Introduction to Julia as a tool in scientific computing

Luiz M. Faria

InfoMath seminar, Sorbonne University

February 5, 2026

Table of Contents

What is Julia?

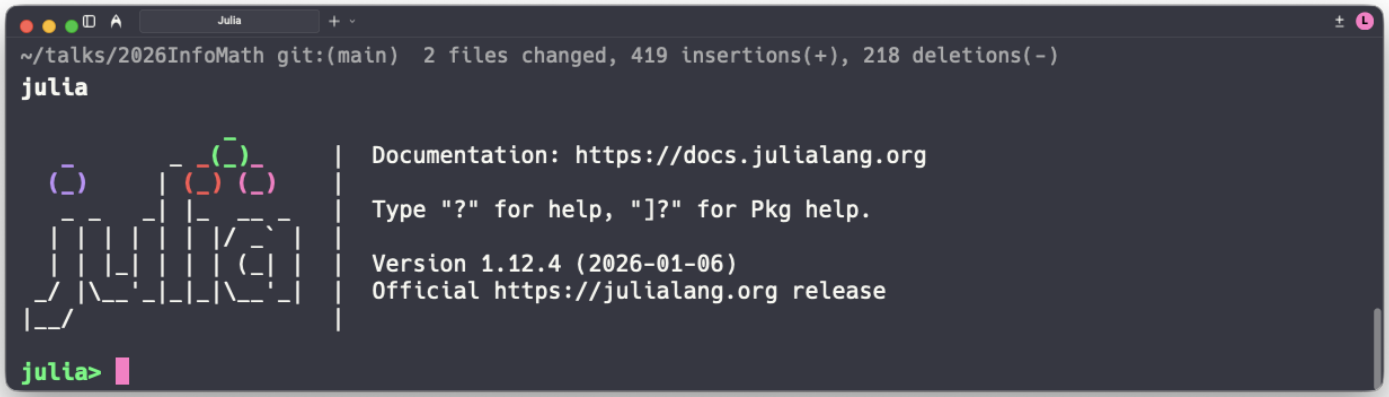
- Julia is a programming language with a focus on scientific computing
- First released in 2012, version 1.0 in 2018
- Open source (MIT license)
- Central promise: combine performance of C with usability of Python
- Addresses the two-language problem: prototype in high-level language, rewrite in low-level for performance

Using Julia

- Installation: julialang.org/downloads

```
$ curl -fsSL https://install.julialang.org | sh
```

- Basic usage: type `julia` in a terminal to open a **REPL**



- IDE: **VS Code** with `Julia` extension

Live demo: `basic_usage.jl`

- Other options: Jupyter, Pluto, emacs, vim, ...

Main language features

Core features:

- Dynamic typing
- Just-In-Time (JIT) compilation via LLVM
- Multiple dispatch as core paradigm
- Parametric types for generic programming

Nice bonuses:

- Built-in package manager (`Pkg`)
- Built-in testing framework (`Test`)
- Powerful metaprogramming (macros, generated functions)
- Native Unicode support (α, β, $\in$, ...)
- Easy interoperability with C, Fortran, Python

Typing system

Static typing (C, Java, Rust):

- Types declared at compile time: `int x = 5;`
- Compiler checks types before running

Dynamic typing (Python, Julia):

- Types belong to *values*, not variables
- `x = 5` then `x = "hello"` is valid
- Type checked at runtime

Julia's twist

Dynamically typed, but the JIT compiler infers types and generates specialized code $\Rightarrow$ flexibility + performance

Live demo: `dynamic_typing.jl`

JIT compilation

Traditional compiled (C, Fortran):

- Compile once, run many times
- Fast execution, but can slow down development cycle

Interpreted (Python, MATLAB):

- No compilation step
- Flexible, but usually slow loops

Just-In-Time (Julia):

- Compile functions on first call
- Specializes code based on argument types
- First call slow, subsequent calls fast

Live demo: `jit_compilation.jl`

Multiple dispatch

- Core paradigm in Julia: functions specialize on all argument types

Example

`+(a::Int, b::Int)` vs `+(a::Float64, b::Float64)`

Same function name, different implementations based on types

- Different from function overloading: can resolve at *runtime*
- Different from OOP (single dispatch)
- Most specific signature chosen
- Quite a powerful feature in practice

Live demo: `multiple_dispatch.jl`

Parametric types

- Types can have parameters:
  - `Vector{Float64}`, `Vector{Float32}`, `Vector{Int}`
  - `Matrix{T}`, `Array{T,N}`
- `Vector{T}` defines an infinite family of types parametrized by `T`
- Think template in C++
- Compiler generates specialized code for each `T` used

Key insight

Generic code + type specialization = no performance penalty

Live demo: `parametric_types.jl`

Summary so far

- We scratched the surface of Julia's core features
- To learn more I strongly suggest the official documentation
- There is also the Julia learning page
- And many other resources online...

Next: case study of using Julia for a scientific computing problem

Table of Contents

The N-body problem

- Classic problem in computational physics
- Simulate motion of N particles under gravitational attraction

$$m_i \ddot{\mathbf{x}}_i = -\sum_{j \neq i} \frac{G m_i m_j}{|\mathbf{x}_i - \mathbf{x}_j|^3} (\mathbf{x}_i - \mathbf{x}_j) \quad \text{for } i = 1, \ldots, N$$

- Focus on the force computation

$$\mathbf{F}_i = -\sum_{j \neq i} \frac{G m_i m_j}{|\mathbf{x}_i - \mathbf{x}_j|^3} (\mathbf{x}_i - \mathbf{x}_j) \quad \text{for } i = 1, \ldots, N$$

- Naive algorithm: double loop over all particles $\Rightarrow \mathcal{O}(N^2)$ complexity
- Better algorithms exist (Barnes-Hut, FMM) but far more complex
- Still need fast force computation kernel

Why this benchmark?

- Very simple to describe

---

function `compute_forces`$(\mathbf{x}, m)$**:**

- **for** $i = 1$ **to** N **do**
  - **for** $j = 1$ **to** N, $j \neq i$ **do**
    - $\mathbf{a}_i \mathrel{-}= G m_j \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|^3}$
- **return a**

- Stress ability to efficiently handle instructions within the language
- $\mathcal{O}(N^2)$ flops with $\mathcal{O}(N)$ bytes allows for some optimizations

Why this benchmark?

- Very simple to describe

---

function `compute_forces`$(\mathbf{x}, m)$**:**

- **for** $i = 1$ **to** N **do**
  - **for** $j = 1$ **to** N, $j \neq i$ **do**
    - $\mathbf{a}_i \mathrel{-}= G m_j \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|^3}$
- **return a**

- Stress ability to efficiently handle instructions within the language
- $\mathcal{O}(N^2)$ flops with $\mathcal{O}(N)$ bytes allows for some optimizations

A word of caution

While one may be tempted to write the algorithm above in the language of linear algebra (e.g., using matrix operations), doing so is usually *not* the most efficient approach for this problem!

Implementations

Compare three implementations of the N-body force computation:

- C

  Live demo: `nbody.c`

- Python

  Live demo: `nbody.py`

- Julia

  Live demo: `nbody.jl`

Goal is *not* to say "X is better Y", but to show differences in performance

Inspired by https://github.com/ahbarnett/floatingspeed/

Performance comparison

Language	Time (s)	vs C	Gpair/s
C	0.13	1×	0.77
Julia	0.14	1.07×	0.72
Python (loops)	21	161×	0.005

Takeaway: Julia can achieve performance comparable to C

Next:

- How to structure larger projects?
- How to create abstractions?
- Can we make the code faster?

Using Julia (revisited)

- Scripting: write code in `.jl` files and run with `julia myscript.jl` or `include("myscript.jl")` in REPL
- Projects: put code inside a `module` and add some metadata files to help manage dependencies

</> Live demo: `create_pkg.jl`

Goal: move from single-file scripts to well-structured packages.

Creating a package

- Use built-in package manager `Pkg`
- Create new package skeleton with `Pkg.generate("MyPackage")`
- Main code goes in `src/MyPackage.jl`
- Tests go in `test/runtests.jl`
- Manage dependencies with `Pkg.add("DependencyName")`
- Activate project environment with `Pkg.activate(".")`

Packages for package generation

In practice people use tools like `PkgTemplates.jl` or `Bestie.jl` to automate some of the boilerplate setup (e.g. CI, documentation, tests, etc.).

Structuring the N-body code

```
function compute_forces!(
        ax::Vector{Float64}, ay::Vector{Float64}, az::Vector{Float64},
        x::Vector{Float64}, y::Vector{Float64}, z::Vector{Float64},
        m::Vector{Float64}, G::Float64 = 1.0
    )
    # ...
end
```

1. Physical and logical restructuring of the code
2. Make types parametric for flexibility
3. Optimize for performance

```
struct GravitationalSystem
    positions::Matrix{Float64}
    acceleration::Matrix{Float64}
    masses::Vector{Float64}
    G::Float64
end

function compute_forces!(sys::GravitationalSystem)
    # Compute gravitational forces and update accelerations
    # ...
end
```

→

```
struct GravitationalSystem{T <: Real}
    positions::Matrix{T}
    acceleration::Matrix{T}
    masses::Vector{T}
    G::T
end

function compute_forces!(sys::GravitationalSystem)
    # Compute gravitational forces and update accelerations
    # ...
end
```

→

```
struct GravitationalSystem{T <: Real}
    positions::Matrix{T}
    acceleration::Matrix{T}
    masses::Vector{T}
    G::T
end

function compute_forces!(sys::GravitationalSystem{T}) where {T}
    N = pick_vector_width(T) # register width
    # Compute gravitational forces and update accelerations
    # ...
end
```

1. Organize
2. Generalize
3. Optimize

Live demo: `gravitational_system.jl`

Going faster

The simple implementation is not bad, but we can do better

- ✓ Memory layout and allocations
- ✗ Use of vectorized instructions (SIMD)
- ✗ Parallelization (multi-threading)

SIMD instructions

Single **I**nstruction, **M**ultiple **D**ata

Scalar: $\boxed{a_1} + \boxed{b_1} = \boxed{c_1}$

SIMD: $\boxed{a_1}\boxed{a_2}\boxed{a_3}\boxed{a_4} + \boxed{b_1}\boxed{b_2}\boxed{b_3}\boxed{b_4} = \boxed{c_1}\boxed{c_2}\boxed{c_3}\boxed{c_4}$

In Julia:

- `@simd` hints
- `@turbo` (LoopVectorization.jl)
- Explicit SIMD via `SIMD.jl`

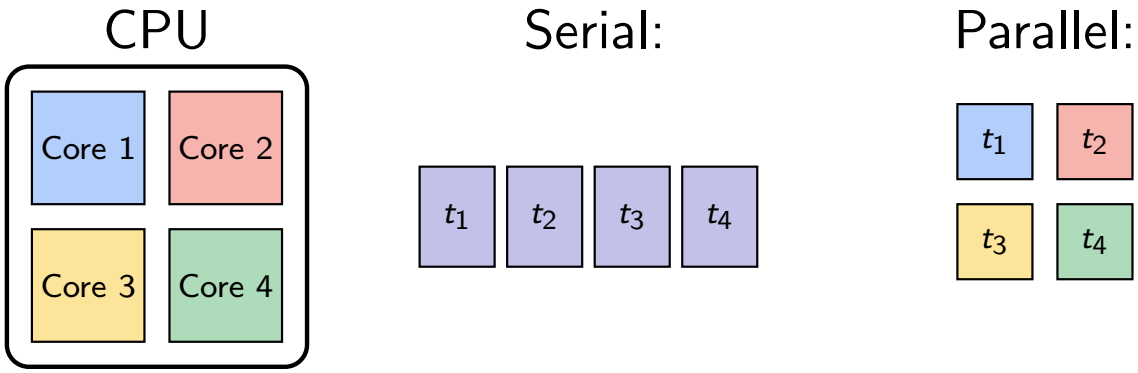
Hardware (processor architectures)

- x86-64: SSE, AVX, AVX2, AVX-512, ...
- ARM: NEON, SVE, ...
- ...

Live demo: `simd.jl`

Multi-threading

Execute work in parallel across multiple CPU cores



In Julia:

- Start with `julia -t auto` or `julia -t 8`
- `Threads.@threads` for loop parallelism
- `@spawn` / `@sync` for task-based parallelism

Considerations

- Race conditions: use locks or atomic operations
- Overhead: threading has a cost, use for large workloads
- N-body force computation is embarrassingly parallel

Optimizations

Next:

- Try to vectorize N-body force computation with SIMD
- Add multi-threading to parallelize over particles

Live demo: `gravitational_system.jl`

Version	Gpairs/s	Speedup
Julia (plain)	0.65	1×
Julia (SIMD)	1.5	2.3
Julia (SIMD + threads)	8.58	12.4

Hardware

- Vector register width: 128-bit
- Number of threads used: 6

Bonus: reduced precision

- Our code is already generic over floating point types
- So does it work with e.g. `Float32`? `Float16`? Faster?

Live demo: `bench.jl`

Version	Gpairs/s	Speedup
Julia (SIMD + threads + `Float64`)	8.58	1×
Julia (SIMD + threads + `Float32`)	17.62	2.05
Julia (SIMD + threads + `Float16`)	34.89	4.06

Bonus: reduced precision

- Our code is already generic over floating point types
- So does it work with e.g. `Float32`? `Float16`? Faster?

Live demo: `bench.jl`

Version	Gpairs/s	Speedup
Julia (SIMD + threads + `Float64`)	8.58	1×
Julia (SIMD + threads + `Float32`)	17.62	2.05
Julia (SIMD + threads + `Float16`)	34.89	4.06

A word of caution

- Effect of round-off seeing much earlier in e.g. `Float32`, `Float16`
- Stability considerations may require algorithmic changes
- Mixed-precision algorithms are an active research area

Table of Contents

Summary

Explored how Julia can be used for number-crunching tasks

. Takeaways:

- ✓ Possible to write fast code in Julia without leaving the language
- ✓ Key features: JIT compilation, multiple dispatch, parametric types
- ✓ Implementation matters: data layout, SIMD, parallelism, ...
- ✓ Easier to learn and use than many traditional compiled languages
- ✓ An interesting language choice for scientific computing projects

Where Julia falls short

I talked about some strengths of Julia, but it's not perfect. Some pain points to be aware of:

- ✗ Latency of JIT compilation can be annoying for scripting workflow
- ✗ External tooling ecosystem still maturing (debuggers, profilers, ...)
- ✗ Type instabilities are a common source of performance issues
- ✗ Lack of strict interface makes it hard to enforce API contracts
- ✗ Hard to compile to a standalone binary (improving!)

Suggestion: try it out on a small project and draw your own conclusions!

https://julialang.org/

Questions?