Chapter 1

An introduction to MATLAB®with applications

MATLAB is a numerical computing environment and a programming language. It is one of the most popular programming tools, whose fields of application range from finance to robotics, from computer vision to communications, and much more. One of the main reasons for its spread is due to its peculiar language, which is based on matrix calculus. Developed by MathWorks, MATLAB works on different operating systems, including Microsoft Windows, Mac OS X, and Linux. Chapter 1 is devoted to an introduction to the MATLAB language and development environment including programming, numerical calculation, and visualization applied to simple calculus and financial problems.

This chapter is structured as follows: first, we introduce some basic elements for computing in MATLAB. Then, we describe how to build an ordered sequence of commands in a single file, in order to easily and efficiently implement all instructions to reproduce a model. Furthermore, we address the main statements of MATLAB that allow the user to execute expressions repeatedly, or to impose or check certain conditions. Finally, some exercises on programming and on Financial Mathematics are proposed. We point out that the first part of this chapter is devoted to teaching MATLAB to those who use it for the first time, while the part concerning Financial Mathematics is addressed to readers who already have at least a basic knowledge of this topic.

1.1 MATLAB®basics

In this section, we discuss how to start programming with MATLAB. In particular, we describe the basic syntax (i.e., how commands should be structured, so

that MATLAB executes appropriate calculations and operations), the commonly used operators and special characters, and some of the variables, constants, and functions, which are predefined in MATLAB.

1.1.1 Preliminary elements

MATLAB is a useful software both for numerical computations and for graphical visualization. MATLAB stands for Mat*rix* Lab*oratory*, and the arrays are the basic unit of its language. The numerical computations, previously mentioned, aim at the practical implementation of mathematical models. MATLAB has several features: it is an interpreter of commands, a programming language, and provides wide graphics facilities. Furthermore, it is able to effectively communicate with different applications (e.g., Excel).

In addition to the basic platform, MATLAB has a variety of tools, called Toolboxes, for specific applications. For our purposes, the Statistics, the Optimization and the Financial Toolboxes are the most useful ones.

A software strictly linked with MATLAB is Simulink, a graphical programming environment for modeling, simulating and analyzing complex and dynamic systems, but this topic is beyond the aim of this book.

Let us start by running MATLAB. One can generally observe a main window divided in several subwindows, as shown in Fig. 1.1. In this case, we have the Current Folder on the left, the Command Window at the center, and the Workspace and the Command History on the right. The Current Folder (or Directory) represents the place where we are working, the Command History lists, in chronological order, the last commands used, and the Command Window contains the prompt (the blank line that follows the symbol >>), where we insert instructions, run MATLAB programs, and so on. Note that the complete path of the Current Folder is generally shown above the Command Window. Fig 1.1 shows an example of the MATLAB main window (corresponding to version R2019b), where, at the top, we can see three sheets: Menu, Plots, and Apps. Generally, these contain elements for managing and elaborating the MATLAB files; for finding bugs; for setting up the parallel programming; for configuring the desktop of MATLAB; for accessing to Help menu. Note that the Help menu and the Mathworks website (http://www.mathworks.it/matlabcentral/) are extremely useful for solving problems, especially for MATLAB beginners.

Variable assignment

Let us consider how to define a variable, for instance, a scalar. As mentioned above, the building block of MATLAB is the array, i.e., an orderly arrangement

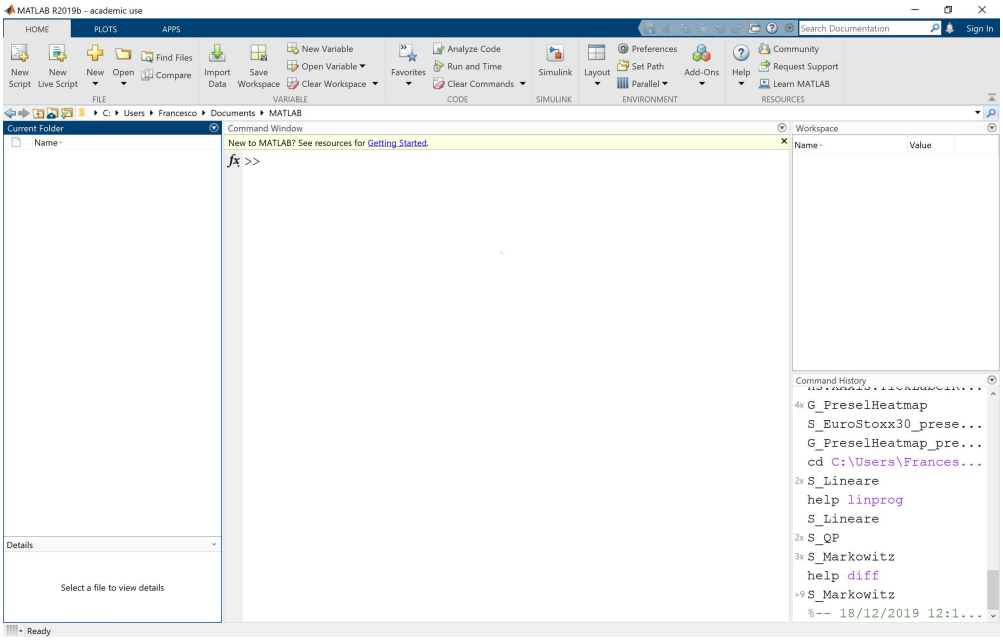

Figure 1.1: Example of the layout of the MATLAB main window.

of cells, usually in rows and columns. If the cells are numbers, we obtain scalars, vectors and matrixes. Otherwise, if the cells are strings, we have an array of strings, and so on. Consider now a scalar, that is a `1×1` matrix. To define a new variable equal to `3.2`, one can write on the prompt the following instruction

```
>> x=3.2;
```

where `x` is the name of the variable, and `3.2` is its assigned value. The name of a variable must not exceed 31 continuous characters and must not contain mathematical operators (`-`,`+`,`*`,`=`), apostrophe, punctuation, slash, or backslash. If one writes a scalar on the prompt (>>), a default variable is generated and it is indicated by `ans`. For instance, one can type

```
>> 1.67;
```

Note that adding the semicolon (`;`) at the end of an instruction avoids showing the output on the Command Window and decreases the running time of an algorithm. Conversely, in order to visualize the assigned variables, one can run the name of a variable (for example, `x` or `ans`) on the prompt; otherwise, it is possible to directly see the variable in the Workspace subwindow (see Fig 1.1).

There are different options to visualize numbers associated to the assigned variables. Below, various formats are illustrated:

`FORMAT (SHORT)`	four digits after the comma	`2.7000`
`FORMAT LONG`	fifteen digits after the comma	`2.700000000000000`
`FORMAT SHORT E`	exponential short form	`2.7000e+000`
`FORMAT LONG E`	exponential long form	`2.700000000000000e+000`
`FORMAT SHORT G`	best representation with four digits	`2.7`
`FORMAT LONG G`	best representation with fifteen digits	`2.7`
`FORMAT BANK`	numbers expressed in euro and euro cents	`2.70`

Furthermore, MATLAB provides some predefined variables listed in the following table:

Variable	**Meaning**
`ans`	Value of the last operation, to which a name is not assigned
`i`	Imaginary unit
`pi`	Approximation of π
`eps`	Machine accuracy
`realmax`	Maximum positive machine number
`realmin`	Minimum positive machine number
`Inf`	inf (namely a number greater than `realmax`)
`NaN`	Not a Number (e.g., $\frac{0}{0}$)
`computer`	the type of your computer
`version`	MATLAB version

Workspace

The Workspace is represented by a subwindow and is the place where assigned variables are stored. It can show several details of the assigned variables, for instance:

Name	name of the variable
Value	assigned value
Size	array dimension (rows $\times$ column)
Bytes	allocated memory
Class	variable types

The MATLAB variable types are *double*, *cell*, *sparse*, *char*, *struct*, and *uint8*. By default, MATLAB works with variables in double precision. A cell array is an array whose elements are, in turn, arrays with different possible dimensions.

A sparse matrix is generally a matrix with most of the elements equal to zero, where, therefore, only the elements different from zero are represented. For more details, see the MATLAB Help. Another feature of the Workspace is that it allow us to open a window similar to an Excel spreadsheet by double-clicking on the variable.

Note that MATLAB is case sensitive. Thus, the variables `x` and `X` are considered different variables.

Arithmetic operations

The basic calculation operations are the arithmetic ones: sum (`+`), subtraction (`-`), multiplication (`*`), division(`/`), and power (`^`). For instance, if we have to calculate

$$x = \frac{6 + 9^3 - 8/2}{2 * (2+1)^4}$$

we can write the following instruction on the command prompt:

```
>> x=(6+9^3-8/2)/(2*(2+1)^4);
```

In the MATLAB environment, the arithmetic operations work in the following order:

Order	**Operation**	**Example**
I	Brackets	`(3+4)*4=7*4`
II	Power	`3^2+2=9+2`
III	$*$ /, from left to right	`2*3/5=6/5`
IV	$+$ $-$, from left to right	`8-5+3=3+3`

However, one can change the order of the operations by appropriately applying parentheses `()`. If an instruction consists in a long expression, one can divide it in two or more lines by writing three consecutive points, as shown in the following example:

```
>> x=1/2+(9*3)^3-...
3+12;
```

These instructions compute the expression $x = \frac{1}{2} + (9 * 3)^3 - 3 + 12$.

1.1.2 Vectors and matrices

The building block of MATLAB is the array. An array is an ordered succession of memory locations that contains a collection of elements of the same type. As specified in Section 1.1.1, an array can be made by six types of variables: `char`,

`double`, `sparse`, `cell`, `uint8`, `struct`. Consider now an array a (1×5), namely 1 row and 5 columns (a row vector), that contains integer numbers from 1 to 5. As shown below, we can create a in several ways:

```
>> a=[1 2 3 4 5];
>> a=[1,2,3,4,5];
>> a=1:5

a =

     1   2   3   4   5
```

To generate an array b (5×1), namely 5 rows and 1 column (a column vector), that contains integer numbers from 1 to 5, for instance, we can write:

```
>> b=[1;2;3;4;5]

b =

     1
     2
     3
     4
     5
```

Note that the space or the comma (`,`) separate elements on the same row, while the semicolon (`;`) separates elements on the same column. To define an array c (2×5), i.e., a matrix with 2 rows and 5 columns, we can opportunely combine the space (or the comma) and the semicolon as follows:

```
>> c=[15 2 4 -1 9; 1 5 -7 0 7]

c =

    15    2    4   -1    9
     1    5   -7    0    7
```

Detecting the dimensions of a matrix becomes very useful when the number of its elements is too large to visualize them directly. There are two built-in functions that allow for the identification of array dimensions, namely `size` and `length`. Applying the function `size` to a generic matrix $m \times n$ generates the number of rows (m) and columns (n) as outputs. In the case of the matrix c just defined, we have

```
>> size(c)

ans  =

     2   5
```

The function `length` is usually used to detect the number of elements in a vector. However, applying the function `length` to a matrix gives the greater number between m and n as output. In the previous example, we obtain

```
>> length(c)

ans  =

     5
```

Then, we have that `length(c)=max(size(c))`.

Another important issue to address is how to access and manipulate the elements of an array. For instance, given the matrix c, we can detect the first element of the second row in the following way:

```
>> c(2,1)

ans  = 1
```

A similar procedure can be used for a vector. For example, given the vector b, to identify its second element, we can write

```
>> b(2)

ans  = 2
```

Furthermore, equating a specific element to a scalar replaces that element with the newly assigned scalar. For instance, continuing the previous example, we have

```
>> b(2)=6

b  =

     1
     6
     3
     4
     5
```

To modify an element or a set of elements in a matrix, we have to suitably specify two indices. For instance, considering the matrix c, to modify the element $c_{1,3}$, we can write

```
>> c(1,3)=21

c =

    15     2    21    -1     9
     1     5    -7     0     7
```

Note that running the command `d(3,4)=3` without predefining the matrix `d` generates a 3×4 matrix with all elements equal to zero, except for that of position $(3, 4)$:

```
>> d(3,4)=7

d =

     0     0     0     0
     0     0     0     0
     0     0     0     7
```

1.1.3 Basic linear algebra operations

As mentioned above, the natural environment of MATLAB is the matrix calculus. Its standard operations are the following:

- `+` $\rightarrow$ element-by-element sum of matrices (or vectors)
- `-` $\rightarrow$ element-by-element difference of matrices (or vectors)
- `*` $\rightarrow$ row-by-column product of matrices (or vectors)

Note that, for the sum and the difference, i.e., for element-by-element operations, the matrices (or the vectors) involved must have the same dimensions. While for the row-by-column product the number of columns of the first matrix must be equal to the number of rows of the second matrix. For instance, given the vectors a and b previously introduced, and the column vector `a_=[1;2;3;4;5]`, we can implement the following operations

```
>> a_+b

ans  =
         2
         8
         6
         8
        10
```

```
>> a_-b

ans  =
         0
        -4
         0
         0
         0
```

```
>> a*b

ans  =
        63
```

Note that, if we compute the difference between the vectors a e b, we obtain the following output:

```
>> a-b

Error using ==> -
Matrix dimensions must agree.
```

Similarly, we can apply the basic linear algebra operations among matrices. For instance, consider the following matrices

$$e = \begin{pmatrix} 2 & 1 & 2 \\ 3 & 0 & 1 \end{pmatrix} \quad \text{and} \quad f = \begin{pmatrix} 2 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix},$$

and calculate their product:

```
>> g = e*f

g =
      5      4
      6      4
```

This result is a consequence of the matrix multiplication rule, which, in terms of dimensions, works as follows: $(2\times3)(3\times2) = (2\times2)$.
Other standard operations are the *transposition* and the *inversion* of matrices. Below we show these operations by means of two examples. Firstly, we transpose the matrix e

```
>> e'

ans =
     2 3
     1 0
     2 1
```

Note that if we consider

```
>> f*e'
```

we obtain the following output

```
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Indeed, in this case, the number of columns of the first matrix is not equal to the number of rows of the second matrix.
Furthermore, we can invert the non-singular square matrix g defined above as follows:

```
>> g_inv=inv(g)

g_inv =
    -1.0000    1.0000
     1.5000   -1.2500
```

Obviously, it must hold that $g^{-1} \cdot g = I$, where I is the identity matrix

```
>> g_inv*g

ans =
     1.0000   0.0000
     0.0000   1.0000
```

For further information about matrix inverse, see the MATLAB Help (`mldivide` \).

Example 1 (Linear equations system) *Let us consider the following system of linear equations*

$$\begin{cases} 3x_1 + 4x_2 &= 25 \\ x_1 - 6x_2 &= 50 \end{cases}$$

In matrix notation, we can write that $Ax = b$, *where* $A = \begin{pmatrix} 3 & 4 \\ 1 & -6 \end{pmatrix}$, $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, *and* $b = \begin{pmatrix} 25 \\ 50 \end{pmatrix}$. *Since* A *is an invertible matrix (i.e.,* $\det(A) = -22 \neq 0$*), we have* $x = A^{-1}b$. *Using MATLAB, we can write:*

```
>> A = [3 4; 1 -6];
>> b = [25;50];
>> x = A\b

x  =

     15.9091
     -5.6818
```

Note that the product between the inverse of A (A^{-1}) and b can be obtained with the backslash (\) command.

1.1.4 Element-by-element multiplication and division

In many cases, it could be useful to perform element-by-element power, multiplication, and division operations among matrices. MATLAB provides these operations by adding a dot (`.`) before the commands `^`, `*` or `/` (i.e., `.^`, `.*`, `./`). For instance, to define a new vector whose elements are squares of the elements of the vector b (modified at the end of the previous section), we can write

```
>> b.^2

ans  =

     1
     36
     9
     16
     25
```

Observe that the expression `b^2` does not make sense, since it corresponds to the matrix product $(5\times1)(5\times1)$.

Example 2 (Element-by-element operations) *Compute the element-by-element product and division between the vectors* `a_2=[2,2,2,2,2]` *and* `b_2=[4,8,16,2,10]`. *On the command prompt, we can type:*

```
>> a_2.*b_2

ans  =

     8   16   32   4   20
```

and

```
>> a_2./ b_2

ans  =

      0.5000   0.2500   0.1250   1.0000   0.2000
```

Note that the involved vectors (or matrices) must have the same dimensions.

1.1.5 Colon (:) operator

This command is frequently adopted for several purposes. For instance, it can be used to create vectors with equally spaced elements, such as a set of numbers that identifies the index $k = 1, 2, \ldots, n$, by writing `>> k = 1:n`. As a consequence, the colon operator (`:`) can be exploited to select rows or columns of an array. For instance, considering the matrix c of Section 1.1.2, we can select the first two rows and columns of c as follows

```
>> c(1:2,1:2)

c  =

    15   2
     1   5
```

or we can extract the third column as follows:

```
>> c(:,3)

ans  =

     21
     -7
```

More generally, the command `Vector=Start:Step:End` generates a vector with equally spaced elements between the number `Start` and the number `End`. For

example, to create a vector of odd numbers that starts from 1 up to 15, we can write:

```
>> h=1:2:15

h  =

     1  3  5  7  9  11  13  15
```

By default, `Step` (when it is omitted) is equal to 1. If `Step` is positive, then the value of the elements of `Vector` increases. Otherwise, if `Step` is negative, then the elements of `Vector` decrease as shown in the following example:

```
>> k=12:-2:4

k  =

     12  10  8  6  4
```

Note that the colon operator (`:`) will be extensively used when addressing the loop scheme in MATLAB (see Section 1.3.2).

When `Step` is not an integer, it could be worth using the *built-in* function `linspace`, where the inputs are the first and the last element of the vector, and the number of components of the vector. For example, we can write

```
>> l=0; m=1; n=8;
>> x=linspace(l,m,n)
x=
   Columns 1 through 4
    0  0.1429  0.2857  0.4286
    Columns 5 through 8
    0.5714  0.7143  0.8571  1.0000
```

It generates a vector x with equally spaced elements with a step equal to $\dfrac{m-l}{n-1}$. More in detail, the elements of x are

$$x_i = l + (i-1)\frac{m-l}{n-1} \quad \text{with} \quad i = 1, ..., n;$$

where x_i indicates the *i-th* element of x.

1.1.6 Predefined and user-defined functions

MATLAB provides a number of mathematical functions. A non-exhaustive list of predefined function is reported in the following table

Function	Meaning
`sin, cos, tan`	sine, cosine, tangent
`asin, acos, atan`	arcsine, arccosine, arctangent
`exp`	exponential
`sinh, cosh`	hyperbolic sine, hyperbolic cosine
`tanh`	hyperbolic tangent
`log, log2, log10`	logarithm to the base e, 2, and 10
`sqrt`	square root
`abs`	absolute value
`sign`	sign function

The scheme of a predefined functions is `Output=function(Input)`, where `Output` and `Input` can be, in general, matrices. For instance, if we apply the function `sin` to the matrix c of Section 1.1.2, we have

```
>> y=sin(c)

y =

    0.6503    0.9093    0.8367   -0.8415    0.4121
    0.8415   -0.9589   -0.6570         0    0.6570
```

`inline` Function

MATLAB also gives the possibility to create any function with one or more independent variables by means of the `inline` function. As specified in the MATLAB Help, to build a generic function `anyf`, we can use the following syntax:

```
>> anyf=inline('expr','arg1','arg2',...,'argn');
```

where the string `expr` provides the mathematical expression of the function considered, and `arg1`,`arg2`,...,`argn` indicate the independent variables of this function. Below we report two examples of functions with one (on the left) and two (on the right) independent variables:

```
>> f=inline('x.^2.*tanh(x)')
f=
     Inline function:
     f(x)=x.^2.*tanh(x)
```

```
>> g=inline('sqrt(x.^2+y.^2)','x','y')
g=
     Inline function:
     g(x,y)=sqrt(x.^2+y.^2)
```

Note that both the expressions and the arguments must be included within apostrophes. Therefore, it is possible to evaluate the two previous functions w.r.t. any fixed points. For instance, we can type

```
>> x=2.5;          | >> x1=1.5; x2=3.7;
>> y=f(x)          | >> z=g(x1,x2)
                   |
y  =               | z  =
                   |
     6.1663        |      3.9925
```

Furthermore, if the mathematical expression `expr` is properly written, then the inputs `arg1,arg2,...,argn` of the generic function `anyf` can also be matrices. For instance, referring to the function `g`, we have

```
>> x1=[0 1]; y1=[1 2];
>> g(x1,y1)
ans =
      1 2.2361
```

Observe that the names of the inputs do not have to be necessarily the same of those of the variables `arg1,arg2,...,argn` used to define the `inline` function.

Anonymous Function

An alternative to `inline` is the Anonymous Function, that will substitute the former in a future release of MATLAB. However, the syntax to define an Anonymous Function is similar to that of `inline`. For instance, to create an anonymous function that calculates the quartic root of a variable, we can write

```
>> q_root=@(x) x.^(1/4)
```

where `q_root` is the name of the function. The content in brackets following `@` represents the independent variables of the mathematical expression of `q_root`. Then, we can calculate the quartic root of $x = 81$ as follows

```
>> x=81;
>> y=q_root(x)
y=
    3
```

Note that an Anonymous Function can include other anonymous functions. However, for further details, see the MATLAB Help.

1.2 M-file: Scripts and Functions

The main programming tools of MATLAB are represented by Scripts and Functions. They can be written by the MATLAB Editor, which provides several facilities in the programming review. Script and Function are saved as `.m` files, a specific extension of MATLAB (also called M-file). In a nutshell, in an M-file we can write all the instructions to implement a model and run Script or Function on the prompt. For convenience, we summarize the main characteristics and differences between Script and Function in the following table

Script	Function
works on variables in the Workspace	*the interior variables are local*
does not accept input variables	*can accept input variables*
does not have output variables	*can have output variables*
useful for running a number of instructions	*useful for solving recurrent applications*

For instance, consider the following M-files:

S_HypRightTri.m

```
a= 3;
b= 4;
c=sqrt(a.^2+b.^2)
```

F_HypRightTri.m

```
function [c] = F_HypRightTri(a,b)
c=sqrt(a.^2+b.^2)
end
```

In this case, both Script (left side) and Function (right side) have the same objective, the calculation of the hypotenuse of a right triangle. We can run these M-files on the Command Window as follows:

```
>> S_HypRightTri
c=
   5
```

```
>>c= F_HypRightTri(3,4)
c=
     5
```

Note that a Function M-file must be saved with the same name as its reference function.

Since the construction of an M-file is a key issue in MATLAB programming, we analyze, in detail, the resolution of a linear system, where the incomplete matrix A is a Hilbert matrix, an example of an ill-conditioned matrix. Then, in

Example 3 the aim is to solve the linear equation system $Ax = b$ and to compute the absolute and relative errors of the obtained solution (due to the instability of the solution when A is an ill-conditioned matrix).

Example 3 (Ill-conditioned matrix: Script) *Open a Script on the Current Folder and call it, e.g.,* `S_IllCond.m`. *Write the following lines of code*

```
clear all                      % delete all variables
                               % in the Workspace
close all                      % close all figures
n=30;
A=hilb(n);                     % define an n-by-n
                               % Hilbert matrix
x=[1:n]';                      % x is a column vector of
                               % n elements
b=A*x;                         % define the vector b
x1=A\b;                        % solve the linear system Ax=b
                               % note that the back-slash
                               % stands for A^{-1} * b
abs_error=norm(x-x1)           % calculate the absolute error
rel_error=abs_error/norm(x)    % calculate the relative error.
```

On the left of the code, we report the syntax used to solve our problem, while, on the right, some comments are provided. Comments are useful in managing long Scripts (think about a code of more than 300 lines) and can be inserted by typing `%` before the desired note. Thus running the nae of the Script on the command prompt, MATLAB automatically executes each code line of the Script in chronological order. In addition, if the Script presents some errors, then it is automatically highlighted on the Command Window with indication of the bugged line's location.
The other important M-file is the Function, which is used to implement a block of commands frequently applied in solving a problem. For convenience, we solve the same problem of Example 3.

Example 4 (Ill-conditioned matrix: Function)
Open the editor, and save an M-file with the same name of the reference function, namely `F_IllCond.m`*, and, therefore, type the following syntax:*

```
function [abs_error,rel_error] = F_IllCond(n)  % define inputs
                                               % and outputs
% Solve a linear equations system              % text visualized
                                               % in the Help
A=hilb(n);                                     % n-by-n Hilbert
                                               % matrix
x=[1:n]';                                      % a column vector
                                               % of n elements
b=A*x;                                         % define vector b
x1=A\b;                                        % namely A^-1 · b
abs_error=norm(x-x1);                          % absolute error
rel_error=abs_error/norm(x);                   % relative error
```

By means of the Function `F_IllCond`, *it is possible to compute the absolute and the relative errors for any dimension n of the square matrix A. Given the* 4×4 *and* 7×7 *Hilbert matrices, one can write the following instructions on the prompt*

```
>> [abs_err4,rel_err4]=F_IllCond(4);
>> [abs_err7,rel_err7]=F_IllCond(7);
```

Note that the variables in a Function (as for `inline` *and for an Anonymous function, see Section 1.1.6) are local, and then the names of input and output variables can be different from their names used in the code.*

Let us examine the elements of a Function separately. The first line generally contains the syntax definition for the new function as follows:

`function [outputs]=NameOfTheFunction(inputs)`.

Note that `inputs` and `outputs` can be one or more arrays of different types (numbers, strings, etc.). The following lines could contain a description of the Function, which is possible to visualize by typing `help NameOfTheFunction` on the prompt. After that, there is the body of the Function, where all the variables are local.
In the following table, we synthetically list all possible contents in a Script, or in a Function:

Content of a Script/Function
Commands to load and save data
Assignments
Comments
Calculations
Calls of other defined functions
`for` and `while` loops
`if`, `elseif`, `else` schemes
Commands to construct graphs
etc.

1.3 Programming fundamentals

In this section, we deal with the main basic schemes of programming in MATLAB.

1.3.1 `if`, `else`, and `elseif` construct

In some cases, it could be useful to run statements only when specific conditions are verified. The command `if` evaluates an expression that represents a condition. If that condition is true, namely the real part of that expression has all non-zero elements as outputs, then it executes a group of statements. The general form of the `if` scheme is

```
if expression
statements
elseif expression
statements
else expression
statements
end
```

where the `elseif` and `else` lines are optional. Indeed, it is possible to construct a scheme of conditional statements by means of the following short form

```
if expression
statements
end
```

To formulate one or more conditions, *relational* and *logical* operators can be exploited. Below we provide a list of all *relational* operators available.

Relational operator	Description
$<$	smaller than
$<=$	smaller than or equal to
$>$	greater than
$>=$	greater than or equal to
$==$	equal to
$\sim=$	different from

The relational operators can be used to verify a condition between arrays with the same dimensions. More precisely, if an expression is true, i.e., the corresponding output is non-empty, and has all non-zero elements, then the statements are executed. Otherwise, the expression is false and the statements are not performed.

Example 5 (Relational operators) *Consider the following two* 3×3 *matrices, named A and B*

```
>> A=[11 2 3; 5 10 3; 2 3 2];
>> B=[2 7 6; 9 10 7; 2 3 2];
```

and check which elements are equal. Thus, we can type the following expression on the prompt

```
>> A==B
ans=
      0   0   0
      0   1   0
      1   1   1
```

Note that, in the output, `1` *indicates that the relation is true, while* `0` *indicates that the relation is false.*

The *logical* operators in MATLAB are summarized in the following table:

Operator	Description
&	and
|	or
$\sim$	not