

Introduction to scientific programming with Julia

Przemysław Szufel
<https://szufel.pl/>

Installing Julia

- Julia Open Source ← **RECOMMENDED**
 - <https://julialang.org/downloads/>
 - 1.0.3 – latest
- Julia Pro
 - <https://juliacomputing.com/products/juliapro.html>
- JuliaBox
 - <https://juliabox.com/>
 - Free, and runs in a web browser
 - OK for today's examples, NOT OK for Tuesday

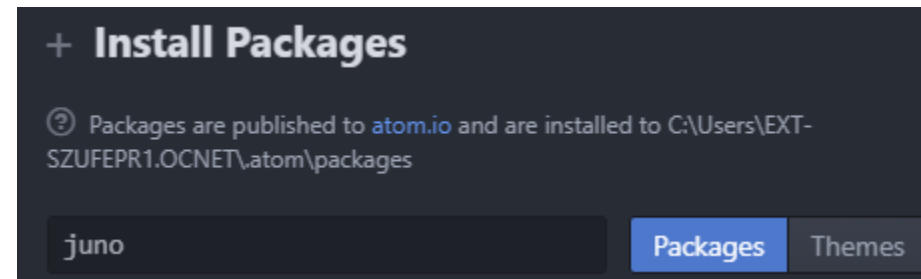
Adding Julia packages

- Start Julia REPL
- Press **]** to start the Julia package manager
(prompt `(v1.0) pkg>` will be seen)
- Sample package installation command

```
(v1.0) pkg> add PyPlot DataFrames Distributions
```

Julia IDEs

- Atom with Juno plugin
 - Go to <https://atom.io/download/>
 - Run Atom and press `Ctrl + ,` (Ctrl and “comma”)
 - In the search box type “juno”



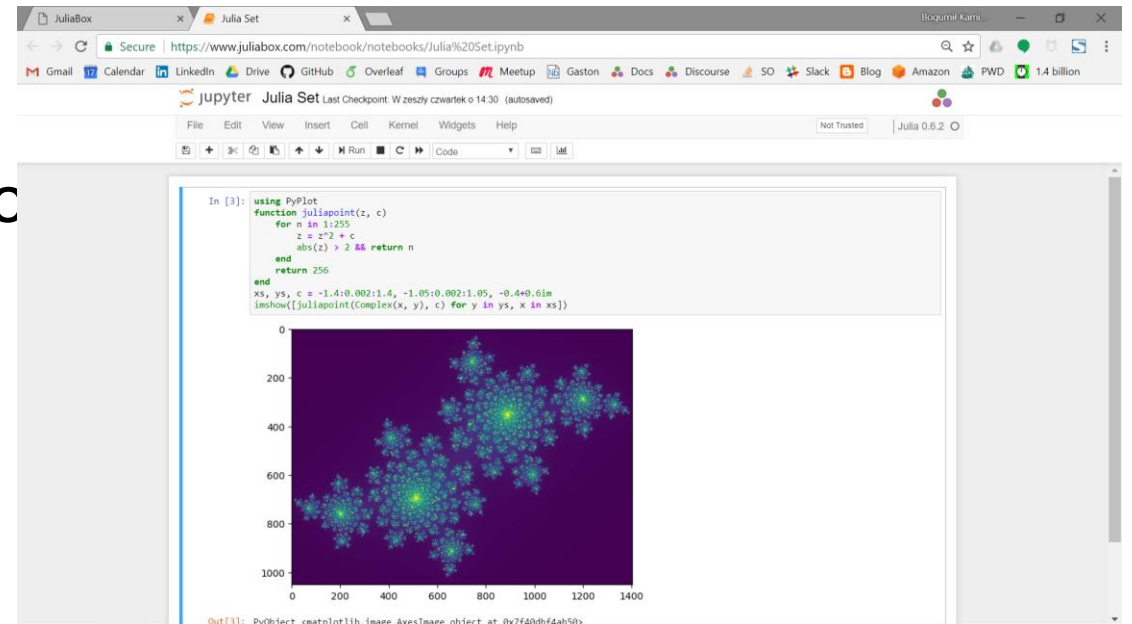
- Press the “Install” button to install Juno



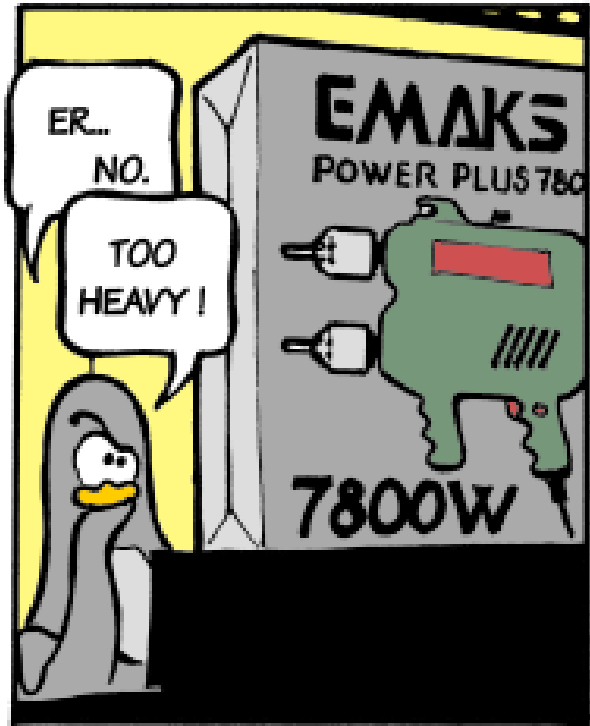
This will install Juno Atom plug-in along with its dependencies

Jupyter notebook and Julia online...

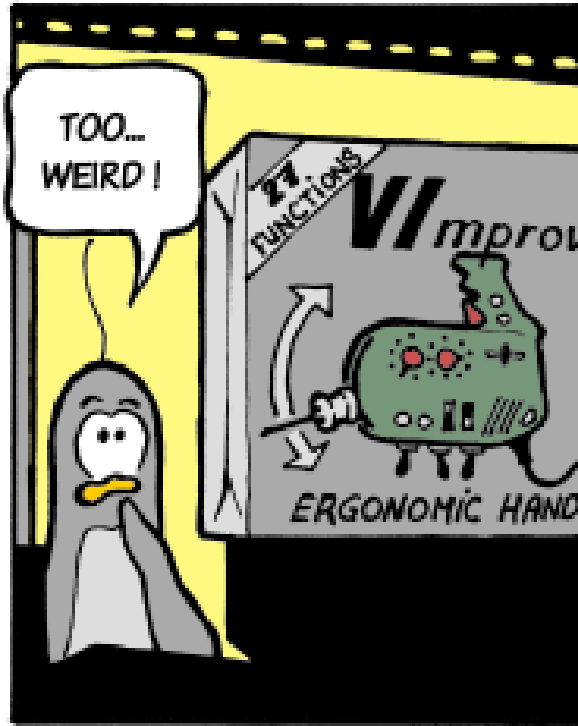
- Alternative: Jupyter notebook
 - `julia> Pkg.add("IJulia")`
 - `julia> using IJulia`
 - `julia> notebook()`
- JuliaBox – Julia on-line, no installation (pure cloud <https://juliabox.com>)



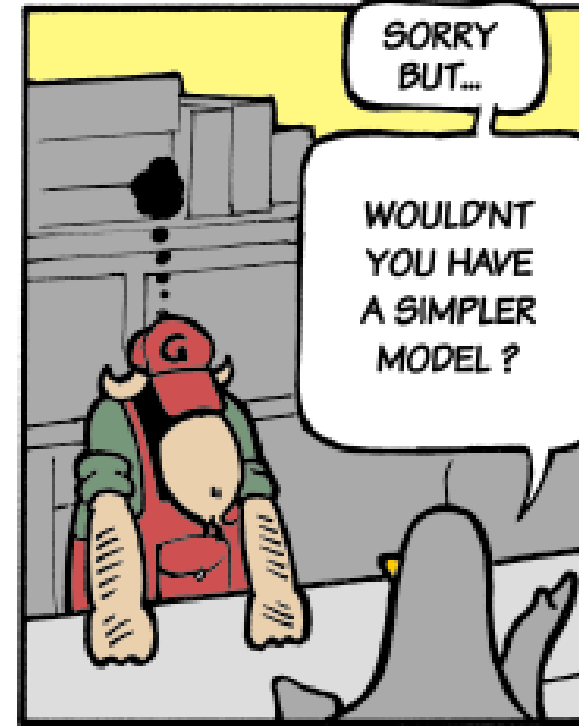
Scala

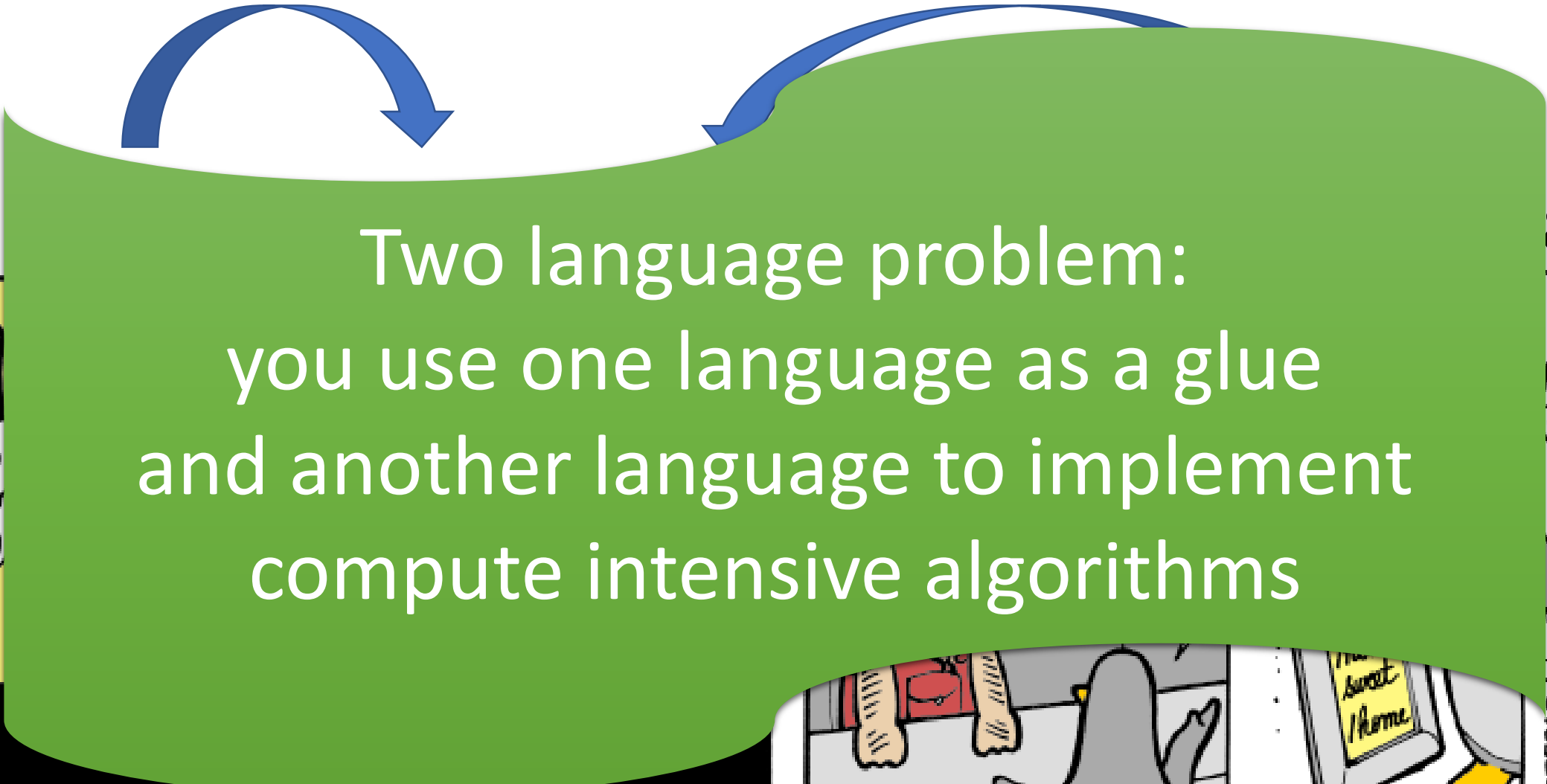


C++



Visual Basic





Two language problem:
you use one language as a glue
and another language to implement
compute intensive algorithms



asic

DID I
REALLY
PAY FOR
THAT ???

Methods of achieving high performance in different environments

Ecosystem	Glue	Hot code
R-based	R	RCpp
Python-based	Python	Numba/Cython
Julia-based	Julia	Julia
NetLogo	R Behaviorspace	Java

Methods of achieving high performance in different environments

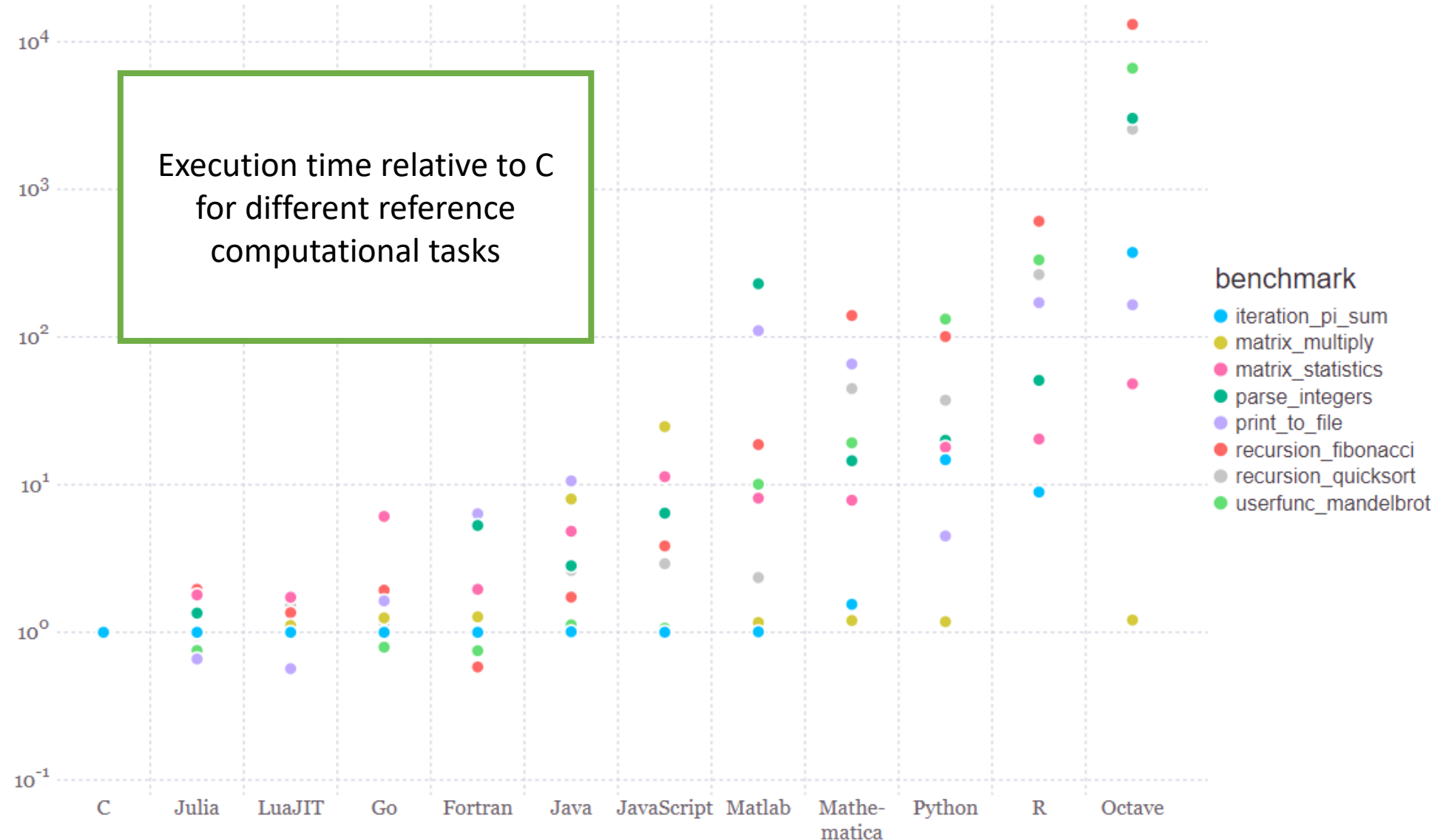
The 'Rcpp' package provides R functions as well as C++ classes which offer a seamless integration of R and C++.

Ecosystem	Glue	Hot code
R-based	R	RCpp
Python-based	Python	Numba/Cython
Julia-based	Julia	Julia
NetLogo	?	Java

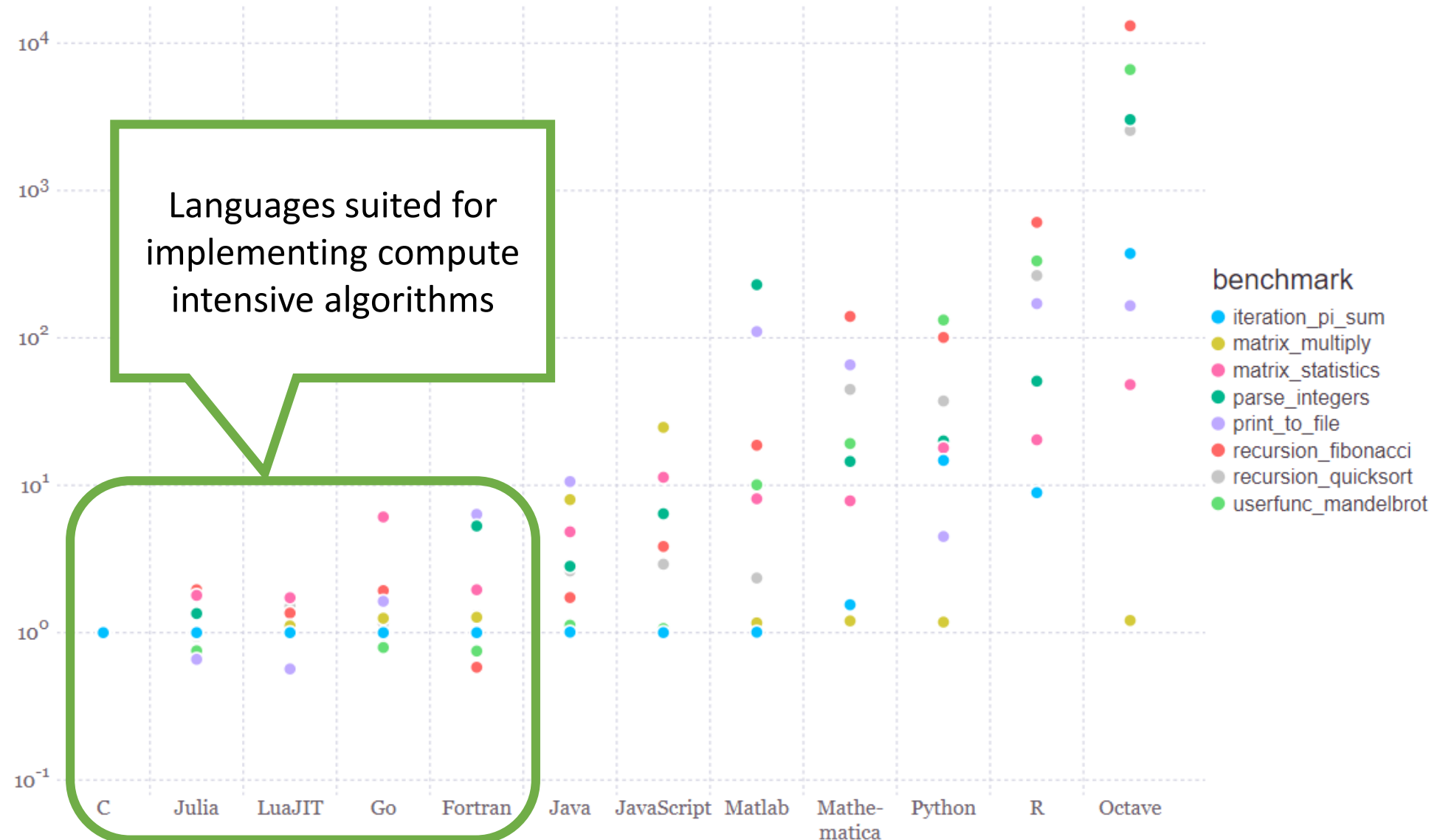
Numba gives you the power to speed up your applications with high performance functions written directly in Python (subset supported).

Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language

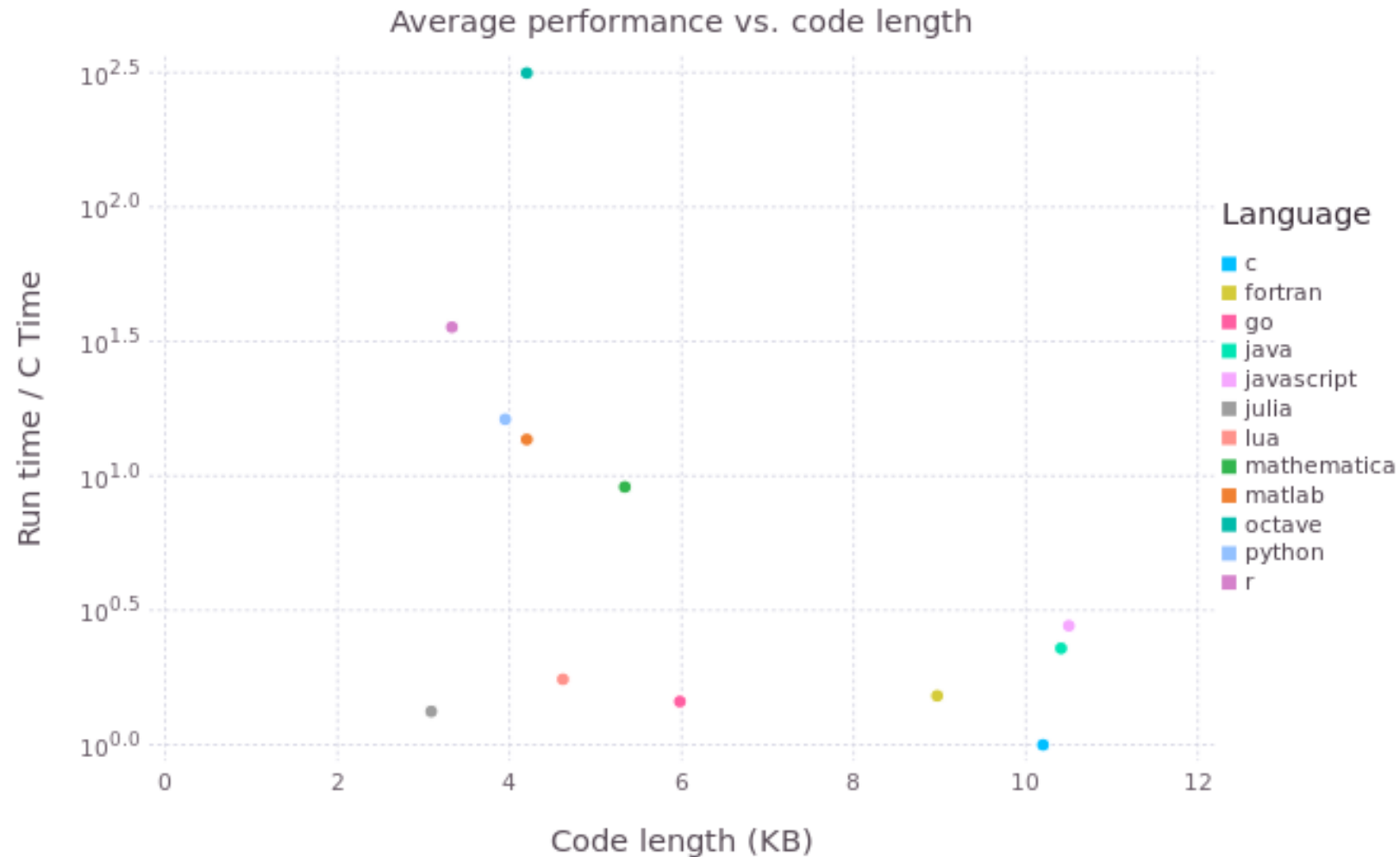
Reference benchmarks from Julia website



Reference benchmarks from Julia website

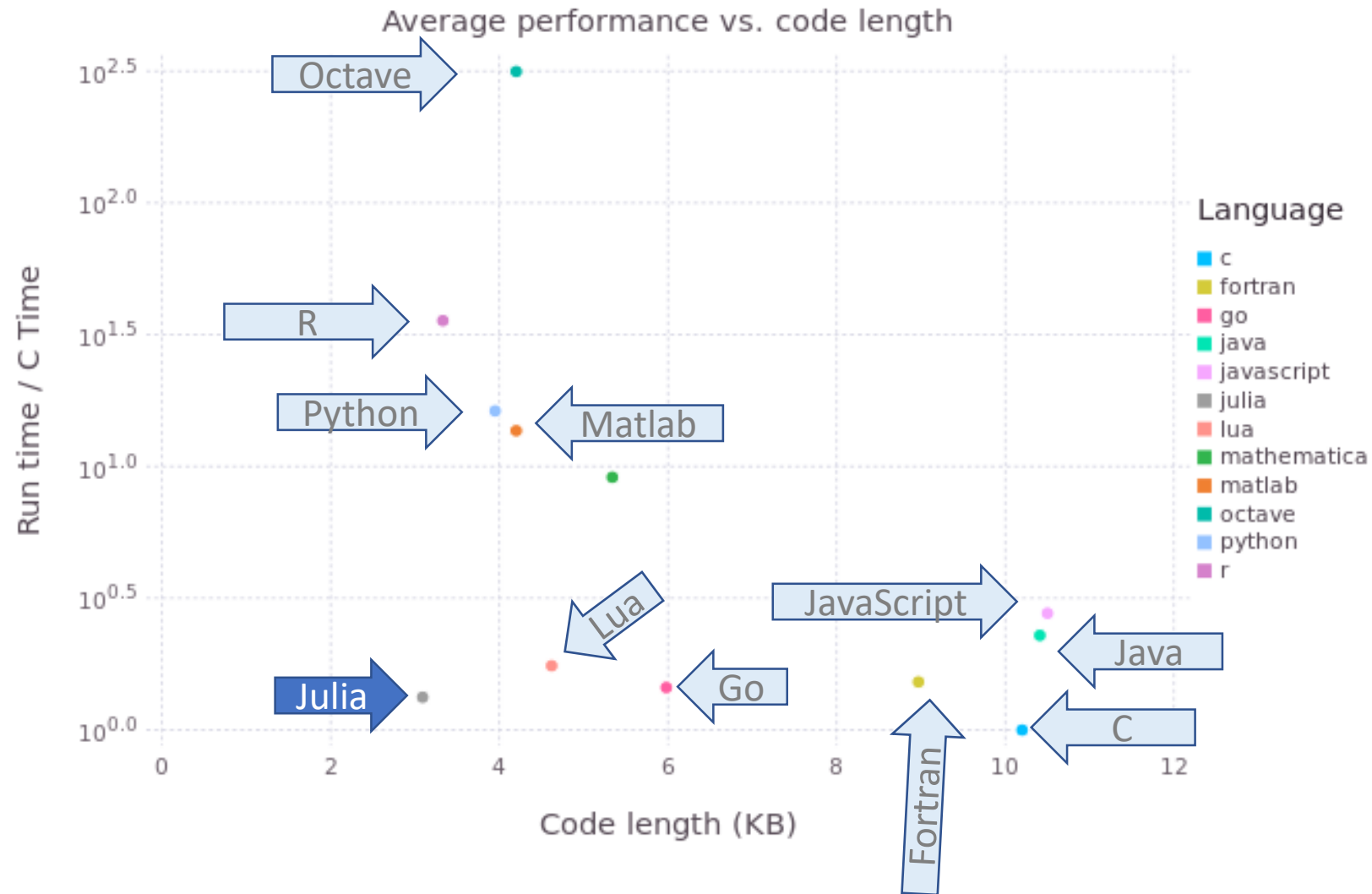


Language Code Complexity vs Execution Speed



Source: <http://www.oceanographerschoice.com/2016/03/the-julia-language-is-the-way-of-the-future/>

Language Code Complexity vs Execution Speed



Source: <http://www.oceanographerschoice.com/2016/03/the-julia-language-is-the-way-of-the-future/>

Example simple problem

Simulate 90% confidence of correlation coefficient of two random uniform vector of length 1000.

```
# Julia
function cord(n, rep)
    x, y = zeros(n), zeros(n)
    quantile([cor(rand!(x), rand!(y)) for i in 1:rep], [0.05, 0.95])
end
```

```
# R
cord <- function(n, rep) {
    quantile(replicate(rep, cor(runif(n), runif(n))), c(0.05, 0.95))
}
```

Example simple problem

Simulate 90% confidence of correlation coefficient of two random uniform vector of length 1000.

Julia

```
function cord(n, rep)
    x, y = zeros(n), zeros(n)
    quantile([cor(rand!(x), rand!(y
end
```

```
julia> @time cord(1000, 10^6)
4.107035 seconds (10.43 k allocations:
15.824 MiB, 0.31% gc time)
2-element Array{Float64,1}:
-0.0519554
 0.0521062
```

R

```
cord <- function(n, rep) {
    quantile(replicate(rep, cor(run
}
```

```
> system.time(cord(1000, 10^6))
   user  system elapsed 
139.19    0.25   142.02
```

Example simple problem

Simulate 90% confidence of correlation coefficient of two random uniform vector of length 1000.

Julia

```
function cord(n, rep)
    x, y = zeros(n), zeros(n)
    quantile([cor(rand!(x), rand!(y
end
```

```
julia> @time cord(1000, 10^6)
4.107035 seconds (10.43 k allocations:
15.824 MiB, 0.31% gc time)
2-element Array{Float64,1}:
-0.0519554
 0.0521062
```

In R you could call C++ via RCpp
to have the same performance
(but then you have to code that C++
which is harder....)

```
> system.time(cord(1000, 10^6))
   user  system elapsed 
139.19    0.25   142.02
```


The promise of Julia

Julia Joins Petaflop Club

September 12, 2017

BERKELEY, Calif., Sept. 12, 2017 — Julia has joined the rarefied ranks of computing languages that have achieved peak performance exceeding one petaflop per second – the so-called ‘Petaflop Club.’

The Julia application that achieved this milestone is called [Celeste](#). It was developed by a team of astronomers, physicists, computer engineers and statisticians from UC Berkeley, Lawrence Berkeley National Laboratory, National Energy Research Scientific Computing Center (NERSC), Intel, Julia Computing and the Julia Lab at MIT.

Celeste uses the Sloan Digital Sky Survey (SDSS), a dataset of astronomical images from the Apache Point Observatory in New Mexico that includes every visible object from over 35% of the sky – hundreds of millions of stars and galaxies. Light from the most distant of these galaxies has been traveling for billions of years and lets us see how the universe appeared in the distant past.

The promise of Julia

Julia Joins Petaflop

September 12, 2017

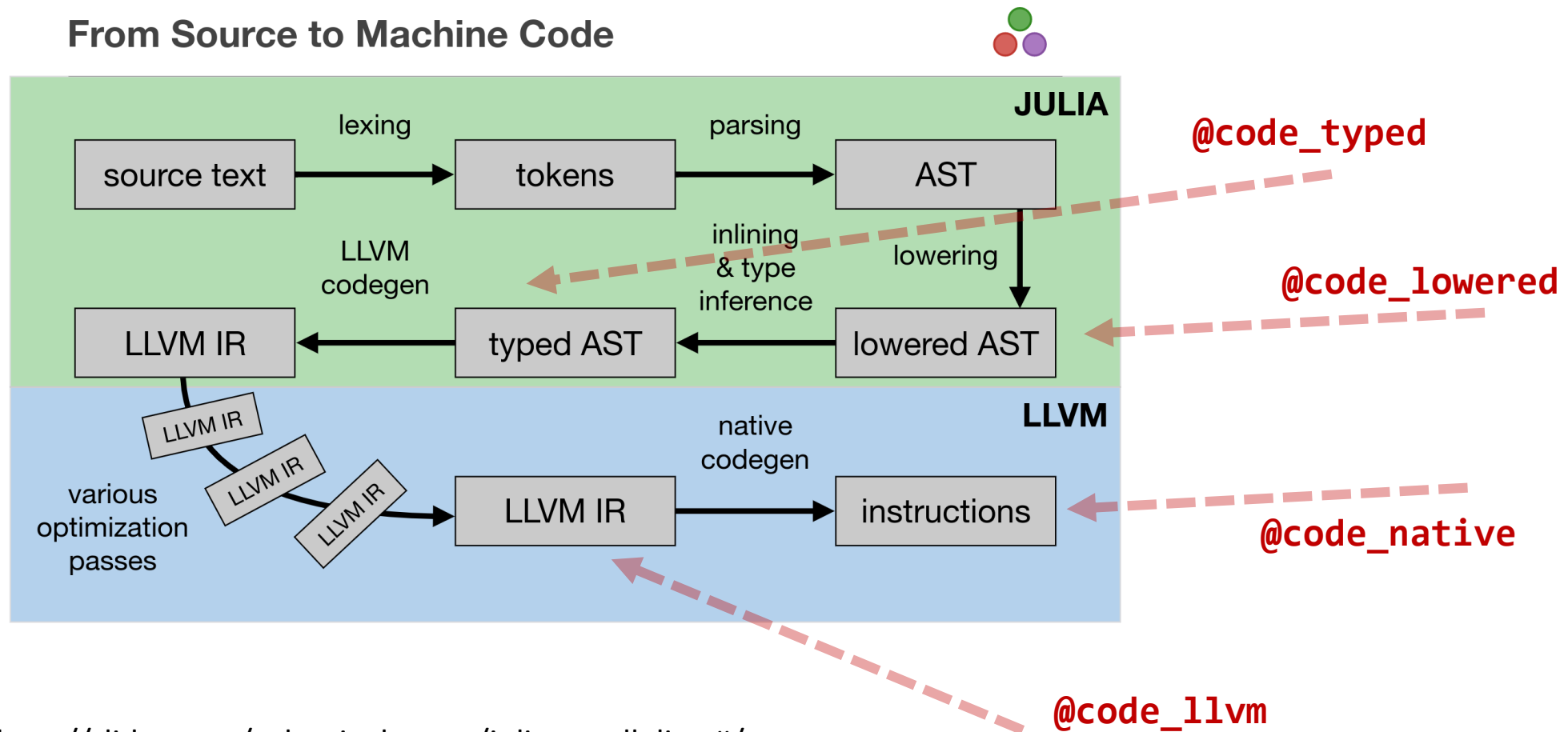
Julia is the only “scripting flavor”
language designed to support such
large deployments

...observatory in New Mexico that
...35% of the sky – hundreds of millions of stars and
...the most distant of these galaxies has been traveling for
...of years and lets us see how the universe appeared in the distant past.

Key features

- Performance
 - Dynamically compiled to optimized native machine code
- Scalability
 - SIMD, Threading, Distributed computing
- Modern design of the language
 - multiple dispatch, metaprogramming, type system
- MIT License
 - corporate-use friendly (also package ecosystem)

Julia code compilation process



Julia @ Aviva Solvency II

- 93% reduction of # lines of code (~14,000 → ~1,000)
 - ↓ development time
 - ↓ testing complexity
 - ↓ team management effort
 - ↓ maintenance cost
- 95% reduction of required servers (100 → 5)
 - ↓ hardware and software cost
 - ↓ Infrastructure management effort
- lower license footprint

<https://youtu.be/P9lF6he8Ed8>

Julia @ Ministry of Health, Poland

Optimal allocation of accelerators in radiotherapy

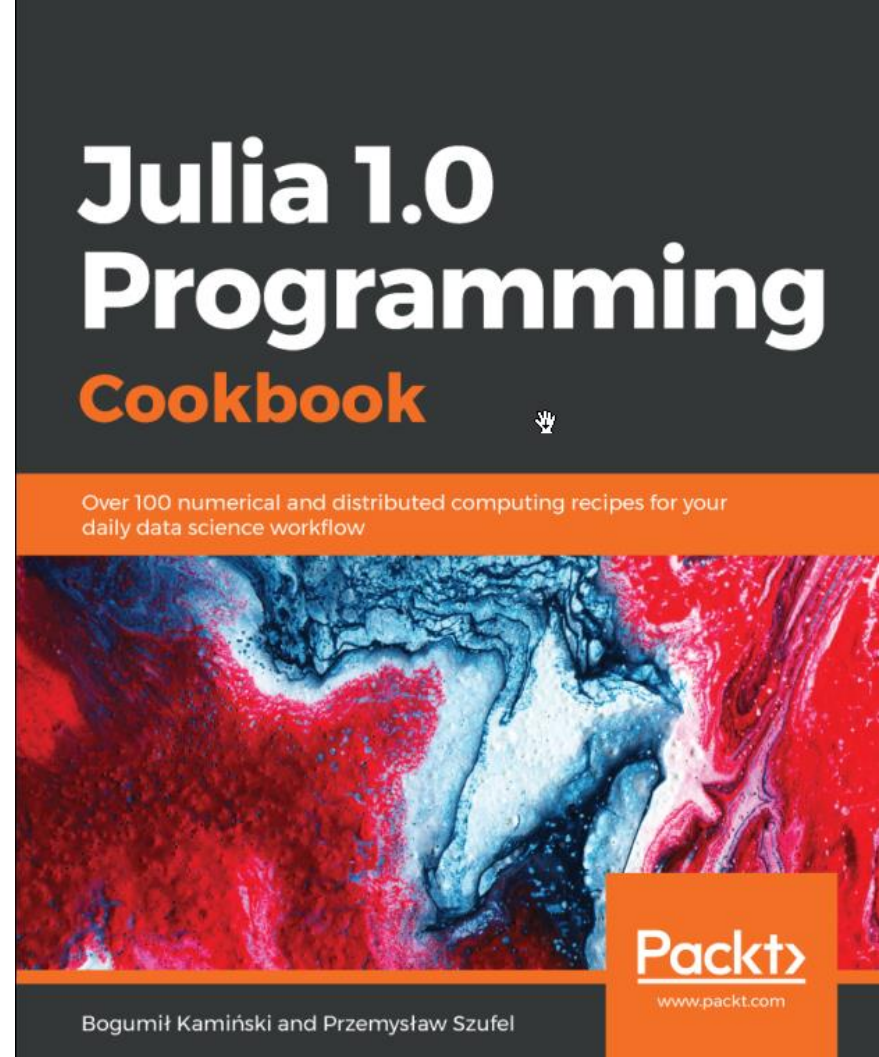
```
using JuMP, Cbc

distances = readcsv("distances.txt", Float64)
patients = readcsv("patients.txt", Float64)
N, L = size(distances)

m = Model(solver=CbcSolver())
@variable(m, y[1:L] >= 0, Int)
@variable(m, x[1:N, 1:L] >= 0)
@objective(m, Min, sum(x[i,j]*distances[i,j] for i=1:N for j=1:L))
for i in 1:N
    @constraint(m, sum(x[i,:]) >= patients[i])
end
for j in 1:L
    @constraint(m, sum(x[:,j]) <= y[j]*450)
end
@constraint(m, sum(y) <= ceil(sum(patients)/450))
solve(m)
```

Learning more about Julia

- Website: <https://julialang.org/>
- Learning materials: <https://julialang.org/learning/>
- Blogs about Julia: <https://www.juliabloggers.com/>
- <https://github.com/bkamins/The-Julia-Express>
- Julia forum: <https://discourse.julialang.org/>
- Q&A for Julia: <https://stackoverflow.com/questions/tagged/julia-lang>



> julia



Documentation: <https://docs.julialang.org>

Type "?" for help, "]?" for Pkg help.

Version 1.0.3 (2018-12-18)

Official <https://julialang.org/> release

julia> versioninfo()

Julia Version 1.0.3

Commit 099e826241 (2018-12-18 01:34 UTC)

Platform Info:

OS: Windows (x86_64-w64-mingw32)

CPU: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz

WORD_SIZE: 64

LIBM: libopenlibm

LLVM: libLLVM-6.0.0 (ORCJIT, skylake)

julia>

Basic Julia commands

(also see <https://github.com/bkamins/The-Julia-Express>)

```
apropos("apropos")  
    # search documentation for "apropos" string
```

```
@less(max(1,2))  
    # show the definition of max function  
    when invoked with arguments 1 and 2
```

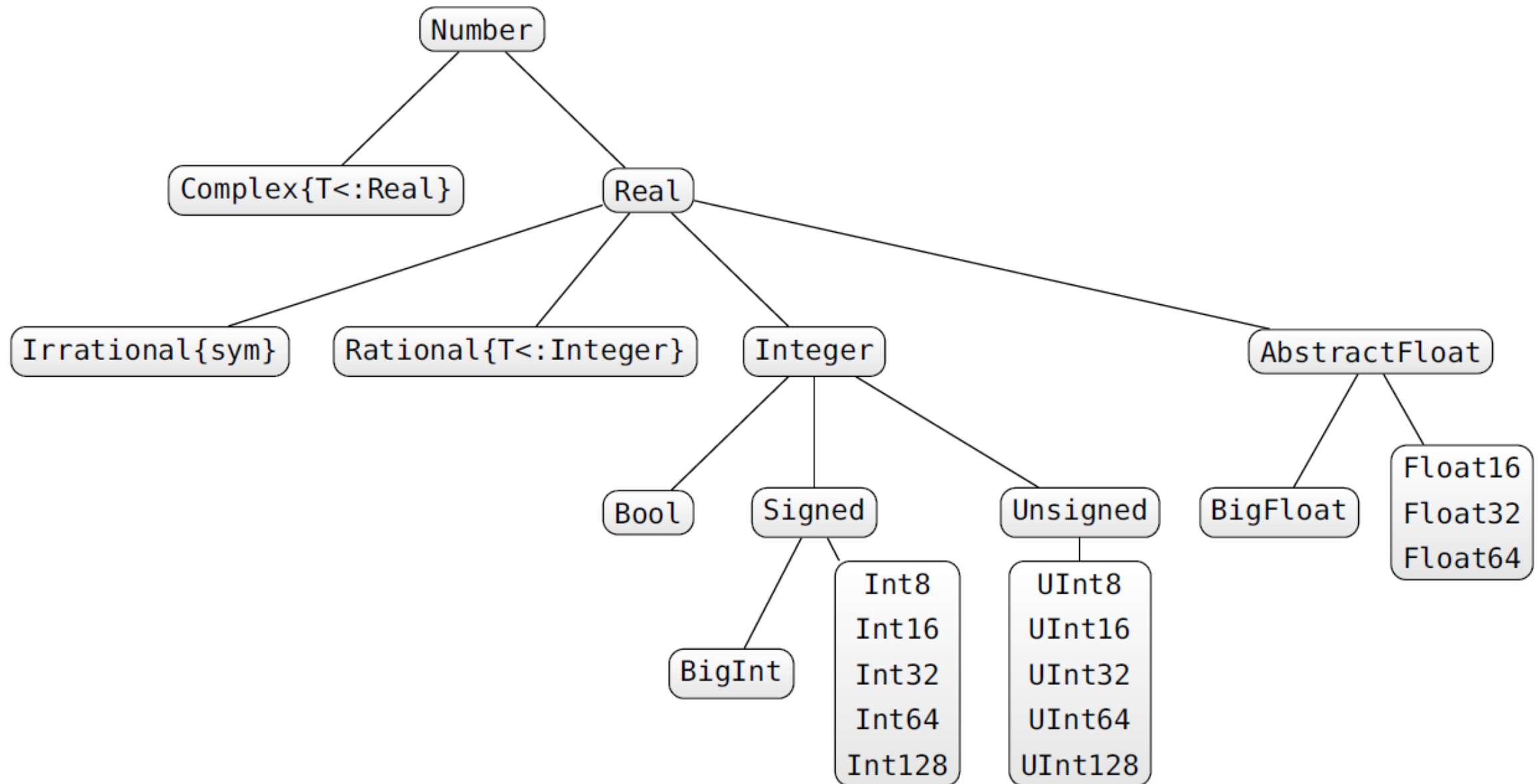
```
cd("D:/") # change working directory to D:/ (on Windows)
```

```
pwd() # get current working directory
```

```
include("file.jl") # execute source file
```

```
exit() # exit Julia
```

Numeric type hierarchy



Type conversion functions

- `Int64('a')` *# character to integer*
- `Int64(2.0)` *# float to integer*
- `Int64(1.3)` *# inexact error*
- `Int64("a")` *# error no conversion possible*
- `Float64(1)` *# integer to float*
- `Bool(1)` *# converts to boolean true*
- `Bool(0)` *# converts to boolean false*
- `Char(89)` *# integer to char*
- `zero(10.0)` *# zero of type of 10.0*
- `one(Int64)` *# one of type Int64*
- `convert(Int64, 1.0)` *# convert float to integer*
- `parse(Int64, "1")` *# parse "1" string as Int64*

Special types

- `Any` # *all objects are of this type*
- `Union{}` # *subtype of all types, no object can have this type*
- `Nothing` # *type indicating nothing, subtype of Any*
- `nothing` # *only instance of Nothing*

Tuples (similar to Python)

- `()` # *empty tuple*
- `(1,)` # *one element tuple*
- `("a", 1)` # *two element tuple*
- `('a', false)::Tuple{Char, Bool}` # *tuple type assertion*
- `x = (1, 2, 3)`
- `x[1]` # *1 (element)*
- `x[1:2]` # *(1, 2) (tuple)*
- `x[4]` # *bounds error*
- `x[1] = 1` # *error - tuple is not mutable*
- `a, b = x` # *tuple unpacking a==1, b==2*

Arrays

```
Array{Char}(undef, 2, 3, 4)      # 2x3x4 array of Chars
Array{Any}(undef, 2, 3)          # 2x3 array of Any
zeros(5)                         # vector of Float64 zeros
ones{Int64}(2, 1)                # 2x1 array of Int64 ones
trues(3), falses(3)             # tuple of vector of trues and of falses
Matrix{Float64}(I, 3, 3)        # 3x3 Float64 identity matrix
x = range(1, stop=2, length=5)
    # iterator having 5 equally spaced elements
collect(x)                      # converts iterator to vector
1:10                            # iterable from 1 to 10
1:2:10                          # iterable from 1 to 9 with 2 skip
reshape(1:12, 3, 4)             # 3x4 array filled with 1:12 values
```

Selecting rows and columns from an array

- `a = reshape(1:12, 3, 4)`
- `a[:, 1:2]` *# 3x2 matrix*
- `a[:, 1]` *# 3 element vector*
- `a[1, :]` *# 4 element vector*

Note: all arrays are 1-based (not zero based like in Python)

Data Structures

```
mutable struct Point
    x::Int64
    y::Float64
    meta
end
p = Point(0, 0.0, "Origin")
println(p.x)                # access field
p.meta = 2                  # change field value
fieldnames(typeof(p))       # get names of instance fields
fieldnames(Point)           # get names of type fields
```

Julia does not support Object Oriented Programming!

Dictionaries

```
x = Dict{Float64, Int64}()
           # empty dictionary mapping floats to integers
y = Dict("a"=>1, "b"=>2)           # filled dictionary
y["a"]                               # element retrieval
y["c"] = 3                           # added element
haskey(y, "b")                       # check if y contains key "b"
keys(y), values(y)
           # tuple of iterators returning keys and values in y
delete!(y, "b")  # delete key from a collection, see also: pop!
get(y, "c", "default")
           # return y["c"] or "default" if not haskey(y, "c")
```

Strings

```
"Hi " * "there!"           # string concatenation
"Ho " ^ 3                   # repeat string
string("a= ", 123.3)        # create using print function
occursin("CD","ABCD")        # check if first string contains second
"\n\t\$"                    # C-like escaping in strings, new \$ escape
x = 123
"$x + 3 = $(x+3)"           # unescaped $ is used for interpolation
"\$199"                      # to get a $ symbol you must escape it

r = r"A|B"                   # create new regexp
occursin(r, "CD")            # false, no match found
m = match(r, "ACBD")
                             # find first regexp match, see documentation for details
```

Functions

```
f(x, y = 10) = x + y           # new function f with y defaulting to 10
f(3, 2)                        # simple call, 5 returned
f(3)                           # 13 returned
function g(x::Int, y::Int)     # type restriction
    return y, x # explicit return of a tuple
end

g(x::Int, y::Bool) = x * y     # add multiple dispatch
g(2, true)                   # second definition is invoked
methods(g)                   # list all methods defined for g
```

Operators

```
true || false      # binary or operator (singeltons only)
[1 2] .& [2 1]      # bitwise and operator (vectorized by .)
1 < 2 < 3          # chaining conditions is OK (singeltons only without .)
[1 2] .< [2 1]      # for vectorized operators need to add '.' in front
a = 5
2a + 2(a+1) # multiplication can be omitted between a literal and a
variable or a left parenthesis

x = [1 2 3]          #1x3 Array{Int64,2}
y = [1, 2, 3]        #3-element Array{Int64,1}
x + y # error
x .+ y # 3x3 matrix, dimension broadcasting
x + y' # 1x3 matrix
x * y # array multiplication, 1-element vector (not scalar)
```

Numerical example – approximating Pi

$$\pi = 2 \sum_{n=0}^{+\infty} \frac{n!}{(2n+1)!!}$$

```
function our_pi(n, T)
    s = one(T)
    f = one(T)
    for i::T in 1:n
        f *= i/(2i+1)
        s += f
    end
    2s
end
```

Testing....

```
for T in [Float16, Float64, BigFloat]
    display([our_pi(2^n, T) for n in 1:10] .- big( $\pi$ ))
end
```

BigFloat

```
julia> our_pi(1000, BigFloat)-pi  
1.03634022661133335504636222353604794853392004373235376620284  
4416420231e-76
```

```
julia> setprecision(1000) do  
    our_pi(1000, BigFloat)-pi  
end  
3.73305447401287551596035817889526867846836578548683209848685  
7359183867643903102537817761308391524409438379959721296970496  
8619500854161295793660832688157230249376426645533006010959803  
0394360732604440196318506045247296205005918373516322071308450  
166041524279351541770592447787925691464383688807065164177119e  
-301
```

Rational numbers

```
julia> [our_pi(n, Rational) for n in 1:10]
10-element Array{Rational{Int64},1}:
8//3
44//15 64//21
976//315
10816//3465
141088//45045
47104//15015
2404096//765765
45693952//14549535
45701632//14549535
```


Julia IO – writing files

- In Julia the open command can be used to read and write to a particular file stream.

```
julia> f = open("some_name.txt", "w")  
IOStream(<file some_name.txt>)
```

- The write command takes a stream handle as the first parameter accepts a wide range of additional parameters.

```
write(f, "first line\nsecond line\n")
```

- Close the stream

```
close(f)
```

Julia IO – reading files

```
f = open("some_name.txt")
```

In order to read a single line from a file use the readline function.

```
julia> readline(f)
```

```
"first line"
```

```
julia> readline(f)
```

```
"second line"
```

```
julia> eof(f)
```

```
true
```

```
julia> close(f)
```

Parallel computing

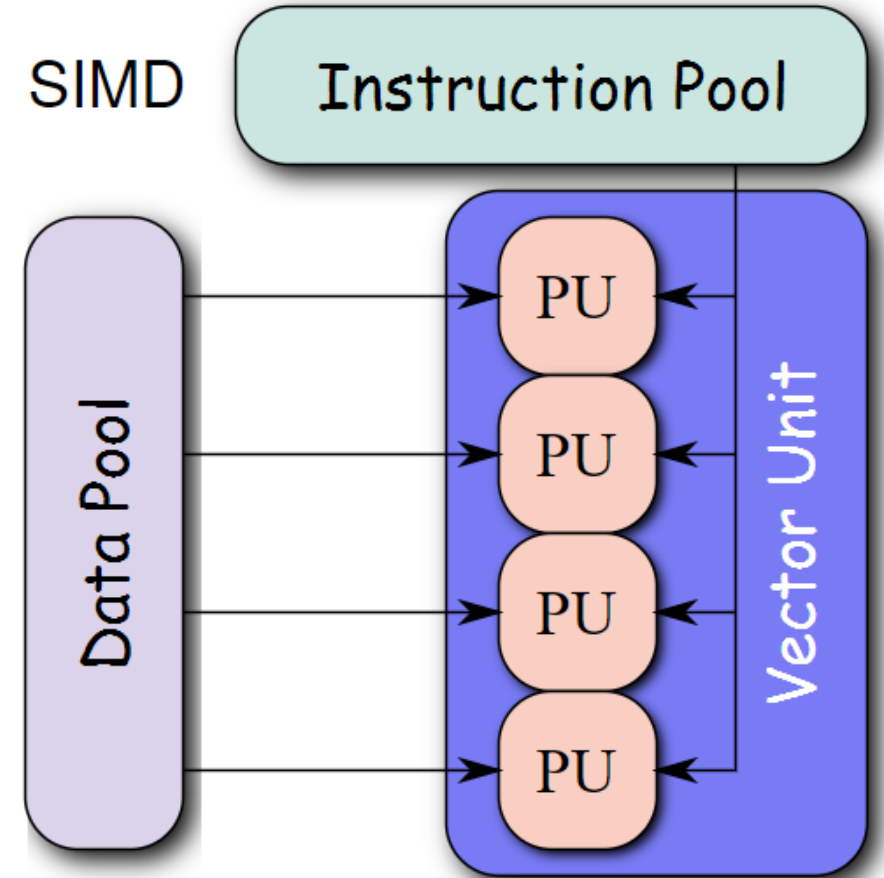
Where Julia shines...

Parallel computing

- Single instruction, multiple data (SIMD)
- Green-threads
- Multi-threading
- Multi-processing
 - local (single machine)
 - distributed (computing clusters)
 - computers within a network
 - managed supercomputer clusters (SLURM, SGE)

SIMD

- Single instruction, multiple data (SIMD) describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.



Source: <https://en.wikipedia.org/wiki/SIMD>

Data level parallelism

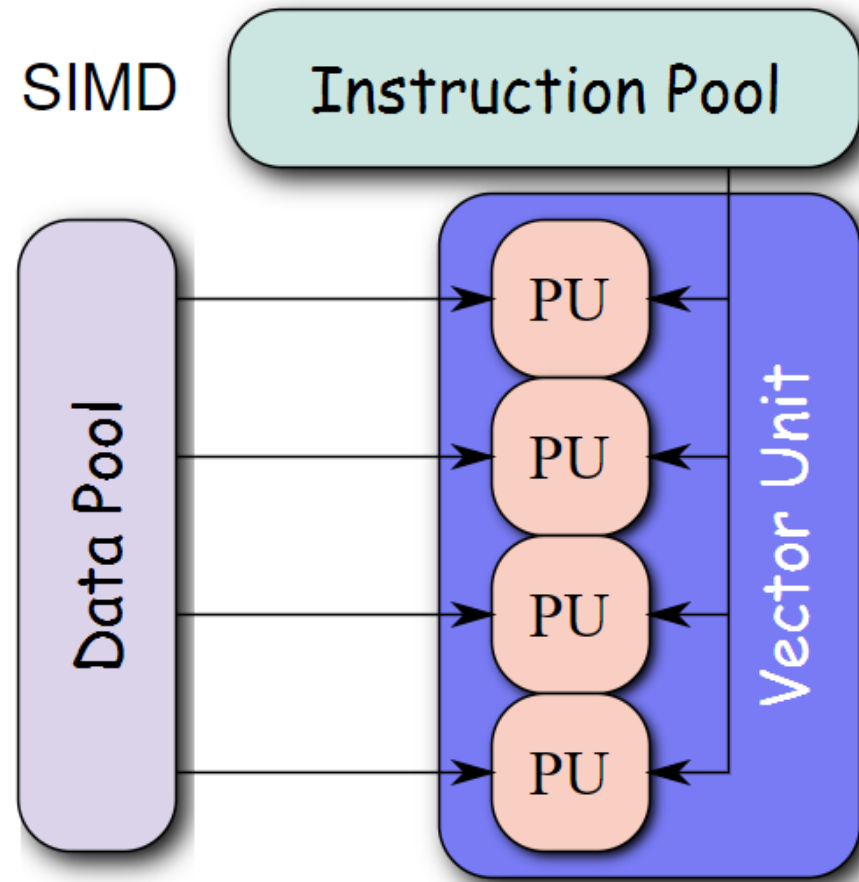


Image source: <https://en.wikipedia.org/wiki/SIMD>

```
# 1_dot/dot_simd.jl
```

```
function dot1(x, y)
    s = 0.0
    for i in 1:length(x)
        @inbounds s += x[i]*y[i]
    end
    s
end
```

```
function dot2(x, y)
    s = 0.0
    @simd for i in 1:length(x)
        @inbounds s += x[i]*y[i]
    end
    s
end
```

Time reduced from 0.83 to 0.5s

Simple example – threading

Single threaded

```
function ssum(x)
    r, c = size(x)
    y = zeros(c)
    for i in 1:c
        for j in 1:r
            y[i] += x[j, i]
        end
    end
    y
end
```

Multithreading

```
function tsum(x)
    r, c = size(x)
    y = zeros(c)
    Threads.@threads for i in 1:c
        for j in 1:r
            y[i] += x[j, i]
        end
    end
    y
end
```

Typical pattern for distributed simulation

```
using Distributed  
addprocs(4);
```

```
@everywhere include("sim_file.jl")
```

```
function init()  
    Random.seed!(myid())  
end
```

```
@sync for wid in workers()  
    @async fetch(@spawnat wid init())  
end
```


Writing distributed loops

```
data = @distributed (vcat) for i = 1:10000
    some_param_A = rand()
    some_param_B = rand()
    res_1, res_2, res_3 = run_sim();
    (sim_stats(res_1,res_2,res_3)... ,
     some_param_A,
     some_param_B,
     myid() )
end
```