2.3 Lab: Introduction to Julia

2.3.1 Getting started

Throughout this notes, we will be using the Integrated Development Environment Visual Studio Code as well as the Jupyter Notebook extension. Julia version used in this tutorials is 1.10.1. To install the packages in Julia, use the Pkg library, typing "using Pkg", and then "Pkg.add("Package_name"). To verify the installed packages, status function can be helpful: "Pkg.status()". This will return a list of the installed packages and their version. Packages-version used in this Lab are the following:

CSV → v0.10.13
CSVFiles → v.1.0.2
DataFrames → v.0.22.7
DelimitedFiles → v.1.9.1
Distributions → v0.24.18
GLM → v1.9.0
Statistics → v1.10.0

2.3.2 Basic Commands

In this section, some of the Julia basic commands will be reviewed. The objective it is have a solid understanding of the basic for further chapters. If you wish to find a deeper description of the topics the following link contains the Julia Documentation: https://docs.julialang.org/en/v1/

To print an output on the screen in Julia we use the println() function and the text between double quotes "".

In [1]:
```
println("fit a model with ", 11, " variables")
```

Out [1]:
```
fit a model with 11 variables
```

Julia's REPL is very friendly and by typing the name of a function plus a question mark "?", results in a short summary.

In [2]: Julia > `?println`

Out [2]:
```
search: println printstyled print sprint isprint

println([io::IO], xs...)

Print (using print) xs to io followed by a newline. If io is not supplied, prints to
the default output    stream stdout.

See also printstyled to add colors etc.
```

```
Examples
========

julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello", ',', " world.")

julia> String(take!(io))
"Hello, world.\n"
```

Integers additions is equal to Python:

In [3]:
```
3 + 5
```

Out [3]: 8

As we already mentioned above, text or string data types can be declared in Julia by “”, note that by using single quotes ‘’, a simple character is declared. Then if a word between ‘’ is declared then we see an ERROR message.

In [4]:
```
‘Hello’
```

Out [4]:
```
ERROR: ParseError:
# Error @ REPL[3]:1:2
'Hello'
#└───┘ ── character literal contains multiple characters
Stacktrace:
[1] top-level scope
@ none:1
```

We can join or concatenate 2 strings using the multiplication “*” operator(In Python this operator duplicate n-times the string).

In [5]:
```
"hello" * "  " * "world"
```

Out [5]: "hello world"

Julia handles several classes of data structures and collections. Collections are groups of elements. Main collections are sets, tuples, named tuples, dictionaries and arrays. The following sentece assigns an array to variable x. By default Julia returns the result of the last line in the code, to avoid this just add ; (semicolon) at the end.

In [6]:
```
x = [3, 4, 5]
```

Out [6]: 3-element Vector{Int64}:
```
3
4
5
```

To add elementwise 2 arrays, we just simply use the addition operator "+". (Another altenative to apply an elemenwise operation it is with the dot "." symbol before any of the operators +, -, *, / or also it is possible to use the @macro syntax, by putting @. before the expression).

In [7]:
```
y = [4, 9, 7]
 x + y
```

Out [7]: 3-element Vector{Int64}:
```
7
13
12
```

On the other hand, to concatenate 2 arrays just use the semicolon symbol(;).

In [8]:
```
[x ; y]
```

Out [8]: 6-element Vector{Int64}:
```
3
4
5
4
9
7
```

2.3.3 Introduction to Numerical Julia

To create a two-dimensional array in Julia, we can type:

In [9]:
```
x = [1 2 ; 3 4]
```

Out [9]: 2×2 Matrix{Int64}:
```
1 2
3 4
```

To display the dimensions of the matrix, we use the ndims() function:

In [10]:
```
ndims(x)
```

Out [10]: 2

Aditionally, if one of the elements contains a non-integer number(Float), the rest of the elements are converted to Float datatypes.

In [12]:
```
eltype([1 2 ; 3.0 4])
```

Out [12]: `Float64`

A second alternative to create a float number matrix, it is by declaring the datatype:

In [13]:
```
Float64[1 2 ; 3 4]
```

Out [13]:
```
2×2 Matrix{Float64}:
1.0 2.0
3.0 4.0
```

x is a 2-dimensional array, size() functions returns the number of rows as well as the number of columns:

In [14]:
```
size(x)
```

Out [14]: `(2, 2)`

The expression sum(object) sums all of the elements contained in the object.

In [15]:
```
sum(x)
```

Out [15]: `10`

Reshape function changes the dimensions of the array, with the same elements, by passing the array´s name and a tuple with the new dimensions (rows, columns).

In [16]:
```
y_col_order = [1 2 3 4 5 6]
println(“beginning y:\n”, y_col_order)
y_reshape_co = reshape(y_col_order, (2, 3))
println(“reshaped y:\n”, y_reshape_co)
```

Out [16]:
```
beginning y:
[1 2 3 4 5 6]
reshaped y:
[1 3 5; 2 4 6]
```

Notice tha Julia uses a column-major ordering to reshape the arrays, unlike Python which uses row-major ordering. Permutedims() functions let us transpose arrays and can help us in changing this column-major order.

In [17]:
```
y_row_order = [1 2 3 4 5 6]
println(“beginning y:\n”, y_row_order)
y_reshape_ro = permutedims(reshape(y_col_order, (3, 2)), [2, 1])
println(“reshaped y:\n”, y_reshape_ro)
```

Out [17]:
```
beginning y:
[1 2 3 4 5 6]
reshaped y:
[1 2 3; 4 5 6]
```

Julia is 1-based indexing, we can acces the first element by typing y_reshape_ro[1, 1].

In [18]:
```
y_reshape_ro[1, 1]
```

Out [18]: 1

To acces the element in the second row and the third column, we type y_reshape_[2, 3].

In [19]:
```
y_reshape_ro[2, 3]
```

Out [19]: 6

y_row_order[3] returns the third element in the array.

In [20]:
```
y_row_order[3]
```

Out [20]: 3

Now, let´s change the first element of the y_reshape_ro.

In [21]:
```
println(“y before we modify y_reshape:\n”, y_row_order)
println(“y_reshape before we modify y_reshape:\n”, reshape_ro)
y_reshape_ro[1, 1] = 5
println(“y_reshape after we modify its top lef elemten:\n”, y_reshape_ro)
```

Out [21]:
```
y before we modify y_reshape:
[1 2 3 4 5 6]
y_reshape before we modify y_reshape:
[5 2 3; 4 5 6]
y_reshape after we modify its top left element:
[5 2 3; 4 5 6]
```

Arrays are muttable objects in Julia, tuples are not, as a consequence, we cannot change a tuple´s element.

In [22]:
```
my_tuple = (3, 4, 5)
my_tuple[1] = 2
```

Out [22]:
```
MethodError: no method matching setindex!(::Tuple{Int64, Int64, Int64}, ::Int64, ::Int64)
Stacktrace: [1] top-level scope @ c:\Users\charl\Desktop\VS Code\2024\ISL 2024\CH-2\Lab-2 P1 Julia.ipynb:2
```

In summary, the functions size(), ndims() and permutedims() help us to obtain some properties of the array. Size results in a tuple containing the dimensions, ndims returns the number of dimensions and permutedims returns the traspose of the array. Also, we can obtain the traspose of a matrix adding the simple quote symbol A’.

In [23]:
```
res = size(y_reshape_ro), ndims(y_reshape_ro), y_reshape_ro’
```

Out [23]: `((2, 3), 2, [5 4; 2 5; 3 6])`

The three elements are stored on a tuple object(a tuple is declared separating its elements by a come). Typeof functions identifies the object type.

In [24]:
```
typeof(res)
```

Out [24]: `Tuple{Tuple{Int64, Int64}, Int64, LinearAlgebra.Adjoint{Int64, Matrix{Int64}}}`

Sometimes we must apply some operation to arrays. To apply an elementwise operation to an array we just add a dot(.) before the operator, for example to compute the square root.

In [25]:
```
sqrt.(y_reshape_ro)
```

Out [25]:
```
2×3 Matrix{Float64}:
2.23607 1.41421 1.73205
2.0 2.23607 2.44949
```

To square the elements, we use the symbol ^.

In [26]:
```
y_reshape_ro.^2
```

Out [26]:
```
2×3 Matrix{Int64}:
25 4 9
16 25 36
```

A second alternative to compute the square root it is by raising to the power or 0.5 or ½.

In [27]:
```
y_reshape_ro.^(1/2)
```

Out [27]:
```
2×3 Matrix{Float64}:
2.23607 1.41421 1.73205
2.0 2.23607 2.44949
```

Sometimes in some analysis we need to use random numbers, in such cases we it is useful to generate those numbers. Julia contains the packages Random and Distributions to accomplish these goals. To install a package in Julia, we use the pakage manager. In the REPL we type] and then add "Pakage_name". Once the package is installed we import it with the keyword using "Pakage_name". To see more details about the Package manager in Julia you can view the documentation here: https://docs.julialang.org/en/v1/stdlib/Pkg/.

In [28]:
```
using Random, Distributions
Random.seed!(42)

d = Normal()
x = rand(d, 50)
```

Out [28]:
```
50-element Vector{Float64}:
 -0.36335748145177754
  0.2517372155742292
 -0.31498797116895605
 -0.31125240132442067
  0.8163067649323273
  ...
```

For simplicity we just showed the firts 5 elements of the vector. The Normal() function admits arguments like the loc and scale. Creating a second array with a mean = 50, std = 1 and 50 elements.

In [29]:
```
y = x + rand(Normal(50, 1), 50)
```

Out [29]:
```
50-element Vector{Float64}:
 49.6985752588623
 50.530143029738234
 49.089187613467
 49.73540698824896
 51.90210078647561
 ...
```

To obtain the correlation matrix between the x and y vectors, we might use the cor() function.

In [30]:
```
cor(x, y)
```

Out [30]:
```
2×2 Matrix{Float64}:
 1.0        0.675424
 0.675424   1.0
```

Like Python, in Julia each time we call the function random(), we'll get diffetent results. For example:

In [31]:
```
println(rand(Normal(5, 1), 2))
println(rand(Normal(5, 1), 2))
```

Out [31]:
```
[3.6042742415429947, 4.7619854550235825]
[3.6207336774070766, 3.3092242444101405]
```

To avoid this situation, Julia contains some algorithms for random numbers. One of them is MersenneTwister. Putting a seed to the MersenneTwister algorithm we guarantee the reproducibility of the results.

In [32]:
```
rng = MersenneTwister(42)
println(rand!(rng, rand(Normal(5, 1), 2)))
rng_2 = MersenneTwister(42)
println(rand!(rng_2, rand(Normal(5, 1), 2))
```

Out [32]:
```
[0.5331830160438613, 0.4540291355871424]
[0.5331830160438613, 0.4540291355871424]
```

The Statistics package contains functions for descriptive analysis. Some of the functions available are mean(), var() and std() for average, variance and standard deviations of arrays.

In [33]:
```
using Statistics

rng = MersenneTwister(1)
y = rand!(rng, rand(Normal(), 10))
mean(y)
```

Out [33]: `0.47928971924837915`

In [34]:
```
var(y), sum((y .- mean(y)).^2)/(length(y)-1)
```

Out [34]: `(0.13361068170423643, 0.13361068170423643)`

By default it is computed the sample variance, as you see the n-1 in the denominator of the above equation. If we are interested in the population variance, we can use the argument corrected=false in the var() function or just remove the -1 in the equation.

In [35]:
```
var(y, corrected=false), sum((y .- mean(y)).^2)/(length(y))
```

Out [35]: `(0.12024961353381278, 0.12024961353381278)`

We can compute the standard deviation through the square root of the variance or with the function std().

In [36]:
```
sqrt(var(y)), std(y)
```

Out [36]: `(0.3655279492791713, 0.3655279492791713)`

The functions mean(), var() and std() are also applicable to a matrix components(rows and columns). To verify this, let's create randomly a 10x3 matrix.

In [37]:
```
rng =MersenneTwister(100)
X = rand!(rng, rand(10, 3))
```

Out [37]:
```
10×3 Matrix{Float64}:
 0.260125    0.172707    0.249832
 0.190313    0.52399     0.348241
 0.660911    0.557837    0.0660258
 0.0671932   0.893169    0.318286
 0.9676      0.199381    0.386709
 0.645691    0.0893719   0.00399661
 0.545968    0.307374    0.579446
 0.526845    0.324303    0.0316995
 0.972755    0.688505    0.35467
 0.868194    0.869716    0.502858
```

In [38]:
```
std(X, dims=1)
```

Out [38]:
```
1×3 Matrix{Float64}:
 0.319487 0.288838 0.196291
```

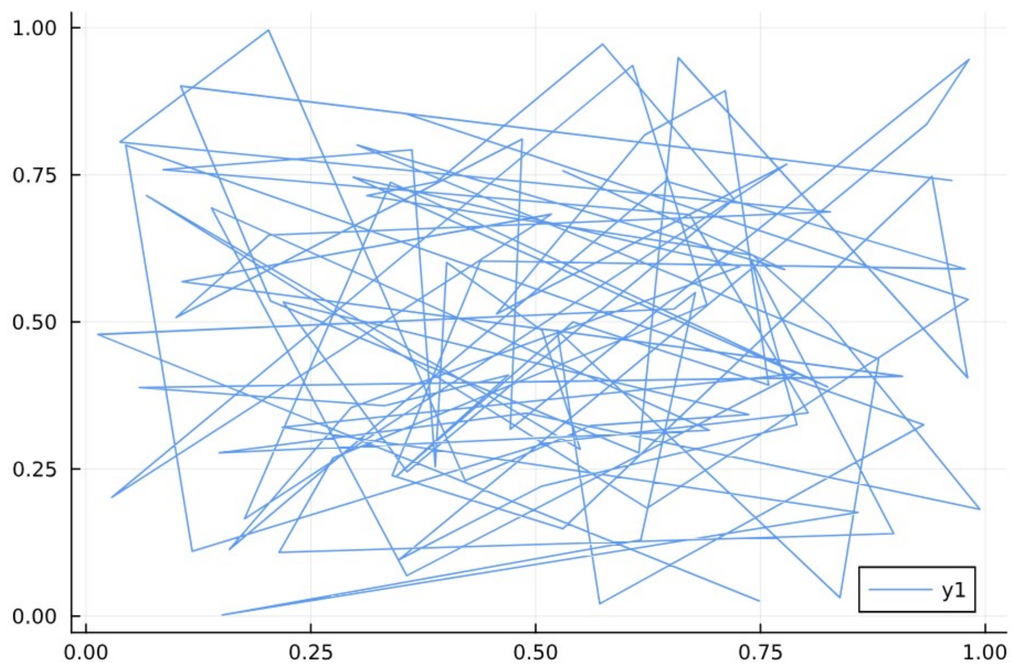
2.3.4 Graphics

There are several packages or libraries for visualizations in Julia. Some of them are Plots.jl, Makie.jl, Gadgly.jl, Vega.jl and VegaLite. In this tutorial we will be mainly using the Plots library.

In [39]:
```
using Plots, Distributions, Random

rng = MersenneTwister(5)
x = rand!(rng, rand(Normal(), 100))
y = rand!(rng, rand(Normal(), 100))
plot(x, y)
```

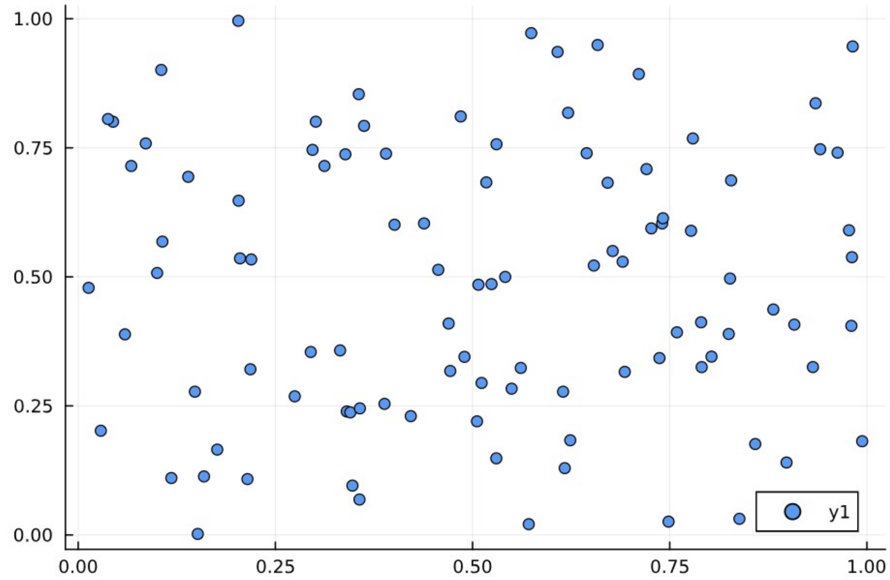
Out [39]:



If we prefer we can build a scatter plot instead of a line plot with the scatter() function.

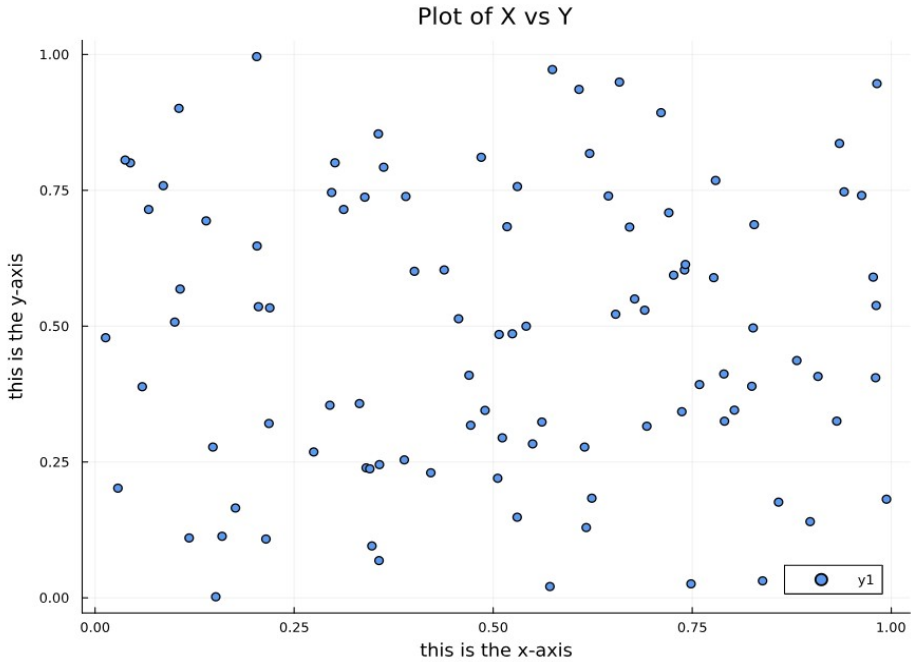
In [40]:
```
scatter(x, y)
```

Out [40]:



In [41]:
```
scatter(x, y, xlabel = “this is the x-axis”, ylabel = “this is the y-axis”,
title = “Plot of X vs Y”, size=(800, 600))
```

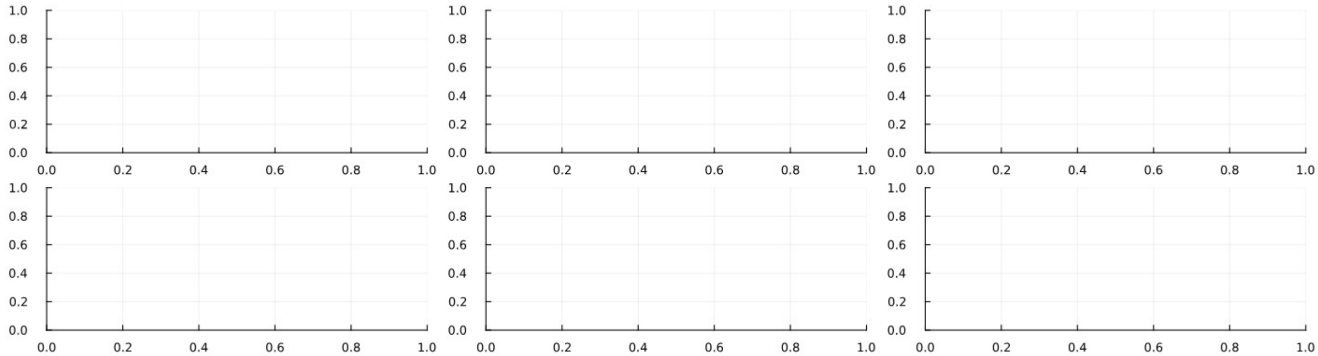
Out [41]:



To show several subplots in Julia, the layout argument of the plot() function can help us.

In [42]:
```
plot(layout=(2, 3), size=(1300, 350)
```

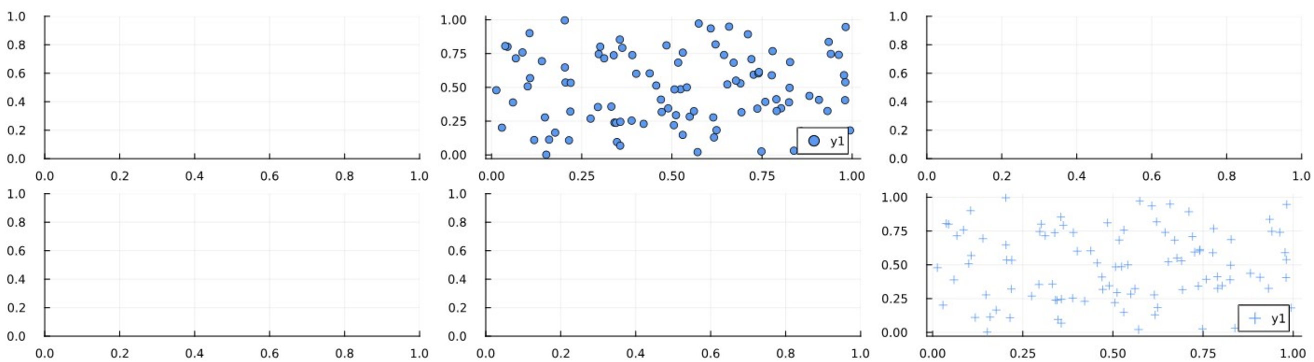
Out [42]:



Now, we can add plots to our template or figure. We add scatter plots, one in the first row second column and another in the second row and third column.

In [43]:
```
p1 = scatter(); p3 = scatter; p4 = scatter(); p5 = scatter()
p2 = scatter(x, y, markershape=:circle)
p6 = scatter(x, y, markershape=:+)
```

Out [43]:



To save the plots, savefig functions() and as arguments we type the name of the file and extensions, formats available are png or pdf.

In [44]:
```
savefig(“Figure.png”)
savefig(“Figure.pdf”)
```

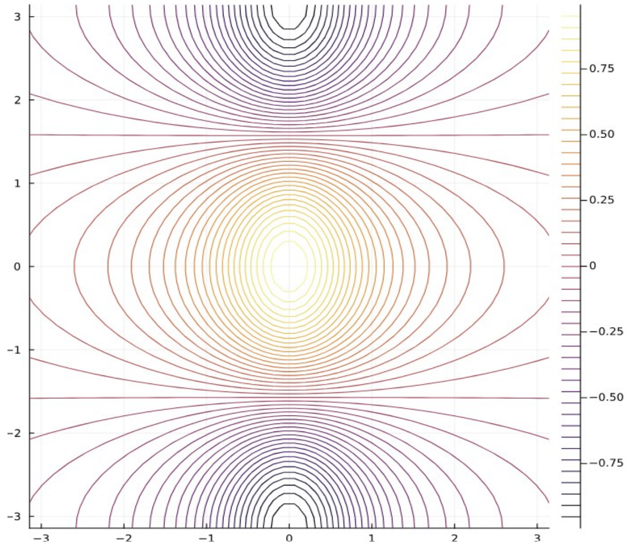
Also we can change range of the x-variable and save it again.

In [45]:
```
p2 = scatter(x, y, markershape=:circle, xlims=(-1, 1))
plot(p1, p2, p3, p4, p5, p6, layout=(2, 3), size=(1300, 500), dpi=400)
png(“Figure_updated.png”)
```

For three-dimensional data, contour() function is capable to plot a map. LinRange() function returns n number of elements on a specified range linearly.

In [46]:
```
x = LinRange(-pi, pi, 50)
y = x
f = (cos.(y)) .* (1 ./ (1 .+ x.^2))'
contour(x, y, f, levels=45)
```

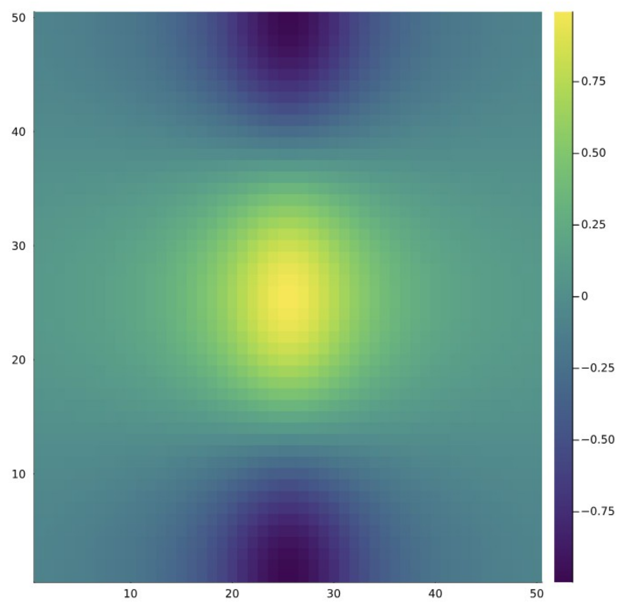
Out [46]:



Or we could gave plot a heatmap instead of a countour map.

In [47]:
```
heatmap(f, size=(500, 500), c=:viridis)
```

Out [47]:



2.3.5 Sequences and Slice Notation

To create a linearly space sequence of number we can use the functions LinRange() and collect().

In [48]:
```
seq1 = collect(LinRange(0, 10, 11))
```

Out [48]:
```
11-element Vector{Float64}:
  0.0
  1.0
  2.0
  3.0
  4.0
  5.0
  6.0
  7.0
  8.0
  9.0
  10.0
```

range() and collect() functions generates a sequence of numbers with a step of 1. A second option it is just to use the syntax init : end in conjuntion with the collect function.

In [49]:
```
seq2 = collect(range(0, 9))
```

Out [49]:
```
10-element Vector{Float64}:
  0.0
  1.0
  2.0
  3.0
  4.0
  5.0
  6.0
  7.0
  8.0
  9.0
```

In [50]:
```
seq3 = collect(0:9)
```

Out [50]:
```
10-element Vector{Float64}:
  0.0
  1.0
  2.0
  3.0
  4.0
  5.0
  6.0
  7.0
  8.0
  9.0
```

Similar to Python, to retrieve a slice of a sequence or a collection in Julia. Just remember that Julia is a one-indexing and last-inclusive element language. For example:

In [51]:
```
"hello world"[4:6]
```

Out [51]:
```
"lo "
```

Also to get a portion of the string, we can employ the substring() function. Note the capital S in the function.

In [52]:
```
SubString("hello world", 4, 6)
```

Out [52]:
```
"lo "
```

2.3.6 Indexing Data

Let's begin creating a 2-dimensional array.

In [53]:
```
A = reshape(0:15, 4, 4)'
```

Out [53]:
```
4×4 adjoint(reshape(::UnitRange{Int64}, 4, 4)) with eltype Int64:
  0   1   2   3
  4   5   6   7
  8   9  10  11
 12  13  14  15
```

To retrieve the element located in the second row and third column, we type.

In [54]:
```
A[2, 3]
```

Out [54]:
```
6
```

To retrieve the element located in the second row and third column, we type.

Indexing Rows, Columns and Submatrices

If we need to retrieve several rows and all the columns, we can pass a list with the row numbers and the colon ":" symbol, this symbol means all the columns.

In [55]:
```
A[[2, 4], :]
```

Out [55]:
```
2×4 Matrix{Int64}:
  4   5   6   7
 12  13  14  15
```

To select the first and third columns, we pass [1, 3] as the second argument and the rows selecting all ":".

In [56]:
```
A[:, [0, 2]]
```

Out [56]:
```
4×2 Matrix{Int64}:
  0   2
  4   6
  8  10
 12  14
```

To get a subset of the original matrix, this is the insersection o the second and fourth row and the first and third columns.

In [57]:
```
A[[2, 4], [1, 3]]
```

Out [57]:
```
2×2 Matrix{Int64}:
  4    6
 12   14
```

Boolean Indexing

In Julia a boolean value of true is equivalent to 1 or 1.0, and a false is equivalent to 0 or 0.0.

In [58]:
```
keep_rows = Bool.(zeros(size(A)[1]))
```

Out [58]:
```
4-element BitVector:
 0
 0
 0
 0
```

Modifying two of the elements to true.

In [59]:
```
keep_rows[[2, 4]] .= true
keep_rows
```

Out [59]:
```
4-element BitVector:
 0
 1
 0
 1
```

To verify the equality, put the comparison operator “==”.

In [60]:
```
keep_rows == [false, true, false, true]
```

Out [60]:
```
true
```

With the Boolean indexing we can access to the second and fourth row, first, third and fourth colums as follows.

In [61]:
```
keep_rows == [false, true, false, true]
```

Out [61]:
```
2×3 Matrix{Int64}:
  4   6   7
 12  14  15
```

2.3.7 Loading Data

To load a file in Julia one of the libraries that we might use it is CSV. jl. If the file is in the same location of the notebook, we just need to write the name of the file and his extension. For those cases where the file is not in the same location, we add the full path.

In [62]:
```
using CSV, DataFrames

auto = CSV.read("C:/Users/charl/Desktop/VS_Code/2024/ISL_2024/Datasets/Auto.csv", DataFrame)
first(auto, 5)
```

Out [62]:
5×9 DataFrame

Row	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
	Float64	Int64	Float64	String3	Int64	Float64	Int64	Int64	String
1	18	8	307	130	3504	12	70	1	chevrolet chevelle malibu
2	15	8	350	165	3693	11.5	70	1	buick skylark 320
3	18	8	318	150	3436	11	70	1	plymouth satellite
4	16	8	304	150	3433	12	70	1	amc rebel sst
5	17	8	302	140	3449	10.5	70	1	ford torino

Also it is possible to read a .data file using the package DelimitedFiles.

In [63]:
```
using DelimitedFiles

auto_data = readdlm("C:/Users/charl/Desktop/VS_Code/2024/ISL_2024/Datasets/Auto.data")
auto_data = DataFrame(auto_data[2:end, 1:end], string.(auto_data[1, 1:end]))
first(auto_data, 5)
```

To retrieve the data from the horsepower column, we can type auto.:horsepower or auto[!, “horsepower”].

In [64]:
```
auto_data.:horsepower
```

Out [64]:
```
397-element Vector{String3}:
 "130"
```

```
"165"
"150"
"150"
"140"
"198"
"220"
"215"
"225"
"190"
  ⋮
"112"
 "96"
 "84"
 "90"
 "86"
 "52"
 "84"
 "79"
 "82"
```

The unique() function returns an array containing one value for each unique value. In the results can see that the character “?” is part of the horsepower column. To avoid this data, we can filter the dataframe including all of those values different from “?”.

In [65]:
```
auto_data = auto_data[auto_data[!, “horsepower”] .!= “?”, :]
sum(auto_data[!, “horsepower”])
```

Out [65]: `40952.0`

Size function returns a tuple with the number of rows and columns of the dataframe. In this case, our dataframe contains after removing the “?” values contains 392 rows or observations and 9 columns.

In [66]:
```
size(auto_data)
```

Out [66]: `(392, 9)`

Basics of Selecting Rows and Columns

The names function returns a vector with the column names of the dataframe.

In [66]:
```
names(auto_data)
```

Out [66]:
```
9-element Vector{String}:
 "mpg"
 "cylinders"
 "displacement"
 "horsepower"
 "weight"
 "acceleration"
```

```
"year"
"origin"
"name"
```

To display the first 3 rows of the dataframe applying indexing rules.

In [67]:
```
auto_data[1:3, :]
```

Out [67]:
3×9 DataFrame

Row	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
	Any	Any	Any	Any	Any	Any	Any	Any	Any
1	18	8	307	130	3504	12	70	1	chevrolet chevelle malibu
2	15	8	350	165	3693	11.5	70	1	buick skylark 320
3	18	8	318	150	3436	11	70	1	plymouth satellite

To filter data one option it is to employ Boolean indenxing. Let´s filter cars by year. For simplicity just the first 5 rows are displayed.

In [68]:
```
auto_data[auto_data[!, “year”] .> 80, :][1:5, :]
```

Out [68]:
5×9 DataFrame

Row	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
	Any	Any	Any	Any	Any	Any	Any	Any	Any
1	27.2	4	135	84	2490	15.7	81	1	plymouth reliant
2	26.6	4	151	84	2635	16.4	81	1	buick skylark
3	25.8	4	156	92	2620	14.4	81	1	dodge aries wagon (sw)
4	23.5	6	173	110	2725	12.6	81	1	chevrolet citation
5	30	4	135	84	2385	12.9	81	1	plymouth reliant

If we need to retrieve several columns type the names of the columns as an array.

In [69]:
```
auto_data[!, [“mpg”, “horsepower”]][1:5, :]
```

Out [69]:
5×2 DataFrame

Row	mpg	horsepower
	Any	Any
1	18	130
2	15	165
3	18	150
4	16	150
5	17	140

nrow() function can help us to generate a range between 1 and the total number of observations.

In [70]:
```
1:nrow(auto_data)
```

Out [70]: `1:392`

Let's suppose we're interested in cars "amc rebel sst" and "ford torino", we can accomplish with the filter() function. Notice we include an anonymous function and the "||" operator which returns true if either of the operands is true.

In [71]:
```
filter(x → x.name == “amc rebel sst” || x.name == “ford torino”, auto_data)
```

Out [71]:
`2×9 DataFrame`

Row	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
	Any	Any	Any	Any	Any	Any	Any	Any	Any
1	16	8	304	150	3433	12	70	1	amc rebel sst
2	17	8	302	140	3449	10.5	70	1	ford torino

To display the fourth and fifth row of the dataframe, we put the number of the rows in square brackets [4, 5] and also we can select the name column onwards with the function Cols().

In [72]:
```
auto_data[[4, 5], Cols(9, :)]
```

Out [72]:
`2×9 DataFrame`

Row	name	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin
	Any	Any	Any	Any	Any	Any	Any	Any	Any
1	amc rebel sst	16	8	304	150	3433	12	70	1
2	ford torino	17	8	302	140	3449	10.5	70	1

To select several colums we just simply apply the indexing rules by putting the number of the columns between square brackets([9, 1, 3, 4]) ans selecting all the rows (!).

In [73]:
```
auto_data[!, [9, 1, 3, 4]][1:3, :]
```

Out [73]:
4×4 DataFrame

Row	name	mpg	displacement	horsepower
	Any	Any	Any	Any
1	chevrolet chevelle malibu	18	307	130
2	buick skylark 320	15	350	165
3	plymouth satellite	18	318	150

Combining the above we can retrieve the 4 and 5 rows, include the name column as in index and also the columns 1, 3 and 4.

In [74]:
```
auto_data[[4, 5], Cols(9, 1, 3, 4)]
```

2×4 DataFrame

Row	name	mpg	displacement	horsepower
	Any	Any	Any	Any
1	amc rebel sst	16	304	150
2	ford torino	17	302	140

If there are non-unique values in the dataframe, they're also retrieved.

In [75]:
```
auto_data[auto_data.name == “ford galaxie 500”, [“name”, “mpg”, “origin”]]
```

Out [75]:
3×3 DataFrame

Row	name	mpg	origin
	Any	Any	Any
1	ford galaxie 500	15	1
2	ford galaxie 500	14	1
3	ford galaxie 500	14	1

More on Selecting Rows and Columns

We can join several conditions, suppose we´re interested in the weight and origins for those cars with the year > 80.

In [76]:
```
auto_data[auto_data.year .> 80, [“name”, “weight”, “origin”]][1:5, :]
```

Out [76]:
5×3 DataFrame

Row	name	weight	origin
	Any	Any	Any
1	plymouth reliant	2490	1
2	buick skylark	2635	1
3	dodge aries wagon (sw)	2620	1
4	chevrolet citation	2725	1
5	plymouth reliant	2385	1

Applying anonymous functions we can obtain cars year > 80 and mpg > 30.

In [77]:
```
filter(x -> x.year > 80 && x.mpg > 30, auto_data)[!, Cols(9, 5, 8)][1:5, :]
```

Out [77]:
5×3 DataFrame

Row	name	weight	origin
	Any	Any	Any
1	toyota starlet	1755	3
2	plymouth champ	1875	1
3	honda civic 1300	1760	3
4	subaru	2065	3
5	datsun 210 mpg	1975	3

We can apply filter combinations, for example, suppose we need to retrieve vehicles Ford and Datsun with displacements < 30, functions any() and occursin can help us.

In [78]:
```
df_filtered = filter(column -> any(occursin.([“ford”, “datsun”], column.name)), auto_data)
df_filt = filter(x -> x.displacement < 300, df_filtered)[1:5, Cols(9, :)]
```

Out [78]:
5×9 DataFrame

Row	name	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin
	Any	Any	Any	Any	Any	Any	Any	Any	Any
1	ford maverick	21	6	200	85	2587	16	70	1
2	datsun pl510	27	4	97	88	2130	14.5	70	3
3	datsun pl510	27	4	97	88	2130	14.5	71	3
4	ford torino 500	19	6	250	88	3302	15.5	71	1
5	ford mustang	18	6	250	88	3139	14.5	71	1

2.3.8 For Loops

Syntactically, Julia loops are very similar to Python loops. Loops are control flow structures. We use them in cases where we need to repeat the execution of certain lines of code several times. Mainly there are two loops, for loop amd while loop. For loop is very useful in Python and Julia because allow us to iterate through iterables and sequences in Python and through collections in Julia. Suppose we must compute the sum of the elements within an array. Notice the abscense of the colon ":" that after the array [], the "end" keyword closing the for loop and also the abscence of indentation. Indentation does not affects the code, because of the "end".

In [79]:
```
total = 0
for value in [3, 2, 19]
total += value
end
println("Total is: $total")
```

Out [79]: 24

Sometimes we need to iterate through several arrays, in such cases nested loops are important.

In [80]:
```
tot = 0
for i in [3, 2, 19]
    for j in [3, 2, 1]
        tot += i*j
    end
end
println("Total is: $tot")
```

Out [80]: 144

We can compute the weigthed average using a for loop.

In [81]:
```
wa = 0
for (value, weight) in collect(zip([2, 3, 19], [0.2, 0.3, 0.5]))
    wa += weight * value
end

println("Weighted average is: $wa")
```

Out [81]: `Weighted average is: 10.8`

String Formatting

Let's build a dataframe with 20 percent missing values. The data comes from a normal distribution(mean = 0 and var = 1).

In [82]:
```
using Distributions, Random
Random.seed!(42)

d = Normal(0, 1)
A = rand(d, (127, 5))
M = wsample([0, NaN], [0.8, 0.2], size(A))
A += M
D = DataFrame(A, ["food", "bar", "pickle", "snack", "popcorn"])
D[1:3, :]
```

Out [82]:
```
3×5 DataFrame
```

Row	food	bar	pickle	snack	popcorn
	Float64	Float64	Float64	Float64	Float64
1	-0.36	-0.15	-0.77	-1.29	-0.27
2	NaN	NaN	NaN	-0.13	-0.26
3	NaN	-0.3	0.06	0.67	0.6

In [83]:
```
using Formatting: printfmt

for col in names(D)
    value = mean(isnan.(D[!, "$col"]))*100
    value = round(value, digits=2)
    println("Column &col has $value% missing values.")
end
```

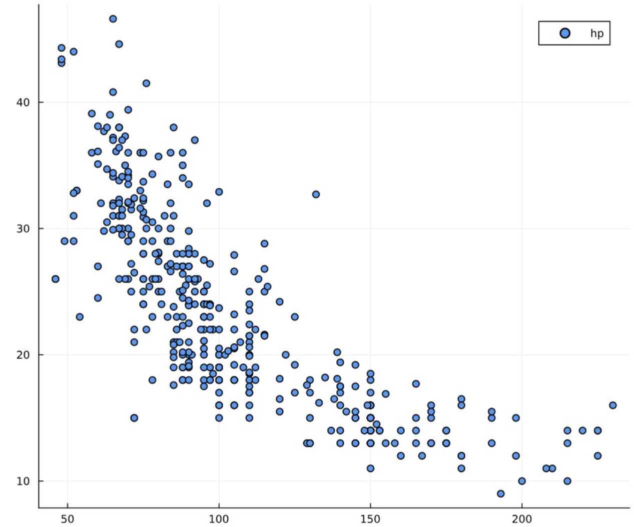
Out [83]:
```
Column food has 21.26% missing values.
Column bar has 21.26% missing values.
Column pickle has 18.9% missing values.
Column snack has 16.54% missing values.
Column popcorn has 21.26% missing values.
```

2.3.9 Additional Graphical and Numerical Summaries

Now, let's plot some of the columns from the dataframe. We've chosen the horsepower and the miles per gallon. As we mentioned the most common ibrary for plotting is Plots.

In [84]:
```
using Plots
scatter(auto_data.:horsepower, auto_data.:mpg, size=(700, 600), label = "hp")
```

Out[84]:



To save the plot as a png file savefig() function is helpful.

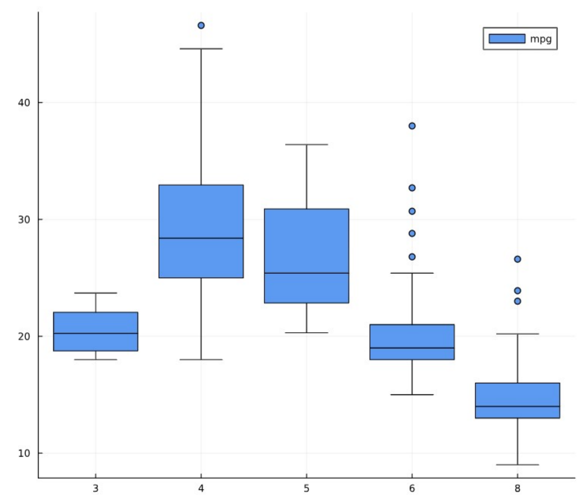
In [85]:
```
savefig("horsepower_mpg.png")
```

StatsPlots package in Julia contains functions to build some graphs. For example, we can plot the miles per gallon by cylinders in our dataset.

In [86]:
```
using StatsPlots
using CategoricalArrays

lev = levels(categorical(auto_data.cylinders))
cat = categorical(auto_data.cylinders)
values = auto_data.mpg
boxplot(cat, values, size=(700, 600), label = "mpg")
```

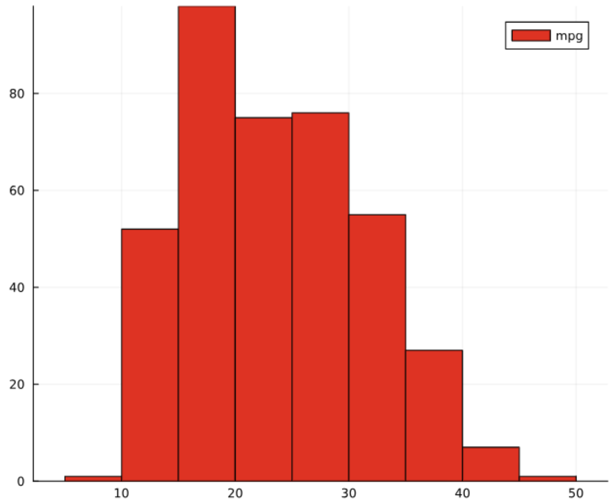
Out [86]:



Often we're interested in the distribution of a variable, histogram() function it is useful.

In [86]:
```
histogram(auto_data.mpg, c=:red, label = "mpg", size = (600, 500))
```

Out [86]:



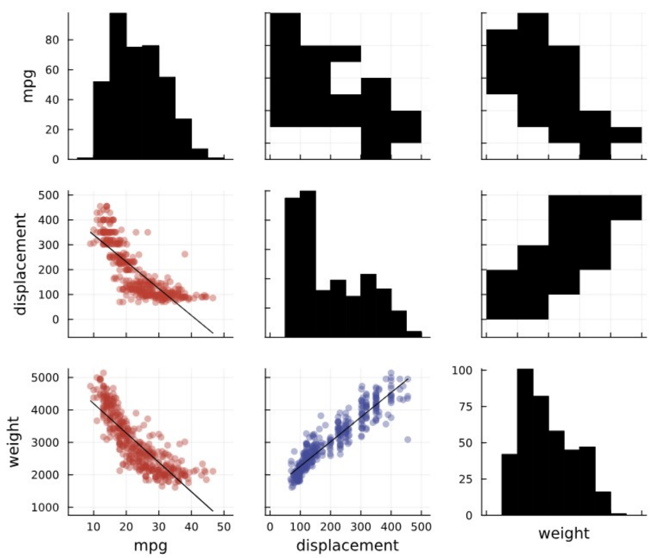
Corrplot() function plots the relationships between variables for the dataframe.

In [86]:
```
@df auto_data=
corrplot([:mpg, :displacement, :horsepower, :weight, :acceleration, :year,
:origin], size = (1300, 900))
```

Or if we just need to plot a subset of variables or columns.

In [87]:
```
@df auto_data corrplot([:mpg, :displacement, :weight], size=(700, 600))
```

Out [87]:



Describe functions returns a statistics summary of the columnes selected.

In [88]:
```
describe(auto_data, :all, cols = [:mpg, :displacement, :weight])
```

Out [88]:

3×9 DataFrame

Row	variable	mean	std	min	q25	median	q75	max	sum
	Symbol	Float64	Float64	Real	Float64	Float64	Float64	Real	Real
1	mpg	23.45	7.81	9	17	22.75	29	46.6	9190.8
2	displacement	194.41	104.64	68	105	151	275.75	455	76209.5
3	weight	2977.58	849.4	1613	2225.25	2803.5	3614.75	5140	1167213