Unit 4: On data files, and basic numerics

Table of contents

1 Unit 4: On data files, and basic numerics

In this unit we deal with data, numerics, and basic numerical algorithms. In terms of numerics, there are other courses at UQ-SMP that focus primarily on numerical mathematics. These include MATH3201 and MATH3204. Here we are only getting a taste of numerical mathematics from the viewpoint of programming. Also, in terms of data, later in this course (Units 6 and 7) we deal with data in a much broader context.

1.1 Data analysis

We first deal with files and file formats include binary, CSV, and JSON. We then discuss basic plotting and animation creation.

1.1.1 Files and file formats

A file is a collection of bytes, stored by your operating system on disk. It can also be transferred via the web.

Here we create a file with 5 ASCII characters `12345`:

```
$ echo -n "12345" > my_file.txt
```

```
$ ls -l my_file.txt
-rw-r--r--@ 1 uqjnazar  staff  5  1 Aug 09:34 my_file.txt
```

Here we download Shakespeare's works from the web:

```
$ curl -o shakespeare.txt https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 5330k  100 5330k    0     0  6043k      0 --:--:-- --:--:-- --:--:-- 6036k

$ ls -l shakespeare.txt
-rw-r--r--@ 1 uqjnazar  staff  5458199  1 Aug 09:45 shakespeare.txt

$ head -4 shakespeare.txt
This is the 100th Etext file presented by Project Gutenberg, and
is presented in cooperation with World Library, Inc., from their
Library of the Future and Shakespeare CDROMS.  Project Gutenberg
often releases Etexts that are NOT placed in the Public Domain!!
```

If a file is encoded as a text file, you can open it in an editor like VS Code and make sense of it. Otherwise it is binary and you cannot. Here we create a binary file of 20 bytes:

```
file = open("my_binary_file.dat","w") #Julia function to open a file. "w" means opening it for
writing.
write(file,Int8.(1:20)) #now f is the handle for the file
close(file)
```

With VSCode, we can use the hexdump plugin to view the file (after you install the plugin - go to `my_binary_file.dat` in the VS code file explorer and choose `Show Hexdump`.

```
 Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10    ................
00000010: 11 12 13 14                                       ....
```

Another alternative is to use the `hexdump` Unix command:

```
$ hexdump my_binary_file.dat
0000000 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10
0000010 11 12 13 14
0000014
```

We won't deal much with binary files, but rather with files that can be interpreted as text. Within text files, one of the most common formats is CSV (Comma Separated Values). Another common format is JSON (Javascript Object Notation). We'll deal with both and in each case we'll use a Julia package that helps us work with such files. These are CSV.jl and JSON.j. We'll also use DataFrames.jl, much more of this package will be discussed in Units 6 and 7.

1.1.2 CSV Files

```
using DataFrames, CSV
df = DataFrame(
   names = ["Jason", "Ella", "Kayley", "Emily"],
   ages = [23, 25, 13, 15],
   gender = [:male, missing, :female, :female]
)
println(df)
```

```
4×3 DataFrame
 Row   names   ages   gender
       String  Int64  Symbol?

   1   Jason      23  male
   2   Ella       25  missing
   3   Kayley     13  female
```

```
   4   Emily      15  female
```

```
CSV.write("people.csv", df)
```

```
"people.csv"
```

In the REPL, if we enter the shell:

```
;
shell> ls -l people.csv
-rw-r--r--  1 uqjnazar  staff  74  1 Aug 10:35 people.csv

shell> cat people.csv
names,ages,gender
Jason,23,male
Ella,25,
Kayley,13,female
Emily,15,female
names,ages,gender
Jason,23,male
Ella,25,
Kayley,13,female
Emily,15,female
```

This form of the CSV file has a header row. There is also a form without

```
CSV.write("people_no_header.csv", df, header = false)
sleep(3) # A crude way to let the OS settle with creating the file (otherwise Julia will beat it)
readlines("people_no_header.csv") #A crude way to read lines of a file into a vector of strings
```

```
4-element Vector{String}:
 "Jason,23,male"
 "Ella,25,"
 "Kayley,13,female"
 "Emily,15,female"
```

Now that the have the files `people.csv` and `people_no_header.csv` we can read them and process them.

```
using DataFrames, CSV
df_with_names = CSV.File("people.csv") |> DataFrame #In Julia f(x) |> g is like g(f(x))
println(df)
```

```
4×3 DataFrame
 Row   names   ages   gender
       String  Int64  Symbol?

   1   Jason      23  male
   2   Ella       25  missing
   3   Kayley     13  female
   4   Emily      15  female
```

```
df_no_names = CSV.File("people_no_header.csv",header = false ) |> DataFrame
println(df_no_names)
```

```
4×3 DataFrame
 Row   Column1  Column2  Column3
       String7  Int64    String7?

   1   Jason         23  male
```

```
   2   Ella          25  missing
   3   Kayley        13  female
   4   Emily         15  female
```

```
n, p = size(df_with_names)
@show n, p
average_age = sum(df_with_names.ages)/n  #notice we can access the field ages via ".ages"
@show average_age;
```

```
(n, p) = (4, 3)
average_age = 19.0
```

```
n, p = size(df_no_names)
@show n, p
average_age = sum(df_no_names[:,2])/n  #Here we just access ages as the second column
@show average_age;
```

```
(n, p) = (4, 3)
average_age = 19.0
```

1.1.3 JSON Files

JSON files are much more structured than CSV files and are used for a variety of settings. As a real world application consider the Github Public API, this provides programmatic web access to github services so that source code and artefacts can be managed via software.

We download the JSON file from the web for the repos of a particular github user and the use `JSON.parse` from the `JSON.jl` package to convert the JSON file into a hierarchical dictionary.

```
using HTTP, JSON
web_request = HTTP.request("GET","https://api.github.com/users/paulbellette/repos")
web_data = JSON.parse(String(web_request.body))
```

```
12-element Vector{Any}:
 JSON.Object{String, Any}("id" => 1224378452, "node_id" => "R_kgDOSPqIVA", "name" => "chladni_inverse_d
 JSON.Object{String, Any}("id" => 1255492163, "node_id" => "R_kgDOStVKQw", "name" => "cycle_moment_sinkl
 JSON.Object{String, Any}("id" => 35654242, "node_id" => "MDEwOlJlcG9zaXRvcnkzNTY1NDI0Mg==", "name" =>
 JSON.Object{String, Any}("id" => 1266940118, "node_id" => "R_kgDOS4P41g", "name" => "graph_block_LM",
 JSON.Object{String, Any}("id" => 1239189550, "node_id" => "R_kgDOSdyILg", "name" => "krylov_style_atte
 JSON.Object{String, Any}("id" => 73240772, "node_id" => "MDEwOlJlcG9zaXRvcnk3MzI0MDc3Mg==", "name" =>
 JSON.Object{String, Any}("id" => 1212270403, "node_id" => "R_kgDOSEHHQw", "name" => "neural_hybrid_ren
 JSON.Object{String, Any}("id" => 137652204, "node_id" => "MDEwOlJlcG9zaXRvcnkxMzc2NTIyMDQ=", "name" =>
 JSON.Object{String, Any}("id" => 394584413, "node_id" => "MDEwOlJlcG9zaXRvcnkzOTQ1ODQ0MTM=", "name" =>
 JSON.Object{String, Any}("id" => 79876825, "node_id" => "MDEwOlJlcG9zaXRvcnk3OTg3NjgyNQ==", "name" =>
 JSON.Object{String, Any}("id" => 673271098, "node_id" => "R_kgDOKCFNOg", "name" => "test_repo", "full_
 JSON.Object{String, Any}("id" => 1210126476, "node_id" => "R_kgDOSCEQjA", "name" => "TinyGPT", "full_na
```

```
length(web_data)
```

```
12
```

```
[(r["name"], r["language"]) for r in web_data]
```

```
12-element Vector{Tuple{String, Any}}:
 ("chladni_inverse_design", "Python")
 ("cycle_moment_sinkhorn_flows", "Python")
 ("displaz", "C++")
 ("graph_block_LM", "Python")
 ("krylov_style_attention", "Python")
```

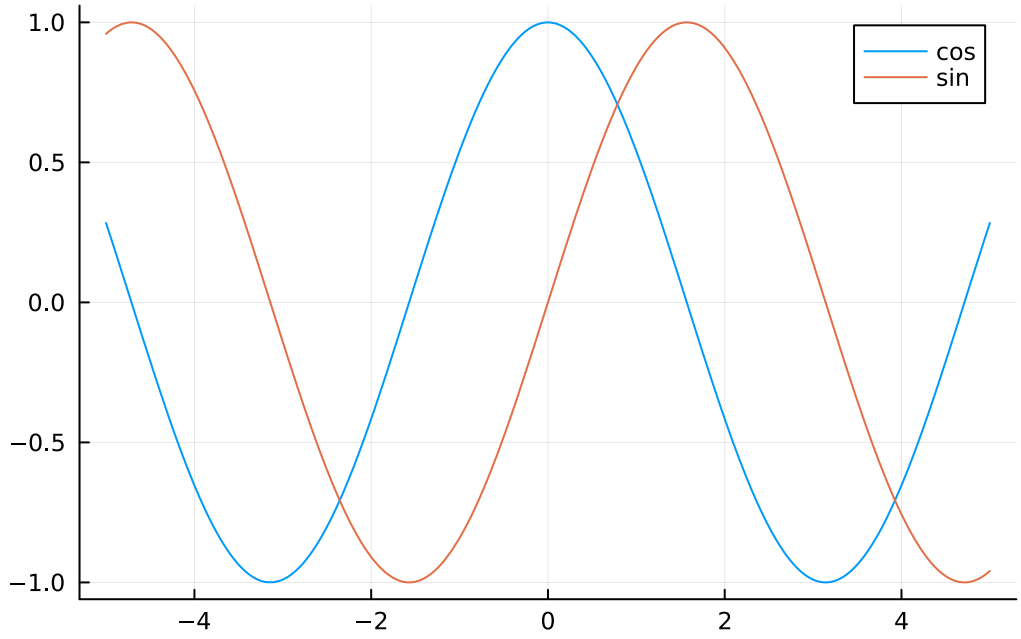
```
("Munkres.jl", "Julia")
("neural_hybrid_render", "Python")
("PaulBellette.github.io", "HTML")
("Pauls_Silly_Test_Repo", "Julia")
("SpatialGrids.jl", "Julia")
("test_repo", nothing)
("TinyGPT", "Julia")
```

1.1.4 Basic scientific work environment

You can use Julia for plotting and statistics. We won't cover all the features of Plots.jl, but instead will show a few examples taken from SWJ which showcase plotting and animation. You may also use the image gallery, where in each case clicking on the example image gives you a link to self contained source code that plots the image. Note that in that source you can ignore (comment out) `pyplot()` which is a function which sets the backend of the plotting as Python's plotting package.

Simple direct plotting of functions:

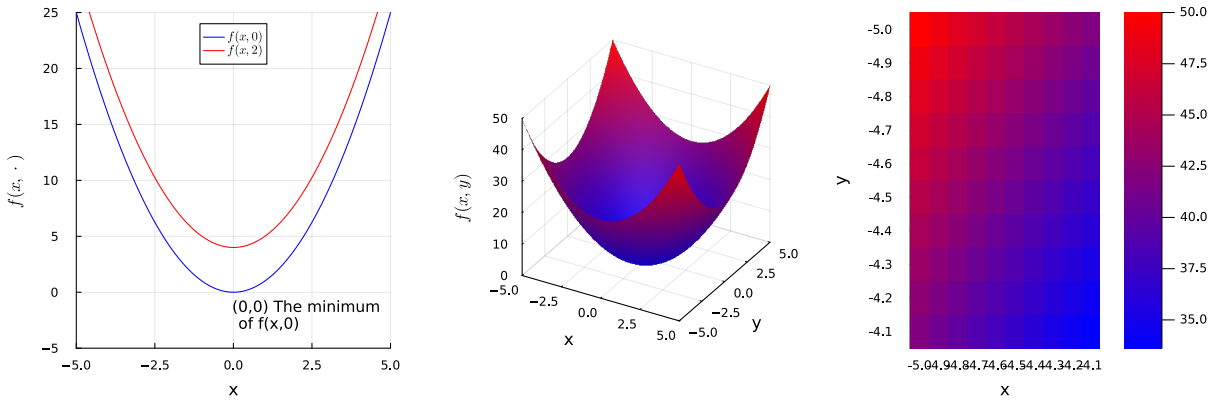
```
using Plots
plot([cos sin], label = ["cos" "sin"])
```



Multiple plots:

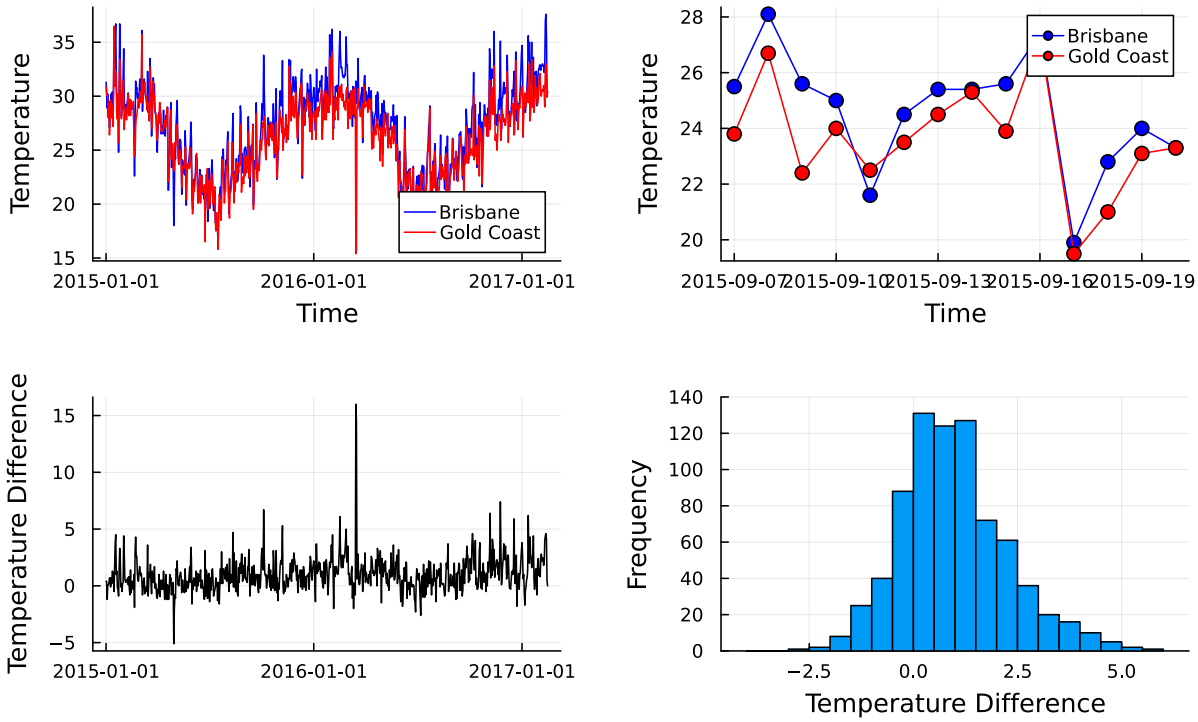
```
using Plots, LaTeXStrings, Measures

f(x,y) = x^2 + y^2
f0(x) = f(x,0)
f2(x) = f(x,2)

xVals, yVals = -5:0.1:5 , -5:0.1:5
plot(xVals, [f0.(xVals), f2.(xVals)],
    c=[:blue :red], xlims=(-5,5), legend=:top,
    ylims=(-5,25), ylabel=L"f(x,\cdot)", label=[L"f(x,0)" L"f(x,2)"])
p1 = annotate!(0, -0.2, text("(0,0) The minimum\n of f(x,0)", :left, :top, 10))
```

```
z = [ f(x,y) for y in yVals, x in xVals ]
p2 = surface(xVals, yVals, z, c=cgrad([:blue, :red]),legend=:none,
    ylabel="y", zlabel=L"f(x,y)")

M = z[1:10,1:10]
p3 = heatmap(M, c=cgrad([:blue, :red]), yflip=true, ylabel="y",
    xticks=([1:10;], xVals), yticks=([1:10;], yVals))

plot(p1, p2, p3, layout=(1,3), size=(1200,400), xlabel="x", margin=5mm)
```



Plotting data:

```
using DataFrames, CSV, Statistics, Dates, Plots, Measures

resp =
HTTP.request("GET","https://raw.githubusercontent.com/h-Klok/StatsWithJuliaBook/master/data/temperatures.csv")
data = CSV.read(IOBuffer(String(resp.body)), DataFrame)

brisbane = data.Brisbane
goldcoast = data.GoldCoast

diff = brisbane - goldcoast
dates = [Date(
   Year(data.Year[i]),
   Month(data.Month[i]),
   Day(data.Day[i])
   ) for i in 1:nrow(data)]

fortnightRange = 250:263
brisFortnight = brisbane[fortnightRange]
goldFortnight = goldcoast[fortnightRange]

p1 = plot(dates, [brisbane goldcoast], label=["Brisbane" "Gold Coast"],
   c=[:blue :red], xlabel="Time", ylabel="Temperature")
p2 = plot(dates[fortnightRange], [brisFortnight goldFortnight], label=["Brisbane" "Gold Coast"],
   c=[:blue :red], m=(:dot, 5, Plots.stroke(1)), xlabel="Time", ylabel="Temperature")
p3 = plot(dates, diff, c=:black, ylabel="Temperature Difference",legend=false)
p4 = histogram(diff, bins=-4:0.5:6,
   ylims=(0,140), legend = false,
   xlabel="Temperature Difference", ylabel="Frequency")
plot(p1,p2,p3,p4, size = (800,500), margin = 5mm)
```



Creating animations:

```
using Plots

function graphCreator(n::Int)
   vertices = 1:n
   complexPts = [exp(2*pi*im*k/n) for k in vertices]
   coords = [(real(p),imag(p)) for p in complexPts]
   xPts = first.(coords)
   yPts = last.(coords)
   edges = []
   for v in vertices, u in (v+1):n
      push!(edges,(v,u))
   end

   anim = Animation()
   scatter(xPts, yPts, c=:blue, msw=0, ratio=1,
   xlims=(-1.5,1.5), ylims=(-1.5,1.5), legend=:none)

   for i in 1:length(edges)
      u, v = edges[i][1], edges[i][2]
      xpoints = [xPts[u], xPts[v]]
      ypoints = [yPts[u], yPts[v]]
      plot!(xpoints, ypoints, line=(:red))
      frame(anim)
   end

   gif(anim, "graph.gif", fps = 60)
end

graphCreator(16)
```

```
[ Info: Saved animation to /home/paul-bellete/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLea
```

```
Plots.AnimatedGif("/home/paul-bellete/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSys
```

1.1.5 Basic statistical summaries

In terms of basic statistics associated with data you can use the in-built package `Statistics` as well as the external package `StatsBase.jl`:

```
using CSV, Statistics, StatsBase
resp =
HTTP.request("GET","https://raw.githubusercontent.com/h-Klok/StatsWithJuliaBook/master/data/temperatures.csv")
data = CSV.read(IOBuffer(String(resp.body)), DataFrame)[:,4] # Take the fourth column

println("Sample Mean: ", mean(data))
println("Harmonic <= Geometric <= Arithmetic ",
    (harmmean(data), geomean(data), mean(data)))
println("Sample Variance: ",var(data))
println("Sample Standard Deviation: ",std(data))
println("Minimum: ", minimum(data))
println("Maximum: ", maximum(data))
println("Median: ", median(data))
println("95th percentile: ", percentile(data, 95))
println("0.95 quantile: ", quantile(data, 0.95))
println("Interquartile range: ", iqr(data),"\n")

summarystats(data)
```

```
Sample Mean: 27.155405405405407
Harmonic <= Geometric <= Arithmetic (26.52181504074163, 26.84561900212437, 27.155405405405407)
Sample Variance: 16.125389558372802
Sample Standard Deviation: 4.015643106449177
Minimum: 16.1
Maximum: 37.6
Median: 27.7
95th percentile: 33.0
0.95 quantile: 33.0
Interquartile range: 6.100000000000001
```

```
Summary Stats:
Length:         777
Missing Count:  0
Mean:           27.155405
Std. Deviation: 4.015643
Minimum:        16.100000
1st Quartile:   24.000000
Median:         27.700000
3rd Quartile:   30.100000
Maximum:        37.600000
```

1.2 Numerics

We now focus on floating point numbers. This is the Double-precision floating-point format. Before we get into the details, here are some illustrations on how it is represented in memory:

```
function pretty_print_float(x::Float64)
  bits = bitstring(x)
  println("Sign: ", bits[1])
```

```
  println("Exponent: ", bits[2:12])
  println("Significand: ", bits[13:64])
end

x = 15.324
@show typeof(x)
@show sizeof(x)
@show bitstring(x);
pretty_print_float(x)

ex = exponent(x)
sig = significand(x)
@show ex, sig
@show 2^ex * sig;
```

```
typeof(x) = Float64
sizeof(x) = 8
bitstring(x) = "0100000000101110101001011110001101010011111101111100111011011001"
Sign: 0
Exponent: 10000000010
Significand: 1110101001011110001101010011111101111100111011011001
(ex, sig) = (3, 1.9155)
2 ^ ex * sig = 15.324
```

And this is the Single-precision floating-point format:

```
function pretty_print_float(x::Float32) #Notice this is a second method of pretty_print_float
  bits = bitstring(x)
  println("Sign: ", bits[1])
  println("Exponent: ", bits[2:9])
  println("Significand: ", bits[10:32])
end

x = 15.324f0
@show typeof(x)
@show sizeof(x)
@show bitstring(x);
pretty_print_float(x)

ex = exponent(x)
sig = significand(x)
@show ex, sig
@show 2^ex * sig;
```

```
typeof(x) = Float32
sizeof(x) = 4
bitstring(x) = "01000001011101010010111100011011"
Sign: 0
Exponent: 10000010
Significand: 11101010010111100011011
(ex, sig) = (3, 1.9155f0)
2 ^ ex * sig = 15.324f0
```

We would ideally like to represent any number on the real line $\mathbb{R}$ via a finite number of bits with the computer. However, this is not possible and any numerical representation of a number $x \in \mathbb{R}$ is only approximated via a number $\tilde{x} \in \mathbb{F}$ where $\mathbb{F}$ is the set of **floating point numbers**. Each such floating point number is represented as,

$$\tilde{x} = \pm(1+f)2^e,$$

where e is a (signed) integer called the **exponent** and $1+f$ is the **mantissa** (or **significand**). The value f is represented as,

$$f = \sum_{i=1}^{d} b_i 2^{-i},$$

where $b_i \in \{0,1\}$ and d is a fixed positive integer counting the number of bits used for the mantissa.

Hence the mantissa, $1+f$, lies in the range $[1,2)$ and is represented in binary form. By multiplying the equation above by 2^{-d} we have,

$$f = 2^{-d}\Big(\sum_{i=1}^{d} b_i 2^{d-i}\Big) = 2^{-d} z.$$

Hence $z \in \{0,1,2,\dots,2^d-1\}$. This means that between 2^e and ending just before 2^e-1 there are exactly 2^d evenly spaced numbers in the set $\mathbb{F}$.

Exercise: Take $d=3$. What are the floating point numbers with $e=0$? How about with $e=1$? How about with $e=-1$?

Exercise: Take $d=2$. Determine $\mathbb{F} \cap [-5,5]$.

Observe now that the smallest element of $\mathbb{F}$ that is greater than 1 is $1+2^{-d}$. This motivates defining machine epsilon as $\varepsilon_{\text{mach}} = 2^{-d}$.

The IEEE 754 double precision standard has $d=52$ bits and single precision (`Float32`) has $d=23$ bits. Hence with `Float64` variables we have

$$\varepsilon_{\text{mach}} = 2^{-52} \approx 2.2 \times 10^{-16}.$$

```
@show eps() #Default is for Float64
@show eps(Float32)
@show eps(Float16)
```

```
eps() = 2.220446049250313e-16
eps(Float32) = 1.1920929f-7
eps(Float16) = Float16(0.000977)
```

```
Float16(0.000977)
```

```
@show 2^(-52)
@show 2^(-23)
@show 2^(-10);
```

```
2 ^ -52 = 2.220446049250313e-16
2 ^ -23 = 1.1920928955078125e-7
2 ^ -10 = 0.0009765625
```

We can suppose there is some (mathematical) function $\text{fl} : \mathbb{F} \to \mathbb{R}$ where $\text{fl}(x)$ takes a real number x and maps it to the nearest $\tilde{x}$ in $\mathbb{F}$. For positive x it lies in the interval $[2^e, 2^{e+1})$ where the spacing between the elements is 2^{e-d}. Hence $|\tilde{x}-x| \le \frac{1}{2}2^{e-d}$. We can now consider the relative error between $\tilde{x}$ and x:

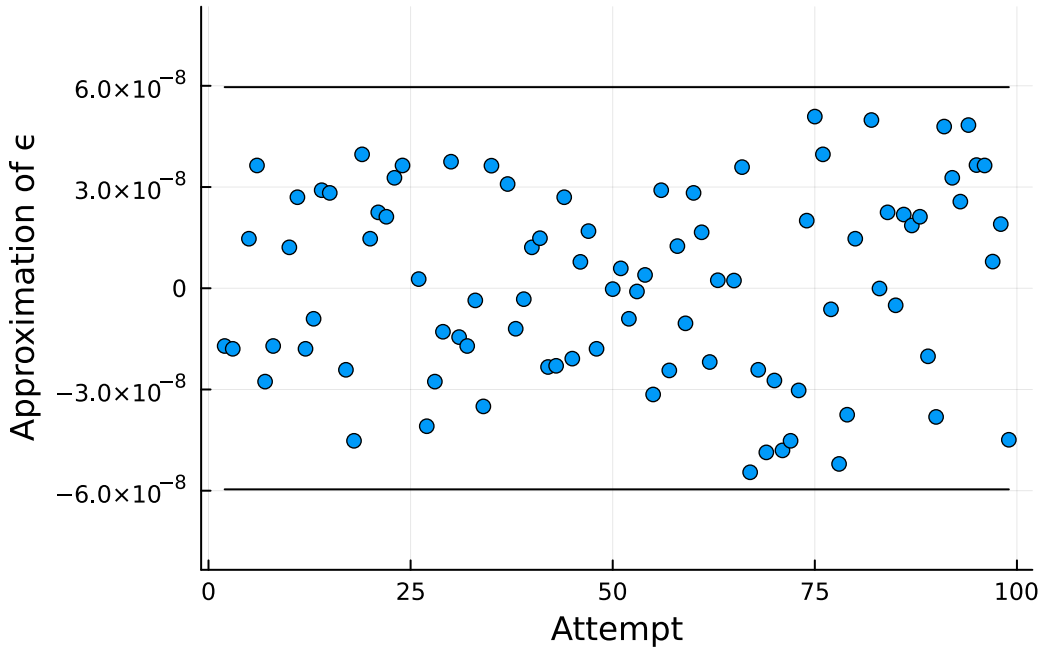
$$\frac{|\tilde{x}-x|}{|x|} \le \frac{2^{e-d-1}}{2^e} \le \frac{1}{2}\varepsilon_{\text{mach}}.$$

An equivalent statement states that for any $x \in \mathbb{R}$ (within the range of the exponent), there is a ε where $|\varepsilon| \le \frac{1}{2}\varepsilon_{\text{mach}}$ and,

$$\text{fl}(x) = x(1+\varepsilon).$$

Here is an example that looks at the irrational square roots of $\{1,2,3,\dots,100\}$ and estimates the ε associated with $\text{fl}(x)$ for each of these square roots. The example does this for `Float32` values and uses `Float64` as an approximation of the absolute truth. The two black bars are at $\pm\frac{1}{2}\varepsilon_{\text{mach}}$.

```
non_squares = setdiff(1:100,[i^2 for i in 1:100])
x̃ = sqrt.(Float32.(non_squares)) #x + \tilde + [TAB]
x = sqrt.(non_squares) #Lets treat 64 bit as infinite precision
  = x̃ ./ x .- 1  #\epsilon + [TAB]
scatter(non_squares, ,legend=false,xlabel = "Attempt", ylabel="Approximation of  ")
plot!([(x)->0.5*eps(Float32) (x)->-0.5eps(Float32)],
   c=:black,ylim=(-0.7*eps(Float32), 0.7*eps(Float32)))
```



Going back to `Float64` (double precision) we have 52 bits in the mantissa. 11 bits for the exponent e and a single sign bit. This makes 64 bits. There are also some special values:

```
bitstring(0/0) #NaN
```

```
"1111111111110000000000000000000000000000000000000000000000000000"
```

```
bitstring(1/0) #Inf
```

```
"0111111111110000000000000000000000000000000000000000000000000000"
```

```
bitstring(-1/0) #-Inf
```

```
"1111111111110000000000000000000000000000000000000000000000000000"
```

1.2.1 Numerical inaccuracy issues

Due to the approximate nature of the floating point numbers there are a range of numerical issues that can arise. The most simple being that some number are not exactly represented, e.g.

```
@show big(0.1)
```

```
big(0.1) = 0.1000000000000000055511151231257827021181583404541015625
```

```
0.1000000000000000055511151231257827021181583404541015625
```

This approximate nature also leads to associativity of addition of floating point numbers not being guaranteed, e.g.

```
@show (0.1 + 0.2) + 0.3 == 0.1 + (0.2 + 0.3)
```

```
(0.1 + 0.2) + 0.3 == 0.1 + (0.2 + 0.3) = false
```

```
false
```

but this is not usually a big error

```
@show (0.1 + 0.2) + 0.3  0.1 + (0.2 + 0.3)
@show (0.1 + 0.2) + 0.3 - (0.1 + (0.2 + 0.3))
```

```
(0.1 + 0.2) + 0.3  0.1 + (0.2 + 0.3) = true
((0.1 + 0.2) + 0.3) - (0.1 + (0.2 + 0.3)) = 1.1102230246251565e-16
```

```
1.1102230246251565e-16
```

Also small numbers can get "absorbed" into a larger number

```
@show (1e9 + 1e-9) == 1e9
```

```
1.0e9 + 1.0e-9 == 1.0e9 = true
```

```
true
```

Obviously repeated addition can compound errors, for example

```
a = 0.1
x = 0.0
n = 1e6

for i = 1 : n
   global x += a
end

error =  a*n - x
@show error
```

```
error = -1.3328826753422618e-6
```

```
-1.3328826753422618e-6
```

There are many examples of such errors and when used in complex operations these errors can lead to the violation of expected mathematical results, e.g.

```
using LinearAlgebra

x = [ones(10,6) zeros(10)]

eigen_values, eigen_vectors = eigen(x'*x)

fraction_bad =  sum(eigen_values .< 0)/length(eigen_values)

@show fraction_bad
```

```
fraction_bad = 0.2857142857142857
```

```
0.2857142857142857
```

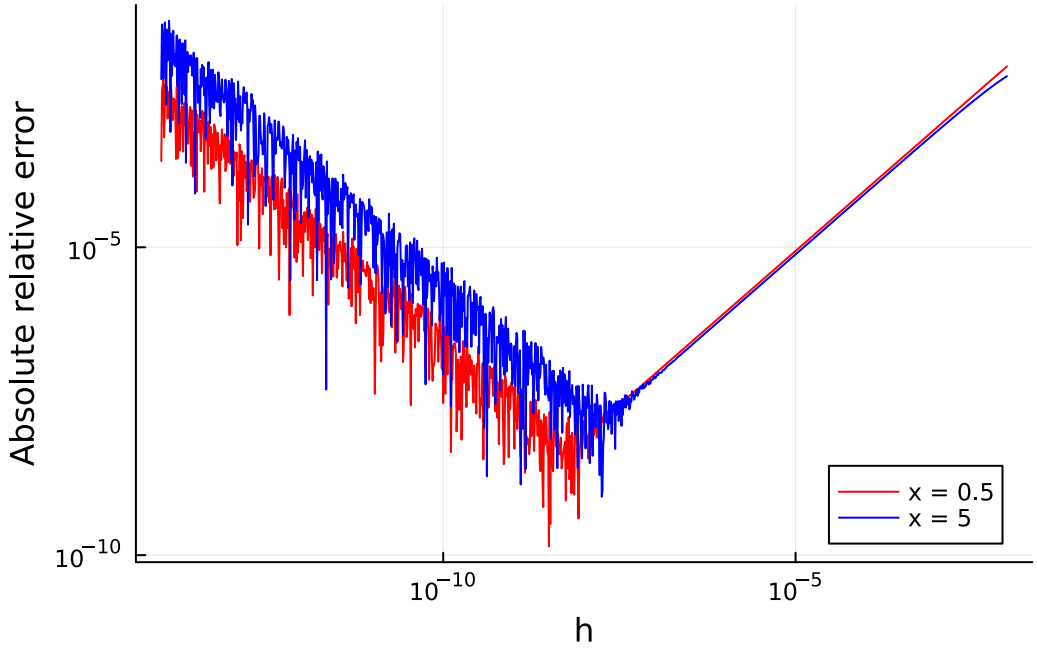
1.2.2 Numerical derivatives

These errors are particularly bad when considering a finite difference approximation to a derivative

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Mathematically we would expect that the error in the mathematical approximation should go to zero as h approaches zero. However floating point errors increase (difference of close numbers divided by a small number) so there is an optimal value for the finite difference step size that reduces the total approximation error.

```
diff_forward(f, x; h = sqrt(eps())) = (f(x+h) - f(x))/h
f(x) = sin(x^2)
f_der(x) = 2x*cos(x^2) #Here we know the analytic derivative
x0, x1 = 0.5, 5
h_range = 10 .^ (-14:0.01:-2)
errs0 = [abs(diff_forward(f, x0; h = h) - f_der(x0))/abs(f_der(x0)) for h in h_range]
errs1 = [abs(diff_forward(f, x1; h = h) - f_der(x1))/abs(f_der(x1)) for h in h_range]
plot(h_range,[errs0 errs1], yaxis = :log,xaxis = :log,
   xlabel="h",ylabel="Absolute relative error",
   label = ["x = $x0" "x = $x1"], c = [:red :blue], legend = :bottomright)
```



1.2.3 Brief intro to automatic differentiation

Doing finite differences suffers from numerical accuracy issues. Doing finite differences for functions of the type $y = f(x)$ where $y \in \mathbb{R}$ and $x \in \mathbb{R}^{\tilde{n}}$ where $ n > 1$ also suffers from a computational complexity issue as $O(n)$ function evaluations are also required. A modern approach to this is called automatic differentiation, where floating point precision accuracy for the derivative is achieved and $O(1)$ function evaluations are required. This works by the cunning application of the chain rule and a special numerical type called a Dual Number. We won't go into details here but this is supported by various Julia packages.

```
using Zygote

@show gradient((x,y) -> 3x + 4y + 1, 0,0)
```

```
gradient(((x, y)->begin
            #= /home/paul-bellete/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/
```

```
            3x + 4y + 1
        end), 0, 0) = (3.0, 4.0)
```

```
(3.0, 4.0)
```

1.2.4 Numerical integration

Numerical integration finds many useful applications in Applied Mathematics, Statistics, Science and Engineering. The standard problem is to find a numerical estimate of the definite integral of some function. Numerical methods for the approximate integration of univariate functions are usually referred to as quadrature methods and higher dimensional problems are referred to as cubature methods. As a starting point consider the 1D integral of the function $f(x)$ from $x = a$ to $x = b$

$$\int_a^b f(x)dx$$

A simple method for estimating the integral is to use the Reimann sum at some finite width to approximate the integral, i.e.

$$\int_a^b f(x)dx \approx (b-a)f\left(\frac{a+b}{2}\right)$$

This can be improved by using interpolating polynomials of successively higher orders to more closely follow the function (called Newton-Cotes formulas). Fox example, for a 1st order polynomial we get the trapezoid rule

$$\int_a^b f(x)dx \approx (b-a)\left(\frac{f(a)+f(b)}{2}\right)$$

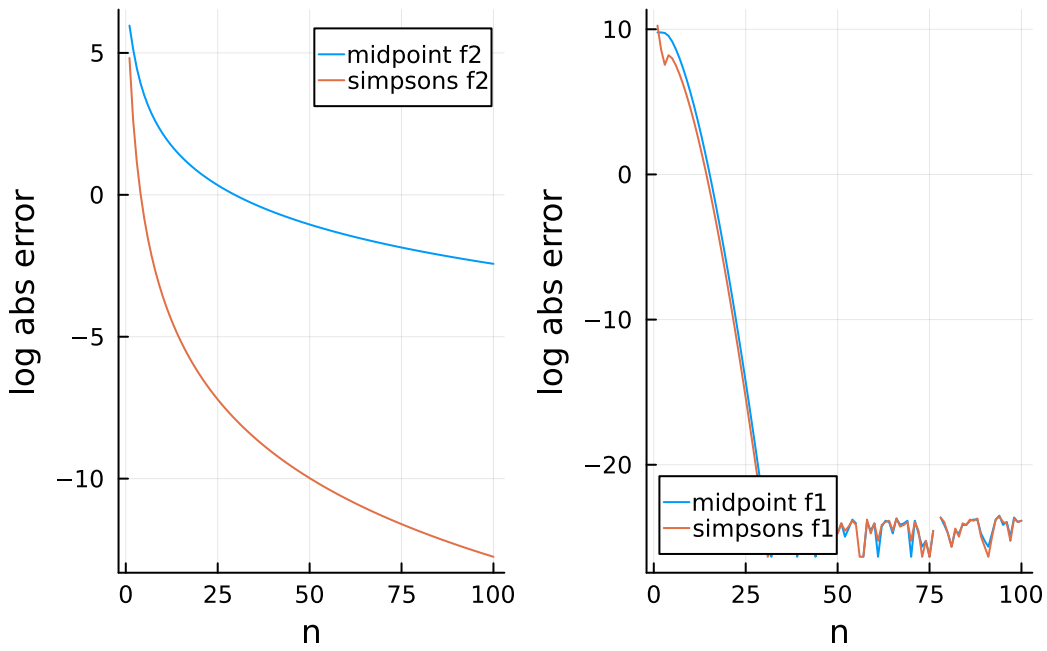
and for a second order polynomial we get Simpson's Rule

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

and so on and so on. In general the error of these methods reduces as the order of the polynomial increases. However an important special case is when the function is smooth is periodic, here midpoint and trapezoid rules have excellent convergence properties (beating the higher order methods), see below

```
using SpecialFunctions

function riemann_sum(f, a, b, method = "midpoint")
   if method == "midpoint"
      return (b - a)*f((a + b)/2)
   elseif method == "simpsons"
      return (b-a) / 6 * (f(a) + 4*f((a+b)/2) + f(b))
   else
      @error("unsupported method given")
   end
end

f1(x) = exp(10*cos(x))
f2(x) = exp(x)

#from 0 to 2
integral_f1 = 2 *besseli(0,10)
```

```
integral_f2 = exp(2 ) - 1

#use a map to implement midpoint method on successively finer grid

f1_int_midpoint = Float64[]
f1_int_simpsons = Float64[]

f2_int_midpoint = Float64[]
f2_int_simpsons = Float64[]

n_range = 1 : 100

for n in n_range
   dx = 2 /n
   xs = 0 .+ dx * (0 : n-1 )
   f1_int_mid = sum(map(x -> riemann_sum(f1, x, x+dx), xs))
   f1_int_simp = sum(map(x -> riemann_sum(f1, x, x+dx, "simpsons"), xs))
   push!(f1_int_midpoint, f1_int_mid)
   push!(f1_int_simpsons, f1_int_simp)
   f2_int_mid = sum(map(x -> riemann_sum(f2, x, x+dx), xs))
   f2_int_simp = sum(map(x -> riemann_sum(f2, x, x+dx, "simpsons"), xs))
   push!(f2_int_midpoint, f2_int_mid)
   push!(f2_int_simpsons, f2_int_simp)
end

p1 = plot(log.(abs.(f2_int_midpoint .- integral_f2)), xlabel = "n", ylabel = "log abs error", label
= "midpoint f2")
plot!(p1, log.(abs.(f2_int_simpsons .- integral_f2)), xlabel = "n", ylabel = "log abs error", label
= "simpsons f2")

p2 = plot(log.(abs.(f1_int_midpoint .- integral_f1)), xlabel = "n", ylabel = "log abs error", label
= "midpoint f1")
plot!(p2, log.(abs.(f1_int_simpsons .- integral_f1)), xlabel = "n", ylabel = "log abs error", label
= "simpsons f1")

plot(p1,p2)
```



The wild success of the simple integration methods for smooth periodic function methods explains why the Fast Fourier Transform sees such widespread use in Science and Engineering (for example in the JPEG compression algorithm, well its close cousin the DCT).

Higher dimensional integration methods (cubature) methods follow a similar approach but due to the poor scaling of computational complexity with dimension tend to feature advance adaptive methods to resize the grid the integral is performed on. See for example the rate of error increase with dimension in the following integral of a multivariate normal distribution using a state of the art integration method:

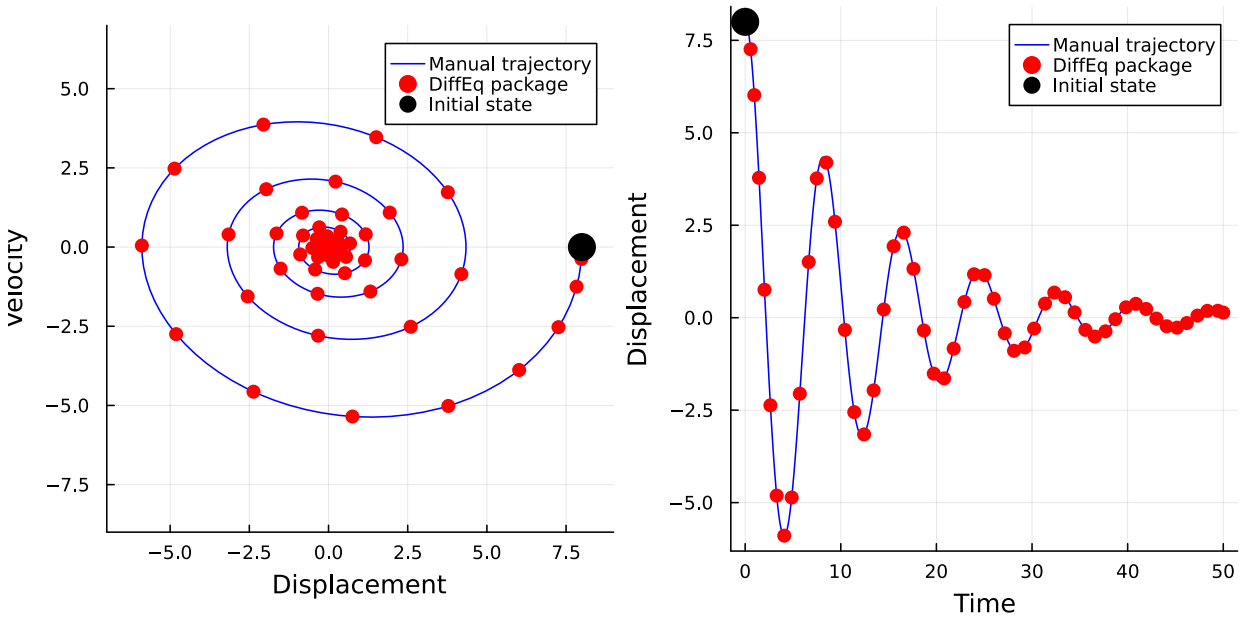
```
using HCubature

M = 4.5
maxD = 10

f(x) = (2*pi)^(-length(x)/2) * exp(-(1/2)*x'x)

for n in 1:maxD
   a = -M*ones(n)
   b = M*ones(n)
   I,e = hcubature(f, a, b, maxevals = 10^7)
   println("n = $(n), integral = $(I), error (estimate) = $(e)")
end
```

```
n = 1, integral = 0.9999932046537504, error (estimate) = 4.3658478221519914e-10
n = 2, integral = 0.9999864091389511, error (estimate) = 1.4879076431447001e-8
n = 3, integral = 0.9999796140804303, error (estimate) = 1.4899542972263743e-8
n = 4, integral = 0.9999728074508335, error (estimate) = 4.444736568416228e-7
n = 5, integral = 0.9999659361030421, error (estimate) = 2.3294669134902988e-5
n = 6, integral = 0.9999639124757738, error (estimate) = 0.00039379544626102076
n = 7, integral = 1.0001623151630636, error (estimate) = 0.003150665016337946
n = 8, integral = 1.0074827348433604, error (estimate) = 0.0232757416645977
n = 9, integral = 1.2233043761463311, error (estimate) = 0.373112534918662
n = 10, integral = 0.4286620931616112, error (estimate) = 0.22089760603668315
```

1.2.5 Differential Equations

Differential equations are the backbone of physical modelling in Science and Engineering and hence a vast amount of literature to the numerical solution of differential equations has been produced. Partial Differential Equations require some special care for numerical analysis and won't be covered here. Ordinary Differential Equations are much more approachable (particularly initial value problems). In fact Julia features a state of the art library for the solution of Ordinary (and Stochastic) Differential Equations in DifferentialEquations.jl. A simple example is given below for a "spring-mass_damper" second order ODE.

```
using DifferentialEquations, LinearAlgebra, Plots

k, b, M = 1.2, 0.3, 2.0
A = [0 1;
   -k/M -b/M]

initX = [8., 0.0]
tEnd = 50.0
tRange = 0:0.1:tEnd

manualSol = [exp(A*t)*initX for t in tRange]

linearRHS(x,Amat,t) = Amat*x
prob = ODEProblem(linearRHS, initX, (0,tEnd), A)
sol = solve(prob)

p1 = plot(first.(manualSol), last.(manualSol),
    c=:blue, label="Manual trajectory")
p1 = scatter!(first.(sol.u), last.(sol.u),
    c=:red, ms = 5, msw=0, label="DiffEq package")
p1 = scatter!([initX[1]], [initX[2]],
    c=:black, ms=10, label="Initial state", xlims=(-7,9), ylims=(-9,7),
    ratio=:equal, xlabel="Displacement", ylabel="Velocity")
p2 = plot(tRange, first.(manualSol),
    c=:blue, label="Manual trajectory")
p2 = scatter!(sol.t, first.(sol.u),
    c=:red, ms = 5, msw=0, label="DiffEq package")
p2 = scatter!([0], [initX[1]],
    c=:black, ms=10, label="Initial state", xlabel="Time",
    ylabel="Displacement")
plot(p1, p2, size=(800,400), legend=:topright)
```



There is a great similarity in the basic method for the numerical solution of ODES and the numerical integration methods outlined above. For example consider the following initial value ODE problem

$$\frac{dy}{dt} = f(y,t), \qquad \text{and} \qquad y(0) = y_0.$$

where $f : \mathbb{R}^n \to \mathbb{R}^n$ and we seek solutions on some time horizon from $t = 0$ to $t = a$.

The most basic approximation method to the derivative is using the approximation

$$\frac{dy}{dt} \approx \frac{y(t+h) - y(t)}{h}$$

which when applied recursively yields a basic numerical method called Euler's Method.

$$y(t+h) = y(t) + hf(y(t)),$$

A popular Higher Order Method are the Runge-Kutta Methods. These use intermediate points to approximate the derivative that have coefficients chosen to reduce the approximation error. The 4th order Runge-Kutta mthod is of the form

$$y(t+h) = y(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

where,

$$\begin{aligned} k_1 &= f\left(y(t)\right) \\ k_2 &= f\left(y(t) + h\,\frac{k_1}{2}\right), \\ k_3 &= f\left(y(t) + h\,\frac{k_2}{2}\right) \\ k_4 &= f\left(y(t) + h\,k_3\right). \end{aligned}$$

Note that if we consider the Fundamental Theorem of Calculus we can recognise that Eulers method and the RK4 method are similar to the Midpoint rule and Simpsons rule for numerical integration.

1.2.6 Matrix operations

You can expect any scientific programming environment to have in-built linear algebra support and Julia does not fall short. See the Linear Algebra documentation. Like MATLAB, and NumPy for Python you can do any sensible operation you would expect on matrices and vectors. This includes solving linear equations, factorizing matrices in various ways (e.g. LU, QR, SVD), computing eigenvalues, and much more.

Almost any linear algebra system uses BLAS and LAPACK under the hood. This is a collection of highly optimized routines for linear algebra. Julia does the same.

Here are some basics:

```
using LinearAlgebra
A = [1 2; 3 4] #A matrix of integers
```

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
A = [1.0 2; 3 4] #A matrix of Float64 values
```

```
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

```
B = inv(A)
```

```
2×2 Matrix{Float64}:
 -2.0   1.0
  1.5  -0.5
```

```
B*A
```

```
2×2 Matrix{Float64}:
 1.0          0.0
 2.22045e-16  1.0
```

In Julia's linear algebra we use `I` for the identity.

```
I
```

```
UniformScaling{Bool}
true*I
```

```
I*A
```

```
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

```
I == inv(A)*A #Will be false due to numerical error
```

```
false
```

```
I  inv(A)*A #\approx + [TAB] shorthand for isapprox()
```

```
true
```

One of the most verstile linear algebra tools is the function "\", sometimes called the left division operator.

```
help?> \
search: \
```

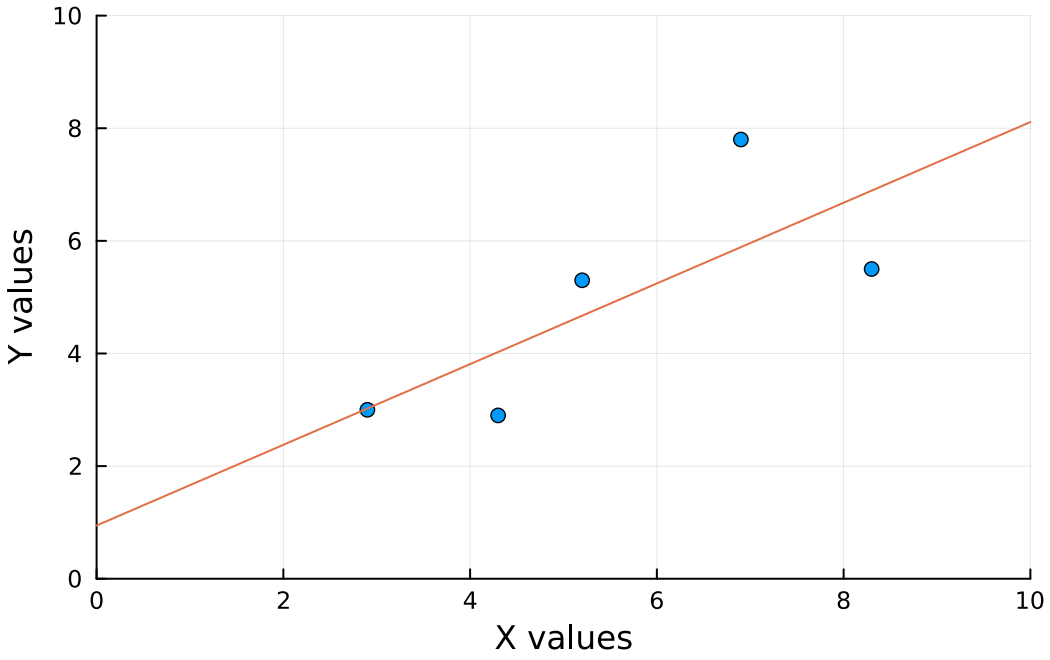
```
  \(x, y)

  Left division operator: multiplication of y by the inverse of x on the left. Gives floating-point resu
  arguments.

  Examples


  julia> 3 \ 6
  2.0

  julia> inv(3) * 6
  2.0

  julia> A = [4 3; 2 1]; x = [5, 6];

  julia> A \ x
  2-element Vector{Float64}:
    6.5
   -7.0

  julia> inv(A) * x
  2-element Vector{Float64}:
    6.5
   -7.0

    ...
```

As shown in th documentation above it is used to "solve" systems of linear equations. However, it can also find least squares solutions to overdetermined systems of equations.

We won't cover all of the linear algebra capabilities but rather use them as they are needed throughout the course. However, let's consider a single example of least squares fitting. Take this very simple dataset of 5 points. The least squares line has an intercept of 0.945 and a slope of 0.716.

```
using DataFrames, GLM, Statistics, LinearAlgebra, CSV, HTTP
resp =
HTTP.request("GET","https://raw.githubusercontent.com/h-Klok/StatsWithJuliaBook/master/data/L1L2data.csv")
data = CSV.read(IOBuffer(String(resp.body)), DataFrame)
xVals, yVals = data.X, data.Y

modelJ = lm(@formula(Y ~ X), data)
b0, b1 = coef(modelJ)
@show b0, b1
scatter(xVals, yVals,
   legend=false,xlim=(0,10),ylim=(0,10),
   xlabel = "X values", ylabel = "Y values")
plot!([0,10], [b0, b0 + 10*b1])
```

```
(b0, b1) = (0.9449358690844867, 0.7164971251658538)
```



This least squares line minimizes (assume $n = 5$),

$$\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)^2.$$

This is just like minimizing $\|A\beta - y\|^2$, where β is the vector of β_0 and β_1 in this case, and A is an $n \times 2$ matrix where the first column has 1's and the second column has the values of the variable x.

It can be shown that the minimizer is a solution of the normal equations,

$$A^T A\beta = A^T y.$$

```
n = length(xVals)
A = [ones(n) xVals] #This is sometimes called the design matrix
```

```
5×2 Matrix{Float64}:
 1.0  2.9
 1.0  4.3
 1.0  5.2
 1.0  6.9
 1.0  8.3
```

Let's check that the normal equations are indeed solved by the solution `b0` and `b1` computed above:

```
 =[b0,b1] #computed via lm() above
lhs_normal_equations = A'A*
rhs_normalequations = A'*yVals
lhs_normal_equations - rhs_normalequations
```

```
2-element Vector{Float64}:
  0.0
 -2.842170943040401e-14
```

Now for the fun of it, and to see several linear algebra features, we'll compute β using 9 alternative methods and in the process see quite a few `LinearAlgebra` functions including `dot`, `pinv`, `qr`, `svd`, and `Diagonal`. As

you can see from the output. In all cases, these different methods of computing the least squares estimates yields the same coefficients. Note that the first two approaches are specific to "simple linear regression" (a single independent variable). However the other approaches will in general work with multiple independent variables as long as the columns of A are linearly independent. The SVD and gradient descent approach will also work in case A is is not a full rank matrix.

```
# Approach A - Using specific formulas for simple linear regression
xBar, yBar = mean(xVals),mean(yVals)
sXX, sXY = ones(n)'*(xVals.-xBar).^2 , dot(xVals.-xBar,yVals.-yBar)
b1A = sXY/sXX
b0A = yBar - b1A*xBar

# Approach B  - Using specific formulas for simple linear regression
b1B = cor(xVals,yVals)*(std(yVals)/std(xVals))
b0B = yBar - b1B*xBar

# Approach C - Solving the normal equations using
b0C, b1C = A'A \ A'yVals

# Approach D
Adag = inv(A'*A)*A'
b0D, b1D = Adag*yVals

# Approach E
b0E, b1E = pinv(A)*yVals

# Approach F
b0F, b1F = A\yVals

# Approach G
F = qr(A)
Q, R = F.Q, F.R
b0G, b1G = (inv(R)*Q')*yVals

# Approach H
F = svd(A)
V, Sp, Us = F.V, Diagonal(1 ./ F.S), F.U'
b0H, b1H = (V*Sp*Us)*yVals

# Approach I - Gradient Descent - we won't describe here
 = 0.002
b, bPrev = [0,0], [1,1]
while norm(bPrev-b) >= √eps()
   global bPrev = b
   global b = b -  *2*A'*(A*b - yVals)
end
b0I, b1I = b[1], b[2]

println(round.([b0A,b0B,b0C,b0D,b0E,b0F,b0G,b0H,b0I],digits=3))
println(round.([b1A,b1B,b1C,b1D,b1E,b1F,b1G,b1H,b1I],digits=3))
```

```
[0.945, 0.945, 0.945, 0.945, 0.945, 0.945, 0.945, 0.945, 0.945]
[0.716, 0.716, 0.716, 0.716, 0.716, 0.716, 0.716, 0.716, 0.716]
```

1.2.7 Sparsity

A very useful data structure for dealing with large matrices is a Sparse Matrix. The idea is that only the non zero elements of a matrix are kept and typical linear algebra methods and extended for this special type. These types of matrices arise very frequently in the numerical solution of PDEs from the finite difference

approximation of certain operators, in applications of graph theory where the edges are sparse and also in Statistics and Machine Learning where the Inverse Covariance Matrix often has a special sparse structure.

Consider the following imaginary social network

```
using SparseArrays, Random, Arpack, LinearAlgebra, InteractiveUtils

n_people = 10000

Random.seed!(0)
rows = Int[]
columns = Int[]

for i = 1 : n_people
   for j = i + 1 : n_people
      is_friend = rand() < 100/n_people
      if is_friend
         push!(rows, i)
         push!(columns, j)
         push!(rows, j)
         push!(columns, i)
      end
   end
end

friend_matrix = sparse(rows, columns, 1)
big_zeros = zeros(n_people, n_people)

#show size in memory of sparse adjacency matrix
varinfo(r"friend_matrix")
varinfo(r"big_zeros")

#now you want to do some spectral graph theory on the network to discover clusters of friends

@time eigs(friend_matrix)

#don't even try to take eigenvalues of a dense matrix this size O(n^2.4ish)!!!!
```

```
  0.412933 seconds (5.05 k allocations: 17.888 MiB)
([100.85219904099387, -19.934428552839307, -19.925178341628687, 19.921069762397796, -19.889223092301965
```