Brief overview of the main features of Julia

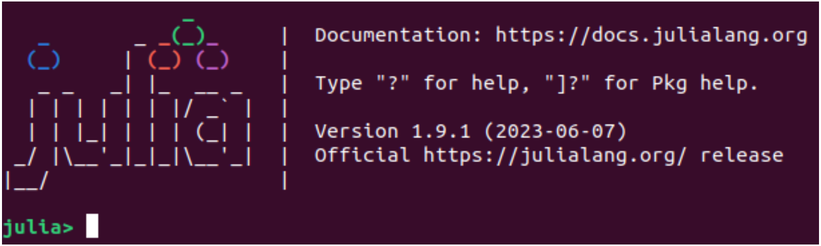
A 21st century programming language for scientific computing

Dr. Alberto F. Martín - `alberto.f.martin@anu.edu.au`
School of Computing, Australian National University, Canberra, 28^{th} Nov 2023

Outline - Overview of Julia features

- The Julia programming language
- Why is Julia great?
- The two language problem
- Julia main features
- `map`, anonymous functions, and do-block syntax
- Some practical aspects

Julia is a 21st century, open-source, multi-platform, high-level, interactive, **high-performance** programming language for technical computing



The Julia REPL

- Developed at MIT by J. Bezanson, A. Edelman, S.Karpinski, V. Shah
- First released in 2012
- First stable release (i.e., v1.0) in 2018
- Current release is 1.9.4 (Nov, 2023)
- High momentum, thriving community ($\sim 10^4$ packages registered)
- $\sim$ two minor releases per year (+3-5 patch releases per minor)

Why is Julia great for scientific computing?

- It is fast – It is indeed **very fast**!
  - Good performance on par with `C/C++` or `Fortran`
  - No need for vectorization as, e.g., in Python (`for` loops are fast!)
  - Click here for an independent benchmark of Julia vs other languages
- It has a friendly syntax
- Very easy to install
  - Cumbersome tools such as `automake` or `CMake` no longer required
- Solves the two-language problem (more on next slide)
  - Easy to prototype new algorithms that are fast out-of-the-box
  - Remarkable balance among expressivity/productivity/performance
- Support for Unicode and LaTeXcharacters (not just a cosmetic feature)
  - Example: bye bye `alpha` or `beta` as variable names, welcome `α` and `β`

Two-language problem

Julia solves the **two-language** problem!

- One language to prototype, another for production
- One language for users, another for under-the-hood (developers)
- Prototype/interface language (typically interpreted):
  - Easy to learn and use
  - Interactive
  - Productive
  - But ... **slow!**
  - Examples: `Python`, `Matlab`, `R`, ...
- Production/developers language (typically compiled):
  - **Fast!**
  - But ... **complicated/verbose/viscous/non-interactive!**
  - Examples: `C`, `C++`, `Fortran`, `Java`, ...
- Many state-of-the-art finite element libraries suffer from the two-language problem (e.g., Fenics, Firedrake, deal.II, OpenFOAM, ...)

How all these can be fulfilled? Julia features at a glance

- Just-In-Time (JIT) compilation (*compilation occurs at run-time!*)
- **Dynamic typing** and **type inference**
- **Multiple type dispatching** and **specialization**
- Garbage collection (memory leak free)
- Extensible design, rich/expressive built-in abstract types/interfaces (e.g., `AbstractArray{T,N}`)
  - User-defined types are as fast and compact as built-ins
  - Supports **type parameterization** (similar to `C++` templates)
- Metaprogramming (Julia code can be generated using Julia)
- Designed for parallelism and distributed computation
  - Only JIT-compiled language in the Petaflop club (HPCWire, 2017)
- Remarkable interoperability with other languages (e.g., `Python` and `C`)
- Fantastic built-in automated package manager. Rich package ecosystem

CAVEAT: with great power comes great responsibility!
(*clueless-written Julia code can perform very poorly*)

Just-in-time (JIT) compilation

- In Julia, the first call to a function is (significantly) slower than subsequent calls **within the same REPL session**
- Known as *first call latency* (a.k.a. *time to first execution*)
- In production environments (e,g., HPC cluster), can be addressed with the use of pre-compilation techniques (`PackageCompiler.jl` package)
- **Reduction of first call latency is a priority for core developers**. Click here for an interesting article on current state of things and future plans

```
julia> using Gridap
julia> @time model=CartesianDiscreteModel((0,1,0,1),(10,10))
  1.558585 seconds (1.74 M allocations: 102.560 MiB,
                    1.89% gc time, 99.30% compilation time)
CartesianDiscreteModel()
julia> @time model=CartesianDiscreteModel((0,1,0,1),(10,10))
  0.000297 seconds (392 allocations: 41.031 KiB)
CartesianDiscreteModel()
```

Dynamic typing and type inference

- Julia is a **dynamically typed language** (*programmer not forced to declare types of variables*) – "It feels like an scripting language"
- Although dynamically typed, it is still **a compiled language**, i.e., native machine code is ultimately generated prior to actual execution
- Compilation happens on first touch at run-time – JIT compilation
- JIT compiler needs the types of all variables to generate machine code
- To this end, Julia uses **type inference** (prior to compilation):
  - When applied to function calls, type inference determines types of all intermediate results and outputs from types of input arguments
  - It's a symbolic process, i.e., **type inference analyzes flow of types** (as opposed to data flow, which is unknown until code execution)
  - Can be introspected with, e.g., `@code_warntype` macro

```julia
function square(x)
  x*x
end
@code_warntype square(3)   # Run type inference
@code_warntype square("3") # Run type inference
```

Multiple type dispatching and specialization

- In Julia, functions are first-class citizenships separated from objects
- A function (can) have many different methods (**multi-method functions**)
  - Methods of a function queried with the `methods` built-in function
- The methods of a function differ in the number of parameters and/or their types (parameters of a method can optionally be type annotated)
- On a function call, **multiple type dispatch** is the process of deciding which method to call from the set of methods of a function
- This decision is based on the types of **all** function arguments
- Methods act as a sort of generic template to be specialized given the type of the arguments
- Specialized code can be seen with `@code_llvm` (intermediate low level representation) and `@code_native` (native machine code)

JuliaCon 2019 presentation on the subject by Stefan Karpinski
The Unreasonable Effectiveness of Multiple Dispatch

Multiple type dispatching and specialization

```julia
foo(bar) = ...                          # Method declaration statement
foo(bar::Integer) = ...                 # Method declaration statement
foo(bar::Float64) = ...                 # Method declaration statement
foo(bar::String,baz::Integer) = ... # Method declaration statement
methods(foo)                            # foo is a multi-method function w/ 4 methods
```

```julia
function square(x)
  x*x
end

# Show LLVM machine code (intermediate representation)
@code_llvm square(3)
@code_llvm square("3")

# Show native machine code (the one actually executed on the CPU)
@code_native square(3)
@code_native square("hello")
```

`map`, anonymous functions, and do-block syntax

- `map` is a built-in Julia function that lets one apply a function entry-wise to the elements in a collection to return a new collection
- The function may have several arguments; in such a case, we have to provide to `map` as many collections as function arguments
- Julia allows one to define anonymous functions with the `->` syntax
- `map` and anonymous functions can be combined using do-block syntax

```
# map applied to single collection
julia> square(x)=x*x
julia> map(square, [1,2,3])
3-element Vector{Int64}:
  1
  4
  9
# map applied to two collections
julia> sum(x,y)=x+y
julia> map(sum, [1,2,3], [4,5,6])
3-element Vector{Int64}:
  5
  7
  9
```

```
# anonymous function and map
julia> map(x->x*x, [1,2,3])
3-element Vector{Int64}:
  1
  4
  9
# Julia's do-block syntax
julia> map([1,2,3]) do x
           x*x
       end
3-element Vector{Int64}:
  1
  4
  9
```

Some practical aspects

Documentation and Getting Help

- Julia is very well-documented
- Julia documentation available here
- It comprises an excellent manual. **USE IT!!!**
- Julia Discourse
  - Go-to place in order to get help
  - Read this before posting
- Julia Slack ($\sim 15,000$ members)
  - Dedicated channels (`#hpc`, `#gpu`, `#machine-learning`, ...)
- Stack Exchange and Stack Overflow not so active

Fast-track to Julia from other languages

- Although Julia has commonalities with other languages, it is NOT a clone of any other language, such as, e.g., Python or MATLAB
- A comprehensive enumeration of noteworthy differences from other languages can be found here
- For example, compared to Python or C/C++:
  - Array indexing in Julia is 1-based not 0-based
  - Julia arrays are stored in column-major order (as in Fortran)
  - Julia is NOT an object-oriented language
  - No classes (methods bundled to objects); no multiple inheritance
- Cheat Sheets available at the workshop's references page:
  - Julia Cheat Sheet
  - Matlab-Python-Julia Cheat Sheet

The Julia package manager (`Pkg.jl`)

- As opposed to, e.g., Python, Julia comes with a built-in package manager (no more `pip` , `conda` , etc.)
- It is bundled into the REPL (Pkg documentation)
- Prompt of the package manager accessed by typing `]` on the Julia REPL
- It standardizes the installation of new Julia software, and also manages **reproducible environments**
- An environment is a record of:
  - Direct package dependencies and compatibilities in `Project.toml` file
  - A full list of package dependencies (direct and indirect) and their current state (package version, `git` revision, etc.) in `Manifest.toml`
  - `Manifest.toml` is fully auto-generated, `Project.toml` mostly, but requires manual edits (e.g., to specify compatibilities)
- `Project.toml` and `Manifest.toml` combined can be shipped to third-parties to 100% reproduce the current state of the software

How to develop your Julia code?

- Mainly two different options/workflows
- Option 1: REPL-based workflow
  - Combine your editor of choice (e.g. `vim`) and the REPL
  - **BIG WARNING**: Use `Revise.jl` package!
  - You can install `Revise.jl` in the global environment, and all other environments will inherit it
- Option 2: Use VSCode IDE with Julia extension (**this workshop**)
  - Matlab-ish experience (e.g., workspace browser, debugger . . .)
  - Powerful built-in debugger. Users' guide here
  - Native support for Jupyter notebooks (e.g., JuliaTutorials)
  - Embedded results (e.g., plots, profiler, databases)
  - Supports remote development (e.g., on Gadi) with SSH extension
  - Collaborative sessions: A "Google Docs" experience with integrated audio and text chat

Some final important resources

- Style guide for writing Julia codes available here
- Comprehensive list of performance tips available here
- `BenchmarkTools.jl` provides tools for statistical measurement of code performance and memory footprint (e.g. `@btime` macro)
- Julia provides built-in profiling capabilities through `Profile` module (Documentation available here)
- The package `ProfileView.jl` can be used to visualize profile data using the so-called FlameGraphs
  - Click here for an explanation of color code map