Introduction to Matlab, A brief tutorial

Harris Dellas

Contents

1 Introduction

Matlab is a high level programming language for technical computing.[1] Its basic data element is a matrix (the name Matlab therefore stands for Matrix laboratory). Since many models and solution procedures in economics are naturally stated as a series of matrix operations, Matlab together with programming languages of the same class is widely used in economics.

In this course we will learn basic Matlab. Matlab offers a huge amount of so called *operators* and *functions*. This manual introduces just a subset of them. However, the Matlab *Help library* offers a complete overview on all existing elements of Matlab. The software bundle also contains a so called *Help desk* that is useable as an Internet page.

[1] Comparable high level programming languages would be GAMS, GAUSS and Mathematica. Lower level languages are Basic, Fortran, C, C++ and Java.

2 Matlab basics

2.1 Preliminaries

We start the software as usual (by double-clicking the respective icon). This opens the Matlab *command window* (sometimes called Matlab *prompt*). One can either work on line by submitting commands. Or use the Matlab *editor/debugger* to write a program (set of instructions) and submit it.

As you work in the command window, Matlab remembers the *commands* you enter as well as the values of any *variable* you create. These commands and variables are said to reside in the Matlab *workspace*.

Sometimes it is useful to delete variables from the workspace. Just type

```
≫ clear
```

and all the variables defined so far are gone.

As in the example above, *Matlab code* is written in `Courier`. There is a command after every ≫. If you want to process a command, press Enter (Return key) and the result will appear in the command window. Using a semicolon (;) at the end of a command line, makes Matlab to skip the display of the results

```
≫ 2+3;
≫
```

in contrast to

```
≫ 2+3
≫ 5
```

Preceding commands can be re-displayed by pressing the *up-cursor*.

When Matlab displays numerical results, it follows several rules. By default, Matlab displays a real number with approximately *four digits* to the right of the decimal point. If the significant digits in the result are

outside this range, Matlab displays the result in *scientific notation*, similar to scientific calculators. If you prefer to see numerical results up to 15 positions behind the dot, use the command

```
» format long
```

In order to set back the default setting, type

```
» format short
```

To get the *current working directory* or folder displayed, type

```
» cd
```

`cd directory-name` sets the current directory to the one specified. `cd ..` moves to the directory above the current one. The command `dir` provides a list of all files in the directory (or folder).

To interrupt a running execution, type Ctrl+C.

2.2 Declaring variables

In Matlab each variable is a matrix (or rectangular data array). Since a matrix contains m rows and n columns, it is said to be of dimension m-by-n. An m-by-*1* or *1*-by-n matrix ist called a *vector*. A *scalar*, finally, is a *1*-by-*1* matrix.

2.2.1 Building a matrix

In order to build a matrix you have several options. One way to do it is to declare a matrix...

... as when you write it by hand

```
» A=[1 2 3
4 5 6]

A =
     1   2   3
     4   5   6
```

... by separating rows with ';'

```
» B=[4 5 6;7 8 9]
```

... or element by element

```
» C(1,1)=3;
» C(1,2)=4;
» C(1,3)=5;
» C(2,1)=6;
» C(2,2)=7;
» C(2,3)=8;
```

You can give a matrix whatever *name* you want. Numbers may be included. (Matlab is sensitive with respect to the upper or lower case.)

If a variable has previously been assigned a value, the new value *overrides* the predecessor.

Suppose you want to look at matrix C. All you have to do is type either

```
» C

C =
     3   4   5
     6   7   8
```

or

```
» disp(C)

     1   2   3
     4   5   6
```

In general, when you enter » `C`, Matlab does the following: It checks to see if C is a *variable* in the Matlab workspace; if not, it checks to see if C is a *built-in-function*; if not, it checks if an *M-file* named $C.m$ exists in the current directory; if not, it checks to see if $C.m$ exists anywhere on the Matlab search path, by searching the path in the order in which it is

specified. Finally, if Matlab can't find *C.m* anywhere on the Matlab search path, an error message will appear in the command window.

2.2.2 Declaring an empty matrix

In Matlab, there is no need to declare a variable before assigning a value to it – but it is recommended.

```
» D=[]

D =
    []
```

2.2.3 Building a matrix out of several submatrices

You can build matrices out of several submatrices. Suppose you have submatrices *A* to *D*.

```
» A=[1 2;3 4];
» B=[5 6 7;8 9 10];
» C=[3 4;5 6];
» D=[1 2 3;4 5 6];
```

In order to build $\boldsymbol{E}$, which is given by $\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix}$, you type:

```
» E=[A B;C D]

E =
     1   2   5   6   7
     3   4   8   9  10
     3   4   1   2   3
     5   6   4   5   6
```

2.2.4 Declaring a string matrix

A variable in Matlab is one of two types: *numeric* or *string*. A string matrix is like any other, except the elements in it are interpreted as ASCII numbers. To create a string variable, we enclose a string of characters in *apostrophes*. Since a string variable is in fact a row vector of numbers, it is possible to create a list of strings by creating a matrix in which each row is a separate string.
Note: As with all standard matrices, the rows must be of the same length.

```
» x=['ab';'cd']

x =
     ab
     cd

» x=['ab' 'cd']

x =
     abcd
```

2.2.5 Declaring a vector

A vector is simply a 1-by-n or m-by-1 matrix.

```
» f=[1 2 3]

f =
     1   2   3
```

Matlab always creates a vector as a row (or 1-by-n) vector. However, it is very easy to create a column (or m-by-1) vector out of it. All you have to do is adding an apostrophe (*transposition of a matrix* - compare matrix operations further below).

```
» f=[1 2 3]'

f =
     1
     2
     3
```

You may be able to create vectors of special form by indicating an *initial value*, the *step* and a *final value*.

```
» g=[1:0.5:5]

g =

 Columns 1 through 5
    1.0000   1.5000   2.0000   2.5000   3.0000
 Columns 6 through 9
    3.5000   4.0000   4.5000   5.0000
```

No step means step 1.

```
» x=[3:5]'

x =
     3
     4
     5
```

2.2.6 Declaring a scalar (or 1-by-1 matrix)

A scalar is given by a 1-by-1 matrix. E.g., consider the following single number:

```
» a=2
```

2.2.7 Managing the Matlab workspace

To see what variable names are in the Matlab workspace, issue the command `who`:

```
» who

   Your variables are:

     A      B      C
```

For more detailed information, use the command `whos`:

```
» whos

     Name  Size  Bytes  Class
     A     2x3      48  double array
     B     2x3      48  double array
     C     2x3      48  double array

     Grand total is 18 elements using 144 bytes
```

or check the workspace window.

As shown earlier, the command

```
» clear
```

deletes variables from the Matlab workspace.

2.3 Basic manipulations of matrices

2.3.1 Using partitions of matrices

One of the most basic operations is to extract some elements of a matrix (called a *partition* of matrices). Consider

```
A =
     1  2  3  4  5
     2  3  4  5  1
     3  4  5  1  2
     4  5  1  2  3
     5  1  2  3  4
```

In paragraph *2.2.1 Building a matrix* we have learnt that e.g. the element a_{23} of Matrix A is called `A(2,3)` in Matlab. Let us now isolate the central matrix

```
B =
     2  3
     3  4
     4  5
     5  1
```

In order to do this we type

```
» B=A(1:4,2:3)

B =
     2   3
     3   4
     4   5
     5   1
```

Suppose we just want to select columns 1 and 3, but take all the lines.

```
» B=A(:,[1 3])

B =
     1   3
     2   4
     3   5
     4   1
     5   2
```

where (:) means *select all*.

2.3.2 Making vectors from matrices and reverse

Suppose that we want to obtain the vectorialization of a matrix, that is, you want to obtain vector B from matrix A.

```
A =
     1   2
     3   4
     5   6

» B=A(:)

B =
     1
     3
     5
     2
     4
     6
```

Suppose we have a vector B. We want to obtain matrix A from B.

```
» A=reshape(B,3,2)

A =
     1   2
     3   4
     5   6
```

2.4 Matrix and array operations

2.4.1 Transposition of a matrix

```
C =
     3   4   5
     6   7   8


» C'

ans =
      3   6
      4   7
      5   8
```

Note: The *ans* variable is created automatically when no output argument is specified. It can be used in subsequent operations.

2.4.2 Basic matrix operations

+ -	Addition and subtraction
*	Multiplication
/	Right division
^	Exponentiation

E.g., consider the vectors a and b and a scalar i:

```
» a=1:5;
» b=[6:10]';
» i=3;
```

The vector d and the matrix E are given by:

```
» d=a+i*a

d =
     4   8   12   16   20

» E=b*a

E =
      6   12   18   24   30
      7   14   21   28   35
      8   16   24   32   40
      9   18   27   36   45
     10   20   30   40   50
```

What, if the matrices involved in the operation are of incompatible sizes? Not surprisingly, Matlab will complain with a statement such as:

```
» E=a*d
???  Error using ==> *
Matrix dimensions must agree.
```

2.4.3 Array operators

To indicate an *array* (element-by-element) operation, precede a standard operator with a period (dot). Matlab array operations include multiplication (.*), division (./) and exponentiation (.^). (Array addition and subtraction are not needed and, in fact, are not allowed, since they would simply duplicate the operations of matrix addition and subtraction.) Thus, the "dot product" of x and y is

```
» x=[1 2 3];
» y=[4 5 6];
» x.*y

ans =
      4   10   18
```

You may divide all the elements in one matrix by the corresponding elements in another, producing a matrix of the same size, as in:
`C = A ./ B`

In each case, one of the operands may be a *scalar*. This proves handy when you wish to raise all elements in a matrix to a power. For example:

```
x =
     1   2   3

» x.^2

ans =
      1   4   9
```

Similarly, you can raise each element to a different power

```
x =
     1   2   3

z =
     2   3   2

» x.^z

ans =
      1   8   9
```

2.5 Relational and logical operators

Matlab relational operators can be used to compare two arrays of the same size or to compare an array to a scalar. In the second case, the scalar is compared with all elements of the array, and the result has the same size as the array.

< >	Less (greater) than
<= >=	Less (greater) than or equal to
==	Equal to
~=	Not equal to

Let us consider the following matrix G. The command `G<=2` finds elements of G that are less or equal 2. The resulting matrix contains

elements set to logical true (1), where the relation is true, and elements set to logical (0), where it is not.

```
» G=[1 2;3 4];
» T=G<=2

T =
     1   1
     0   0
```

Suppose we want to select the elements that are greater or equal to 5 in the matrix A on page 5 and store them in vector B. We do this in two steps. Firstly, we tell Matlab to create a 5-by-5 matrix which contains 1, if an element in A is greater or equal than 5 and 0 otherwise. Secondly, we tell Matlab to collect all the elements of A in a row vector, for which we got a logical true (1) answer in the first step.

```
» B=A(A>=5)

B =
     5
     5
     5
     5
     5
```

Logical operators provide a way to combine or negate relational expressions. Matlab logical operators include:

&	AND
|	OR
∼	NOT

An example is:

```
» T=∼(G<=2)

T =
     0   0
     1   1
```

Suppose you want to find the location of the elements in a matrix that satisfy a particular condition. Consider matrix A. You want to know whether it has any elements that are greater than 2 but less or equal to 4 as well as their location i,j. The instruction

$t = A <= 4\&A > 2$ generates a matrix t of the same size as A with ones where the condition is satisfied. Alternatively, and more usefully, you can use *find* to locate the indexes of such elements:

$[i, j] = find(A <= 4\&A > 2); [ij]$

2.6 Building special matrices

2.6.1 Declaring a zero (or Null) matrix

`zeros(n)` returns an n-by-n matrix of zeros. An error message appears if n is not a scalar. `zeros(m,n)` or `zeros([m n])` returns an m-by-n matrix of zeros. Thus,

```
» D=zeros(2,3)

D =
     0   0   0
     0   0   0
```

2.6.2 Declaring a ones matrix

```
» E=ones(2,3)

E =
     1   1   1
     1   1   1
```

2.6.3 Declaring an identity matrix

```
» F=eye(3)

F =
     1   0   0
     0   1   0
     0   0   1
```

2.6.4 Declaring a random matrix

The `rand` function generates arrays of random numbers, whose elements are uniformly distributed in the interval (0,l).

```
» R=rand(5,1)

R =
    0.6154
    0.7919
    0.9218
    0.7382
    0.1763
```

The `randn` function generates arrays of random numbers, whose elements are normally distributed with mean 0 and variance 1.

```
» N=randn(2,3)

N =
    -0.4326   0.1253  -1.1465
    -1.6656   0.2877   1.1909
```

2.7 More built-in functions

The type of commands used to build special matrices are called *built-in functions*. There is a large number of built-in functions in Matlab. Apart from those mentioned above, the following are particularly useful:

2.7.1 Display text or array

`disp(X)` displays an array, without printing the array name. If X contains a text string, the string is displayed.

```
» disp('    Corn     Oats     Hay')
        Corn     Oats     Hay
```

2.7.2 Sorting a matrix

`sort(A)` sorts the elements in ascending order. If A is a matrix, `sort(X)` treats the columns of A as vectors, returning sorted columns. E.g.:

```
» A=[1 2;3 5;4 3]

A =
    1   2
    3   5
    4   3

» sort(A)

ans =
     1   2
     3   3
     4   5
```

2.7.3 Sizes of each dimension of an array

`size(A)` returns the sizes of each dimension of matrix A in a vector. `[m,n]=size(A)` returns the size of matrix A in variables m and n (recall: in Matlab, arrays are defined as m-by-n matrices). `length(A)` returns the size of the longest dimension of A. E.g.:

```
» [m,n]=size(A)

m =
    3

n =
    2

» length(A)

ans =
     3
```

2.7.4 Sum of elements of a matrix

If A is a vector, `sum(A)` returns the sum of the elements. If A is a matrix, `sum(A)` treats the columns of A as vectors, returning a row vector of the sums of each column.

```
» B=sum(A)

B =
     8   10
```

If A is a vector, `cumsum(A)` returns a vector containing the cumulative sum of the elements of A. If A is a matrix, `cumsum(A)` returns a matrix of the same size as A containing the cumulative sums for each column of A.

```
» B=cumsum(A)

B =
     1    2
     4    7
     8   10
```

2.7.5 Smallest (largest) elements of an array

If A is a matrix, `min(A)` treats the columns of A as vectors, returning a row vector containing the minimum element from each column. If A is a vector, `min(A)` returns the smallest element in A. `max(A)` returns the maximum elements.

```
» min(A)

ans =
      1   2
```

2.7.6 Descriptive statistics of matrices

`mean(X)` returns the *mean* values of the elements along the *columns* of an array. `std(X)` returns the *standard deviation* using

$$std = \left(\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2 \right)^{\frac{1}{2}}$$

where n is the number of elements in the sample.

For matrices where each row is an observation and each column a variable `cov(X)` is the *covariance matrix*. Likewise, `corrcoef(X)` returns a *matrix of correlation coefficients* calculated from an input matrix whose rows are observations and whose columns are variables.

2.7.7 Determinant of a matrix

`det(A)` returns the determinant of the square matrix A.

```
» A=[4 2;1 3];
» B=det(A)
B =
    10
```

2.7.8 Inverse of a matrix

`inv(A)` returns the inverse of the square matrix A.

```
» C=inv(A)

C =
     0.3000   -0.2000
    -0.1000    0.4000
```

In practice, it is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises, when solving the system of linear equations $Ax = b$. One way to solve this is with `x = inv(A)*b`. A better way from both an execution time and numerical accuracy standpoint is to use the matrix division operator `x = A\b`.

2.7.9 Eigenvectors and eigenvalues of a matrix

The n-by-n (quadratic) matrix A can often be decomposed into the form $\mathbf{A = PDP^{-1}}$, where D is the matrix of eigenvalues and P is the matrix of eigenvectors. Consider

```
A =
    0.9500   0.0500
    0.2500   0.7000


» [P,D]=eig(A)

P =
    0.7604   -0.1684
    0.6495    0.9857
D =
    0.9927         0
         0   0.6573
```

Probably you are just interested in the eigenvalues.

```
» eig(A)

ans =
      0.9927
      0.6573
```

2.8 Getting help

Typing `help topic` displays help about that particular topic if it exists, e.g.

```
» help det
```

shows information about the use of the determinant function. Another way is to consult the *help desk* in the *Help* menu.

3 Graphics

Matlab can produce both planar plots and 3D plots. The first important instruction is the `clf` instruction that clears the graphic screen.

3.1 Histogram

The instruction `hist(x)` draws a 10-bin histogram for the data in vector `x`. `hist(x,c)`, where `c` is a vector, draws a histogram using the bins specified in `c`. Here is an example:

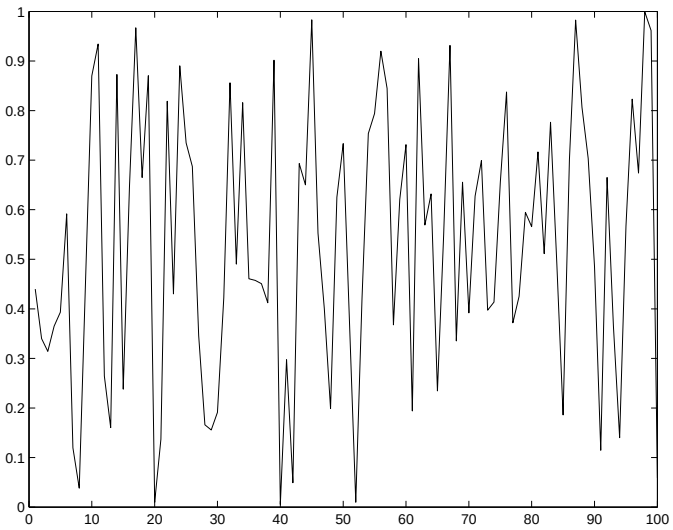
```
» c=-2.9:0.2:2.9;
» x=randn(5000,1);
» hist(x,c);
```

3.2 Plotting series versus their index

2D graphs essentially rely on the instruction `plot`. `plot(y)` plots the data in vector `y` versus its index, i.e.

```
» y=rand(100,1);
» clf;
» plot(y)
```

produces the following graph:



`plot([y x])` plots the two column vectors of the matrix `[y x]` versus their index (within the same graph).

3.3 Scatterplots (2D plots)

The general form of the `plot` instruction is

```
plot(x,y,S)
```

where `x` and `y` are vectors or matrices and `S` is a one, two or three character string specifying colour, marker symbol or line style. `plot(x,y,S)` plots `y` against `x`, if `y` and `x` are vectors. If `y` and `x` are matrices, the columns of `y` are plotted versus the columns of `x` in the same graph. If only `y` or `x` is a matrix, the vector is plotted versus the rows or columns of the matrix. Note that `plot(x,[a b],S)` and `plot(x,a,S,x,b,S)`, where `a` and `b` are vectors, are alternative ways to create exactly the same scatterplot.

The `S` string is optional and is made of the following (and more) characters:

Line style:

	Solid	Dashed	Dotted line	Dash-dot line
Symbol	-	--	:	-.

Colour:

	Yellow	Red	Green	Blue	Cyan	Magenta	White	Black
Symbol	Y	r	g	b	C	M	w	k

Marker specifier:

	Point	Circle	Asterisk	Cross
Symbol	.	O	*	x

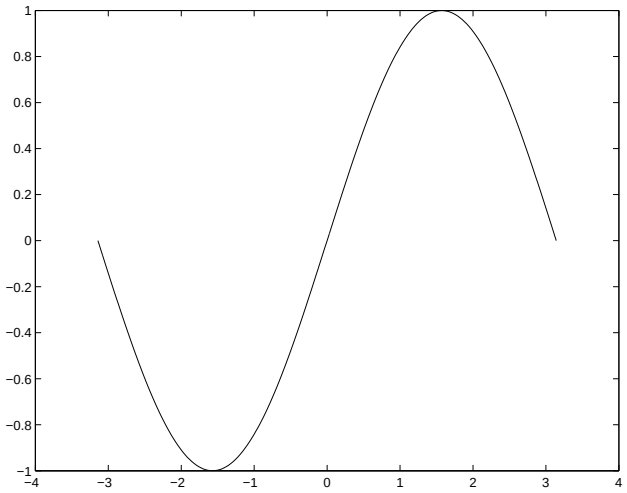
Thus,

```
» plot(X,Y,'b*')
```

plots a blue asterisk at each point of the data set.

Here is an example:

```
» x=pi*(-1:.01:1);
» y=sin(x);
» clf;
» plot(x,y,'b-');
```



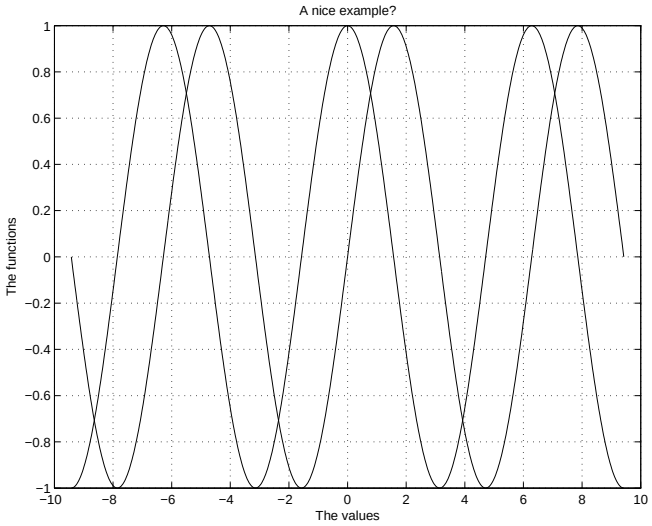
3.4 Adding titles, labels and grids

You can add titles and labels to your plot, using instructions:

```
» title('atitle') % Obvious
» xlabel('alabel') % Label on the x axis
» ylabel('alabel') % Label on the y axis
```

Using `grid`, you can even add a grid to your plot. E.g.:

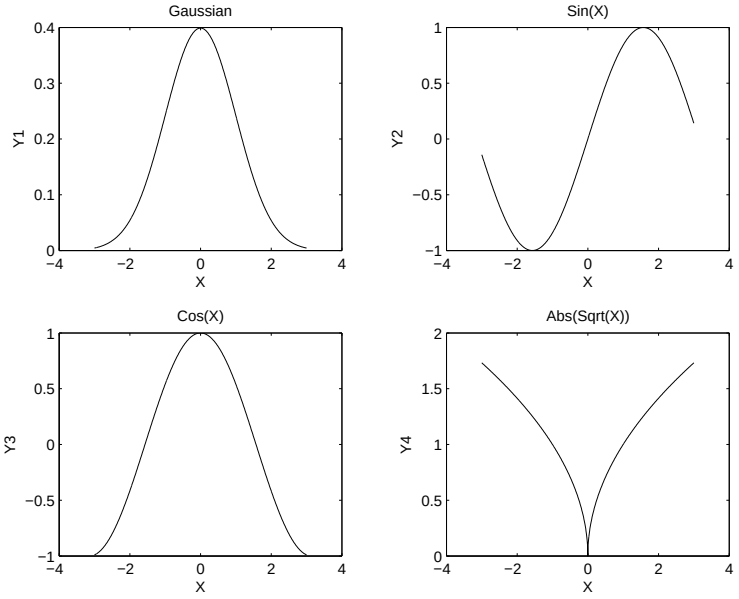
```
» x=pi*(-3:.01:3);
» y1=sin(x);
» y2=cos(x);
» clf;
» plot(x,y1,'b',x,y2,'r-');
» title('A nice example?');
» ylabel('The functions');
» xlabel('The values');
» grid;
```



3.5 Creating subplots

You can also have many graphs on the screen, using the instruction `subplot(rcn)` where `r`, `c`, and `n`, respectively, denotes the *number of rows*, *columns* and the *number of the graph*:

```
» x=[-3:.01:3];
» y1=exp(-0.5*x.^2)/sqrt(2*pi);
» y2=sin(x);
» y3=cos(x);
» y4=abs(sqrt(x));
» clf;
» subplot(221); % figure 1 out of 4
» plot(x,y1);
» ylabel('Y1');
» xlabel('X');
» title('Gaussian');
» subplot(222); % figure 2 out of 4
» plot(x,y2);
» ylabel('Y2');
» xlabel('X');
» title('Sin(X)');
» subplot(223); % figure 3 out of 4
» plot(x,y3);
» ylabel('Y3');
» xlabel('X');
» title('Cos(X)');
» subplot(224); % figure 4 out of 4
» plot(x,y4);
» ylabel('Y4');
» xlabel('X');
» title('Abs(Sqrt(X))');
```



3.6 3D plots

The `plot` command from the two-dimensional world can be extended into three dimensions with `plot3`. The format is the same as the two-dimensional `plot`, except the data are in triples rather than in pairs. The generalized format of `plot3` is `plot3`$(\mathtt{x}_1, \mathtt{y}_1, \mathtt{z}_1, \mathtt{S}_1, \mathtt{x}_2, \mathtt{y}_2, \mathtt{z}_2, \mathtt{S}_2, \ldots)$, where $\mathtt{x}_n$, $\mathtt{y}_n$, and $\mathtt{z}_n$ are vectors or matrices, and $\mathtt{S}_n$ are optional character strings specifying colour, marker symbol or line style. Here is an example:

```
» t=linspace(0,10*pi);
» plot3(sin(t),cos(t),t,'b*');
» xlabel('sin(t)'),zlabel('t');
```

Next we consider mesh plots. Matlab defines a mesh surface by the z-coordinates of points above a rectangular grid in the x-y plane. It forms a plot by joining adjacent points with straight lines. The result looks like a fishing net with the knots at the data points. Mesh plots are very useful for visualizing large matrices or for plotting functions of two variables.

The first step in generating the mesh plot of a function of two variables, $z = f(x, y)$, is to generate `X` and `Y` matrices consisting of repeated rows and columns, respectively, over some range of the variables x and y. Matlab provides the function `meshgrid` for this purpose. `[X,Y]=meshgrid(x,y)` creates a matrix `X` whose rows are copies of the vector `x`, and a matrix `Y` whose columns are copies of the vector `y`. This pair of matrices may then be used to evaluate functions of the two variables using Matlab's array mathematics features. Here is an example:

```
» x=[-7.5:.5:7.5];
» y=x;
» [X,Y]=meshgrid(x,y);
» R=sqrt(X.^2+Y.^2)+eps;
» Z=sin(R)./R;;
» mesh(X,Y,Z);
```

3.7 Save graphs

To save the resulting graph in a file just use the instruction `print`. Its general syntax is given by:

```
print -options name_of_file
```

To save the graph in a jpg-format you type:

```
print -djpeg graph.jpg
```

To save the graph in a ps-format you type:

```
print -dps graph.eps
```

4 Scripts and functions

Up to now, we were content to input *individual commands* in the Matlab command window. For more complex functions or frequently needed *sequences of instructions* this is, however, extremely unsatisfying. Matlab offers us two possibilities of processing or of automating such sequences more comfortably: *script M-files* and *function M-files*.

4.1 Script M-files

Matlab allows you to place Matlab commands in a simple *text file*, and then tell Matlab to open the file and evaluate commands exactly as it would if you had typed them in the Matlab command window. These files are called *script M-files*. The term "script" symbolizes the fact that Matlab simply reads from "script" found in the file. The term "M-file" recognizes the fact that script filenames must end with the extension .m. Further, the name of an M-file has to begin with an alphabetical letter.

To create a script M-file, choose **New** from the **File** menu and select **M-file** (alternatively you just type `edit` in the Matlab prompt and press Enter). This procedure brings up a text editor window, the *Matlab editor/debugger*. Let's type the following sequence of statements in the editor window:

```
A=[1 2];
B=[3 4];
C=A+B
```

This file can be saved as the M-file *example.m* on your disk by choosing **Save** from the **File** menu. Matlab executes the commands in *example.m* when you simply type `example` in the Matlab command window (provided there is a file *example.m* in your working directory or path[2]).

[2]Do modify the default adjustments over **File/Set Path/Path** or by pressing the button "Path Browser" in the command window toolbar. A third way would be to use DOS-commands: ≫ `dir` lists the files in the current directory; ≫ `cd` prints out the current directory; ≫ `cd..` changes to the directory above the current one

```
≫ example

C =
     4   6
```

The use of scripts is particularly handy when you are coding problems that consist of a large number of connected and complex indexing steps. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, to be used in subsequent computations. Moreover, scripts can produce graphical output using functions like plot.

4.2 Function M-files

A *function M-file* is similar to a script file in that it is a text file having a .m extension. As with script M-files, function M-files are created with a text editor. A function M-file is different from a script file in that a function communicates with the Matlab workspace only through their variables passed to it and through the output variables it created. Intermediate variables within the function do not appear in, or interact with, the Matlab workspace. Functions operate on variables within their own workspace, separate from the workspace you access at the Matlab command window. If we call a function within a code, Matlab will search the work directory for the name of the invoked M-file, minus its extension.

We have encountered examples of function M-files before. Some *built-in functions*, are part of the Matlab core. Although they are very efficient, the computational details are not readily accessible. Others, are implemented in M-files. You can see the code and even modify it if you want. Apart from the function M-files incorporated in the Matlab package, you can *write* functions *on your own*, *download* them from other users homepages on the internet or *buy* them separately in so called *toolboxes*.

4.2.1 Defining your own function M-files

We add new functions to Matlab's vocabulary by expressing them in terms of existing commands and functions. The general syntax for defining a *function* is

```
Function[output1,..]=<name of function>(input1,..);
% include here the text of your online help!
statements;
```

The first line of a function M-file defines the M-file as a function and specifies its name (its file name without the *.m* extension). The name of the M-file and the one of the function should be the same. It also defines its *input* and *output variables*.
Next, there is a sequence of comment lines with the text displayed in response to the `help` command:

```
» help <name of function>.
```

Finally the remainder of the M-file contains Matlab commands that create the output variables.
Let's consider an example: We would like to build a function, that gives us the mean and the standard deviation of a vector. (Suppose we don't know that there is a respective built-in function.)

```
function [mx,stx]=stat(x);
%
% function [mx,stx]=stat(x)
% Computes the mean and standard deviation of a
% vector x
% x:    a vector (column or row)
% mx:   mean of vector
% stx:  standard deviation of vector
%
lx=length(x);
mx=sum(x)/lx;
stx=(sum((x-mx).^2)/(lx-1))^(1/2);
```

The name of the function is `stat`. The input vector is called `x`. Output consists of two respective variables, `mx` and `stx`. After every line that starts with a `%` there is your text of the online help. What follows next are some familiar Matlab statements. First we define the scalar `lx` as the length of the input vector. Then we define the scalars `mx` and `stx`.
You now want to include and use your new function. For this purpose you change to the command window. Suppose, your input is a 1-by-100 random vector.

```
» x=randn(1,100);
» [mx,stx]=stat(x)

mx =
     0.0407

stx =
      0.8797
```

You get the variables `mx` and `stx` as an output.

A common problem in dynamic economic models involves the computation of the steady state of the model. This can be accomplished using the `fcsolve` routine that solves non linear systems of equations. For example in the standard growth model, you will have to solve the system

$$1 = \beta(\alpha A k^{\alpha-1} + 1 - \delta) \quad (1)$$

$$\delta k = A k^{\alpha} - c \quad (2)$$

Then you will create a function of 2 variables, let's call it `steady`, in a file `steady.m` :

```
function z=steady(x,param );
k=x(1);
c=x(2);

z=zeros(2,1);                                   % Initialization of z.

z(1)=1-beta*(alpha*A*k^(alpha-1)+1-\delta);   % Note that the function is
z(2)=delta*k-(A*k^alpha-c);                     % written as f(x)=0
```

and then create a script file in which you define an initial condition as well as parameter values for the fcsolve algorithm and then call fcsolve :

```
Define parameter values
alpha=0.35; beta=0.99; delta=0.025; A=1; param=[alpha beta delta A];


Give the initial guess of k and c, k0, c0
$k0=10; c0=1; x0=[k0; c0];

sol=fcsolve('steady',x0,[],param);
```

4.2.2 Downloading and using function M-files from the internet

Instead of writing a function on your own, you can download required M-files (e.g., an algorithm or an estimation procedure etc.) from the internet. You will probably find it convenient to create your own library of tools that you will use often.

4.2.3 Toolboxes

Finally, function M-files can be part of a so-called *toolbox*. A toolbox is a collection of function M-files that extend the capability of Matlab. In addition to the toolboxes installed by default,[3] toolboxes to specific topics (like *statistics* or *optimization*) can be bought separately.

[3] In your Matlab folder you will find a folder called toolbox. By default it contains three toolboxes: *local*, *matlab* and *tour*.

5 Controlling the flow

The structure of the sequences of instructions used so far was rather straightforward; commands were executed one after the other, running *from top to bottom*. Like any computer programming language and programmable calculators Matlab offers, however, features that allow you to *control the flow of command execution*. If you have used these features before, this section will be familiar to you. On the other hand, if controlling the flow is new to you, this material may seem complicated at first; if this is the case, take it slow.

Controlling the flow is extremely powerful, since it lets past computations influence future operations. With the three following sets of instruction it is possible to cope with almost all the problems we shall encounter in computation. Because they often encompass numerous Matlab commands, they frequently appear in M-files, rather than being typed directly at the Matlab command window.

5.1 The FOR-loop

The most common use of a FOR-loop arises when a set of statements is to be repeated a fixed number of times n. The general form of a FOR-loop is:

```
For variable = expression;
    statements;
end;
```

The variable `expression` is thereby a row vector of the length n that is processed element-by-element (in many cases it is the row vector (`1:n`)). `Statements` stands for a sequence of statements to the program. The columns of the expression are stored one at a time in the variable while the following statements, up to the end, are executed. The individual instructions are generally separated by semicolons, in order to avoid a constant output from section results in the command window.

A simple example of such a loop is:

```
for i=1:10;
    x(i)=i;
end;
disp(x')
```

5.2 The WHILE-loop

In contrast to the FOR-loop, with which the loop will pass through a fixed number n, WHILE-loops are executed until an abort condition is fulfilled. Note: You have to care that the abort condition is achieved by the program in finite time in each case, in order to avoid a continuous loop.

The general syntax for a WHILE loop is the following:

```
while condition;
    statements;
end;
```

So while the condition is satisfied, the statements will be executed. Consider the following example:

```
eps = 1;
while (1+eps) > 1
    eps = eps/2;
end;
eps = eps*2
```

This example shows one way of computing the special Matlab value `eps`, which is the smallest number that can be added to 1 such that the result is greater than 1 using finite precision. (We use uppercase `EPS` so that the Matlab value `eps` is not overwritten.) In this example, `EPS` starts at 1. As long as `(1+EPS)>1` is True (nonzero), the commands inside the WHILE loop are evaluated. Since `EPS` is continually divided in two, `EPS` eventually gets so small that adding `EPS` to 1 is no longer greater than 1. (Recall that this happens because a computer uses a fixed number of digits to represent numbers. Matlab uses 16 digits, so you would expect `EPS` to be near 10^{-16}.) At this point, `(1+EPS)>1` is False (zero) and the WHILE loop terminates. Finally, `EPS` is multiplied by 2, because the last division by 2 made it too small by a factor of two.

5.3 The IF-statement

A third possibility of controlling the flow in Matlab is called the IF-statement. The IF-statement executes a set of instructions *if* a condition is satisfied. The general form is

```
if condition 1;
    % commands if condition 1 is TRUE
    statements 1;
elseif condition 2;
    % commands if condition 2 is TRUE
    statement 2;
else;
    % commands if condition 1 and 2 are both FALSE
    statements 3;
end;
```

The last set of commands is executed if both condition 1 and 2 are false (i.e., not true). When you have just two conditions you skip the `elseif` condition and immediately go to `else`. An example should make the point clearer. Assume you have a demand function of the kind

$$D(P) = \begin{cases} 0 & \text{if } p \geq 2 \\ 1 - 0.5P & \text{if } 1 \leq p < 2 \\ 2P^{-2} & \text{otherwise} \end{cases}$$

The code will be:

```
P=input('enter a price :'); % displays the message
                            % '.'  on the screen and
                            % waits for an answer
if P>=2;
    D=0;                    % statement 1
elseif (P>=1)&(P<2);
    D=1-0.5*P;              % statement 2
else;                       % otherwise
    D=2*P^(-2);             % statement 3
end;
disp(D);
```

5.4 How to break a FOR or WHILE-loop

`break` terminates the execution of a for FOR- or WHILE-loop.

6 Importing and exporting data

With Matlab you can both *export* your data or calculated results and *import* external data.

6.1 The diary command

The diary command creates a log of keyboard input and the resulting output (except it does not include graphics). The log is saved as an ASCII text file in the current directory or folder. For instance let's type the following statements:

```
» diary result.out;
» disp(A);
» diary off;
```

In our directory appears a file named "result" with the extension .out. We can open it in any editor. It contains the (string or numeric) matrix A.

Note: The instruction `diary` does not overwrite an existing file. We have to delete it first with `delete <afile>` if we do *not* want to *add* information on the existing file but *overwrite* it.

6.2 Save workspace variables on disk

The **Save workspace as...** menu item in the **File** menu opens a standard file dialog box for saving all current variables. Saving variables does not delete them from the Matlab workspace. Besides, Matlab provides the command `save`.

```
» save
```

stores the workspace in Matlab binary format in the file *matlab.mat*.

```
» save data
```

saves the workspace in Matlab binary format in the file *data.mat*.

```
» save capital capital
```

saves the variable `capital` in Matlab binary format in the file *capital.mat*.

```
» save capital capital -ascii
```

saves the variable `capital` in 8-digit ASCII text format in the file *capital.txt*. ASCII-formatted files may be edited using any common text editor.

6.3 Retrieving data and results

The `load` command uses the same syntax, with the obvious difference of loading variables into the Matlab workspace.

For example assume the data file, *data.txt*, takes the form:

```
.....1.0000 –>.....2.0000 ¶
.....3.0000 –>.....4.0000 ¶
.....5.0000 –>.....6.0000 ¶
```

We make sure that *data.txt* is in the right directory and then type

```
load data.txt -ascii;
```

Our data are stored in a matrix called `data` that we will manipulate as any other matrix.

If our data is stored in an *Excel spreadsheet* then there is an even lazier way to import `data`. We highlight the data to be imported in the Excel spreadsheet, copy them, swap to the Matlab prompt, define a matrix (e.g. `data=`), open the angular brackets and then paste the data. Finally we close the brackets. We end up with a matrix called `data` containing our data.

6.4 Request user input

If we want the user to input information from the keyboard we use the statement `input`:

```
n=input('Give a number');
```

displays the message "Give a number" on the screen and wait for an answer. The result will be evaluated in the variable `n`.

References

[1] *Einführung in Matlab*. University of Munich
http://www.stat.uni-muenchen.de/∼boehme/matlab-kurs/lecture1.html

[2] Noisser, Robert (1998). *Matlab Einführung*. University of Vienna
http://www.iert.tuwien.ac.at/support/matlab_1.htm

[3] Sigmon, Kermit (1992). *Matlab Primer* 2nd edition. Department of Mathematics, University of Florida

[4] *The Student Edition of MATLAB*. Version 5, User's Guide (1997)