2018/2019

Financial Risk Management

an application to MATLAB

University Paris-Dauphine
Master Degree & Pre-Doctoral Program

Sylvain Benoit[1] *sylvain.benoit@dauphine.fr*
Gauthier Vermandel [2] *gauthier.vermandel@dauphine.fr*

Chapter 1:
An Introduction to MATLAB Programming

Content of the Lecture

Objectives of the lecture

- Utilize the basic mathematical operations;
- Get know the general purpose commands of Matlab;
- Manipulate matrices, functions and loops;

[1]Department of Economics, Paris-Dauphine University.
[2]Department of Economics, Paris-Dauphine University. Codes available on my website: : http://vermandel.fr/rfm.

1. MATLAB in a nutshell

MATLAB (matrix laboratory) is a multi-paradigm numerical computing environment and fourth-generation programming language. A proprietary programming language developed by MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, C#, Java, Fortran and Python.

MATLAB (the name stands for: *Matrix Laboratory*) is a high performance programming language and a computing environment that uses vectors and matrices as one of its basic data types (MATLAB®is a registered trademark of the MathWorks, Inc.). It is a powerful tool for mathematical and technical calculations and it can also be used for creating various types of plots.[3]

1.1 Basic calculus

It is the main window in which the user communicates with the software. In the command window, the user can view the prompt symbol ">>" which indicates that MATLAB is ready to accept various commands by the user. Via this window, the user can employ the basic arithmetic operators like: "+" (addition), "-" (subtraction), "*" (multiplication), "/" (division), "^" (powers) and the "( )" (brackets), as well as many other build in elementary and other functions and commands that will be referred to later.

As a first example, enter the following command that performs a basic calculation:

```
>> 5+6/12+9/3-2
ans =
6.5000
```

The MATLAB displays the results into a variable named as anss. This is a default variable name that is used by the command window to display the most recent results instructed by the user that has not defined a specific name. Continue by entering the following command that creates a fourelement vector:

```
>> [1 2 3 4]
ans =
1     2     3     4
```

The square brackets "[ ]" indicate the definition of the vector. Also, the space between the vector numbers separates the vector's elements. The comma "," is another way of separating the elements. Additionally, note that the default variable anss has lost its previous value in order to store the results of the most recent operation.

As another example, enter the following command that creates a 3-by-3 magic square saved in the matrix variable M:

```
>> M=magic(3)
M =
```

[3]This section is buildt on the work of Panayiotis Andreou entitled "Introductory Course to Matlab with Financial Case Studies".

```
8     1     6
3     5     7
4     9     2
```

The magic square is created using the element function magiicc that is already built in Matlab.

Type also the following:

```
>> x=[(2^2+1)^2-15/4*6.1, 1.23e-2]
x =
2.1250    0.0123
```

Observe that the very first command that you have entered in the command window has vanished with the additional of the last one (of course it depends on the size of the window, in here the command window is minimized so earlier commands vanish too quickly). You can use the scroll bar on the right to navigate the command window and go up to see the earlier commands (or view them via the command history window). MATLAB calculates the quantities as follows:

- First come the quantities in brackets;
- Power calculation follow: (*e.g.* 5 + 2^2 = 5 +4 = 9);
- "*" and "/" follow by working from left to right: (*e.g.* 2*8/4 = 16/4 = 4);
- "+" and "-" follow last, from left to right: (*e.g.* 6-7+2= -1+2 = 1).

Note that "e" notation is used for very large or very small numbers. By definition: 1e1=1X101 and 1e-1=1X10-1.

Trigonometric		*Exponential*	
sin	sine	exp	exponential
sinh	hyperbolic sine	log	natural logarithm
asin	inverse sine	log10	common (base10) logarithm
asinh	inverse hyperbolic sine	log2	base 2 power and scale floating point
cos	cosine	pow2	base 2 power and scale floating point
cosh	hyperbolic cosine	realpow	power that will error out on complex
acos	inverse cosine	reallog	natural logarithm of real number
acosh	inverse hyperbolic cosine	realsqrt	square root of number greater than $x^{0.5}$
tan	tangent	sqrt	square root
tanh	hyperbolic tangent	nextpow2	next higher power of 2
atan	inverse tangent	*Complex*	
atan2	four quadrant inverse	abs	absolute value
atanh	inverse hyperbolic tangent	angle	phase angle
sec	secant	complex	construct complex data from real & imaginary
sech	hyperbolic secant	conj	complex conjugate
asec	inverse secant	imag	complex imaginary part
asech	inverse hyperbolic secant	real	complex real part
csc	cosecant	unwrap	unwrap phase angle
csch	hyperbolic cosecant	isreal	true for real array
acsc	inverse cosecant	cplxpair	sort numbers into complex conjugate pairs
acsch	inverse hyperbolic cosecant	*Rounding and Remainder*	
cot	cotangent	fix	round towards zero
coth	hyperbolic cotangent	floor	round towards minus infinity
acot	inverse cotangent	ceil	round towards plus infinity
acoth	inverse hyperbolic cotangent	round	round toward nearest integer
		mod	modulus
		rem	remained after division
		sign	signum

Table 1: MATLAB's elementary build-in functions

Matlab can handle three different kinds of numbers: integers, real numbers and complex numbers (with imaginary parts). Moreover, it can handle non-number expressions like: NaN (Not-a-Number) produced from mathematically undefined operations like: $0/0$, $\infty^* \infty$ and inf produced by operations like $1/0$. Matlab as a calculator includes a variety of build-in mathematical functions like: trigonometric, exponential, complex, rounding and remainder, etc. Table 1 below, depicts these elementary mathematical build-in functions.

Note the build-in functions exhibited in Table 1 have their own calling syntax. For example, if you type sin in the command window, MATLAB returns the following error massage:

```
>> sin
Error using sin
Not enough input arguments.
```

This happened because the sin function requires an input expression like a number enclosed in brackets. If you enter:

```
>> sin(5)
ans =
-0.9589
```

In the rest of this handout, only the name of the build in functions will be given.

Beside some exceptions where the correct calling syntax of a function is fully tabulated, the user is responsible in knowing the necessary input arguments to the function. Additional the use of the command window will be apparent as you read this manuscript since the learning of a computer programming language is pure a "learning by doing" process.

1.2 Matricial calculus

A matrix or an array is the basic element on which Matlab can operate. A 1-by-1 matrix forms a scalar or a single number, whereas a matrix with only one row or column forms a row or column vector respectively. This section exhibits the mathematical manipulation of vectors (arrays) and of two dimensional matrices. Most of build-in function for matricial calculation are reported in Table 4.

1.2.1 Row of vectors

A vector is a list of numbers separated by either space or commas. Each different number/entry located in the vector is termed as either element or component. The number of the vector elements/components determines the length of the vector. In Matlab, square brackets "[ ]" are used to both define a vector and a matrix. For instance, the following command returns a row vector with 5 elements:

```
>> y=[ 5 exp(2) sign(-5) sqrt(9) pi]
y =
5.0000    7.3891   -1.0000    3.0000    3.1416
```

Note that the definition of the row vector, the user it free to use any built-in function as long as this is used properly. In the above definition, exp(.) is the exponential, sign(.) returns the sign, sqrt(.) is the square root and pi represents π. General speaking and except some special cases, when a function is applied to a 1-by-1 scalar, the result is a scalar, when applied to a row or column vector is a row or column vector and when applied to a matrix the output is again a matrix. This happe ns because MATLAB applied the build-in functions element-wise. The lleengtth of the above vector is obtained via the following command:

```
>> length(y)
ans =
5
```

Note that length(.) is a build-in function that given a vector, it returns its length. Since the result of this function is not stored in a user-defined variable, MATLAB saves the result in the ans variable. If it is need to save the vector's length to a variable named Yleength the command would be:

```
>> Ylength=length(y)
Ylength =
5
```

With MATLAB, vectors can be easily multiplied by a scalar and added or subtracted with other vectors of similar length. Moreover, a scalar can be added or subtracted to

or from a vector and smaller vectors can be used to construct larger ones. All these operations are performed element-wise. Note that each vector represents a variable.

An element-by-element multiplication of vector a by 5:

```
>> a=[-1 2 -3]; b=[2 1 2];
c=5*a
c =
-5    10   -15
```

An element-by-element addition of vector b with 5:

```
>> c1=5+b
c1 =
7     6     7
```

An element-by-element addition of two equal length vectors:

```
>> d=a+b
d =
1     3    -1
```

A larger vector is created after certain manipulations:

```
>> d=a+b
d =
1     3    -1
```

To refer to specific elements of the vector, the vector's name is followed by the element's rank in brackets. For instance we can change the values of the ee vector with the following command:

```
>> e(2)=-99; e(4)=-99; e(6)=-99;
e
e =
0   -99     0   -99     0   -99
```

1.2.2 The colon notation

The colon notation ":" can be used to pick out selected rows, columns and elements of vectors, matrices, and arrays. It is a shortcut that is used to create row vectors. Moreover, the colon notation is used to view or extract parts of vectors (afterwards, with matrices, the colon can be used to view a certain part of a matrix). For instance, the following commands produce three different row vectors:

```
>> v1=1:8, v2=-4:2:2, v3=[0.1:0.2:0.6]
v1 =
1     2     3     4     5     6     7     8
v2 =
-4    -2     0     2
v3 =
0.1000    0.3000    0.5000
```

zeros	Create array of all zeros	isscalar	Determine whether input is scalar
ones	Create array of all ones	isvector	Determine whether input is vector
rand	Uniformly distributed random numbers	ismatrix	Determine whether input is matrix
true	Logical 1 (true)	isrow	Determine whether input is row vector
false	Logical 0 (false)	iscolumn	Determine whether input is column vector
eye	Identity matrix	isempty	Determine whether array is empty
diag	Create diagonal matrix	sort	Sort array elements
blkdiag	Construct block diagonal matrix	sortrows	Sort rows of array, table, or timetable
cat	Concatenate arrays	issorted	Determine whether array is sorted
horzcat	Concatenate arrays horizontally	flip	Flip order of elements
vertcat	Concatenate arrays vertically	fliplr	Flip array left to right
repelem	Repeat copies of array elements	flipud	Flip array up to down
repmat	Repeat copies of array	rot90	Rotate array 90 degrees
linspace	Generate linearly spaced vector	transpose	Transpose vector or matrix
logspace	Generate logarithmically spaced vector	ctranspose	Complex conjugate transpose
freqspace	Frequency spacing for frequency response	permute	Rearrange dimensions of N-D array
meshgrid	2-D and 3-D grids	ipermute	Inverse permute dimensions of N-D array
ndgrid	Rectangular grid in N-D space	circshift	Shift array circularly
length	Length of largest array dimension	shiftdim	Shift dimensions
size	Array size	reshape	Reshape array
ndims	Number of array dimensions	squeeze	Remove singleton dimensions
numel	Number of array elements	colon	Create vectors, array subscripting, and for-loop

Table 2: MATLAB's Matrices and Arrays build-in functions

The careful viewer should have notice that the vector creation via the use of the colon notation has the form: starting_value:step:finishing_value. The starting_value consists the first value/element of the vector, the finishing_value is the last value/element of the vector and all other elements differ by a value equal to step. Also, when the interval between finishing_value and starting_value is not divisible by the step, the last value of the vector is not the finishing_value (*i.e.* in v3 the last element is 0.5 and not 0.6). The colon notation is also used to view or extract parts of a vector.

Extracting the 2th, 3rd and 4th elements of v1:

```
>> v4=v1(2:4)
v4 =
2     3     4
```

Extracting the 2th, 5th and 8th elements of v1:

```
>> v1(2:3:8)
ans =
2     5     8
```

1.2.3 The column vectors

Column vectors are created via the use of semi-colon ";" instead of commas and spaces. Operations with column vectors are similar as with row vectors. The following examples depict the manipulation of column vectors.

Creating a four-element column vector:

```
>> cv=[1;4;7;9]
cv =
1
4
7
9
```

Taking the vector's length:

```
>> length(cv)
ans =
4
```

Creating CV that is a four-element column vector. Its first two elements are the two first elements of ccvv increased by 2 whereas the last two elements are the 2nd and 3rd elements of cv multiplied by 5:

```
>> CV=[cv(1:2)+2; cv(2:3)*5]
CV =
3
6
20
35
```

1.2.4 Matrices scalar product

Matlab can perform, scalar products (or inner products) and dot products, as well as dot division and power operations. The only restriction is that the length of the vectors must be the same. The priorities concerning vector manipulations is the same as in the case that we use MATLAB as a calculator (power operations first followed by "*" and "/" followed by "+" and "-").

The scalar product or otherwise termed as inner product, concerns the multiplication of two equal length vectors. The symbol "*" is used to carry out this operation. Given a row vector W and a column vector U of length N:

$$W = \left[\begin{array}{cccc} w_1, & w_2, & ..., & w_N \end{array} \right], \qquad U = \left[\begin{array}{c} u_1 \\ u_2 \\ ... \\ u_N \end{array} \right]$$

The inner product is defined as the linear combination:

$$W \times U = \sum_n^N w_n u_n$$

Producing the inner product of various operations. For example prod1 is computed as: prod1=(1*2)+(0*4)+(2*(-2))+((-1)*0.5)=-2.5.

```
>> w=[1 0 2 -1]; u=[2; 4; -2; 0.5];
prod1=w*u, prod2=(2+w)*(u/2), prod3=w*w', prod4=w*u*w*u
prod1 =
```

```
-2.5000
prod2 =
3.2500
prod3 =
6
prod4 =
6.2500
```

1.2.5 Matrices dot product

The dot product involves the multiplication of two similar vectors (to be either column or row vectors with same length) element-by-element. With the dot product, a new vector is created. The dot product between the row vectors W and U^T (where the superscript T represents the transpose symbol) is another row vector with the following form:

$$W \times U^T = \begin{bmatrix} w_1u_1, & w_2u_2, & ..., & w_Nu_N \end{bmatrix}$$

The dot product in Matlab is performed via the ".*" symbol. By the use of dot product we can get the inner product. This is explained in the following examples.

Producing the dot product of various operations. For example dprod1 is computed as: dprod1=[1*2, 0*4, 2*(-2), (-1)*0.5].

```
>> dprod1=w.*u', dprod2=u'.*u'.* w
dprod1 =
2.0000         0   -4.0000   -0.5000
dprod2 =
4.0000         0    8.0000   -0.2500
```

1.2.6 Matrices dot power

The dot power of vectors works in a similar way as with other dot products. Usually there is the need to square (or to rise to some other power) the elements of a vector. This can be done with the dot power operation as explained in the following example set.

```
>> x=-5:5;
d_power=sqrt(x.^2)
d_power =
5     4     3     2     1     0     1     2     3     4     5
```

Exercise 1

1. Create the following matrices: $U = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$:
   (a) by hand.
   (b) using function linspace.
   (c) Using

Exercise 2

Download the exercise1.m code which loads macro data for the US economy:

```
%% EXERCICE 1
% loading macro data
load Data_NelsonPlosser
% select data from 1910-1970
ourData = Data(51:end,:);
% time period vector 1910-1970
Time        = dates(51:end);
%% Extract data
% Stock price
SP = ourData(:,14);
% Bond Yield
BY = ourData(:,13);
% real per capital GNP
GNPPC = ourData(:,3);
```

1. Compute the stock price return using the variable SP with $r_t = \log(SP_t/SP_{t-1})$. Same for the real GDP growth rate g_t. *Tips:* look for function diff(.).

2. Assuming that we want to know the how much asset price return r_t are affected by the GDP growth g_t. To do so, the simple regression equation is given by: $Y_i = b_0 + b_1 X_{i1} + e_i$ with $e_i \backsim \mathcal{N}(0, \sigma_e^2)$. The equation above can be expressed more compactly by a set of matrices: $Y = Xb + e$.

   (a) Create a matrix of all ones with functions ones() with the same size as g_t named X0.

   (b) Create the matrix $X = [\mathrm{X0}, g_t]$ composed of two vectors: one of all ones, and another with g_t.

   (c) Regression estimates of $b = [b_0, b_1]$ are typically found via least squares. The minimization problem of least squares boils down to:

   $$b = \left(X^T X\right)^{-1} X^T Y$$

   Compute b and interpret the result.

   (d) Confirm your result using an internal function of MATLAB performing linear regressions (use 2 methods).

3. Now include the treasury bond yield (variable BY in MATLAB) in the regression model and estimate b. Are your results consistent with empirical evidence? *Tips:* the first value of BY must be removed to have the same number of observations as for g_t.

1.2.7 Matrix operators

MATLAB comparison operators are:

==	equal	>	strictly higher than
~=	different	<=	less or equal to
<	strictly less than	>=	greater than or equal to

Table 3: MATLAB's Matrices operators

The comparison of two scalars returns 1 if the relationship is true and 0 if false. It is also possible to make comparison between matrices of same size or between a matrix and a scalar. The following examples illustrates the flow controls overs matrices:

```
>> A = [1 2;3 4]; B = 2*ones(2);
>> A == B
ans =
0     1
0     0
>> A > 2
ans =
0     0
1     1
```

To assess how two matrices are identical, it is possible to use the IsEqual:

```
>> isequal(A,B)
ans =
0
```

The IsEqual function is one of the many boolean functions offered by MATLAB, characterized by their names starting with is statement. The table below summarizes some of them:

ischar(x)	True if x is a character	isnan(x)	True if the elements of x are not a number
isempty(x)	True if x is empty	islogical(x)	True if element is logical
isequal(x,y)	True if the elements of x and y are identical	isnumeric(x)	True if the elements of x are numeric
isfinite(x)	True if the elements of x are finite	isreal(x)	True if the elements of x are real
isinf(x)	True if the elements of x are infinite		

Table 4: MATLAB's functions checking some propreties of arrays/matrices

The find() command can be used to return the indices corresponding to a non-zero element of the vector. for example:

```
>> x = [-3 1 0 -inf 0];
f = find(x)
f =
1     2     4
```

Find the results can then be used to extract the elements of the vector:

```
>> x(f)
ans =
-3     1  -Inf
```

You can also use the find command to get the finite elements of the vector x above:

```
>> x(find(isfinite(x)))
ans =
-3     1     0     0
```

If one wishes to replace the negative elements of x with a zero, we do this:

```
>> x(find(x < 0)) = 0
x =
0     1     0     0     0
```

Exercise 3

1. Generate 10000 random numbers drawn from a normal distribution (0,1). Draw the histogram of the normal distribution.
2. Suppose that we want to truncate the distribution its last decile (i.e. removing 10% highest values). Determine the critical threshold. Replace the last decile of the distribution by NaN and plot the truncated distribution.

1.3 Plotting

It is quite easy to create a plot or a graph by using the variables or parameters that have already been stored in the MATLAB's workspace. MATLAB offers a variety of build-in functions for creating simple two dimensional plots, 3 dimensional surface plots, to combine plots to a larger one and so on. The following subsections are a very brief and an elementary reference to the visualization capabilities of MATLAB.

1.3.1 Creating 2D line plots

The basic function for the creation of a simple 2D line plot is called: plot(·). Let be a row or column vector with real data named Y, with elements "y_1, y_2, ...,y_n". If plot(·) is called as: "plot(Y)", then a linear plot of the elements of Y versus its index will appear (the plot(·) function creates/plots the pairs: (1, y_1), (2, y_2),..., (n, y_n) and connects them with a line). For example, to plot X^2:

```
X=-5:5;
Y=X.^2;
plot(Y)
```

If for each "y_i" we have an accompanied "x_j" coordinate, then the function "plot(X,Y)" plots all pairs: (x_1,y_1), (x_2, y_2), ..., (x_n, y_n) and connects them with a line. Note that vectors Y and X must share the same length. Additionally, if the data to be plotted are stored in a two dimensional array, then the arguments for the plot function can be

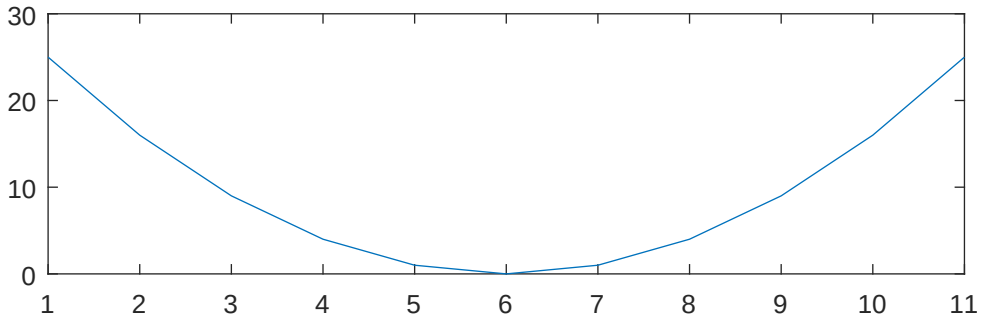

Figure 1: Simple plot(Y)

done via the use of colon notation ":". It is very easy to plot the following function in the range [-5,5]:

```
X=-5:5;
Y=X.^2;
plot(X,Y)
```

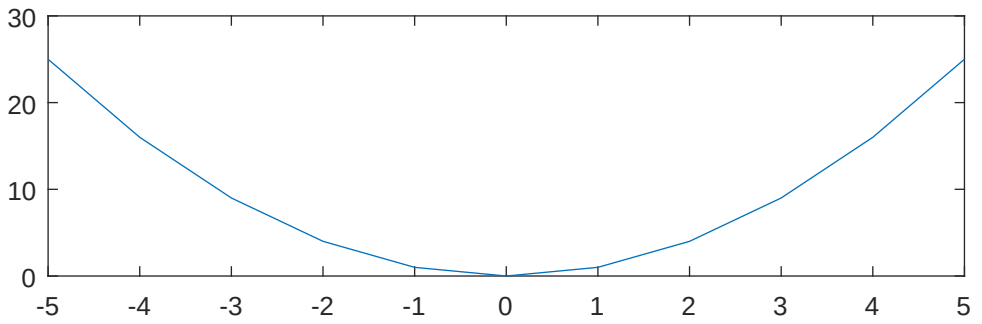

Figure 2: Simple plot(X,Y)

In addition, it is possible to include a title and labels for each axis using functions title('text'), xlabel('text') and ylabel('text'):

```
X=-5:5;
Y=X.^2;
plot(X,Y)
title('plotting function x^2')
xlabel('x')
ylabel('y')
```

1.3.2 Creating 3D plots

There is the function: plot3(·) which its use is similar as with the function: plot(·), with the difference that an additional input argument corresponding to the additional dimension is required (the additional dimension is symbolized as "z"). plot3(·) is used to plot lines or point coordinates in the 3D space. Let use the plot3(·) to report $(XZ)^2$:

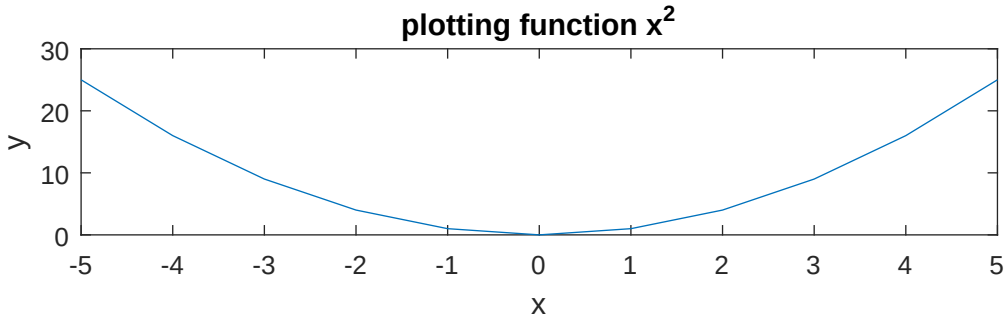

Figure 3: Simple plot(X,Y) with titles

```
X=-5:5;
Y=-5:5;
Z=(X.*Y).^2;
plot3(X,Y,Z)
```

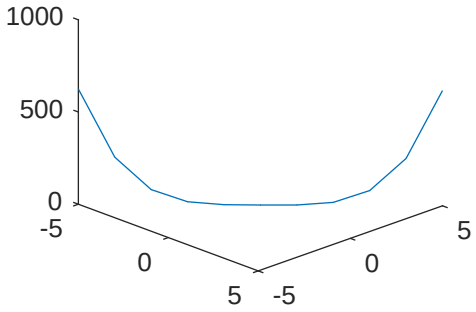

Figure 4: 3-D plot(X,Y,Z)

There are also many commands that lead to the creation of 3D surfaces. Recall that to create a surface of two variables, at each pair of (x, y) a functional expression, $z = f(x, y)$, is evaluated. Usually, we evaluate the z behavior in a certain area of the (x, y)-plane in which z has some interesting properties (*e.g.* it presents a minimum, a maximum or a saddle point). Suppose that a x vector that defines the x-axis (abscissa) and a y vector that defines the y-axis (ordinate) exist and define the (x, y)-plane on which we would like to create the 3D surface of $z = f(x, y)$. To plot the surface, it is needed to create a grid of sample points (most preferable with high density and this is defined by the difference between the elements of x and y) that covers the rectangular domain of the (x, y) plane in order to generate X and Y matrices consisting of repeated rows and columns of x and y, respectively, over the domain of the function. Then these matrices will be used to evaluate and graph the function.

The meshgrid$(\cdot)$ function transforms the domain specified by two vectors, x and y, into matrices, X and Y. You then use these matrices to evaluate the $z = f(x, y)$. The rows of X are copies of the vector x and the columns of Y are copies of the vector y.

Examining the function $(xy)^2$ in the rectangular space (x, y)-plane reads as follows:

```
X=-5:5;
Y=-5:5;
% generating the grid
[x,y]=meshgrid(X,Y);
% computing Z
Z = (x.*y).^2;
% plotting
subplot(1,2,1)
mesh(X,Y,Z)
xlabel('x'); ylabel('y'); zlabel('Z');
title('mesh(X,Y,Z)')
subplot(1,2,2)
surf(X,Y,Z)
xlabel('x'); ylabel('y'); zlabel('Z');
title('surf(X,Y,Z)')
```

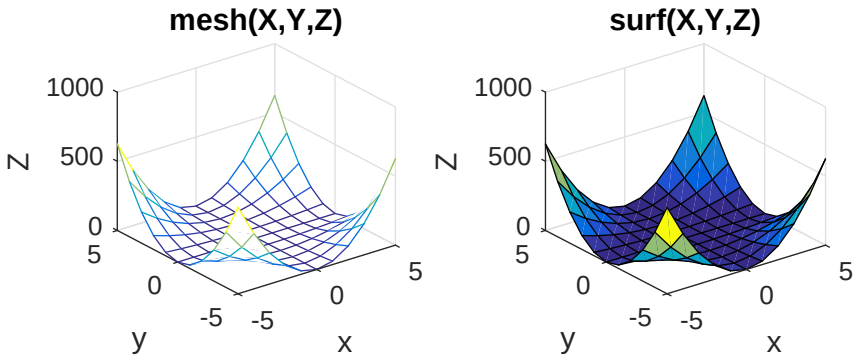

Figure 5: Using mesh(X,Y,Z) and surf(X,Y,Z)

It is also possible to plot the contours:

```
X=-10:10;
Y=-10:10;
% generating the grid
[x,y]=meshgrid(X,Y);
% computing Z
Z = (x.*y).^2;
% plotting contours
[c,h] = contour(x,y,Z); clabel(c,h), colorbar; hold
```

Exercise 4

Take back the code of exercise1.m.

1. Plot the return of assets with time in abscissa.
2. Recalling that we have estimated $r_t = b_0 + b_1 g_t + b_2 BY_t + e_t$, and denoting for tractability that $x_1 = g_t$ and $x_2 = BY_t$. Let us plot the points and the regression plan through a 3D plot:

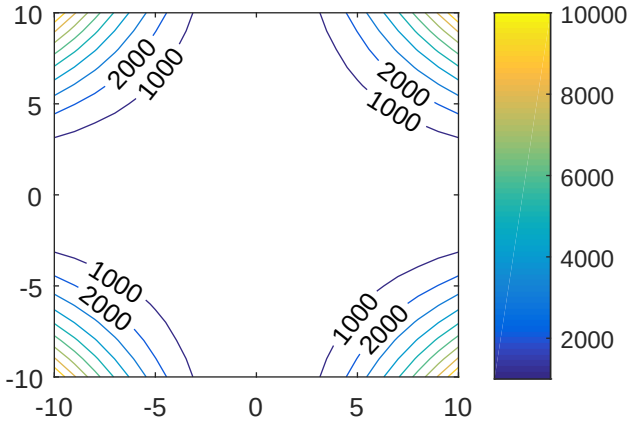

Figure 6: Using contour(X,Y,Z)

(a) Use the scatter plot function scatter3(·) (with 'filled' option) to plot the points.

(b) Next line, write "hold on;" to continue the modification of the plot.

(c) For x_1, create a vector of 15 evenly spaced points in the interval [min(x_1),max(x_1)] named x1fit using the function linspace(·) (check the documentation on MATLAB website). Same for x_1 with a vector called x2fit.

(d) Use the meshgrid(.) to create the grid for each explanotary variable x_1 and x_2 and store the output in [X1FIT,X2FIT]

(e) Create the Z-grid based on estimated paramaters:

```
Z = b(1) + b(2)*X1FIT + b(3)*X2FIT;
```

(f) Perform a 3D plot via mesh() function using X1FIT, X2FIT and Z. *tips:* use labelling with xlabel('GDP Growth'); ylabel('Tbill rate'); zlabel('Asset return')

1.4 Declaring functions

Functions are m-files that can accept input arguments and return output arguments. The name of the m-file and of the funcction should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the Matlab command prompt. Functions are useful for extending the existing Matlab language for personal applications For example a practitioner might want to write various functions that return the theoretical price of a call option contract according to various options pricing models. Options pricing models like the Black and Scholes, the Merton's Jump's diffusion, the displaced diffusion model, Heston's stochastic volatility model, and other can easily be implemented via a different function.

As an example, a function computing the asset return from the stock price is given by:

```
function dl_x = logdiff(x)
        dl_x =  diff(log(x));
end
```

Then we can call this function:

```
r =  logdiff(SP);
g =  logdiff(GNPPC);
```

This will provide the growth rates of each variable.

It is also possible to declare functions with multiple inputs/outputs. Suppose that we are willing to get both the log and the log difference as output, and we want to multiply by 100 or 1 the result:

```
function [dl_x,logx] = logdiff2(x,factor)
        dl_x =  factor*diff(log(x));
        logx = factor*log(x);
end
```

This function must be called this way:

```
% gross value
[r,logSP] =  logdiff2(SP,1);
% percentages
[r100,logSP100] =  logdiff2(SP,100);
```

Note that functions are always stored in external files (i.e. you are not allowed to store function directly in the script file), however you can store multiple function in a single matlab file.

1.5 Conditional statements

Conditional statements enable you to select at run time which block of code to execute. The simplest conditional statement is an if statement.

For example, to report when a random number between [0,100] is above 50, we define the following conditional statement:

```
% Generate a random number
a = randi(100, 1);
% If above 50, send a console message
if a > 50
        disp('a is above 50')
end
```

note: num2str(.) converts a real number into a string, disp(.) displays the argument in the console and [string1 string2] concanates strings into a single string.

It is also possible to contrast the above/below situation using else statement:

```
% Generate a random number
a = randi(100, 1);
% If above 50, send a console message
if a > 50
        disp('a is above 50')
else
        disp('a is equal or below 50')
end
```

Or even contrast three situations: above/below/equal using elseif statement:

```
% Generate a random number
a = randi(100, 1);
% If above 50, send a console message
if a > 50
        disp('a is above 50')
elseif a == 50
        disp('a is equal to 50')
else
        disp('a is below 50')
end
```

1.6 Loops

The for loop repeats a group of statements a fixed, predetermined number of times. A matching end delineates the statements. for statements loop a specific number of times, and keep track of each iteration with an incrementing index variable. For example, preallocate a 3-element vector, and show its content for each row:

```
% taking 3 random numbers in a vector
a = rand(3,1);
% open a loop to display the content of the vector
for i = 1:length(a)
        % show in console
        disp(['row ' num2str(i) ' has value ' num2str(a(i))]);
end
```

Where variable i is the incrementing index variable. Running this code in the command windows returns:

```
row 1 has value 0.49809
row 2 has value 0.90085
row 3 has value 0.57466
```

It is also possible to nest loops. As an example, if we want to show the content of a 3x3 matrix:

```
% taking 9 random numbers in matrix 3x3
a = rand(3,3);
% parsing rows
for i = 1:size(a,1)
        % parsing columns
        for j = 1:size(a,2)
                % show in console
                disp(['row ' num2str(i) ' and column ' num2str(j) ...
                                ' has value ' num2str(a(i,j))]);
        end
end
```

This code returns:

```
row 1 and column 1 has value 0.58145
row 1 and column 2 has value 0.016983
row 1 and column 3 has value 0.4843
row 2 and column 1 has value 0.92831
row 2 and column 2 has value 0.12086
row 2 and column 3 has value 0.84486
row 3 and column 1 has value 0.58009
row 3 and column 2 has value 0.86271
row 3 and column 3 has value 0.20941
```

The while loop repeats a group of statements an indefinite number of times under control of a logical condition. That is, as long as an expression is TRUE, then the segment of executable programming code that is included in the while statement is executed. A matching end delineates the statements. The syntax for the while loop is given by:

```
% taking 3 random numbers in a vector
a = rand(3,1);
% defining the starting value of the incremental variable
i = 0;
while i < length(a)
        % incrementing
        i = i + 1;
        % show in console
        disp(['row ' num2str(i) ' has value ' num2str(a(i))]);
end
```

This code returns exactly the same output as in the first loop example with for statement. The while statement is requires more code than the for statement, however the while loop offers more liberty for the stop conditions of loops.

We can also nests loops and provide exactly the same output as in the second example for for statement:

```
% taking 9 random numbers in matrix 3x3
a = rand(3,3);
% defining the starting value of incremental variables
i = 0;
j = 0;
% parsing rows
while i < size(a,1)
        % incrementing
        i = i + 1;
        % parsing columns
        while j < size(a,2)
                % incrementing
                j = j + 1;
                % show in console
                disp(['row ' num2str(i) ' and column ' num2str(j) ...
                                ' has value ' num2str(a(i,j))]);
        end
        % restarting the increment for columns
        j = 0;
end
```

Exercise 5

Take back the code of exercise1.m.

1. Compute the stock price return using the variable SP with $r_t = \log(SP_t/SP_{t-1})$ through a loop (*i.e.* don't employ the diff() function).
2. Same question, but replace negative values of r_t by 0.
3. Same question but write an external function.

2. Interest rates and bonds valuation exercises

Exercise 6

Consider a P=100 coupon bond of nominal value (face value) N=100 with time maturity T=5 and annual coupon rate c=0.08.

1. After declaring P, N, T and C in MATLAB, define the vector line, denoted cf, in which is reported cash flows inherent to the purchase of the obligation.
2. Determine the internal rate of return (or discount rate) or the rate of return at maturity that cancels the net present value of the income stream.

Remember that:

$$P = \sum_{t=1}^{T} \frac{C_t}{(1+\rho)^t} + \frac{N}{(1+\rho)^T}$$

To do so, employ functions roots(·) and flplr(·). Tips: compte the roots of the problem, and select the unique real root discarding imaginary ones.

3. Determine the internal rate of return using the command irr(·).

4. Determine the internal rate of return using the non-linear solver fsolve(·): A working example of fsolve() for a two period bond: $P = (1+\rho)^{-1}cN + (1+\rho)^{-2}(1+c)N$ is obtained:

```
f=@(x)([
        -P + C/(1+x(1)) + N*(1+c)/(1+x(1))^2
]);
x0=[0];
rho=fsolve(f,x0)
```

Adapt this code for the exercise.

Exercise 7

Electricité de France SA has capped a $12.4 billion global fundraising in 2014 by selling the first 100-year bonds in Europe with coupon rate of 6.125%.

1. Assuming that each obligation has P=N=100, determine the internal rate of return of this bond.

2. Three years later, the price of bond is now 90. Determine the new rate of return.

Exercise 8

Consider a bond, repaid at maturity with facial value N, maturity n and coupon c. We want to determine the price of this obligation P with the rate of return r. (i) the first method aims at determining the analytical solution of P and code it. (ii) the second solution use a loop parsing the vector of cash flows. (iii) the last one employs the function pvvar(·).

1. Determine the price of the obligation using the three methods, assuming that N=100, c=0.08 and n=5.

2. Adapt the code for each method in order to compute the bond price for different internal rate of return. Construct a vector r=[0.08 0.09 0.10 0.11 0.12] and use this variable internal rate of return.

3. Discuss your result regarding the bond price.

Exercise 9

Consider a zero-coupon bond with price P, nominal value N and maturity n. Letting r denote the variable internal rate of return, the bond price reads as:

$$P = \frac{N}{(1+r)^n}$$

We seek to measure the implication of maturity n over the bond price P following a change in the rate of return r.
We fix $N = 100$ and consider two possible maturities $n = [5, 20]$.

1. Determine the variation of the bond price when r change from 0.08 to 0.09. Contrast your result for the two maturities $n = [5, 20]$.
2. Same exercise, but employ loops to store your calculation of bonds price into a matrix 2x2.

Exercise 10

These a pure-programming exercises:

1. Write a function to print all natural numbers from 1 to n using loop.
2. Write a function to print all natural numbers in reverse (from n to 1).
3. Write a program to count number of digits in any number.
4. Write a function to enter any number and print its reverse.
5. Write a function that use as input a person's name and greets her with her name.
6. Modify the previous program such that only the users Trump and Clinton are greeted with their names.
7. Write a program that takes as input a number n and gives him the possibility to choose between computing the sum and computing the product of 1,. . . ,n.
8. Write a program that prints a multiplication table for numbers up to 12.
9. Write a program that prints the next 20 leap years.

Exercise 11

We consider a bond with a 8% interest rate, with a coupon rate equal to 10%. This bond was acquired on the 10th of August 2007 and matures 31st of December 2020.

1. Using the function bndprice, find the clean price (i.e. le pied de coupon) of the purchase date and the dirty price (i.e. le coupon couru).[a]
2. Determine (using the bndprice function matlab price) the **current** clean

and dirty price. To do this, you use the command today. Use functions datestr(today) and datenum (datestr(today)).

3. Determine (using the bndprice function matlab price) the clean and dirty price when the coupon is paid once per year + the bond face value is 200 + the convention is to consider all the months of the year are assumed to have the same number of days, namely 30 days, a year of 360 days (European 30/360).

[a]The acquisition of a bond may occur between two coupon payments. Thus the price of a bond is not constant over time: each time a coupon is detached (paid), the bond price decreases in the value of the coupon. This leads us to distinguish the clean price (the bond price regardless of the coupon) and the dirty price (the price "accrued interests" to which is added the accrued interest).

Exercise 12

Consider an obligation acquired on the 19th of March 2000 which matures on the 15th of June 2016. The face value is 1000 and the coupon rate is 5%.
Calculate and plot the curve of the price "accrued interests" based on performance, for performance values between 1% and 20% (the return distribution will be uniform) using two different methods. Your chart should include a title. The x-axis and ordinates must have a title. The curve should be black. You will also show the horizontal and vertical lines that extend the numerical values of the axes.

Exercise 13

Consider an obligation acquired on the 31st of January 2006 which matures on the 3st December 2016. The coupon rate is 8%.
Calculate and plot the price "accrued interests" based on performance for the price values between 50 and 150 (the prize must go from 5 to 5). Your chart should include a title. The x-axis and ordinates must have a title. The curve will be green. To do this you use the command bndyield whose options are similar to that of bndprice order.

References