The sequential and parallel AIJ matrix storage formats by default employ *i-nodes* (identical nodes) when possible. We search for consecutive rows with the same nonzero structure, thereby reusing matrix information for increased efficiency. Related options database keys are `-mat_aij_no_inode` (do not use inodes) and `-mat_aij_inode_limit <limit>` (set inode limit (max limit=5)). Note that problems with a single degree of freedom per grid node will automatically not use I-nodes. By default the internal data representation for the AIJ formats employs zero-based indexing. For compatibility with standard Fortran storage, thus enabling use of external Fortran software packages such as SPARSKIT, the option `-mat_aij_oneindex` enables one-based indexing, where the stored row and column indices begin at one, not zero. All user calls to PETSc routines, regardless of this option, use zero-based indexing.

Preallocation of Memory for Sequential AIJ Sparse Matrices

The dynamic process of allocating new memory and copying from the old storage to the new is *intrinsically very expensive*. Thus, to obtain good performance when assembling an AIJ matrix, it is crucial to preallocate the memory needed for the sparse matrix. The user has two choices for preallocating matrix memory via `MatCreateSeqAIJ()`.

One can use the scalar `nz` to specify the expected number of nonzeros for each row. This is generally fine if the number of nonzeros per row is roughly the same throughout the matrix (or as a quick and easy first step for preallocation). If one underestimates the actual number of nonzeros in a given row, then during the assembly process PETSc will automatically allocate additional needed space. However, this extra memory allocation can slow the computation, If different rows have very different numbers of nonzeros, one should attempt to indicate (nearly) the exact number of elements intended for the various rows with the optional array, `nnz` of length `m`, where `m` is the number of rows, for example

```
int nnz[m];
nnz[0] = <nonzeros in row 0>
nnz[1] = <nonzeros in row 1>
....
nnz[m-1] = <nonzeros in row m-1>
```

In this case, the assembly process will require no additional memory allocations if the nnz estimates are correct. If, however, the nnz estimates are incorrect, PETSc will automatically obtain the additional needed space, at a slight loss of efficiency. Using the array nnz to preallocate memory is especially important for efficient matrix assembly if the number of nonzeros varies considerably among the rows. One can generally set nnz either by knowing in advance the problem structure (e.g., the stencil for finite difference problems on a structured grid) or by precomputing the information by using a segment of code similar to that for the regular matrix assembly. The overhead of determining the nnz array will be quite small compared with the overhead of the inherently expensive mallocs and moves of data that are needed for dynamic allocation during matrix assembly. Always guess high if exact value is not known (since extra space is cheaper than too little). Thus, when assembling a sparse matrix with very different numbers of nonzeros in various rows, one could proceed as follows for finite difference methods:

- Allocate integer array nnz.
- Loop over grid, counting the expected number of nonzeros for the row(s)
  associated with the various grid points.
- Create the sparse matrix via MatCreateSeqAIJ () or alternative.
- Loop over the grid, generating matrix entries and inserting in matrix via MatSetValues ().

For (vertex-based) finite element type calculations, an analogous procedure is as follows:

- Allocate integer array nnz.
- Loop over vertices, computing the number of neighbor vertices, which determines the
  number of nonzeros for the corresponding matrix row(s).
- Create the sparse matrix via MatCreateSeqAIJ () or alternative.
- Loop over elements, generating matrix entries and inserting in matrix via MatSetValues ().

The -log_info option causes the routines MatAssemblyBegin () and MatAssemblyEnd () to print information about the success of the preallocation. Consider the following example for the MATSEQAIJ matrix format:

MatAssemblyEnd_SeqAIJ:Matrix size 10 X 10; storage space:20 unneeded, 100 used
MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues is 0

The first line indicates that the user preallocated 120 spaces but only 100 were used. The second line indicates that the user preallocated enough space so that PETSc did not have to internally allocate additional space (an expensive operation). In the next example the user did not preallocate sufficient space, as indicated by the fact that the number of mallocs is very large (bad for efficiency):

MatAssemblyEnd_SeqAIJ:Matrix size 10 X 10; storage space:47 unneeded, 1000 used
MatAssemblyEnd_SeqAIJ:Number of mallocs during MatSetValues is 40000

Although at first glance such procedures for determining the matrix structure in advance may seem unusual, they are actually very efficient because they alleviate the need for dynamic construction of the matrix data structure, which can be very expensive.

Parallel AIJ Sparse Matrices

Parallel sparse matrices with the AIJ format can be created with the command

MatCreateMPIAIJ (MPI_Comm comm ,int m,int n,int M,int N,int d_nz,
int *d_nnz, int o_nz,int *o_nnz,Mat *A);

A is the newly created matrix, while the arguments m, M, and N, indicate the number of local rows and the number of global rows and columns, respectively. In the PETSc partitioning scheme, all the matrix columns are local and n is the number of columns corresponding to local part of a parallel vector. Either the local or global parameters can be replaced with PETSC_DECIDE, so that PETSc will determine them. The matrix is stored with a fixed number of rows on each processor, given by m, or determined by PETSc if m is PETSC_DECIDE. If PETSC_DECIDE is not used for the arguments m and n, then the user must ensure that they are chosen to be compatible with the vectors. To do this, one first considers the matrix-vector product $y = Ax$. The m that is used in the matrix creation routine MatCreateMPIAIJ () must match the local size used in the vector creation routine VecCreateMPI () for y. Likewise, the n used must match that used as the local size in VecCreateMPI () for x.

The user must set d_nz=0, o_nz=0, d_nnz=PETSC_NULL, and o_nnz=PETSC_NULL for PETSc to control dynamic allocation of matrix memory space. Analogous to nz and nnz for the routine MatCreateSeqAIJ (), these arguments optionally specify nonzero information for the diagonal (d_nz and d_nnz) and off-diagonal (o_nz and o_nnz) parts of the matrix. For a square global matrix, we define each processor's diagonal portion to be its local rows and the corresponding columns (a square submatrix); each processor's off-diagonal portion encompasses the remainder of the local matrix (a rectangular submatrix). The rank in the MPI communicator determines the absolute ordering of the blocks. That is, the process with rank 0 in the communicator given to MatCreateMPIAIJ contains the top rows of the matrix; the ith process in that communicator contains the ith block of the matrix.

Preallocation of Memory for Parallel AIJ Sparse Matrices

As discussed above, preallocation of memory is critical for achieving good performance during matrix assembly, as this reduces the number of allocations and copies required. We present an example for three processors to indicate how this may be done for the MATMPIAIJ matrix format. Consider the 8 by 8 matrix, which is partitioned by default with three rows on the first processor, three on the second and two on the third.

$$\left(\begin{array}{ccc|ccc|cc}
1 & 2 & 0 & 0 & 3 & 0 & 0 & 4 \\
0 & 5 & 6 & 7 & 0 & 0 & 8 & 0 \\
9 & 0 & 10 & 11 & 0 & 0 & 12 & 0 \\
\hline
13 & 0 & 14 & 15 & 16 & 17 & 0 & 0 \\
0 & 18 & 0 & 19 & 20 & 21 & 0 & 0 \\
0 & 0 & 0 & 22 & 23 & 0 & 24 & 0 \\
\hline
25 & 26 & 27 & 0 & 0 & 28 & 29 & 0 \\
30 & 0 & 0 & 31 & 32 & 33 & 0 & 34
\end{array}\right)$$

The "diagonal" submatrix, d, on the first processor is given by

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 5 & 6 \\ 9 & 0 & 10 \end{pmatrix},$$

while the "off-diagonal" submatrix, o, matrix is given by

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 4 \\ 7 & 0 & 0 & 8 & 0 \\ 11 & 0 & 0 & 12 & 0 \end{pmatrix}.$$

For the first processor one could set d_nz to 2 (since each row has 2 nonzeros) or, alternatively, set d_nnz to $\{2,2,2\}$. The o_nz could be set to 2 since each row of the o matrix has 2 nonzeros, or o_nnz could be set to $\{2,2,2\}$. For the second processor the d submatrix is given by

$$\begin{pmatrix} 15 & 16 & 17 \\ 19 & 20 & 21 \\ 22 & 23 & 0 \end{pmatrix}.$$

Thus, one could set `d_nz` to 3, since the maximum number of nonzeros in each row is 3, or alternatively one could set `d_nnz` to {3,3,2}, thereby indicating that the first two rows will have 3 nonzeros while the third has 2. The corresponding `o` submatrix for the second processor is

$$\begin{pmatrix} 13 & 0 & 14 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 \\ 0 & 0 & 0 & 24 & 0 \end{pmatrix}$$

so that one could set `o_nz` to 2 or `o_nnz` to {2,1,1}. Note that the user never directly works with the `d` and `o` submatrices, except when preallocating storage space as indicated above. Also, the user need not preallocate exactly the correct amount of space; as long as a sufficiently close estimate is given, the high efficiency for matrix assembly will remain.

As described above, the option `-log_info` will print information about the success of preallocation during matrix assembly. For the MATMPIAIJ format, PETSc will also list the number of elements owned by on each processor that were generated on a different processor. For example, the statements

```
MatAssemblyBegin_MPIAIJ:Number of off processor values 10
MatAssemblyBegin_MPIAIJ:Number of off processor values 7
MatAssemblyBegin_MPIAIJ:Number of off processor values 5
```

indicate that very few values have been generated on different processors. On the other hand, the statements

```
MatAssemblyBegin_MPIAIJ:Number of off processor values 100000
MatAssemblyBegin_MPIAIJ:Number of off processor values 77777
```

indicate that many values have been generated on the "wrong" processors. This situation can be very inefficient, since the transfer of values to the "correct" processor is generally expensive. By using the command `MatGetOwnershipRange()` in application codes, the user should be able to generate most entries on the owning processor. *Note*: It is fine to generate some entries on the "wrong" processor. Often this can lead to cleaner, simpler, less buggy codes. One should never make code overly complicated in order to generate all values locally. Rather, one should organize the code in such a way that *most* values are generated locally.

3.1.2 Dense Matrices

PETSc provides both sequential and parallel dense matrix formats, where each processor stores its entries in a column-major array in the usual Fortran style. To create a sequential, dense PETSc matrix, `A` of dimensions `m` by `n`, the user should call

```
MatCreateSeqDense(PETSC_COMM_SELF,int m,int n,PetscScalar *data,Mat *A);
```

The variable `data` enables the user to optionally provide the location of the data for matrix storage (intended for Fortran users who wish to allocate their own storage space). Most users should merely set `data` to `PETSC_NULL` for PETSc to control matrix memory allocation. To create a parallel, dense matrix, `A`, the user should call

```
MatCreateMPIDense(MPI_Comm comm,int m,int n,int M,int N,PetscScalar *data,Mat *A)
```

The arguments `m`, `n`, `M`, and `N`, indicate the number of local rows and columns and the number of global rows and columns, respectively. Either the local or global parameters can be replaced with PETSC_DECIDE, so that PETSc will determine them. The matrix is stored with a fixed number of rows on each processor, given by `m`, or determined by PETSc if `m` is PETSC_DECIDE.

PETSc does not provide parallel dense direct solvers. Our focus is on sparse iterative solvers.

3.2 Basic Matrix Operations

Table 2 summarizes basic PETSc matrix operations. We briefly discuss a few of these routines in more detail below. The parallel matrix can multiply a vector with `n` local entries, returning a vector with `m` local entries. That is, to form the product

`MatMult` (`Mat` A,`Vec` x,`Vec` y);

the vectors `x` and `y` should be generated with

`VecCreateMPI` (MPI_Comm `comm` ,n,N,&x);
`VecCreateMPI` (MPI_Comm `comm` ,m,M,&y);

By default, if the user lets PETSc decide the number of components to be stored locally (by passing in PETSC_DECIDE as the second argument to `VecCreateMPI` () or using `VecCreate` ()), vectors and matrices of the same dimension are automatically compatible for parallel matrix-vector operations. Along with the matrix-vector multiplication routine, there is a version for the transpose of the matrix,

`MatMultTranspose` (`Mat` A,`Vec` x,`Vec` y);

There are also versions that add the result to another vector:

`MatMultAdd` (`Mat` A,`Vec` x,`Vec` y,`Vec` w);
`MatMultTransposeAdd` (`Mat` A,`Vec` x,`Vec` y,`Vec` w);

These routines, respectively, produce $w = A * x + y$ and $w = A^T * x + y$. In C it is legal for the vectors `y` and `w` to be identical. In Fortran, this situation is forbidden by the language standard, but we allow it anyway. One can print a matrix (sequential or parallel) to the screen with the command

`MatView` (`Mat` mat,PETSC_VIEWER_STDOUT_WORLD);

Other viewers can be used as well. For instance, one can draw the nonzero stucture of the matrix into the default X-window with the command

`MatView` (`Mat` mat,PETSC_VIEWER_DRAW_WORLD);

Also one can use

`MatView` (`Mat` mat,`PetscViewer` viewer);

where `viewer` was obtained with PetscViewerDrawOpenX(). Additional viewers and options are given in the `MatView` () man page and Section 14.2. The `NormType` argument to `MatNorm` () is one of `NORM_1`, `NORM_INFINITY`, and `NORM_FROBENIUS`.

3.3 Matrix-Free Matrices

Some people like to use matrix-free methods, which do not require explicit storage of the matrix, for the numerical solution of partial differential equations. To support matrix-free methods in PETSc, one can use the following command to create a `Mat` structure without ever actually generating the matrix:

`MatCreateShell` (MPI_Comm `comm` ,int m,int n,int M,int N,void *ctx,`Mat` *mat);

Here `M` and `N` are the global matrix dimensions (rows and columns), `m` and `n` are the local matrix dimensions, and `ctx` is a pointer to data needed by any user-defined shell matrix operations; the manual page has additional details about these parameters. Most matrix-free algorithms require only the application of the linear operator to a vector. To provide this action, the user must write a routine with the calling sequence

Function Name	Operation
MatAXPY (PetscScalar *a,Mat X, Mat Y,MatStructure);	$Y = Y + a * X$
MatMult (Mat A,Vec x, Vec y);	$y = A * x$
MatMultAdd (Mat A,Vec x, Vec y,Vec z);	$z = y + A * x$
MatMultTranspose (Mat A,Vec x, Vec y);	$y = A^T * x$
MatMultTransposeAdd (Mat A,Vec x, Vec y,Vec z);	$z = y + A^T * x$
MatNorm (Mat A,NormType type, double *r);	$r = \|\|A\|\|_{type}$
MatDiagonalScale (Mat A,Vec l,Vec r);	$A = \text{diag}(l) * A * \text{diag}(r)$
MatScale (PetscScalar *a,Mat A);	$A = a * A$
MatConvert (Mat A,MatType type,Mat *B);	$B = A$
MatCopy (Mat A,Mat B,MatStructure);	$B = A$
MatGetDiagonal (Mat A,Vec x);	$x = \text{diag}(A)$
MatTranspose (Mat A,Mat * B);	$B = A^T$
MatZeroEntries (Mat A);	$A = 0$
MatShift (PetscScalar *a,Mat Y);	$Y = Y + a * I$

Table 2: PETSc Matrix Operations

UserMult(Mat mat,Vec x,Vec y);

and then associate it with the matrix, `mat`, by using the command

MatShellSetOperation (Mat mat,MatOperation MATOP_MULT,
(void(*)(void)) int (*UserMult)(Mat ,Vec ,Vec));

Here `MATOP_MULT` is the name of the operation for matrix-vector multiplication. Within each user-defined routine (such as `UserMult()`), the user should call MatShellGetContext () to obtain the user-defined context, `ctx`, that was set by MatCreateShell (). This shell matrix can be used with the iterative linear equation solvers discussed in the following chapters. The routine MatShellSetOperation () can be used to set any other matrix operations as well. The file `${PETSC_DIR}/include/petscmat.h` provides a complete list of matrix operations, which have the form `MATOP_<OPERATION>`, where `<OPERATION>` is the name (in all capital letters) of the user interface routine (for example, MatMult () $\rightarrow$ `MATOP_MULT`). All user-provided functions have the same calling sequence as the usual matrix interface routines, since the user-defined functions are intended to be accessed through the same interface, e.g., `MatMult(Mat,Vec,Vec)` $\rightarrow$ `UserMult(Mat,Vec,Vec)`. The final argument for MatShellSetOperation () needs to be cast to a `void *`, since the final argument could (depending on the MatOperation) be a variety of different functions. Note that MatShellSetOperation () can also be used as a "backdoor" means of introducing user-defined changes in matrix operations for other storage formats (for example, to override the default LU factorization routine supplied within PETSc for the MATSEQAIJ format). However, we urge anyone who introduces such changes to use caution, since it would be very easy to accidentally create a bug in the new routine that could affect other routines as well. See also Section 5.5 for details on one set of helpful utilities for using the matrix-free approach for nonlinear solvers.

3.4 Other Matrix Operations

In many iterative calculations (for instance, in a nonlinear equations solver), it is important for efficiency purposes to reuse the nonzero structure of a matrix, rather than determining it anew every time the matrix is generated. To retain a given matrix but reinitialize its contents, one can employ

MatZeroEntries(Mat A);

This routine will zero the matrix entries in the data structure but keep all the data that indicates where the nonzeros are located. In this way a new matrix assembly will be much less expensive, since no memory allocations or copies will be needed. Of course, one can also explicitly set selected matrix elements to zero by calling MatSetValues(). In the numerical solution of elliptic partial differential equations, it can be cumbersome to deal with Dirichlet boundary conditions. In particular, one would like to assemble the matrix without regard to boundary conditions and then at the end apply the Dirichlet boundary conditions. In numerical analysis classes this process is usually presented as moving the known boundary conditions to the right-hand side and then solving a smaller linear system for the interior unknowns. Unfortunately, implementing this requires extracting a large submatrix from the original matrix and creating its corresponding data structures. This process can be expensive in terms of both time and memory.

One simple way to deal with this difficulty is to replace those rows in the matrix associated with known boundary conditions, by rows of the identity matrix (or some scaling of it). This action can be done with the command

MatZeroRows(Mat A,IS rows,PetscScalar *diag_value);

For sparse matrices this removes the data structures for certain rows of the matrix. If the pointer `diag_value` is `PETSC_NULL`, it even removes the diagonal entry. If the pointer is not null, it uses that given value at the pointer location in the diagonal entry of the eliminated rows.

Another matrix routine of interest is

MatConvert(Mat mat,MatType newtype,Mat *M)

which converts the matrix `mat` to new matrix, `M`, that has either the same or different format. Set `newtype` to MATSAME to copy the matrix, keeping the same matrix format. See `${PETSC_DIR}/include/petscmat.h` for other available matrix types; standard ones are MATSEQDENSE, MATSEQAIJ, MATMPIAIJ, MATMPIROWBS, MATSEQBDIAG, MATMPIBDIAG, MATSEQBAIJ, and MATMPIBAIJ. In certain applications it may be necessary for application codes to directly access elements of a matrix. This may be done by using the the command (for local rows only)

MatGetRow(Mat A,int row, int *ncols,int (*cols)[],PetscScalar (*vals)[]);

The argument `ncols` returns the number of nonzeros in that row, while `cols` and `vals` returns the column indices (with indices starting at zero) and values in the row. If only the column indices are needed (and not the corresponding matrix elements), one can use `PETSC_NULL` for the `vals` argument. Similarly, one can use `PETSC_NULL` for the `cols` argument. The user can only examine the values extracted with MatGetRow(); the values *cannot* be altered. To change the matrix entries, one must use MatSetValues(). Once the user has finished using a row, he or she *must* call

MatRestoreRow(Mat A,int row,int *ncols,int **cols,PetscScalar **vals);

to free any space that was allocated during the call to MatGetRow().

3.5 Partitioning

For almost all unstructured grid computation, the distribution of portions of the grid across the processor's work load and memory can have a very large impact on performance. In most PDE calculations the grid partitioning and distribution across the processors can (and should) be done in a "pre-processing" step before the numerical computations. However, this does not mean it need be done in a separate, sequential program, rather it should be done before one sets up the parallel grid data structures in the actual program. PETSc provides an interface to the ParMETIS (developed by George Karypis; see the docs/installation/index.htm file for directions on installing PETSc to use ParMETIS) to allow the partitioning to be done in parallel. PETSc does not currently provide directly support for dynamic repartitioning, load balancing by migrating matrix entries between processors, etc. For problems that require mesh refinement, PETSc uses the "rebuild the data structure" approach, as opposed to the "maintain dynamic data structures that support the insertion/deletion of additional vector and matrix rows and columns entries" approach. Partitioning in PETSc is organized around the `MatPartitioning` object. One first creates a parallel matrix that contains the connectivity information about the grid (or other graph-type object) that is to be partitioned. This is done with the command

`MatCreateMPIAdj` (MPI_Comm `comm` ,int mlocal,int n,const int ia[],const int ja[],
int *weights,`Mat` *Adj);

The argument `mlocal` indicates the number of rows of the graph being provided by the given processor, `n` is the total number of columns; equal to the sum of all the `mlocal`. The arguments `ia` and `ja` are the row pointers and column pointers for the given rows, these are the usual format for parallel compressed sparse row storage, using indices starting at 0, **not** 1.

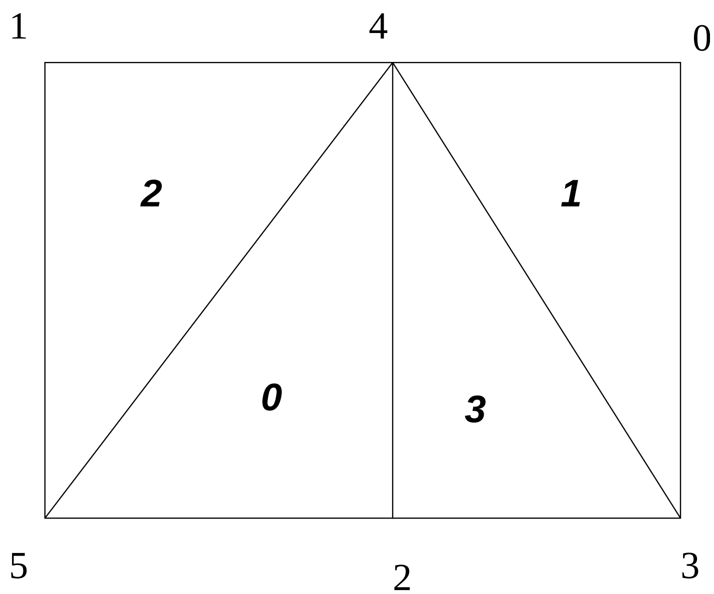

Figure 11: Numbering on Simple Unstructured Grid

This, of course, assumes that one has already distributed the grid (graph) information among the processors. The details of this initial distribution is not important; it could be simply determined by assigning to the first processor the first n_0 nodes from a file, the second processor the next n_1 nodes, etc. For example, we demonstrate the form of the `ia` and `ja` for a triangular grid where we
(1) partition by element (triangle)

- Processor 0, $mlocal = 2, n = 4, ja = \{2, 3, |3\}, ia = \{0, 2, 3\}$
- Processor 1, $mlocal = 2, n = 4, ja = \{0, |0, 1\}, ia = \{0, 1, 3\}$

Note that elements are not connected to themselves and we only indicate edge connections (in some contexts single vertex connections between elements may also be included). We use a | above to denote the transition between rows in the matrix. and (2) partition by vertex.

- Processor 0, $mlocal = 3, n = 6, ja = \{3, 4, |4, 5, |3, 4, 5\}, ia = \{0, 2, 4, 7\}$

- Processor 1, $mlocal = 3, n = 6, ja = \{0, 2, 4, |0, 1, 2, 3, 5, |1, 2, 4\}, ia = \{0, 3, 8, 11\}$.

Once the connectivity matrix has been created the following code will generate the renumbering required for the new partition

```
MatPartitioningCreate(MPI_Comm comm,MatPartitioning *part);
MatPartitioningSetAdjacency(MatPartitioning part,Mat Adj);
MatPartitioningSetFromOptions(MatPartitioning part);
MatPartitioningApply(MatPartitioning part,IS *is);
MatPartitioningDestroy(MatPartitioning part);
MatDestroy(Mat Adj);
ISPartitioningToNumbering(IS is,IS *isg);
```

The resulting `isg` contains for each local node the new global number of that node. The resulting `is` contains the new processor number that each local node has been assigned to. Now that a new numbering of the nodes has been determined one must renumber all the nodes and migrate the grid information to the correct processor. The command

```
AOCreateBasicIS(isg,PETSC_NULL,&ao);
```

generates, see Section 2.3.1, an `AO` object that can be used in conjunction with the `is` and `gis` to move the relevant grid information to the correct processor and renumber the nodes etc.

PETSc does not currently provide tools that completely manage the migration and node renumbering, since it will be dependent on the particular data structure you use to store the grid information and the type of grid information that you need for your application. We do plan to include more support for this in the future, but designing the appropriate user interface and providing a scalable implementation that can be used for a wide variety of different grids requires a great deal of time.

Chapter 4

SLES: Linear Equations Solvers

The object SLES is the heart of PETSc, because it provides uniform and efficient access to all of the package's linear system solvers, including parallel and sequential, direct and iterative. SLES is intended for solving nonsingular systems of the form

$$Ax = b, \tag{4.1}$$

where A denotes the matrix representation of a linear operator, b is the right-hand-side vector, and x is the solution vector. SLES uses the same calling sequence for both direct and iterative solution of a linear system. In addition, particular solution techniques and their associated options can be selected at runtime. The combination of a Krylov subspace method and a preconditioner is at the center of most modern numerical codes for the iterative solution of linear systems. See, for example, [7] for an overview of the theory of such methods. SLES creates a simplified interface to the lower-level KSP and PC modules within the PETSc package. The KSP package, discussed in Section 4.3, provides many popular Krylov subspace iterative methods; the PC module, described in Section 4.4, includes a variety of preconditioners. Although both KSP and PC can be used directly, users should employ the interface of SLES .

4.1 Using SLES

To solve a linear system with SLES , one must first create a solver context with the command

```
SLESCreate(MPI_Comm comm,SLES *sles);
```

Here `comm` is the MPI communicator, and `sles` is the newly formed solver context. Before actually solving a linear system with SLES , the user must call the following routine to set the matrices associated with the linear system:

```
SLESSetOperators(SLES sles,Mat Amat,Mat Pmat,MatStructure flag);
```

The argument `Amat`, representing the matrix that defines the linear system, is a symbolic place holder for any kind of matrix. In particular, SLES *does* support matrix-free methods. The routine `MatCreateShell`() in Section 3.3 provides further information regarding matrix-free methods. Typically the *preconditioning matrix* (i.e., the matrix from which the preconditioner is to be constructed), `Pmat`, is the same as the matrix that defines the linear system, `Amat`; however, occasionally these matrices differ (for instance, when a preconditioning matrix is obtained from a lower order method than that employed to form the linear system matrix). The argument `flag` can be used to eliminate unnecessary work when repeatedly solving linear systems of the same `size` with the same preconditioning method; when solving just one linear system, this flag is ignored. The user can set `flag` as follows:

- SAME_NONZERO_PATTERN - the preconditioning matrix has the same nonzero structure during successive linear solves,
- DIFFERENT_NONZERO_PATTERN - the preconditioning matrix does not have the same nonzero structure during successive linear solves,
- SAME_PRECONDITIONER - the preconditioner matrix is identical to that of the previous linear solve.

If the structure of a matrix is not known a priori, one should use the flag DIFFERENT_NONZERO_PATTERN. Much of the power of SLES can be accessed through the single routine

SLESSetFromOptions(SLES sles);

This routine accepts the options -h and -help as well as any of the KSP and PC options discussed below. To solve a linear system, one merely executes the command

SLESSolve(SLES sles,Vec b,Vec x,int *its);

where b and x respectively denote the right-hand-side and solution vectors. On return, the parameter its contains either the iteration number at which convergence was successfully reached, or the *negative* of the iteration at which divergence or breakdown was detected. Section 4.3.2 gives more details regarding convergence testing. Note that multiple linear solves can be performed by the same SLES context. Once the SLES context is no longer needed, it should be destroyed with the command

SLESDestroy(SLES sles);

The above procedure is sufficient for general use of the SLES package. One additional step is required for users who wish to customize certain preconditioners (e.g., see Section 4.4.4) or to log certain performance data using the PETSc profiling facilities (as discussed in Chapter 12). In this case, the user can optionally explicitly call

SLESSetUp(SLES sles,Vec b,Vec x);

before calling SLESSolve() to perform any setup required for the linear solvers. The explicit call of this routine enables the separate monitoring of any computations performed during the set up phase, such as incomplete factorization for the ILU preconditioner. The default solver within SLES is restarted GMRES, preconditioned for the uniprocessor case with ILU(0), and for the multiprocessor case with the block Jacobi method (with one block per processor, each of which is solved with ILU(0)). A variety of other solvers and options are also available. To allow application programmers to set any of the preconditioner or Krylov subspace options directly within the code, we provide routines that extract the PC and KSP contexts,

SLESGetPC(SLES sles,PC *pc);
SLESGetKSP(SLES sles,KSP *ksp);

The application programmer can then directly call any of the PC or KSP routines to modify the corresponding default options.

To solve a linear system with a direct solver (currently supported by PETSc for sequential matrices, and by several external solvers through PETSc interfaces (see Section 4.6)) one may use the options -pc_type lu (see below). By default, if a direct solver is used, the factorization is *not* done in-place. This approach prevents the user from the unexpected surprise of having a corrupted matrix after a linear solve. The routine PCLUSetUseInPlace(), discussed below, causes factorization to be done in-place.

4.2 Solving Successive Linear Systems

When solving multiple linear systems of the same size with the same method, several options are available. To solve successive linear systems having the *same* preconditioner matrix (i.e., the same data structure with exactly the same matrix elements) but different right-hand-side vectors, the user should simply call SLESSolve () multiple times. The preconditioner setup operations (e.g., factorization for ILU) will be done during the first call to SLESSolve() only; such operations will *not* be repeated for successive solves. To solve successive linear systems that have *different* preconditioner matrices (i.e., the matrix elements and/or the matrix data structure change), the user *must* call SLESSetOperators () and SLESSolve () for each solve. See Section 4.1 for a description of various flags for SLESSetOperators () that can save work for such cases.

4.3 Krylov Methods

The Krylov subspace methods accept a number of options, many of which are discussed below. First, to set the Krylov subspace method that is to be used, one calls the command

```
KSPSetType (KSP ksp,KSPType method);
```

The type can be one of KSPRICHARDSON, KSPCHEBYCHEV, KSPCG, KSPGMRES, KSPTCQMR, KSPBCGS, KSPCGS, KSPTFQMR, KSPCR, KSPLSQR, KSPBICG, or KSPPREONLY. The KSP method can also be set with the options database command -ksp_type, followed by one of the options richardson, chebychev, cg, gmres, tcqmr, bcgs, cgs, tfqmr, cr, lsqr, bicg, or preonly. There are method-specific options for the Richardson, Chebychev, and GMRES methods:

```
KSPRichardsonSetScale (KSP ksp,double damping_factor);
KSPChebychevSetEigenvalues (KSP ksp,double emax,double emin);
KSPGMRESSetRestart (KSP ksp,int max_steps);
```

The default parameter values are damping_factor=1.0, emax=0.01, emin=100.0, and max_steps=30. The GMRES restart and Richardson damping factor can also be set with the options -ksp_gmres_restart <n> and -ksp_richardson_scale <factor>. The default technique for orthogonalization of the Hessenberg matrix in GMRES is the iterative refinement Gram-Schmidt method. This can be set by using the command line option -ksp_gmres_irorthog. Or via

```
KSPGMRESSetOrthogonalization (KSP ksp,KSPGMRESModifiedGramSchmidtOrthogonalization);
```

A slightly faster approach is to use the unmodified (classical) Gram-Schmidt method, which can be set with

```
KSPGMRESSetOrthogonalization (KSP ksp,KSPGMRESUnmodifiedGramSchmidtOrthogonalization);
```

or the options database command -ksp_gmres_unmodifiedgramschmidt. Note that this algorithm is numerically unstable, but may deliver slightly better speed performance. One can also use modifed Gram-Schmidt, by setting the orthogonalization routine, KSPGMRESModifiedGramSchmidtOrthogonalization(), by using the command line option -ksp_gmres_modifiedgramschmidt. For the conjugate gradient method with complex numbers, there are two slightly different algorithms depending on whether the matrix is Hermitian symmetric or truly symmetric (the default is to assume that it is Hermitian symmetric). To indicate that it is symmetric, one uses the command

```
KSPCGSetType (KSP ksp,KSPCGType KSP_CG_SYMMETRIC);
```

Note that this option is not valid for all matrices. The LSQR algorithm does not involve a preconditioner, any preconditioner set to work with the KSP object is ignored if LSQR was selected. By default, KSP assumes an initial guess of zero by zeroing the initial value for the solution vector that is given; this zeroing is done at the call to SLESSolve () (or KSPSolve ()). To use a nonzero initial guess, the user *must* call

KSPSetInitialGuessNonzero (KSP ksp,PetscTruth flg);

4.3.1 Preconditioning within KSP

Since the rate of convergence of Krylov projection methods for a particular linear system is strongly dependent on its spectrum, preconditioning is typically used to alter the spectrum and hence accelerate the convergence rate of iterative techniques. Preconditioning can be applied to the system (4.1) by

$$(M_L^{-1} A M_R^{-1}) \, (M_R x) = M_L^{-1} b, \tag{4.2}$$

where M_L and M_R indicate preconditioning matrices (or, matrices from which the preconditioner is to be constructed). If $M_L = I$ in (4.2), right preconditioning results, and the residual of (4.1),

$$r \equiv b - Ax = b - A M_R^{-1} \, M_R x,$$

is preserved. In contrast, the residual is altered for left ($M_R = I$) and symmetric preconditioning, as given by

$$r_L \equiv M_L^{-1} b - M_L^{-1} A x = M_L^{-1} r.$$

By default, all KSP implementations use left preconditioning. Right preconditioning can be activated for some methods by using the options database command `-ksp_right_pc` or calling the routine

KSPSetPreconditionerSide (KSP ksp,PCSide PC_RIGHT);

Attempting to use right preconditioning for a method that does not currently support it results in an error message of the form

KSPSetUp_Richardson:No right preconditioning for KSPRICHARDSON

We summarize the defaults for the residuals used in KSP convergence monitoring within Table 3. Details regarding specific convergence tests and monitoring routines are presented in the following sections. The preconditioned residual is used by default for convergence testing of all left-preconditioned KSP methods. For the conjugate gradient, Richardson, and Chebyshev methods the true residual can be used by the options database command `ksp_norm_type unpreconditioned` or by calling the routine

KSPSetNormType (KSP ksp,KSP_UNPRECONDITIONED_NORM);

Note: the bi-conjugate gradient method requires application of both the matrix and its transpose plus the preconditioner and its transpose. Currently not all matrices and preconditioners provide this support and thus the `KSPBICG` cannot always be used.

4.3.2 Convergence Tests

The default convergence test, KSPDefaultConverged (), is based on the l_2-norm of the residual. Convergence (or divergence) is decided by three quantities: the relative decrease of the residual norm, `rtol`, the absolute size of the residual norm, `atol`, and the relative increase in the residual, `dtol`. Convergence is detected at iteration k if

$$\|r_k\|_2 < \max(rtol * \|r_0\|_2, atol),$$

Method	KSPType	Options Database Name	Default Convergence Monitor†
Richardson	KSPRICHARDSON	richardson	true
Chebychev	KSPCHEBYCHEV	chebychev	true
Conjugate Gradient [12]	KSPCG	cg	true
BiConjugate Gradient	KSPBICG	bicg	true
Generalized Minimal Residual [17]	KSPGMRES	gmres	precond
BiCGSTAB [20]	KSPBCGS	bcgs	precond
Conjugate Gradient Squared [19]	KSPCGS	cgs	precond
Transpose-Free Quasi-Minimal Residual (1) [8]	KSPTFQMR	tfqmr	precond
Transpose-Free Quasi-Minimal Residual (2)	KSPTCQMR	tcqmr	precond
Conjugate Residual	KSPCR	cr	precond
Least Squares Method	KSPLSQR	lsqr	precond
Shell for no KSP method	KSPPREONLY	preonly	precond

†true - denotes true residual norm, precond - denotes preconditioned residual norm

Table 3: KSP Defaults. All methods use left preconditioning by default.

where $r_k = b - Ax_k$. Divergence is detected if

$$\|r_k\|_2 > dtol * \|r_0\|_2.$$

These parameters, as well as the maximum number of allowable iterations, can be set with the routine

KSPSetTolerances(KSP ksp,double rtol,double atol,double dtol,int maxits);

The user can retain the default value of any of these parameters by specifying `PETSC_DEFAULT` as the corresponding tolerance; the defaults are `rtol`=10^{-5}, `atol`=10^{-50}, `dtol`=10^{5}, and `maxits`=10^{5}. These parameters can also be set from the options database with the commands `-ksp_rtol <rtol>`, `-ksp_atol <atol>`, `-ksp_divtol <dtol>`, and `-ksp_max_it <its>`. In addition to providing an interface to a simple convergence test, KSP allows the application programmer the flexibility to provide customized convergence-testing routines. The user can specify a customized routine with the command

KSPSetConvergenceTest(KSP ksp,int (*test)(KSP ksp,int it,double rnorm,
KSPConvergedReason *reason,void *ctx),void *ctx);

The final routine argument, `ctx`, is an optional context for private data for the user-defined convergence routine, `test`. Other `test` routine arguments are the iteration number, `it`, and the residual's l_2 norm, `rnorm`. The routine for detecting convergence, `test`, should set reason to positive for convergence, 0 for no convergence, and negative for failure to converge. A list of possible KSPConvergedReason is given in `include/petscksp.h`.

4.3.3 Convergence Monitoring

By default, the Krylov solvers run silently without displaying information about the iterations. The user can indicate that the norms of the residuals should be displayed by using `-ksp_monitor` within the options database. To display the residual norms in a graphical window (running under X Windows), one should use `-ksp_xmonitor [x,y,w,h]`, where either all or none of the options must be specified. Application programmers can also provide their own routines to perform the monitoring by using the command

KSPSetMonitor(KSP ksp,int (*mon)(KSP ksp,int it,double rnorm,void *ctx), void *ctx,int (*mondestroy)(void *));

The final routine argument, `ctx`, is an optional context for private data for the user-defined monitoring routine, `mon`. Other `mon` routine arguments are the iteration number (`it`) and the residual's l_2 norm (`rnorm`). A helpful routine within user-defined monitors is `PetscObjectGetComm((PetscObject)ksp,MPI_Comm *comm)`, which returns in `comm` the MPI communicator for the KSP context. See section 1.3 for more discussion of the use of MPI communicators within PETSc. Several monitoring routines are supplied with PETSc, including

KSPDefaultMonitor(KSP ,int,double, void *);
KSPSingularValueMonitor(KSP ,int,double, void *);
KSPTrueMonitor(KSP ,int,double, void *);

The default monitor simply prints an estimate of the l_2-norm of the residual at each iteration. The routine KSPSingularValueMonitor() is appropriate only for use with the conjugate gradient method or GMRES, since it prints estimates of the extreme singular values of the preconditioned operator at each iteration. Since KSPTrueMonitor() prints the true residual at each iteration by actually computing the residual using the formula $r = b - Ax$, the routine is slow and should be used only for testing or convergence studies, not for timing. These monitors may be accessed with the command line options `-ksp_monitor`, `-ksp_singmonitor`, and `-ksp_truemonitor`. To employ the default graphical monitor, one should use the commands

PetscDrawLG lg;
KSPLGMonitorCreate(char *display,char *title,int x,int y,int w,int h,PetscDrawLG *lg);
KSPSetMonitor(KSP ksp,KSPLGMonitor,lg,0);

When no longer needed, the line graph should be destroyed with the command

KSPLGMonitorDestroy(PetscDrawLG lg);

The user can change aspects of the graphs with the `PetscDrawLG*()` and `PetscDrawAxis*()` routines. One can also access this functionality from the options database with the command `-ksp_xmonitor [x,y,w,h]`. , where `x, y, w, h` are the optional location and size of the window. One can cancel hardwired monitoring routines for KSP at runtime with `-ksp_cancelmonitors`. Unless the Krylov method converges so that the residual norm is small, say 10^{-10}, many of the final digits printed with the `-ksp_monitor` option are meaningless. Worse, they are different on different machines; due to different round-off rules used by, say, the IBM RS6000 and the Sun Sparc. This makes testing between different machines difficult. The option `-ksp_smonitor` causes PETSc to print fewer of the digits of the residual norm as it gets smaller; thus on most of the machines it will always print the same numbers making cross system testing easier.

4.3.4 Understanding the Operator's Spectrum

Since the convergence of Krylov subspace methods depends strongly on the spectrum (eigenvalues) of the preconditioned operator, PETSc has specific routines for eigenvalue approximation via the Arnoldi or Lanczos iteration. First, before the linear solve one must call

KSPSetComputeEigenvalues(KSP ksp,PETSC_TRUE);

Then after the SLES solve one calls

KSPComputeEigenvalues(KSP ksp, int n,double *realpart,double *complexpart,int *neig);

Here, `n` is the `size` of the two arrays and the eigenvalues are inserted into those two arrays. `Neig` is the number of eigenvalues computed; this number depends on the `size` of the Krylov space generated during the linear system solution, for GMRES it is never larger than the restart parameter. There is an additional routine

`KSPComputeEigenvaluesExplicitly`(`KSP` ksp, int n,double *realpart,double *complexpart);

that is useful only for very small problems. It explicitly computes the full representation of the preconditioned operator and calls LAPACK to compute its eigenvalues. It should be only used for matrices of `size` up to a couple hundred. The `PetscDrawSP*()` routines are very useful for drawing scatter plots of the eigenvalues. The eigenvalues may also be computed and displayed graphically with the options data base commands `-ksp_plot_eigenvalues` and `-ksp_plot_eigenvalues_explicitly`. Or they can be dumped to the screen in ASCII text via `-ksp_compute_eigenvalues` and `-ksp_compute_eigenvalues_explicitly`.

4.3.5 Other KSP Options

To obtain the solution vector and right hand side from a `KSP` context, one uses

`KSPGetSolution`(`KSP` ksp,`Vec` *x);
`KSPGetRhs`(`KSP` ksp,`Vec` *rhs);

During the iterative process the solution may not yet have been calculated or it may be stored in a different location. To access the approximate solution during the iterative process, one uses the command

`KSPBuildSolution`(`KSP` ksp,`Vec` w,`Vec` *v);

where the solution is returned in `v`. The user can optionally provide a vector in `w` as the location to store the vector; however, if `w` is `PETSC_NULL`, space allocated by PETSc in the `KSP` context is used. One should not destroy this vector. For certain `KSP` methods, (e.g., GMRES), the construction of the solution is expensive, while for many others it requires not even a vector copy.

Access to the residual is done in a similar way with the command

`KSPBuildResidual`(`KSP` ksp,`Vec` t,`Vec` w,`Vec` *v);

Again, for GMRES and certain other methods this is an expensive operation.

4.4 Preconditioners

As discussed in Section 4.3.1, the Krylov space methods are typically used in conjunction with a preconditioner. To employ a particular preconditioning method, the user can either select it from the options database using input of the form `-pc_type <methodname>` or set the method with the command

`PCSetType`(`PC` pc,`PCType` method);

In Table 4 we summarize the basic preconditioning methods supported in PETSc. The `PCSHELL` preconditioner uses a specific, application-provided preconditioner. The direct preconditioner, `PCLU`, is, in fact, a direct solver for the linear system that uses LU factorization. `PCLU` is included as a preconditioner so that PETSc has a consistent interface among direct and iterative linear solvers. Each preconditioner may have associated with it a set of options, which can be set with routines and options database commands provided for this purpose. Such routine names and commands are all of the form `PC<TYPE>Option` and `-pc_<type>_option [value]`. A complete list can be found by consulting the manual pages; we discuss just a few in the sections below.

Method	PCType	Options Database Name
Jacobi	PCJACOBI	jacobi
Block Jacobi	PCBJACOBI	bjacobi
SOR (and SSOR)	PCSOR	sor
SOR with Eisenstat trick	PCEISENSTAT	eisenstat
Incomplete Cholesky	PCICC	icc
Incomplete LU	PCILU	ilu
Additive Schwarz	PCASM	asm
Linear solver	PCSLES	sles
Combination of preconditioners	PCCOMPOSITE	composite
LU	PCLU	lu
Cholesky	PCCholesky	cholesky
No preconditioning	PCNONE	none
Shell for user-defined PC	PCSHELL	shell

Table 4: PETSc Preconditioners

4.4.1 ILU and ICC Preconditioners

Some of the options for ILU preconditioner are

PCILUSetLevels(PC pc,int levels);
PCILCCSetLevels(PC pc,int levels);
PCILUSetReuseOrdering(PC pc,PetscTruth flag);
PCILUSetUseDropTolerance(PC pc,double dt,double dtcol,int dtcount);
PCILUDTSetReuseFill(PC pc,PetscTruth flag);
PCILUSetUseInPlace(PC pc);
PCILUSetAllowDiagonalFill(PC pc);

When repeatedly solving linear systems with the same SLES context, one can reuse some information computed during the first linear solve. In particular, PCILUSetReuseOrdering() causes the ordering (for example, set with `-pc_ilu_ordering_type order`) computed in the first factorization to be reused for later factorizations. The `PCILUDTSetReuseFill()` causes the fill computed during the first drop tolerance factorization to be reused in later factorizations. PCILUSetUseInPlace() is often used with PCASM or PCBJACOBI when zero fill is used, since it reuses the matrix space to store the incomplete factorization it saves memory and copying time. Note that in-place factorization is not appropriate with any ordering besides natural and cannot be used with the drop tolerance factorization. These options may be set in the database with

```
-pc_ilu_levels <levels>
-pc_ilu_reuse_ordering
-pc_ilu_use_drop_tolerance <dt>,<dtcol>,<dtcount>
-pc_ilu_reuse_fill
-pc_ilu_in_place
-pc_ilu_nonzeros_along_diagonal
-pc_ilu_diagonal_fill
```

See Section 13.4.2 for information on preallocation of memory for anticipated fill during factorization. By alleviating the considerable overhead for dynamic memory allocation, such tuning can significantly enhance performance. PETSc supports incomplete factorization preconditioners for several matrix types for the uniprocessor case. In addition, for the parallel case we provide an interface to the ILU and ICC preconditioners of BlockSolve95 [13]. PETSc enables users to employ the preconditioners within BlockSolve95 by using the BlockSolve95 matrix format `MATMPIROWBS` and invoking either the `PCILU` or `PCICC` method within the linear solvers. Since PETSc automatically handles matrix assembly, preconditioner setup, profiling, etc., users who employ BlockSolve95 through the PETSc interface need not concern themselves with many details provided within the BlockSolve95 users manual. Consult the file `docs/installation/index.htm` for details on installing PETSc to allow the use of BlockSolve95. One can create a matrix that is compatible with BlockSolve95 by using `MatCreate` () with the option `-mat_mpirowbs`, or by directly calling

`MatCreateMPIRowbs` (MPI_Comm `comm` ,int m,int M,int nz,int *nnz,`Mat` *A)

`A` is the newly created matrix, while the arguments `m` and `M` indicate the number of local and global rows, respectively. Either the local or global parameter can be replaced with PETSC_DECIDE, so that PETSc will determine it. The matrix is stored with a fixed number of rows on each processor, given by `m`, or determined by PETSc if `m` is PETSC_DECIDE. The arguments `nz` and `nnz` can be used to preallocate storage space, as discussed in Section 3.1 for increasing the efficiency of matrix assembly; one sets `nz=0` and `nnz=PETSC_NULL` for PETSc to control all matrix memory allocation. If the matrix is symmetric, one *may* call

`MatSetOption` (`Mat` mat,MAT_SYMMETRIC);

to improve efficiency, but in this case one cannot use the ILU preconditioner, only ICC. Internally, PETSc inserts zero elements into matrices of the `MATMPIROWBS` format if necessary, so that nonsymmetric matrices are considered to be symmetric in terms of their sparsity structure; this format is required for use of the parallel communication routines within BlockSolve95. In particular, if the matrix element $A[i, j]$ exists, then PETSc will internally allocate a 0 value for the element $A[j, i]$ during `MatAssemblyEnd` () if the user has not already set a value for the matrix element $A[j, i]$. When manipulating a preconditioning matrix, A, BlockSolve95 internally works with a scaled and permuted matrix, $\hat{A} = PD^{-1/2}AD^{-1/2}$, where D is the diagonal of A, and P is a permutation matrix determined by a graph coloring for efficient parallel computation. Thus, when solving a linear system, $Ax = b$, using ILU/ICC preconditioning and the matrix format `MATMPIROWBS` for *both* the linear system matrix and the preconditioning matrix, one actually solves the scaled and permuted system $\hat{A}\hat{x} = \hat{b}$, where $\hat{x} = PD^{1/2}x$ and $\hat{b} = PD^{-1/2}b$. PETSc handles the internal scaling and permutation of x and b, so the user does *not* deal with these conversions, but instead always works with the original linear system. In this case, by default the scaled residual norm is monitored; one must use the option `-ksp_truemonitor` to print both the scaled and unscaled residual norms. *Note*: If one is using ILU/ICC via BlockSolve95 and the `MATMPIROWBS` matrix format for the preconditioner matrix, but using a different format for a different linear system matrix, then this scaling and permuting is done only internally during the application of the preconditioner.

4.4.2 SOR and SSOR Preconditioners

PETSc provides only a sequential SOR preconditioner that can only be used on sequential matrices or as the subblock preconditioner when using block Jacobi or ASM preconditioning (see below). The options for SOR preconditioning are

`PCSORSetOmega` (`PC` pc,double omega);
`PCSORSetIterations` (`PC` pc,int its,int lits);
`PCSORSetSymmetric` (`PC` pc,`MatSORType` type);

The first of these commands sets the relaxation factor for successive over (under) relaxation. The second command sets the number of inner iterations `its` and local iterations `lits` (the number of smoothing sweeps on a processor before doing a ghost point update from the other processors) to use between steps of the Krylov space method. The total number of SOR sweeps is given by `its*lits`. The third command sets the kind of SOR sweep, where the argument `type` can be one of `SOR_FORWARD_SWEEP`, `SOR_BACKWARD_SWEEP` or `SOR_SYMMETRIC_SWEEP`, the default being `SOR_FORWARD_SWEEP`. Setting the type to be `SOR_SYMMETRIC_SWEEP` produces the SSOR method. In addition, each processor can locally and independently perform the specified variant of SOR with the types `SOR_LOCAL_FORWARD_SWEEP`, `SOR_LOCAL_BACKWARD_SWEEP`, and `SOR_LOCAL_SYMMETRIC_SWEEP`. These variants can also be set with the options `-pc_sor_omega <omega>`, `-pc_sor_its <its>`, `-pc_sor_lits <lits>`, `-pc_sor_backward`, `-pc_sor_symmetric`, `-pc_sor_local_forward`, `-pc_sor_local_backward`, and `-pc_sor_local_symmetric`.

The Eisenstat trick [5] for SSOR preconditioning can be employed with the method `PCEISENSTAT` (`-pc_type eisenstat`). By using both left and right preconditioning of the linear system, this variant of SSOR requires about half of the floating-point operations for conventional SSOR. The option `-pc_eisenstat_no_diagonal_scaling`) (or the routine `PCEisenstatNoDiagonalScaling`()) turns off diagonal scaling in conjunction with Eisenstat SSOR method, while the option `-pc_eisenstat_omega <omega>` (or the routine `PCEisenstatSetOmega(PC pc,double omega)`) sets the SSOR relaxation coefficient, `omega`, as discussed above.

4.4.3 LU Factorization

The LU preconditioner provides several options. The first, given by the command

`PCLUSetUseInPlace`(`PC` pc);

causes the factorization to be performed in-place and hence destroys the original matrix. The options database variant of this command is `-pc_lu_in_place`. Another direct preconditioner option is selecting the ordering of equations with the command

`-pc_lu_ordering_type <ordering>`

The possible orderings are

- `MATORDERING_NATURAL` - Natural
- `MATORDERING_ND` - Nested Dissection
- `MATORDERING_1WD` - One-way Dissection
- `MATORDERING_RCM` - Reverse Cuthill-McKee
- `MATORDERING_QMD` - Quotient Minimum Degree

These orderings can also be set through the options database by specifying one of the following: `-pc_lu_ordering_type natural`, `-pc_lu_ordering_type nd`, `-pc_lu_ordering_type 1wd`, `-pc_lu_ordering_type rcm`, `-pc_lu_ordering_type qmd`. In addition, see `MatGetOrdering` (), discussed in Section 16.2. The sparse LU factorization provided in PETSc does not perform pivoting for numerical stability (since they are designed to preserve nonzero structure), thus occasionally a LU factorization will fail with a zero pivot when, in fact, the matrix is non-singular. The option `-pc_lu_nonzeros_along_diagonal <tol>` will often help eliminate the zero pivot, by preprocessing the the column ordering to remove small values from the diagonal. Here, `tol` is an optional tolerance to decide if a value is nonzero; by default it is $1.e-10$.

In addition, Section 13.4.2 provides information on preallocation of memory for anticipated fill during factorization. Such tuning can significantly enhance performance, since it eliminates the considerable overhead for dynamic memory allocation.

4.4.4 Block Jacobi and Overlapping Additive Schwarz Preconditioners

The block Jacobi and overlapping additive Schwarz methods in PETSc are supported in parallel; however, only the uniprocessor version of the block Gauss-Seidel method is currently in place. By default, the PETSc implentations of these methods employ ILU(0) factorization on each individual block (that is, the default solver on each subblock is `PCType=PCILU, KSPType=KSPPREONLY`); the user can set alternative linear solvers via the options `-sub_ksp_type` and `-sub_pc_type`. In fact, all of the `KSP` and `PC` options can be applied to the subproblems by inserting the prefix `-sub_` at the beginning of the option name. These options database commands set the particular options for *all* of the blocks within the global problem. In addition, the routines

`PCBJacobiGetSubSLES` (`PC` pc,int *n_local,int *first_local,`SLES` **subsles);
`PCASMGetSubSLES` (`PC` pc,int *n_local,int *first_local,`SLES` **subsles);

extract the `SLES` context for each local block. The argument `n_local` is the number of blocks on the calling processor, and `first_local` indicates the global number of the first block on the processor. The blocks are numbered successively by processors from zero through $gb - 1$, where gb is the number of global blocks. The array of `SLES` contexts for the local blocks is given by `subsles`. This mechanism enables the user to set different solvers for the various blocks. To set the appropriate data structures, the user *must* explicitly call `SLESSetUp` () before calling `PCBJacobiGetSubSLES` () or `PCASMGetSubSLES` (). For further details, see the example `${PETSC_DIR}/src/sles/examples/tutorials/ex7.c`. The block Jacobi, block Gauss-Seidel, and additive Schwarz preconditioners allow the user to set the number of blocks into which the problem is divided. The options database commands to set this value are `-pc_bjacobi_blocks n` and `-pc_bgs_blocks n`, and, within a program, the corresponding routines are

`PCBJacobiSetTotalBlocks` (`PC` pc,int blocks,int *`size`);
`PCASMSetTotalSubdomains` (`PC` pc,int n,`IS` *is);
`PCASMSetType` (`PC` pc,`PCASMType` type);

The optional argument `size`, is an array indicating the `size` of each block. Currently, for certain parallel matrix formats, only a single block per processor is supported. However, the `MATMPIAIJ` and `MATMPIBAIJ` formats support the use of general blocks as long as no blocks are shared among processors. The `is` argument contains the index sets that define the subdomains.

The object `PCASMType` is one of `PC_ASM_BASIC`, `PC_ASM_INTERPOLATE`, `PC_ASM_RESTRICT`, `PC_ASM_NONE` and may also be set with the options database `-pc_asm_type [basic,interpolate,restrict,none]`. The type `PC_ASM_BASIC` (or `-pc_asm_type basic`) corresponds to the standard additive Schwarz method that uses the full restriction and interpolation operators. The type `PC_ASM_RESTRICT` (or `-pc_asm_type restrict`) uses a full restriction operator, but during the interpolation process ignores the off-processor values. Similarly, `PC_ASM_INTERPOLATE` (or `-pc_asm_type interpolate`) uses a limited restriction process in conjunction with a full interpolation, while `PC_ASM_NONE` (or `-pc_asm_type none`) ignores off-processor valies for both restriction and interpolation. The ASM types with limited restriction or interpolation were suggested by Xiao-Chuan Cai and Marcus Sarkis [3]. `PC_ASM_RESTRICT` is the PETSc default, as it saves substantial communication and for many problems has the added benefit of requiring fewer iterations for convergence than the standard additive Schwarz method. The user can also set the number of blocks and sizes on a per-processor basis with the commands

`PCBJacobiSetLocalBlocks` (`PC` pc,int blocks,int *`size`);
`PCASMSetLocalSubdomains` (`PC` pc,int N,`IS` *is);

For the ASM preconditioner one can use the following command to set the overlap to compute in constructing the subdomains.

PCASMSetOverlap (PC pc,int overlap);

The overlap defaults to 1, so if one desires that no additional overlap be computed beyond what may have been set with a call to PCASMSetTotalSubdomains() or PCASMSetLocalSubdomains (), then overlap must be set to be 0. In particular, if one does *not* explicitly set the subdomains in an application code, then all overlap would be computed internally by PETSc, and using an overlap of 0 would result in an ASM variant that is equivalent to the block Jacobi preconditioner. Note that one can define initial index sets is with *any* overlap via PCASMSetTotalSubdomains () or PCASMSetLocalSubdomains (); the routine PCASMSetOverlap () merely allows PETSc to extend that overlap further if desired.

4.4.5 Shell Preconditioners

The shell preconditioner simply uses an application-provided routine to implement the preconditioner. To set this routine, one uses the command

PCShellSetApply (PC pc,int (*apply)(void *ctx,Vec ,Vec),void *ctx);

The final argument ctx is a pointer to the application-provided data structure needed by the preconditioner routine. The three routine arguments of apply() are this context, the input vector, and the output vector, respectively. For a preconditioner that requires some sort of "setup" before being used, that requires a new setup everytime the operator is changed, one can provide a "setup" routine that is called everytime the operator is changed (usually via SLESSetOperators ()).

PCShellSetSetUp (PC pc,int (*setup)(void *ctx));

The argument to the "setup" routine is the same application-provided data structure passed in with the PCShellSetApply () routine.

4.4.6 Combining Preconditioners

The PC type PCCOMPOSITE allows one to form new preconditioners by combining already defined preconditioners and solvers. Combining preconditioners usually requires some experimentation to find a combination of preconditioners that works better than any single method. It is a tricky business and is not recommended until your application code is complete and running and you are trying to improve performance. In many cases using a single preconditioner is better than a combination; an exception is the multigrid/multilevel preconditioners (solvers) that are always combinations of some sort, see Section 4.4.7. Let B_1 and B_2 represent the application of two preconditioners of type type1 and type2. The preconditioner $B = B_1 + B_2$ can be obtained with

PCSetType (pc,PCCOMPOSITE);
PCCompositeAddPC (pc,type1);
PCCompositeAddPC (pc,type2);

Any number of preconditioners may added in this way.

This way of combining preconditioners is called additive, since the actions of the preconditioners are added together. This is the default behavior. An alternative can be set with the option

PCCompositeSetType (PC pc,PCCompositeType PC_COMPOSITE_MULTIPLICATIVE);

In this form the new residual is updated after the application of each preconditioner and the next preconditioner applied to the next residual. For example, with two composed preconditioners: B_1 and B_2; $y = Bx$ is obtained from

$$y = B_1 x$$
$$w_1 = x - Ay$$
$$y = y + B_2 w_1$$

Loosely, this corresponds to a Gauss-Siedel iteration, while additive corresponds to a Jacobi iteration. Under most circumstances the multiplicative form requires one-half the number of iterations as the additive form; but the multiplicative form does require the application of A inside the preconditioner.

In the multiplicative version, the calculation of the residual inside the preconditioner can be done in two ways: using the original linear system matrix or using the matrix used to build the preconditioners `B1`, `B2`, etc. By default it uses the "preconditioner matrix", to use the true matrix use the option

```
PCCompositeSetUseTrue(PC pc);
```

The individual preconditioners can be accessed (in order to set options) via

```
PCCompositeGetPC(PC pc,int count,PC *subpc);
```

For example, to set the first sub preconditioners to use ILU(1)

```
PC subpc;
PCCompositeGetPC(pc,0,&subpc);
PCILUSetFill(subpc,1);
```

These various options can also be set via the options database. For example, `-pc_type composite -pc_composite_pcs jacobi,ilu` causes the composite preconditioner to be used with two preconditioners: Jacobi and ILU. The option `-pc_composite_type multiplicative` initiates the multiplicative version of the algorithm, while `-pc_composite_type additive` the additive version. Using the true preconditioner is obtained with the option `-pc_composite_true`. One sets options for the subpreconditioners with the extra prefix `-sub_N_` where `N` is the number of the subpreconditioner. For example, `-sub_0_pc_ilu_fill 0`. PETSc also allows a preconditioner to be a complete linear solver. This is achieved with the `PCSLES` type.

```
PCSetType(PC pc,PCSLES PCSLES);
PCSLESGetSLES(pc,&sles);
/* set any SLES/KSP/PC options */
```

From the command line one can use 5 iterations of bi-CG-stab with ILU(0) preconditioning as the preconditioner with `-pc_type sles -sles_pc_type ilu -sles_ksp_max_it 5 -sles_ksp_type bcgs`.

By default the inner `SLES` preconditioner uses the outer preconditioner matrix as the matrix to be solved in the linear system; to use the true matrix use the option

```
PCSLESSetUseTrue(PC pc);
```

or at the command line with `-pc_sles_true`. Naturally one can use a `SLES` preconditioner inside a composite preconditioner. For example, `-pc_type composite -pc_composite_pcs ilu,sles -sub_1_pc_type jacobi -sub_1_ksp_max_it 10` uses two preconditioners: ILU(0) and 10 iterations of GMRES with Jacobi preconditioning. Though it is not clear whether one would ever wish to do such a thing.

4.4.7 Multigrid Preconditioners

A large suite of routines is available for using multigrid as a preconditioner. In the PC framework the user is required to provide the coarse grid solver, smoothers, restriction, and interpolation, as well as the code to calculate residuals. The PC package allows all of that to be wrapped up into a PETSc compliant preconditioner. We fully support both matrix-free and matrix-based multigrid solvers. See also Chapter 7 for a higher level interface to the multigrid solvers for linear and nonlinear problems using the DMMG object. A multigrid preconditioner is created with the four commands

```
SLESCreate(MPI_Comm comm,SLES *sles);
SLESGetPC(SLES sles,PC *pc);
PCSetType(PC pc,PCMG);
MGSetLevels(pc,int levels,MPI_Comm *comms);
```

A large number of parameters affect the multigrid behavior. The command

```
MGSetType(PC pc,MGType mode);
```

indicates which form of multigrid to apply [18]. For standard V or W-cycle multigrids, one sets the `mode` to be `MGMULTIPLICATIVE`; for the additive form (which in certain cases reduces to the BPX method, or additive multilevel Schwarz, or multilevel diagonal scaling), one uses `MGADDITIVE` as the `mode`. For a variant of full multigrid, one can use `MGFULL`, and for the Kaskade algorithm `MGKASKADE`. For the multiplicative and full multigrid options, one can use a W-cycle by calling

```
MGSetCycles(PC pc,int cycles);
```

with a value of `MG_W_CYCLE` for `cycles`. The commands above can also be set from the options database. The option names are `-pc_mg_type [multiplicative, additive, full, kaskade]`, and `-pc_mg_cycles <cycles>`. The user can control the amount of pre- and postsmoothing by using either the options `-pc_mg_smoothup m` and `-pc_mg_smoothdown n` or the routines

```
MGSetNumberSmoothUp(PC pc,int m);
MGSetNumberSmoothDown(PC pc,int n);
```

Note that if the command MGSetSmoother() (discussed below) has been employed, the same amounts of pre- and postsmoothing will be used. The multigrid routines, which determine the solvers and interpolation/restriction operators that are used, are mandatory. To set the coarse grid solver, one must call

```
MGGetCoarseSolve(PC pc,SLES *sles);
```

and set the appropriate options in `sles`. Similarly, the smoothers are set by calling

```
MGGetSmoother(PC pc,int level,SLES *sles);
```

and setting the various options in `sles.` To use a different pre- and postsmoother, one should call the following routines instead.

```
MGGetSmootherUp(PC pc,int level,SLES *upsles);
```

and

```
MGGetSmootherDown(PC pc,int level,SLES *downsles);
```

Use

```
MGSetInterpolate(PC pc,int level,Mat P);
```

and

MGSetRestriction (PC pc,int level,Mat R);

to define the intergrid transfer operations. It is possible for these interpolation operations to be matrix free (see Section 3.3), he or she should make sure that these operations are defined for the (matrix-free) matrices passed in. Note that this system is arranged so that if the interpolation is the transpose of the restriction, the same `mat` argument can be passed to both MGSetRestriction () and MGSetInterpolation(). On each level except the coarsest, one must also set the routine to compute the residual. The following command suffices:

MGSetResidual (PC pc,int level,int (*residual)(Mat ,Vec ,Vec ,Vec),Mat mat);

The residual() function can be set to be MGDefaultResidual () if one's operator is stored in a `Mat` format. In certain circumstances, where it is much cheaper to calculate the residual directly, rather than through the usual formula $b - Ax$, the user may wish to provide an alternative.

Finally, the user must provide three work vectors for each level (except on the finest, where only the residual work vector is required). The work vectors are set with the commands

MGSetRhs (PC pc,int level,Vec b);
MGSetX (PC pc,int level,Vec x);
MGSetR (PC pc,int level,Vec r);

The user is responsible for freeing these vectors once the iteration is complete. One can control the KSP and PC options used on the various levels (as well as the coarse grid) using the prefix `mg_levels_` (`mg_coarse_` for the coarse grid). For example,

-mg_levels_ksp_type cg

will cause the CG method to be used as the Krylov method for each level. Or

-mg_levels_pc_type ilu -mg_levels_pc_ilu_levels 2

will cause the the ILU preconditioner to be used on each level with two levels of fill in the incomplete factorization.

4.5 Solving Singular Systems

Sometimes one is required to solver linear systems that are singular. That is systems with the matrix has a null space. For example, the discretization of the Laplacian operator with Neumann boundary conditions as a null space of the constant functions. PETSc has tools to help solve these systems. First, one must know what the null space is and store it using an orthonormal basis in an array of PETSc Vecs . (The constant functions can be handled seperately, since they are such a common case). Create a MatNullSpace object with the command

MatNullSpaceCreate (MPI_Comm,PetscTruth hasconstants,int dim,Vec *basis,
MatNullSpace *nsp);

Here `dim` is the number of vectors in `basis` and `hasconstants` indicates if the null space contains the constant functions. (If the null space contains the constant functions you do not need to include it in the `basis` vectors you provide). One then tells the PC object you are using what the null space is with the call

PCNullSpaceAttach (PC pc,MatNullSpace nsp);

The PETSc solvers will now handle the null space during the solution processor. *But* if one chooses a direct solver (or an incomplete factorization) it may still detect a zero pivot. You can run with the additional options `-pc_lu_damping <dampingfactor>` or `-pc_ilu_damping <dampingfactor>` to prevent the zero pivot. A good choice for the `damping` factor is 1.e-10.

4.6 Using PETSc interface to external linear solvers

PETSc interfaces to several external linear solvers (see Acknowledgments). To use these solvers, one needs to do the followings.

1. Install the external software.
2. Enable using the external software from PETSc by editing `${PETSC_DIR}/bmake/${PETSC_ARCH}/packages`. For example, to use SuperLU, one would specify the following variables with the appropriate paths:

```
SUPERLU_LIB        = /home/petsc/software/SuperLU/superlu_linux_gcc_pgf90.a
SUPERLU_INCLUDE    = -I/home/petsc/software/SuperLU/SRC
PETSC_HAVE_SUPERLU = -DPETSC_HAVE_SUPERLU
```

3. Build the PETSc libraries.
4. Use the runtime option: -mat_<mattype>_<package> -pc_type <pctype>.

Package	**MatType**	**PCType**	**Runtime Options**
SUPERLU	aij	lu	`-mat_aij_superlu`
SUPERLU_DIST	aij	lu	`-mat_aij_superlu_dist`
SPOOLES	aij	lu	`-mat_aij_spooles`
	sbaij	cholesky	`-mat_sbaij_spooles`
DSCPACK	baij	cholesky	`-mat_baij_dscpack`

Table 5: Options for External Solvers

The default and available input options for each external software can be found by specifying -help (or -h) at runtime.

Chapter 5

SNES: Nonlinear Solvers

The solution of large-scale nonlinear problems pervades many facets of computational science and demands robust and flexible solution strategies. The `SNES` library of PETSc provides a powerful suite of data-structure-neutral numerical routines for such problems. Built on top of the linear solvers and data structures discussed in preceding chapters, `SNES` enables the user to easily customize the nonlinear solvers according to the application at hand. Also, the `SNES` interface is *identical* for the uniprocessor and parallel cases; the only difference in the parallel version is that each processor typically forms only its local contribution to various matrices and vectors. The `SNES` class includes methods for solving systems of nonlinear equations of the form

$$\boldsymbol{F}(\boldsymbol{x}) = 0, \tag{5.1}$$

where $\boldsymbol{F} : \Re^n \to \Re^n$. Newton-like methods provide the core of the package, including both line search and trust region techniques, which are discussed further in Section 5.2. Following the PETSc design philosophy, the interfaces to the various solvers are all virtually identical. In addition, the `SNES` software is completely flexible, so that the user can at runtime change any facet of the solution process. The general form of the n-dimensional Newton's method for solving (5.1) is

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - [\boldsymbol{F}'(\boldsymbol{x}_k)]^{-1}\boldsymbol{F}(\boldsymbol{x}_k), \quad k = 0, 1, \ldots, \tag{5.2}$$

where $\boldsymbol{x}_0$ is an initial approximation to the solution and $\boldsymbol{F}'(\boldsymbol{x}_k)$ is nonsingular. In practice, the Newton iteration (5.2) is implemented by the following two steps:

$$1. \quad \text{(Approximately) solve} \quad \boldsymbol{F}'(\boldsymbol{x}_k)\Delta\boldsymbol{x}_k = -\boldsymbol{F}(\boldsymbol{x}_k). \tag{5.3}$$

$$2. \quad \text{Update} \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}_k. \tag{5.4}$$

5.1 Basic Usage

In the simplest usage of the nonlinear solvers, the user must merely provide a C, C++, or Fortran routine to evaluate the nonlinear function of Equation (5.1). The corresponding Jacobian matrix can be approximated with finite differences. For codes that are typically more efficient and accurate, the user can provide a routine to compute the Jacobian; details regarding these application-provided routines are discussed below. To provide an overview of the use of the nonlinear solvers, we first introduce a complete and simple example in Figure 12, corresponding to `${PETSC_DIR}/src/snes/examples/tutorials/ex1.c`.

```
/*$Id: ex1.c,v 1.26 2001/08/07 03:04:16 balay Exp $*/

static char help[] = "Newton's method to solve a two-variable system, se-
quentially.\n\n";
```

```
/*T
   Concepts: SNES^basic uniprocessor example
   Processors: 1
T*/

/*
   Include "petscsnes.h" so that we can use SNES solvers.  Note that this
   file automatically includes:
     petsc.h       - base PETSc routines   petscvec.h - vectors
     petscsys.h    - system routines       petscmat.h - matrices
     petscis.h     - index sets            petscksp.h - Krylov subspace meth-
ods
     petscviewer.h - viewers               petscpc.h  - preconditioners
     petscsles.h   - linear solvers
*/
#include "petscsnes.h"

/*
   User-defined routines
*/
extern int FormJacobian1(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
extern int FormFunction1(SNES,Vec,Vec,void*);
extern int FormJacobian2(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
extern int FormFunction2(SNES,Vec,Vec,void*);

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **argv)
{
  SNES         snes;         /* nonlinear solver context */
  SLES         sles;         /* linear solver context */
  PC           pc;           /* preconditioner context */
  KSP          ksp;          /* Krylov subspace method context */
  Vec          x,r;         /* solution, residual vectors */
  Mat          J;            /* Jacobian matrix */
  int          ierr,its,size;
  PetscScalar  pfive = .5,*xx;
  PetscTruth   flg;

  PetscInitialize(&argc,&argv,(char *)0,help);
  ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
  if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create nonlinear solver context
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = SNESCreate(PETSC_COMM_WORLD,&snes);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

     Create matrix and vector data structures; set corresponding routines
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*/
```

```
/*
   Create vectors for solution and nonlinear function
*/
ierr = VecCreateSeq(PETSC_COMM_SELF,2,&x);CHKERRQ(ierr);
ierr = VecDuplicate(x,&r);CHKERRQ(ierr);

/*
   Create Jacobian matrix data structure
*/
ierr = MatCreate(PETSC_COMM_SELF,PETSC_DECIDE,PETSC_DECIDE,2,2,&J);CHKERRQ(ierr);
ierr = MatSetFromOptions(J);CHKERRQ(ierr);

ierr = PetscOptionsHasName(PETSC_NULL,"-hard",&flg);CHKERRQ(ierr);
if (!flg) {
  /*
   Set function evaluation routine and vector.
  */
  ierr = SNESSetFunction(snes,r,FormFunction1,PETSC_NULL);CHKERRQ(ierr);

  /*
   Set Jacobian matrix data structure and Jacobian evaluation routine
  */
  ierr = SNESSetJacobian(snes,J,J,FormJacobian1,PETSC_NULL);CHKERRQ(ierr);
} else {
  ierr = SNESSetFunction(snes,r,FormFunction2,PETSC_NULL);CHKERRQ(ierr);
  ierr = SNESSetJacobian(snes,J,J,FormJacobian2,PETSC_NULL);CHKERRQ(ierr);
}

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Customize nonlinear solver; set runtime options
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

/*
   Set linear solver defaults for this problem. By extracting the
   SLES, KSP, and PC contexts from the SNES context, we can then
   directly call any SLES, KSP, and PC routines to set various options.
*/
ierr = SNESGetSLES(snes,&sles);CHKERRQ(ierr);
ierr = SLESGetKSP(sles,&ksp);CHKERRQ(ierr);
ierr = SLESGetPC(sles,&pc);CHKERRQ(ierr);
ierr = PCSetType(pc,PCNONE);CHKERRQ(ierr);
ierr = KSPSetTolerances(ksp,1.e-4,PETSC_DEFAULT,PETSC_DEFAULT,20);CHKERRQ(ierr);

/*
   Set SNES/SLES/KSP/PC runtime options, e.g.,
       -snes_view -snes_monitor -ksp_type <ksp> -pc_type <pc>
   These options will override those specified above as long as
   SNESSetFromOptions() is called _after_ any other customization
   routines.
*/
ierr = SNESSetFromOptions(snes);CHKERRQ(ierr);

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   Evaluate initial guess; then solve nonlinear system
```

```
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
  if (!flg) {
    ierr = VecSet(&pfive,x);CHKERRQ(ierr);
  } else {
    ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
    xx[0] = 2.0; xx[1] = 3.0;
    ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
  }
  /*
     Note: The user should initialize the vector, x, with the initial guess
     for the nonlinear solver prior to calling SNESSolve().  In particular,
     to employ an initial guess of zero, the user should explicitly set
     this vector to zero by calling VecSet().
  */

  ierr = SNESSolve(snes,x,&its);CHKERRQ(ierr);
  if (flg) {
    Vec f;
    ierr = VecView(x,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
    ierr = SNESGetFunction(snes,&f,0,0);CHKERRQ(ierr);
    ierr = VecView(r,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
  }

  ierr = PetscPrintf(PETSC_COMM_SELF,"number of Newton iterations = %d\n\n",its);CHKERRQ(ie

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = VecDestroy(x);CHKERRQ(ierr); ierr = VecDestroy(r);CHKERRQ(ierr);
  ierr = MatDestroy(J);CHKERRQ(ierr); ierr = SNESDestroy(snes);CHKERRQ(ierr);

  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
/* ------------------------------------------------------------------- */
#undef __FUNCT__
#define __FUNCT__ "FormFunction1"
/*
   FormFunction1 - Evaluates nonlinear function, F(x).

   Input Parameters:
.  snes - the SNES context
.  x - input vector
.  dummy - optional user-defined context (not used here)

   Output Parameter:
.  f - function vector
 */
int FormFunction1(SNES snes,Vec x,Vec f,void *dummy)
{
  int    ierr;
  PetscScalar *xx,*ff;
```

```
  /*
     Get pointers to vector data.
       - For default PETSc vectors, VecGetArray() returns a pointer to
         the data array.  Otherwise, the routine is implementation depen-
dent.
       - You MUST call VecRestoreArray() when you no longer need access to
         the array.
  */
  ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
  ierr = VecGetArray(f,&ff);CHKERRQ(ierr);

  /*
     Compute function
  */
  ff[0] = xx[0]*xx[0] + xx[0]*xx[1] - 3.0;
  ff[1] = xx[0]*xx[1] + xx[1]*xx[1] - 6.0;

  /*
     Restore vectors
  */
  ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
  ierr = VecRestoreArray(f,&ff);CHKERRQ(ierr);

  return 0;
}
/* ------------------------------------------------------------------- */
#undef __FUNCT__
#define __FUNCT__ "FormJacobian1"
/*
   FormJacobian1 - Evaluates Jacobian matrix.

   Input Parameters:
.  snes - the SNES context
.  x - input vector
.  dummy - optional user-defined context (not used here)

   Output Parameters:
.  jac - Jacobian matrix
.  B - optionally different preconditioning matrix
.  flag - flag indicating matrix structure
*/
int FormJacobian1(SNES snes,Vec x,Mat *jac,Mat *B,MatStructure *flag,void
*dummy)
{
  PetscScalar *xx,A[4];
  int    ierr,idx[2] = {0,1};

  /*
     Get pointer to vector data
  */
  ierr = VecGetArray(x,&xx);CHKERRQ(ierr);

  /*
     Compute Jacobian entries and insert into matrix.
      - Since this is such a small problem, we set all entries for
```

```
        the matrix at once.
  */
  A[0] = 2.0*xx[0] + xx[1]; A[1] = xx[0];
  A[2] = xx[1]; A[3] = xx[0] + 2.0*xx[1];
  ierr = MatSetValues(*jac,2,idx,2,idx,A,INSERT_VALUES);CHKERRQ(ierr);
  *flag = SAME_NONZERO_PATTERN;

  /*
     Restore vector
  */
  ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);

  /*
     Assemble matrix
  */
  ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

  return 0;
}


/* ------------------------------------------------------------------- */
#undef __FUNCT__
#define __FUNCT__ "FormFunction2"
int FormFunction2(SNES snes,Vec x,Vec f,void *dummy)
{
  int    ierr;
  PetscScalar *xx,*ff;

  /*
     Get pointers to vector data.
       - For default PETSc vectors, VecGetArray() returns a pointer to
         the data array.  Otherwise, the routine is implementation depen-
dent.
       - You MUST call VecRestoreArray() when you no longer need access to
         the array.
  */
  ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
  ierr = VecGetArray(f,&ff);CHKERRQ(ierr);

  /*
     Compute function
  */
  ff[0] = PetscSinScalar(3.0*xx[0]) + xx[0];
  ff[1] = xx[1];

  /*
     Restore vectors
  */
  ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
  ierr = VecRestoreArray(f,&ff);CHKERRQ(ierr);

  return 0;
}
```

```
/* ------------------------------------------------------------------- */
#undef __FUNCT__
#define __FUNCT__ "FormJacobian2"
int FormJacobian2(SNES snes,Vec x,Mat *jac,Mat *B,MatStructure *flag,void
*dummy)
{
  PetscScalar *xx,A[4];
  int    ierr,idx[2] = {0,1};

  /*
     Get pointer to vector data
  */
  ierr = VecGetArray(x,&xx);CHKERRQ(ierr);

  /*
     Compute Jacobian entries and insert into matrix.
      - Since this is such a small problem, we set all entries for
        the matrix at once.
  */
  A[0] = 3.0*PetscCosScalar(3.0*xx[0]) + 1.0; A[1] = 0.0;
  A[2] = 0.0;                     A[3] = 1.0;
  ierr = MatSetValues(*jac,2,idx,2,idx,A,INSERT_VALUES);CHKERRQ(ierr);
  *flag = SAME_NONZERO_PATTERN;

  /*
     Restore vector
  */
  ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);

  /*
     Assemble matrix
  */
  ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

  return 0;
}
```

Figure 12: Example of Uniprocessor SNES Code

To create a `SNES` solver, one must first call `SNESCreate`() as follows:

`SNESCreate`(MPI_Comm `comm`,`SNES` *snes);

The user must then set routines for evaluating the function of equation (5.1) and its associated Jacobian matrix, as discussed in the following sections. To choose a nonlinear solution method, the user can either call

`SNESSetType`(`SNES` snes,`SNESType` method);

or use the the option `-snes_type <method>`, where details regarding the available methods are presented in Section 5.2. The application code can take complete control of the linear and nonlinear techniques used in the Newton-like method by calling

SNESSetFromOptions(snes);

This routine provides an interface to the PETSc options database, so that at runtime the user can select a particular nonlinear solver, set various parameters and customized routines (e.g., specialized line search variants), prescribe the convergence tolerance, and set monitoring routines. With this routine the user can also control all linear solver options in the SLES, KSP, and PC modules, as discussed in Chapter 4. After having set these routines and options, the user solves the problem by calling

SNESSolve(SNES snes,Vec x,int *iters);

where iters is the number of nonlinear iterations required for convergence and x indicates the solution vector. The user should initialize this vector to the initial guess for the nonlinear solver prior to calling SNESSolve(). In particular, to employ an initial guess of zero, the user should explicitly set this vector to zero by calling VecSet(). Finally, after solving the nonlinear system (or several systems), the user should destroy the SNES context with

SNESDestroy(SNES snes);

5.1.1 Nonlinear Function Evaluation

When solving a system of nonlinear equations, the user must provide a vector, f, for storing the function of Equation (5.1), as well as a routine that evaluates this function at the vector x. This information should be set with the command

SNESSetFunction(SNES snes,Vec f,
int (*FormFunction)(SNES snes,Vec x,Vec f,void *ctx),void *ctx);

The argument ctx is an optional user-defined context, which can store any private, application-specific data required by the function evaluation routine; PETSC_NULL should be used if such information is not needed. In C and C++, a user-defined context is merely a structure in which various objects can be stashed; in Fortran a user context can be an integer array that contains both parameters and pointers to PETSc objects. ${PETSC_DIR}/src/snes/examples/tutorials/ex5.c and ${PETSC_DIR}/src/snes/examples/tutorials/ex5f.F give examples of user-defined application contexts in C and Fortran, respectively.

5.1.2 Jacobian Evaluation

The user must also specify a routine to form some approximation of the Jacobian matrix, A, at the current iterate, x, as is typically done with

SNESSetJacobian(SNES snes,Mat A,Mat B,int (*FormJacobian)(SNES snes,
Vec x,Mat *A,Mat *B,MatStructure *flag,void *ctx),void *ctx);

The arguments of the routine FormJacobian() are the current iterate, x; the Jacobian matrix, A; the preconditioner matrix, B (which is usually the same as A); a flag indicating information about the preconditioner matrix structure; and an optional user-defined Jacobian context, ctx, for application-specific data. The options for flag are identical to those for the flag of SLESSetOperators(), discussed in Section 4.1. Note that the SNES solvers are all data-structure neutral, so the full range of PETSc matrix formats (including "matrix-free" methods) can be used. Chapter 3 discusses information regarding available matrix formats and options, while Section 5.5 focuses on matrix-free methods in SNES. We briefly touch on a few details of matrix usage that are particularly important for efficient use of the nonlinear solvers. During successive calls to FormJacobian(), the user can either insert new matrix

Method	SNES Type	Options Name	Default Convergence Test
Line search	SNESLS	ls	SNESConverged_LS()
Trust region	SNESTR	tr	SNESConverged_TR()
Test Jacobian	SNESTEST	test	

Table 6: PETSc Nonlinear Solvers

contexts or reuse old ones, depending on the application requirements. For many sparse matrix formats, reusing the old space (and merely changing the matrix elements) is more efficient; however, if the matrix structure completely changes, creating an entirely new matrix context may be preferable. Upon subsequent calls to the `FormJacobian()` routine, the user may wish to reinitialize the matrix entries to zero by calling `MatZeroEntries` (). See Section 3.4 for details on the reuse of the matrix context. If the preconditioning matrix retains identical nonzero structure during successive nonlinear iterations, setting the parameter, `flag`, in the `FormJacobian()` routine to be `SAME_NONZERO_PATTERN` and reusing the matrix context can save considerable overhead. For example, when one is using a parallel preconditioner such as incomplete factorization in solving the linearized Newton systems for such problems, matrix colorings and communication patterns can be determined a single time and then reused repeatedly throughout the solution process. In addition, if using different matrices for the actual Jacobian and the preconditioner, the user can hold the preconditioner matrix fixed for multiple iterations by setting `flag` to `SAME_PRECONDITIONER`. See the discussion of `SLESSetOperators` () in Section 4.1 for details. The directory `${PETSC_DIR}/src/snes/examples/tutorials` provides a variety of examples.

5.2 The Nonlinear Solvers

As summarized in Table 5.2, `SNES` includes several Newton-like nonlinear solvers based on line search techniques and trust region methods. Each solver may have associated with it a set of options, which can be set with routines and options database commands provided for this purpose. A complete list can be found by consulting the manual pages or by running a program with the `-help` option; we discuss just a few in the sections below.

5.2.1 Line Search Techniques

The method `SNESLS` (`-snes_type ls`) provides a line search Newton method for solving systems of nonlinear equations. By default, this technique employs cubic backtracking [4]. An alternative line search routine can be set with the command

```
SNESSetLineSearch(SNES snes,
int (*ls)(SNES,Vec,Vec,Vec,Vec,double,double*,double*),void *lsctx);
```

Other line search methods provided by PETSc are `SNESQuadraticLineSearch` (), `SNESNoLineSearch` (), and `SNESNoLineSearchNoNorms` (), which can be set with the option `-snes_ls [cubic, quadratic`
The line search routines involve several parameters, which are set to defaults that are reasonable for many applications. The user can override the defaults by using the options `-snes_ls_alpha <alpha>`, `-snes_ls_maxstep <max>`, and `-snes_ls_steptol <tol>`.

5.2.2 Trust Region Methods

The trust region method in `SNES` for solving systems of nonlinear equations, `SNESTR` (`-snes_type tr`), is taken from the MINPACK project [14]. Several parameters can be set to control the variation of the trust region size during the solution process. In particular, the user can control the initial trust region radius, computed by

$$\Delta = \Delta_0 \|F_0\|_2,$$

by setting Δ_0 via the option `-snes_tr_delta0 <delta0>`.

5.3 General Options

This section discusses options and routines that apply to all SNES solvers and problem classes. In particular, we focus on convergence tests, monitoring routines, and tools for checking derivative computations.

5.3.1 Convergence Tests

Convergence of the nonlinear solvers can be detected in a variety of ways; the user can even specify a customized test, as discussed below. The default convergence routines for the various nonlinear solvers within SNES are listed in Table 5.2; see the corresponding manual pages for detailed descriptions. Each of these convergence tests involves several parameters, which are set by default to values that should be reasonable for a wide range of problems. The user can customize the parameters to the problem at hand by using some of the following routines and options. One method of convergence testing is to declare convergence when the norm of the change in the solution between successive iterations is less than some tolerance, `stol`. Convergence can also be determined based on the norm of the function (or gradient for a minimization problem). Such a test can use either the absolute size of the norm, `atol`, or its relative decrease, `rtol`, from an initial guess. The following routine sets these parameters, which are used in many of the default SNES convergence tests:

SNESSetTolerances(SNES snes,double atol,double rtol,double stol,
int its,int fcts);

This routine also sets the maximum numbers of allowable nonlinear iterations, `its`, and function evaluations, `fcts`. The corresponding options database commands for setting these parameters are `-snes_atol <atol>`, `-snes_rtol <rtol>`, `-snes_stol <stol>`, `-snes_max_it <its>`, and `-snes_max_funcs <fcts>`. A related routine is SNESGetTolerances(). Convergence tests for trust regions methods often use an additional parameter that indicates the minimum allowable trust region radius. The user can set this parameter with the option `-snes_trtol <trtol>` or with the routine

SNESSetTrustRegionTolerance(SNES snes,double trtol);

An additional parameter is sometimes used for unconstrained minimization problems, namely the minimum function tolerance, `ftol`, which can be set with the option `-snes_fmin <ftol>` or with the routine

SNESSetMinimizationFunctionTolerance(SNES snes,double ftol);

Users can set their own customized convergence tests in SNES by using the command

SNESSetConvergenceTest(SNES snes,int (*test)(SNES snes,double xnorm,
double gnorm,double f,SNESConvergedReason reason,
void *cctx),void *cctx);

The final argument of the convergence test routine, `cctx`, denotes an optional user-defined context for private data. When solving systems of nonlinear equations, the arguments `xnorm`, `gnorm`, and `f` are the current iterate norm, current step norm, and function norm, respectively. Likewise, when solving unconstrained minimization problems, the arguments `xnorm`, `gnorm`, and `f` are the current iterate norm, current gradient norm, and the function value. `SNESConvergedReason` should be set positive for convergence and negative for divergence. See `include/petscsnes.h` for a list of values for SNESConvergedReason.

5.3.2 Convergence Monitoring

By default the `SNES` solvers run silently without displaying information about the iterations. The user can initiate monitoring with the command

`SNESSetMonitor` (`SNES` snes,int (*mon)(`SNES` ,int its,double norm,void* mctx),
void *mctx,int (*monitordestroy)(void *));

The routine, `mon`, indicates a user-defined monitoring routine, where `its` and `mctx` respectively denote the iteration number and an optional user-defined context for private data for the monitor routine. The argument `norm` is the function norm. The routine set by `SNESSetMonitor` () is called once after every successful step computation within the nonlinear solver. Hence, the user can employ this routine for any application-specific computations that should be done after the solution update. The option `-snes_monitor` activates the default `SNES` monitor routine, `SNESDefaultMonitor` (), while `-snes_xmonitor` draws a simple line graph of the residual norm's convergence. Once can cancel hardwired monitoring routines for `SNES` at runtime with `-snes_cancelmonitors`. As the Newton method converges so that the residual norm is small, say 10^{-10}, many of the final digits printed with the `-snes_monitor` option are meaningless. Worse, they are different on different machines; due to different round-off rules used by, say, the IBM RS6000 and the Sun Sparc. This makes testing between different machines difficult. The option `-snes_smonitor` causes PETSc to print fewer of the digits of the residual norm as it gets smaller; thus on most of the machines it will always print the same numbers making cross processor testing easier. The routines

`SNESGetSolution` (`SNES` snes,`Vec` *x);
`SNESGetFunction` (`SNES` snes,`Vec` *r,void *ctx,
int(**func)(`SNES` ,`Vec` ,`Vec` ,void*));

return the solution vector and function vector from a `SNES` context. These routines are useful, for instance, if the convergence test requires some property of the solution or function other than those passed with routine arguments.

5.3.3 Checking Accuracy of Derivatives

Since hand-coding routines for Jacobian matrix evaluation can be error prone, `SNES` provides easy-to-use support for checking these matrices against finite difference versions. In the simplest form of comparison, users can employ the option `-snes_type test` to compare the matrices at several points. Although not exhaustive, this test will generally catch obvious problems. One can compare the elements of the two matrices by using the option `-snes_test_display` , which causes the two matrices to be printed to the screen. Another means for verifying the correctness of a code for Jacobian computation is running the problem with either the finite difference or matrix-free variant, `-snes_fd` or `-snes_mf`. see Section 5.6 or Section 5.5). If a problem converges well with these matrix approximations but not with a user-provided routine, the problem probably lies with the hand-coded matrix.

5.4 Inexact Newton-like Methods

Since exact solution of the linear Newton systems within (5.2) at each iteration can be costly, modifications are often introduced that significantly reduce these expenses and yet retain the rapid convergence of Newton's method. Inexact or truncated Newton techniques approximately solve the linear systems using an iterative scheme. In comparison with using direct methods for solving the Newton systems, iterative methods have the virtue of requiring little space for matrix storage and potentially saving significant computational work. Within the class of inexact Newton methods, of particular interest are Newton-Krylov methods, where the subsidiary iterative technique for solving the Newton system is chosen from the class of

Krylov subspace projection methods. Note that at runtime the user can set any of the linear solver options discussed in Chapter 4, such as `-ksp_type <ksp_method>` and `-pc_type <pc_method>`, to set the Krylov subspace and preconditioner methods. Two levels of iterations occur for the inexact techniques, where during each global or outer Newton iteration a sequence of subsidiary inner iterations of a linear solver is performed. Appropriate control of the accuracy to which the subsidiary iterative method solves the Newton system at each global iteration is critical, since these inner iterations determine the asymptotic convergence rate for inexact Newton techniques. While the Newton systems must be solved well enough to retain fast local convergence of the Newton's iterates, use of excessive inner iterations, particularly when $\|\boldsymbol{x}_k - \boldsymbol{x}_*\|$ is large, is neither necessary nor economical. Thus, the number of required inner iterations typically increases as the Newton process progresses, so that the truncated iterates approach the true Newton iterates. A sequence of nonnegative numbers $\{\eta_k\}$ can be used to indicate the variable convergence criterion. In this case, when solving a system of nonlinear equations, the update step of the Newton process remains unchanged, and direct solution of the linear system is replaced by iteration on the system until the residuals

$$\boldsymbol{r}_k^{(i)} = \boldsymbol{F}'(\boldsymbol{x}_k)\Delta\boldsymbol{x}_k + \boldsymbol{F}(\boldsymbol{x}_k)$$

satisfy

$$\frac{\|\boldsymbol{r}_k^{(i)}\|}{\|\boldsymbol{F}(\boldsymbol{x}_k)\|} \leq \eta_k \leq \eta < 1.$$

Here $\boldsymbol{x}_0$ is an initial approximation of the solution, and $\|\cdot\|$ denotes an arbitrary norm in $\Re^n$.

By default a constant relative convergence tolerance is used for solving the subsidiary linear systems within the Newton-like methods of `SNES` . When solving a system of nonlinear equations, one can instead employ the techniques of Eisenstat and Walker [6] to compute η_k at each step of the nonlinear solver by using the option `-snes_ksp_ew_conv` . In addition, by adding one's own `KSP` convergence test (see Section 4.3.2), one can easily create one's own, problem-dependent, inner convergence tests.

5.5 Matrix-Free Methods

The `SNES` class fully supports matrix-free methods. The matrices specified in the Jacobian evaluation routine need not be conventional matrices; instead, they can point to the data required to implement a particular matrix-free method. The matrix-free variant is allowed *only* when the linear systems are solved by an iterative method in combination with no preconditioning (`PCNONE` or `-pc_type none`), a user-provided preconditioner matrix, or a user-provided preconditioner shell (`PCSHELL`, discussed in Section 4.4); that is, obviously matrix-free methods cannot be used if a direct solver is to be employed. The user can create a matrix-free context for use within `SNES` with the routine

```
MatCreateSNESMF(SNES snes,Vec x, Mat *mat);
```

This routine creates the data structures needed for the matrix-vector products that arise within Krylov space iterative methods [2] by employing the matrix type `MATSHELL`, discussed in Section 3.3. The default `SNES` matrix-free approximations can also be invoked with the command `-snes_mf`. Or, one can retain the user-provided Jacobian preconditioner, but replace the user-provided Jacobian matrix with the default matrix free variant with the option `-snes_mf_operator`. See also

```
MatCreateMF(Vec x, Mat *mat);
```

for users who need a matrix-free matrix but are not using `SNES` . The user can set one parameter to control the Jacobian-vector product approximation with the command

```
MatSNESMFSetFunctionError(Mat mat,double rerror);
```

The parameter `rerror` should be set to the square root of the relative error in the function evaluations, e_{rel}; the default is 10^{-8}, which assumes that the functions are evaluated to full double precision accuracy. This parameter can also be set from the options database with

-snes_mf_err `<err>`

In addition, `SNES` provides a way to register new routines to compute the differencing parameter (h); see the manual page for `MatSNESMFSetType` () and `MatSNESMFRegisterDynamic`) . We currently provide two default routines accessible via

-snes_mf_type `<default or wp>`

For the default approach there is one "tuning" parameter, set with

`MatSNESMFDefaultSetUmin` (`Mat` mat,PetscReal umin);

This parameter, `umin` (or u_{min}), is a bit involved; its default is 10^{-6} . The Jacobian-vector product is approximated via the formula

$$F'(u)a \approx \frac{F(u + h * a) - F(u)}{h}$$

where h is computed via

$$\begin{aligned} h &= e_{rel} * u^T a/||a||_2^2 \quad \text{if} |u'a| > u_{min} * ||a||_1 \\ &= e_{rel} * u_{min} * sign(u^T a) * ||a||_1/||a||_2^2 \quad \text{otherwise.} \end{aligned}$$

This approach is taken from Brown and Saad [2]. The parameter can also be set from the options database with

-snes_mf_umin `<umin>`

The second approach, taken from Walker and Pernice, [16], computes h via

$$h = \frac{\sqrt{1 + ||u||}e_{rel}}{||a||}$$

This has no tunable parameters, but note that (a) for GMRES with left preconditioning $||a|| = 1$ and (b) for the entire **linear** iterative process u does not change hence $\sqrt{1 + ||u||}$ need be computed only once. This information may be set with the options

`MatSNESMFWPSetComputeNormA` (`Mat` mat,`PetscTruth`);
`MatSNESMFWPSetComputeNormU` (`Mat` mat,`PetscTruth`);

or

-snes_mf_compute_norma `<true or false>`
-snes_mf_compute_normu `<true or false>`

This information is used to eliminate the redundant computation of these parameters, therefor reducing the number of collective operations and improving the efficiency of the application code. It is also possible to monitor the differencing parameters h that are computed via the routines

`MatSNESMFSetHHistory` (`Mat` ,`PetscScalar` *,int);
`MatSNESMFResetHHistory` (`Mat` ,`PetscScalar` *,int);
`MatSNESMFGetH` (`Mat` ,`PetscScalar` *);
MatSNESMFKSPMonitor(`KSP` ,int,double,void *);

and the runtime option -snes_mf_ksp_monitor. We include an example in Figure 13 that explicitly uses a matrix-free approach. Note that by using the option -snes_mf one can easily convert any SNES code to use a matrix-free Newton-Krylov method without a preconditioner. As shown in this example, SNESSetFromOptions () must be called *after* SNESSetJacobian () to enable runtime switching between the user-specified Jacobian and the default SNES matrix-free form. Table 7 summarizes the various matrix situations that SNES supports. In particular, different linear system matrices and preconditioning matrices are allowed, as well as both matrix-free and application-provided preconditioners. All combinations are possible, as demonstrated by the example, ${PETSC_DIR}/src/snes/examples/tutorials/ex5.c, in Figure 13.

Matrix Use	**Conventional Matrix Formats**	**Matrix-Free Versions**
Jacobian Matrix	Create matrix with MatCreate (). * Assemble matrix with user-defined routine. †	Create matrix with MatCreateShell (). Use MatShellSetOperation () to set various matrix actions. Or use MatCreateSNESMF ().
Preconditioning Matrix	Create matrix with MatCreate (). * Assemble matrix with user-defined routine. †	Use SNESGetSLES () and SLESGetPC () to access the PC , then use PCSetType (pc,PCSHELL); followed by PCSetApply().

* Use either the generic MatCreate () or a format-specific variant such as MatCreateMPIAIJ ().

† Set user-defined matrix formation routine with SNESSetJacobian ().

Table 7: Jacobian Options

```
/*$Id: ex6.c,v 1.71 2001/08/07 03:04:16 balay Exp $*/

static char help[] = "u`` + u^{2} = f. Different matrices for the Jacobian
and the preconditioner.\n\
Demonstrates the use of matrix-free Newton-Krylov methods in conjunction\n\
with a user-provided preconditioner.  Input arguments are:\n\
   -snes_mf : Use matrix-free Newton methods\n\
   -user_precond : Employ a user-defined preconditioner.  Used only with\n\
                   matrix-free methods in this example.\n\n";

/*T
   Concepts: SNES^different matrices for the Jacobian and preconditioner;
   Concepts: SNES^matrix-free methods
   Concepts: SNES^user-provided preconditioner;
   Concepts: matrix-free methods
   Concepts: user-provided preconditioner;
   Processors: 1
T*/
```

```
/*
   Include "petscsnes.h" so that we can use SNES solvers.  Note that this
   file automatically includes:
     petsc.h       - base PETSc routines   petscvec.h - vectors
     petscsys.h    - system routines       petscmat.h - matrices
     petscis.h     - index sets            petscksp.h - Krylov subspace meth-
ods
     petscviewer.h - viewers               petscpc.h  - preconditioners
     petscsles.h   - linear solvers
*/
#include "petscsnes.h"

/*
   User-defined routines
*/
int FormJacobian(SNES,Vec,Mat*,Mat*,MatStructure*,void*);
int FormFunction(SNES,Vec,Vec,void*);
int MatrixFreePreconditioner(void*,Vec,Vec);

int main(int argc,char **argv)
{
  SNES         snes;                /* SNES context */
  SLES         sles;                /* SLES context */
  PC           pc;                  /* PC context */
  KSP          ksp;                 /* KSP context */
  Vec          x,r,F;               /* vectors */
  Mat          J,JPrec;             /* Jacobian,preconditioner matrices */
  int          ierr,it,n = 5,i,size;
  int          *Shistit = 0,Khistl = 200,Shistl = 10;
  PetscReal    h,xp = 0.0,*Khist = 0,*Shist = 0;
  PetscScalar  v,pfive = .5;
  PetscTruth   flg;

  PetscInitialize(&argc,&argv,(char *)0,help);
  ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
  if (size != 1) SETERRQ(1,"This is a uniprocessor example only!");
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
  h = 1.0/(n-1);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Create nonlinear solver context
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = SNESCreate(PETSC_COMM_WORLD,&snes);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

     Create vector data structures; set function evaluation routine
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*/

  ierr = VecCreate(PETSC_COMM_SELF,&x);CHKERRQ(ierr);
  ierr = VecSetSizes(x,PETSC_DECIDE,n);CHKERRQ(ierr);
  ierr = VecSetFromOptions(x);CHKERRQ(ierr);
  ierr = VecDuplicate(x,&r);CHKERRQ(ierr);
```

```
  ierr = VecDuplicate(x,&F);CHKERRQ(ierr);

  ierr = SNESSetFunction(snes,r,FormFunction,(void*)F);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

     Create matrix data structures; set Jacobian evaluation routine
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*/

  ierr = MatCreateSeqAIJ(PETSC_COMM_SELF,n,n,3,PETSC_NULL,&J);CHKERRQ(ierr);
  ierr = MatCreateSeqAIJ(PETSC_COMM_SELF,n,n,1,PETSC_NULL,&JPrec);CHKERRQ(ierr);

  /*
     Note that in this case we create separate matrices for the Jacobian
     and preconditioner matrix.  Both of these are computed in the
     routine FormJacobian()
  */
  ierr = SNESSetJacobian(snes,J,JPrec,FormJacobian,0);CHKERRQ(ierr);

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Customize nonlinear solver; set runtime options
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  /* Set preconditioner for matrix-free method */
  ierr = PetscOptionsHasName(PETSC_NULL,"-snes_mf",&flg);CHKERRQ(ierr);
  if (flg) {
    ierr = SNESGetSLES(snes,&sles);CHKERRQ(ierr);
    ierr = SLESGetPC(sles,&pc);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-user_precond",&flg);CHKERRQ(ierr);
    if (flg) { /* user-defined precond */
      ierr = PCSetType(pc,PCSHELL);CHKERRQ(ierr);
      ierr = PCShellSetApply(pc,MatrixFreePreconditioner,PETSC_NULL);CHKERRQ(ierr);
    } else {ierr = PCSetType(pc,PCNONE);CHKERRQ(ierr);}
  }

  ierr = SNESSetFromOptions(snes);CHKERRQ(ierr);

  /*
     Save all the linear residuals for all the Newton steps; this enables
us
     to retain complete convergence history for printing after the conclu-
sion
     of SNESSolve().  Alternatively, one could use the monitoring options
           -snes_monitor -ksp_monitor
     to see this information during the solver's execution; however, such
     output during the run distorts performance evaluation data.  So, the
     following is a good option when monitoring code performance, for ex-
ample
     when using -log_summary.
  */
  ierr = PetscOptionsHasName(PETSC_NULL,"-rhistory",&flg);CHKERRQ(ierr);
  if (flg) {
    ierr = SNESGetSLES(snes,&sles);CHKERRQ(ierr);
    ierr = SLESGetKSP(sles,&ksp);CHKERRQ(ierr);
```

```
    ierr = PetscMalloc(Khistl*sizeof(PetscReal),&Khist);CHKERRQ(ierr);
    ierr = KSPSetResidualHistory(ksp,Khist,Khistl,PETSC_FALSE);CHKERRQ(ierr);
    ierr = PetscMalloc(Shistl*sizeof(PetscReal),&Shist);CHKERRQ(ierr);
    ierr = PetscMalloc(Shistl*sizeof(int),&Shistit);CHKERRQ(ierr);
    ierr = SNESSetConvergenceHistory(snes,Shist,Shistit,Shistl,PETSC_FALSE);CHKERRQ(ierr);
  }

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Initialize application:
     Store right-hand-side of PDE and exact solution
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  xp = 0.0;
  for (i=0; i<n; i++) {
    v = 6.0*xp + pow(xp+1.e-12,6.0); /* +1.e-12 is to prevent 0^6 */
    ierr = VecSetValues(F,1,&i,&v,INSERT_VALUES);CHKERRQ(ierr);
    xp += h;
  }

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Evaluate initial guess; then solve nonlinear system
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = VecSet(&pfive,x);CHKERRQ(ierr);
  ierr = SNESSolve(snes,x,&it);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_SELF,"Newton iterations = %d\n\n",it);CHKERRQ(ierr);

  ierr = PetscOptionsHasName(PETSC_NULL,"-rhistory",&flg);CHKERRQ(ierr);
  if (flg) {
    ierr = KSPGetResidualHistory(ksp,PETSC_NULL,&Khistl);CHKERRQ(ierr);
    ierr = PetscRealView(Khistl,Khist,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
    ierr = PetscFree(Khist);CHKERRQ(ierr);CHKERRQ(ierr);
    ierr = SNESGetConvergenceHistory(snes,PETSC_NULL,PETSC_NULL,&Shistl);CHKERRQ(ierr);
    ierr = PetscRealView(Shistl,Shist,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
    ierr = PetscIntView(Shistl,Shistit,PETSC_VIEWER_STDOUT_SELF);CHKERRQ(ierr);
    ierr = PetscFree(Shist);CHKERRQ(ierr);
    ierr = PetscFree(Shistit);CHKERRQ(ierr);
  }

  /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     Free work space.  All PETSc objects should be destroyed when they
     are no longer needed.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

  ierr = VecDestroy(x);CHKERRQ(ierr);     ierr = VecDestroy(r);CHKERRQ(ierr);
  ierr = VecDestroy(F);CHKERRQ(ierr);     ierr = MatDestroy(J);CHKERRQ(ierr);
  ierr = MatDestroy(JPrec);CHKERRQ(ierr); ierr = SNESDestroy(snes);CHKERRQ(ierr);
  ierr = PetscFinalize();CHKERRQ(ierr);

  return 0;
}
/* ------------------------------------------------------------------- */
/*
   FormInitialGuess - Forms initial approximation.
```

```
   Input Parameters:
   user - user-defined application context
   X - vector

   Output Parameter:
   X - vector
 */
int FormFunction(SNES snes,Vec x,Vec f,void *dummy)
{
  PetscScalar *xx,*ff,*FF,d;
  int    i,ierr,n;

  ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
  ierr = VecGetArray(f,&ff);CHKERRQ(ierr);
  ierr = VecGetArray((Vec)dummy,&FF);CHKERRQ(ierr);
  ierr = VecGetSize(x,&n);CHKERRQ(ierr);
  d = (PetscReal)(n - 1); d = d*d;
  ff[0]   = xx[0];
  for (i=1; i<n-1; i++) {
    ff[i] = d*(xx[i-1] - 2.0*xx[i] + xx[i+1]) + xx[i]*xx[i] - FF[i];
  }
  ff[n-1] = xx[n-1] - 1.0;
  ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
  ierr = VecRestoreArray(f,&ff);CHKERRQ(ierr);
  ierr = VecRestoreArray((Vec)dummy,&FF);CHKERRQ(ierr);
  return 0;
}
/* ------------------------------------------------------------------- */
/*
   FormJacobian - This routine demonstrates the use of different
   matrices for the Jacobian and preconditioner

   Input Parameters:
.  snes - the SNES context
.  x - input vector
.  ptr - optional user-defined context, as set by SNESSetJacobian()

   Output Parameters:
.  A - Jacobian matrix
.  B - different preconditioning matrix
.  flag - flag indicating matrix structure
*/
int FormJacobian(SNES snes,Vec x,Mat *jac,Mat *prejac,MatStructure *flag,void
*dummy)
{
  PetscScalar *xx,A[3],d;
  int    i,n,j[3],ierr;

  ierr = VecGetArray(x,&xx);CHKERRQ(ierr);
  ierr = VecGetSize(x,&n);CHKERRQ(ierr);
  d = (PetscReal)(n - 1); d = d*d;

  /* Form Jacobian.  Also form a different preconditioning matrix that
     has only the diagonal elements. */
  i = 0; A[0] = 1.0;
```

```
  ierr = MatSetValues(*jac,1,&i,1,&i,&A[0],INSERT_VALUES);CHKERRQ(ierr);
  ierr = MatSetValues(*prejac,1,&i,1,&i,&A[0],INSERT_VALUES);CHKERRQ(ierr);
  for (i=1; i<n-1; i++) {
    j[0] = i - 1; j[1] = i;                   j[2] = i + 1;
    A[0] = d;     A[1] = -2.0*d + 2.0*xx[i];  A[2] = d;
    ierr = MatSetValues(*jac,1,&i,3,j,A,INSERT_VALUES);CHKERRQ(ierr);
    ierr = MatSetValues(*prejac,1,&i,1,&i,&A[1],INSERT_VALUES);CHKERRQ(ierr);
  }
  i = n-1; A[0] = 1.0;
  ierr = MatSetValues(*jac,1,&i,1,&i,&A[0],INSERT_VALUES);CHKERRQ(ierr);
  ierr = MatSetValues(*prejac,1,&i,1,&i,&A[0],INSERT_VALUES);CHKERRQ(ierr);

  ierr = MatAssemblyBegin(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyBegin(*prejac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(*jac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(*prejac,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

  ierr = VecRestoreArray(x,&xx);CHKERRQ(ierr);
  *flag = SAME_NONZERO_PATTERN;
  return 0;
}
/* ------------------------------------------------------------------- */
/*
   MatrixFreePreconditioner - This routine demonstrates the use of a
   user-provided preconditioner.  This code implements just the null
   preconditioner, which of course is not recommended for general use.

   Input Parameters:
.  ctx - optional user-defined context, as set by PCShellSetApply()
.  x - input vector

   Output Parameter:
.  y - preconditioned vector
*/
int MatrixFreePreconditioner(void *ctx,Vec x,Vec y)
{
  int ierr;
  ierr = VecCopy(x,y);CHKERRQ(ierr);
  return 0;
}
```

Figure 13: Example of Uniprocessor SNES Code - Both Conventional and Matrix-Free Jacobians

5.6 Finite Difference Jacobian Approximations

PETSc provides some tools to help approximate the Jacobian matrices efficiently via finite differences. These tools are intended for use in certain situations where one is unable to compute Jacobian matrices analytically, and matrix-free methods do not work well without a preconditioner, due to very poor conditioning. The approximation requires several steps:

- First, one colors the columns of the (not yet built) Jacobian matrix, so that columns of the same color do not share any common rows.
- Next, one creates a `MatFDColoring` data structure that will be used later in actually computing

the Jacobian.

- Finally, one tells the nonlinear solvers of SNES to use the SNESDefaultComputeJacobianColor () routine to compute the Jacobians.

A code fragment that demonstrates this process is given below.

```
ISColoring iscoloring;
MatFDColoring fdcoloring;
MatStructure str;
/*
This initializes the nonzero structure of the Jacobian. This is artificial
because clearly if we had a routine to compute the Jacobian we wouldn't
need to use finite differences.
*/ FormJacobian(snes,x,&J,&J,&str,&user);
/*
Color the matrix, i.e. determine groups of columns that share no common
rows. These columns in the Jacobian can all be computed simulataneously.
*/
MatGetColoring (J,MATCOLORING_SL,&iscoloring);
/*
Create the data structure that SNESDefaultComputeJacobianColor () uses
to compute the actual Jacobians via finite differences.
*/
MatFDColoringCreate (J,iscoloring,&fdcoloring);
ISColoringDestroy (iscoloring);
MatFDColoringSetFromOptions (fdcoloring);
/*
Tell SNES to use the routine SNESDefaultComputeJacobianColor ()
to compute Jacobians.
*/
SNESSetJacobian (snes,J,J,SNESDefaultComputeJacobianColor ,fdcoloring);
```

Of course, we are cheating a bit. If we do not have an analytic formula for computing the Jacobian, then how do we know what its nonzero structure is so that it may be colored? Determining the structure is problem dependent, but fortunately, for most grid-based problems (the class of problems for which PETSc is designed) if one knows the stencil used for the nonlinear function one can usually fairly easily obtain an estimate of the location of nonzeros in the matrix. One need not necessarily use the routine MatGetColoring () to determine a coloring. For example, if a grid can be colored directly (without using the associated matrix), then that coloring can be provided to MatFDColoringCreate (). Note that the user must always preset the nonzero structure in the matrix regardless of which coloring routine is used. For sequential matrices PETSc provides three matrix coloring routines from the MINPACK package [14]: smallest-last (`sl`), largest-first (`lf`), and incidence-degree (`id`). These colorings, as well as the "natural" coloring for which each column has its own unique color, may be accessed with the command line options

-mat_coloring_type <sl,id,lf,natural>

Alternatively, one can set a coloring type of `COLORING_SL`, `COLORING_ID`, `COLORING_LF`, or `COLORING_NATURAL` when calling MatGetColoring (). As for the matrix-free computation of Jacobians (see Section 5.5), two parameters affect the accuracy of the finite difference Jacobian approximation. These are set with the command

MatFDColoringSetParameters(MatFDColoring fdcoloring,double rerror,double umin);

The parameter `rerror` is the square root of the relative error in the function evaluations, e_{rel}; the default is 10^{-8}, which assumes that the functions are evaluated to full double-precision accuracy. The second parameter, `umin`, is a bit more involved; its default is $10e^{-8}$. Column i of the Jacobian matrix (denoted by $F_{:i}$) is approximated by the formula

$$F'_{:i} \approx \frac{F(u + h * dx_i) - F(u)}{h}$$

where h is computed via

$$h = e_{rel} * u_i \quad \text{if } |u_i| > u_{min}$$
$$h = e_{rel} * u_{min} * sign(u_i) \qquad \text{otherwise.}$$

These parameters may be set from the options database with

-mat_fd_coloring_err ¡err¿
-mat_fd_coloring_umin ¡umin¿

Note that the MatGetColoring() routine currently works only on sequential routines. Extensions may be forthcoming. However, if one can compute the coloring `iscoloring` some other way, the routine MatFDColoringCreate() does work in parallel. An example of this for 2D distributed arrays is given below that uses the utility routine DAGetColoring().

DAGetColoring(da,IS_COLORING_GHOSTED,&iscoloring);
MatFDColoringCreate(J,iscoloring,&fdcoloring);
MatFDColoringSetFromOptions(fdcoloring);
ISColoringDestroy(iscoloring);

Note that the routine MatFDColoringCreate() currently is only supported for the AIJ matrix format.

Chapter 6

TS: Scalable ODE Solvers

The TS library provides a framework for the scalable solution of ODEs arising from the discretization of time-dependent PDEs, and of steady-state problems using pseudo-timestepping. **Time-Dependent Problems:** Consider the ODE

$$u_t = F(u,t),$$

where u is a finite-dimensional vector, usually obtained from discretizing a PDE with finite differences, finite elements, etc. For example, discretizing the heat equation

$$u_t = u_{xx}$$

with centered finite differences results in

$$(u_i)_t = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}.$$

The TS library provides code to solve these equations (currently using the forward or backward Euler method) as well as an interface to other sophisticated ODE solvers, in a clean and easy manner, where the user need only provide code for the evaluation of $F(u,t)$ and (optionally) its associated Jacobian matrix. **Steady-State Problems:** In addition, TS provides a general code for performing pseudo timestepping with a variable timestep at each physical node point. For example, instead of directly attacking the steady-state problem

$$F(u) = 0,$$

we can use pseudo-transient continuation by solving

$$u_t = F(u).$$

By using time differencing with the backward Euler method, we obtain

$$\frac{u^{n+1} - u^n}{dt^n} = F(u^{n+1}).$$

More generally we can consider a diagonal matrix Dt^n that has a pseudo-timestep for each node point to obtain the series of nonlinear equations

$$Dt^{n^{-1}}(u^{n+1} - u^n) = F(u^{n+1}).$$

For this problem the user must provide $F(u)$ and the diagonal matrix Dt^n, (or optionally, if the timestep is position independent, a scalar timestep) as well as optionally the Jacobian of $F(u)$.

6.1 Basic Usage

The user first creates a TS object with the command

int TSCreate (MPI_Comm comm ,TSProblemType problemtype,TS *ts);

The TSProblemType is one of TS_LINEAR or TS_NONLINEAR, to indicate whether $F(u,t)$ is given by a matrix A, or $A(t)$, or a function $F(u,t)$. One can set the solution method with the routine

TSSetType (TS ts,TSType type);

Currently supported types are TS_EULER, TS_BEULER, and TS_PSEUDO or the command line option `-ts_type euler, beuler, pseudo`. Set the initial time and timestep with the command

TSSetInitialTimeStep (TS ts,double time,double dt);

One can change the timestep with the command

TSSetTimeStep (TS ts,double dt);

One can determine the current timestep with the routine

TSGetTimeStep (TS ts,double* dt);

Here, "current" refers to the timestep being used to attempt to promote the solution form u^n to u^{n+1}. One sets the total number of timesteps to run or the total time to run (whatever is first) with the command

TSSetDuration (TS ts,int maxsteps,double maxtime);

One sets up the timestep context with

TSSetUp (TS ts);

destroys it with

TSDestroy (TS ts);

and views it with

TSView (TS ts,PetscViewer viewer);

6.1.1 Solving Time-dependent Problems

To set up TS for solving an ODE, one must set the following:

- Solution:

  TSSetSolution (TS ts, Vec initialsolution);

  The vector `initialsolution` should contain the "initial conditions" for the ODE.

- Function:

  - For linear functions (solved with implicit timestepping), the user must call

TSSetRHSMatrix (TS ts,Mat A, Mat B,
int (*f)(TS ,double,Mat *,Mat *,MatStructure *,void*),void *fP);

The matrix B (although usually the same as A) allows one to provide a different matrix to be used in the construction of the preconditioner. The function f is used to form the matrices A and B at each timestep if the matrices are time dependent. If the matrix does not depend on time, the user should pass in PETSC_NULL for f. The variable fP allows users to pass in an application context that is passed to the f() function whenever it is called, as the final argument. The user must provide the matrices A and B; if they have the right-hand side only as a linear function, they must construct a MatShell matrix. Note that this is the same interface as that for SNESSetJacobian ().

- For nonlinear problems (or linear problems solved using explicit timestepping methods) the user passes the function with the routine

TSSetRHSFunction (TS ts,int (*f)(TS ,double,Vec ,Vec ,void*),void *fP);

The arguments to the function f() are the timestep context, the current time, the input for the function, the output for the function, and the (optional) user-provided context variable fP.

- Jacobian: For nonlinear problems the user must also provide the (approximate) Jacobian matrix of F(u,t) and a function to compute it at each Newton iteration. This is done with the command

  TSSetRHSJacobian (TS ts,Mat A, Mat B,int (*fjac)(TS ,double,Vec ,Mat *,Mat *,
  MatStructure *,void*),void *fP);

  The arguments for the function fjac() are the timestep context, the current time, the location where the Jacobian is to be computed, the Jacobian matrix, an alternative approximate Jacobian matrix used as a preconditioner, and the optional user-provided context, passed in as fP. The user must provide the Jacobian as a matrix; thus, if using a matrix-free approach is used, the user must create a MatShell matrix. Again, note the similarity to SNESSetJacobian ().

Similar to SNESDefaultComputeJacobianColor () is the routine TSDefaultComputeJacobianColor () and TSDefaultComputeJacobian() that corresponds to SNESDefaultComputeJacobian ().

6.1.2 Using PVODE from PETSc

PVODE is a parallel ODE solver developed by Hindmarsh et al. at LLNL. The TS library provides an interface to use PVODE directly from PETSc. (To install PETSc to use PVODE, see the installation guide, docs/installation/index.htm.)

To use the PVODE integrators, call

TSSetType (TS ts,TSType TS_PVODE);

or use the command line option -ts_type pvode. PVODE comes with to main integrator families, Adams and BDF (backward differentiation formula). One can select these with

TSPVodeSetType (TS ts,TSPVodeType [PVODE_ADAMS,PVODE_BDF]);

or the command line option -ts_pvode_type <adams,bdf>. BDF is the default. PVODE does not use the SNES library within PETSc for its nonlinear solvers, so one cannot change the nonlinear solver options via SNES . Rather, PVODE uses the preconditioners within the PC package of PETSc, which can be accessed via

TSPVodeGetPC(TS ts,PC *pc);

The user can then directly set preconditioner options; alternatively, the usual runtime options can be employed via `-pc_xxx`. Finally, one can set the PVODE tolerances via

TSPVodeSetTolerance(TS ts,double abs,double rel);

where `abs` denotes the absolute tolerance and `rel` the relative tolerance. Other PETSc-PVode options include

TSPVodeSetGramSchmidtType(TS ts,TSPVodeGramSchmidtType type);

where `type` is either `PVODE_MODIFIED_GS` or `PVODE_UNMODIFIED_GS`. This may be set via the options data base with `-ts_pvode_gramschmidt_type <modifed,unmodified>`. The routine

TSPVodeSetGMRESRestart(TS ts,int restart);

sets the number of vectors in the Krylov subpspace used by GMRES. This may be set in the options database with `-ts_pvode_gmres_restart restart`.

6.1.3 Solving Steady-State Problems with Pseudo-Timestepping

For solving steady-state problems with pseudo-timestepping one proceeds as follows.

- Provide the function `F(u)` with the routine

  TSSetRHSFunction(TS ts,int (*f)(TS ,double,Vec ,Vec ,void*),void *fP);

  The arguments to the function `f()` are the timestep context, the current time, the input for the function, the output for the function and the (optional) user-provided context variable `fP`.

- Provide the (approximate) Jacobian matrix of `F(u,t)` and a function to compute it at each Newton iteration. This is done with the command

  TSSetRHSJacobian(TS ts,Mat A, Mat B,int (*f)(TS ,double,Vec ,Mat *,Mat *,
  MatStructure *,void*),void *fP);

  The arguments for the function `f()` are the timestep context, the current time, the location where the Jacobian is to be computed, the Jacobian matrix, an alternative approximate Jacobian matrix used as a preconditioner, and the optional user-provided context, passed in as `fP`. The user must provide the Jacobian as a matrix; thus, if using a matrix-free approach, one must create a `MatShell` matrix.

In addition, the user must provide a routine that computes the pseudo-timestep. This is slightly different depending on if one is using a constant timestep over the entire grid, or it varies with location.

- For location-independent pseudo-timestepping, one uses the routine

  TSPseudoSetTimeStep(TS ts,int(*dt)(TS ,double*,void*),void* dtctx);

  The function `dt` is a user-provided function that computes the next pseudo-timestep. As a default one can use `TSPseudoDefaultTimeStep(TS,double*,void*)` for `dt`. This routine updates the pseudo-timestep with one of two strategies: the default

$$dt^n = dt_{increment} * dt^{n-1} * \frac{||F(u^{n-1})||}{||F(u^n)||}$$

or, the alternative,

$$dt^n = dt_{increment} * dt^0 * \frac{||F(u^0)||}{||F(u^n)||}$$

which can be set with the call

TSPseudoIncrementDtFromInitialDt(TS ts);

or the option -ts_pseudo_increment_dt_from_initial_dt. The value $dt_increment$ is by default 1.1, but can be reset with the call

TSPseudoSetTimeStepIncrement(TS ts,double inc);

or the option -ts_pseudo_increment <inc>.

- For location-dependent pseudo-timestepping, the interface function has not yet been created.

Chapter 7

High Level Support for Multigrid with DMMG

PETSc provides an easy to use high-level interface for multigrid on a single structured grid using the PETSc DA object (or the VecPack object) to decompose the grid across the processors. This DMMG code is built on top of the lower level PETSc multigrid interface provided in the PCType of MG, see Section 4.4.7. Currently we only provide piecewise linear and piecewise constant interpolation, but can add more if needed. The DMMG routines only provide linear multigrid but they can be used easily with either SLES (for linear problems) or SNES (for nonlinear problems). For linear problems the examples `src/sles/examples/tutorials/ex22.c` and `ex25.c` can be used to guide your development. We give a short summary here.

DMMG *dmmg;
DA da;
/* Create the DA that stores information about the coarsest grid you wish to use */
ierr = DACreate3d (PETSC_COMM_WORLD,DA _NONPERIODIC,DA _STENCIL_STAR,
3,3,3,PETSC_DECIDE,PETSC_DECIDE,PETSC_DECIDE,1,1,0,0,0,&da);CHKERRQ (ierr);
/* Create the DMMG data structure
- the second argument indicates the number of levels you wish to use and
can be changed with the option -dmmg_nlevels
ierr = DMMGCreate (PETSC_COMM_WORLD,3,PETSC_NULL,&dmmg);CHKERRQ (ierr);
/* Tell the DMMG object to use the da to define the coarsest grid
ierr = DMMGSetDM (dmmg,(DM)da);
/* Tell the DMMG we are solving a linear problem (hence SLES) and provide the
callback function to compute the right hand side and matrices for each level
ierr = DMMGSetSLES (dmmg,ComputeRHS,ComputeJacobian);CHKERRQ (ierr);
/* Solve the problem */
ierr = DMMGSolve (dmmg);CHKERRQ (ierr);
/* One can access the solution with */
DMMGGetx (dmmg)
ierr = DMMGDestroy (dmmg);CHKERRQ (ierr);
ierr = DADestroy (da);CHKERRQ (ierr);

The option `-dmmg_ksp_monitor` causes the DMMG code to print the residual norms for each level of the solver to the screen so that the coarser the grid the more indented the print out. The option `-dmmg_grid_sequence` causes the DMMG solve to use grid sequencing to generate the initial guess by solving the same problem on the previous coarser grid; this often results in a much faster time to solution. The solver (smoother) used on each level but the coarsest can be controled via the options database with any PC or KSP option prefixed as -mg_levels_[pc/ksp]_. The solver options on the finest grid can be set with -mg_coarse_[pc/ksp]_. The DMMG has many other options that can view by running the DMMG program with the option -help. You should generally run your code with the option -sles_view to see exactly what solvers are being used. For nonlinear problems one replaces the DMMGSetSLES () with

DMMGSetSNES (DMMG *dmmg,int (*function)(SNES ,Vec ,Vec ,void*),
int (*jacobian)(SNES ,Vec ,Mat *,Mat *,MatStructure *,void*))

or the prefered approach

DMMGSetSNESLocal (DMMG *dmmg,
int (*localfunction)(DALocalInfo *info,void *x,void *f,void* appctx),
int (*localjacobian)(DALocalInfo *,void *x,Mat J,void *appctx),ad_function,ad_mf_function);

The ad_function and ad_mf_function are described in the next chapter. See examples src/snes/examples/tutorials/ex18.c and ex19.c for complete details. For scalar problems (problems with one degree of freedom per node), the localfunction x and f arguments are simply multi-dimensional arrays of double precision (or complex) numbers (according to the dimension of the grid) that should be indexed using *global* i, j, k indices on the entire grid. For multi-component problems you must create a C struct with an entry for each component and the x and f arguments are appropriately dimensioned arrays of that struct. For example, for a 3d scalar problem the function would be

int localfunction(DALocalInfo *info,double ***x, double ***f,void *ctx)

For a 2d multi-component problem with u, v, and p components one would write

typedef struct {
PetscScalar u,v,p;
} Field;
...
int localfunction(DALocalInfo *info,Field **x,Field **f,void *ctx)

For many nonlinear problems it is too difficult to compute the Jacobian analytically, thus if jacobian or localjacobian is not provided, (indicated by passing in a PETSC_NULL) the DMMG will compute the sparse Jacobian reasonably efficiently automatically using finite differencing. See the next chapter on computing the Jacobian via automatic differentiation. The option -dmmg_jacobian_mf_fd causes the code to not compute the Jacobian explicitly but rather to use differences to apply the matrix vector product of the Jacobian. The option -dmmg_snes_monitor can be used to monitor the progress of the nonlinear solver. The usual -snes_ options may be used to control the nonlinearr solves. Again we recommend using the option -dmmg_grid_sequence and -snes_view for most runs.

Chapter 8

Using ADIC and ADIFOR with PETSc

Automatic differentiation is an incredible technique to generate code that computes Jacobians and other differentives directly from code that only evaluates the function. For structured grid problems, via the `DMMG` interface (see Chapter 7) PETSc provides a way to use ADIFOR and ADIC to compute the sparse Jacobians or perform matrix free vector products with them. See `src/snes/examples/tutorials/ex18.c` and `ex5f.F` for example usage.

First one indicates the functions for which one needs Jacobians by adding in the comments in the code

/* Process adiC: FormFunctionLocal FormFunctionLocali */

where one lists the functions. In Fortran use

! Process adifor: FormFunctionLocal

Next one uses the call

`DMMGSetSNESLocal`(`DMMG` *dmmg,
int (*localfunction)(`DALocalInfo` *info,void *x,void *f,void* appctx),PETSC_NULL,
ad_localfunction,ad_mf_localfunction);

where the names of the last two functions are obtained by prepending an `ad_` and `ad_mf_` in front of the function name. In Fortran, this is done by prepending a `g_` and `m_`. Two useful options are `-dmmg_jacobian_mf_ad` and `-dmmg_jacobian_mf_ad_operator`, with the former is uses the matrix-free automatic differentiation to apply the operator and to define the preconditioner operator. The latter form uses the matrix-free for the matrix-vector product but still computes the Jacobian (by default with finite differences) used to construct the preconditioner.

8.1 Work arrays inside the local functions

In C you can call `DAGetArray`() to get work arrays (this is low overhead). In Fortran you can provide a `FormFunctionLocal()` that had local arrays that have hardwired sizes that are large enough or somehow allocate space and pass it into an inner FormFunctionLocal() that is the one you differentiate; this second approach will require some hand massaging. For example,

subroutine TrueFormFunctionLocal(info,x,f,ctx,ierr)
double precision x(gxs:gxe,gys:gye),f(xs:xe,ys:ye)
`DA` info(`DA`_LOCAL_INFO_SIZE)
integer ctx,ierr
double precision work(gxs:gxe,gys:gye)

```
.... do the work ....
return
subroutine FormFunctionLocal(info,x,f,ctx,ierr)
double precision x(*),f(*)
DA info(DA_LOCAL_INFO_SIZE)
integer ctx,ierr
double precision work(10000)
call TrueFormFunctionLocal(info,x,f,work,ctx,ierr)
return
```

Chapter 9

Using Matlab with PETSc

There are three basic ways to use Matlab with PETSc: (1) dumping files to be read into Matlab, (2) automatically sending data from a running PETSc program to a Matlab process where you may interactively type Matlab commands (or run scripts) and (3) automatically sending data back and forth between PETSc and Matlab where Matlab commands are issued not interactively but from a script or the PETSc program.

9.1 Dumping Data for Matlab

One can dump PETSc matrices and vectors to the screen (and thus save in a file via `> filename.m`) in a format that Matlab can read in directly. This is done with the command line options `-vec_view_matlab` or `-mat_view_matlab`. This causes the PETSc program to print the vectors and matrices every time a VecAssemblyXXX() and MatAssemblyXXX() is called. To provide finer control over when and what vectors and matrices are dumped one can use the `VecView` () and `MatView` () functions with a viewer type of ASCII (see `PetscViewerASCIIOpen` (), `PETSC_VIEWER_STDOUT_WORLD`, `PETSC_VIEWER_STDOUT_SELF`, or `PETSC_VIEWER_STDOUT_(MPI_Comm)`). Before calling the viewer set the output type with, for example,

```
PetscViewerSetFormat (PETSC_VIEWER_STDOUT_WORLD,PETSC_VIEWER_ASCII_MATLAB);
VecView (A,PETSC_VIEWER_STDOUT_WORLD);
```

or

```
PetscViewerPushFormat (PETSC_VIEWER_STDOUT_WORLD,PETSC_VIEWER_ASCII_MATLAB);
MatView (B,PETSC_VIEWER_STDOUT_WORLD);
```

The name of each PETSc variable printed for Matlab may be set with

```
PetscObjectSetName ((PetscObject )A,"name");
```

9.2 Sending Data to Interactive Running Matlab Session

One creates a viewer to Matlab via

```
PetscViewerSocketOpen (MPI_Comm,char *machine,int port,PetscViewer *v);
```

(port is usally set to `PETSC_DEFAULT`, use `PETSC_NULL` for the machine if the Matlab interactive session is running on the same machine as the PETSc program) and then sends matrices or vectors via

```
VecView (Vec A,v);
MatView (Mat B,v);
```

One can also send arrays or integer arrays via

`PetscViewerSocketPutScalar` (v,int m,int n,`PetscScalar` *array);
`PetscViewerSocketPutReal` (v,int m,int n,double *array);
`PetscViewerSocketPutInt` (v,int m,int *array);

One may start the Matlab program manually or use the PETSc command `PetscStartMatlab(MPI_Comm,char *machine,char *script,FILE **fp);` where machine and script may be `PETSC_NULL`. To receive the objects in Matlab you must first make sure that `${PETSC_DIR}/bin/matlab` is in your Matlab path. Use `p = openport;` (or `p = openport(portnum)` if you provided a port number in your call to `PetscViewerSocketOpen` ()), then `a = receive(p);` returns the object you have passed from PETSc. `receive()` may be called any number of times. Each call should correspond on the PETSc side with viewing a single vector or matrix. You many call `closeport()` to close the connection from Matlab. It is also possible to start your PETSc program from Matlab via `launch()`.

9.3 Using the Matlab Compute Engine

One creates access to the Matlab engine via

`PetscMatlabEngineCreate` (MPI_Comm `comm` ,char *machine,`PetscMatlabEngine` *e);

(`machine` may be `PETSC_NULL`). One can send objects to Matlab via

`PetscMatlabEnginePut` (`PetscMatlabEngine` e,`PetscObject` obj);

One can get objects via

`PetscMatlabEngineGet` (`PetscMatlabEngine` e,`PetscObject` obj);.

Similarly one can send arrays via

`PetscMatlabEnginePutArray` (`PetscMatlabEngine` e,int m,int n,`PetscScalar` *array,char *name);

and get them back via

`PetscMatlabEngineGetArray` (`PetscMatlabEngine` e,int m,int n,`PetscScalar` *array,char *name);

One cannot use Matlab interactively in this mode but you can send Matlab commands via

`PetscMatlabEngineEvaluate` (`PetscMatlabEngine` ,"format",...);

where `format` has the usual `printf()` format. For example,

`PetscMatlabEngineEvaluate` (`PetscMatlabEngine` ,"x =

The name of each PETSc variable passed to Matlab may be set with

`PetscObjectSetName` ((`PetscObject`)A,"name");

Text responses can be returned from Matlab via

`PetscMatlabEngineGetOutput` (`PetscMatlabEngine` ,char **);

or

PetscMatlabEnginedPrintOutput(`PetscMatlabEngine` ,FILE*).

There is a short-cut to starting the Matlab engine with `PETSC_MATLAB_ENGINE_(MPI_Comm)`.

Chapter 10

Using ESI with PETSc

The Equation Solver Interface (location of official headers) is an attempt to define a common linear solver interface for a variety of scalable solver packages. It is currently only for C++ and defines a set of abstract C++ classes (the header files for these classes are in `include/esi`). PETSc provides code that allows

- ESI objects to be wrapped as PETSc objects and then be used with “regular” PETSc code and
- PETSc objects to be wrapped as ESI objects and then used with other ESI code.

The wrapping does not involve data copies and thus is efficient. To wrap ESI objects as PETSc objects use

```
VecCreate (MPI_Comm,Vec *x);
VecSetType (x,VECESI);
VecESISetVector (s,::esi::Vector<double,int> *v);

MatCreate (MPI_Comm,...,Mat *A);
MatSetType (A,MATESI);
MatESISetOperator(A,::esi::Operator<double,int> *a);

PCCreate (MPI_Comm,PC *pc);
PCSetType (pc,PCESI);
PCESISetPreconditioner (A,::esi::Preconditioner<double,int> *p);
```

To wrap PETSc objects as ESI objects

```
int VecESIWrap(Vec xin,::esi::Vector<double,int> **v);
eis = new esi::petsc::IndexSpace<int>::IndexSpace(PetscMap map);
evec = new esi::petsc::Vector<double,int>::Vector(Vec vec);
emat = new esi::petsc::Operator<double,int>::Operator(Mat mat);
epc = new esi::petsc::Preconditioner<double,int>::Preconditioner(PC pc);
```

One can also create ESI objects directly with

```
eis = new esi::petsc::IndexSpace<int>::IndexSpace(MPI_Comm icomm, int n, int N)
esi = new esi::petsc::IndexSpace<int>::IndexSpace(::esi::IndexSpace<int> &sourceIndexSpace)
evec = new esi::petsc::Vector<double,int>::Vector( ::esi::IndexSpace<int> *inmap)
emat = new esi::petsc::Matrix<double,int>::Matrix(esi::IndexSpace<int> *inrmap,esi::IndexSpace<int> *incmap)
```

PETSc also provides mechanisms to load ESI classes directly from dynamic libraries to be used with PETSc.

VecESISetType (Vec pc,char *name);
MatESISetType(Mat pc,char *name)
PCESISetType (PC pc,char *name)

where `name` is the full name of the class; for example, `esi::petsc::Vector` or `Petra_ESI_Vector`. These are implemented by loading a factory which then creates the actual class object. This can also be achieved from the command line with

-is_esi_type esi::petsc::IndexSpace
-vec_type esi -vec_esi_type esi::petsc::Vector
-mat_type esi -mat_esi_type esi::petsc::Matrix
-pc_type esi -pc_esi_type esi:petsc::Preconditioner

Note that the `is` in `-is_esi_type` is short for IndexSpace, **not** PETSc IS objects. Finally when using the PETSc ESI objects (not Trilinos or some other implementation), one can set the underly PETSc object type to the PETSc objected that is wrapped as ESI with

-vec_type esi -vec_esi_type esi::petsc::Vector -esi_vec_type <seq,mpi>
-mat_type esi -mat_esi_type esi::petsc::Matrix -esi_mat_type <seqaij,mpiaij,...>
-pc_type esi -pc_esi_type esi:petsc::Preconditioner -esi_pc_type <lu,ilu,....>

Mostly for testing purposes PETSc provides a short hand way of using ESI with the PETSc ESI objects. One can use

-vec_type petscesi -esi_vec_type <seq,mpi>
-mat_type petscesi -esi_mat_type <seqaij,mpiaij,...>
-pc_type petscesi -esi_pc_type <lu,ilu,....>

Chapter 11

PETSc for Fortran Users

Most of the functionality of PETSc can be obtained by people who program purely in Fortran 77 or Fortran 90. The PETSc Fortran interface works with both F77 and F90 compilers. Since Fortran77 does not provide type checking of routine input/output parameters, we find that many errors encountered within PETSc Fortran programs result from accidentally using incorrect calling sequences. Such mistakes are immediately detected during compilation when using C/C++. Thus, using a mixture of C/C++ and Fortran often works well for programmers who wish to employ Fortran for the core numerical routines within their applications. In particular, one can effectively write PETSc driver routines in C/C++, thereby preserving flexibility within the program, and still use Fortran when desired for underlying numerical computations.

11.1 Differences between PETSc Interfaces for C and Fortran

Only a few differences exist between the C and Fortran PETSc interfaces, all of which are due to Fortran 77 syntax limitations. Since PETSc is primarily written in C, the FORTRAN 90 dynamic allocation is not easily accessible. All Fortran routines have the same names as the corresponding C versions, and PETSc command line options are fully supported. The routine arguments follow the usual Fortran conventions; the user need not worry about passing pointers or values. The calling sequences for the Fortran version are in most cases identical to the C version, except for the error checking variable discussed in Section 11.1.2 and a few routines listed in Section 11.1.10.

11.1.1 Include Files

The Fortran include files for PETSc are located in the directory `${PETSC_DIR}/include/finclude` and should be used via statements such as the following:

#include ”include/finclude/includefile.h”

Since one must be very careful to include each file no more than once in a Fortran routine, application programmers must manually include each file needed for the various PETSc libraries within their program. This approach differs from the PETSc C/C++ interface, where the user need only include the highest level file, for example, `petscsnes.h`, which then automatically includes all of the required lower level files. As shown in the examples of Section 11.2, in Fortran one must explicitly list *each* of the include files. One must employ the Fortran file suffix `.F` rather than `.f`. This convention enables use of the CPP preprocessor, which allows the use of the *#include* statements that define PETSc objects and variables. (Familarity with the CPP preprocessor is not needed for writing PETSc Fortran code; one can simply begin by copying a PETSc Fortran example and its corresponding makefile.)

11.1.2 Error Checking

In the Fortran version, each PETSc routine has as its final argument an integer error variable, in contrast to the C convention of providing the error variable as the routine's return value. The error code is set to be nonzero if an error has been detected; otherwise, it is zero. For example, the Fortran and C variants of `SLESSolve` () are given, respectively, below, where `ierr` denotes the error variable:

```
call SLESSolve (SLES sles,Vec b,Vec x,int its,int ierr)
SLESSolve (SLES sles,Vec b,Vec x,int *its);
```

Fortran programmers can check these error codes with `CHKERRQ(ierr)`, which terminates all process when an error is encountered. Likewise, one can set error codes within Fortran programs by using `SETERRQ(ierr,p,' ')`, which again terminates all processes upon detection of an error. Note that complete error tracebacks with `CHKERRQ()` and `SETERRQ()`, as described in Section 1.4 for C routines, are *not* directly supported for Fortran routines; however, Fortran programmers can easily use the error codes in writing their own tracebacks. For example, one could use code such as the following:

```
call SLESSolve (sles,x,y,ierr)
if ( ierr .ne. 0) then
print*, 'Error in routine ...'
return
endif
```

The most common reason for crashing PETSc Fortran code is forgetting the final `ierr` argument.

11.1.3 Array Arguments

Since Fortran 77 does not allow arrays to be returned in routine arguments, all PETSc routines that return arrays, such as `VecGetArray` (), `MatGetArray` (), `ISGetIndices` (), and `DAGetGlobalIndices` () are defined slightly differently in Fortran than in C. Instead of returning the array itself, these routines accept as input a user-specified array of dimension one and return an integer index to the actual array used for data storage within PETSc. The Fortran interface for several routines is as follows:

```
double precision xx_v(1), aa_v(1)
integer ss_v(1), dd_v(1), ierr, nloc
PetscOffset ss_i, xx_i, aa_i, dd_i
Vec x
Mat A
IS s
DA d
call VecGetArray (x,xx_v,xx_i,ierr)
call MatGetArray (A,aa_v,aa_i,ierr)
call ISGetIndices (s,ss_v,ss_i,ierr)
call DAGetGlobalIndices (d,nloc,dd_v,dd_i,ierr)
```

To access array elements directly, both the user-specified array and the integer index *must* then be used together. For example, the following Fortran program fragment illustrates directly setting the values of a vector array instead of using `VecSetValues()`. Note the (optional) use of the preprocessor `#define` statement to enable array manipulations in the conventional Fortran manner.

```
#define xx_a(ib) xx_v(xx_i + (ib))
double precision xx_v(1)
```

```
PetscOffset xx_i
integer i, ierr, n
Vec x
call VecGetArray (x,xx_v,xx_i,ierr)
call VecGetLocalSize (x,n,ierr)
do 10, i=1,n
xx_a(i) = 3*i + 1
10 continue
call VecRestoreArray (x,xx_v,xx_i,ierr)
```

Figure 15 contains an example of using `VecGetArray` () within a Fortran routine. Since in this case the array is accessed directly from Fortran, indexing begins with 1, not 0 (unless the array is declared as `xx_v(0:1)`). This is different from the use of `VecSetValues` () where, indexing always starts with 0. *Note*: If using `VecGetArray` (), `MatGetArray` (), `ISGetIndices` (), or `DAGetGlobalIndices` () from Fortran, the user *must not* compile the Fortran code with options to check for "array entries out of bounds" (e.g., on the IBM RS/6000 this is done with the `-C` compiler option, so never use the `-C` option with this).

11.1.4 Calling Fortran Routines from C (and C Routines from Fortran)

Different machines have different methods of naming Fortran routines called from C (or C routines called from Fortran). Most Fortran compilers change all the capital letters in Fortran routines to small. On some machines, the Fortran compiler appends an underscore to the end of each Fortran routine name; for example, the Fortran routine `Dabsc()` would be called from C with `dabsc_()`. Other machines change all the letters in Fortran routine names to capitals.

PETSc provides two macros (defined in C/C++) to help write portable code that mixes C/C++ and Fortran. They are `PETSC_HAVE_FORTRAN_UNDERSCORE` and `PETSC_HAVE_FORTRAN_CAPS` , which are defined in the file `${PETSC_DIR}/bmake/${PETSC_ARCH}/petscconf.h`. The macros are used, for example, as follows:

```
#if defined(PETSC_HAVE_FORTRAN_CAPS)
#define dabsc_ DABSC
#elif !defined(PETSC_HAVE_FORTRAN_UNDERSCORE)
#define dabsc_ dabsc
#endif
.....
dabsc_(&n,x,y); /* call the Fortran function */
```

11.1.5 Passing Null Pointers

In several PETSc C functions, one has the option of passing a 0 (null) argument (for example, the fifth argument of `MatCreateSeqAIJ` ()). From Fortran, users *must* pass `PETSC_NULL_XXX` to indicate a null argument (where XXX is `INTEGER`, `DOUBLE`, `CHARACTER`, or `SCALAR` depending on the type of argument required); passing 0 from Fortran will crash the code. Note that the C convention of passing `PETSC_NULL` (or 0) *cannot* be used. For example, when no options prefix is desired in the routine `PetscOptionsGetInt` (), one must use the following command in Fortran:

```
call PetscOptionsGetInt (PETSC_NULL_CHARACTER,'-name',N,flg,ierr)
```

This Fortran requirement is inconsistent with C, where the user can employ `PETSC_NULL` for all null arguments.

11.1.6 Duplicating Multiple Vectors

The Fortran interface to VecDuplicateVecs () differs slightly from the C/C++ variant because Fortran does not allow arrays to be returned in routine arguments. To create `n` vectors of the same format as an existing vector, the user must declare a vector array, `v_new` of `size n`. Then, after VecDuplicateVecs () has been called, `v_new` will contain (pointers to) the new PETSc vector objects. When finished with the vectors, the user should destroy them by calling VecDestroyVecs (). For example, the following code fragment duplicates `v_old` to form two new vectors, `v_new(1)` and `v_new(2)`.

```
Vec v_old, v_new(2)
integer ierr
PetscScalar alpha
....
call VecDuplicateVecs (v_old,2,v_new,ierr)
alpha = 4.3
call VecSet (alpha,v_new(1),ierr)
alpha = 6.0
call VecSet (alpha,v_new(2),ierr)
....
call VecDestroyVecs (v_new,2,ierr)
```

11.1.7 Matrix and Vector Indices

All matrices and vectors in PETSc use zero-based indexing, regardless of whether C or Fortran is being used. The interface routines, such as MatSetValues () and VecSetValues (), always use zero indexing. See Section 3.2 for further details.

11.1.8 Setting Routines

When a routine is set from within a Fortran program by a routine such as KSPSetConvergenceTest (), that routine is assumed to be a Fortran routine. Likewise, when a routine is set from within a C program, that routine is assumed to be written in C.

11.1.9 Compiling and Linking Fortran Programs

Figure 21 shows a sample makefile that can be used for PETSc programs. In this makefile, one can compile and run a debugging version of the Fortran program `ex3.F` with the actions `make BOPT=g ex3` and `make runex3`, respectively. The compilation command is restated below:

```
ex3: ex3.o
 -${FLINKER} -o ex3 ex3.o $PETSC_FORTRAN_LIB $PETSC_LIB
 ${RM} ex3.o
```

Note that the PETSc Fortran interface library, given by `${PETSC_FORTRAN_LIB}`, *must* precede the base PETSc libraries, given by `${PETSC_LIB}`, on the link line.

11.1.10 Routines with Different Fortran Interfaces

The following Fortran routines differ slightly from their C counterparts; see the manual pages and previous discussion in this chapter for details:

PetscInitialize (char *filename,int ierr)
PetscError (int err,char *message,int ierr)
VecGetArray (), MatGetArray ()
ISGetIndices (), DAGetGlobalIndices ()
VecDuplicateVecs (), VecDestroyVecs ()
PetscOptionsGetString ()

The following functions are not supported in Fortran:

PetscFClose (), PetscFOpen (), PetscFPrintf (), PetscPrintf ()
PetscPopErrorHandler (), PetscPushErrorHandler ()
PetscLogInfo ()
PetscSetDebugger ()
VecGetArrays (), VecRestoreArrays ()
PetscViewerASCIIGetPointer (), PetscViewerBinaryGetDescriptor ()
PetscViewerStringOpen (), PetscViewerStringSPrintf ()
PetscOptionsGetStringArray ()

11.1.11 Fortran90

PETSc includes limited support for direct use of Fortran90 pointers. Current routines include:

VecGetArrayF90 (), VecRestoreArrayF90 ()
VecDuplicateVecsF90 (), VecDestroyVecsF90 ()
DAGetGlobalIndicesF90 ()
MatGetArrayF90 (), MatRestoreArrayF90 ()
ISGetIndicesF90 (), ISRestoreIndicesF90 ()

See the manual pages for details and pointers to example programs. To use the routines VecGetArrayF90 (), VecRestoreArrayF90 () VecDuplicateVecsF90 (), and VecDestroyVecsF90 (), one must use the Fortran90 vector include file,

#include "include/finclude/petscvec.h90"

Analogous include files for other libraries are `petscda.h90`, `petscmat.h90`, and `petscis.h90`. Unfortunately, these routines currently work only on certain machines with certain compilers. They currently work with the SGI, Solaris, the Cray T3E, the IBM and the NAG Fortran 90 compiler.

11.2 Sample Fortran77 Programs

Sample programs that illustrate the PETSc interface for Fortran are given in Figures 14 — 17, corresponding to `${PETSC_DIR}/src/vec/examples/tests/ex19.F`, `${PETSC_DIR}/src/vec/examples/tutorials/ex4f.F`, `${PETSC_DIR}/src/draw/examples/tests/ex5.F`, and `${PETSC_DIR}/src/snes/examples/ex1f.F`, respectively. We also refer Fortran programmers to the C examples listed throughout the manual, since PETSc usage within the two languages differs only slightly.

```
!
!    "$Id: ex19.F,v 1.38 2001/08/10 03:08:33 balay Exp $";
!
      program main
#include "include/finclude/petsc.h"
```

```
#include "include/finclude/petscvec.h"
!
!  This example demonstrates basic use of the PETSc Fortran interface
!  to vectors.
!
       integer          n,ierr,flg
       PetscScalar      one,two,three,dot
       PetscReal        norm,rdot
       Vec              x,y,w

       n     = 20
       one   = 1.0
       two   = 2.0
       three = 3.0

       call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
       call PetscOptionsGetInt(PETSC_NULL_CHARACTER,'-n',n,flg,ierr)

! Create a vector, then duplicate it
       call VecCreate(PETSC_COMM_WORLD,x,ierr)
       call VecSetSizes(x,PETSC_DECIDE,n,ierr)
       call VecSetFromOptions(x,ierr)
       call VecDuplicate(x,y,ierr)
       call VecDuplicate(x,w,ierr)

       call VecSet(one,x,ierr)
       call VecSet(two,y,ierr)

       call VecDot(x,y,dot,ierr)
       rdot = PetscRealPart(dot)
       write(6,100) rdot
  100  format('Result of inner product ',f10.4)

       call VecScale(two,x,ierr)
       call VecNorm(x,NORM_2,norm,ierr)
       write(6,110) norm
  110  format('Result of scaling ',f10.4)

       call VecCopy(x,w,ierr)
       call VecNorm(w,NORM_2,norm,ierr)
       write(6,120) norm
  120  format('Result of copy ',f10.4)

       call VecAXPY(three,x,y,ierr)
       call VecNorm(y,NORM_2,norm,ierr)
       write(6,130) norm
  130  format('Result of axpy ',f10.4)

       call VecDestroy(x,ierr)
       call VecDestroy(y,ierr)
       call VecDestroy(w,ierr)
       call PetscFinalize(ierr)
       end
```

Figure 14: Sample Fortran Program: Using PETSc Vectors

```
!
!      "$Id: ex4f.F,v 1.28 2001/08/07 03:02:34 balay Exp $";
!
!  Description:  Illustrates the use of VecSetValues() to set
!  multiple values at once; demonstrates VecGetArray().
!
!/*T
!   Concepts: vectors^assembling;
!   Concepts: vectors^arrays of vectors;
!   Processors: 1
!T*/
! -----------------------------------------------------------------------


      program main
      implicit none

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!                    Include files
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!
!  The following include statements are required for Fortran programs
!  that use PETSc vectors:
!     petsc.h       - base PETSc routines
!     petscvec.h    - vectors

#include "include/finclude/petsc.h"
#include "include/finclude/petscvec.h"

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                   Macro definitions
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Macros to make clearer the process of setting values in vectors and
!  getting values from vectors.
!
!   - The element xx_a(ib) is element ib+1 in the vector x
!   - Here we add 1 to the base array index to facilitate the use of
!     conventional Fortran 1-based array indexing.
!
#define xx_a(ib)  xx_v(xx_i + (ib))
#define yy_a(ib)  yy_v(yy_i + (ib))

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!                 Beginning of program
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


       PetscScalar xwork(6)
```

```
      PetscScalar xx_v(1),yy_v(1)
      integer     i,n,ierr,loc(6)
      PetscOffset xx_i,yy_i
      Vec         x,y

      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
      n = 6

!  Create initial vector and duplicate it

      call VecCreateSeq(PETSC_COMM_SELF,n,x,ierr)
      call VecDuplicate(x,y,ierr)

!  Fill work arrays with vector entries and locations.  Note that
!  the vector indices are 0-based in PETSc (for both Fortran and
!  C vectors)

      do 10 i=1,n
         loc(i) = i-1
         xwork(i) = 10.0*i
 10   continue

!  Set vector values.  Note that we set multiple entries at once.
!  Of course, usually one would create a work array that is the
!  natural size for a particular problem (not one that is as long
!  as the full vector).

      call VecSetValues(x,6,loc,xwork,INSERT_VALUES,ierr)

!  Assemble vector

      call VecAssemblyBegin(x,ierr)
      call VecAssemblyEnd(x,ierr)

!  View vector

      write(6,20)
 20   format('initial vector:')
      call VecView(x,PETSC_VIEWER_STDOUT_SELF,ierr)
      call VecCopy(x,y,ierr)

!  Get a pointer to vector data.
!    - For default PETSc vectors, VecGetArray() returns a pointer to
!      the data array.  Otherwise, the routine is implementation dependent.
!    - You MUST call VecRestoreArray() when you no longer need access to
!      the array.
!    - Note that the Fortran interface to VecGetArray() differs from the
!      C version.  See the users manual for details.

      call VecGetArray(x,xx_v,xx_i,ierr)
      call VecGetArray(y,yy_v,yy_i,ierr)

!  Modify vector data

      do 30 i=1,n
```

```
          xx_a(i) = 100.0*i
          yy_a(i) = 1000.0*i
  30   continue

!  Restore vectors

       call VecRestoreArray(x,xx_v,xx_i,ierr)
       call VecRestoreArray(y,yy_v,yy_i,ierr)

!  View vectors

       write(6,40)
  40   format('new vector 1:')
       call VecView(x,PETSC_VIEWER_STDOUT_SELF,ierr)

       write(6,50)
  50   format('new vector 2:')
       call VecView(y,PETSC_VIEWER_STDOUT_SELF,ierr)

!  Free work space.  All PETSc objects should be destroyed when they
!  are no longer needed.

       call VecDestroy(x,ierr)
       call VecDestroy(y,ierr)
       call PetscFinalize(ierr)
       end
```

Figure 15: Sample Fortran Program: Using VecSetValues() and VecGetArray()

```
!
!    "$Id: ex5.F,v 1.29 2001/08/07 03:01:54 balay Exp $";
!
      program main
#include "include/finclude/petsc.h"
#include "include/finclude/petscdraw.h"
!
!  This example demonstrates basic use of the Fortran interface for
!  PetscDraw routines.
!
      PetscDraw         draw
      PetscDrawLG       lg
      PetscDrawAxis     axis
      integer           n,i,ierr,x,y,width,height,flg
      PetscScalar       xd,yd

      n      = 20
      x      = 0
      y      = 0
      width  = 300
      height = 300

      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
```

```
      call PetscOptionsGetInt(PETSC_NULL_CHARACTER,'-width',width,      &
     &     flg,ierr)
      call PetscOptionsGetInt(PETSC_NULL_CHARACTER,'-height',height,    &
     &     flg,ierr)
      call PetscOptionsGetInt(PETSC_NULL_CHARACTER,'-n',n,flg,ierr)

!     call PetscDrawOpenX(PETSC_COMM_SELF,PETSC_NULL_CHARACTER,         &
!                   PETSC_NULL_CHARACTER,x,y,width,height,draw,ierr)
      call PetscDrawCreate(PETSC_COMM_SELF,PETSC_NULL_CHARACTER,        &
     &               PETSC_NULL_CHARACTER,x,y,width,height,draw,ierr)
      call PetscDrawSetType(draw,PETSC_DRAW_X,ierr)

      call PetscDrawLGCreate(draw,1,lg,ierr)
      call PetscDrawLGGetAxis(lg,axis,ierr)
      call PetscDrawAxisSetColors(axis,PETSC_DRAW_BLACK,PETSC_DRAW_RED, &
     &     PETSC_DRAW_BLUE,ierr)
      call PetscDrawAxisSetLabels(axis,'toplabel','xlabel','ylabel',    &
     &     ierr)

      do 10, i=0,n-1
        xd = i - 5.0
        yd = xd*xd
        call PetscDrawLGAddPoint(lg,xd,yd,ierr)
 10   continue

      call PetscDrawLGIndicateDataPoints(lg,ierr)
      call PetscDrawLGDraw(lg,ierr)
      call PetscDrawFlush(draw,ierr)

      call PetscSleep(10,ierr)

      call PetscDrawLGDestroy(lg,ierr)
      call PetscDrawDestroy(draw,ierr)
      call PetscFinalize(ierr)
      end
```

Figure 16: Sample Fortran Program: Using PETSc PetscDraw Routines

```
!
! "$Id: ex1f.F,v 1.33 2001/08/07 03:04:16 balay Exp $";
!
!  Description: Uses the Newton method to solve a two-variable system.
!
!/*T
!  Concepts: SNES^basic uniprocessor example
!  Processors: 1
!T*/
!
! -----------------------------------------------------------------------


      program main
      implicit none
```

```
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                    Include files
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  The following include statements are generally used in SNES Fortran
!  programs:
!     petsc.h       - base PETSc routines
!     petscvec.h    - vectors
!     petscmat.h    - matrices
!     petscksp.h    - Krylov subspace methods
!     petscpc.h     - preconditioners
!     petscsles.h   - SLES interface
!     petscsnes.h   - SNES interface
!  Other include statements may be needed if using additional PETSc
!  routines in a Fortran program, e.g.,
!     petscviewer.h - viewers
!     petscis.h     - index sets
!
#include "include/finclude/petsc.h"
#include "include/finclude/petscvec.h"
#include "include/finclude/petscmat.h"
#include "include/finclude/petscksp.h"
#include "include/finclude/petscpc.h"
#include "include/finclude/petscsles.h"
#include "include/finclude/petscsnes.h"
!
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                   Variable declarations
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Variables:
!     snes        - nonlinear solver
!     sles        - linear solver
!     pc          - preconditioner context
!     ksp         - Krylov subspace method context
!     x, r        - solution, residual vectors
!     J           - Jacobian matrix
!     its         - iterations for convergence
!
      SNES     snes
      SLES     sles
      PC       pc
      KSP      ksp
      Vec      x,r
      Mat      J
      integer  ierr,its,size,rank
      PetscScalar   pfive
      double precision   tol
      PetscTruth  setls
```

```
!  Note: Any user-defined Fortran routines (such as FormJacobian)
!  MUST be declared as external.

      external FormFunction, FormJacobian, MyLineSearch

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- -
!                   Macro definitions
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- -
!
!  Macros to make clearer the process of setting values in vectors and
!  getting values from vectors.  These vectors are used in the routines
!  FormFunction() and FormJacobian().
!   - The element lx_a(ib) is element ib in the vector x
!
#define lx_a(ib) lx_v(lx_i + (ib))
#define lf_a(ib) lf_v(lf_i + (ib))
!
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!                 Beginning of program
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
      call MPI_Comm_size(PETSC_COMM_WORLD,size,ierr)
      call MPI_Comm_rank(PETSC_COMM_WORLD,rank,ierr)
      if (size .ne. 1) then
         if (rank .eq. 0) then
            write(6,*) 'This is a uniprocessor example only!'
         endif
         SETERRQ(1,' ',ierr)
      endif

! - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -
!  Create nonlinear solver context
! - - - - - - - - -- - - - - - - - - - - - - - - - - - - - - - - - - - -

      call SNESCreate(PETSC_COMM_WORLD,snes,ierr)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Create matrix and vector data structures; set corresponding routines
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


!  Create vectors for solution and nonlinear function

      call VecCreateSeq(PETSC_COMM_SELF,2,x,ierr)
      call VecDuplicate(x,r,ierr)

!  Create Jacobian matrix data structure

      call MatCreate(PETSC_COMM_SELF,PETSC_DECIDE,PETSC_DECIDE,2,2,J,   &
```

```
     &                ierr)
      call MatSetFromOptions(J,ierr)

!  Set function evaluation routine and vector

      call SNESSetFunction(snes,r,FormFunction,PETSC_NULL_OBJECT,ierr)

!  Set Jacobian matrix data structure and Jacobian evaluation routine

      call SNESSetJacobian(snes,J,J,FormJacobian,PETSC_NULL_OBJECT,     &
     &     ierr)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Customize nonlinear solver; set runtime options
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


!  Set linear solver defaults for this problem. By extracting the
!  SLES, KSP, and PC contexts from the SNES context, we can then
!  directly call any SLES, KSP, and PC routines to set various options.

      call SNESGetSLES(snes,sles,ierr)
      call SLESGetKSP(sles,ksp,ierr)
      call SLESGetPC(sles,pc,ierr)
      call PCSetType(pc,PCNONE,ierr)
      tol = 1.e-4
      call KSPSetTolerances(ksp,tol,PETSC_DEFAULT_DOUBLE_PRECISION,     &
     &                      PETSC_DEFAULT_DOUBLE_PRECISION,20,ierr)

!  Set SNES/SLES/KSP/PC runtime options, e.g.,
!      -snes_view -snes_monitor -ksp_type <ksp> -pc_type <pc>
!  These options will override those specified above as long as
!  SNESSetFromOptions() is called _after_ any other customization
!  routines.


      call SNESSetFromOptions(snes,ierr)

      call PetscOptionsHasName(PETSC_NULL_CHARACTER,'-setls',setls,ierr)

      if (setls .eq. PETSC_TRUE) then
        call SNESSetLineSearch(snes,MyLineSearch,                       &
     &                         PETSC_NULL_OBJECT,ierr)
      endif

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Evaluate initial guess; then solve nonlinear system
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


!  Note: The user should initialize the vector, x, with the initial guess
!  for the nonlinear solver prior to calling SNESSolve().  In particular,
!  to employ an initial guess of zero, the user should explicitly set
```

```
!  this vector to zero by calling VecSet().

      pfive = 0.5
      call VecSet(pfive,x,ierr)
      call SNESSolve(snes,x,its,ierr)
      if (rank .eq. 0) then
         write(6,100) its
      endif
  100 format('Number of Newton iterations = ',i5)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Free work space.  All PETSc objects should be destroyed when they
!  are no longer needed.
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


      call VecDestroy(x,ierr)
      call VecDestroy(r,ierr)
      call MatDestroy(J,ierr)
      call SNESDestroy(snes,ierr)
      call PetscFinalize(ierr)
      end
! ---------------------------------------------------------------------
!
!  FormFunction - Evaluates nonlinear function, F(x).
!
!  Input Parameters:
!  snes - the SNES context
!  x - input vector
!  dummy - optional user-defined context (not used here)
!
!  Output Parameter:
!  f - function vector
!
      subroutine FormFunction(snes,x,f,dummy,ierr)
      implicit none

#include "include/finclude/petsc.h"
#include "include/finclude/petscvec.h"
#include "include/finclude/petscsnes.h"

      SNES     snes
      Vec      x,f
      integer  ierr,dummy(*)

!  Declarations for use with local arrays

      PetscScalar  lx_v(1),lf_v(1)
      PetscOffset  lx_i,lf_i

!  Get pointers to vector data.
!    - For default PETSc vectors, VecGetArray() returns a pointer to
!      the data array.  Otherwise, the routine is implementation dependent.
!    - You MUST call VecRestoreArray() when you no longer need access to
```

```
!      the array.
!    - Note that the Fortran interface to VecGetArray() differs from the
!      C version.  See the Fortran chapter of the users manual for details.

      call VecGetArray(x,lx_v,lx_i,ierr)
      call VecGetArray(f,lf_v,lf_i,ierr)

!  Compute function

      lf_a(1) = lx_a(1)*lx_a(1)                                        &
     &          + lx_a(1)*lx_a(2) - 3.0
      lf_a(2) = lx_a(1)*lx_a(2)                                        &
     &          + lx_a(2)*lx_a(2) - 6.0

!  Restore vectors

      call VecRestoreArray(x,lx_v,lx_i,ierr)
      call VecRestoreArray(f,lf_v,lf_i,ierr)

      return
      end

! ---------------------------------------------------------------------
!
!  FormJacobian - Evaluates Jacobian matrix.
!
!  Input Parameters:
!  snes - the SNES context
!  x - input vector
!  dummy - optional user-defined context (not used here)
!
!  Output Parameters:
!  A - Jacobian matrix
!  B - optionally different preconditioning matrix
!  flag - flag indicating matrix structure
!
      subroutine FormJacobian(snes,X,jac,B,flag,dummy,ierr)
      implicit none

#include "include/finclude/petsc.h"
#include "include/finclude/petscvec.h"
#include "include/finclude/petscmat.h"
#include "include/finclude/petscpc.h"
#include "include/finclude/petscsnes.h"

      SNES         snes
      Vec          X
      Mat          jac,B
      MatStructure flag
      PetscScalar  A(4)
      integer      ierr,idx(2),dummy(*)

!  Declarations for use with local arrays

      PetscScalar lx_v(1)
```

```
      PetscOffset lx_i

!  Get pointer to vector data

      call VecGetArray(x,lx_v,lx_i,ierr)

!  Compute Jacobian entries and insert into matrix.
!   - Since this is such a small problem, we set all entries for
!     the matrix at once.
!   - Note that MatSetValues() uses 0-based row and column numbers
!     in Fortran as well as in C (as set here in the array idx).

      idx(1) = 0
      idx(2) = 1
      A(1) = 2.0*lx_a(1) + lx_a(2)
      A(2) = lx_a(1)
      A(3) = lx_a(2)
      A(4) = lx_a(1) + 2.0*lx_a(2)
      call MatSetValues(jac,2,idx,2,idx,A,INSERT_VALUES,ierr)
      flag = SAME_NONZERO_PATTERN

!  Restore vector

      call VecRestoreArray(x,lx_v,lx_i,ierr)

!  Assemble matrix

      call MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY,ierr)
      call MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY,ierr)

      return
      end


      subroutine MyLineSearch(snes,lctx,x,f,g,y,w,fnorm,ynorm,gnorm,     &
     &                        flag,ierr)
#include "include/finclude/petsc.h"
#include "include/finclude/petscvec.h"
#include "include/finclude/petscmat.h"
#include "include/finclude/petscksp.h"
#include "include/finclude/petscpc.h"
#include "include/finclude/petscsles.h"
#include "include/finclude/petscsnes.h"

      SNES             snes
      integer          lctx
      Vec              x, f,g, y, w
      double  precision fnorm,ynorm,gnorm
      integer          flag,ierr

      PetscScalar      mone

      mone = -1.0d0
      flag = 0
      call VecNorm(y,NORM_2,ynorm,ierr)
```

```
call VecAYPX(mone,x,y,ierr)
call SNESComputeFunction(snes,y,g,ierr)
call VecNorm(g,NORM_2,gnorm,ierr)
return
end
```

Figure 17: Sample Fortran Program: Using PETSc Nonlinear Solvers

Part III

Additional Information

Chapter 12

Profiling

PETSc includes a consistent, lightweight scheme to allow the profiling of application programs. The PETSc routines automatically log performance data if certain options are specified at runtime. The user can also log information about application codes for a complete picture of performance. In addition, as described in Section 12.1.1, PETSc provides a mechanism for printing informative messages about computations. Section 12.1 introduces the various profiling options in PETSc, while the remainder of the chapter focuses on details such as monitoring application codes and tips for accurate profiling.

12.1 Basic Profiling Information

If an application code and the PETSc libraries have been compiled with the `-DPETSC_USE_LOG` flag (which is the default for all versions), then various kinds of profiling of code between calls to `PetscInitialize()` and `PetscFinalize`() can be activated at runtime. Note that the flag `-DPETSC_USE_LOG` can be specified for an installation of PETSc in the file `${PETSC_DIR}/bmake/${PETSC_ARCH}/variables`, as discussed in Section 15.2. The profiling options include the following:

- `-log_summary` - Prints an ASCII version of performance data at program's conclusion. These statistics are comprehensive and concise and require little overhead; thus, `-log_summary` is intended as the primary means of monitoring the performance of PETSc codes.

- `-log_info` - Prints verbose information about code to the screen. This option provides details about algorithms, data structures, etc. Since the overhead of printing such output slows a code, this option should not be used when evaluating a program's performance.

- `-log_trace [logfile]` - Traces the beginning and ending of all PETSc events. This option, which can be used in conjunction with `-log_info`, is useful to see where a program is hanging without running in the debugger.

As discussed in Section 12.1.3, additional profilng can be done with MPE.

12.1.1 Interpreting `-log_summary` Output: The Basics

As shown in Figure 7 (in Part I), the option `-log_summary` activates printing of profile data to standard output at the conclusion of a program. Profiling data can also be printed at any time within a program by calling `PetscLogPrintSummary()`. We print performance data for each routine, organized by PETSc libraries, followed by any user-defined events (discussed in Section 12.2). For each routine, the output data include the maximum time and floating point operation (flop) rate over all processors. Information about parallel performance is also included, as discussed in the following section. For the purpose of PETSc floating point operation counting, we define one *flop* as one operation of any of the following types: multiplication, division, addition, or subtraction. For example, one VecAXPY () operation, which computes $y = \alpha x + y$ for vectors of length N, requires $2N$ flops (consisting of N additions and N multiplications). Bear in mind that flop rates present only a limited view of performance, since memory loads and stores are the real performance barrier. For simplicity, the remainder of this discussion focuses on interpreting profile data for the SLES library, which provides the linear solvers at the heart of the PETSc package. Recall the hierarchical organization of the PETSc library, as shown in Figure 1. Each SLES solver is composed of a PC (preconditioner) and a KSP (Krylov subspace) part, which are in turn built on top of the Mat (matrix) and Vec (vector) modules. Thus, operations in the SLES module are composed of lower-level operations in these packages. Note also that the nonlinear solvers library, SNES , is build on top of the SLES module, and the timestepping library, TS , is in turn built on top of SNES . We briefly discuss interpretation of the sample output in Figure 7, which was generated by solving a linear system on one processor using restarted GMRES and ILU preconditioning. The linear solvers in SLES consist of two basic phases, SLESSetUp () and SLESSolve (), each of which consists of a variety of actions, depending on the particular solution technique. For the case of using the PCILU preconditioner and KSPGMRES Krylov subspace method, the breakdown of PETSc routines is listed below. As indicated by the levels of indentation, the operations in SLESSetUp () include all of the operations within PCSetUp (), which in turn include MatILUFactor (), and so on.

- SLESSetUp - Set up linear solver
  - PCSetUp - Set up preconditioner
    - MatILUFactor - Factor preconditioning matrix
      - MatILUFactorSymbolic - Symbolic factorization phase
      - MatLUFactorNumeric - Numeric factorization phase
- SLESSolve - Solve linear system
  - PCApply - Apply preconditioner
    - MatSolve - Forward/backward triangular solves
  - KSPGMRESOrthog - Orthogonalization in GMRES
    - VecDot or VecMDot - Inner products
  - MatMult - Matrix-vector product
  - MatMultAdd - Matrix-vector product + vector addition
  - VecScale , VecNorm , VecAXPY , VecCopy , ...

The summaries printed via `-log_summary` reflect this routine hierarchy. For example, the performance summaries for a particular high-level routine such as SLESSolve include all of the operations accumulated in the lower-level components that make up the routine.

Admittedly, we do not currently present the output with `-log_summary` so that the hierarchy of PETSc operations is completely clear, primarily because we have not determined a clean and uniform way to do so throughout the library. Improvements may follow. However, for a particular problem, the user should generally have an idea of the basic operations that are required for its implementation (e.g., which operations are performed when using GMRES and ILU, as described above), so that interpreting the `-log_summary` data should be relatively straightforward.

12.1.2 Interpreting `-log_summary` Output: Parallel Performance

We next discuss performance summaries for parallel programs, as shown within Figures 18 and 19, which present the combined output generated by the `-log_summary` option. The program that generated this data is `${PETSC_DIR}/src/sles/examples/ex21.c`. The code loads a matrix and right-hand-side vector from a binary file and then solves the resulting linear system; the program then repeats this process

```
mpirun ex21 -f0 medium -f1 arco6 -ksp_gmres_unmodifiedgramschmidt -log_summary -mat_mpibaij \
-matload_block_size 3 -pc_type bjacobi -options_left
Number of iterations =  19
Residual norm = 7.7643e-05
Number of iterations =  55
Residual norm = 6.3633e-01
--------------------------------------------- PETSc Performance Summary: ---------------------------------------------
ex21 on a rs6000 named p039 with 4 processors, by mcinnes Wed Jul 24 16:30:22 1996
Max         Min        Avg        Total
Time (sec):          3.289e+01      1.0   3.288e+01
Objects:             1.130e+02      1.0   1.130e+02
Flops:               2.195e+08      1.0   2.187e+08   8.749e+08
Flops/sec:           6.673e+06      1.0               2.660e+07
MPI Messages:        2.205e+02      1.4   1.928e+02   7.710e+02
MPI Message Lengths: 7.862e+06      2.5   5.098e+06   2.039e+07
MPI Reductions:      1.850e+02      1.0
Summary of Stages:  ---- Time ------  ----- Flops -------  -- Messages -- -- Message-lengths -- Reductions -
Avg      %Total    Avg     %Total   counts   %Total    avg      %Total   counts  %Total
 0:  Load System 0: 1.191e+00   3.6% 3.980e+06   0.5%  3.80e+01    4.9%  6.102e+04   0.3%  1.80e+01   9.7%
 1:    SLESSetup 0: 6.328e-01   2.5% 1.479e+04   0.0%  0.00e+00    0.0%  0.000e+00   0.0%  0.00e+00   0.0%
 2:    SLESSolve 0: 2.269e-01   0.9% 1.340e+06   0.0%  1.52e+02   19.7%  9.405e+03   0.0%  3.90e+01  21.1%
 3:  Load System 1: 2.680e+01 107.3% 0.000e+00   0.0%  2.10e+01    2.7%  1.799e+07  88.2%  1.60e+01   8.6%
 4:    SLESSetup 1: 1.867e-01   0.7% 1.088e+08   2.3%  0.00e+00    0.0%  0.000e+00   0.0%  0.00e+00   0.0%
 5:    SLESSolve 1: 3.831e+00  15.3% 2.217e+08  97.1%  5.60e+02   72.6%  2.333e+06  11.4%  1.12e+02  60.5%

------------------------------------------------------------------------------------------------------------------------
.... [Summary of various phases, see part II below] ...
------------------------------------------------------------------------------------------------------------------------
Memory usage is given in bytes:
Object Type      Creations   Destructions   Memory  Descendants' Mem.
Viewer               5             5           0     0
Index set           10            10      127076     0
Vector              76            76     9152040     0
Vector Scatter       2             2      106220     0
Matrix               8             8     9611488     5.59773e+06
Krylov Solver        4             4       33960     7.5966e+06
Preconditioner       4             4          16     9.49114e+06
SLES                 4             4           0     1.71217e+07
```

Figure 18: Profiling a PETSc Program: Part I - Overall Summary

for a second linear system. This particular case was run on four processors of an IBM SP, using restarted GMRES and the block Jacobi preconditioner, where each block was solved with ILU. Figure 18 presents an overall performance summary, including times, floating-point operations, computational rates, and message-passing activity (such as the number and `size` of messages sent and collective operations). Summaries for various user-defined stages of monitoring (as discussed in Section 12.3) are also given. Information about the various phases of computation then follow (as shown separately here in Figure 19). Finally, a summary of memory usage and object creation and destruction is presented. We next focus on the summaries for the various phases of the computation, as given in the table within Figure 19. The summary for each phase presents the maximum times and flop rates over all processors, as well as the ratio of maximum to minimum times and flop rates for all processors. A ratio of approximately 1 indicates that computations within a given phase are well balanced among the processors; as the ratio increases, the balance becomes increasingly poor. Also, the total computational rate (in units of MFlops/sec) is given for each phase in the final column of the phase summary table.

$$\text{Total Mflop/sec} \;=\; 10^{-6} * (\text{sum of flops over all processors})/(\text{max time over all processors})$$

Note: Total computational rates <1 MFlop are listed as 0 in this column of the phase summary table. Additional statistics for each phase include the total number of messages sent, the average message length, and the number of global reductions. As discussed in the preceding section, the performance summaries for higher-level PETSc routines include the statistics for the lower levels of which they are made up. For example, the communication within matrix-vector products `MatMult` () consists of vector scatter operations, as given by the routines `VecScatterBegin` () and `VecScatterEnd()`. The final data presented are the percentages of the various statistics (time (`\%T`), flops/sec (`\%F`), messages(`\%M`), average message length (`\%L`), and reductions (`\%R`)) for each event relative to the total computation and to any user-defined stages (discussed in Section 12.3). These statistics can aid in optimizing performance, since they indicate the sections of code that could benefit from various kinds of tuning. Chapter 13 gives suggestions about achieving good performance with PETSc codes.

```
mpirun ex21 -f0 medium -f1 arco6 -ksp_gmres_unmodifiedgramschmidt -log_summary -mat_mpibaij \
-matload_block_size 3 -pc_type bjacobi -options_left
---------------------------------------------- PETSc Performance Summary: ----------------------------------------------
.... [Overall summary, see part I] ...
Phase summary info:
Count: number of times phase was executed
Time and Flops/sec: Max - maximum over all processors
Ratio - ratio of maximum to minimum over all processors
Mess: number of messages sent
Avg. len: average message length
Reduct: number of global reductions
Global: entire computation
Stage: optional user-defined stages of a computation. Set stages with PLogStagePush() and PLogStagePop().
%T - percent time in this phase         %F - percent flops in this phase
%M - percent messages in this phase     %L - percent message lengths in this phase
%R - percent reductions in this phase
Total Mflop/s: 10^6 * (sum of flops over all processors)/(max time over all processors)
------------------------------------------------------------------------------------------------------------------------
Phase                 Count      Time (sec)       Flops/sec                          --- Global ---   --- Stage ---    Total
Max     Ratio       Max    Ratio  Mess  Avg len  Reduct %T %F %M %L %R  %T %F %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
...
--- Event Stage 4: SLESSetUp 1
MatGetReordering       1  3.491e-03   1.0  0.0e+00   0.0  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   2  0  0  0  0     0
MatILUFctrSymbol       1  6.970e-03   1.2  0.0e+00   0.0  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   3  0  0  0  0     0
MatLUFactorNumer       1  1.829e-01   1.1  3.2e+07   1.1  0.0e+00 0.0e+00 0.0e+00  1  2  0  0  0  90 99  0  0  0   110
SLESSetUp              2  1.989e-01   1.1  2.9e+07   1.1  0.0e+00 0.0e+00 0.0e+00  1  2  0  0  0  99 99  0  0  0   102
PCSetUp                2  1.952e-01   1.1  2.9e+07   1.1  0.0e+00 0.0e+00 0.0e+00  1  2  0  0  0  97 99  0  0  0   104
PCSetUpOnBlocks        1  1.930e-01   1.1  3.0e+07   1.1  0.0e+00 0.0e+00 0.0e+00  1  2  0  0  0  96 99  0  0  0   105
--- Event Stage 5: SLESSolve 1
MatMult               56  1.199e+00   1.1  5.3e+07   1.0  1.1e+03 4.2e+03 0.0e+00  5 28 99 23  0  30 28 99 99  0   201
MatSolve              57  1.263e+00   1.0  4.7e+07   1.0  0.0e+00 0.0e+00 0.0e+00  5 27  0  0  0  33 28  0  0  0   187
VecNorm               57  1.528e-01   1.3  2.7e+07   1.3  0.0e+00 0.0e+00 2.3e+02  1  1  0  0 31   3  1  0  0 51    81
VecScale              57  3.347e-02   1.0  4.7e+07   1.0  0.0e+00 0.0e+00 0.0e+00  0  1  0  0  0   1  1  0  0  0   184
VecCopy                2  1.703e-03   1.1  0.0e+00   0.0  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecSet                 3  2.098e-03   1.0  0.0e+00   0.0  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecAXPY                3  3.247e-03   1.1  5.4e+07   1.1  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0   200
VecMDot               55  5.216e-01   1.2  9.8e+07   1.2  0.0e+00 0.0e+00 2.2e+02  2 20  0  0 30  12 20  0  0 49   327
VecMAXPY              57  6.997e-01   1.1  6.9e+07   1.1  0.0e+00 0.0e+00 0.0e+00  3 21  0  0  0  18 21  0  0  0   261
VecScatterBegin       56  4.534e-02   1.8  0.0e+00   0.0  1.1e+03 4.2e+03 0.0e+00  0  0 99 23  0   1  0 99 99  0     0
VecScatterEnd         56  2.095e-01   1.2  0.0e+00   0.0  0.0e+00 0.0e+00 0.0e+00  1  0  0  0  0   5  0  0  0  0     0
SLESSolve              1  3.832e+00   1.0  5.6e+07   1.0  1.1e+03 4.2e+03 4.5e+02 15 97 99 23 61  99 99 99 99 99   222
KSPGMRESOrthog        55  1.177e+00   1.1  7.9e+07   1.1  0.0e+00 0.0e+00 2.2e+02  4 39  0  0 30  29 40  0  0 49   290
PCSetUpOnBlocks        1  1.180e-05   1.1  0.0e+00   0.0  0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
PCApply               57  1.267e+00   1.0  4.7e+07   1.0  0.0e+00 0.0e+00 0.0e+00  5 27  0  0  0  33 28  0  0  0   186
------------------------------------------------------------------------------------------------------------------------
.... [Conclusion of overall summary, see part I] ...
```

Figure 19: Profiling a PETSc Program: Part II - Phase Summaries

12.1.3 Using `-log_mpe` with Upshot/Jumpshot

It is also possible to use the *Upshot* (or *Jumpshot*) package [11] to visualize PETSc events. This package comes with the MPE software, which is part of the MPICH [9] implementation of MPI. The option

-log_mpe [logfile]

creates a logfile of events appropriate for viewing with *Upshot*. The user can either use the default logging file, `mpe.log`, or specify an optional name via `logfile`.

To use this logging option, the user may employ any implementation of MPI (not necessarily MPICH), but must build and link the MPE part of the MPICH. The user must compile the PETSc library with the `-DPETSC_HAVE_MPE` flag, which is *not* activated by default. The user can turn on MPE logging by specifying `-DPETSC_HAVE_MPE` in the `PCONF` variable within `${PETSC_DIR}/bmake/${PETSC_ARCH}/packages` and (re)compiling all of PETSc. By default, not all PETSc events are logged with MPE. For example, since `MatSetValues` () may be called thousands of times in a program, by default its calls are not logged with MPE. To activate MPE logging of a particular event, one should use the command

PetscLogEventMPEActivate(int event);

To deactivate logging of an event for MPE, one should use

PetscLogEventMPEDeactivate(int event);

The `event` may be either a predefined PETSc event (as listed in the file `${PETSC_DIR}/include/petsclog.h`) or one obtained with `PetscLogEventRegister` () (as described in Section 12.2).

These routines may be called as many times as desired in an application program, so that one could restrict MPE event logging only to certain code segments. To see what events are logged by default, the user can view the source code; see the files `src/plot/src/plogmpe.c` and `include/petsclog.h`. A simple program and GUI interface to see the events that are predefined and their definition is being developed. The user can also log MPI events. To do this, simply consider the PETSc application as any MPI application, and follow the MPI implementation's instructions for logging MPI calls. For example, when using MPICH, this merely required adding `-llmpi` to the library list *before* `-lmpi`.

12.2 Profiling Application Codes

PETSc automatically logs object creation, times, and floating-point counts for the library routines. Users can easily supplement this information by monitoring their application codes as well. The basic steps involved in logging a user-defined portion of code, called an *event*, are shown in the code fragment below:

```
#include "petsclog.h"
int USER_EVENT;
PetscLogEventRegister(&USER_EVENT,"User event name",0);
PetscLogEventBegin(USER_EVENT,0,0,0,0);
/* application code segment to monitor */
PetscLogFlops(number of flops for this code segment);
PetscLogEventEnd(USER_EVENT,0,0,0,0);
```

One must register the event by calling `PetscLogEventRegister()`, which assigns a unique integer to identify the event for profiling purposes:

```
PetscLogEventRegister(int *e,const char string[]);
```

Here `string` is a user-defined event name, and `color` is an optional user-defined event color (for use with *Upshot/Nupshot* logging); one should see the manual page for details. The argument returned in `e` should then be passed to the `PetscLogEventBegin`() and `PetscLogEventEnd`() routines. Events are logged by using the pair

```
PetscLogEventBegin(int event,PetscObject o1,PetscObject o2,
PetscObject o3,PetscObject o4);
PetscLogEventEnd(int event,PetscObject o1,PetscObject o2,
PetscObject o3,PetscObject o4);
```

The four objects are the PETSc objects that are most closely associated with the event. For instance, in a matrix-vector product they would be the matrix and the two vectors. These objects can be omitted by specifying 0 for `o1` - `o4`. The code between these two routine calls will be automatically timed and logged as part of the specified event. The user can log the number of floating-point operations for this segment of code by calling

```
PetscLogFlops(number of flops for this code segment);
```

between the calls to `PetscLogEventBegin`() and `PetscLogEventEnd`(). This value will automatically be added to the global flop counter for the entire program.

12.3 Profiling Multiple Sections of Code

By default, the profiling produces a single set of statistics for all code between the `PetscInitialize` () and `PetscFinalize` () calls within a program. One can independently monitor up to ten stages of code by switching among the various stages with the comands

`PetscLogStagePush` (int stage);
`PetscLogStagePop` ();

where `stage` is an integer (0-9); see the manual pages for details. The command

`PetscLogStageRegister` (int stage,char *name)

allows one to associate a name with a stage; these names are printed whenever summaries are generated with `-log_summary` or `PetscLogPrintSummary` (). The following code fragment uses three profiling stages within an program.

`PetscInitialize` (int *argc,char ***args,0,0);
/* stage 0 of code here */
`PetscLogStageRegister` (0,"Stage 0 of Code");
for (i=0; i¡ntimes; i++) {
PetscLogStagePush(1);
PetscLogStageRegister(1,"Stage 1 of Code");
/* stage 1 of code here */
PetscLogStagePop();
PetscLogStagePush(2);
PetscLogStageRegister(1,"Stage 2 of Code");
/* stage 2 of code here */
PetscLogStagePop();
} `PetscFinalize` ();

Figures 18 and 19 show output generated by `-log_summary` for a program that employs several profiling stages. In particular, this program is subdivided into six stages: loading a matrix and right-hand-side vector from a binary file, setting up the preconditioner, and solving the linear system; this sequence is then repeated for a second linear system. For simplicity, Figure 19 contains output only for stages 4 and 5 (linear solve of the second system), which comprise the part of this computation of most interest to us in terms of performance monitoring. This code organization (solving a small linear system followed by a larger system) enables generation of more accurate profiling statistics for the second system by overcoming the often considerable overhead of paging, as discussed in Section 12.8.

12.4 Restricting Event Logging

By default, all PETSc operations are logged. To enable or disable the PETSc logging of individual events, one uses the commands

`PetscLogEventActivate` (int event);
`PetscLogEventDeactivate` (int event);

The `event` may be either a predefined PETSc event (as listed in the file `${PETSC_DIR}/include/petsclog.h`) or one obtained with `PetscLogEventRegister` () (as described in Section 12.2). PETSc also provides routines that deactivate (or activate) logging for entire components of the library. Currently, the components that support such logging (de)activation are `Mat` (matrices), `Vec` (vectors), `SLES` (linear solvers, including `KSP` and `PC`), and `SNES` (nonlinear solvers):

```
PetscLogEventDeactivateClass(MAT_COOKIE);
PetscLogEventDeactivateClass(SLES_COOKIE); /* includes PC and KSP */
PetscLogEventDeactivateClass(VEC_COOKIE);
PetscLogEventDeactivateClass(SNES_COOKIE);
```

and

```
PetscLogEventActivateClass(MAT_COOKIE);
PetscLogEventActivateClass(SLES_COOKIE); /* includes PC and KSP */
PetscLogEventActivateClass(VEC_COOKIE);
PetscLogEventActivateClass(SNES_COOKIE);
```

Recall that the option `-log_all` produces extensive profile data, which can be a challenge for PETScView to handle due to the memory limitations of Tcl/Tk. Thus, one should generally use `-log_all` when running programs with a relatively small number of events or when disabling some of the events that occur many times in a code (e.g., `VecSetValues()`, `MatSetValues()`). Section 12.1.3 gives information on the restriction of events in MPE logging.

12.5 Interpreting `-log_info` Output: Informative Messages

Users can activate the printing of verbose information about algorithms, data structures, etc. to the screen by using the option `-log_info` or by calling `PetscLogInfoAllow(PETSC_TRUE)`. Such logging, which is used throughout the PETSc libraries, can aid the user in understanding algorithms and tuning program performance. For example, as discussed in Section 3.1.1, `-log_info` activates the printing of information about memory allocation during matrix assembly. Application programmers can employ this logging as well, by using the routine

```
PetscLogInfo(void* obj,char *message,...)
```

where `obj` is the PETSc object associated most closely with the logging statement, `message`. For example, in the line search Newton methods, we use a statement such as

```
PetscLogInfo(snes,"Cubically determined step, lambda '%'g\n",lambda);
```

One can selectively turn off informative messages about any of the basic PETSc objects (e.g., `Mat`, `SNES`) with the command

```
PetscLogInfoDeactivateClass(int object_cookie)
```

where `object_cookie` is one of `MAT_COOKIE`, `SNES_COOKIE`, etc. Messages can be reactivated with the command

```
PetscLogInfoActivateClass(int object_cookie)
```

Such deactivation can be useful when one wishes to view information about higher-level PETSc libraries (e.g., `TS` and `SNES`) without seeing all lower level data as well (e.g., `Mat`). One can deactivate events at runtime for matrix and linear solver libraries via `-log_info [no_mat, no_sles]`.

12.6 Time

PETSc application programmers can access the wall clock time directly with the command

```
PetscLogDouble time;
PetscGetTime(&time);CHKERRQ(ierr);
```

In addition, as discussed in Section 12.2, PETSc can automatically profile user-defined segments of code.

12.7 Saving Output to a File

All output from PETSc programs (including informative messages, profiling information, and convergence data) can be saved to a file by using the command line option `-log_history [filename]`. If no file name is specified, the output is stored in the file `$HOME/.petschistory`. Note that this option only saves output printed with the `PetscPrintf` () and `PetscFPrintf` () commands, not the standard `printf()` and `fprintf()` statements.

12.8 Accurate Profiling: Overcoming the Overhead of Paging

One factor that often plays a significant role in profiling a code is paging by the operating system. Generally, when running a program only a few pages required to start it are loaded into memory rather than the entire executable. When the execution procedes to code segments that are not in memory, a pagefault occurs, prompting the required pages to be loaded from the disk (a very slow process). This activity distorts the results significantly. (The paging effects are noticeable in the the log files generated by `-log_mpe`, which is described in Section 12.1.3.) To eliminate the effects of paging when profiling the performance of a program, we have found an effective procedure is to run the **exact same code** on a small dummy problem before running it on the actual problem of interest. We thus ensure that all code required by a solver is loaded into memory during solution of the small problem. When the code procedes to the actual (larger) problem of interest, all required pages have already been loaded into main memory, so that the performance numbers are not distorted. When this procedure is used in conjunction with the user-defined stages of profiling described in Section 12.3, we can focus easily on the problem of interest. For example, we used this technique in the program `${PETSC_DIR}/src/sles/examples/tutorials/ex10.c` to generate the timings within Figures 18 and 19. In this case, the profiled code of interest (solving the linear system for the larger problem) occurs within event stages 4 and 5. Section 12.1.2 provides details about interpreting such profiling data. In particular, the macros

`PreLoadBegin` (`PetscTruth` ,char* stagename),
`PreLoadStage` (char *stagename),

and

`PreLoadEnd` ()

can be used to easily convert a regular PETSc program to one that uses preloading. The command line options `-preload true` and `-preload false` may be used to turn on and off preloading at run time for PETSc programs that use these macros.

Chapter 13

Hints for Performance Tuning

This chapter presents some tips on achieving good performance within PETSc codes. We urge users to read these hints before evaluating the performance of PETSc application codes.

13.1 Compiler Options

Code compiled with the `BOPT=O` option generally runs two to three times faster than that compiled with `BOPT=g`, so we recommend using one of the optimized versions of code (`BOPT=O`, `BOPT=O_c++`, or `BOPT=O_complex`) when evaluating performance. The user can specify alternative compiler options instead of the defaults set in the PETSc distribution. One can set the compiler options for a particular architecture (`PETSC_ARCH`) and `BOPT` by editing the file `${PETSC_DIR}/bmake/${PETSC_ARCH}/variables`. Section 15.1.2 gives details.

13.2 Profiling

Users should not spend time optimizing a code until after having determined where it spends the bulk of its time on realistically sized problems. As discussed in detail in Chapter 12, the PETSc routines automatically log performance data if certain runtime options are specified. We briefly highlight usage of these features below.

- Run the code with the option `-log_summary` to print a performance summary for various phases of the code.
- Run the code with the option `-log_mpe [logfilename]`, which creates a logfile of events suitable for viewing with Upshot or Nupshot (part of MPICH).

13.3 Aggregation

Performing operations on chunks of data rather than a single element at a time can significantly enhance performance.

- Insert several (many) elements of a matrix or vector at once, rather than looping and inserting a single value at a time. In order to access elements in of vector repeatedly, employ VecGetArray() to allow direct manipulation of the vector elements.
- When using `MatSetValues` (), if the column indices of the values being inserted have been sorted in monotonically increasing order, call the routine `MatSetOption(mat,MAT_COLUMNS_SORTED)` before setting the values to reduce the insertion time significantly.

- When possible, use VecMDot () rather than a series of calls to VecDot ().

13.4 Efficient Memory Allocation

13.4.1 Sparse Matrix Assembly

Since the process of dynamic memory allocation for sparse matrices is inherently very expensive, accurate preallocation of memory is crucial for efficient sparse matrix assembly. One should use the matrix creation routines for particular data structures, such as MatCreateSeqAIJ() and MatCreateMPIAIJ () for compressed, sparse row formats, instead of the generic MatCreate () routine. For problems with multiple degrees of freedom per node, the block, compressed, sparse row formats, created by MatCreateSeqBAIJ () and MatCreateMPIBAIJ (), can significantly enhance performance. Section 3.1.1 includes extensive details and examples regarding preallocation.

13.4.2 Sparse Matrix Factorization

When symbolically factoring an AIJ matrix, PETSc has to guess how much fill there will be. Careful use of the fill parameter in the MatILUInfo structure when calling MatLUFactorSymbolic () or MatILUFactorSymbolic () can reduce greatly the number of mallocs and copies required, and thus greatly improve the performance of the factorization. One way to determine a good value for f is to run a program with the option -log_info. The symbolic factorization phase will then print information such as

Info:MatILUFactorSymbolic _AIJ:Realloc 12 Fill ratio:given 1 needed 2.16423

This indicates that the user should have used a fill estimate factor of about 2.17 (instead of 1) to prevent the 12 required mallocs and copies. The command line option

-pc_ilu_fill 2.17

will cause PETSc to preallocate the correct amount of space for incomplete (ILU) factorization. The corresponding option for direct (LU) factorization is -pc_lu_fill <fill_amount\trl{>}.

13.4.3 PetscMalloc() Calls

Users should employ a reasonable number of PetscMalloc () calls in their codes. Hundreds or thousands of memory allocations may be appropriate; however, if tens of thousands are being used, then reducing the number of PetscMalloc () calls may be warranted. For example, reusing space or allocating large chunks and dividing it into pieces can produce a significant savings in allocation overhead. Section 13.5 gives details.

13.5 Data Structure Reuse

Data structures should be reused whenever possible. For example, if a code often creates new matrices or vectors, there often may be a way to reuse some of them. Very significant performance improvements can be achieved by reusing matrix data structures with the same nonzero pattern. If a code creates thousands of matrix or vector objects, performance will be degraded. For example, when solving a nonlinear problem or timestepping, reusing the matrices and their nonzero structure for many steps when appropriate can make the code run significantly faster.

A simple technique for saving work vectors, matrices, etc. is employing a user-defined context. In C and C++ such a context is merely a structure in which various objects can be stashed; in Fortran a user context can be an integer array that contains both parameters and pointers to PETSc objects. See `${PETSC_DIR}/snes/examples/tutorials/ex5.c` and `${PETSC_DIR}/snes/examples/tutorials/ex5f.F` for examples of user-defined application contexts in C and Fortran, respectively.

13.6 Numerical Experiments

PETSc users should run a variety of tests. For example, there are a large number of options for the linear and nonlinear equation solvers in PETSc, and different choices can make a *very* big difference in convergence rates and execution times. PETSc employs defaults that are generally reasonable for a wide range of problems, but clearly these defaults cannot be best for all cases. Users should experiment with many combinations to determine what is best for a given problem and customize the solvers accordingly.

- Use the options `-snes_view`, `-sles_view`, etc. (or the routines `SLESView` (), `SNESView` (), etc.) to view the options that have been used for a particular solver.
- Run the code with the option `-help` for a list of the available runtime commands.
- Use the option `-log_info` to print details about the solvers' operation.
- Use the PETSc monitoring discussed in Chapter 12 to evaluate the performance of various numerical methods.

13.7 Tips for Efficient Use of Linear Solvers

As discussed in Chapter 4, the default linear solvers are

- uniprocessor: GMRES(30) with ILU(0) preconditioning
- multiprocessor: GMRES(30) with block Jacobi preconditioning, where there is 1 block per processor, and each block is solved with ILU(0)

One should experiment to determine alternatives that may be better for various applications. Recall that one can specify the `KSP` methods and preconditioners at runtime via the options:

-ksp_type <ksp_name> -pc_type <pc_name>

One can also specify a variety of runtime customizations for the solvers, as discussed throughout the manual. In particular, note that the default restart parameter for GMRES is 30, which may be too small for some large-scale problems. One can alter this parameter with the option `-ksp_gmres_restart` <restart> or by calling `KSPGMRESSetRestart` (). Section 4.3 gives information on setting alternative GMRES orthogonalization routines, which may provide much better parallel performance.

13.8 Detecting Memory Allocation Problems

PETSc provides a number of tools to aid in detection of problems with memory allocation, including leaks and use of uninitialized space. We briefly describe these below.

- The PETSc memory allocation (which collects statistics and performs error checking), is employed by default for codes compiled in a debug mode (`BOPT=g`, `BOPT=g_c++`, `BOPT=g_complex`). PETSc memory allocation can be activated for other other cases, such as `BOPT=O`, with the option `-trmalloc`, while `-trmalloc_off` forces the use of conventional memory allocation for the `BOPT=g`, `BOPT=g_c++`, and `BOPT=g_complex` versions. When running timing tests, one should use the `BOPT=O` version of the libraries.

- When the PETSc memory allocation routines are used, the option `-trdump` will print a list of unfreed memory at the conclusion of a program. If all memory has been freed, only a message stating the maximum allocated space will be printed. However, if some memory remains unfreed, this information will be printed. Note that the option `-trdump` merely activates a call to `PetscTrDump` () during `PetscFinalize` (); the user can also call `PetscTrDump` () elsewhere in a program.

- Another useful option for use with PETSc memory allocation routines is `-trmalloc_log`, which activates logging of all calls to malloc and reports memory usage, including all Fortran arrays. This option provides a more complete picture than `-trdump` for codes that employ Fortran with hardwired arrays. Note that the option `-trmalloc_log` activates calls to `PetscTrLog` (), `PetscTrLogDump()`, and `PetscGetResidentSetSize` () during `PetscFinalize()`; the user can also call these routines elsewhere in a program.

- The option `-trmalloc_nan` is useful for tracking down the allocated memory that is used before it has been initialized. This option calls PetscInitializeNans() which marks an array as being uninitialized, so that if values are used for computation without first having been set, a floating point exception is generated. This option also calls PetscInitializeLargeInts(); see the manual pages for details. Note that so far these work only on the certain systems.

13.9 System-Related Problems

The performance of a code can be affected by a variety of factors, including the cache behavior, other users on the machine, etc. Below we briefly describe some common problems and possibilities for overcoming them.

- **Problem too large for physical memory size**: When timing a program, one should always leave at least a ten percent margin between the total memory a process is using and the physical `size` of the machine's memory. One way to estimate the amount of memory used by given process is with the UNIX `getrusage` system routine. Also, the PETSc option `-log_summary` prints the amount of memory used by the basic PETSc objects, thus providing a lower bound on the memory used. Another useful option is `-trmalloc_log` which reports all memory, including any Fortran arrays in an application code.

- **Effects of other users**: If other users are running jobs on the same physical processor nodes on which a program is being profiled, the timing results are essentially meaningless.

- **Overhead of timing routines on certain machines**: On certain machines, even calling the system clock in order to time routines is slow; this skews all of the flop rates and timing results. The file `${PETSC_DIR}/src/benchmarks/PetscTime.c` contains a simple test problem that will approximate the ammount of time required to get the current time in a running program. On good systems it will on the order of 1.e-6 seconds or less.

- **Problem too large for good cache performance**: Certain machines with lower memory bandwidths (slow memory access) attempt to compensate by having a very large cache. Thus, if a significant portion of an application fits within the cache, the program will achieve very good performance; if the code is too large, the performance can degrade markedly. To analyze whether this situation affects a particular code, one can try plotting the total flop rate as a function of problem `size` . If the flop rate decreases rapidly at some point, then the problem may likely be too large for the cache `size` .

- **Inconsistent timings**: Inconsistent timings are likely due to other users on the machine, thrashing (using more virtual memory than available physical memory), or paging in of the initial executable. Section 12.8 provides information on overcoming paging overhead when profiling a code. We have found on all systems that if you follow all the advise above your timings will be consistent within a variation of less than five percent.

Chapter 14

Other PETSc Features

14.1 Runtime Options

Allowing the user to modify parameters and options easily at runtime is very desirable for many applications. PETSc provides a simple mechanism to enable such customization. To print a list of available options for a given program, simply specify the option `-help` (or `-h`) at runtime, e.g.,

`mpirun` -np 1 ex1 -help

Note that all runtime options correspond to particular PETSc routines that can be explicitly called from within a program to set compile-time defaults. For many applications it is natural to use a combination of compile-time and runtime choices. For example, when solving a linear system, one could explicitly specify use of the Krylov subspace technique BiCGStab by calling

`KSPSetType` (ksp,KSPBCGS);

One could then override this choice at runtime with the option

-ksp_type tfqmr

to select the Transpose-Free QMR algorithm. (See Chapter 4 for details.) The remainder of this section discusses details of runtime options.

14.1.1 The Options Database

Each PETSc process maintains a database of option names and values (stored as text strings). This database is generated with the command PETScInitialize(), which is listed below in its C/C++ and Fortran variants, respectively:

`PetscInitialize` (int *argc,char ***args,const char *file,const char *help);
call `PetscInitialize` (character file,integer ierr)

The arguments `argc` and `args` (in the C/C++ version only) are the addresses of usual command line arguments, while the `file` is a name of a file that can contain additional options. By default this file is called `.petscrc` in the user's home directory. The user can also specify options via the environmental variable `PETSC_OPTIONS`. The options are processed in the following order:

- file
- environmental variable

- command line

Thus, the command line options supersede the environmental variable options, which in turn supersede the options file.

The file format for specifying options is

-optionname possible_value
-anotheroptionname possible_value
...

All of the option names must begin with a dash (-) and have no intervening spaces. Note that the option values cannot have intervening spaces either, and tab characters cannot be used between the option names and values. The user can employ any naming convention. For uniformity throughout PETSc, we employ the format `-package_option` (for instance, `-ksp_type` and `-mat_view_info`). Users can specify an alias for any option name (to avoid typing the sometimes lengthy default name) by adding an alias to the `.petscrc` file in the format

alias -newname -oldname

For example,

alias -kspt -ksp_type
alias -sd -start_in_debugger

Comments can be placed in the .petscrc file by using one of the following symbols in the first column of a line: `#`, `%`, or `!`.

14.1.2 User-Defined PetscOptions

Any subroutine in a PETSc program can add entries to the database with the command

`PetscOptionsSetValue` (char *name,char *value);

though this is rarely done. To locate options in the database, one should use the commands

`PetscOptionsHasName` (char *pre,char *name,`PetscTruth` *flg);
`PetscOptionsGetInt` (char *pre,char *name,int *value,`PetscTruth` *flg);
`PetscOptionsGetReal` (char *pre,char *name,double *value,`PetscTruth` *flg);
`PetscOptionsGetString` (char *pre,char *name,char *value,int maxlen,`PetscTruth` *flg);
`PetscOptionsGetStringArray` (char *pre,char *name,char **values,int *maxlen,`PetscTruth` *flg);
`PetscOptionsGetIntArray` (char *pre,char *name,int *value,int *nmax,`PetscTruth` *flg);
`PetscOptionsGetRealArray` (char *pre,char *name,double *value, int *nmax,`PetscTruth` *flg);

All of these routines set `flg=PETSC_TRUE` if the corresponding option was found, `flg=PETSC_FALSE` if it was not found. The optional argument `pre` indicates that the true name of the option is the given name (with the dash "-" removed) prepended by the prefix `pre`. Usually `pre` should be set to `PETSC_NULL` (or `PETSC_NULL_CHARACTER` for Fortran); its purpose is to allow someone to rename all the options in a package without knowing the names of the individual options. For example, when using block Jacobi preconditioning, the `KSP` and `PC` methods used on the individual blocks can be controlled via the options `-sub_ksp_type` and `-sub_pc_type`.

14.1.3 Keeping Track of Options

One useful means of keeping track of user-specified runtime options is use of `-optionstable`, which prints to `stdout` during `PetscFinalize()` a table of all runtime options that the user has specified. A related option is `-options_left`, which prints the options table and indicates any options that have *not* been requested upon a call to `PetscFinalize()`. This feature is useful to check whether an option has been activated for a particular PETSc object (such as a solver or matrix format), or whether an option name may have been accidentally misspelled.

14.2 Viewers: Looking at PETSc Objects

PETSc employs a consistent scheme for examining, printing, and saving objects through commands of the form

XXXView(XXX obj,`PetscViewer` viewer);

Here `obj` is any PETSc object of type `XXX`, where `XXX` is `Mat` , `Vec` , `SNES` , etc. There are several predefined viewers:

- Passing in a zero for the viewer causes the object to be printed to the screen; this is most useful when viewing an object in a debugger.
- `PETSC_VIEWER_STDOUT_SELF` and `PESC_VIEWER_STDOUT_WORLD` cause the object to be printed to the screen.
- `PETSC_VIEWER_DRAW_SELF` and `PETSC_VIEWER_DRAW_WORLD` causes the object to be drawn in a default X window.
- Passing in a viewer obtained by PetscViewerDrawOpenX() causes the object to be displayed graphically.
- To save an object to a file in ASCII format, the user creates the viewer object with the command `PetscViewerASCIIOpen(MPI_Comm comm, char* file, PetscViewer *viewer)`. This object is analogous to `PETSC_VIEWER_STDOUT_SELF` (for a communicator of `MPI_COMM_SELF`) and `PETSC_VIEWER_STDOUT_WORLD` (for a parallel communicator).
- To save an object to a file in binary format, the user creates the viewer object with the command `PetscViewerBinaryOpen(MPI_Comm comm, char* file, PetscViewerBinaryType type, PetscViewer *viewer)`. Details of binary I/O are discussed below.
- Vector and matrix objects can be passed to a running Matlab process with a viewer created by

  PetscViewerMatlabOpen(MPI_Comm `comm` ,char *machine,int port,`PetscViewer` *viewer).

  On the Matlab side, one must first run `v = openport(int port)` and then `A = receive(v)` to obtain the matrix or vector. Once all objects have been received, the port can be closed from the Matlab end with `closeport(v)`. On the PETSc side, one should destroy the viewer object with `PetscViewerDestroy()`. The corresponding Matlab `mex` files are located in `${PETSC_DIR}/src/viewer/impls/matlab`.

The user can control the format of ASCII printed objects with viewers created by `PetscViewerASCIIOpen` () by calling

PetscViewerSetFormat(PetscViewer viewer,int format);

Possible formats include PETSC_VIEWER_ASCII_DEFAULT, PETSC_VIEWER_ASCII_MATLAB, and

PETSC_VIEWER_ASCII_IMPL. The implementation-specific format, PETSC_VIEWER_ASCII_IMPL, displays the object in the most natural way for a particular implementation. For example, when viewing a block diagonal matrix that has been created with MatCreateSeqBDiag(), PETSC_VIEWER_ASCII_IMPL prints by diagonals, while PETSC_VIEWER_ASCII_DEFAULT uses the conventional row-oriented format. The routines

PetscViewerPushFormat(PetscViewer viewer,int format);
PetscViewerPopFormat(PetscViewer viewer);

allow one to temporarily change the format of a viewer. As discussed above, one can output PETSc objects in binary format by first opening a binary viewer with PetscViewerBinaryOpen() and then using MatView(), VecView(), etc. The corresponding routines for input of a binary object have the form XXXLoad(). In particular, matrix and vector binary input is handled by the following routines:

MatLoad(PetscViewer viewer,MatType outtype,Mat *newmat);
VecLoad(PetscViewer viewer,Vec *newvec);

These routines generate parallel matrices and vectors if the viewer's communicator has more than one processor. The particular matrix and vector formats are determined from the options database; see the manual pages for details. One can provide additional information about matrix data for matrices stored on disk by providing an optional file matrixfilename.info, where matrixfilename is the name of the file containing the matrix. The format of the optional file is the same as the .petscrc file and can (currently) contain the following:

-matload_block_size <bs>
-matload_bdiag_diags <s1,s2,s3,...>

The block size indicates the size of blocks to use if the matrix is read into a block oriented data structure (for example, MATSEQBDIAG or MATMPIBAIJ). The diagonal information s1,s2,s3,... indicates which (block) diagonals in the matrix have nonzero values.

14.3 Debugging

PETSc programs may be debugged using one of the two options below.

- -start_in_debugger [noxterm,dbx,xxgdb] [-display name] - start all processes in debugger
- -on_error_attach_debugger [noxterm,dbx,xxgdb] [-display name] - start debugger only on encountering an error

Note that, in general, debugging MPI programs cannot be done in the usual manner of starting the programming in the debugger (because then it cannot set up the MPI communication and remote processes). By default the GNU debugger *gdb* is used when -start_in_debugger or -on_error_attach_debugger is specified. To employ either *xxgdb* or the common UNIX debugger *dbx*, one uses command line options as indicated above. On HP-UX machines the debugger *xdb* should be used instead of *dbx*; on RS/6000 machines the *xldb* debugger is supported as well. By default, the debugger will be started in a new xterm (to enable running separate debuggers on each process), unless the option noxterm is used. In order to handle the MPI startup phase, the debugger command “cont” should be used to continue execution of the program within the debugger. Rerunning the program through the debugger requires terminating the first job and restarting the processor(s); the usual “run” option in the debugger will not correctly handle the MPI startup and should not be used. Not all debuggers work on all machines, so the user may have to experiment to find one that works correctly. You can select a subset of the processors to be debugged (the rest just run without the debugger) with the option

-debugger_nodes node1,node2,...

where you simply list the nodes you want the debugger to run with.

14.4 Error Handling

Errors are handled through the routine `PetscError()`. This routine checks a stack of error handlers and calls the one on the top. If the stack is empty, it selects `PetscTraceBackErrorHandler` (), which tries to print a traceback. A new error handler can be put on the stack with

`PetscPushErrorHandler` (int (*HandlerFunction)(int line,char *dir,char *file,
char *message,int number,void*),void *HandlerContext)

The arguments to `HandlerFunction()` are the line number where the error occurred, the file in which the error was detected, the corresponding directory, the error message, the error integer, and the `HandlerContext.` The routine

`PetscPopErrorHandler` ()

removes the last error handler and discards it. PETSc provides two additional error handlers besides `PetscTraceBackErrorHandler` ():

`PetscAbortErrorHandler` ()
PetscAttachErrorHandler()

The function `PetscAbortErrorHandler` () calls abort on encountering an error, while PetscAttachErrorHandler() attaches a debugger to the running process if an error is detected. At runtime, these error handlers can be set with the options `-on_error_abort` or `-on_error_attach_debugger [noxterm, dbx, xxgdb, xldb] [-display DISPLAY]`. All PETSc calls can be traced (useful for determining where a program is hanging without running in the debugger) with the option

-log_trace [filename]

where `filename` is optional. By default the traces are printed to the screen. This can also be set with the command `PetscLogTraceBegin(FILE*)`. It is also possible to trap signals by using the command

`PetscPushSignalHandler` (int (*Handler)(int,void *),void *ctx);

The default handler `PetscDefaultSignalHandler` () calls `PetscError()` and then terminates. In general, a signal in PETSc indicates a catastrophic failure. Any error hander that the user provides should try to clean up only before exiting. By default all PETSc programs use the default signal handler, although the user can turn this off at runtime with the option `-no_signal_handler` . There is a separate signal handler for floating-point exceptions. The option `-fp_trap` turns on the floating-point trap at runtime, and the routine

`PetscSetFPTrap` (int flag);

can be used in-line. A `flag` of `PETSC_FP_TRAP_ON` indicates that floating-point exceptions should be trapped, while a value of `PETSC_FP_TRAP_OFF` (the default) indicates that they should be ignored. Note that on certain machines, in particular the IBM RS/6000, trapping is very expensive. A small set of macros is used to make the error handling lightweight. These macros are used throughout the PETSc libraries and can be employed by the application programmer as well. When an error is first detected, one should set it by calling

`SETERRQ` (int flag,int pflag,char *message);

The user should check the return codes for all PETSc routines (and possibly user-defined routines as well) with

ierr = PetscRoutine(...);`CHKERRQ` (int ierr);

Likewise, all memory allocations should be checked with

ierr = `PetscMalloc` (n*sizeof(double),&ptr);`CHKERRQ` (ierr);

If this procedure is followed throughout all of the user's libraries and codes, any error will by default generate a clean traceback of the location of the error.

Note that the macro `__FUNCT__` is used to keep track of routine names during error tracebacks. Users need not worry about this macro in their application codes; however, users can take advantage of this feature if desired by setting this macro before each user-defined routine that may call `SETERRQ()`, `CHKERRQ()`. A simple example of usage is given below.

```
#undef __FUNCT__
#define __FUNCT__ "MyRoutine1"
int MyRoutine1() {
/* code here */
return 0;
}
```

14.5 Incremental Debugging

When developing large codes, one is often in the position of having a correctly (or at least believed to be correctly) running code; making a change to the code then changes the results for some unknown reason. Often even determining the precise point at which the old and new codes diverge is a major pain. In other cases, a code generates different results when run on different numbers of processors, although in exact arithmetic the same answer is expected. (Of course, this assumes that *exactly* the same solver and parameters are used in the two cases.) PETSc provides some support for determining exactly where in the code the computations lead to different results. First, compile both programs with different names. Next, start running both programs as a single MPI job. This procedure is dependent on the particular MPI implementation being used. For example, when using MPICH on workstations, *procgroup* files can be used to specify the processors on which the job is to be run. Thus, to run two programs, `old` and `new`, each on two processors, one should create the procgroup file with the following contents:

```
local 0
workstation1 1 /home/bsmith/old
workstation2 1 /home/bsmith/new
workstation3 1 /home/bsmith/new
```

(Of course, workstation1, etc. can be the same machine.) Then, one can execute the command

`mpirun` -p4pg <procgroup_filemame> old -compare <tolerance> [options]

Note that the same runtime options must be used for the two programs. The first time an inner product or norm detects an inconsistency larger than `<tolerance\trl{>}`, PETSc will generate an error. The usual runtime options `-start_in_debugger` and `-on_error_attach_debugger` may be used. The user can also place the commands

`PetscCompareDouble` ()
`PetscCompareScalar` ()
`PetscCompareInt` ()

in portions of the application code to check for consistency between the two versions.

14.6 Complex Numbers

PETSc supports the use of complex numbers in application programs written in C, C++, and Fortran. To do so, we employ C++ versions of the PETSc libraries in which the basic "scalar" datatype, given in PETSc codes by `PetscScalar`, is defined as `complex` (or `complex\trl{<}double\trl{>}` for machines using templated complex class libraries). To work with complex numbers, the user should compile the PETSc libraries (including the Fortran interface library) and the application code with `BOPT=[g_complex,O_complex]` for debugging, optimized, and profiling versions, respectively. The file `${PETSC_DIR}/docs/installation/index.htm` provides detailed instructions for installing PETSc. We recommend using optimized Fortran kernels for some key numerical routines with complex numbers (such as matrix-vector products, vector norms, etc.) instead of the default C++ routines. See the "Complex Numbers" section of the file `${PETSC_DIR}/docs/installation/index.htm` for details on building these kernels. This implementation exploits the maturity of Fortran compilers while retaining the identical user interface. For example, on rs6000 machines, the base single-node performance when using the Fortran kernels is 4-5 times faster than the default C++ code. Recall that each variant of the PETSc libraries is stored in a different directory, given by `${PETSC_DIR}/lib/lib${BOPT}/${PETSC_ARCH}`, according to the architecture and `BOPT` optimization variable. Thus, the libraries for complex numbers are maintained separately from those for real numbers. When using any of the complex numbers versions of PETSc, *all* vector and matrix elements are treated as complex, even if their imaginary components are zero. Of course, one can elect to use only the real parts of the complex numbers when using the complex versions of the PETSc libraries; however, when working *only* with real numbers in a code, one should use a version of PETSc for real numbers for best efficiency. The program `${PETSC_DIR}/src/sles/examples/tutorials/ex11.c` solves a linear system with a complex coefficient matrix. Its Fortran counterpart is `${PETSC_DIR}/src/sles/examples/tutorials/ex11f.F`.

14.7 Emacs Users

If users develop application codes using Emacs (which we highly recommend), the `etags` feature can be used to search PETSc files quickly and efficiently. To use this feature, one should first check if the file, `${PETSC_DIR}/TAGS` exists. If this file is not present, it should be generated by running `make etags` from the PETSc home directory. Once the file exists, from Emacs the user should issue the command

M-x visit-tags-table

where "M" denotes the Emacs Meta key, and enter the name of the TAGS file. Then the command "M-." will cause Emacs to find the file and line number where a desired PETSc function is defined. Any string in any of the PETSc files can be found with the command "M-x tags-search". To find repeated occurrences, one can simply use "M-," to find the next occurrence.

14.8 Parallel Communication

When used in a message-passing environment, all communication within PETSc is done through MPI, the message-passing interface standard [15]. Any file that includes petsc.h (or any other PETSc include file), can freely use any MPI routine.

14.9 Graphics

PETSc graphics library is not intended to compete with high-quality graphics packages. Instead, it is intended to be easy to use interactively with PETSc programs. We urge users to generate their publication-quality graphics using a professional graphics package. If a user wants to hook certain packages in PETSc, he or she should send a message to petsc-maint@mcs.anl.gov, and we will see whether it is reasonable to try to provide direct interfaces.

14.9.1 Windows as PetscViewers

For drawing predefined PETSc objects such as matrices and vectors, one must first create a viewer using the command

PetscViewerDrawOpenX(MPI_Comm comm ,char *display,char *title,int x,
int y,int w,int h,PetscViewer *viewer);

This viewer may be passed to any of the XXXView() routines. To draw into the viewer, one must obtain the Draw object with the command

PetscViewerDrawGetDraw(PetscViewer viewer,PetscDraw *draw);

Then one can call any of the PetscDrawXXX commands on the draw object. If one obtains the draw object in this manner, one does not call the PetscDrawOpenX () command discussed below. Predefined viewers, PETSC_VIEWER_DRAW_WORLD and PETSC_VIEWER_DRAW_SELF, may be used at any time. Their initial use will cause the appropriate window to be created. By default, PETSc drawing tools employ a private colormap, which remedies the problem of poor color choices for contour plots due to an external program's mangling of the colormap (e.g, Netscape tends to do this). Unfortunately, this causes flashing of colors as the mouse is moved between the PETSc windows and other windows. Alternatively, a shared colormap can be used via the option -draw_x_shared_colormap.

14.9.2 Simple PetscDrawing

One can open a window that is not associated with a viewer directly under the X11 Window System with the command

PetscDrawOpenX (MPI_Comm comm ,char *display,char *title,int x,
int y,int w,int h,PetscDraw *win);

All drawing routines are done relative to the windows coordinate system and viewport. By default the drawing coordinates are from `(0,0)` to `(1,1)`, where `(0,0)` indicates the lower left corner of the window. The application program can change the window coordinates with the command

`PetscDrawSetCoordinates`(`PetscDraw` win,double xl,double yl,double xr,double yr);

By default, graphics will be drawn in the entire window. To restrict the drawing to a portion of the window, one may use the command

`PetscDrawSetViewPort`(`PetscDraw` win,double xl,double yl,double xr,double yr);

These arguments, which indicate the fraction of the window in which the drawing should be done, must satisfy $0 \leq \mathtt{xl} \leq \mathtt{xr} \leq 1$ and $0 \leq \mathtt{yl} \leq \mathtt{yr} \leq 1$.

To draw a line, one uses the command

`PetscDrawLine`(`PetscDraw` win,double xl,double yl,double xr,double yr,int cl);

The argument `cl` indicates the color (which is an integer between 0 and 255) of the line. A list of predefined colors may be found in `include/petscdraw.h` and includes `PETSC_DRAW_BLACK`, `PETSC_DRAW_RED`, `PETSC_DRAW_BLUE` etc. To ensure that all graphics actually have been displayed, one should use the command

`PetscDrawFlush`(`PetscDraw` win);

When displaying by using double buffering, which is set with the command

`PetscDrawSetDoubleBuffer`(`PetscDraw` win);

all processors must call

`PetscDrawSynchronizedFlush`(`PetscDraw` win);

in order to swap the buffers. From the options database one may use `-draw_pause n`, which causes the PETSc application to pause `n` seconds at each `PetscDrawPause`(). A time of `-1` indicates that the application should pause until receiving mouse input from the user. Text can be drawn with either of the two commands

`PetscDrawString`(`PetscDraw` win,double x,double y,int color,char *text);
`PetscDrawStringVertical`(`PetscDraw` win,double x,double y,int color,char *text);

The user can set the text font `size` or determine it with the commands

`PetscDrawStringSetSize`(`PetscDraw` win,double width,double height);
`PetscDrawStringGetSize`(`PetscDraw` win,double *width,double *height);

14.9.3 Line Graphs

PETSc includes a set of routines for manipulating simple two-dimensional graphs. These routines, which begin with `PetscDrawAxisDraw`(), are usually not used directly by the application programmer. Instead, the programmer employs the line graph routines to draw simple line graphs. As shown in the program, within Figure 20, line graphs are created with the command

`PetscDrawLGCreate`(`PetscDraw` win,int ncurves,`PetscDrawLG` *ctx);

The argument `ncurves` indicates how many curves are to be drawn. Points can be added to each of the curves with the command

PetscDrawLGAddPoint(PetscDrawLG ctx,double *x,double *y);

The arguments `x` and `y` are arrays containing the next point value for each curve. Several points for each curve may be added with

PetscDrawLGAddPoints(PetscDrawLG ctx,int n,double **x,double **y);

The line graph is drawn (or redrawn) with the command

PetscDrawLGDraw(PetscDrawLG ctx);

A line graph that is no longer needed can be destroyed with the command

PetscDrawLGDestroy(PetscDrawLG ctx);

To plot new curves, one can reset a linegraph with the command

PetscDrawLGReset(PetscDrawLG ctx);

The line graph automatically determines the range of values to display on the two axes. The user can change these defaults with the command

PetscDrawLGSetLimits(PetscDrawLG ctx,double xmin,double xmax,double ymin,double ymax);

It is also possible to change the display of the axes and to label them. This procedure is done by first obtaining the axes context with the command

PetscDrawLGGetAxis(PetscDrawLG ctx,PetscDrawAxis *axis);

One can set the axes' colors and labels, respectively, by using the commands

PetscDrawAxisSetColors(PetscDrawAxis axis,int axis_lines,int ticks,int text);
PetscDrawAxisSetLabels(PetscDrawAxis axis,char *top,char *x,char *y);

```
/*$Id: ex3.c,v 1.42 2001/08/07 21:28:44 bsmith Exp $*/

static char help[] = "Plots a simple line graph.\n";

#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **argv)
{
  PetscDraw          draw;
  PetscDrawLG        lg;
  PetscDrawAxis      axis;
  int                n = 20,i,ierr,x = 0,y = 0,width = 300,height = 300,nports
= 1;
  PetscTruth         flg;
  char               *xlabel,*ylabel,*toplabel;
  PetscReal          xd,yd;
  PetscDrawViewPorts *ports;

  xlabel = "X-axis Label";toplabel = "Top Label";ylabel = "Y-axis Label";
```

```
  ierr = PetscInitialize(&argc,&argv,(char*)0,help);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(PETSC_NULL,"-width",&width,PETSC_NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(PETSC_NULL,"-height",&height,PETSC_NULL);CHKERRQ(ierr);
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
  ierr = PetscOptionsHasName(PETSC_NULL,"-nolabels",&flg);CHKERRQ(ierr);
  if (flg) {
    xlabel = (char *)0; toplabel = (char *)0;
  }
  /* ierr = PetscDrawOpenX(PETSC_COMM_SELF,0,"Title",x,y,width,height,&draw);CHKERRQ(ierr)
  ierr = PetscDrawCreate(PETSC_COMM_SELF,0,"Title",x,y,width,height,&draw);CHKERRQ(ierr);
  ierr = PetscDrawSetFromOptions(draw);CHKERRQ(ierr);

  ierr = PetscOptionsGetInt(PETSC_NULL,"-nports",&nports,PETSC_NULL);CHKERRQ(ierr);
  ierr = PetscDrawViewPortsCreate(draw,nports,&ports);CHKERRQ(ierr);
  ierr = PetscDrawViewPortsSet(ports,0);CHKERRQ(ierr);

  ierr = PetscDrawLGCreate(draw,1,&lg);CHKERRQ(ierr);
  ierr = PetscDrawLGGetAxis(lg,&axis);CHKERRQ(ierr);
  ierr = PetscDrawAxisSetColors(axis,PETSC_DRAW_BLACK,PETSC_DRAW_RED,PETSC_DRAW_BLUE);CHKER
  ierr = PetscDrawAxisSetLabels(axis,toplabel,xlabel,ylabel);CHKERRQ(ierr);

  for (i=0; i<n ; i++) {
    xd = (PetscReal)(i - 5); yd = xd*xd;
    ierr = PetscDrawLGAddPoint(lg,&xd,&yd);CHKERRQ(ierr);
  }
  ierr = PetscDrawLGIndicateDataPoints(lg);CHKERRQ(ierr);
  ierr = PetscDrawLGDraw(lg);CHKERRQ(ierr);
  ierr = PetscDrawFlush(draw);CHKERRQ(ierr);
  ierr = PetscSleep(2);CHKERRQ(ierr);

  ierr = PetscDrawViewPortsDestroy(ports);CHKERRQ(ierr);
  ierr = PetscDrawLGDestroy(lg);CHKERRQ(ierr);
  ierr = PetscDrawDestroy(draw);CHKERRQ(ierr);
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

Figure 20: Example of PetscDrawing Plots

It is possible to turn off all graphics with the option `-nox`. This will prevent any windows from being opened or any drawing actions to be done. This is useful for running large jobs when the graphics overhead is too large, or for timing.

14.9.4 Graphical Convergence Monitor

For both the linear and nonlinear solvers default routines allow one to graphically monitor convergence of the iterative method. These are accessed via the command line with `-ksp_xmonitor` and `-snes_xmonitor`. See also Sections 4.3.3 and 5.3.2.

The two functions used are KSPLGMonitor() and `KSPLGMonitorCreate` () . These can easily be modified to serve specialized needs.

14.9.5 Disabling Graphics at Compile Time

To disable all x-window-based graphics, edit the file ${PETSC_DIR}/bmake/${PETSC_ARCH}/packages and comment the variable PETSC_DHAVE_X11. Then (re)compile the PETSc libraries.

Chapter 15

Makefiles

This chapter describes the design of the PETSc makefiles, which are the key to managing our code portability across a wide variety of UNIX and Windows systems.

15.1 Our Makefile System

To make a program named `ex1`, one may use the command

make BOPT=[g,O] PETSC_ARCH=arch ex1

which will compile a debugging, optimized, or profiling version of the example and automatically link the appropriate libraries. The architecture, `arch`, is one of `solaris, rs6000, IRIX, hpux`, etc. Note that when using command line options with make (as illustrated above), one must *not* place spaces on either side of the "=" signs. The variables `BOPT` and `PETSC_ARCH` can also be set as environmental variables. Although PETSc is written in C, it can be compiled with a C++ compiler. For many C++ users this may be the preferred route. To compile with the C++ compiler, one should use the option `BOPT=g_c++` or `BOPT=O_c++`. The options `BOPT=g_complex` and `BOPT=O_complex` will create C versions that use complex double-precision numbers.

15.1.1 Makefile Commands

The directory `${PETSC_DIR}/bmake` contains virtually all makefile commands and customizations to enable portability across different architectures. Most makefile commands for maintaining the PETSc system are defined in the file `${PETSC_DIR}/bmake/common`. These commands, which process all appropriate files within the directory of execution, include

- `lib` - Updates the PETSc libraries based on the source code in the directory.
- `libfast` - Updates the libraries faster. Since `libfast` recompiles all source files in the directory at once, rather than individually, this command saves time when many files must be compiled.
- `clean` - Removes garbage files.

The `tree` command enables the user to execute a particular action within a directory and all of its subdirectories. The action is specified by `ACTION=[action]`, where `action` is one of the basic commands listed above. For example, if the command

make BOPT=g ACTION=lib tree

were executed from the directory `${PETSC_DIR}/src/sles/ksp`, the debugging library for all Krylov subspace solvers would be built.

15.1.2 Customized Makefiles

The directory ${PETSC_DIR}/bmake contains a subdirectory for each architecture that contains machine-specific information, enabling the portability of our makefile system. For instance, for Sun workstations running OS 5.7, the directory is called solaris. Each architecture directory contains three makefiles:

- packages - locations of all needed packages for a particular site. This file (discussed below) is usually the only one that the user needs to alter.
- variables - definitions of the compilers, linkers, etc.
- rules - some build rules specific to this machine.

The architecture independent makefiles, are located in ${PETSC_DIR}/bmake/common, and the machine-spcecific makefiles get included from here.

15.2 PETSc Flags

PETSc has several flags that determine how the source code will be compiled. The default flags for particular versions are specified by the variable PETSCFLAGS within the base files of ${PETSC_DIR}/bmake/${PETSC_ARCH}, discussed in Section 15.1.2. The flags include

- PETSC_USE_DEBUG - The PETSc debugging options are activated. We recommend always using this.
- PETSC_USE_COMPLEX - The version with scalars represented as complex numbers is used.
- PETSC_USE_LOG - Various monitoring statistics on floating-point operations, and message-passing activity are kept.

15.2.1 Sample Makefiles

Maintaining portable PETSc makefiles is very simple. In Figures 21, 22, and 23 we present three sample makefiles.

The first is a "minimum" makefile for maintaining a single program that uses the PETSc libraires. The most important line in this makefile is the line starting with include:

include ${PETSC_DIR}/bmake/common/base

This line includes other makefiles that provide the needed definitions and rules for the particular base PETSc installation (specified by ${PETSC_DIR}) and architecture (specified by ${PETSC_ARCH}). (See 1.2 for information on setting these environmental variables.) As listed in the sample makefile, the appropriate include file is automatically completely specified; the user should *not* alter this statement within the makefile.

```
ALL: ex2
CFLAGS   =
   FFLAGS   =
   CPPFLAGS =
FPPFLAGS =
include ${PETSC_DIR}/bmake/common/base
ex2: ex2.o chkopts
${CLINKER} -o ex2 ex2.o  ${PETSC_LIB}
${RM} ex2.o
```

Figure 21: Sample PETSc Makefile for a Single Program

Note that the variable `${PETSC_LIB}` (as listed on the link line in the above makefile) specifies *all* of the various PETSc libraries in the appropriate order for correct linking. For users who employ only a specific PETSc library, can use alternative variables like `${PETSC_SYS_LIB}`, `${PETSC_VEC_LIB}`, `${PETSC_MAT_LIB}`, `${PETSC_DM_LIB}`, `${PETSC_SLES_LIB}`, `${PETSC_SNES_LIB}` or `${PETSC_TS_LIB}`. The second sample makefile, given in Figure 22, controls the generation of several example programs.

```
CFLAGS   =
   FFLAGS   =
   CPPFLAGS =
FPPFLAGS =
include ${PETSC_DIR}/bmake/common/base
ex1: ex1.o
        -${CLINKER} -o ex1 ex1.o  ${PETSC_LIB}
${RM} ex1.o
ex2: ex2.o
        -${CLINKER} -o ex2 ex2.o  ${PETSC_LIB}
${RM} ex2.o
ex3: ex3.o
        -${FLINKER} -o ex3 ex3.o  ${PETSC_FORTRAN_LIB} ${PETSC_LIB}
${RM} ex3.o
ex4: ex4.o
        -${CLINKER} -o ex4 ex4.o  ${PETSC_LIB}
${RM} ex4.o
runex1:
-@${MPIRUN} ex1
runex2:
-@${MPIRUN} -np 2 ex2 -mat_seqdense -options_left
runex3:
-@${MPIRUN} ex3 -v -log_summary
runex4:
-@${MPIRUN} -np 4 ex4 -trdump
RUNEXAMPLES_1 = runex1 runex2
RUNEXAMPLES_2 = runex4
RUNEXAMPLES_3 = runex3
EXAMPLESC     = ex1.c ex2.c ex4.c
EXAMPLESF     = ex3.F
EXAMPLES_1    = ex1 ex2
EXAMPLES_2    = ex4
EXAMPLES_3    = ex3
include ${PETSC_DIR}/bmake/common/test
```

Figure 22: Sample PETSc Makefile for Several Example Programs

Again, the most important line in this makefile is the `include` line that includes the files defining all of the macro variables. Some additional variables that can be used in the makefile are defined as follows:

- `CFLAGS, FFLAGS` - User specified additional options for the C compiler and fortran compiler.
- `CPPFLAGS, FPPFLAGS` - User specified additional flags for the C preprocessor and fortran preprocesor.

- CLINKER, FLINKER - the C and Fortran linkers.
- RM - the remove command for deleting files.
- EXAMPLES_1 - examples that will be built with make BOPT=[g,O] examples (see Section 15.1.1)
- RUNEXAMPLES_1 - examples that will be run with make runexamples (see Section 15.1.1)
- EXAMPLESC - all C examples that will be checked in/out of RCS with make ci and make co (not generally needed by users).
- EXAMPLESF - all Fortran examples that will be checked in/out of RCS with make ci and make co (not generally needed by users).
- PETSC_LIB - all of the base PETSc libraries.
- PETSC_FORTRAN_LIB - the PETSc Fortran interface library.

Note that the PETSc example programs are divided into several categories, which currently include:

EXAMPLES_1 – basic C suite used in installation tests
EXAMPLES_2 – additional C suite including graphics
EXAMPLES_3 – basic Fortran .F suite
EXAMPLES_4 – subset of 1 and 2 that runs on only a single processor
EXAMPLES_5 – examples that require complex numbers
EXAMPLES_6 – C examples that do not work with complex numbers
EXAMPLES_8 – Fortran .F examples that do not work with complex numbers
EXAMPLES_9 – uniprocessor version of 3
EXAMPLES_10 – Fortran .F examples that require complex numbers

We next list in Figure 23 a makefile that maintains a PETSc library. Although most users do not need to understand or deal with such makefiles, they are also easily used.

```
ALL: lib
CFLAGS =
SOURCEC = sp1wd.c spinver.c spnd.c spqmd.c sprcm.c
SOURCEF = degree.f fnroot.f genqmd.f qmdqt.f rcm.f fn1wd.f gen1wd.f
genrcm.f qmdrch.f rootls.f fndsep.f gennd.f qmdmrg.f qmdupd.f
SOURCEH =
OBJSC = sp1wd.o spinver.o spnd.o spqmd.o sprcm.o
OBJSF = degree.o fnroot.o genqmd.o qmdqt.o rcm.o fn1wd.o gen1wd.o
genrcm.o qmdrch.o rootls.o fndsep.o gennd.o qmdmrg.o qmdupd.o
LIBBASE = libpetscmat
MANSEC = Mat
include ${PETSC_DIR}/bmake/common/base
```

Figure 23: Sample PETSc Makefile for Library Maintenance

The library's name is libpetscmat.a, and the source files being added to it are indicated by SOURCEC (for C files) and SOURCEF (for Fortran files). Note that the OBJSF and OBJSC are identical to SOURCEF and SOURCEC, respectively, except they use the suffix .o rather than .c or .f.

The variable MANSEC indicates that any manual pages generated from this source should be included in the Mat section.

15.3 Limitations

This approach to portable makefiles has some minor limitations, including the following:

- Each makefile must be called “makefile”.
- Each makefile can maintain at most one archive library.

Chapter 16

Unimportant and Advanced Features of Matrices and Solvers

This chapter introduces additional features of the PETSc matrices and solvers. Since most PETSc users should not need to use these features, we recommend skipping this chapter during an initial reading.

16.1 Extracting Submatrices

One can extract a (parallel) submatrix from a given (parallel) using

MatGetSubMatrix(Mat A,IS rows,IS cols,int csize,MatReuse call,Mat *B);

This extracts the `rows` and `columns` of the matrix `A` into `B`. If call is `MAT_INITIAL_MATRIX` it will create the matrix `B`. If call is `MAT_REUSE_MATRIX` it will reuse the `B` created with a previous call. The argument `csize` is ignored on sequential matrices, for parallel matrices it determines the "local columns" if the matrix format supports this concept. Often one can use the default by passing in PETSC_DECIDE. To create a `B` matrix that may be multiplied with a vector `x` one can use

VecGetLocalSize(x,&csize);
MatGetSubMatrix(Mat A,IS rows,IS cols,int csize,MatReuse call,Mat *B);

16.2 Matrix Factorization

Normally, PETSc users will access the matrix solvers through the SLES interface, as discussed in Chapter 4, but the underlying factorization and triangular solve routines are also directly accessible to the user. The

LU and Cholesky matrix factorizations are split into two or three stages depending on the user's needs. The first stage is to calculate an ordering for the matrix. The ordering generally is done to reduce fill in a sparse factorization; it does not make much sense for a dense matrix.

MatGetOrdering(Mat matrix,MatOrderingType type,IS * rowperm,IS * colperm);

The currently available alternatives for the ordering `type` are

- `MATORDERING_NATURAL` - Natural

- `MATORDERING_ND` - Nested Dissection
- `MATORDERING_1WD` - One-way Dissection
- `MATORDERING_RCM` - Reverse Cuthill-McKee
- `MATORDERING_QMD` - Quotient Minimum Degree

These orderings can also be set through the options database by specifying one of the following: `-pc_lu_ordering_type natural`, `-pc_lu_ordering_type nd`, `-pc_lu_ordering_type 1wd`, `-pc_lu_ordering_type rcm`, `-pc_lu_ordering_type qmd`. Certain matrix formats may support only a subset of these; more options may be added. Check the manual pages for up-to-date information. All of these orderings are symmetric at the moment; ordering routines that are not symmetric may be added. Currently we support orderings only for sequential matrices. Users can add their own ordering routines by providing a function with the calling sequence

int reorder(`Mat` A,`MatOrderingType` type,`IS` * rowperm,`IS` * colperm);

Here `A` is the matrix for which we wish to generate a new ordering, `type` may be ignored and `rowperm` and `colperm` are the row and column permutations generated by the ordering routine. The user registers the ordering routine with the command

`MatOrderingRegisterDynamic` (`MatOrderingType` inname,char *path,char *sname,
int (*reorder)(`Mat` ,`MatOrderingType` ,`IS` *,`IS` *)));

The input argument `inname` is a string of the user's choice, `iname` is either an ordering defined in `petscmat.h` or a users string, to indicate one is introducing a new ordering, while the output See the code in `src/mat/impls/order/sorder.c` and other files in that directory for examples on how the reordering routines may be written. Once the reordering routine has been registered, it can be selected for use at runtime with the command line option `-pc_lu_ordering_type sname`. If reordering directly, the user should provide the `name` as the second input argument of `MatGetOrdering` (). The following routines perform complete, in-place, symbolic, and numerical factorizations for symmetric and nonsymmetric matrices, respectively:

`MatCholeskyFactor` (`Mat` matrix,`IS` permutation,double pf);
`MatLUFactor` (`Mat` matrix,`IS` rowpermutation,`IS` columnpermutation,`MatLUInfo` *info);

The argument `info->fill > 1` is the predicted fill expected in the factored matrix, as a ratio of the original fill. For example, `info->fill=2.0` would indicate that one expects the factored matrix to have twice as many nonzeros as the original. For sparse matrices it is very unlikely that the factorization is actually done in-place. More likely, new space is allocated for the factored matrix and the old space deallocated, but to the user it appears in-place because the factored matrix replaces the unfactored matrix. The two factorization stages can also be performed separately, by using the out-of-place mode:

`MatCholeskyFactorSymbolic` (`Mat` matrix,`IS` perm, double pf,`Mat` *result);
`MatLUFactorSymbolic` (`Mat` matrix,`IS` rowperm,`IS` colperm,`MatLUInfo` *info,`Mat` *result);
`MatCholeskyFactorNumeric` (`Mat` matrix,`Mat` *result);
`MatLUFactorNumeric` (`Mat` matrix, `Mat` *result);

In this case, the contents of the matrix `result` is undefined between the symbolic and numeric factorization stages. It is possible to reuse the symbolic factorization. For the second and succeeding factorizations, one simply calls the numerical factorization with a new input `matrix` and the *same* factored `result` matrix. It is *essential* that the new input matrix have exactly the same nonzero structure as the original factored matrix.

(The numerical factorization merely overwrites the numerical values in the factored matrix and does not disturb the symbolic portion, thus enabling reuse of the symbolic phase.) In general, calling `XXXFactorSymbolic` with a dense matrix will do nothing except allocate the new matrix; the `XXXFactorNumeric` routines will do all of the work.

Why provide the plain `XXXfactor` routines when one could simply call the two-stage routines? The answer is that if one desires in-place factorization of a sparse matrix, the intermediate stage between the symbolic and numeric phases cannot be stored in a `result` matrix, and it does not make sense to store the intermediate values inside the original matrix that is being transformed. We originally made the combined factor routines do either in-place or out-of-place factorization, but then decided that this approach was not needed and could easily lead to confusion. We do not currently support sparse matrix factorization with pivoting for numerical stability. This is because trying to both reduce fill and do pivoting can become quite complicated. Instead, we provide a poor stepchild substitute. After one has obtained a reordering, with `MatGetOrdering`(`Mat` A,MatOrdering type,`IS` *row,`IS` *col) one may call

`MatReorderForNonzeroDiagonal`(`Mat` A,double tol,`IS` row, `IS` col);

which will try to reorder the columns to ensure that no values along the diagonal are smaller than `tol` in a absolute value. If small values are detected and corrected for, a nonsymmetric permutation of the rows and columns will result. This is not guaranteed to work, but may help if one was simply unlucky in the original ordering. When using the `SLES` solver interface the options `-pc_ilu_nonzeros_along_diagonal <tol>` and `-pc_lu_nonzeros_along_diagonal <tol>` may be used. Here, `tol` is an optional tolerance to decide if a value is nonzero; by default it is $1.e-10$.

Once a matrix has been factored, it is natural to solve linear systems. The following four routines enable this process:

`MatSolve`(`Mat` A,`Vec` x, `Vec` y);
`MatSolveTranspose`(`Mat` A, `Vec` x, `Vec` y);
`MatSolveAdd`(`Mat` A,`Vec` x, `Vec` y, `Vec` w);
`MatSolveTransposeAdd`(`Mat` A, `Vec` x, `Vec` y, `Vec` w);

The matrix `A` of these routines must have been obtained from a factorization routine; otherwise, an error will be generated. In general, the user should use the `SLES` solvers introduced in the next chapter rather than using these factorization and solve routines directly.

16.3 Unimportant Details of KSP

Again, virtually all users should use `KSP` through the `SLES` interface and, thus, will not need to know the details that follow.

It is possible to generate a Krylov subspace context with the command

`KSPCreate`(MPI_Comm `comm`,`KSP` *kps);

Before using the Krylov context, one must set the matrix-vector multiplication routine and the preconditioner with the commands

`PCSetOperators`(`PC` pc,`Mat` mat,`Mat` pmat,`MatStructure` flag);
`KSPSetPC`(`KSP` ksp,`PC` pc);

In addition, the `KSP` solver must be initialized with

`KSPSetUp`(`KSP` ksp);

Solving a linear system is done with the command

KSPSolve (KSP ksp,int *its);

Finally, the KSP context should be destroyed with

KSPDestroy (KSP ksp);

It may seem strange to put the matrix in the preconditioner rather than directly in the KSP ; this decision was the result of much agonizing. The reason is that for SSOR with Eisenstat's trick, and certain other preconditioners, the preconditioner has to change the matrix-vector multiply. This procedure could not be done cleanly if the matrix were stashed in the KSP context that PC cannot access. Any preconditioner can supply not only the preconditioner, but also a routine that essentially performs a complete Richardson step. The reason for this is mainly SOR. To use SOR in the Richardson framework, that is,

$$u^{n+1} = u^n + B(f - Au^n),$$

is much more expensive than just updating the values. With this addition it is reasonable to state that *all* our iterative methods are obtained by combining a preconditioner from the PC package with a Krylov method from the KSP package. This strategy makes things much simpler conceptually, so (we hope) clean code will result. *Note*: We had this idea already implicitly in older versions of SLES , but, for instance, just doing Gauss-Seidel with Richardson in old SLES was much more expensive than it had to be. With PETSc this should not be a problem.

16.4 Unimportant Details of PC

Most users will obtain their preconditioner contexts from the SLES context with the command SLESGetPC (). It is possible to create, manipulate, and destroy PC contexts directly, although this capability should rarely be needed. To create a PC context, one uses the command

PCCreate (MPI_Comm comm ,PC *pc);

The routine

PCSetType (PC pc,PCType method);

sets the preconditioner method to be used. The two routines

PCSetOperators (PC pc,Mat mat,Mat pmat,MatStructure flag);
PCSetVector (PC pc,Vec vec);

set the matrices and type of vector that are to be used with the preconditioner. The `vec` argument is needed by the PC routines to determine the format of the vectors. The routine

PCGetOperators (PC pc,Mat *mat,Mat *pmat,MatStructure *flag);

returns the values set with PCSetOperators (). The preconditioners in PETSc can be used in several ways. The two most basic routines simply apply the preconditioner or its transpose and are given, respectively, by

PCApply (PC pc,Vec x,Vec y);
PCApplyTranspose (PC pc,Vec x,Vec y);

In particular, for a preconditioner matrix, `B`, that has been set via `PCSetOperators(pc,A,B,flag)`, the routine `PCApply(pc,x,y)` computes $y = B^{-1}x$ by solving the linear system $By = x$ with the specified preconditioner method. Additional preconditioner routines are

```
PCApplyBAorAB(PC pc,PCSide right,Vec x,Vec y,Vec work,int its);
PCApplyBAorABTranspose(PC pc,PCSide right,Vec x,Vec y,Vec work,int its);
PCApplyRichardson(PC pc,Vec x,Vec y,Vec work,PetscReal rtol,PetscReal atol, PetscReal dtolint its);
```

The first two routines apply the action of the matrix followed by the preconditioner or the preconditioner followed by the matrix depending on whether the `right` is `PC_LEFT` or `PC_RIGHT`. The final routine applies `its` iterations of Richardson's method. The last three routines are provided to improve efficiency for certain Krylov subspace methods. A `PC` context that is no longer needed can be destroyed with the command

```
PCDestroy(PC pc);
```

Index

Bibliography

[1] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997. also available as Argonne preprint ANL/MCS-P634-0197.

[2] Peter N. Brown and Youcef Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.

[3] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. Technical Report CU-CS 843-97, Computer Science Department, University of Colorado-Boulder, 1997. (accepted by SIAM J. of Scientific Computing).

[4] J. E. Dennis Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[5] S. Eisenstat. Efficient implementation of a class of CG methods. *SIAM J. Sci. Stat. Comput.*, 2:1–4, 1981.

[6] S. C. Eisenstat and H. F. Walker. Choosing the forcing terms in an inexact Newton method. *SIAM J. Sci. Stat. Comput.*, 17:16–32, 1996.

[7] R. Freund, G. H. Golub, and N. Nachtigal. *Iterative Solution of Linear Systems*, pages 57–100. Acta Numerica. Cambridge University Press, 1992.

[8] Roland W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Stat. Comput.*, 14:470–482, 1993.

[9] William Gropp and Ewing Lusk. MPICH Web page. http://www.mcs.anl.gov/mpi/mpich.

[10] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[11] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, August 1991.

[12] Magnus R. Hestenes and Eduard Steifel. Methods of conjugate gradients for solving linear systems. *J. Research of the National Bureau of Standards*, 49:409–436, 1952.

[13] Mark T. Jones and Paul E. Plassmann. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, December 1995.

[14] Jorge J. Moré, Danny C. Sorenson, Burton S. Garbow, and Kenneth E. Hillstrom. The MINPACK project. In Wayne R. Cowell, editor, *Sources and Development of Mathematical Software*, pages 88–111, 1984.

[15] MPI: A message-passing interface standard. *International J. Supercomputing Applications*, 8(3/4), 1994.

[16] M. Pernice and H. F. Walker. NITSOL: A newton iterative solver for nonlinear systems. *SIAM J. Sci. Stat. Comput.*, 19:302–318, 1998.

[17] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[18] Barry F. Smith, Petter Bjørstad, and William D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[19] Peter Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[20] H. A. van der Vorst. BiCGSTAB: A fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.