Introduction to julia

Akira Endo

London School of Hygiene & Tropical Medicine

Purpose of this talk

- What it does and why it’s worth trying
- Key concepts and distinct features compared with R
- Caveats and typical pitfalls when switching from R
- Some useful packages

Warnings

- I’m rather novice so some explanations may be inaccurate/insufficient
- I have COI (more people use Julia -> easier to collaborate)

What’s julia ?

- General-purpose programming language released in 2012
- Fast execution with just-in-time (JIT) compilation via LLVM
- Some unique features (multiple dispatch, dot-syntax, @macro)
- Quickly growing community (25M total downloads, 5000+ pkgs)
- Current ver 1.6.2 (as of 14 Jul)

Why julia?

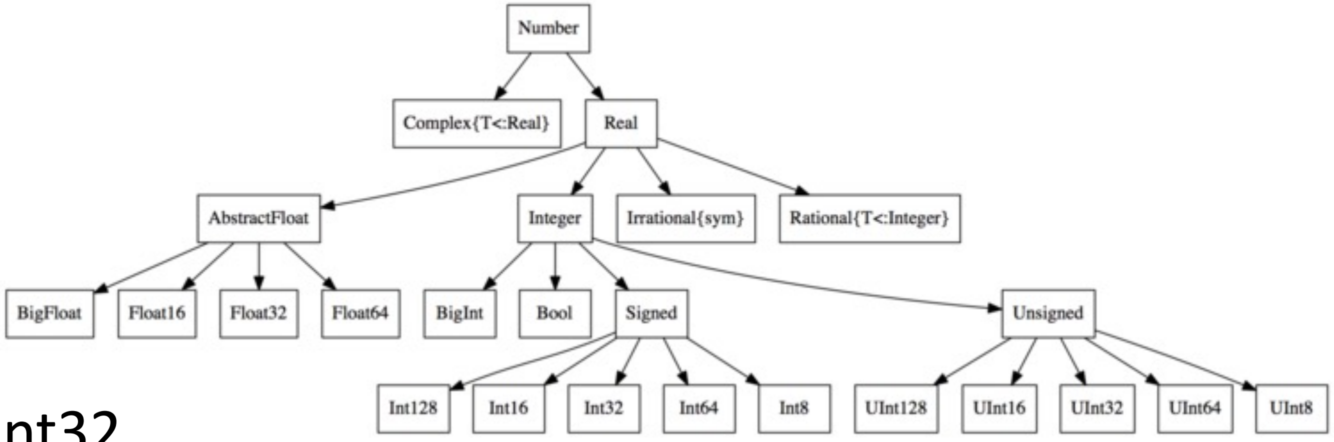
I’ve been using it for 2 years and now it’s my first-choice language

- Can avoid “two-language problem”: easy to prototype and runs fast
- Supports many modern features, less stressful in coding
- Can call R, Python, etc. within with simple syntax anyway

Execution speed

- When called for the first time, a function is JIT-compiled (can incur small overhead time) and from the second call it runs much faster
- When well written, Julia runs almost as fast as C
- Some coding style can make it slower, but still no slower than interpreter languages
- Launching Julia and loading packages have certain overhead; developing on REPL or Jupyter recommended (rather than running xxx.jl one after another)

Types and hierarchy



- Concrete types
  Bool, Float64 (double in R), Int64, UInt32,…
- Abstract types ("umbrella" for types)
  Int64 <: Signed <: Integer <: Real <: Number (<: Any)

Types can be "parameterised" by other types

- Vector{Int64} (= Array{Int64, 1}): [1, 2, 3],   Vector{Float64}: [0.1, 0.5, 0.2]
- Matrix{Real} (= Array{Real, 2}): $\begin{bmatrix} 1 & 0.1 \\ 0.9 & 2 \end{bmatrix}$
- Vector{Vector{Int64}}: [[1, 2, 3], [4, 5]]
- Vector{Any}: [1, 0.5, [1, 2, 3], "Hello World"]

Multiple dispatch

- In Julia, functions are distinguished not only by names but also by **argument types**
- Define f(x::Int64)=x+1 and f(x::Float64)=x+2:

  They coexist & you get **2** for f(1) and **3.0** for f(1.0)

  —if not specified, f(x) is just treated as f(x::Any)
- Easy to extend existing functions/pkgs:
  - Define “polar vector” as a type. PVector: [r, θ]
  - Define Base.:+(x::PVector, y::PVector)=…
  - Then sum() for PVector is ready to use
  - Likewise: override a few basic functions and PVector can be used anywhere

```
Julia: A fresh approach to technical computing.

[1]: f(x::Int64)=x+1
[1]: f (generic function with 1 method)

[2]: f(x::Float64)=x+2
[2]: f (generic function with 2 methods)

[3]: f(1)
[3]: 2

[4]: f(1.0)
[4]: 3.0
```

```
Julia: A fresh approach to technical computing.

[1]: struct PVector{T<:Number}
         r::T
         θ::T
     end
     Base.:+(x::PVector, y::PVector)=PVector(sqrt((x.r*cos(x.θ)+y.r*cos(y.θ))^2+
     (x.r*sin(x.θ)+y.r*sin(y.θ))^2),
     atan(x.r*sin(x.θ)+y.r*sin(y.θ),x.r*cos(x.θ)+y.r*cos(y.θ)))

[2]: PVector(1.,π/6)+PVector(1.,π/3)
[2]: PVector{Float64}(1.9318516525781366, 0.7853981633974483)

[3]: sum([PVector(1.,π/6),PVector(1.,π/3)])
[3]: PVector{Float64}(1.9318516525781366, 0.7853981633974483)
```

Some more convenient features

- Dot syntax (broadcasting): apply function element-wise

  sqrt.([1, 4, 9]) -> [1, 2, 3] *# sqrt(v) in R*

  min.([1, 2, 3], [2, 1, 4]) -> [1, 1, 3] *# pmin(v ,w) in R*

  [1, 2, 3] .+ [0 1 2] -> $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$

```
Julia: A fresh approach to technical computing.

[1]: m=rand(10000,10000);

[2]: using Statistics

[3]: @time mean(m)

       0.106950 seconds (126.82 k allocations: 6.948 MiB)
[3]: 0.49994340525897085

[4]: @time mean(m)

       0.046512 seconds (1 allocation: 16 bytes)
[4]: 0.49994340525897085
```

- @macro

  @time: count execution time and memory allocation

  @views: access sub-array (e.g. m[1:3, 1:2]) without copying the array

  @threads: multi-thread execution of for loop

- And more…
  - +=, *=, etc. operators (instead of x <- x + 1 in R…)
  - Supports math-y style: e.g. σ^2 as a variable name (\sigma TAB \^2 TAB)
  - List comprehension: [i^2 for i in 1:10]

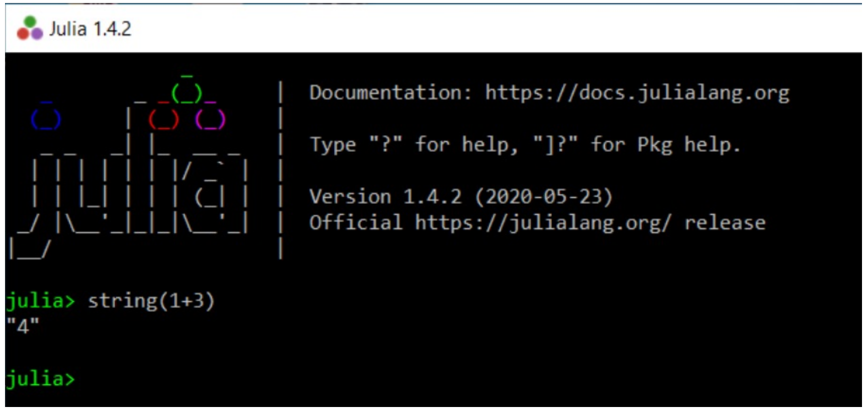
Use R/Python in Julia

RCall.jl/PyCall.jl provides simple interface to call R/Python

- @rimport to load packages
- R”” expression to directly write R code
- So you still have access to those ecosystems
- Simpler than the other way round (e.g. via JuilaConnectoR)

```
Julia: A fresh approach to technical computing.

[1]: using RCall

[2]: @rimport stats as rs

[3]: rs.rnorm(10,0,1)

[3]: RObject{RealSxp}
      [1] -0.333175027  2.028126902 -0.704663144  0.366013614 -0.354683235
      [6] -0.126341587  1.123303843  1.136265648 -0.862661693 -0.009227246

[4]: R"rpois(10, 0.5)"

[4]: RObject{IntSxp}
      [1] 0 0 1 5 0 2 0 1 0 1
```

Gradual steps to Julia



1. Install Julia

2. Try some snippets on REPL to learn basic syntax

3. Speed up core functions by JuliaConnectR (instead of Rcpp)

4. Work in Julia but call R functions via RCall wherever necessary

Pitfalls when switching from R

- 1:3+1 is 1:4, not 2:4
- Most operations only make shallow copy:
  a = [1, 2]; b = a; a[1] = 0 then b is [0, 2]
- Don’t forget dot-syntax for elementwise operation: v[1:2] .= 0
- Functions are passed-by-reference: modified arguments inside a function remains so outside. Function with names ending with ! alter arguments (e.g. push!(v, e) changes v)
- Beware memory allocation for best efficiency

Limitations of Julia

- Ecosystem is still in development
  Many packages are underdocumented/abandoned. Functionalities of some packages may be limited cf. the R/Python counterparts
- Version updates of major packages are rapid
  Package management via Pkg.jl is important; otherwise your code may not run as intended in a few months
- Slightly more intellectually intensive to code
  If you want the fastest code you need to be thinking about it throughout coding, e.g. type inference, memory allocation, etc.

Useful packages for modelling

- Distributions.jl
  Julia ver of {distr}: provides many types of distributions. Some convenient functionalities like mean(d), MixtureModel, etc.
- DifferentialEquations.jl
  Fast (~20x than {deSolve}?) and high-end DE solver. Also available in R via {diffeqr}, but native env provides more flexible usage.
- Turing.jl
  Provides interface for probabilistic programming. Supports MCMC, VI, Particle Gibbs, etc.

Quick introduction to Turing.jl

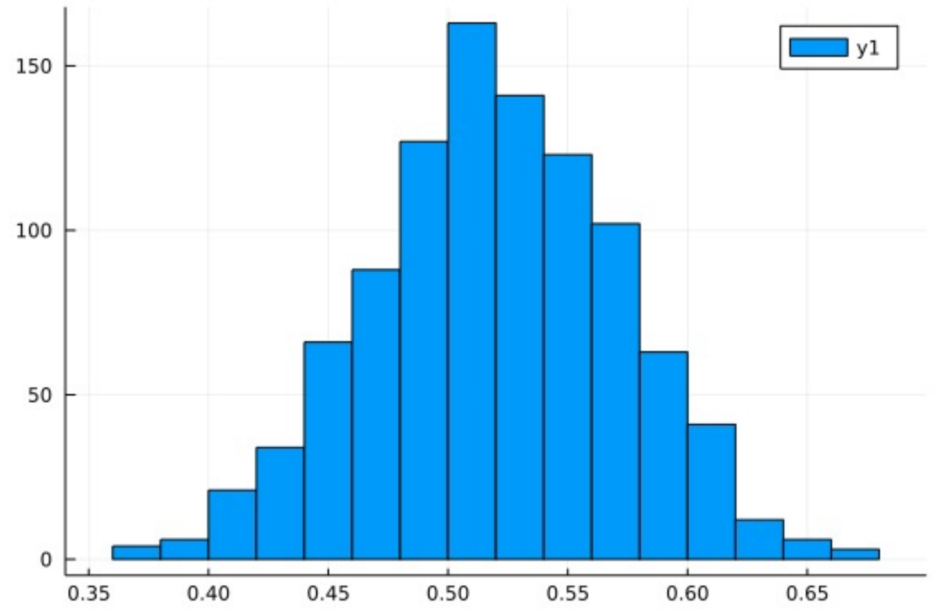
- Mainly for MCMC (MLE also supported)
- Stan-like probabilistic programming interface:

```
@model coinflip(y) = begin
    p ~ Beta(1, 1) # prior.
    N = length(y)
    for n in 1:N
        y[n] ~ Bernoulli(p)
    end
end
```

- Supported algorithms: NUTS, HMC, Gibbs, M-H, etc.
- Runs on Julia—anything runnable on Julia can be used in Turing

Minimum example (from official tutorial)

```
@model coinflip(y) = begin
        p ~ Beta(1, 1) # prior.
        N = length(y)
        for n in 1:N
                y[n] ~ Bernoulli(p)
        end
end
iterations = 1000
ϵ = 0.05
τ = 10
data = [true, false, true, true, false]

chain = sample(coinflip(data), HMC(ϵ, τ), iterations);
plot(chain[:p], seriestype = :histogram)
```



Pitfalls when using Turing.jl

- Model has to be auto-differentiable (at least by default). Functions should be able to handle AbstractFloat (not only eg Float64)
- Algorithms that don't use autodiff are rather scarce; not the best choice if your model can't be or takes too long to be autodiff'ed
- By default, the initial values are sampled from the prior, so if uninformative prior is used MCMC may not take off due to -Inf logposterior

To wrap up:

- Julia can solve two-language problem
- Some nice features including multiple dispatch
- Using Julia doesn’t necessarily mean you’re abandoning R
- Join #julia slack channel for introductory materials and discussion!