Julia Part I
Julia for Matlab Users

Prof. Matthew Roughan
matthew.roughan@adelaide.edu.au

UoA

Oct 31, 2017
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 it's 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

High-level languages

Easy
interpreted
interactive
exploratory programming
dynamic types
cool
e.g. Matlab
R
Python

Fast
compiled
static types
old/boring
e.g. C/C++
Fortran

Julia
Faster: Their Benchmarks

- y-axis is powers of 10
- Relative to C performance
- Smaller is better
Faster: My Benchmarks

Simple function that calculates whether 3 points in $\mathbb{R}^2$ are in clockwise or counter-clockwise order.
Less Obvious, But Important Differences

• Lots, let's 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 = \text{Array\{Int64\}\(2,3\)} \]
\[ F = ["\text{string1}\" \ "\text{string2}\"] \]
\[ G = \text{zeros}(2,3) \]
\[ H = \text{ones}(\text{Int64}, \ 3) \]
\[ ?\text{ones} \]
Array Indexing

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

```plaintext
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, 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
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!**

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

- Used all over the place, e.g.,
  - function argument lists
  - returning multiple arguments from functions
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!\(^2\)

\[
x = 3 +
\]
\[
\begin{array}{c}
2
\end{array}
\]

\(^2\)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!**

```
[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!

```julia
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!**

```julia
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 `i=0` before the loop.
Separate Char and String types (yay!)

- Single-quotes to define a Char
- Double-quotes to define a String
- Concatenation operator is *

**Try It!**

```plaintext
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

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

Julia

```julia
x = [i*i for i in 1:10]
```

In Julia this will be (probably) faster than

```julia
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
  
  \[ [i*i \text{ for } i \text{ in } 1:10] \]

- Syntactic sugar for defining one array in terms of another array or iterator
  - Python-like syntax
  - Can replace “in” with ∈, or =

**Try It!**

\[
[ x \text{ for } x \in 1:2] \\
[ x*y \text{ for } x=1:2, \ y=3:4]
\]
Dictionaries (associative arrays)

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

**Try It!**

```python
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-Volterra 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

<table>
<thead>
<tr>
<th>Alpha</th>
<th>\u0391</th>
<th>Beta</th>
<th>\u0392</th>
<th>Gamma</th>
<th>\u0393</th>
<th>Delta</th>
<th>\u0394</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epsilon</td>
<td>\u0395</td>
<td>Zeta</td>
<td>\u0396</td>
<td>Eta</td>
<td>\u0397</td>
<td>Theta</td>
<td>\u0398</td>
</tr>
<tr>
<td>Iota</td>
<td>\u0399</td>
<td>Kappa</td>
<td>\u039a</td>
<td>Lambda</td>
<td>\u039b</td>
<td>Mu</td>
<td>\u039c</td>
</tr>
<tr>
<td>Nu</td>
<td>\u039d</td>
<td>Xi</td>
<td>\u039e</td>
<td>Omicron</td>
<td>\u039f</td>
<td>Pi</td>
<td>\u03a0</td>
</tr>
<tr>
<td>Rho</td>
<td>\u03a1</td>
<td>Sigma</td>
<td>\u03a3</td>
<td>Tau</td>
<td>\u03a4</td>
<td>Upsilon</td>
<td>\u03a5</td>
</tr>
<tr>
<td>Phi</td>
<td>\u03a6</td>
<td>Chi</td>
<td>\u03a7</td>
<td>Psi</td>
<td>\u03a8</td>
<td>Omega</td>
<td>\u03a9</td>
</tr>
<tr>
<td>Alpha</td>
<td>\u03b1</td>
<td>Beta</td>
<td>\u03b2</td>
<td>Gamma</td>
<td>\u03b3</td>
<td>Delta</td>
<td>\u03b4</td>
</tr>
<tr>
<td>Epsilon</td>
<td>\u03b5</td>
<td>Zeta</td>
<td>\u03b6</td>
<td>Eta</td>
<td>\u03b7</td>
<td>Theta</td>
<td>\u03b8</td>
</tr>
<tr>
<td>Iota</td>
<td>\u03b9</td>
<td>Kappa</td>
<td>\u03ba</td>
<td>Lambda</td>
<td>\u03bb</td>
<td>Mu</td>
<td>\u03bc</td>
</tr>
<tr>
<td>Nu</td>
<td>\u03bd</td>
<td>Xi</td>
<td>\u03be</td>
<td>Omicron</td>
<td>\u03bf</td>
<td>Pi</td>
<td>\u03c0</td>
</tr>
<tr>
<td>Rho</td>
<td>\u03c1</td>
<td>Altsigma</td>
<td>\u2102</td>
<td>Sigma</td>
<td>\u03c3</td>
<td>Tau</td>
<td>\u03c4</td>
</tr>
<tr>
<td>Upsilon</td>
<td>\u03c5</td>
<td>Phi</td>
<td>\u03c6</td>
<td>Chi</td>
<td>\u03c7</td>
<td>Psi</td>
<td>\u03c8</td>
</tr>
<tr>
<td>Omega</td>
<td>\u03c9</td>
<td>Complex</td>
<td>\u2102</td>
<td>Naturals</td>
<td>\u2115</td>
<td>Rationals</td>
<td>\u211a</td>
</tr>
<tr>
<td>Reals</td>
<td>\u211d</td>
<td>Integers</td>
<td>\u211d</td>
<td>Forall</td>
<td>\u2200</td>
<td>Exists</td>
<td>\u2203</td>
</tr>
<tr>
<td>Triangle</td>
<td>\u2206</td>
<td>Uptri</td>
<td>\u2207</td>
<td>Isin</td>
<td>\u220a</td>
<td>Pm</td>
<td>\u2213</td>
</tr>
<tr>
<td>Sqrt</td>
<td>\u221a</td>
<td>Int</td>
<td>\u222b</td>
<td>Leq</td>
<td>\u2264</td>
<td>Geq</td>
<td>\u2265</td>
</tr>
<tr>
<td>Subset</td>
<td>\u2283</td>
<td>Intersection</td>
<td>\u22c2</td>
<td>Union</td>
<td>\u22c3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For more see

There are lots more differences between Matlab and Julia ... but I hope they won’t bite you this week.
Some useful references

- https://learnxinyminutes.com/docs/julia/
- https://docs.julialang.org/en/release-0.6/manual/noteworthy-differences/
- https://cheatsheets.quantecon.org/
Section 3

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

- [https://projecteuler.net/problem=89](https://projecteuler.net/problem=89)

Use standard (modern) form Roman numerals

**Skeleton**

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

Save your function into a `.jl` file, and “include” it.
Bonus frames
tic()/toc() performance

M.Roughan (UoA)