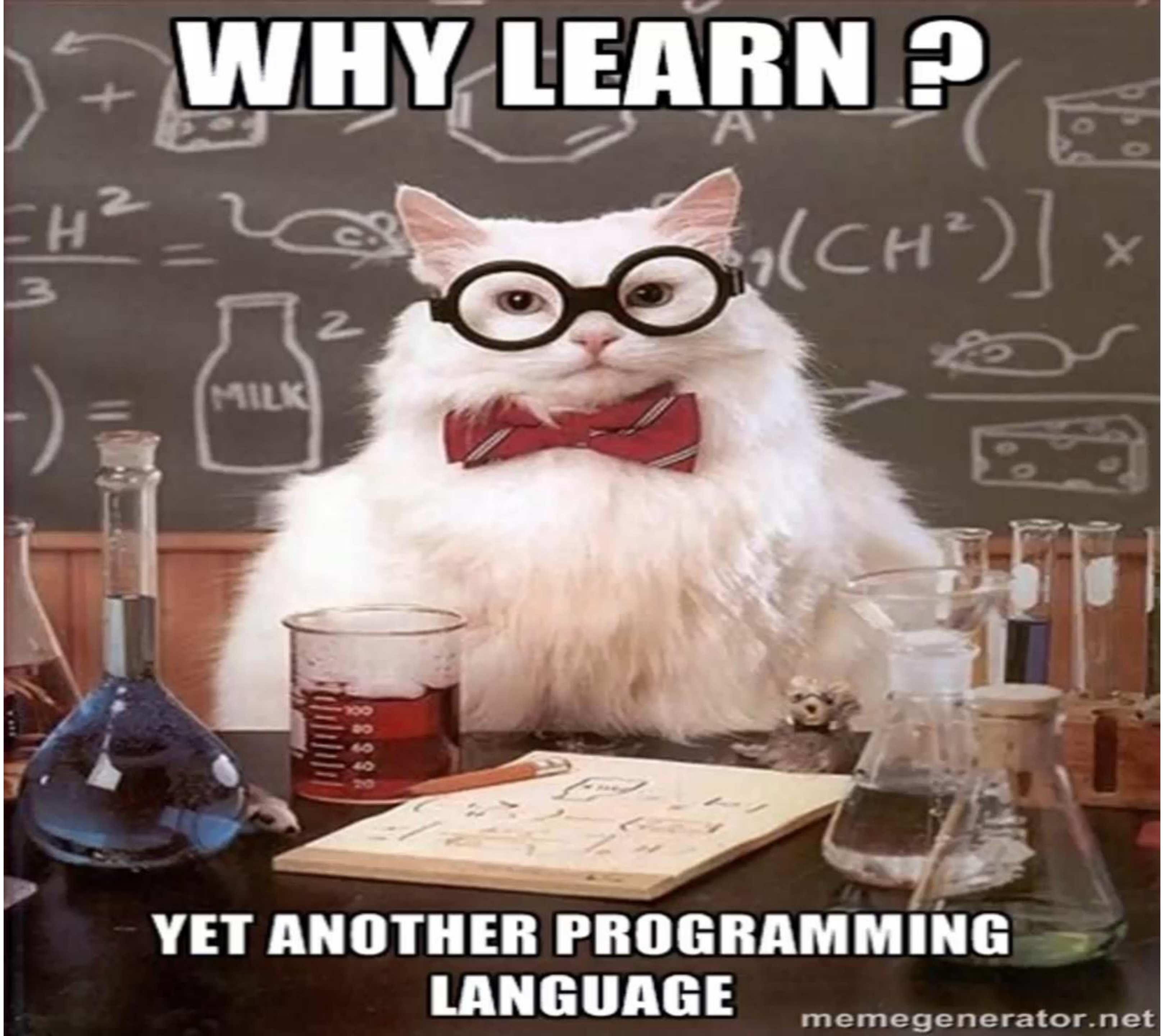
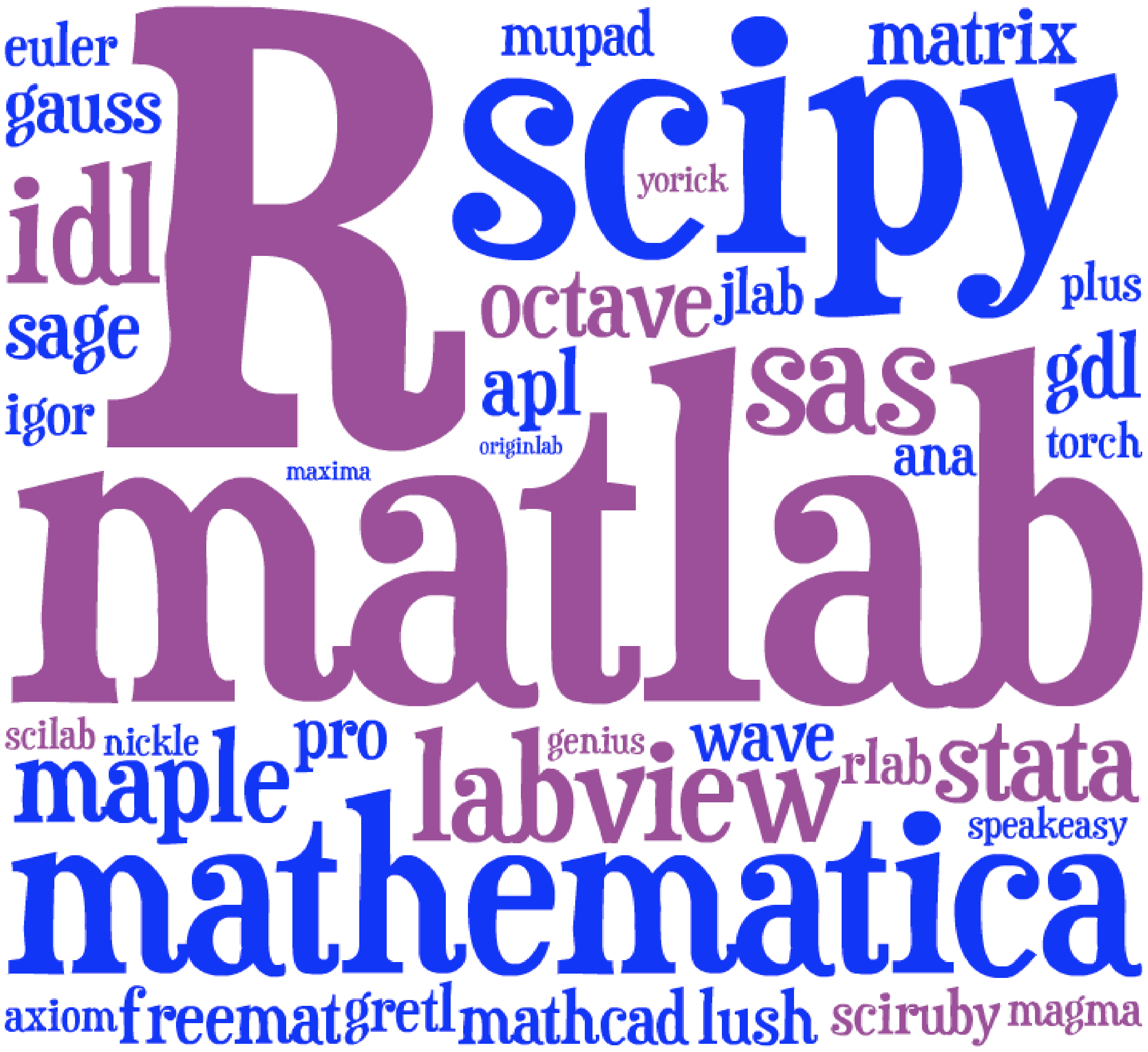
Julia: A New Language for Technical Computing

Presented by

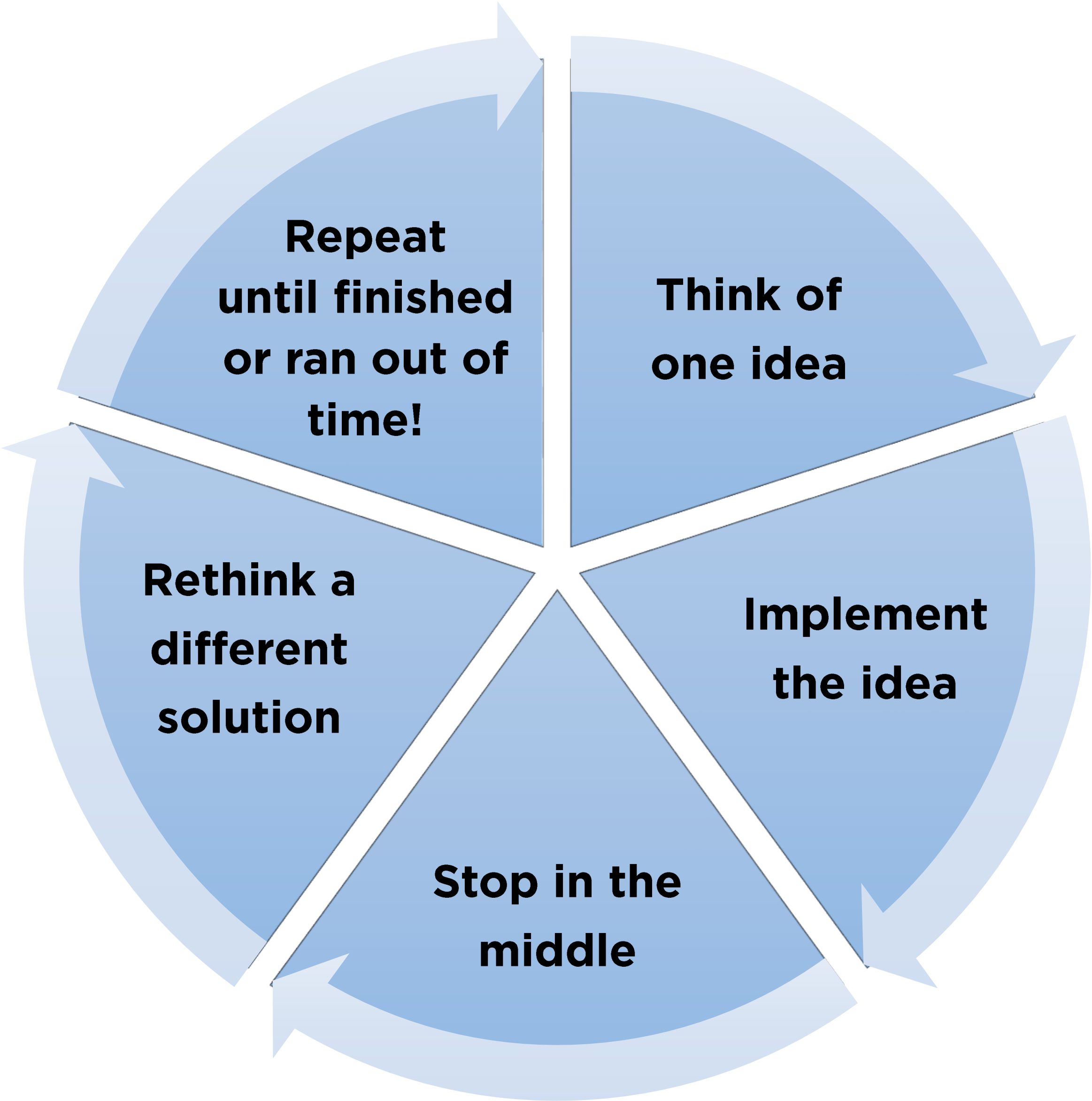
Dr. Seif Shebl
Electrical Engineering Department


WHY LEARN ?
MILK
YET ANOTHER PROGRAMMING
LANGUAGE
memegenerator.net

Why do we need one more?



Typical scenario for researchers/scientists



Design Tradeoffs

	C	MATLAB
Execution speed		
Code development time		

The two-language problem

Non-application based development is typically interactive nowadays

MATLAB has shown us how a dynamically typed, interpreted language can be easily used to solve our problems

Rapid prototyping and low barrier to entry are important language characteristics

But what happens when you need to scale things up?

The research question?

Why is a language fast?

Better question:

What causes slowness?

The answer!

Slowness is usually attributed to

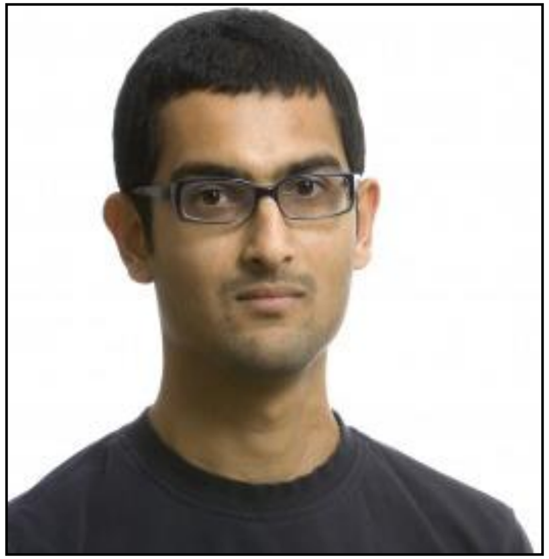
Uncertainty!

The creators of Julia ([why we created Julia?](#))

Jeff Bezanson

Stephen Karpinski

Viral Shah



Alan Edelman

What is Julia?

- A high-level, **high-performance**, **dynamic** programming language.
- **Easy syntax** similar to other technical computing environments.
- Provides a smart **compiler**, distributed **parallel** execution, numerical **accuracy**, and extensive mathematical **library**.
- Julia’s Base library is largely written in Julia itself
- Integrates mature, best-of-breed open source **C** and **Fortran** libraries for **linear algebra**, **random number** generation, **signal processing**, and **string** processing

Solving the two-language problem

Julia fills this role: it is a flexible dynamic language, suitable for scientific and numerical computing, with performance comparable to traditional statically-typed languages.

```
x = [1,2,3]
y = [1 2 3]
A = [1 2 3 4;5 6 7 8;9 10 11 12]
A[2, 1] = 0
u,v  = (15.03, 1.2e-27)
f(x) = 3x
map(x -> 3x, A)
x[2:12]
x[2:end]
A[5,1:3]
A[5,:]
```

```
for animal in ["dog", "cat", "mouse"]
    println("$animal is a mammal")
end

map(x -> x^2+2x-1, [1, 2, 3])

[add_10(i) for I in [1, 2, 3]]
```

```
... and keyword arguments too
```

Syntax that is familiar to MATLAB users

```
function randmatstat(t; n=10)
    v = zeros(t)
    w = zeros(t)
    for i = 1:t
        a = randn(n,n)
        b = randn(n,n)
        c = randn(n,n)
        d = randn(n,n)
        P = [a b c d]
        Q = [a b; c d]
        v[i] = trace((P'*P)^4)
        w[i] = trace((Q'*Q)^4)
    end
    std(v)/mean(v), std(w)/mean(w)
end
```

Keyword arguments

Familiar array syntax

Common matrix operations

Common statistics

Last expression is return value

Some noteworthy features

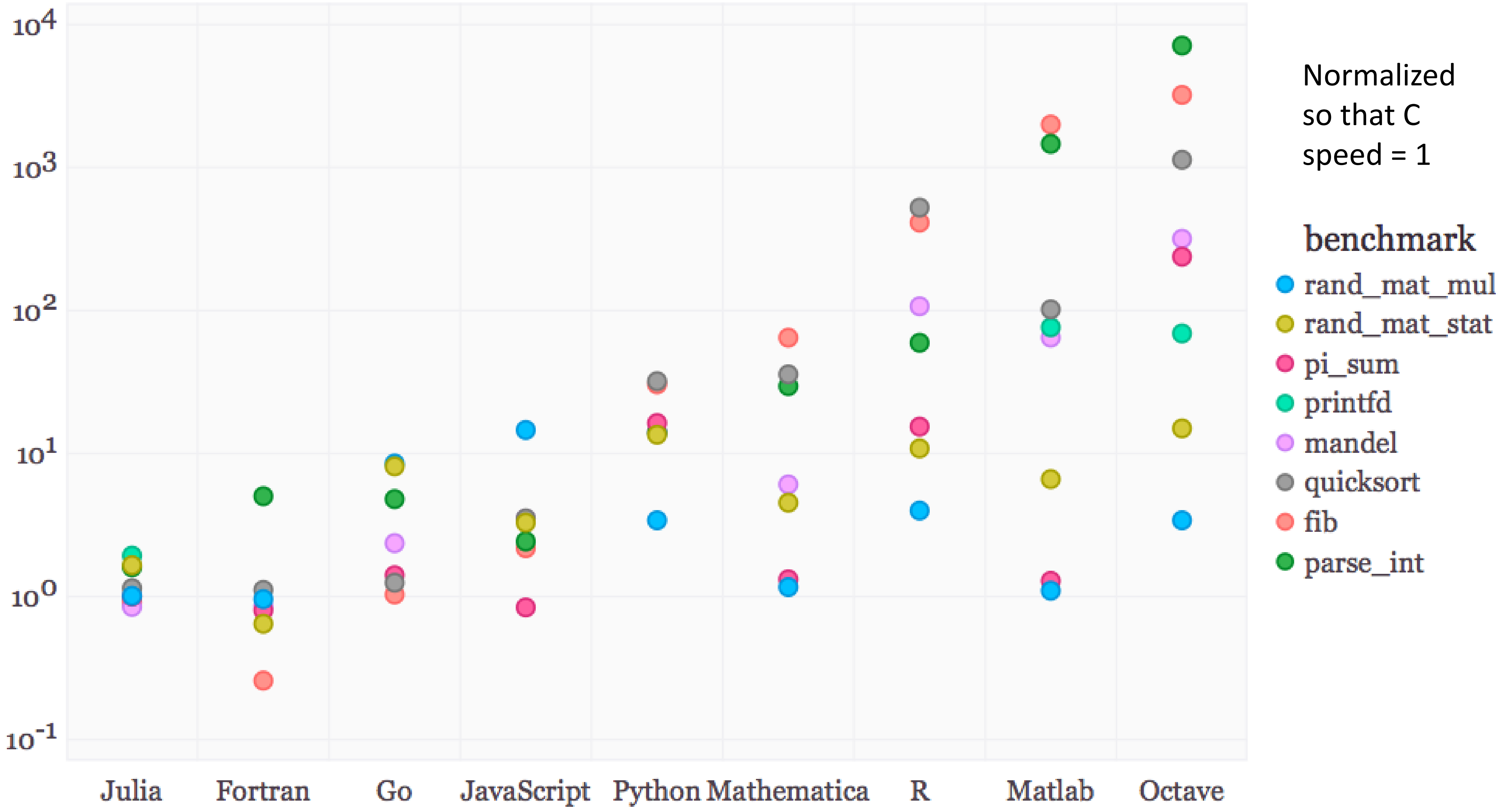
- Open source with an MIT licensed core
- Easy installation - Just download a precompiled binary and run
- Dynamically typed with fast user-defined types
- Multiple dispatch with a sophisticated parametric type system
- JIT compiler - no need to vectorize for performance
- Distributed memory parallelism
- Effortlessly call **C**, **Fortran**, **Python**, and **MATLAB**
- Unicode support

JIT advantages

JIT: Just-in-time compilation in Julia proved **faster than native** code on certain scenarios since JIT can do additional optimizations based on runtime values which a static compiler doesn't know about.

Similarly, **garbage collection** can be more performant than explicit memory cleanup on certain cases.

Performance on synthetic benchmarks

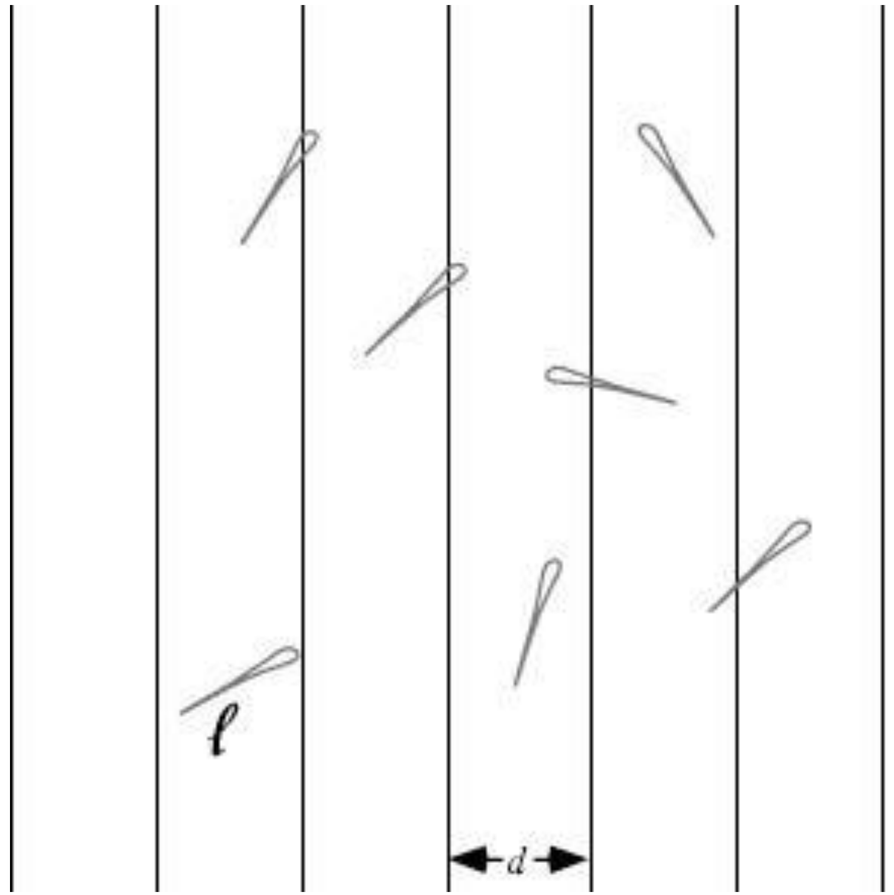


Normalized so that C speed = 1

Let's compute π : Buffon needle problem

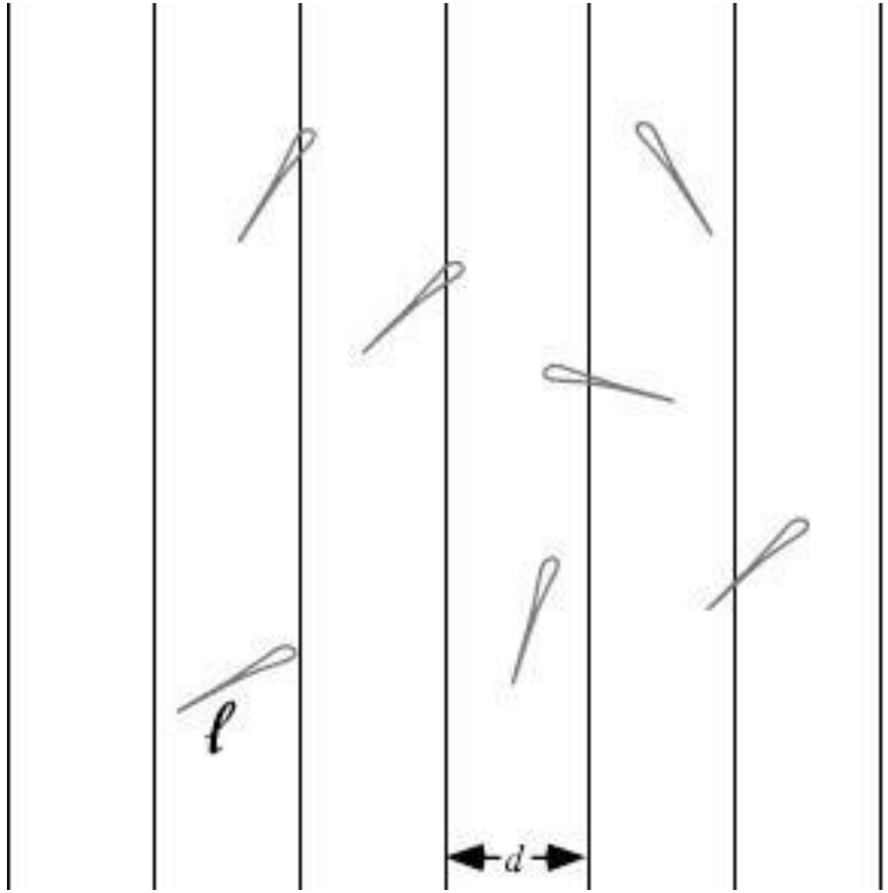
```
function buffon(m)
  hit = 0
  for l = 1:m
    mp  = rand()
    phi = (rand()* pi) - pi/2
    xrechts = mp + cos(phi)/2
    xlinks  = mp - cos(phi)/2
    if xrechts >= 1 || xlinks <= 0
        hit += 1
    end
    end
    miss = m – hit
  piapprox = m / hit * 2

end
```



Let’s compute π in parallel

```
function buffon_par(m)
  hit = @parallel (+) for l = 1:m
    mp = rand()
    phi = (rand() * pi) - pi / 2
    xrechts = mp + cos(phi)/2
    xlinks  = mp - cos(phi)/2
    (xrechts>=1 || xlinks<=0)? 1:0
  end
  miss = m – hit
  piapprox = m / hit * 2
end
```



How to run?

1. From the command window
2. From the REPL (Read-Eval-Print Loop)
3. Inside an IDE: Juno, Sublime Text, etc.
4. In the cloud: JuliaBox

Demo

Questions?

Thank you