Introduction to the Julia language

Bogumił Kamiński

Przemysław Szufel

Warsaw School of Economics, Poland

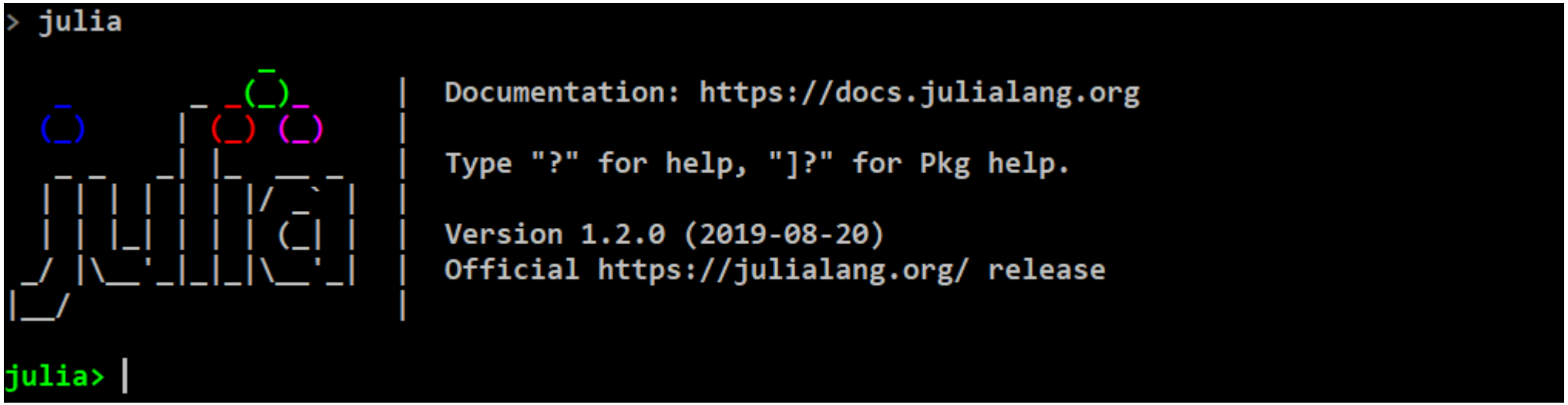
Download the slides! https://szufel.pl/ssw2020/

Installing and running Julia

- Free and Open Source
  - **https://julialang.org/downloads/**
  - v1.5.1 – the latest stable version
- Other options:
  - Julia Pro (version for the business, free)
    - https://juliacomputing.com/products/juliapro.html
  - JuliaBox
    - https://juliabox.com/
    - Free, and runs in a web browser
    - Great for quick hands-on experience

Download the slides! https://szufel.pl/ssw2020/

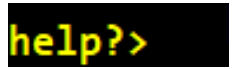
Julia Command Line (REPL)



pressing] changes REPL to package installation mode

pressing ; changes REPL to package installation mode

pressing ? changes REPL to help mode



to go back to normal mode press **BACKSPACE**

Download the slides! https://szufel.pl/ssw2020/

Adding Julia packages

- Start Julia REPL
- Press `]` to start the Julia package manager
  (prompt `(v1.5) pkg>` will be seen)
- Sample package installation command

```
(v1.5) pkg> add PyPlot DataFrames Distributions
```

to go back to normal mode press **BACKSPACE**

Download the slides! https://szufel.pl/ssw2020/

Managing packages

(press] for the package management REPL mode)

```
(v1.2) pkg> status
    Status `C:\JuliaPkg\Julia-1.2.0\environments\v1.2\Project.toml`
  [7c4d4715] AmplNLWriter v0.5.0
  [c52e3926] Atom v0.10.1
  [aae01518] BandedMatrices v0.11.0
  [6e4b80f9] BenchmarkTools v0.4.3
  [8e7c35d0] BlockArrays v0.10.0
  [ffab5731] BlockBandedMatrices v0.5.0
  [336ed68f] CSV v0.5.11
  [9961bab8] Cbc v0.6.4
```

```
(v1.2) pkg> add DataFrames
  Updating registry at `C:\JuliaPkg\Julia-1.2.0\registries\General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `C:\JuliaPkg\Julia-1.2.0\environments\v1.2\Project.toml`
 [no changes]
  Updating `C:\JuliaPkg\Julia-1.2.0\environments\v1.2\Manifest.toml`
 [no changes]
```

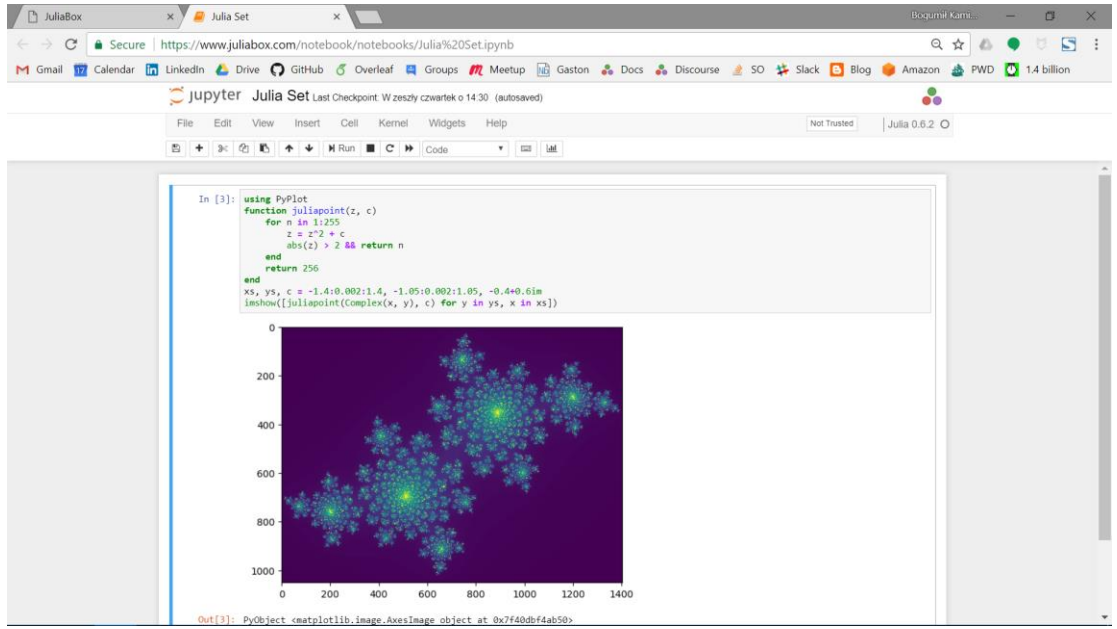
```
(v1.2) pkg> precompile
Precompiling project...
Precompiling DataFrames
[ Info: Recompiling stale cache
-5684-b7b6-d8193f3e46c0]
```

Download the slides! https://szufel.pl/ssw2020/

Jupyter notebook

- Jupyter notebook
  - `using Pkg; Pkg.add("IJulia")`
  - `using IJulia`
  - `notebook(dir=".")`

  - Press Ctrl+C to exit



Download the slides! https://szufel.pl/ssw2020/

Why another language

Download the slides! https://szufel.pl/ssw2020/

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

Download the slides! https://szufel.pl/ssw2020/

Methods of achieving high performance in different data science environments

Ecosystem	Glue	Hot code	GPU
R-based	**R**	**RCpp**	**C**
Python-based	**Python**	**Numba/Cython/C**	**C**
NetLogo	**R, Behaviorspace**	**Java**	**?**
Julia-based	**Julia**	**Julia**	**Julia**
Matlab-based	**Matlab**	**C**	**GPU coder**

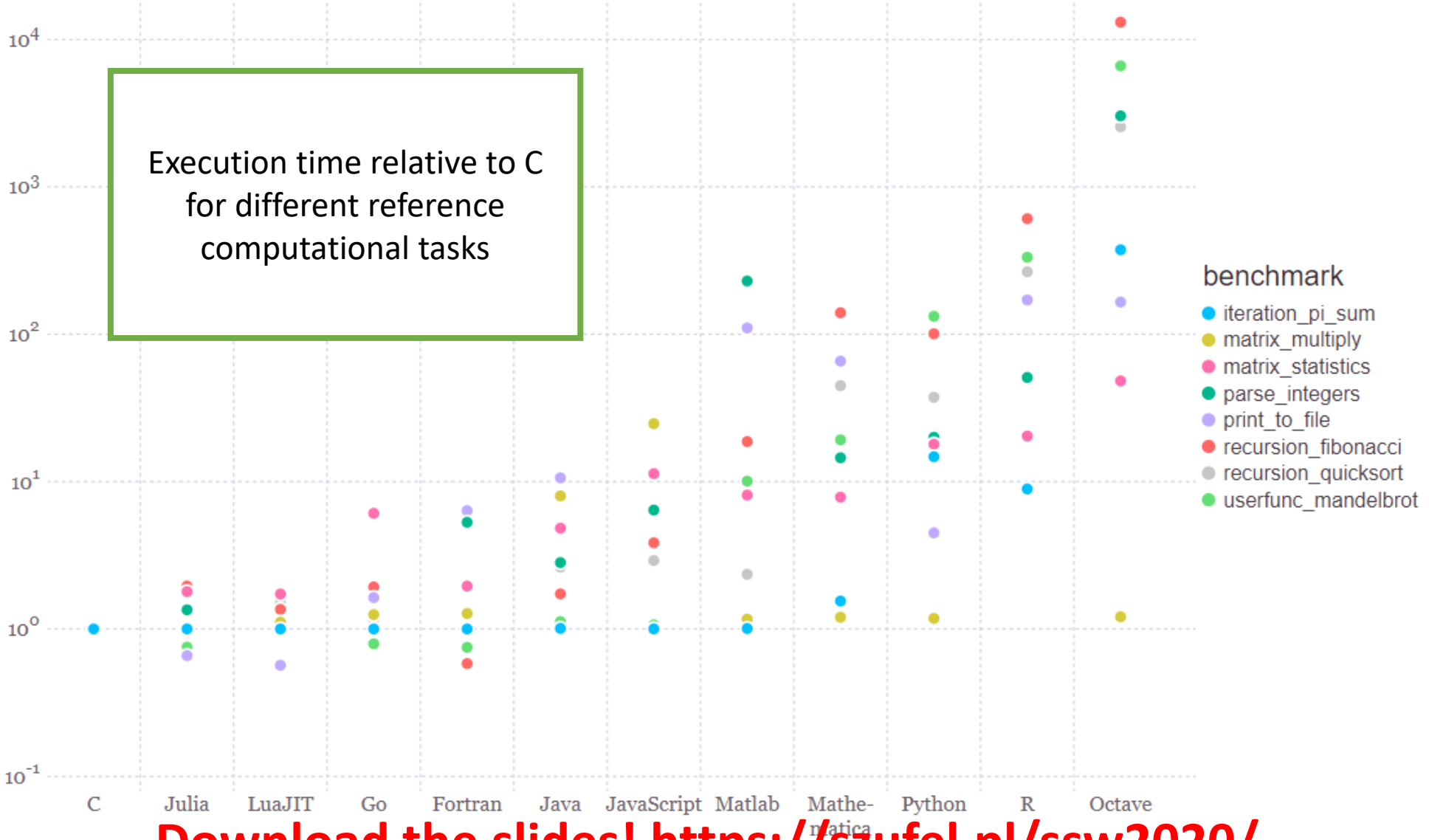
Download the slides! https://szufel.pl/ssw2020/

Matlab?

“We had to renew our licenses and got a bill for 30’000’000 USD” – overheard during a talk with a director of big supercomputer research center

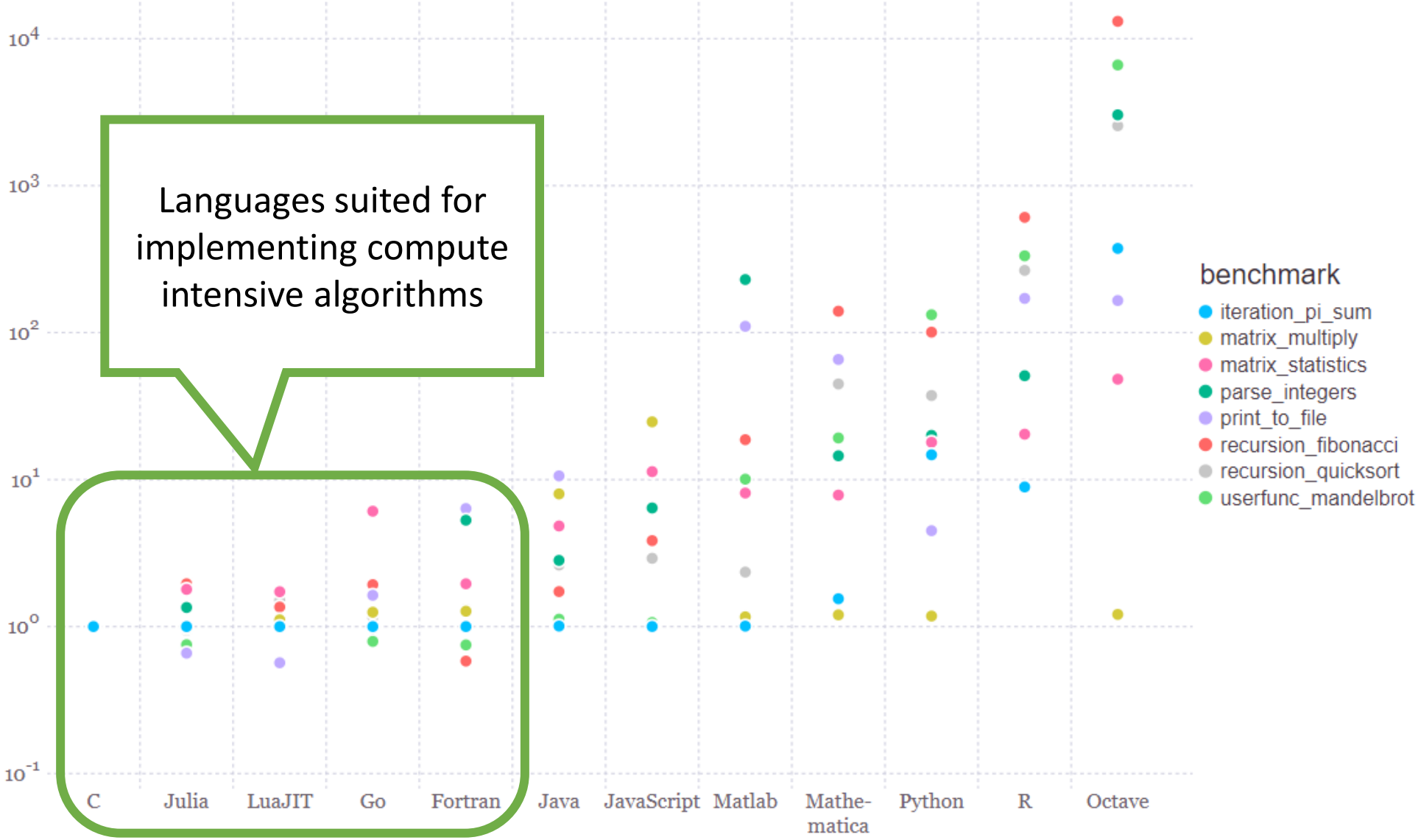
Download the slides! https://szufel.pl/ssw2020/

Reference benchmarks from Julia website



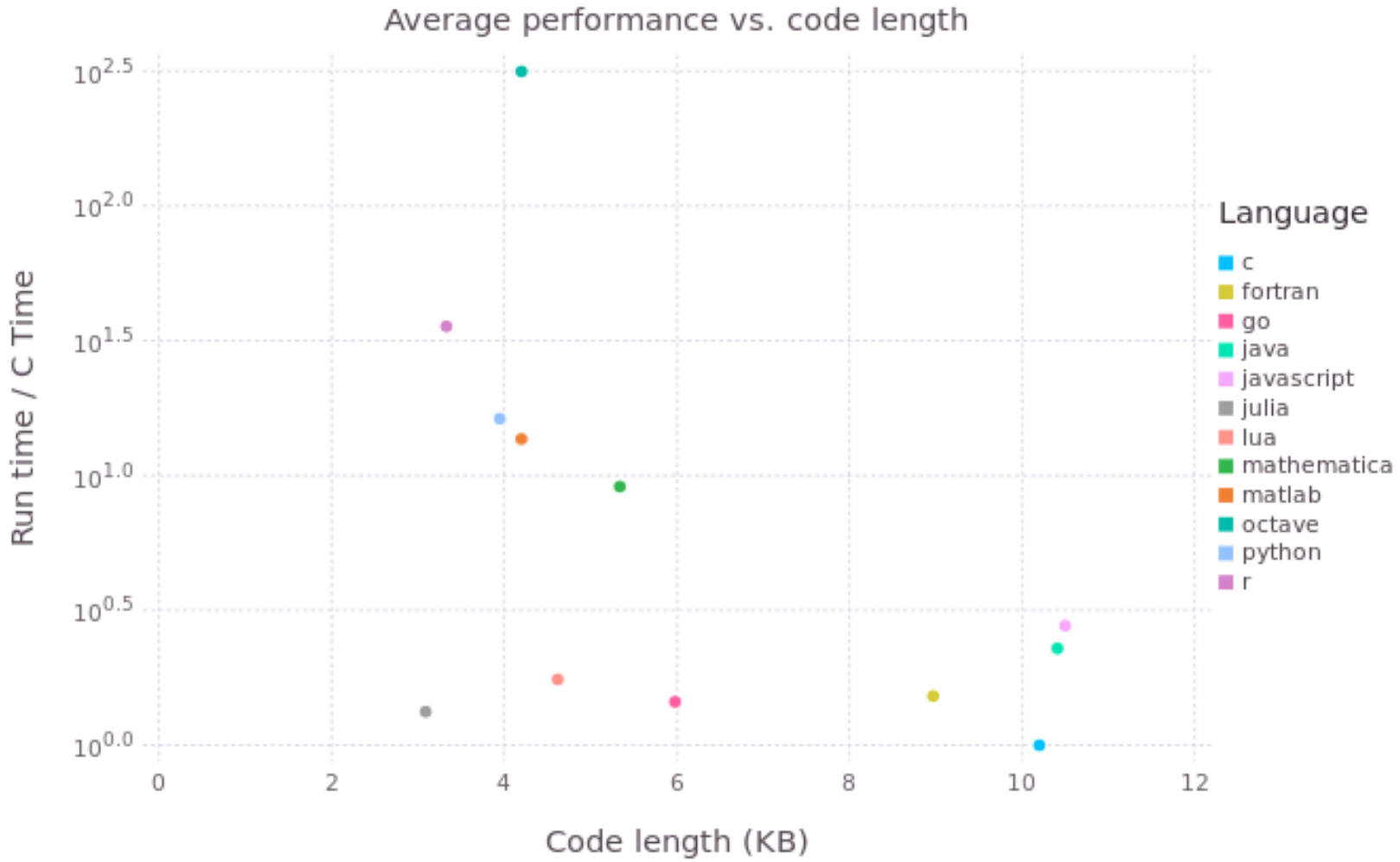
Download the slides! https://szufel.pl/ssw2020/

Reference benchmarks from Julia website



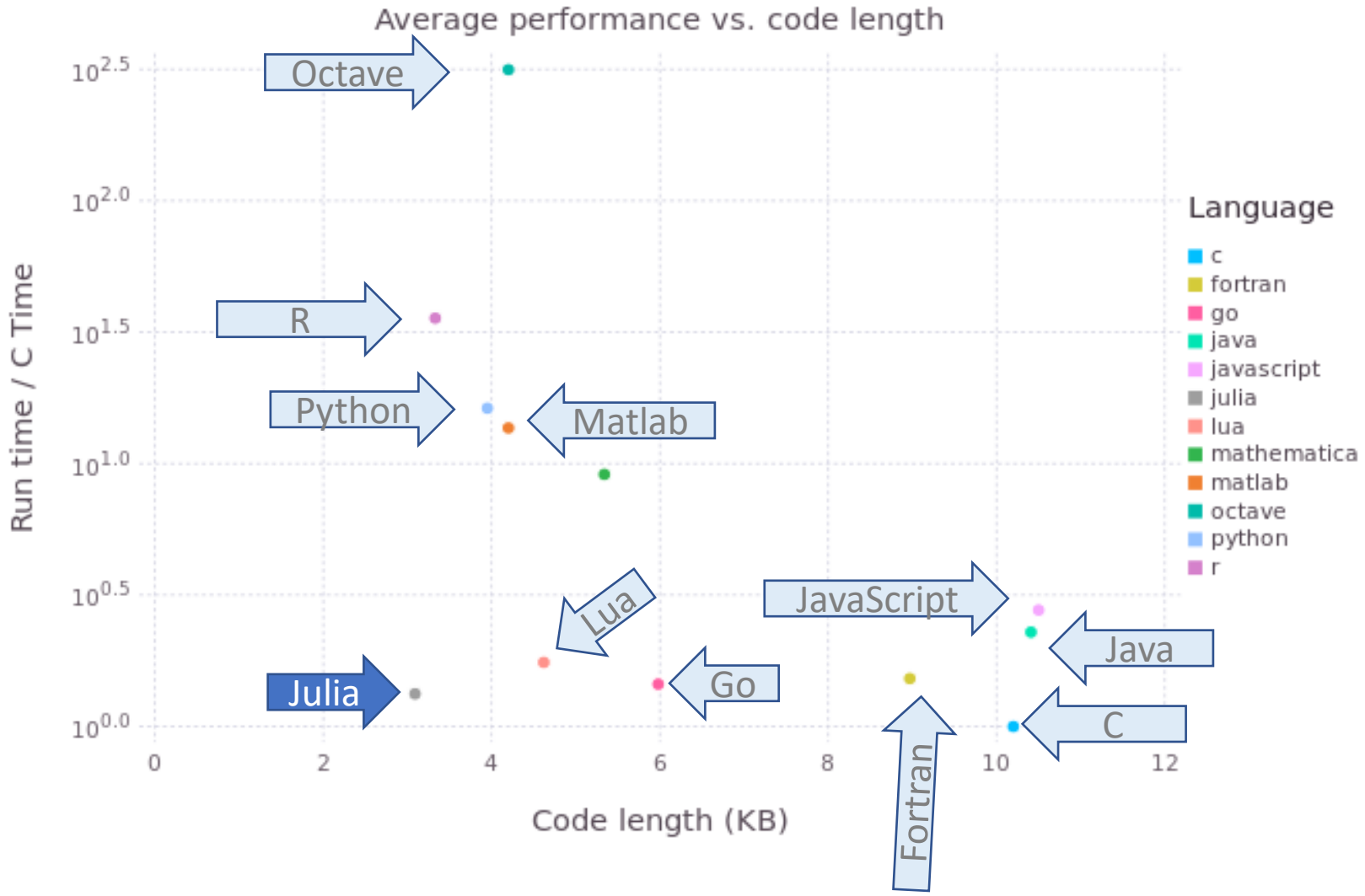
Source: http://www.oceanographerschoice.com/2016/03/the-julia-language-is-the-way-of-the-future/

Language Code Complexity vs Execution Speed



Download the slides! https://szufel.pl/ssw2020/

Language Code Complexity vs Execution Speed



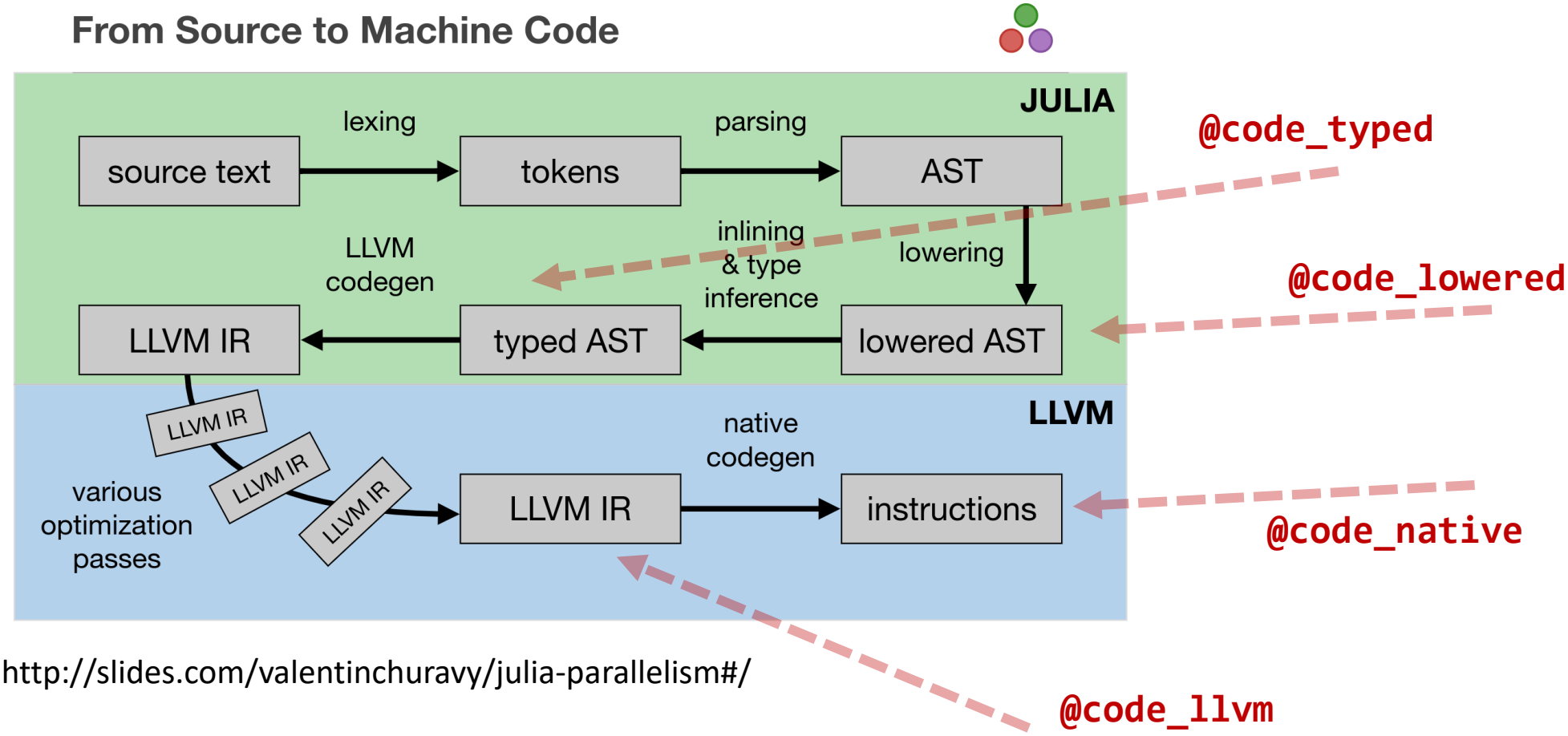
Source: http://www.oceanographerschoice.com/2016/03/the-julia-language-is-the-way-of-the-future/

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)

Download the slides! https://szufel.pl/ssw2020/

Julia code compilation process



http://slides.com/valentinchuravy/julia-parallelism#/

Download the slides! https://szufel.pl/ssw2020/

Polish use case@ Ministry of Health, Poland
Optimal allocation of accelerators in radiotherapy

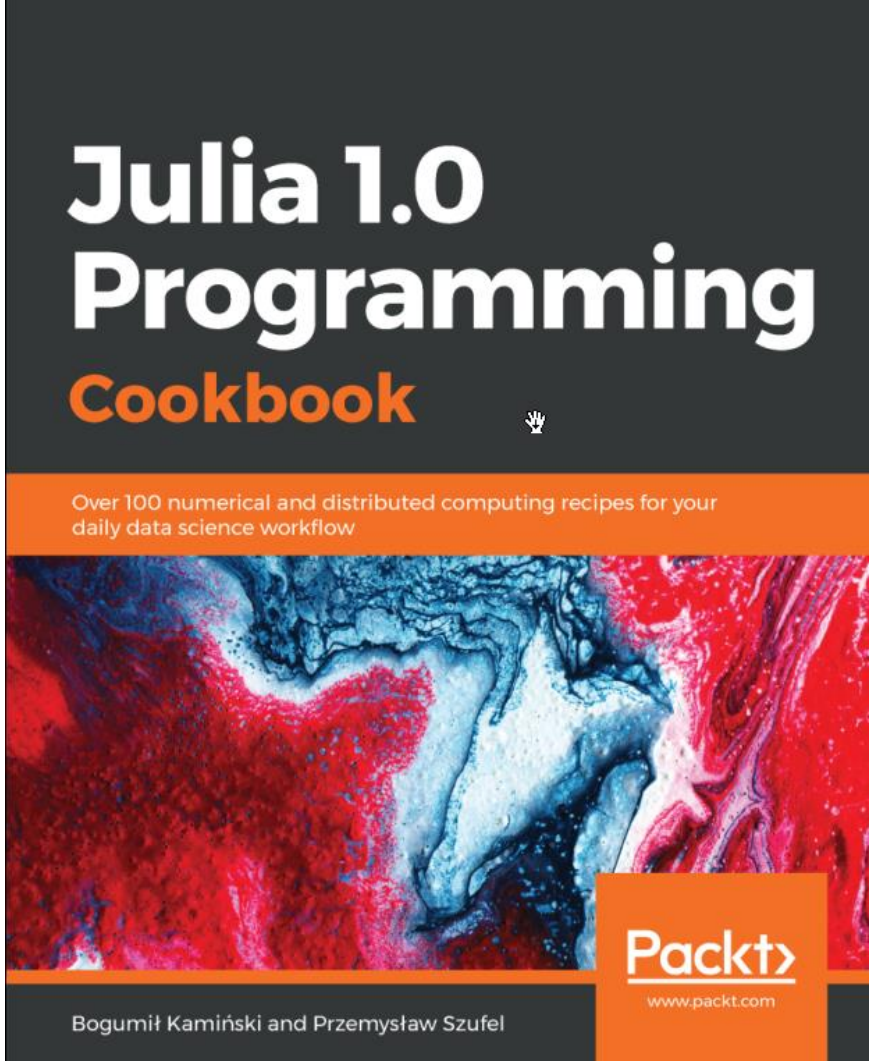
```
using JuMP, Cbc

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)
```

Download the slides! https://szufel.pl/ssw2020/

Learning more about Julia

- Website: https://julialang.org/
- Learning materials: https://julialang.org/learning/
- Blogs about Julia: https://www.juliabloggers.com/

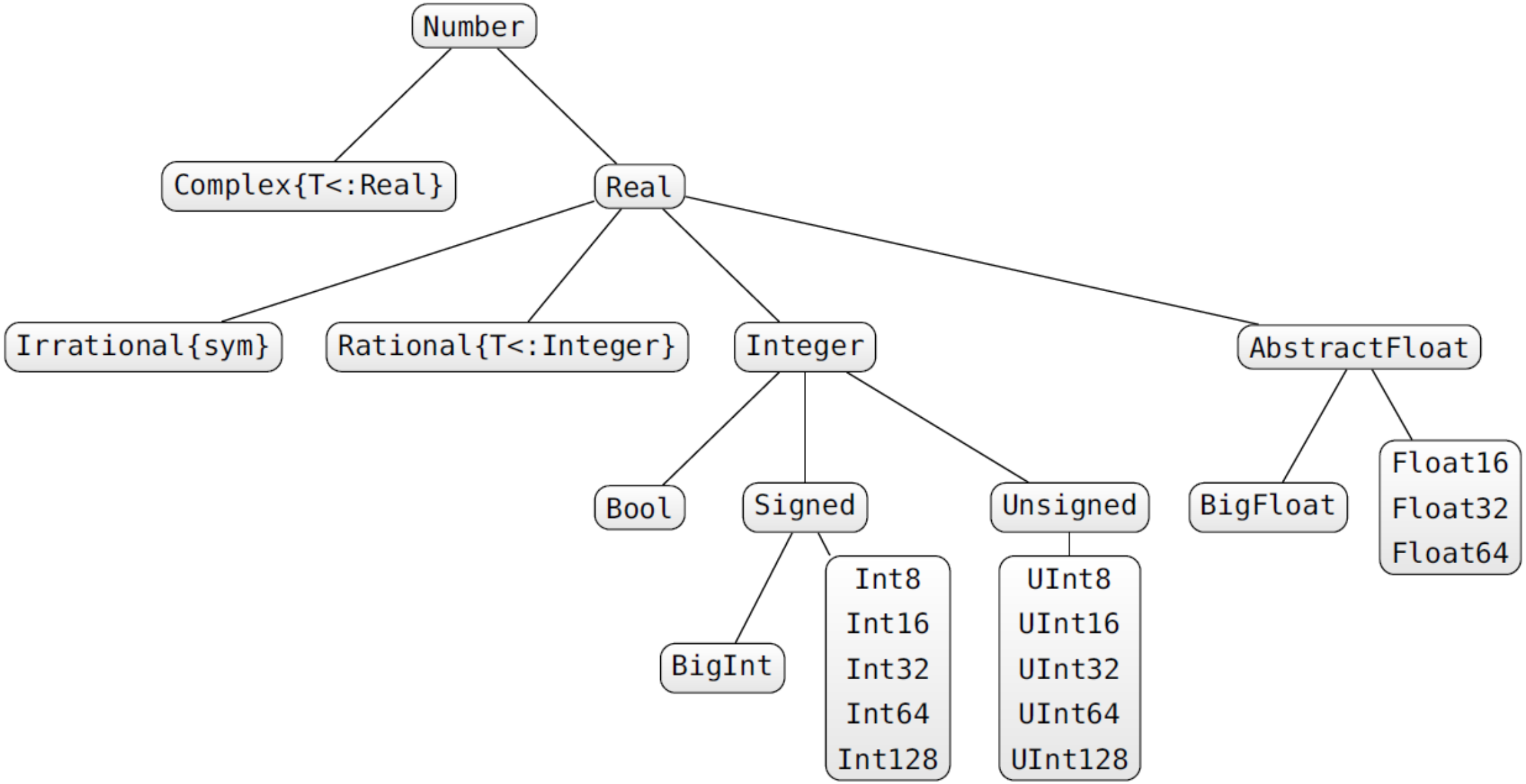
- **https://github.com/bkamins/The-Julia-Express**



- Julia forum: https://discourse.julialang.org/
- Q&A for Julia: https://stackoverflow.com/questions/tagged/julia-lang

Download the slides! https://szufel.pl/ssw2020/

Numeric type hierarchy



Download the slides! https://szufel.pl/ssw2020/

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
```

Download the slides! https://szufel.pl/ssw2020/

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`

Download the slides! https://szufel.pl/ssw2020/

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)
```

Metaprogramming and symbolic computing

JuliaDiff

Differentiation tools in Julia. JuliaDiff on GitHub.

Stop approximating derivatives!

Derivatives are required at the core of many numerical algorithms. Unfortunately, they are usually computed *inefficiently* and *approximately* by some variant of the finite difference approach

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}, h \text{ small}.$$

This method is *inefficient* because it requires $\Omega(n)$ evaluations of $f : \mathbb{R}^n \to \mathbb{R}$ to compute the gradient $\nabla f(x) = \left(\frac{\partial f}{\partial x_1}(x), \cdots, \frac{\partial f}{\partial x_n}(x)\right)$, for example. It is *approximate* because we have to choose some finite, small value of the step length h, balancing floating-point precision with mathematical approximation error.

What can we do instead?

One option is to explicitly write down a function which computes the exact derivatives by using the rules that we know from Calculus. However, this quickly becomes an error-prone and tedious exercise. **There is another way!** The field of automatic differentiation provides methods for automatically computing *exact* derivatives (up to floating-point error) given only the function f itself. Some methods use many fewer evaluations of f than would be required when using finite differences. In the best case, **the exact gradient of f can be evaluated for the cost of $O(1)$ evaluations of f itself**. The caveat is that f cannot be considered a black box; instead, we require either access to the source code of f or a way to plug in a special type of

`Calculus.jl` – symbolic differantion at compile time

```
julia> using Calculus

julia> differentiate(:(sin(x)))
:(1 * cos(x))

julia> expr = differentiate(:(sin(x) + x*x+5x))
:(1 * cos(x) + (1x + x * 1) + (0x + 5 * 1))

julia> x = 0; eval(expr)
```

Parallel computing

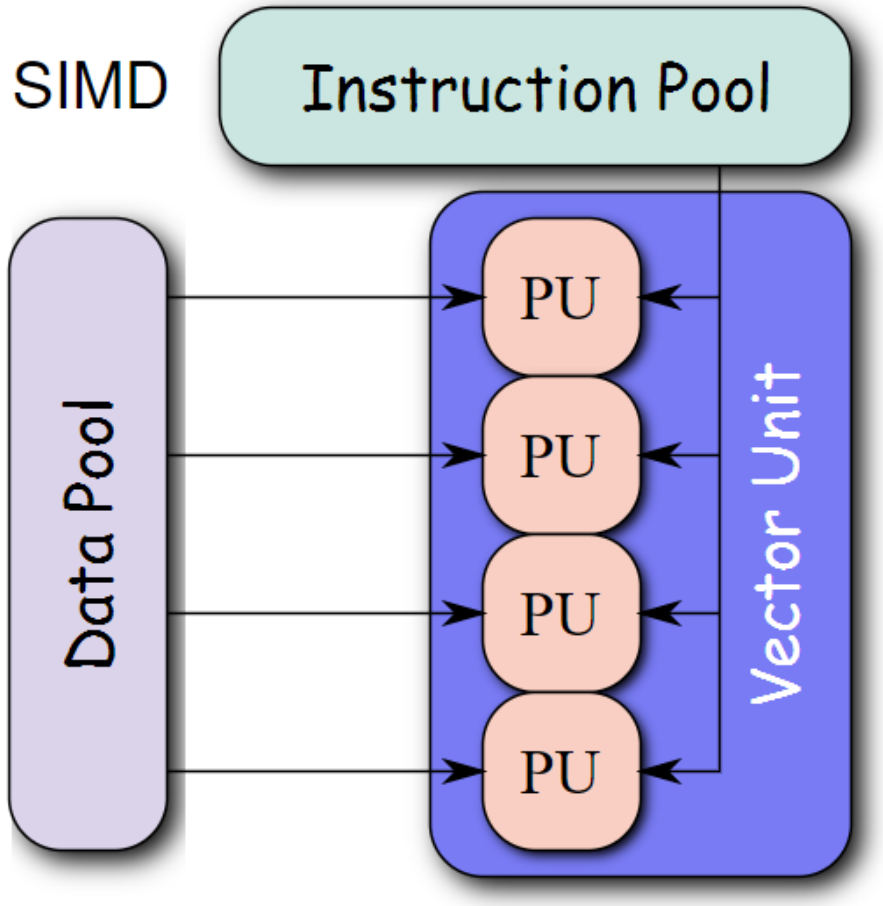
This is where Julia shines!!!

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

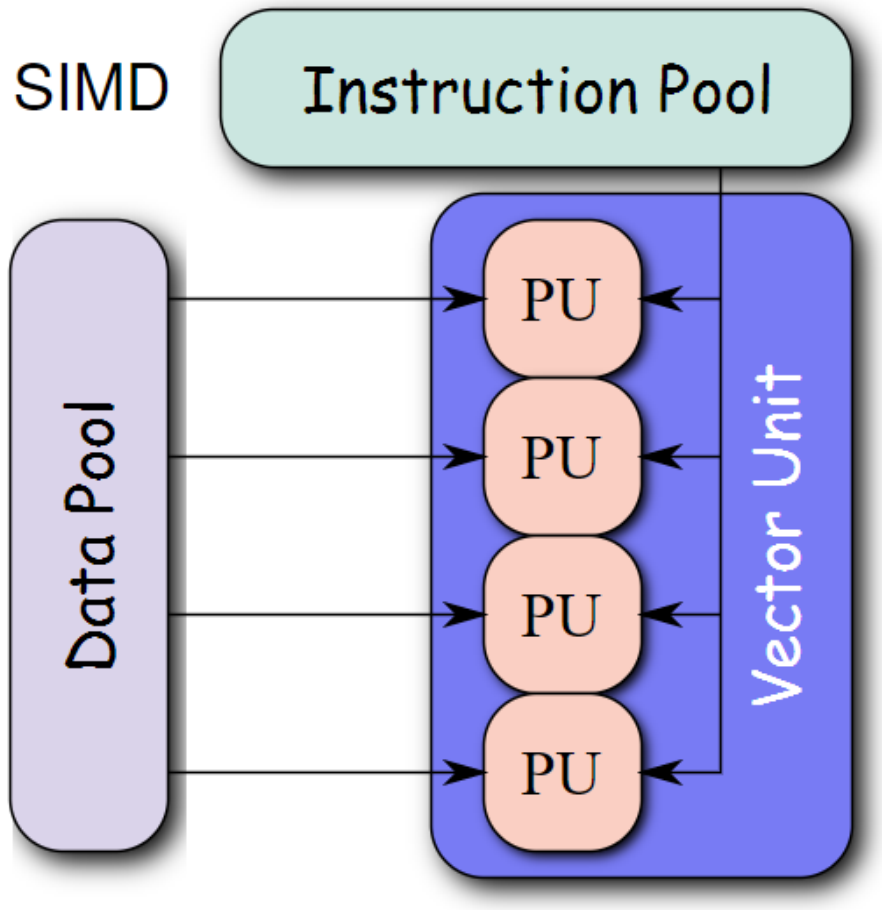


Image source: https://en.wikipedia.org/wiki/SIMD

```
# 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
```