Introduction to Julia

Hayden Andersen

21st October

What is Julia?

- Julia is a programming language
- Initially created by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman
- Aims to have the simplicity of Python, the speed of C, and the functionality of Lisp

Background and Notable Uses

- First released in 2012.
- Version 1.0 was released in 2018
- Won the 2019 James H. Wilkinson Prize for Numerical Software
- Used both at NASA and at CERN
  - Used at NASA to model spacecraft separation dynamics - 15000 times faster than MATLAB
  - Used at CERN for one of the Large Hadron Collider experiments
- Many major changes over the years - is finally now in a good place to be used by everyday users!

What makes Julia good?

- Built around the concept of multiple dispatch
- High performance
- Native support for parallelism
- Optional typing/duck typing
- Strong support for metaprogramming
- Support for Unicode
- And more!

Ease of Use

Language Features

Performance Improvements

Useful Libraries for Research

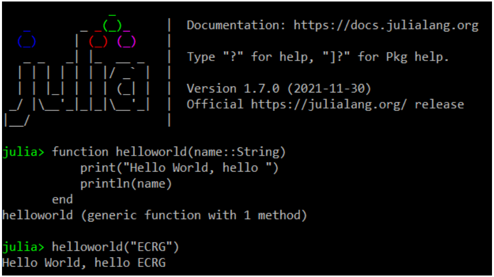
Downsides

Questions/Discussion

Ease of Use

REPL

- While Julia is a compiled language, it provides a read–eval–print loop (REPL) to interactively write code
- Similar to Lisp and Python
- Allows for on-the-fly testing of code during development



Dynamic Typing

- Similar to Python, Julia supports dynamic (or duck) typing
- Optional static typing can improve computation speed and aid multiple dispatch
- Can write full programs in Julia without using types
- The same name can refer to multiple different types throughout execution of the code

String Interpolation

- Julia supports C++ style string interpolation
- Instead of format strings, variables and computations can be directly inserted into the string

```
'Using dataset {}, seed {}, and the {} algorithm with
population size {}, {}-tournament selection, {} elitism,
{} crossover and {} mutation'.format(dataset, seed,
algorithm, population, tournament, elitism, crossover,
mutation)
```

String Interpolation

- Julia supports C++ style string interpolation
- Instead of format strings, variables and computations can be directly inserted into the string

```
'Using dataset {}, seed {}, and the {} algorithm with
population size {}, {}-tournament selection, {} elitism,
{} crossover and {} mutation'.format(dataset, seed,
algorithm, population, tournament, elitism, crossover,
mutation)
```

```
"Using dataset $dataset, seed $seed, and the $algorithm
algorithm with population size $population, $(tournament)-
tournament selection, $elitism elitism,
$crossover crossover and $mutation mutation"
```

Other Usability Bonuses

- Short circuit evaluation
  `conflicting(rule, v) && (return true)`
  `best isa Nothing || (fitness = best.fitness)`
- 1-indexed
- Single line function definitions
  `addtwo(a) = a + 2`
- Easy vectorization of functions
  `addtwo.(somelist)`

Language Features

Multiple Dispatch

- The main concept/core paradigm behind Julia!
- Each function can have an arbitrary number of method implementations, each operating on different types
- Julia decides which method to run as the most specific method based on parameter types
- Methods can be typed as abstract types
- Allows for a huge amount of code reuse/code sharing.
- After some time really changes your coding style

Multiple Dispatch

- Many languages we are used to use single dispatch (OOP)
  `Cat bella = Cat(); cat.meowAt(dog)`
- It can take some getting used to the style of multiple dispatch
  `Cat bella = Cat(); meowAt(cat, dog)`
- New method definitions can be added at any time - all code that already used that function now works with the new method!
- Does not need to be inside the class like it would have to be for OOP
- I highly recommend watching "The Unreasonable Effectiveness of Multiple Dispatch" https://www.youtube.com/watch?v=kc9HwsxE1OY

Interfaces

- Thanks to multiple dispatch, Julia provides some easy to implement interfaces!
- To make a type iterable, only have to implement `iterate(iter)` and `iterate(iter, state)`

```
julia> struct Squares
           count::Int
       end

julia> iterate(s::Squares, state=1) = state > s.count ? nothing : (state*state, state+1)
iterate (generic function with 232 methods)

julia> for i in Squares(4)
           println(i)
       end
1
4
9
16
```

Interfaces

- We can also index a type by implementing `getindex(X, i)`, `setindex!(X, v, i)`, `firstindex(X)`, and `lastindex(X)`

```
julia> function getindex(S::Squares, i::Int)
           1 <= i <= S.count || throw(BoundsError(S, i))
           return i*i
       end
getindex (generic function with 218 methods)

julia> getindex(S::Squares, I) = [S[i] for i in I]
getindex (generic function with 218 methods)

julia> firstindex(S::Squares) = 1
firstindex (generic function with 17 methods)

julia> lastindex(s::Squares) = s.count
lastindex (generic function with 14 methods)

julia> Squares(6)[3:end]
4-element Vector{Int64}:
  9
 16
 25
 36
```

Interfaces

```
julia> struct SquaresVector <: AbstractVector{Int}
           count::Int
       end

julia> size(S::SquaresVector) = (S.count,)
size (generic function with 101 methods)

julia> IndexStyle(::Type{<:SquaresVector}) = IndexLinear()
IndexStyle

julia> getindex(S::SquaresVector, i::Int) = i*i
getindex (generic function with 217 methods)

julia> s = SquaresVector(4)
4-element SquaresVector:
  1
  4
  9
 16

julia> length(s)
4

julia> s[s .> 8]
2-element Vector{Int64}:
  9
 16

julia> s + s
4-element Vector{Int64}:
  2
  8
 18
 32
```

Metaprogramming

- As with Lisp, Julia represents code as a data structure in the language itself
- This means we can generate and transform code within the code itself!

```
julia> exp = Expr(:call, :+, Expr(:call, :*, 4, 2), Expr(:call, :-, 6, :x))
:(4 * 2 + (6 - x))

julia> x = 5
5

julia> eval(exp)
9
```

Metaprogramming

- As with Lisp, Julia represents code as a data structure in the language itself
- This means we can generate and transform code within the code itself!

```
julia> exp.args[2]
:(4 * 2)

julia> exp.args[2] = Expr(:call, :/, 10, 5)
:(10 / 5)

julia> exp
:(10 / 5 + (6 - x))

julia> eval(exp)
3.0

julia> x = 6
6

julia> eval(exp)
2.0
```

Metaprogramming

- As with Lisp, Julia represents code as a data structure in the language itself
- This means we can generate and transform code within the code itself!
- This looks a lot like genetic programming!
  - This makes sense, with GP's roots in Lisp
- Can also use metaprogramming to hold arbitrary information - many libraries use Symbols (eg. `:callable`) to represent settings in functions

Parallelism

Julia has inbuilt support for multiple type of parallelism

- LoopVectorization.jl allows for specific lines of code to be parallelised
- Julia base has support for classic multi-threading
  - Loops can be parallelised with `@threads`
- Built in GPU/CUDA support
- Utilise multiple machines with distributed computing

Unicode

- Full support in both strings and names for Unicode - including UTF and emoji
- This seems like a small feature, but it has a lot of benefits
- Code can directly relate to mathematical expressions it implements - no more spelling out Greek letters!
  `area(c::Circle) = π * c.r^2`

```
julia> struct Circle
           r::Float64
       end

julia> 🔴(c::Circle) = π * c.r^2
🔴 (generic function with 1 method)

julia> 🔴(Circle(2))
12.566370614359172
```

Interface with Other Languages

- Julia has functions and packages to easily call code from other languages!
- Call C functions with ccall
- Similar libraries exist for others - PyCall.jl, RCall.jl, and JavaCall.jl are all easy to use
- Helps with the infancy of Julia - just use complex packages from more mature languages!

Conversions and Promotions

- As with most other languages, Julia automatically converts data types when it can and needs to
  - Assigning to a typed field/variable/array
  - Returning from a typed function
  - Math: `1 + 1.5 -> 1.0 + 1.5 = 2.5`
- Unlike other languages, we can define our own conversions!
  `convert(::Type{MyType}, x) = MyType(x)`
- For math, types will be promoted to a common type. We can also define these rules:

```
promote_rule(::Type{Float64}, ::Type{Float32})
    = Float64
```

Performance Improvements

Performance

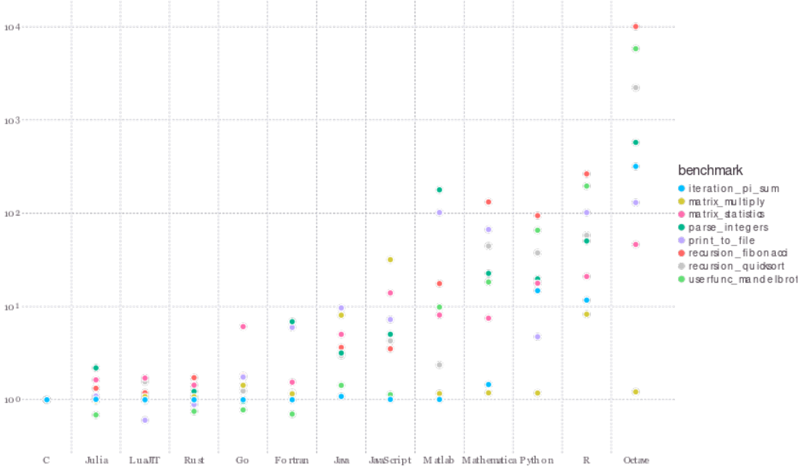
- As a compiled language, Julia can achieve much higher performance than languages it emulates
- No C backend - directly compiles itself
- Unlike Python, for loops perform just as well as vectorisations
- Sample benchmark - square a list → add 3 to all items → square root → sum:

```
Single for loop in Julia = 7.059 ms ± 4.517 ms
Single for loop in Python = 721.5896 ms

Multiple for loops in Julia = 8.468 ms ±  3.799 ms
Multiple for loops in Python = 957.65503 ms

Julia vectorisation = 5.051 ms ±  3.889 ms
Python (numpy) vectorisation = 4.814175158 ms
```

Benchmarks



10^4
10^3
10^2
10^1
10^0
C
Julia
LuaJIT
Rust
Go
Fortran
Java
JavaScript
Matlab
Mathematica
Python
R
Octave
benchmark
iteration_pi_sum
matrix_multiply
matrix_statistics
parse_integers
print_to_file
recursion_fibonacci
recursion_quicksort
userfunc_mandelbrot

Useful Libraries for Research

Plotting

- There are two "main" plotting libraries for Julia
- Plots.jl provides a simpler interface that is more familiar to those used to matplotlib

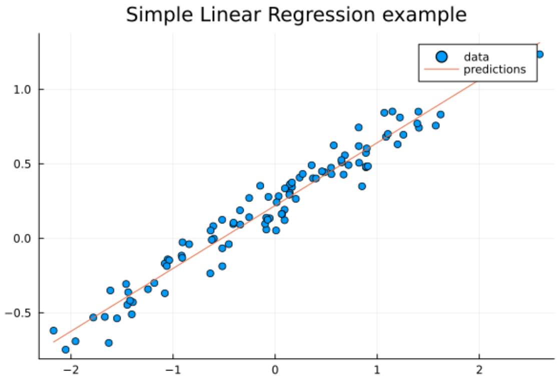
Plotting

- There are two “main” plotting libraries for Julia
- Plots.jl provides a simpler interface that is more familiar to those used to matplotlib

```
function sampleplotting(X, y, pred)
    scatter(X[1], y, title = "Simple Linear Regression
        example", label="data")
    plot!(X[1], pred, label="predictions")
end
```

Plotting

- There are two “main” plotting libraries for Julia
- Plots.jl provides a simpler interface that is more familiar to those used to matplotlib



Plotting

- There are two “main” plotting libraries for Julia
- Makie.jl provides a more complex interface that is more powerful than Plots

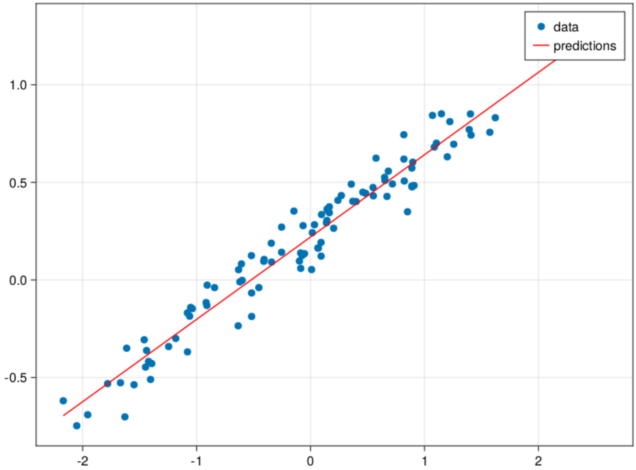
Plotting

- There are two “main” plotting libraries for Julia
- Makie.jl provides a more complex interface that is more powerful than Plots

```
function sampleplotting(X, y, pred)
    Makie.scatter(X[1], y, label="data")
    lines!(X[1], pred, label="predictions", color=:red)
    axislegend()
    current_figure()
end
```

Plotting

- There are two “main” plotting libraries for Julia
- Makie.jl provides a more complex interface that is more powerful than Plots



Machine Learning

- Similar to sklearn in Python, Julia has MLJ.jl
- Partially developed at University of Auckland
- Unified interface for many ML packages
- Slightly more complex to use than sklearn, but after a small learning curve works just as well
- Has support for sklearn models!

Machine Learning

- Similar to sklearn in Python, Julia has MLJ.jl
- Unified interface for many ML packages
- Slightly more complex to use than sklearn, but after a small learning curve works just as well
- Has support for sklearn models!
- Flux.jl also provides powerful deep learning functionality

Evolutionary Computation

- Of particular interest to this group will be Evolutionary.jl
- Implements algorithms for GA, DE, GP, and more
- Works about as well as DEAP - problems and all
- Initially strange workflow - quick to pick up!
- Few contributors, so not complete in places

Downsides

Compilation Times

- In order to achieve high performance, the compiler does a lot of work
- This is very slow
- Improved in recent versions of Julia - now attempts to compile packages when they are installed through the package manager
- Still very slow for some packages - the worst I've found is plotting packages
- In runtime needs to compile each dynamic dispatch method - JIT

Variable Performance

While Julia can have very good performance, this requires it to be used in a specific way:

- Code is only fast when it is inside a function
- Global variables slow down computation
- Containers slow down with abstract types
    - A `Vector{Real}` is much slower than a `Vector{Float64}`!
- Fields with abstract types are slow
- Essentially - the compiler can only do so much!

Very Young Language

- Bugs in core code
- Poor documentation
- Interfaces hard to find information on
- Parts of the language still very subject to change

Questions/Discussion