High performance in dynamic languages: julia

6.172 guest lecture

Prof. Steven G. Johnson
MIT Applied Mathematics, MIT Physics

Dynamic languages for interactive math…

The two-language approach:

High-level dynamic language for productivity,

+ low-level language (C, Fortran, Cython, …) for performance-critical code.

= Huge jump in complexity, loss of generality.

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.

A new programming language?

Alan Edelman
[MIT]
Viral Shah
Jeff Bezanson
Stefan Karpinski

[30+ developers with 100+ commits,
1000+ external packages, 4th JuliaCon in 2017]

julia

julialang.org

[begun 2009, "0.1" in 2013, ~40k commits,
"0.6" release in June 2017,
1.0 release in August 2018]

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**.

Generating Vandermonde matrices

given x = [α_1, α_2, …], 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 $x = [\alpha_1, \alpha_2, \ldots]$, generate:

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

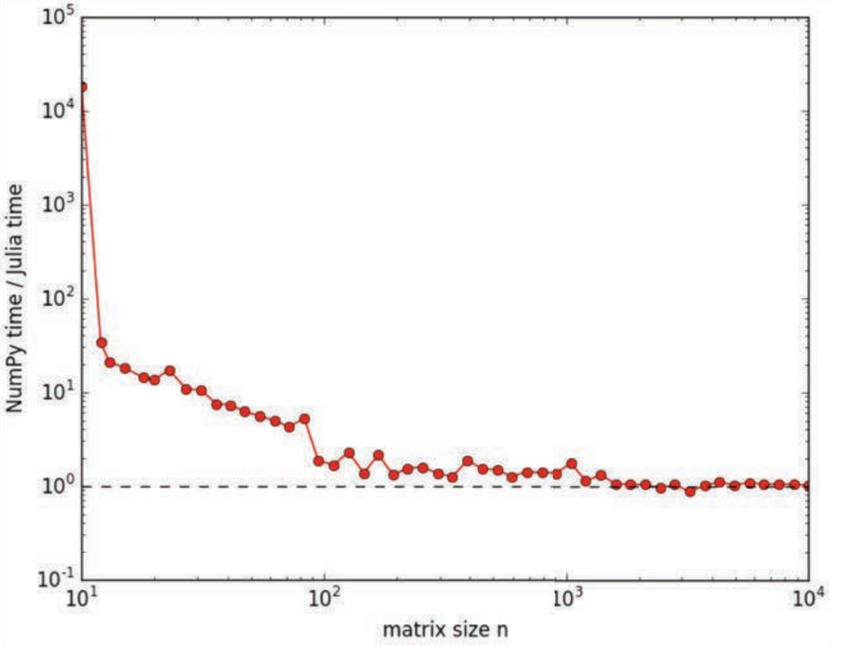
NumPy (`numpy.vander`): *[follow links]*

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

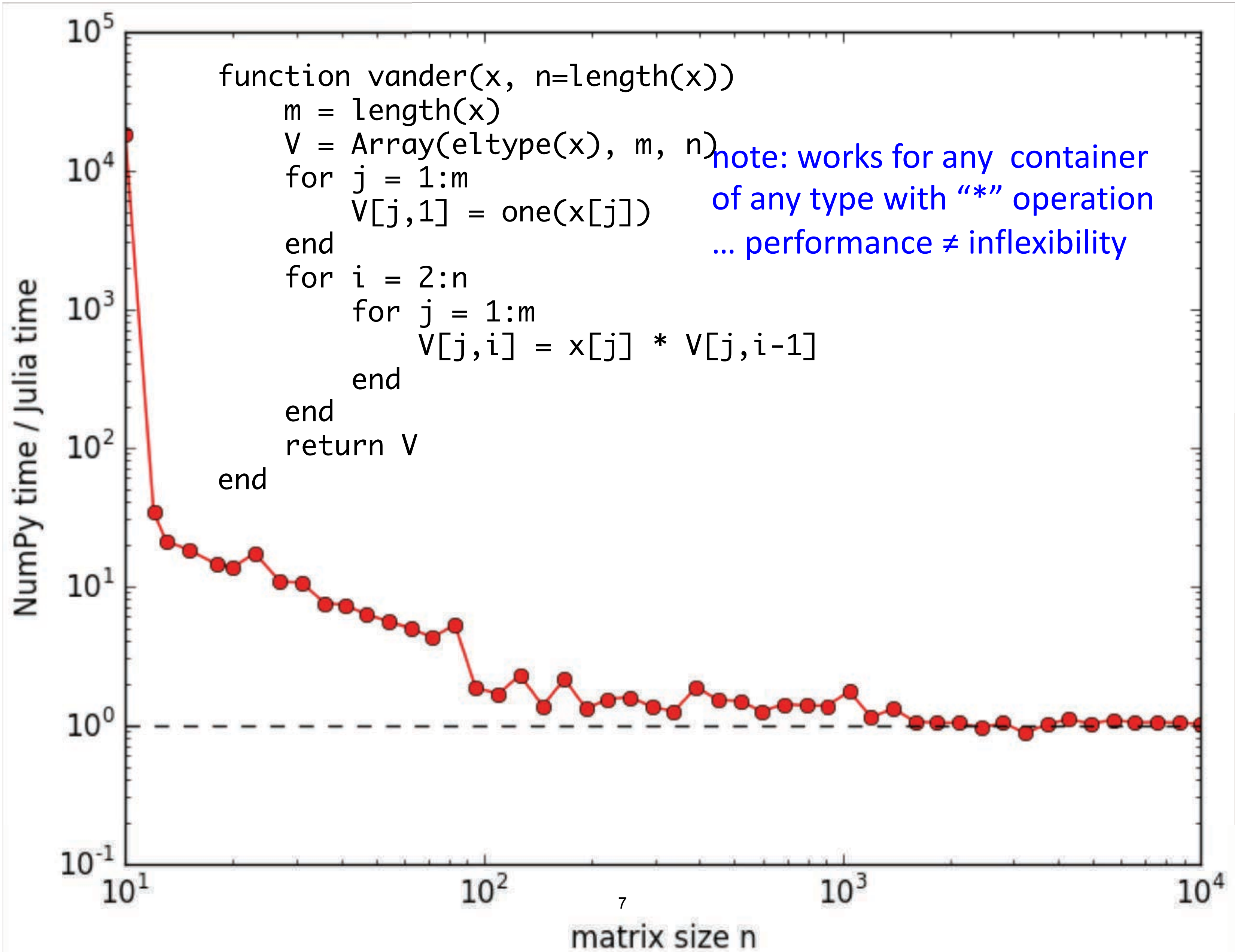
type-generic at high-level, but
low level limited to small set of types.

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



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 Γ 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.

Why can Julia be fast?

First need to understand: Why is Python slow?

`goto` Jupyter/IJulia notebooks from 18.S096.