Crash Course on Julia

Undergraduate Computational Macro

Jesse Perla

jesse.perla@ubc.ca

University of British Columbia

Table of contents

Introduction to Julia

Introductory Lectures

- See intro lecture for environment setup instructions
- Assuming you are familiar with Matlab or Python, Julia will be easy to learn
- Adapted from QuantEcon lectures coauthored with John Stachurski and Thomas J. Sargent
  - → Julia by Example
  - → Essentials
  - → Fundamental Types
- SciML Cheat Sheet for Matlab/Python/Julia

Using Packages

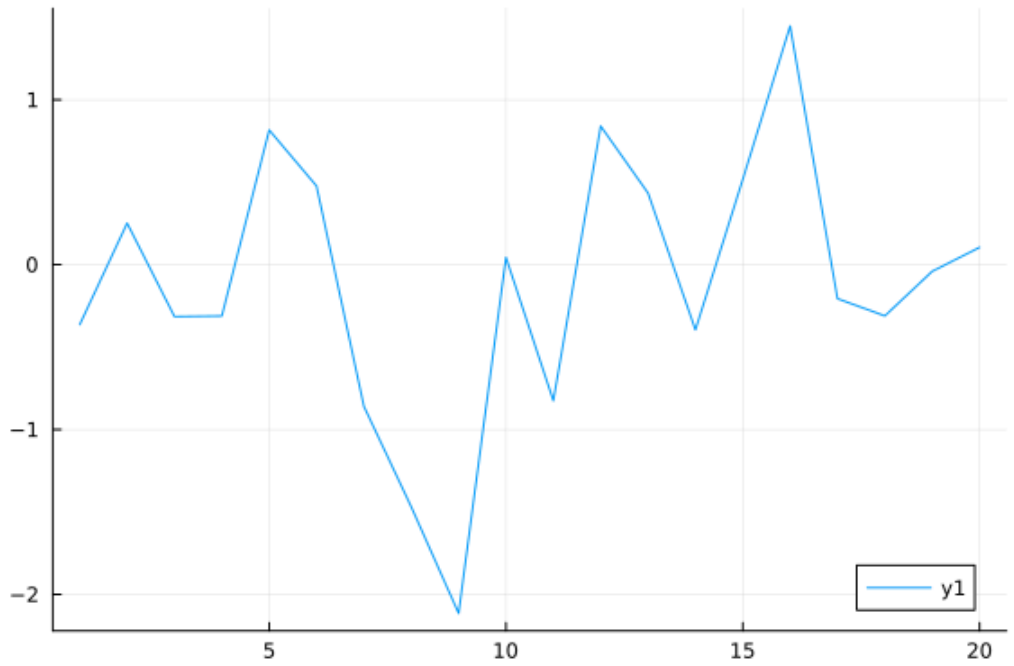
- First ensure your project is activated and packages instantiated

```
1 using LinearAlgebra, Statistics, Plots, Distributions, Random
```

Basic Operations and Control Flow

Plotting Random Numbers

```
1  Random.seed!(42) # for reproducibility
2  n = 20
3  ep = randn(n)
4  plot(1:n, ep;size=(600,400))
```



Loops

```
n = 100
ep = zeros(n)
for i in 1:n
    ep[i] = randn()
end
println(ep[1:5])
```

```
[-0.9654904870197227, 0.9656607495563969, 1.3110173557334994, 0.4041754007591603, 1.1979596698125403]
```

While Loops and `break`

```
1  i = 1
2  total = 0.0
3  while i <= 10
4      total += rand()
5      if total > 2.0
6          break # leave the loop early when condition met
7      end
8      i += 1
9  end
10 @show i, total;
```

```
(i, total) = (4, 2.1678515649076444)
```

Comprehensions

```
1  # Comprehensions
2  @show [2 * i for i in 1:4];
```

```
[2i for i = 1:4] = [2, 4, 6, 8]
```

Manually Calculated Mean

```
1 ep_sum = 0.0 # careful to use 0.0 here, instead of 0
2 for ep_val in ep
3     ep_sum = ep_sum + ep_val
4 end
5 @show ep_mean = ep_sum / length(ep)
6 @show ep_mean ≈ mean(ep)
7 @show ep_mean
8 @show sum(ep) / length(ep)
9 @show sum(ep_val for ep_val in ep) / length(ep); # generator/comprehension
```

```
ep_mean = ep_sum / length(ep) = -0.10436111001880369
ep_mean ≈ mean(ep) = true
ep_mean = -0.10436111001880369
sum(ep) / length(ep) = -0.10436111001880365
sum((ep_val for ep_val = ep)) / length(ep) = -0.10436111001880369
```

String Interpolation

- Use `$name` for variables or `$(expr)` for expressions inside strings

```
1 name = "Julia"
2 x = 3
3 println("Hello $name, 2x = $(2x)")
```

```
Hello Julia, 2x = 6
```

Macros and `@show`

- Macros (prefixed with `@`) transform code before execution; `@show` is a handy debug macro

```
1 a = 1 + 2
2 @show a
3 @show sum(randn(3));
```

```
a = 3
sum(randn(3)) = 0.9905223195566424
```

Functions and Functional Programming

Functions

```
function generatedata(n)
    ep = randn(n) # use built in function
    for i in eachindex(ep) # or i in 1:length(ep)
        ep[i] = ep[i]^2 # squaring the result
    end
    return ep
end
data = generatedata(5)
println(data)
```

```
[0.456293906611111663, 2.239276519973084, 0.012093086182622361, 0.8178133947622638, 0.029520710239712283]
```

Broadcasting

```
function generatedata(n)
    ep = randn(n) # use built in function
    return ep .^ 2
end
@show generatedata(5)
generatedata2(n) = randn(n) .^ 2
@show generatedata2(5);
```

```
generatedata(5) = [0.7936585609350724, 2.54008455063787, 0.03630079956534469, 1.0596951950031361, 0.026581091545836936]
generatedata2(5) = [0.12012915959661571, 0.14908580134436417, 1.2037454134039276, 0.014073031953049788,
3.7808480859126736]
```

Higher Order Functions

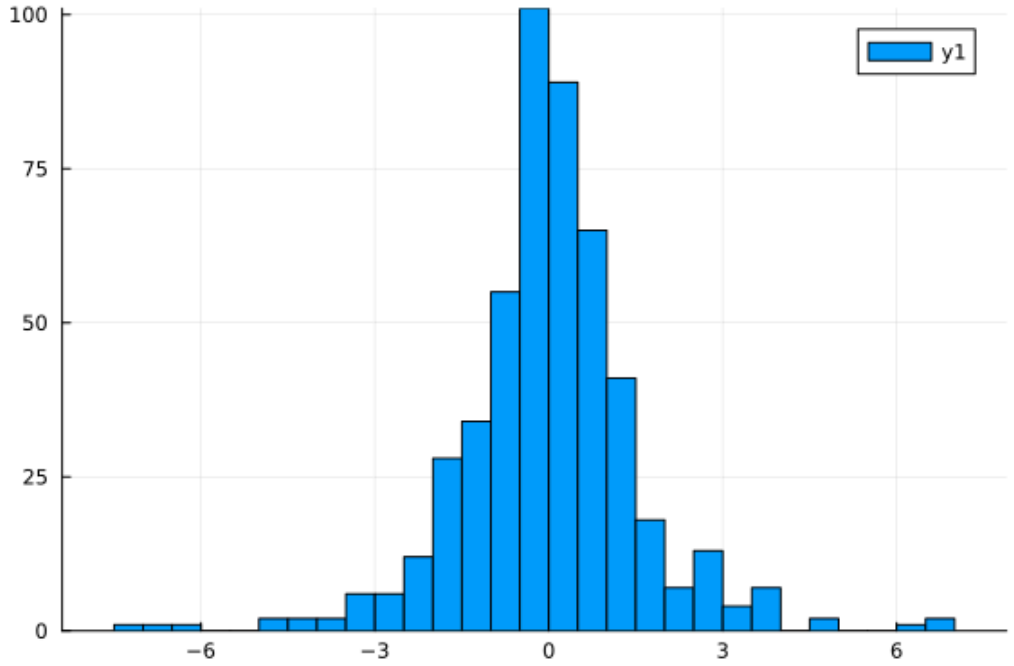
```
1 generatedata3(n, gen) = gen.(randn(n)) # broadcasts on gen
2 f(x) = x^2 # simple square function
3 @show generatedata3(5, f); # applies f
```

```
generatedata3(5, f) = [0.7121322893565936, 0.0185955776623056, 0.0912022660336257, 0.11442078211374904,
0.25477412900679614]
```

Plotting and Visualization

More Plotting Examples

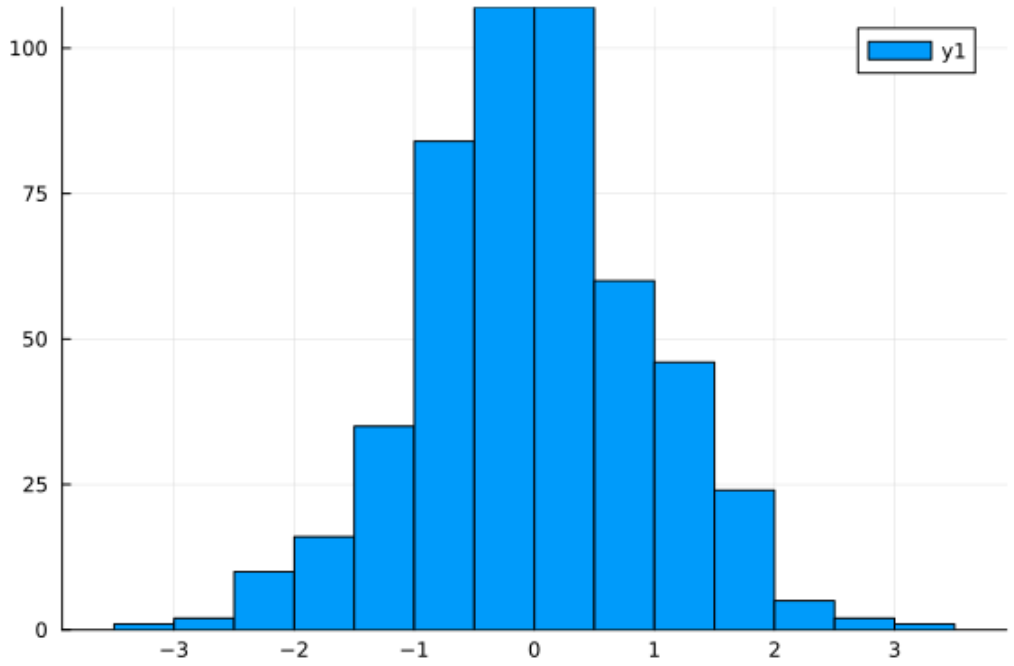
```
1  using Distributions
2  function plothistogram(dist, n)
3      # n draws from distribution
4      ep = rand(dist, n)
5      return histogram(ep;size=(600,400))
6  end
7  dist = Laplace() # dist != dist in function
8  plothistogram(dist, 500)
```



Changing Types

- The `rand(dist, n)` changes its behavior based on the type of `dist`

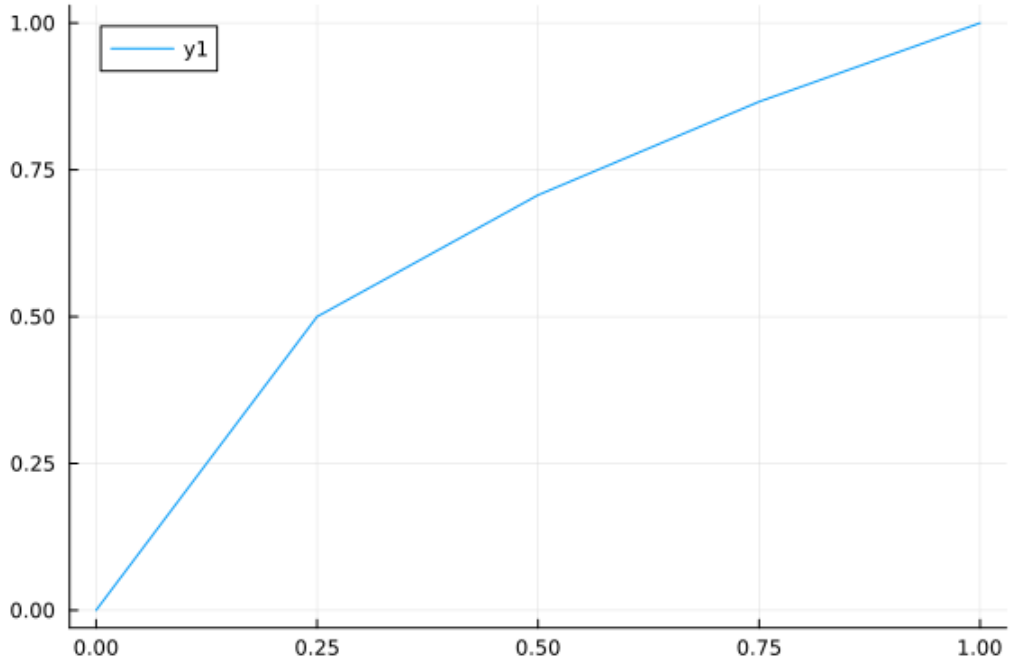
```
1 dist = Normal()
2 plothistogram(Normal(), 500)
```



Ranges

```
1  x = range(0.0, 1.0; length = 5)
2  @show x
3  @show Vector(x)
4  plot(x, sqrt.(x); size=(600,400))
```

```
x = 0.0:0.25:1.0
Vector(x) = [0.0, 0.25, 0.5, 0.75, 1.0]
```



Defining Functions

- You can create anonymous functions as in R, but it is harder for the compiler because the type `f3` can change. Avoid `->` if name required

```
1 f(x) = x^2
2 function f2(x)
3     return x^2
4 end
5 f3 = x -> x^2 # assignment not required
6 @show f(2), f2(2), f3(2);
```

```
(f(2), f2(2), f3(2)) = (4, 4, 4)
```

Default Function Arguments

```
1 f(x, a = 1) = exp(cos(a * x))
2 @show f(pi)
3 @show f(pi, 2);
```

```
f(pi) = 0.36787944117144233
f(pi, 2) = 2.718281828459045
```

Keyword Arguments

```
1  f2(x; a = 1) = exp(cos(a * x))  # note the ; in the definition
2  # same as longform
3  function f(x; a = 1)
4      return exp(cos(a * x))
5  end
6  @show f(pi)
7  @show f(pi; a = 2) # passing in adate
8  a = 2
9  @show f(pi; a); # equivalent to f(pi; a = a)
```

```
f(pi) = 0.36787944117144233
f(pi; a = 2) = 2.718281828459045
f(pi; a) = 2.718281828459045
```

Closures

- In general, try to avoid globals and closures outside of functions

```
1 a = 0.2
2 f(x) = a * x^2  # refers to the `a` in the outer scope
3 @show f(1)
4 # The a is captured in this scope by name.  Careful!
5 a = 0.3
6 @show f(1);
```

```
f(1) = 0.2
f(1) = 0.3
```

Closures Inside Functions

- But within a function they are safe, common, and usually free of overhead

```
function g(a)
    f(x) = a * x^2  # refers to the `a` passed in the function
    return f(1)
end
a = 123.5 # Different scope than the `a` in function
@show g(0.2);
```

```
g(0.2) = 0.2
```

Data Structures

Tuples and Named Tuples

```
1 t = (1, 2.0, "hello")
2 @show t[1]
3 nt = (; a = 1, b = 2.0, c = "hello")
4 @show nt
5 @show nt.a; # can't use nt[1] or nt["a"]
```

```
t[1] = 1
nt = (a = 1, b = 2.0, c = "hello")
nt.a = 1
```

Tuples Packing and Unpacking

```
1  function solve_model(x)
2      a = x^2
3      b = 2 * a
4      c = a + b
5      return (; a, b, c)  # note local scope of tuples!
6  end
7  @show solve_model(0.1)
8  # can unpack in different order, or use subset of values
9  (; c, a) = solve_model(0.1)
10 println("a = $a, c = $c");
```

```
solve_model(0.1) = (a = 0.010000000000000002, b = 0.020000000000000004, c = 0.030000000000000006)
a = 0.010000000000000002, c = 0.030000000000000006
```

Arrays and Linear Algebra

Array Basics

```julia
b = [1.0, 2.1, 3.0] # 1d array
A = [1 2; 3 4] # 2x2 matrix
@show size(b)
@show size(A)
@show typeof(b)
@show typeof(A)
@show zeros(3)
@show ones(2, 2)
@show fill(1.0, 2, 2)
@show similar(A)
@show A[1, 1]
@show A[1, :]
@show A[1:end, 1];
```

```
size(b) = (3,)
size(A) = (2, 2)
typeof(b) = Vector{Float64}
typeof(A) = Matrix{Int64}
zeros(3) = [0.0, 0.0, 0.0]
ones(2, 2) = [1.0 1.0; 1.0 1.0]
fill(1.0, 2, 2) = [1.0 1.0; 1.0 1.0]
similar(A) = [140033842892576 140033842892608;
140033842892592 140034887560608]
A[1, 1] = 1
A[1, :] = [1, 2]
A[1:end, 1] = [1, 3]
```

Useful Array Helpers

```
x = [1.0, -1.0, 2.5, -0.5]
@show cumsum(x)
g(x) = x > 1.0
@show findfirst(g, x) # 1-based index of first match
# equivalent anonymous function versions:
val_1 = findfirst(x_val -> x_val > 1.0, x)
val_2 = findfirst(x -> x > 1.0, x) # note scope!
@show val_1, val_2;
```

```
cumsum(x) = [1.0, 0.0, 2.5, 2.0]
findfirst(g, x) = 3
(val_1, val_2) = (3, 3)
```

Linear Algebra Basics

```
1 A = [1 2; 3 4]
2 b = [1, 2]
3 @show A * b # Matrix product
4 @show A' # transpose
5 @show dot(b, [5.0, 2.0]) # dot product
6 @show b' * b # dot product
7 @show Diagonal([1.0, 2.0]) # diagonal matrix
8 @show I # identity matrix
9 @show inv(A); # inverse
```

```
A * b = [5, 11]
A' = [1 3; 2 4]
dot(b, [5.0, 2.0]) = 9.0
b' * b = 5
Diagonal([1.0, 2.0]) = Diagonal([1.0, 2.0])
I = LinearAlgebra.UniformScaling{Bool}(true)
inv(A) = [-1.9999999999999996 0.9999999999999998;
1.4999999999999998 -0.4999999999999999]
```

Norms

- Defaults to the Euclidean (2) norm; use `norm(v, 1)` or `norm(v, Inf)` for others

```
1 v = [3.0, -4.0, 1.0]
2 manual_euclid1 = sqrt(sum(v.^2))
3 manual_euclid2 = sqrt(sum(abs2, v)) # sum can apply function first
4 @show norm(v)
5 @show manual_euclid1, manual_euclid2
6 @show norm(v, 1)
7 @show norm(v, Inf);
```

```
norm(v) = 5.0990195135927845
(manual_euclid1, manual_euclid2) = (5.0990195135927845, 5.0990195135927845)
norm(v, 1) = 8.0
norm(v, Inf) = 4.0
```

Linear Solves

```
1 A = [2.0 1.0; 1.0 3.0]
2 b = [1.0, 0.0]
3 x = A \ b # preferred over inv(A) * b
4 @show x
5 @show norm(A * x - b); # residual norm
```

```
x = [0.6, -0.2]
norm(A * x - b) = 5.551115123125783e-17
```

Eigenvalues and Eigenvectors

```
1 A = [2.0 1.0; 1.0 2.0]
2 vals = eigvals(A)
3 F = eigen(A)
4 @show vals
5 @show F.vectors[:, 1];
```

```
vals = [1.0, 3.0]
F.vectors[:, 1] = [-0.7071067811865475, 0.7071067811865475]
```

Modifying Vectors

- Scalars and tuples/named tuples are immutable
- Vectors and matrices are mutable

```
1  A = [1 2; 3 4]
2  A[1, 1] = 2
3  @show A
4  b = [1, 2]
5  b[1] = 2
6  @show b
7  b .= [3, 4] # otherwise just renamed
8  @show b
9  A[1, :] .= [3, 4] # assign slice
10 @show A;
```

```
A = [2 2; 3 4]
b = [2, 2]
b = [3, 4]
A = [3 4; 3 4]
```

Slicing Higher-D Arrays

- Julia is column-major; slicing by the last index stays contiguous and fast
- Loop order: vary first index fastest for best cache performance

```julia
X = reshape(collect(1:24), 2, 3, 4) # 2×3×4 tensor
@show size(X)
M2 = X[:, :, 2] # 2×3 matrix view, contiguous in memory
@show M2
@show view(X, :, :, 2) # does not allocate, nice if contiguous
@show X[1, :, 2]; # row slice from that matrix
```

```
size(X) = (2, 3, 4)
M2 = [7 9 11; 8 10 12]
view(X, :, :, 2) = [7 9 11; 8 10 12]
X[1, :, 2] = [7, 9, 11]
```

Learning More

- Clone the QuantEcon lectures
  - → `> Git: Clone` the `https://github.com/quantecon/lecture-julia.notebooks`
- This covers part of Julia Essentials and Fundamental Types
- Other more advanced lectures, not required for this course, are
  - → Introduction to Types and Generic Programming
  - → Generic Programming
  - → Visual Studio and Other Tools