Introduction to Matlab, Version 6.0.0

Harris Dellas Kurt Schmidheiny Manuel Wälti

Winter Term 2004

Contents

1 Introduction

This short note aims at introducing you to the basics of Matlab.

Preliminaries

In what follows, `typeset` letters denote matlab code.

After typing a command you execute it by pressing the *enter* key. The output generated by your commands is stored in Matlab and can be accessed by simply typing the appropriate name. Issuing the command `clear x` clears the value of variable x from the memory and `clear` clears all values that have been created during a matlab session.

To interrupt a running execution, type Ctrl+C.

In general, when you enter $\gg$ `C`, Matlab does the following: It first 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.

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

In the appendix you can find explanations for the functions and operations that are not fully described in the text. Alternatively, if you need help, type `help` for a list of help topics. If you need help regarding a particular command, say, log, type `help log`.

When Matlab displays numerical results it displays real numbers with *four digits* to the right of the decimal point. If you want to have more

than four digits displayed issue the command `format long`. To go back to displaying four digits after the decimal point (the default setting), type `format short`.

Note also that a ";" at the end of a line tells Matlab not to display the results from the command it executes. Text following % is interpreted as a comment (plain text) and is not executed. A . . . means that the line does not end at the physical end of the line but rather spills into the next line.

2 Matrix construction and manipulation

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.

You can give a matrix whatever *name* you want. Numbers may be included in the names. 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 old one.

2.1 Building a matrix

There are several ways to build a matrix.

- One way to do it is to declare a matrix as if you wrote it by hand

```
» A=[1 2 3
4 5 6]

A =
```

This will produce the matrix

```
1  2  3
4  5  6
```

- Another way is to separate rows with ';'

```
» B=[4 5 6;7 8 9]
```

- A final way is to do it element by element

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

There are some special matrices that are extremely useful:

- The zero (or Null) matrix: `zeros(m,n)` creates a m-by-n matrix of zeros. Thus,

```
» D=zeros(2,3)
```

produces
```
D =
     0   0   0
     0   0   0
```

- The ones matrix: `ones(m,n)` creates a m-by-n matrix of ones.
```
» E=ones(2,3)

E =
     1   1   1
     1   1   1
```

- The identity matrix: `eye(n)` creates a $(n \times n)$ matrix with ones along the diagonal and zeros everywhere else.

```
» F=eye(3)

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

- You can generate a $(n \times m)$ matrix of random elements using the command `rand(n,m)`, for uniformly distributed elements, or `randn(n,m)`, for normally distributed elements. That is, `rand` will draw numbers in [0;1] while `randn` will draw numbers from a $\mathcal{N}(0,1)$ distribution.

- The empty matrix is useful for initializing a matrix :

```
A=[ ];
```

  will set `A` to be an empty matrix.

- A string matrix. A variable in Matlab can be 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 Basic manipulations

You can build matrices out of several submatrices. Suppose you have the matrices *A,B,C,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
```

Now, suppose you want to extract particular elements from a matrix. Consider the matrix :

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \\ 3 & 4 & 5 & 1 & 2 \\ 4 & 5 & 1 & 2 & 3 \\ 5 & 1 & 2 & 3 & 4 \end{pmatrix}$$

Suppose you want to isolate the central matrix :

$$B = \begin{pmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \\ 5 & 1 \end{pmatrix}$$

All we have to type in matlab is :

```
B=A(1:4,2:3)
```

`1:4` means the sequence 1 2 3 4 and `2:3` the sequence 2 3, so that `B=A(1:4,2:3)` will just select in `A` the rows 1 2 3 4 and columns 2 and 3.

Assume we just want to select columns 1 and 3, but take all the lines. We have to write :

```
B=A(:,[1 3]);
```

The `:` means select all, while `[1 3]` means select all the elements of the the 1st and 3rd column of the matrix.

Deletion Now suppose you want to delete some elements of matrix A.

```
A(:, 3)= []
```

This deletes the third column of A.

Replacement

```
A(2:3,1:2)= zeros(2)
```

replaces the specified elements with zeros. Similarly,

```
A(2,:)= 8
```

replaces the specified element with 8.

Vectorization Consider the matrix :

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

And assume that we just want to obtain its vectorization, then we just have to type :

```
B=A(:);
```

to get :

$$B = \begin{pmatrix} 1 \\ 3 \\ 5 \\ 2 \\ 4 \\ 6 \end{pmatrix}$$

If we want to obtain the A matrix from B, we just use the `reshape` command :

```
A=reshape(B,3,2)
```

that means that it will take the B vector and form a matrix (3×2).

Replication

This is how we can replicate a matrix (tile it) horizontally or vertically

```
B=repmat(A,3,2)
```

This creates the matrix

```
A   A   A
A   A   A
```

Transpose

Finally, in order to obtain the transpose of a matrix we just use the ' operator

```
B=A';
```

Here is a table of some other useful manipulations :

`rot90(A)`	rotation of A
`fliplr(A)`	flip matrix left to right
`flipud(A)`	flip matrix up and down
`diag(A)`	create or extract the diagonal of A
`diag(A,i)`	extract the i-th diagonal above the main one of A
`diag(A,-i)`	extract the i-th diagonal below the main one of A
`tril(A)`	Lower triangular part of A
`triu(A)`	Upper triangular part of A

Logical and relational operators

You can also extract (or add) elements from a matrix using logical or relational operators. Suppose that we want to select the elements that are greater or equal to 3 in the matrix and store them in a vector B :

```
B=A(A>=3);
```

if $A(i,j) >= 3$ then it will be stored in **vector** B.

Now suppose you simply want to know the location of the elements that satisfy a particular property (let as say, equal or greater than 3). The command

```
B=A>=3;
```

will reproduce the A matrix A but with elements that take the value of 1 (if the condition is satisfied) or 0 (if the condition is not satisfied. That is,

$$B = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$$

If you simply want to know the *location* of the elements that satisfy this condition, you type

```
[i,j]=find(A>=3);
```

This can be very handy when you have a very large matrix and you are searching for particular elements.

Other relational operators are

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

Logical operators can play a similar role. For instance, suppose you want to extract those elements of A that are not equal to 3 and at the same time they are greater than 4. This can be accomplished by

```
B=A(A~=3 & A>4);
```

The logical operators include

&	AND
|	OR
~	NOT

2.2.1 Basic matrix operations

Arithmetic operations on matrices are straightforward. Some attention need to be paid when doing divisions.

+ -	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, will not do it statement such as:

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

Element-by-element operations

To do *array* (element-by-element) operation rather than matrix operations 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
```

Basic Operations :

Here is a table of basic operations you can do in Matlab :

Action	Math. equivalent	Matlab	Comment
Size	A is $(r \times c)$	`[r,c]=size(A)`	
Transposition	A'	`A'`	
Addition	$A + B$	`A+B`	$dim(A) = dim(B)$
Product	$A\,B$	`A*B`	Compatility
Product element by element	$A_{ij}B_{ij}$	`A.*B`	$dim(A) = dim(B)$
Division 1	X solution of $A\,X = B$	`A\ B`	Compatility
Division 2	X solution of $X\,A = B$	`A\ B`	Compatility
Division element by element	A_{ij}/B_{ij}	`A./B`	$dim(A) = dim(B)$
Power of a matrix	A^n	`A^ n`	A square
Power element by element	A^n_{ij}	`A.^ n`	
Trace	$tr(A)$	`trace(A)`	A square
Determinant	$det(A)$	`det(A)`	A square
Kronecker product of 2 matrices	$A \otimes B$	`kron(A,B)`	
Inverse	A^{-1}	`inv(A)`	A square
Rank	rank(A)	`rank(A)`	
Null space	$(A\,V = 0)$	`V=null(A)`	
Eigenvalue decomposition	$A = D\,P^{-1}$	`[P,D]=eig(A)`	A square
Sum columnwize	$\sum_{i=1}^{rows(A)} A_{i.}$	`sum(A)`	
Product columnwize	$\Pi_{i=1}^{rows(A)} A_{i.}$	`prod(A)`	

2.3 Some useful functions

2.3.1 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
```

2.3.2 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
```

2.3.3 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. `[m,n]=size(A) length(A)` returns the size of the longest dimension of A and `size(A,i)` gives the size of the i-th dimension (1 for rows and 2 for columns). E.g.:

```
» [m,n]=size(A)

m =
    3

n =
    2

» length(A)

ans =
     3
```

2.3.4 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
```

2.3.5 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
```

2.3.6 Descriptive statistics of matrices

`mean(X,i), i=1,2` returns the *mean* values of the elements along the *i-th* dimension of the matrix (1 = columns, 2 = rows). `std(X)` returns the *standard deviation*. 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.

2.3.7 Determinant of a matrix

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

```
» A=[4 2;1 3];
» B=det(A)
```

```
B =
    10
```

2.3.8 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`.

2.3.9 Eigenvectors and eigenvalues of a matrix

The n-by-n (quadratic) matrix A can often be decomposed into the form $\mathbf{A} = \mathbf{PDP^{-1}}$, 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
```

If you are just interested in the eigenvalues,

» `eig(A)` will do.

3 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.

3.1 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;
```

A simple example of such a loop is: `for i=1:10;`

```
    x(i)=i;
end;
disp(x')
```

Some other examples

Example 2 : Assume you want to get the generate the matrix A such that :

$$A_{ij} = i^2 - j^3 + 1, \text{ for } i = 1, \ldots, 20 \text{ and } j = 1, \ldots, 10$$

The code will be :

```
for i=1:1:20;           % i runs from 1 to 20 with step 1 : for i=init:step:final
  for j=1:1:10;         % j runs from 1 to 10 with step 1 : for j=init:step:final
    A(i,j)=i^2-j^3+1;
  end;
end;
```

Example 3 : Assume you want to compute A_{ij}^b, for b running from 0 to 1 with a step 0.1, and display the result at each iteration. The code will be :

```
A=[1 2;2 1];
for b=0:0.1:1;          % i runs from 0 to 1 with step 0.1
  C=A.^b;
  disp(C)               % displays C
end;
```

3.2 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 a condition is fulfilled.

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:

Example 1 : Assume you want to find the fixed point of a dynamic relation that admits no analytical solution :

$$x_{t+1} = 1 + x_t^{0.2}$$

All you have to do is to iterate on the relation and stop when the absolute difference between two consecutive iterations is less than a tolerance criterion. The code will be :

```
tol=1e-6;               % tolerance criterion
crit=1;                 % initialization of criterion
x0=1;                   % initial value for x
while crit>tol;
  x=1+x0^0.2;           % The dynamic relation
  crit=abs(x-x0);       % Absolute difference between 2 consecutive iterations
  x0=x;                 % the new x(t) is the one found.
end;
```

3.3 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);
```

4 Input–Output tools

Input – output instructions are important because they allow you to display, save and use your results in other codes.

4.1 Input – output of texts

The first command to know is the display command : `disp`. It allows you to display a message, a value or whatever you want provided it's a text. Thus :

```
disp('This is a nice message')
```

will display "`This is a nice message`" on the screen. Assume A is a matrix of the form :

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

then :

```
disp(A)
```

will display :

```
1 2 3 4
```

on the screen.

If now you want to store your results in a file, all you have to do is to open a file and display your results as if you wanted to see them printed on the screen. This is done with `diary name_of_file`. Do not forget to close your file with `diary off`. For example :

```
diary result.out disp('Hello world') diary off
```

will create a file named `result.out` containing the string "`hello world`".

Be careful instruction `diary` does not overwrite an existing file. You have to delete it first with `delete name_of_file`, if you do not want to add information on the existing file but overwrite it.

If you want the user to input information from the keyboard, use instruction `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`.

If now you want to load data, all you have to do is to create an ASCII file containing the data. For example assume your data file, `dat.txt`, takes the form :

1 2
3 4
5 6

Just type :

```
load dat.txt -ascii;
```

And your data are stored in a matrix called `dat`, that you will manipulate as any other matrix.

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

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.

5 Graphics

Matlab can produce both 2D and 3D plots.

5.1 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:

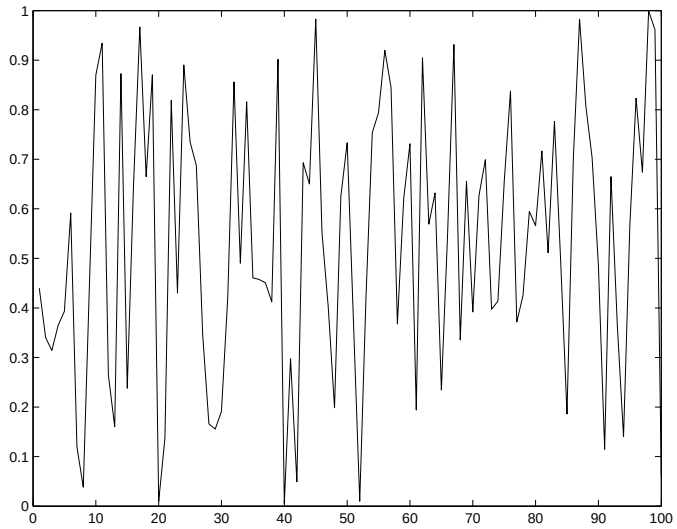
```
» c=-2.9:0.2:2.9;
» x=randn(5000,1);
» hist(x,c);
```

5.2 Plots

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).

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

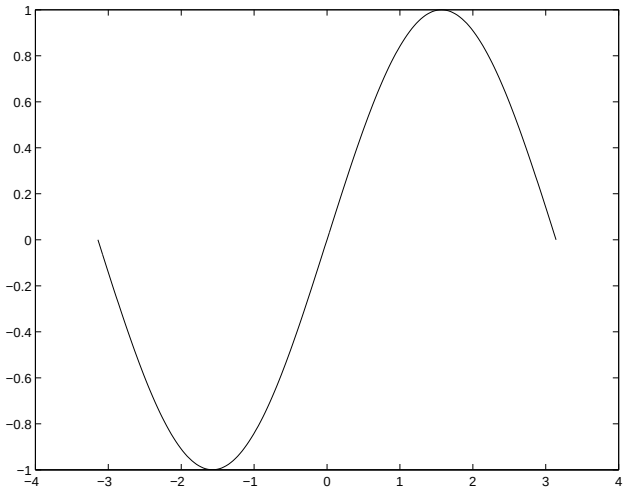
Thus,

```
» 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-');
```

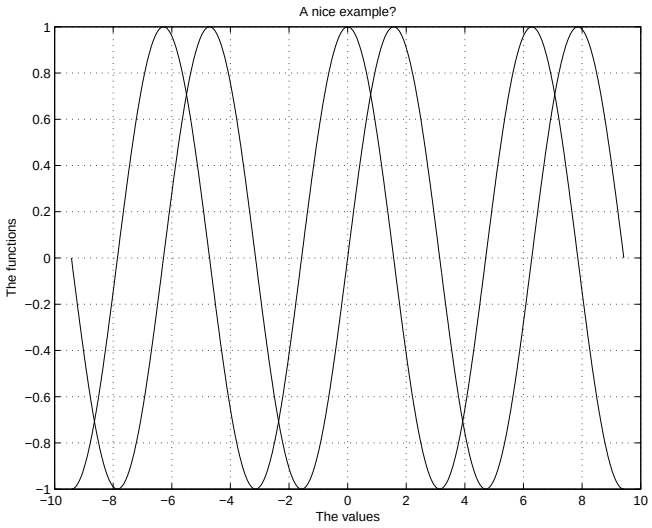


Adding titles, labels and grids You can add titles and labels to your plot, using instructions:

```
» 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.:

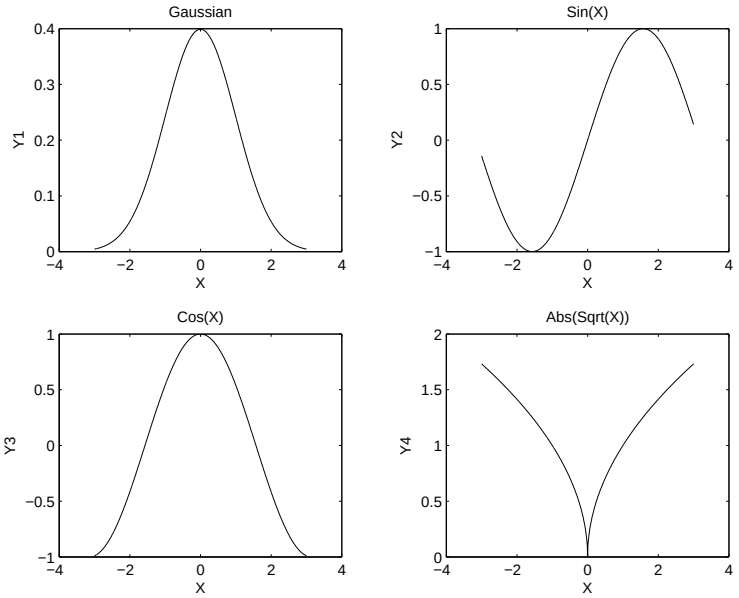
```
» x=pi*(-3:.01:3);
» y1=sin(x);
» y2=cos(x);
» clf;
» plot(x,y1,'b',x,y2,'r-');
» title('A nice example?');
» ylabel('The functions');
» xlabel('The values');
» grid;
```



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))');
```



5.3 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);
```

Saving 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
```

6 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*.

6.1 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[1]). ≫ `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.

6.2 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 in paragraph 2, already. Some *built-in functions*, like ≫ `ones`, are part of the Matlab core. Others, like ≫ `std`, are implemented in M-files. You can see the code and even modify it if you want[2]. 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*.

6.2.1 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
```

[1] 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

[2] In order to check whether a built-in function is an M-file or not, you can use your Windows Explorer **Tools/Find/Files or Folders** and look for the respective name of the function plus its extension, e.g. std.m. If it is indeed an M-file you can open it.

```
% 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 more sophisticated example

An often encountered problem in economics is the computation of the steady state of a model. This can be accomplished using the `fsolve` routine that solves non linear systems. For example in the standard growth case you will have to solve the system[3] :

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

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

[3]This can be solved by hand, but it is a simple and appealing example.

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

```
function z=steady(x);

alpha=0.35; beta=0.99; delta=0.025; A=1;

k=x(1); c=x(2);

z=zeros(2,1);                                  % Initialization of z.
                                               % Not necessary but recommended
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 f(x)=0
```

and then create another file in which you define an initial condition for the fsolve algorithm and call fsolve :

```
k0=10; c0=1; x0=[k0;c0]; sol=fsolve('steady',x0);
```

6.2.2 Downloading and using function M-files from the internet

The internet contains a huge number of Matlab M-files. For instance, visit http://www.helsinki.fi/WebEc/framec.html

6.2.3 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 that accompany basic Matlab, many other specific toolboxes (such as *statistics* or *optimization*) can be bought separately.

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)