

MATLAB Tutorial: Part I.

Conchi Ausín Olivera
Econometrics

Ph.D. Program in Business Administration and Quantitative Methods.

September 26, 2005

Contents

1	Introduction to MATLAB.	2
1.1	Basic syntax and arrays: vectors and matrices.	2
1.1.1	Basic syntax and array exercises.	5
1.2	Relational and logical operations.	6
1.2.1	Relational and logical exercises.	7
1.3	Condition constructs: the <code>if</code> statement.	8
1.3.1	If-blocks exercises.	8
1.4	Loop constructs: <code>for</code> and <code>while</code> loops.	9
1.4.1	Loops exercises.	10
1.5	Graphics.	11
1.5.1	Planar plots	11
1.5.2	3-D mesh plots	12
1.6	Loading and saving data.	12
1.7	Executable files and subroutines: <code>*.m</code> files.	13
1.7.1	Script files.	13
1.7.2	Function files.	14
1.7.3	Programming exercises.	17

1 Introduction to MATLAB.

The name MATLAB is an abbreviation for "Matrix Laboratory". As its name implies, this software package is designed for efficient vector and matrix computations. The interface follows a language that is designed to look a lot like the notation use in linear algebra. The primary data type in MATLAB is an $N \times M$ two-dimensional array, with integer, real, or complex elements. Of course, $N \times 1$ and $1 \times M$ arrays are vectors, and a 1×1 array represents a scalar quantity. MATLAB can also produce both planar plots and 3-D mesh surface plots. The syntax is simple and straightforward, and is similar to C/C++ and Fortran. In general, it is easier to program in MATLAB than in C or Fortran, although MATLAB is usually slower. However, MATLAB is an interactive, high-level, high-performance matrix-based system for doing scientific and technical computation and visualization. MATLAB contains a wide range of basic built-in functions and also various specialized libraries (toolboxes). These notes will only refer to the Statistics Toolbox and are based on the MATLAB 6.5 version, but the contents are almost completely applicable to previous versions.

To run MATLAB, click on the MATLAB icon and the command window will be automatically opened. This is where you can type, for example, the basic commands to compute simple operations, as shown below. From now on, the symbol `>>` represents a command line.

```
>> 9*8
>> 9*(1-4/5)
>> cos(pi/2)^2
```

1.1 Basic syntax and arrays: vectors and matrices.

Almost all of basic commands in MATLAB revolve around the use of vectors. A vector is defined by placing a sequence of numbers within square braces:

```
>> v = [3 1 4]
```

This creates a row vector which has the label `v`. It has three elements, 3, 1 and 4. If you put a semi-colon (`;`) at the end of the line, the result will not be printed out. If you want to view the vector just type its label. You can view individual entries in this vector:

```
>> v(1)
```

To simplify the creation of large vectors, you can define a vector by specifying the first entry, an increment, and the last entry. For example:

```
>> v=0:2:8
```

If you want to only look at the first three entries in a vector you can use the same notation you used to create the vector:

```
>> v(1:3)
>> v(1:2:4)
```

The transposed vector can be obtained with:

```
>> v'
```

You can define directly a column vector as follows:

```
>> w=[1;4;3]
```

You can perform standard operations with vectors:

```
>> v(1:3)-v(2:4)
```

MATLAB also keeps track of the last result in a variable called `ans`. Observe the results obtained with the following operations:

```
>> v = [1 2 3]'  
>> b = [2 4 6]'  
>> v+b  
>> v-b
```

Multiplication of vectors and matrices must follow strict rules:

```
>> v*b  
??? Error using ==> *  
Inner matrix dimensions must agree.  
>> v*b'  
>> v'*b  
>> (v*b')^2
```

The matrix operations of addition and subtraction already operate entry-wise but the other matrix operations given above do not, they are *matrix* operations. It is important to observe that these other operations: `*`, `^`, `\`, `/`, can be made to operate entry-wise by preceding them by a period (`.`), for example, the following operation:

```
>> v.*b
```

allows to find $[v(1)*b(1) \ v(2)*b(2) \ v(3)*b(3)]$. Observe also:

```
>> v./b  
>> (v*b').^2
```

In MATLAB there are some elementary built-in functions as `log`, `cos`,... that can be applied to arrays.

```
>> sin(v)
```

See `help elfun` to obtain a list with the elementary math functions. See also `help ops` and `help arith`.

With MATLAB you can perform operations with high dimensional vectors, for example:

```
>> x = [0:0.1:100];  
>> length(x)  
>> y = sin(x).*x./(1+cos(x));  
>> plot(x,y)  
>> plot(x,y,'rx')  
>> help plot  
>> plot(x,y,'y',x,y,'g.')
```

The `who` and `whos` commands let you know all of the variables you have in your work space.

```
>> whos
```

To clear all variables from memory use:

```
>> clear  
>> whos
```

Be careful, it does not ask you for a second opinion and its results are final.

Defining a matrix is similar to defining a vector. To define a matrix, you can treat it like a column of row vectors:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

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

You can also treat it like a row of column vectors:

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 7 & 8 \end{bmatrix}$$

```
>> B = [ [1 2 3]' [2 4 7]' [3 5 8]']
```

The notation used by MATLAB is the standard linear algebra notation. If you want to multiply the matrix B times a vector x , where:

```
>> x = [ 2 4 6 ]'
```

you can just type:

```
>> b = B*x
```

or equivalently:

```
>> b = (x'*B)'
```

The number of columns of the thing on the left must be equal to the number of rows of the thing on the right of the multiplication symbol:

```
>> B*x'
```

Matrix product of two matrices can be calculated as simply as follows:

```
>> A*B
```

Element-wise product can be done with the `.*` operation:

```
>> A.*B
```

You can work with different parts of a matrix, just as you can with vectors:

```
>> A(1:2,2:3)
```

```
>> A(2,3)
```

```
>> A(:,2:3)
```

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

```
>> A(:)
```

MATLAB solves linear systems as $Bx = b$. The solution x of this system is obtained with:

```
>> x=inv(B)*b
```

```
>> x=B^(-1)*b
```

```
>> x=B\b
```

which is different from:

```
>> y=b'\B
```

A warning message is printed if the matrix is badly scaled or nearly singular

```
>> inv(A)
```

Some elementary functions to define matrices are given by:

```

>> I=eye(3)      % 3x3 identity matrix
>> Z=zeros(3)    % 3x3 matrix of zeros
>> u=ones(3,1)   % 3x3 vector of ones
>> R=rand(6,6)   % 6x6 matrix of random numbers from an U(0,1)
>> d=diag(R)     % main diagonal of R

```

Some operations with matrices

```

>> det(A)        % determinant of the square matrix A
>> rank(A)       % matrix rank of A
>> [f,c]=size(A) % number of rows and columns in A
>> trace(A)      % trace of A
>> [V,D]=eig(A)  % eigenvectors and eigenvalues of A (so that A*V = V*D)

```

1.1.1 Basic syntax and array exercises.

1. Let $x = [3 \ 2 \ 6 \ 8]'$ and $y = [4 \ 1 \ 3 \ 5]'$.

- Add the sum of the elements in x to y .
- Raise each element of x to the power specified by the corresponding element in y .
- Divide each element of y by the corresponding element in x .
- Multiply each element in x by the corresponding element in y , calling the result z .
- Add up the elements in z and assign the result to a variable called w .
- Compute $x'*y - w$ and interpret the result.

2. Evaluate the following expressions:

a) `round(6 / 9)` *b)* `floor(6 / 9)` *c)* `ceil(6 / 9)`

3. Create a vector x with the elements:

$$x_n = (-1)^n \frac{n+1}{2n-1}.$$

4. Plot the functions x , x^3 , e^x and e^{x^2} over the interval $0 < x < 4$.

5. Given $x = [3 \ 1 \ 5 \ 7 \ 9 \ 2 \ 6]$, explain the meaning of the following commands:

- `x(3)`
- `x(1:7)`
- `x(1:end)`
- `x(1:end-1)`
- `x(6:-2:1)`
- `x([1 6 2 1 1])`

6. Given the array $A = [\ 2 \ 4 \ 1 \ ; \ 6 \ 7 \ 2 \ ; \ 3 \ 5 \ 9]$, provide the commands needed to:

- Assign the first row of A to a vector $x1=[\ 1 \ 2 \ 3 \]$.

- (b) Assign the last 2 rows of A to an array $Y=[2 \ 3 \ 1;5 \ 3 \ 2]$.
- (c) Compute the sum over the columns of A and also over the rows of A .
7. Given the arrays $x = [1 \ 4 \ 8]$, $y = [2 \ 1 \ 5]$ y $A = [3 \ 1 \ 6 \ ; \ 5 \ 2 \ 7]$, determine which of the following statements will and will not correctly execute.
- (a) $A - [x' \ y']$
- (b) $[x \ ; \ y']$
- (c) $[x \ ; \ y]$
- (d) $A - 3$
8. Given the array $A = [2 \ 7 \ 9 \ 7 \ ; \ 3 \ 1 \ 5 \ 6 \ ; \ 8 \ 1 \ 2 \ 5]$, explain the results of the following commands:
- (a) A'
- (b) $A(:, [1 \ 4])$
- (c) $A([2 \ 3], [3 \ 1])$
- (d) $A(:)$
- (e) $[A \ A]$
- (f) $A(1:3, :)$
- (g) $[A \ ; \ A(1:2, :)]$
9. Given the array A from the previous problem, above, provide the command that will:
- (a) Assign the even-numbered columns of A to an array $B=[2 \ 3; \ 4 \ 5; \ 5 \ 4]$
- (b) Compute the square-root of each element of A .

1.2 Relational and logical operations.

As in many packages, in MATLAB, you should think of 1 as *true* and 0 as *false*. Some common logical operator are the following. See also `help ops` and `help relop`.

Relational operators:		Logical operators:	
Equal:	<code>==</code>	And:	<code>&</code>
Not equal:	<code>~=</code>	Or:	<code> </code>
Less than:	<code><</code>	True if any element of vector is $\neq 0$:	<code>any</code>
Greater than:	<code>></code>	True if all elements of vector are $\neq 0$:	<code>all</code>
Less than or equal:	<code><=</code>		
Greater than or equal:	<code>>=</code>		

1.2.1 Relational and logical exercises.

1. Given that $\mathbf{x} = [1\ 5\ 2\ 8\ 9\ 0\ 1]$ and $\mathbf{y} = [5\ 2\ 2\ 6\ 0\ 0\ 2]$, execute and explain the results of the following commands:

- (a) $\mathbf{x} > \mathbf{y}$
- (b) $\mathbf{x} == \mathbf{y}$
- (c) $\mathbf{y} \geq \mathbf{x}$
- (d) $(\mathbf{x} \sim 0) \& (\mathbf{y} \sim 0)$
- (e) $(\mathbf{x} > \mathbf{y}) | (\mathbf{x} < \mathbf{y})$
- (f) $(\mathbf{x} > 0) | (\mathbf{y} > 0)$
- (g) $(\mathbf{x} \sim 0) \& (\mathbf{y} == 0)$
- (h) $(\mathbf{x} > \mathbf{y}) \& (\mathbf{x} < \mathbf{y})$

2. The exercises here show the techniques of logical-indexing (indexing with 0-1 vectors). Given $\mathbf{x} = 1:10$ and $\mathbf{y} = [3\ 1\ 5\ 6\ 8\ 2\ 9\ 4\ 7\ 0]$, execute and interpret the results of the following commands:

- (a) $(\mathbf{x} > 3) \& (\mathbf{x} < 8)$
- (b) $\mathbf{x}(\mathbf{x} > 5)$
- (c) $\mathbf{y}(\mathbf{x} \leq 4)$
- (d) $\mathbf{x}(\mathbf{x} < 2) | (\mathbf{x} \geq 8)$
- (e) $\mathbf{y}(\mathbf{x} < 2) | (\mathbf{x} \geq 8)$
- (f) $\mathbf{x}(\mathbf{y} < 0)$

3. Given $\mathbf{x} = [3\ 15\ 9\ 12\ -1\ 0\ -12\ 9\ 6\ 1]$, provide the command(s) that will:

- (a) Set the values of \mathbf{x} that are greater or equal than zero.
- (b) Extract the values of \mathbf{x} that are greater than 10 into a vector called \mathbf{y} .
- (c) Extract the values of \mathbf{x} that are greater than the mean of \mathbf{x} .

4. Create the vector $\mathbf{x} = \text{randperm}(35)$ and then, create another vector, \mathbf{y} , evaluating the following function for each element of \mathbf{x} .

$$f(x) = \begin{cases} 2, & \text{if } x < 6, \\ x - 4, & \text{if } 6 \leq x \leq 20 \\ 36 - x & \text{if } 20 \leq x \leq 35 \end{cases}$$

1.3 Condition constructs: the if statement.

There are times when you want certain parts of your program to be executed only in limited circumstances. The way to do that is to put the code within an `if` statement. The most basic structure for an `if` statement is the following:

```
>> if <condition 1>
    <commands 1>
elseif <condition 2>
    <commands 2>
...
    else <condition k>
        <commands k>
end
```

Each set of commands is executed if its corresponding condition is verified (that is, the statements are executed if the real part of `condition` has all non-zero elements). The `else` and `elseif` parts are optional.

It is important to know that the `if` command check `condition 1`, after that `condition 2`, and so on. Then, at the moment one `condition` is verified, you will exit the if-block. For example, in the following case:

```
>> n=0; if 1>0 n=2;elseif 2>0;n=3;else;end
The final value of n is 2.
```

1.3.1 If-blocks exercises.

1. In each of the following questions, evaluate the given MATLAB code fragments for each of the cases indicated. Use MATLAB to check your answers. Test with: $n = 7, 0, -10$, $z = 1, 9, 60, 200$, and $T = 50, 15, 0$.

(a) if $n > 1$
 $m = n + 1$
else
 $m = n - 1$
end

(b) if $z < 5$
 $w = 2 * z$
elseif $z < 10$
 $w = 9 - z$
elseif $z < 100$
 $w = \text{sqrt}(z)$
else
 $w = z$
end

```

(c) if T < 30
      h = 2*T + 1
elseif T < 10
      h = T - 2
else
      h = 0
end

```

2. Write a sequence of sentences to evaluate the following functions.

(a)

$$h(t) = \begin{cases} t - 10, & \text{if } 0 < t < 100, \\ 0.45t + 900, & \text{if } t \geq 100. \end{cases}$$

Test with $h(5) = -5$ and $h(110) = 949.5$.

(b)

$$f(x) = \begin{cases} -1, & \text{if } x < 0, \\ 1, & \text{if } x \geq 100. \end{cases}$$

Compare your results to the MATLAB function `sign`.

(c)

$$g(y) = \begin{cases} 200, & \text{if } y < 10000, \\ 200 + 0.1(y - 10000), & \text{if } 10000 \leq y < 20000, \\ 1200 + 0.15(y - 20000) & \text{if } 20000 \leq y < 50000, \\ 5700 + 0.25(y - 50,000) & \text{if } 50000 \leq y. \end{cases}$$

Test with $g(5000) = 200$, $g(17000) = 900$, $g(25000) = 1950$ and $g(75000) = 11950$.

(d) Explain why the following if-block would not be a correct solution to the previous exercise.

```

if y < 10000
    t = 200
elseif 10000 < y < 20000
    t = 200 + 0.1*(y - 10000)
elseif 20000 < y < 50000
    t = 1200 + 0.15*(y - 20000)
elseif y > 50000
    t = 5700 + 0.25*(y - 50000)
end

```

1.4 Loop constructs: for and while loops.

The commands to construct loops in MATLAB are `for` and `while`.

A `for` loop is a construction of the form:

```
>> for i=1:n, <program>, end
```

The sentences indicated in `<program>` will be repeated once for each index value `i`, (`n` times). A simple example is the following:

```
>> for j=1:4,j,end
```

For example, if you want to define a vector, \mathbf{v} , whose elements are given by $v_n = 2^n$, for $n = 1, \dots, 10$, you can use:

```
>> for n=1:10; v(n) = 2^n;end
```

However, in this case, it is faster to define directly:

```
>> n=1:10; v= 2.^n
```

Another example is one in which we want to perform operations on the rows of a matrix. If you want to start at the second row of a matrix and subtract the previous row of the matrix and then repeat this operation on the following rows, a `for` loop can do this in short order:

```
>> A = [ [1 2 3]' [3 2 1]' [2 1 3]' ]
>> for j=2:3; A(j,:) = A(j,:) - A(j-1,:);end
```

A `while` loop is a construction of the form:

```
>> while <condition>, <program>, end
```

The `program` will be executed successively as long as the value of `condition` is true (that is, if `condition` is not 0). A simple example is:

```
>> a=0, while a<4, a=a+1,end
```

Another example:

```
>> n=0, while 2^n<20, n=n+1,end
```

`While` loops carry an implicit danger in that there is no guarantee in general that you will exit it.

1.4.1 Loops exercises.

- Given the vector $\mathbf{x} = [1 \ 8 \ 3 \ 9 \ 0 \ 1]$, create a short set of commands that will:
 - Add up the values of the elements (check with `sum`.)
 - Computes the running sum (check with `cumsum`.)
- Create an 3×4 array of random numbers (use `rand`). Move through the array, element by element, and set any value that is less than 0.2 to 0 and any value that is greater than (or equal to) 0.2 to 1.
- Given $\mathbf{x} = [4 \ 1 \ 6]$ and $\mathbf{y} = [6 \ 2 \ 7]$, compute the following arrays:
 - A matrix, \mathbf{A} , whose elements are $a_{ij} = x_i y_j$
 - A matrix, \mathbf{B} , whose elements are $b_{ij} = \frac{x_i}{y_j}$
 - A vector, \mathbf{c} , whose elements are $c_i = x_i y_i$
- Write a sequence of sentences that will use the random-number generator `rand` to determine the following:
 - The number of random numbers it takes to add up to 20 (or more).
 - The number of random numbers it takes before a number between 0.8 and 0.85 occurs.
 - The number of random numbers it takes before the mean of those numbers is within 0.01 of 0.5 (the mean of this random-number generator).

1.5 Graphics.

MATLAB can produce both planar plots and 3-D mesh surface plots.

1.5.1 Planar plots

The `plot` command creates linear x-y plots; if `x` and `y` are vectors of the same length, the command `plot(x,y)` opens a graphics window and draws an x-y plot of the elements of `x` versus the elements of `y`. You can, for example, draw the graph of the sine function over the interval -4 to 4 with the following commands:

```
>> x = -4:.01:4; y = sin(x); plot(x,y)
```

Give it a try. The vector `x` is a partition of the domain with meshsize 0.01 while `y` is a vector giving the values of sine at the nodes of this partition (recall that `sin` operates entrywise).

As a second example, we will draw the graph of an exponential function:

```
>> x = -1.5:.01:1.5; y = exp(-x.^2); plot(x,y)
```

Note that one must precede `^` by a period to ensure that it operates entrywise.

Plots of parametrically defined curves can also be made. Try, for example,

```
>> t=0:.001:2*pi; x=cos(3*t); y=sin(2*t); plot(x,y)
```

The command `grid` will place grid lines on the current graph.

The graphs can be given titles, axes labeled and text placed within the graph with the commands `title`, `xlabel`, `ylabel` and `gtext`, respectively. All of them take a string as an argument. For example, the command,

```
>> title('This is my graph')
```

gives the graph a title. The command `gtext('The Spot')` allows a mouse or the arrow keys to position a crosshair on the graph, at which the text will be placed when any key is pressed. Instead, you can also use the `insert` command in the figure menu to insert `xlabel`, `ylabel`, `title`, `text...` and also the icons in the figure.

By default, the axes are auto-scaled. This can be overridden by the command `axis`. If `c = [xmin,xmax,ymin,ymax]` is a 4-element vector, then `axis(c)` sets the axis scaling to the prescribed limits. The command `axis('square')` ensures that the same scale is used on both axes. For more information on `axis` see `help axis`.

Two ways to make multiple plots on a single graph are illustrated by

```
>> x=0:.01:2*pi;y1=sin(x);y2=sin(2*x);y3=sin(4*x);plot(x,y1,y2,y3)
```

and by forming a matrix `Y` containing the functional values as columns

```
>> x=0:.01:2*pi; Y=[sin(x)', sin(2*x)', sin(4*x)']; plot(x,Y)
```

Another way is with the `hold` command. The command `hold` freezes the current graphics screen so that subsequent plots are superimposed on it. Entering `hold` again releases the "hold". The commands `hold on` and `hold off` are also available.

One can override the default linetypes and pointtypes. For example,

```
>> x=0:.01:2*pi; y1=sin(x); y2=sin(2*x); y3=sin(4*x);
```

```
>> plot(x,y1,'--',x,y2,':',x,y3,'+')
```

renders a dashed line and dotted line for the first two graphs while for the third the symbol `+` is placed at each node. See `help plot` for line and mark colors.

The command `subplot` can be used to partition the screen so that up to four plots can be viewed simultaneously. See `help subplot`.

```
>> subplot(3,1,1), plot(x,y1,'--')
>> subplot(3,1,2), plot(x,y2,':')
>> subplot(3,1,3), plot(x,y3, '+')
```

1.5.2 3-D mesh plots

Three dimensional mesh surface plots are drawn with the function `mesh`. The command `mesh(z)` creates a three-dimensional perspective plot of the elements of the matrix `z`. The mesh surface is defined by the z-coordinates of the points above a rectangular grid in the x-y plane. Try `mesh(eye(10))`.

To draw the graph of a function $z = f(x, y)$ over a rectangle, one first defines vectors `xx` and `yy` which gives partitions of the sides of the rectangle. With the function `meshgrid` (mesh grid) one then creates a matrix `x`, each row of which equals `xx` and whose column length is the length of `yy`, and similarly a matrix `y`, each column of which equals `yy`. One then computes a matrix `z`, obtained by evaluating f entrywise over the matrices `x` and `y`, to which `mesh` can be applied. The following example will draw the graph of $z = \exp(-\sqrt{x} - \sqrt{y})$ over the square $[-2, 2] \times [-2, 2]$.

```
>> xx = -2:.1:2;
>> yy = xx;
>> [x,y] = meshgrid(xx,yy);
>> z = exp(-x.^2 - y.^2);
>> mesh(z)
```

Alternatively, you could replace the first three lines of the preceding example with:

```
>> [x,y] = meshgrid(-2:.1:2,-2:.1:2);
```

Others command related to 3-D graphics are `plot3` and `surf`; where `plot3` is used to draw a line in the 3-D space while `surf` is used to draw a surface in the 3-D space. See the online help for further details of these commands.

You can obtain a hardcopy of the graphics screen by using the command `print`.

```
>> print -dps dibujito
```

This command will send a high-resolution copy of the current graphics screen into a file called `dibujito.ps`. See `help print`.

1.6 Loading and saving data.

The MATLAB `load` and `save` routines can be used to read and write both ASCII and binary data. With these two commands, you can save and load all the variables (vectors, matrices,..) you have in your workspace.

```
save filename                               % saves workspace to binary file (*.mat)
save filename X Y Z                         % saves variables X Y Z to binary file
save filename X Y Z -ascii                  % saves variables X Y Z to ASCII file
save filename X Y Z -ascii -tabs -double   % saves variables X Y Z in tab delimited
                                           % 16-digit ASCII form
```

The `load` command will read any MAT-file created on any platform or any ASCII file with numeric space or tab delimited data.

```
load file                % loads file assuming file.mat exists
load file.txt -ascii    % loads file as ASCII
```

See `help load` and `help save` for further details on these topics.

Try this example:

```
>> clear;
>> x=rand(10)
>> y=rand(10)
>> save vas x y
>> clear
>> x
>> load vas
>> x
```

If you want to save your variables in ascii format, it is better to save each variable in a different file.

```
>> clear;
>> x=rand(10)
>> save x.txt x -ascii
>> clear
>> x
>> load x.txt -ascii
>> x
```

1.7 Executable files and subroutines: *.m files.

MATLAB can execute a sequence of statements stored in a file. Such files are called "M-files" because they must have an extension of ".m" for its filename. Much of your work with MATLAB will be creating and refining M-files.

M-files allows you to store a long sequence of sentences and repeat them any time you want. First, you will need to create the *.m file. The easiest editor on our system is to just use the built in MATLAB editor (M-file editor).

There are two types of M-files: script files and function files.

1.7.1 Script files.

A script file consists of a sequence of standard MATLAB statements. For example, imagine we define a sequence of sentences to compute the Body Mass Index (BMI) in a created M-file, say, BMI.m. Then, the MATLAB command `BMI` will cause the statements in the file to be executed. See Figure 1.

Variables in a script file are global and will change the value of variables of the same name in the environment of the current MATLAB session.

Script files are also often used to enter data into a large matrix. In such a file, entry errors can be easily edited out. If, for example, one enters in a diskfile `data.m`

```

C:\Docencia\MATLAB\BMI.m
File Edit View Text Debug Breakpoints Web Window Help
[Icons]
1 - clear;
2 - weight=input('How much do you weigh in kilograms?');
3 - height=input('How tall are you in meters?');
4 - BMI=weight/height^2;
5 - disp('Your body mass index is')
6 - disp(BMI)
7 - if BMI<=18.5
8 -     disp('You are underweight')
9 - elseif (18.5<BMI)&(BMI<=25)
10 -     disp('You are normal weight')
11 - elseif (25<BMI)&(BMI<30)
12 -     disp('You are overweight')
13 - else
14 -     disp('You have an obesity problem')
15 - end
16 -

```

Figure 1: Example of script file to calculate the Body Mass Index (BMI)

```

A = [
1 2 3 4
5 6 7 8
];

```

then the MATLAB statement `data` will cause the assignment given in `data.m` to be carried out.

An M-file can also reference other M-files, including referencing itself recursively.

1.7.2 Function files.

Function files provide extensibility to MATLAB. You can create new functions specific to your problem which will then have the same status as other MATLAB functions. Variables in a function file are by default local. However, you can declare a variable to be global if you wish.

We first illustrate with a simple example of a function file.

```

function y = randint(m,n)
% RANDINT Randomly generated integral matrix.
% randint(m,n) returns an m-by-n such matrix with
% entries between 0 and 9.
y = floor(10*rand(m,n));

```

A more general version of this function is given as follow:

```

function y = randint(m,n,a,b)
% RANDINT Randomly generated integral matrix.
% randint(m,n) returns an m-by-n such matrix with

```

```

% entries between 0 and 9.
% randint(m,n,a,b) returns entries between integers a and b.
if nargin < 3, a=0; b=9; end
y = floor((b-a+1)*rand(m,n))+a;

```

This should be placed in a diskfile with filename `randint.m` (corresponding to the function name). The first line declares the function name, input arguments and output arguments; without this line the file would be a script file. Then a MATLAB statement `z = randint(4,5)`, for example, will cause the numbers 4 and 5 to be passed to the variables `m` and `n` in the function file with the output result being passed out to the variable `z`. Since variables in a function file are local, their names are independent of those in the current MATLAB environment.

Note that use of `nargin` ("number of input arguments") permits one to set a default value of an omitted input variable, such as `a` and `b` in the example given above.

A function may also have multiple output arguments. For example:

```

function [mean, stdev] = stat(x)
% STAT Mean and standard deviation
% For a vector x, stat(x) returns the
% mean and standard deviation of x.
% For a matrix x, stat(x) returns two row
% vectors containing, respectively, the
% mean and standard deviation of each column.
[m n] = size(x);
if m == 1
m = n; % handle case of a row vector
end
mean = sum(x)/m;
stdev = sqrt(sum(x.^2)/m -mean.^2);

```

Once this is placed in a diskfile `stat.m`, a MATLAB command `[xm, xd] = stat(x)`, for example will assign the mean and standard deviation of the entries in the vector `x` to `xm` and `xd`, respectively. Single assignments can also be made with a function having multiple output arguments. For example, `xm = stat(x)` (no brackets needed around `xm`) will assign the mean of `x` to `xm`.

The `%` symbol indicates that the rest of the line is a comment; MATLAB will ignore the rest of the line. However, the first few comments lines, which document the M-file, are available to the on-line help facility and will be displayed if, for example, `help stat` is entered. Such documentation should always be included in a function file.

This function illustrates some of the MATLAB features that can be used to produce efficient code. Note, for example, that `x.^2` is the matrix squares of the entries of `x`, that `sum` is a vector function, that `sqrt` is a scalar function, and that the division in `sum(x)/m` is a matrix-scalar operation.

The following function, which gives the greatest common divisor of two integers via the Euclidean algorithm, illustrates the use of an error message.

```

function y = gcd(a,b)
% GCD Greatest common divisor
% gcd(a,b) is the greatest common divisor
% of the integers a and b, not both zero.

```

```

a = round(abs(a)); b = round(abs(b));
if a == 0 & b == 0
error('The gcd is not defined when both numbers are zero')
else
while b ~= 0
r = rem(a,b);
a = b; b = r;
end
end

```

Some more advanced features are illustrated by the following function. As noted earlier, some of the input arguments of a function (such as `tol` in the following example), may be made optional through use of `nargin` ("number of input arguments"). The variable `nargout` can be similarly used. Note that the fact that a relation is a number (1 when true; 0 when false) is used and that, when `while` or `if` evaluates a relation, "nonzero" means "true" and "zero" means "false". Finally, the MATLAB function `feval` permits one to have as an input variable a string naming function.

```

function [b, steps] = bisect(fun, x, tol)
% BISECT Zero of a function of one variable via bisection method.
% bisect(fun,c) returns a zero of the function. fun is a string
% containing the name of a real-valued function of a single real
% variable; ordinarily functions are defined in M-files. x is a
% starting guess. The value returned is near a point where fun
% changes sign. For example, bisect('sin',3) is pi. Note the
% quote around sin.
%
% An optional third input argument sets a tolerance for the
% relative accuracy of the result. The default is eps. An
% optional second output arguments gives a matrix containing
% a trace of the steps; the rows are of the form [c f(c)].
% Initialization
if nargin < 3, tol = eps; end
trace = (nargout == 2);
if x ~= 0, dx = x/20; else, dx = 1/20; end
a = x - dx; fa = feval(fun,a);
b = x + dx; fb = feval(fun,b);

% Find change of sign
while (fa > 0) == (fb > 0)
dx = 2.0*dx;
a = x - dx; fa = feval(fun,a);
if (fa > 0) ~= (fb > 0), break, end
b = x + dx; fb = feval(fun,b);
end
if trace, steps = [a fa; b fb]; end
% Main loop

```

```

while abs(b - a) > 2.0*tol*max(abs(b),1.0)
c = a + 0.5*(b - a); fc = feval(fun,c);
if trace, steps = [steps; [c fc]]; end
if (fb > 0) == (fc > 0)
b = c; fb = fc;
else
a = c; fa = fc;
end
end
end

```

Some of MATLAB's functions are built-in while others are distributed as M-files. The actual listing of any M-file (MATLAB's or your own) can be viewed with the MATLAB command `type functionname`. Try entering `type eig` and `type rank`.

1.7.3 Programming exercises.

The following exercises are meant to be answered by either a script or a function Mfile (you decide). The descriptions are complete but the nature of the input/output and display options is left to you.

1. The Fibonacci numbers are commuted according to the following relation,

$$F_n = F_{n-1} + F_{n-2},$$

with $F_0 = F_1 = 1$.

- (a) Compute the first 10 Fibonacci numbers.
- (b) For the first 50 Fibonacci numbers, compute the ratio,

$$\frac{F_n}{F_{n-1}},$$

It is claimed that this ratio approaches the value of the golden mean ($\frac{(1+\sqrt{5})}{2}$). What do your results show?

2. The Legendre polynomials, $P_n(x)$, are defined by the following recurrence relation,

$$(n + 1)P_{n+1}(x) - (2n + 1)P_n(x) + nP_{n-1}(x) = 0,$$

with $P_0(x) = 1$, $P_1(x) = x$ and $P_2(x) = (3x^2 - 1)/2$. Compute the next three Legendre polynomials and plot all 6 over the interval $[-1, 1]$.

3. Below, it is shown the Gauss-Legendre algorithm to approximate π found in this web page: http://www.netcom.com/~hjsmith/Pi/Gauss_L.html

- (a) Set $a = 1$, $b = 1/\text{sqrt}(2)$, $t = 1/4$ and $x = 1$.

- (b) Repeat the following commands until the difference between a and b is within some desired accuracy:

```
y = a
a = (a + b)/2
b = sqrt(b*y)
t = t - x*(y - a)^2
x = 2*x
```

- (c) From the resulting values of a , b and t , an estimate of π is,

$$\hat{\pi} = \frac{(a + b)^2}{4t}.$$

How many repeats are needed to estimate pi to an accuracy of 1e-8? 1e-12?

4. Compute and plot the path(s) of a set of random walkers which are confined by a pair of barriers at $+B$ units and $-B$ units from the origin (where the walkers all start from).

A random walk is computed by repeatedly performing the calculation,

$$x_{j+1} = x_j + s$$

where s is a number drawn from the standard normal distribution (`randn` in MATLAB). For example, a walk of N steps would be handled by the code fragment:

```
>> x(1) = 0;
>> for j = 1:N
>> x(j+1) = x(j) + randn(1,1);
>> end
```

There are three possible ways that the walls can "act":

- (a) Reflecting - In this case, when the new position is outside the walls, the walker is "bounced" back by the amount that it exceeded the barrier. That is,

- i. When $x_{j+1} > B$,

$$x_{j+1} = B - |B - x_{j+1}|.$$

- ii. When $x_{j+1} < -B$,

$$x_{j+1} = -B + |-B - x_{j+1}|.$$

If you plot the paths, you should not see any positions that are beyond $|B|$ units from the origin.

- (b) Absorbing - In this case, if a walker hits or exceeds the wall positions, it "dies" or is absorbed and the walk ends. For this case, it is of interest to determine the mean lifetime of a walker (i.e., the mean and distribution of the number of steps the "average" walker will take before being absorbed).

- (c) Partially absorbing - This case is a combination of the previous two cases. When a walker encounters a wall, "a coin is flipped" to see if the walker reflects or is absorbed. Assuming a probability p for reflection, the pseudo-code fragment that follows uses the MATLAB uniform random-number generator to make the reflect/absorb decision:

```
if rand < p
    reflect
else
    absorb
end
```

What do you do with all the walks that you generate? Compute statistics, of course. Answering questions like:

- i. What is the average position of the walkers as a function of time?
- ii. What is the standard deviation of the position of the walkers as a function of time?
- iii. Does the absorbing or reflecting character influence these summaries?
- iv. For the absorbing/partial-reflection case, a plot of the number of surviving walkers as a function of step numbers is a very interesting thing.

Is useful, informative and interesting, particularly if graphically displayed.

5. Write a function which computes the cumulative product of the elements in a vector. The cumulative product of the j^{th} element of the vector \mathbf{v} is defined by,

$$p_j = v_1 \times v_2 \times \dots \times v_j,$$

for $j = 1 : \text{length of the vector } \mathbf{v}$. Create 2 different versions of this function:

- (a) One that uses two for-loops to explicitly carry out the calculations, element by element. An "inner" loop should accumulate the product and an "outer" loop should more through the elements of the vector \mathbf{p} .
- (b) One that uses the built-in function `prod` to replace the inner loop.

In each case, you can check your results with the built-in function, `cumprod`.

6. Follow the directions for Exercise 5 but create a function that computes the cumulative sum of the elements of a vector. The elements of the cumulative sum vector are defined by,

$$s_j = v_1 + v_2 + \dots + v_j,$$

for $j = 1 : \text{length of the vector } \mathbf{v}$. The built-in functions `sum` and `cumsum` should be used instead of `prod` and `cumprod`, respectively.

7. Use the function `randint` created in Section 1.7.2. to generate random arrays of integers between \mathbf{a} and \mathbf{b} . Test this function with the following piece of code:

```
(a) >> x = randint(100000,1,10,17);
>> hist(x,10:17)
```

The resulting histogram should be nearly flat from 10 to 17. Pay particular attention to the endpoints of the distribution.

```
(b) >> x = randint(100000,1,-30,5);
>> hist(x,-30:5)
>> x = randint(100000,1,-45,-35);
>> hist(x,-45:-35)
>> x = randint(100000,1,7,-2);
>> hist(x,-2:7)
```

Consider that each of these uses is valid.

References

- [1] Matlab Official Page <http://www.mathworks.com/>
- [2] Matlab Workshop. Geology and Geophysics Department Computer Seminars. University of Utah. http://www.mines.utah.edu/gg_computer_seminar/matlab/matlab.html
- [3] Matlab Tutorial Information. Dept. of Mathematics. University of Florida. <http://www.math.ufl.edu/help/matlab-tutorial/>
- [4] Matlab Tutorial Information. Dept. of Mathematics. University of Utah. <http://www.math.utah.edu/lab/ms/matlab/matlab.html>
- [5] Exercises for Practice. Department of Chemical Engineering. Bucknell University. <http://www.facstaff.bucknell.edu/maneval/help211/exercises.html>
- [6] J.G. de Jalón, J.H. Rodríguez, A. Brazález (2001). Aprende Matlab 6.5 como si estuviera en primero. Escuela Técnica Superior de Industriales. Universidad Politécnica de Madrid. (*In Spanish*). <http://mat21.etsii.upm.es/ayudainf/aprendainf/Matlab65/matlab65pro.pdf>
- [7] Matlab Tutorial Information. Dept. of Mathematics and Statistics. University of New Hampshire. <http://www.cyclismo.org/tutorial/matlab/>