Basic Julia Tutorial

Running/Installing Julia

This is a notebook. To run it you need Julia and Jupyter/VS Code.

1. To run Julia without a local installation, use (for instance) CoCalc or JuliaHub. Also Colab works, but the support is still (as of mid 2025) a bit tentative.
2. To install Julia on your machine, download and install it. This link contains both instructions for the Microsoft Store and binaries for manual download.
3. You may also want to use VS Code with the Julia extension. It is an IDE (editor and more) that can run script files and notebooks. It works well with windsurf and copilot.
4. To run notebooks from your local installation, you need either (1) VS Code with the Julia extension (see above) or (2) Jupyter/JupyterLab.
5. To create and run script files instead of notebooks, see the comments towards the end of this notebook.

Installing Jupyter/JupyterLab - When You Don't Have Python Installed

1. Install IJulia and issue the command `notebook()`. This will make a local Python installation.
2. To use Jupyter, start Julia and run `using IJulia; notebook(dir=pwd())`. Change the directory as needed. Instead, to run JupyterLab do `using IJulia;jupyterlab(dir=pwd())`.

Installing Jupyter/JupyterLab - When You Already Have Python Installed

1. If you already have a Python installation (with Jupyter/JupyterLab), run the following in Julia `ENV["JUPYTER"] = "C:\\Miniconda3\\Scripts\\jupyter.exe"` (change the path as needed) before you install IJulia. The best is perhaps to add this to your *startup.jl* file (for instance, `C:\Users\yourusername\.julia\config\startup.jl` if you are on Windows). You can test whether it works by running `run(`$(ENV["JUPYTER"]) --version`)` from the Julia REPL.
2. On Windows, it helps to allow the Python installer to add python to the system path, in spite of the warnings.
3. Install IJulia and start using Jupyter/JupyterLab (see above).

Documentation and Help

1. Cheat sheet at QuantEcon
2. Wiki book
3. Short tutorials
4. ThinkJulia is a free on-line book
5. The official Julia on-line manual
6. Discussion lists are found at
   - https://discourse.julialang.org/
   - https://stackoverflow.com/questions/tagged/julia-lang
   - https://www.reddit.com/r/Julia/
   - https://forem.julialang.org/
7. In Julia, do `?cos` or `@doc cos` to get help with the cos function (see below for how to get help in VS Code).
8. chatgpt often generate decent Julia code, which can serve as a starting point

About Notebooks

This cell is a "Markdown" cell. This is meant for comments and documentation, not computations.

You can change a cell to "Code" or "Markdown" in the menu.

Markdown cells can handle LaTeX. An example: $\alpha = \beta/2$. A Markdown cell can also contain some *formatting*, like lists of this kind:

1. To insert a new cell, use the menu.

2. The next cell is “Code”. You can run it. Text after a # sign is treated as a comment.

```
a = 2                   #this is a comment
                        #run this cell by using the menu, or by Shift+Enter
```

```
2
```

Load Packages and Extra Functions

There are many packages for Julia, for instance, for plotting or statistical methods (see julialang for a list of lists). To install a package, you do either

1. (works everywhere) run `import Pkg` and then `Pkg.add("Packagename")`
2. (works in the Julia console, the REPL) enter the “package manager mode” by typing `]`, then run `add PackageName`. You leave the package manager mode by backspace.

Once a package is installed, you can use it by running `using PackageName`.

```
using Printf                        #installed by default, used for more control over printing

include("src/printmat.jl");     #just a function for prettier matrix printing, semicolon (;) t
```

How to Get Help

Typing `?cos` or `@doc cos` prints the documentation for the `cos` function, except in a notebook in VS Code. There, do `@doc2 cos` instead. (The `@doc2` macro is defined in the printmat.jl file included above.)

```
@doc2 cos           #Needed in VS Code, works everywhere
# ? cos             #works in REPL and jupyter, not in VS Code
```

```
cos(x)
```

Compute cosine of `x`, where `x` is in radians.

See also `cosd`, `cospi`, `sincos`, `cis`.

```
cos(A::AbstractMatrix)
```

Compute the matrix cosine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the cosine. Otherwise, the cosine is determined by calling `exp`.

Examples

```
julia> cos(fill(1.0, (2,2)))
2×2 Matrix{Float64}:
  0.291927  -0.708073
 -0.708073   0.291927
```

Scalars and Matrices

Create a Scalar and a Matrix

```
q = 1                              #create a scalar
Q = [ 1 2 3;                       #create 2x3 matrix
      4 5 6 ]
println("q is a scalar. To print, use println() or printlnPs()")
println(q)

println("\nQ is a matrix. To print, use display() or printmat()")
printmat(Q)                        #case sensitive (q and Q are different)
                                   #the \n adds a line break
```

```
q is a scalar. To print, use println() or printlnPs()
1

Q is a matrix. To print, use display() or printmat()
     1         2         3
   6           5
```

Picking Out Parts of a Matrix

```
println("\n","element [1,2] of Q: ",            #commands continue on
        Q[1,2])                                 #the next line (until finished)

println("\ncolumns 2 and 3 of Q: ")
printmat(Q[:,2:3])

println("\nline 1 of Q (as a vector): ")
printmat(Q[1,:])
```

```
element [1,2] of Q: 2

columns 2 and 3 of Q:
     2         3
     5         6


line 1 of Q (as a vector):
     1
     2
     3
```

Basic Linear Algebra

The syntax for linear algebra is similar to the standard text book approach. For instance, * `Q'Q` (or `Q'*Q`) multiplies the transpose (Q') with the matrix (Q) * `A*B` does matrix multiplication * `100*Q` multiplies each element of the matrix (Q) by 100. (You can also do `100Q`)

However, to add a scalar to each element of a matrix, use `100 .+ Q`. Notice the dot.

```
println("transpose of Q:")
printmat(Q')

println("Q'Q:")
printmat(Q'Q)

println("scalar * matrix:")
printmat(100*Q)

println("scalar .+ matrix:")        #notice the dot
printmat(100 .+ Q)
```

```
transpose of Q:
     1         4
     2         5
     3         6

Q'Q:
    17        22        27
    22        29        36
    27        36        45

scalar * matrix:
   100       200       300
   400       500       600

scalar .+ matrix:
   101       102       103
   104       105       106
```

Creating a Sequence and a Vector

```
θ = 1:10:21                  #a range, type \theta[TAB] to get this symbol
println("\n","θ is a sequence: ",θ)

ρ = collect(θ)               #make the sequence into a vector, \rho[TAB]
println("\n","ρ is a vector: ")
printmat(ρ)
```

```
θ is a sequence: 1:10:21

ρ is a vector:
     1
    11
    21
```

Comparing Things

To see if the scalar $z \leq 0$, do

```
vv = z <= 0
```

to get a single output (true or false).

Instead, if x is an array, do

```
vv = x .<= 0                    #notice the dot.
```

to get an array of outputs (with the same dimensions as x)

```
x =  [-1.5,-1.0,-0.5,0,0.5]             #this is a vector

println("x values: ")
printmat(x)

vv = -1 .< x .<= 0                      #true for x values (-1,0], vv is a vector
println("true if x is in (-1,0]: ")
printmat(vv)

x2 = x[vv]                              #x values for which vv==true
println("x values that are in (-1,0]: ")
printmat(x2)
```

```
x values:
    -1.500
    -1.000
    -0.500
     0.000
     0.500

true if x is in (-1,0]:
     0
     0
     1
     1
     0

x values that are in (-1,0]:
    -0.500
     0.000
```

Finding Things (extra)

can be done by indexing as shown before. However, it might be more efficient to use one of `findfirst()`, `findall()` and `indexin()`.

Also, if you just need to check if the number `z` is in an array (or any collection) `x`, then use `in(z,x)`.

```
x =  [-1.5,-1.0,-0.5,0,0.5]             #this is a vector

println("x values: ")
printmat(x)

v1 = findfirst(x.==0)
println("(first) index v in x such x[v]==0: ",v1)

v2 = findall(x.>=0)
println("\nall indices v in x such x[v]>=0: ")
printmat(v2)

y = [-1.0,0]
v3 = indexin(y,x)
println("\nindices in x so that x[v] equals the vector y=$y: ")
printmat(v3)

v4 = in(0,x)
println("\ntesting if 0 is in x: ",v4)
```

```
x values:
    -1.500
    -1.000
    -0.500
     0.000
     0.500

(first) index v in x such x[v]==0: 4

all indices v in x such x[v]>=0:
     4
     5
```

```
indices in x so that x[v] equals the vector y=[-1.0, 0.0]:
     2
     4


testing if 0 is in x: true
```

Simple Functions

The next cell defines two new functions, `fn0()` and `fn1()`. They take a scalar input (`x`) and return a scalar output (`y`).

If you instead use a vector as the input, then the computations in these functions fail. (The reason is that you cannot do `x^2` on a vector. You would need `x.^2`.)

However, using the "dot" syntax

```
y = fn1.(x)
```

gives an array as output where element `y[i,j] = fn1(x[i,j])`.

```
fn0(x) = (x-1.1)^2 - 0.5      #define a one-line function

function fn1(x)               #define a function that could have many lines
  y = (x-1.1)^2 - 0.5
  return y
end
```

```
fn1 (generic function with 1 method)
```

```
y = fn1(1.5)
println("result from fn1(1.5): ",y," and compare with fn0(1.5) ",fn0(1.5))

x = [1;1.5]
#y = fn1(x)                  #would give an error
y = fn1.(x)                  #calling on the function, dot. to do for each element in x
println("\nresult from fn1.(x): ")
printmat(y)
```

```
result from fn1(1.5):     -0.340 and compare with fn0(1.5)     -0.340

result from fn1.(x):
    -0.490
    -0.340
```

if-else-end

allows you to run different commands depending on a condition that you specify.

(extra) There are also other (more compact) possibilities for the `if-else` case

1. `y = z <= 2 ? z : 2` or
2. `y = ifelse(z <= 2,z,2)`

```
z = 1.05

if z <= 2          #(a) if true, run the next command (y=z) and then jump to end
    y = z
else               #(b) if (a) is false, do this instead
    y = 2
end

#y = z <= 2 ? z : 2       #these two versions also work
#y = ifelse(z <= 2,z,2)


println(y)
```

```
1.05
```

```
if z < 1               #(a) if true, run the next command (y=1) and then jump to end
    y = 1
elseif 1 <= z <= 2     #(b) if (a) is false, try this instead
    y = z
else                   #(c) if also (b) is false, do this
    y = 2
end

println(y)
```

```
1.05
```

Loops

The are two types of loops: “for loops” and “while loops”.

The *for loop* is best when you know how many times you want to loop, for instance, over all m rows in a matrix.

The *while loop* is best when you want to keep looping until something happens, for instance, that x_0 and x_1 get really close.

The default behaviour in *IJulia* (notebooks), *inside functions* and also at the *REPL* promt is that assignments of `x` inside the loop overwrites `x` defined before the loop. To get the same behavior in scripts, you need to add `global x` somewhere inside the loop. In contrast, a variable (here `z2`) that does not exist before the loop is local to the loop.

To make sure that the `y` calculated inside the loop *does not* affect `y` outside the loop, say `local y` inside the loop.

A Simple “for loop”

The “for loop” in the next cell iterates over all elements in a vector and changes them.

```
x = zeros(3)

for i in 1:length(x)           #1,2,3
    x[i] = 100 + i
end

printmat(x)
```

```
  101.000
  102.000
  103.000
```

A More Complicated “for loop”

The “for loop” in the next cell makes 3 iterations and changes a global x variable. It also illustrates difference between local and global variables.

```
x = 0
y = -999
println("\nBefore loop: x=$x and y=$y \n")    #"$x" inserts the value of x into a string#

for i in 3:3:9                          #or `i = 3:3:9`, or `v=[3,6,9]; i in v`
    #global x                          #only needed in script
    local y                            #don't overwrite y outside loop
    x = x + i                          #adding i to the "old" x
    y = i
    z2 = -998                          #notice: z2 has not been used before
    println("i=$i, x=$x, y=$y and z2=$z2")          #$x prints the value of x
end

println("\nAfter loop: x=$x and y=$y \n")

try                         #to "drive around" errors
    println(z2)             #does not work: z2 is local to the loop
catch
    println("z2 is not defined outside the loop")
end
```

```
Before loop: x=0 and y=-999

i=3, x=3, y=3 and z2=-998
i=6, x=9, y=6 and z2=-998
i=9, x=18, y=9 and z2=-998

After loop: x=18 and y=-999

z2 is not defined outside the loop
```

A Double “for loop”

The next cell uses a double loop to fill a matrix.

```
(m,n) = (4,3)                       #same as m=4;n=3
x = fill(-999,m,n)                  #to put results in, initialized as -999
for i in 1:m, j in 1:n                #can be written on the same line
    x[i,j] = 10*i + j
end
```

```
println("new x matrix: \n")
printmat(x)
```

```
new x matrix:

    11        12        13
    21        22        23
    31        32        33
    41        42        43
```

A Simple “while loop”

```
x = 5

while x > 0.05
    #global x            #only needed in script
    x = x/2
    println("x=$x")
end

println("\nAfter loop: x=$x")
```

```
x=2.5
x=1.25
x=0.625
x=0.3125
x=0.15625
x=0.078125
x=0.0390625

After loop: x=0.0390625
```

A First Plot

With the Plots package you create a a simple plot like this:

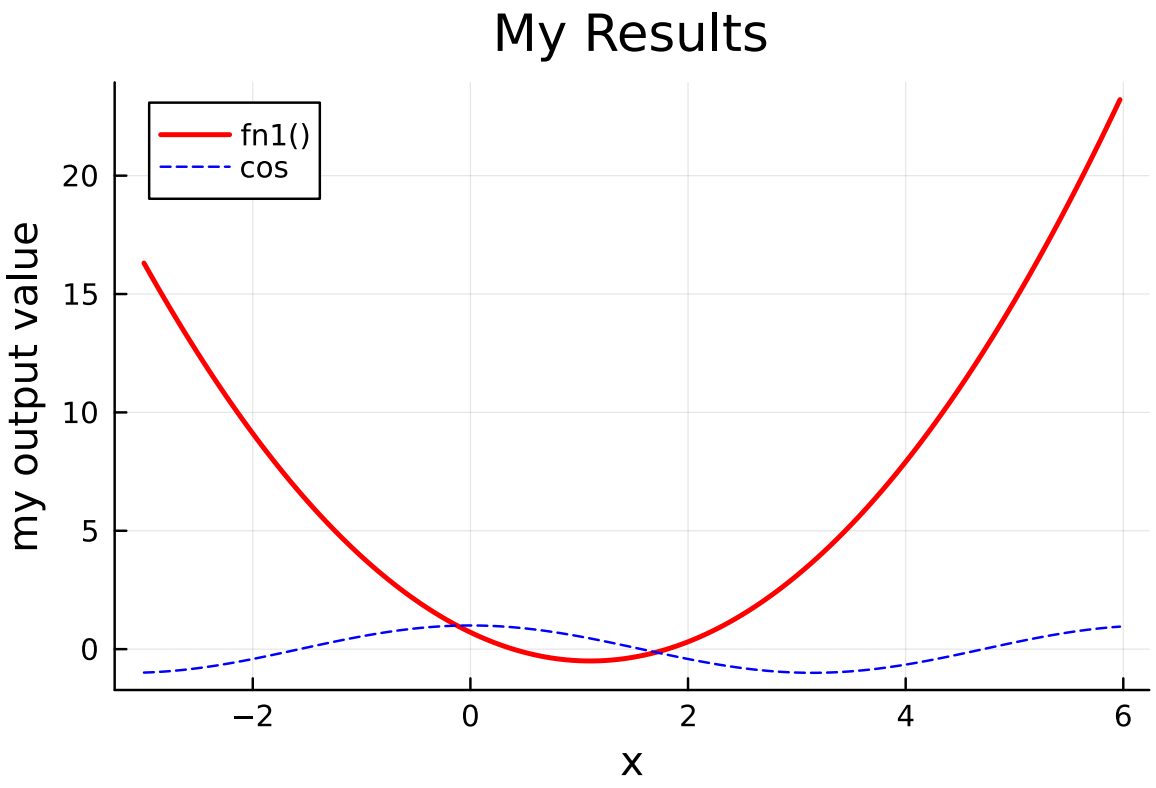
1. Plot two curve by using the `plot([x1 x2],[y1 y2])` command

2. Configure curves by `linecolor =`, `linestyle =` etc
3. Add titles and labels by `title =`, `xlabel =`, etc

Notice: the *first plot is slow*.

```
using Plots                                 #this loads the Plots package
default(size = (480,320),fmt = :png)        # :svg gives prettier plots
```

```
x = -3:6/99:6

p1 = plot( [x x],[fn1.(x) cos.(x)],
           linecolor = [:red :blue],
           linestyle = [:solid :dash],
           linewidth = [2 1],
           label = ["fn1()" "cos"],
           title = "My Results",
           xlabel = "x",
           ylabel = "my output value" )
display(p1)       #not needed in notebook, but useful in script
```



Types: Integers, Floats, Bools and Others

Julia has many different types of variables: signed integers (like 2 or -5), floating point numbers (2.0 and -5.1), bools (false/true), bitarrays (similar to bools, but with more efficient use of memory), strings ("hello"), Dates (2017-04-23) and many more types.

Integers and Floats

```
a = 2                   #integer, Int (Int64 on most machines)
b = 2.0                 #floating point, (Float64 on most machines)
A = [1,2]
B = [1.0,2.0]

println("Finding the type of a, b, A and B:")
println(typeof(a)," ",typeof(b)," ",typeof(A)," ",typeof(B))
```

```
Finding the type of a, b, A and B:
 Float64 Vector{Int64} Vector{Float64}
```

Bools and BitArrays

Bools are "true" or "false". BitArrays are (more memory efficient) versions of this.

```
c = 2 > 1.1
C = A .> 1.5        #A is an array, so C is too

println("Finding the type of c and C:")
println(typeof(c)," ",typeof(C))
```

```
Finding the type of c and C:
Bool BitVector
```

Calculations with Mixed Types

A calculation like "integer" + "float" works and the type of the result will be a float (the more flexible type). Similarly, "bool" + "integer" will give an integer. These promotion rules make it easy to have mixed types in calculations, and also provide a simple way of converting a variable from one type to another. (There are also an explicit convert() function that might be quicker.)

```
println(1+2.0)                  #integer + Float
println((1>0) + 2)              #bool + integer
```

```
3.0
3
```

Running Script Files

instead of notebooks.

Once you have created a script file (called "myscript.jl" below), then you can, for instance, run it from VS Code or directly from the Julia REPL. For the latter, do as follows:

1. Open the Julia REPL and perhaps run `cd("folder_name")` to set the working directory (typically to where your script file is).
2. Run `include("myscript.jl")`

As an alternative, instead do the following:

0. Add this close to the top of the script file: `cd(dirname(@__FILE__))`
1. Open the Julia REPL
2. Run `include("foldername/myscript.jl")`

Performance Tips (extra)

To benchmark your code, install/use the BenchmarkTools.jl package.

```
using BenchmarkTools

function fn2(x)
    n = length(x)
    z = zero(x)       #array of zeros, same size as x
    for i in 2:n
        z[i] = (x[i] + x[i-1])/2   #moving average
    end
    return z
end

x = [1;zeros(101)]
@btime fn2($x)        #notice the $
println()
```

```
59.797 ns (2 allocations: 928 bytes)
```

Stats

This notebook loads some data, reports simple descriptive statistics (means, standard deviations etc) and shows a number of useful plots (scatter plots, histograms, time series plots).

Most of the descriptive stats use the standard package Statistics. The plots rely on the Plots.jl package and the pdf and quantiles are from the Distributions.jl package.

For more stat functions, see the StatsBase.jl package. (Not used here.)

Statistical calculations are often reported in tables. This notebook uses my own `printmat()` function. For more powerful alternatives, consider the PrettyTables.jl package.

Load Packages and Extra Functions

```
using Statistics, Printf, Dates, LinearAlgebra, DelimitedFiles, Distributions

include("src/printmat.jl")
```

```
@doc2
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Load Data from a csv File

The next cell displays the first few lines of a data file in a raw format. The subsequent cell loads the data into a matrix.

```
printblue("the first 5 lines of Data/MyData.csv:\n")

printmat(readlines("Data/MyData.csv")[1:5])
```

```
the first 5 lines of Data/MyData.csv:

date,Mkt_RF,RF,SmallGrowth
197901,4.18,0.77,10.96
197902,-3.41,0.73,-2.09
197903,5.75,0.81,11.71
197904,0.05,0.8,3.27
```

```
x = readdlm("Data/MyData.csv",',',skipstart=1)  #reading the csv file
                                                #skip 1st line
printblue("\nfirst four lines of x:\n")
printmat(x[1:4,:])
```

```
first four lines of x:

197901.000     4.180     0.770    10.960
197902.000    -3.410     0.730    -2.090
197903.000     5.750     0.810    11.710
197904.000     0.050     0.800     3.270
```

Creating Variables

```
ym  = round.(Int,x[:,1])             #yearmonth, like 200712
(Rme,Rf,R) = (x[:,2],x[:,3],x[:,4])  #creating variables from columns of x
Re  = R - Rf                         #do R .- Rf if R has several columns
dN = Date.(string.(ym),"yyyymm") #convert to string and then to Julia Date

printblue("first 4 observations:\n")
printmat(dN[1:4],Rme[1:4,:],Re[1:4,:],colNames=["","Rme","Re"])
```

```
first 4 observations:

                 Rme        Re
```

```
1979-01-01     4.180    10.190
1979-02-01    -3.410    -2.820
1979-03-01     5.750    10.900
1979-04-01     0.050     2.470
```

Some Descriptive Statistics

Means and Standard Deviations

The next few cells estimate means, standard deviations, covariances and correlations of the variables `Rme` (US equity market excess return) and `Re` (excess returns for a segment of the market, small growth firms).

```
μ = mean([Rme Re],dims=1)    #,dims=1 to calculate average along a column
σ = std([Rme Re],dims=1)     #do \sigma[Tab] to get σ

printmat([μ;σ];colNames=["Rme","Re"],rowNames=["mean","std"])
```

```
           Rme        Re
mean     0.602     0.303
std      4.604     8.572
```

Covariances and Correlations

```
printblue("\n","variance-covariance matrix:")
printmat(cov([Rme Re]);colNames=["Rme","Re"],rowNames=["Rme","Re"])

printblue("\n","correlation matrix:")
printmat(cor([Rme Re]);colNames=["Rme","Re"],rowNames=["Rme","Re"])
```

```
variance-covariance matrix:
          Rme        Re
Rme    21.197    28.426
Re     28.426    73.475


correlation matrix:
```

```
          Rme        Re
Rme     1.000     0.720
Re      0.720     1.000
```

OLS

A linear regression $y_t = x_t'b + u_t$, where $x_t = [1; R^e_{m,t}]$.

Clearly, the first element of b is the intercept and the second element is the slope coefficient.

The code below creates a $T \times 2$ matrix x with x_t' in row t. y is a T-element vector (or $T \times 1$).

The GLM.jl package (not used here) has more powerful regression methods.

```
c   = ones(size(Rme,1))          #a vector with ones, no. rows from variable
x   = [c Rme]                    #x is a Tx2 matrix
y   = copy(Re)                   #to get standard OLS notation

b2  = inv(x'x)*x'y               #OLS according to a textbook
b   = x\y                        #also OLS, numerically more stable
u   = y - x*b                    #OLS residuals
R2  = 1 - var(u)/var(y)          #R2

printblue("OLS coefficients, regressing Re on constant and Rme:\n")
printmat([b b2];colNames=["calculation 1","calculation 2"],rowNames=["c","Rme"],width=15)
println("R²: ",R2)
println("no. of observations: ",size(Re,1))
```

```
OLS coefficients, regressing Re on constant and Rme:

     calculation 1  calculation 2
c           -0.504         -0.504
Rme          1.341          1.341

      R²:      0.519
no. of observations:    388
```

```
Covb  = inv(x'x)*var(u)          #traditional covariance matrix of b estimates
Stdb  = sqrt.(diag(Covb))        #std of b estimates
tstat = (b .- 0)./Stdb           #t-stats, replace 0 with your null hypothesis

printblue("\ncoefficients and t-stats:\n")
printmat([b tstat];colNames=["coef","t-stat"],rowNames=["c","Rme"])
```

```
coefficients and t-stats:

          coef    t-stat
c       -0.504    -1.656
Rme      1.341    20.427
```

Drawing Random Numbers and Finding Critical Values

Random Numbers: Independent Variables

```
T = 100
x = randn(T,2)    #T x 2 matrix, N(0,1) distribution

printblue("\n","mean and std of random draws: ")
μ = mean(x,dims=1)
σ = std(x,dims=1)

colNames = ["x₁","x₂"]
printmat([μ;σ];colNames,rowNames=["mean","std"])

printblue("covariance matrix:")
printmat(cov(x);colNames,rowNames=colNames)
```

```
mean and std of random draws:
             x₁         x₂
mean      0.056     -0.222
std       1.065      0.980

covariance matrix:
           x₁         x₂
x₁      1.134      0.093
x₂      0.093      0.961
```

Random Numbers: Correlated Variables

```
μ = [-1,10]          #vector of means
Σ = [1 0.5;          #covariance matrix
     0.5 2]

T = 100
x = rand(MvNormal(μ,Σ),T)'  #random numbers, T x 2, drawn from bivariate N(μ,Σ)
colNames = ["x₁","x₂"]

printblue("covariance matrix:")
printmat(cov(x);colNames,rowNames=colNames)
```

```
covariance matrix:
          x₁        x₂
x₁     1.343     0.558
x₂     0.558     1.893
```

Quantiles ("critical values") of Distributions

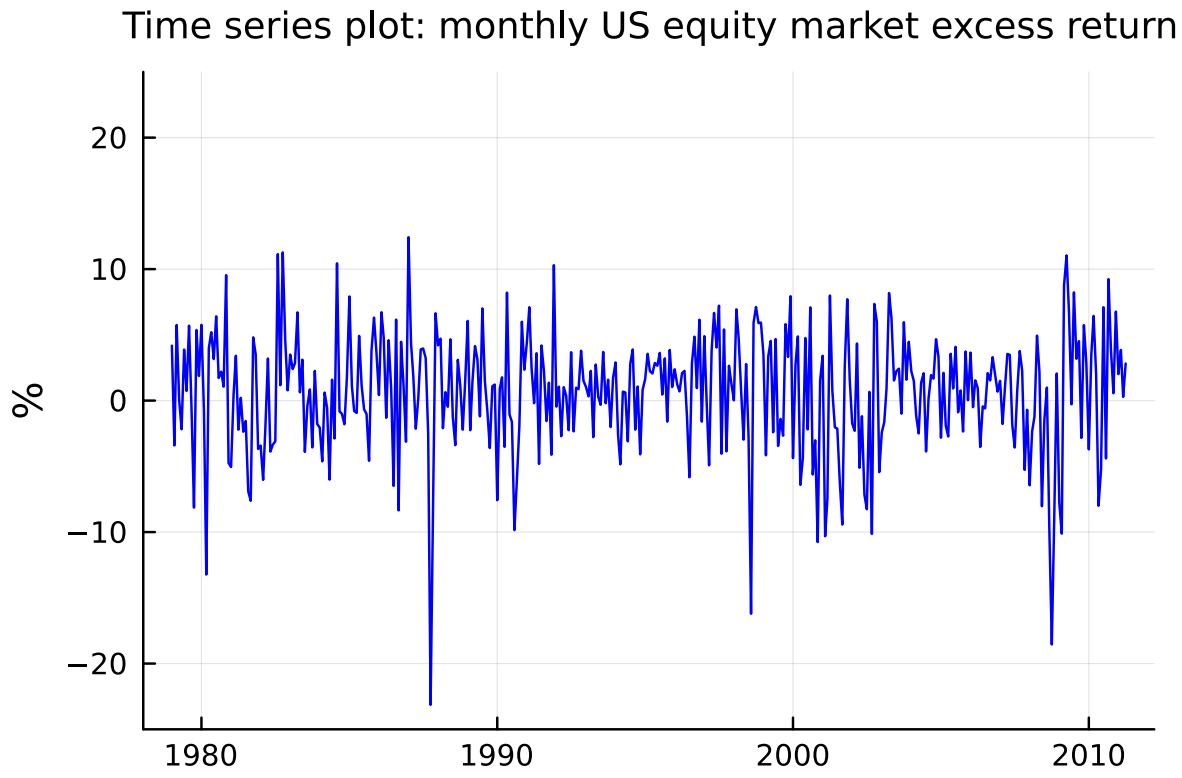
```
N05     = quantile(Normal(0,1),0.05)            #from the Distributions package
Chisq05 = quantile(Chisq(5),0.95)

printblue("\n","5th percentile of N(0,1) and 95th of χ²(5):")
printmat([N05 Chisq05])
```

```
5th percentile of N(0,1) and 95th of χ²(5):
    -1.645    11.070
```

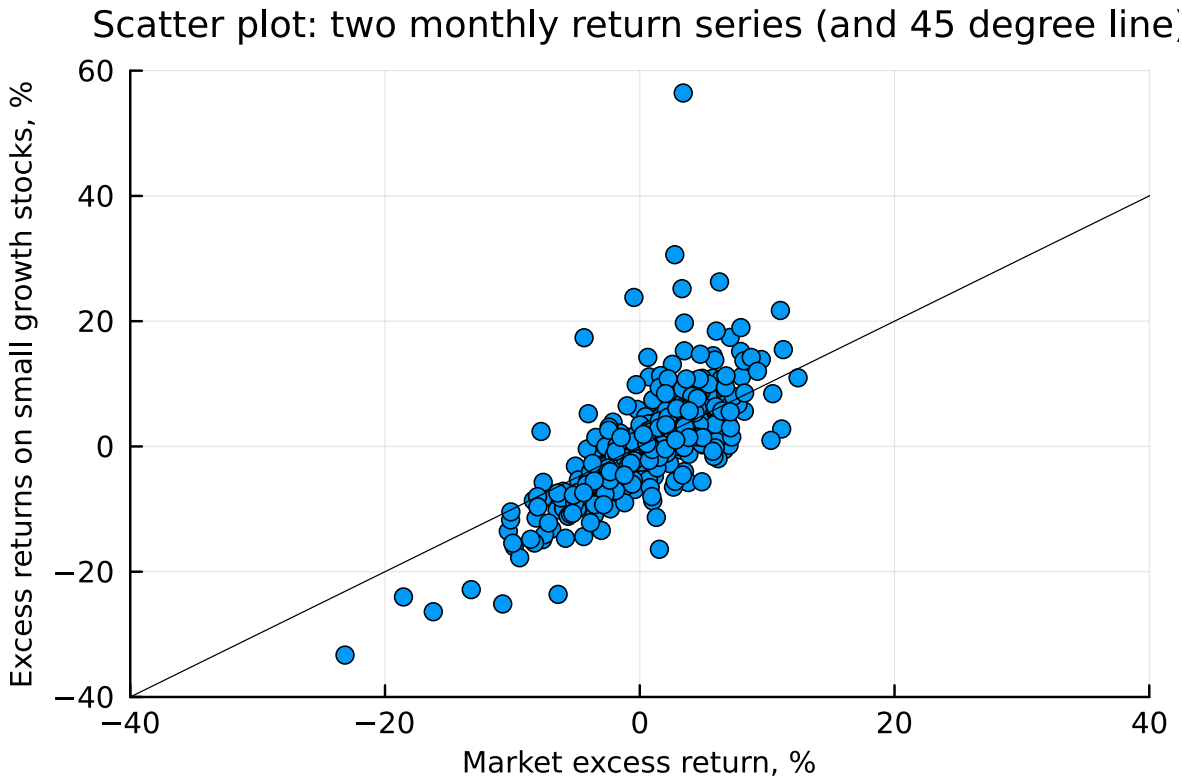
Statistical Plots

```
xTicksLoc = [Date(1980),Date(1990),Date(2000),Date(2010)]
xTicksLab = Dates.format.(xTicksLoc,"Y")

p1 = plot( dN,Rme,
           linecolor = :blue,
           legend = false,
```

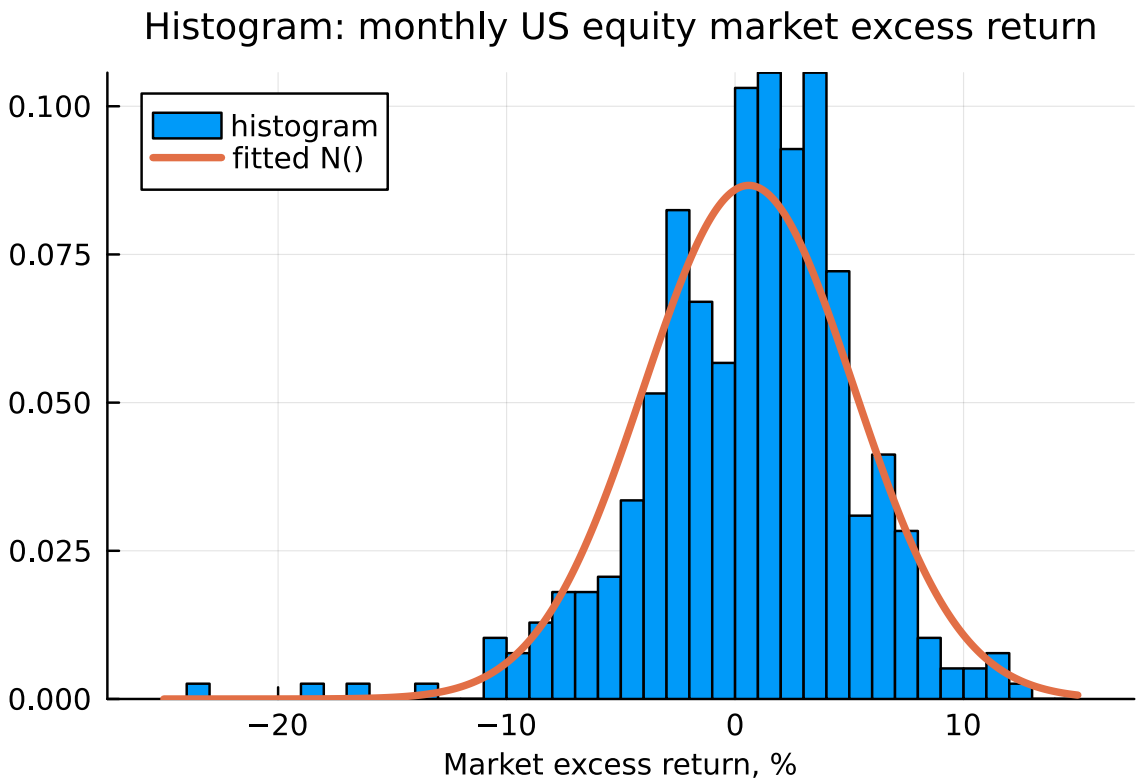
```
            xticks = (xTicksLoc,xTicksLab),
            ylim = (-25,25),
            title = "Time series plot: monthly US equity market excess return",
            titlefontsize = 10,
            ylabel = "%" )
display(p1)
```




```
p1 = scatter( Rme,Re,
              fillcolor = :blue,
              legend = false,
              xlim = (-40,40),
              ylim = (-40,60),
              title = "Scatter plot: two monthly return series (and 45 degree line)",
              titlefontsize = 10,
              xlabel = "Market excess return, %",
              ylabel = "Excess returns on small growth stocks, %",
              guidefontsize = 8 )
plot!([-40;60],[-40;60],color=:black,linewidth=0.5)   #easier to keep this outside plot()
display(p1)
```




```
xGrid = -25:0.1:15
pdfX  = pdf.(Normal(mean(Rme),std(Rme)),xGrid) #"Distributions" wants σ, not σ^2

p1 = histogram( Rme,bins = -25:1:15,
                normalized = true,     #normalized to have area=1
                label = "histogram",
                title = "Histogram: monthly US equity market excess return",
                titlefontsize = 10,
                xlabel = "Market excess return, %",
                legend = :topleft,
                guidefontsize = 8 )   #font size of axis labels
plot!(xGrid,pdfX,linewidth=3,label="fitted N()")
display(p1)
```


Histogram: monthly US equity market excess return
histogram
fitted N()
0.100
0.075
0.050
0.025
0.000
−20
−10
0
10
Market excess return, %

Functions

This notebook shows how to define functions with one (or several) inputs and outputs. It also shows how to use to dot (.) syntax to apply a function to each element of an array.

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");
```

Functions with One Output

The Basic Approach

The basic approach to define a function with the name `fn` is

```
function fn(x,b)
    ...(some code that calculates ys)
    return y
end
```

Once you have defined a function, you can use it (call on it) by, for instance, `y1 = fn(2,1)`. This will generate a `y1` variable (not a `y` variable) in the workspace. Inside the function, `x` is then 2 and `b` is 1. Clearly, if `x1=2 and b1=1`, you get the same result by calling as `y1 = fn(x1,b1)`.

```
function fn1(x,b)                    #x and b are the inputs
    c = 0.5                          #c is only "seen" inside the function
    y = b*(x-1.1)^2 - c
    return y                         #this is the output
end

y1 = fn1(2,1)
println("result from fn1(2,1): ",y1)
```

```
result from fn1(2,1):      0.310
```

Performance Tips (extra)

Wrapping performance critical code in functions often speeds up the computation, since it helps Julia/the compiler to make optimisations.

Performance Tips (extra)

Do not use any global variables inside a function. It slows down the computation and can lead to surprising behaviour (see further down in this notebook). Instead, make all the inputs explicit function arguments.

(Alternatively, declare the global variables to be constants by `const a = 2`).

Default Values for the Inputs

You can change the first line of the function to specify default values as

```
function fn2(x,b=1)
```

In this case you can call on the function as `fn2(x)` and the value of `b` will default to 1. (Clearly, inputs with default values should be towards the end of the list of inputs.)

```
function fn2(x,b=1)                  #b=1 is the default in case we call as fn2(x)
    y = b*(x-1.1)^2 - 0.5
    return y
end

println("result from fn2(2,1) and fn2(2): ",fn2(2,1),fn2(2))
```

```
result from fn2(2,1) and fn2(2):      0.310     0.310
```

Elementwise Evaluation

To apply the function to each element of arrays `x` and `b`, use the dot syntax (broadcasting):

```
y = fn.(x,b)
```

This calculates `fn(x[i],b[i])` for each pair `(x[i],b[i])`.

Instead, with `fn.(x,2)`, you calculate `fn(x[i],2)` for each element `x[i]`.

```
x1 = [1,1.5]
b1 = [2,2]

println("\nresult from fn2.(x1,2): ")
printmat(fn2.(x1,2))

println("\nresult from fn2.(x1,b1): ")
printmat(fn2.(x1,b1))
```

```
result from fn2.(x1,2):
    -0.480
    -0.180


result from fn2.(x1,b1):
    -0.480
    -0.180
```

Elementwise Evaluation over *Some* Inputs (extra)

To apply the function to each element of the array `x`, but keep `b` fixed, use :

```
y = fn.(x,Ref(b))
```

For instance, `fn301.(x1,Ref(b1))` in the next cell will create the 2-element vector

```
1 + (10+20+30)
1.5 + (10+20+30)
```

```
function fn301(x,b)
    y = x + sum(b)
    return y
end

x1 = [1,1.5]
b1 = [10,20,30]

println("result from fn301.(x1,Ref(b1)):")
printmat(fn301.(x1,Ref(b1)))
```

```
result from fn301.(x1,Ref(b1)):
    61.000
    61.500
```

Short Form (extra)

We can also use short forms of a function as in the cell below.

The first version (`fn3`) is just a single expression. It could span several lines. The second version (`fn3b`) is a sequence of expressions (a "compound expression") separated by semicolons (`;`), where the last expression is the function output.

```
fn3(x,b) = b*(x-1.1)^2 - 0.5           #short form of a function

fn3b(x,b) = (c = 0.5;b*(x-1.1)^2 - c)  #this works too. Notice the ;

printlnPs("result from fn3(1.5,1) and fn3b(1.5,1): ",fn3(1.5,1),fn3b(1.5,1))
```

```
result from fn3(1.5,1) and fn3b(1.5,1):     -0.340    -0.340
```

Explicit Names of the Inputs: Keyword Arguments

You can also define functions that take *keyword arguments* like in

```
function fn(x;b,c)
```

Notice the semi-colon (`;`). You can also specify default values as `fn(x;b=1,c=0.5)`

In this case, you *call* on the function by `fn(x;c=3,b=2)` or just `fn(x)` if you want to use the default values. This helps remembering/interpreting what the arguments represent. When calling on the function, you can pass the keyword arguments in any order and you can use comma (`,`) instead of semi-colon (`;`). Using the semi-colon has an advantage, though: if you already have `b` defined, then you can call as `fn(x;c=3,b)`.

(Extra) You can also use a `Dict()` to supply the keyword arguments as in

```
opt = Dict(:b=>1,:c=>0.5)
fn(x;opt...)
```

Notice that the dictionary must use symbols and that you need to use `;` and `...` in the call. It is also possible to mix traditional keywords with a dictionary as in `opt = Dict(:c⇒0.5); fn(x;b=1,opt...)`

```
function fn4(x;b=1,c=0.5)
    y = b*(x-1.1)^2 - c
    return y
end

b = 2
printlnPs("result from fn4(1,c=3,b=2): ",fn4(1;c=3,b))
```

```
result from fn4(1,c=3,b=2):     -2.980
```

Variable Number of Inputs (extra)

When *defining a function*, we can use the `x...` syntax (also called "slurping") on the last non-keyword input to capture a variable number of inputs. Later, when calling on this function, several inputs will be combined into a vector `x`.

For instance, `fn5(-9,1,2,b=10)` in the next cell will create the 2-element vector

```
-9 + 1*10
-9 + 2*10
```

```
function fn5(a,x...;b=1)   #a is traditional input, x... captures several inputs; b=1 is a key
    n = length(x)
    y = zeros(n)
    for i = 1:n            #loop over the elements in x
        y[i] = a + x[i]*b
    end
    return y
end

y = fn5(-9,1,2,b=10)
println("result from fn5(-9,1,2,b=10): ")
printmat(y)
```

```
result from fn5(-9,1,2,b=10):
     1.000
    11.000
```

Conversely, you can also use the `x...` syntax (also called "splatting") when *calling on a function* that requires several inputs.

In the next cell, when running `fn5b([1,2]...)`, `x1` will be 1 and `x2` will be 2.

```
function fn5b(x1,x2)
    println(x1," and then also ",x2)
    return nothing
end

fn5b([1,2]...)
```

```
1 and then also 2
```

Functions with Several Outputs

Several Outputs 1: Basic Approach

A function can produce a "tuple" like `(y1,y2,y3)` as output.

In case you only want the first two outputs, call as `(y1,y2,) = fn(x)`.

Instead, if you only want the second and third outputs, call as `(_,y2,y3) = fn(x)`

You can also extract the second output as `y2 = fn(x)[2]`

If `Y1` is an already existing array, then `(Y1[3],y2,) = fn(x)` will change element 3 of `Y1` (and also assign a value to `y2`).

```
function fn11(x,b=1)
    y1 = b*(x-1.1)^2 - 0.5
    y2 = b*x
    y3 = 3
    return y1, y2, y3
end

(y1,y2,) = fn11(1,2)
println("The first 2 outputs from fn11(1,2): ",y1,y2)

y2 = fn11(1,2)[2]          #grab the second output
println("\nThe result from calling fn11(1,2)[2]: ",y2)

Y1 = zeros(5)              #create an array
(Y1[3],y2,) = fn11(1,2)    #put result in existing array
println("\nY1 array after calling fn11(1,2): ",Y1)
```

```
The first 2 outputs from fn11(1,2):     -0.480     2

The result from calling fn11(1,2)[2]:      2

Y1 array after calling fn11(1,2):     0.000     0.000    -0.480     0.000     0.000
```

Several Outputs 2: Named Tuples and Dictionaries (extra)

Instead of returning several values, it might be easier to combine them into either a "named tuple" or a dictionary and then export that.

You could end the function with

```
    y = (a=y1,b=y2,c=y3)                    #named tuple
    return y
end
```

or

```
   y  = Dict(:a=>y1,:b=>y2,:c=>y3)           #dictionary
   return y
end
```

Several Outputs 3: Elementwise Evaluation (extra)

...can be tricky, because you get an array (same dimension as the input) of tuples instead of a tuple of arrays (which is probably what you want).

One way around this is to reshuffle the output to get a tuple of arrays.

Alternatively, you could loop over the function calls or write the function in such a way that it directly handles array inputs (without the dot). The latter is done in `fn14()`.

```
y = fn11.([1,1.5],2)
println("y, a 2-element vector of tuples (3 elements in each):")
printmat(y)

(y1,y2,y3) = ntuple(i->getindex.(y,i),3)    #split up into 3 vectors

println("\nthe vectors y1, y2 and y3:")
printmat([y1 y2 y3])
```

```
y, a 2-element vector of tuples (3 elements in each):
(-0.48, 2.0, 3)
(-0.18000000000000016, 3.0, 3)


the vectors y1, y2 and y3:
    -0.480     2.000     3.000
    -0.180     3.000     3.000
```

```
function fn14(x,b=1)                  #x can be an array
    y1 = b*(x.-1.1).^2 .- 0.5
    y2 = b*x
    y3 = 3
    return y1, y2, y3
end

(y1,y2,y3) = fn14(x1,2)                #function written to handle arrays
println("result from fn14(x1,2): ")
```

```
printmat([y1 y2])
printmat(y3)
```

```
result from fn14(x1,2):
    -0.480     2.000
    -0.180     3.000

     3
```

Documenting Your Function

To use Julia's help function (`?FunctionName`, `@doc FunctionName`) or `@doc2 FunctionName` (in VS Code, using the macro from printmat.jl), put the documentation in triple quotes, just above the function definition. (No empty lines between the last triple quote and the start of the function.) The cell below illustrates a simple case.

```
"""
    fn101(x,b=1)

Calculate `b*(x-1.1)^2 - 0.5`.

### Arguments
- `x::Number`:    an important number
- `b::Number`:    another number

"""
function fn101(x,b=1)
    y = b*(x-1.1)^2 - 0.5
    return y
end
```

```
fn101
```

```
@doc2 fn101      #get help on the fn101 function, macro defined in printmat.jl
#?fn101          #works in REPL and jupyter, not in VS Code
```

```
fn101(x,b=1)
```

Calculate `b*(x-1.1)^2 - 0.5`.

Arguments

- `x::Number`: an important number
- `b::Number`: another number

Sending a Function to a Function

You can use a function name (for instance, `cos`) as an input to another function. Alternatively, use an anonymous function (see → below).

```
function fn201(f,a,b)   #f is a function, could also write f::Function
    y = f(a) + b
    return y
end

#here f is the cos function
y1 = fn201(cos,3.145,10)
println("result from fn201(cos,3.145,10): ",y1)

#here f is z->mod(z,2), an anonymous function
y2 = fn201(z->mod(z,2),3.145,10)
println("\nresult from fn201(z->mod(z,2),3.145,10): ",y2)
```

```
result from fn201(cos,3.145,10):      9.000

result from fn201(z->mod(z,2),3.145,10):     11.145
```

Performance Tips (extra)

Anonymous functions like `z->fn(z,c)` can be slow when `c` is a global variable. You can speed it up by wrapping the call in a function (see below).

However, defining both `c` and `z->fn(z,c)` inside a loop will not cause a slowdown (`c` is now local to the loop).

```
DoTheCalc(c) = fn201(z->mod(z,c),3.145,10)    #wrapper

c = 2
println(DoTheCalc(c),"\n")
```

```
x = fill(NaN,2)
for i = 1:2                                          #loop
    local c                                          #`c` is a local variable
    c = 2/i
    x[i] = fn201(z->mod(z,c),3.145,10)
end
printmat(x)
```

```
11.145

    11.145
    10.145
```

Global Variables in a Function (extra)

is a bad idea. It slows down the computations and can lead to surprising behaviour (illustrated in the first cell below).

The problem (and the surprise below) is that global variables in a function are taken from the scope where the function was *defined*. For instance, the function `fn101()` below is defined at the top level (this notebook) so it will use the global variable a found there. This holds even if `fn101()` is called from another function with its own a variable.

```
fn101(x) = println("a and a+x: ",a," ",a + x)
a = 1
printblue("From fn101(0):")
fn101(0)

function fn102(x,a)          #does not solve the problem
    println("a according to fn102: ",a)
    fn101(x)                 #since `fn101()` (used inside `fn102`) still uses the global `a`
end
printblue("\nFrom fn102(0,10):")
fn102(0,10)

fn103(x,a) = println("a and a+x: ",a," ",a + x)       #this works the way it should
printblue("\nFrom fn103(0,10):")
fn103(0,10)
```

```
From fn101(0):
```

```
a and a+x: 1 1

From fn102(0,10):
a according to fn102: 10
a and a+x: 1 1

From fn103(0,10):
a and a+x: 10 10
```

In contrast, this is not a problem if the function is *defined inside another function*. (This way of 'capturing' variables is often used.)

```julia
function fn202(x,a)      #allows us to set a, could also set a inside fn
    fn201(x) = println("a and a+x: ",a," ",a + x)   #same as before, but defined inside fn202(
    fn201(x)
end

fn202(0,10)
```

```
a and a+x: 10 10
```

Data Containers

This notebook shows how to put data into different types of "containers" (arrays, dictionaries, tuples, ...).

(This notebook does not discuss DataFrames.jl.)

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");
```

Arrays

are used everywhere in finance and statistics/econometrics.

Vectors, Matrices and High-dimensional Arrays

can be created in many ways: the code below demonstrates just a few of them. See the tutorial on Arrays for (many) more details.

To access a vector element, just do `A[2]` or similarly. Also, you can change an matrix element as in `B[2,1] = -999`, that is, arrays are mutable.

Notice that `D = [A B]` creates an independent copy, so later changing `B` does not affect `D`. However, if we define `E = B`, then a change of `B` will affect both itself and `E`.

```
A = [100,101]           #a vector
printmat(A)             #or display(A)

B = [1 2;               #a matrix
     0 10]
printmat(B)

D = [A B]               #a 2x3 matrix
printmat(D)
```

```
   100
   101

     1         2
     0        10

   100         1         2
   101         0        10
```

```
println("A[2] is ",A[2])          #access an element of a vector

B[2,1] = -999                     #change an element of a matrix
println("\nB is now")
printmat(B)

println("\nD is not affected")
printmat(D)                       #D is not changed when B is
```

```
A[2] is 101

B is now
     1         2
  -999        10


D is not affected
   100         1         2
   101         0        10
```

```
C = rand(2,3,2)         #a 4x3x2 array
display(C)
```

```
2×3×2 Array{Float64, 3}:
[:, :, 1] =
 0.814824  0.607881  0.975816
 0.681557  0.32344   0.369023

[:, :, 2] =
 0.661005  0.908389  0.387029
 0.829191  0.124879  0.861941
```

Arrays of Arrays (or other types)

You can store very different things (a mixture of numbers, matrices, strings) in an array. For instance, if `a` is a vector, `str` is a string and `C` is a matrix, then `x = [a,str,C]` puts them into a vector, where `x[1]` equals the string in `a`.

If you later change elements of the matrix `C` then it will affect `x` (discussed at the end of the notebook).

To first allocate an array of arrays and later fill it, use, for instance, `y = Vector{Any}(undef,3)` can be filled with any sort of elements, while `y = Vector{Array}(undef,3)` can be filled with arrays.

```
a   = 1:3
str = "Hazel"
C   = [11 12;21 22]
x  = [a,str,C]          #element 1 of x is a
foreach(printmat,x)     #loops over the elements of x

println("\n2nd try:")
y    = Vector{Any}(undef,3)
y[1] = 1:3
y[2] = "Hazel"
y[3] = [11 12;21 22]
foreach(printmat,y)
```

```
1
2
3
```

```
 Hazel

11  12
21  22


2nd try:
 1
 2
 3

 Hazel

11  12
21  22
```

Tuples and Named Tuples

are very useful for collecting very different types of data (a number, a string, and a couple of vectors, say). They are also often used as inputs or outputs of functions.

Once created, you cannot change tuples (they are immutable). (Exception: *changing an element of an array* that belongs to the tuple will affect the tuple too.)

The next few cells show how to create tuples and named tuples, how to extract parts of them, merge them and what happens when you to try to change them.

```
a   = 1:3              #how to create tuples and named tuples
str = "Hazel"
C   = [11 12;21 22]

t = (a,str,C)           #a tuple, or tuple(a,str,C)
display(t)

nt = (a=a,str=str,C=C)  #a named tuple, (a2=a,str2=str,C2=C) would also work
display(nt)

nt_b = (;a,str,C)        #also a named tuple, names are given by variables
display(nt_b)
```

```
(1:3, "Hazel", [11 12; 21 22])
```

```
(a = 1:3, str = "Hazel", C = [11 12; 21 22])
```

```
(a = 1:3, str = "Hazel", C = [11 12; 21 22])
```

```
(a2,str2,C2) = t                          #extract the tuple into variables ("destructuring")
println("a2 and str2 are: $a2 $str2 \n")  #a2 is the same as t[1]

println("t[3] is ",t[3],"\n")             #can index into (tuple) t

(a3,str3...) = t                          #str3 will be a tuple of t[2:end]
println("a3 is ",a3,"\n")

println("nt.C is ",nt.C)                  #we can use nt.C as a name (nt is a named tuple)

(;C,a) = nt                               #extract by symbol name, not position
println("C and a are:")
printmat(C)
printmat(a)
```

```
a2 and str2 are: 1:3 Hazel

t[3] is [11 12; 21 22]

a3 is 1:3

nt.C is [11 12; 21 22]
C and a are:
    11        12
    21        22

     1
     2
     3
```

```
t = (t...,3.14)                       #add an element like this
display(t)

nt = (;nt...,abc=3.14)
display(nt)

nt_c = nt[(:a,:C)]                    #create a new named tuple as a subset of another one
```

```
display(nt_c)

nt_d = merge(nt,(abc=3.14,x2="a"))   #merge named tuples to create a new one
display(nt_d)
```

```
(1:3, "Hazel", [11 12; 21 22], 3.14)
```

```
(a = 1:3, str = "Hazel", C = [11 12; 21 22], abc = 3.14)
```

```
(a = 1:3, C = [11 12; 21 22])
```

```
(a = 1:3, str = "Hazel", C = [11 12; 21 22], abc = 3.14, x2 = "a")
```

```
#t[1] = -999                          #cannot change the tuple, uncomment to get an error
#t[4] = 34                            #cannot add elements like this, uncomment to get an error
```

Create a Tuple Dynamically (extra)

when the values and/or names are created dynamically in the program.

Suppose `values` and `names` in the next cell may differ in length from one run of the program to the next. Using `tuple(values...)` and `NamedTuple{names}(values)` allows you to still create tuples/named tuples.

```
values = [a,str,C]

t2 = tuple(values...)                         #or (values...,)
display(t2)

names  = (:a, :b, :c)                         #should be a tuple of symbols (:a)
nt2    = NamedTuple{names}(values)            #or (;zip(names,values)...)
display(nt2)
```

```
(1:3, "Hazel", [11 12; 21 22])
```

```
(a = 1:3, b = "Hazel", c = [11 12; 21 22])
```

Dictionaries

offer a flexible way to collect different types of data. Dictionaries can (in contrast to tuples) be changed, they are mutable. (As usual, changing elements of an array that belongs to the dictionary will affect the dictionary too.)

A dictionary is organised as (key,value) pairs, where the key is the name of the element. You can loop over the elements (see below) and also change/add elements in a loop.

By using `get.(Ref(D),[:a,:C],missing)` you can extract several variables at once.

```
a   = 1:10
str = "Hazel"
C   = [11 12;21 22]

D = Dict(:a⇒a,:str⇒str,:C⇒C)       #dictionary, "a" instead of :a works too
#D = Dict([(:a,a),(:str,str),(:C,C)])  #alternative syntax

println("D[:C] is ",D[:C])

D[:a] = -999                       #can change an element
D[:verse2] = "Stardust"            #can add an element this way
display(D)

D_b = merge(D,Dict(:abc⇒3.14))     #merge Dicts
display(D_b)
```

```
D[:C] is [11 12; 21 22]

Dict{Symbol, Any} with 4 entries:
  :a      => -999
  :verse2 => "Stardust"
  :str    => "Hazel"
  :C      => [11 12; 21 22]

Dict{Symbol, Any} with 5 entries:
  :a      => -999
  :abc    => 3.14
  :verse2 => "Stardust"
  :str    => "Hazel"
  :C      => [11 12; 21 22]
```

From a Dict to a NamedTuple and Back Again (extra)

```
nt = (;D...)      #create a named tuple from a dict
display(nt)

D2 = Dict(pairs(nt))  #create a dict from a named tuple
display(D2)
```

```
(a = -999, verse2 = "Stardust", str = "Hazel", C = [11 12; 21 22])
```

```
Dict{Symbol, Any} with 4 entries:
  :a      => -999
  :verse2 => "Stardust"
  :str    => "Hazel"
  :C      => [11 12; 21 22]
```

A Potential Pitfall in Adding to a Dict (extra)

If you have created a dict with only numbers by

```
D = Dict(:aa=>1)
```

then you cannot add eg. a string by `D[:cc] = "hello"` since `D` is only set up to accept variables of the type `Int`.

```
D = Dict(:aa⇒1)
#D[:cc] = "hello"            #error since D only accepts Int

D = Dict{Any,Any}(:aa⇒1)    #this works
D[:cc] = "hello"
display(D)
```

```
Dict{Any, Any} with 2 entries:
  :aa => 1
  :cc => "hello"
```

Create a Dictionary Dynamically (extra)

See below for examples.

Remark: if you have the names as an array of strings (`names = ["a","b","c"]`), but want symbol names (`:a` etc), then use `Symbol.(names)`.

```
names  = (:a, :b, :c)           #or ["a","b","c"]
values = [a,str,C]

D = Dict(zip(names,values))
display(D)
```

```
Dict{Symbol, Any} with 3 entries:
  :a => 1:10
  :b => "Hazel"
  :c => [11 12; 21 22]
```

```
D = Dict()                    #empty dictionary
for i = 1:length(values)      #loop
    D[names[i]] = values[i]  #add this to the dictionary
end
display(D)
```

```
Dict{Any, Any} with 3 entries:
  :a => 1:10
  :b => "Hazel"
  :c => [11 12; 21 22]
```

Performance Tips (extra)

Named tuples are often faster to create than Dictionaries. However, Dictionaries can be changed afterwards.

Loop over a Tuple/Named Tuple/Dictionary

by using `(key,value) in pairs()`

```
for (key,value) in pairs(nt)              #or over `t` or `D`
    println("$key: $value")
end
```

```
a: -999
verse2: Stardust
str: Hazel
C: [11 12; 21 22]
```

Your Own Tailor Made Data Type (`struct`)

It is sometime conventient to define your own `struct` as a container. The `struct` command creates an immutable type (you cannot change it, except for elements of arrays that belong to it). There is also a `mutable struct` approach.

```
a   = 1:10
str = "Hazel"
C   = [11 12;21 22]

struct MyType             #change to `mutable struct` to be able to change it later
   x                      #can be anything
   s::String              #has to be a String
   z::Array               #has to be an Array
end

x1 = MyType(a,str,C)    #has to specify all arguments

println("x1: ",x1)
println("x1.s: ",x1.s)

#x1 = MyType(1:10,10,[1;2])      #error since 10 is not a string
#x1.x = 3                        #error since we cannot change
```

```
x1: MyType(1:10, "Hazel", [11 12; 21 22])
x1.s: Hazel
```

A Potential Pitfall in Using Arrays in a `struct` (extra)

It is also possible to specify array types (for instance, `z::Array{Float64}` instead of just `z::Array`). This has the effect of converting (if possible) an input array to Float64. While this might have its uses, it also comes with a potential drawback: the conversion breaks the link between the input array and the array inside `MyType`.

A Potential Pitfall when Using an Array in another Data Container (extra)

Suppose you create an array of arrays (or a tuple or a dictionary) called `y`, and that the array `C` is one of the elements.

If you later change *elements* of `C` then it will affect `y` as well (and vice versa). This happens with *arrays*, since they are designed to conserve memory space. For instance, even if `C` is a very large array (several GB, say), creating `y = ["hello",C]` will require very little additional memory space.

If you want an independent copy, use `copy(C)`, for instance, `y = ["hello",copy(C)]`.

In contrast, if you change the shape of `C` then it will *not* affect `y` (but you don't save any memory).

```
a   = 1:10
str = "Hazel"
C   = [11 12;21 22]

x = [a,str,C]
t = (a,str,C)
d = Dict(:a⇒a,:str⇒str,:C⇒C)
e = MyType(a,str,C)

C[1,1] = -999                  #changing an element of C affects x,t,d,e

display(x)
display(t)
display(d)
display(e)
```

```
3-element Vector{Any}:
 1:10
 "Hazel"
 [-999 12; 21 22]
```

```
(1:10, "Hazel", [-999 12; 21 22])
```

```
Dict{Symbol, Any} with 3 entries:
  :a   => 1:10
  :str => "Hazel"
  :C   => [-999 12; 21 22]
```

```
MyType(1:10, "Hazel", [-999 12; 21 22])
```

```
C = 0                #changing the shape of C does not affect x,t,d
display(t)
```

```
(1:10, "Hazel", [-999 12; 21 22])
```

Plots

This file demonstrates how to create plots in Julia by using the Plots.jl package.

The notebook uses the default backend (GR), but Plots.jl supports several alternatives. (See the package document for details.)

See Julia Plots Gallery for examples (with code snippets). Also, see the Plots documentation on colors for instructions to change colours, and the Plots documentation on themes for changing the overall plot theme (to dark, say).

Load Packages and Extra Functions

```
using Dates, Plots

default(size = (480,320),fmt = :png)   #:svg gives prettier plots, but may not work well on Gi
#pythonplot()                          #to use another backend
```

A First Plot

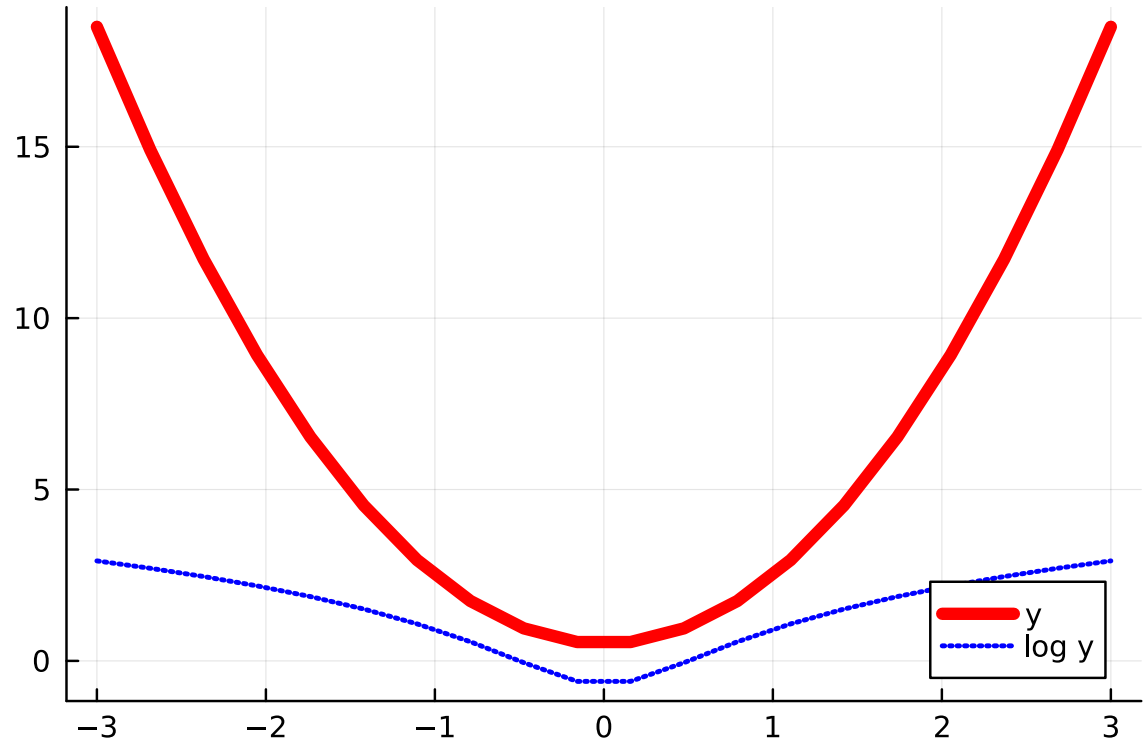
The next cell creates and shows a first plot.

The first plot takes a bit of time. Subsequent plots are much quicker.

The x and y values in this file are in vectors and matrices, so all the examples can be applied to (statistical) data. (In contrast, when your aim is to plot functions, then you can actually avoid generating the y values. See the manual for a discussion.)

The `savefig()` command (see below) saves the plot to a graphics file.

```
x = range(-3,3,length=20)      #something to plot
y = 2*x.^2 .+ 0.5

p1 = plot( [x x],[y log.(y)],
           label = ["y" "log y"],
           linecolor = [:red :blue],
           linestyle = [:solid :dot],
           linewidth = [5 2] )
display(p1)       #not needed in notebook, but useful in script, not needed with pytonplot
```



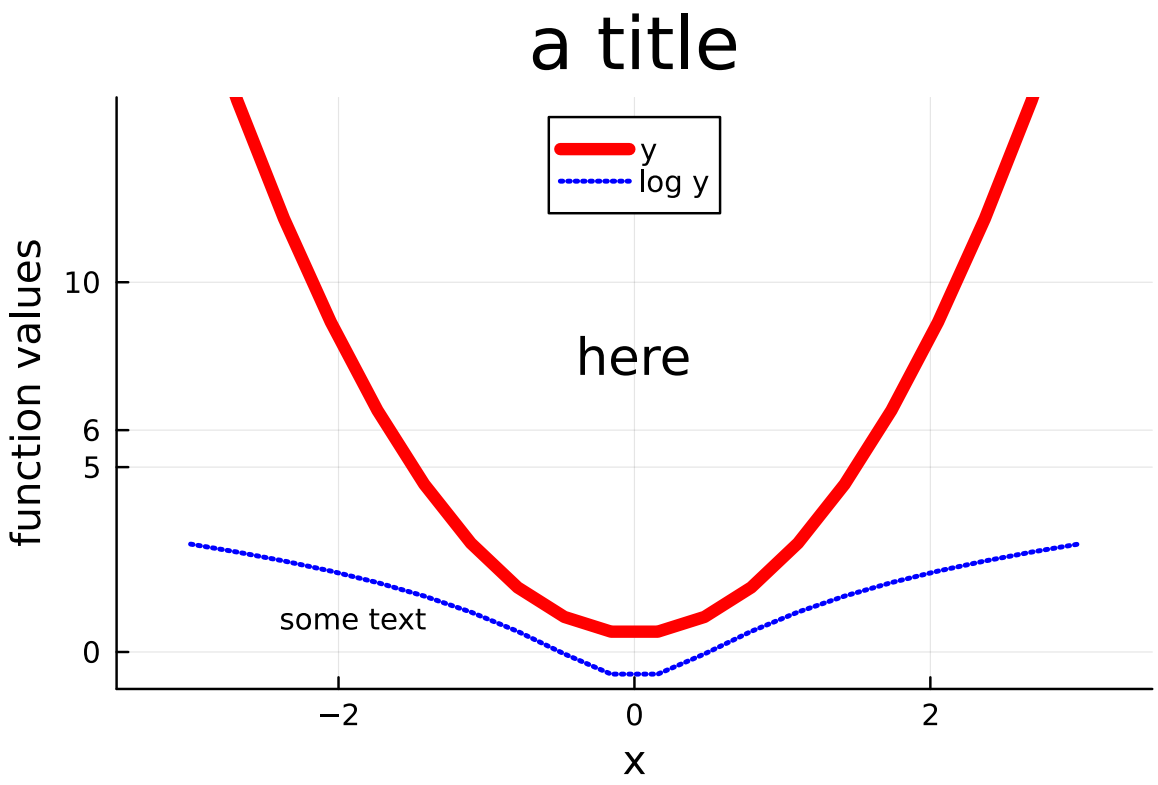

```
plotattr("linestyle")             #see all available options
```

```
:linestyle

Style of the line (for path and bar stroke). Choose from [:auto, :solid, :dash, :dot, :dashdot, :da

Aliases: (:linestyles, :ls, :s, :style).
```

Type: Symbol.

`Series` attribute, defaults to `solid`.

```
#now with title, labels and more

p1 = plot( [x x],[y log.(y)],
           label = ["y" "log y"],
           legend = :top,
           linecolor = [:red :blue],
           linestyle = [:solid :dot],
           linewidth = [5 2],
           title = "a title",
           titlefontsize = 20,
           xlabel = "x",
           ylabel = "function values",
           xlims = (-3.5,3.5),
           ylims = (-1,15),
           xticks = [-2;0;2],
           yticks = [0;5;6;10],
           annotation = ([-1.9,0],[0.9,8],[text("some text",8),"here"]) )
display(p1)
```




```
savefig("AFirstPlot.pdf");                #change to .svg or .png
println("Check that the graphics file is now in the same folder as the notebook.")
```

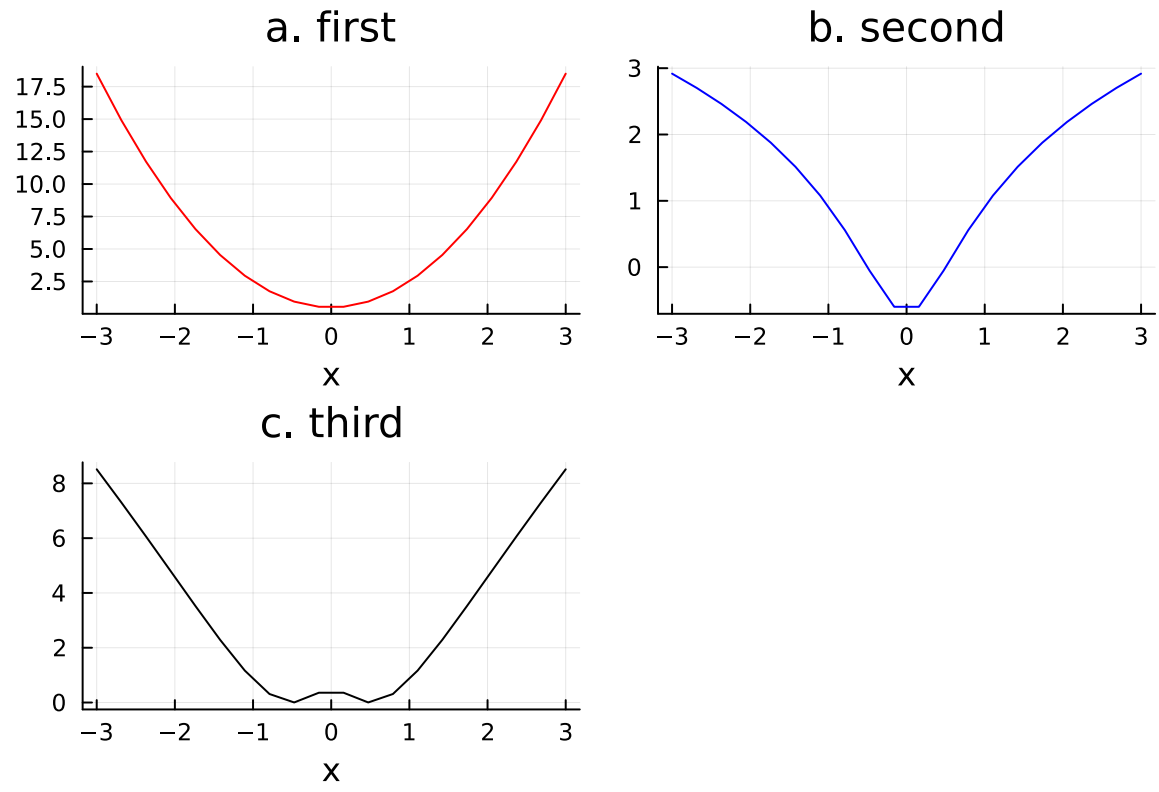
```
Check that the graphics file is now in the same folder as the notebook.
```

Subplots

Use `layout = (2,2)` to create a 2x2 plot and `layout = @layout[a a;a _]` to create a 2x2 plot where the last subplot is blank. In the latter case, a just indicates that there should be a visible subplot (you could use another symbol, eg b) and _ that the subpot should be blank.

```
p1 = plot( [x x x],[y log.(y) log.(y).^2],
           #layout = (2,2),
           layout = @layout[a a;a _],
           size = (600,400),
           linecolor = [:red :blue :black],
           title = ["a. first" "b. second" "c. third"],
```

```
            xlabel = "x",
            legend = false )
display(p1)
```

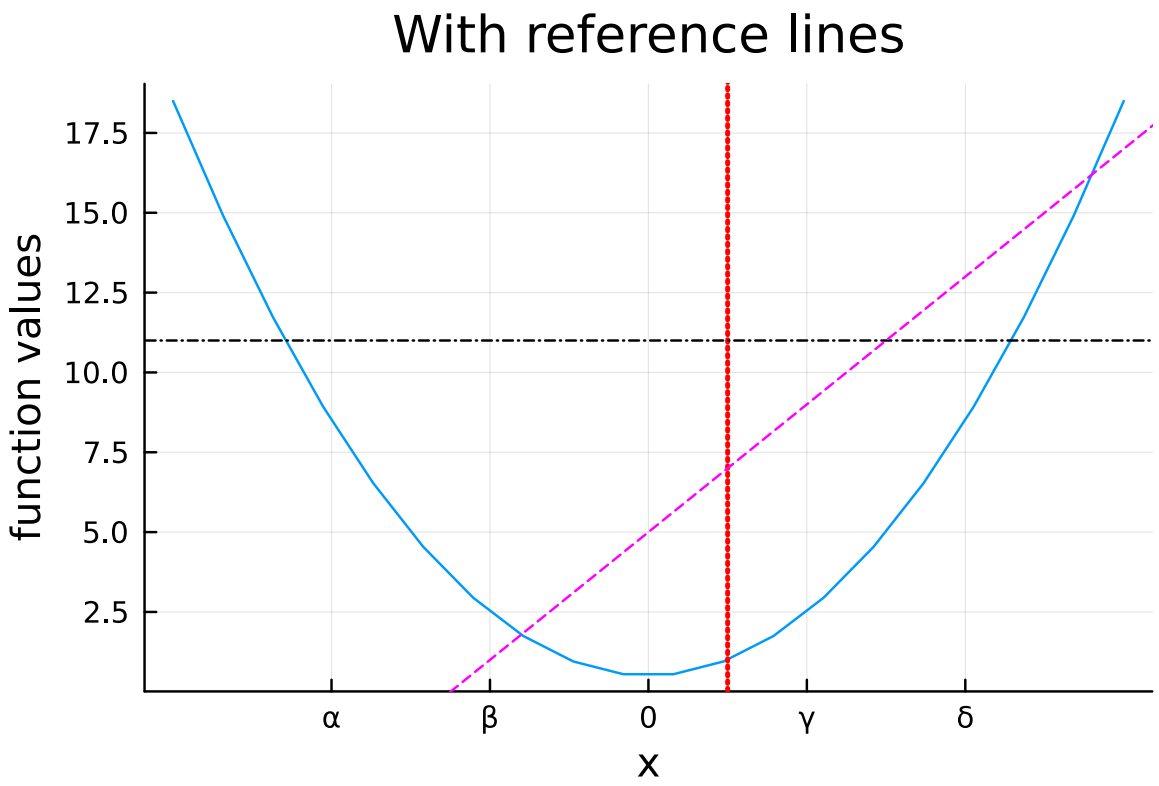


Adding Lines (Horizontal, Vertical and More)

and changing the x-tick marks.

```
#xtickStr = ["low","not so low","0","high","very high"]
xtickStr = ["α","β","0","γ","δ"]

p1 = plot( x,y,
            legend = false,
            title = "With reference lines",
            xlabel = "x",
            ylabel = "function values",
            xticks = (-2:2,xtickStr) )
```

```
vline!([0.5],linecolor=:red,line=(:dot,2))   #easiest to not integrate this in plot()
hline!([11],linecolor=:black,line=:dashdot)
Plots.abline!(4,5,linecolor=:magenta,line=:dash)  #yes, it currently needs the Plots. prefix
display(p1)
```



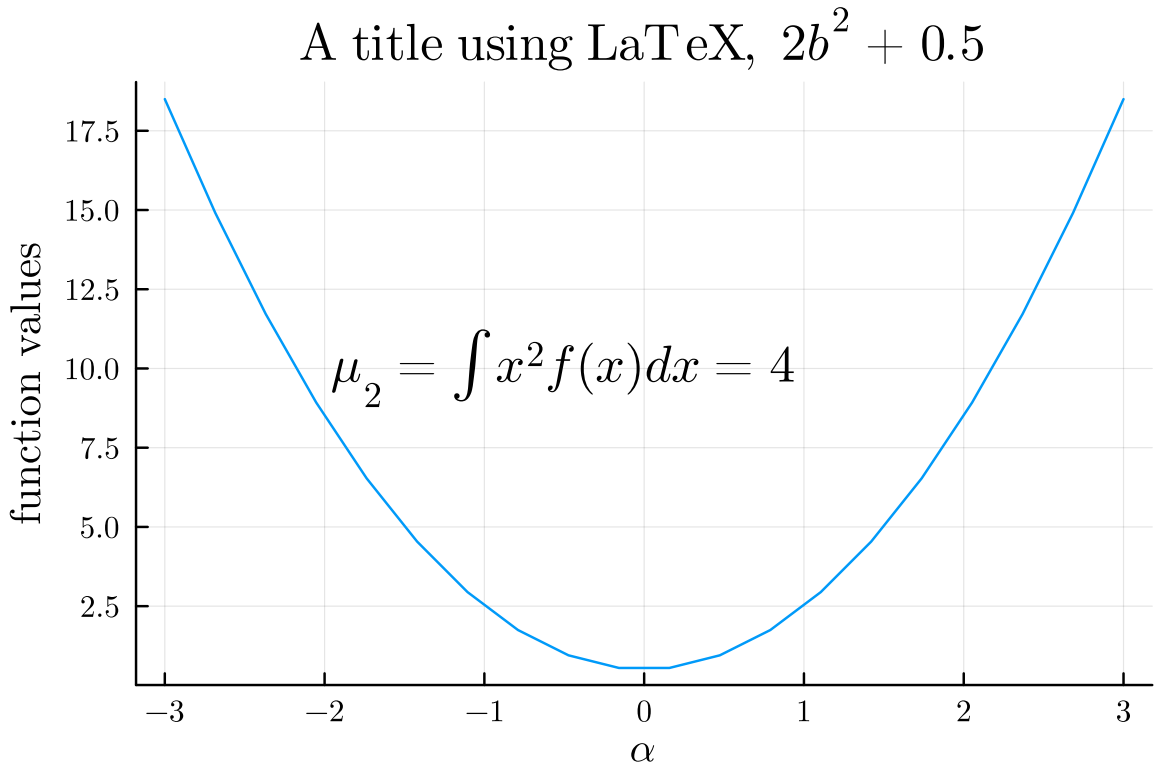
LaTeX in the Figure

`gr()` can include LaTeX elements. Such strings need to be just LaTeX code, so you need a workaround to combine it with text: see `title` in the cell below. In particular, notice that `\mathrm{}` creates ordinary text and that `\` gives a space. To make all text (including tick labels etc) use the traditional TeX font, use `fontfamily="Computer Modern"`.

To insert the value of the variable `z` into the LaTeX string ("string interpolation"), use `%$z` instead of the standard `$z`.

(You may also consider using the `pythonplot()` backend which has extensive support for LaTeX.)

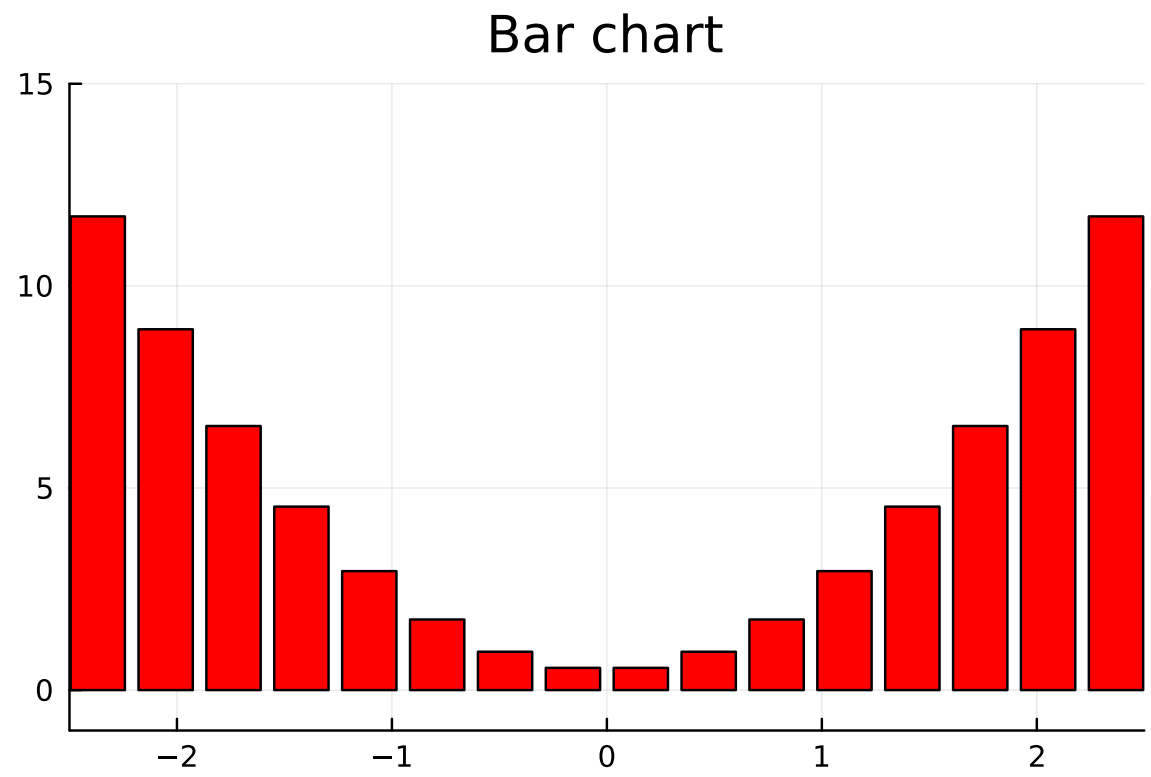
```
using LaTeXStrings                    #add some LaTeX to the figure

z = 4                                 #a value to be used inside the string

p1 = plot( x,y,legend = false,
           title = L"$\mathrm{A \ title \ using \ LaTeX,\ } 2 b^2 + 0.5$",
           xlabel = L"$\alpha$",
           ylabel = "function values",
           annotation = (-0.5,10,L"$\mu_2 = \int x^2 f(x) dx = %$z $"),  #notice the % in %$z
           fontfamily = "Computer Modern" ) #comment out this if it does not work
display(p1)
```



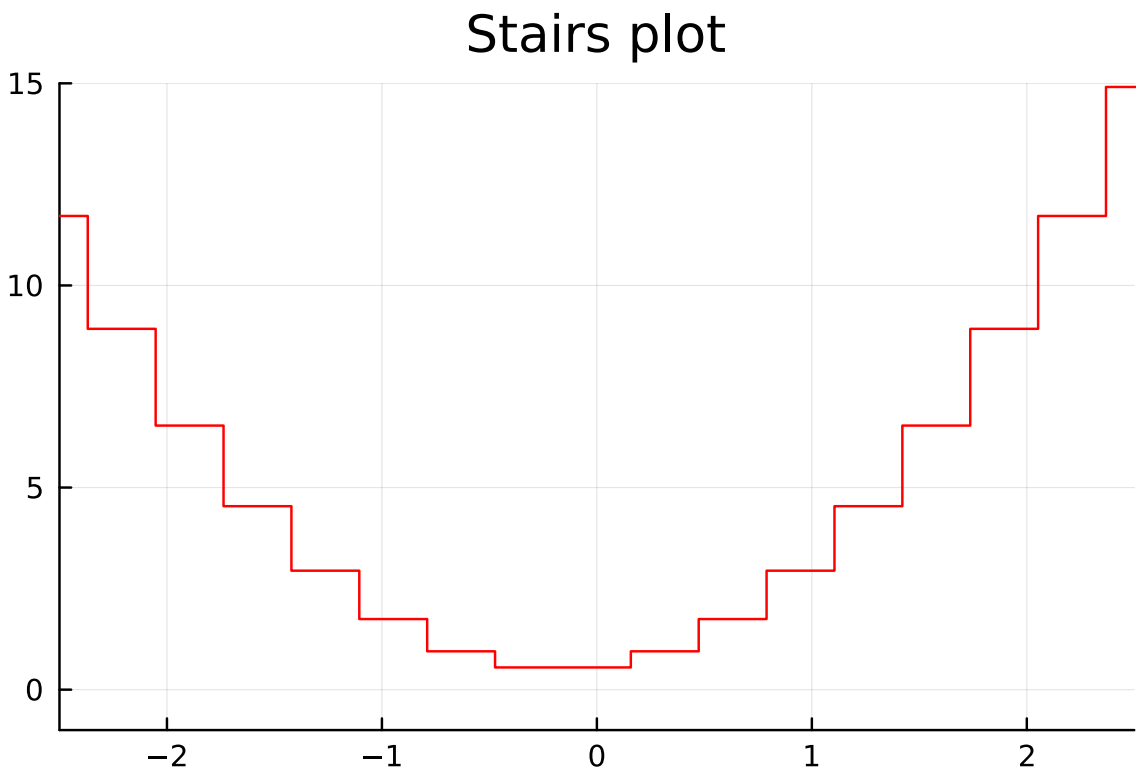

```
savefig("ASecondPlot.pdf");
```

Bars and Stairs Plots

```
p1 = bar( x,y,
          legend = false,
          fillcolor = :red,
          xlims = (-2.5,2.5),
          ylims = (-1,15),
          title = "Bar chart" )
display(p1)
```



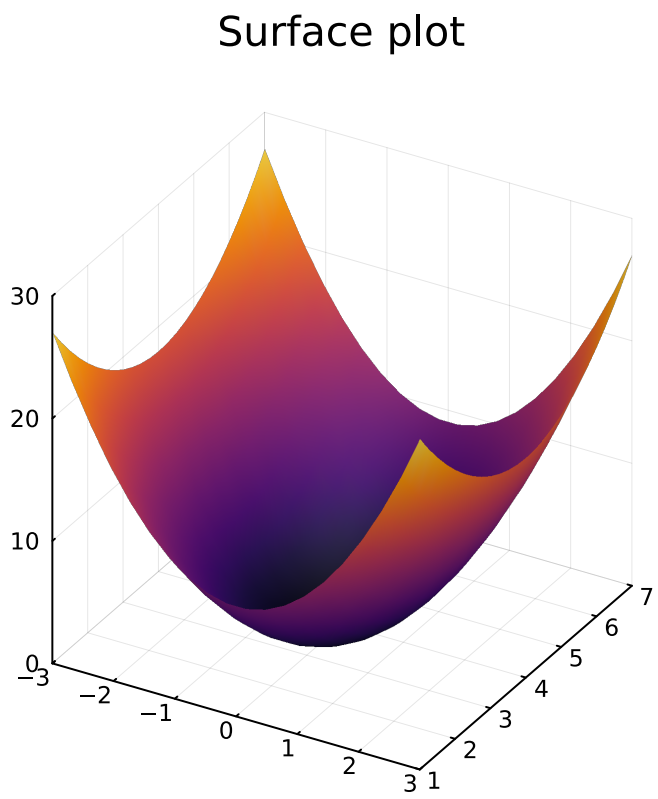

```
p1 = plot( x,y,
           linetype = :steppre,
           linecolor = :red,
           legend = false,
           xlims = (-2.5,2.5),
           ylims = (-1,15),
           title = "Stairs plot" )
display(p1)
```



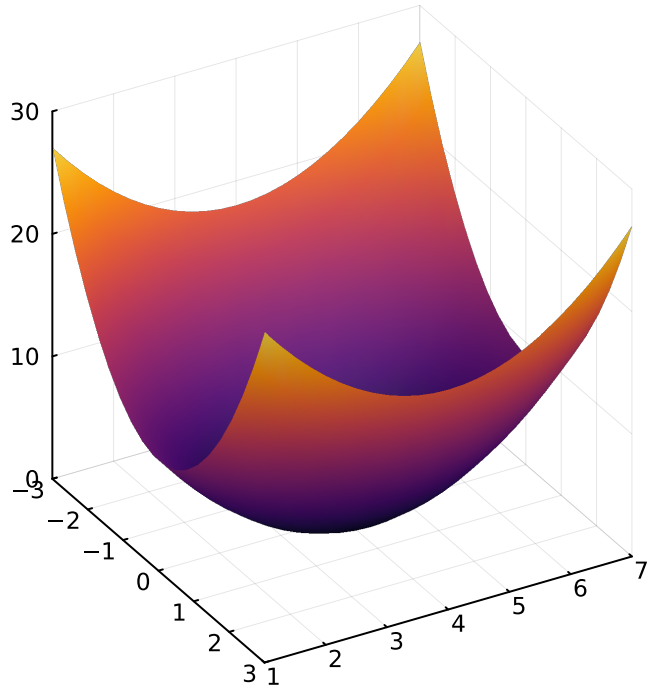
Surface Plots

```
x = range(-3,3,length=20)       #create some "data" to plot
y = range(1,7,length=25)
z = 2*x.^2 .+ (y' .-4).^2

#notice the arguments: x,y,z'
println(size(x),size(y),size(z'))

p1 = surface( x,y,z',
              size = (600,433),
              legend = false,
              xlims = (-3,3),
              ylims = (1,7),
              zlims = (0,30),
              title = "Surface plot" )
display(p1)
```

```
(20,)(25,)(25, 20)
```

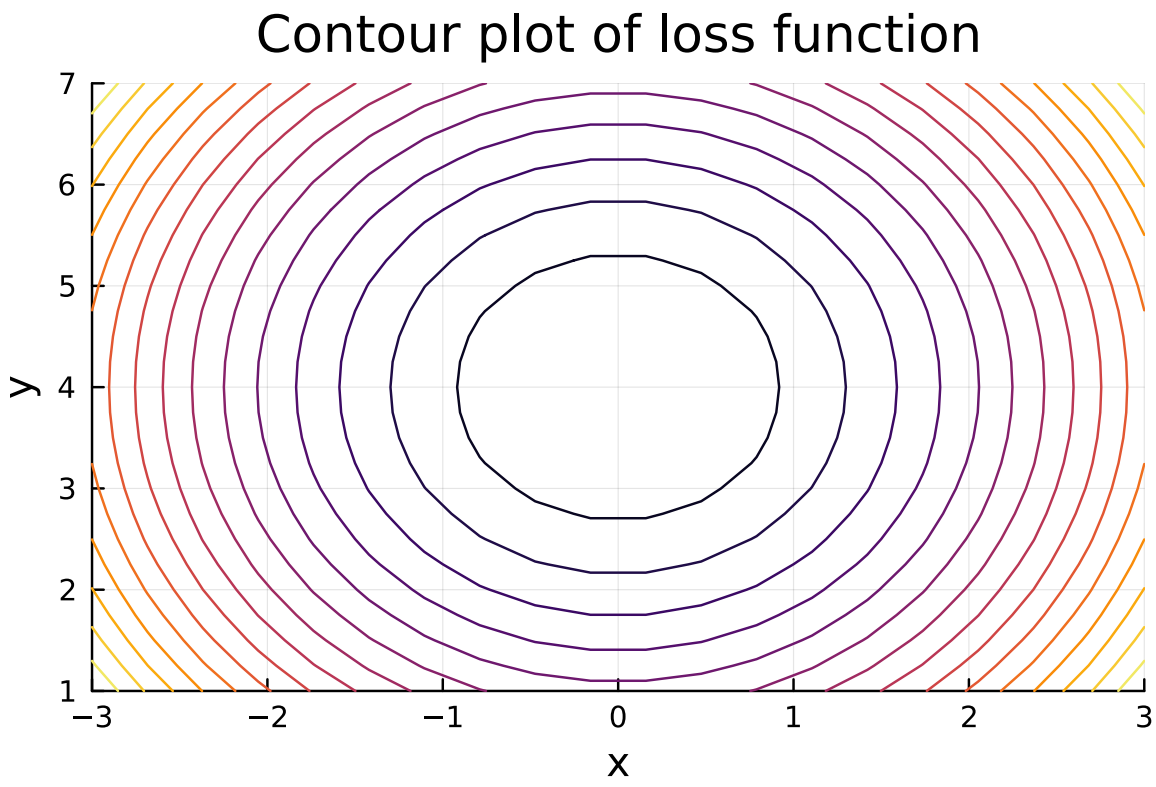



```
p1 = surface( x,y,z',
              size = (600,433),
              camera = (60,30),
              legend = false,
              xlims = (-3,3),
              ylims = (1,7),
              zlims = (0,30),
              title = "Surface plot, rotated" )
display(p1)
```



Contour Plot

```
p1 = contour(x,y,z',                #notice the transpose: z'
             legend = false,
             title = "Contour plot of loss function",
             xlabel = "x",
             ylabel = "y")
display(p1)
```



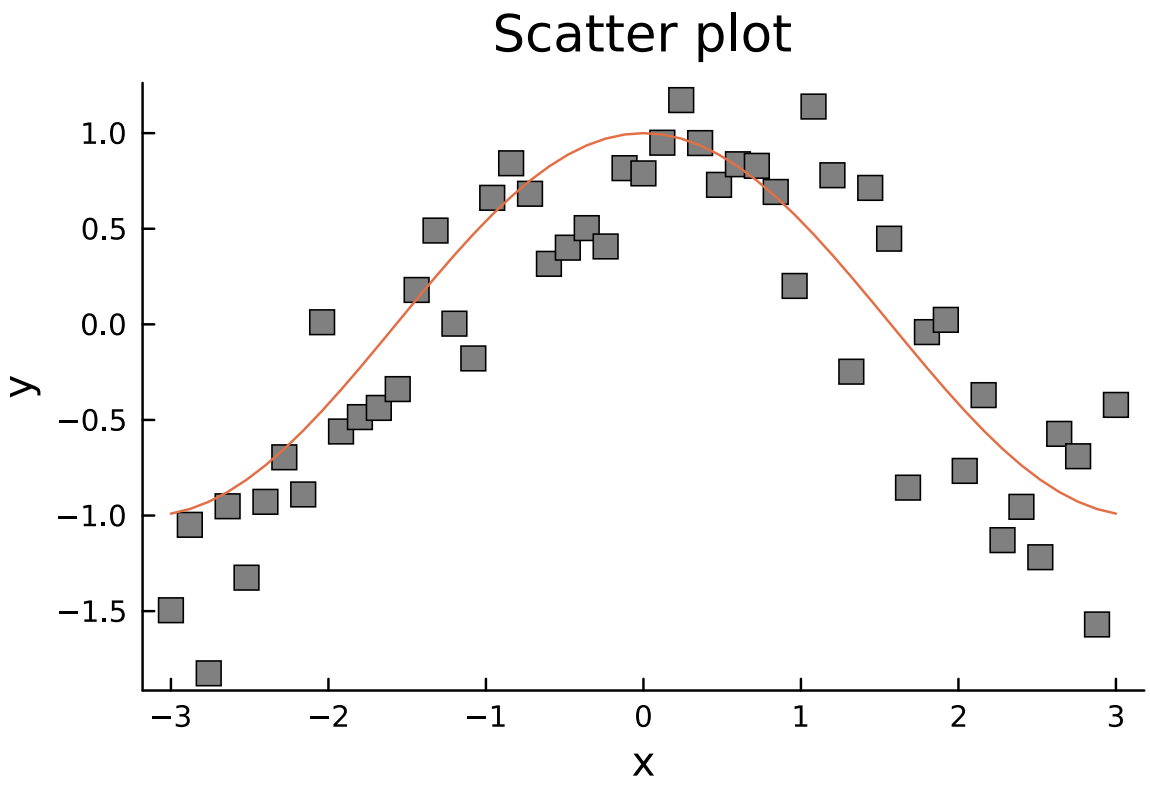
Scatter and Histogram

```
plotattr("markershape")                #to see available options
```

```
:markershape

Choose from [:none, :auto, :circle, :rect, :star5, :diamond, :hexagon, :cross, :xcross, :utriangle

Aliases: (:markershapes, :shape).

Type: Union{Symbol, Shape, AbstractVector}.

`Series` attribute, defaults to `none`.
```

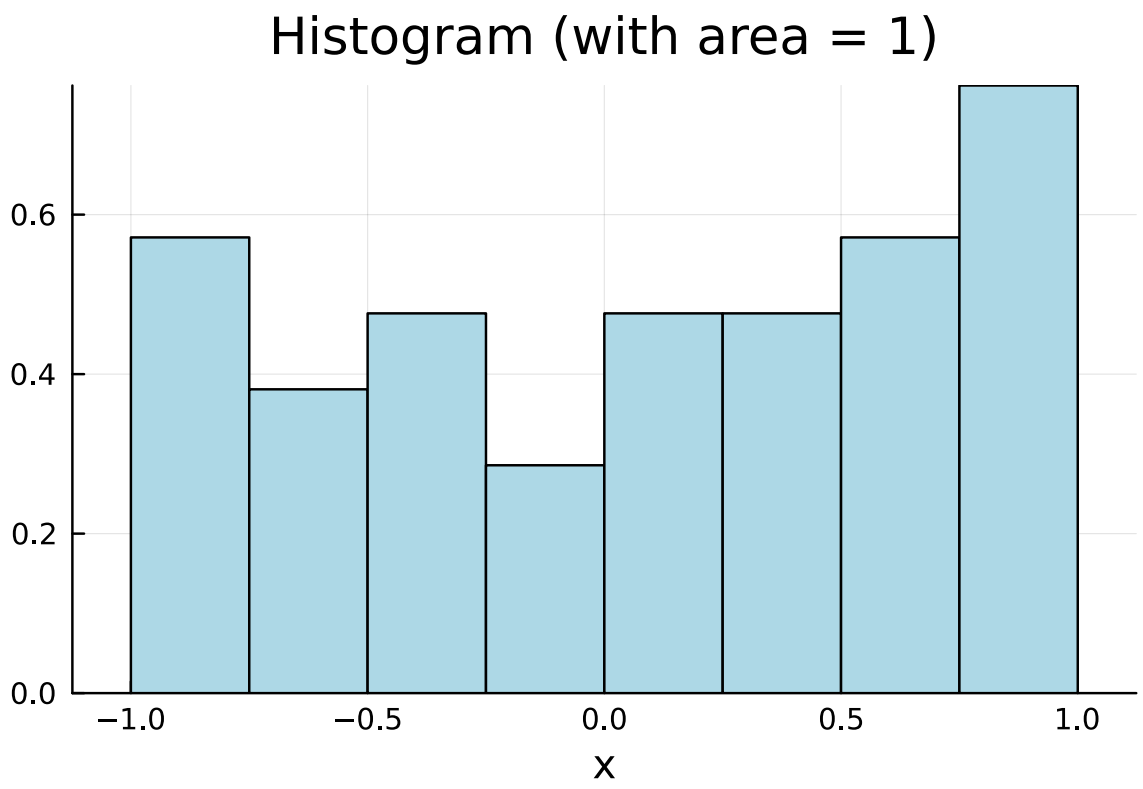
```
N = 51
x = range(-3,3,length=N)
```

```
y = cos.(x) + randn(N)/3

p1 = scatter( x,y,
              markersize = 5,
              markercolor = :grey,
              markershape = :rect,
              legend = false,
              grid = false,
              title = "Scatter plot",
              xlabel = "x",
              ylabel = "y" )
plot!(x,cos.(x))
display(p1)
```




```
p1 = histogram( y,
                bins = -1:0.25:1,
                normalize = true,
                fillcolor = :lightblue,
                legend = false,
```

```
                title = "Histogram (with area = 1)",
                xlabel = "x" )
display(p1)
```



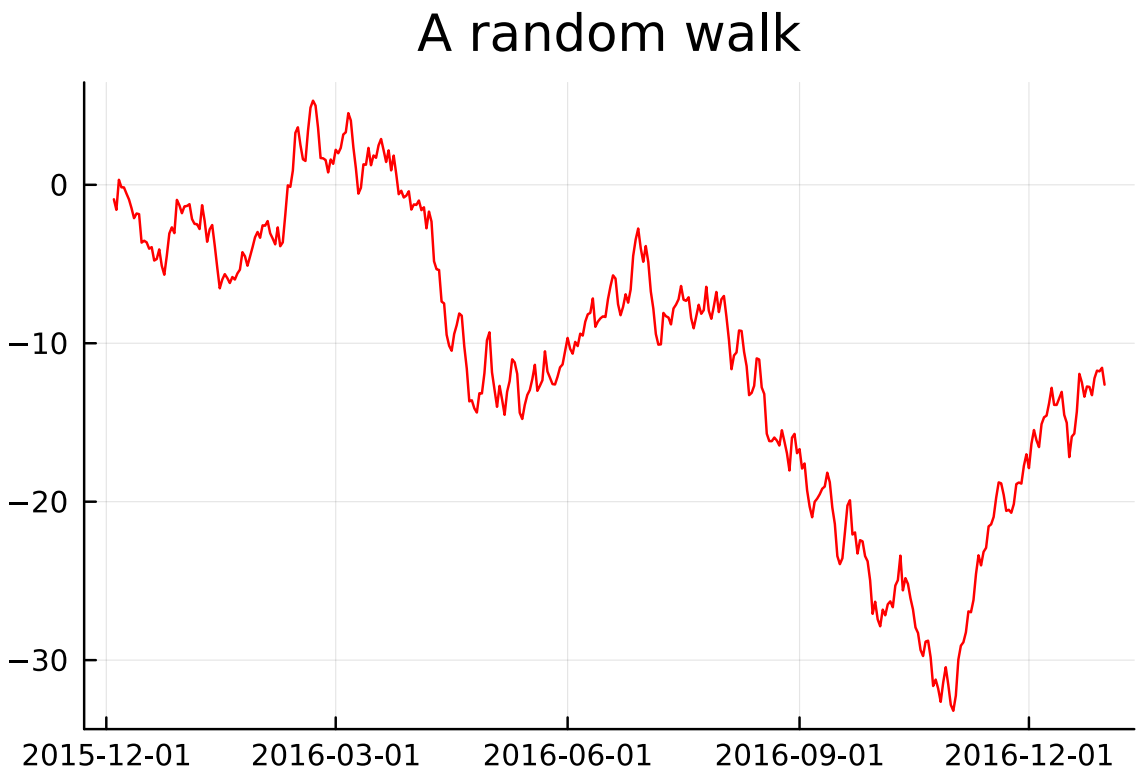
Time Series Plots

We can use a `Date()` vector as the x-axis variable.

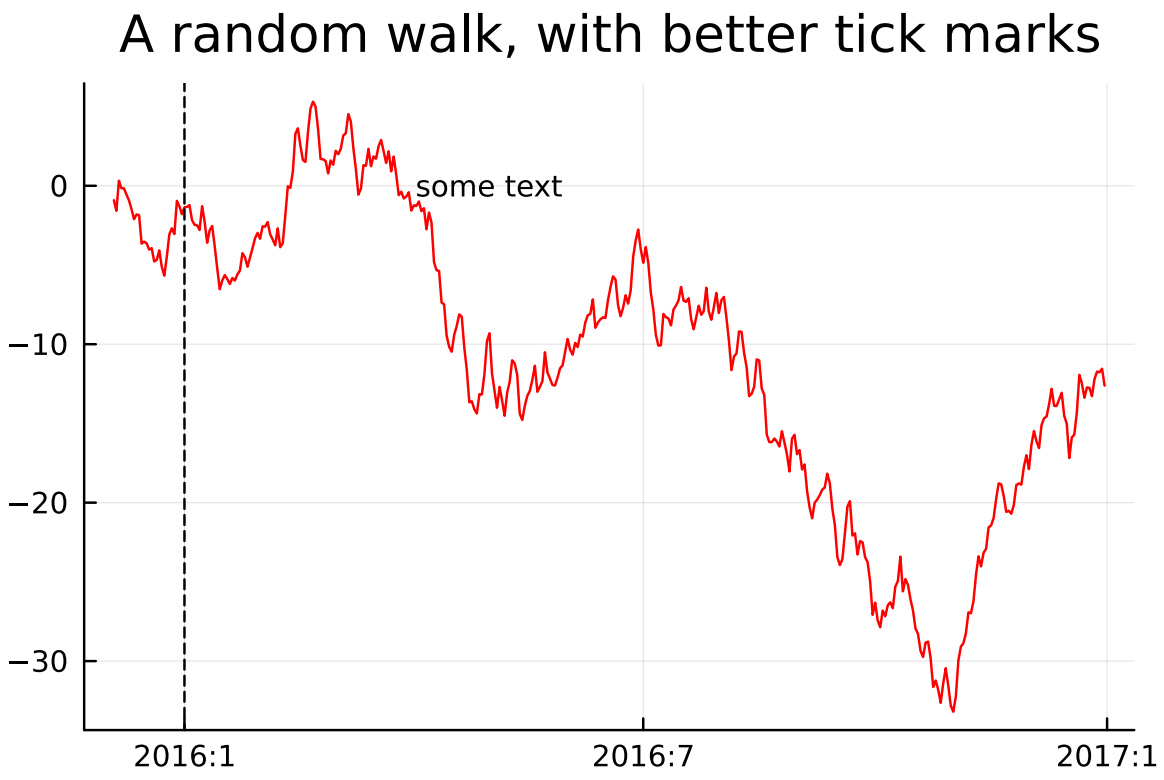
To insert lines, annotations and tick marks, the x values are defined as, for instance, `Date(2016,8,15)`.

```
dN = Date(2015,12,4):Day(1):Date(2016,12,31)   #just faking some dates
y  = randn(length(dN))                               #some random numbers to plot

p1 = plot( dN,cumsum(y),
           linecolor = :red,
           legend = false,
```

```
            title = "A random walk" )
display(p1)
```




```
xTicksLoc = [Date(2016,1),Date(2016,7),Date(2017,1)]
xTicksLab = Dates.format.(xTicksLoc,"Y:m")    #see Dates.format for more options

p1 = plot( dN,cumsum(y),
            linecolor = :red,
            legend = false,
            xticks = (xTicksLoc,xTicksLab),
            title = "A random walk, with better tick marks",
            annotation = (Date(2016,5),0,text("some text",8)) )

vline!([Date(2016)],linecolor=:black,line=(:dash,1))
display(p1)
```



Animations

The following cell animates the probability density function (pdf) of a normally distributed variable x with mean $x_0 + m\mu$ and variance $m\sigma^2$ as m increases.

(Background: this shows how the distribution of a random walk with drift would change as the horizon m is extended.)

```
using Distributions

(x₀,μ,σ) = (0.25,0.1,0.2)               #parameters of the N() distribution

x       = range(-0.25,0.75,length=101)
m_range = range(1/52,1/2,step=1/52)     #different horizons

anim = @animate for m in m_range        #create an animation, like a loop
    dist = Normal(x₀+m*μ,sqrt(m*σ^2))     #computations
    pdf_m = pdf.(dist,x)
```

```
    txt = "m = $(round(m,digits=2))"
    plot(x,pdf_m,                          #create the plot
         ylims = (0,7),
         title = "Pdf of x",
         xlabel = "x",
         annotation = (-0.1,6.5,text(txt,8)),
         legend = false)
         vline!([x₀+m*μ],linecolor=:black,line=(:dot,1))
end

gif(anim; fps=7,show_msg=false)        #show the animation
```

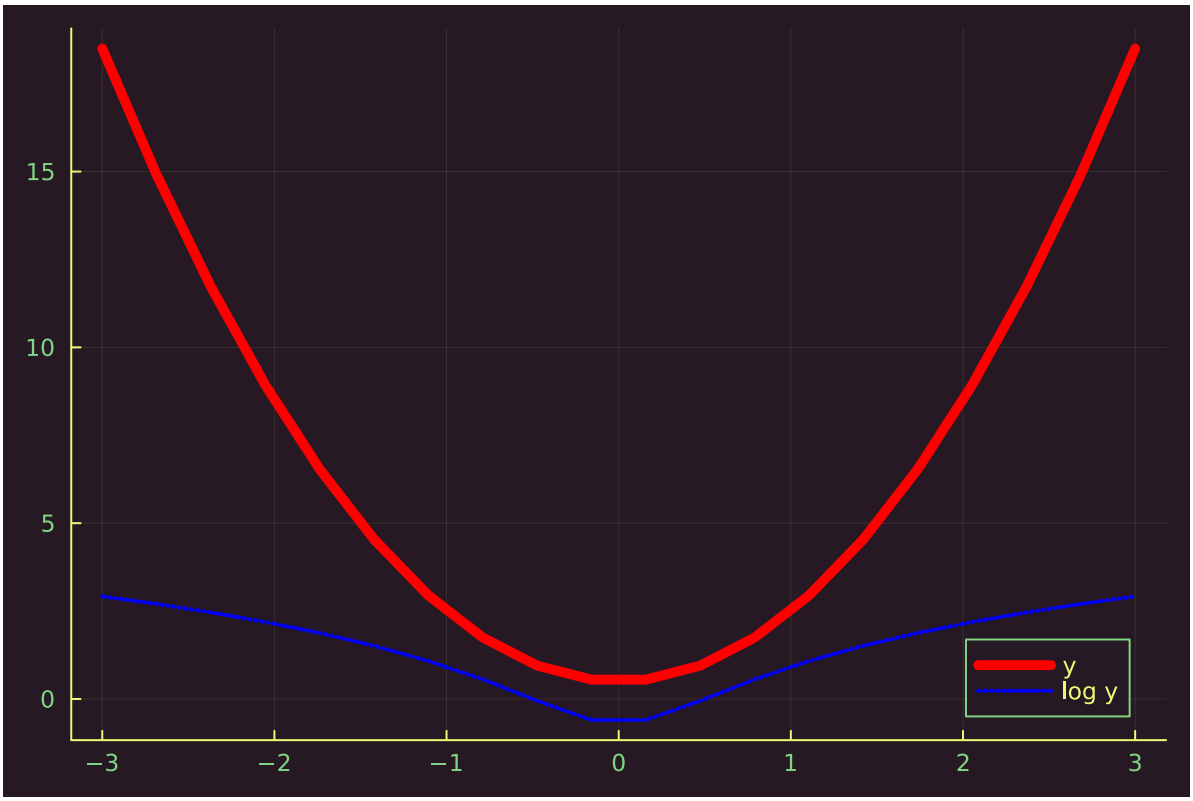
```
Plots.AnimatedGif("C:\\Users\\PSoderlind\\PSDataE\\JL\\JuliaTutorial\\tmp.gif")
```

Plot Themes (extra)

...to give the entire plot another style.

```
using PlotThemes
theme(:lime)
```

```
x = range(-3,3,length=20)      #something to plot
y = 2*x.^2 .+ 0.5

p1 = plot( [x x],[y log.(y)],
           label = ["y" "log y"],
           linecolor = [:red :blue],
           linestyle = [:solid :dot],
           linewidth = [5 2] )
display(p1)
```


15
10
5
0
−3
−2
−1
0
1
2
3
y
log y

Some Typical Financial Calculations

This notebook showcases some classical finance calculations, which requires regressions, loops, root solving, optimisation and more. Details about these techniques are found in the other tutorials (notebooks).

The notebook *(a)* estimates CAPM equations and autocorrelations; *(b)* implements a simple trading strategy; *(c)* calculates Value at Risk using a simple model for time-varying volatility; *(d)* calculates the Black-Scholes option price and implied volatility; *(e)* calculates and draws the mean-variance frontier.

Load Packages and Extra Functions

The Roots package solves non-linear equations, Distributions.jl defines distribution functions for a large number of distrunitions, and StatsBase has methods for estimating autocorrelations etc. The other packages are Standard Libraries (they come with the Julia installer).

```
using Printf, Dates, DelimitedFiles, LinearAlgebra, Roots, Distributions, StatsBase

include("src/printmat.jl");
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Load Data

```
x   = readdlm("Data/MyData.csv",',',skipstart=1)        #monthly return data
ym  = round.(Int,x[:,1])     #yearmonth, like 200712
Rme = x[:,2]                 #market excess return
Rf  = x[:,3]                 #interest rate
```

```
R   = x[:,4]                 #return on small growth stocks
Re  = R - Rf                 #excess returns
T   = size(Rme,1)

dN = Date.(string.(ym),"yyyymm")  #convert to string and then Julia Date
printmat(dN[1:4],Re[1:4],Rme[1:4])
```

```
1979-01-01    10.190     4.180
1979-02-01    -2.820    -3.410
1979-03-01    10.900     5.750
1979-04-01     2.470     0.050
```

CAPM

The CAPM regression is

$R_t^e = \alpha + \beta R_{mt}^e + \varepsilon_t$,

where R_t^e is the excess return of test asset and R_{mt}^e is the market excess return. Theory says that $\alpha = 0$, which is easily tested.

A Remark on the Code

- The OLS coeffcients on the regressors in `x` are called `b` in the code below. Given the way `x` is defined, `b[1]` corresponds to α and `b[2]` to β.
- There are clearly packages that can do OLS, but we do not use them here.

```
x    = [ones(T) Rme]            #regressors
y    = copy(Re)                 #to get standard OLS notation
b    = x\y                      #OLS, intercept and slope
u    = y - x*b                  #residuals
covb = inv(x'x)*var(u)          #cov(b), see any textbook
stdb = sqrt.(diag(covb))        #std(b)
R2   = 1 - var(u)/var(y)

printblue("Regression results:\n")
printmat([b stdb b./stdb],colNames=["coeff","std","t-stat"],rowNames=["α","β"])
println("R2: ",R2)
println("no. of observations: ",T)
```

```
Regression results:

      coeff       std    t-stat
α    -0.504     0.304    -1.656
β     1.341     0.066    20.427

      R2:      0.519
no. of observations:    388
```

Return Autocorrelation

That is, the correlation of R^e_t and R^e_{t-s}.

It can be shown that the t-stat of an autocorrelation is $\sqrt{T}$ times the autocorrelation.

A Remark on the Code

- `autocor(Re,plags)` from the `StatsBase` package estimates the autocorrelations for the lags defines in the rang (or vector) `plags`.

```
plags = 1:5
ρ = autocor(Re,plags)         #using the StatsBase package

printblue("Autocorrelations:\n")
printmat([ρ sqrt(T)*ρ],colNames=["autocorr","t-stat"],rowNames=string.(plags),cell00="lag")
```

```
Autocorrelations:

lag  autocorr    t-stat
1       0.216     4.253
2       0.002     0.046
3      -0.018    -0.359
4      -0.065    -1.289
5      -0.027    -0.536
```

A Trading Strategy

The next cell implements a very simple momentum trading strategy.

1. If $R^e_{t-1} \geq 0$, then we hold the market index and shorten the riskfree from $t-1$ to t. This means that we will earn R^e_t.
2. Instead, if $R^e_{t-1} < 0$, then we do the opposite. This means that we will earn $-R^e_t$.

This simple strategy could be coded without using a loop, but "vectorization" does not speed up much.

We compare with a "passive" portfolio, which is just holding the asset all the time (and earning R^e_t in each period).

```
(w,Rp)  = (fill(NaN,T),fill(NaN,T))
for t = 2:T
  w[t]  = ifelse(Re[t-1] < 0,-1,1)         #w is -1 or 1
  Rp[t] = w[t]*Re[t]
end

R_all = hcat(Rp[2:end],Re[2:end])          #Tx2, matrix of both returns
μ     = mean(R_all,dims=1)
σ     = std(R_all,dims=1)

printblue("The annualized mean excess return and Sharpe ratios of the strategy and a passive p
xx = [μ*12;sqrt(12)*μ./σ]
printmat(xx;rowNames=["mean","SR"],colNames=["strategy","passive"])
```

```
The annualized mean excess return and Sharpe ratios of the strategy and a passive portfolio:

      strategy   passive
mean    26.733     3.329
SR       0.932     0.112
```

Value at Risk

The next cell constructs an simple estimate of σ^2_t as a backward looking moving average:

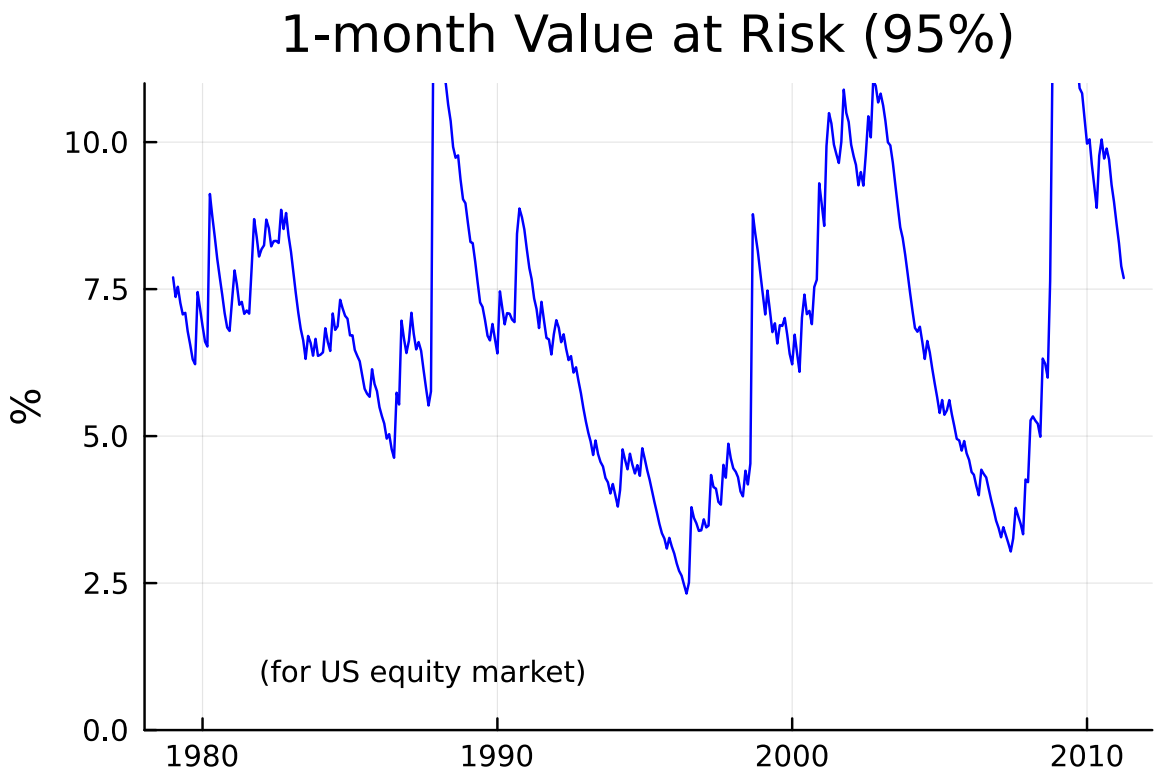
$\sigma^2_t = \lambda\sigma^2_{t-1} + (1-\lambda)(R^e_{t-1} - \mu_{t-1})^2$, where $\mu_t = \lambda\mu_{t-1} + (1-\lambda)R^e_{t-1}$

Then, we calculate the 95% VaR by assuming a $N(\mu, \sigma^2_t)$ distribution:

$\text{VaR}_t = -(\mu_t - 1.64\sigma_t)$.

```
(μ₀,σ₀) = (0.5,5.0)          #some starting values for obs t=0

λ = 0.94
(μ,σ²) = (fill(μ₀,T),fill(σ₀^2,T))    #vectors, time-varying mean and variance
for t in 2:T
    μ[t]  = λ*μ[t-1]  + (1-λ)*Rme[t-1]
    σ²[t] = λ*σ²[t-1] + (1-λ)*(Rme[t-1]-μ[t-1])^2    #RiskMetrics approach
end

VaR95 = -(μ .- 1.64*sqrt.(σ²));      #VaR at 95% level
```

```
xTicksLoc = [Date(1980);Date(1990);Date(2000);Date(2010)]
xTicksLab = Dates.format.(xTicksLoc,"Y")

p1 = plot( dN,VaR95,
           color = :blue,
           legend = false,
           xticks = (xTicksLoc,xTicksLab),
           ylim = (0,11),
           title = "1-month Value at Risk (95%)",
           ylabel = "%",
           annotation = (Date(1982),1,text("(for US equity market)",8,:left)) )
display(p1)
```



Options

Black-Scholes Option Price

Let S be the the current spot price of an asset and y be the interest rate.

The Black-Scholes formula for a European call option with strike price K and time to expiration m is

$C = S\Phi(d_1) - e^{-ym}K\Phi(d_2)$, where

$d_1 = \frac{\ln(S/K)+(y+\sigma^2/2)m}{\sigma\sqrt{m}}$ and $d_2 = d_1 - \sigma\sqrt{m}$

and where $\Phi(d)$ denotes the probability of $x \leq d$ when x has an $N(0,1)$ distribution. All variables except the volatility (σ) are directly observable.

A Remark on the Code

- `cdf(Normal(0,1),x)` defines the cdf of a N(0, 1) distribution, using the `Distributions.jl` package.

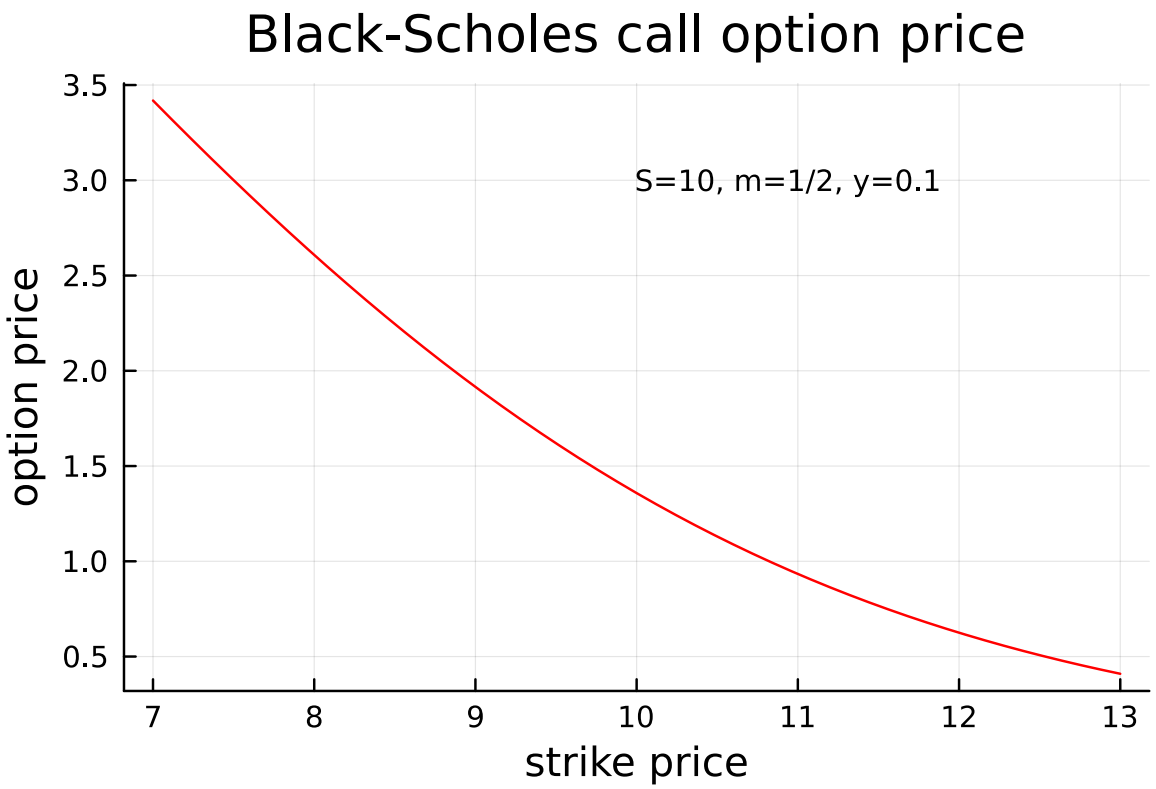
```
"""
Pr(z<=x) for N(0,1)
"""
Φ(x) = cdf(Normal(0,1),x)     #a one-line function, to get standard notation


"""
Calculate Black-Scholes european call option price
"""
function OptionBlackSPs(S,K,m,y,σ)

  d1 = ( log(S/K) + (y+1/2*σ^2)*m ) / (σ*sqrt(m))
  d2 = d1 - σ*sqrt(m)
  c  = S*Φ(d1) - exp(-y*m)*K*Φ(d2)
  return c
end
```

```
OptionBlackSPs
```

```
σ = 0.4
c1 = OptionBlackSPs(10,10,0.5,0.1,σ)         #call price for a single K value
println("\n","call price according to Black-Scholes for a single K value: ",c1)

K = range(7,13,length=51)                    #call prices for several K values
c = OptionBlackSPs.(10,K,0.5,0.1,σ);
```

```
call price according to Black-Scholes for a single K value:      1.358
```

```
p1 = plot( K,c,
           color = :red,
           legend = false,
           title = "Black-Scholes call option price",
           xlabel = "strike price",
           ylabel = "option price",
           annotation = (10,3,text("S=10, m=1/2, y=0.1",8,:left)) )
display(p1)
```



Implied Volatility

...is the σ value that makes the Black-Scholes equation give the same option price as observed on the market. It is often interpreted as the "market uncertainty."

The next cell uses the call option price calculated above as the market price. The implied volatility should then equal the volatility used above (this is a way to check your coding).

The subsequent cells instead use some data on options on German government bonds and calculate the σ for each of them.'

A Remark on the Code

- the `find_zero()` function from the `Roots.jl` package solve a non-linear function for its root (in terms of one of the inputs of the function). In the cell below, we thus solve for the σ value that makes `OptionBlackSPs(10,10,0.5,0.1,σ)-c1` equal to zero.

```
                                #solve for implied vol
iv = find_zero(σ->OptionBlackSPs(10,10,0.5,0.1,σ)-c1,(0.00001,5))

println("Implied volatility: ",iv,", compare with: $σ")
```

```
Implied volatility:      0.400, compare with: 0.4
```

```
#  LIFFE Bunds option data, trade date April 6, 1994
K = [                          #strike prices
      92.00;  94.00;  94.50;  95.00;  95.50;  96.00;  96.50;  97.00;
      97.50;  98.00;  98.50;  99.00;  99.50;  100.0;  100.5;  101.0;
     101.5;  102.0;  102.5;  103.0;  103.5 ];
C = [                          #call prices
      5.13;    3.25;    2.83;    2.40;    2.00;    1.64;    1.31;    1.02;
      0.770;   0.570;   0.400;   0.280;   0.190;   0.130;  0.0800;  0.0500;
      0.0400;  0.0300;  0.0200;  0.0100;  0.0100 ];
S = 97.05                  #spot price
m = 48/365                 #time to expiry in years
y = 0.0                    #Interest rate: LIFFE⇒no discounting
N = length(K)
```

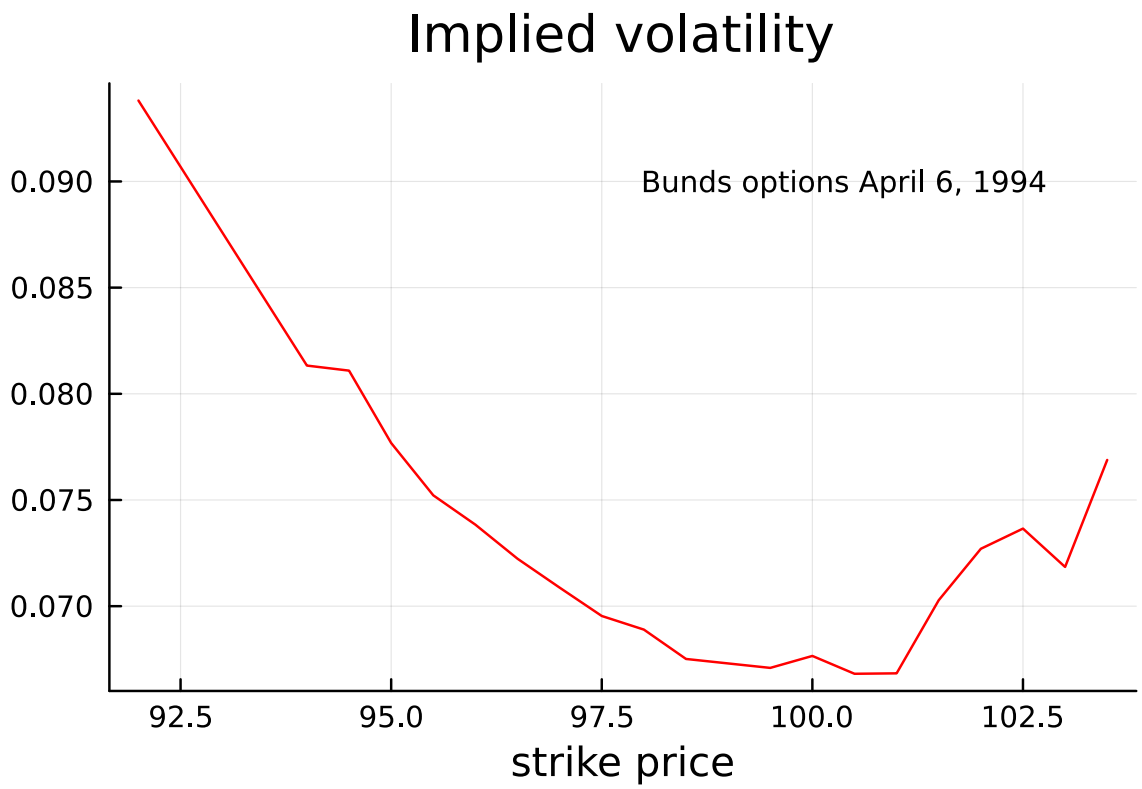
```
21
```

```
iv = fill(NaN,N)       #looping over strikes
for i in 1:N
  iv[i] = find_zero(sigma->OptionBlackSPs(S,K[i],m,y,sigma)-C[i],(0.00001,5))
end

printblue("From Bunds options data:\n")
printmat(K,iv,colNames=["strike","σ"])
```

```
From Bunds options data:

    strike         σ
    92.000     0.094
    94.000     0.081
    94.500     0.081
    95.000     0.078
    95.500     0.075
    96.000     0.074
```

```
 96.500     0.072
 97.000     0.071
 97.500     0.070
 98.000     0.069
 98.500     0.068
 99.000     0.067
 99.500     0.067
100.000     0.068
100.500     0.067
101.000     0.067
101.500     0.070
102.000     0.073
102.500     0.074
103.000     0.072
103.500     0.077
```

```
p1 = plot( K,iv,
           color = :red,
           legend =false,
           title = "Implied volatility",
           xlabel = "strike price",
           annotation = (98,0.09,text("Bunds options April 6, 1994",8,:left)) )
display(p1)
```



Mean-Variance Frontier

Given a vector of average returns (μ) and a variance-covariance matrix (Σ), the mean-variance frontier shows the lowest possible portfolio uncertainty for a given expected portfolio return (denoted μ^x below).

It is thus the solution to a quadratic minimization problem. The cells below will use matrix formulas for this solution, but we often have to resort to numerical methods (especially when there are portfolio restrictions).

The mean-variance frontier is typically plotted with the portfolio standard deviation (*not* the variance) on the horizontal axis and the portfolio expected return μstar on the vertical axis.

We calculate and plot two different mean-variance frontiers: *(1)* when we only consider risky assets; *(2)* when we also consider a risk-free asset.

A Remark on the Code

The calculations for the mean-variance frontiers are in two functions (for the case of only risky assets, and for risky and a riskfree asset). The calculations involve solving a system of linear equations (or calculating a matrix inverse) for each value of the expected return μ^x. To speed up the calculations we use the `factorize()` function from the `LinearAlgebra` standard library.

```
μ = [11.5, 9.5, 6]/100          #expected returns
Σ  = [166  34  58;              #covariance matrix
       34  64   4;
       58   4 100]/100^2
Rf = 0.03                       #riskfree return (an interest rate)

println("μ: ")
printmat(μ)
println("Σ: ")
printmat(Σ)
println("Rf: ")
printmat(Rf)
```

```
μ:
     0.115
     0.095
     0.060

Σ:
     0.017     0.003     0.006
     0.003     0.006     0.000
     0.006     0.000     0.010

Rf:
     0.030
```

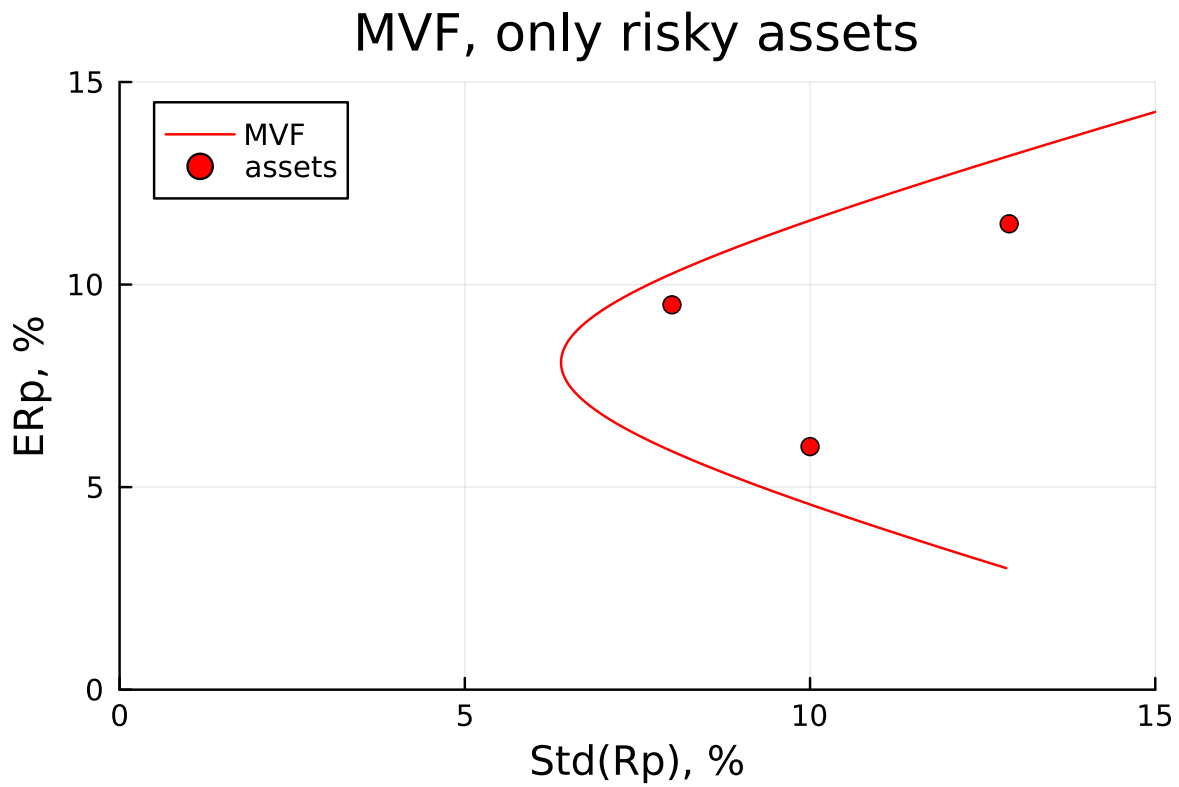
```
"""
    MVCalc(μˣ,μ,Σ)

Calculate the std and weights of a portfolio (with mean return μˣ) on MVF of risky assets.

# Input
- `μˣ::Vector`:    K vector, mean returns to calculate results for
- `μ::Vector`:     n vector, mean returns
- `Σ::Matrix`:     nxn, covariance matrix of returns, can contain riskfree assets
```

```
# Output
- `StdRp::Vector`:    K vector, standard deviation of mean-variance portfolio (risky only) wit
- `w_p::Matrix`:      Kxn, portfolio weights of       ""

"""
function MVCalc(μˣ,μ,Σ) #the std of a portfolio on MVF of risky assets
    (K,n) = (length(μˣ),length(μ))
    A = [Σ        μ ones(n);   #A is just a name of this matrix, it's not asset A
         μ'       0 0;
         ones(n)' 0 0];
    Af = factorize(A)          #factorize so we can solve wλδ_i repeatedly (and fast)
    (w,StdRp) = (fill(NaN,K,n),fill(NaN,K))
    for i in 1:K                #loop over μˣ[i]
        wλδ_i    = Af\[zeros(n);μˣ[i];1]
        w_i      = wλδ_i[1:n]
        StdRp[i] = sqrt(w_i'Σ*w_i)
        w[i,:]   = w_i
    end
    return StdRp,w
end


"""
Calculate the std of a portfolio (with mean μˣ) on MVF of (Risky,Riskfree)
"""
function MVCalcRf(μˣ,μ,Σ,Rf)
    (K,n) = (length(μˣ),length(μ))
    μᵉ = μ .- Rf
    A  = [Σ   μᵉ;
          μᵉ' 0]
    Af = factorize(A)
    (w,StdRp) = (fill(NaN,K,n),fill(NaN,K))
    for i in 1:K
      wλ_i     = Af\[zeros(n);(μˣ[i]-Rf)]
      w_i      = wλ_i[1:n]
      StdRp[i] = sqrt(w_i'Σ*w_i)
      w[i,:]   = w_i
    end
    return StdRp,w
end
```

```
MVCalcRf
```

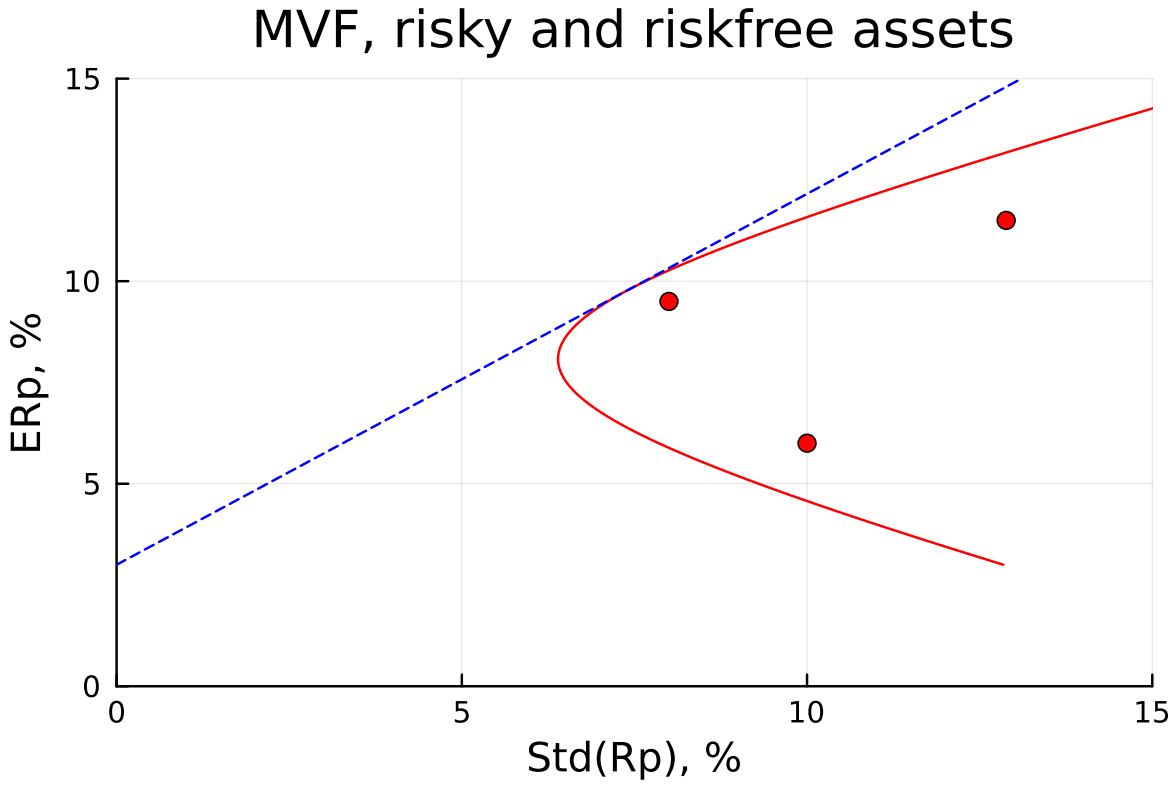
```julia
μˣ = range(Rf,0.15,length=201)
L     = length(μˣ)

StdRp = MVCalc(μˣ,μ,Σ)[1];    #risky assets only, [1] to get the first output
```

```julia
p1 = plot( StdRp*100,μˣ*100,
           linecolor = :red,
           xlim = (0,15),
           ylim = (0,15),
           label = "MVF",
           legend = :topleft,
           title = "MVF, only risky assets",
           xlabel = "Std(Rp), %",
           ylabel = "ERp, %" )
scatter!(sqrt.(diag(Σ))*100,μ*100,color=:red,label="assets")
display(p1)
```



```
StdRpRf = MVCalcRf(μˣ,μ,Σ,Rf)[1];          #with riskfree too
```

```
p1 = plot( [StdRp StdRpRf]*100,[μˣ μˣ]*100,
           legend = nothing,
           linestyle = [:solid :dash],
           linecolor = [:red :blue],
           xlim = (0,15),
           ylim = (0,15),
           title = "MVF, risky and riskfree assets",
           xlabel = "Std(Rp), %",
           ylabel = "ERp, %" )
scatter!(sqrt.(diag(Σ))*100,μ*100,color=:red)
display(p1)
```



Arrays

This notebook illustrates how to create and reshuffle arrays. Other notebooks focus on matrix algebra and functions applied to arrays.

Load Packages and Extra Functions

```
using Printf, DelimitedFiles

include("src/printmat.jl");
```

Scalars, Vectors and Multi-dimensional Arrays

are treated as different things in Julia, even if they happen to "look" similar. For instance, a 1×1 array is not a scalar and an $n \times 1$ array is not a vector. This is discussed in some detail further down.

However, we first present some common features of all arrays (vectors or multi-dimensional arrays).

Creating Arrays

The typical ways of getting an array are

- hard coding the contents
- reading in data from a file
- as a result from computations
- allocating the array and then changing the elements
- (often not so smart) growing the array by adding rows (or columns,..)

- by list comprehension

The next few cells give simple examples.

1. Hard Coding the Contents or Reading from a File

```
z = [11 12;                       #typing in your matrix
     21 22]
printblue("A matrix that we typed in:")
printmat(z)

x = readdlm("Data/MyData.csv",',',skipstart=1)  #read matrix from file
printblue("First four lines of x from csv file:")
printmat(x[1:4,:])

#to create a vector: [1,2] or [1;2]
#to create a 2x3x2 array:  [1 2 3;4 5 6;;;11 12 13;14 15 16]
```

```
A matrix that we typed in:
    11        12
    21        22

First four lines of x from csv file:
197901.000     4.180     0.770    10.960
197902.000    -3.410     0.730    -2.090
197903.000     5.750     0.810    11.710
197904.000     0.050     0.800     3.270
```

2a. Allocating an Array and Then Changing the Elements: Fill

An easy way to create an array is to use the `fill()` function.

```
A = fill(0,10,2)             #10x2, integers (0)
B = fill(0.0,10)             #vector with 10 elements, floats (0.0)
C = fill(NaN,10,2)           #10x2, floats (NaN)
D = fill("",3)               #vector with 3 elements, strings ("")
E = fill(Date(1),3)          #vector with 3 elements, dates (0001-01-01)
```

In contrast, do *not* use `fill([1,2],7)`, since all 7 arrays will refer to the same underlying array (changing one changes all). Instead, use a comprehension (see below).

```
x = fill(0.0,3,2)     #creates a 3x2 matrix filled with 0.0
printblue("so far, x is filled with 0.0. For instance, x[1,1] is $(x[1,1])")

for i in 1:size(x,1), j in 1:size(x,2)
    x[i,j] = i/j
end

printblue("\nx after some computations:")
printmat(x)
```

```
so far, x is filled with 0.0. For instance, x[1,1] is 0.0

x after some computations:
     1.000     0.500
     2.000     1.000
     3.000     1.500
```

2b. Allocating an Array and Then Changing the Elements: A More General Approach (extra)

You can also create an array by

```
A = Array{Int}(undef,10,2)       #10x2, integers
F = Array{Any}(undef,3)          #vector with 3 elements, can include anything
```

The `undef` signals that the matrix is yet not initialized. This is more cumbersome than `fill()`, but sometimes more flexible.

```
F    = Array{Any}(undef,3)
F[1] = [1;2;3;4]             #F[1] contains a vector
F[2] = "Sultans of Swing"    #F[2] a string
F[3] = 1978                  #F[3] an integer

printmat(F)
```

```
[1, 2, 3, 4]
Sultans of Swing
  1978
```

3. Growing an Array

Growing an array (vector, matrix or higher-dimensional array) is done by `[A;B]`, `[A;;B]` (which is the same as `[A B]`), or `[A;;;B]`, etc. Alternatively, use the `vcat`, `hcat` and `cat` functions.

There are special (faster) functions for growing a *vector* (not a matrix):

```
push!(old vector,new_element_1,new element_2)       #or pushfirst!()
```

If you instead want to append all elements of a vector, then do

```
append!(old vector,vector1_to_append,vector2_to_append)     #or prepend!()
```

```
A = [11 12;
     21 22]
B = [1 2;
     0 10]

z = [A;B]                #same as vcat(A,B)
printblue("\n","stacking A and B vertically")
printmat(z)

z2 = [A B]                 #same as hcat(A,B)
printblue("\n","stacking A and B horizontally")
printmat(z2)
```

```
stacking A and B vertically
    11        12
    21        22
     1         2
     0        10


stacking A and B horizontally
    11        12         1         2
    21        22         0        10
```

```
B = Float64[]                  #empty vector, to include floats
for i = 1:3
    x_i = 2.0 + 10^i
    push!(B,x_i)               #adding an element at the end
end
printblue("a vector with 3 elements:")
printmat(B)
```

```
a vector with 3 elements:
    12.000
   102.000
  1002.000
```

Performance Tips (extra)

Avoid growing a matrix/higher dimensional array ([],hcat, vcat, cat) in long loops since it is slow. Instead, pre-allocate and *change* the elements in the loop.

However, growing a *vector* with push or append is not that slow.

4. List Comprehension and map (extra)

List comprehension is a simple way of creating an array from repeated calculations. It is similar to the combination of pre-allocation and a "for loop."

(You can achieve the same thing with map (for instance, by map(i→collect(1:i),1:3)).)

```
A = [collect(1:i) for i=1:3]         #this creates a vector of vectors

printblue("A[1] is vector with 1 element, A[2] a vector with 2 elements,...")
printmat(A)
```

```
A[1] is vector with 1 element, A[2] a vector with 2 elements,...
       [1]
    [1, 2]
 [1, 2, 3]
```

Using Parts of a Matrix

The most common way to use parts of an array is by indexing. For instance, to use the second column of `A`, do `A[:,2]`.

Notice that `x = A[1,:]` gives a (column) vector (yes, it does), while `z = A[1:1,:]` gives a $1 \times k$ matrix.

Also notice that `z = A[1,:]` creates an independent copy, so changing (elements of) `z` will *not* change `A`.

A shortcut to loop over all rows of `A` is `for i in eachrow(A)`. There is also `eachcol()`.

```
A = [11 12;
     21 22]
printblue("A:")
printmat(A)

printblue("\nsecond column of A:")
printmat(A[:,2])

printblue("\n","first row of A (as a vector): ")
printmat(A[1,:])                          #notice 1 makes it a vector

printblue("\n","first row of A (as a 1x2 matrix): ")
printmat(A[1:1,:])                        #use 1:1 to keep it as a 1x2 matrix

for i in eachrow(A)          #looping over all rows
    printblue("another row: ")
    printmat(i)
end
```

```
A:
    11        12
    21        22


second column of A:
    12
    22


first row of A (as a vector):
```

```
    11
    12


first row of A (as a 1x2 matrix):
    11        12

another row:
    11
    12

another row:
    21
    22
```

Performance Tips (extra)

In case you do not need an independent copy, then `y = view(A,1,:)` creates a *view* of the first row of `A`. This saves memory and is sometimes faster. Notice, however, that changing `y` by `y .= [1,2]` will now change also the first row of `A`. Notice that the dot `.` is needed for this. (In contrast, `y = [1,2]` would create a new `y` and not affect `A`.)

To make a *copy or a view?* If you need to save memory: a view. Instead, if you need speed: try both. (Copies are often quicker when you need to do lots of computations on the matrix, for instance, in a linear regression.)

```
printblue("\n","view of first row of A (although it prints like a column vector): ")
y = view(A,1,:)
printmat(y)

y .= [1,2]                    #changing y and thus the first row of A, notice the dot (.)
printblue("A after changing y by y .= [1,2]")
printmat(A)
```

```
view of first row of A (although it prints like a column vector):
    11
    12

A after changing y by y .= [1,2]
     1         2
    21        22
```

Performance Tips (extra)

Avoid creating and destroying lots of arrays in loops: it takes time. If possible re-use the existing arrays instead. The next cell provides an illustration.

```
function fn1(N)
  for i = 1:N
    tmp = zeros(N,N)         #create a new tmp in each loop
    tmp[i,i] = i             #do something with tmp
  end
  return nothing
end

function fn2(N)
  tmp = zeros(N,N)
  for i = 1:N
    tmp .= 0.0             #re-use the existing tmp, reset to zeros
    tmp[i,i] = i           #do something with tmp
  end
  return nothing
end


using BenchmarkTools       #a package for benchmarking computations

@btime fn1(300)            #timing
@btime fn2(300)
```

```
  9.636 ms (900 allocations: 206.02 MiB)
  2.610 ms (3 allocations: 703.20 KiB)
```

Splitting up an Array (extra)

Sometimes you want to create new variables from the columns (or rows) of a matrix. The next cell shows an example.

```
printblue("A simple way...which works well when you want to create a few variables")
x1 = A[:,1]
x2 = A[:,2]      #or (x1,x2) = (A[:,1],A[:,2])
printmat(x2)
```

```
printblue("Another way")
(z1,z2) = [A[:,i] for i = 1:2]
printmat(z2)
```

```
A simple way...which works well when you want to create a few variables
     2
    22

Another way
     2
    22
```

Arrays vs. Vectors vs. Scalars

Matrices, vectors and scalars are different things in Julia, even if they contain the same number of elements. In particular,

(a) an $n \times 1$ matrix is not the same thing as an n-vector.

(b) a 1×1 matrix or a 1-element vector are not the same thing as a scalar.

As you will see further on, vectors are often more convenient than $n \times 1$ matrices.

To convert a 1-element vector or 1×1 matrix `C` to a scalar, just do `C[]` or `only(C)`

```
A = ones(3,1)          #a 3x1 matrix
B = ones(3)            #a vector with 3 elements

printblue("The sizes of matrix A and vector B: $(size(A)) $(size(B))")

printblue("\nTesting if A==B: ",isequal(A,B))

printblue("\nThe nx1 matrix A and n-element vector B can often be used together, for instance,
printmat(A+B)
```

```
The sizes of matrix A and vector B: (3, 1) (3,)

Testing if A==B: false

The nx1 matrix A and n-element vector B can often be used together, for instance, as in A+B, whose s
```

```
     2.000
     2.000
     2.000
```

```
C = ones(1,1)                    #a 1x1 matrix
c = 1                            #a scalar

printblue("\nc/C would give an error since C is a (1x1) matrix")

printblue("\nInstead, do c/only(C): ",c/only(C))     #or c/C[]
```

```
c/C would give an error since C is a (1x1) matrix

Instead, do c/only(C): 1.0
```

Vectors: x'x and x'A*x Create Scalars (if x is a vector)

If `x` is a vector and `A` a matrix, then `x'x` and `x'A*x` are scalars. This is what a linear algebra text book would teach you, so this is an example of when vectors are convenient. This is *not* true if `x` is a matrix of size $n \times 1$. In that case, the result is a 1×1 matrix.

Recommendation: use vectors (instead of $n \times 1$ matrices) if you can.

```
x = ones(2)                                #this is a vector
A = [11 12;
     21 22]
printblue("\nx'x and x'A*x when x is a 2 element vector: ",x'x," ",x'A*x)

x   = ones(2,1)                          #this is a 2x1 matrix (array)
printblue("\nx'x and x'A*x when x is a 2x1 array: ",x'x," ",x'A*x)
```

```
x'x and x'A*x when x is a 2 element vector: 2.0 66.0

x'x and x'A*x when x is a 2x1 array: [2.0;;] [66.0;;]
```

An Array of Arrays (extra)

If `x1` and `x2` are two arrays, then `y=[x1,x2]` is a vector (of arrays) where `y[1] = x1` and `y[2] = x2`.

In this case `y[1]` is actually a view of `x1` so changing elements of one changes the other.

(If you instead want to stack `x1` and `x2` into a single matrix, use `[x1 x2]`, `[x1;x2]` or one of the `cat` functions discussed above.)

```
x1 = ones(3,2)
x2 = [1;2]
y = [x1,x2]               #a vector of arrays

println(size(y))
printmat(y[1])
printmat(y[2])
```

```
(2,)
     1.000     1.000
     1.000     1.000
     1.000     1.000

     1
     2
```

Arrays are Different...

Vectors and matrices (arrays) can take lots of memory space, so **Julia is designed to avoid unnecessary copies of arrays**. In short, notice the following. Let `A` be and array and you do either of these:

- `B = A`, `B = reshape(A,n,m)`, `B = vec(A)`, or `B = A'`, and then followed by `B[1] = -999`
- `f1!(A)` where `f1!` is a function like

```
function f1!(B)
    B[1] = -999     #change some elements of B inside the function
  return B
end
```

- `B = [A,A]` (an array of arrays) followed by `B[1][1] = -999`

then also `A` will change.

Notice that in all cases you are changing some *elements* of `B`, not redefining the entire `B` (like in `B = [1,2,3]`). Other ways to change some *elements* are `B[:] = [1,2]` or `B .= [1,2]` so the same behaviour applies to those cases.

If you do not like this behaviour, then use `copy(A)` to create an independent copy of the array.

```
function f1!(B)            #! is a convention for indicating that the function changes the inp
    B[1] = -999            #changing ELEMENTS of B, affects outside value
    #B = B/2               #this would NOT affect the outside value
  return B
end

A  = [1.0,2.0]
printblue("original A:")
printmat(A)

C = f1!(A)
printblue("A after calling f1!(A): ")
printmat(A)
```

```
original A:
     1.000
     2.000

A after calling f1!(A):
  -999.000
     2.000
```

Basic Matrix Algebra

This notebook presents some basic linear algebra and some matrix reshuffling/computations often needed in finance and econometrics.

```
using Printf, LinearAlgebra

include("src/printmat.jl");
```

Adding and Multiplying: An Array and a Scalar

With an array A (for instance, a vector or a matrix, but also higher-dimensional arrays) and a scalar c, do

1. `A*c` (textbook: Ac) to multiply each element of A by c
2. `A .+ c` (textbook: $A + cJ$, where J is an array of ones) to add c to each element of A, and similarly `A .- c` (textbook: $A - cJ$). Notice the dot.

Watch out when the number comes first: `2.+A` is not allowed since it is ambiguous. However, `2.0.+A` and `2 .+ A` both work.

```
A = [1 3;3 4]
c = 10

printblue("A:")
printmat(A)
printblue("c:")
printmat(c)

printblue("A*c:")
printmat(A*c)
```

```
printblue("A .+ c:")
printmat(A .+ c)          #notice the dot in .+
```

```
A:
     1          3
     3          4

c:
    10

A*c:
    10         30
    30         40

A .+ c:
    11         13
    13         14
```

Adding and Multiplying Two Arrays

With two arrays of the same dimensions (A and B), do

`A+B` (textbook: $A + B$) to add them (element by element), and similarly `A-B` (textbook: $A - B$).

To multiply arrays (A and B) of conformable dimensions: `A*B` (textbook: AB).

```
A = [1 3;3 4]               #A and B are 2x2 matrices
B = [1 2;3 -2]
printblue("A:")
printmat(A)
printblue("B:")
printmat(B)

printblue("A+B:")
printmat(A+B)

printblue("A*B:")
printmat(A*B)
```

```
A:
     1          3
```

```
     3          4

B:
     1          2
     3         -2

A+B:
     2          5
     6          2

A*B:
    10         -4
    15         -2
```

Transpose

You can transpose a numerical vector/matrix `A` by `A'`.

Notice that (in Julia) `A` and `B = A'` share the same elements (changing one changes the other). If you want an independent copy, use `B = permutedims(A)` or perhaps `B = copy(A')`.

For an array of other elements (for instance, strings), use `permutedims(A)` to swap the dimensions.

```
A = [1 2 3;4 5 6]
printblue("A: ")
printmat(A)
printblue("A': ")
printmat(A')
```

```
A:
     1          2          3
     4          5          6

A':
     1          4
     2          5
     3          6
```

Vectors: Inner and Outer Products

There are several different ways to think about a vector in applied mathematics: as a $K \times 1$ matrix (a column vector), a $1 \times K$ matrix (a row vector) or just a flat K vector. Julia uses flat vectors but they are mostly interchangable with column vectors.

The inner product of two (column) vectors with K elements is calculated as `x'z` or `dot(x,y)` (textbook: $x'z$ or $x \cdot z$) to get a scalar. (The dot is obtained by `\cdot + TAB`, but this is sometimes hard to distinguish from or things like `x.z`.)

In contrast, the outer product of two (column) vectors with K elements is calculated as `x*z'` (textbook: xz') to get a $K \times K$ matrix.

```
x = [10,11]                 #[10;11] gives the same
z = [2,5]
printblue("x and z")
printmat([x z])

printblue("x'z: ")
println(x'z)                #dot(x,z) gives the same

printblue("x*z':")
printmat(x*z')
```

```
x and z
    10          2
    11          5

x'z:
    75
x*z':
    20         50
    22         55
```

Vectors: Quadratic Forms

A quadratic form (A is an $n \times n$ matrix and x is an n vector): `x'A*x` (textbook: $x'Ax$) to get a scalar. There is also the form `dot(x,A,x)`.

```
A = [1 3;3 4]
x = [10,11]

printblue("x:")
printmat(x)
printblue("A:")
printmat(A)

printblue("x'A*x: ")
println(x'A*x)          #or dot(x,A,x)
```

```
x:
    10
    11

A:
     1          3
     3          4

x'A*x:
  1244
```

Kronecker Product

The Kronecker product of two matrices

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

is calculated by using `kron()`

```
A = [1 3;2 4]
B = [10 11]
z = kron(A,B)

printblue("A:")
printmat(A)
printblue("B:")
printmat(B)
```

```
printblue("kron(A,B):")
printmat(z)
```

```
A:
     1          3
     2          4

B:
    10         11

kron(A,B):
    10         11         30         33
    20         22         40         44
```

Matrix Inverse

A matrix inverse of an $n \times n$ matrix A:

`inv(A)` or `A^(-1)` (textbook: A^{-1})

The inverse is such that $AA^{-1} = I$ and $A^{-1}A = I$, where I is the identity matrix (ones along the main diagonal, zeros elsewhere).

```
A = [1 3;3 4]
printblue("A:")
printmat(A)

printblue("inv(A):")
printmat(inv(A))

printblue("inv(A)*A:")
printmat(inv(A)*A)
```

```
A:
     1          3
     3          4

inv(A):
    -0.800      0.600
     0.600     -0.200
```

```
inv(A)*A:
     1.000    -0.000
     0.000     1.000
```

Solving Systems of Linear Equations

If you have the system $Ab = y$, where A is a matrix and b and y are vectors, then we can solve as `b = inv(A)*y` or `b = A\y`. The latter is typically more robust.

```
y = [10,11]

b1 = inv(A)*y
b2 = A\y

printmat(b1,b2,colNames=["sol 1","sol 2"])
```

```
     sol 1     sol 2
    -1.400    -1.400
     3.800     3.800
```

The Identity Matrix

The identity matrix I_n can often be represented by `I` and then Julia will compare with the surrounding code to create the right dimension. For instance, if `A` is a square matrix, then `I + A` and `[A I]` both work. Notice that you need to do `using LinearAlgebra` before `I` works.

If you still need to specify the dimension, then use `1I(3)` or `1I[1:3,1:3]`.

You can create a basis vector as `1I[1:3,2]`. This 3-vector is filled with zeros, except that element 2 is 1.

```
printblue("I + A")
printmat(I + A)

printblue("1I[1:3,1:3]")
printmat(1I[1:3,1:3])

printblue("1I[1:3,2]")
printmat(1I[1:3,2])
```

```
I + A
     2          3
     3          5

1I[1:3,1:3]
     1          0          0
     0          1          0
     0          0          1

1I[1:3,2]
     0
     1
     0
```

Vectors: Extracting Vectors from Matrices

Notice that `A[1,:]` and `A[:,1]` both give flat vectors. In case you want a row vector use `A[1:1,:]`.

```
A[1,:]
```

```
2-element Vector{Int64}:
 1
 3
```

Functions of Arrays

This notebook illustrates how to apply a function to arrays and how to iterate over array elements. How to create/reshuffle arrays and matrix algebra are covered in other notebooks.

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");   #a function for prettier matrix printing
```

Elementwise Functions of Arrays: The Dot (.)

Let `X` be an array, and `a` and `b` be scalars. Then, `y = fn.(X,a,b)` generates an array `y` where `y[i,j] = fn(X[i,j],a,b)`

(You would achieve the same thing with `map(xi->fn(xi,a,b),x)`.)

```
fn(x,a,b) = a/x + b             #x has to be a scalar for this to work

X = [1 2;
     0 10]

printmat(fn.(X,100,10))         #notice the dot (.)
```

```
   110.000    60.000
       Inf    20.000
```

Looping over Elements of an Array

There are several ways of looping over all elements in an array. The next cell summarises some of them. We cover `for z in x`, `for i in 1:length(x)` and `for i in eachindex(x)`. The `enumerate()` approach is discussed further below.

```
x = [11 12; 21 22]

printblue("elements of x")
for z in x                      #when we do not need the index
    println(z)
end
println()

y = similar(x)
for i in 1:length(x)            #when we want to use the index i for an output
    y[i] = x[i] + i*100         #works only with traditional arrays (first index is 1)
end                             #`for i = 1:length(x)` is the same
printblue("y:")
printmat(y)

y = similar(x)
for i in eachindex(x)           #again when we want to use the index i for an output
    y[i] = x[i] + i*100         #more general and would work with non-traditional arrays
end
printblue("y calculated in another way:")
printmat(y)
```

```
elements of x
11
21
12
22

y:
   111       312
   221       422

y calculated in another way:
   111       312
   221       422
```

Looping over Columns or Rows

Suppose you want to calculate the sum of each row of a matrix. The classical way of doing that is to loop and extract each row as `X[i,:]`

But, there is also a direct way to loop over all rows by using `for r in eachrow(X)`. To also get the row number `i`, we use `for (i,row) in enumerate(eachrow(X))`. For looping over columns, use `eachcol()`.

To change `X` in such a loop, use `row .= new_vector`. Notice the dot in the assignment (`.=`).

```
X = [1 2;
     0 10]
printblue("X:")
printmat(X)

m = size(X,1)

z = zeros(Int,m)               #to fill with results, zeros(Int,m) to get Int
for i in 1:m                   #traditional loop over rows
    z[i] = sum(X[i,:])         #or sum(view(X,i,:)) to save memory
end
printblue("sum of each row:")
printmat(z)

z = zeros(Int,m)
for (i,row) in enumerate(eachrow(X))   #enumerate to get both index and value
    z[i] = sum(row)
end
printblue("sum of each row:")
printmat(z)

Xb = copy(X)
for (i,row) in enumerate(eachrow(Xb))   #enumerate to get both index and value
    row .= [i*10+1,i*10+2]              #need .= to change Xb
end
printblue("changing rows by using `eachrow()` ")
printmat(Xb)
```

```
X:
     1         2
     0        10
```

```
sum of each row:
     3
    10

sum of each row:
     3
    10

changing rows by using `eachrow()`
    11        12
    21        22
```

Functions with Built-in `dims` Argument

May functions have a `dims` argument that allows you to avoid looping, for instance, `sum`.

```
printmat(sum(X,dims=2))
```

```
     3
    10
```

Functions with Inside Functions (Predicates)

Several functions allow you to also apply an elementwise function to `X` before doing the rest of the calculations, for instance, `any`, `all`, `sum`, `prod`, `maximum`, and `minimum`. This speeds up things since it avoids creating intermediate/temporary arrays.

```
sum(abs2,X,dims=2)           #same as sum(abs2.(X),dims=2) but faster
```

```
2×1 Matrix{Int64}:
   5
 100
```

Apply Your Own Function on Each Column: mapslices (extra)

...or each row (or some other dimension)

The `mapslices(fun,x,dims=1)` applies `fun(x[:,i])` to each column of a matrix `x`. This is a flexible alternative to looping over the columns, since it easily allows you to change the dimension over which too loop.

The cell below illustrates this by calling a function which calculates the moving average of `x[t]` and `x[t-1]` for each column (or row) of a matrix `X`.

```
function MovingAvg2(x)              #moving average of t and t-1
    T = length(x)
    y = fill(NaN,T)
    for t = 2:T
        y[t] = (x[t] + x[t-1])/2
    end
    return y
end

X = [1:5 101:105]
Y = mapslices(MovingAvg2,X,dims=1)          #could change to dims=2
printblue("Y and MA(1) of X:")
printmat([X Y],colNames=["x₁","x₂","MA(x₁)","MA(x₂)"])
```

```
Y and MA(1) of X:
        x₁         x₂     MA(x₁)     MA(x₂)
     1.000    101.000        NaN        NaN
     2.000    102.000      1.500    101.500
     3.000    103.000      2.500    102.500
     4.000    104.000      3.500    103.500
     5.000    105.000      4.500    104.500
```

Types

This notebook provides *(a)* a brief introduction to types (for instance, integers, bools and strings), *(b)* discussion of how to test types and *(c)* also how to convert from one type to another.

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");
```

Some Important Types

Julia has many different types of variables: signed integers (like 2 or -5), floating point numbers (2.0 and -5.1), bools (false/true), bitarrays (similar to bools, but with more efficient use of memory), strings ("hello"), Dates (2017-04-23) and many more.

The numerical types also come with subtypes for different precisions, for instance, Float16, Float32 and Float64. Unless you specify otherwise, code like

```
a = 2
b = 2.0
```

gives an Int64 and a Float64 respectively (at least on the 64 bit version of Julia).

Integers and Floats

```
a = 2                   #integer, Int (Int64 on most machines)
b = 2.0                 #floating point, (Float64 on most machines)
A = [1,2]
B = [1.0,2.0]

println("a: ",typeof(a))
println(a)

println("\nb: ",typeof(b))
println(b)

println("\nA: ",typeof(A))
printmat(A)

println("B: ",typeof(B))
printmat(B)
```

```
a: Int64
2

b: Float64
2.0

A: Vector{Int64}
     1
     2

B: Vector{Float64}
     1.000
     2.000
```

Why Use Int When There Are Floats?

That is, why bother with sometimes using 3 when you could use 3.0 everywhere? Mostly because you cannot use 3.0 everywhere. For instance, you cannot pick out element `x[3.0]` from a vector. It has to be `x[3]`.

(In contrast, `1.0 + 2.0` will be exactly `3.0` in Julia, without a floating point rounding error, so that is not a main concern. For instance, try `1.0 + 2.0 == 3`)

```
x = [1,10,100,1000]

#println(x[3.0])    #uncomment and run. Will give an error

println(x[3])
```

```
100
```

Bools and BitArrays

```
c = 2 > 1.1                    #Bool
println("c: ",typeof(c))
println(c)

C = A .> 1.5                   #BitArray (here a BitVector)
println("\nC: ",typeof(C))
printmat(C)

println("A BitArray is a more economical array version of Bool, but prints as 0/1 by printmat.
        "Notice that typeof(C[1]) gives: ",typeof(C[1]))
```

```
c: Bool
true

C: BitVector
     0
     1

A BitArray is a more economical array version of Bool, but prints as 0/1 by printmat.
Notice that typeof(C[1]) gives: Bool
```

Char and Strings

```
t = 'a'                                    #Char, just one letter
println(typeof(t))

txt = "Dogs are nicer than cats."          #String, could be a long novel
println(typeof(txt))
```

```
Char
String
```

Calculations with Mixed Types and Converting Types

A calculation like "integer" + "float" works automatically. The type of the result is a float (the more flexible type). Similarly, "Bool" + "Int" will give an integer.

There are also direct ways of converting a variable from one type to another using the `convert()` function.

Some Calculations with Mixed Types ("promotion")

```
println("Int + Float64: ",1+2.0)
println("Bool + Int: ",(1 > 0) + 2)
```

```
Int + Float64: 3.0
Bool + Int: 3
```

Converting from Int to Float and Vice Versa

```
x = [1.1;10.1;100.1]
println("x: ",typeof(x))
printmat(x)

B_to_Int = round.(Int,x)                    #Float64 -> Int by rounding
println("rounding x to Int: ",typeof(B_to_Int))
printmat(B_to_Int)

A = [1;2]
println("A: ",typeof(A))
printmat(A)

A_to_Float64 = convert.(Float64,A)           #Int -> Float64
println("after converting A to Float64: ",typeof(A_to_Float64))
printmat(A_to_Float64)                       #Float64.(A) also works
```

```
x: Vector{Float64}
     1.100
    10.100
   100.100

rounding x to Int: Vector{Int64}
     1
    10
   100

A: Vector{Int64}
     1
     2

after converting A to Float64: Vector{Float64}
     1.000
     2.000
```

Converting from Bools and BitArrays to Int and Vice Versa

```
C = A .> 1.5
C_to_Int = convert.(Int,C)                #BitArray --> Int
println(typeof(C_to_Int))                 #Int.(C) also works
printmat(C_to_Int)

D = [1;0;1]
D_to_Bool = convert.(Bool,D)               #Int --> BitArray
println(typeof(D_to_Bool))                 #Bool.(D) also works
printmat(D_to_Bool)
```

```
Vector{Int64}
     0
     1

BitVector
     1
     0
     1
```

From Bools and BitArrays to Int: A Tricky Case (extra)

`false` is a "strong zero" in the sense that `false*NaN == 0` and `false*Inf == 0`.

If you do not want that behaviour in your code, transform `false` to 0 and then multiply.

```
println(false*NaN)
println(false*Inf)

println(convert(Int,false)*NaN)
println(convert(Int,false)*Inf)
```

```
0.0
0.0
NaN
NaN
```

Testing the Type

The perhaps easiest way to test the type is by using the `isa(variable,Type)` function. The type can be a union of other types (see below for an example).

Notice that an array has the type `Array`; more specifically `Array{Float64}` if it is an array with Float64 numbers.

```
x = 1.2
z = [1.2,1.3]

println("$x is a Number: ",isa(x,Number))
println("$x is an Int: ",isa(x,Int))
println("$x is an Int or a Float64: ",isa(x,Union{Int,Float64}))

println("$z is a Float64: ",isa(z,Float64))
println("$z is an Array: ",isa(z,Array))
```

```
1.2 is a Number: true
1.2 is an Int: false
1.2 is an Int or a Float64: true
[1.2, 1.3] is a Float64: false
[1.2, 1.3] is an Array: true
```

Performance Tips (extra)

Your code will often run faster if your variables do not change type in the computations (that is, if they are "type stable"). For instance, do something like this

```
x = 0.0                  #better than using x = 0 here
x = x + 0.1
```

If you want to initialise an array of zeros with the same type and size as `z`, use `zero(z)`. This is particularly useful inside function where the input `z` might sometimes contain floats and other times integers, etc. To get another size than `z`, do eg. `zeros(eltype(z),3,2)`. There are corresponding `one()` and `ones()` functions. Instead, if you just need a array of the same type as `z`, use `similar(z)`. (It will be filled with garbage.)

Days and Dates

This notebook *(a)* introduces the `Date` type; *(b)* discusses how to convert from other date formats (for instance, Excel and Matlab) to Julia dates; *(c)* and how to do date arithmetics.

Load Packages and Extra Functions

```
using Printf, Dates

include("src/printmat.jl");
```

Building a Calendar

```
dNb = Date(2014,1,31):Month(1):Date(2014,12,31)    #build a monthly calendar

printmat([dNb dayofweek.(dNb)];colNames=["date","day of week"],width=13)
```

```
         date  day of week
   2014-01-31            5
   2014-02-28            5
   2014-03-31            1
   2014-04-30            3
   2014-05-31            6
   2014-06-30            1
   2014-07-31            4
   2014-08-31            7
   2014-09-30            2
   2014-10-31            5
   2014-11-30            7
   2014-12-31            3
```

Converting from Other Date Formats

Converting from yyyymmdd

Background: financial data is often downloaded as CSV files (eg. from finance.yahoo), where the date may look like 20160331. The next cell shows a simple way to create a Julia Date.

```
csvDate = [20160331;20160401]                #two dates

jlDate = Date.(string.(csvDate),"yyyymmdd")  #convert to string and then Julia Date

printmat(csvDate,jlDate;colNames=["Original date","Julia date"],width=13)
```

```
Original date   Julia date
 20160331       2016-03-31
 20160401       2016-04-01
```

Converting from DateTime (Excel) to Date

Background: importing xls sheets with XLSX.jl gives an `Any` vector, if the sheet contains a column with dates. If needed, this can be converted to a vector of Dates.

```
xlsDate = Any[Date(2016,3,31);Date(2016,4,1)]     #to be converted

jlDate  = Date.(xlsDate)                          #convert to vector of Dates
println(typeof(xlsDate),"\n",typeof(jlDate))
```

```
Vector{Any}
Vector{Date}
```

Converting from Matlab's datenum

Background: in Matlab `datenum(2016,3,31)` gives 736420.0. In contrast, in Julia (which follows the ISO 8601 standard), `Dates.value(Date(2016,3,31))` gives 736054, which is 366 less, and it is an integer. A conversion is therefore required.

```
dNml = [736420.0;736421.0]            #to be converted, 2016-03-31;2016-04-01

jlDate = Date.(rata2datetime.(round.(Int,dNml) .- 366))  #to Julia Date from matlab datenum

#mlDate = Dates.datetime2rata.(jlDate)[:,:] .+ 366.0     #to matlab datenum from Julia Date

printmat([dNml jlDate];colNames=["Matlab","Julia"],width=13)
```

```
      Matlab        Julia
  736420.000   2016-03-31
  736421.000   2016-04-01
```

Time Arithmetics

You can add and subtract Dates from each other.

```
d1 = Date(2016,3,31)
d2 = Date(2016,4,30)

dif    = d2 - d1                          #count the number of days between d2 and d1
difRel = Dates.value(dif)/daysinyear(d1)   #Dates.value() is the datenumber, needs prefix Date

println("difference between two dates: ",dif)
printlnPs("as a fraction of the year: ",difRel)
```

```
difference between two dates: 30 days
as a fraction of the year:      0.082
```

```
d3 = d1 + Month(1)          #one month after d1

println("d1 and one month later: ",d1," ",d3)
```

```
d1 and one month later: 2016-03-31 2016-04-30
```

Looking up Day of the Week and More

```
println("day of the week of date: ",d1," ",dayofweek(d1))
println("day of the year of date: ",d1," ",dayofyear(d1))

(y,m,d)= yearmonthday(d1)            #splitting up a date
println("\nSplitting up a date: ",y," ",m," ",d)
```

```
day of the week of date: 2016-03-31 4
day of the year of date: 2016-03-31 91

Splitting up a date: 2016 3 31
```

Printing a Date

with your own formatting (see the manual for many other formatting options)

```
println(Dates.format(d1,"d u yyyy"))               #needs prefix Dates
println(Dates.format(d1,"dd-mm-yyyy"))
```

```
31 Mar 2016
31-03-2016
```

Loading and Saving Data

to/from csv, jld2, xlsx, hdf5, and mat files.

There are also packages for reading R data files (RData.jl), numpy data files (NPZ.jl), JSON, Arrow and more but they are not covered in this tutorial.

Load Packages and Extra Functions

The packages are loaded in the respective sections below. This allows you to run parts of this notebook without having to install all packages.

The data files created by this notebook are written to and loaded from the subfolder "Results".

```
using Printf, Dates
include("src/printmat.jl")

if !isdir("Results")
    error("create the subfolder Results before running this program")
end
```

Loading a csv File

The csv ("comma-separated values") format provides a simple and robust method for moving data and it can be read by most software.

For instance, for *reading* a data file delimited by comma (,) and where the first line of the file contains variable names, then use the following

```
(x,header) = readdlm(FileName,',',header=true)
```

Alternatively, use

```
x = readdlm(FileName,',',skipstart=1)
```

to disregard the first line.

Extra arguments control the type of data (Float64, Int, etc), suppression of comment lines and more.

If you need more powerful write/read routines, try the CSV.jl package.

```
println("the raw contents of Data/CsvFile.csv:\n")
println(read("Data/CsvFile.csv",String))
```

```
the raw contents of Data/CsvFile.csv:

X,Y,Z
1.1,1.2,1.3
2.1,2.2,2.3
```

```
using DelimitedFiles

(x,header) = readdlm("Data/CsvFile.csv",',',header=true)  #read csv file

println("header of csv file:")
printmat(header)
println("x:")
printmat(x)
```

```
header of csv file:
         X          Y          Z

x:
     1.100      1.200      1.300
     2.100      2.200      2.300
```

Saving a csv File

To *write* csv data, the simplest approach is to create the matrix you want to save and then run

```
writedlm(FileName,matrix)
```

Alternatively, to write several matrices to the file (without having to first combine them), use

```
fh = open(Filename, "w")
    writedlm(fh,matrix1,',')
    writedlm(fh,matrix2,',')
close(fh)
```

Another syntax which does the same thing is

```
open(Filename, "w") do fh
    writedlm(fh,matrix1,',')
    writedlm(fh,matrix2,',')
end
```

If you only need/want limited precision in the file, use `round.(matrix1,digits=5)` instead of just `matrix1` in the `writedlm()` call.

```
x   = [1.1 1.2 1.3;
       2.1 2.2 2.3]
header = ["X" "Y" "Z"]

xx = [header; x]                          #to save
writedlm("Results/NewCsvFile.csv",xx,',')  #write csv file

printblue("NewCsvFile.csv has been created in the subfolder Results. Check it out.")
```

```
NewCsvFile.csv has been created in the subfolder Results. Check it out.
```

Loading csv with Dates and Missing Values (extra)

The next cells show how to load a csv files with dates (for instance, 15/01/1979) and data with some missing values.

The code does the following: 1. reads the csv file 2. converts `x2[:,1]` to Date, 3. finds all elements in `x = x2[:,2:end]` that are not numbers and converts them to `NaN` (use with `Float64`) or `missing` (use when data is not `Float64`).

```
"""
    readdlmFix(x,Typ=Float64,missVal=NaN)

Change elements with missing data (' ') to either NaN or missing.
`x` is the input matrix, `Typ` is the type of the output (Float64, Int, etc) and
`missval` is your choice of either `NaN` or `missing`
"""
function readdlmFix(x,Typ=Float64,missVal=NaN)
    y = replace(z->!isa(z,Number) ? missVal : z,x)
    ismissing(missVal) && (Typ = Union{Missing,Typ})   #allow missing
    y = convert.(Typ,y)
  return y
end
```

```
readdlmFix
```

```
x2 = readdlm("Data/CsvFileWithDates.csv",',',skipstart=1)
dN = Date.(x2[:,1],"d/m/y")     #to Date, "d/m/y" is the date format in the file
x  = x2[:,2:end]                #the data, but Any[] since missing data

println("dates and data (first 4 obs):")
printmat([dN[1:4] x[1:4,:]])

x = readdlmFix(x)

println("after fix of missing data (first 4 obs):")
printmat([dN[1:4] x[1:4,:]])
```

```
dates and data (first 4 obs):
1979-01-02    96.730
1979-01-03               9.310
1979-01-04    98.580     9.310
1979-01-05    99.130     9.340

after fix of missing data (first 4 obs):
1979-01-02    96.730       NaN
1979-01-03       NaN     9.310
1979-01-04    98.580     9.310
1979-01-05    99.130     9.340
```

Loading and Saving jld2

jld2 files can store very different types of data: integers, floats, strings, dictionaries, etc. It is a dialect of hdf5, designed to save different Julia objects (including `Date`).

The basic syntax of the JLD2.jl package is

```
(A,B) = load(FileName,"A","B")        #load some data
xx = load(FileName)                   #load all data into a Dict()
save(FileName,"Aaaa",A,"B",B)         #save data
jldsave(FileName;Aaaa=A,B)            #also to save data
```

(It also possible to use the same syntax as for HDF5 below, except that we use `jldopen` instead of `h5open`.)

```
using FileIO, JLD2

xx = load("Data/JldFile.jld2")                    #load entire file
println("The variables are: ",keys(xx))           #list contents of the file
x = xx["x"]                                       #extract the x variable
printmat(x)

(x,dN) = load("Data/JldFile.jld2","x","d")        #alternative way to read some of the data
println("\ndates and x from jld2 file is")
printmat([dN x])
```

```
The variables are: ["B", "C", "x", "d"]
     1.100     1.200     1.300
     2.100     2.200     2.300


dates and x from jld2 file is
2019-05-14     1.100     1.200     1.300
2019-05-15     2.100     2.200     2.300
```

```
x   = [1.1 1.2 1.3;                                #to save in a JLD2 file
       2.1 2.2 2.3]
d   = [Date(2019,5,14);                            #Julia dates
       Date(2019,5,15)]
B   = 1
C   = "Nice cat"
```

```
save("Results/NewJldFile.jld2","xxxx",x,"d",d,"B",B,"C",C)       #write jld2 file
jldsave("Results/NewJldFile2.jld2";xxxx=x,d,B,C)                 #alternative way

println("Two jld2 files have been created in the subfolder Results")
```

```
Two jld2 files have been created in the subfolder Results
```

Loading and Saving xlsx

The XLSX.jl package allows you to read and write xlsx files.

When reading a file, you typically get data of the `Any` type (for instance, `x1` in the cell below). It often makes sense to convert this to type (for instance, `Float64`) that you want.

For an automatic approach to get the right types, consider the `XLSX.gettable()` method mentioned in the cell below.

```
using XLSX

data1 = XLSX.readxlsx("Data/xlsxFile.xlsx")   #reading the entire file
x1    = data1["Data!B2:C6"]                   #extracting a part of the sheet "Data"
#x1 = XLSX.readdata("Data/XlsFile.xlsx","Data","B2:C6")   #does the same thing

x1    = convert.(Float64,x1)                  #converting from Any to Float64

println("part of the xlsx file:")
printmat(x1)

xx1 = XLSX.gettable(data1["Data"],infer_eltypes=true)   #to get types automatically
(a1,a2,a3) = xx1.data;                                  #extract "columns" from xx1
printmat(a1,a2,a3)
```

```
part of the xlsx file:
    16.660  -999.990
    16.850  -999.990
    16.930  -999.990
    16.980  -999.990
    17.080  -999.990

1950-01-03    16.660  -999.990
```

```
1950-01-04    16.850  -999.990
1950-01-05    16.930  -999.990
1950-01-06    16.980  -999.990
1950-01-09    17.080  -999.990
1950-01-10    17.030     7.000
1950-01-11    17.090     8.000
1950-01-12    16.760  -999.990
1950-01-13    16.670  -999.990
1950-01-16    16.720  -999.990
```

```
x   = [1.1 1.2 1.3;                          #writing a matrix to an xlsx file
       2.1 2.2 2.3]
y =  [11 12]

XLSX.openxlsx("Results/NewxlsxFile.xlsx",mode="w") do xf
  xf[1]["C2"] = x  #write to first sheet, matrix with upper left corner in cell C2
  XLSX.addsheet!(xf,"SecondSheet")  #create 2nd sheet
  xf[2]["A1"] = y                   #write to 2nd sheet
end

println("NewxlsxFile.xlsx has been created in the subfolder Results")
```

```
NewxlsxFile.xlsx has been created in the subfolder Results
```

Loading and Saving hdf5 (extra)

hdf5 files are used in many computer languages. They can store different types of data: integers, floats, strings (but not Julia Dates).

The basic syntax of the HDF5.jl package is

```
fh = h5open(FileName,"r")   #open for reading
    (x,y) = read(fh,"x","y")
close(fh)

fh = h5open(FileName,"w")   #open for writing
    write(fh,"x",x,"y",y)
close(fh)
```

(There are also `h5write(filename,...)` and `h5read(filename,...)` commands that writes/reads files directly.)

To save dates, save either a matrix `[y m d]` (see eg. `month(date)`) or a date value (see eg. `Dates.value(date)`).

The HDFView program allows you to look at the contents of a hdf5 file. (It is not needed here.)

```
using HDF5

fh = h5open("Data/H5File.h5","r")                    #open for reading
     println("\nVariables in h5 file: ",keys(fh))
    (x,B,ymd) = read(fh,"x","B","ymd")               #load some of the data
close(fh)

dN = Date.(ymd[:,1],ymd[:,2],ymd[:,3])               #reconstructing dates

println("\ndates and x from h5 file is")
printmat([dN x])
```

```
Variables in h5 file: ["B", "C", "x", "ymd"]

dates and x from h5 file is
2019-05-14     1.100     1.200     1.300
2019-05-15     2.100     2.200     2.300
```

```
x   = [1.1 1.2 1.3;
       2.1 2.2 2.3]
ymd = [2019 5 14;
       2019 5 15]
B   = 1
C   = "Nice cat"

fh = h5open("Results/NewH5File.h5","w")    #open file for writing
    write(fh,"x",x,"ymd",ymd,"B",B,"C",C)
close(fh)                                  #close file

println("NewH5File.h5 has been created in the subfolder Results")
```

```
NewH5File.h5 has been created in the subfolder Results
```

Saving and Loading Matlab mat files (extra)

The MAT.jl package allows you to load/save (Matlab) mat files (which is a dialect of HDF5).

```
using MAT

function DateMLtoDate(dNum)         #Matlab datenum to Julia date
  dNum   = round.(Int,dNum) .- 366
  dNTime = rata2datetime.(dNum)
  dN     = Date.(dNTime)
  return dN
end


fh = matopen("Data/MatFile.mat")
    println("\nVariables in mat file: ",names(fh))
    (x,dM) = read(fh,"x","dM")
close(fh)

d = DateMLtoDate(dM)                #Matlab datenum to Julia date

println("\ndates and x from mat file is")
printmat([d x])
```

```
Variables in mat file: ["B", "C", "dM", "x"]

dates and x from mat file is
2019-05-14     1.100     1.200     1.300
2019-05-15     2.100     2.200     2.300
```

```
x   = [1.1 1.2 1.3;
       2.1 2.2 2.3]
d   = [Date(2019,5,14);                          #Julia dates
       Date(2019,5,15)]
B   = 1
C   = "Nice cat"

dM  = Dates.value.(d)[:,:] .+ 366.0  #Julia Date to Matlab's datenum(), Float64

fh = matopen("Results/NewMatFile.mat","w")
```

```
    write(fh,"x",x)             #write one variable at a time
    write(fh,"B",B)
    write(fh,"dM",dM)
    write(fh,"C",C)
close(fh)

println("\nNewMatFile.mat has been created in the subfolder Results")
```

```
NewMatFile.mat has been created in the subfolder Results
```

Creating Variables from Variable Names (extra)

Suppose you have 1. a matrix `A` with data 2. a list of the variable names for each column of `A`, for instance, from `header` in a CSV file or saved as a vector of strings in a hdf5/mat/jld2 file.

If there are few variables, then you can manually create each of them from the loaded matrix, but this becomes tedious when there are many variables.

However, it is easy to create a Dictionary or NamedTuple which allows us to later use `D[:X]` or `N.X` to refer to variable `X`.

```
using DelimitedFiles
(A,header) = readdlm("Data/CsvFile.csv",',',header=true)
n = size(A,2)

(X,Y,Z) = [A[:,i] for i=1:n]                        #manually creating X,Y,Z

D = Dict([(Symbol(header[i]),A[:,i]) for i=1:n])    #Creating D with :X,:Y,:Z

namesB = tuple(Symbol.(header)...)                  #a tuple (:X,:Y,:Z)
N      = NamedTuple{namesB}([A[:,i] for i=1:n])     #NamedTuple with N.X, N.Y and N.Z

println("A[:,1], X, D[:X] and N.X\n")
printmat([A[:,1] X D[:X] N.X],colNames=["A[:,1]","X","D[:X]","N.X"])
```

```
A[:,1], X, D[:X] and N.X

    A[:,1]         X     D[:X]       N.X
     1.100     1.100     1.100     1.100
     2.100     2.100     2.100     2.100
```

Strings

This notebook demonstrates some basic string commands.

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");
```

String Basics

The next few cells show how to

1. combine several strings
2. test if a string contains a specific substring
3. replace part of a string with something else
4. split a string into a vector of words (and then to join them back into a single string again)
5. sort a vector of words in alphabetical order

```
str1 = "Hello"
str2 = "world!\n"
str3 = "Where are you?"

str3b = string(str1," ",str2,str3)          #combine into one string
println(str3b)
```

```
Hello world!
Where are you?
```

```
str4 = "Highway 62 Revisited"

if occursin("Highway",str4)                    #test if the string contains a substring
    println(str4)
    printblue("contains the word Highway")
end

str4 = replace(str4,"62" ⇒ "61")               #replace part of a string
printblue("\nNew, better string after a replacement: ")
println(str4)
```

```
Highway 62 Revisited
contains the word Highway

New, better string after a replacement:
Highway 61 Revisited
```

```
words = split(str4)
printblue("split a string into a vector of words:")
printmat(words)

printblue("\nand join the words into a single string:")
println(join(words," "))
```

```
split a string into a vector of words:
   Highway
        61
 Revisited


and join the words into a single string:
Highway 61 Revisited
```

```
printblue("sort the words alphabetically:")
printmat(sort(words))
```

```
sort the words alphabetically:
```

```
       61
  Highway
Revisited
```

Reading an Entire File as a String

The next cell reads a file into one single string. It keeps the formatting (spaces, line breaks etc).

```
txtFile = "Data/FileWithText.txt"

str = read(txtFile,String)       #read as string
```

```
"Dogs are nicer\nthan cats.\n        \n     This\n     is a\nfairly short file."
```

```
printblue("testing if str is a String: ")   #test if a String
println(isa(str,String),"\n")

println(str)                #printing the string read from a file
```

```
testing if str is a String:
true

Dogs are nicer
than cats.

      This
      is a
fairly short file.
```

Reading all Lines of a File into a Vector of Strings

The next cell reads a file into a vector of strings: one string per line of the file. The second cell joins those lines into one string.

```
lines = readlines(txtFile)

printmat(lines)
```

```
Dogs are nicer
than cats.

      This
      is a
fairly short file.
```

```
linesJoined = join(lines,"\n")      #join the lines of the array,
println(linesJoined)                # "\n" to create line breaks
```

```
Dogs are nicer
than cats.

      This
      is a
fairly short file.
```

Strings and Indexing (extra)

can be tricky when the string contains non-ascii characters.

Notice that you cannot change a string by indexing. For instance, `str[1] = "D"` does not work. However, you can *read* strings by indexing, if you are careful.

The next cell gives two versions of a string.

```
str1 = "Dx = -0.9x"
str2 = "Δx = -0.9x"
```

```
"Δx = -0.9x"
```

```
println(str1[1])           #works
#println(str2[2])          #uncomment to get an error
```

```
D
```

`str2[2]` does not work since the first character takes more than one byte to store. Julia has commands to get around this. For instance, see the next cell.

```
                                  #this should work in all cases
println(str1[nextind(str1,1)])   #nextind() gives the starting point of the character after th
println(str2[nextind(str2,1)])
```

```
x
x
```

Looping over All Characters in a String

in a way that works even if there are some non-ascii characters.

```
i = 1
for c in str2              #alternatively, while i <= lastindex(str)
    #global i              #only needed in script
    println(i," ",c)
    i = nextind(str2,i)
end
```

```
1 Δ
3 x
4
5 =
6
7 -
8 0
9 .
10 9
11 x
```

Creating a Long String (extra)

can be done with `string()`, but it is often quicker to write to an `IOBuffer()`.

Both approaches are demonstrated below by combining a vector of words into a string. (This is just meant as an illustration since `join(txt," ")` would be a better way to do this particular operation.)

```
txt = ["The","highway","is","for","gamblers,","better","use","your","sense\n",    #a vector of
       "Take","what","you","have","gathered","from","coincidence"];
```

```
BabyBlue1 = ""                                  #an empty string
for i = 1:length(txt)
    #global BabyBlue1                           #only needed in script
    BabyBlue1 = string(BabyBlue1," ",txt[i])    #add to the string
end
println(BabyBlue1)
```

```
 The highway is for gamblers, better use your sense
 Take what you have gathered from coincidence
```

```
iob = IOBuffer()                                #an IOBuffer
for i = 1:length(txt)
   write(iob," ",txt[i])                        #write to the buffer
end
BabyBlue2 = String(take!(iob))                  #convert to a string
println(BabyBlue2)
```

```
 The highway is for gamblers, better use your sense
 Take what you have gathered from coincidence
```

Printing

This notebook is focused on how to print (arrays of) numbers with formatting (width, number of digits shown, etc).

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl")
include("src/printTeXTable.jl");
```

Basic Printing

is done with `println()` or `display()`.

```
x = [11.111 12.12;21 22]        #a matrix to print

println(x)
```

```
[11.111 12.12; 21.0 22.0]
```

```
display(x)                     #gives basic formatting
```

```
2×2 Matrix{Float64}:
 11.111  12.12
 21.0    22.0
```

printmat() and printTeXTable

My simple function `printmat()` allows basic formatting and `printTeXTable()` prints a simple LaTeX table.

These functions use keyword arguments, for instance, `colNames=["A","B"]`. But, if you already have a variable `colNames` defined in the notebook, then using just `colNames` works fine.

`printTeXTable()` is somewhat similar, but creates a string that can be used for a LaTeX table.

```
printmat(x;colNames=["A","B"],width=7,prec=2)
```

```
      A      B
  11.11  12.12
  21.00  22.00
```

```
colNames = ["A","B"]
rowNames = ["row 1","row 2"]
printmat(x;colNames,rowNames,width=7,prec=2)                      #notice the ;
#printmat(x;colNames=colNames,rowNames=rowNames,width=7,prec=2)   #this works too
```

```
           A      B
row 1  11.11  12.12
row 2  21.00  22.00
```

```
str = printTeXTable(x;colNames,rowNames,width=7,prec=2);   #str can be printed to a file, see
```

```
\begin{table}
  \begin{tabular}{lrr}
     & A & B \\ \hline
   row 1 &   11.11 &   12.12 \\
   row 2 &   21.00 &   22.00 \\
  \hline
  \end{tabular}
\end{table}
```

The PrettyTables.jl Package

The PrettyTables.jl package provides powerful methods for formatted printing, including printing LaTeX tables.

The lines around/in the table are defined by the `tf` keyword.

The formatting is defined by the `formatters` keyword. For instance, to print a decimal floating point number with a width of 5 and 2 digits after the decimal point, use `ft_printf("%5.2f")`. (See `@printf` in the manual.)

```
using PrettyTables

pretty_table(x;header=colNames,row_labels=rowNames)

pretty_table(x;header=colNames,row_labels=rowNames,tf=tf_borderless,formatters=ft_printf("%5.2

pretty_table(x;header=colNames,row_labels=rowNames,tf=tf_ascii_dots,formatters=ft_printf("%10.
```

```
┌─────────┬────────┬─────────┐
│         │      A │       B │
├─────────┼────────┼─────────┤
│ row 1   │ 11.111 │  12.12  │
│ row 2   │   21.0 │   22.0  │
└─────────┴────────┴─────────┘
```

```
             A       B
  row 1   11.11   12.12
  row 2   21.00   22.00
.....................................
:       :           A :           B :
:.......:.............:.............:
: row 1 :    11.1110  :    12.1200  :
: row 2 :    21.0000  :    22.0000  :
:.......:.............:.............:
```

```
pretty_table(x;header=colNames,formatters=ft_printf("%5.2f"),backend=Val(:latex))   #LaTeX tab
```

```
\begin{tabular}{rr}
  \hline
  \textbf{A} & \textbf{B} \\\hline
```

```
  11.11 & 12.12 \\
  21.00 & 22.00 \\\hline
\end{tabular}
```

Printing Output to a Text File

is simple. You `open()` the file, write to it and then `close()` it. The next cells demonstrate this.

```
fh = open("Results/NewTxtFile.txt", "w")      #open the file, "w" for writing

println(fh,"Dogs are ")                       #printing to the file, notice the fh
println(fh,"nicer than cats.\n")

printmat(fh,x,prec=2)
printmat(fh,x,colNames=colNames,rowNames=rowNames,prec=2)
printTeXTable(fh,x,colNames=colNames,rowNames=rowNames,prec=2)

pretty_table(fh,x,header=colNames,row_labels=rowNames,formatters=ft_printf("%5.2f"))

close(fh)                                     #close the file
```

```
printblue("Results/NewTxtFile.txt has been closed. Check it out. The raw contents are:\n")
println(read("Results/NewTxtFile.txt",String))
```

```
Results/NewTxtFile.txt has been closed. Check it out. The raw contents are:

Dogs are
nicer than cats.

     11.11     12.12
     21.00     22.00

              A         B
row 1     11.11     12.12
row 2     21.00     22.00

\begin{table}
  \begin{tabular}{lrr}
     & A & B \\ \hline
   row 1 &        11.11 &        12.12 \\
   row 2 &        21.00 &        22.00 \\
```

```
  \hline
  \end{tabular}
\end{table}
```

	A	B
row 1	11.11	12.12
row 2	21.00	22.00

Redirecting the Output (extra)

You can also *redirect* the output from the screen to a file. This gives an simple mechanism for dumping all outputs to a file without having to insert the IO stream (called `fh` above) in all print statements.

To do that, use something like the next cell

```
redirect_stdio(stdout="Results/NewTxtFile2.txt") do     #redirect the output to a file
  println("NewTxtFile2.txt should contain this string and the next table")
  printmat(x,prec=2)
end  #end redirection

printblue("Results/NewTxtFile2.txt has been closed. Check it out. The raw contents are:\n")
println(read("Results/NewTxtFile2.txt",String))
```

```
Results/NewTxtFile2.txt has been closed. Check it out. The raw contents are:

NewTxtFile2.txt should contain this string and the next table
     11.11     12.12
     21.00     22.00
```

Formatted Printing (extra)

by using the `Printf` libary.

`@printf()` prints to a file or to the screen (and IOBuffers etc), while `@sprintf()` prints to a string.

To print with dynamic formatting (that is, where the format is decided at run-time), use `Printf.format()`. The second cell below illustrates by drawing the width and precision randomly.

```
z = 3.14159
                                        #the format string is constant
@printf("%10.2f\n",z)                    #width 10, 2 digits, floating point

str = @sprintf("%5d %g",1828,exp(1))    #width 5 and integer, compact floating point
println("my string is just this: ",str)
```

```
      3.14
my string is just this:  1828 2.71828
```

```
n = 6
(width,prec) = (10,1:n)
for i = 1:n
    #local fmt,str            #only needed in script
    fmt = Printf.Format("%$(width).$(prec[i])f")
    str = Printf.format(fmt,z)
    println(i,str)
end
```

```
1       3.1
2      3.14
3     3.142
4    3.1416
5   3.14159
6  3.141590
```

Missing Values

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");
```

NaN and missing

The `NaN` (Not-a-Number) can be used to indicate that a floating point number (like 2.1) is missing or otherwise strange. For other types of data (for instance, 2), use a `missing` instead.

Most computations involving NaNs/missings give `NaN` or `missing` as a result.

```
println(2.0 + NaN + missing)
```

```
missing
```

Loading Data

When your data (loaded from a csv file, say) has special values for missing data points (for instance, `-999.99`), then you can simply replace those values with `NaN`. This works since `NaN` is a Float64 value, so you can change an existing array of `Float64`s to `NaN`. More generally, use `missing`.

(See the tutorial on loading and saving data for more information.)

```
data = [1.0 -999.99;
        2.0 12.0;
        3.0 13.0]

z  = replace(data,-999.99⇒NaN)    #replace -999.99 by NaN or missing
z2 = replace(data,-999.99⇒missing)
printblue("z and z2: ")
printmat(z)
printmat(z2)
```

```
z and z2:
     1.000       NaN
     2.000    12.000
     3.000    13.000

     1.000   missing
     2.000    12.000
     3.000    13.000
```

Testing for NaNs/missings in an Array

You can test whether a number is `NaN` or missing by using `isunordered()`. (Use `isnan()` or `ismissing()` if you want to test specifically for one of them.)

```
z = [1.0     NaN;
     2.0     12.0;
     missing 13.0;
     4       14   ]

if any(isunordered,z)                  #check if any NaNs/missins
  println("z has some NaNs/missings")  #can also do any(isunordered.(z))
end

printblue("\nThe sum of each column: ")
printmat(sum(z,dims=1))
```

```
z has some NaNs/missings

The sum of each column:
   missing       NaN
```

Disregarding NaNs/missings in a Vector

can often be done by just `!filter()` the vector to get rid of all elements that are NaN/missing.

```
z1 = z[:,1]             #the first column of `z`

sum(filter(!isunordered,z1))    #finds all elements in z1 that are not unordered, and sums the
```

```
7.0
```

Getting Rid of Matrix Rows with any NaNs/missings

It is a common procedure in statistics to throw out all cases with NaNs/missing values. For instance, let z be a matrix and `z[t,:]` the data for period *t* which contains one or more `NaN/missing` values. It is then common (for instance, in linear regressions) to throw out that entire row of the matrix.

This is a reasonable approach if it can be argued that the fact that the data is missing is random - and not related to the subject of the investigation. It is much less reasonable if, for instance, the returns for all poorly performing mutual funds are listed as "missing" - and you want to study what fund characteristics that drive performance.

The code below shows a simple way of how to through out all rows of a matrix with at least one `NaN/missing`.

For statistical computations, you may also consider the NaNStatistics.jl package.

```
printblue("z:")
printmat(z)

vb = any(isunordered,z,dims=2) #indicates rows with NaNs/missings, gives a Tx1 matrix
vc = .!vec(vb)                 #indicates rows without NaNs/missings, vec to make a vector

z2 = z[vc,:]           #keep only rows without NaNs/missings
printblue("z2: a new matrix where all rows with any NaNs/missings have been pruned:")
printmat(z2)
```

```
z:
     1.000       NaN
     2.000    12.000
   missing    13.000
```

```
     4.000    14.000

z2: a new matrix where all rows with any NaNs/missings have been pruned:
     2.000    12.000
     4.000    14.000
```

Converting to a Standard Array

Once you have pruned all rows with missings, you may want to convert the matrix to, for instance, Float64. This might simplify some of the later code. Notice that if there were no missings (just NaNs), then no conversion is needed.

```
println("The type of z2 is ", typeof(z2))

z3 = convert.(Float64,z2)            #could also use `disallowmissing()` from the `Missings.jl
println("\nThe type of z3 is ", typeof(z3))
```

```
The type of z2 is Matrix{Union{Missing, Float64}}

The type of z3 is Matrix{Float64}
```

Downloading Files from Internet

is done by

```
import Downloads
Downloads.download(url)
```

Loading Packages

```
using Downloads
```

```
if !isdir("Results")
    error("create the subfolder Results before running this program")
end
```

Some Data from Kenneth French's Homepage

```
http = string("http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/ftp/",
              "F-F_Research_Data_Factors_daily_CSV.zip")

println("File to download: ",http)
result = Downloads.download(http,"Results/WhatIJustDownloaded.zip")

println("\nDone. Check $result \n")
```

```
File to download: http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/ftp/F-
F_Research_Data_Factors_daily_CSV.zip

Done. Check Results/WhatIJustDownloaded.zip
```

Solving Non-linear Equations

The Roots.jl package provides methods for solving a non-linear equation (one variable, one function).

For a system of non-linear equations, use NLsolve.jl.

Load Packages and Extra Functions

```
using Printf, Roots, NLsolve

include("src/printmat.jl")
```

```
@doc2
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Defining and Plotting the Function

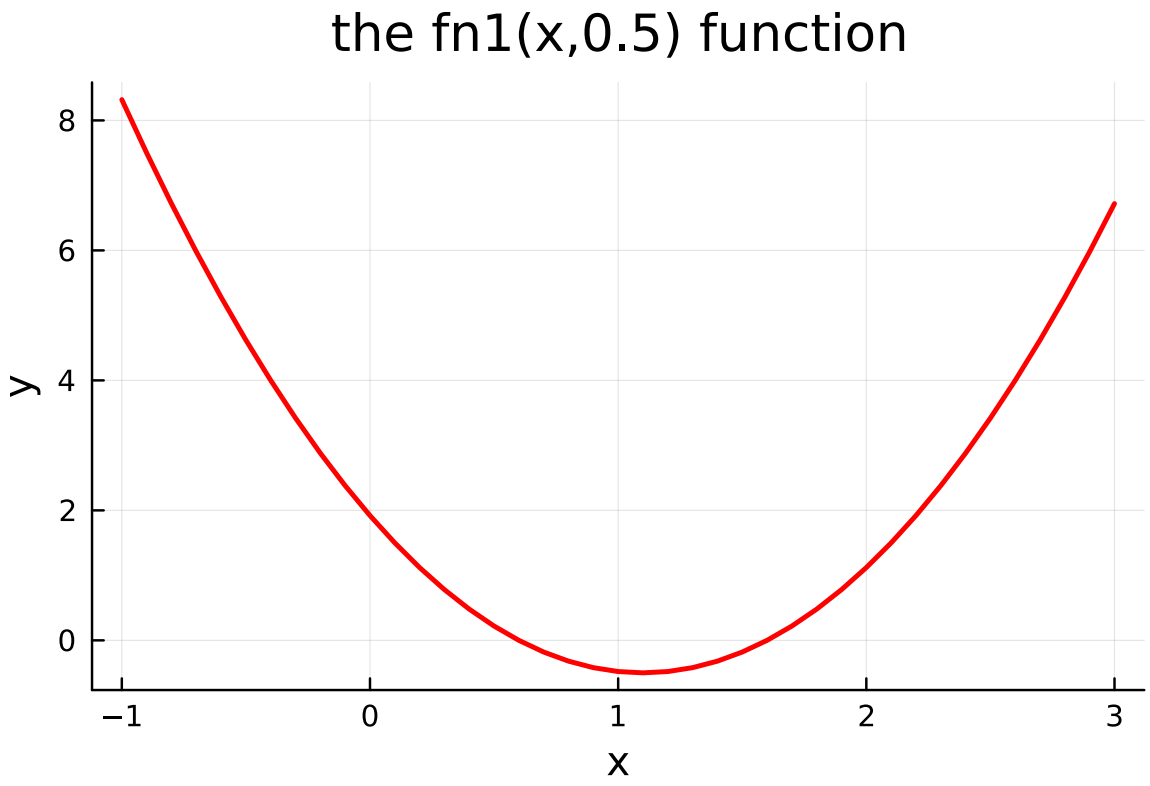
In the next few cells, we define a fairly simple function and then plot it.

If possible, plot your function before trying to find the roots. Maybe you see something strange, perhaps there are several roots? It also helps you set the initial guesses (or brackets) for root solving.

```
fn1(x,c) = 2*(x - 1.1)^2 - c
```

```
fn1 (generic function with 1 method)
```

```julia
x = range(-1,3,length=41)

p1 = plot( x,fn1.(x,0.5),
           linecolor = :red,
           linewidth = 2,
           legend = nothing,
           title = "the fn1(x,0.5) function",
           xlabel = "x",
           ylabel = "y" )
display(p1)
```



There seems to be two roots: around 0.6 and 1.6.

Solving a Non-Linear Equation

The Roots package wants a function with only one input and one output. An easy way to turn `fn1(a,0.5)` into that form is by defining an anonymous function:

`x->fn1(x,0.5)`

Then, running

`find_zero(x->fn1(x,0.5),(x₀,x₁))`

searches for a root in the `[x₀,x₁]` interval. Alternatively, you can do

`find_zero(x->fn1(x,0.5),x₂)`

where `x₂` is a single starting guess.

```
xRoot1 = find_zero(x->fn1(x,0.5),(-1,1))         #searches for roots in [-1,1]
println("at which x is fn1(x,0.5) = 0? ",xRoot1)

xRoot2 = find_zero(x->fn1(x,0.5),2)              #searches for roots around 2
println("at which x is fn1(x,0.5) = 0? ",xRoot2)

printblue("\nyes, there are several roots. Just look at it (in the plot)")
```

```
at which x is fn1(x,0.5) = 0?      0.600
at which x is fn1(x,0.5) = 0?      1.600

yes, there are several roots. Just look at it (in the plot)
```

Performance Tips (extra)

Anonymous functions like `x->fn1(x,b)` where `b` is a global variable might be slow when used as above. To speed up, either wrap the call in another function or by create *both* `b` and the anonymous function `x->fn1(x,b)` inside a loop. See the cell below.

```
FindTheRoot(b) = find_zero(x->fn1(x,b),(-1,1))        #approach 1: a short wrapping function
b = 0.5
println(FindTheRoot(b),"\n")

xRoot4 = fill(NaN,2)                                  #appraoch 2: create things inside a loop
for i = 1:2                                           #are local to the loop
    local b
    b = 0.5/i
```

```
    xRoot4[i] = find_zero(x->fn1(x,b),(-1,1))
end
printmat(xRoot4)
```

```
     0.600

     0.600
     0.746
```

Finding Many Roots

Instead, running

`find_zeros(x->fn1(x,0.5),x₀,x₁)`

searches for all roots between x_0 and x_1. (Notice the *s* in `find_zeros`.)

```
xRootsAll = find_zeros(x->fn1(x,0.5),-1,3)            #find_zeros (notice the "s")

printlnPs("at which x is fn1(x,0.5) = 0? ",xRootsAll)
```

```
at which x is fn1(x,0.5) = 0?      0.600     1.600
```

Solving a System of Non-Linear Equations

The NLsolve.jl package has many options. The cells below illustrate a very simple case (2 non-linear equations with 2 unknowns, no information about the derivatives).

The two equations are

$y - x^2 - 1 = 0$

$y - x - 1 = 0$

and the roots are at $(x, y) = (0, 1)$ and also at $(1, 2)$.

```
function fn2(p)              #p is a vector with 2 elements, the output too
    (x,y) = (p[1],p[2])
    z = [y-x^2-1;y-x-1]      #equal to [0,0]
    return z
end
```

```
fn2 (generic function with 1 method)
```

```
Sol2a = nlsolve(fn2,[0.0,0.5])
printlnPs("There is a solution at         ",Sol2a.zero)

Sol2b = nlsolve(fn2,[1.0,0.0])  #try again, using another starting guess
printlnPs("There is a another solution at ",Sol2b.zero)
```

```
There is a solution at             -0.000     1.000
There is a another solution at      1.000     2.000
```

Optimization

of linear-quadratic problems, using the OSQP.jl package.

The optimization problems are (for pedagogical reasons) the same as in the other notebook about optimization. Otherwise, the methods illustrated here are well suited for cases when the objective involves the portfolio variance ($w'\Sigma w$) or when the estimation problem is based on minimizing the sum of squared residuals ($u'u$), and the restrictions are linear expressions.

Load Packages and Utility Functions

```
using Printf, LinearAlgebra, SparseArrays, OSQP

include("src/printmat.jl");
```

```
using Plots, LaTeXStrings    #LaTeXStrings to facilitate using LaTeX in plots
default(size = (480,320),fmt = :png)
```

The OSQP.jl Optimization Package

The OSQP.jl package is tailor made for solving linear-quadratic problems (with linear restrictions). It solves problems of the type

$\min 0.5\theta' P\theta + q'\theta$ subject to $l \leq A\theta \leq u$,

where θ is a vector of choice variables.

To get an equality restriction in row `i`, set `l[i]=u[i]`.

Notice that (P, A) to should be `Sparse` matrices and (q, l, u) vectors with `Float64` numbers. We consider several cases below: no restrictions on θ, bounds on θ, and a linear equality restriction.

Unconstrained Minimization

We will define the following minimization problem

$$P = 2\begin{bmatrix} 1 & 0 \\ 0 & 16 \end{bmatrix}$$

and

$$q = \begin{bmatrix} -4 \\ 24 \end{bmatrix}.$$

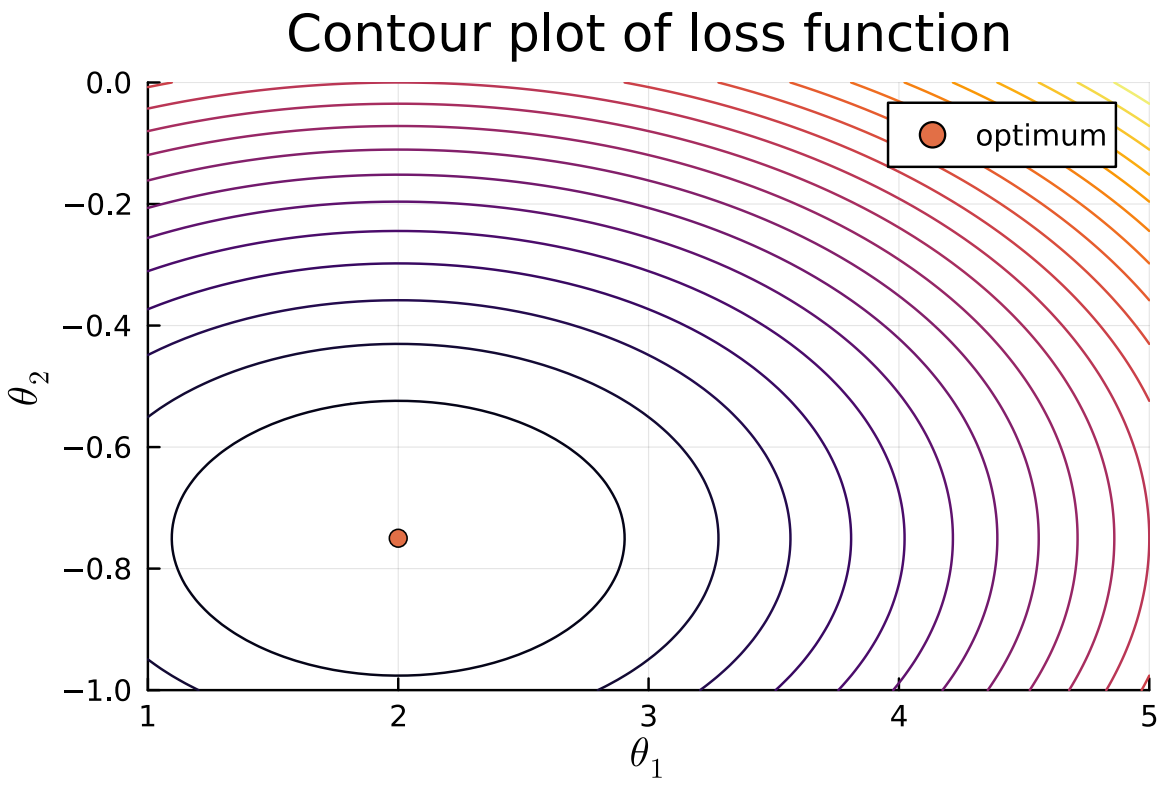
If there are no constraints, then the solution should be $\theta = (2, -3/4)$.

It is straightforward to see that this is the same as minimizing $(\theta_1 - 2)^2 + (4\theta_2 + 3)^2$

Before doing the optimization, we plot the contours of the loss function.

```
P = 2*[1  0;           #we minimize 0.5θ'Pθ + q'θ, hence the 2*[]
       0  16]          #this is the same as minimizing (x-2)^2 + (4y+3)^2
q =   [-4.0, 24.0]     #vector, Float64

function Lossfn(P,q,θ)
     L = 0.5*θ'*P*θ + q'*θ
    return L
end

(n1,n2) = (82,122)
θ₁ = range(1,5,length=n1)
θ₂ = range(-1,0,length=n2)

loss2d = fill(NaN,n1,n2)      #matrix with loss fn values
for i in 1:n1, j in 1:n2
    loss2d[i,j] = Lossfn(P,q,[θ₁[i],θ₂[j]])
end

p1 = contour( θ₁,θ₂,loss2d',       #notice: loss2d'
              xlims = (1,5),
              ylims = (-1,0),
              legend = false,
              levels = 21,
              title = "Contour plot of loss function",
             xlabel = L"\theta_1",
```

```
            ylabel = L"\theta_2" )
scatter!([2],[-0.75],label="optimum",legend=true)
display(p1)
```




```
A = zeros(1,2)          #effectively no restriction
l = [0.0]               #vectors, Float64
u = [0.0]

settings = Dict(:verbose => true)
model = OSQP.Model()
OSQP.setup!(model; P=sparse(P), q=q, A=sparse(A), l=l, u=u, settings...)
result = OSQP.solve!(model)

printblue("Unconstrained minimization: the solution should be (2,-3/4):")
printmat(result.x)
printred("\nand lots of information about the iterations:")
```

```
Unconstrained minimization: the solution should be (2,-3/4):
```

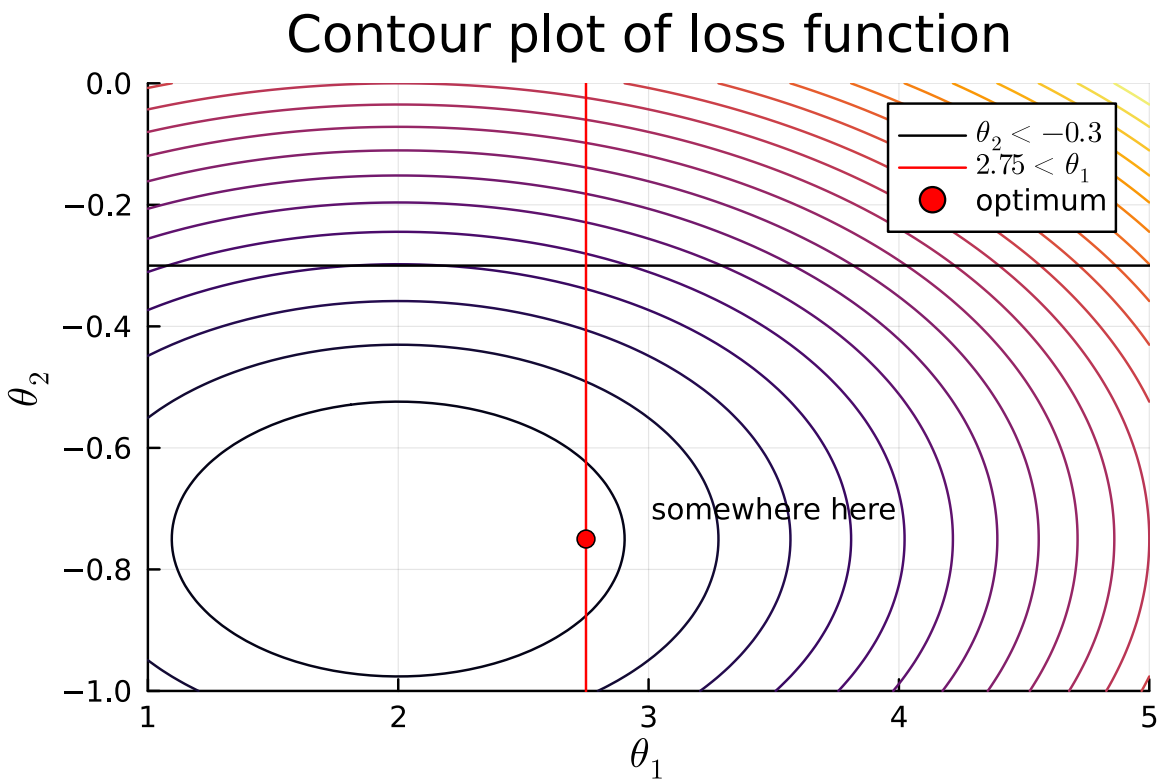
```
 2.000
-0.750


and lots of information about the iterations:
-----------------------------------------------------------------
           OSQP v0.6.2  -  Operator Splitting QP Solver
              (c) Bartolomeo Stellato,  Goran Banjac
        University of Oxford  -  Stanford University 2021
-----------------------------------------------------------------
problem:  variables n = 2, constraints m = 1
          nnz(P) + nnz(A) = 2
settings: linear system solver = qdldl,
          eps_abs = 1.0e-003, eps_rel = 1.0e-003,
          eps_prim_inf = 1.0e-004, eps_dual_inf = 1.0e-004,
          rho = 1.00e-001 (adaptive),
          sigma = 1.00e-006, alpha = 1.60, max_iter = 4000
          check_termination: on (interval 25),
          scaling: on, scaled_termination: off
          warm start: on, polish: off, time_limit: off

iter  objective    pri res    dua res    rho        time
   1 -8.3200e+000  0.00e+000  1.44e+001  1.00e-001  1.04e-004s
  25 -1.3000e+001  0.00e+000  6.82e-005  1.00e-001  1.21e-004s

status:               solved
number of iterations: 25
optimal objective:    -13.0000
run time:             1.28e-004s
optimal rho estimate: 1.00e-006
```

Constrained Minimization

Bounds on the solution: $2.75 \leq \theta_1$ and $\theta_2 \leq -0.3$.

```
p1 = contour( θ₁,θ₂,loss2d',
              xlims = (1,5),
              ylims = (-1,0),
              legend = false,
              levels = 21,
              title = "Contour plot of loss function",
              xlabel = L"\theta_1",
```

```
            ylabel = L"\theta_2",
            annotation = (3.5,-0.7,text("somewhere here",8)) )
hline!([-0.3],linecolor=:black,label=L"\theta_2 < -0.3")
vline!([2.75],linecolor=:red,line=:solid,label=L"2.75 < \theta_1")
scatter!([2.75],[-0.75],color=:red,label="optimum",legend = true)
display(p1)
```



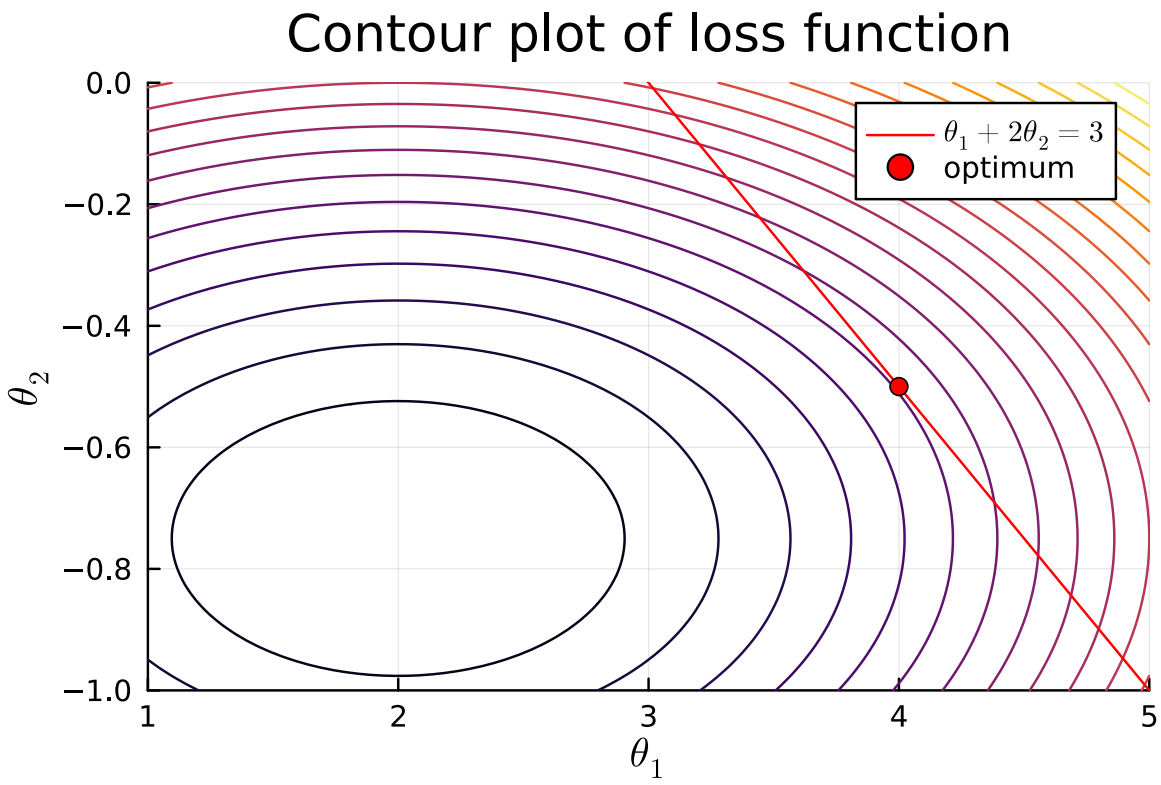

```
A = I[1:2,1:2]     #identity matrix, I₂
l = [2.75,-Inf]    #2.75 <= θ₁ <= Inf, -Inf <= θ₂ <= -0.3
u = [Inf,-0.3]

settings = Dict(:verbose ⇒ false)
model = OSQP.Model()
OSQP.setup!(model; P=sparse(P), q=q, A=sparse(A), l=l, u=u, settings...)
result = OSQP.solve!(model)

printblue("with bounds on the solution: the solution should be (2.75,-0.75):")
printmat(result.x)
```

```
with bounds on the solution: the solution should be (2.75,-0.75):
     2.750
    -0.750
```

Constrained Minimization

A linear equality constraint: $\theta_1 + 2\theta_2 = 3$.

```
p1 = contour(θ₁,θ₂,loss2d',
             xlims = (1,5),
             ylims = (-1,0),
             legend = false,
             levels = 21,
             title = "Contour plot of loss function",
             xlabel = L"\theta_1",
             ylabel = L"\theta_2" )
Plots.abline!(-0.5,1.5,linecolor=:red,line=:solid,label=L"\theta_1+2\theta_2=3")
scatter!([4],[-0.5],markercolor=:red,label="optimum",legend=true)
display(p1)
```



```
A = [1 2]                    #equality constraint
l = [3.0]
u = [3.0]

model = OSQP.Model()
OSQP.setup!(model; P=sparse(P), q=q, A=sparse(A), l=l, u=u, settings...)
result = OSQP.solve!(model)

printblue("equality constraint: the solution should be (4,-1/2):\n")
printmat(result.x)
```

```
equality constraint: the solution should be (4,-1/2):

     4.000
    -0.500
```

Updating the Problem (extra)

When you just want to change some of the inputs to the optimization problem, then it may pay off to do `update!()`. This is especially useful when you resolve the problem many times (in a loop, say).

```
model = OSQP.Model()                #setting up the problem for the first time
OSQP.setup!(model; P=sparse(P), q=q, A=sparse(A), l=l, u=u, settings...)
result = OSQP.solve!(model)
printmat(result.x)

q_new = [-2.0,20.0]                 #resolve the problem with different q values
OSQP.update!(model;q=q_new)         #(partially) update the problem
result = OSQP.solve!(model)
printmat(result.x)
```

```
     4.000
    -0.500

     3.600
    -0.300
```

An Alternative Package (extra)

The Clarabel.jl package is an interesting alternative to OSQP.jl. It has a slightly different syntax, but the `DoClarabel()` function in `src/DoClarabel.jl` provides a way to call it using the same syntax as for OSQP.jl.

```
using LinearAlgebra, SparseArrays, Clarabel
include("src/DoClarabel.jl");
```

```
(x,result,fn0) = DoClarabel(P,q,A,l,u,Clarabel.Settings(verbose = false))
printblue("equality constraint: the solution should be (4,-1/2), as before:")
printmat(x)
```

```
equality constraint: the solution should be (4,-1/2), as before:
     4.000
    -0.500
```

Numerical Optimization

This notebook uses the Optim.jl package which has general purpose routines for optimization. (Other packages that do similar things are Optimization.jl, NLopt.jl and JuMP.jl)

The optimization problems are (for pedagogical reasons) the same as in the other notebook about optimization. This means that the solutions should be very similar and that the contour plots in the other notebook can be used as references.

However, the current notebook is focused on methods for solving general optimization problems. In contrast, the other notebook focuses on linear-quadratic problems (mean-variance, least squares, etc), where there are faster algorithms.

Load Packages and Extra Functions

```
using Printf, Optim

include("src/printmat.jl");
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Optimization with One Choice Variable

Running

```
Sol = optimize(x->fn1(x,0.5),a,b)
```

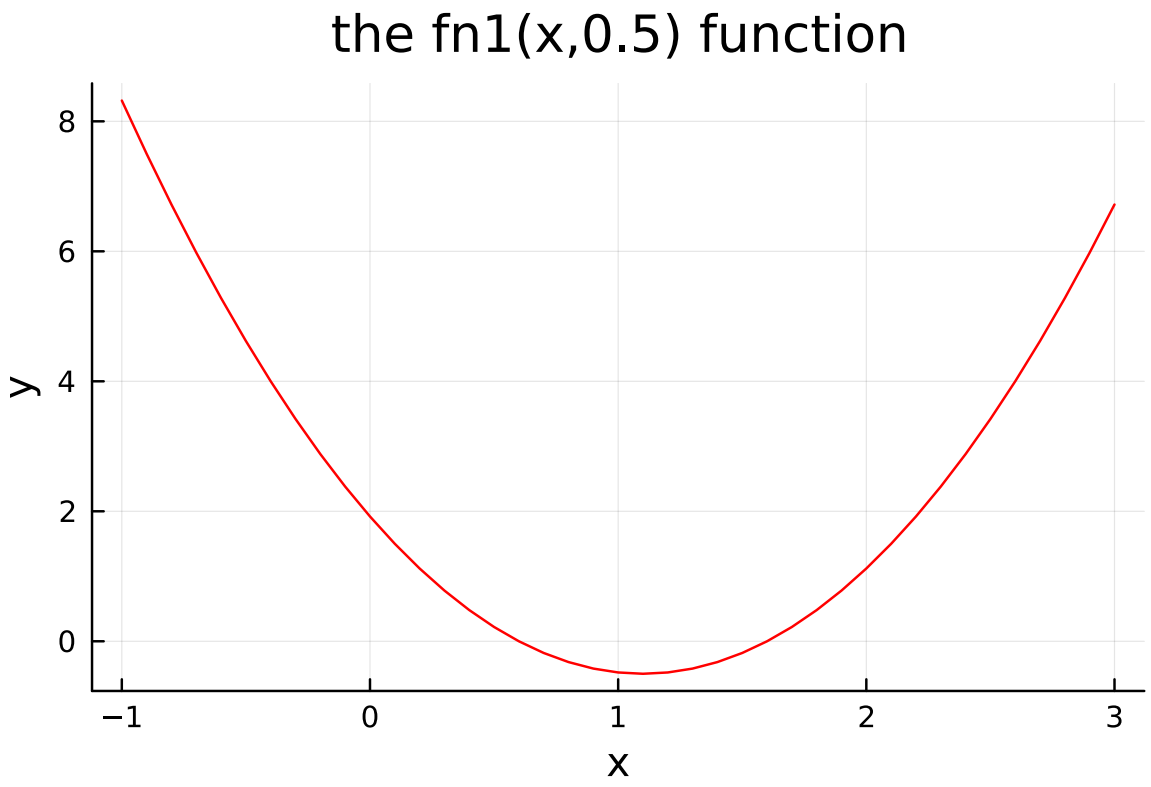
finds the x value (in the interval [a,b]) that *minimizes* fn1(x,0.5). The x→fn1(x,0.5) syntax makes this a function of x only, which is what the optimize() function wants.

The output (Sol) contains a lot of information.

```
function fn1(x,c)                    #notice: the function has two arguments
  value = 2*(x - 1.1)^2 - c
  return value
end
```

```
fn1 (generic function with 1 method)
```

```
x = -1:0.1:3

p1 = plot( x,fn1.(x,0.5),                 #plotting the function can help identify
           linecolor = :red,              #starting points and possible issues
           linewidth = 1,
           legend = nothing,
           title = "the fn1(x,0.5) function",
           xlabel = "x",
           ylabel = "y" )
display(p1)
```




```
Sol = optimize(x->fn1(x,0.5),-2.0,3.0)
println(Sol)

printblue("\nThe minimum is at: ", Optim.minimizer(Sol))     #the optimal x value
printblue("Compare with the plot above")
```

```
Results of Optimization Algorithm
 * Status: success
 * Algorithm: Brent's Method
 * Search Interval: [-2.000000, 3.000000]
 * Minimizer: 1.100000e+00
 * Minimum: -5.000000e-01
 * Iterations: 5
 * Convergence: max(|x - x_upper|, |x - x_lower|) <= 2*(1.5e-08*|x|+2.2e-16): true
 * Objective Function Calls: 6

The minimum is at: 1.1
Compare with the plot above
```

One Choice Variable: Supplying a Starting Guess Instead (extra)

If you prefer to give a starting guess c instead of an interval, then supply it as as a vector [c]. The solution is then also a vector (with one element.)

```
Solb = optimize(x->fn1(x[],0.5),[0.1])

printlnPs("The minimum is at: ",Optim.minimizer(Solb))
```

```
The minimum is at:      1.100
```

Several Choice Variables: Unconstrained Optimization

In the example below, we choose (x, y) so as to minimize the fairly simple objective function

$(x-2)^2 + (4y+3)^2$,

without any constraints. The solution should be $(x, y) = (2, -3/4)$.

```
function fn2(θ)
    (x,y) = (θ[1],θ[2])          #unpack the choice variables and get nicer names
     L    = (x-2)^2 + (4*y+3)^2
    return L
end
```

```
fn2 (generic function with 1 method)
```

```
θ₀ = [4.0,-0.5]                 #starting guess

Sol = optimize(fn2,θ₀)          #use θ->lossfn(θ,other arguments) if there are
                                #additional (non-choice) function arguments
printlnPs("minimum at (x,y)= ",Optim.minimizer(Sol))
```

```
minimum at (x,y)=      2.000    -0.750
```

Several Choice Variables: Bounds on the Solutions

The next few cells discuss how to impose bounds on the solution.

In the example below, we impose $2.75 \leq x$ (a lower bound) and $y \leq -0.3$ (an upper bound). We will see that only one of these restrictions binds.

```
lower_θ = [2.75, -Inf]
upper_θ = [Inf, -0.3]

Sol = optimize(fn2,lower_θ,upper_θ,θ₀)
println("The optimum is at (x,y) = ",Optim.minimizer(Sol))
```

```
The optimum is at (x,y) =      2.750    -0.750
```

Several Choice Variables: Equality Restrictions (extra)

We now impose the constraint that $x + 2y - 3 = 0$ (that is, $y = 1.5 - 0.5x$) The `Optim.jl` package can handle both equality and inequality constraints, but the syntax is a bit more involved than in the earlier examples. The remarks below discuss this. (Clearly, we could handle this single linear equality constraint by rewriting the objecive function instead, so consider this as just a simple illustration.) The solution should be at $(x, y) = (4, -0.5)$

Remarks on the Code

1. `TwiceDifferentiable(fn2,θ₀)` and `TwiceDifferentiableConstraints(EqConstrfun!,lower_θ,upper_θ` tell the `optimize()` function that the objective function and constraint are differentiable (twice).
2. The function for the constraint (`EqConstrfun!(c,θ)`) takes two inputs: `c` will be filled (by `optimize()`) with the calculations in the function and `θ` is the vector of choice variables.
3. `lower_θ` and `upper_θ` are (vectors of) bounds on the choice variables. With no bounds, use `[-Inf,-Inf]` and `[Inf,Inf]`.
4. `lower_c` and `upper_c` are (vectors of) bounds on the output from the constraint function. With a single equality constraint, use `[0.0]` for both.

```
function EqConstrfun!(c,θ)        #a function for a constraint
    (x,y) = (θ[1],θ[2])           #c is an array where results are written to
    c[1]  = x + 2*y - 3           #update the input c
    return c
end

lower_θ = [-Inf,-Inf]                 #effectively no bounds on the parameters, but needed
upper_θ = [Inf,  Inf]
lower_c = [0.0]                       #c from EqConstrfun! should be lower_c <= c <= upper_c
upper_c = [0.0]

df  = TwiceDifferentiable(fn2,θ₀)       #informing about the differentiability of the problem
dfc = TwiceDifferentiableConstraints(EqConstrfun!,lower_θ,upper_θ,lower_c,upper_c)
Sol = optimize(df,dfc,θ₀,IPNewton())
println("Constrained, x+2y-3=0")
printlnPs(Optim.minimizer(Sol))
```

```
Constrained, x+2y-3=0
     4.000    -0.500
```

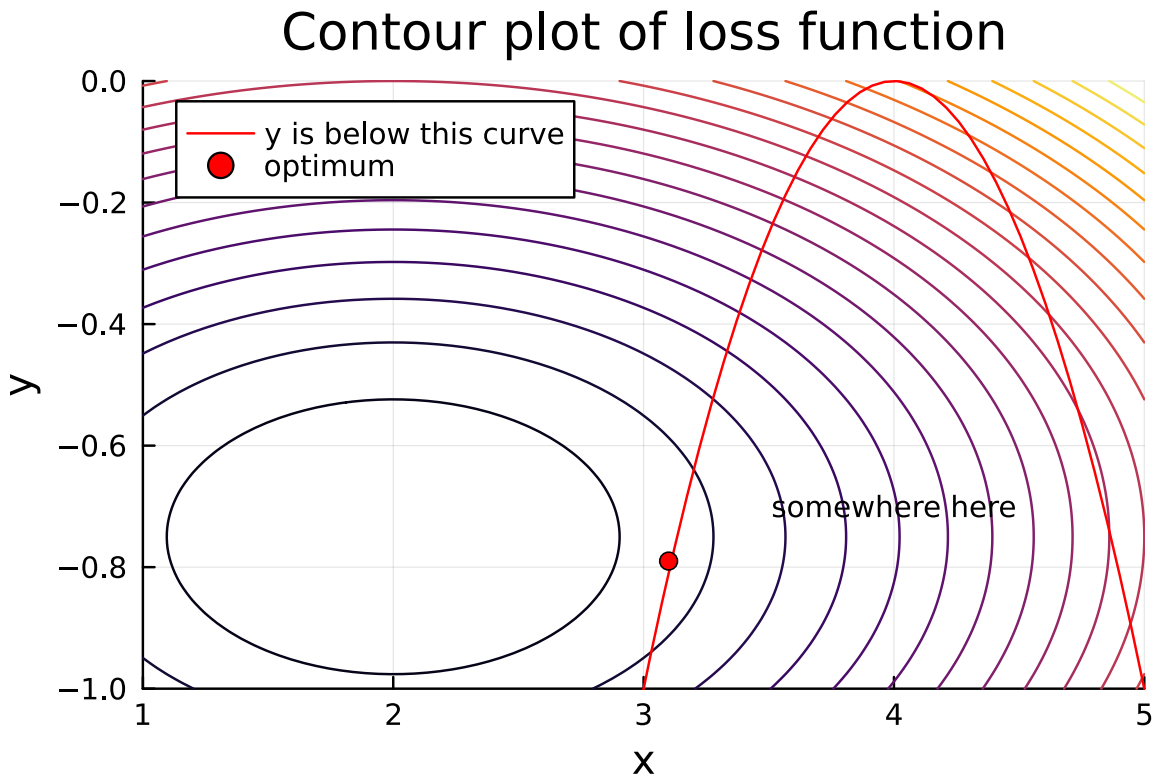
Optimization with Inequality Constraints (extra)

We now impose the non-linear constraint that $y \leq -(x-4)^2$, that is, $y + (x-4)^2 \leq 0$.

This is very similar to the linear constraint above, except that we need a new constraint function and that `lower_c = [-Inf]` and `upper_c = [0.0]`.

```
nx = 2*41
ny = 2*61
x = range(1,5,length=nx)
y = range(-1,0,length=ny)

loss2d = fill(NaN,(nx,ny))      #matrix with loss fn values
for i in 1:nx, j in 1:ny
    loss2d[i,j] = fn2([x[i];y[j]])
end

yRestriction = -(x.-4).^2               #y should be less than this

p1 = contour( x,y,loss2d',
```

```
                legend = false,
                levels = 21,
                title = "Contour plot of loss function",
                xlabel = "x",
                ylabel = "y",
                xlims = (1,5),
                ylims = (-1,0),
                annotation = (4.0,-0.7,text("somewhere here",8)) )
plot!(x,yRestriction,linecolor=:red,linewidth=1,label="y is below this curve")
scatter!([3.1],[-0.79],markercolor=:red,label="optimum",legend=:topleft)
display(p1)
```




```
function IneqConstrfun!(c,θ)              #another restriction
    (x,y) = (θ[1],θ[2])
    c[1]  = y + (x-4)^2                   #lower_c <= c[1] <= upper_c
  return c
end

lower_θ = [-Inf,-Inf]                     #effectively no bounds on the parameters
```

```
upper_θ = [Inf,Inf]
lower_c = [-1_000_000.0]              # -Inf <= c <= 0.0, but a very negative value works bet
upper_c = [0.0]

df  = TwiceDifferentiable(fn2,θ₀)
dfc = TwiceDifferentiableConstraints(IneqConstrfun!,lower_θ,upper_θ,lower_c,upper_c)
Sol = optimize(df,dfc,θ₀,IPNewton())
println("Constrained, y + (x-4)^2 <= 0")
println∫Ps(Optim.minimizer(Sol))
```

```
Constrained, y + (x-4)^2 <= 0
     3.112    -0.789
```

Several Choice Variables: Supplying the Gradient (extra)

Supplying a function for calculating the derivatives improves speed and accuracy. See below for an example.

```
function g2!(G,θ)                          #derivatives of fn2 wrt. θ[1] and θ[2]
    (x,y) = (θ[1],θ[2])
    G[:] = [2*(x-2), 2*4*(4*y+3)]          #fills/updates an existing vector
    return G
end

Sol3 = optimize(fn2,g2!,[1.0,-0.5])
println(Sol3)
printblue("Probably faster than the solution above")
printblue("Minimum at: ",round.(Optim.minimizer(Sol3),digits=3))
```

```
 * Status: success

 * Candidate solution
    Final objective value:     9.860761e-30

 * Found with
    Algorithm:     L-BFGS

 * Convergence measures
    |x - x'|               = 9.34e-01 ≰ 0.0e+00
```

```
    |x - x'|/|x'|          = 4.67e-01 ≰ 0.0e+00
    |f(x) - f(x')|         = 8.75e-01 ≰ 0.0e+00
    |f(x) - f(x')|/|f(x')| = 8.88e+28 ≰ 0.0e+00
    |g(x)|                 = 2.49e-14 ≤ 1.0e-08

 * Work counters
    Seconds run:   0  (vs limit Inf)
    Iterations:    2
    f(x) calls:    7
    ∇f(x) calls:   7

Probably faster than the solution above
Minimum at: [2.0, -0.75]
```

Numerical Integration and Differentiation

This notebook illustrates how to perform numerical integration and differentiation.

There are several packages for doing this. Here, the focus is on `QuadGK.jl` and `FiniteDiff.jl` packages. Towards the end of the notebook, there are also some (extra) comments about the `ForwardDiff.jl` and `ReverseDiff.jl` packages.

Load Packages and Extra Functions

```
using Printf

include("src/printmat.jl");
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Numerical Integration

with the QuadGK.jl package.

There are many alternative packages (for instance, HCubature.jl), but they are not discussed in this notebook.

```
using QuadGK
```

The Pdf of the N() Distribution

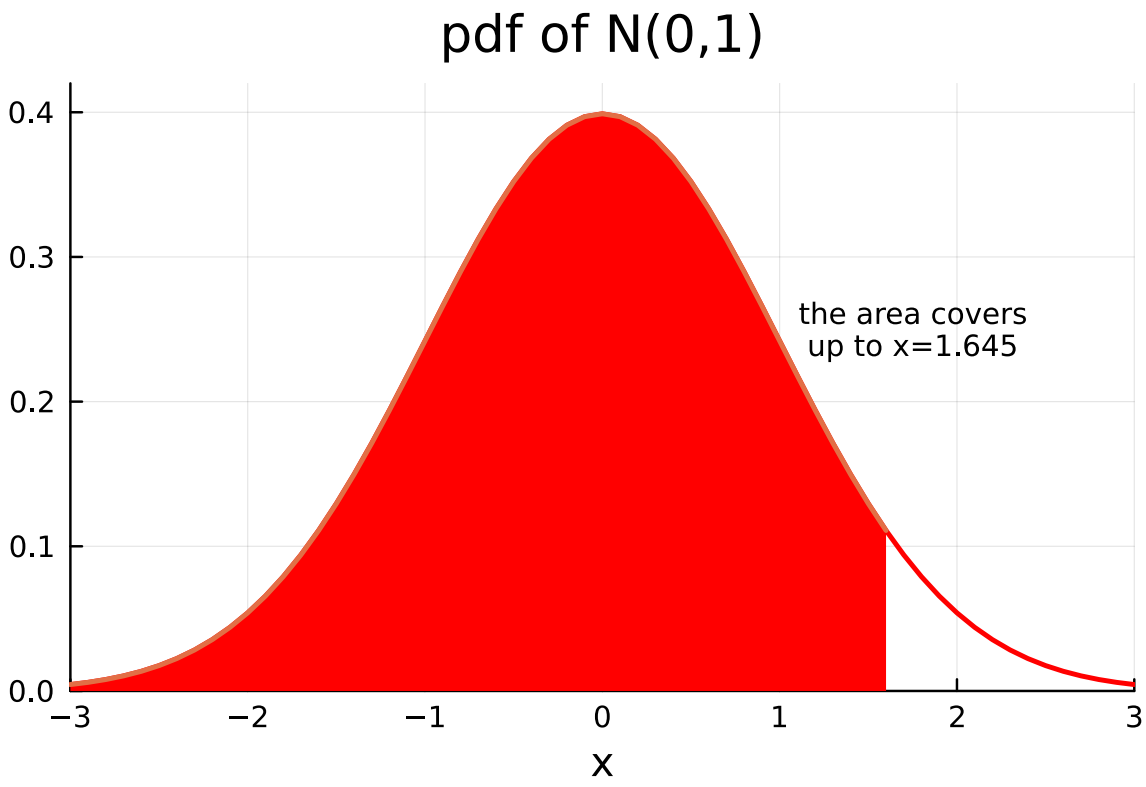
As a simple illustration, the next cells plot and integrate the $N(0,\sigma)$ pdf.

Notice: the function takes μ and σ (not σ^2) as inputs, similar to the `Distributions.jl` package.

```
function ϕNS(x,μ=0,σ=1)       #pdf of N(μ,σ), defaults to N(0,1)
    z   = (x - μ)/σ
    pdf = exp(-0.5*z^2)/(sqrt(2*pi)*σ)
    return pdf
end
```

```
ϕNS (generic function with 3 methods)
```

```
x  = -3:0.1:3
xb = filter(<=(1.645),x)          #same as x[x.<=1.645]

p1 = plot( x,ϕNS.(x),
           linecolor = :red,
           linewidth = 2,
           ylims = (0,0.42),
           xlims = (-3,3),
           legend = nothing,
           title = "pdf of N(0,1)",
           xlabel = "x",
           ylabel = "",
           annotation = (1.75,0.25,text("the area covers\nup to x=1.645",8)) )
plot!(xb,ϕNS.(xb),fillcolor=:red,linewidth=2,legend=nothing,fill=(0,:red))
display(p1)
```



Calculating Prob(x<=1.645)

Let $\phi(x, \mu, \sigma)$ be the pdf of an $N(\mu, \sigma)$ variable. The next cell calculates

$$\int_{-\infty}^{1.645} \phi(x, 0, \sigma) dx,$$

by numeric integration.

The input to `quadgk` should be a function with only one argument. We do that by creating an anonymous function `x->ϕNS(x,0,2)` or just `ϕNS` is we want to use the default values of the `(μ,σ)` inputs.

```
cdf1, = quadgk(ϕNS,-Inf,1.645)              #N(0,1), default values
println∫Ps("\nPr(x<=1.645) according to N(0,1):", cdf1)

cdf2, = quadgk(x->ϕNS(x,0,2),-Inf,1.645)    #N(0,σ=2)
println∫Ps("\nPr(x<=1.645) according to N(0,2):", cdf2)
```

```
Pr(x<=1.645) according to N(0,1):     0.950

Pr(x<=1.645) according to N(0,2):     0.795
```

Numerical Derivatives

We use the FiniteDiff.jl package to calculate numerical derivatives.

The `FiniteDiff.jl` package does not export its functions and the names are very long, for instance, `FiniteDiff.finite_difference_derivative`. In the next cell we therefore rename the relevant functions.

```
using FiniteDiff: finite_difference_derivative as δ, finite_difference_gradient as ∇
```

A Derivative of a Function with One Input and One Output

```
function fn1(x,a)                  #a simple function, to be differentiated
   return (x - 1.1)^2 - a          #see below for a plot
end
```

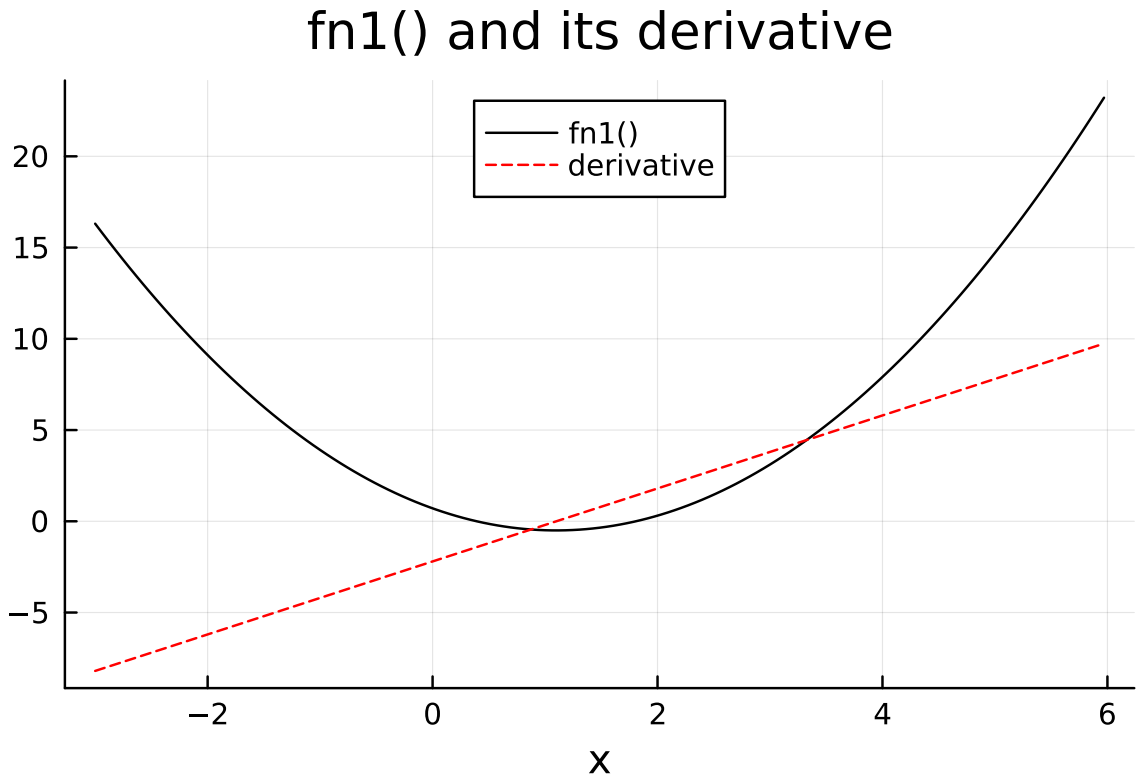
```
fn1 (generic function with 1 method)
```

```
x₀ = 2.0
dydx_A = δ(x->fn1(x,0.5),x₀)              #make sure x₀ is a Float64

println∏s("The derivative at x=$x₀ is: ",dydx_A)
```

```
The derivative at x=2.0 is:      1.800
```

```
x = -3:6/99:6          #calculate the derivative at many points

dydx_A = [δ(z->fn1(z,0.5),xᵢ) for xᵢ in x];              #xᵢ should be Float64
```

```
p1 = plot( [x x],[fn1.(x,0.5) dydx_A],
           line = [:solid :dash],
           linecolor = [:black :red],
           label = ["fn1()" "derivative"],
           legend = :top,
           title = "fn1() and its derivative",
           xlabel = "x",
           ylabel = "" )
display(p1)
```



A Derivative of a Function with Many Inputs and One Output (Gradient)

```
fn2(w,Σ) = w'*Σ*w                  #a quadratic form. The derivative wrt w is 2Σw

Σ = [1 0.5
     0.5 2]
w₀ = [1.0,2.0]                    #floats
```

```
grad = ∇(z->fn2(z,Σ),w₀)

printblue("The gradient is:\n")
printmat(grad,2*Σ*w₀;colNames=["numerical","exact"])
```

```
The gradient is:

 numerical     exact
     4.000     4.000
     9.000     9.000
```

Comments on Using the ForwardDiff or ReverseDiff Packages (extra)

The ForwardDiff.jl and ReverseDiff.jl packages are exact and fast. (Use the former for functions with few inputs and many outputs, and the latter for vice versa.)

However, they require that your code can handle also other types than Floats. In most cases, that is not a problem, but you may have to watch out if you create arrays to store (intermediate?) results inside the function. See the examples below.

```
using ForwardDiff: ForwardDiff.gradient as ForwardDiff_∇
using ReverseDiff: ReverseDiff.gradient as ReverseDiff_∇
```

```
grad_B = ForwardDiff_∇(z->fn2(z,Σ),w₀)                    #this works
grad_C = ReverseDiff_∇(z->fn2(z,Σ),w₀)

printmat(grad_B,grad_C,2*Σ*w₀;colNames=["ForwardDiff","ReverseDiff","exact"],width=15)
```

```
    ForwardDiff    ReverseDiff          exact
          4.000          4.000          4.000
          9.000          9.000          9.000
```

An Illustration of the Pitfall of ForwardDiff.jl and ReverseDiff.jl

```
function fnDoesNotWork(b,a)
    z = zeros(length(b))              #will not work with ForwardDiff, since
    for i in 1:length(z)              #since z cannot store dual numbers
        z[i] = b[i]*i
    end
    return sum(z) + a
end

function fnDoesWork(b,a)
    z = similar(b)                    #will work with ForwardDiff, since
    z .= 0                            #when b is a dual number, so is z
    for i in 1:length(z)
        z[i] = b[i]*i
    end
    return sum(z) + a
end
```

```
fnDoesWork (generic function with 1 method)
```

```
b0 = [2]              #should be a vector for use with the gradient functions

grad_x1 = ForwardDiff_∇(b->fnDoesWork(b,1),b0)
#grad_x1 = ForwardDiff_∇(b->fnDoesNotWork(b,1),b0)    #uncomment to get an error

grad_x2 = ReverseDiff_∇(b->fnDoesWork(b,1),b0)
#grad_x2 = ReverseDiff_∇(b->fnDoesNotWork(b,1),b0)    #uncomment to get an error

printlnPs(grad_x1,grad_x2)
```

```
     1         1
```

Interpolation

This notebook uses the Interpolations.jl package. As alternatives, consider Dierckx.jl and DataInterpolations.jl.

Load Packages and Extra Functions

```
using Printf, Interpolations

include("src/printmat.jl");
```

```
using Plots
default(size = (480,320),fmt = :png)
```

Interpolation of y = f(x)

Interpolations are particularly useful when *(a)* we repeatedly want to evaluate a function $f(x)$ that is time consuming to calculate but we are willing to accept approximate results, or *(b)* when we only know $f(x)$ for a grid of x values but we are willing to assume that the function is pretty smooth (perhaps even linear over short intervals).

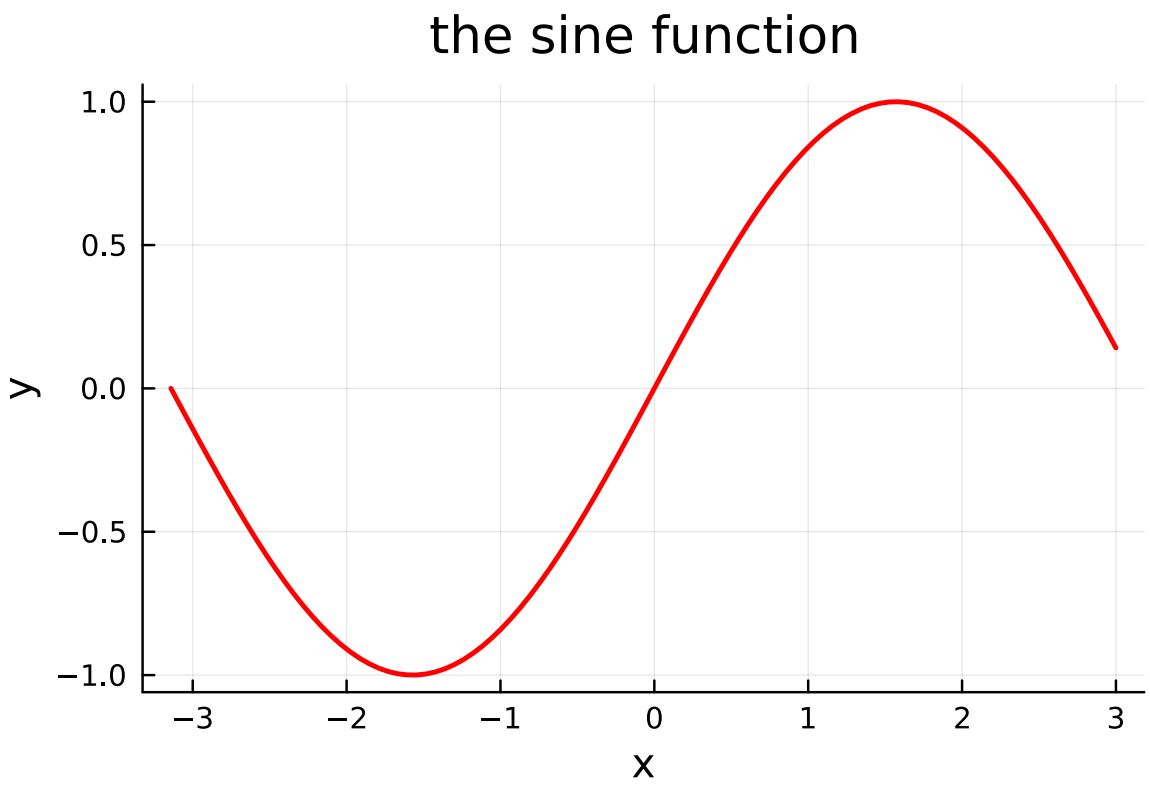
In either case, we do something like this:

1. Calculate $f(x)$ values for a grid of x. This creates a "look-up" table.
2. Replace the calculation of $f(z)$ by interpolating from the "look-up" table.

Some Values to Be Interpolated

As a simple illustration, we interpolate the sine function. (In practice, the interpolation technique is typically applied to more complicated functions.)

```
xGrid = range(-pi,3.0,length=101)  #uniformly spaced grid
yGrid = sin.(xGrid)                #y values at xGrid

p1 = plot( xGrid,yGrid,
           linecolor = :red,
           linewidth = 2,
           legend = nothing,
           title = "the sine function",
           xlabel = "x",
           ylabel = "y" )
display(p1)
```



Interpolate

The next cell calls on `cubic_spline_interpolation()` to create the "look-up" table (more precisely, create an interpolation object).

To use a cubic spline it is required that the x_i grid is *uniformly spaced* (for instance, 0.1,0.2,...). Actually, it has to be a `range`. The case of a non-uniformly spaced x grid is discussed later.

The option `extrapolation_bc=...` determines how extrapolation beyond the range of the x_i grid is done.

The second cell interpolates and extrapolates y at some specific x values.

```
itp = cubic_spline_interpolation(xGrid,yGrid,extrapolation_bc=Flat());
```

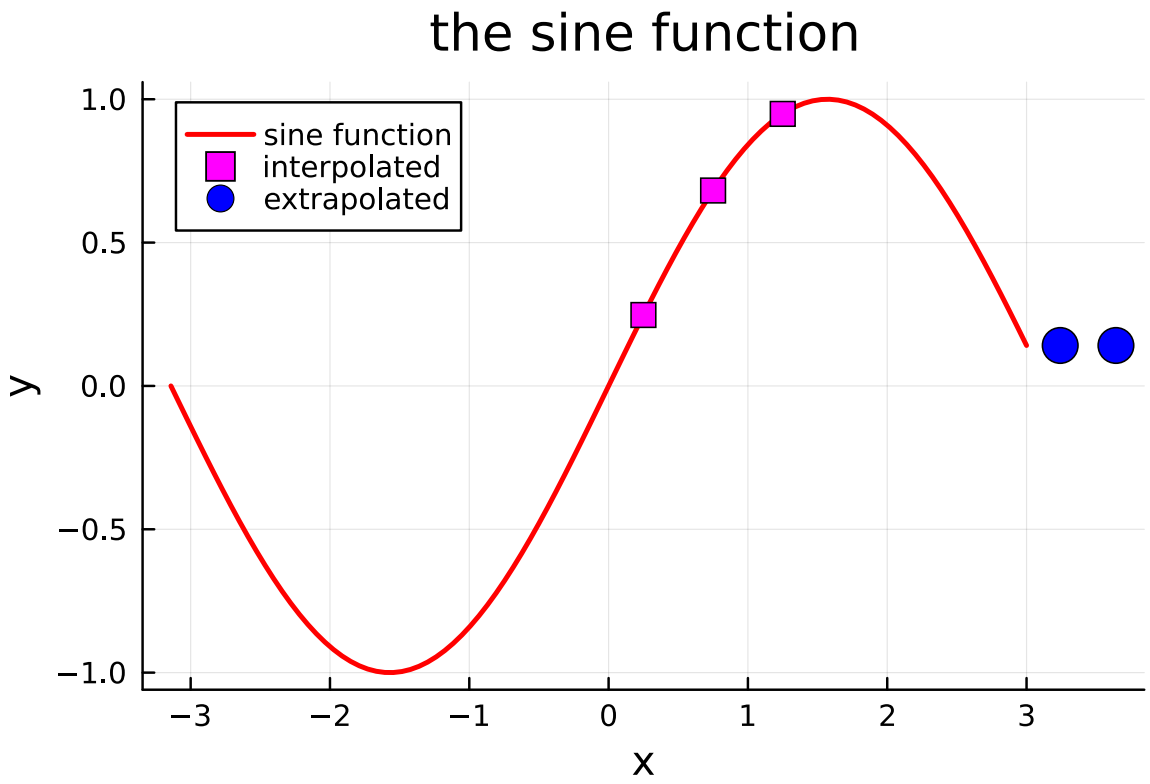
```
x  = [0.25,0.75,1.25]                          #to interpolate the y values at

y_interpolated = itp(x)
printmat([x y_interpolated sin.(x)],colNames=["x","interpolated y","true y"],width=17)

x2             = [pi+0.1,pi+0.5]                 #to extrapolate the y values at
y_extrapolated = itp(x2)

printmat([x2 y_extrapolated],colNames=["x2","extrapolated y"],width=17)
```

```
                x   interpolated y           true y
            0.250            0.247            0.247
            0.750            0.682            0.682
            1.250            0.949            0.949

               x2   extrapolated y
            3.242            0.141
            3.642            0.141
```

Plotting the Results

```
p1 = plot( xGrid,yGrid,
           linecolor = :red,
           linewidth = 2,
           label = "sine function",
           title = "the sine function",
```

```
            xlabel = "x",
            ylabel = "y" )
scatter!(x,y_interpolated,markercolor=:magenta,markersize=5,marker=:square,label="interpolated
scatter!(x2,y_extrapolated,markercolor=:blue,markersize=8,label="extrapolated")
display(p1)
```



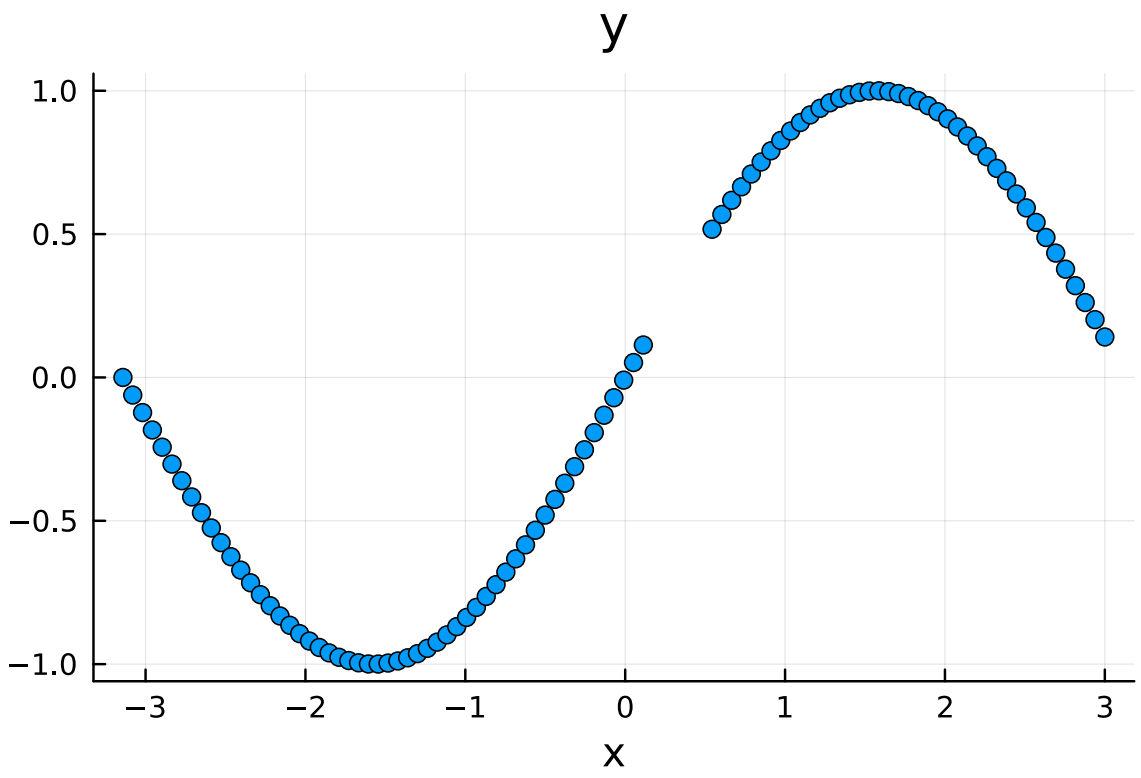
Interpolation of y = f(x) for General x Vectors

That is, when we cannot guarantee that the look-up table of $y_i = f(x_i)$ is from uniformly spaced x_i values (the grid is irregular). This is useful, for instance, when we have empirical data on (x_i, y_i).

The approach works similar to before, except that the `cubic_spline_interpolation` must be replaced by `linear_interpolation`.

```
xGrid2 = deleteat!(collect(xGrid),55:60)          #non-uniformly spaced grid
yGrid2 = sin.(xGrid2)

p1 = scatter( xGrid2,yGrid2,
```

```
              linecolor = :red,
              legend = false,
              title = "y",
              xlabel = "x" )
display(p1)
```




```
itp2 = linear_interpolation(xGrid2,yGrid2,extrapolation_bc=Flat())

y_interpolated = itp2(x)
printmat([x y_interpolated sin.(x)],colNames=["x","interpolated y","true y"],width=17)

y_extrapolated = itp2(x2)
printmat([x2 y_extrapolated],colNames=["x2","extrapolated y"],width=17)
```

```
               x   interpolated y           true y
           0.250           0.241            0.247
           0.750           0.681            0.682
           1.250           0.949            0.949
```

x2	extrapolated y
3.242	0.141
3.642	0.141

Running Python or C Code from Julia

This notebook provides a basic introduction to how to run Python, using the `PyCall.jl` package and also C code (requires no package) from Julia. Please see the PyCall.jl homepage for instructions for how to either use an existing Python installation, or let the package make one.

You need Python's `statsmodels` package installed to run the code below. If you have let PyCall install Python for you, use Conda.jl to add packages: `import Conda; Conda.add("statsmodels")`.

An alternative package (not used here) for running Python is PythonCall.jl, which seems to be gaining popularity.

Another notebooks discusses how to run R code.

```
using Printf, DelimitedFiles
include("src/printmat.jl");
```

Load Data

```
x = readdlm("Data/MyData.csv",',',skipstart=1)  #reading the csv file

(Rme,Rf,R) = (x[:,2],x[:,3],x[:,4])  #creating variables from columns of x
y  = R - Rf                          #do R .- Rf if R has several columns

c = ones(length(Rme))
x = [c Rme]

b = x\y
println("OLS coeffs according to Julia")
printmat(b)
```

```
OLS coeffs according to Julia
    -0.504
     1.341
```

Python

In the next cells we *(a)* load the PyCall.jl package and activates the (Python) package `statsmodels`; *(b)* call some functions (eg. `OLS()`) from statsmodels.

```
using PyCall
sm = pyimport("statsmodels.api");     #activate this package and call it `sm`
```

```
resultsP = sm.OLS(y, x).fit()        #can use Python functions directly

println(resultsP.summary())
```

```
PyObject <class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.519
Model:                            OLS   Adj. R-squared:                  0.518
Method:                 Least Squares   F-statistic:                     416.2
Date:                Mon, 25 Nov 2024   Prob (F-statistic):           2.72e-63
Time:                        09:42:56   Log-Likelihood:                -1241.7
No. Observations:                 388   AIC:                             2487.
Df Residuals:                     386   BIC:                             2495.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.5042      0.305     -1.654      0.099      -1.103       0.095
x1             1.3410      0.066     20.401      0.000       1.212       1.470
==============================================================================
Omnibus:                      259.682   Durbin-Watson:                   1.870
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             5249.944
Skew:                           2.482   Prob(JB):                         0.00
Kurtosis:                      20.323   Cond. No.                         4.68
==============================================================================
```

```
Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
"""
```

```
println(keys(resultsP))              #print all keys (field names)
```

```
[:HC0_se, :HC1_se, :HC2_se, :HC3_se, :_HCCM, :__class__, :__delattr__, :__dict__, :__dir__, :__do
```

```
b_P = resultsP.params                #the numerical results are now a Julia vector

printblue("Comparing the estimates in Julia and Python:")
printmat([b b_P];colNames=["Julia","Python"])
```

```
Comparing the estimates in Julia and Python:
     Julia    Python
    -0.504    -0.504
     1.341     1.341
```

```
#we can run blocks of code like this, notice: $x and $y
py"""
import numpy as np
xx = np.matmul(np.matrix.transpose($x),$x)
xy = np.matmul(np.matrix.transpose($x),$y)
b_p = np.linalg.solve(xx,xy)
"""

py"b_p"               #to print the result
```

```
2-element Vector{Float64}:
 -0.5041626034967046
  1.3410486453848383
```

C

This section shows some simple examples of how to call a C function. The functions are in the file `My_C_Stuff.c` (printed in the next cell). The first function `c_dot` defines a dot product between two vectors and the second function `c_ols` a simple linear regression.

```
println(read("Data/My_C_Stuff.c",String))
```

```
#include <stddef.h>

// calculate the inner (dot) product of vectors Y and Y, returns the result (Sxy)
double c_dot(size_t n, double *Y, double *X) {
    double Sxy = 0.0;
    for (size_t i = 0; i < n; ++i) {
        Sxy += X[i]*Y[i];
    }
    return Sxy;
}

// calculate a simple regression, Y = a + b*X + u, puts (a,b) in vector ab, returns nothing
void c_ols(size_t n, double *Y, double *X, double *ab) {
    double Sx = 0.0, Sy = 0.0, Sxx = 0.0, Sxy = 0.0;
    for (size_t i = 0; i < n; ++i) {
        Sx  += X[i];
        Sy  += Y[i];
        Sxx += X[i]*X[i];
        Sxy += X[i]*Y[i];
    }
    ab[1] = (Sxy-Sx*Sy/n)/(Sxx-Sx*Sx/n);   //slope
    ab[0] = (Sy - ab[1]*Sx)/n;             //intercept
}
```

To compile to a dynamlic library (dll on windows), I use gcc (for x86_64) from mingw-64 and run the following in the mingw terminal

```
gcc -shared -fPIC My_C_Stuff.c -o My_C_Stuff.dll
```

To call the C functions, place the dll file in the current folder and then run the following cells.

```
mylibc = "My_C_Stuff.dll"
x2     = x[:,2];               #get a vector with the regressor values
```

A Function which Returns a Number

In the next example, we a function `c_dot` in `My_C_Stuff.dll`. The function calculates the inner product of two vectors.

The details are: 1. `mylibc.c_dot` is the library.function 2. `length(y)::Csize_t` is the first input and its type (an integer indicating the number of elements in y) 3. `y::Ptr{Float64}` is the second input (a pointer to an array of Floats) and similarly for `x2` 4. `Float64` is the type of the output

(We could potentially wrap this in a Julia function that checks for the right input types and outputs the result.)

```
z = @ccall mylibc.c_dot(length(y)::Csize_t, y::Ptr{Float64}, x2::Ptr{Float64})::Float64

printlnPs("The inner product of x2 and y in Julia and C:  ",x2'y," ",z)
```

```
The inner product of x2 and y in Julia and C:   11071.648           11071.648
```

A Function which Returns a Vector

The details are as above, except that 1. `mylibc.c_ols` is the library.function 2. `Cvoid` is the type of the output, which here indicates that the function does not have an output. Rather, the function modifies the vector `b_c` by putting the OLS results there.

```
b_c = zeros(2)          #where C will store the regression results

@ccall mylibc.c_ols(length(y)::Csize_t, y::Ptr{Float64}, x2::Ptr{Float64}, b_c::Ptr{Float64}):

println("Comparing the estimates in Julia and C")
printmat([b b_c];colNames=["Julia","C"])
```

```
Comparing the estimates in Julia and C
     Julia         C
    -0.504    -0.504
     1.341     1.341
```

Running R Code from Julia

This notebook provides a basic introduction to how to run R code, using the RCall.jl package, from Julia. See the the RCall.jl homepage (the installation instructions) for how to use your existing R installation or for letting RCall.jl install R for you.

```
using Printf, DelimitedFiles
include("src/printmat.jl");
```

Load Data

```
x = readdlm("Data/MyData.csv",',',skipstart=1)  #reading the csv file

(Rme,Rf,R) = (x[:,2],x[:,3],x[:,4])  #creating variables from columns of x
y  = R - Rf                          #do R .- Rf if R has several columns

c = ones(length(Rme))
x = [c Rme]

b = x\y
println("OLS coeffs according to Julia")
printmat(b)
```

```
OLS coeffs according to Julia
    -0.504
     1.341
```

R

The next few cells load the RCall.jl package, transfers the data from Julia to R (`@rput`) and then run an OLS regression (`reval()`).

```
using RCall
```

```
@rput x y                           #send x and y to R

resultsR = reval("summary(mod <- lm(y ~ x-1))")  #run R code
println(resultsR)                                #and print output
```

```
RObject{VecSxp}

Call:
lm(formula = y ~ x - 1)

Residuals:
    Min      1Q  Median      3Q     Max
-17.981  -3.131  -0.359   2.281  52.361

Coefficients:
   Estimate Std. Error t value Pr(>|t|)
x1 -0.50416    0.30483  -1.654    0.099 .
x2  1.34105    0.06573  20.401   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.954 on 386 degrees of freedom
Multiple R-squared:  0.5194,    Adjusted R-squared:  0.5169
F-statistic: 208.6 on 2 and 386 DF,  p-value: < 2.2e-16
```

```
println(names(resultsR))            #print all keys (field names)
```

```
[:call, :terms, :residuals, :coefficients, :aliased, :sigma, :df, Symbol("r.squared"), Symbol("ad
```

```
b_R = rcopy(resultsR[:coefficients])  #the numerical results are now a Julia array

printblue("Comparing the estimates in Julia and R")
printmat([b b_R[:,1]];colNames=["Julia","R"])
```

```
Comparing the estimates in Julia and R
     Julia         R
    -0.504    -0.504
     1.341     1.341
```

```
R"model <- lm(y ~ x-1)"                                #an alternative approach
resultsR2 = rcall(:summary, R"model")
b_R2      = rcopy(resultsR2[:coefficients])
printmat([b b_R[:,1] b_R2[:,1]];colNames=["Julia","R","R ver, 2"])
```

```
     Julia         R  R ver, 2
    -0.504    -0.504    -0.504
     1.341     1.341     1.341
```

```
#we can also run blocks of code like this
#do @rput x y or use $x and $y in the code below

#solve x'x*b = x'y
R"""
xx <- t(x)%*%x
xy <- t(x)%*%y
b_R <- solve(xx,xy)
"""
```

```
RObject{RealSxp}
           [,1]
[1,] -0.5041626
[2,]  1.3410486
```

Interactive Usage of R

If you are using the Julia REPL ("command prompt"), then you can work more interactively with R. Unfortunaterly, that does not work in a notebook. The following trivial example illustrates the idea (`>julia` means that you are at the julia prompt and `R>` at the R prompt).

```
julia> x = 1;y=2
julia> @rput x y
julia> $         #to switch over to R
R> z <- x + y
R> hit ESC       #to switch over to Julia
julia> @rget z
```

At this point you can work with z in Julia.

Threading

means using the multiples cores that modern CPUs have. Some things in Julia are automatically threaded, for instance, many linear algebra computations and the package manager. Also, many packages use are threaded. However, to get your own code threaded, there is some work to do (and some pitfalls to avoid).

The Simplest Kind of Threading

in Julia is of this type:

```
using Base.Threads: @threads

Z = zeros(N)
@threads for i in 1:N
    local y
    y = SomeFunction(i,W)
    Z[i] = SomeOtherFunction(i,X,y,Z[i],but not Z[j])
end
U = YetAnotherFunction(Z)
```

In this case, `Z` is used for storing the results. Variables that are created in the loop (like `y`) should be declared `local` to avoid that they are shared across threads. Notice that is important to *not* let iteration `i` depend on results created by another iteration (`Z[j]`).

The `@threads` is a simple approach that works well when the iterations are similar (uniform). In contrast, if some iterations are much more costly or you want to use threads in a nested setting, the `@spawn/fetch/@sync` might be better (not discussed in this notebook).

Threading typically only pays off when the iterations involve heavy computations. Otherwise, the cost of setting of the threading might dominate. It is also important to avoid too many allocations (for instance, creating and deleting arrays) inside the threaded loop, or otherwise the memory allocation system ("gc") will impact the performance.

Activating Threading

Julia is (currently, as of version 1.10) started with a single thread. To change that default, set the environment variable `JULIA_NUM_THREADS=4` (it is often recommended to set it to the number of physical CPU cores).

In case you cannot change the environment variable, or simply want more fine-grained control, consider the following:

1. start julia from a command line as `julia --threads=4`
2. Set up jupyter/lab by first doing the following in the REPL

```
using IJulia
installkernel("Julia (4 threads)", env=Dict("JULIA_NUM_THREADS"=>"4"))
```

and then pick the right kernel when running the notebook.

3. In VScode, go to the Julia extension settings and search for `threads`. Then, change to `"julia.NumThreads": 4`

```
using Printf
using Base.Threads: @threads         #load the threading functions

include("src/printmat.jl");
```

```
println("Number of threads: ", Threads.nthreads())   #check how many threads that are availabl
```

```
Number of threads: 4
```

Comparing w/wo Threads

We will calculate

$\mu_t = \lambda\mu_{t-1} + (1-\lambda)x_{t-1}$, for $t = 2...T$

on each column of a large matrix (t refers to rows).

The function `ExpMA(x,λ)` does the calculation for a vector (that is, a column in a bigger matrix). `ExpMA_1()` and `ExpMA_2()` loops over the columns without and with threading, respectively.

```julia
function ExpMA(x,λ)             #exponential MA, creates a vector
    T = length(x)
    μ  = zeros(T)
    for t in 2:T
        μ[t]  = λ*μ[t-1]  + (1-λ)*x[t-1]
    end
    return μ
end
```

```
ExpMA (generic function with 1 method)
```

```julia
function ExpMA_1(X,λ)          #wrap ExpMA(X,λ) in loop over columns of X
    (T,N) = (size(X,1),size(X,2))
    result = zeros(T,N)
    for i in 1:N
        result[:,i] = ExpMA(X[:,i],λ)
    end
    return result
end

function ExpMA_2(X,λ)         #wrap but with threaded loop
    (T,N) = (size(X,1),size(X,2))
    result = zeros(T,N)
    @threads for i in 1:N
        result[:,i] = ExpMA(X[:,i],λ)
    end
    return result
end
```

```
ExpMA_2 (generic function with 1 method)
```

```julia
λ = 0.94
T = 100_000

N = 500
X = randn(T,N)

Y1 = ExpMA_1(X,λ)
Y2 = ExpMA_2(X,λ)

println("Test if the same results (Y1==Y2): ", Y1==Y2)
```

```
Test if the same results (Y1==Y2): true
```

Avoiding (Memory) Allocations

is often a good idea. But, it becomes perhaps even more important in threaded applications.

The next cell defines a new function `ExpMAx!()` which writes the result to an existing matrix (the first function argument, μ) instead of creating a new vector every time. Then, the function `ExpMAx_2()` does a threaded loop over the columns in `X`. We verify that we get the same result as before.

```
function ExpMAx!(μ,x,λ,i)      #exponential MA, writes to column in an existing matrix μ
    T = size(x,1)
    for t in 2:T
        μ[t,i]  = λ*μ[t-1,i]  + (1-λ)*x[t-1,i]
    end
    return μ
end

function ExpMAx_1(X,λ)             #no threading, but create
    (T,N) = (size(X,1),size(X,2))  #fewer intermediate vectors
    result = zeros(T,N)
    for i in 1:N
        ExpMAx!(result,X,λ,i)
    end
    return result
end


function ExpMAx_2(X,λ)             #wrap but with threaded loop and creating
    (T,N) = (size(X,1),size(X,2))  #fewer intermediate vectors
    result = zeros(T,N)
    @threads for i in 1:N
        ExpMAx!(result,X,λ,i)
    end
    return result
end

Y3 = ExpMAx_1(X,λ)
Y4 = ExpMAx_2(X,λ)
println("Test if the same results (Y1==Y3==Y4): ", Y1==Y3==Y4)
```

```
Test if the same results (Y1==Y3==Y4): true
```

Speed Comparison

without thread, with threads (but creating intermediater vectors) and with threads but no intermediate vectors.

```
using BenchmarkTools           #package for benchmarking

printblue("standard loop:")
@btime ExpMA_1($X,λ)           #use $X to get more accurate timing

printblue("threaded loop:")
@btime ExpMA_2($X,λ)

printblue("standard loop, but without intermediate vectors:")
@btime ExpMAx_1($X,λ)

printblue("threaded loop, but without intermediate vectors:")
@btime ExpMAx_2($X,λ)

printmagenta("\n...reducing memory allocations and using threads can both speed up the code")
```

```
standard loop:
  345.360 ms (3003 allocations: 1.12 GiB)
threaded loop:
  135.887 ms (3027 allocations: 1.12 GiB)
standard loop, but without intermediate vectors:
192.443 ms (3 allocations: 381.47 MiB)
threaded loop, but without intermediate vectors:
  90.615 ms (25 allocations: 381.47 MiB)

...reducing memory allocations and using threads can both speed up the code
```

Things that Could Go Wrong with Threading (extra)

are often related to data races (different threads writing to the same memory location).

Case 1: Several Threads Changing the Same Elements of an Array

When threads write to the same memory locations, anything can happen.

```
N = 10_000

x = [0]
@threads for i in 1:N
  x[1] = x[1] + 1            #all threads writing to x[1]
end

println("This ought to be $N, but it is ",x[])
```

```
This ought to be 10000, but it is 3222
```

Case 2: Threads Writing to BitArrays

Code like in the next cell can also create unexpected results since the threads are trying write to the same BitArray (which has a packed format). Run a few times to see tha problem.

This is solved by instead using `Z = fill(false,N)` which is an array of Bools.

```
N = 10
Z = falses(N)
#Z = fill(false,N)              #ucomment to solve the problem
@threads for i in 1:N
    Z[i] = true
    sleep(0.5)                   #give the thread something to do
end

println("All $N values in Z should be `true`, but only ", sum(Z)," are. If needed, rerun to se
```

```
All 10 values in Z should be `true`, but only 10 are. If needed, rerun to see.
```

Case 3: @threads and Variable Scope

Code like this

```
v = 1:2
@threads for i in 1:N
    v = something
    x = SomeFunction(v)
end
```

can create unexpected results since the threads are sharing `v`. This is solved by declaring `v` inside the loop to be `local`.

```
using LinearAlgebra

function f2(N)
  v = falses(N+1)
  x = zeros(Int,N,N)
  @threads for i = 1:N
    #local v                   #uncomment to solve the problem
    v    = falses(N)
    v[i] = true
    x[v,i] .= i
  end
  return x
end

println("This should always be zero. Run a few times to check if that holds.\n")
M = 100
dev = zeros(M)
for i = 1:M
  dev[i] = maximum(abs,f2(i) - diagm(1:i))
end
println("All $M values should be 0, but only ", sum(dev.==0), " are.")
```

```
This should always be zero. Run a few times to check if that holds.

All 100 values should be 0, but only 69 are.
```

Projects and Modules

This notebook shows how to create your projects and modules (kind of a package).

A Module

...is, for instance, a collection of your own functions that you often use.

One of the advantages with a module is that you can do `using .TutorialModule` instead of having to `include()` every file. It is also likely to be quicker.

Step 1: Create a Module File

Create a file that defines the module.

The next cell shows the contents of the module file. Change it, for instance, by exporting also `printred`.

```
printstyled("the contents of src/TutorialModule.jl:\n\n";color=:blue,bold=true)
println(read("src/TutorialModule.jl",String))
```

```
the contents of src/TutorialModule.jl:

module TutorialModule

import Printf                                   #the module can use other packages
#import LinearAlgebra

export printmat, printlnPs, printblue, printTeXTable  #available after `using .TutorialModule`

include("printmat.jl")                                 #files with the functions
include("printTeXTable.jl")                   #could also type in the code here
```

```
end
```

Step 2: Include and Use the Module

```
include("src/TutorialModule.jl")
using .TutorialModule        #notice the dot(.)
```

```
x = [11 12;21 22]

printmat(x)                 #the printmat() function should now be available
```

```
    11        12
    21        22
```

Steps 0-2: Alternative Approach to Creat/Use a Module (extra)

You should probably **restart the Julia kernel** before running the cells below, since we are going to load the same module, but using another apporach.

0. Tell Julia where to find the files for the module by adding to `LOAD_PATH`. This has to be redone every time you run code using the module. (For instance, by putting it in your `startup.jl` file.)
1. use the same module file as above
2. run `using TutorialModule` (without the dot)

This approach has the advantage that the module will be available from everywhere and also pre-compiled.

```
PathToMyFolder = joinpath(pwd(),"src")  #change as needed
push!(LOAD_PATH,PathToMyFolder);
```

```
using TutorialModule         #no dot

x = [11 12;21 22]

printmat(x)                 #the printmat() function should now be available
```

```
    11        12
    21        22
```

A Project

A project is folder where you can make specific package installations, without affecting other projects (or Julia, more generally).

You can also define a module in that project, using one of the approaches discussed before, or by using the `Package` approach dicussed later on.

Step 1: Create the Project

You should probably **restart the Julia kernel** before running the cells below.

Create the project folder (`LitteProject`).

```
import Pkg
mkdir("LittleProject")      #run once to create the project folder
```

```
"LittleProject"
```

Step 2: Use the Project

You should probably **restart the Julia kernel** before running the cells below.

1. Copy this notebook (or create a new one with at least the next few cells) to the project folder.
2. Move to the project folder. (In VS code, remember to close the current folder and open up the new.)
3. Do `Pkg.activate(".")` to activate the project. Needs to be done every time you work with that project
4. Optionally add packages. Because of point 2, the package installations will now be to the his project, and not to the general Julia environment.
5. run code...

```
file = "Tutorial_28_Modules_Projects.ipynb"
dir = "LittleProject"
cp(file,joinpath(dir,file);force=true)
println("contents of $dir: ",readdir(dir))

printstyled("\nMove to the project and continue working from there.";color=:red,bold=true)
```

```
contents of LittleProject: ["Tutorial_28_Modules_Projects.ipynb"]

Move to the project and work from there.
```

```
#run this in the LittleProject folder
import Pkg
Pkg.activate(".")              #always run this
Pkg.add("Printf")              #optionally run to install packages
```

```
using Printf                   #use the (locally) installed package, run some code

fmt = Printf.Format("%10.3f")
str = Printf.format(fmt,pi)
println(str)
```

```
     3.142
```

A Local Package (Project + Module)

A local package is a project with a subfolder `src` containing a module file with the same name as the project.

You should probably **restart the Julia kernel** before running the cells below.

Step1: Create the Package

To create a package in a (non-existing) subfolder `LittlePackage`, do `Pkg.generate("LittlePackage")`. This creates the subfolder, with `.toml` files (they define the packages you have added to the project—so far none) and a `src` subfolder with a skeleton module file.

```
import Pkg
Pkg.generate("LittlePackage")          #uncomment to run this once
```

```
  Generating  project LittlePackage:
    LittlePackage\Project.toml
    LittlePackage\src\LittlePackage.jl

Dict{String, Base.UUID} with 1 entry:
  "LittlePackage" => UUID("02a44c26-f219-4aec-a305-47be40cf3ac0")
```

Step 2: Transfer Files to the Package

...both this notebook and files for setting up a module

1. Copy this notebook (or create a new one with at least the next few cells) to the project folder. Move there. (In VS code, remember to close the current folder and open up the new.)
2. Copy the `LittlePackage.jl`, `printmat.jl`, and `printTeXTable.jl` to `LittlePacke.src`. Yes, overwrite the existing file there.

```
file = "Tutorial_28_Modules_Projects.ipynb"
dir = "LittlePackage"
cp(file,joinpath(dir,file);force=true)

files = ["LittlePackage.jl","printmat.jl","printTeXTable.jl"]
for file in files
  cp(joinpath("src",file),joinpath(dir,"src",file);force=true)
end

println("contents of $dir: ",readdir(dir))

printstyled("\nMove to the project and and continue working from there.";color=:blue,bold=true
```

```
contents of LittlePackage: ["Project.toml", "Tutorial_28_Modules_Projects.ipynb", "src"]

Move to the project and and continue working from there.
```

Step 3: Use the Package

1. Do `Pkg.activate(".")` to activate the project. Needs to be done every time you work with that project
2. Optionally add packages. Because of point 2, the package installations will now be to the his project, and not to the general Julia environment.
3. run some code
4. `using LittlePackage` and the run functions from the module `LittlePackage`

```
#run this in the LittlePackage folder
import Pkg
Pkg.activate(".")              #always run this
Pkg.add("Printf")              #optionally run to install packages
```

```
using LittlePackage             #use the package

printmat(randn(4,3))
```

```
      0.908     1.045    -0.502
     -0.386     0.002    -1.376
      0.014     0.824    -0.794
     -0.196    -1.897     0.138
```

PyPlot

This file demonstrates how to create plots in Julia by using the PyPlot.jl package.

PyPlot relies on the matplotlib library, which is part of Python. If you have Python installed, then it will be used as is. Otherwise, see PyPlot's homepage for instructions on how to install.

PyPlot seems to behave better (on Windows, at least), if you run `ENV["MPLBACKEND"]="TkAgg"` before running `using PyPlot`. One possibility is to put that command in the startup.jl file.

Collections of examples are available at Plot Examples and the Julia Plots Gallery.

As an alternative, consider PythonPlot.jl. It is mostly a drop-in replacement of PyPlot.jl (a few small differences), but uses another approach for calling on the matplotlib library.

Load Packages and Extra Functions

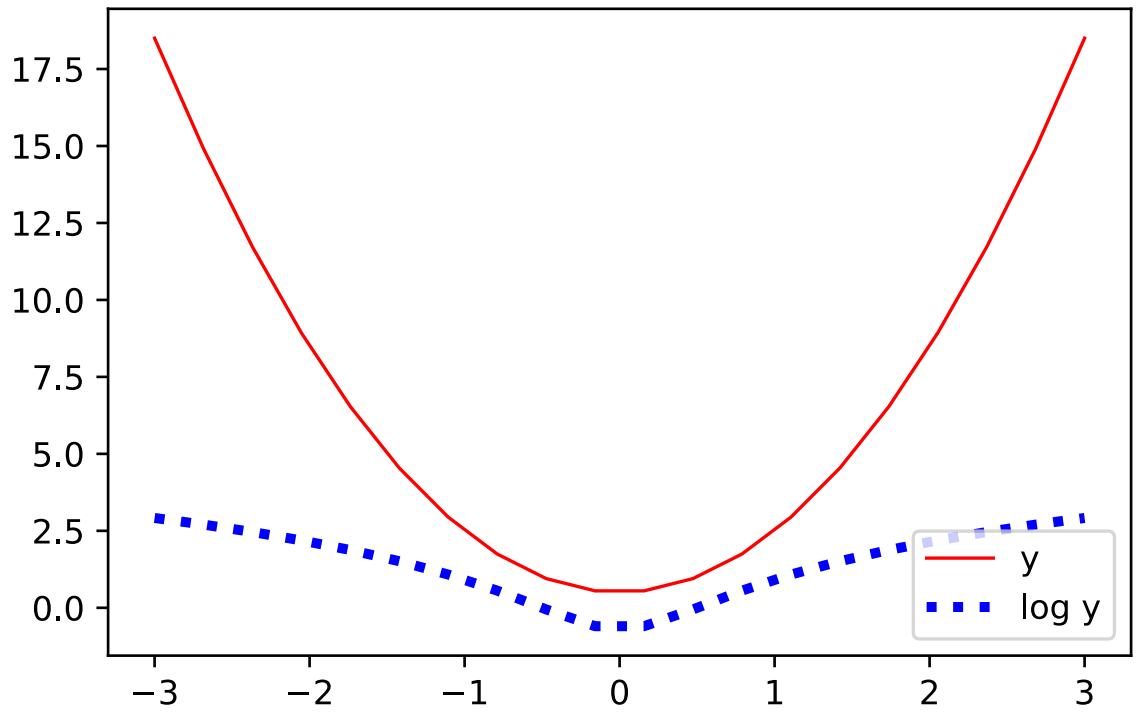
```
using Dates

using PyPlot
PyPlot.svg(true);          #prettier figures if `true`, but Github may not render it
```

A First Plot

The next figure shows several curves.

The subsequent figure adds a title, axis labels and legends. Text and font sizes are illustrated. The axis limits and the tick marks are set manually.

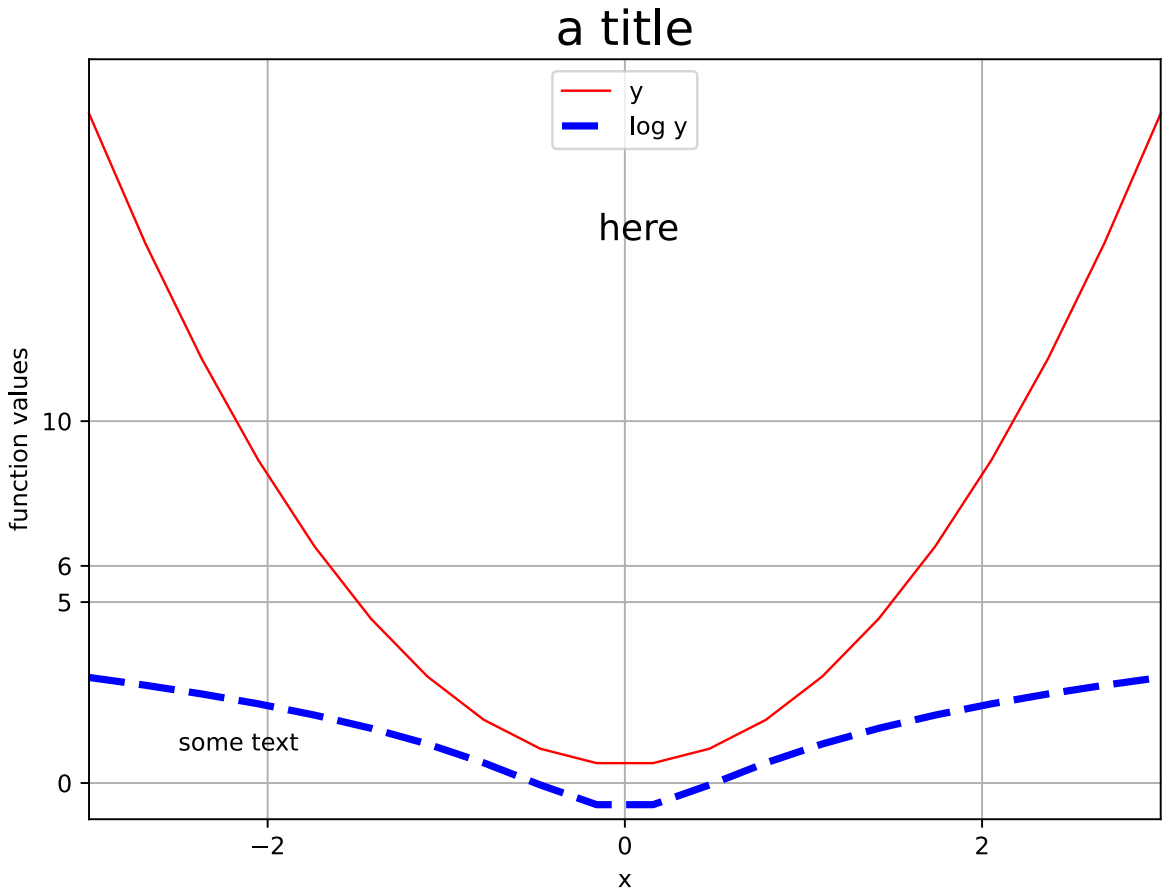
```
x = range(-3,3,length=20)
y = 2*x.^2 .+ 0.5

figure(figsize=(5.5,3.5))       #width and height, in inches
  plot(x,y,linestyle="-",color="r",linewidth=1.0,label="y")
  plot(x,log.(y),linestyle=":",color="b",linewidth=3.0,label="log y")
  legend(loc="lower right")
#display(gcf())              #uncomment if the plot is not shown (eg. in REPL or VS Code)
```




```
PyObject <matplotlib.legend.Legend object at 0x000001641568B9650>
```

```
#now with title, labels and more - and larger

figure(figsize=(8,5.71))
  plot(x,y,linestyle="-",color="r",linewidth=1.0,label="y")
  plot(x,log.(y),linestyle=(0,(5,2)),color="b",linewidth=3.0,label="log y")
  xticks([-2;0;2])
  yticks([0;5;6;10])
  grid(true)
```

```
  title("a title",fontsize=20)
  xlim(-3,3)               # set limits of the x-axis
  ylim(-1,20)              # set limits of the y-axis
  xlabel("x")
  ylabel("function values")
  text(-2.5,0.9,"some text")
  text(-0.15,15,"here",fontsize=15)
  legend(loc="upper center")
savefig("AFirstPlot.pdf")      #save pdf file of the plot
#display(gcf())
```

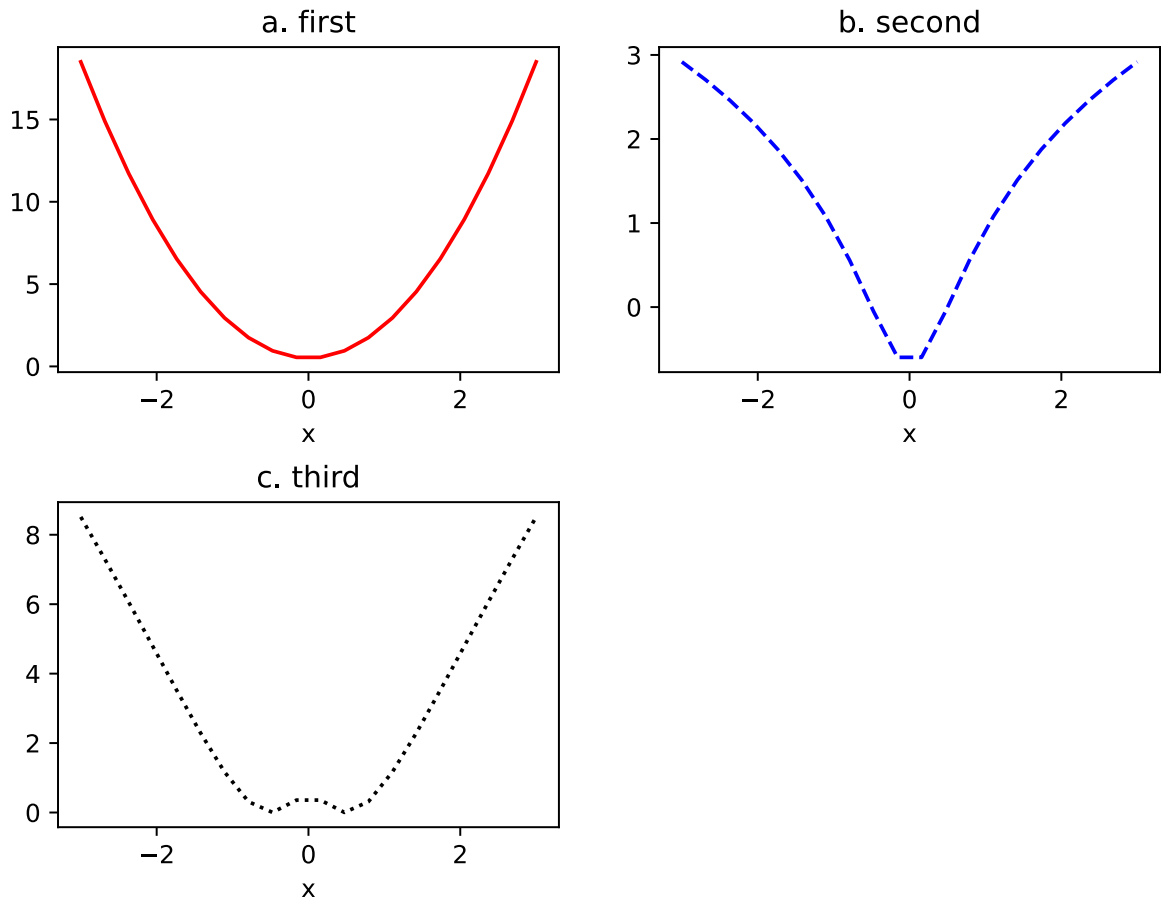


Subplots

are easily constructed using the `subplot()` command. For a the first subplot in a 2x2 grid, use `subplot(2,2,1)`.

The `fig.set_size_inches(16,10.7,forward=false)` before the `savefig()` command (see below) makes sure the size of the saved graphics is not cut by my (smallish) screen size.

```
fig = figure(figsize=(8,5.7))                       #subplots
subplot(2,2,1)
  plot(x,y,"r-")
  title("a. first")
  xlabel("x")
subplot(2,2,2)
  plot(x,log.(y),"b--")
  title("b. second")
  xlabel("x")
subplot(2,2,3)
  plot(x,log.(y).^2,"k:")
  title("c. third")
  xlabel("x")
subplots_adjust(hspace = 0.4)                  #to give more vertical space between "row" 1 and

#fig.set_size_inches(16,10.7,forward=false)    #to save really large figure
#savefig("ASecondPlot.pdf")
#display(gcf())
```

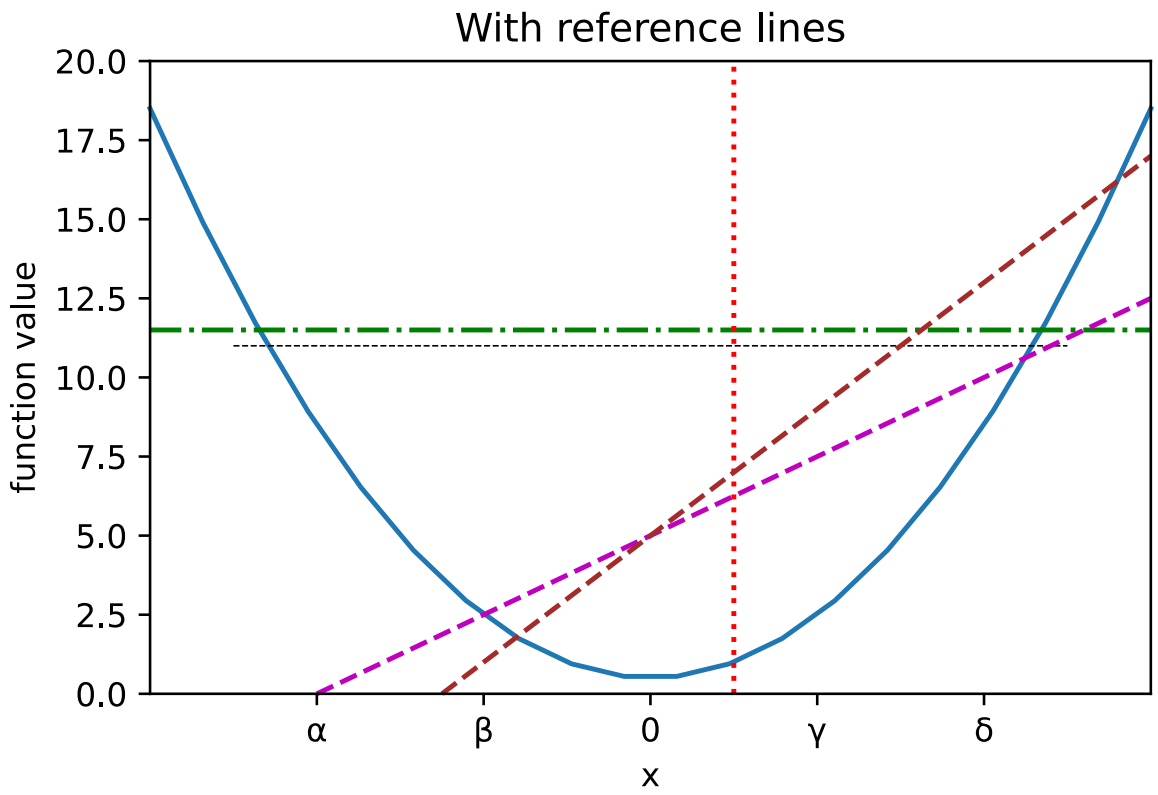


Adding Lines

and changing the x-tick marks.

```
figure(figsize=(5.5,3.5))
  plot(x,y)
  xlim(-3,3)
  ylim(0,20)
  title("With reference lines")
  xlabel("x")
  ylabel("function value")
  xticks(-2:2,["α","β","0","γ","δ"])
  hlines(11,-2.5,2.5,linestyle="--",linewidth=0.5,color="black")   #stretches over x=[-2.5;2.5
  axhline(11.5,linestyle="-.",color="g")             #stretches over all x
```

```
  vlines(0.5,0,20,linestyle=":",color="r")
  axline((0,5),(2,10),linestyle="--",color="m")
  axline((0,5),slope=4,linestyle="--",color="brown")
#display(gcf())
```



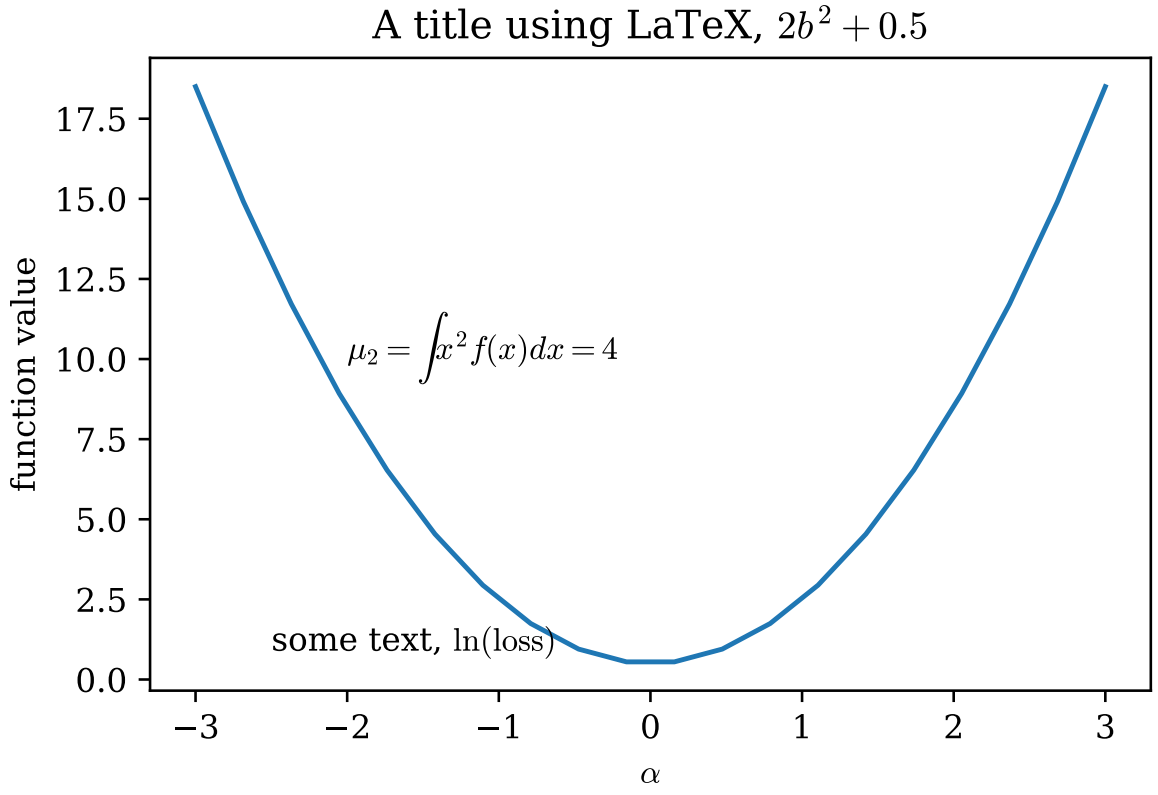

```
PyObject <matplotlib.lines.AxLine object at 0x0000016417703B90>
```

LaTeX in the Figure

PyPlot calls on LaTeXStrings so you can us, for instance, `xlabel(L"$\alpha$")` to get the x label in LaTeX. You should probably also change the general font to "cm" (see below) to make the various text elements look more similar.

To insert the value of the variable z into the LaTeX string ("string interpolation"), use `%$z` instead of the standard `$z`.

```
rcParams = PyPlot.PyDict(PyPlot.matplotlib."rcParams")  #allows you to make global changes
rcParams["mathtext.fontset"] = "cm"             #setting the math font
rcParams["font.family"]      = "serif"          #setting the text font family

z = 4                               #a value to be used inside the string

figure(figsize=(5.5,3.5))
  plot(x,y)
  title(L"A title using LaTeX, $2 b^2 + 0.5$")
  xlabel(L"$\alpha$")
  ylabel("function value")
  text(-2.5,0.9,L"some text, $\ln(\mathrm{loss})$")
  text(-2.0,10,L"$\mu_2 = \int x^2 f(x) dx = %$z $")        #notice: %$z
#display(gcf())
```

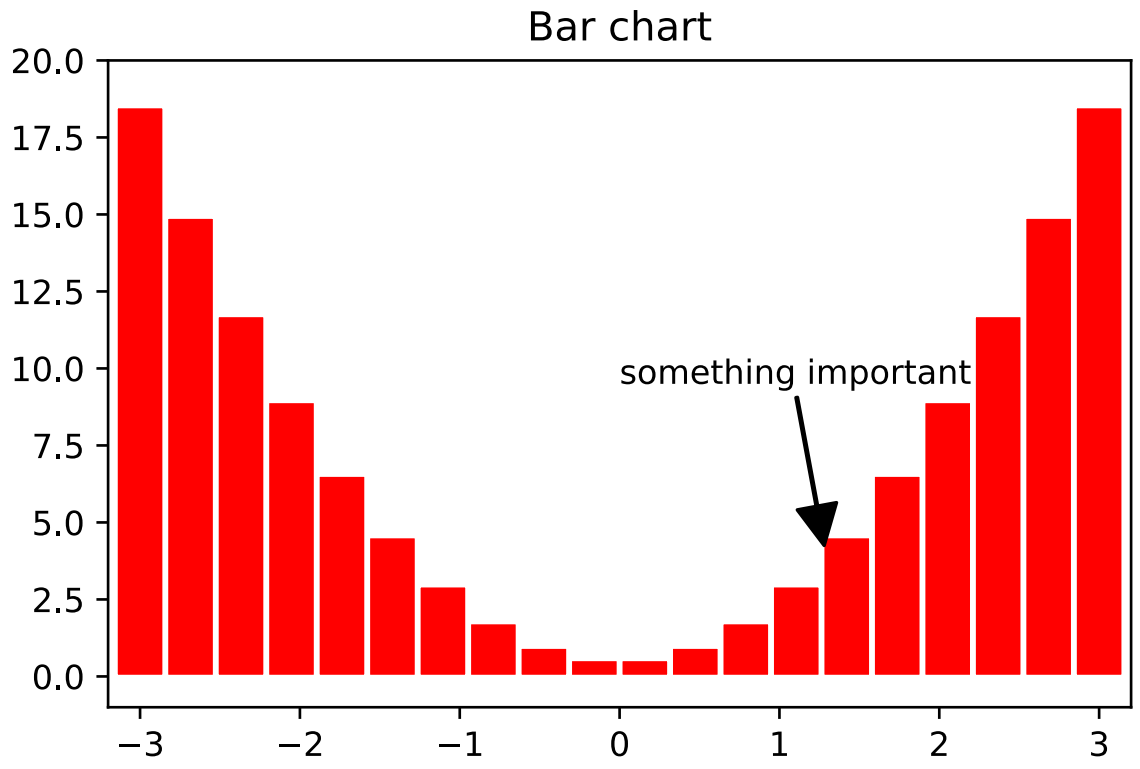



```
PyObject Text(-2.0, 10, '$\\mu_2 = \\int x^2 f(x) dx = 4 $')
```

Bars and Stairs

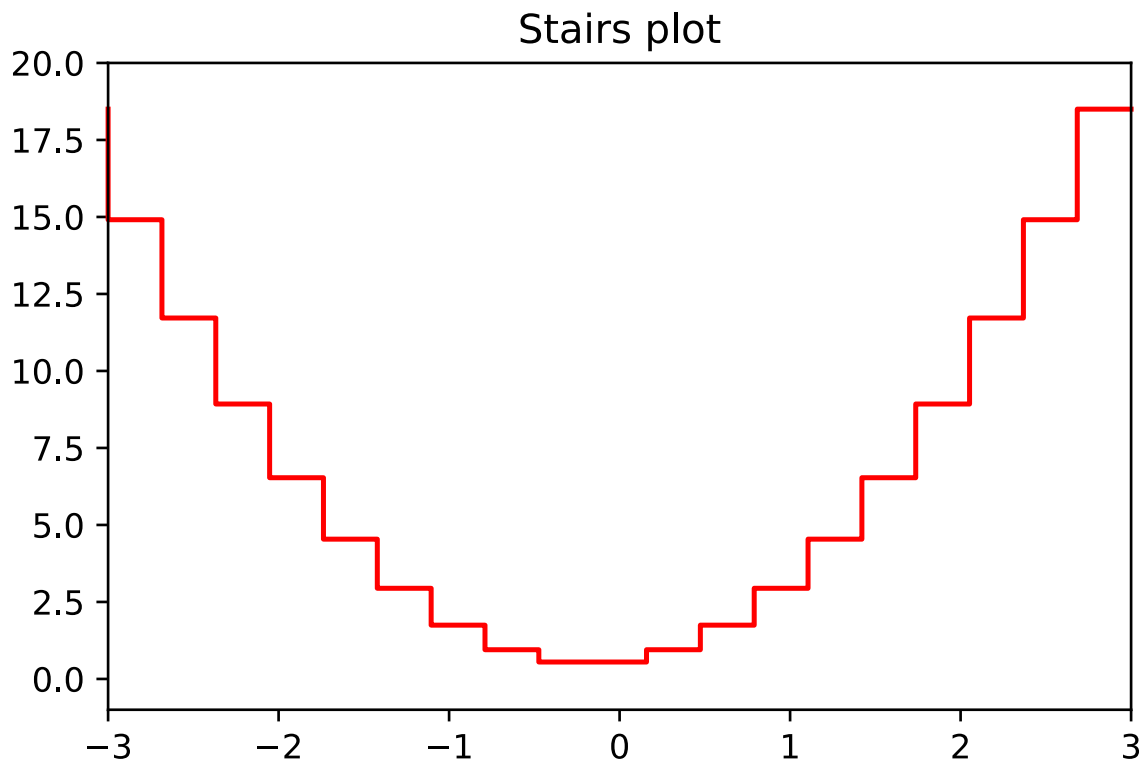
(with some annotations)

```
rcParams["mathtext.fontset"] = "dejavusans"            #resetting the fonts
rcParams["font.family"] = "sans-serif"                 #after the LaTeXing
rcParams["font.size"] = 10

figure(figsize=(5.5,3.5))
  bar(x,y,facecolor="red",edgecolor="white",align="center",width=0.3)
  xlim(-3.2,3.2)
  ylim(-1,20)
  title("Bar chart")
  txt = "something important"
  annotate(txt,xy=[0.7;0.25],xycoords="axes fraction",xytext=[0.5;0.5],
           textcoords="axes fraction",arrowprops=Dict("facecolor"⇒"black","width"⇒0.5))
#display(gcf())
```




```
PyObject Text(0.5, 0.5, 'something important')
```

```
figure(figsize=(5.5,3.5))
  step(x,y,linewidth=1.5,color="r")
  xlim(-3,3)
  ylim(-1,20)
  title("Stairs plot")
#display(gcf())
```




```
PyObject Text(0.5, 1.0, 'Stairs plot')
```

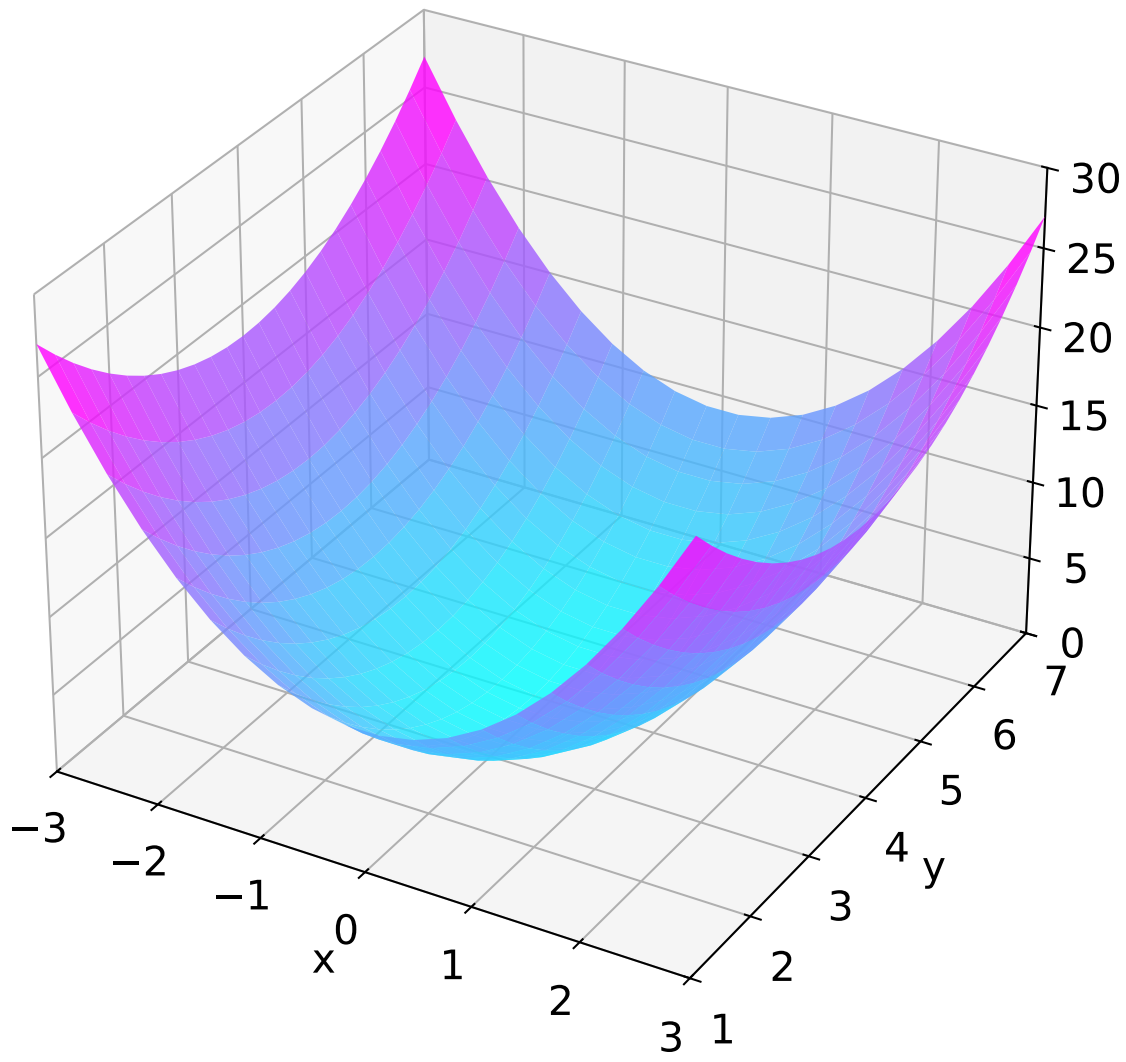
Surface (3D) Plots

The next plot creates a surface plot. Notice that if `x` is a 20 vector, `y` is a 25 vector, and `z` is 20x25, then you need to use `surf(x,y,z')`.

In the subsequent plot, we rotate the view and also customize the axes and tick marks. This requires a slightly more involved approach (see below).

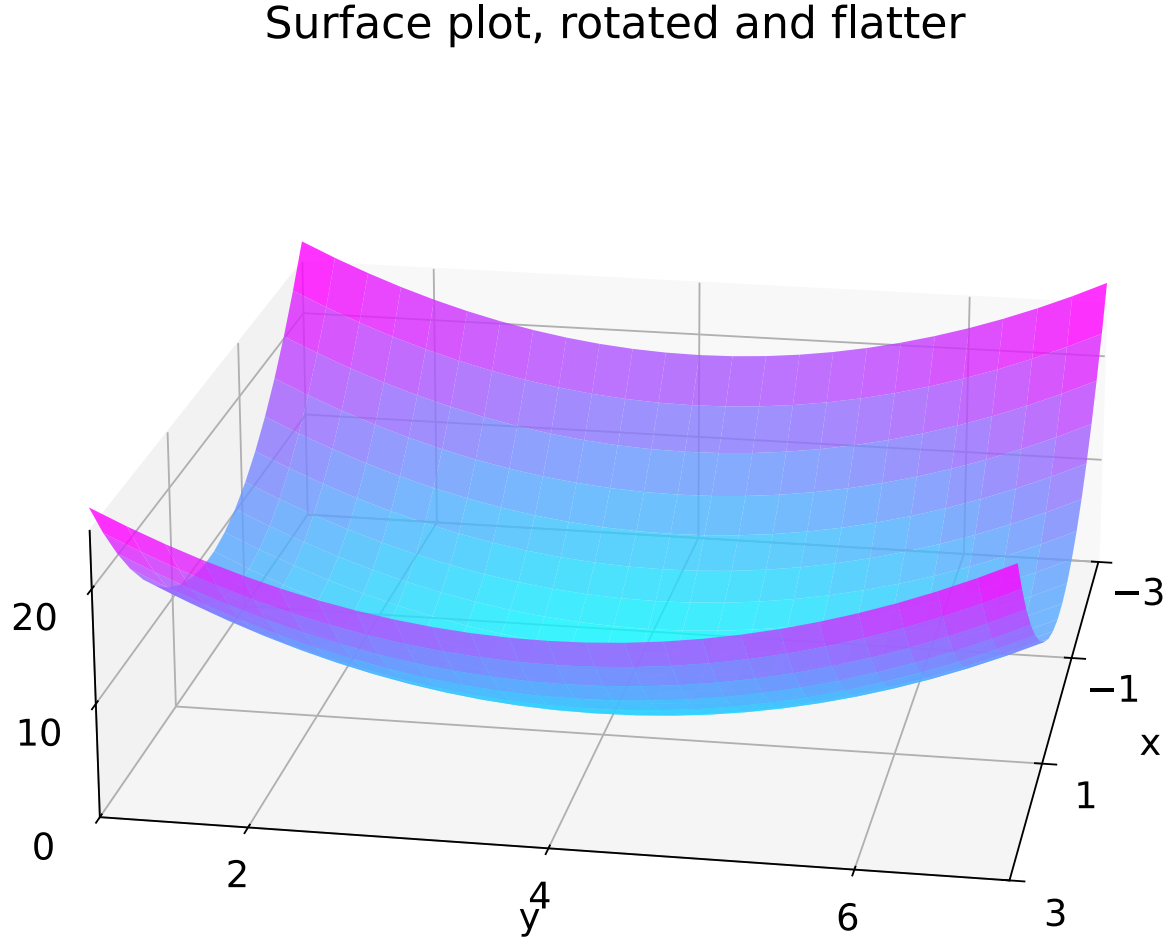
```
x = range(-3,3,length=20)       #create some "data" to plot
y = range(1,7,length=25)

z = 2*x.^2 .+ (y' .-4).^2

#notice the arguments: x,y,z'
println(size(x),size(y),size(z'))

PyPlot.matplotlib.rc("font",size=16)
fig = figure(figsize=(12,8))
ax = PyPlot.axes(projection="3d")
  surf(x,y,z',rstride=1,cstride=1,cmap=ColorMap("cool"),alpha=0.8)
  xlim(-3,3)              #change rstride and cstride to improve the look
  ylim(1,7)
  zlim(0,30)
  xlabel("x")
  ylabel("y")
  title("Surface plot")
#display(gcf())

println("\nthe aspect ratio: ",round.(gca().get_box_aspect(),digits=2))
println("\nthe elevation and azimuth: ",gca().elev," ",gca().azim)
```

```
(20,)(25,)(25, 20)

[1.19, 1.19, 0.89]

the elevation and azimuth: 30 -60
```




```
PyPlot.matplotlib.rc("font",size=16)
figure(figsize=(12,8))
ax = PyPlot.axes(projection="3d")       #ax = gca(projection="3d") in old matplotlib versions
  surf(x,y,z',rstride=1,cstride=1,cmap=ColorMap("cool"),alpha=0.8)
  xlim(-3,3)
  ylim(1,7)
  zlim(0,30)
```

```
  ax.view_init(elev=20.0,azim=10)                #notice: elevation and azimuth
  gca().set_box_aspect((6.0,6.0,2.0),zoom=1.2)  #matplotlib 3.3.0 is needed for this
  xticks(-3:2:3)
  yticks(2:2:6)
  zlim([0,25])
  zticks([0;10;20])
  xlabel("x")
  ylabel("y")
  title("Surface plot, rotated and flatter")
#display(gcf())
```

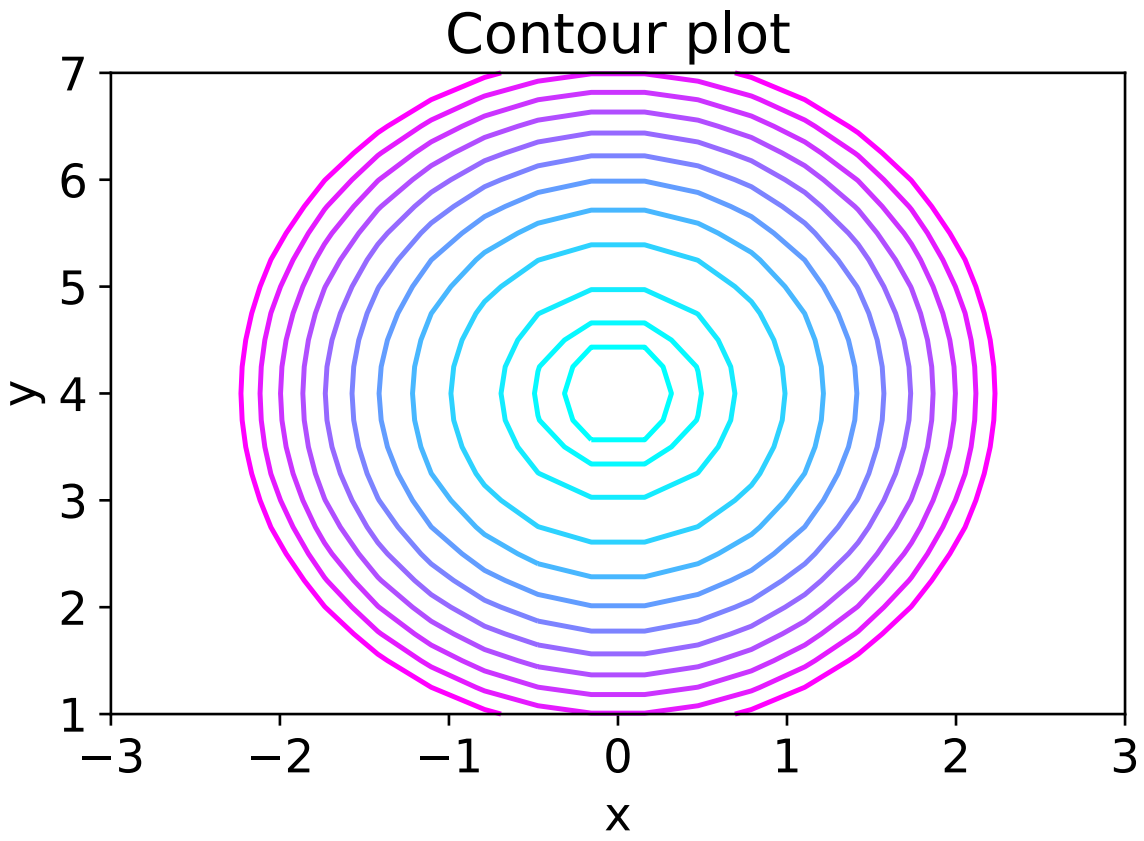


```
PyObject Text(0.5, 1.0, 'Surface plot, rotated and flatter')
```

Contour Plot

As an alternative to a surface plot, consider a contour plot. The syntax is `contour(x,y,z',...)`.

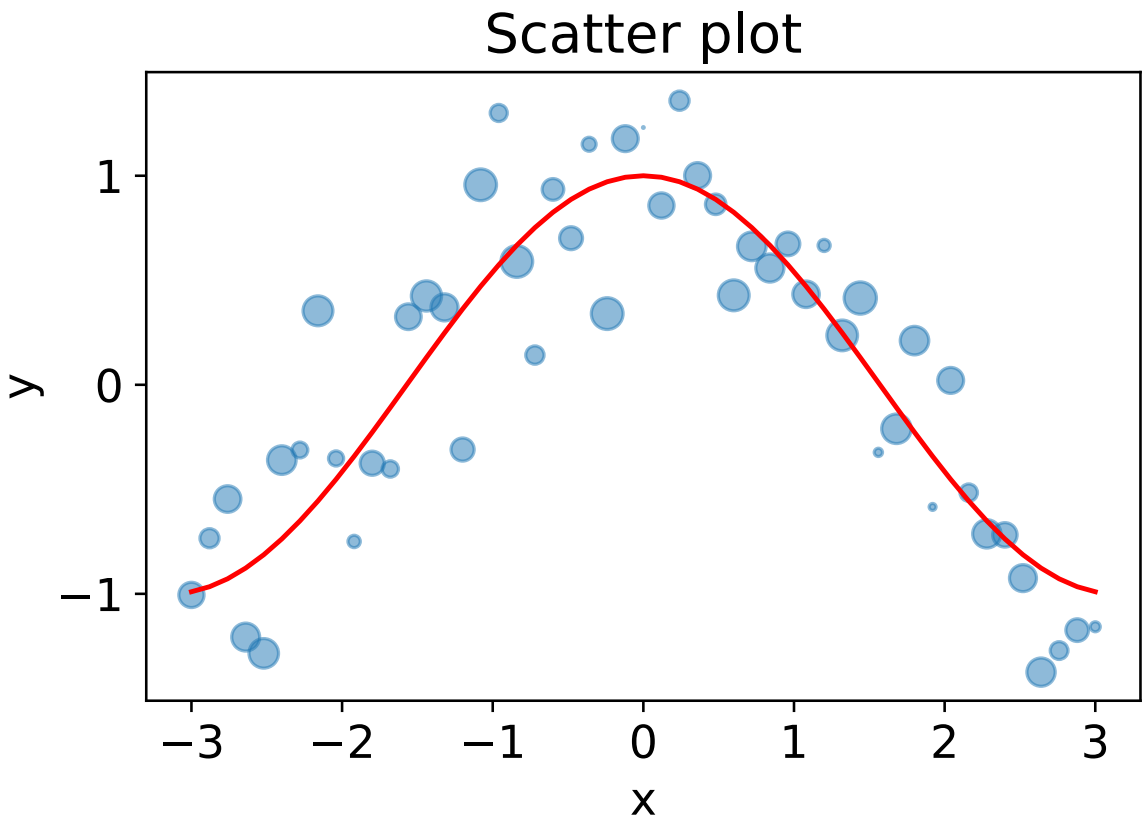
```
PyPlot.matplotlib.rc("font",size=14)
fig1 = figure(figsize=(5.5,3.5))
  lev = [0.25;0.5;1:1:10]
  contour(x,y,z',lev,cmap=ColorMap("cool"))
  xlim(-3,3)
  ylim(1,7)
  xlabel("x")
  ylabel("y")
  title("Contour plot")
#display(gcf())
```



```
PyObject Text(0.5, 1.0, 'Contour plot')
```

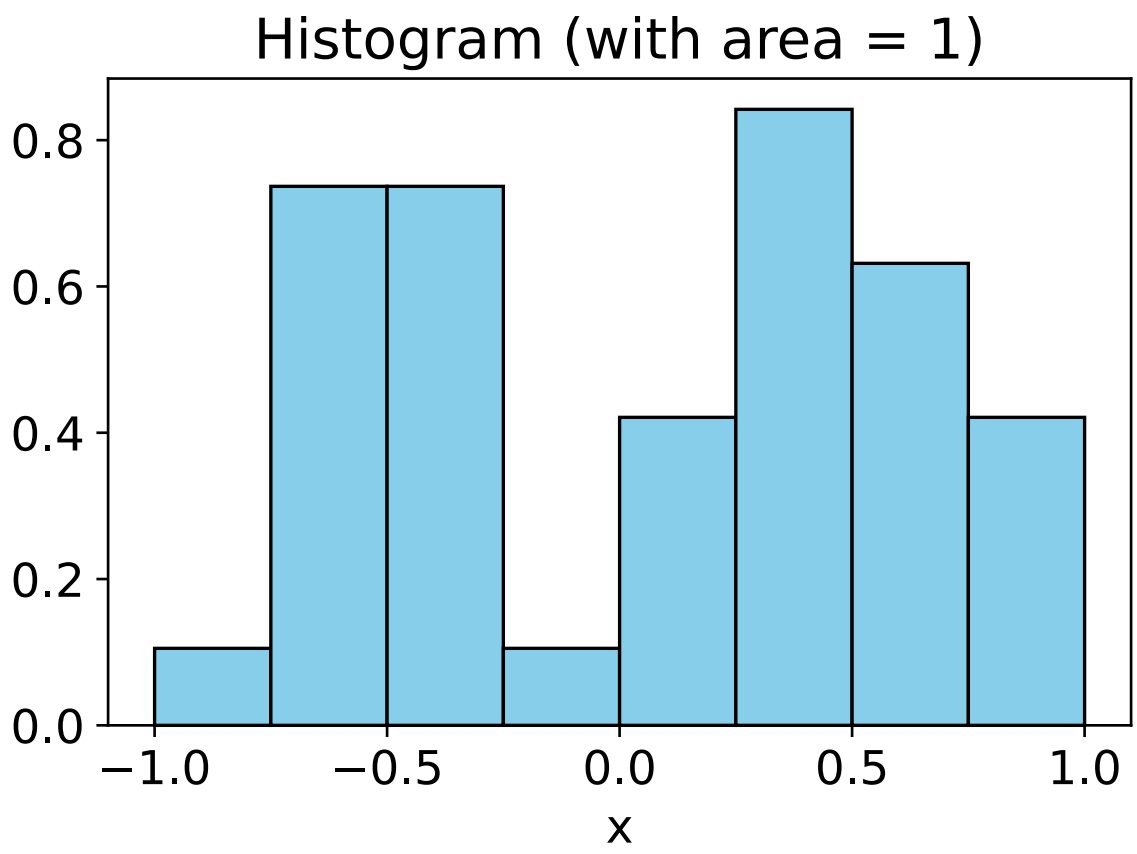
Scatter and Histogram

```
N = 51
x = range(-3,3,length=N)                          #SCATTER, HISTOGRAM
y = cos.(x) + randn(N)/3
areas = rand(51)*100                      #size of the scatter points, could be a scalar

figure(figsize=(5.5,3.5))
  scatter(x,y,s=areas,alpha=0.5)          #s is the size of the circles
  plot(x,cos.(x),"r-")
  title("Scatter plot")
  xlabel("x")
  ylabel("y")
#display(gcf())
```




```
PyObject Text(28.999999999999993, 0.5, 'y')
```

```
figure(figsize=(5.5,3.5))
  Bins = -1:0.25:1
  hist(y,bins=Bins,density=true,color="skyblue",edgecolor="k")
  title("Histogram (with area = 1)")
  xlabel("x")
#display(gcf())
```



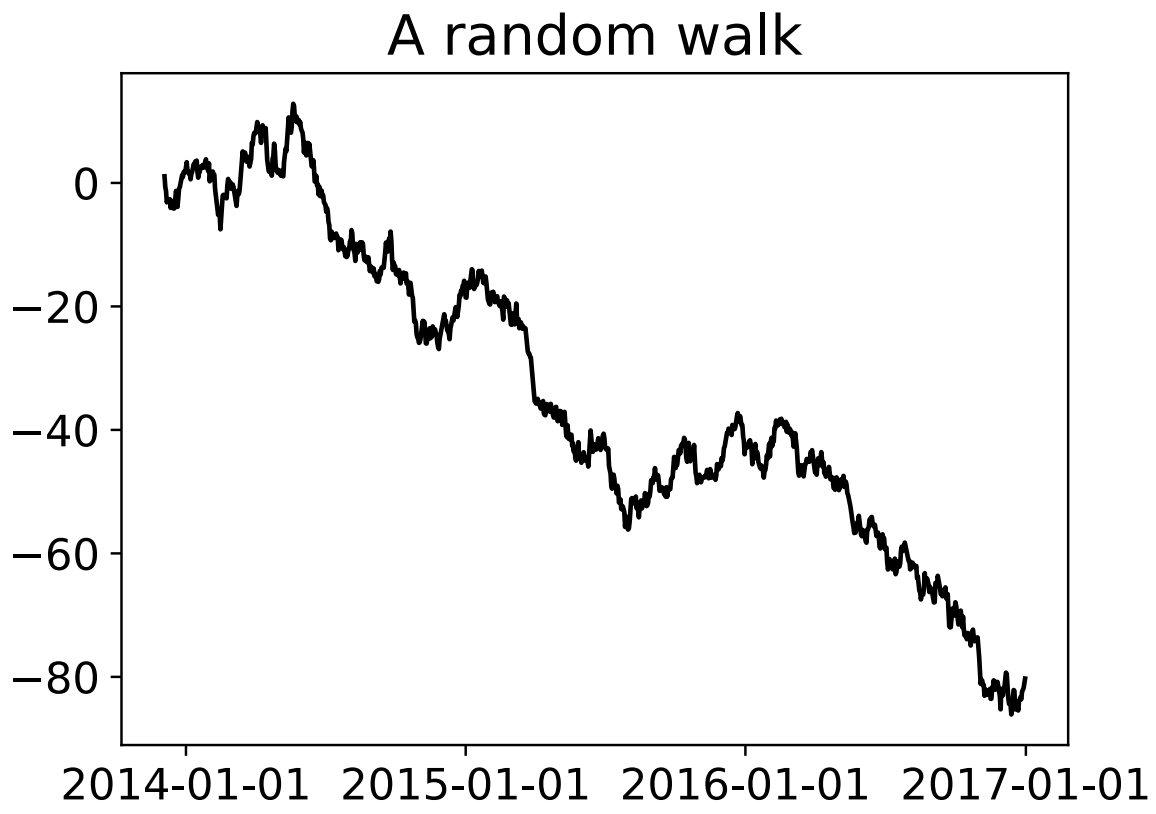

```
PyObject Text(0.5, 29.0, 'x')
```

Time Series Plots

The (time) tick marks in time series plots often need tweaking. The cell below does that by first creating a vector of dates (`xTicks`) where I want the time marks, and then calls on `xticks(xTicks)`. When the `x` variable is Julia dates, then the usual `plot()` function works well. (Otherwise, consider `plot_date()`).

```
dN = Date(2013,12,4):Day(1):Date(2016,12,31)   #just faking some dates
y  = randn(length(dN))                         #some random numbers to plot

xTicks = Date(2014,1,1):Year(1):Date(2017,1,1) #tick marks on x axis

figure(figsize=(5.5,5.5/1.4))          #basic time series plot
```

```
  plot(dN,cumsum(y),"k-")
  xticks(xTicks)
  title("A random walk",fontsize=18)
#display(gcf())
```




```
PyObject Text(0.5, 1.0, 'A random walk')
```

```
figure(figsize=(5.5,5.5/1.4))
  plot(dN,cumsum(y),"k-")
  xticks(xTicks)
  gca().xaxis.set_major_formatter(matplotlib.dates.DateFormatter("%Y"))
  title("A random walk, with better tick marks",fontsize=18)
#display(gcf())
```


```
PyObject Text(0.5, 1.0, 'A random walk, with better tick marks')
```

Adjusting the Resolution (extra)

In case you use PyPlot in a script (not a notebook), then you may experience that your figures do not work well with the resolution of your screen (the figures are too small/big and the elements look garbled). It sometimes help to change the resolution of your figures by adding the following to your script

```
rcParams = PyPlot.PyDict(PyPlot.matplotlib."rcParams")
rcParams["figure.dpi"] = 50       #experiment with different numbers
```

Notice: this is typically not needed in a notebook.

DataFrames

This notebook illustrates the *basics* of DataFrames.jl. There are many packages that work with such data frames, for instance, TidierData.jl which sticks close to R syntax. To do R style regressions, combine with StatsModels.jl and GLM.jl.

As an alternative to `DataFrames.jl`, you may want to condsider InMemoryDatasets.jl.

Load Packages

```
using Printf, Dates, Statistics, DataFrames, CSV
include("src/printmat.jl");
```

Loading Some Data with CSV.jl

The CSV.jl package is very powerful - and also quick for large files.

(You could also use `DelimitedFiles.jl`, but it is not as well integrated with the `DataFrames.jl` as `CSV.jl`.)

```
println("the raw contents of Data/dfData1.csv:")
println(read("Data/dfData1.csv",String))
```

```
the raw contents of Data/dfData1.csv:
Date,          id,         B,      C,   D,           E
01/05/2019,     1,      9.998,   0.000, The,        true
01/05/2019,     2,      9.994,   0.037, highway,    true
01/05/2019,     3,     10.044,   0.092, is,         true
02/05/2019,     1,     10.061,   0.098, for,        true
02/05/2019,     2,     10.076,   0.078, gamblers,   true
02/05/2019,     3,     10.064,   0.061, Better,     false
```

To use `CSV.jl` to create a dataframe on the fly, do as in the cell below. Use `normalizenames` to get names that can be used in Julia as variables names and specify the `dateformat` used in the csv file (to convert to proper Julia dates).

```
df1 = CSV.read("Data/dfData1.csv",DataFrame,normalizenames=true,dateformat="dd/mm/yyyy")
show(df1;allrows=true,summary=false)    #to explicitly print the df, can be configured
```

Row	Date	id	B	C	D	E
	Date	Int64	Float64	Float64	String15	Bool
1	2019-05-01	1	9.998	0.0	The	true
2	2019-05-01	2	9.994	0.037	highway	true
3	2019-05-01	3	10.044	0.092	is	true
4	2019-05-02	1	10.061	0.098	for	true
5	2019-05-02	2	10.076	0.078	gamblers	true
6	2019-05-02	3	10.064	0.061	Better	false

Picking out Rows and Columns

```
df1[:,:B]     #create a traditional Julia vector
#df1.B        #works too
```

```
6-element Vector{Float64}:
  9.998
  9.994
 10.044
 10.061
 10.076
 10.064
```

```
df1b = df1[(df1.id.>=2) .& (df1.E.==true), :]         #create new df by picking out some rows,
```

	Date	id	B	C	D	E
	Date	Int64	Float64	Float64	String15	Bool
1	2019-05-01	2	9.994	0.037	highway	1
2	2019-05-01	3	10.044	0.092	is	1
3	2019-05-02	2	10.076	0.078	gamblers	1

```
df1b = df1[2:4,[:Date,:B]]  #creates a new df by picking out some rows and cols
show(stdout,MIME("text/csv"),df1b)                              #explicit printing, here as c
show(stdout,MIME("text/html"),df1b;summary=false,eltypes=false)  #and as html
```

```
"Date","B"
"2019-05-01",9.994
"2019-05-01",10.044
"2019-05-02",10.061
```

Row	Date	B
1	2019-05-01	9.994
2	2019-05-01	10.044
3	2019-05-02	10.061

Changing a DataFrame

```
df1[2:3,:C] = [100,101]              #changing some values in :C
show(df1)
```

```
6×6 DataFrame
 Row │ Date        id     B        C        D          E
     │ Date        Int64  Float64  Float64  String15   Bool
─────┼──────────────────────────────────────────────────────
   1 │ 2019-05-01      1    9.998    0.0    The         true
   2 │ 2019-05-01      2    9.994  100.0    highway     true
   3 │ 2019-05-01      3   10.044  101.0    is          true
   4 │ 2019-05-02      1   10.061    0.098  for         true
   5 │ 2019-05-02      2   10.076    0.078  gamblers    true
   6 │ 2019-05-02      3   10.064    0.061  Better     false
```

```
rename!(df1,:D ⇒ :BabyBlue)          #renaming a column
```

	Date	id	B	C	BabyBlue	E
	Date	Int64	Float64	Float64	String15	Bool
1	2019-05-01	1	9.998	0.0	The	1
2	2019-05-01	2	9.994	100.0	highway	1
3	2019-05-01	3	10.044	101.0	is	1
4	2019-05-02	1	10.061	0.098	for	1
5	2019-05-02	2	10.076	0.078	gamblers	1
6	2019-05-02	3	10.064	0.061	Better	0

```
df1c = select(df1,[:BabyBlue,:C])              #new df
show(df1c)

println("\n")
df1b = select(df1,Not([:BabyBlue,:C]))         #new df, select and Not: all cols except those
show(df1b)
```

```
6×2 DataFrame
 Row │ BabyBlue   C
     │ String15   Float64
─────┼─────────────────────
   1 │ The          0.0
   2 │ highway    100.0
   3 │ is         101.0
   4 │ for          0.098
   5 │ gamblers     0.078
   6 │ Better       0.061

6×4 DataFrame
 Row │ Date        id     B        E
     │ Date        Int64  Float64  Bool
─────┼──────────────────────────────────
   1 │ 2019-05-01      1    9.998   true
   2 │ 2019-05-01      2    9.994   true
   3 │ 2019-05-01      3   10.044   true
   4 │ 2019-05-02      1   10.061   true
   5 │ 2019-05-02      2   10.076   true
   6 │ 2019-05-02      3   10.064  false
```

Converting to and from a (traditional) Julia Matrix

```
arraydata = Matrix(df1[:,[:id,:B,:E]])
```

```
6×3 Matrix{Float64}:
 1.0   9.998  1.0
 2.0   9.994  1.0
 3.0  10.044  1.0
 1.0  10.061  1.0
 2.0  10.076  1.0
 3.0  10.064  0.0
```

```
df_from_matrix = DataFrame(arraydata,:auto)
```

	x1	x2	x3
	Float64	Float64	Float64
1	1.0	9.998	1.0
2	2.0	9.994	1.0
3	3.0	10.044	1.0
4	1.0	10.061	1.0
5	2.0	10.076	1.0
6	3.0	10.064	0.0

Reshuffling a DataFrame (Groupby)

We now create a group for each id. These groups can be referred to as `dataG1[key]`.

We list the keys below. For instance `(id=2,)` is one of the keys. This is a tuple with a single element (similar to `(x,)`) where the element is `id=2`.

```
dataG1 = groupby(df1,:id)    #grouped by id
```

GroupedDataFrame with 3 groups based on key: id

First Group (2 rows): id = 1

	Date	id	B	C	BabyBlue	E
	Date	Int64	Float64	Float64	String15	Bool
1	2019-05-01	1	9.998	0.0	The	1
2	2019-05-02	1	10.061	0.098	for	1

…

Last Group (2 rows): id = 3

	Date	id	B	C	BabyBlue	E
	Date	Int64	Float64	Float64	String15	Bool
1	2019-05-01	3	10.044	101.0	is	1
2	2019-05-02	3	10.064	0.061	Better	0

```
dataG2 = vcat(dataG1...)          #put together again, but now id=1 first, then id=2
```

	Date	id	B	C	BabyBlue	E
	Date	Int64	Float64	Float64	String15	Bool
1	2019-05-01	1	9.998	0.0	The	1
2	2019-05-02	1	10.061	0.098	for	1
3	2019-05-01	2	9.994	100.0	highway	1
4	2019-05-02	2	10.076	0.078	gamblers	1
5	2019-05-01	3	10.044	101.0	is	1
6	2019-05-02	3	10.064	0.061	Better	0

```
println("The keys of the grouped DataFrame are: ")
printmat(DataFrames.GroupKeys(dataG1))
```

```
The keys of the grouped DataFrame are:
GroupKey: (id = 1,)
GroupKey: (id = 2,)
GroupKey: (id = 3,)
```

```
dataG1[(id=2,)]        #the group for (id=2,),
```

	Date	id	B	C	BabyBlue	E
	Date	Int64	Float64	Float64	String15	Bool
1	2019-05-01	2	9.994	100.0	highway	1
2	2019-05-02	2	10.076	0.078	gamblers	1

Merging DataFrames

It is possible to stack DataFrames, if they are comformable.

More generally, we can use one of the many `join` functions, for instance, `innerjoin()`.

```
df3 = [df1 DataFrame(ff=1:6)]        #stacking 2 dataframes horizontally
```

	Date	id	B	C	BabyBlue	E	ff
	Date	Int64	Float64	Float64	String15	Bool	Int64
1	2019-05-01	1	9.998	0.0	The	1	1
2	2019-05-01	2	9.994	100.0	highway	1	2
3	2019-05-01	3	10.044	101.0	is	1	3
4	2019-05-02	1	10.061	0.098	for	1	4
5	2019-05-02	2	10.076	0.078	gamblers	1	5
6	2019-05-02	3	10.064	0.061	Better	0	6

```
                        #loading another DataFrame, it has another order of observations
df2 = CSV.read("Data/dfData2.csv",DataFrame,normalizenames=true,dateformat="dd/mm/yyyy")
```

	Date	id	G
	Date	Int64	Int64
1	2019-05-01	1	11
2	2019-05-02	1	12
3	2019-05-01	2	21
4	2019-05-02	2	22
5	2019-05-01	3	31
6	2019-05-02	3	32

```
df3 = innerjoin(df1,df2,on=[:Date,:id])        #joining 2 DataFrames, match both :Date and :id
                                               #innerjoin() for intersection of data points
```

	Date	id	B	C	BabyBlue	E	G
	Date	Int64	Float64	Float64	String15	Bool	Int64
1	2019-05-01	1	9.998	0.0	The	1	11
2	2019-05-02	1	10.061	0.098	for	1	12
3	2019-05-01	2	9.994	100.0	highway	1	21
4	2019-05-02	2	10.076	0.078	gamblers	1	22
5	2019-05-01	3	10.044	101.0	is	1	31
6	2019-05-02	3	10.064	0.061	Better	0	32

```
df3[:,:c] .= 1                 #add a constant to the DataFrame
show(df3;summary=false)
```

Row	Date	id	B	C	BabyBlue	E	G	c
	Date	Int64	Float64	Float64	String15	Bool	Int64	Int64
1	2019-05-01	1	9.998	0.0	The	true	11	1
2	2019-05-02	1	10.061	0.098	for	true	12	1

3	2019-05-01	2	9.994	100.0	highway	true	21	1
4	2019-05-02	2	10.076	0.078	gamblers	true	22	1
5	2019-05-01	3	10.044	101.0	is	true	31	1
6	2019-05-02	3	10.064	0.061	Better	false	32	1

Using the Data

```
median(df1[:,:B])
```

```
10.0525
```

```
combine(groupby(df1,:id),:B.⇒[mean,std])  #mean and std of B for each id
```

	id	B_mean	B_std
	Int64	Float64	Float64
1	1	10.0295	0.0445477
2	2	10.035	0.0579828
3	3	10.054	0.0141421

Tricky Stuff

This file highlights some tricky aspects of Julia (from the perspective of a Matlab user).

Load Packages and Extra Functions

```
using Printf, LinearAlgebra

include("src/printmat.jl");
```

An Nx1 Matrix Is Not a Vector

and it sometimes matters.

Julia has both vectors and Nx1 matrices (the latter being a special case of NxM matrices). They can often be used interchangeably, but not always.

```
v  = ones(Int,2)                    #a vector with two elements
v2 = ones(Int,2,1)                  #a 2x1 matrix (Array)

println("v and v2 'look' similar:")
printmat(v)
printmat(v2)
println("but they have different sizes: ",size(v)," ",size(v2))
```

```
v and v2 'look' similar:
     1
     1

     1
```

```
     1

but they have different sizes: (2,) (2, 1)
```

X[1,:] Gives a (Flat) Vector

If X is a $T \times n$ matrix, then X[1,:] gives a flat vector, *not* a $1 \times n$ matrix (or row vector).

```
X = [11 12;21 22]

x1 = X[1,:]                      #this gives a flat vector
println("size of X[1,:]: ", size(x1))
printmat(x1)

x1b = X[1:1,:]                   #however, this gives a 1x2 matrix
println("size of X[1:1,:]: ", size(x1b))
printmat(x1b)
```

```
size of X[1,:]: (2,)
    11
    12

size of X[1:1,:]: (1, 2)
    11        12
```

Array .+ Scalar Requires a Dot (.)

```
y = [1;2] .+ 1              #do not forget the dot (.)
printmat(y)
```

```
     2
     3
```

Creating Variables in a Loop

```
for i = 1:5
    global Tor          #without this, Tor is not seen outside the loop
    Tor = cos(i)
end
println("Tor: $Tor")


Oden = Inf               #define `Oden` before the loop
for i = 1:5
    #global Oden         #only needed in script
    Oden = sin(i)        #will overwrite an existing value
end
println("Oden: $Oden")
```

```
Tor: 0.28366218546322625
Oden: -0.9589242746631385
```

An Array of Arrays

can be initialized by comprehension (see below).

Do **not** use `fill(array,n)` for this, since all n arrays will refer to the same underlying array (changing one changes all).

```
x = [zeros(2,2) for i=1:2]         #a vector of two matrices
x[1][1,1] = -99                    #change an element of x[1]

println("x[1]")
printmat(x[1])

println("x[2]")
printmat(x[2])
```

```
x[1]
   -99.000     0.000
     0.000     0.000

x[2]
     0.000     0.000
     0.000     0.000
```

Arrays are Different...

Vectors and matrices (arrays) can take lots of memory space, so **Julia is designed to avoid unnecessary copies of arrays**. Notice the following: * `B = A` creates two names of the *same* array (changing one changes the other) * `B = reshape(A,n,m)`, `B = vec(A)`, and `B = A'` and create *another view* of the same array (changing one changes the other) * `E = [C,D]` (where `C` and `D` are arrays) refers to underlying arrays. Changing `C` or `D` will affect `E` and vice versa. * When an input an array to a function, then this array is shared between the function and the calling program (scope). Changing *elements* of the array (inside the function) will then change the array outside the function.

If you do not like this behaviour, then use `copy(A)` to create an independent copy of the array.

See the chapter on Arrays for more details.