KICK OFF MATLAB

Marco Aiolfi Antonio Falanga

General Introduction

1. Introduction

These notes are intended as something you read in order to quickly learn how to do things in Matlab®. The organization is quite different from a book about Matlab®. The idea is to try to explain things in the natural way in which they will pop up in your mind when you will start using this software to perform economics and financial applications.
In the following there is what you really need to know about Matlab® to start writing your own codes, nothing more. If you really want to know everything about Matlab®, a good suggestion is the excellent book *Mastering MATLAB 7* by Duane Hanselman, and Bruce R. Little.

2. What is MATLAB®?

Matlab® is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation.
Matlab® features a family of add-on applications for specific solutions called *toolboxes*. Very important to most users of Matlab®, toolboxes allow you to learn and apply specialized technology. Toolboxes are comprehensive collections of Matlab® functions (M-files) that extend the Matlab® environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.
Matlab® consists of 3 main parts:

- a command window through which you communicate with the Matlab® interpreter;
- a workspace containing the variables;
- an editor and a debugger to write and check your own codes.

3. Variables

The heart of Matlab® is linear algebra. In fact, “MATLAB” was originally a contraction of “matrix laboratory”. More so than any other language, Matlab® encourages and expects you to make heavy use of arrays, vectors, and matrices.
An *array* is a collection of numbers, called elements or entries, referenced by one or more indices running over different index sets. In Matlab®, the index sets are always sequential integers starting with 1. The dimension of the array is the number of indices needed to specify an element. The size of an array is a list of the sizes of the index sets.

A matrix is a two-dimensional array with special rules for addition, multiplication, and other operations. It represents a mathematical linear transformation. The two dimensions are called the rows and the columns.
A *vector* is a matrix for which one dimension has only the index 1. A row vector has only one row and a column vector has only one column. Although an array is much more general and less mathematical than a matrix, the terms are often used interchangeably. In addition Matlab® has no formal distinction between a scalar and a 1 × 1 matrix.

4. Entering matrices and vectors

In the main window (command window) you communicate with the Matlab® interpreter. Matlab® displays a prompt (>>) indicating that it's ready to accept command from you.
Matrices can be introduced in different ways: entered by a list of elements,by generated by built-in or user supplied functions, loaded from external data and so on.

To enter the matrix **A** = $\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$ by a list of elements, type in the command line the following expression:

```
>> A=[1,2,3;4,5,6;7,8,9]
A =
1 2 3
4 5 6
7 8 9
```

The entries of the matrix are enclosed in square brackets [ ]. The values within a row are separated by commas (,) while rows are separated from each other by semicolons (;).
With the command above we initialized a new variable, **A**. Matlab® is case sensitive in the names of variables and functions, so **A** is considered a different variable from **a**. In the workspace window on the left side of the screen are listed the variables currently in use. Typing *whos* in the command line will list the variables currently in the workspace, their size and their type.
To enter a row vector **B** = $1 \quad 2 \quad 3$, type in the command window the following:

```
>> B=[1,2,3]
B =
1 2 3
```

As stated above we can generate matrices by built-in or user supplied functions. Matlab provides the following built-in function to generate matrices:

• **eye**(M,N)	identifies (MxN) matrix,
• **zeros**(M,N)	identifies (MxN) matrix of zeros,
• **ones**(M,N)	identifies (MxN) matrix of ones,
• **diag**(X)	if X is a vector produces a diagonal matrix with X down the diagonal; if X is a matrix produces a vector consisting of the diagonal of X.

- **magic**(M) produces a magic square
- **randn**(M,N) (MxN) matrix with random entries, chosen from a normal(0,1) distribution.

An important feature of working with Matlab® is the chance to combine and concatenate matrices. In fact we can create a matrix C by the two previous matrices **A** and **B**, in such a way that **C** = $\begin{matrix} A \\ B \end{matrix}$. In Matlab® it enough to write:

```
>>C= [A;B]
C =
1 2 3
4 5 6
7 8 9
1 2 3
```

In this way we created a new variable **C** adding the vector **B** to the end of the matrix **A**. Arrays can be built out of other arrays, as long as the sizes are compatible.

5. Referencing elements

It is frequently necessary to access one or more of the elements of a matrix. Each dimension is given a single index or vector of indices. The result is a block extracted from the matrix. For example, if we want to refer to the element placed in the second row and third column in the matrix **A**, we can write:

```
>>A(2,3)

ans  =
        6
```

otherwise we can call multiple elements:

```
>>A([1 3])

ans =
       1  7
```

Vectors can be given a single subscript. In fact, any matrix can be accessed via a single subscript. Multidimensional arrays (matrices) are actually stored linearly in memory, varying over the first dimension, then the second, and so on. (Think of the columns of a matrix being stacked on top of each other.) In this sense the matrix is equivalent to a vector, and a single subscript will be interpreted in this context.
An especially important construct is the **colon** operator, both for building vectors and for calling elements. For example:

```
>> 1:8

ans =
1 2 3 4 5 6 7 8

>> 0:2:10
```

```
ans =
0 2 4 6 8 10

>> 1:-.5:-1

ans =
1.0      0.5000 0 -0.5000 -1.0000
```

The format is ***first:step:last*** and the result is always a row vector.
A different kind of indexing is **logical indexing**. Logical indices usually arise from a **relational operator** (see table below). The result of applying a relational operator is a **logical array**, whose elements are 0 and 1 with interpretation as “false” and “true”. Using a logical array as an index returns those values where the index is 1 (in the single-index sense above).

==	equal to	~=	not equal to
<	less than	>	grater than
<=	less than or equal to	>=	grater than or equal to

Considering the previous matrix **A**, we get:

```
>> A>3

ans =
    0    0    0
    1    1    1
    1    1    1

>> A(A>3)

ans =
    4
    7
    5
    8
    6
    9
```

6. Multidimensional Arrays

A natural extension of two-dimensional matrices are multidimensional arrays.
For a three-dimensional array you need to use three subscripts: (*row, column, page*). As usual you can generate multidimensional arrays by entering the list of elements, using built-in functions or loading external data.

```
>>zeros(2,2,2)

ans(:,:,1) =
0 0
0 0
```

```
ans(:,:,2) =
0 0
0 0

>> a(:,:,1)=magic(2)

a =
1 3
4 2

>> a(:,:,2)=eye(2)

a(:,:,1) =
1 3
4 2
a(:,:,2) =
1 0
0 1
```

7. String and text

Text strings are entered in Matlab® enclosed in single quotes. The following line assigns the given text string to the variable **a**: *a='this is a string'*. By typing *whos* in the command line Matlab® will answer that a is a 1x16 character array.

```
>> whos
Name Size Bytes Class
a 1x16 32 char array
```

The best way to display a text string is typing *disp(varname)* or *disp('text')* :

```
>> disp(a)
this is a string
```

The command *strvcat('s1','s2',...)* vertically concatenate strings s1,s2 as in the following example:

```
>>cn=strvcat('us','uk','fr','gy','jp','cn');

cn =
us
uk
fr
gy
jp
cn
```

By indexing you can have access to a single entry of a string array:

```
>> cn(3,:)
```

```
ans =
fr
```

Now suppose you have the following string array:

```
>>varname=strvcat('rgdp','ip');
```

If you type whos you will see that **varname** is a 2x4 string array, because the first element of the string has 4 characters. In this case if you need the second string of the array and you type *varname(2,:)* you wil get 'ip ', ip followed by 2 blank spaces. To get just 'ip' you have to write cellstr(varname(2,:)) and then apply again *strvcat* to have a string. Indeed *cellstr* creates a cell array of strings and *strvcat* converts this cell array into a string

```
>>strvcat(cellstr(varname(2,:)))

ans='ip'
```

Other useful functions are *horzcat* which concatenates horizontally different strings or matrices, and *int2str* which converts a number into a string.

```
>>hor=2

fname=horzcat(strvcat(cellstr(varname(2,:))),'@',cn(3,:),'_',int2str(hor));

>> fname
fname =
ip@fr_2
```

This way of managing strings can be very useful to load and save data using nested loops.

8. Concatenation

Concatenation is the process of joining small matrices to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, [ ], is the concatenation operator. For an example, start with the 4-by-4 magic square, A, and form

```
>>B = [A A+32; A+48 A+16]
```

The result is an 8-by-8 matrix, obtained by joining the four submatrices.

```
B =

16 3 2 13 48 35 34 45
5 10 11 8 37 42 43 40
9 6 7 12 41 38 39 44
4 15 14 1 36 47 46 33
64 51 50 61 32 19 18 29
53 58 59 56 21 26 27 24
```

```
57 54 55 60 25 22 23 28
52 63 62 49 20 31 30 17
```

This matrix is halfway to being another magic square. Its elements are a rearrangement of the integers 1:64. Its column sums are the correct value for an 8-by-8 magic square.

```
>>sum(B)

ans =
260 260 260 260 260 260 260 260
```

But its row sums, sum(B')', are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square.

9. Deleting Rows and Columns

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

```
X = A;
```

Then, to delete the second column of X, use X(:,2) = []. This changes X to

```
X =
16 2 13
5 11 8
9 7 12
4 14 1
```

If you delete a single element from a matrix, the result is not a matrix anymore.
So, expressions like

```
X(1,2) = []
```

result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector. So

```
X(2:2:10) = []
```

results in

```
X =
16 9 2 7 13 12 1
```

10. Importing/Exporting data from Excel

As soon as you start Matlab®, you will have the problem of importing data from external sources. Matlab® can interact with different applications related to data providers. However a student usually has the problem of importing data from Excel. There are two kind of links that directly allow the user to easily import the data from Excel: the function *xlsread* and the toolbox **Excel-Link**.

The function xlsread import directly the data from Excel, opening the file, copying the data and closing it. It has the following format:

[N T]= xlsread('filename', sheet, 'range'),

returning numeric data in array N and text data in cell array T.
For example to import only rows 4 and 5 from worksheet 1of the excel file ‘textdata’, specify the range as 'A4:B5', use the following expression

```
>>A = xlsread('testdata.xls', 1, 'A4:B5')
```

On the other hand, the toolbox Excel-Link is an excel add-in that enables the user to import and export data, evaluate Matlab® functions directly from Excel.
After having done some computations on data, you usually want to export the results to Excel. As for importing, in this case you can use the xlswrite function:

Status=xlswrite('filename', M, sheet, 'range')

It writes matrix M to a rectangular region specified by range in worksheet sheet of the file filename. With *Status* it returns the completion status of the write operation in status. If the write completed successfully, status is equal to 1 (or true). Otherwise, status is 0 (or false). Unless you specify an output for *xlswrite*, no status is displayed in the Command Window.

TWO MOST IMPORTANT THINGS TO KNOW

- If Matlab embarks on some lengthy process that you didn't intend it to do you can put a stop to it by pressing **Ctrl + c**.
- The second important thing is that if you don't want Matlab to echo the stuff you type and the calculations it makes, then type a semicolon at the end of each command. Forgetting to use the semicolon is the most common reason for having to us the **Ctrl + c** command. The program execution time is greatly increased if all the calculations are being displayed in the Matlab window.