Julia for Matlab Users

Prof. Matthew Roughan
matthew.roughan@adelaide.edu.au
http://www.maths.adelaide.edu.au/matthew.roughan/

UoA

March 5th, 2021

THE UNIVERSITY of ADELAIDE

I write to find out what I think about something.

Neil Gaiman, The View From the Cheap Seats

Section 1

Get Started

- The reason I feel like we can do this is because (I hope) you all know some Matlab, and Julia is syntactically and operationally very much like Matlab
  - syntax is very similar
  - REPL[1] is similar
    - ★ tab completion, and up arrows work
    - ★ ? = help
    - ★ ; = shell escape to OS
  - JIT compiler
  - Use cases are similar

[1] REPL = Read-Evaluate-Print Loop; old-school name is the shell, or CLI.

So have a go

- You should have installed Julia before the workshop
- Start it up
  - start up varies depending on IDE, and OS
  - I am using simplest case (for me): the CLI, on a Mac
  - it's all very Unix-y
- Type some calculations

```
a = 3
b = a + 2
c = a + b^2
```

- Create a script, *e.g.*, "test.jl", and "include" it

```
include("test.jl")
```

  - its a little more cumbersome than Matlab

Section 2

Julia Isn't Matlab (or Octave)

Julia may look a lot like Matlab but

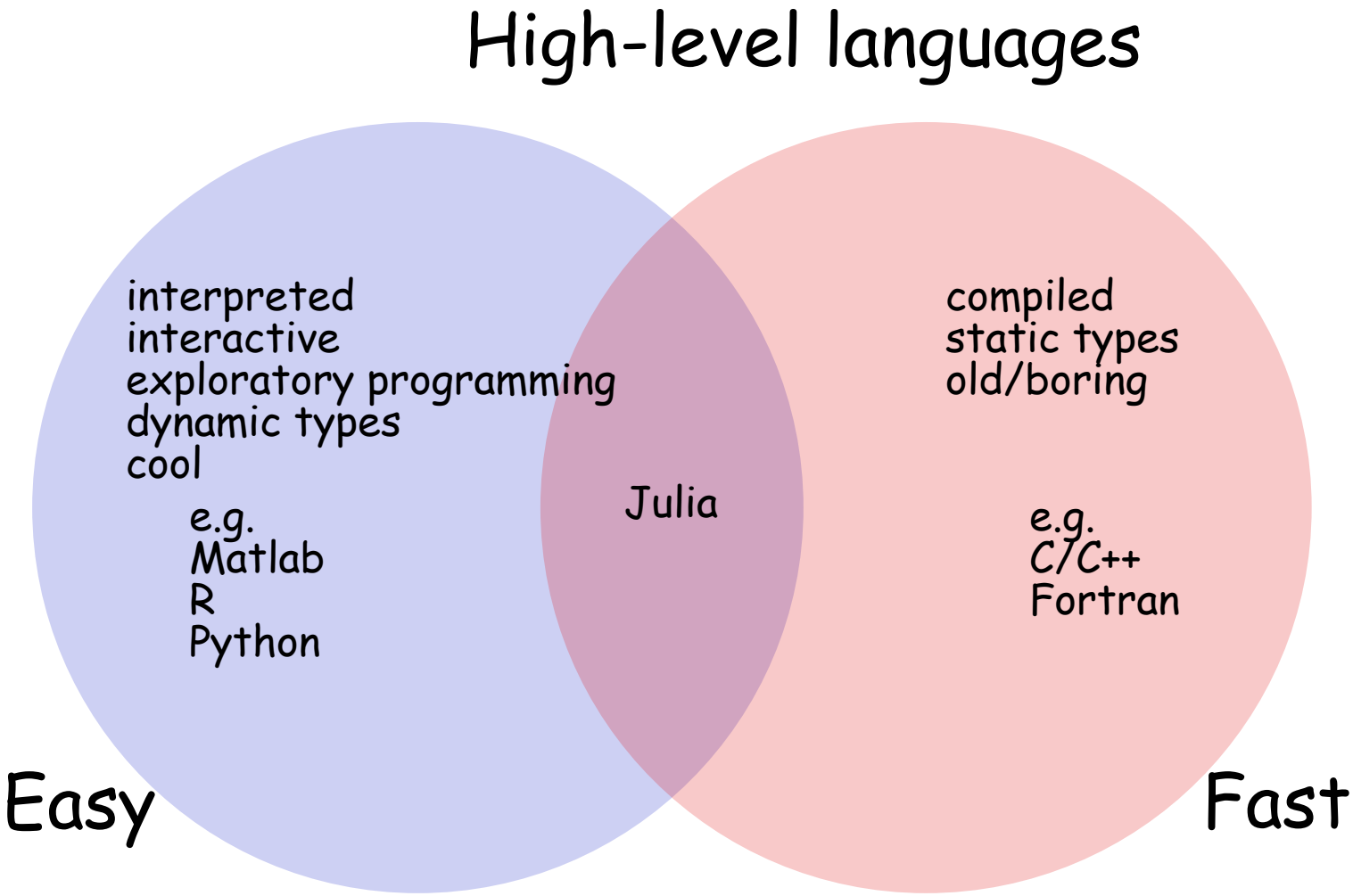
- under the hood its very different
- and there are a lot of changes that affect you

otherwise why would we bother?

Why Julia? Big Differences

- Faster (natively)
  - depends on what you are doing though
- Better name spaces
  - better for modules
- Better Support for Types and Data Structures
  - Strongly typed, but dynamic
  - Lots of useful types
    - ★ *e.g.,* Dictionaries (associative arrays)
- Homoiconic: Julia parses its code into Julia data structures (which we can potentially manipulate)
- Concurrency

Native Speed is Key



Less Obvious, But Important Differences

- Lots, lets deal with 1 by 1
- I will focus on the points that gave me the most pain or pleasure

1D and 2D Arrays

- Similar to Matlab
  - row based definition (as in Matlab)
  - similar constructors: `zeros, ones, ...`
- Array definition is slightly different
  - no commas in row definition
  - commas or semicolons separate rows, but with slightly different meaning
  - can have any type of element
- Julia has true one-dimensional arrays, *i.e.,* vectors
  - a single column of a 2D array is not the same as a vector
  - for me there are some slight weirdnesses in this
  - Can lead to confusing bugs to start with, but can also allow for more efficient code.
    - ★ how many Matlab functions begin by checking row or col vector input, or changing it around?

1D and 2D Arrays

Try It!

```
A = [1  2    3]
B = [1, 2.0, 3]
C = [1, 2,   3 // 4]
D1 = [ [1 2 3], [4 5 6] ]
D2 = [ 1 2 3; 4 5 6]
D3 = [ 1 2 3
       4 5 6 ]
E = Array{Int64,2}(undef, 2,3)
F = ["string1"  "string2"]
G = zeros(2,3)
H = ones(Int64, 3)
?ones
```

Array Indexing

- Can still use Matlab forms `:` and `end`
- But use square brackets for array indexing
- **Try It!**

```
A[2]
D3[2,3]
D3[2, :]
D3[2, end]
```

- Square brackets are better
  - separates functions from arrays
  - consistent with array definition
  - avoids name clashes, and hence bugs
- But I keep typing it wrong :(

Like Matlab (and unlike Python and C), Julia starts indexing from 1, not 0

Julia arrays are assigned by reference

- If you type `A = B`, you are not creating a copy of `B`, you are creating a reference, so
- **Try It!**

```
X = [1 2 3]
Y = X
Y[1] = 3
X
Z = copy(X) # create an actual copy, not a ref
Z[1] = 4
X
```

- Same is true of function array arguments: they are passed by reference
  - a function can alter its inputs
- This is efficient, but can lead to some obscure bugs
  - Matlab has a fancy hybrid system, that is actually pretty nice IMHO

Range Objects and Iterators

- In Julia `a:b` constructs a **Range** object, not a vector
- You can iterate over a Range
  - more efficient because it lazily calculates values
    - ★ doesn't use as much memory
    - ★ saves effort if you break out of the loop
- If you want the vector use `collect`, but often you don't need to

Try It!

```
x = 3:2:11
for i = x
    println(i)
end
x[3:end-1]
x + 10
collect(x)
```

Semicolons, Ellipsis, and Comments

- Matlab
  - ; at the end of a line suppresses output
  - ... extends a line
  - Matlab comments preceded by `%`
    Julia comments preceded by `#`
- Julia
  - ; at end of line doesn't do anything except when typing interactively in REPL
    - ★ *e.g.,* don't need semi-colons in function defs
  - incomplete lines are automatically continued
- **Try It!**[2]

```
x = 3 +
        2
```

[2] I notice that the Atom-based IDE doesn't do line continuation in its console.

.* notation for everything

- The Matlab idea of `.*` is extended to most other operators
  Try It!

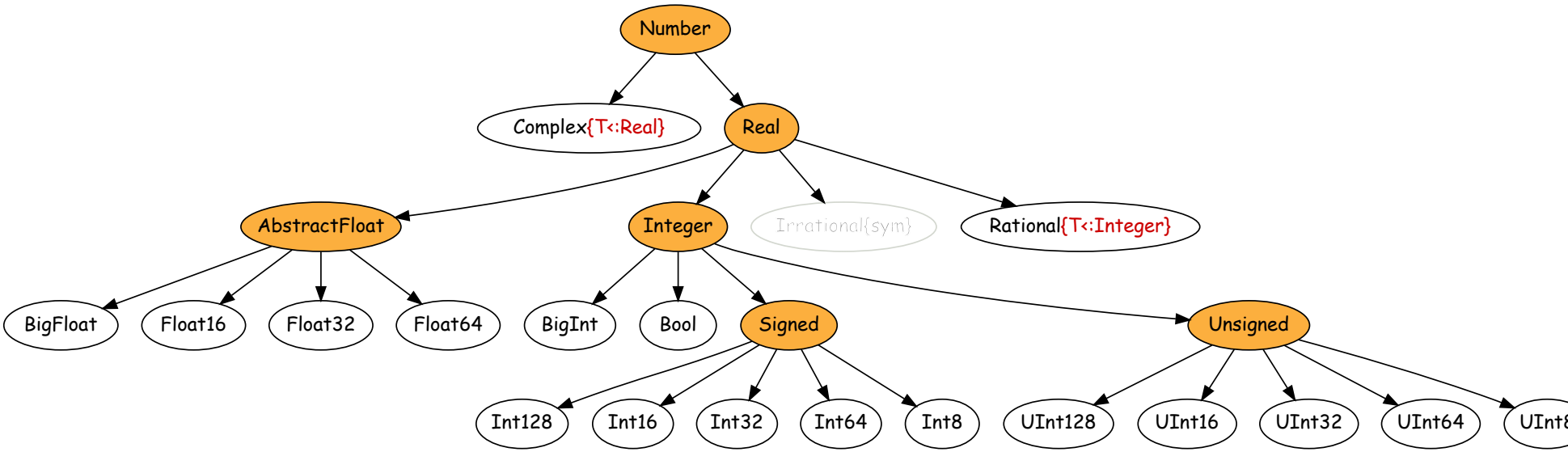
```
[2,4] .* [10, 20]
[1,2,3] .- [1,2,3]
[3,4] .== [3,5]
[3,4] .< [3,5]
```

- And BTW, we can use C-like syntax to

```
x = 1
x *= 2
x -= 7
```

  but not `i++`

Stronger support for data types with multiple dispatch



Try It!

```
a = 3
b = 2.3
c = 3 // 6
typeof(a), typeof(b), typeof(c)
sqrt(-1)
sqrt(complex(-1))
```

Tighter scoping rules

- Variables have scope of the block they are defined in
  Try It!

```
n = 3
for i=1:n
    x = 2i
end
i
x
```

- You need to pre-define the variable outside the loop to use it outside the loop
  - *e.g.,* set `x=0` before the loop
  - still have no direct access to counter

Separate Char and String types (yay!)

- Single-quotes to define a `Char`
- Double-quotes to define a `String`
- Concatenation operator is `*`
  Try It!

```
a = 'a'
b = 'x'
ab = "ab"
abc = ab * "c"
abc = ab * b
abc = ab * string(b)
```

- Julia has better string handling in lots of other ways
  - regular expressions

Julia Doesn't Automatically Grow Arrays

- This is somewhat annoying but
  - avoids inefficient code
  - avoids some bugs
- An alternative approach is to use a **comprehension**

Matlab

```
for i=1:10
    x(i) = i^2
end
```

Julia

```
x = [i^2 for i in 1:10]
```

In Julia this will be (probably) faster than `x = collect(1:10).^2`

List Comprehensions

- **List comprehensions** represent in a more mathematical syntax
  - *e.g.,*

    $$\{i^2 \mid i = 1, 2, \ldots, 10\}$$

    becomes

    ```
    [i^2 for i in 1:10]
    ```
- Syntactic sugar for defining one array in terms of another array or iterator
  - Python-like syntax
  - Can replace “in” with $\in$, or =

Try It!

```
[ x for x ∈ 1:2]
[ x*y for x=1:2, y=3:4]
```

Dictionaries (associative arrays)

- Dictionaries associate (key, value) pairs
- Looks like an array indexed by arbitrary objects
  Try It!

```
x = Dict()
x[1] = "five"
x["three"] = 3
x["three"]
```

  Note I **can** grow this as I go
- They are called variously
  - dictionaries in Smalltalk, Swift, Python, ...
  - hashes in Perl, Ruby, ...
  - maps in Java, Go, Scala, Haskell, **Matlab** in latest versions via Java
- Julia also has **Sets**

More on Dictionaries

- Constructing dictionaries

  Try It!

```
dict = Dict("a" => 1, "b" => 2, "c" => 3)
dict = Dict{String,Integer}("a" => 1, "b" => 2)
dict = Dict(string(i) =>sin(pi*i/180) for i=0:360)
dict["90"]
```

- Useful functions

  Try It!

```
dict = Dict("a" => 1, "b" => 2, "c" => 3);
keys(dict)      # which is an iterator
values(dict)    # which is also an iterator
for key in keys(dict)
    println("$key => $(dict[key])")
end
```

- Note that entries are **not** ordered
  - use `sort(collect(keys(dict)))`
  - use `SortedDict` from `DataStructures` package

Unicode Support

Julia has Unicode support, so the following should be a valid Lotka-Volerra simulation

```
😺 = 10    # number of cats
😻 = 100   # number of mice
for i=1:n
    😺 = 😺 + α*😺 + β*😺*😻
    😻 = 😻 + δ*😻 - γ*😺*😻
end
```

From https://twitter.com/eloceanografo/status/790939841223589888

Try It!

```
CTRL-SHIFT-u 03b1
\alpha TAB = 1
\pi TAB
c = '\u03b1'
```

Unicode Support

Alpha	\u0391	Beta	\u0392	Gamma	\u0393	Delta	\u0394
Epsilon	\u0395	Zeta	\u0396	Eta	\u0397	Theta	\u0398
Iota	\u0399	Kappa	\u039a	Lambda	\u039b	Mu	\u039c
Nu	\u039d	Xi	\u039e	Omicron	\u039f	Pi	\u03a0
Rho	\u03a1	Sigma	\u03a3	Tau	\u03a4	Upsilon	\u03a5
Phi	\u03a6	Chi	\u03a7	Psi	\u03a8	Omega	\u03a9
alpha	\u03b1	beta	\u03b2	gamma	\u03b3	delta	\u03b4
epsilon	\u03b5	zeta	\u03b6	eta	\u03b7	theta	\u03b8
iota	\u03b9	kappa	\u03ba	lambda	\u03bb	mu	\u03bc
nu	\u03bd	xi	\u03be	omicron	\u03bf	pi	\u03c0
rho	\u03c1	altsigma	\u03c2	sigma	\u03c3	tau	\u03c4
upsilon	\u03c5	phi	\u03c6	chi	\u03c7	psi	\u03c8
omega	\u03c9	complex	\u2102	naturals	\u2115	rationals	\u211a
reals	\u211d	integers	\u2124	forall	\u2200	exists	\u2203
triangle	\u2206	uptri	\u2207	isin	\u220a	pm	\u2213
sqrt	\u221a	int	\u222b	leq	\u2264	geq	\u2265
subset	\u2283	intersection	\u22c2	union	\u22c3		

For more see

https://docs.julialang.org/en/latest/manual/unicode-input/

Namespaces

Julia has them. Matlab doesn't.

There are lots of other differences, for instance, in specific behaviours of functions, or the nomenclature of notionally equivalent functions in Matlab and Julia. We won't try to list them all.

Comparisons

- https://docs.julialang.org/en/v1/manual/noteworthy-differences/
- https://juliafs.readthedocs.io/en/latest/manual/noteworthy-differences.html
- https://cheatsheets.quantecon.org/
- https://tobydriscoll.net/blog/matlab-vs.-julia-vs.-python/
- https://www.juliabloggers.com/why-numba-and-cython-are-not-substitutes-for-julia

Some general references

- https://learnxinyminutes.com/docs/julia/
- https://cheatsheets.quantecon.org/
- https://docs.julialang.org/en/stable/

Please note that some online help was written regarding earlier versions of Julia and there have been changes. Refer only to help/discussions from versions 1.0 forward, i.e., from around 2019 onwards.

Section 3

Activity

Activity

Create a function to translate an arbitrary positive integer into Roman numerals.

- https://projecteuler.net/problem=89
- http://www.rapidtables.com/convert/number/roman-numerals-converter.htm
- https://en.wikipedia.org/wiki/Roman_numerals

Use standard (modern) form Roman numerals
Skeleton

```
function int2roman(n::Int)
    # output a Roman numeral string

end
```

Save your function into a `.jl` file, and "include" it.

Bonus frames

Function or methods

There are a lot of differences in the way functions are created and used in Julia (as compared to Matlab). It's probably best to just forget the Matlab approach and start afresh.

Logical operators

In Julia you need to be aware of types when using logical operators. Integers are not the same as Booleans.

- `&` and `|` are *bitwise* AND and OR and there is also `⊻` for XOR and shift operators also exist.
- Logical (Boolean) AND and OR use `&&` and `||`
- bitwise NOT is denoted `~` but `!` also works for bools
- Can use `!=`, or $\neq$ for testing not equals
- operations on vectors/arrays need to use the dot syntax, *e.g.,* `.&` and `.|` and `.==`
- Julia's operator precedence rules are slightly different https://docs.julialang.org/en/v1/manual/mathematical-operations/#Operator-Precedence-and-Associativity

See also https://www.geeksforgeeks.org/operators-in-julia/ and https://github.com/JuliaLang/julia/issues/5187

Julia has “tuples”

- Almost like an array
  - ordered sequence of values
  - denoted by round braces
  - but can index them as with arrays
- But they are **immutable**
  - once created you can’t change them
  - can be very efficient
- **Try It!**

```
t = (1,2,3,4)
t[3:end]
t[1] = 2
```

- Used all over the place, *e.g.*,
  - function argument lists
  - returning multiple arguments from functions

tic()/toc() performance