Introduction to Julia

Second BioMat4CAST Training

Fredrik Hedman, Noruna

2024-09-24

Schedule

Programming in Julia

- Introduction to Julia (15 min)
- Julia Syntax and Basic Concepts (30 min)
- Ecosystem and Packages (15 min)
- Scientific Computing in Julia (30 min)

Julia Tutorial

- TBD

Julia a brief history: what?

What is it?

- A high-level, high-performance programming language
- Designed specifically for technical and scientific computing
- Created in 2009 by Bezanson, Karpinski, Shah and Edelman
- Bridge the gap between high-level languages used for prototyping (like Python or MATLAB) and low-level languages used for performance-critical code (like C or Fortran)
- Julia aims to be
  - Fast
  - Dynamic
  - Reproducible
  - Composable
  - General
  - Open source

Julia a brief history: timeline

History of Julia

- Key milestones in Julia's history include:
  - 2012: Julia was officially unveiled to the public
  - 2014: The first JuliaCon conference was held
  - 2018: Julia 1.0 was released
  - 2024: Julia 1.10.5 is the latest release from Aug 27
- Popular in various fields, including data science, machine learning, and scientific computing

Julia design philosophy

Design philosophy

Combining the ease of use of dynamic languages with the performance of statically-typed compiled languages

Aim at a "no compromises" language

As easy to use as Python, as fast as C, as capable for statistics as R, as natural for linear algebra as MATLAB, and as good at gluing programs together as Perl.

Julia key aspects

Key aspects

- Performance: Julia aims to provide performance comparable to low-level languages like C and Fortran.
- Ease of use: The language is designed to be easy to write and read, with a syntax that closely resembles mathematical formulas.
- Dynamic typing: Julia uses a dynamic type system, but also allows optional type annotations for clarity and optimization.
- Multiple dispatch: This is a core feature of Julia, allowing function behavior to be defined across many combinations of argument types.

Julia key aspects

Key aspects contd.

- Designed for parallelism: The language is well-suited for parallel and distributed computing.
- Extensibility: Julia's type system and multiple dispatch mechanism allow for easy extension of the language.
- Interoperability: Julia can easily call C and Fortran libraries without wrappers or special APIs.
- Powerful metaprogramming capabilities, including Lisp-like macros.
- Open-source: Julia is developed as an open-source project with a permissive license.

Key features for scientific computing: performance

High performance

Julia is designed to deliver high performance comparable to low-level languages like C and Fortran:

- It uses just-in-time (JIT) compilation to generate optimized native machine code at runtime.
- Julia programs can achieve near-C performance without sacrificing high-level productivity.
- It combines the ease of use of dynamic languages with the speed of statically-typed compiled languages

Key features for scientific computing: easy

Ease of use

Julia provides a syntax that is easy to learn and use for scientific computing:

- It has a syntax similar to MATLAB and Python, making it accessible to scientists and engineers.
- Julia is dynamically typed and supports interactive use through REPLs and notebooks.
- It allows writing clear, high-level code that closely resembles mathematical formulas.

Key features for scientific computing: libraries

Extensive scientific libraries

Julia has a rich ecosystem of libraries and tools for scientific computing:

- It comes with built-in support for linear algebra, including a preinstalled LinearAlgebra.jl package.
- There are mature plotting libraries like Plots.jl and Makie for data visualization.
- Julia offers powerful packages for differential equations, numerical integration, symbolic computation, and more.
- Its package manager makes it easy to install and use scientific libraries.
- Supports parallel computing paradigms like multithreading and distributed computing
- Supports GPU programming

Jupyter installation

Installation and setup of jupyter. To install Jupyter using conda, follow these steps:

- Install Anaconda or Miniconda (`https://docs.anaconda.com/miniconda/`)
- Anaconda comes with Jupyter pre-installed, while Miniconda requires manual installation.
- Open a terminal and create a new conda environment (optional but recommended)

```
conda create --name julia_env python=3.11
conda activate julia_env
conda install -c conda-forge notebook
jupyter notebook --version
jupyter notebook
```

Julia Installation

Installation and setup of julia

- Installation instructions found at `https://julialang.org/downloads/`
- Use the ‘juliaup‘ installation manager to keep it up to date
- Do not use the version of "Julia" shipped by unix package managers
- Start julia

```
julia --version
julia
```

- Install the ‘IJulia‘ package in the julia REPL

```
using Pkg
Pkg.add("IJulia")
using IJulia
notebook()
```

Julia REPL

The REPL allows users to enter Julia code and immediately see the results.

REPL prompt modes

- Julian mode: The default mode, indicated by ‘julia>‘. Here, you can enter and execute Julia code.
- Help mode: Activated by typing ‘¿, it provides documentation for functions, modules, and types.
- Shell mode: Accessed by typing ‘;‘, it allows running system shell commands.
- Package mode: Entered by typing ‘]‘, it’s used for managing Julia packages.
- Enter backspace on an empty line to get back to the Julian mode.

Example

Atomic masses

```
function molecular_mass(formula)
    mass = 0.0
    for (atom, count) in formula
        mass += atomic_masses[atom] * count
    end
    return mass
end

atomic_masses = Dict("H" => 1.008, "C" => 12.011, "O" => 15.999)
water = Dict("H" => 2, "O" => 1)
methanol = Dict("C" => 1, "H" => 4, "O" => 1)

println("Mass of water: $(molecular_mass(water)) g/mol")
println("Mass of methanol: $(molecular_mass(methanol)) g/mol")
```

Ecosystem of packages

Julia's package ecosystem contains over 10,000 registered packages in the General registry (`https://julialang.org/packages/`)

Example

```
using Pkg
Pkg.add("Unitful")
using Unitful

# Convert temperature from Kelvin to Celsius
T_kelvin = 300.0u"K"
T_celsius = uconvert(u"°C", T_kelvin)
println("$T_kelvin is equal to $T_celsius")
```

Just-In-Time (JIT) compilation

- Dynamically compiles code to native machine code at runtime, just before it is executed
- Combines the flexibility of an interpreted language with the performance of compiled code
- Can introduce some initial latency when functions are first called
  - Latency often called "time to first execution" (TTFX)
- Use tools like precompilation and the PackageCompiler to reduce TTFX

Type inference

```
struct Atom
    symbol::String
    eps::Float64  # Depth of potential well
    sigma::Float64  # Distance at which potential is zero
end

function lj_potential(a1::Atom, a2::Atom, r::Float64)
    eps = sqrt(a1.eps * a2.eps)
    sigma = (a1.sigma + a2.sigma) / 2
    return 4 * ((sigma/r)^12 - (sigma/r)^6)
end
```

Type inference (contd)

```
Ar = Atom("Ar", 0.997, 3.40)
Kr = Atom("Kr", 1.320, 3.65)

r = 4.0  # Distance in Angstroms
println("LJ potential for Ar-Ar at $r Å:
         $(lj_potential(Ar, Ar, r)) kJ/mol")
println("LJ potential for Ar-Kr at $r Å:
         $(lj_potential(Ar, Kr, r)) kJ/mol")
```

Schrödinger equation for a particle in a box

```
using LinearAlgebra

function particle_in_box(n, L)
    H = zeros(n, n)
    for i in 1:n
        H[i,i] = (i^2 * ^2) / (2 * L^2)
    end
    return eigvals(H), eigvecs(H)
end
```

Schrödinger equation for a particle in a box

```
n = 10  # Number of basis functions
L = 1.0  # Box length
energies, wavefunctions = particle_in_box(n, L)
println("First 3 energy levels: $(energies[1:3])")
```

Summary and Conclusion

Let's recap the key points:

1. Julia's design philosophy: Julia was created to address the "two-language problem" in scientific computing
2. Performance: Julia's just-in-time compilation and multiple dispatch system allow for C-like speeds while maintaining a high-level syntax
3. Ecosystem: Julia has a rich ecosystem of packages for scientific computing, data handling, visualization, and domain-specific applications, making it suitable for a wide range of scientific tasks.
4. Parallel computing: Julia offers native support for multi-threading, distributed computing, and GPU acceleration, enabling efficient use of modern hardware for complex simulations.