

Introduction to Julia: Why are we doing this to you?

(Spring 2019)

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...

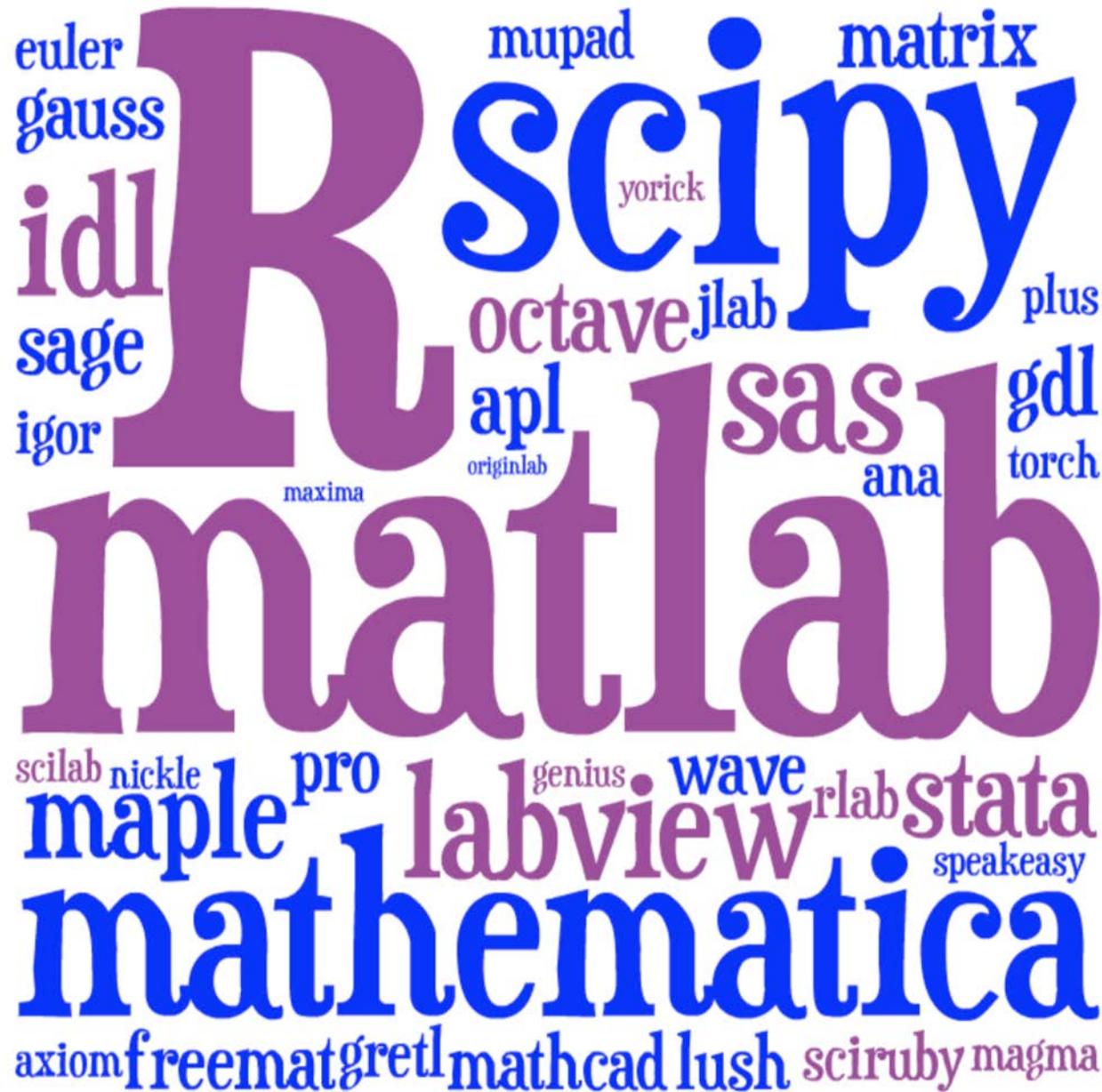


Image courtesy of Viral Shah. Used with permission.

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?

Jeff Bezanson

Viral Shah

julialang.org

Alan Edelman (MIT)

[begun 2009, “0.1” in 2013, ~40k commits,
1.0 release in Aug. 2018, 1.1 in Jan. 2019]

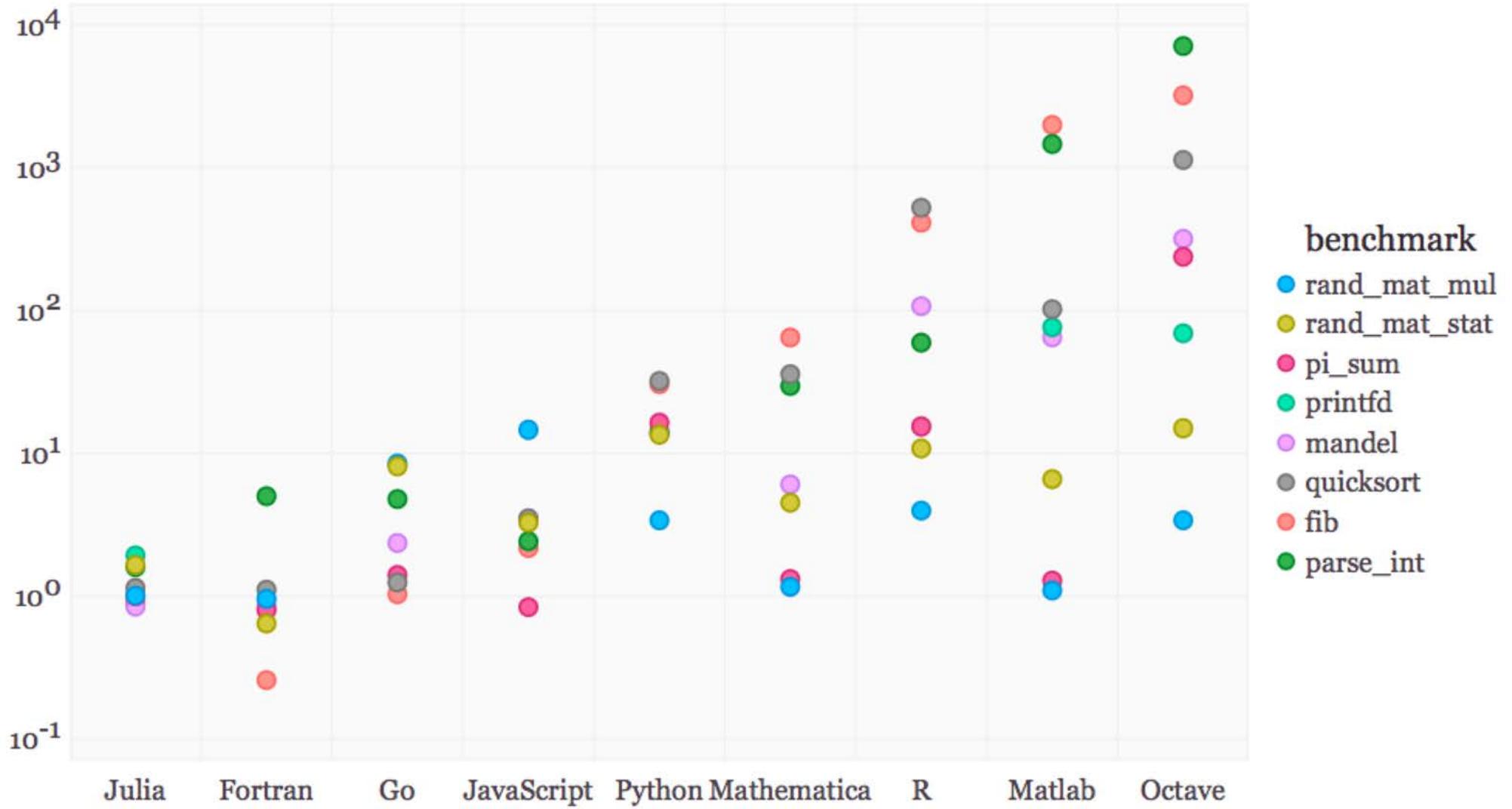
Stefan Karpinski

[30+ developers with 100+ commits,
1000+ external packages, 6th JuliaCon in 2019]

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)` [= $(m+1)^{\text{th}}$ derivative of the $\ln \Gamma$ function]

~ 2× faster than SciPy's (C/Fortran) for real z

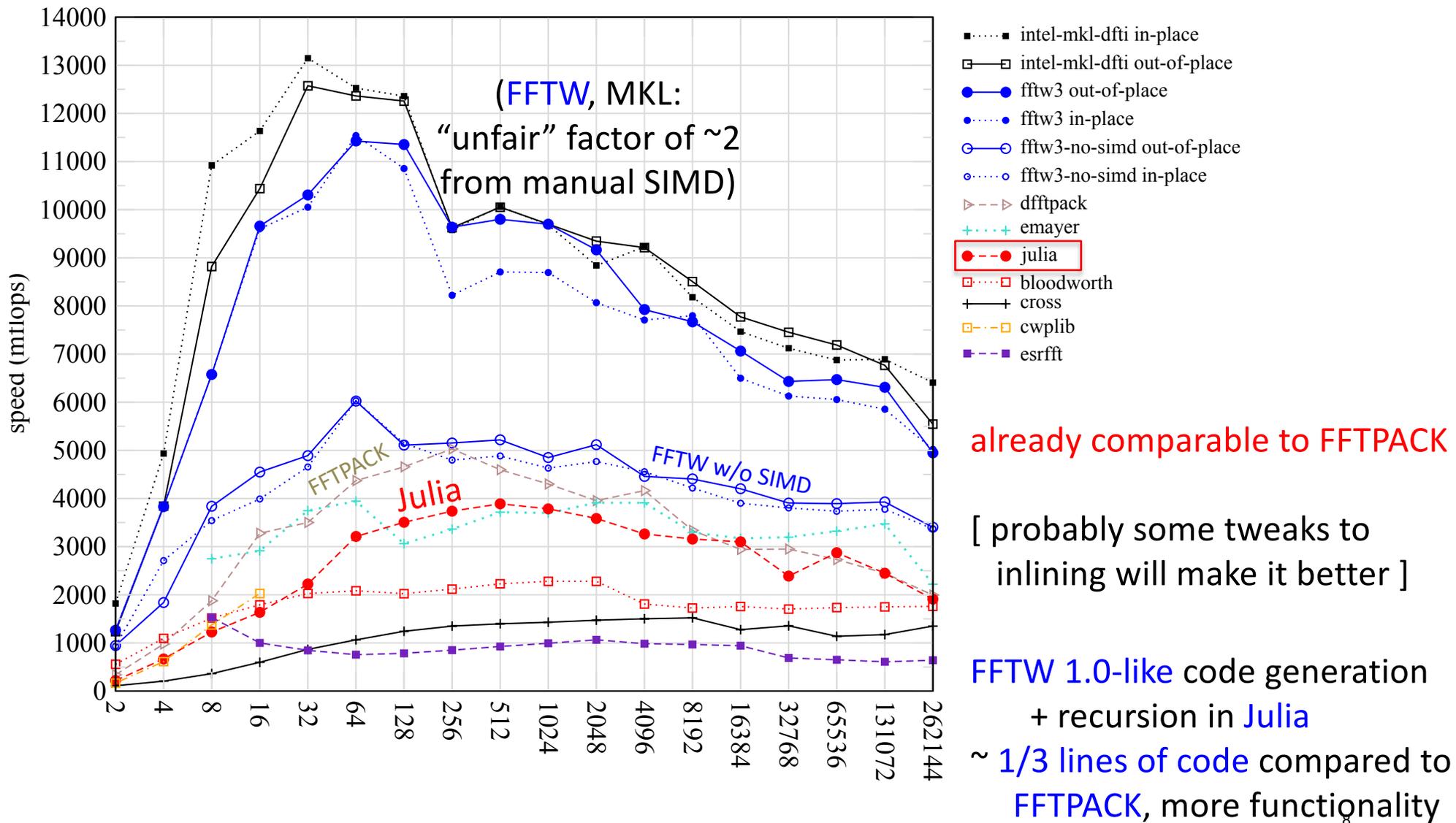
... and unlike SciPy's, *same code* supports complex argument z

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

Pure-Julia FFT performance

double-precision complex, 1d transforms

powers of two



Generating Vandermonde matrices

given $x = [\alpha_1, \alpha_2, \dots]$, generate:

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

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 $x = [\alpha_1, \alpha_2, \dots]$, generate:

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

NumPy (`numpy.vander`): *[follow links]*

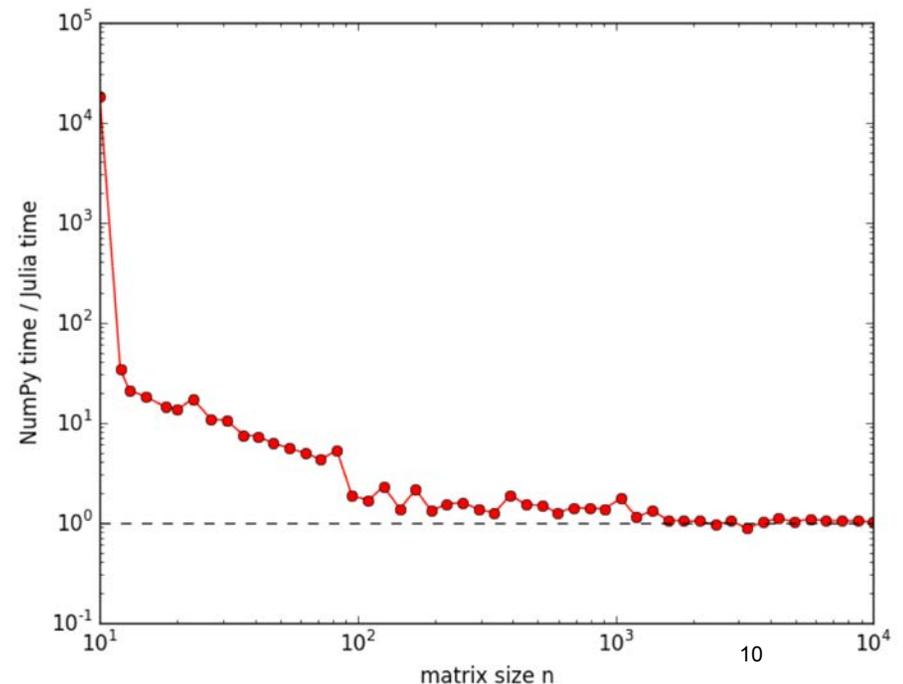
Python code ...wraps C code

... wraps generated C code

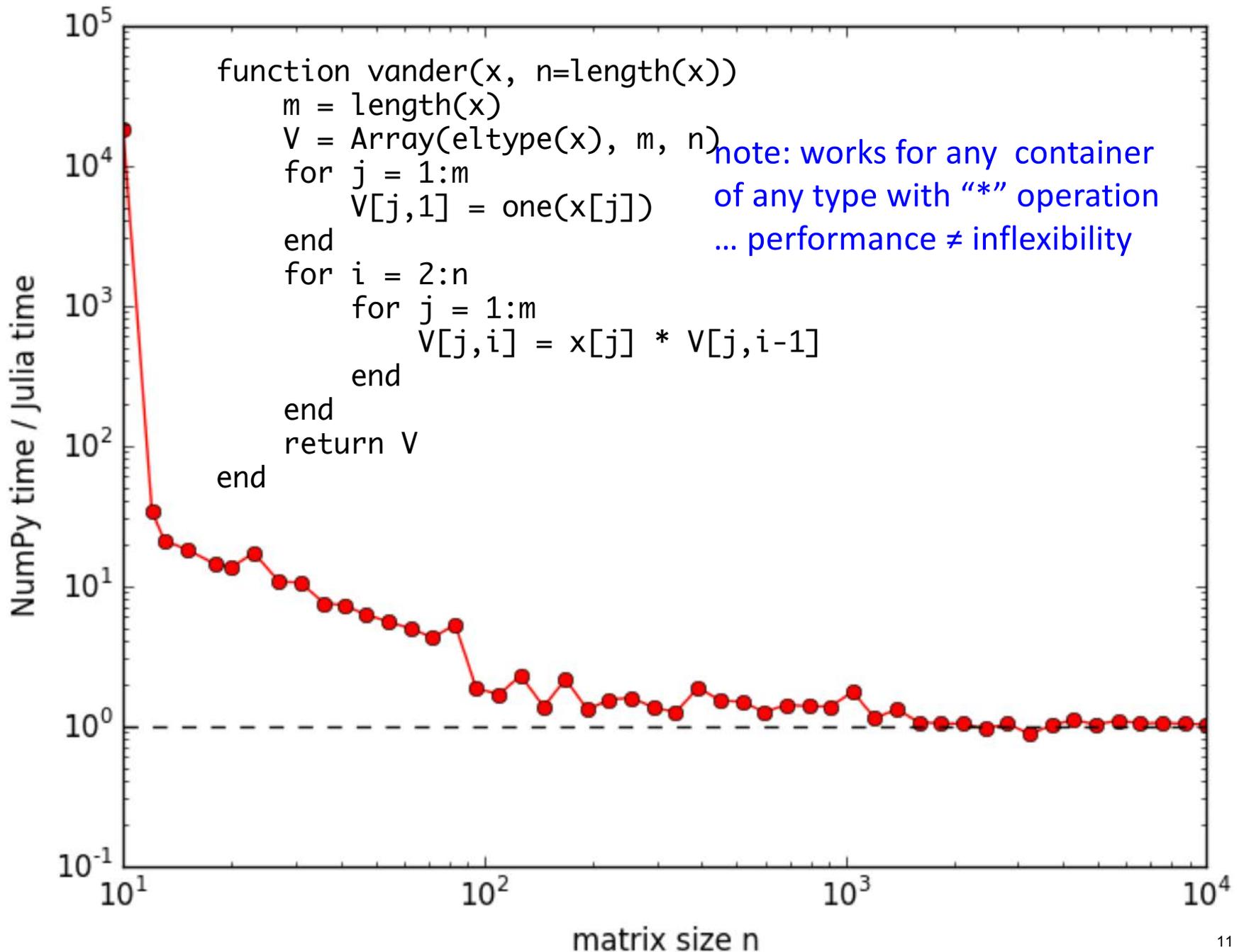
type-generic at high-level, but
low level limited to small set of types.

Julia (type-generic code):

```
function vander(x, n=length(x))
    m = length(x)
    V = Array{eltype(x), 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



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), and also...



via IPython/Jupyter:

Modern multimedia
interactive **notebooks**
mixing **code**, **results**,
graphics, **rich text**,
equations, **interaction**

“Julia”

goto live [Julia notebook](#) demo...

Go to [juliabox.org](#) for install-free Julia on the Amazon cloud

See also [julia.org](#) for more tutorial materials...

MIT OpenCourseWare

<https://ocw.mit.edu>

18.335J Introduction to Numerical Methods

Spring 2019

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.