Julia: A Fresh Approach to Numerical Computing

Mosè Giordano
@giordano  m.giordano@ucl.ac.uk

Knowledge Quarter Codes Tech Social

October 16, 2019
Julia’s Facts

- **v1.0.0** released in 2018 at UCL
- Development started in 2009 at MIT, first public release in 2012
- Julia co-creators won the 2019 James H. Wilkinson Prize for Numerical Software
- Julia adoption is growing rapidly in numerical optimisation, differential equations, machine learning, differentiable programming
- It is used and taught in several universities (https://julialang.org/teaching/)
Julia: come for the syntax, stay for the speed

Researchers often find themselves coding algorithms in one programming language, only to have to rewrite them in a faster one. An up-and-coming language could be the answer.

Jeffrey M. Perkel

Solving the Two-Language Problem: Julia

- Multiple dispatch
- Dynamic type system
- Good performance, approaching that of statically-compiled languages
- JIT-compiled scripts
- User-defined types are as fast and compact as built-ins
- Lisp-like macros and other metaprogramming facilities
- No need to vectorise: for loops are fast
- Garbage collection: no manual memory management
- Interactive shell (REPL) for exploratory work
- Call C and Fortran functions directly: no wrappers or special APIs
- Call Python functions: use the PyCall package
- Designed for parallelism and distributed computation
using DifferentialEquations, Measurements, Plots

\[ g = 9.79 \pm 0.02; \] # Gravitational constant
\[ L = 1.00 \pm 0.01; \] # Length of the pendulum

# Initial speed & angle, time span
\[ u_0 = [0 \pm 0, \pi / 60 \pm 0.01] \]
\[ tspan = (0.0, 6.3) \]

# Define the problem
function pendulum(du, u, p, t)
    \[
    \theta = u[1] \\
    d\theta = u[2] \\
    du[1] = d\theta \\
    du[2] = -(g/L) \times \theta
    \]
end

# Pass to solvers
prob = ODEProblem(pendulum, u0, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u0[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")

From DifferentialEquations.jl tutorial “Numbers with Uncertainties”, by Mosè Giordano & Chris Rackauckas

JuliaCon 2019 talk “The Unreasonable Effectiveness of Multiple Dispatch”: [https://www.youtube.com/watch?v=kc9HwsxE1OY](https://www.youtube.com/watch?v=kc9HwsxE1OY)
Multiple Dispatch: An Example

Define the types

```plaintext
# The abstract type `Shape`
abstract type Shape end
# Followings are subtypes of the abstract type `Shape`
struct Paper <: Shape end
struct Rock <: Shape end
struct Scissors <: Shape end
```

Define the rules of the game

```plaintext
play(::Type{Paper}, ::Type{Rock}) = "Paper wins"
play(::Type{Paper}, ::Type{Scissors}) = "Scissors win"
play(::Type{Rock}, ::Type{Scissors}) = "Rock wins"
play(::Type{T}, ::Type{T}) where {T<:Shape} = "Tie, try again"
play(a::Type{<:Shape}, b::Type{<:Shape}) =
    play(b, a) # Commutativity
```
Let’s play!

```julia
julia> play(Scissors, Rock)
"Rock wins"

julia> play(Scissors, Scissors)
"Tie, try again"

julia> play(Rock, Paper)
"Paper wins"

julia> play(Scissors, Paper)
"Scissors win"
```
Extend the game by adding a new shape

```
julia> struct Well <: Shape end

julia> play(::Type{Well}, ::Type{Rock}) = "Well wins";

julia> play(::Type{Well}, ::Type{Scissors}) = "Well wins";

julia> play(::Type{Well}, ::Type{Paper}) = "Paper wins";

julia> play(Paper, Well)
"Paper wins"

julia> play(Well, Rock)
"Well wins"

julia> play(Well, Well)
"Tie, try again"
```

https://giordano.github.io/blog/2017-11-03-rock-paper-scissors/
Metaprogramming

- Like Lisp, Julia is homoiconic: it represents its own code as a data structure of the language itself.
- Since code is represented by objects that can be created and manipulated from within the language, it is possible for a program to transform and generate its own code. This allows sophisticated code generation without extra build steps, and also allows true Lisp-style macros operating at the level of abstract syntax trees (ASTs).
- In contrast, preprocessor "macro" systems, like that of C and C++, perform textual manipulation and substitution before any actual parsing or interpretation occurs.
- Julia’s macros allow you to modify an unevaluated expression and return a new expression at parsing-time.
- Macros allows the creation of domain-specific languages (DSLs). See https://julialang.org/blog/2017/08/dsl

For more information, read the manual: https://docs.julialang.org/en/v1/manual/metaprogramming/

MP is powerful but hard: https://www.youtube.com/watch?v=mSgXWpvQEHE
Lotka-Volterra equations (predator-prey model):

\[
\begin{align*}
\frac{dx}{dt} &= ax - bxy \\
\frac{dy}{dt} &= -cy + dxy
\end{align*}
\]

You can define this problem as follows:

```julia
function lotka_volterra!(du,u,p,t)
end
```
Lotka-Volterra equations (predator-prey model):

\[
\frac{dx}{dt} = ax - bxy \\
\frac{dy}{dt} = -cy + dxy
\]

You can define this problem as follows:

```julia
function lotka_volterra!(du,u,p,t)
end
```

Or use `@ode_def` macro from `ParameterizedFunctions.jl`:

```julia
lotka_volterra! = @ode_def LotkaVolterra begin
    dx = a*x - b*x*y
    dy = -c*y + d*x*y
end a b c d
```
\begin{align*}
f & = \mathbf{ode\_def} \begin{align*}
d \mathbf{\cdot} & = a \mathbf{\cdot} \mathbf{\cdot} - \beta \mathbf{\cdot} \mathbf{\cdot} \mathbf{\cdot} \\
d \mathbf{\cdot} & = -\gamma \mathbf{\cdot} \mathbf{\cdot} + \delta \mathbf{\cdot} \mathbf{\cdot} \mathbf{\cdot}
\end{align*} \\
\text{end} & \quad a \quad \beta \quad \gamma \quad \delta
\end{align*}
Do you have code in other languages that you want to be able to use? Don’t worry!

```
julia> ccall((:exp, "libm.so.6"), Cdouble, (Cdouble,), 1.57)
4.806648193775178

julia> my_shell = ccall((:getenv, "libc.so.6"),
                              Cstring, (Cstring,), "SHELL")
Cstring(0x00007ffdf927c6b6)

julia> unsafe_string(my_shell)
"/bin/zsh"
```


**JuliaCon 2019** talk: [https://www.youtube.com/watch?v=ez-KVi0le0w](https://www.youtube.com/watch?v=ez-KVi0le0w)
Calling Other Languages (cont.)

```julia
julia> using PyCall

julia> const math = pyimport("math");

julia> math.sin(math.pi / 4) - sin(pi / 4)
0.0

julia> const np = pyimport("numpy");

julia> np.random.rand(3, 4)
3×4  Array{Float64,2}:
  0.423639  0.863076  0.164781  0.160279
  0.452385  0.368733  0.779607  0.474547
  0.139557  0.777287  0.226157  0.493904
```

If you come to Julia from another language, keep in mind the following differences:
https://docs.julialang.org/en/v1/manual/noteworthy-differences/
Best Programming Practices

- Packages are git repositories
- Testing framework in standard library
- Continuous integration with several different services (Travis, AppVeyor, Cirrus, Drone, Gitlab Pipelines, Azure Pipelines, GitHub Actions, etc...)
- Code coverage: Coveralls, Codecov
- Documentation: docstrings, doctests
- PkgEval: test all registered packages

Tutorial on how to develop Julia packages:
https://www.youtube.com/watch?v=QVmU29rCjaA
Reproducibility

- Package manager integrated with the language
- “Artifacts” (binary packages, data, etc…) treated as packages
- Reproducible environments:
  - `Project.toml`: direct dependencies and their minimum required versions
  - `Manifest.toml`: complete checkout of the environment (all “packages” with fixed versions). It allows full reproducibility
Laundry list (just so you know I know)

- Compiler latency (time to first plot)
- Better static compilation support
- Better support for immutable arrays
- What am I allowed to mutate?
- Array optimizations are too many manual in-place ops
- Need protocols ("what do I do argument?")
- Better traits (to replace big interface)
- Parser error messages are too error prone
- Macro hygiene needs a lot of work
- Incomplete notation, e.g. for N-d arrays
- Special objects: Array, String, Symbol
- map(f, [])
- missing vs. DataFrame vs. nothing vs. ...

JuliaCon 2019 talk: https://www.youtube.com/watch?v=TPuJsgyu87U
What’s Bad About Julia (cont.)

- Compilation latency can be annoying during development
- Plotting framework not exciting
- Global variables are bad
- Ecosystem still young
Platforms 1: GPU

- High-level programming without GPU experience
- Low-level programming for high-performance and flexibility
- Rich ecosystem: CUDAnative.jl, CuArrays.jl, GPUifyLoops.jl, etc...
julia> f(x) = 3x^2 + 5x + 2;

julia> A = [1f0, 2f0, 3f0];

julia> A .= f.(2 .* A.^2 .+ 6 .* A.^3 .- sqrt.(A))
3-element Array{Float32,1}:
  184.0
  9213.753
  96231.72

julia> using CuArrays

julia> B = CuArray([1f0, 2f0, 3f0]);

julia> B .= f.(2 .* B.^2 .+ 6 .* B.^3 .- sqrt.(B))
3-element CuArray{Float32,1}:
  184.0
  9213.753
  96231.72

More info in https://doi.org/10.1109/TPDS.2018.2872064
Platforms 2: TPU

- Tensor Processing Units are developed by Google for neural network machine learning
- Julia supports TPUs via https://github.com/JuliaTPU/XLA.jl
- Kernels are pure Julia code, but calls require @tpu macro
- JuliaCon 2019 talk: https://www.youtube.com/watch?v=QeG1IWeVKek
Platforms 3: WebAssembly (*experimental*)

The Julia Language

- Julia Web Interface
- Interactive Prompt
- Julia Home
- Documentation
- Issues

Color Scheme

Light

Quick Reference

For help, try one of these:

- help()
- help(function)
- apropos("string")

```
@jscall Reflect.get(this, String(sym))
end

function setproperty!(this::JS::Object, sym::Symbol, val)
    @jscall Reflect.set(this, String(sym), val)
end
typeof(Base.setproperty!())

julia> nothing
nothing

julia> window.document
JS::Object(0x00000004)

julia> window.document.body.style.backgroundColor = "#ffffff"

julia> window.document.body.style.backgroundColor = "#ffffff"

julia> window.document.body.style.backgroundColor = "#000000"

julia> window.document.body.style.backgroundColor = "#ab0000"

julia> window.document.body.style.backgroundColor = "#000000"

julia> window.document.body.style.backgroundColor = "#ffffff"

julia> window.document.body.style.backgroundColor = "#ffffff"
```

Credits: Keno Fisher on Twitter: https://twitter.com/KenoFischer/status/1158517084642582529

Mozilla awarded a grant to develop Julia support for WebAssembly
Platforms 4: FPGA (very experimental)

Credits: Keno Fisher on Twitter: https://twitter.com/KenoFischer/status/1154865907472183296
Applications: Past – Celeste.jl

Project goals:

1. Catalog all galaxies and stars that are visible through the next generation of telescopes
   - The Large Synoptic Survey Telescope (LSST) will house a 3200-megapixel camera producing 15 TB of images nightly

2. Replace non-statistical approaches to building astronomical catalogs from photometrical data

3. Identify promising galaxies for spectrograph targeting
   - Better understand dark energy and the geometry of the Universe

4. Develop and extensible model and inference procedure, for use by the astronomical community
   - Future applications might include finding supernovae and detecting near-Earth asteroids
Accomplishments:

1. Reached 1.54 petaFLOPS performance (first First Julia application to exceed 1 petaFLOPS)
   - Julia is probably the first dynamic high-level language to enter the petaFLOPS club (other languages in it: Assembly, Fortran, C/C++)
   - Code ran on 9568 Intel Xeon Phi nodes of Cori (Phase II)
   - 1.3 million threads on 650,000 KNL cores

2. Processed most of SDSS dataset in 14.6 minutes
   - Loaded and analysed 178 TB
   - Optimised 188 million stars and galaxies

3. First comprehensive catalog of visible objects with state-of-the-art point and uncertainty estimates

4. Demonstration of Variational Inference on 8 billion parameters
   - 2 orders of magnitude larger than other reported results

Discover more:

- [https://github.com/jeff-regier/Celeste.jl](https://github.com/jeff-regier/Celeste.jl)
- [JuliaCon 2017 talk](https://www.youtube.com/watch?v=uecducADM3hY)
Applications: Present – PuMaS

PharmaceUtical Modeling And Simulation

- Suite of tools for developing, simulating, fitting, and analyzing pharmaceutical models
- Bring efficient implementations of all aspects of pharmaceutical modeling under one cohesive package
- Deliver personalised treatment schedules for each individual
- Seemless integration with the rest of Julia ecosystem (Measurements.jl, JuliaDB.jl, Query.jl, etc.)
- Collaboration between Center for Translational Medicine of University of Maryland, Baltimore and Julia Computing

Talks at JuliaCon 2018: https://www.youtube.com/watch?v=KQ4Vtsd9XNw and JuliaCon 2019: https://www.youtube.com/watch?v=i8LgmT0mKmE
Collaboration between Caltech, NASA JPL, MIT, Naval Postgraduate School, funded among others by NSF: https://clima.caltech.edu/

First Earth model that automatically learns from diverse data sources

Modeling platform that is scalable and built for growth

It will need to run on the world’s fastest supercomputers and on the cloud, using both GPU and CPUs

Scalable for different resolutions, to have local and global climate

Julia chosen to ensure performance on modern heterogeneous architectures without sacrificing scientific productivity information

Talk at JuliaCon 2019: https://www.youtube.com/watch?v=gD5U_U9kZk8
Take-Home Messages

- Great **composability**: complex packages can work together
- Incremental **optimisation**: from prototype to final product step by step
  - https://mitmath.github.io/18337/lecture2/optimizing
- Julia programs are organised around **multiple dispatch**
- **Metaprogramming** capabilities
- Most of Julia is written in Julia itself
- My 2 cents: main Julia’s strength is **genericity**, which increases productivity

Got interested?

- Official website: https://julialang.org/
- List of registered packages: https://pkg.julialang.org/
- GitHub repository: https://github.com/JuliaLang/julia
- Discussion forum: https://discourse.julialang.org/
- Slack workspace: https://slackinvite.julialang.org/
Take-Home Messages

- Great **composability**: complex packages can work together
- Incremental **optimisation**: from prototype to final product step by step
  - https://mitmath.github.io/18337/lecture2/optimizing
- Julia programs are organised around **multiple dispatch**
- **Metaprogramming** capabilities
- Most of **Julia** is written in Julia itself
- My 2 cents: main Julia’s strength is **genericity**, which increases productivity

Got interested?

- **Official website**: https://julialang.org/
- **Manual**: https://docs.julialang.org/en/
- List of **registered packages**: https://pkg.julialang.org/
- **GitHub** repository: https://github.com/JuliaLang/julia
- **Discussion forum**: https://discourse.julialang.org/
- **Slack** workspace: https://slackinvite.julialang.org/
JuliaCon 2020 in Lisbon!

https://juliacon.org/2020/

Mosè Giordano (RSDG @ UCL)  Julia: A Fresh Approach to Numerical Computing  October 16, 2019