Introduction to

julia

Programming

Table of content

- **Overview of Julia**
- **Advantages for Scientists**
- **Performance and Type System**
- **Julia Ecosystem**
- **Interoperability and Legacy Code**
- **Resources for Learning Julia**
- **Practical Applications and Exercises**

julia - in a nutshell

High-performance	**High-level code**
Open source	Designed for technical computing
Active developer community	Rich ecosystem of libraries
Familiar syntax for users of	

2009	Work started on Julia by (Python: 1989, C: 1969, Fortran: early 1950s)
2012	release on github (0.1 alpha)
2015	First JuliaCon
2018	v1.0
Post 2018	Continuous development (currently 1.9 with rc 1.10)

Now:

Fork 5.3k Star 43.4k

>10,000 Julia packages
>45 Million downloads

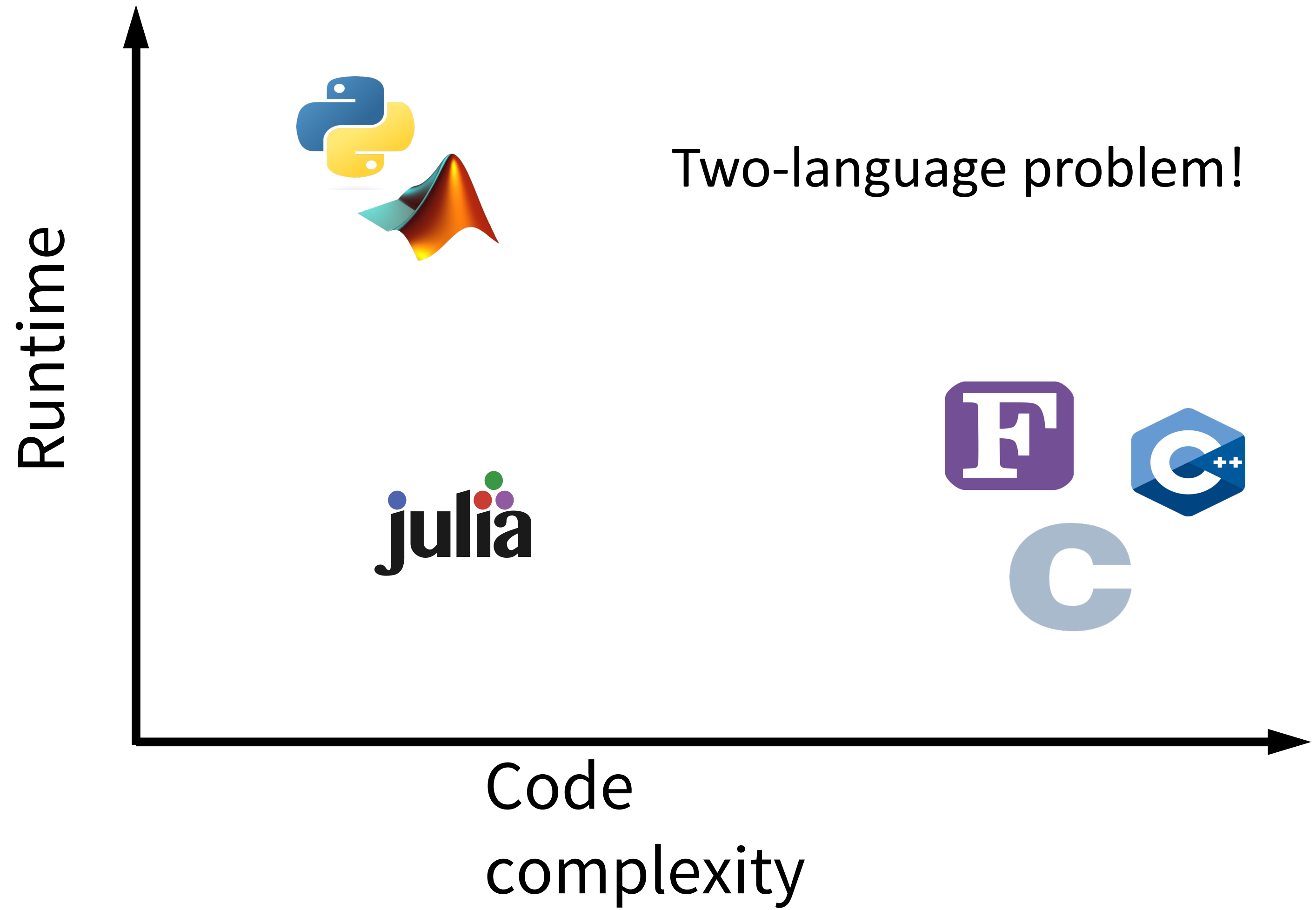
Why should you bother (as a scientist)?

You want a programming language that is:

Easy to write and read

Fast and scalable

interactive



```julia
hypotenuse(a, b) = sqrt(a^2 + b^2)
hypotenuse(3, 4)  # Output: 5.0
```

```julia
μ = mean(A)
σ = std(A)
```

```julia
Ω = [α, β, γ, ...]
😸 = 1
😺 = 1
😹 = 😺 + 😺
```

```julia
A = [1 2; 3 4]
B = [5 6; 7 8]
C = A * B
```

```julia
# elementwise multiplication
# dot syntax
D = A .* B
```

```julia
using ForwardDiff.derivative
f(x) = x^2 + sin(x)
df(x) = derivative(f, x)
df(0.3)
```

```julia
s = 0
for i in 1:10
    s += i^2
end
```

```julia
b = [1,2]
# Solve Ax = b for x
x = A \ b
```

Easy to read and write – just like math

How can Julia be **fast** ?

Short answer: Julia is a compiled language

- Compiler: LLVM

Code interpretation	Just-in-time compilation	Ahead-of-time compilation

Parse source into syntax tree

↓

Expand macros

↓

Lower syntax tree

↓ code_lowered

Type Inference

↓ code_warntype

Build LLVM code

↓

Optimize LLVM code

↓ code_llvm

Emit machine code

code_native

How can Julia be **fast** ?

Short answer: Julia is a compiled language

- Compiler: LLVM

```
julia> function sumup()
           x = 0
           for i in 1:100
               x += i
           end
           x
       end
sumup (generic function with 2 methods)

julia> @code_llvm debuginfo=:none sumup()

; Function Attrs: uwtable
define i64 @julia_sumup_12626() #0 {
top:
  ret i64 5050
}
```

Just returns the answer!
(The for loop was compiled away)

Parse source into syntax tree
↓
Expand macros
↓
Lower syntax tree
↓ code_lowered
Type Inference
↓ code_warntype
Build LLVM code
↓
Optimize LLVM code
↓ code_llvm
Emit machine code
code_native

Type system

Julia is a fully typed language

```
julia> typeof(1)
Int64

julia> typeof(1.5)
Float64

julia> typeof(0x3)
UInt8

julia> typeof(1.5f0)
Float32
```

```
julia> isthree(x) = x == 3
isthree (generic function with 1 method)

julia> isthree(3)
true

julia> isthree(3.)
true

julia> isthree(0x3)
true

julia> isthree(3f0)
true
```

Multiple dispatch

- Functions are global
- Functions with the same name are specialized on their input types

```
julia> square(x) = x^2
square (generic function with 1 method)

julia> @code_llvm debuginfo=:none square(2)
define i64 @julia_square_3025(i64 signext %0) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}

julia> @code_llvm debuginfo=:none square(2.)
define double @julia_square_3027(double %0) #0 {
top:
  %1 = fmul double %0, %0
  ret double %1
}
```

```
f( x :: Int ) = "This is an Int: $(x)"
f( x :: Float64 ) = "This is a Float: $(x)"
f( x :: Any) = "This is a generic fallback"

f(1) # "This is an Int: 1"
f(1.0) # "This is a Float: 1.0"
f("Hello") # "This is a generic fallback"
```

More examples of multiple dispatch

Specializing existing functions

```
import SpecialFunctions:gamma
using BenchmarkTools

function gamma(x :: Int)
    x > 0 ? factorial(x-1) : Inf
end

@btime gamma(15.0) # 59.827 ns
@btime gamma(15)   # 2.500 ns
```

Creating new types

```
struct DiagonalMatrix
    diag
end

Base.:*(A::DiagonalMatrix, x::AbstractVector) = A.diag .* x

DiagonalMatrix([1,2,3]) * [1,2,3] # [1,4,9]
```

Write generic functions – get specialized code

```
f(x) = x^3 - 2x

A = [1 2;3 4]
@btime f($A)          # 209.633 ns (4 allocations: 384 bytes)

A_static = @SMatrix [1 2;3 4]
@btime f($A_static) # 7.083 ns (0 allocations: 0 bytes)
```

Summary: How can Julia be **fast and readable**

- Strong type system – allows specialized compiled code
- Multiple dispatch – allows writing generic high level code
- Tackles two-language problem

Interactivity

REPL

TERMINAL

```
julia> f(x) = x^3 - 2x
f (generic function with 1 method)

julia> A = [1 2;3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> f(A)
2×2 Matrix{Int64}:
 35   50
 75  110
```

Jupyter noebooks

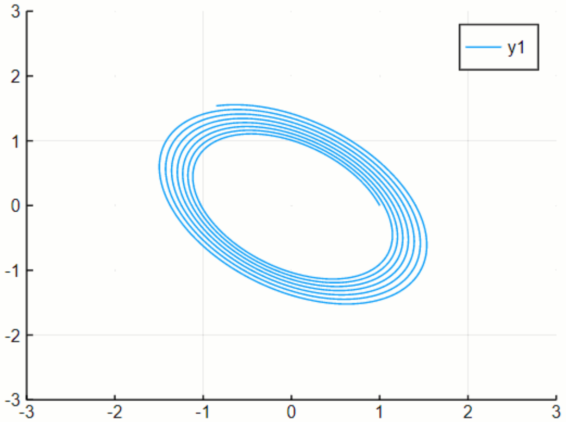
\+ Code \+ Markdown | Run All Restart Stop ··· Julia 1.9.3

```
[1]  f(x) = x^3 - 2x
     ✓ 1.9s                                         Julia
...  f (generic function with 1 method)

[2]  A = [1 2;3 4]
     ✓ 1.0s                                         Julia
...  2×2 Matrix{Int64}:
      1  2
      3  4

[3]  f(A)
     ✓ 0.6s                                         Julia
...  2×2 Matrix{Int64}:
      35   50
      75  110

[ ]                                                 Julia
```

Pluto notebooks (reactive!)

```
2×2 Array{Float64,2}:
 -0.4  -1.0
  1.0   0.41
```

```
A = [ -.4 -1
       1   .41]
```

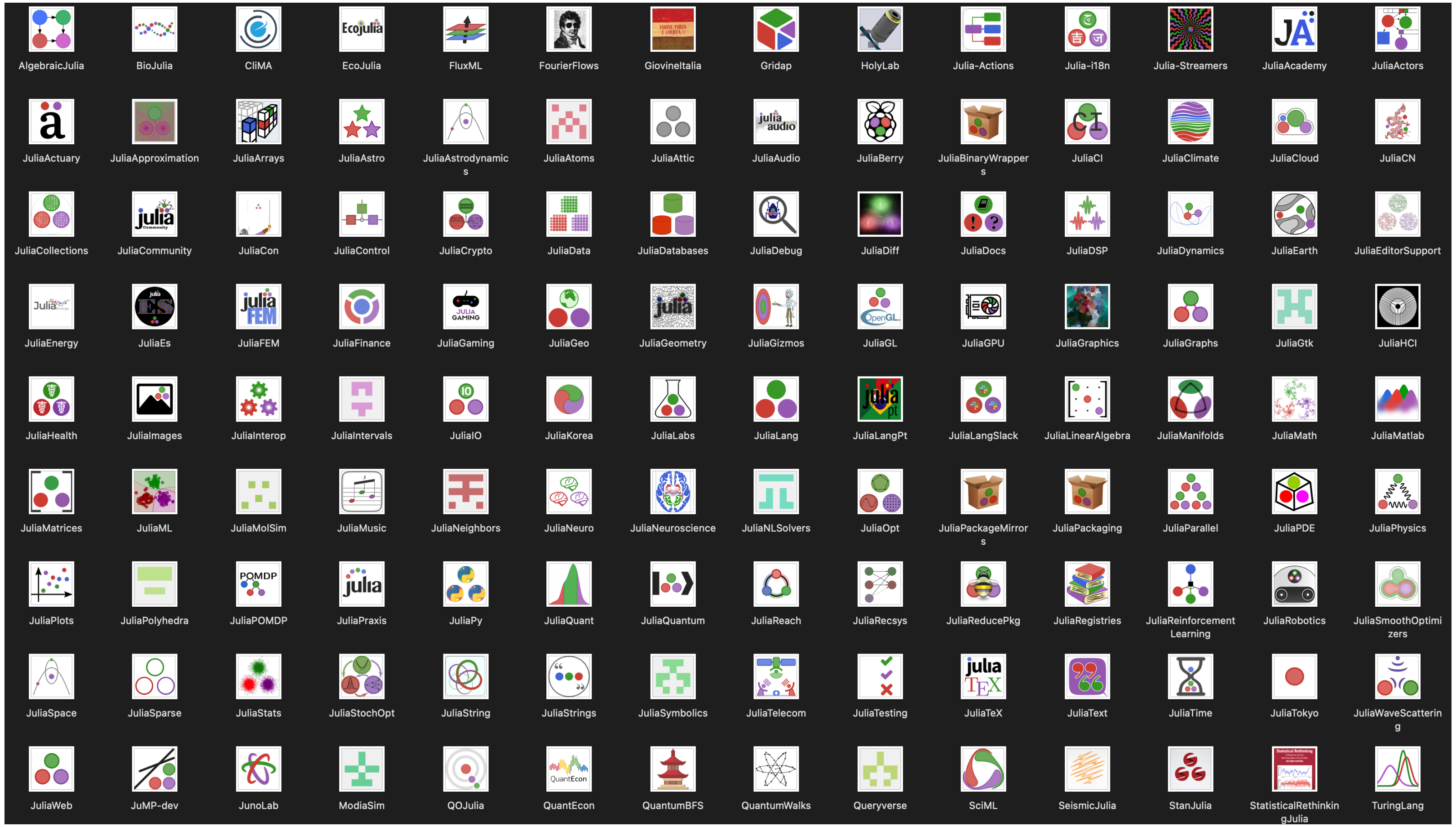



```
let
  x, y = integrate_ODE(A, 50.0)
  plot(x, y, xlims=(-3,3), ylims=(-3,3), size=
(400,300))
end
```

```
Enter cell code...
```

Julia ecosystem

- DifferentialEquations.jl
- Flux.jl / MLJ.jl / SciML
- Optim.jl
- JuliaGPU (CUDA.jl, AMD, AppleSilicon)
- JuliaDiff
- A lot more!!



Julia GitHub Organizations (https://julialang.org/community/organizations/)

But what about my undocumented 20k loc I used over the past years?? 😭

- Natively can call into C and Fortran code
- Packages for calling Python, R, Matlab, etc

Julia crash course

```julia
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end


for i in 1:10
    println(i)
end


function add(a, b)
    return a + b
end
# Shorthand
add(a, b) = a + b
```

```julia
arr = [2, 4, 8]
arr[1] # 1
arr[end] # 8
arr[1:2] # [2, 4]
arr[1:end] # [2, 4, 8]
# start:step:finish
arr[1:2:end] # [2, 8]

A = [1, 2, 3]
B = A .+ 1 # Results in [2, 3, 4]

# List comprehensions are very similar to Python
squares = [i^2 for i in 1:5] # [1, 4, 9, 16, 25]

name = "world"
greeting = "Hello, $name!" # "Hello, world!"
```

Julia cheat sheet: https://cheatsheet.juliadocs.org/

From here on…

- Official documentation (very well written)
- https://modernjuliaworkflows.github.io/
- For questions: Julia discourse and slack and stackexchange
- ChatGPT, github copilot
- Official youtube channel

Some ideas:

- Convert some of your C, Fortran, Python, etc code to Julia
- Use data visualization with Pluto

Github repository

- https://github.com/jamblejoe/SFB_PhD_Introduction-to-Julia

Notebooks with puzzles about

- Basics of Julia
- Learn about structs and matrix free multiplication and eigenvalue solve
- Explore multithreading with Monte-Carlo simulations
- Visualize the Mandelbrot set with Sliders and Plots
- Call python code from Julia