CONTENTS

CHAPTER 1

INTRODUCTION TO MATLAB

During this course you will learn how to use MatLab, to design, and to perform mathematical computations. You will also get acquainted with basic programming. If you learn to use this program well, you will find it very useful in future, since many technical or mathematical problems can be solved using MatLab. This text includes all material (with some additional information) that you need to know, however, many things are treated briefly. It is important that you spend enough time to learn the MatLab basics.

1.1 MatLab Basics

1.1.1 Command-Line

MatLab is an interactive system; commands followed by Enter are executed immediately. The results are, if desired, displayed on screen. However, execution of a command will be possible if the command is typed according to the rules. Table 1.1 shows a list of commands used to solve indicated mathematical equations (a,b, x and y are numbers).

Below you find basic information to help you starting with MatLab:

- Commands in MatLab are executed by pressing Enter or Return. The output will be displayed on screen immediately. Try the following:

  $>> 3 + 7.5$
  $>> 18/4$

Mathematical notation	MatLab command
$a + b$	a+b
$a - b$	a-b
$a * b$	a*b
$\frac{a}{b}$	a/b
x^b	x ^ b
$\sqrt{x}$	sqrt(x) or x ^ 0.5
$\|x\|$	abs(x)
π	pi
$4 \cdot 10^3$	4e3 or 4*10 ^ 3
i	i or j
$3 - 4i$	3-4*i or 3-4*j
e, e^x	exp(1), exp(x)
$\ln x$, $\log x$	log(x), log10(x)
$\sin x, \arctan x, \ldots$	sin(x), atan(x), ...

Table 1.1 Translation of mathematical notation to MatLab commands.

>> 3 * 7
Note that spaces are not important in MatLab.

- The result of the last performed computation is ascribed to the variable ans, which is an example of a MatLab built-in variable. It can be used in the next command. For instance:
>> 14/4
ans = 3.5000
>> ans ^ (-6)
ans = $5.4399e-04$
$5.4399e-04$ is a computer notation of $5.4399 * 10^{-4}$. Note that ans is always overwritten by the last command.

- You can also define your own variables. Look how the information is stored in the variables a and b:
>> a = 14/4
a = 3.5000
>> b = a ^ (-6)
b =$5.4399e-04$

- When the command is followed by a semicolon ';', the output is suppressed. Check the difference between the following expressions:
>> 3 + 7.5
>> 3 + 7.5;

- It is possible to execute more than one command at the same time; the separate commands should then be divided by commas (to display the output) or by semicolons (to suppress the output display), e.g.:
>> sin(pi/4), cos(pi); sin(0)

ans = 0.7071
ans = 0
Note that the value of cos(pi) is not printed.

- By default, MatLab displays only 5 digits. The command format long increases this number to 15, format short reduces it to 5 again. For instance:
  >> 312/56
  ans = 5.5714
  >> format long
  >> 312/56
  ans = 5.57142857142857

- MatLab is case sensitive, for example, a is written as a in MatLab; A will result then in an error.

- All text after a percent sign % until the end of a line is treated as a comment. Enter e.g. the following:
  >> sin(3.14159) % this is an approximation of sin(pi)
  You will notice that some examples in this text are followed by comments. They are meant for you and you should skip them while typing those examples.

- Previous commands can be fetched back with the ↑ key. The command can also be changed, the ← and → keys may be used to move around in a line and edit it. In case of a long line, Ctrl-a and Ctrl-e might be useful; they allow to move the cursor at the beginning or the end of the line, respectively.

- To recall the most recent command starting from e.g. c, type c at the prompt followed by the ↑ key. Similarly, cos followed by the ↑ key will find the last command starting from cos.

1.1.2 Variables

MatLab variables can be created by an assignment. There is also a number of built-in variables, e.g. pi, eps or i, summarized in Table 1.2. In addition to creating variables by

Variable name	Value/meaning
ans	the default variable name used for storing the last result
pi	$\pi = 3.14159\ldots$
eps	the smallest positive number that added to 1, creates a result larger than 1
inf	representation for positive infinity, e.g. 1/0
nan or NaN	representation for not-a-number, e.g. 0/0
i or j	i = j = $\sqrt{-1}$
nargin/nargout	number of function input/output arguments used
realmin/realmax	the smallest/largest usable positive real number

Table 1.2 Built-in variables in MatLab.

assigning values to them, another possibility is to copy one variable, e.g. b into another, e.g. a. In this way, the variable a is automatically created (if a already existed, its previous

value is lost):
```
>> b = 10.5;
>> a = b;
```
A variable can be also created as a result of the evaluated expression:
```
>> a = 10.5; c = a^2 + sin(pi*a)/4;
```
or by loading data from text or '*.mat' files. If min is the name of a function (see help min), then a defined, e.g. as:
```
>> b = 5; c = 7;
>> a = min (b,c);        % create a as the minimum of b and c
```
will call that function, with the values b and c as parameters. The result of this function (its return value) will be written (assigned) into a. So, variables can be created as results of the execution of built-in or user-defined functions. Important: do not use variable names which are defined as function names (for instance *mean* or *error*)! If you are going to use a suspicious variable name, use help <*name*> to find out if the function already exists.

1.1.3 Matrix and Operations

The basic element of MatLab is a matrix (or an array). Special cases are:

- a 1×1-matrix: a scalar or a single number;
- a matrix existing only of one row or one column: a vector.

Note that MatLab may behave differently depending on the input, whether it is a number, a vector or a 2D matrix.

1.1.3.1 Vectors Row vectors are lists of numbers separated either by commas or by spaces. They are examples of simple arrays. First element has index 1. The number of entries is known as the length of the vector (the command length exists as well). Their entities are referred to as elements or components. The entries must be enclosed in [ ]:
```
>> v = [-1 sin(3) 7]
v = -1.0000 0.1411 7.0000
>> length(v)
ans = 3
```
A number of operations can be done on vectors. A vector can be multiplied by a scalar, or added/subtracted to/from another vector with the same length, or a number can be added/subtracted to/from a vector. All these operations are carried out element-by-element. Vectors can be also built from the already existing ones.
```
>> v = [-1 2 7]; w = [2 3 4];
>> z = v + w          % an element-by-element sum
z = 1 5 11
>> vv = v + 2         % add 2 to all elements of vector v
vv = 1 4 9
>> t = [2*v, -w]
ans = -2 4 14 -2 -3 -4
```
Also, a particular value can be changed or displayed:
```
>> v(2) = -1          % change the 2nd element of v
v = -1 -1 7
>> w(2)               % display the 2nd element of w
ans = 3
```

1.1.3.2 Colon notation and extracting parts of a vector A colon notation is an important shortcut, used when producing row vectors (see Table 3 and help colon):

```
>> 2:5
ans = 2 3 4 5
>> -2:3
ans = -2 -1 0 1 2 3
```

In general, first:step:last produces a vector of entities with the value first, incrementing by the step until it reaches last:

```
>> 0.2:0.5:2.4
ans = 0.2000 0.7000 1.2000 1.7000 2.2000
>> -3:3:10
ans = -3 0 3 6 9
>> 1.5:-0.5:-0.5        % negative step is also possible
ans =
1.5000 1.0000 0.5000 0 -0.5000
```

Parts of vectors can be extracted by using a colon notation:

```
>> r = [-1:2:6, 2, 3, -2]        % -1:2:6 => -1 1 3 5
r = -1 1 3 5 2 3 -2
>> r(3:6)        % get elements of r which are on the positions from 3 to 6
ans = 3 5 2 3
>> r(1:2:5)        % get elements of r which are on the positions 1, 3 and 5
ans = -1 3 2
>> r(5:-1:2)        % what will you get here?
```

1.1.3.3 Product, divisions and powers of vectors You can now compute the inner product between two vectors x and y of the same length, $x^T y = \sum_1 x_1 y_1$, in a simple way:

```
>> u = [-1; 3; 5]        % a column vector
>> v = [-1; 2; 7]        % a column vector
>> u * v        % you cannot multiply a column vector by a column vector
??? Error using ==> *
Inner matrix dimensions must agree.
>> u' * v        % this is the inner product
ans = 42
```

An another way to compute the inner product is by the use of the dot product, i.e. .*, which performs element-wise multiplication. For two vectors x and y, of the same length, it is defined as a vector $[x_1 y_1, x_2 y_2, \ldots, x_n y_n]$, thus, the corresponding elements of two vectors are multiplied. For instance:

```
>> u .* v        % this is an element-by-element multiplication
     1
     6
    35
>> sum(u.*v)        % this is an another way to compute the inner product
ans = 42
>> z = [4 3 1];        % z is a row vector
>> sum(u'.*z)        % this is the inner product
ans = 10
>> u'*z'        % since z is a row vector, u'*z' is the inner product
ans = 10
```

You can now tabulate easily values of a function for a given list of arguments. For instance:
>> x = 1:0.5:4;
>> y = sqrt(x) .* cos(x)
y = 0.5403 0.0866 -0.5885 -1.2667 -1.7147 -1.7520 -1.3073
Mathematically, it is not defined how to divide one vector by another. However, in MatLab, the operator ./ is defined to perform an element-by-element division. It is, therefore, defined for vectors of the same size and type:
>> x = 2:2:10
x = 2 4 6 8 10
>> y = 6:10
y = 6 7 8 9 10
>> x./y
ans = 0.3333 0.5714 0.7500 0.8889 1.0000
>> z = -1:3
z = -1 0 1 2 3
>> x./z % division 4/0, resulting in Inf
Warning: Divide by zero.
ans = -2.0000 Inf 6.0000 4.0000 3.3333
>> z./z % division 0/0, resulting in NaN
Warning: Divide by zero.
ans = 1 NaN 1 1 1
The operator ./ can be also used to divide a scalar by a vector:
>> x=1:5; 2/x % this is not possible
??? Error using ==> /
Matrix dimensions must agree.
>> 2./x % but this is!
ans = 2.0000 1.0000 0.6667 0.5000 0.4000

Command	Result
A(i,j)	A_{ij}
A(:,j)	j-th column of A
A(i,:)	i-th row of A
A(k:l,m:n)	(l-k + 1) × (n -m+ 1) matrix with elements A_{ij} with $k \leq i \leq l, m \leq j \leq n$
v(i:j)'	'vector-part' $(v_i, v_{i+1}, \ldots, v_j)$ of vector v

Table 1.3 Manipulation of (groups of) matrix elements.

1.1.3.4 Matrices Row and column vectors are special types of matrices. An n × k matrix is a rectangular array of numbers having n rows and k columns. Defining a matrix in MatLab is similar to defining a vector. The generalization is straightforward, if you see that a matrix consists of row vectors (or column vectors). Commas or spaces are used to separate elements in a row, and semicolons are used to separate individual rows.

- **Define a matrix:**

For example, the matrix $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ is defined as:

```
>> A = [1 2 3; 4 5 6; 7 8 9]      % row by row input
A =
     1   2   3
     4   5   6
     7   8   9
```

Command	Result
C = A + B	sum of two matrices
C = A - B	subtraction of two matrices
C = A * B	multiplication of two matrices
C = A .* B	'element-by-element' multiplication (A and B are of equal size)
C = A ^ k	power of a matrix (k 2 Z; can also be used for A1)
C = A .^k	'element-by-element' power of a matrix
C = A'	the transposed of a matrix; AT
C = A ./ B	'element-by-element' division (A and B are of equal size)
X = A \ B	finds the solution in the least squares sense to the system AX = B
X = B / A	finds the solution of XA = B, analogous to the previous command

Table 1.4 Frequently used matrix operations.

Another examples are, for instance:

```
>> A2 = [1:4; -1:2:5]
A2 =
     1   2   3   4
    -1   1   3   5
>> A3 =  [   1   3
            -4   7  ]
A3 =
     1   3
    -4   7
```

From that point of view, a row vector is a 1 × k matrix and a column vector is an n × 1 matrix. Transposing a vector changes it from a row to a column or the other way around. This idea can be extended to a matrix, where transposing interchanges rows with the corresponding columns, as in the example:

```
>> A2
     1   2   3   4
    -1   1   3   5
>> A2'      % transpose of A2
ans =
```

Command	Result
n = rank(A)	n is the rank of matrix A
x = det(A)	x is the determinant of matrix A
x = size(A)	x is a row-vector with 2 elements: the number of rows and columns of A
x = trace(A)	x is the trace (sum of diagonal elements) of matrix A
x = norm(v)	x is the Euclidean length of vector v
C = inv(A)	C becomes the inverse of A
C = null(A)	C is an orthonormal basis for the null space of A obtained from the singular value decomposition
C = orth(A)	C is an orthonormal basis for the range of A
C = rref(A)	C is the reduced row echelon form of A
L = eig(A)	L is a vector containing the eigenvalues of a square matrix A
[X,D] = eig(A)	produces a diagonal matrix D of eigenvalues and a full matrix X whose columns are the corresponding eigenvectors of A
S = svd(A)	S is a vector containing the singular values of A
[U,S,V] = svd(A)	S is a diagonal matrix with nonnegative diagonal elements in decreasing order; columns of U and V are the accompanying singular vectors
x = linspace(a,b,n)	generates a vector x of n equally spaced points between a and b
x = logspace(a,b,n)	generates a vector x starting at 10a and ended at 10b containing n values
A = eye(n)	A is an n × n identity matrix
A = zeros(n,m)	A is an n × m matrix with zeros (default m = n)
A = ones(n,m)	A is an n × m matrix with ones (default m = n)
A = diag(v)	gives a diagonal matrix with the elements $v_1, v_2, \ldots, v_n$ on the diagonal
X = tril(A)	X is lower triangular part of A
X = triu(A)	X is upper triangular part of A
A = rand(n,m)	A is an n × m matrix with elements uniformly distributed between 0 & 1
A = randn(n,m)	ditto - with elements standard normal distributed
v = max(A)	v is a vector with the maximum value of the elements in each column of A or v is the maximum of all elements if A is a vector
v = min(A)	ditto - with minimum
v = sum(A)	ditto - with sum

Table 1.5 Frequently used matrix functions.

```
     1   -1
     2    1
     3    3
     4    5
>> size(A2)      % returns the size (dimensions) of A2: 2 rows, 4 columns
ans =
     2 4
```

```
>> size(A2')
ans =
     4 2
```

- **Special matrices:**

There is a number of built-in matrices of size specified by the user (see Table 4). A few examples are given below:

```
>> E = [ ]        % an empty matrix of 0-by-0 elements!
E =
     [ ]
>> size(E)
ans =
     0 0
>> I = eye(3);        % the 3-by-3 identity matrix
I =
     1   0   0
     0   1   0
     0   0   1
>> x = [2; -1; 7]; I*x        % I is such that for any 3-by-1 x holds I*x = x
ans =
      2
     −1
      7
>> r = [1 3 -2]; R = diag(r)        % create a diagonal matrix with r on the diagonal
R =
     1   0    0
     0   3    0
     0   0   −2
>> A = [1 2 3; 4 5 6; 7 8 9];
>> diag(A)        % extracts the diagonal entries of A
ans =
     1
     5
     9
>> B = ones(3,2)
B =
     1   1
     1   1
     1   1
>> C = zeros (size(C'))        d% a matrix of all zeros of the size given by C'
C =
     0   0   0
     0   0   0
>> D = rand(2,3)        % a matrix of random numbers; you will get a different one!
D =
```

```
    0.0227   0.9101   0.9222
    0.0299   0.0640   0.3309
>> v = linspace(1,2,4)        % a vector is also an example of a matrix
v =
    1.0000   1.3333   1.6667   2.0000
```

- **Building matrices and extracting parts of matrices:**

It is often needed to build a larger matrix from the smaller ones:

```
>> x = [4; -1], y = [-1 3]
x =
     4
    -1
y =
    -1   3
>> X = [x y']        % X consists of the columns x and y'
X =
     4   -1
    -1    3
>> T = [ -1 3 4; 4 5 6]; t = 1:3;
>> T = [T; t]        % add to T a new row, namely the row vector t
T =
    -1   3   4
     4   5   6
     1   2   3
>> G = [1 5; 4 5; 0 2];        % G is a matrix of the 3-by-2 size; check size(G)
>> T2 = [T G]        % concatenate two matrices
T2 =
    -1   3   4   1   5
     4   5   6   4   5
     1   2   3   0   2
>> T3 = [T; G ones(3,1)]        % G is 3-by-2, T is 3-by-3
T3 =
    -1   3   4
     4   5   6
     1   2   3
     1   5   1
     4   5   1
     0   2   1
>> T3 = [T; G'];        % this is also possible; what do you get here?
>> [G' diag(5:6); ones(3,2) T]        % you can concatenate many matrices
ans =
```

```
    1   4    0      5   0
    5   5    2      0   6
    1   1   -1      3   4
    1   1    4      5   6
    1   1    1   2 * 3
```

A part can be extract from a matrix in a similar way as from a vector. Each element in the matrix is indexed by a row and a column to which it belongs. Mathematically, the element from the i-th row and the j-th column of the matrix A is denoted by Aij ; Matlab provides the A(i,j) notation.

```
>> A = [1:3; 4:6; 7:9]
A =
    1   2   3
    4   5   6
    7   8   9
>> A(1,2), A(2,3), A(3,1)
ans =
     2
ans =
     6
ans =
     7
>> A(4,3)       % this is not possible: A is a 3-by-3 matrix!
??? Index exceeds matrix dimensions.
>> A(2,3) = A(2,3) + 2*A(1,1)       % change the value of A(2,3)
A =
    1   2   3
    4   5   8
    7   8   9
```

It is easy to extend a matrix automatically. For the matrix A it can be done e.g. as follows:

```
>> A(5,2) = 5       % assign 5 to the position (5,2); the uninitialized
A =       % elements become zeros
    1   2   3
    4   5   8
    7   8   9
    0   0   0
    0   5   0
```

If needed, the other zero elements of the matrix A can be also de

ned, by e.g.:

```
>> A(4,:) = [2, 1, 2];       % assign vector [2, 1, 2] to the 4th row of A
>> A(5,[1,3]) = [4, 4];       % assign: A(5,1) = 4 and A(5,3) = 4
>> A       % how does the matrix A look like now?
```

Different parts of the matrix A can be now extracted:

```
>> A(3,:)       % extract the 3rd row of A
ans =
```

```
     7 8 9
>> A(:,2)       % extract the 2nd column of A
ans =
     2
     5
     8
     1
     5
>> A(1:2,:)      % extract the rows 1st and 2nd of A
ans =
     1   2   3
     4   5   8
>> A([2,5],1:2)      % extract a part of A
ans =
     4   5
     4   5
```

As you have seen in the examples above, it is possible to manipulate (groups of) matrix-elements. The commands are shortly explained in Table 3. The concept of an empty matrix [] is also very useful in Matlab. For instance, a few columns or rows can be removed from a matrix by assigning an empty matrix to it. Try for example:

```
>> C = [1 2 3 4; 5 6 7 8; 1 1 1 1];
>> D = C; D(:,2) = []       % now a copy of C is in D; remove the 2nd column of D
>> C ([1,3],:) = []       % remove the rows 1 and 3 from C
```

- **Operations on matrices**

Table 4 shows some frequently used matrix operations and functions. The important ones are dot operations on matrices, matrix-vector products and matrix-matrix products. In the class of the dot operations, there are dot product, dot division and dot power. Those operations work as for vectors: they address matrices in the element-by-element way, therefore they can be performed on matrices of the same sizes. They also allow for scalar-matrix operations. For the dot product or division, corresponding elements are multiplied together or divided one by another. A few examples of basic operations are provided below:

```
>> B = [1 -1 3; 4 0 7]
B =
     1   −1   3
     4    0   7
>> B2 = [1 2; 5 1; 5 6];
>> B = B + B2'       % add two matrices; why B2' is needed instead of B2?
B =
     2   4    8
     6   1   13
>> B-2       % subtract 2 from all elements of B
ans =
     0    2    6
     4   −1   11
>> ans = B./4       % divide all elements of the matrix B by 4
ans =
```

```
    0.5000   1.0000   2.0000
    1.5000   0.2500   3.2500
>> 4/B       % this is not possible
??? Error using ==> /
Matrix dimensions must agree.
>> 4./B      % this is possible; equivalent to: 4.*ones(size(B)) ./ B
ans =
    2.0000   1.0000   0.5000
    0.6667   4.0000   0.3077
>> C = [1 -1 4; 7 0 -1];
>> B .* C       % multiply element-by-element
ans =
     2   -4    32
    42    0   -13
>> ans.^3 - 2       % do for all elements: raise to the power 3 and subtract 2
ans =
         6   -66   32766
     74086    -2   -2199
>> ans ./ B.^2       % element-by-element division
ans =
    0.7500   -1.0312   63.9961
  342.9907   -2.0000   -1.0009
>> r = [1 3 -2]; r * B2       % this is a legal operation: r is a 1-by-3 matrix and B2 is
ans =       % 3-by-2 matrix; B2 * r is an illegal operation
6 -7
```

Concerning the matrix-vector and matrix-matrix products, two things should be reminded. First, an n × k matrix A (having n rows and k columns) can be multiply by a k × 1 (column) vector x, resulting in a column n × 1 vector y, i.e.: Ax = y such that $y_i = \sum_{p=1}^{k} A_{ip}x_p$. Multiplying a 1 × n (row) vector x by a matrix A, results in a 1 × k (row) vector y. Secondly, an n × k matrix A can be multiply by a matrix B, only if B has k rows, i.e. B is k × m (m is arbitrary). As a result, you get n × m matrix C, such that AB = C, where $C_{ij} = \sum_{p=1}^{k} A_{ip}B_{pj}$.

```
>> b = [1 3 -2];
>> B = [1 -1 3; 4 0 7]
B =
    1   -1   3
    4    0   7
>> b * B       % not possible: b is 1-by-3 and B is 2-by-3
??? Error using ==¿ *
Inner matrix dimensions must agree.
>> b * B'       % this is possible: a row vector multiplied by a matrix
ans =
    -8 -10
>> B' *ones(2,1)
ans =
     5
    -1
```

```
      10
>> C = [3 1; 1 -3];
>> C * B
ans =
       7   -3    16
     -11   -1   -18
>> C.^3       % this is element-by-element power
ans =
      27     1
       1   -27
27 1 1 -27  >> C^3        % this is equivalent to C*C*C
ans =
      30    10
      10   -30
>> ones(3,4)./4 * diag(1:4)
ans =
    0.2500    0.5000    0.7500    1.0000
    0.2500    0.5000    0.7500    1.0000
    0.2500    0.5000    0.7500    1.0000
```

1.2 M-File

1.2.1 Scripts

MatLab commands can be entered at the MatLab prompt. When a problem is more complicated this becomes inefficient. A solution is using script m-files. They are useful when the number of commands increases or when you want to change values of some variables and re-evaluate them quickly. Formally, a script is an external file that contains a sequence of MatLab commands (statements). However, it is not a function, since there are no input/output parameters and the script variables remain in the workspace. So, when you run a script, the commands in it are executed as if they have been entered through the keyboard.

1.2.2 Function

Functions m-files are true subprograms, since they take input arguments and/or return output parameters. They can call other functions, as well. Variables defined and used inside a function, different from the input/output arguments, are invisible to other functions and the command environment. The general syntax of a function is presented below:

function [outputArgs] = function_name (inputArgs)

outputArgs are enclosed in [ ]:

- a comma-separated list of variable names;

  is optional when only one argument is present;

- functions without outputArgs are legal.

inputArgs are enclosed in ( ):

- a comma-separated list of variable names;

- functions without inputArgs are legal.

MatLab provides a structure for creating your own functions. The first line of the file should be a definition of a new function (also called a header). After that, a continuous sequence of comment lines should appear. Their goal is to explain what the function does, especially when this is not trivial. Not only a general description, but also the expected input parameters, returned output parameters and synopsis should appear there. The comment lines (counted up to the first non-comment line) are important since they are displayed in response to the help command. Finally, the remainder of the function is called the body. Function m-files terminate execution and return when they reached the end of the file or, alternatively, when the command return is encountered. As an example, the function average is defined as follows:

```
1  function avr = average (x)
2  %AVERAGE computes the average value of a vector x
3  %     and returns it in avr
4
5  % Notes: an example of a function
6  n = length(x);
7  avr = sum(x)/n;
8  return;
```

Marks: the first line must be the function definition. The second and third lines are explanation of file for help of average. A blank line within the comment is used as a break point of help. Notes information will NOT appear when you ask for help.

Important: The name of the function and the name of the file stored on disk should be identical. In our case, the function should be stored in a file called average.m.

1.2.3 Execution of M-File

For a script of M-file, just simply use the name of the file to execute each command in the file sequently.

However, for a function of M-file, all input variables are needed to define first. You can define them first, or you can use the command line to specify each variable.

Symbol	Color	Symbol	Line style
r	red	., o	point, circle
g	green	*	star
b	blue	x, +	x-mark, plus
y	yellow	-	solid line
m	magenta	--	dash line
c	cyan	:	dot line
k	black	-.	dash-dot line

Table 1.6 Plot colors and styles.

1.3 Visualization

MatLab can be used to visualize the results of an experiment. Therefore, you should define variables, each of them containing all values of one parameter to plot.

1.3.1 Simple Plots

With the command plot, a graphical display can be made. For a vector y, plot(y) draws the points [1, y(1)], [2, y(2)], ..., [n, y(n)] and connects them with a straight line. plot(x,y) does the same for the points [x(1), y(1)], [x(2), y(2)], ..., [x(n), y(n)]. Note that x and y have to be both either row or column vectors of the same length (i.e. the number of elements). The commands loglog, semilogx and semilogy are similar to plot, except that they use either one or two logarithmic axes.

Type the following commands after predicting the result:

```
>> x = 0:10;
>> y = 2.^ x;        % this is the same as y = [1 2 4 8 16 32 64 128 256 512 1024]
>> plot(x,y)         % to get a graphic representation
>> semilogy(x,y)     % to make the y-axis logarithmic
```

As you can see, the same figure is used for both plot commands. The previous function is removed as soon as the next is displayed. The command figure gives you an extra figure. Repeat the previous commands, but generate a new figure before plotting the second function, so that you can see both functions in separate windows. You can also switch back to a figure using figure(n), where n is its number.

To plot a graph of a function, it is important to sample the function sufficiently well. Compare the following examples:

```
>> n = 5;
>> x = 0:1/n:3;      % coarse sampling
>> y = sin(5*x);
>> plot(x,y)

>> n = 25;
>> x = 0:1/n:3;      % good sampling
>> y = sin(5*x);
>> plot(x,y)
```

The solid line is used by plot by default. It is possible to change the style and the color, e.g.:

```
>> x = 0:0.4:3; y = sin(5*x);
>> plot(x,y,'r- -')
```

produces the dashed red line. The third argument of plot specifies the color (optional) and the line style. Table 1.6 shows a few possibilities, help plot shows all. To add a title, grid and to label the axes, one uses:

```
>> title('Function y = sin(5*x)');
>> xlabel('x-axis');
>> ylabel('y-axis');
>> grid on       % remove grid by calling grid off
```

Command	Result
grid on/off	adds a grid to the plot at the tick marks or removes it
axis([xmin xmax ymin ymax])	sets the minimum and maximum values of the axes
box off/on	removes the axes box or shows it
xlabel('text')	plots the label text on the x axis
ylabel('text')	plots the label text on the y axis
title('text')	plots a title above the graph
text(x,y,'text')	adds text at the point (x,y)
gtext('text')	adds text at a manually (with a mouse) indicated point
legend('fun1','fun2')	plots a legend box (move it with your mouse) to name your functions
legend off	deletes the legend box
clf	clear the current figure
subplot	create a subplot in the current gure

Table 1.7 Plot-manipulations.

1.3.2 Multiple curves in a Plot or plots

There are different ways to draw several functions in the same figure. The first one is with the command *hold on*. After this command, all functions will be plotted in the same figure until the command *hold off* is used. When a number of functions is plotted in a figure, it is useful to use different symbols and colors. An example is:

```
>> x1 = 1:.1:3.1; y1 = sin(x1);
>> plot(x1,y1,'md');
>> x2 = 1:.3:3.1; y2 = sin(-x2+pi/3);
>> hold on
>> plot(x2,y2,'k*-.')
>> plot(x1,y1,'m-')
>> hold off
```

A second method to display a few functions in one figure is to plot several functions at the same time. The next commands will produce the same output as the commands in the previous example:

```
>> x1 = 1:.1:3.1; y1 = sin(x1);
>> x2 = 1:.3:3.1; y2 = sin(-x2+pi/3);
>> plot(x1, y1,'md', x2, y2, 'k*-.', x1, y1, 'm-')
```

To make the axes better fitting the curves, perform:

```
>> axis([1,3.1,-1,1])
```

The same can be achieved by axis tight. It might be also useful to exercise with axis options (see help), e.g. axis on/off, axis equal, axis image or axis normal. A descriptive legend can be included with the command legend, e.g.:

```
>> legend ('sin(x)', 'sin(-x+pi/3)');
```

It is also possible to produce a few subplots in one figure window. With the command subplot, the window can be horizontally and vertically divided into p × r subfigures, which are counted from 1 to pr, row-wise, starting from the top left. The commands: plot, title, grid etc work only in the current subfigure.

```
>> x = 1:.1:4;
```

```
>> y1 = sin(3*x);
>> y2 = cos(5*x);
>> y3 = sin(3*x).*cos(5*x);
>> subplot(1,3,1); plot(x,y1,'m-'); title('sin(3*x)')
>> subplot(1,3,2); plot(x,y2,'g-'); title('cos(5*x)')
>> subplot(1,3,3); plot(x,y3,'k-'); title('sin(3*x) * cos(5*x)')
```

1.3.3 2D and 3D plots

Some commands similar to plot, loglog, semilogx and semilogy were mentioned. There are, however, more ways to display data. MatLab has a number of functions designed for plotting specialized 2D graphs, e.g.: fill, polar, bar, barh, pie, hist, errorbar or stem. In the example below, fill is used to create a polygon:

```
>> N = 5; k = -N:N;
>> x = sin(k*pi/N);
>> y = cos(k*pi/N);        % x and y - vertices of the polygon to be filled
>> fill(x,y,'g')
>> axis square
>> text(-0.45,0,'I am a green polygon')
```

To get an impression of other visualizations, type the following commands and describe the result (note that the command figure creates a new figure window):

```
>> figure        % bar plot of a bell shaped curve
>> x = -2.9:0.2:2.9;
>> bar(x,exp(-x.*x));
>> figure        % stairstep plot of a sine wave
>> x = 0:0.25:10;
>> stairs(x,sin(x));
>> figure        % errorbar plot
>> x = -2:0.1:2;
>> y = erf(x);        % error function; check help if you are interested
>> e = rand(size(x)) / 10;
>> errorbar (x,y,e);
>> figure
>> r = rand(5,3);
>> subplot(1,2,1); bar(r,'grouped')        % bar plot
>> subplot(1,2,2); bar(r,'stacked')
>> figure
>> x = randn(200,1);        % normally distributed random numbers
>> hist(x,15)        % histogram with 15 bins
```

The command plot3 to plot lines in 3D is equivalent to the command plot in 2D. The format is the same as for plot, it is, however, extended by an extra coordinate. An example is plotting the curve r defined parametrically as r(t) = [t sin(t), t cos(t); t] over the interval $[-10\pi, 10\pi]$.

```
>> t = linspace(-10*pi,10*pi,200);
>> plot3(t.*sin(t), t.*cos(t), t, 'md-');        % plot the curve in magenta
>> title('Curve r(t) = [t sin(t), t cos(t), t]');
>> xlabel('x-axis'); ylabel('y-axis'); zlabel('z-axis');
>> grid
```

Exercise:

Make a 3D smooth plot of the curve defined parametrically as: [x(t), y(t), z(t)] = [sin(t), cos(t), sin2(t)] for $t = [0, 2\pi]$. Plot the curve in green, with the points marked by circles. Add a title, description of axes and the grid. You can rotate the image by clicking Tools at the Figure window and choosing the Rotate 3D option or by typing rotate3D at the prompt. Then by clicking at the image and dragging your mouse you can rotate the axes. Exercise with this option.

MatLab provides a number of commands to plot 3D data. A surface is defined by a function f(x, y), where for each pair of (x, y), the height z is computed as z = f(x, y). To plot a surface, a rectangular domain of the (x; y)-plane should be sampled. The mesh (or grid) is constructed by the use of the command **meshgrid** as follows:

```
>> [X, Y] = meshgrid (-1:.5:1, 0:.5:2)
X =
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
Y =
         0         0         0         0         0
    0.5000    0.5000    0.5000    0.5000    0.5000
    1.0000    1.0000    1.0000    1.0000    1.0000
    1.5000    1.5000    1.5000    1.5000    1.5000
    2.0000    2.0000    2.0000    2.0000    2.0000
```

The domain [-1, 1] × [0, 2] is now sampled with 0:5 in both directions and it is described by points [X(i, j), Y (i, j)]. To plot a smooth surface, the chosen domain should be sampled in a more dense way. To plot a surface, the command **mesh** or **surf** can be used:

```
>> [X,Y] = meshgrid(-1:.05:1, 0:.05:2);
>> Z = sin(5*X) .* cos(2*Y);
>> mesh(X,Y,Z);
>> title ('Function z = sin(5x) * cos(2y)')
```

You can also try **waterfall** instead of mesh.

The MatLab function peaks is a function of two variables, obtained by translating and scaling Gaussian distributions. Perform, for instance:

```
>> [X,Y,Z] = peaks;        % create values for plotting the function
>> surf(X,Y,Z);        % plot the surface
>> figure
>> contour (X,Y,Z,30);        % draw the contour lines in 2D
>> colorbar        % adds a bar with colors corresponding to the z-axis
>> title('2D-contour of PEAKS');
>> figure
>> contour3(X,Y,Z,30);        % draw the contour lines in 3D
>> title('3D-contour of PEAKS');
>> pcolor(X,Y,Z);        % z-values are mapped to the colors and presented as
                            % a 'checkboard' plot; similar to contour
```

1.3.4 Printing

Before printing a figure, you might want to add some information, such as a title, or change somewhat in the lay-out. Table 1.7 shows some of the commands that can be used.

Example:

Plot the functions y1 = sin(4x), y2 = x cos(x), $y3 = (x+1)^{-1}\sqrt{x}$ for x = 1 : 0:25 : 10; and a single point (x, y) = (4, 5) in one figure. Use different colors and styles. Add a legend, labels for both axes and a title. Add also a text to the single point saying: 'single point'. Change the minimum and maximum values of the axes such that one can look at the function y3 in more detail.

When you like the displayed figure, you can print it to paper. The easiest way is to click on **File** in the menu-bar and to choose **Print**. If you click OK in the print window, your figure will be sent to the printer indicated there. There exists also a print command, which can be used to send a figure to a printer or output it to a file. You can optionally specify a print device (i.e. an output format such as *tiff* or *postscript*) and options that control various characteristics of the printed file (i.e., which Figure to print etc). You can also print to a file if you specify the file name. If you do not provide an extension, print adds one. Since they are many parameters they will not be explained here (check **help print** to learn more). Instead, try to understand the examples:

```
>> print -dwinc          % print the current Figure to the current printer in color
>> print -f1 -deps myfile.eps        % print Figure no.1 to the file myfile.eps in black
>> print -f1 -depsc myfilec.eps      % print Figure no.1 to the file myfilec.eps in color
>> print -dtiff myfile1.tiff         % print the current Figure to the file myfile1.tiff
>> print -dpsc myfile1c.ps           % print the current Figure to the file myfile1.ps in color
>> print -f2 -djpeg myfile2          % print Figure no.2 to the file myfile2.jpg
```

1.4 Input/Output

1.4.1 Data Input/Output

The easiest way to save or load MatLab variables is by using (clicking) the File menu-bar, and then selecting the Save Workspace as... or Load Workspace... items respectively. Also MatLab commands exist which save data to files and which load data from files. The command save allows for saving your workspace variables either into a binary file or an ASCII file (check Preliminaries on binary and ASCII files). Binary files automatically get the '**.mat**' extension, which is not true for ASCII files. However, it is recommended to add a '**.txt**' or **.dat** extension.

Learn how to use the **save** command by exercising:

```
>> s1 = sin(pi/4);
>> c1 = cos(pi/4); c2 = cos(pi/2);
>> str = 'hello world';        % this is a string
>> save          % saves all variables in binary format to matlab.mat
>> save data     % saves all variables in binary format to data.mat
>> save numdata s1, c1         % saves numeric variables s1 and c1 to numdata.mat
>> save strdata str            % saves a string variable str to strdata.mat
>> save allcos.dat c* -ascii   % saves c1,c2 in 8-digit ascii format to allcos.dat
```

The load command allows for loading variables into the workspace. It uses the same syntax as **save**. Try to load variables from the created files. Before each load command, clear the workspace and after loading check which variables are present in the workspace (use **who**).

```
>> load        % loads all variables from the file matlab.mat
>> load data s1 c1        % loads only specified variables from the file data.mat
>> load str data        % loads all variables from the file strdata.mat
```

It is also possible to read ASCII files that contain rows of space separated values. Such a file may contain comments that begin with a percent character. The resulting data is placed into a variable with the same name as the ASCII file (without the extension). Check, for example:

```
>> load allcos.dat        % loads data from allcos.dat into variable allcos
>> who        % lists variables present in the workspace now
```

1.4.2 Screen Output

The **fprintf** command converts data to character strings and displays it on screen or writes it to a file. The general syntax is:

```
fprintf (fid,format,a,...)
```

Consider the following example:

```
>> x = 0:0.1:1;
>> y = [x; exp(x)];
>> fid = fopen ('exptab.txt','w');
>> fprintf(fid, 'Exponential function\n');
>> fprintf(fid, '%6.2f %12.8f\n',y);
>> fclose(fid);
```

1.4.3 File Input/Output

MatLab file input and output (I/O) functions read and write arbitrary binary and formatted text files. This enables you to read data collected in other formats and to save data for other programs, as well. Before reading or writing a file you must open it with the **fopen** command:

```
>> fid = fopen (file_name, permission);
```

The permission string specifies the type of access you want to have:

- 'r' - for reading only
- 'w' - for writing only
- 'a' - for appending only
- 'r+' - both for reading and writing

Here is an example:

```
>> fid = fopen ('results.txt','w')        % tries to open the file results.txt for writing
```

The **fopen** statement returns an integer *file identifier*, which is a handle to the file (used later for addressing and accessing your file). When **fopen** fails (e.g. by trying to open a non-existing file), the file identifier becomes -1. It is also possible to get an error message, which is returned as the second optional output argument. It is a good habit to test the file

identifier each time when you open a file, especially for reading. Below, the example is given, when the user provides a string until it is a name of a readable file:

```
fid = 0;
while fid < 1
    fname = input ('Open file: ', 's');
    [fid, message] = fopen (fname, 'r');
    if (fid == -1)
        disp (message);
    end
end
```

When you finish working on a file, use **fclose** to close it up. MatLab automatically closes all open files when you exit it. However, you should close your file when you finished using it:

```
fid = fopen ('results.txt', 'w');
...
fclose(fid);
```

1.5 Control Flow

A control ow structure is a block of commands that allows conditional code execution and making loops.

1.5.1 Logical and Relational Flow

To use control flow commands, it is necessary to perform operations that result in logical values: **TRUE** or **FALSE**. In MatLab the result of a logical operation is 1 if it is true and 0 if it is false. Table 1.8 shows the relational and logical operations. Another way to get to know more about them is to type **help relop**. The relational operators $<, <=, >, >=, ==$ and $\sim=$ can be used to compare two arrays of the same size or an array to a scalar. The logical operators &, | and allow for the logical combination or negation of relational operators. In addition, three functions are also available: **xor**, **any** and **all** (use **help** to find out more).

Command	Result
a = (b > c)	a is 1 if b is larger than c. Similar are: <, >= and <=
a = (b == c)	a is 1 if b is equal to c
a = (b ~= c)	a is 1 if b is not equal c
a = ~b	logical complement: a is 1 if b is 0
a = (b & c)	logical AND: a is 1 if b = TRUE AND c = TRUE
a = (b | c)	logical OR: a is 1 if b = TRUE OR c = TRUE

Table 1.8 Relational and logical operations.

Important: The logical & and — have the equal precedence in MatLab, which means that those operators associate from left to right. A common situation is:

```
>> b = 10;
>> 1 | b > 0 & 0
ans =
     0
>> (1 | b > 0) & 0
ans =
     0
>> 1 | (b > 0 & 0)
ans =
     1
```

This shows that you should always use brackets to indicate in which way the operators should be evaluated.

1.5.2 Conditional Flow

Selection control structures, **if**-blocks, are used to decide which instruction to execute next depending whether *expression* is **TRUE** or not. The general description is given below. In the examples below the command **disp** is frequently used. This command displays on the screen the text between the quotes.

- **if ... end**

	Syntax		Example
if	logical_expression	if	$(a > 0)$
	statement1		$b = a;$
	statement2		disp ('a is positive');
	...	end	
end			

- **if ... else ... end**

	Syntax		Example
if	logical_expression	if	(temperature > 100)
	block of statements		disp ('Above boiling.');
	evaluated if TRUE		toohigh = 1;
else		else	
	block of statements		disp ('Temperature is OK.');
	evaluated if FALSE		toohigh = 0;
end		end	

Another selection structure is **switch**, which switches between several cases depending on an expression, which is either a scalar or a string. The statements following the first **case** where the expression matches the choice are executed. This construction can be very handy to avoid long **if** ... **elseif** ... **else** ... **end** constructions.

Syntax	Example

switch expression
  case choice1
  block of commands1
  case {choice2a, choice2b, ...}
  block of commands2
  ...
  otherwise
  block of commands
end
	method = 2;
	switch method
	  case {1, 2}
	  disp('Method is linear.');
	  case 3
	  disp('Method is cubic.');
	  case 4
	  disp('Method is nearest.');
	  otherwise
	  disp('Unknown method.');
	end

1.5.3 Loops

Iteration control structures, loops, are used to repeat a block of statements until some condition is met. Two types of loops exist:

- the **for** loop that repeats a group of statements a fixed number of times; You can

Syntax	Example

for index = first:step:last
  block of commands
end
	sumx = 0;
	for i=1:length(x)
	  sumx = sumx + x(i);
	end

  specify any step, including a negative value. The index of the for-loop can be also a vector.See some examples of possible variations:

Example 1	Example 2	Example 3	Example 4

for i=1:2:n
  ...
end
	for i=n:-1:3
	  ...
	end
		for x=0:0.5:4
		  disp(x^2);
		end
			for x=[25 9 81]
			  disp(sqrt(x));
			end

- **while** loop, which evaluates a group of commands as long as expression is TRUE.

A simple example how to use the loop construct can be to draw graphs of f(x) = cos(n x) for $n = 1, \ldots, 9$ in different subplots. Execute the following script:

```
figure
hold on
x = linspace(0,2*pi);
for n=1:9
    subplot(3,3,n);
    y = cos(n*x);
    plot(x,y);
```

Syntax	Example
while expression	N = 100;
statement1	iter = 1;
statement2	msum = 0;
statement3	while iter <= N
...	msum = msum + iter;
end	iter = iter + 1;
	end

```
    axis tight
end
```

Given two vectors x and y, an example use of the loop construction is to create a matrix A whose elements are defined, e.g. as $A_{ij} = x_i y_j$. Enter the following commands to a script:

```
n = length(x);
m = length(y);
for i=1:n
    for j=1:m
        A(i,j) = x(i) * y(j);
    end
end
```

and create A for x = [1 2 -1 5 -7 2 4] and y = [3 1 -5 7]. Note that A is of size n-by-m. The same problem can be solved by using the while-loop, as follows:

```
n = length(x);
m = length(y);
i = 1; j = 1;       % initialize i and j
while i <= n
    while j <= m
        A(i,j) = x(i) * y(j);
        j = j+1;        % increment j; it does not happen automatically
    end
    i = i+1;        % increment i
end
```