Crossing Language Barriers with julia, SciPy, and IP[y]thon

Steven G. Johnson
MIT Applied Mathematics

Where I'm coming from…

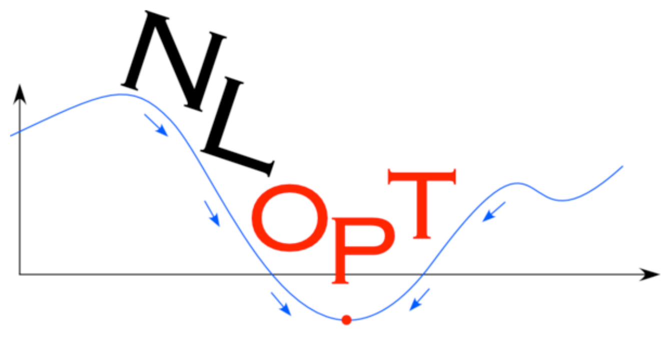
[google "Steven Johnson MIT"]

Computational software you may know…
… mainly C/C++ libraries & software …
… often with Python interfaces …
(& Matlab & Scheme & …)

FFTW

www.fftw.org

jdj.mit.edu/nlopt

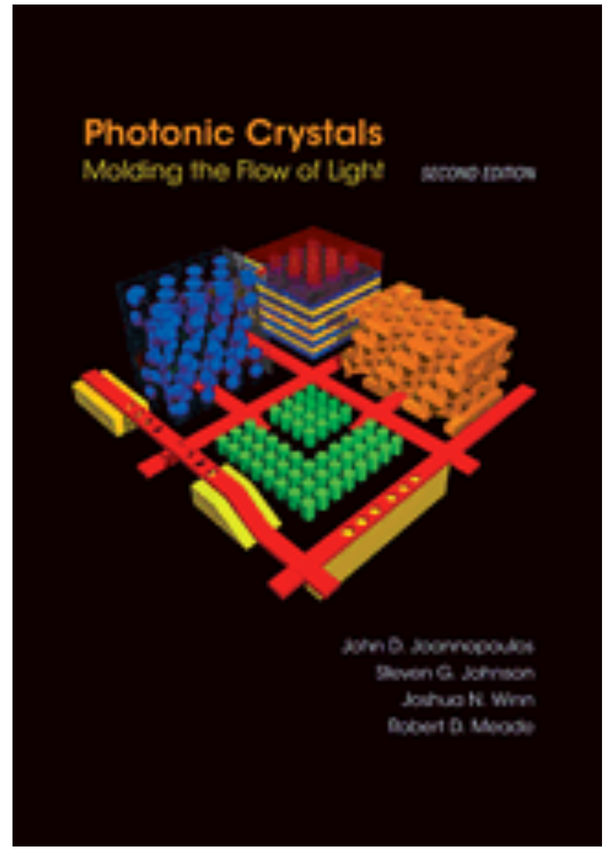


jdj.mit.edu/meep

& other EM simulators…

erf(z) (and erfc, erfi, …)
in SciPy 0.12+

Nanophotonics



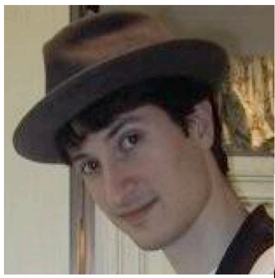
jdj.mit.edu/book

Confession: I've used Python's internal C API more than I've coded in Python…

A new programming language?

Viral Shah

Jeff Bezanson



Alan Edelman

Stefan Karpinski

[17+ developers with 100+ commits]

julia

julialang.org

[begun 2009, “0.1” in 2013, ~20k commits]

First reaction: **You’re ~~doomed~~.** [usual fate of all new languages]

… *subsequently*: **… ~~probably doomed~~**

… still might be doomed

but, in the meantime,

I'm having fun with it...

... and it solves a real problem
with technical computing
in high-level languages.

The "Two-Language" Problem

Want a high-level language that you can work with interactively
= easy development, prototyping, exploration
⇒ dynamically typed language

Plenty to choose from: Python, Matlab / Octave, R, Scilab, …

(& some of us even like Scheme / Guile)

Historically, can't write performance-critical code ("inner loops")
in these languages… have to switch to C/Fortran/… (static).
[e.g. SciPy git master is ~70% C/C++/Fortran]

Workable, but Python → Python+C = a huge jump in complexity.

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 may be you.
  — some problems are impossible or
  just very awkward to vectorize.

Dynamic languages don't have to be slow.

Lots of progress in JIT compilers, driven by web applications.
& excellent free/open-source JIT via LLVM.

Javascript in modern browsers achieves C-like speed.

Many other efforts to speed up dynamic languages, e.g. PyPy, Numba / Cython (really 2nd lower-level language embedded in Python).

What if a dynamic language were designed for JIT from the beginning, with the goal of being as high-level as possible while staying within 2× C speed?

julia

(and it’s easier to call SciPy from Julia than from PyPy)

Today

A brief introduction to Julia,
its key features,
and how it gets performance.

How Julia leverages Python and IPython
to lessen the “infrastructure problem” of new languages

time permitting:
How tools can flow in the other direction too…

julia

[julialang.org]

- Dynamically typed
- Multiple dispatch: a generalization of OO
- Metaprogramming (code that writes code)
- Direct calling of C and Python functions
- Coroutines, asychronous I/O
- Designed for Unicode
- Distributed-memory parallelism
- User-defined types == builtin types … extensible promotion and conversion rules, etc.
- Large built-in library: regex, linear algebra, special functions, integration, etcetera…
- git-based package manager

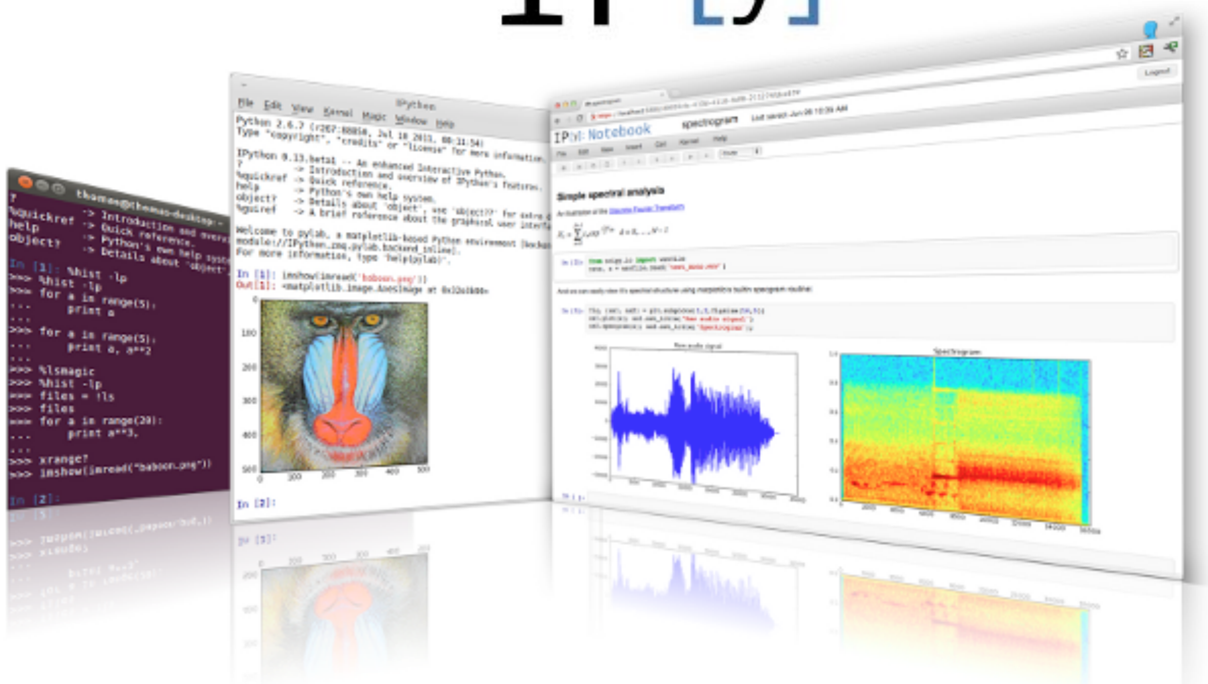
Most of Julia (70%+) is written in Julia.

`goto` live Julia REPL demo…

This command line is so 1990...

[e.g. GNU Readline, 1989]

vs. IP[y]



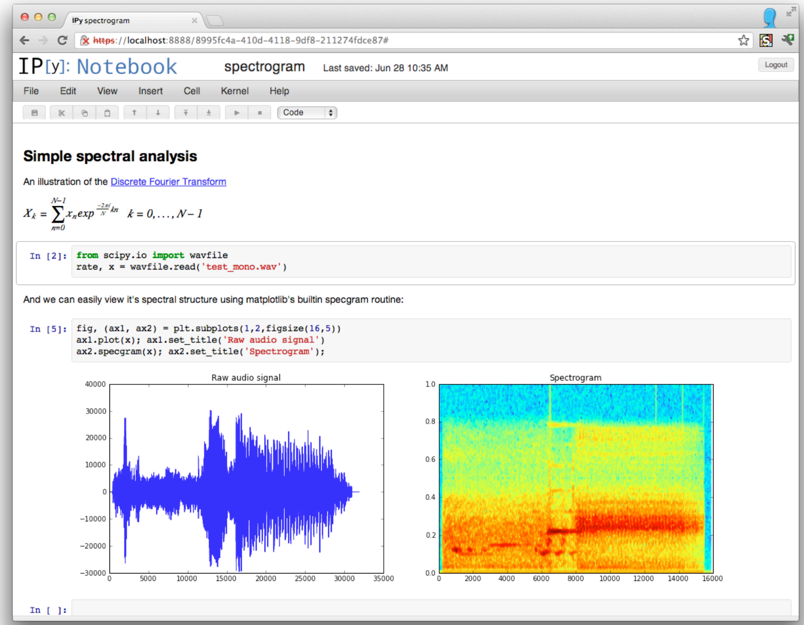
[ipython.org]

Come back in 5 years once you implement something more modern...

(roughly) How IPython Notebooks Work

programmer

enter Python code



Web browser:

Notebook Display/Interaction

HTTP

code

HTML + JS

IPython Web Server

ZeroMQ

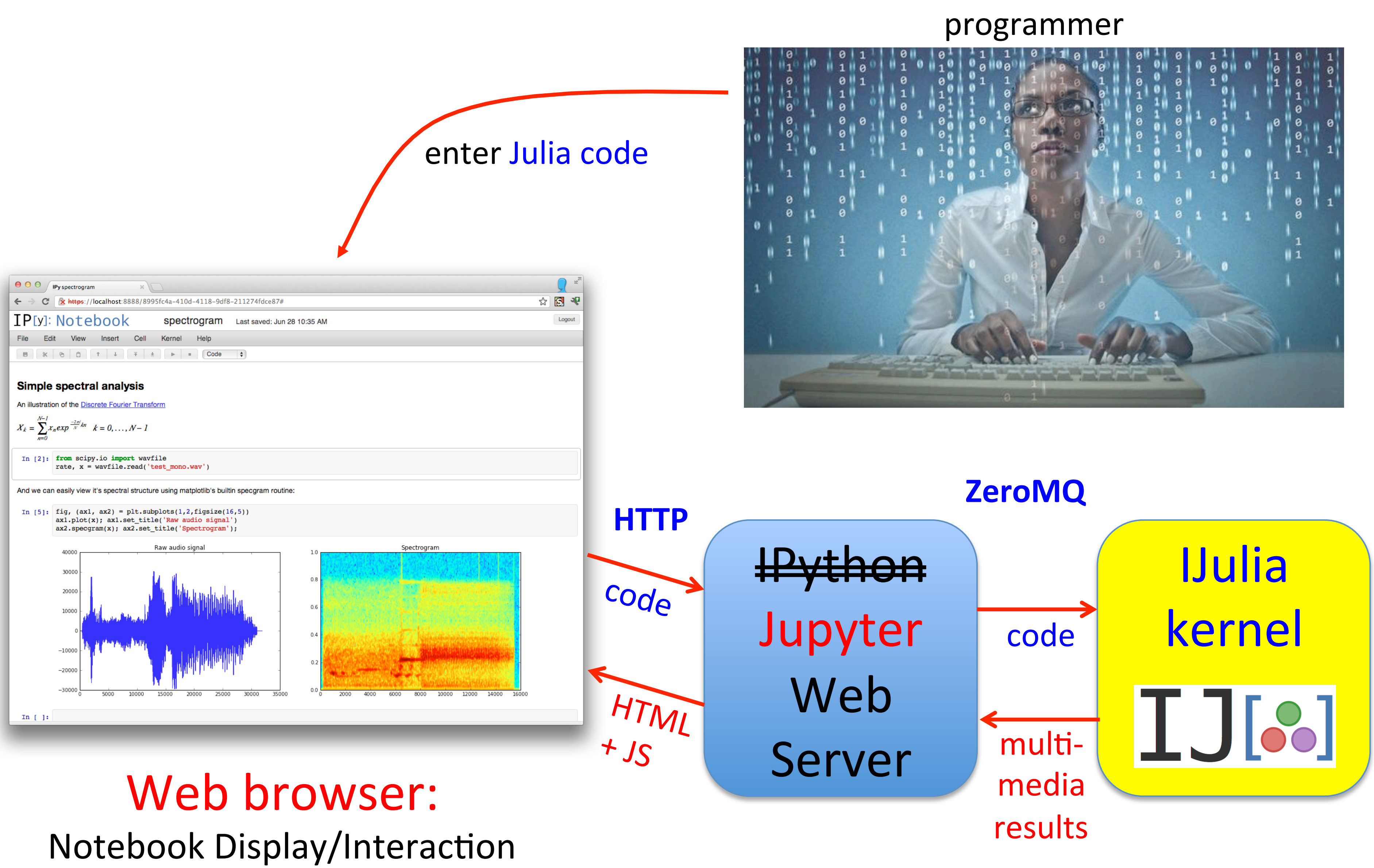
code

multi-media results

IPython kernel

IP[y]

How IJulia Notebooks Work



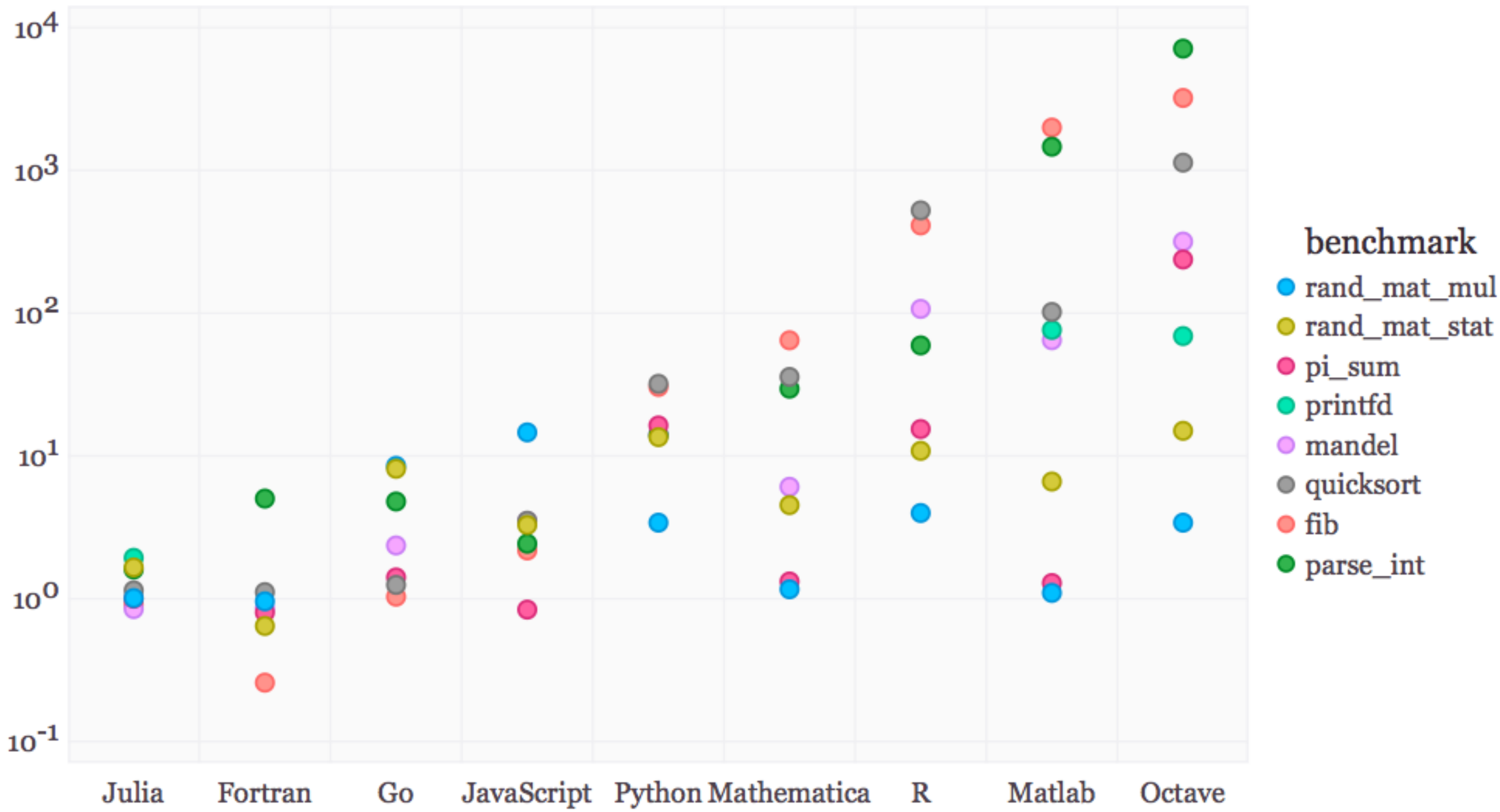
`goto` live IJulia notebook demo…

Why is Julia fast?

~~Why~~ is Julia fast?

Julia performance on synthetic benchmarks

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



What about real problems,
compared to highly optimized code?

Special Functions in Julia

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

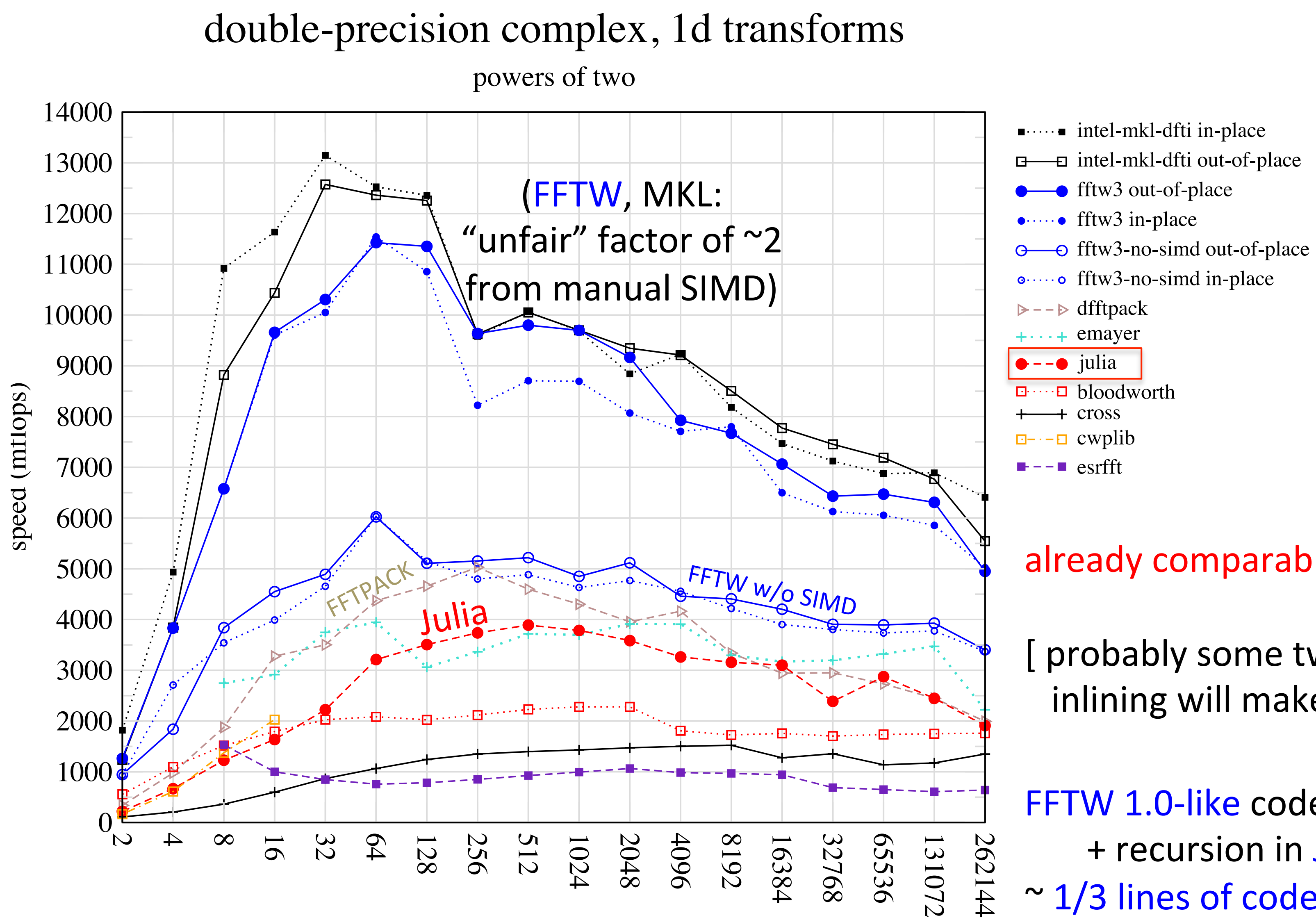
Many of Julia's special functions come from the usual C/Fortran libraries, but some are written in pure Julia code.

Pure Julia erfinv(x) [= $\mathrm{erf}^{-1}(x)$]
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/ code generation] that are hard in a low-level language.

Pure-Julia FFT performance



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

Why is Julia fast?

~~Why is Julia fast?~~

Why can Julia be fast?

(You can write slow code in any language, of course.)

… and couldn't Python do the same thing?

Type Inference

To generate fast code for a function f(x,y), the compiler needs to be able to infer the types of variables in f, map them to hardware types (registers) where possible, and call specialized code paths for those types (e.g. you want to inline +, but this depends on types).

At compile-time, the compiler generally only knows types of x,y, not values, and it needs to be able to cascade this information to infer types throughout f and in any functions called by f.

Julia and its standard library are designed so type inference is possible for code following straightforward rules.

… sometimes this requires subtle choices that would be painful to retrofit onto an existing language.

Type Stability

Key to predictable, understandable type inference:

- the type of function's return value should only depend on the types of its arguments

A counter-example in Matlab and GNU Octave:

sqrt(1) == 1.0 — real floating-point
sqrt(–1) == 0.0+1.0i — complex floating-point

Hence, any non-vector code that calls sqrt(x) in Matlab cannot be compiled to fast code even if x is known to be real scalar — anything "touched" by the sqrt(x) is "poisoned" with an unknown type — unless the compiler knows x ≥ 0.

Better to throw an exception for sqrt(–1), requiring sqrt(-1+0i).

Type Stability

Key to predictable, understandable type-inference:

- the type of function's return value
  should only depend on the types of its arguments

Common counter-examples in Python

Typical idiom:

foo(x) returns y, or None if [exceptional condition]

[e.g. numpy.ma.notmasked_edges, scipy.constants.find, …]

Better: Throw an exception.

Type Stability

Key to predictable, understandable type-inference:

- the type of function's return value
  should only depend on the types of its arguments

A counter-example in Python

integer arithmetic

Integer arithmetic in Python automatically uses bignums
to prevent overflow. Unless the compiler can detect that
overflow is impossible [which may be detectable sometimes!],
integers can't be compiled to integer registers & hw arithmetic.

Julia tradeoff: default integers are 64-bit, overflow possible
… use explicit BigInt type if you are doing number theory.

`goto` live IJulia notebook demo…

Julia: fun, fast, and
you don’t lose your Python stuff.

New languages are always a risk…

…but maybe not doomed?

Acknowledgements

julialang.org

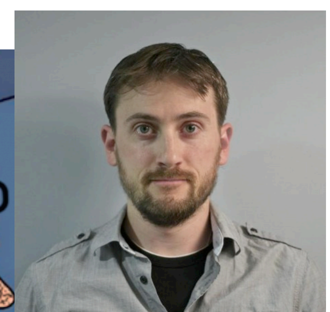
Julia core team:

- Jeff Bezanson (MIT)
- Stefan Karpinski (MIT)
- Viral Shah
- *…(17+ developers with 100+ commits)…*
- Prof. Alan Edelman (MIT)

ipython.org

Fernando Perez

Matthias Bussonier



Min RK

& Shashi Gowda (GSoC)

& Jake Bolewski (pyjulia)