Your first steps with Julia

Przemysław Szufel, PhD

Assistant Professor – SGH Warsaw School of Economics
https://szufel.pl/

Preparation of this workshop has been supported by the Polish National Agency for Academic Exchange under the Strategic Partnerships programme, grant number BPI/PST/2021/1/00069/U/00001.

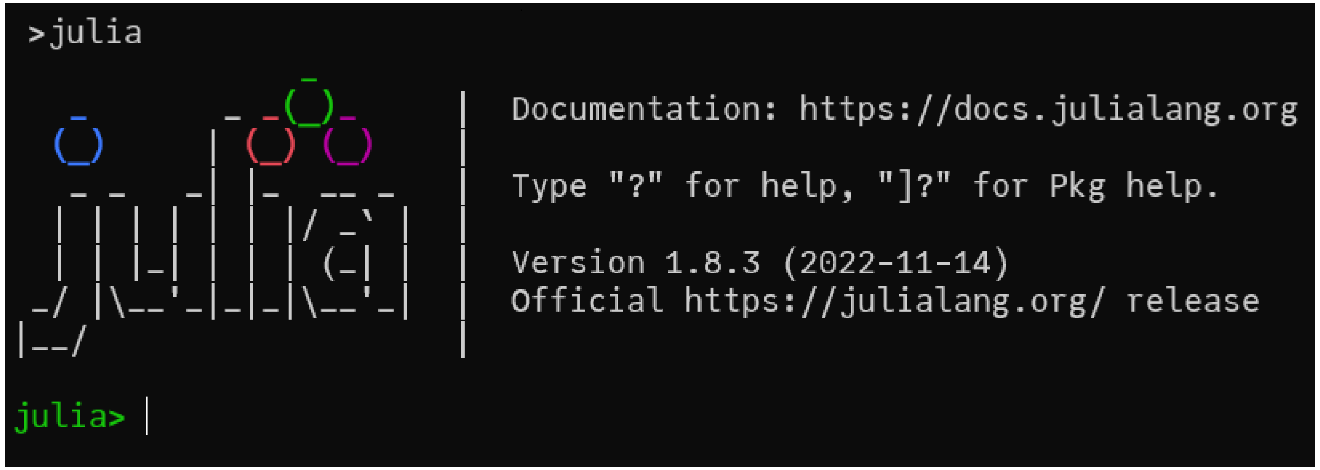
SGH Warsaw School of Economics

NAWA POLISH NATIONAL AGENCY FOR ACADEMIC EXCHANGE

Installing and running Julia

- Download Julia
  - Free and Open Source
  - **https://julialang.org/downloads/**
  - v1.10.0 – the latest stable version
- Programming environment – VS Code
  - https://code.visualstudio.com/download/)
- Jupyter notebook
  - Available via IJulia package

Julia Command Line (REPL)



pressing **]** changes REPL to package installation mode `(v1.2) pkg>`

pressing **;** changes REPL to package installation mode `shell>`

pressing **?** changes REPL to help mode `help?>`

to go back to normal mode press **BACKSPACE** `julia>`

Adding Julia packages

- Start Julia REPL

- Press **]** to start the Julia package manager (prompt `(v1.10) pkg>` will be seen)

- Sample package installation command

```
(v1.10) pkg> add PyPlot DataFrames Distributions
```

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

Managing packages
(press] for the package management REPL mode)

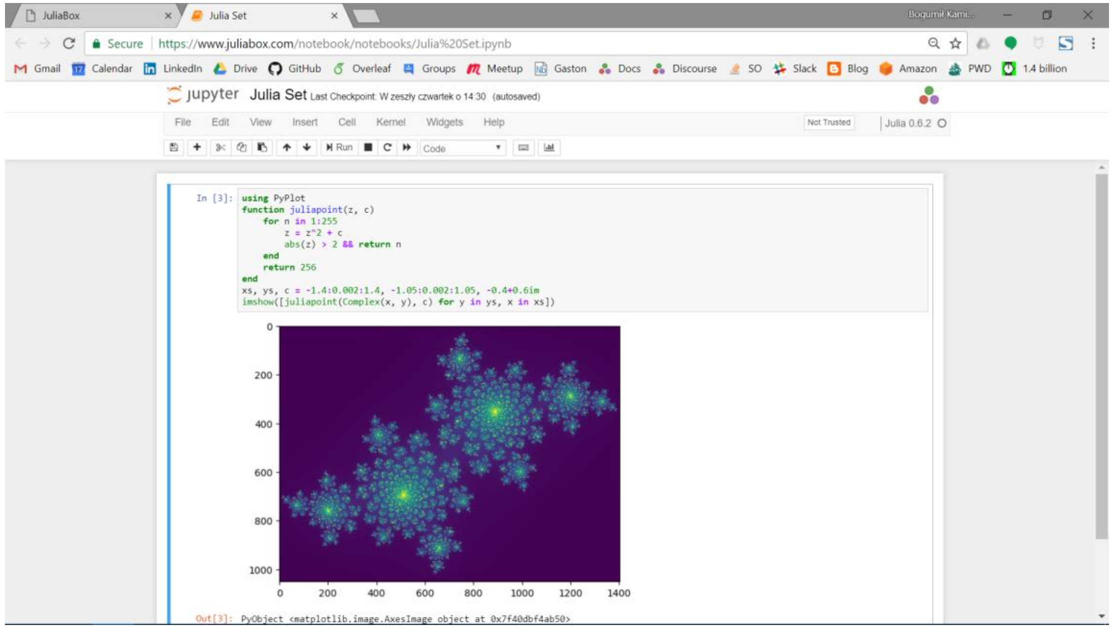
```
(@v1.6) pkg> status
     Status `C:\JuliaPkg\Julia-1.6.3\env
  [46ada45e] Agents v4.5.6
  [6e4b80f9] BenchmarkTools v1.2.0
  [336ed68f] CSV v0.8.5
  [34f1f09b] ClusterManagers v0.4.2
  [5ae59095] Colors v0.12.8
  [8f4d0f93] Conda v1.5.2
  [a93c6f00] DataFrames v1.2.2
```

```
(@v1.6) pkg> add RCall
   Updating registry at `C:\JuliaPkg\Julia-1.6.3\registries\General`
   Updating git-repo `https://github.com/JuliaRegistries/General.git`
  Resolving package versions...
  Installed ShiftedArrays ───── v1.0.0
  Installed WinReg ──────────── v0.3.1
  Installed StatsModels ─────── v0.6.26
  Installed RCall ───────────── v0.13.12
  Installed CategoricalArrays ─ v0.10.1
   Updating `C:\JuliaPkg\Julia-1.6.3\environments\v1.6\Project.toml`
  [6f49c342] + RCall v0.13.12
   Updating `C:\JuliaPkg\Julia-1.6.3\environments\v1.6\Manifest.toml`
  [324d7699] + CategoricalArrays v0.10.1
  [6f49c342] + RCall v0.13.12
  [1277b4bf] + ShiftedArrays v1.0.0
  [3eaba693] + StatsModels v0.6.26
  [1b915085] + WinReg v0.3.1
   Building RCall → `C:\JuliaPkg\Julia-1.6.3\scratchspaces\44cfe95a-1eb2-52ea-b672-
Precompiling project...
  4 dependencies successfully precompiled in 13 seconds (282 already precompiled)
```

Jupyter notebook

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

  - Press Ctrl+C to exit



Julia 10,000 feet overview

- Exponential growth, in several areas became a standard for scientific and high performance computing
- “walks like Python runs like C”
- Syntax in-between Pyhton/numpy and Matlab
- Compiles to assembly
- Compiles to GPU
- Distributed computing built into the language (known to scale up to millions of CPU cores)
- Best option for number crunching

Why another language for data science?

Two language problem of data science – programming languages

- are either fast (C++, Fortran)
- or are convenient (Python, R, Matlab)

Main features of Julia

1. Efficiency
2. Expressiveness
3. Integrability
4. Metaprogramming – DSLs for various data science subproblems
5. Integration and toolboxes

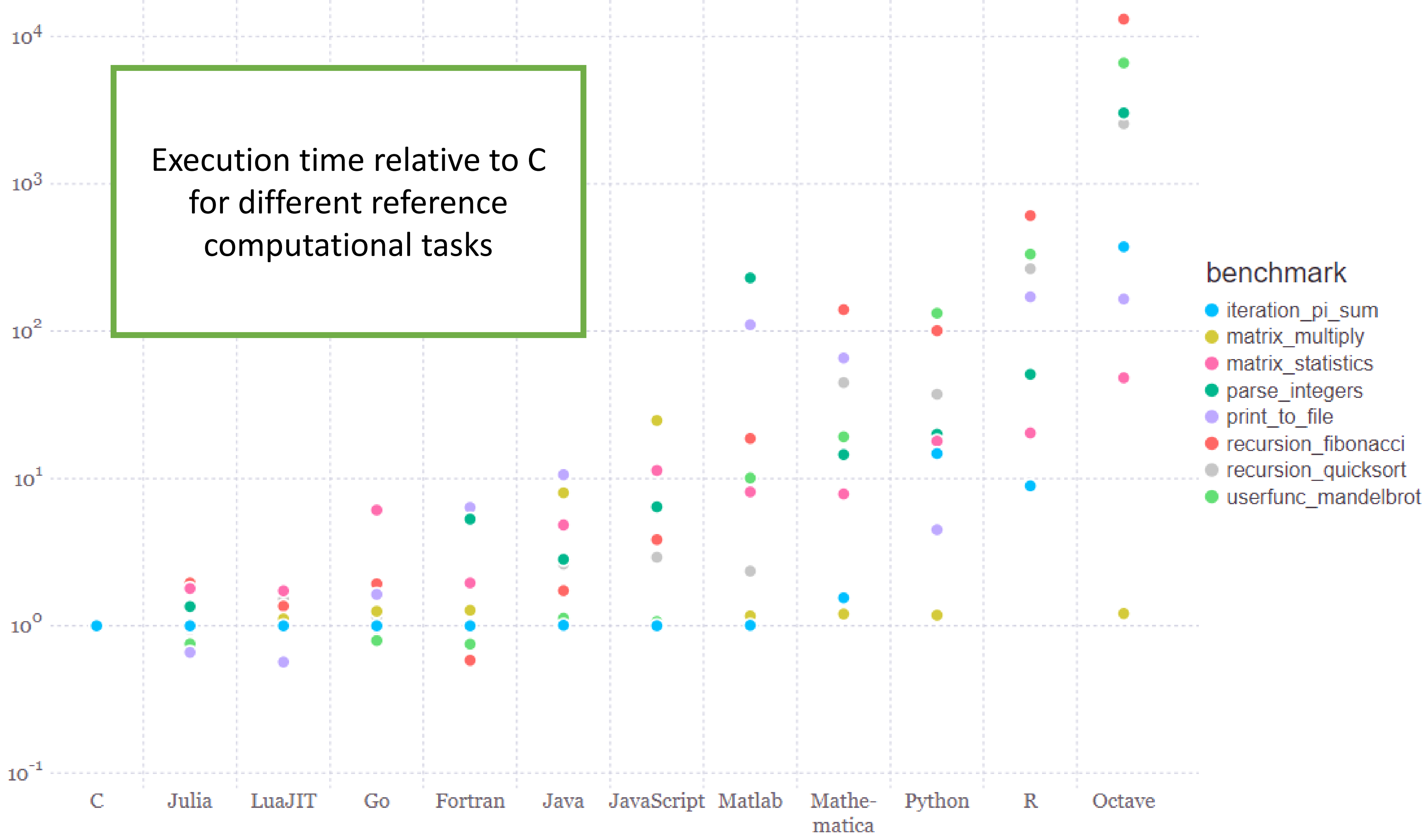
Methods of achieving high performance in different data science environments

Ecosystem	Glue	Hot code	GPU
R-based	R	RCpp	C
Python-based	Python	Numba/Cython/C	C
Julia-based	Julia	Julia	Julia
Matlab-based	Matlab	C	GPU coder

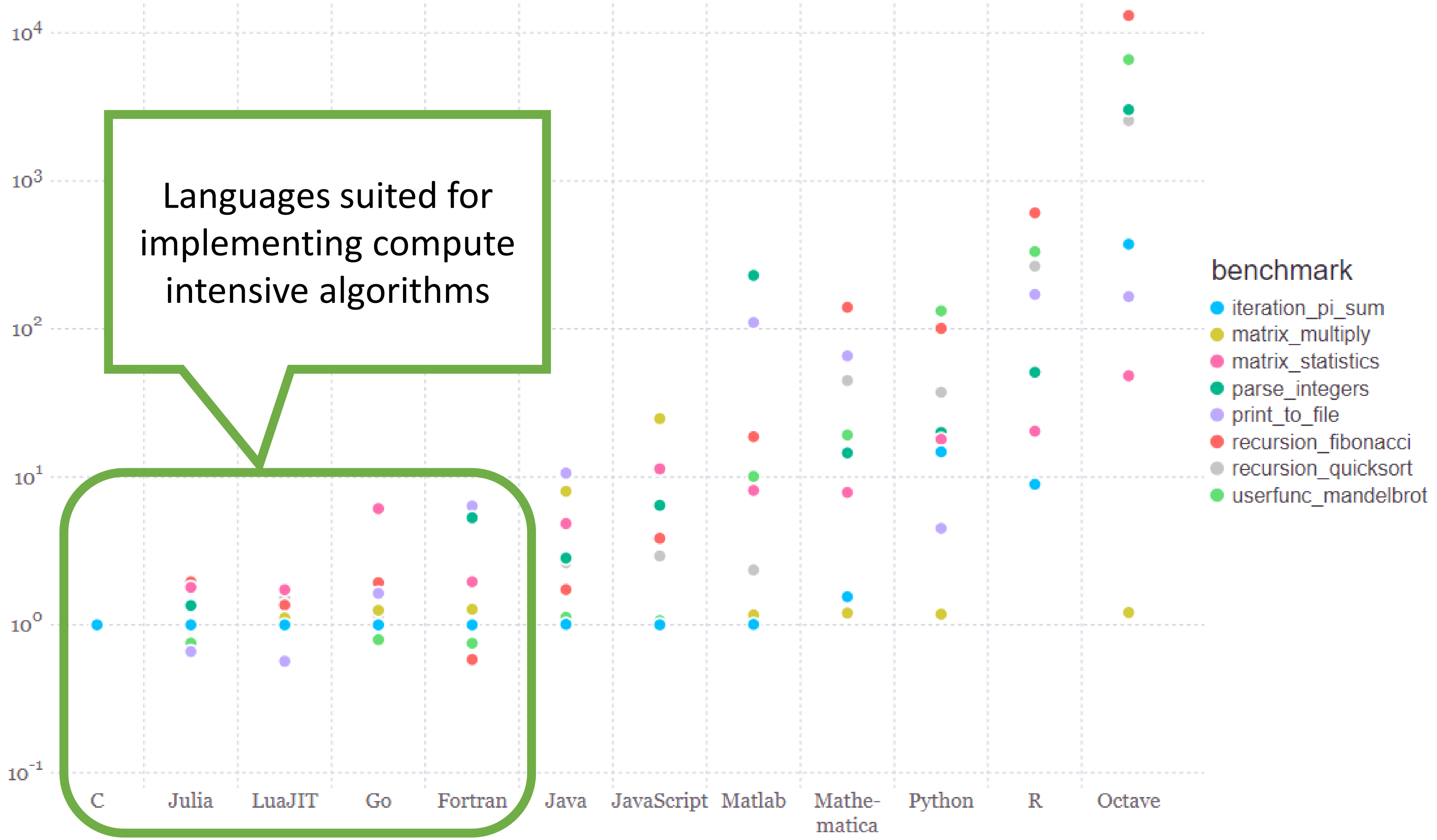
Matlab?

“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

Reference benchmarks from Julia website

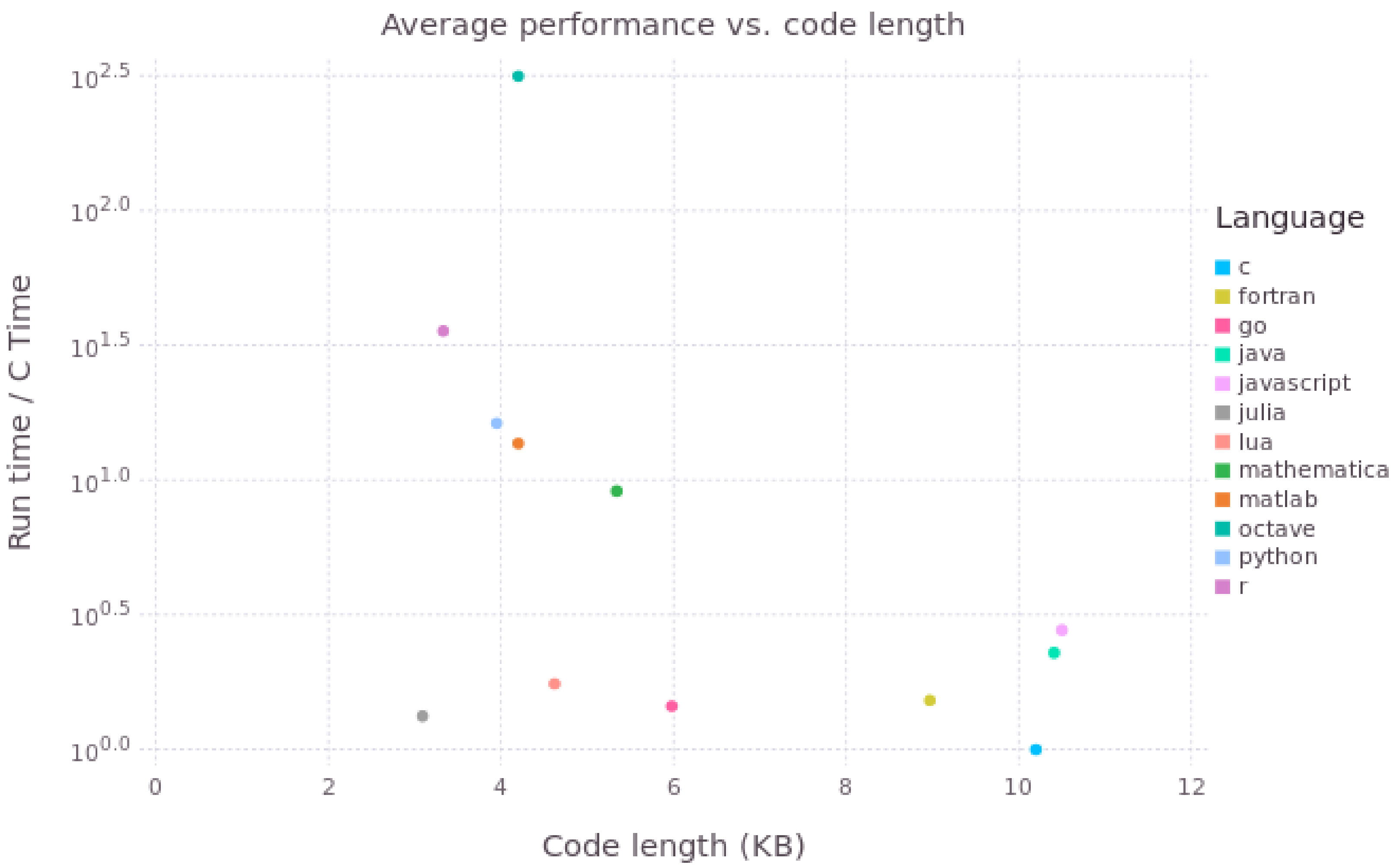


Reference benchmarks from Julia website

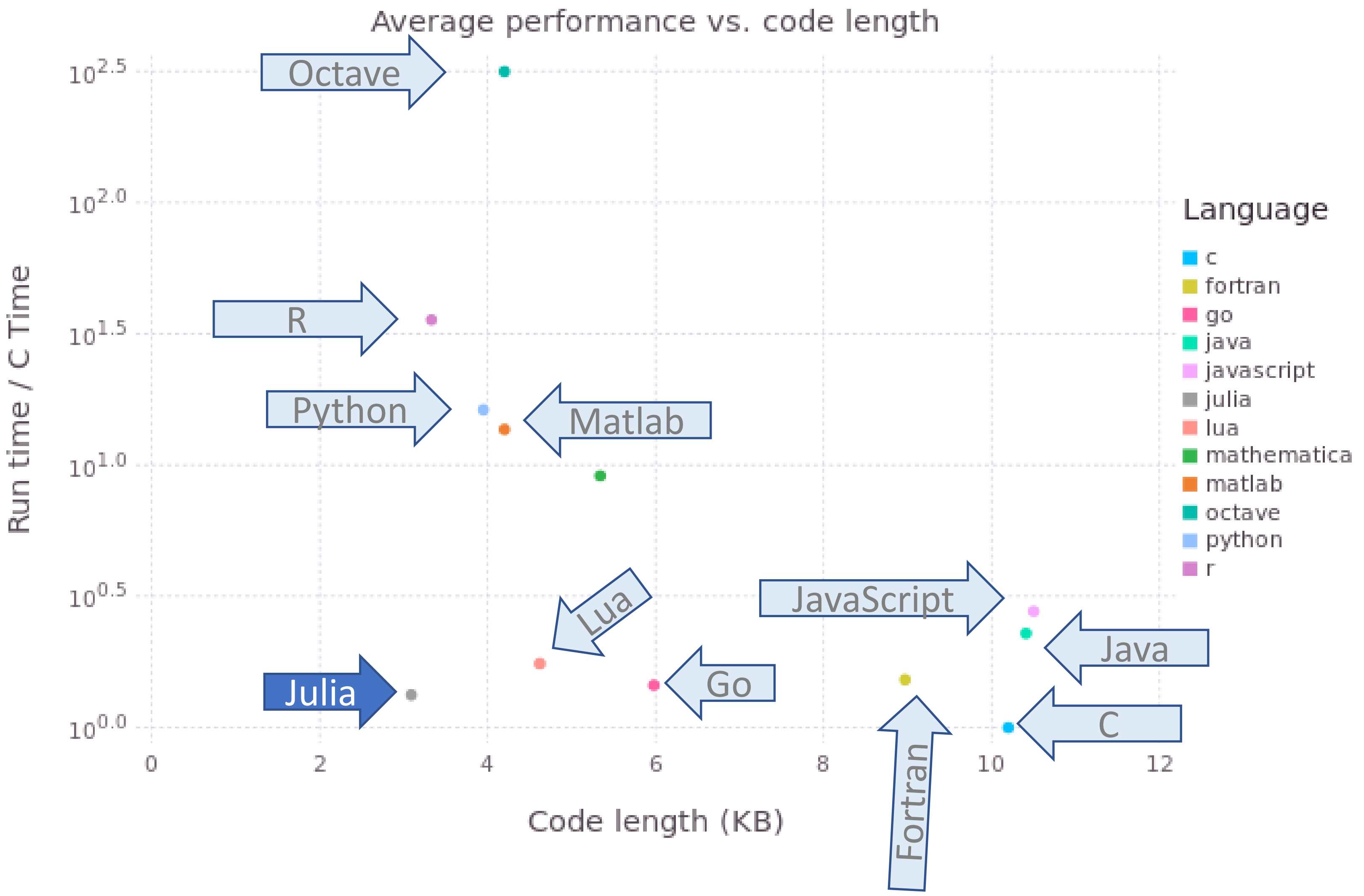


Source: http://www.oceanographerschoice.com/2016/03/the-julia-language-is-the-way-of-the-future/

Language Code Complexity vs Execution Speed



Language Code Complexity vs Execution Speed

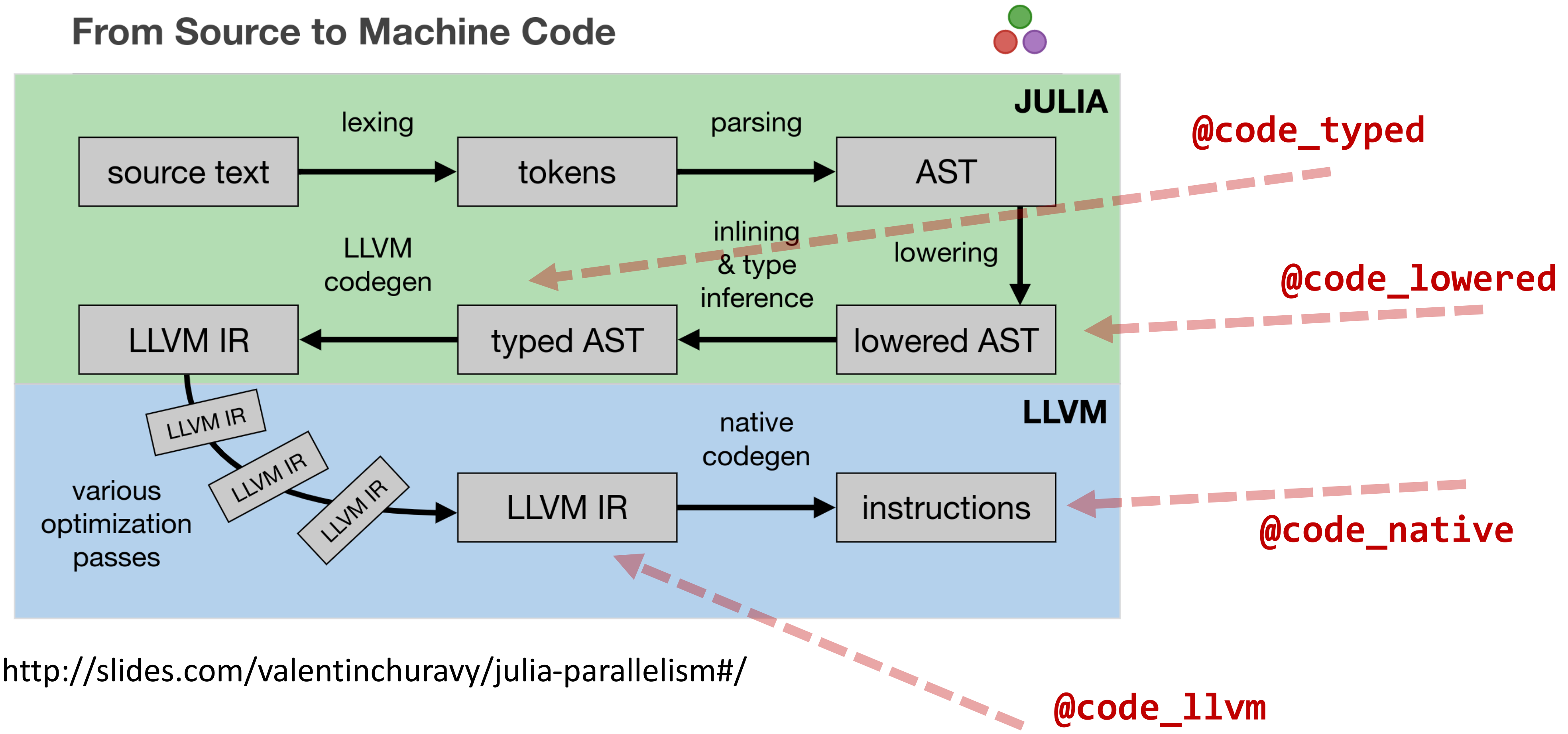


Source: http://www.oceanographerschoice.com/2016/03/the-julia-language-is-the-way-of-the-future/

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)

Julia code compilation process



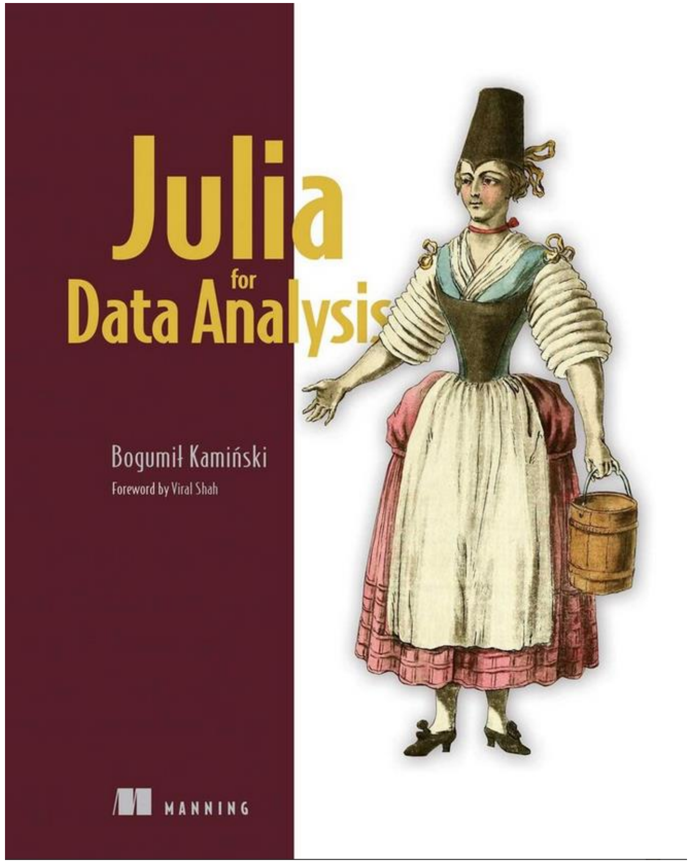
http://slides.com/valentinchuravy/julia-parallelism#/

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**

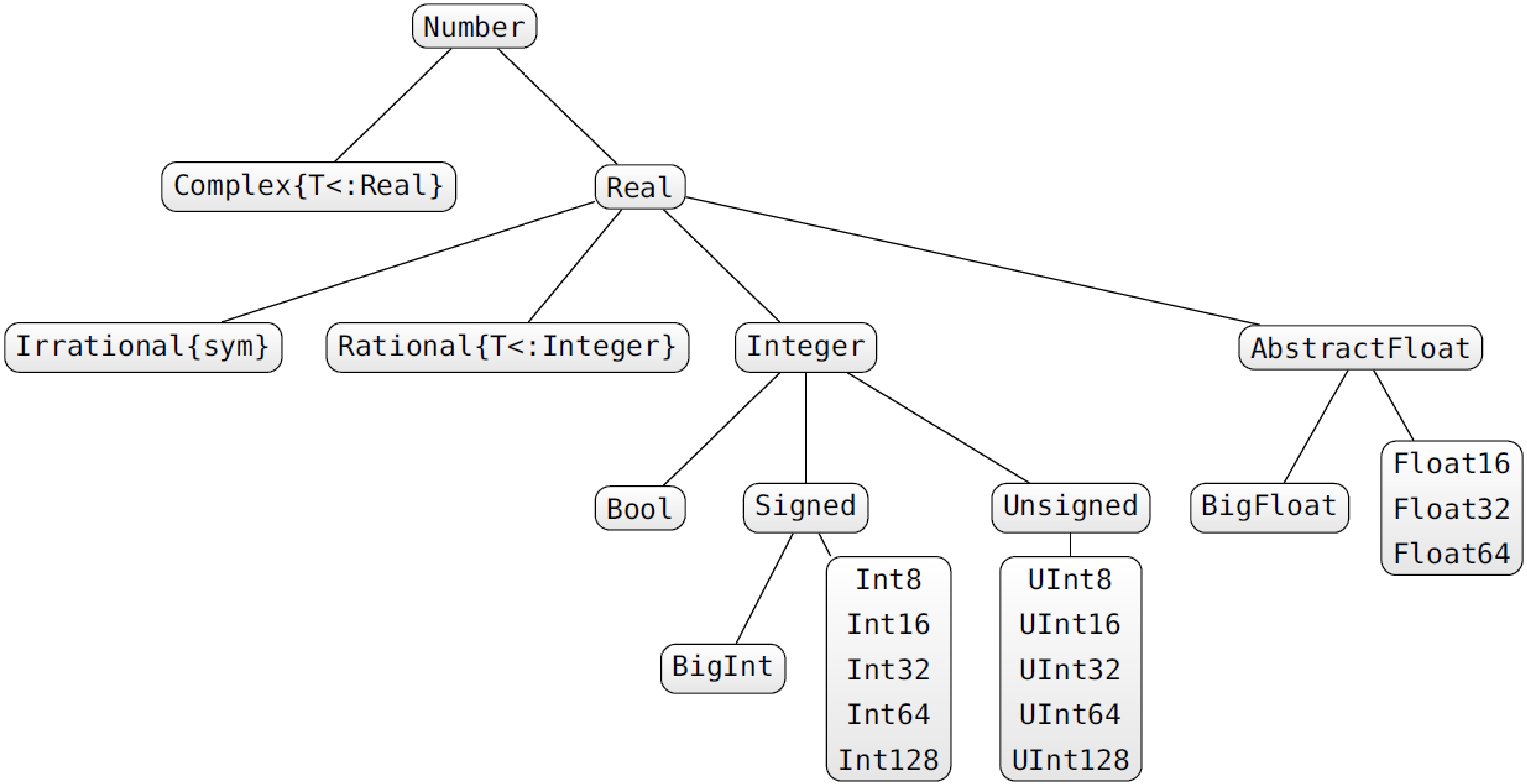
- Julia forum: https://discourse.julialang.org/
- Q&A for Julia: https://stackoverflow.com/questions/tagged/julia-lang



Some basic commands
(see also https://github.com/bkamins/The-Julia-Express)

```
@less(max(1,2))
     # show function source code

cd("D:/") # change working directory to D:/
cd(raw"c:\temp")

pwd() # current directory

include("file.jl") # run file

exit() # end your Julia session
```

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 – just like in 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]                   # first 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`

Tuples are immutable, and the Julia compiler makes a good use of that!

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

x = range(1, stop=2, length=5)
  # iterator having 5 equally spaced elements (1.0:0.25:2.0)
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
```

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 is not object oriented language – multiple dispatch is used instead

Default values require a macro

```
Base.@kwdef struct A
    a::Int = 6
    b::Float64 = -1.1
    c::UInt8 = 1
end
A()
A(a=2, c=4)
```

Dictionaries

```
x = Dict{Int, Float64}()
     # empty dictionary (types for keys and values are defined)
y = Dict(1=>5.5, 2=>4.5)          # dictionary
y[2]                              # return element
y[3] = 30.0                       # add element

keys(y), values(y)                # iterators
haskey(y)
```

Texts and interpolations

```
"Hi " * "there!"           # concatenation
string("a= ", 123.3)       # concatenation

x = 123
"$x + 3 = $(x+3)"          # $ is used for interpolation
"\$199"                    # and needs to be escaped with a `\`

occursin("CD", "ABCD")     # occurrence
occursin(r"A|B", "ABCD")  # occurrence with RegExp
```

Functions

```
f(x, y = 10) = x + y
               # default value for y is 10


function g(x::Int, y::Int)     # ograniczenie typu
     return y, x # yields a tuple
end

g(x::Int, y::Bool) = x * y     # multiple dispatch
g(2, true)                     # 2nd definition will be called
methods(g)                     # list of methods for g
```

Operators

```
true || false   # binary or operator (singeltons only)
1 < 2 < 3       # condition chaining
[1 2] .< [2 1]  # vectorization with a dot "."
a = 5
2a + 2(a+1) # multiplication “*” can be ommited

x = [1 2 3]         #matrix 1×3 Array{Int64,2}
y = [1, 2, 3]      #vector of 3-elements Array{Int64,1}
# Vectors are vertical and algebra rules apply
x + y # error
x .+ y # 3x3 matrix, dimension broadcasting
x + y' # 1x3 matrix
x * y # array multiplication, 1-element vector (not scalar)
```

Numerical example – pi approximation

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

```
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(π))
end
```

BigFloat

```
julia> our_pi(1000, BigFloat)-π
1.036340226611333355046362223536047948533920043732353766202844416420231e-76

julia> setprecision(1000) do
    our_pi(1000, BigFloat)-π
end
3.73305447401287551596035817889526867846836578548683209848685735918386764390310253781776130839152440943837995972129697049686195008541612957936608326881572302493764266455330060109598030394360732604440196318506045247296205005918373516322071308450166041524279351541770592447787925691464383688807065164177119e-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)
```

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 number using operator overloading.

`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)
```