



# SKILLPILLS

## Skill Pill: Julia

### Lecture 2: Data Processing and Plotting

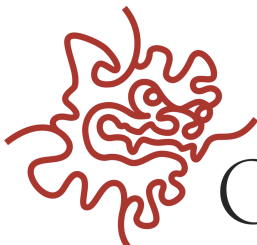
Juan Polo { material  
taken/based on  
Ankur Dhar &  
James Schloss }

*[juan.polo@oist.jp](mailto:juan.polo@oist.jp)*

*[ankurd@oist.jp](mailto:ankurd@oist.jp)*

*[james.schloss@oist.jp](mailto:james.schloss@oist.jp)*

February 24th, 2021



OIST

1 Arrays

2 Data Handling

3 Plotting

Arrays in Julia are defined using square brackets `[ ]`

By default arrays are row vectors, but can be transposed to column vectors

Arrays in Julia are defined using square brackets `[ ]`

By default arrays are row vectors, but can be transposed to column vectors

Sequences of numbers can be generated using the `:`

Ranges can be defined as `start:step:end`

To create a vector from a range use the function `collect()`

Arrays in Julia are defined using square brackets `[ ]`

By default arrays are row vectors, but can be transposed to column vectors

Sequences of numbers can be generated using the `:`

Ranges can be defined as `start:step:end`

To create a vector from a range use the function `collect()`

```
x = [1 2 3 4]           # 1x4 Array{Int64,2}
y = collect(1:4)         # 4-element Array{Int64,1}
z = [1, 2, 3, 4]         # 4-element Array{Int64,1}
m = [1 2; 3 4]           # 2x2 Array{Int64,2}
w=x'                     # 4x1 {Int64,Array{Int64,2}}
array = Int64[]          # 0-element Array{Int64,1}
```

Arrays in Julia are defined using square brackets `[ ]`

By default arrays are row vectors, but can be transposed to column vectors

Sequences of numbers can be generated using the `:`

Ranges can be defined as `start:step:end`

To create a vector from a range use the function `collect()`

```
x = [1 2 3 4]           # 1x4 Array{Int64,2}
y = collect(1:4)         # 4-element Array{Int64,1}
z = [1, 2, 3, 4]         # 4-element Array{Int64,1}
m = [1 2; 3 4]           # 2x2 Array{Int64,2}
w=x'                    # 4x1 {Int64,Array{Int64,2}}
array = Int64[]          # 0-element Array{Int64,1}
```

Please try to create a vector from 0 to 0.9 in steps of 0.1

Arrays in Julia are defined using square brackets `[ ]`

By default arrays are row vectors, but can be transposed to column vectors

Sequences of numbers can be generated using the `:`

Ranges can be defined as `start:step:end`

To create a vector from a range use the function `collect()`

```
x = [1 2 3 4]           # 1x4 Array{Int64,2}
y = collect(1:4)         # 4-element Array{Int64,1}
z = [1, 2, 3, 4]         # 4-element Array{Int64,1}
m = [1 2; 3 4]           # 2x2 Array{Int64,2}
w=x'                    # 4x1 {Int64,Array{Int64,2}}
array = Int64[]          # 0-element Array{Int64,1}
```

Please try to create a vector from 0 to 0.9 in steps of 0.1

**?range** This allows you to create a range without thinking too much. You can combine stop, step and length `collect(range(0,stop=10,length=10))`

`zeros(S)` Makes an array of size `S` filled with zeros

`ones(S)` Makes an array of size `S` filled with ones

`repeat(A,c,r)` Repeats array `A` column-wise `c` times and row-wise `r` times

`rand(S)` Generates array of size `S` with random numbers between 0 and 1

`Type[]` Creates empty array of type `Type`



Arrays are indexed by square brackets after the array `A[1,2]`.

Arrays are indexed by square brackets after the array `A[1,2]`.

```
data = rand(50,50)
data[1,:]
data[:,1]
inner = x*y
outer = y*x
square = x .* x
```

To execute scalar operations on an array, **you can use a `.` to the operator**. This is also applicable to functions as well.

Arrays are indexed by square brackets after the array `A[1,2]`.

```
data = rand(50,50)
data[1,:]
data[:,1]
inner = x*y
outer = y*x
square = x .* x
```

To execute scalar operations on an array, **you can use a `.` to the operator**. This is also applicable to functions as well.

**Multiply all elements of "x" by a constant "2" (careful with spaces)**

Arrays are indexed by square brackets after the array `A[1,2]`.

```
data = rand(50,50)
data[1,:]
data[:,1]
inner = x*y
outer = y*x
square = x .* x
```

To execute scalar operations on an array, **you can use a `.` to the operator**. This is also applicable to functions as well.

Multiply all elements of "x" by a constant "2" (careful with spaces)

`2 .* x`

Element-by-element binary operations on arrays of different sizes are possible. Dotted operators, `.+` or `.*`, are equivalent to broadcast calls

```
data = collect(0:.1:1)
round.(data)
f(x) = x+2
f.(data)
```

Element-by-element binary operations on arrays of different sizes are possible. Dotted operators, `.+` or `.*`, are equivalent to broadcast calls

```
data = collect(0:.1:1)
round.(data)
f(x) = x+2
f.(data)
```

Create a vector “v” of  $\text{dim} = 2$  (e.g. elements (1, 2)) and add it to the matrix “m” of  $\text{dim} = 2 \times 2$

Element-by-element binary operations on arrays of different sizes are possible. Dotted operators, `.+` or `.*`, are equivalent to broadcast calls

```
data = collect(0:.1:1)
round.(data)
f(x) = x+2
f.(data)
```

Create a vector “v” of  $\text{dim} = 2$  (e.g. elements (1, 2)) and add it to the matrix “m” of  $\text{dim} = 2 \times 2$

```
v = [1, 2]
m = [1 2; 3 4]
v .+ m
```

Element-by-element binary operations on arrays of different sizes are possible. Dotted operators, `.+` or `.*`, are equivalent to broadcast calls

```
data = collect(0:.1:1)
round.(data)
f(x) = x+2
f.(data)
```

Create a vector “v” of  $\text{dim} = 2$  (e.g. elements (1, 2)) and add it to the matrix “m” of  $\text{dim} = 2 \times 2$

```
v = [1, 2]
m = [1 2; 3 4]
v .+ m
```

Extra cool things

```
data[data .< 0.5]
```



# Data Handling

To begin working with files, you must know where your working directory is. When launching Julia from an application menu (Windows/MacOS), the default directory is predefined. For Linux it will be the home directory.

```
pwd()  
# cd("C:\\Users\\M\\Documents\\JuliaStuff")  
readdir()
```

You can use `pwd()` to print your working directory, and `cd()` to change it to whichever new directory you would like. Once you are in the directory you want, you can list the files within using `readdir()`.

The simplest data files are often delimited text files or CSV files, which can be manipulated like any other variable in Julia. To load any general delimited file, load the DelimitedFiles package. Reading in these files will automatically generate 1D or 2D arrays depending on the data being read in.

```
julia> using DelimitedFiles
julia> data = rand(50,50)
julia> writedlm("Random.txt",data)
julia> randData = readdlm("Random.txt")
```

Similarly, any 1D or 2D can be written to a file, with the actual delimiter being based on the file extension used (txt for space and csv for comma).

Typical CSV files are a bit more complex than standard delimited files, with headers or labels. For this use case the CSV package is recommended. When reading in a file, the first row will be taken as the header row, but this can be explicitly defined.

```
using CSV, DataFrames
data =
    DataFrame(CSV.File("simplemaps-worldcities-basic.csv"))
names(data)
populations = data[:, :pop]
populations = data.pop
data[5677, :]
```

Small trick, you can write "data." and press "TAB" twice to see the available columns

```
data[data.city .== "Barcelona",:]
findall(x->x=="Barcelona",data.city)
```

## Exercise 1

Creates a file `squares.txt` consisting of the first 5 square numbers

## Exercise 2

Write a script which creates a new file called `large_cities.txt`. The file should contain one line for each of the cities which have a population larger than 10,000,000., formatted as follows:

Buenos Aires, Argentina: population 11862073

Sao Paulo, Brazil: population 14433147.5

...

## Exercise 1

Creates a file `squares.txt` consisting of the first 5 square numbers

## Exercise 2

Write a script which creates a new file called `large_cities.txt`. The file should contain one line for each of the cities which have a population larger than 10,000,000., formatted as follows:

Buenos Aires, Argentina: population 11862073

Sao Paulo, Brazil: population 14433147.5

...

Fast way to look at data using Dataframes `data[data.pop .> 107,:]`

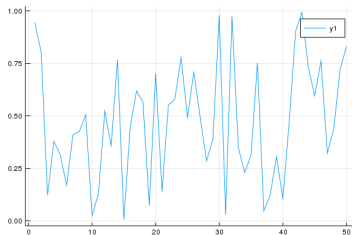
# Plots

To start with, let's add and load the general Plots package

```
# inside pkg>
add Plots
# inside REPL
using Plots
# plot(Plots.fakedata(50))
```

This will let us call the general Plot commands, in this case plot plots 1D data as a line plot.

More information can be found at <https://docs.juliaplots.org/latest/>





`scatter(X,Y)` Scatter plot data with XY coordinates

`bar(x,y)` Bar plot following similar rules to plot

`histogram(x,bins=n)` Plots histogram of 1D data in n bins.

`plot( $\theta$ ,r,proj=:polar)` Polar plot of data following r and  $\theta$

`heatmap(x,y,z)` Plots heatmap following XY axes with intensity array z

`fakedata(L,S)` Generates random S numbers of series data of length L

`savefig(filename)` Saves a generated plot as an image file

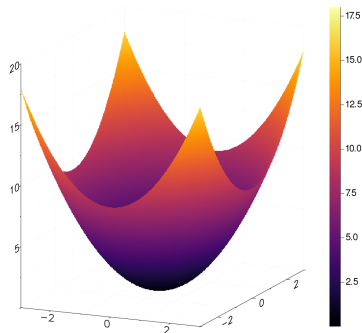
There are a couple of additional options for plotting 3D data:

`surface(x,y,z)` Draws surface in 3D space

`contour(x,y,z)` Draws contours on 2D plane

`heatmap(x,y,z)` Draws a heatmap on 2D plane

The plotting commands `plot`, `scatter`, `bar`, and `heatmap` also can accept 3D data.

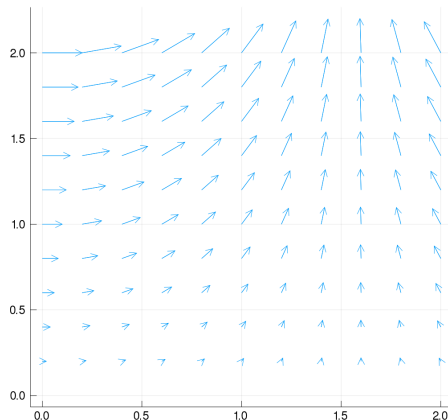


For vector data the current option is  
quiver:

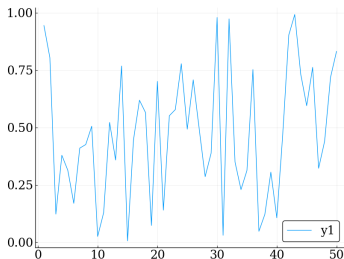
```
help?> quiver  
search: quiver quiver!
```

```
quiver(x,y,quiver=(u,v))  
quiver!(x,y,quiver=(u,v))
```

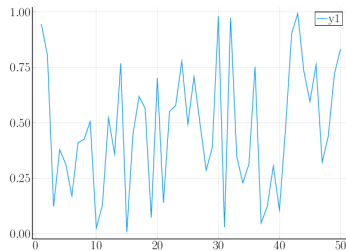
Make a quiver (vector field)  
plot. The  $i$ th vector  
extends from  $(x[i], y[i])$   
to  $(x[i] + u[i], y[i] + v[i])$ .



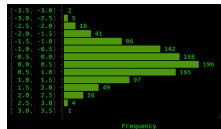
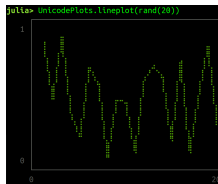
These commands are agnostic to the plotting backend, meaning they will work with a number of plotting engines in similar fashion. Each backend has pros and cons, but most commonly used are **GR** and **PyPlot**.



**Figure:** PyPlot  
using PyPlot; PyPlot.plot(rand(10))



**Figure:** PGFPlots



**Figure:** UnicodePlots  
using UnicodePlots; UnicodePlots.lineplot(rand(10))  
UnicodePlots.histogram(randn(1000), nbins=15, closed=:left)

You can load one or multiple plot libraries but they might have conflicts. Using “Plots.plot” or “PyPlot.plot” will force a particular library.

The default backend can be changed by “backend(:gr)” or “backend(:pyplot)”

You can load one or multiple plot libraries but they might have conflicts. Using “Plots.plot” or “PyPlot.plot” will force a particular library.

The default backend can be changed by “backend(:gr)” or “backend(:pyplot)”

```
using Plots; import PyPlot; plt = PyPlot;  
# Import avoids conflicts, however PyPlot.XXX or plt.XXX is  
# need to use that library
```

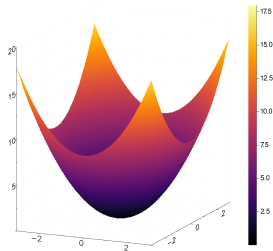
```
plot(rand(20))      # generates a plot  
plot(rand(20))      # generates a new plot  
plot!(rand(20))     # add line to previous plot
```

```
p1 = plot(rand(20)); # the semicolon will prevent plotting  
p2 = plot(rand(20));  
p3 = plot(rand(20));  
plot(p1,p2,p3) # Automatically generates a layout
```

```
# Manually generating a layout  
plot(p1,p2,p3, layout = @layout grid(3,1))  
plot(p1,p2,p3, layout = @layout [a [b; c]])
```

```
using Plots; import PyPlot; plt = PyPlot;  
# we have to use plt.XXX for the main commands  
  
fig, axs = plt.subplots(ncols=2,nrows=2);  
  
axs[1,1].plot(rand(20));  
axs[1,2].plot(rand(20));  
axs[2,1].plot(rand(20));  
axs[2,2].plot(rand(20));  
  
axs[1,1].set_title("(a)")  
axs[1,1].set_xlabel("x_label")  
axs[1,1].set_ylabel("y_label")  
  
plt.clf(); # this will clear the figure
```

```
using Plots
x = y = collect(-1:0.01:1)
#surface(x,y,(x,y)->x^2+y^2)
#heatmap(x,y,(x,y)->x^2+y^2)
f(x,y) = x^2 + y^2
surface(x,y,f.(x,y'))
contour(x,y,f.(x,y'))
heatmap(x,y,f.(x,y'))
```





Each of these commands can be expressed in-line with the plotting command or beforehand within a call to the plotting backend.

`font(fontname,size)` Defines a Font object with a given size

`size=(X,Y)` Sets size of plot to X by Y pixels

`xlabel=string` Sets X-Y labels to string, also `ylabel`

`title=string` Sets title to string, also `colorbar_title` for heatmap.

`xtickfont=font` Sets the font of x tick marks, also `ytickfont`, `titlefont`, `guidelfont`

`left_margin=length` Sets margin for left side of plot, also `top_margin`, `bottom_margin`, and `right_margin`

`yscale=:log10` Sets x scale to log10, also `yscale`

## Exercise 3

Read data.txt given in the Public Folder and plot the results. What do you see?

## Exercise 4

Plot a histogram of the longitudes of the world's cities. What is the mean and median longitude?

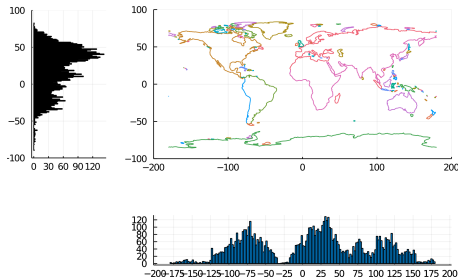
using Plots, DataFrames, CSV

```
data = DataFrame(CSV.File("simplemaps-worldcities-basic.csv"))
```

```
using Plots, Images, ImageMagick, DataFrames, CSV

data =
    DataFrame(CSV.File("simplemaps-worldcities-basic.csv"))

histogram(data.lng, bins=200)
histogram(data.lat)
```



```
using Plots, Images, ImageMagick, DataFrames, CSV, Shapefile

data = DataFrame(CSV.File("simplemaps-worldcities-basic.csv"))

shp =
    Shapefile.shapes(Shapefile.Table("ne_110m_coastline.shp"));

p3 = histogram(data.lng, bins=200, xticks = (-
    200:25:200), label=false);
p2 = plot(shp, xlim=(-200,200), ylim=(-100,100));
p2_size = p2.attr.explicit[:size];

p1 = histogram(data.lat, bins=200, ylim = (-100,100),
    orientation=:horizontal,
    aspect_ratio=p2_size[1]/p2_size[2], label=false);

plot(p1,p2,p3,p3,
    layout=@layout [[a{0.2w} b{0.8h}]; [c d]])
```

Last Session Problem Sets with James

Also next week Advanced Topics with Valentin!