A basic problem about any body of data is to make it more easily and effectively handleable by minds – our minds, her mind, his mind.

*John W. Tukey, Exploratory Data Analysis, Addison-Wesley, 1977*
Section 1

Get Started
Interface Cuteness

- Matlab uses `help`, Julia switches into help mode by typing `?`
  - `lookfor` in Matlab becomes `apropos`, e.g.,
    `apropos("determinant")`

- In Julia can access OS commands by typing `;`, e.g.,
  `;pwd`

- Useful things to know
  - history with up and down keys
  - matches partial strings
  - auto-complete with TAB

- Standard shell-commands
  - `Ctrl-c` interrupt process
  - `Ctrl-a` start of the line
  - `Ctrl-e` end of the line
  - `Ctrl-d` exit

- Startup file `~/.juliarc.jl`
Other useful bits and pieces

- Comments in shell-style #
- Functions that modify their arguments have a name like sort!
- Useful commands

  ```
  whos()
  @which sin(2)
  versioninfo()
  ```

- Numerical constants

  ```
  pi
golden
e
im
eulergamma
  ```

- Long numbers: 1_000_000
- Others useful constants

  ```
  JULIA_HOME # path to julia executable
  nothing    # function that returns void
  ```
Section 2

Plotting
There are several plotting packages

- **PyPlot**: emulates Matlab, through Python’s matplotlib
- **Gadfly**: emulates R’s ggplot
- **Plots**: aims to become front end for all backends
- **GR, UnicodePlots, Plotly, PlotlyJS, Vega, Winston, StatsPlots, PlotRecipes, GLVisualize, PGFPlots, Qwt, ...**
You should have it installed (see startup sheet)
  ▶ it uses PyCall to call Python
  ▶ uses Julia’s multimedia backend to display using various Julia graphical backends (Qt, GTK, ...)
  ▶ it should be fairly portable

Syntax is intended to be similar to Matlab
  ▶ as implemented in matplotlib
  
```
using PyPlot
x = linspace(0,2*pi,1000);
y = sin.(3*x + 4*cos.(2*x));
plot(x, y, color="red", linewidth=2.0,
     linestyle="--")
title("A sinusoidally modulated sinusoid")
```
Main commands

You can get a listing of commands by typing `PyPlot.TAB TAB`.

Some examples

```
plot
gcf()
xlim
xlabel
xkcd
surf
bar
figure
fill
pie
text
scatter
```

When running in a script, you need to use `show()` to get the fig to display.
Example 1

using PyPlot

```python
x = 0:0.1:2*pi;
y = 0:0.1:pi;
X = repmat(x, 1, length(y));
Y = repmat(y', length(x), 1);
S = [cos(x[i]) + sin(y[j]) for i=1:length(x),
     j=1:length(y)]
surf(X, Y, S, cmap=ColorMap("jet"), alpha=0.7)
xlabel("x")
ylabel("y")
```
Example 2

using PyPlot
xkcd()
plot([0,1], [0,1])
title(L"Plot of $\Gamma_3(x)$")
savefig("plot.svg")
    # or PNG or EPS or PDF

LaTeXString defined by L"...."
More Examples

https://gist.github.com/gizmaa/7214002
https://lectures.quantecon.org/jl/julia_plots.html
Section 3

A Stupidly Short Tour of Packages
Installing Packages

Packages are a collection of code encapsulated into a set of Modules, and (usually) put on GitHub in a standard format.

- Adding a package can be done in a few ways, but the most standard is
  
  ```julia
  Pkg.add("PyPlot")
  Pkg.update()
  ```

  - takes care of dependencies
  - installs code

- Get status, and see where code is

  ```julia
  Pkg.status()
  Pkg.Dir.path()
  LOAD_PATH
  ```
Using Packages

**Packages** are a collection of code encapsulated into a set of **Modules**, and (usually) put on GitHub in a standard format

- **Commands to use or import**
  
  ```
  using PyPlot
  import PyPlot
  ```

  - **using** simple access to all exported functions
  - **import** uses names space of module, *e.g.*, `PyPlot.plot`

- **Other ways to import code**

  ```
  include( "Code/my_code.jl" )
  reload( "PyPlot" )
  ```
Lots of Packages

https://pkg.julialang.org/

- 1518 registered packages!
- Some trending packages
  
  https://github.com/trending/julia
  
  - Deep Learning https://github.com/denizyuret/Knet.jl
  - IJulia is a Jupyter interactive environment
    https://github.com/JuliaLang/IJulia.jl
  - Gadfly is ggplot-like plotting
    https://github.com/GiovineItalia/Gadfly.jl
  - PyCall lets you call Python
    https://github.com/JuliaPy/PyCall.jl
  - Convex programming
    https://github.com/JuliaOpt/Convex.jl

- I will talk about a couple of direct use in Data Science
DataFrames

- Concept comes from R (as far as I know)
- Like a 2D array except
  - can have missing values
  - multiple data types
    - quantitative
    - categorical (strings)
  - labelled columns
- Nice mapping from Frame to CSV (or similar)

using DataFrames

data = readtable("Data/Titanic.csv",
    nastrings=["NA", "na", "n/a", "missing"])

head(data)
size(data)
showcols(data)
data[:,:Name]

temp = deepcopy(data)
push!( temp, @data([1314, "my bit", "nth", NA, "male"])
tail(temp)
deleterows!(temp, 3:5)
data[ data[:,:Sex] .=="female", : ]
data[ :height ] = @data( rand(size(data,1)) )
sort!(data, cols = [order(:Sex), order(:Age)])
JSON

- JSON = JavaScript Object Notation
- Data exchange format
  - increasingly popular
  - lightweight
  - portable
- Stores name/value pairs
  - so it maps to a Dictionary well
  - but lots of other data can be stored as JSON

http://www.json.org/
Download the following dataset, and put in a local folder called Data

https://raw.githubusercontent.com/corysimmons/colors.json/master/colors.json

```
import JSON
c = JSON.parsefile("Data/colors.json")
c["purple"]
JSON.print(c)
```
Distributions

- Package for probability distributions and associate facilities
  - moments
  - pdf, cdf, logpdf, mgf
  - samples
  - Estimation: MLE, MAP
- Included here because
  - it’s useful
  - it’s a nice example of a Julia package
    - type hierarchy used to provide structure to RVs
      - e.g., Distributions → Univariate → Continuous → Normal
    - multiple dispatch used to call correct version of generically named functions
    - easy to add a new one

https://juliastats.github.io/Distributions.jl/latest/
using Distributions
srand(123)

d = Normal(0.0, 1.0)
x = rand(d, 10)
quantile.( d, [ 0.5, 0.975] )
params(d)
minimum(d)
location(d)
scale(d)

x = rand(d, 100)
fit_mle(Normal, x)
Section 4

Parallel Processing
Julia Macros

- Macros look a bit like functions, but begin with @, e.g.,

```julia
@printf("Hello %s\n", "World!")
@printf "Hello %s\n"  "World!"
```

- Why?
  - Macros are parsed at compile time, to construct custom code for run time
    - e.g., for `@printf`, we want to interpret the **format string** at compile time,
    - In C, the printf function re-parses the format string each time it is called, which is inefficient
    - Also means that C compilers need to be very smart to avoid many hard-to-debug mistakes of the wrong types of arguments being passed to printf
Julia Macros

- Julia uses quite a few macros, and you can define your own
  ```julia
  @time [sin(i) for i in 1:100000]
  @which sin(1)
  @show 2 + 2
  macroexpand(quote @time sin(i) end)
  ```
- Macros can be MUCH faster ways of implementing code
  [https://statcompute.wordpress.com/2014/10/10/julia-function-vs-macro/](https://statcompute.wordpress.com/2014/10/10/julia-function-vs-macro/)
- Macros can be used to automate annoying bits of replicated code, e.g., `@time`
- It’s part of the **meta-programming** paradigm of Julia
  - ideas from Lisp
  - Julia code is represented (internally) as Julia data
  - so you can change the “data”
What Julia Does

1. Raw Julia code is parsed
   - converted into an Abstract Syntax Tree (AST), held in Julia
   - syntax errors are found
2. Create a deeper AST
   - Macros play here - they can create and modify *unevaluated* code
3. Parsed code is run
   - hopefully really fast
So what does that have to do with Parallel Programming?

- Julia has several functions and macros to aid in parallel processing.
- I think the coolest is the “Map/Reduce” functionality introduced by the @parallel macro.
  - maybe you can see why it is a macro?
Setting up for Multi-Processor Ops

There are two approaches for a single, multicore machine

> julia -p 4

julia > addprocs(3)
julia > procs()
julia > nprocs()

I’m not going to get into how to build a cluster
Map Reduce

Many simple processes can be massively parallelised easily by decomposing them into Map-Reduce operations.

- **Map**: apply an (independent) function or mapping to a small piece of data.
- **Reduce**: combine the results of all the mappings into a summary.

It’s a particularly good framework for multiple simulations run in parallel.
First make sure that all processes have the required environment

```julia
@everywhere cd("/home/mroughan/Presentation/Julia/Code")
@everywhere include("my_code.jl")
```

Now run parallelised loop, aggregating results with operator +

```julia
nheads = @parallel (+) for i = 1:200_000_000
    Int(rand(Bool))
end
```

But take care – data is not automatically shared!!!!!!!
Section 5

Tips and tricks
Type stability

Use `@time` to compare the speed of these two functions for large $n$

```julia
function t1(n)
    s = 0
    for i in 1:n
        s += s/i
    end
end

function t2(n)
    s = 0.0
    for i in 1:n
        s += s/i
    end
end
```
Don’t avoid loops

Use `@time` to compare the speed of these two functions for large $n$

```julia
function t1(n)
    x = zeros(n)
    for i in 1:n
        x[i] = i^2
    end
    return x
end

function t2(n)
    x = collect(1:n).^2
    end
```

Avoid global variables

- Apart from the usual arguments
- Hard for compiler to optimise around, because type may change
  - if you need them, and they don’t change, define them as constants
    ```
    const DEFAULT_VAL = 0
    ```
- Note variables defined in the REPL are global
- Execute code in functions, not global scope
  - write functions, not scripts
Pre-allocate outputs

Use \texttt{@time} to compare the speed of these two functions for large $n$

```julia
function t1(n)
    x = zeros(Int64, n)
    for i in 1:n
        x[i] = i^2
    end
    return x
end

function t2(n)
    x = [1]
    for i in 2:n
        push!(x, i^2)
    end
    return x
end
```
Access arrays in memory order, along columns

- 2D arrays stored in column order (as in Fortran)
  - C and Python `numpy` are in row order
- Accessing in this order avoids jumping around in memory
  - get the best value out of pipeline and cache
Lots more tips

- https://github.com/Gnimuc/JuliaSO
- http://blog.translusion.com/posts/julia-tricks/
- https://julialang.org/blog/2017/01/moredots
Standard Tools

- **Debugging** [https://github.com/Keno/Gallium.jl](https://github.com/Keno/Gallium.jl)
- **BenchmarkTools** package [https://github.com/JuliaCI/BenchmarkTools.jl](https://github.com/JuliaCI/BenchmarkTools.jl)
- **Profiler** [https://docs.julialang.org/en/latest/manual/profile/](https://docs.julialang.org/en/latest/manual/profile/)
- **Lint** package [https://github.com/tonyhffong/Lint.jl](https://github.com/tonyhffong/Lint.jl)
- **Literate programming (aka Knitr, ...)** [https://github.com/mpastell/Weave.jl](https://github.com/mpastell/Weave.jl), and iJulia
Standard Tools

- There is a lot more to learn
  - function definition
  - creating modules
  - types
  - interfaces to other languages
  - ...

- I tried to concentrate on things where I think it is hard to get started learning yourself
Final Comment

Julia is v.shiny, but it’s not all roses

- Current version is 0.6
  - each 0.1 increment has introduced “breaking” changes
  - the core is still evolving
  - it’s getting better, but change is painful

- Some libraries aren’t all there
  - stagnation, ...

- Plotting
  - argggh!
I don’t like endings, so here are some quotes to go on with.

We – or the Black Chamber – have a little agreement with [Knuth]; he doesn’t publish the real Volume 4 of the Art of Computer Programming, and they don’t render him metabolically challenged.

*Charles Stross, The Atrocity Archive, 2001*
Some more useful references

- https://github.com/trending/julia
Bonus frames