Introduction to Julia:
Why are we doing this to you?
(Spring 2017)

Steven G. Johnson, MIT Applied Math

MIT classes 18.06, 18.303, 18.330, 18.08[56],
18.335, 18.337, ...
What language for teaching scientific computing?

For the most part, these are not hard-core programming courses, and we only need little “throw-away” scripts and toy numerical experiments.

Almost any high-level, interactive (dynamic) language with easy facilities for linear algebra \((Ax=b, Ax=\lambda x)\), plotting, mathematical functions, and working with large arrays of data would be fine.

And there are lots of choices...
Lots of choices for interactive math...
Just pick the most popular? Matlab or Python or R?

We feel guilty pushing a language on you that we are starting to abandon ourselves.

Traditional HL computing languages hit a performance wall in “real” work ... eventually force you to C, Cython, ...
A new programming language?

As high-level and interactive as Matlab or Python+IPython,
as general-purpose as Python,
as productive for technical work as Matlab or Python+SciPy,
but as fast as C.
Performance on synthetic benchmarks

[ loops, recursion, etc., implemented in most straightforward style ]

(normalized so that C speed = 1)
Special Functions in Julia

Special functions $s(x)$: classic case that cannot be vectorized well
... switch between various polynomials depending on $x$

Many of Julia’s special functions come from the usual C/Fortran libraries, but some are written in pure Julia code.

Pure Julia `erfinv(x)` [ = $\text{erf}^{-1}(x)$ ]
3–4× faster than Matlab’s and 2–3× faster than SciPy’s (Fortran Cephes).

Pure Julia `polygamma(m, z)` [ = $(m+1)^{th}$ derivative of the ln $\Gamma$ function ]
~ 2× faster than SciPy’s (C/Fortran) for real $z$
... and unlike SciPy’s, same code supports complex argument $z$

Julia code can actually be faster than typical “optimized” C/Fortran code, by using techniques [metaprogramming/code generation] that are hard in a low-level language.
Pure-Julia FFT performance

double-precision complex, 1d transforms
powers of two

(FFTW, MKL: “unfair” factor of \(~2\) from manual SIMD)

already comparable to FFTPACK

[ probably some tweaks to inlining will make it better ]

FFTW 1.0-like code generation
+ recursion in Julia
~ 1/3 lines of code compared to FFTPACK, more functionality
Generating Vandermonde matrices

given \( x = [\alpha_1, \alpha_2, \ldots], \) generate:

\[
V = \begin{bmatrix}
1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^{n-1} \\
1 & \alpha_2 & \alpha_2^2 & \ldots & \alpha_2^{n-1} \\
1 & \alpha_3 & \alpha_3^2 & \ldots & \alpha_3^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \alpha_m & \alpha_m^2 & \ldots & \alpha_m^{n-1} 
\end{bmatrix}
\]

NumPy (numpy.vander): [follow links]

Python code...wraps C code
...wraps generated C code

type-generic at high-level, but
low level limited to small set of types.

Writing fast code “in” Python or Matlab = mining the standard library
for pre-written functions (implemented in C or Fortran).

If the problem doesn’t “vectorize” into built-in functions,
if you have to write your own inner loops ... sucks for you.
Generating Vandermonde matrices

given \(x = [\alpha_1, \alpha_2, ...]\), generate:

\[
V = \begin{bmatrix}
1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^{n-1} \\
1 & \alpha_2 & \alpha_2^2 & \ldots & \alpha_2^{n-1} \\
1 & \alpha_3 & \alpha_3^2 & \ldots & \alpha_3^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \alpha_m & \alpha_m^2 & \ldots & \alpha_m^{n-1}
\end{bmatrix}
\]

NumPy (numpy.vander): [follow links]

Python code ... wraps C code
... wraps generated C code

type-generic at high-level, but
low level limited to small set of types.

Julia (type-generic code):

```julia
function vander{T}(x::AbstractVector{T}, n=length(x))
    m = length(x)
    V = Array(T, m, n)
    for j = 1:m
        V[j,1] = one(x[j])
    end
    for i = 2:n
        for j = 1:m
            V[j,i] = x[j] * V[j,i-1]
        end
    end
    return V
end
```
Generating Vandermonde matrices

function vander{T}(x::AbstractVector{T}, n=length(x))
    m = length(x)
    V = Array(T, m, n)
    for j = 1:m
        V[j,1] = one(x[j])
    end
    for i = 2:n
        for j = 1:m
            V[j,i] = x[j] * V[j,i-1]
        end
    end
    return V
end

note: works for any container of any type with "*" operation
... performance ≠ inflexibility
But I don’t “need” performance!

For lots of problems, especially “toy” problems in courses, Matlab/Python performance is good enough.

But if use those languages for all of your “easy” problems, then you won’t be prepared to switch when you hit a hard problem. When you need performance, it is too late.

You don’t want to learn a new language at the same time that you are solving your first truly difficult computational problem.
Just vectorize your code?
= rely on mature external libraries, operating on large blocks of data, for performance-critical code

Good advice! But...

• Someone has to write those libraries.

• Eventually that person will be you.
  — some problems are impossible or just very awkward to vectorize.
But everyone else is using Matlab/Python/R/…

Julia is still a young, niche language. That imposes real costs — lack of familiarity, rough edges, continual language changes. These are real obstacles.

But it also gives you advantages that Matlab/Python users don’t have.
But I lose access to all the libraries available for other languages?

Very easy to call C/Fortran libraries from Julia, and also to call Python...
Julia leverages Python...

Directly call Python libraries (PyCall package), e.g. to plot with Matplotlib (PyPlot package)

via IPython/Jupyter:
Modern multimedia interactive notebooks mixing code, results, graphics, rich text, equations, interaction

“IJulia”
goto live IJulia notebook demo...

Go to juliabox.org for install-free IJulia on the Amazon cloud

See also julialang.org for more tutorial materials...