University of BRISTOL

May 2019

A Brief Introduction to Julia

And Applications to Dynamical Systems

David A.W. Barton

Engineering Mathematics, University of Bristol

Jeff Bezanson, Stefan Karpinski, and Viral B. Shah

Winners of 2019 James H. Wilkinson Prize for Numerical Software

for the creation of Julia, an innovative environment for the creation of high-performance tools that enable the analysis and solution of computational science problems.

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
- Dynamic
- Optionally typed
- Easy to use
- General
- Open source

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
  Compiled to machine code and designed to be fast by judicious choice of language features
- Dynamic
- Optionally typed
- Easy to use
- General
- Open source

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
- Dynamic
  Variables don't have to have a fixed type; easy interactive use
- Optionally typed
- Easy to use
- General
- Open source

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
- Dynamic
- Optionally typed
  Functions can have types specified but not necessary for speed
  (helps catch errors and allows multiple dispatch)
- Easy to use
- General
- Open source

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
- Dynamic
- Optionally typed
- Easy to use
  High-level syntax, a range of useful tools, and a very supportive community
- General
- Open source

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
- Dynamic
- Optionally typed
- Easy to use
- General
  Not just for science, lots of packages for general-purpose programming
- Open source

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
- Dynamic
- Optionally typed
- Easy to use
- General
- Open source
  All development on GitHub, many different contributors (quite a lot have academic backgrounds)

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
- Dynamic
- Optionally typed
- Easy to use
- General
- Open source

JuliaComputing — a consultancy firm set up by the creators of Julia

What Julia is

Headings from `julialang.org` with my comments —

- Julia is fast
- Dynamic
- Optionally typed
- Easy to use
- General
- Open source

JuliaComputing — a consultancy firm set up by the creators of Julia
NumFocus — charitable contributions via NumFocus (like NumPy, SciPy, Juypter, etc)

What Julia isn't

the answer to everything… Use the right tool for the job!

- Language is largely fixed post-v1.0 but will continue to evolve
- Ecosystem is not as large as Python/Matlab but growing rapidly
- Proliferation of packages for the same thing as people experiment
- Tooling is still developing though all key things in place
  - Profiler and development environment (Juno, Jupyter) are mature
  - Debugger is available but relatively new
  - Plotting is great but still has some rough edges (interactivity)
- Not suitable for system-level programming (use C/Rust…)
- Stylistically quite different to object-orientated programming
  - Though you can adopt many different styles

Example code — naive Newton iteration

Short functions are the typical style

```
derivative(f, x; ϵ=1e-6) = (f(x + ϵ) - f(x - ϵ)) / 2ϵ

function newton(f, x; ϵ=1e-6, itmax=10)
    i = 0
    while (abs(f(x)) > ϵ) && (i < itmax)
        x -= f(x)/derivative(f, x)
        i += 1
    end
    return x
end

my_f(x) = x*exp(x) - 2

ans = newton(my_f, 1.0)  # 0.8526055263689221
```

Multiple dispatch — naive Newton iteration

```
derivative(f, x; ϵ=1e-6) = (f(x + ϵ) - f(x - ϵ)) / 2ϵ

function newton(f, x; ϵ=1e-6, itmax=10)
    i = 0
    while (abs(f(x)) > ϵ) && (i < itmax)
        x -= f(x)/derivative(f, x)
        i += 1
    end
    return x
end

my_f(x) = x*exp(x) - 2
derivative(::typeof(my_f), x) = (1 + x)*exp(x)  # Overload!

ans = newton(my_f, 1.0)  # 0.8526055263689221
```

Generic programming

Want more precision?

- BigFloats (arbitrary precision arithmetic, built-in but slow)
- Quadmath.jl (128 bit IEEE floating point numbers)

Install the package

```
] add Quadmath
```

Run the code

```
ans = newton(my_f, Float128(1.0), ϵ=1e-20)
  # 0.852605502013725491346472414695478028
```

All compiled to efficient machine code

More generic programming

Integrate *dimensional* ODEs — Newton's law of cooling

```
using BenchmarkTools
using Unitful: W, m, K, s, kg, J
using OrdinaryDiffEq

p = (h = 10000.0W/(m^2*K),
     A = 0.0048m^2,
     m = 1kg,
     Cₚ = 4186J/(kg*K),
     T₀ = 273.15K)
f = (u, p, t) -> (p.h*p.A)/(p.m*p.Cₚ)*(p.T₀ - u)
u₀ = 300.0K
prob = ODEProblem(f, u₀, (0.0s, 1000.0s), p)
@btime sol = solve(prob, Tsit5())
  # 7.755 μs with units, 8.036 μs without; MATLAB 363 μs
```

Metaprogramming

Julia has macros that manipulate code rather than values; consider

```
a = 0.5
x = 2.3
myfunction(2a*x)  # calls myfunction with the value 2.3
@mymacro(2a*x)    # calls mymacro with the expression 2a*x
```

This enables arbitrary code transformations; for example, Interact.jl provides an interactive web interface

```
using Interact

t = range(0, stop=10, length=101)  # linspace in MATLAB
@manipulate for ω in slider(0.01:0.1:4π)
    plot(t, sin.(ω.*t))
end
```

Metaprogramming

`@macroexpand` shows what the macro does

```
children = (Widgets.OrderedDict)(:ω =>
    (ω = (Widgets.widget)(slider(0.01:0.1:4π), label="ω")))
output = begin
            tmp = ((ω,)->begin
                            plot(t, sin.(ω .* t))
                        end)
            (Widgets.map)(tmp, ω)
        end
layout = (Widgets.manipulatelayout)((Widgets.get_backend)())
(Widgets.Widget){:manipulate}(children, output=output,
                            layout=layout)
```

Macros save a lot of boilerplate code!

Domain Specific Languages

Macros enable the creation of Domain Specific Languages (DSLs) within Julia; for example ModelingToolkit.jl

```
using ModelingToolkit

# Define some variables
@parameters t σ ρ β
@variables x(t) y(t) z(t)
@derivatives D'~t  # can also handle higher derivatives

# Lorenz equations
eqs = [D(x) ~ σ*(y - x),
       D(y) ~ x*(ρ - z) - y,
       D(z) ~ x*y - β*z]
# Final ODE system (with symbolic derivatives as necessary)
de = ODESystem(eqs)
```

Don't throw away old code

Julia is great at interoperability (FFI — Foreign Function Interface)

- C/Fortran — built-in
- C++ — Cxx.jl, CxxWrap.jl
- Python — PyCall.jl
- MATLAB — MATLAB.jl
- R — RCall.jl
- Java — JavaCall.jl
- Mathematica — Mathematica.jl (needs updating)

Allows for graceful migration of code

Notable packages: DifferentialEquations.jl

A tour de force: `juliadiffeq.org` (Chris Rackauckas et al)

- Many different ODE solvers for different types of ODE; includes wrappers for existing libraries (e.g., Sundials)
- Delay differential equation solvers
- DAE solvers
- Stochastic differential equation solvers
- Routines for parameter estimation
- Sensitivitiy analysis
- Parallel Monte Carlo simulations
- Lots more…

Notable packages: DynamicalSystems.jl

`juliadynamics.github.io/DynamicalSystems.jl/latest/`
(George Datseris et al)

- Delay embeddings
- Poincarè sections
- Lyapunov exponents
- Entropies and dimension estimations
- Recurrance plots

Lots of nice tutorials and introductory material

If I had more time…

- Open Science & Reproducible Science!
- JuMP.jl — optimisation: linear, quadratic, mixed integer
- Flux.jl — Flux is the ML library that doesn't make you tensor
- LightGraphs.jl & companions — tools for networks
- Optim.jl & NLsolve.jl — nonlinear optimisation and solvers
- Turing.jl — probabilistic programming
- ForwardDiff.jl & Zygote.jl — automatic differentiation
- CUDAnative.jl — GPU programming via Julia
- Broadcasting — vectorisation done well
- Parallel & distributed computing — built in to the language

People come for the speed, but stay for the type-dispatch system