http://lucaregis.altervista.org

Additional Statistical Training

Dr. Luca Regis • 2015/2016 • University of Torino

Last Revision: March 1, 2016

Table of Contents

MATLAB Basics

MATLAB (whose name derives from *Matrix Laboratory*) is a mathematical software that uses vectors and matrices as basic data types. It is both a computing environment and a programming language. It is a commercial software, developed by Mathworks, Inc. Nonetheless, it has become widely popular, thanks to the possibility of using both built-in commands and user-defined functions. Several packages of commands, called toolboxes, are present, containing tools to deal with particular types of applications. Most useful toolboxes to finance students are the Financial Toolbox, the Statistics Toolbox, the Optimization Toolbox. An open-source software, very similar to Matlab, exists, called Scilab. The language and commands are very similar, although syntax can sometimes be different.

MATLAB's interface

MATLAB's interface (Desktop) has the following elements:

- Command window: commands are inserted here;
- Command history: lists the most recent commands inserted in the command window. They can be dragged and dropped to the command window;
- Current folder: the current folder the user is working in. It can be changed from the bar in the upper part of the screen;
- Workspace: shows the objects that the user can handle;
- Home: it is the bar containing basic File, Variable and Code commands, Preferences and Resources
- Plots and Apps: are shortcuts to plot commands and to built-in applications.

Basic use of the command window: Matlab as a calculator

The command window can be used directly as a place to insert calculations. Basic arithmetic operators are:

- + : sum
- - : difference
- * : product
- / : division

Operations are executed following the usual priority rules (products and divisions before sums and differences, operations in brackets have priority). As for other basic operators, arguments are placed in round brackets. Here is a list of the most used basic commands and operators:

- "^": $n\hat{}k$ computes the k-th power of the number n;
- sqrt (n): computes the square root of the number n;
- exp(n): computes the exponential of the number n;
- log(n): computes the natural logarithm of the number n;
- log2, log10: compute the base 2 or 10 logarithm;
- sin(n), cos(n), tan(n): compute the sine, cosine and tangent of n (expressed in radiants, the function pi can be used to transform degrees into radiants)

In order to get informations about any MATLAB built-in function, it is possible to type "help <function-name>" on the command window. The command *clc* clears the command window, while the command *diary* creates a txt file in which the whole session is recorded, until the user calls the command "diary off".

Exercise Verify the relationship $tan(45°) = \frac{sin(45°)}{cos(45°)}$.

Variables

Results of the calculations we have described in the previous section through commands in the command window are collected in an "ans" variable. Variables are tools which are used to store data. Is a sort of "box" which the user can define directly from the command window:

```
» a=4
a=4
```

Once a variable has been defined, it displays in the Workspace and it can be used in further computations.

```
» b= 3+a
b=7
```

Notice that putting a ";" after the command prevents MATLAB from showing the output in the command window (but not from storing the new variables/modifying the existing ones in the Workspace). Variables are almost always stored as arrays or matrices and can belong to different classes. We will consider almost exclusively numeric variables, in particular numeric arrays/matrices

with double-precision floating numbers as elements. Strings can be stored into variables of class *char*, which are again arrays containing numeric codes for the characters. Other interesting and more advanced classes are:

1. *struct*: they are data types that group "collections" of related data, i.e. arrays of varying classes and size, using data containers called fields;
2. *cell*: the cell arrays store arrays of varying classes and size, using indexed data containers called cells, where each cell can contain any type of data.

The command *save* is used to save the variables in the workspace into a file (whose extension is .mat). The command *load* can then be used to load the variables in a .mat file. The variables contained in the .mat file will be displayed in the Workspace. *clear* removes all the variables from the Workspace. *who* (*whos*) lists the current variables in the workspace (adding their properties).

Exercise
Create a variable $a = 3 * \ln 5 + 2$. Use it to compute $b = \frac{\sqrt{a}}{7+a}$. Save both variables in a file named 'both.mat'. Then, save only variable a in a file called 'one.mat'. Use MATLAB help to figure out the correct syntax of the *save* command.

Precision, rounding and other useful commands

By default, the number of digits with which variables are stored is 32. The command *digits* shows this and allows to modify it when working with the *vpa*, variable precision arithmetics, function, that allows to control the precision, i.e. the accuracy of approximations in the numerical computations that Matlab performs. Also the format in which the variables are displayed can be modified using the *format* command. The standard is *short*, which displays 4 digits. Other standard formats are:

1. *long*: it displays 16 digits;
2. *bank*: it displays 2 digits;
3. *short/long e*: they display 4/16 digits plus the exponent;
4. *rat*: it displays the closest rational expression.

The following commands can be used to round numbers in Matlab:

1. fix: rounds towards zero;
2. floor: rounds towards minus infinity;
3. ceil: rounds towards plus infinity;

4. round: rounds towards nearest integer.

Other useful commands are:

1. rem: Computes the remainder of a quotient. Syntax is rem(X,Y) to obtain the remainder of X/Y;
2. sign: returns the signum of the input, returning a value +1 if the input is positive, -1 if negative;
3. abs: returns the absolute value of the input.

In some cases, if one wants to obtain results with a high degree of accuracy, standard functions (exp, log) are not sufficient. This is the case for instance when one wants to compute $exp(x) - 1$ or $log(1 + x)$. The problem lies in the fact that the distance between 1 and the following next largest double-precision floating number arises because of the way the software stores numbers. This distance can be displayed using the function eps(1). Its approximate value is $2.2204 * 10^{-}16$, which is called the roundoff level. In this specific case, one can make use of the functions expm1 and logp1 to overcome this rounding issue.

Exercise Check out the difference between exp(1*10^{-}17)-1 and expm1(1*10^{-}17).
Try the different rounding commands on the number 5/4. What is the difference between the result obtained with fix and floor? Try again with -5/4.

Arrays and matrices

As mentioned above, matrices are the basic structure in MATLAB. Even variables containing one single number are stored as 1x1 double arrays.

Creating vectors and matrices

Vectors are nx1 (column vector) or 1xm (row vector) matrices, containing n and m scalars respectively. They can be created in Matlab from the command window:

```
»R=[1 2 3]
»C=[1;2;3]
```

The semicolon separates different rows. The transpose operator is simply $'$. Once a vector is created, it is possible to access any element in it: R(1) accesses the first element of vector R, R(2) the second and so on. $R(i:n)$ identifies the vector whose elements go from the i-th element of R to the n-th. The command *end* defines the index of the last element of the vector: R(1:end)=R.

Notice that the operator ":" can also be used to define arithmetic progressions. The command "1:2:11" returns a vector whose elements are in an arithmetic progression which starts from 1 and ends with 11 and has step 2. The step can also be negative. Matrices can be easily created from the command window as well:

```
»A=[1 2;3 4]
```

It is possible to view or access (and modify) any element of a matrix by specifying its position: A(i,j) denotes the element of A in the i-th row and j-th column. The colon operator can be used to denote all the elements in row or column of a matrix. For instance, A(2,:) returns all the elements in the second row.

Handling vectors and matrices

Some useful commands:

- length: returns the number of elements in a vector or the maximum between the dimensions of a matrix;
- size: returns a vector with the number of rows and columns of a matrix;
- sort: sorts the values in a vector or in the columns of a matrix;
- linspace: creates a vector of equally spaced points. The syntax is *linspace(a,b,npoints)* for a vector which starts at a, ends at b and contains npoints equally spaced points.
- zeros(k,m): creates a k by m matrix of zeros.
- ones(k,m): creates a k by m matrix of ones.
- eye(k): creates a k by k identity matrix.
- diag(A): returns a vector containing the diagonal elements of a matrix if A is a matrix, creates a diagonal matrix with the elements contained in A on the diagonal of A is a vector.
- tril, triu: returns the lower and upper triangular matrix obtained by setting to 0 all the elements of a square matrix above or below the diagonal respectively.

Sub-matrices can be defined by using the operator ":" or vectors of indexes. For instance, to create a sub-matrix B containing the first two rows and columns of the 3 by 3 A matrix one can do the following:

```
B=A(1:2,1:2)
```

or, alternatively

```
B=A([1 2],[1 2])
```

An important operator is the empty vector, [], which can be used to delete columns or rows of a matrix.

Exercises

1. Create a vector containing the first 20 positive integer numbers in reversed order in two different ways, by using the operator ":" and the function linspace. Then, sort them.
2. Create a 3 by 3 identity matrix. Delete its first row.
3. Create the matrix A=[4 0 0; 4 4 0; 4 4 4] using the commands tril and ones.
4. Consider the matrix A=[6 7 1; 3 5 6; -1 3 8]. Create a vector B containing the elements on the diagonal in increasing order.

Matrix and vector operations

Operations between matrices and vectors are very easy to handle in Matlab (of course, provided that the matrices or vectors are compatible, that is the operation can be performed). The following list reports the syntax used to perform basic operations between two compatible matrices A and B:

- sum: A+B;
- difference: A-B;
- matrix classical product: A*B;
- element-wise product: A.*B.
- k-th power of a square matrix: A^k. This operation computes the product of the square matrix A multiplied by itself k times.

Exercises

1. Given the two matrices A=[3 -2; 1 -5], B=[4.3 0 2; -5 7 -4]. Compute their sum, difference, product, dot product when possible. When not possible, modify B to make the two matrices compatible.

Here is a list of other useful functions:

- norm: computes the norm of a vector;
- max: returns the maximum of a vector;

- min: returns the minimum of a vector;
- sum: returns the sum of the elements of a vector;
- prod: returns the product of the elements of a vector;
- det: returns the determinant of a square matrix;
- inv: returns the inverse of a square matrix;
- rank: computes the rank of a matrix (number of linearly independent rows or columns);
- mldivide(A,B): computes the solution of a system of linear equations Ax=B for x. Alternatively, the symbol "\" can be used.

Exercises

Consider the vector v=[5.5 -2 11], matrix A=[1 0 5;-3 4 2; 6 5.5 -3].

1. Compute the L-1 norm of the vector v.
2. Compute the infinity norm of vector v.
3. Find the maximum element in matrix A using "max".
4. Consider the linear system Ax=v'. How can you determine the number of solutions? Solve the problem by using the function "inv" and the command "mldivide".

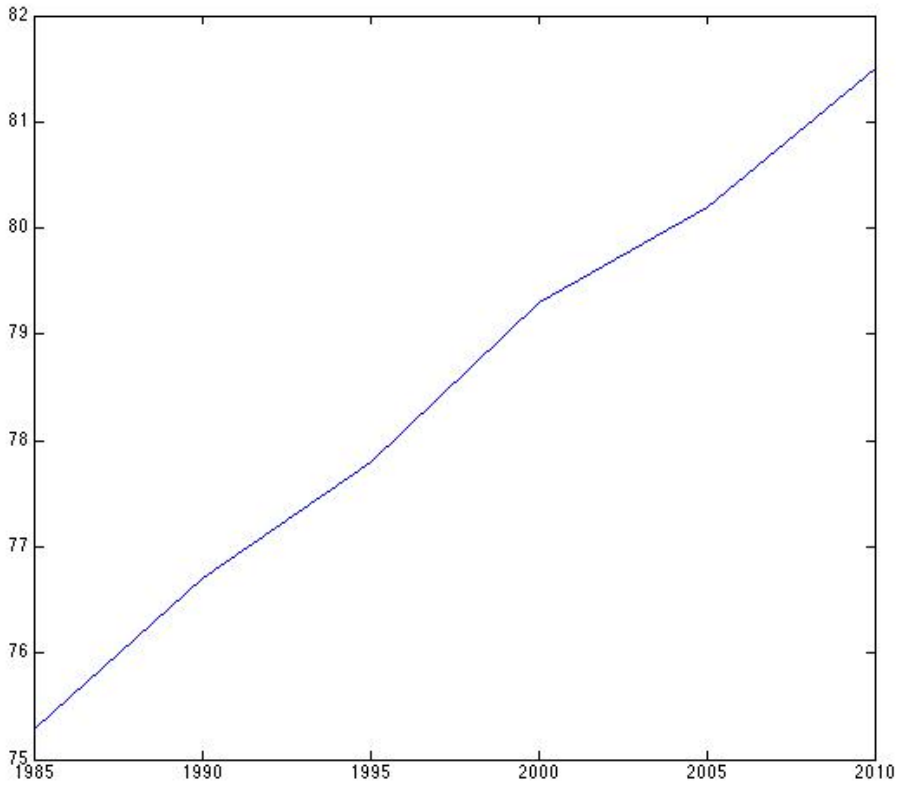
Plots and Graphics

Some built-in functions allow MATLAB's users to easily produce graphics using the information stored in the variables present in the Workspace. In this section we will briefly review the basic tools to produce 2-dimensional graphs, surface plots, histograms. The command *figure* creates a new "figure" object.

2-d plots

The most basic graphical function for a 2D line plot is *plot*. The basic syntax is *plot(X,Y)*, which produces a plot of Y against X. If only one vector is specified, plot(X) produces a plot of X against the indices of the vector, i.e. [1:1:length(X)].
For instance, let' s produce a 2-d graph of the evolution of life expectancy at birth in Italy in the last 25 years:

```
X=[1985:5:2010]
Y=[75.3 76.7 77.8 79.3 80.2 81.5]
plot(X,Y)
```



After running the "plot" command, a window with the Figure object opens. Additional commands

can be used to modify this object:

- xlabel: adds a string to label the x-axis;
- ylabel: adds a string to label the y axis;
- title: adds a title to the plot;
- legend: adds a legend to the plot.

Alternatively, the plot can be modified directly using the tools on the newly opened window in the so-called Edit mode. In order to plot more than one line on the same graph, two alternative ways can be used. First, the *plot* command can accomodate for multiple lines. The syntax is plot(X1,Y1,X2,Y2,...,Xn,Yn) to plot Y1 against X1,...in the same graph. Alternatively, after plotting one line, the *hold on* command prevents MATLAB from creating a new figure when another plot command is run. *hold off* allows the software to create a different Figure object. Several graphical options allow to modify the layout of the graph:

- line colour: 'y', 'r','b','g','w',...
- shape of points: .,x,+,*,o,...;
- style of the line: continuous (-), dashed (- -), dotted (:),...

If only one of these graphical aspects is specified, it is sufficient to insert the option after the Y variable. Otherwise, it is necessary to specify what type of option is being modified ('Color', 'Marker', 'LineStyle'). Figures can be saved to the most used graphic formats (jpg, png, eps, pdf,...) although the default extension is fig. The command *fplot* allows to plot a function. For instance,
`fplot('`x^{-1}`',[0 1])`
plots the function $y = x^{-}1$ in the interval [0,1]. The command *subplot* allows to produce several plots in sub-windows in the same figure. The syntax of this function is:
`subplot(m,n,k)`
The function produces an empty plot in the k-th subwindow of the m by n matrix of sub-windows in the figure. All of the following commands will operate on this k-th sub-window. Another useful 2d plot function is *scatter*(X,Y), that produces a scatter plot of Y against X. The syntax of additional properties is similar to the one of the plot function.

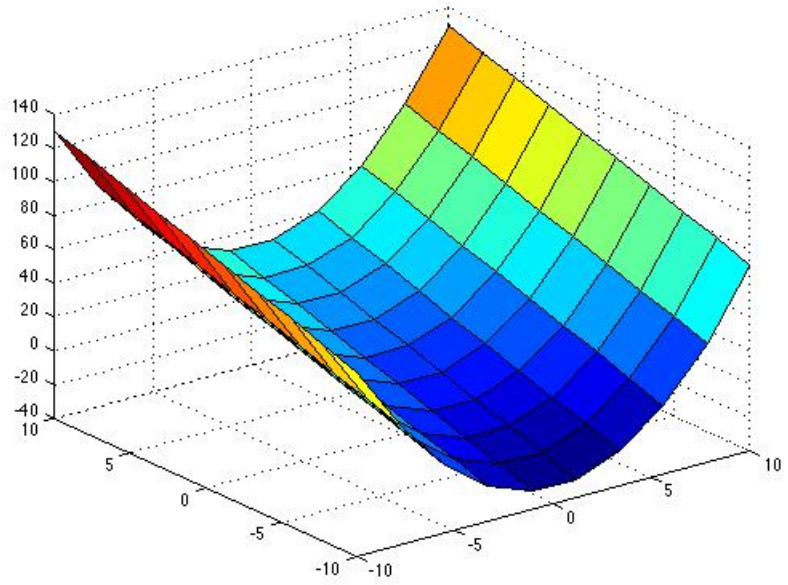
Exercise

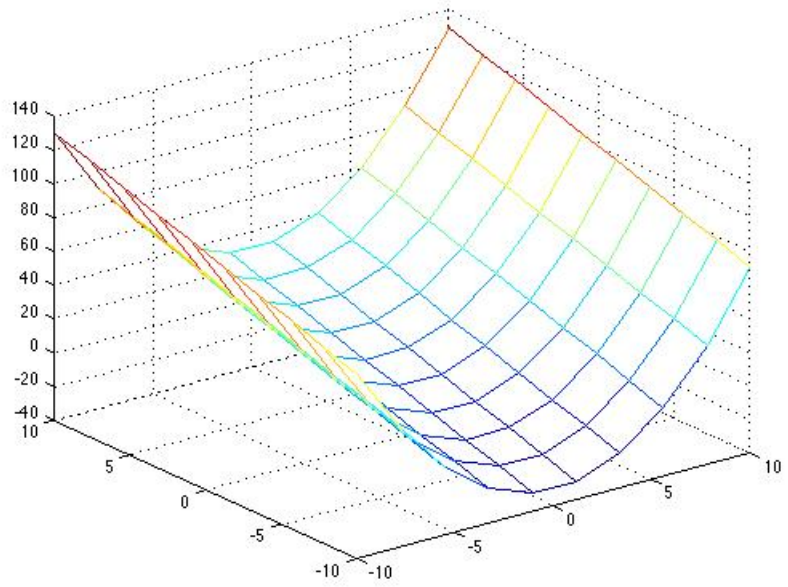
1. Produce a plot of the function y=sin(x) in the domain [0,2π] using "plot". Produce the plot by computing the function in 100 equally spaced points of the domain.
2. Plot the function above in a dashed red line with circles as markers.
3. Add the $y = cos(x)$ function on the same domain in the same plot, with a dotted line.

3-D Plots

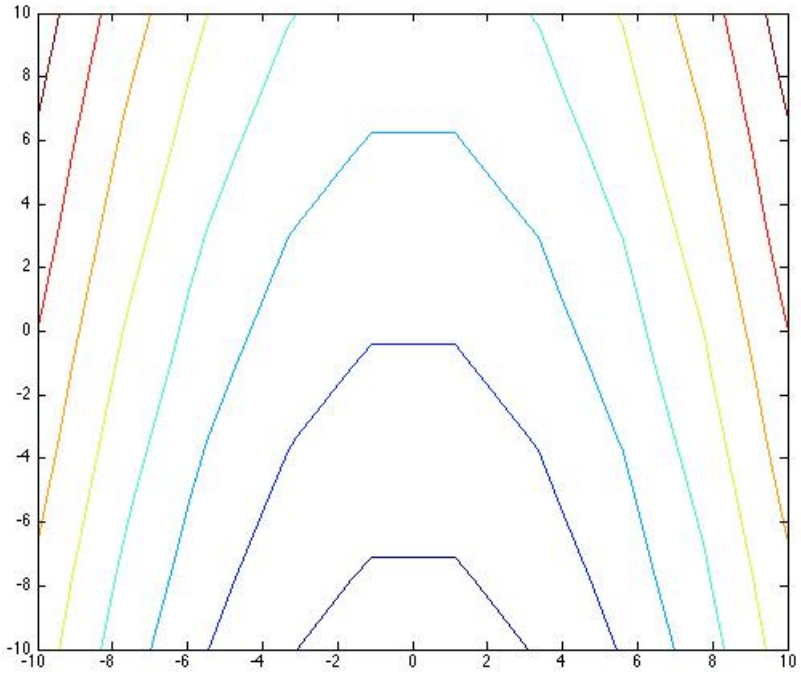
Functions of two variables can be drawn easily using the commands *surf* and *mesh*. For instance, we want to plot the bivariate function $f(x, y) = (x)^2 + 3 * y$ on the domain [-10,10] x [-10,10].

```
» x=linspace(-10,10,10);
» x=linspace(-10,10,10);
» [X,Y]=meshgrid(x,y);
» Z=X.^2+3.*Y;
» mesh(X,Y,Z)
```





The function *meshgrid* replicates the vectors x and y a sufficient number of times, creating a matrix that can be used to compute the value of the function at (x, y).
The commands *contour* and *contour3* allow to visualize the isocurves of the function (i.e. the curves passing from the points in which the function has the same value).



Other plots

Other useful types of plot are:

- histogram: it is drawn using the commands *bar* or *hist*. The former plots data in an array, while hist produces an histogram based on the number of occurrence of the same observation in the data array. A related function *histc* counts the number of values in each specified bin range;
- 3d histogram: using the command *bar3* or *bar3h* when bars are horizontal;
- pie: using the command *pie* or *pie3* with a 3d effect. Notice that MATLAB automatically normalizes the input data, so that they sum up to 1. Functions *pie* and *pie3* take displayed labels as an input. Labels can be stored in a cell array, using the following syntax: » `labels='a','b','c';`

Exercise

1. Given the following data vector v=[21, 18, 13, 12, 12, 20, 30, 30, 28, 25, 25, 22, 17, 16, 19, 19, 20, 13, 15, 16, 24, 25, 30], produce an histogram of v using the function "hist".
2. Set the number of bins to 5.
3. Produce a pie chart of the occurrences of each value in the v vector ≥ 18 (hint: use the function *histc*!). What happens to the values with zero occurrences?

Programming in MATLAB

Up to now, all the commands we wanted to use were passed through the command window. Matlab' s flexibility lies in the possibility of saving a sequence of commands to be ran sequentially into an m-file, the so-called script, and to allow the users to build his own functions, alongside the built-in ones.

Scripts

A script is a file that contains a sequence of commands. It can be ran from the command window and sequentially executes the commands it contains. The result of running a script can precisely be obtained by copying and pasting all of the content of the script on the command window. In order to create a script, it is first necessary to create a new m-file from the command bar, then write the instructions and save the file. It will be sufficient to write the name of the file in the command window to execute the script. Be careful:

- the current folder must contain the script, otherwise MATLAB will return an error!
- The workspace must contain all the variables the script is using other than the ones which are generated by the commands inside the script.
- NEVER save scripts using the names of built-in Matlab functions (the software is going to warn you).

Example.

We now want to write a script that solves a second order equation of the type $ax^2 + bx + c = 0$. We first define the vector of coefficients in the command window.

```
»coeff =[a b c]
```

Then, we open a new m-file and write the script:

```
% Solution of a second order equation
a=coeff(1);
b=coeff(2);
c=coeff(3);
delta=b^2-4*a*c;
x1=(-b+sqrt(delta))/(2*a);
x2=(-b-sqrt(delta))/(2*a);
x=[x1  x2];
disp(sprintf('The solutions to the second order equation are %f %f.', x1, x2));
```

Here, we used the functions:

- disp: which displays on the screen the argument variable;
- sprintf: it creates a string. The string can be composed calling the variables in the workspace, after specifying their format. (%f for floating point numbers, %d for integer numbers, %s for strings).

Moreover, we started the script with a comment, a line preceded by the sign % which is not executed.
Three things must be noticed when running a script:

- all variables created by the commands in the script are stored in the workspace;
- variables existing in the workspace may be overwritten;
- the execution of the script can be affected by variables already existing in the workspace.

Exercises

1. Write a script that, given a 3 by 1 vector a, computes the vector b whose elements are the sum of the corresponding element of a and the length of a and computes the L-1 norm of this new vector b. Call the script "createb.m".
2. Write a script that plots the functions $y = x^2$ and $y = x^3$ between -1 and 1 on the same plot, entitles the figure 'Graph', the x axis 'x' and the y axis 'y'.

Functions

Functions are routines, programs that accept input arguments and return outputs. In practice, they are user-defined MATLAB commands.
M-files containing functions start with the keyword *function*, which defines the output variable(s), the name of the function and the input variable(s):

```
function [output]= name (input)
```

The list of output variables is in square brackets, while the list of input variables, which have to be separated by commas, is in round brackets. It is customary to have one or more lines of comment after the function statement, which will be displayed when the help command on the function is called.
As an example, we build a function that finds the solutions of a second order equation: `function`

```
x=seceq(coeff)
% x=SECEQ(a,b,c) This function returns the vector x of solutions of the second order
equation ax^2 + bx + c = 0, where coeff=[a,b,c].
a=coeff(1);
```

```
b=coeff(2);
c=coeff(3);
delta=b^2-4*a*c;
x1=(-b-sqrt(delta))/(2*a);
x2=(-b+sqrt(delta))/(2*a);
x=[x1  x2];
```

One important difference with respect to scripts is that functions store variables in a workspace internal to the function.

Exercises

1. Compare the results obtained with the function above with those obtained with the matlab function *roots*.

2. Write a function that converts Fahrenheit degrees into Celsius degrees (recall the conversion formula $C = 5/9 * (F - 32)$).

3. Write a function that, given the annual coupon of a bond and a certain (integer) maturity, computes its present value at a fixed compound interest rate i.

Control flow and operators

As any other programming language, MATLAB has some structures that allow to control command execution. These control flows structures include for loops, while loops, and if-else-end constructions.

The if/else/elseif ...end structure

It is very useful in programming to have the possibility to create a structure in which a series of commands are executed or not based on whether a certain condition is verified. The syntax of an "if..else" structure is

```
if [condition]
command or sequence of commands


else..
command or sequence of commands


end
```

The presence of "else" allows to list a series of commands that will be executed in case the first condition is not met. For instance, let us write a script that computes the solutions of a second

degree equation if they are real and that displays a string saying that the solutions are not real otherwise:

```
delta=b^2 - 4 * a * c;
if delta>=0
x1=(-b-sqrt(delta))/(2*a);
x2=(-b+sqrt(delta))/(2*a);
else
disp('Non ci sono soluzioni reali.');
end
```

Conditions usually concern comparisons between variables and/or contain logical operators:

- ">", "<", ">=", "<=";
- the notation "==" indicates "equal to";
- the symbol "∼=" indicates "not equal to";
- & & is the logical operator "and";
- || is the logical operator "or".

Exercises

1. Write a script that, given a matrix A and an array b, solves the linear system $Ax = b$ when it has a unique solution.

2. Write a function that, given a certain input integer a, determines if it is a multiple of a certain other integer k.

3. Write a script that checks whether a certain number lies inside an interval [a,b].

While...end

The instruction "if...end" allows to execute a group of commands in case a certain condition is verified. The instruction "while...end" is instead used when it is necessary to repeat a group of instructions several times, until a certain condition is not met anymore. For instance, let us see how "while" can be used to look for the maximum among the elements of a vector x (without using the function "max"):

```
 n=length(x);
i=1;
```

```
p=-inf;
while i<=n
if x(i)>p
p=x(i);
end
i=i+1;
end
```

The series of commands between "while" and "end" is repeated until the condition "$i <= n$" is verified. When it becomes false, the iteration stops. Please be aware that, in case the condition will be verified forever, MATLAB will never stop running the sequence of commands!!! The above example allows us also to appreciate how control flow operators can be nested. Indeed, the sequence of commands inside a "if" or "while"... statement can contain other control flow operators.

Exercises

1. Use the "while..end" instruction to obtain in a vector the multiplication table of 7.
2. Use the "while...end" instruction to obtain the number of positive elements inside a vector x.
3. Given a matrix A, return the number of the first row that displays an element greater than or equal a certain threshold.

For

When a series of commands needs to be repeated a fixed number of times, or for a certain series of objects or variables, and there is no further condition that needs to be verified in order to execute the commands, it is better to use the instruction "for" rather than "while".
For instance, let us assume that, as in the exercise above, we would like to count the number of positive elements in a vector. This can be accomplished simply using the for instruction:

```
 p=0;
for i=1:length(x)
if x(i)>=0
p=p+1;
end
end
```

Notice that the counting variable i is defined once and for all after the "for" statement and does not need to be changed at every step as in the "while" instruction. For loops are very useful and

easy to use, even though they are time-consuming and should be avoided when possible when it comes to maximizing the computational efficiency.

Exercises

1. Write a script that determines the number of null and of negative elements of an array x.

2. Write a script that, given two arrays x and y, creates a new array z (whose dimension is equal to the minimum between the dimensions of the two vectors) containing the element-wise maximum between the two corresponding elements of the vectors.

3. Write a script that checks how many elements of a matrix are above a certain threshold k. (Hint: use a nested double for loop!)

4. Write a script that determines the position (row and column) of the minimum element of a matrix.

In order to stop the execution of "for" and "while" instructions, there is an instruction called break. For instance, let us assume we want to find the first positive element of a matrix for each row:

```
 [r,c]=size(A);
pos=zeros(1,r);
for i=1:r
for j=1:c
if A(i,j)>0
pos(i)=A(i,j);
break
end
end
end
```

Here, the instruction "break" stops the inner if in which it is located (but not the upper level ones!), allowing to return exactly the first positive element of the matrix. Notice that the second line of code defined the vector pos. This line of command allows to "pre-allocate" the vector. It avoids MATLAB from enlarging the vector at each iteration, which is a VERY inefficient and time consuming operation. Remember to always pre-allocate vectors and matrices which are then modified inside a loop to save computational time.

Switch

The instruction *switch* can be used to avoid using nested if instructions when there are different actions to be taken in different cases, corresponding to different values that a variable can assume.

For instance:

```
function x=evaluate(X,a)
switch a
case 1
x=mean(X);
case 2
x=var(X);
case 3
x=prctile(X,99.5);
otherwise
disp('Second argument must be 1,2 or 3');
end
end
```

Importing and exporting data

It is very useful to use data generated outside Matlab in many circumstances. In some other cases, it is convenient to export the output produced by Matlab in other formats. Matlab has a series of useful commands that can be used to either import or export data. Let us consider first how to import and export data from or to .txt files first.

MATLAB opens .txt files using the command *fopen*. For instance:

```
»ref=fopen('data.txt');
```

fopen allows to create a reference number "ref" for the opened file. This variable can now be used to refer to the file in other MATLAB commands. The input of the function *fopen* is the file path. In order to read the contents of the file, one can use the command *fscanf*:

```
» data=fscanf(ref,'%f %f',[2 inf]);
```

The above command generates an array variable called "data", in which data from the file data.txt are stored. The %f %f instruction specifies in which format the data read using fscan are converted (floating numbers in this case). The last option tells how many rows and columns must be read (inf indicates that MATLAB should read until the end of the file). Notice that filling of the output matrix is done by columns. In order to be able to write on a .txt file, it must be opened first using the 'w' (or 'wt') option. In case the file does not exist, MATLAB creates it. As an alternative, the option 'r+' allows to both read and write an already existing .txt file. The command *load* can be used to open data which are organized in matrices in a .txt file. The variable generated using the command load, unless specified, will have the same name of the file.

The other most common format to handle is the .xls format. Data in Excel format can be read using the command "xlsread". If no options are specified, the command opens the file and reads the numeric data in the first sheet:

```
filename='data.xls';
Data=xlsread(filename);
```

It is possible to specify:

- the sheet to be read;
- the range of cells to be read.

Indeed, to read from a specific file, a specific sheet and a specific range of cells we write:

```
filename='data.xls';
sheet='Italy';
range='A3:M11'; A=xlsread(filename,sheet,range);
```

Analogously, to write on an .xls file, one can use the command *xlswrite*. The options are analogous to *xlsread*. The first argument of the function is the name of the Excel file to write on, while the second argument is the content (variable) to write. Other arguments follow as for *xlsread* (sheet, range,...). Notice that *xlsread* and *xlswrite* do not accept inputs such as sheet and range if you are working on MAC OSX. Some user-defined functions (available in the File Exchange forum of MATLAB Central) overcome this problem.

Some useful tools

Descriptive statistics and statistical functions

The Statistics Toolbox of MATLAB provides the user with many basic and advanced commands to handle the problem of analyzing data. MATLAB obviously has commands which automatically compute the basic statistical quantities relative to a vector of observations:

- mean(x): mean of the elements of vector x (or of column elements of matrix x);
- var(x): variance of the elements of vector x(or of column elements of matrix x);
- std(x): standard deviation of the elements of vector x (or of column elements of matrix x);
- cov(x): covariance matrix of matrix x (columns of x represent different variables);
- median(x): median value of vector x (or of column elements of matrix x);
- mode(x): modal value of vector x (or of column elements of matrix x);
- prctile(x,k): computes the k-th percentile of vector x (or of column elements of matrix x);

Notice that it is possible to compute var or standard deviation along the row dimension of matrices instead of default column dimension just by specifying it as an input. For instance, *std(x,2)* computes the standard deviation of each row vector of matrix x.
Exercises

1. Compute the value at risk at confidence level 99.5% of a vector X=rand(1000,1);

Linear regression

As a simple example of the use of the statistical toolbox, let us now focus on linear regressions. A useful command, *LinearModel.fit* fits a linear model.

```
output=LinearModel.fit(X,y)
```

X can be either a vector or a matrix and describes the independent variables. y is the vector containing the values of the dependent variable. Notice that the output produced is not a single variable, but a more complex object which contains several variables of different types. Fitted values, residuals, loglikelihood, R^2, RMSE, are all reported in the output. The result of the fit can be used for prediction, using the command *feval*. For instance, let us assume we fitted a linear model of the mortality rates between age 15 to 50 with age as the independent variable. Now let us use the result of our fit to predict mortality rates between age 51 and 80.

```
Xpred=[51:1:80]
pred=feval(output,Xpred);
```

The plot function, used on the "output" model structure, produces a plot of the observations, the fit and confidence intervals. Analogous commands allow to deal with non-linear regression models: the function *NonLinearModel.fit* fits a non-linear model. The command works iteratively to minimize the RMSE between the observed values and the fitted ones, produced according to a user-specified functional form. As a consequence, initial values for the independent variables are needed in order to start the procedure. The following example illustrates the syntax of the function:

```
funG=@(b,X) b(1)*exp(b(2)*X);
b0=[0.1 0.1];
mdlG=NonLinearModel.fit(X,y,funG,b0);
```

While defining the functional form of the fitting function, we made use of the symbol @, which defines a so-called "function handle", which is a way of defining a function indirectly. In the command above, the value funG is obtained by executing a function with inputs b and X which computes the Gompertz law (b(1)*exp(b(2)*X) at points contained in X with parameters as contained in vector b.

Notice that in future releases "LinearModel.fit" and "NonLinearModel.fit" will be replaced by "fitlm" and "fitnlm", which are already present in the 2014 release.

Solving nonlinear equations, (un)constrained optimization problems

MATLAB provides commands that return the numerical solutions of nonlinear equations (*fzero*) or systems of nonlinear equations (*fsolve*). Initial values are indeed necessary inputs for these commands, alongside the functions defining the equation or system of equations to be solved. In many circumstances, procedures to optimize a certain function are required, and numerical methods are necessary when the objective function presents for instance non-linearities. MATLAB provides several algorithms to minimize functions, which may account for (non-linear) constraints. In particular, the Optimization and Global Optimization Toolboxes contain functions to implement highly sophisticated methods to solve Linear and Non Linear programming problems. The following list collects the most widely used functions:

- unconstrained optimization: *fminsearch*;
- constrained optimization: *fmincon*, *linprog*;

Inputs to these functions can either be specified in a "normal" way and include specifying the function to be minimized, the starting values of parameters, their boundaries, the (non)linear constraints,... or can be stored in complex objects that contain all of the informations needed by the function.
For instance, the function *fmincon*, can be called either as

```
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub)
```

or as

```
x=fmincon(problem),
```

where problem can/must have the following variables:

- objective: the objective function to be minimied;
- x0: the initial point for the algorithm;
- Aineq: matrix of linear inequality constraints;
- bineq: vector for linear inequality constraints;
- Aeq: matrix of linear equality constraints;
- beq: vector for linear equality constraints;
- lb: vector of lower bounds for parameters;

- ub: vector of upper bounds for parameters;
- nonlcon: nonlinear constraint function;
- solver: 'fmincon';
- options: contains additional options on the properties of the algorithm (tolerance, maximum number of iterations and function evaluations,...).

Interpolation and numerical integration

Interpolation methods are useful in many contexts. The function *interp1* produces a piecewise linear interpolant of the nodes specified:

```
x=[0:pi/8:2*pi];
y=sin(x);
points=linspace(0,2*pi);
interpv=interp1(x,y,points);
```

These lines of command, starting from a set of values of y, evaluated at a certain number of nodes (in x), returns the set of interpolated values computed at the points specified in the vector points itself. Some options can be used to move from the piecewise linear interpolation to other interpolating functions. *'cubic'* is the option for a cubic interpolant. Analogously, the specific command *spline* can be used to perform cubic interpolation. The function *ppval* is used to evaluate the interpolant at certain points. In some cases, it is necessary to evaluate an integral using numerical methods. Some functions allow us to perform this operation:

- *integral*: computes an integral using a global adaptive quadrature method;
- *quad*: computes the integral using the adaptive Simpson quadrature method;
- *dblquad*: computes a double integral using quadrature method;
- *trapz*: computes the integral using a trapezoidal integration method

Notice that, while *integral, quad, dblquad* accept function handles as inputs, *trapz* does not, as it is applied to a set of values of the integrand function. *trapz(x,y)* applies the trapezoidal rule, where y are the values of the integrand function at points specified in x.
These functions return the value of the integral and require as an input the function and the extremes of integration.

Exercises

1. Fit the Gompertz model to the data in the file 'data.txt' using the mortality intensities of ages 0 to 100. Import the data using the *load* command and then extract from the resulting matrix the third column. Minimize the RMSE between observed and fitted values, under the constraint that all of the intensities are positive. Use the function *fmincon*.
2. Find the roots of the non-linear equation $x^2 e^{\sqrt{2x}} - (3x + x^3)$ using the function *fzero*.
3. Compute the value of the integral $\int_0^5 x^2 e^{\sqrt{2x}} dx$ using the functions quad and trapz.

Generating random variables

Uniform random variables can be generated using the command *rand*, which generates uniformly distributed numbers in the interval [0,1]. The following command produces an $n \times m$ matrix of uniformly distributed numbers in [0,1]:

```
X=rand(n,m)
```

Notice that one can use the inverse cdf of any distribution to generate samples using uniformly distributed random numbers. Alternatively, one can make use of the *makedist* and *random* commands to sample from a distribution object defined using the former command. The command *makedist* is particularly useful to handle any sort of distributions with known parameters. Notice that the function creates an object. However, the Statistics toolbox of MATLAB provides the user also with useful functions that:

- compute the pdfs and cdfs of many distributions;
- generate random numbers from several distributions.

Here is a list of the most common distributions and associated functions:

- Normal r.v.: *normcdf*, *normpdf*, *randn*;
- Multivariate Normal:*mvrpdf*,*mvrcdf* *mvrnd*;
- Lognormal:*lognpdf*,*logncdf* *lognrnd*;
- Binomial:*binopdf*,*binocdf*, *binornd*;
- Gamma:*gampdf*,*gamcdf* *gamrnd*;
- Exponential:*exppdf*,*expcdf*, *exprnd*;
- Chi square: *chi2pdf*,*chi2cdf*, *chi2rnd*.

The syntax of pdf and cdf functions requires to insert the value at which the function is computed as first input, followed by the parameters of the distribution. Random numbers generator functions require the size of the matrix of the sample followed by the parameters of the distribution.