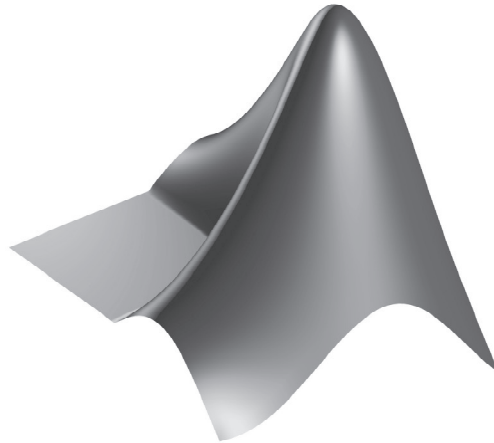


# Introduction to MATLAB



Ela Pekalska and many others

Quantitative Imaging Group  
Department of Imaging Science and Technology  
Faculty of Applied Sciences  
Delft University of Technology  
2001-2010

send remarks to: Bernd Rieger, [b.rieger@tudelft.nl](mailto:b.rieger@tudelft.nl)



1	Getting started with Matlab.....	5
1.1	Key Concepts .....	5
1.2	Starting Matlab.....	6
1.3	Input and output via the Command Window .....	6
1.4	Interrupting a command or program .....	7
1.5	Help-facilities.....	8
1.6	Introduction to the Workspace .....	8
1.6.1	Path.....	9
1.6.2	Saving and loading Workspace variables .....	9
2	Basic syntax and variables in MATLAB .....	9
2.1	Matlab as a calculator .....	10
2.2	Assignments and variables.....	10
2.3	Matlab syntax and semantics issues.....	11
2.3.1	Syntax trees and logical data type.....	12
3	Matlab programs: script and function m-files .....	14
3.1	Script m-files.....	14
3.2	Function m-files .....	16
3.3	Workspace issues .....	17
3.3.1	Local and global variables .....	17
3.3.2	Special function variables.....	17
3.4	Scripts vs. functions .....	17
4	Mathematics with vectors and matrices in Matlab.....	18
4.1	Vectors .....	18
4.1.1	Colon notation and extracting parts of a vector .....	18
4.1.2	Column vectors and transposing.....	19
4.1.3	Product, division and powers of vectors .....	20
4.2	Matrices .....	23
4.2.1	Special matrices .....	24
4.2.2	Building matrices and extracting parts of matrices.....	25
4.2.3	Operations on matrices .....	27
5	Text and cell arrays.....	29
5.1	Character strings .....	29
5.2	Text input and output .....	31
5.3	Cell arrays .....	34
6	Control flow.....	35
6.1	Expressions and logical expressions .....	35
6.2	Sequential commands .....	35
6.3	Conditional commands .....	35
6.4	Iterative commands.....	37
6.4.1	Definite iteration.....	37
6.4.2	Indefinite iteration .....	37
6.5	Evaluation of logical and relational expressions in the control flow structures .....	38
7	Visualization .....	39
7.1	Simple plots .....	39
7.2	Several functions in one figure .....	40
7.3	Printing.....	42
8	Numerical analysis .....	42
8.1	Curve fitting .....	43
8.2	Evaluation of a function .....	44
8.3	Integration and differentiation.....	45
8.4	Numerical computations and the control of flow structures .....	46
9	Optimizing the performance of Matlab code.....	47
9.1	Vectorization - speed-up of computations .....	47
9.2	Array pre-allocations.....	48
9.3	Matlab's tricks and tips .....	48

10	Writing and debugging Matlab programs .....	51
10.1	Structural programming .....	51
10.2	Recommended programming style.....	53

## Introduction

During this course you will be introduced to designing and performing mathematical computations assisted by Matlab. You will also get acquainted with basic programming skills. If you learn to use this program well, you will find it very useful in the future, since many technical or mathematical problems (in thesis projects) can be solved with the help of Matlab.

This text includes all necessary material (with some additional information), however, many things are treated briefly. Therefore, we recommend one of the following books as an additional resource of information during the course:

- D.C. Hanselman, B. Littlefield, “Mastering Matlab 6: A Comprehensive Tutorial and Reference”, Prentice Hall, 2001, ISBN 0-13-019468-9 (\$60)
- D.M. Etter, D.C. Kuncicky, D.W. Hull, “Introduction to Matlab 6”, Prentice Hall, 2002, ISBN 0-13-032845-6 (\$32)

Exercises are marked with [☞] in the left margin. The end of an exercise is marked with [■].

Please test after each section whether you have **sufficient understanding of the issues** discussed. Use the lists provided below. You should be able to:

Section 1-2: MATLAB’s basis syntax and variables;

- use on-line `help` and `lookfor` to get more information on a command;
- recognize built-in variables;
- define variables and perform basic mathematical operations using them;
- add, remove and inspect variables in the MATLAB workspace;
- use the `load` and `save` commands to read/save Workspace data to/from a file.
- know how to suppress display with `;` (semicolon);
- use the `format` command to adjust output to the Command Window;
- access files at different directories (manipulate path-changing commands);
- use relational operators: `<`, `<=`, `>`, `>=`, `==`, `~=`, and logical operators: `&`, `|` and `~`;
- understand the logical addressing.

Section 3: MATLAB programs

- edit and run an m-file (both functions and scripts);
- identify the difference between scripts and functions;
- understand the concept of local and global variables;
- create a function with one or more input arguments and one or more output arguments;
- use comment statements to document scripts and functions.

Section 4: Mathematics with matrices

- create matrices (including vectors) with direct assignment, using `[ ]` (brackets);
- use `linspace` to create vectors;
- create random vectors and matrices;
- create matrices via the commands: `eye`, `ones`, `zeros` and `diag`;
- build a larger matrix from smaller ones;
- use `:` (colon) notation to create vectors and extract ranges of elements from vectors and matrices;
- extract elements from vectors and matrices with subscript notation, e.g., `x(5)`, `A(i,j)`;
- apply transpose operations to vectors and matrices;
- perform legal addition, subtraction, and multiplication operations on vectors and matrices;
- understand the use of dot operations, such as `.*`, `./`, `...` and know why they are different from the regular `*`, `/`, `...` operators;
- delete elements from vectors and matrices;
- compute inner products and the Euclidean length of (complex) vectors.

## Section 5: Text and cell arrays

- create and manipulate string variables, e.g., compare two strings, concatenate them, find a substring in a string, convert a number/string into a string/number, etcetera;
- use freely and with understanding the text input/output commands: `input`, `disp` and `fprintf`;
- operate on cell arrays.

## Section 6: Control flow

- use `if...end`, `if...elseif...end` and `if...elseif...else...end` and `switch` constructs
- use `for`-loops and `while`-loops and know the difference between them;
- understand how logical expressions are evaluated in the control flow structures.

## Section 7: Visualization

- use the `plot` command to make simple plots;
- know how to use `hold on/off`
- plot several functions in one figure either in one graphical window or by creating a few smaller ones (the use of `subplot`);
- add a title, grid and legend, describe the axes, change the range of the axes;
- use logarithmic axes;
- send plots to a printer or print them to a file.

## Section 8: Numerical analysis

- create and manipulate MATLAB polynomials;
- fit a polynomial to the data;
- interpolate the data;
- evaluate a function;
- integrate and differentiate a function;
- understand how to make approximations of Taylor expansions with given precision.

## Section 9: Optimize MATLAB commands on performance

- pre-allocate memory for vectors or matrices and know why and when this is beneficial;
- replace basic loops with vectorized operations;
- use `:` (colon) notation to perform vectorized operations;
- understand the two ways of addressing matrix elements using a vector as an index: traditional and logical indexing;
- use array indexing instead of loops to select elements from a matrix;
- use logical indexing and logical functions instead of loops to select elements from matrices;
- understand MATLAB's tricks.

## Section 10: Writing and debugging MATLAB programs

- know and understand the importance of structural programming and debugging;
- know how to debug your program;
- have an idea how to write programs using the recommended programming style.

# 1 Getting started with MATLAB

MATLAB (matrix laboratory) is a tool for mathematical (technical) calculations (computing): integrating computation, visualization and programming. It can be used as a scientific calculator. Apart from that, it allows you to plot or visualize data in many different ways, perform matrix algebra, work with polynomials or integrate functions. Like with a programmable calculator, you can create, execute and save a sequence of commands (script) in order to make your computational process automatic. It can be used to store and retrieve data. MATLAB can be treated as a programming language, which offers the possibility to handle calculations in an easy way. Note that MATLAB is especially designed to work with data sets as a whole, such as vectors, matrices and images. For that reason, PRTOOLS, a toolbox for Pattern Recognition purposes, and DIPLIB, a toolbox for Image Processing, have been developed under MATLAB.

## 1.1 Key Concepts

Please read the following basic definitions on computers and programming needed in the coming sections.

- A *bit* (short for *binary digit*) is the smallest unit of information on a computer. A single bit or electrical unit having only two states: charged and uncharged. It can hold one of two values: either 0 or 1.
- The *binary system* is a number system that has two unique digits: 0 and 1. Each digit position represents a different power of 2. The powers of 2 increase while moving from the rightmost to the leftmost position, starting from  $2^0 = 1$ . Here is an example of a binary number and its representation in the decimal system:
  - $\text{bin}(10110) = \text{dec}(1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) = 16 + 0 + 4 + 2 + 0 = 22$
- A *byte* is a unit of 8 bits. It can hold values (from 0 to 255) and is used to store a single character. Large amounts of memory are indicated in terms of kilobytes ( $1 \text{ kB} = 2^{10} = 1024$  bytes), megabytes ( $1 \text{ MB} = 1024 \text{ kB}$ ), and gigabytes ( $1 \text{ GB} = 1024 \text{ MB}$ ). So  $1 \text{ MB} = 2^{20} = 1,048,576$  bytes, and  $1 \text{ GB} = 2^{30} = 1,073,741,824$  bytes. (When it involves selling these bytes, hard disk manufacturers bend the rules: they use  $1 \text{ GB} = 10^9 \text{ bytes}$ ).
- *Floating point representation* is way to store very small or very large numbers with a fixed amount of bits/bytes. This representation can hold a much larger range of numbers than integer of fix point representation in the same amount of memory. It is comparable to the way scientist write very large and very small number e.g.  $c = 299792458 \text{ m/s}$   $2.99792458 \cdot 10^8 \text{ m/s}$ . Any number  $x$  is represented as  $x = m b^e$ , with  $m$  the mantissa,  $b$  the base and  $e$  the exponent. As base you chose 2 for computing. The name stems from the fact that the decimal point is not fixed.
- *Data* is information represented with symbols, e.g., numbers, words, signals or images.
- A **variable is a container that can hold a value**. For example, in the expression  $x+y$ ,  $x$  and  $y$  are variables. **It can represent data:** numeric values, such as 25.5, characters, such as 'c' or an array of characters (string) such as 'Matlab'. Every variable is identified with a name (called the variable name) and has a data type. A variable's data type indicates how the data are stored and displayed, e.g.:
  - *integer*: a whole number; a number that has no fractional part, e.g., 3;
  - *floating-point*: a number with a decimal point, e.g. 3.5 or  $1.2 \cdot 10^{-16}$  (this stands for  $1.2 \cdot 10^{-16}$ );
  - *character*: a standardized (using ASCII standard), readable text character (can be part of an ASCII text file).
- A *command* is an instruction to manipulate the contents of variables (e.g.  $y = x+2$ , meaning assign the value of  $x+2$  to the variable  $y$ ).
- An *algorithm* is a sequence of instructions for the solution of a specific problem.
- A *program* is a sequence of commands (implementing the algorithm) to do a specific task: solving a problem on a computer. When a program is executed, the variables represent a type of real data. That is why the same program can process different sets of data.
- A *bug* is an error in a program: the program does not solve your problem but another problem. The error can cause the program to stop running or provide wrong results.
- *Debugging* is the process of finding and removing bugs.

- A *file* is a collection of data or information that has a name: (e.g. data-, program-, or text-files).
- A *binary* file is a file stored in bit format (not meant to be read by humans). Most numeric data and all executable programs are stored in binary files. MATLAB binary files are those with the file-extension '`*.mat`' as opposed to:
- An *ASCII file* is a file stored in byte format (using human readable ASCII standard), such as a `*.m`-file.

## 1.2 Starting MATLAB

Before you start up MATLAB under Windows you must create your own directory `H:/MATLAB`. So, change your current directory with the `cd` command in `H:/MATLAB`. Now you can start MATLAB by double clicking on the MATLAB icon that should be on the desktop of your computer. Running MATLAB creates one or more windows on your screen. The most important window is the Command Window, which is the place you interact with MATLAB, i.e. it is used to enter commands and display text results. The string `>>` is the MATLAB prompt (or `EDU>>` for the Student Edition). When the Command Window is active, a cursor appears after the prompt, indicating that MATLAB is waiting for your command. MATLAB responds by writing text in the Command Window or by creating a Figure Window for graphics. To exit MATLAB use the command `exit` or `quit`.

## 1.3 Input and output via the Command Window

MATLAB is an interactive system; commands are executed immediately when typed in the command window and followed by Enter. The results are, if desired, displayed on screen. However, execution of a command will only be possible if the command is typed according to the rules (according to the MATLAB syntax).

Table 1 shows a list of commands used to solve indicated mathematical problems (`a`, `b`, `x` and `y` represent 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 in the Command Window immediately. Try the following:

```
>> 3 + 7.5
>> 18 / 4
>> 3 * 7
```

Note that additional spaces are not important in MATLAB.

The result of the last performed computation is assigned to the variable `ans`, which is an example of a MATLAB built-in variable. For instance:

```
>> 14/4
ans =
    3.5000
>> ans^(-6)
ans =
    5.4399e-004
```

Note that `5.4399e-004` is the **notation** that MATLAB uses to **represent the contents of the variable** `ans` on the screen. In fact it is not the exact contents of the variable (see 1.1 Key concepts). `0.000543991` is another notation for the contents of `ans`. Also note that the contents of `ans=3.5` is overwritten with `5.4399e-004` without any warning.

By default, MATLAB displays only 5 digits. The command `format long` increases this number to 15: `format short` reduces it to 5 again. Try:

```
>> help format
>> 312/56
ans =
    5.5714
>> format long
>> ans
ans =
    5.57142857142857
```



You can also define your own variables:

```
>> a = 14/4          % Text after % until the end of a line is treated as comment.
a =
    3.5000
>> b = a^(-6);       % The semicolon ';' suppresses writing the result of a command.
>> b
b =
    5.4399e-004
```

You will learn more on MATLAB variables in section 2.2.

You will notice that some examples in this text are followed by comments. Use comments in your own programs to explain the task of the command or group of commands in your program.

It is possible to enter more than one command on the same line. The separate commands should then be separated by a comma (to display the output) or by a semicolon (to suppress the display of output), e.g.:

```
>> sin(pi/4), cos(pi); sin(0)
ans =
    0.70710678118655
ans =
    0
```

The output may contain some empty lines; this can be suppressed by the command `format compact`. In contrast, the command `format loose` will insert extra empty lines.

To enter a single statement using two lines, use three periods `'...'` followed by Enter or Return. For instance:

```
>> sin(1) + sin(2) - sin(3) + sin(4) - sin(5) + sin(6) - ...
sin(8) + sin(9) - sin(10) + sin(11) - sin(12)
ans =
    1.03571874568747
```

MATLAB is case sensitive, for example, `a` is written as `a` in MATLAB ; using `A` will then result in an error or display the contents of another variable.

Previous commands can be fetched with the  $\uparrow$ -key. The command can also be changed: the  $\leftarrow$  and  $\rightarrow$ -keys may be used to move around in a line and edit it. In case of a long line, `Ctrl-a` and `Ctrl-e` may be useful: they allow to move the cursor to the beginning or the end of the line, respectively. To recall the most recent command starting from e.g. `c`, type `c` at the command prompt followed by the  $\uparrow$ -key. Similarly, `cos` followed by the  $\uparrow$ -key will find the last command starting from `cos`.

Since MATLAB executes the command immediately, it might be useful to have an idea of the expected outcome. You might be surprised how long it takes to print out a 1000 x 1000 matrix!

#### 1.4 Interrupting a command or program

Sometimes you might spot an error in your command or program. Due to this error it can happen that the command or program does not stop. Pressing `Ctrl-Break` (or `Ctrl-C` on a UNIX machine) forces MATLAB to stop the process. Sometimes, however, you may need to press a few times. After this, the MATLAB prompt (`>>`) re-appears. This may take a while, though.

Mathematical notation	MATLAB command
$a + b$	<code>a+b</code>
$a - b$	<code>a-b</code>
$ab$	<code>a*b</code>
$a/b$	<code>a/b</code> or <code>b\ a</code>
$x^b$	<code>x^b</code>
$\sqrt{x}$	<code>sqrt(x)</code> or <code>x^0.5</code>
$ x $	<code>abs(x)</code>
$\pi$	<code>pi</code>

$4 \cdot 10^3$	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)
$\sin(x), \arctan(x)$	sin(x), atan(x)

**Table 1,** Translation of mathematical notation to MATLAB commands.

## 1.5 Help-facilities

MATLAB provides assistance through extensive help. The `help` command is the simplest way to get help. It displays the list of all possible topics. To get a more general introduction to help, try:

```
>> help help
```

If you already know the topic or command, you can ask for a more specific help. For instance:

```
>> help ops
```

gives information on the operators and specific characters in MATLAB. The topics you want help on must be exact and spelled correctly. The `'lookfor'` command is more useful if you do not know the exact name of the command or topic. For example:

```
>> lookfor inverse
```

Displays a list of commands, with a short description, for which the word `inverse` is included in its help text. You can also use an incomplete name, e.g. `lookfor inv`. Beside the `help` and `lookfor` commands, there is also a separate mouse driven help. The `helpwin` command opens a new window on screen which can be browsed in an interactive way.

### Exercise 1

- Is the inverse cosine function, known as  $\cos^{-1}$  or  $\arccos$ , one of the MATLAB's elementary functions?
- Does MATLAB have a mathematical function to calculate the greatest common divisor?
- Look for information on logarithms.

Use `help` or `lookfor` to find out. ■

## 1.6 Introduction to the Workspace

If you work in the Command Window, MATLAB memorizes all commands that you entered and all variables that you created. These commands and variables are said to reside in the MATLAB Workspace. They might be easily recalled when needed, e.g. to recall previous commands, the `'-'` key is used. Variables can be verified with the commands `who`, that gives a list of variables present in the workspace, and `whos`, that includes also information on name, number of allocated bytes and class of variables. For example, assuming that you performed all commands from section 1.1, after typing `who` you should get the following information:

```
>> who
Your variables are:
a      ans      b      x
```

The command `clear <name>` deletes the variable `<name>` from the MATLAB workspace, `clear` or `clear all` removes all variables. This is useful when starting a new exercise. For example:

```
>> clear a x
>> who
Your variables are:
ans      b
```

### 1.6.1 Path

In MATLAB, commands or programs are contained in m-files, which are just plain text files and have an extension `‘.m’` (see Chapter 3). The m-files must be located in one of the directories which MATLAB automatically searches. The list of these directories can be listed by the command `path`. One of the directories that is always taken into account is the current working directory, which can be identified by the command `pwd`. Use the `path`, `addpath` and `rmpath` functions to modify the path. It is also possible to access the path browser from the File menu-bar, instead.

#### Exercise 2

Type `path` to check which directories are placed on your path. If you didn't before you have to create your own directory `H:/MATLAB` with the `cd` command. ■

### 1.6.2 Saving and loading Workspace variables

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.

#### Exercise 3

The command `save` allows saving your workspace variables into either a binary file or an ASCII file (see 1.1 Key concepts on binary and ASCII files). Binary files automatically get the `‘.mat’` extension, which is not true for ASCII files. It is recommended, however, to add a `‘.txt’` or `‘.dat’` extension to the latter.

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 ascii format to allcos.dat ■
```

The `load` command allows for loading variables into the workspace. It uses the same syntax as `save`.

#### Exercise 4

Having done the previous exercise, 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 the specified variables from the file data.mat
>> load Matlab str data           % loads only str from the file matlab.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 all variables from the file allcos.dat
>> load Matlab data s1 c1         % loads specified variables from the file matlab.mat
■
```

## 2 Basic syntax and variables in MATLAB

You have already got some experience with MATLAB and you know it can be used as a calculator. This chapter discusses in further detail the syntax of MATLAB -commands with which you can manipulate variables.

## 2.1 MATLAB as a calculator

There are three kinds of numbers used in MATLAB: integers, real numbers and complex numbers. In addition, MATLAB has representations of the non-numbers: `Inf`, for positive infinity, generated e.g. by `1/0`, and `NaN`, Not-a-Number, obtained as a result of the mathematically undefined operations such as `0/0` or  $\infty - \infty$ .

MATLAB has six basic arithmetic operations, such as: `+`, `-`, `*`, `/` and `\` (right and left divisions) and `^` (involution, or raising to a power). Note that the two division operators are different:

```
>> 19\3, 3/19      % mathematically both 3/19
ans =
    0.1579
ans =
    0.1579
```

Basic built-in functions, trigonometric, exponential, etc., are available for a user. Try `help elfun` to get the list of elementary functions.

### Exercise 5

Modify the format in MATLAB such that empty lines are suppressed and the output is given with 15 digits. Calculate:

```
>> pi
>> sin(pi)
```

Note that the answer is not exactly 0: why?. Use the command `format` to put MATLAB in its standard-format. ■

## 2.2 Assignments and variables

Working with complex numbers is easily done with MATLAB.

### Exercise 6

Choose two complex numbers, for example  $-3+2i$  and  $5-7i$ . Add, subtract, multiply, and divide these two numbers. ■

During this exercise, the complex numbers had to be typed four times. To reduce this, assign each number to a variable. For the previous exercise, this results in:

```
>> z = -3 + 2*i; w = 5 - 7*i
>> y1 = z + w; y2 = z - w;
>> y3 = z * w;
>> y4 = z / w; y5 = w \ z;
```

Formally, there is no need to declare (i.e. define the name, size and the type of) a new variable in MATLAB. A variable is simply created through an assignment (see key concepts), for example:

```
>> x = pi / 3 ; f = x^2 + 4 * sin(x)
```

Each newly created numerical variable is always of the type `double`, i.e., the container size is 64 bit by default: real numbers are approximated with the highest possible precision (see `help eps`).

You can change this type by converting it into, e.g., the single type with `a = single(a)`. In some cases, when huge matrices should be handled and precision is not very important, this might be a way to proceed. Also, when only integers are taken into consideration, it might be useful to convert the double representation into e.g. `int32` integer type with `a = int32(a)`.

Note that integer numbers are represented exactly within a certain range, no matter which numeric type is used, as long as the number can be represented in the number of bits used in the numeric type for instance: a byte is a unit of 8 bits. It can hold  $2^8=256$  values.

Bear in mind that undefined values cannot be assigned to variables. So, the following is not possible:

```
>> clear x      % to make sure that x does not exist
>> f = x^2 + 4 * sin(x)
```

Variable names (identifiers) begin with a letter, followed by letters, numbers or underscores. Only 31 characters may be used for MATLAB identifiers.

## Exercise 7

Here are some examples of different types of MATLAB variables. You do not need to understand them all now, since you will learn more about them during the course. Create them manually in MATLAB:

```
>> this_is_my_very_simple_variable_today = 5    % what happens?
>> 2t = 8                                       % what is the problem with this command?
>> M = [1 2; 3 4; 5 6]                         % a matrix
>> c = 'E'                                     % a character
>> str = 'Hello world'                         % a string
>> m = ['J', 'o', 'h', 'n']                   % try to guess what it is
```

Check the types by using the command `whos`. Use `clear <name>` to remove a variable from the workspace. ■

As you already know, a MATLAB variable is created by an assignment if it does not exist in the workspace; if it already existed, its previous value is lost. For example:

```
>> clear;
>> b = 5; c = 7;
>> a = min(b,c);                               % create a as the minimum of b and c
>> whos
  Name      Size      Bytes  Class
  a         1x1         8  double array
  b         1x1         8  double array
  c         1x1         8  double array
Grand total is 3 elements using 24 bytes
>> a='hello';
>> whos
  Name      Size      Bytes  Class
  a         1x5        10  char array
  b         1x1         8  double array
  c         1x1         8  double array
Grand total is 7 elements using 26 bytes
```

In the previous example the function `min` is called with `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 (you will learn more on how to build your own functions in section 3.2).

**Important:** do not use variable names which are defined as function names (e.g., `mean` or `error`). If you are going to use a suspicious variable name, use `help <name>` to find out whether the function already exists or not.

### 2.3 MATLAB syntax and semantics issues

A natural language is so called because it evolves naturally: it is whatever the members of a human community ‘agree’ to use for communication among themselves. It might well be rich in ambiguities. This is tolerable because listeners and readers are also intelligent. On the other hand, a computer language must be simple and direct, since it will be used to instruct an uncomprehending machine (no unkind remark to MATLAB).

In ordinary life there are many things that we wish to specify, such as engineering designs and knitting patterns. Scientists concerned with the design, implementation and use of algorithms (computer languages) – i.e., all scientists – should be acquainted with the principal methods of specifying a language formally: the theme of this section.

Every language has its syntax and semantics. Syntax is concerned with form, semantics with meaning. Each language has a vocabulary of symbols (e.g., `=`, `+`, `>`), and rules (the grammar) for how these symbols may be put together to form phrases: expressions, commands and programs:

- The meaning of an expression is a computation that **yields a value** (e.g. `a + 2` might yield 202).

- The meaning of a command is a computation that **manipulates variables** (e.g.  $b = a + 2$  might **change** the value of  $b$  to 202).

MATLAB automatically prepares (translates) commands for executing them on a computer. This is exactly the motivation to use MATLAB and not try to create your own Pentium or other processor-specific code. This section explains the fundamental idea of syntax: how to write well-formed phrases in the MATLAB language which can be understood (interpreted) by MATLAB.

### 2.3.1 Syntax trees and logical data type

Consider the phrase '**a&b-c**'. This phrase consists of a number of sub-phrases. Without rules the decomposition into sub-phrases is not obvious. We can make the phrase structure explicit by means of syntax trees. Each phrase corresponds to a tree. Branching represents the subdivision of phrases into subphrases. Without precedence rules it is not obvious which decomposition of '**a&b-c**' is meant by the programmer.

Correct decomposition	Incorrect decomposition
<p>First calculate <math>b - c</math>, yielding a numeric value. Next, calculate the logical expressions <math>(a \&amp; (b - c))</math>. Verify that:  <math>1 \&amp; 3 - 3 = 0</math>                      <math>1 \&amp; 0 - 1 = 1</math>  And not:  <math>1 \&amp; 3 - 3 = -2</math>                      <math>1 \&amp; 0 - 1 = -1</math>  Note that that not only <math>1 == \text{TRUE}</math> but also <math>3 == \text{TRUE}</math>, and <math>-1 == \text{TRUE}</math>.</p>	<p>First calculate the logical expressions <math>(a \&amp; b)</math>: which yields a logical value: <b>TRUE</b> or <b>FALSE</b>. In MATLAB the result of a logical expression is 1 if it is true and 0 if it is false. Next, calculate <math>(a \&amp; b) - c</math>: which yields a numeric value. In this case <math>(1 - c)</math> or <math>(0 - c)</math>.</p>

The previous example introduced two kinds of expressions. Truth-valued expressions and number-valued expressions.

Production rules specify how commands, expressions and numerals are formed. First there are the complete commands (such as  $a = 1 + 2$ ). Then there are expressions (such as  $1 + 2$ ). Then there are numerals (such as 2).

The execution precedence of the arithmetic, relational and logical operators in expressions is:

- *involution*: ^
- *multiplication*: \* or *division*: / or: \
- *addition*: + or *subtraction*: - *greater than*: > or *not less than*: >= or *equal*: == or *not greater than*: <= or *less than*: <
- *logical AND*: & or *logical OR*: | or *logical complement NOT*: ~

Input A	Input B	~A (NOT)	A&B (AND)	A B (OR)
0	0	1	0	0
0	any value ~= 0	1	0	1
any value ~= 0	0	0	0	1
any value ~= 0	any value ~= 0	0	1	1

**Table 2,** logical operators.

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

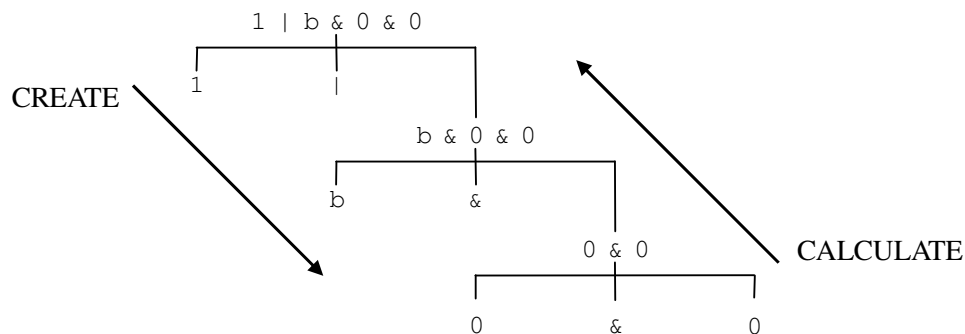
**Table 3,** Relational and logical operators

**Note:** Be careful with implementing the *equal* command using ==. In the computer the value of pi is approximated. Knowing this limitation easily provides the solution (see help eps).

```
>> sin(pi) == 0
ans =
    0
>> abs( sin(pi) ) <= eps
ans =
    1
```

Note: The logical & and | have 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 =
    1
```



In case of ambiguity the association of operators is from left to right: creating the tree. After building the tree first (0 & 0) is evaluated yielding 0. Subsequently (b & 0) is evaluated, yielding 0 regardless the value of b. Finally 1 | (b & 0 & 0) is evaluated. We now see that this expression always will yield 1 regardless the value of b.

$(1 \text{ or } (b \& (0 \& 0))) = 1 \text{ or } (b \& 0) = (1 \text{ or } 0) = 1$

This shows that you should always use parentheses to indicate in which way you want MATLAB to understand your expression in case of ambiguity.

The representation of the logical data type using logical 0 and 1 values can be used for indexing. You can see the differences between the use of logical and non-logical values by executing the following commands that attempt to extract the elements of y that correspond to either the odd or the even elements of x:

```
>> x = [9 8 7 6 5] % a vector with five values
>> use = logical([1 2 3 4 0]) % a vector with five truth-values
use = % 0 is FALSE and > 0 denotes TRUE
     1  1  1  1  0
>> x(use) % select only those indices for which use is TRUE
ans =
     9  8  7  6
```

```
>> use = [1 2 3 4 0]      % If we try this then Matlab tries to create a new
>> x(use)                 % vector with the first, second, ... , zeroth element
                           % However, the zeroth element does not exist
??? Index into matrix is negative or zero. See release notes on changes to
logical indices.
```

To select all odd elements from a vector  $x$  we can now simply compute (see `help rem`):

```
>> x( logical(rem(x,2)) )    % remainder after division by 2
```

## ✎ Exercise 8

Let  $x = [3 \ 16 \ 9 \ 12 \ -1 \ 0 \ -12 \ 9 \ 6 \ 1]$  and  $y = [3 \ 5 \ 6 \ 1 \ 8 \ 2 \ 9 \ 4 \ 0 \ 7]$ . The exercises here show the techniques of logical-indexing. Execute and interpret the results of the following commands:

<code>x(x&gt;5)</code>	<code>(x&gt;3) &amp; (x&lt;8)</code>	• set the positive values of $x$ to zero
<code>y(x&lt;=4)</code>	<code>x((x&lt;2)   (x&gt;=8))</code>	• set values of $x$ that are multiples of 3 to 3 (use <code>rem</code> )
<code>x(y&lt;0)</code>	<code>y((x&lt;2)   (x&gt;=8))</code>	• extract the values of $x$ that are $> 10$ ■

## ✎ Exercise 9

Evaluate the following expressions by hand and use MATLAB to check the answers. Note the difference between the left and the right divisors. Use `help` to learn more on commands rounding numbers, such as `round`, `floor` and `ceil`.

<code>2/2*3</code>	<code>3^2/4</code>
<code>8*5\4</code>	<code>3^2^3</code>
<code>8*(5\4)</code>	<code>2+round(6/9+3*2)/2</code>
<code>7-5*4\9</code>	<code>2+floor(6/9+3*2)/2</code>
<code>6-2/5+7^2-1</code>	<code>2+ceil(6/9+3*2)/2</code>
<code>10/2\5-3+2*4</code>	<code>x=pi/3, x=x-1, x=x+5, x=abs(x)/x</code> ■

Before moving on, check whether you now understand the following relations:

```
>> a = randperm(10);      % random permutations
>> b = 1:10;
>> b - (a <= 7)           % subtract from b taking 1 for a<=7 and 0 otherwise
>> (a >= 2) & (a < 4)     % ones at positions where 2 <= a < 4
>> ~(b > 4)               % ones at positions where b <= 4
>> (a == b) | b == 3       % ones at positions where a == b or b == 3
>> any(a > 5)              % 1 when ANY of the a elements are larger than 5
>> any(b < 5 & a > 8)     % 1 when in the evaluated expression is at least one 1
>> all(b > 2)              % 1 when ALL b-elements are larger than 2
```

## 3 MATLAB programs: script and function m-files

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 you want to change values of some variable and re-evaluate them quickly.

### 3.1 Script m-files

Formally, a script file is an external file that contains a sequence of MATLAB commands: a MATLAB program. When you run a script, the commands in it are executed as if they have been executed from the command line.

The sequence of commands in a script is executed using the file-name (without the extension) at the MATLAB command line. It is also possible to execute script files from within other MATLAB programs.

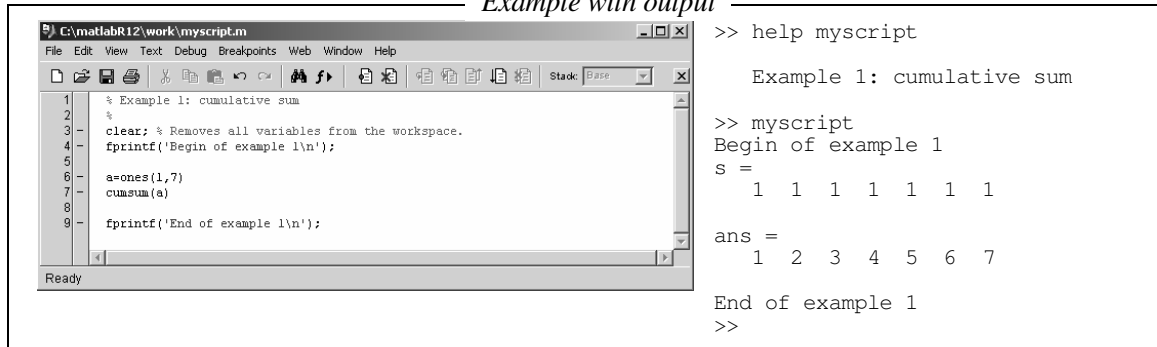
In the following example the `fprintf` function is used to write data (e.g. a string) to the display. Special format characters can be used such as `'\n'` for a linefeed or `'\t'` to jump to the next tabulator (fixed horizontal location).



### Syntax

```
% The text after the percentage sign at the lines before  
% the bracket will appear when you type help 'name of script m-file'.  
[  
C1; % The script-body contains a collection of subcommands.
```

### Example with output



The screenshot shows the MATLAB editor window with a script named 'myscript.m'. The script contains the following code:

```
1 % Example 1: cumulative sum  
2 %  
3 clear; % Removes all variables from the workspace.  
4 fprintf('Begin of example 1\n');  
5  
6 a=ones(1,7)  
7 cumsum(a)  
8  
9 fprintf('End of example 1\n');
```

The output of the script is shown in the command window:

```
>> help myscript  
Example 1: cumulative sum  
  
>> myscript  
Begin of example 1  
s =  
    1    1    1    1    1    1    1  
  
ans =  
    1    2    3    4    5    6    7  
  
End of example 1  
>>
```

**Note:** MATLAB variables, names, script names and function names may neither contain spaces nor they may start with a number!

### Exercise 10

Create a script: open the MATLAB editor (go to the File menu-bar, choose the New option and then M-file; the MATLAB Editor Window will appear), enter the lines listed below and save these lines in a M-file named `sinplot.m`.

```
x = 0:0.2:6;  
y = sin(x);  
plot(x,y);  
title('Plot of y = sin(x)');
```

and then run it by:

```
>> sinplot
```

The `sinplot` script affects the workspace. Check:

```
>> clear % all variables are removed from the workspace  
>> who % no variables present  
>> sinplot  
>> who ■
```

These generic names, `x` and `y`, may be easily used in further computations and this can cause side effects. Side effects occur in general when a set of commands change variables other than the input arguments. Since scripts create and change variables in the workspace (without warning), a bug, hard to track down, may easily occur.

So it is important to remember that the commands within a script have access to all variables in the workspace and all variables created in a script become part of the workspace (think of the `clear` command in a script). Therefore it is **better** to use function m-files when there is a specific problem to be solved.

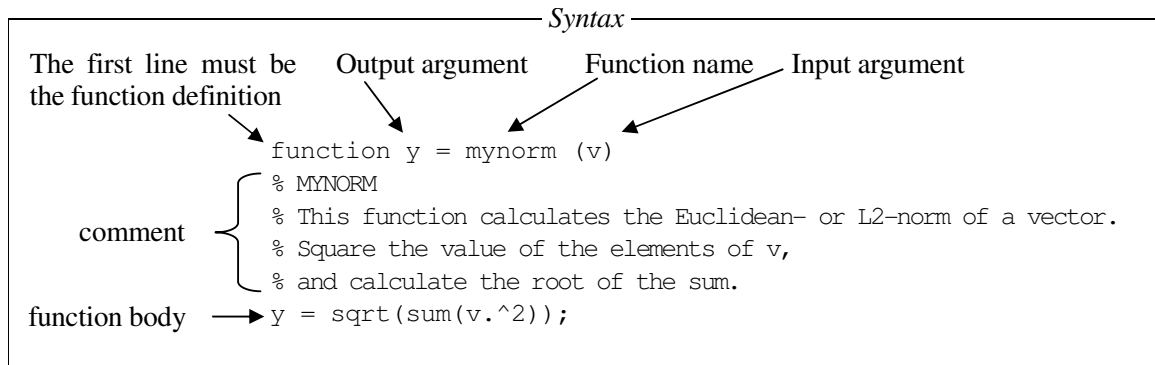
### Exercise 11

Create a script which plots (see `help plot` where you can omit the parameter `s`), in the same figure, the functions  $y_1 = 2^{\cos(x)}$ ,  $y_2 = 2^{\sin(x)}$  ( $y_1 = 2.^{\cos(x)}$ , the operator `.` means element-by-element operation see chapter 4) in the range  $[-10,10]$ . Choose a sufficient length for `x` in order to obtain a smooth plot. Add a title, legend and axis descriptions (see chapter 7 for title) ■

### 3.2 Function m-files

New functions can be added to MATLAB through function m-files. Through the use of input and output-variables it is possible to communicate with other files. Variables defined and used inside a function, other than the inputs/ outputs, are invisible to other functions and the command environment.

The name of the file is the name of the function which can be used to call the function from an other file or command-line.



Try entering these commands in an M-file called `mynorm.m`. This function accepts a single input argument and returns a single output argument. To call the `mynorm` function, enter:

```
in = [2 3 6];
mynorm(in)
ans =
    7
```

**note:** Variables inside a function, are local, and only known within the function.

```
function y=fname(x)
a=3;      % a exists only within (the context of) this function.
y=a*x;
```

#### Exercise 12

Create the function `average` and store it on disk as `average.m`. Remember about the comment lines. Check its usability by calling:

```
help average
avr1 = average(1:10);
```

■

**Note:** more than one output arguments are enclosed in `[ ]`.

```
function [avr,sd] = mystat(x)
% STAT Simple statistics.
% Computes the average value and the standard deviation of a vector x.
n = length(x);
avr = sum(x)/n;
sd = sqrt(sum((x-avr).^2)/(n-1))
return;
```

**Warning:** The functions `mean` and `std` already exist in MATLAB. As long as a function name is used as variable name, MATLAB cannot perform that function. Many other, easily appealing names, such as `sum` or `prod` are reserved by MATLAB functions, so be careful when choosing your names. Hint: personalize your variables (e.g. `tu_mean`).

When controlling the proper use of parameters use the function `error`. It displays an error message, stops function execution and returns to the command environment.

```
function y = mytest(x)
% COMMENT
if (x >= 1)
    error ('x must be smaller than 1');
```

```

end
% function body
return;

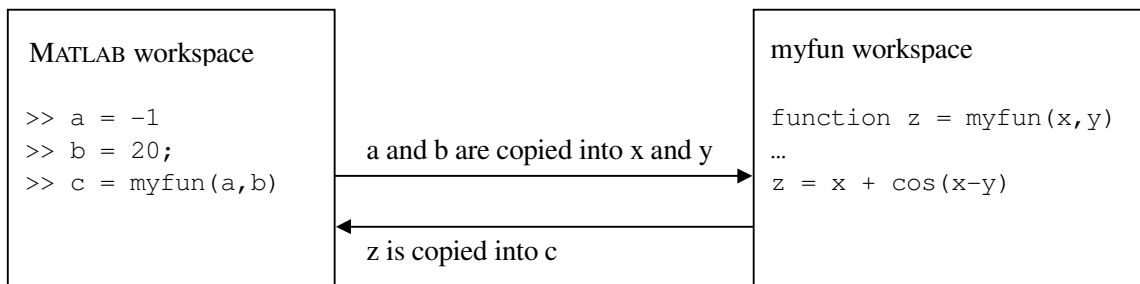
```

### 3.3 Workspace issues

The workspace is an environment at a certain time. If you design a function, you create your own workspace at the time of execution.

#### 3.3.1 Local and global variables

Each m-file function has access to a part of memory separate from MATLAB's workspace. This is called the Function Workspace. This means that each m-file function has its own local variables, which are separate from those of other functions and from the (general) workspace. To understand it better analyze the following diagram:



At the time of executing the `myfun` function in the MATLAB workspace, variables `a` and `b` are available. The variables `a` and `b` are copied to the variables `x` and `y` via the function parameters `x` and `y` of the function. The variables `x`, `y` and `z` are visible only in the function `myfun`. The function result available in `z` is copied into the newly created variable `c` in the MATLAB workspace.

#### 3.3.2 Special function variables

Each function has two internal variables: `nargin` - the number of function input arguments that were used to call the function and `nargout` - the number of output arguments. Analyze the following function:

```

function [out1] = checkarg (in1,in2,in3)
% CHECKARG Demo on using the nargin and nargout variables.
if (nargin ~= 3)
    disp('wrong number of arguments');
    return; % leave the function early
end
s = in1 + in2 + in3;
if (nargout == 0)
    return; % leave the function early
end
out1 = s;

```

### 3.4 Scripts vs. functions

The most important difference between a script and a function is that all script variables are externally accessible (i.e. in the MATLAB workspace), whereas function variables are not. Therefore, a script is a good tool for documenting work, designing experiments and testing. In general, create a function to solve a given problem for an arbitrary value of the parameters. When using a function, only the input and output variables are shared in the MATLAB workspace: this makes it possible to create your own function workspace hiding the variables in a function from the MATLAB workspace. Use a script to run functions for specific parameters required by the assignment.

## 4 Mathematics with vectors and matrices in MATLAB

The basic element of MATLAB is a **matrix** with dimensions  $n \times k$  (number of rows  $\times$  number of columns) (see section 4.2). Special cases are:

- If  $n = k = 1$  the element is a **scalar** (a single number)
- For  $k = 1$  the element is a **column vector**; for  $n = 1$  this is a **row vector**.

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

Dimensions of matrices do not have to be entered explicitly. They are derived from the input.

### 4.1 Vectors

Row vectors are lists of numbers separated either by commas or by spaces. They are examples of simple arrays. Their entries are referred to as elements or components. The entries must be enclosed in brackets `[ ]`. The number of entries is known as the length of the vector (the command `length` exists as well).

```
>> v = [-1 sin(3) 7]      % ALSO v = [-1, sin(3), 7]
v =
   -1.0000    0.1411    7.0000
>> length(v)
ans =
     3
>> v(2)                  % between parentheses(): indices
ans =
    0.1411
```

**Note:** The first element `v(1) = -1` has `index=1`; use `index>0` because `v(0)` does not exist!

A vector can be multiplied by a scalar, or added to (or subtracted from) another vector with the same length, or a (scalar) number can be added to (or subtracted from) every component of a vector. All these operations are carried out element by element. Vectors can also be built from 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]           % note the use of brackets
t =
    -2     4    14    -2    -3    -4
! >> u = 2*v -w
u =
    -4     1    10        % note difference with previous example!
>>
```

A particular element of a vector can be manipulated as well.

```
>> v(2) = -1              % change the 2nd element of v
v =
   -1   -1     7
```

#### 4.1.1 Colon notation and extracting parts of a vector

The colon (`:`) notation provides an important shortcut, used when producing row vectors (see Table 4 and `help colon`):

```
>> 2:5                    % Start:Step:End or Start:End
ans =
     2     3     4     5
```

```
>> -2:3                                % Equal to -2:1:3
ans =
    -2    -1     0     1     2     3
```

In general, `first:step:last` produces a vector of elements with as first value `first`, incrementing every next element by `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    % a negative step is also possible
ans =
    1.5000    1.0000    0.5000         0   -0.5000
>>
```

Parts of a vector can be extracted by using a colon notation: the indices are enclosed in parentheses `()`:

```
>> r = [-1:2:6 2 3, -2] % Note inconsistent use of commas
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 positions 1, 3 and 5
ans =
    -1     3     2
```

A very useful concept in Matlab is demonstrated in the following command:

```
>> r(5:-1:2)              % what will you get here?
```

For example `x(1:2:end)=x(1:2:end)+11` implements  $x_i+11$  for  $i=1, 3, \dots$

#### 4.1.2 Column vectors and transposing

To create a column vector, you should separate entries by new lines or by a semicolon `(;)`. An alternative is by transposing a row vector:

```
>> z = [1
7
7]
z =
     1
     7
     7
>> u = [-1; 3; 5]
u =
    -1
     3
     5
```

The same operations as on row vectors can be performed on column vectors. However, you cannot, for example add a column vector to a row vector. To do that, you need an operation called **transposing**, which converts a column vector into a row vector and vice versa:

```
>> u'                    % u is a column vector and u' is a row vector
ans =
    -1     3     5
>> v = [-1 2 7];         % v is a row vector
>> u + v                  % you cannot add a column vector u to a row vector v
??? Error using ==> +
Matrix dimensions must agree.
```

```
>> u' + v          % row vector
ans =
    -2     5    12
>> u + v'          % column vector
ans =
    -2
     5
    12
```

If  $z$  is a **complex** vector, then  $z'$  gives the **conjugate transpose** of  $z$ , e.g.:

```
>> z = [1+2i, -1+i]
z =
    1.0000 + 2.0000i   -1.0000 + 1.0000i
>> z'                % this is the conjugate transpose
ans =
    1.0000 - 2.0000i
   -1.0000 - 1.0000i
>> z.'               % this is the traditional transpose
ans =
    1.0000 + 2.0000i
   -1.0000 + 1.0000i
```

### 4.1.3 Product, division and powers of vectors

You can now compute the inner product between two vectors  $x$  and  $y$  of the same length, in a simple way:

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

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, thus, the corresponding elements of two vectors are multiplied. For instance:

```
>> u .* v % this is an element-by-element multiplication, of similar vectors
ans =
     1
     6
    35
>> sum(u.*v) % this is 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
Note that:
>> u*z
ans =
    -4    -3    -1
    12     9     3
    20    15     5
```

You can now easily **tabulate** 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. In MATLAB, however, the operator `./` is defined to perform an element-by-element division. It is, therefore, defined (only) 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 also be 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
```

### Exercise 13

Get acquainted with operating on row and column vectors. Perform, for instance:

- Create a vector consisting of the even numbers between 21 and 47.
- Let  $x = [4 \ 5 \ 9 \ 6]$ .
  - Subtract 3 from each element
  - Add 11 to the odd-index elements.
  - Compute the square root of each element.
  - Raise each element to the power 3.
- Given a vector  $t$ , write down the MATLAB expressions that will compute:
  - $\ln(2 + t + t^2)$
  - $\cos^2(t) - \sin^2(t)$
  - $e^t(1 + \cos(3t))$
  - $\tan^{-1}(t)$

Test them for  $t = 1 : 0.2 : 2$ .

Given  $x = [2 \ 1 \ 3 \ 7 \ 9 \ 4 \ 6]$ , explain what the following commands do (note that  $x(\text{end})$  points to the last element of  $x$ ):

$x(3)$	$x(2:2:6)$	$\text{sum}(x)$
$x(1:7)$	$x(6:-2:1)$	$\text{mean}(x)$
$x(1:\text{end}-1)$	$x(\text{end}-2:3:2)$	$\text{min}(x)$

Let  $x = [1 + 3i, 2 - 2i]$  be a complex vector. Check the following expressions:

$x'$	$x * x'$
$x.''$	$x * x.'$

■

#### Exercise 14

Use the knowledge on computing the inner product to find the Euclidean length of the vector

$x = [2 \ 1 \ 3 \ 7 \ 9 \ 4 \ 6]$ .

The angle between two column vectors is defined (by the 'cosine rule') as:  $\cos \alpha = \frac{x^T y}{\|x\| \|y\|}$ .

Compute the angle between the vectors (note the difference between  $[3 \ 2 \ 1]$  and  $[3;2;1]$ ):

$x = [3;2;1]$  and  $y = [1;2;3]$

$x = [1:5]$  and  $y = [6:10]$  ■

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

**Table 4.** Manipulation of (groups of) matrix elements

MATLAB Command	Result
$\sin(4)$	$\sin(4)$ is approximately -0.7568
$\text{Sin } 4$	$\sin 4 = \sin(52) = 0.9866$ : because 4 is represented by ASCII 52.
$x = \text{rank}(A)$	$x$ becomes the rank of matrix $A$
$x = \text{det}(A)$	$x$ becomes the determinant of matrix $A$
$x = \text{size}(A)$	$x$ becomes a row-vector with 2 elements: the number of rows and columns of $A$
$x = \text{trace}(A)$	$x$ becomes the trace (sum of diagonal elements) of matrix $A$
$x = \text{norm}(v)$	$x$ becomes the Euclidean length of vector $v$

MATLAB Command	Result
$C = A + B, C = A - B$	addition, subtraction of two (similar) matrices
$C = A * B$	multiplication of two matrices
$C = A .* B$	'element-by-element' multiplication ( $A$ and $B$ are similar)
$C = A ^ B$	power of a matrix (with $B$ scalar: can also be used for $A^{-1}$ ).
$C = A .^ k$	'element-by-element' power of a matrix; $k$ is an integer
$C = A'$	the transpose of a matrix: $A^T$
$C = A ./ B$	'element-by-element' division ( $A$ and $B$ are similar), reciprocal: $y = 1 ./ x$
$X = A \setminus B$	finds the solution in the least squares sense to the system of equations $AX=B$
$X = B / A$	finds the solution of $XA = B$ , analogous to the previous command



MATLAB Command	Result
<code>C = inv(A)</code>	C is the inverse of A
<code>L = eig(A)</code>	L is a vector containing the (possibly complex) eigenvalues of a square matrix A
<code>[X,D] = eig(A)</code>	produces a diagonal matrix D of eigenvalues and a full matrix X whose columns are the corresponding eigenvectors of A
<code>S = svd(A)</code>	S is a vector containing the singular values of A
<code>[U,S,V] = svd(A)</code>	S is a diagonal matrix with nonnegative diagonal elements in decreasing order; columns of U and V are the accompanying singular vectors

MATLAB Command	Result
<code>x= linspace(a,b,n)</code>	x is a vector of n equally spaced points between (and including) a and b
<code>x= logspace(a,b,n)</code>	generates a vector x starting at $10^a$ and ending at $10^b$ containing n values
<code>A = eye(n)</code>	A is an n x n identity matrix
<code>A = zeros(n,m)</code>	A is an n x m matrix with zeros (default m = n)
<code>A = ones(n,m)</code>	A is an n x m matrix with ones (default m = n)
<code>A = diag(v)</code>	A is an n x n matrix with elements $v_1, v_2, \dots, v_n$ on the diagonal
<code>X = tril(A)</code>	X is the lower triangular part of A
<code>X = triu(A)</code>	X is the upper triangular part of A
<code>A = rand(n,m)</code>	A is an n x m matrix with elements uniformly distributed between 0 and 1
<code>A = randn(n,m)</code>	ditto - with elements standard normal distributed

MATLAB Command	Result
<code>v = max(A)</code>	v is a <b>row</b> vector with the maximum value of the elements in each <b>column</b> of A (if A is a vector, v is the maximum of all elements)
<code>v=min(A), v=sum(A)</code>	ditto - with minimum, ditto - with sum

## 4.2 Matrices

Row and column vectors are special types of matrices. An n x k matrix is a rectangular array of variables 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. Examine the definition of the matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

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

Other examples are:

```
>> 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
```

As stated earlier (see 4.0), a row vector is a  $1 \times k$  matrix and a column vector is an  $n \times 1$  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 transposition interchanges rows with the corresponding columns, as in the example

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

#### 4.2.1 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(B'))          % a matrix of all zeros of the size given by B'
C =  0      0      0
     0      0      0
>> D = rand(2,3)               % a matrix of random numbers; you will get a different one!
D =
    0.9501    0.6068    0.8913
    0.2311    0.4860    0.7621
>> v = linspace(1,2,4)         % a vector is also an example of a matrix
v =
    1.0000    1.3333    1.6667    2.0000
```

## 4.2.2 Building matrices and extracting parts of matrices

It is often needed to build a larger matrix from 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 size 3 by 2; 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 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 extracted 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  $A_{ij}$ ; MATLAB provides the  $A(i,j)$  notation (syntax).

```

>> 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 impossible: 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 unspecified
               % elements are initialized to zero
A =
     1     2     3
     4     5     8
     7     8     9
     0     0     0
     0     5     0

```

Different parts of the matrix A can now be 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
     0
     5
>> A(1:2,:) % extract the 1st and 2nd rows of A
ans =
     1     2     3
     4     5     8
>> A([2,5],1:2) % extract a part of A
ans =
     4     5
     0     5

```

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

```

As you have seen in the examples above it is possible to manipulate (groups of) matrix-elements (variables). The commands are shortly explained in Table 4.

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
```

### ☞ Exercise 15

Clear all variables. Define the matrix: `A = [1:4; 5:8; 1,1,1,1]`. Predict and check the result of:

```
x = A(:,3)                Y = A(3:3,1:4)
B = A(1:3,2:2)            A = [A; 2 1 7 7; 7 7 4 5]
A(1,1) = 9 + A(2,3)       C = A([1 3],2)
A(2:3,1:3) = [0 0 0;0 0 0] D = A([2,3,5],[1,3,4])
A(2:3,1:2) = [1 1; 3 3]   D(2,:) = [] ■
```

### ☞ Exercise 16

Let `A = [2 7 9 7; 3 1 5 6; 8 1 2 5]`. Explain the results or perform the following commands:

<code>A'</code>	<code>sum(A)</code>	<code>max(min(A))</code>	
<code>A(1,:)'</code>	<code>sum(A)'</code>	<code>[[A;sum(A)][sum(A,2);sum(A(:))]]</code>	
<code>A(:,[1,4])</code>	<code>mean(A)</code>		• assign even-numbered columns of A to a matrix B
<code>A([2,3],[3,1])</code>	<code>mean(A')</code>		• assign the odd numbered rows to a matrix C
<code>reshape(A,2,6)</code>	<code>sum(A,2)</code>		• convert A into a 4-by-3 matrix
<code>A(:)</code>	<code>mean(A,2)</code>		• compute the reciprocal of each element of A
<code>flipud(A)</code>	<code>min(A)</code>		• compute the square-root of each element of A
<code>fliplr(A)</code>	<code>max(A')</code>		• remove the second column of A
<code>[A; A(end,:)]</code>	<code>min(A(:,4))</code>		• add a row of all 1's at the beginning and at the end
<code>[A; A(1:2,:)]</code>	<code>[min(A)',max(A)']</code>		• swap the 2 <sup>nd</sup> row with the last row ■

### 4.2.3 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. These 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 -3 3; 4 0 7]
B =
     1     -3     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      2      8
     6      1     13
>> B-2    % subtract 2 from all elements of B
ans =
     0      0      6
     4     -1     11
```

```

>> B./4 % divide all elements of the matrix B by 4
ans =
    0.5000    0.5000    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    2.0000    0.5000
    0.6667    4.0000    0.3077
>> C = [1 -1 4; 7 0 -1];
>> B .* C % multiply element-by-element
ans =
     2     -2    32
    42     0   -13
>> ans.^3 - 2 % do for all elements: raise to the power 3 and subtract 2
ans =
         6        -10       32766
    74086         -2       -2199
>> ans ./ B.^2 % element-by-element division
ans =
    1.0e+003 *
    0.0015   -0.0025    0.5120
    2.0579   -0.0020   -0.0130
>> r = [1 3 -2];
>> r * B2 % this is a legal operation: r is a 1-by-3 matrix and B2 is
          % a 3-by-2 matrix; B2 * r, however, is an illegal operation
ans =
     6     -7

```

Concerning the matrix-vector and matrix-matrix products, two things should be reminded. First, an  $n \times k$  matrix  $A$  (having row index  $i=1:n$  and column index  $j=1:k$ ) can be multiplied by a  $k \times 1$  (column) vector  $x$ , resulting in a  $n \times 1$  (column) vector  $y$ , i.e.:  $Ax = y$  such that:

$$y_i = \sum_{p=1}^k A_{ip} x_p$$

Multiplying a  $1 \times n$  (row) vector  $x$  by a matrix  $A$ , results in a  $1 \times k$  (row) vector:

$$y_j = \sum_{p=1}^n x_p A_{pj}$$

Secondly, an  $n \times k$  matrix  $A$  can be multiplied by a matrix  $B$ , only if  $B$  has  $k$  rows, i.e.  $B$  is  $k \times m$  (with  $m$  an arbitrary value). As a result, you get an  $n \times m$  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
>> 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

```

### Exercise 17

Given: vectors  $x = [1 \ 3 \ 7]$  and  $y = [2 \ 4 \ 2]$ , and matrices  $A = [3 \ 1 \ 6; 5 \ 2 \ 7]$  and  $B = [1 \ 4; 7 \ 8; 2 \ 2]$ , determine which of the following statements will correctly execute (and if not, try to understand why); also provide the result:

$x + y$	$[x \ ; \ y']$	$B * A$	$B ./ x'$
$x + A$	$[x; \ y]$	$A. * B$	$B ./ [x'x']$
$x' + y$	$A - 3$	$A' .* B$	$2 / A$
$A - [x'y']$	$A + B$	$2 * B$	$\text{ones}(1,3) * A$
$[x; \ y] + A$	$B' + A$	$2 .* B$	$\text{ones}(1,3) * B$ ■

### Exercise 18

Let  $A$  be a random  $3 \times 3$  matrix, and  $b$  a random  $3 \times 1$  vector (use the command `randn`). Given that  $Ax = b$ , then  $x = A^{-1}b$ . Find  $x$ . Explain what the difference is between the operators `\`, `/` and the command `inv` (look at Table 4). Having found  $x$ , check whether  $Ax - b$  is close to a zero vector. ■

## 5 Text and cell arrays

### 5.1 Character strings

Although MATLAB mainly works with numbers, it might occasionally be necessary to operate on text. Text is saved as a character string, i.e., a vector of ASCII values which are displayed as their character string representation. Since text is a vector of characters, it can be addressed in the same way as any vector. For example:

```

>> t = 'This is a character string'
t =
This is a character string

>> size(t)
ans =
     1     26

```

```
>> whos
      Name      Size      Bytes  Class
      ans       1x2        16    double array
      t         1x26        52    char array
Grand total is 28 elements using 68 bytes
>> t(10:19)
ans =
      character
>> t([2,3,10,17])      % note brackets [ ] inside parentheses ( )
                        % to denote collection of indices

ans =
hi t

To see the underlying ASCII representation, the string has to be converted using the command
double or abs:
>> double(t(1:12))
ans =
      84      104      105      115      32      105      115      32      97      32      99      104

The function char provides the reverse transformation:
>> t(16:17)
ans =
ct
>> t(16:17)+3
ans =
      102      119
>> t(16:17)-3
ans =
      96      113
>> char(t(16:17)-2)
ans =
ar
```

## Exercise 19

Perform the commands described. Use string `t` to form a new string `u` that contains the word “character” only. Convert string `u` to form the same word spelled backwards, i.e., `u1` = ‘retcarahc’.

With `findstr`, a string can be searched for a character or a combination of characters. Some examples on the use of different string functions are given below:

```
>> findstr(t, 'c')
ans =
      11      16

>> findstr(t, 'racter')
ans =
      14

>> findstr(t, u)
ans =
      11

>> strcat(u,u1)
ans =
characterretcarahc
>> strcmp(u,u1)
ans =
      0

>> q = num2str(34.35)
q =
34.35
```



```

>> z = str2num(q)
z =
    34.3500

>> whos q z
      Name      Size      Bytes  Class
      q         1x5         10  char array
      z         1x1          8  double array
Grand total is 6 elements using 18 bytes
>> t = str2num('1 -7 2')
t =
     1     -7     2
>> t = str2num('1 - 7 2')
t =
    -6     2
>> A = round(4*rand(3,3))+0.5;
>> ss = num2str(A)
ss =
4.5      2.5      2.5
1.5      4.5      0.5
2.5      3.5      3.5
>> whos ss
      Name      Size      Bytes  Class
      ss        3x27      162  char array
Grand total is 81 elements using 162 bytes
>> ss(2,1), ss(2,1:27)
ans =
1
ans =
1.5      4.5      0.5
>> ss(1:2,1:3)
ans =
4.5
1.5

```

## Exercise 20

Perform the commands shown in the example above. Define another string, e.g. `s = 'Nothing wastes more energy than worrying'` and exercise with `findstr`. ■

## Exercise 21

Become familiar with `num2str` and `str2num`. Check, e.g., what happens with `ss = num2str([1 2 3; 4 5 6])` or `q = str2num('1 3.25; 5 5.5')`. Analyze also the commands `str2double` and `int2str` (use `help`). ■

## 5.2 Text input and output

The `input` command can be used to prompt (ask) the user for numeric or string input:

```

>> myname = input('Enter your name: ', 's');
age = input('Enter your age: ');

```

There are two common text output functions: `disp` and `fprintf`. The `disp` function only displays the value of one parameter, either a numerical array or a string (the recognition is done automatically). For example:

```

>> disp('This is a statement.') % a string
This is a statement

```

```
>> disp(rand(3))
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
```

The `fprintf` function (familiar to C programmers) is useful for writing data to a file or screen when precise formatting is important. Try for instance (an explanation will follow):

```
>> x = 2;
>> fprintf('Square root of %g is %8.6f.\n' , x, sqrt(x));
Square root of 2 is 1.414214.
```

Formally, `fprintf` converts, formats and prints its arguments to a file or displays them on screen according to a format specification. For displaying on screen, the following syntax is used:

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

The format string contains ordinary characters, which are copied to the output, and conversion specifications, each of which converts and prints the next successive argument to `fprintf`. Each conversion specification is introduced by the character `'%'` and ended by a conversion character. Between the `'%'` and the conversion character there may appear:

- a minus sign; controlling the alignment within the field defined.
- a digit string specifying a minimum field length; The converted number will be printed in the field at least this wide, and wider if necessary.
- a period which separates the field width from the next digit string;
- a digit string - the precision which specifies the maximum number of characters to be printed from a string or the number of digits printed after the decimal point of a single or double type.

Format specification is given below in terms of the conversion characters and their meanings:

Command	The argument
d	is converted into decimal notation;
u	is converted into unsigned decimal notation
c	is taken to be a single character;
s	is a string
e	is taken to be a single or double and converted into decimal notation of the form: <code>[-]m.nnnnnnE[±]xx</code> , where the length of <code>n</code> 's is specified by the precision;
f	is taken to be a single or double and converted into decimal notation of the form: <code>[-]mmm.nnnnn</code> , where the length of <code>n</code> 's is specified by the precision;
g	is specified by <code>e</code> or <code>f</code> , whichever is shorter; non-significant zeros are not printed;

**Table 5,** Format specification

The special formats `\n`, `\r`, `\t`, `\b` can be used to produce next line (i.e. makes sure that the next command or output will be written on the next line), carriage return, tab and backspace respectively. Use `\\` to produce a backslash character and `%%` to produce the percent character.

Analyze the following example:

```
>> fprintf('Look at %20.6e!\nSee help fprintf', 1000*sqrt(2))
Look at      1.414214e+003!
See help fprintf
>> fprintf('Look at %-20.6e!See help fprintf\n', 1000*sqrt(2))
Look at 1.414214e+003      !See help fprintf
```

For both commands, the minimum field length is 20 and the number of digits after the decimal point is 6. In the first case, the value of `1000*sqrt(2)` is padded on the right, in the second case, because of the `'-'`, it appears on the left.

## Exercise 22

Try to understand how to use the `input`, `disp` and `fprintf` commands. For instance, try to read a vector with real numbers using `input`. Then, try to display this vector, both by calling `disp` and formatting an output by `fprintf`. Make a few variations. Try the same with a string. ■

### Exercise 23

Study the following examples in order to become familiar with the `fprintf` possibilities and exercise yourself to understand how to use these specifications:

```
>> str = 'life is beautiful';
>> fprintf('My sentence is: %s\n', str);    % note the \n format
My sentence is: life is beautiful
>> fprintf('My sentence is: %30s\n', str);
My sentence is:                life is beautiful
>> fprintf('My sentence is: %30.10s\n', str);
My sentence is:                life is be
>> fprintf('My sentence is: %-20.10s\n', str);
My sentence is: life is be
>>
>> name = 'John';
>> age = 30;
>> salary = 6130.50;
>> fprintf('My name is %4s. I am %2d. My salary is $ %7.2f.\n', name, age,
salary);
My name is John. I am 30. My salary is $ 6130.50.
>>
>> x = [0, 0.5, 1]; y = [x; exp(x)];
>> fprintf('%6.2f  %12.8f\n', y);
    0.00    1.00000000
    0.50    1.64872127
    1.00    2.71828183
```

Observe that the `fprintf` command applies its format along columns until all elements of a matrix are processed.

```
>> fprintf('%6.1e  %12.4e\n', y);
0.0e+000    1.0000e+000
5.0e-001    1.6487e+000
1.0e+000    2.7183e+000
>>
>> x = 1:3:7; y = [x; sin(x)];
>> fprintf('%2d  %10.4g\n', y);
 1      0.8415
 4     -0.7568
 7      0.657
```

■

**Warning:** `fprintf` uses its first argument to decide how many arguments follow and what their types are. If you provide a wrong type or there are not enough arguments, you will get nonsense for an answer. So, be careful with formatting your output. Look, e.g. what happens in the following case (age is not provided):

```
>> fprintf('My name is %4s. I am %2d. My salary is $ %7.2f.\n', name,
salary);
My name is John. I am 6.130500e+003. My salary is $
```

**Remark:** The function `sprintf` is related to `fprintf`, but writes to a string instead. Analyze the example:

```
>> str = sprintf('My name is %4s. I am %2d. My salary is $ %7.2f.\n', name,
age, salary)
str =
My name is John. I am 30. My salary is $ 6130.50.
```

## Exercise 24

Define a string `s = 'A woodchuck couldn't chuck any amount of wood because a woodchuck can't chuck wood, but if a woodchuck could chuck and would chuck wood, how much wood would a woodchuck chuck? Oh, shut up!'`.

**Note:** the symbol `'` within a string is represented by a double `''`. Exercise with `findstr`, i.e., find all appearances of the substring `'wood'`, `'o'`, `'uc'` or `'could'`. Try to build a new string `ss` by concatenation of separate words in a few ways; use `strcat`, `disp`, `sprintf` or `fprintf`. ■

## 5.3 Cell arrays

Cell arrays are arrays whose elements are cells. Each cell can contain any data, including numeric arrays, strings, cell arrays, etcetera. For instance, one cell can contain a string, another a numeric array, etc. Below, there is a schematic representation of an exemplar cell array:

cell 1,1	cell 1,2	cell 1,3																
<table><tr><td>3</td><td>1</td><td>-7</td></tr><tr><td>7</td><td>2</td><td>4</td></tr><tr><td>0</td><td>-1</td><td>6</td></tr><tr><td>7</td><td>3</td><td>7</td></tr></table>	3	1	-7	7	2	4	0	-1	6	7	3	7	<table><tr><td>John Smith</td></tr><tr><td>35</td></tr><tr><td>5900 6000</td></tr></table>	John Smith	35	5900 6000	<table><tr><td>1 + 3i</td></tr></table>	1 + 3i
3	1	-7																
7	2	4																
0	-1	6																
7	3	7																
John Smith																		
35																		
5900 6000																		
1 + 3i																		
cell 2,1	cell 2,2	cell 2,3																
<table><tr><td>'this is a text'</td></tr></table>	'this is a text'	<table><tr><td>'Hi'</td><td>[7,7]</td></tr><tr><td>-1 0 0 -1</td><td>'Bye'</td></tr></table>	'Hi'	[7,7]	-1 0 0 -1	'Bye'	<table><tr><td>'Living'</td></tr><tr><td>'implies effort'</td></tr></table>	'Living'	'implies effort'									
'this is a text'																		
'Hi'	[7,7]																	
-1 0 0 -1	'Bye'																	
'Living'																		
'implies effort'																		

Cell arrays can be built up by assigning data to each cell. The cell contents are accessed by brackets `{}`, e.g.

```
>> A(1,1) = {[3 1 -7; 7 2 4; 0 -1 6; 7 3 7]};
>> A(2,1) = {'this is a text' };
>> B(1,1) = {'John Smith'}; B(2,1) = {35}; B(3,1) = {[5900 6000]}

B =
    'John Smith'
    [          35]
    [1x2 double]
>> A(1,2) = {B}; % cell array B becomes one of the cells of A
>> C(1,1) = {'Hi'}; C(2,1) = {[ -1 0; 0 -1]}; C(1,2) = {[7,7]}; C(2,2) =
{'Bye'};
>> A(2,2) = {C}; %cell array C becomes one of the cells of A
>> A
A =
    [4x3 double]    {3x1 cell}
    'this is a text'    {2x2 cell}
>> A(2,2) %access the cell but not the contents
ans =
    {2x2 cell}
>> A{2,2} %use {} to display the contents
ans =
    'Hi'            [1x2 double]
    [2x2 double]    'Bye'
```

```
>> A{2,2}(2,1) %take the (2,1) element of the cell A{2,2}
ans =
    -1     0
     0    -1
```

There are also two useful functions with meaningful names: `celldisp` and `cellplot`. Use help to learn more.

The common application of cell arrays is the creation of text arrays. Consider the following example:

```
>> Month = {'January'; 'February'; 'March'; 'April'; 'May'; 'June'; 'July'; ...
'August'; 'September'; 'October'; 'November'; 'December'};
>> fprintf('It is %s.\n', Month{9});
It is September.
```

### Exercise 25

Exercise with the concept of a cell array, first by typing the examples presented above. Next create a cell array `w` with the names of week days, and, using the command `fprintf`, display on screen the current date with the day of the week. The goal of this exercise is also to use `fprintf` with a format for a day name and a date, in the spirit of the above example. ■

## 6 Control flow

Since variables are updated by commands (e.g., `b=10`; `c=b+1`), the order in which the commands are executed is important. Control flow ensures that commands will be executed in a suitable order. This chapter deals with ways of composing commands to achieve different control flows.

### 6.1 Expressions and logical expressions

To create different control flows logical expressions are a means to decide which group of subcommands to execute. These expressions result in a logical value: TRUE or FALSE. In MATLAB the result of a logical expression is 1 if it is true (e.g., `1 == 2-1`) and 0 if it is false.

First check section 2.3 on MATLAB syntax.

### 6.2 Sequential commands

The most common control flow is the sequential composition of two (or more) commands: '`C1; C2`'; denotes that command `C1` is executed before command `C2`.

### 6.3 Conditional commands

A conditional command has a number of subcommands (each consisting of a sequential group of commands). Exactly one group of subcommands (starting with `if`, `else`, or `elseif`, and ending with `else`, `elseif`, or `end`) is selected for execution. The choice among the various candidate subcommands is based upon the truth-value yielded by a logical expression `Elogical` (e.g. `a > b`). In the example below the `disp` function is frequently used.

Syntax	Example
<pre>if E<sub>logical</sub>     C<sub>1</sub>; else     C<sub>2</sub>; end</pre>	<pre>if (a &gt;= b)     max = a;     disp('a &gt;= b'); else     max = b;     disp('b &gt; a'); end</pre>
Syntax	Example
<pre>if E<sub>1</sub>     C<sub>1</sub>; % executed if E<sub>1</sub> is True elseif E<sub>2</sub>     C<sub>2</sub>; % executed if E<sub>2</sub> is True else     C<sub>3</sub>; % executed if no other         % expression is True end</pre>	<pre>if (height &gt; 190)     max = a;     disp('a &gt;= b'); elseif (height &gt; 170)     disp('tall'); elseif (height &lt; 150)     disp('small'); else     disp('average'); end</pre>

### Exercise 26

Create a script that asks for a number and computes the drag coefficient  $C$ , depending on the Reynold's number (make use of `elseif`). The command input might be useful here (use `help` if needed).

$$C = \begin{cases} 0, & N \leq 0 \\ 24/N, & N \in (0, 0.1] \\ (24/N)(1+0.14N^{0.7}), & N \in (0.1, 1000] \\ 0.43, & N \in (1000, 500000] \\ 0.19 - 80000/N, & N > 500000 \end{cases}$$

Check whether your script works correctly. Compute the drag coefficient for e.g.  $N = -3e3$ ,  $0.01$ ,  $56$ ,  $1e3$ ,  $1e6$  (remember that e.g.  $3e3$  is a MATLAB notation of  $3 \cdot 10^3$ ). ■

### Exercise 27

Write a script that asks for an integer and checks whether it can be divided by 2 or 3. Consider all possibilities, such as: divisible by both 2 and 3, divisible by 2 and not by 3 etc. (use the command `rem`). ■

It is also possible to select a group of subcommands on the basis of values other than truth values (e.g., scalars or a string). The value yielded by the expression  $E$  must equal the value of one of the literals  $L_i$  and then the corresponding subcommand  $C_i$  is chosen.

Syntax	Example
<pre> switch E     case L1         C1; % executed if L1 equals E     case {L2, L3, ...}         C2; % executed if L2 or L3 equals E     otherwise         C3; % executed if no literal Li             % equals E end         </pre>	<pre> method = 'lin' switch method     case {'lin', 'lin2'}         disp('Method is linear. ');     case {'near'}         disp('Method is nearest. ');     otherwise         disp('Unknown method. '); end         </pre>

### Exercise 28

Assume that the months are represented by the numbers from 1 to 12. Write a script that asks you to provide a month and returns the number of days in that particular month. Alternatively, write a script that asks you to provide a month name (e.g. 'June') instead of a number. Use the `switch`-construction. ■

### Exercise 29

Write a MATLAB function that implements :

*This is **not** MATLAB syntax*

```

if 0 < x < 7
    y = 4x
elseif 7 < x < 55
    y = -10x
else
    y = 333
end
        
```

Verify its functionality using

- $x = -1$ ,  $x = 5$ ,  $x = 30$ , and  $x = 56$  ■

## 6.4 Iterative commands

An iterative command (commonly known as a **loop**) has a subcommand (the loop body) that is to be executed repeatedly, and some kind of phrase that determines when iteration will cease. We can classify iterative commands according to whether the number of iterations is determined in advance (**definite**) or not (**indefinite**).

### 6.4.1 Definite iteration

Definite iteration is characterized by the use of a **control variable**. The loop body is executed with the control variable taking each of a predetermined sequence of values. This sequence of values is called the control sequence.

**note 1:** the control sequence can be a vector.

**note 2:** the step can have a negative value.

Syntax	Example
<pre>for a = first : step : last     C1; end</pre>	<pre>for a = 0 : 0.5 : 4     disp(a^2) end</pre>
Syntax	Example
<pre>for a = vector     C1; end</pre>	<pre>v = [0 : 0.5 : 4]; for a = v     disp(a^2); end</pre>

In MATLAB, the `for` command is self-contained: control variables are created within the loop body. Thus the control variable has no value (a in the example) outside the loop (it does not exist). The loop body cannot cause an assignment to the control variable - it is in fact a constant!

### 6.4.2 Indefinite iteration

This concerns loops in which the number of iterations is not determined in advance. The `while` command consists of a loop body preceded by a truth-valued expression  $E_{\text{logical}}$  (the loop condition) that controls whether iteration is to be (started or) continued.

**note 1:** The number of executions of a command can also be 0!

**note 2:** if  $E_{\text{logical}}$  is always true the loop will not have a (natural) end!

Syntax	Example
<pre>while E_logical     C1; end</pre>	<pre>sum = 0; delta = 1; iter = 0; while(delta &gt; eps)     sum = sum + delta;     delta = delta / 2;     iter = iter + 1; end disp(sum); disp(iter);</pre>

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 in a script:

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

Now 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 two `while` loops:

```
n = length(x); m = length(y); ii = 1 % initialize ii
while ii <= n
    jj = 1; % initialize jj
```

```

while jj <= m
    A(ii,jj) = x(ii)*y(jj);
    jj=jj+1    % increment jj: it does not happen automatically!
end
ii=ii+1    % increment ii
end

```

### Exercise 30

Use a loop construction to carry out the computations. Write short scripts.

- Given the vector  $x = [1 \ 8 \ 3 \ 9 \ 0 \ 1]$ , create a short set of commands that will:
  - add up the values of the elements (check with `sum`);
  - computes the running sum (for element  $ii$ , the running sum is the sum of the elements from 1 to  $ii$ ; check with `cumsum`).
- Given  $x = [4 \ 1 \ 6 \ -1 \ -2 \ 2]$  and  $y = [6 \ 2 \ -7 \ 1 \ 5 \ -1]$ , compute matrices whose elements are created according to the following formulas:
  - $a_i = x_i y_i$  and add up the elements of  $a$ ;
  - $b_{ij} = x_i / (2 + x_i + y_j)$ ;
  - $c_{ij} = 1 / \max(x_i, y_j)$ ;
- Determine the sum of the first 50 squared numbers with a control loop.
- Write a script to find the largest value  $n$  such that the sum:  $\sqrt{1^3} + \sqrt{2^3} + \dots + \sqrt{n^3}$  is less than 1000. ■

## 6.5 Evaluation of logical and relational expressions in the control flow structures

The relational and logical expressions may become quite complicated. It is not difficult to operate on them if you understand how they are evaluated. To explain this in more detail, let us consider the following example:

```

if (~isempty(data)) & (max(data) < 5)
    ...
end

```

This construction of the if-block is necessary to avoid comparison if `data` happens to be an empty matrix. In such a case you cannot evaluate the right logical expression and MATLAB returns an error. The `&` operator returns 1 only if both expressions: `~isempty(data)` and `max(data) < 5` are TRUE, and 0 otherwise. When `data` is an empty matrix, the next expression is not evaluated since the whole `&`-expression is already known to be false (lazy evaluation). The second expression is checked only if `data` is a non-empty matrix. Don't forget to put logical expression units between brackets to avoid unexpected evaluations!

### Exercise 31

Consider the following example:.

```

max_iter = 50;
tolerance = 1e-4;
iter = 0;
xold = 0.1;
x = 1;
while (abs (x-xold) > tolerance) & (iter < max_iter)
    xold = x;
    x = cos(xold);
    iter = iter + 1;
end

```



This short program tries to solve the equation  $\cos(x) = x$  ( $x$  is the solution found). Make the script `solve_cos` from the presented code. Note that the while-loop is repeated as long as both conditions are true. If either the condition  $(|x - x_{old}| \leq \text{tolerance})$  or  $(\text{iter} \geq \text{max\_iter})$  is fulfilled then the loop is quitted. Run the script `solve_cos` for different tolerance parameters, e.g.  $1e-3$ ,  $1e-6$ ,  $1e-8$ ,  $1e-10$ , etc. Use `format long` to check more precisely how much the found  $x$  is really different from  $\cos(x)$ . For each tolerance value check the number of performed iterations (iter value). ■

## ☞ Exercise 32

Create the script `solve_cos2`, which is equal to the given, replacing the while-loop condition by:

```
while (abs (x-xold) > tolerance) | (iter < max_iter)
```

Try to understand the difference and confirm your expectations by running `solve_cos2`. What happens to iter? ■

## 7 Visualization

MATLAB can be used to visualize the results of an experiment. To that end, you should define your variables such that each of them contains all values of one parameter to plot.

### 7.1 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 same number of elements).

The commands `loglog`, `semilog` and `semilogy` are similar to `plot`, except that they use either one or two logarithmic axes. Type the following commands, after predicting the result:

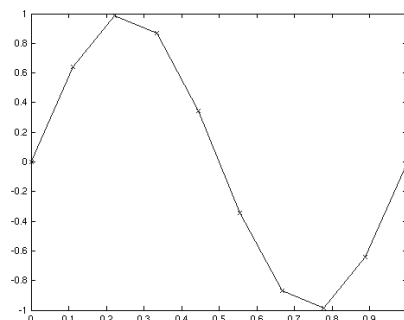
```
>> x = 0:10;
>> y = 2.^x; % the same as y = [1 2 4 8 16 32 64 128 256 512 1024]
>> plot(x,y); % to get a graphic representation
>> semilogy(x,y); % to make the y-axis logarithmic
```

As you can see, the same figure is used for both commands. The previous function is removed as soon as the next is displayed. The command `figure` gives you an extra figure-window. 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.

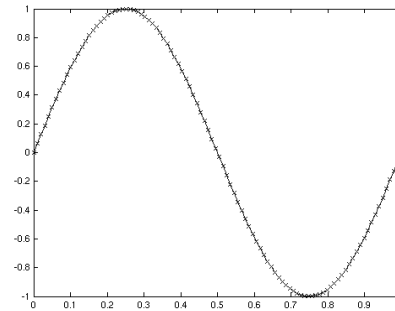
For plotting a graph of a function, it is important to **sample** the function sufficiently well, a much higher sampling than Nyquist sampling is required.

```
>> x = [0 0.5 1]; % Nyquist sampling
>> y = sin(2*pi*x);
>> plot(x,y,'x-');

>> x = linspace(0,1,10); % coarse sampling
>> y = sin(2*pi*x);
>> plot(x,y,'x-');
```



```
>> x = linspace(0,1,100);    % fine sampling
>> y = sin(2*pi*x);
>> plot(x,y, 'x-');
```



The plot command uses a blue solid line by default. It is possible to change both the style and colour, e.g.:

```
>> x = 0:0.4:3; y = sin(5*x);
>> plot(x,y, 'r--') % produces a dashed red line.
```

Symbol	Colour	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 6,** Plot colours and styles

The third argument of `plot` specifies the colour (optional) and the line style. Table 6 shows a few possibilities, `help plot` shows all. To add a title, grid and to label the axis, one uses:

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

### ✎ Exercise 33

Make a plot connecting the coordinates: (2,6), (2.5, 18), (5,17.5), (4.2, 12.5) and (2,12) by a line. ■

### ✎ Exercise 34

Plot the function  $y = \sin(x) + x - x \cos(x)$  in two separate figures for the intervals:

$0 < x < 30$  and  $-100 < x < 100$ . Add a title and axis description. ■

## 7.2 Several functions in one figure

There are different ways to draw several functions in the same figure-window. The first one uses the command `hold on`. After this command, all functions will be plotted in the same figure until the command `hold off` is used. When more than one function is plotted in a window, it is useful to use different symbols and colours. An example is:

```
>> x1 = 1:.1:3.1; y1 = sin(x1);
>> plot(x1, y1, 'md'); % 'd' for diamond symbol
>> 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 extend the list of arguments handled by the plot command. The next command 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 axis better fitting the curves, perform

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

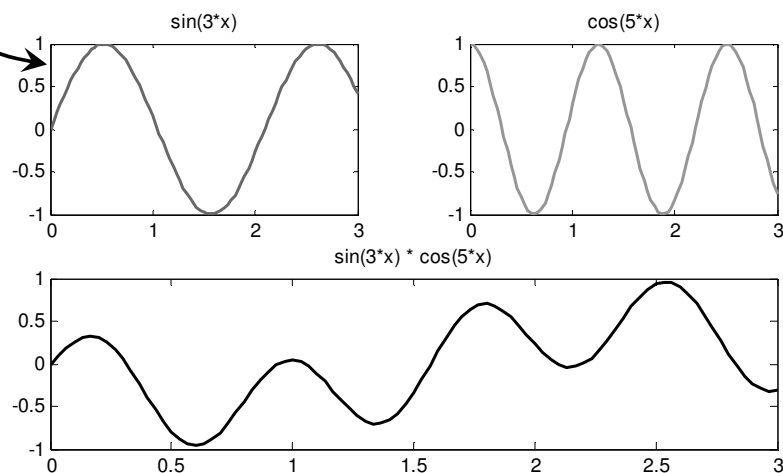
The same can be achieved by `axis tight`. It might be also useful to exercise with axis options (see help), e.g. `axis on/of`, `axis equal`, `axis image` or `axis normal`. A descriptive legend can be included with the command `legend`, e.g.:

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

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

```
>> x = 1:.1:4;
>> y1 = sin(3*x);
>> y2 = cos(5*x);
>> y3 = sin(3*x).*cos(5*x)
>> subplot(2,2,1); plot(x,y1,'m-'); title('sin(3*x)')
>> subplot(2,2,2); plot(x,y2,'g-'); title('cos(5*x)')
>> subplot(2,1,2); plot(x,y3,'k-'); title('sin(3*x) * cos(5*x)')
```

`subplot(2,2,1)`: if the plot window is a figure matrix with dimensions  $n \times k$  (row  $\times$  column) this would be the subfigure at the first position of a  $2 \times 2$  matrix.



### Exercise 35

Plot the functions  $f(x) = x$ ,  $g(x) = x^3$ ,  $h(x) = e^x$  and  $z(x) = e^{(x^2)}$  over the interval  $[0,4]$  using both the normal and the log-log scales. Use an appropriate sampling to get smooth curves. Describe your plots by using the functions: `xlabel`, `ylabel`, `title` and `legend`. ■

Command	Result
<code>grid on/off</code>	adds a grid to the plot at the tick marks or removes it
<code>axis([xmin xmax ymin ymax])</code>	sets the minimum and maximum values of the axes
<code>box off/on</code>	removes the axes box or shows it
<code>xlabel('text')</code>	plots the label text on the x axis

<code>ylabel('text')</code>	plots the label text on the y axis
<code>title('text')</code>	plots a title above the graph
<code>text(x,y,'text')</code>	adds text at the point (x,y)
<code>gtext('text')</code>	adds text at a manually (with the mouse) indicated point
<code>legend('fun1','fun2')</code>	plots a legend box (move it with the mouse)
<code>legend off</code>	deletes the legend box
<code>clf</code>	clears the current figure
<code>subplot</code>	creates a subplot in the current figure

**Table 7,** Plot/print commands

### 7.3 Printing

Before printing a figure, you might want to add some information, such as a title, or change somewhat in the lay-out. Table 7 shows some of the commands that can be used.

#### Exercise 36

Plot the functions  $y_1 = \sin(4x)$ ,  $y_2 = x \cos(x)$ ,  $y_3 = \sqrt{x} (x+1)^{-1}$  for  $x = 1:0.25:10$ , and plot a single point  $(x,y) = (4,5)$ , all in one figure. Use different colours 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  $y_3$  in more detail. ■

When you like the displayed figure you can print it to paper. The easiest way is to click 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 also exists 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 there are many parameters they will not be explained here (click `help print` to learn more) instead, try to understand the examples:

```
>> print -dwinc          % print current figure to current printer in colour
>> print -f1 -deps myf.eps % print Figure no.1 to file myf.eps in black
>> print -f1 -depssc myg.eps % print Figure no.1 to file myg.eps in colour
>> print -dtiff myfile.tif % print current figure to file myfile.tif
>> print -f2 -djpeg myfile.jpg % print Figure no.2 to the file myfile.jpg
```

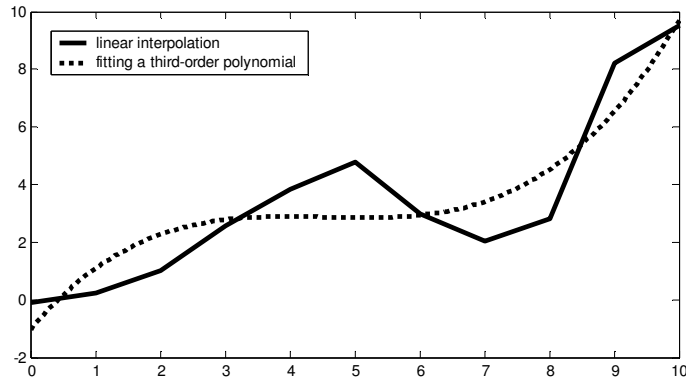
## 8 Numerical analysis

Numerical analysis can be used whenever it is very difficult or impossible to determine the analytical solution. MATLAB can be used to find, for instance, the minimum, maximum or integral of a particular function.

There are two ways to describe data with an analytical function:

- curve fitting or regression; finding a smooth curve that best fits (approximates) the data according to a criterion (e.g., best least square fit). The curve does not have to go through any of the data points.
- interpolation; the data are assumed to be error-free: desired is a way to describe what happens between the data points. In other words, given a finite set of points, the goal of interpolation is to find a function from a given class of functions (e.g. polynomials) which passes through the specified points.

The figure below illustrates the difference between regression (curve fitting) and interpolation:



In this section we will briefly discuss some issues on curve fitting, interpolation, function evaluation, integration and differentiation. It is also possible to solve differential equations (see Matlab books mentioned in the Introduction or the Projects on Blackboard).

### 8.1 Curve fitting

MATLAB interprets the best fit of a curve as ‘least squares curve fitting’. The curves used are restricted to polynomials. Which the command `polyfit` any polynomial can be fitted to the data. `polyfit(x,y,n)` finds the coefficients of a polynomial of degree `n` that fits the data (finds the linear combination between `x` and `y`). Let’s start with finding a linear regression of some data:

```
>> x = 0:10;
>> y = [-0.10 .24 1.02 1.58 2.84 2.76 2.99 4.05 4.83 5.22 7.51]
>> p = polyfit(x,y,1) % find the fitting polynomial of order 1
p =
    0.6772   -0.3914
```

The output of `polyfit` is a row vector of the polynomial coefficients; the solution of this example is therefore  $y = 0.6772x - 0.3914$ .

#### ☞ Exercise 37

Execute the above example. Plot the data and the fitted curve in a figure. Determine the 2nd and 9th order polynomial of these data as well, and display the solution in a figure.

Hint: `x1 = linspace(xmin, xmax, n);` results in a vector with `n` elements evenly distributed from `xmin` till `xmax`. `z = polyval(p,x1);` calculates the values of the polynomial for every element of `x1`. ■

#### ☞ Exercise 38

Define `x` and `y` as follows:

```
>> x = -5:5;
>> y = 0.2 * x.^3 - x + 2; % a 3rd order polynomial
>> y = y + randn(1,11); % add some noise to y
```

Determine the 3rd order polynomial fitting `y` (note that because of added noise the coefficients are different from those originally chosen). Try some other polynomials as well. Plot the results in a figure. ■

### 8.2 Interpolation

The simplest way to examine various interpolations is by displaying the function plots with the command `plot`. When the neighbouring data points are connected with straight lines, this is called linear interpolation. When more data points are taken into account, the effect of interpolation becomes less visible. Analyse:

```
>> x1 = linspace(0,2*pi, 2);
>> x2 = linspace(0,2*pi,4);
>> x3 = linspace(0,2*pi,16);
>> x4 = linspace(0,2*pi,256);
>> plot(x1, sin(x1), x2, sin(x2),x3,sin(x3),x4, sin(x4))
>> legend('2 points','4 points','16 points','256 points')
```

There also exist MATLAB functions that interpolate between points, e.g., `interp1` or `spline`, as a 1D interpolation and `interp2`, as a 2D interpolation. Perform the following commands:

```
>> N = 50;
>> x = linspace(0,5,N);
>> y = sin(x) .* sin(6*x);
>> subplot(2,1,1); plot(x,y);
>> hold on
>> p = randperm(N);
>> pp = p(1:round(N/2)); % choose randomly N/2 indices of points from [0,5]
>> pp = sort(pp);        % sort the indices
>> xx = x(pp);           % select the points
>> yy = y(pp);
>> plot(xx,yy,'ro-')      % this is a 'coarse' version
>> yn = interp1(xx,yy,x,'nearest'); % nearest neighbour interpolation on
all x points
>> plot(x,yn,'g')
>> axis tight
>> legend('Original','Crude version'); % Nearest neighbour interpolation
>> subplot(2,1,2); plot(xx,yy,'ro-');
>> hold on
>> yc = interp1(xx,yy,x,'linear');    % spline interpolation
>> plot(x,yc,'g');
>> ys = spline(xx,yy,x);
>> plot(x,ys,'k')
>> axis tight
>> legend('Crude version','Linear interpolation','Spline interpolation');
```

### 8.3 Evaluation of a function

As said before, the command `plot` plots a function by connecting defined data points.

MATLAB Command	Result
<code>fplot('f', [min_x max_x])</code>	plots the function <code>f</code> on <code>[min_x, max_x]</code>
<code>fminbnd('f', min_x, max_x, options)</code>	returns the <code>x</code> -value for which the function <code>f</code> reaches a minimum on <code>[min_x, max_x]</code>
<code>fzero('f', x0)</code>	returns the <code>x</code> -value for which the function <code>f</code> is zero in the neighborhood of <code>x0</code>

**Table 8,** Evaluation of a function

When a function is constant and not very interesting over some range, and next shows an unexpected wild behaviour over another, it might be misinterpreted by using `plot`. Then, the command `fplot` is more useful. An example is:

```
>> x = 0: pi/8:2*pi; y=sin(8*x); plot(x,y,'b')
>> hold on
>> fplot('sin(8*x)', [0 2*pi], 'r')
>> title('sin(8*x)')
>> hold on
```

Execution of these commands results in a figure with an almost straight blue line and a red sine. `fplot` can also be used to plot a defined function:

```
>> f = 'sin(8*x)';
>> fplot(f,[0 2*pi], 'r');title(f)
```

Let:

$$f(x) = \frac{1}{(x-0.1)^2 + 0.1} + \frac{1}{(x-1)^2 + 0.1}.$$

The point at which  $f(x)$  takes its minimum is found using `fminbnd`. By default, the relative error is  $1e-4$ . It is possible however, to obtain a higher accuracy, by specifying an extra parameter while calling `fminbnd`.

```
>> format long          % you need this format to see the change in accuracy
>> f = '1./((x-0.1).^2 + 0.1) + 1. / ((x-1).^2 + 0.1)';
>> fplot(f,[0 2]);
>> options = optimset('TolX',1e-4);
>> [xm1,fm1] = fminbnd(f,0.3,1, options)
>> options = optimset('TolX',1e-12);
>> [xm2,fm2] = fminbnd(f,0.3,1, options)
>> [xm1,xm2]          % compare the two answers
```

Important: Note that `format` does not change the accuracy of the numbers or calculations; it just changes the numbers of digits displayed on the screen.

### Exercise 39

Determine the maximum of the function  $f(x) = x \cos(x)$  over the interval  $10 < x < 15$ . Hint: there is no function `fmax`. Find the minimum of  $-f(x)$ , instead. ■

### Exercise 40

Find the zero, i.e.  $x_0$ , of the functions  $f(x) = \cos(x)$  and  $g(x) = \sin(2x)$  around the point  $x = 2$ . Use the command from table 8. Check the sign of the functions for  $x_0$ , and that the values are approximately zero. ■

## 8.4 Integration and differentiation

The integral, or the surface underneath a 2D function, can be determined with the command `trapz`. `trapz` does this by measuring the surface between the x-axis and the data points, connected by straight lines. The accuracy of this method depends on the distance between the data points:

```
>> x = 0:0.5:10; y=0.5*sqrt(x)+ x .* sin(x);
>> integrall = trapz(x,y)
integrall =
    18.16550108689927
>> x = 0:0.05:10; y=0.5*sqrt(x)+ x .* sin(x);
>> integral2 = trapz(x,y)
integral2 =
    18.38461257420812
```

A more accurate result can be obtained by using the command `quad` or `quadl`, which also numerically evaluate an integral of the function `f` specified by a string.

Let again be:

$$f(x) = \frac{1}{(x-0.1)^2 + 0.1} + \frac{1}{(x-1)^2 + 0.1}.$$

```
>> f = '1./((x-0.1).^2 + 0.1) + 1./((x-1).^2 + 0.1)';
>> integrall = quad(f,0,10)
>> integrall = quadl(f,0,10)
```

You can also add an extra parameter to `quad` and `quadl`, specifying the accuracy.

### Exercise 41

Find the integral of the function  $f(x) = e^{-\frac{1}{2}x^2}$  over the interval  $[-3,3]$ . Exercise with different MATLAB commands and different accuracies. ■

Differentiation is performed by determining the slope in each data point. Numerically this is done by first fitting a polynomial, followed by differentiating this function:

```
>> x = 0:10;
>> y = [-0.10 .24 1.02 -1.58 2.84 2.76 7.75 .35 4.83 5.22 7.51];
>> p = polyfit(x,y,5)
>> dp = polyder(p)
>> x1 = linspace(0,10,200);
>> z = polyval(p,x1);
>> dz = polyval(dp, x1);
>> plot(x,y, 'g.-', x1, z, 'b', x1, dz, 'r:')
>> legend('data points','curve fit','derivative')
```

### Exercise 42

Determine the integral and the derivative of the 4th polynomial fitted through (x,y) with  $x = 4:13$  and  $y = [12.84 \ 12.00 \ 7.24 \ -0.96 \ -11.16 \ -20.96 \ -27.00 \ -24.96 \ -9.56 \ 25.44]$ . ■

## 8.5 Numerical computations and the control of flow structures

Take a look at the problem of a Taylor expansion. The series will be determined numerically until the additional terms are smaller than a defined precision. Look at the Taylor expansion of :

$$\frac{1}{1-x} = 1 + x + x^2 + \dots \text{ with } x = 0.42:$$

*Example*

```
s = 0; x = 1;
x0 = 0.42;
while (x > 1e-6) % absolute
    s = s + x;
    x = x * x0;
end
```

*Example*

```
s = 0; x = 1;
x0 = 0.42;
while (x > 1e-6 * s) % relative
    s = s + x;
    x = x * x0;
end
```

The approximation of  $1/(1-x)$  is returned in s (from sum). In the first case (on the left), the iteration process is stopped as soon as the next term t is smaller than an **absolute** value of  $1e-6$ . In the other case, this value is **relative** to the outcome.

### Exercise 43

Find the Taylor expansion of  $\frac{1}{1-x}$  and  $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$  with an absolute precision of  $1e-5$

for  $x=0.4$  and  $x=-0.4$ . ■

It is also possible to determine the Taylor expansion of  $\frac{1}{1-x} = 1 + x + x^2 + \dots$  with a fixed number of 30 terms ( $x=0.42$  again):

*Example*

```
s = 0; x = 1;
x0 = 0.42;
while (x > 1e-6)
    s = s + x;
    x = x * x0;
end
```

*Example*

```
s = 0; x = 1;
x0 = 0.42;
n1 = 1; step = 1; n2 = 30;
for k = n1:step:n2
    x = x * x0;
    s = s + x;
end
```

Note that the index k can be indicated with a starting value, step size and end value.



## 9 Optimizing the performance of Matlab code

MATLAB is a matrix language, so it is designed for matrix operations. For best performance you should take advantage of this fact.

### 9.1 Vectorization - speed-up of computations

Vectorization is simply the use of (compact) expressions that operate on all elements of a vector without explicitly executing a loop. MATLAB is optimized such that vector or matrix operations are much more efficient (faster) than loops. Most built-in functions support vectorized operations. So, try to replace loops with vector operations whenever possible. For example, instead of using:

```
for k = 1:10
    t(k) = 2*k;
    y(k) = sin(t(k));
end
```

try this:

```
t=2:2:20;
y = sin(t);
```

Copying or transforming matrices can be vectorized as well. Check the equivalence between the **scalar** version:

```
n = 10;
A = rand(n,n);
B = ones(n,n);
for k = 1:n
    B(2,k) = A(2,k)
end
```

and the **vectorized** code of the last loop:

```
B(2,:) = A(2,:);
```

Or, check the equivalence between the following loop:

```
for k=1:n
    B(k,1) = A(k,k);
end
```

and the vectorized expression:

```
B(1:n,1) = diag(A(1:n,1:n));
```

Logical operations and proper use of the colon notation make programs work faster. However, some experience is needed to do this correctly. Therefore, the advice is: start first with the scalar code and then vectorize it where possible, by removing loops.

#### Exercise 44

Look both at your script or function m-files already written, especially your loop exercises. Try to vectorize their codes wherever possible. ■

#### Exercise 45

Try to vectorize the following two code examples !:

<i>code 1</i>	<i>code 2</i>
<pre>n=20; m=10; A = rand(n,m); for ii = 1:n     for jj = 1:m;         if (A(ii,jj) &gt; 0.5)             A(ii,jj) = 0;         else             A(ii,jj) = 1;         end     end end end</pre>	<pre>n=20; m=10; A = randn(n,m); x = randn(n,1); p = zeros(1,m); % this is an array pre-allocation for ii = 1:m     p(ii) = sum(x .* A(:,ii));     % what is p here? end</pre>

## 9.2 pre-allocation an array

In ‘real’ programming languages the programmer has to allocate (reserve) memory. In MATLAB, this is not required, but it improves the execution speed and, apart from that, it improves the readability of the code. Preallocating an arrays that stores output results, prevents MATLAB from resizing this array each time you enlarge it. Preallocating also helps to reduce memory fragmentation if you work with very large matrices. During the MATLAB session memory can be fragmented. As a result, there may be plenty of free memory, but not enough contiguous space to store a large variable.

### Exercise 46

Create two scripts with the listings below. Run them and compare the results, i.e., measure their execution times. The pair of commands `tic` and `toc` is used to this end.

*code 1*

```
clear s x;
tic;
x = -500:0.1:500;
% without array pre-allocation
for k = 1:length(x)
    if (x(k) > 0)
        s(k) = sqrt(x(k));
    else
        s(k) = 0;
    end
end
toc;
```

*code 2*

```
clear s x;
tic;
x = -500:0.1:500;
s = zeros (size(x)); % preallocate
for k = 1:length(x)
    if (x(k) > 0)
        s(k) = sqrt(x(k));
    else
        s(k) = 0;
    end
end
toc;
```

Remember that scripts work in the workspace, inferring with the existing variables. Therefore, for a fair comparison the variables `s` and `x` should be removed before running each script. Having done this, try to vectorize the code. ■

## 9.3 MATLAB’s tricks and tips

Matrix elements can be addressed in MATLAB using a vector as an index:

- If `x` and `y` are vectors, then `x(y)` is the vector `[x(y(1)), x(y(2)), ..., x(y(n))]`, where `n=length(y)`. For instance:

```
>> x = [3 -1 4 2 7 2 3 5 4];
>> y = 1:2:9; x(y)
ans =
     3     4     7     3     4
>> y = [5 5 5]; x(y)
ans =
     7     7     7
>> y = [1 5 1 1 7]; x(y)
ans =
     3     7     3     3     3
```

- If `x` and `y` are vectors of the same size and `y` only consists of 0’s and 1’s then MATLAB interprets `y` via logical-indexing. As a result, the elements of `x` are returned whose position corresponds to the location of a 1 in `y`. For instance:

```
>> x = [3 -1 4 2 7 2 3 5 4];
>> y = x < 4, x(y)
y =
     1     1     0     1     0     1     1     0     0
ans =
     3    -1     2     2     3
>> y = (x == 2) | (x > 6), x(y)
y =
     0     0     0     1     1     1     0     0     0
ans =
     2     7     2
```

The examples before should serve you to optimize your MATLAB routines. Use `help` when necessary to learn more about commands used and test them with small matrices. When more solutions are given, try to use `tic` and `toc` to measure the performance for large matrices, in order to decide which solution is faster. Depending on the version of MATLAB, the computer hardware, and the dimension of the matrices (the value of `n`) the speed of the given solutions may vary.

- Create a row (column) vector of `n` uniformly spaced elements:

```
>> a = -1; b = 1; n = 50;
>> x1 = [a:2/(n-1):b];           % a row vector
>> y1 = [a:2/(n-1):b]';         % a column vector
>> x2 = linspace(a,b,n);         % a row vector
>> y2 = linspace(a,b,n)';       % a column vector
```

- Shift `k` (`k` should be positive) elements of a vector:

```
>> x = [3 -1 4 2 7 2 3 5 4];
>> x([end 1:end-1]);             % shift right (or down for columns) 1 element
>> k = 5;
>> x([end-k+1:end 1:end-k])     % shift right (or down for columns) k elements
>> x([2: end 1]);               % shift left (or up for columns) 1 element
>> x[k+1:end 1:k]);             % shift left (or up for columns) k element
```

- Initialize a vector with a constant

```
>> n = 10000;
>> x = 5 * ones(n,1);           % 1st solution
>> y = repmat(5,n,1);           % 2nd solution
>> z = zeros(n,1); z(:)=5;      % 3rd solution
```

- Create an `nxn` matrix of 3's

```
>> n = 1000;
>> A = 3 * ones(n,n);           % 1st solution
>> B = repmat(3,n,n);           % 2nd solution
```

- Create a matrix consisting of a row vector duplicated `m` times:

```
>> m=10000
>> n = 10000;
>> x = 1:5;
>> A = ones(n,1) * x;           % 1st solution
>> B = x(ones(n,1),:);         % 2nd solution
```

- Create a matrix consisting of a column vector duplicated `n` times:

```
>> n = 100;
>> x = (1:5)';
>> A = x * ones(1,n);           % 1st solution
>> B = x(:,ones(n,1));          % 2nd solution
```

- Given a vector `x`, create a vector `y` in which each element is replaced `n` times:

```
>> n = 10;
>> x = [2 1 4];
>> y = x(ones(1,n),:); y = y(:)';
>> x = [2 1 4]';
>> y = x(:,ones(1,n))'; y = y(:);
```

- Reverse a vector:

```
>>n = 100000;
>> x = [1:n];
>> x(n:-1:1) % 1st solution
>> fliplr(x) % 2nd solution
```

- Reverse one column of a matrix:

```
>> A = round (5 * rand(10,20));
>> c = 2;
>> A(:,c) = flipud (A(:,c));
```

- Interchange rows or columns of a matrix:

```
>> A = round (5*rand(10,20));
>> i1 = 1; i2 = 4;
>> A([i1,i2],:) = A([i2,i1],:); % swap rows
>> A(:, [i1,i2]) = A(:, [i2,i1]); % swap columns
```

- Matrix A can be transformed into a row vector by writing A(:).

Find out which elements are shared by two matrices (or vectors):

```
>> A = round (5*rand(10,20));
>> B = round (6*rand(30,50));
>> intersect (A(:), B(:)); % different sizes of A and B are permitted
```

- Combine two vectors into one, removing repetitive elements:

```
>> x = [1:10];
>> y = [1 5 4 1 7 -1 2 2 6];
>> union (x,y)
```

- Find unique elements in a vector

```
>> x = [1 5 4 1 7 -1 2 2 6 1 1 ];
>> unique(x)
```

- Keep only the diagonal elements of the matrix multiplications, i.e., vectorize the loop (both A and B are n-by-m matrices):

```
>> A = round (5*rand(10,20));
>> B = round (7*rand(10,20));

>> z = zeros(n,1); % 1st solution
>> for ii = 1:n
>>     z(ii) = A(ii,:) * B(ii,:)' ;
>> end

>> C = diag(A * B'); % 2nd solution
>> D = sum (A.* B, 2); % 3rd solution
```

- Scale all columns of the matrix A by a column vector x:

```
>> A = round (7*rand(10,20));
>> [n,m] = size(A);
>> x = [1:n]';
>> B = A ./ x(:,ones(m,1));
```

- Scale all rows of the matrix A by a row vector x:

```
>> A = round (7*rand(10,20));
>> [n,m] = size(A);
>> x = [1:m]';
>> B = A ./ x(ones(n,1),:);
```

Other useful tips are:

- Use the `ginput` command to input data with a mouse. Try, for instance:

```
>> x = 0; y = 0;
>> while ~isempty(x)
>>     [x1,y1] = ginput(1);
>>     plot([x x1], [y y1], 'b.-');
>>     hold on
>>     x = x1; y = y1;
>> end
```

- If you want to find out what takes so long in your MATLAB code, use the command `profile`, which ‘helps’ you debug and optimize M-files by tracking their execution time. For each function, the profiler records information about execution time, number of calls, parent functions, child functions, code line hit count, and code line execution time. Try, e.g.:

```
>> profile on - detail builtin
>> rgb = imread('ngc6543a.jpg');
>> profile report
```

## 10 Writing and debugging MATLAB programs

The recommendations in this section are general for programming in any language. Learning them now will turn out to be beneficial in the future or while learning real programming languages like C, when structured programming is indispensable.

### 10.1 Structured programming

Never write all code at once; program in small steps and make sure that each of these small steps works as expected, before proceeding to the next one. Each step should be devoted to only one task. Do not try to solve many tasks in one module, because you may easily make a mess. This is called a **structured** or **modular** way of programming. Formally, modularity is the hierarchical organization of a system or a task into self-contained subtasks and subsystems, each having a prescribed input-output communication. It is an essential feature of a well-designed program. The benefits of structured programming are: easier error detection and correction, modifiability, extensibility and portability. A general approach to program development is presented below:

#### Specification

Read and understand the problem. The computer cannot do anything on its own: you have to tell it how to operate. Before using the computer, some level of preparation and thought is required. Some basic questions to be asked are:

- What are the parameters/inputs for the problem?
- What are the results/outputs for this problem?
- In what form should the inputs/outputs be provided?
- What kind of algorithm is needed to find the outputs from the inputs?

## Design

Split your problem into a number of smaller and easier tasks. Decide how to implement them. Start with a schematic implementation to solve your problem, e.g. create function headers or script descriptions (decide about the input and output arguments). To do this, you may use, e.g., a top-down approach. You start at the most general level, where your first functions are defined. Each function may be again composed of a number of functions (subfunctions). While ‘going down’ your functions become more precise and more detailed.

Imagine that you have to compare the results of the given problem for two different datasets, stored in the files `data1.dat` and `data2.dat`. Schematically, such a top-down approach could be designed as follows:

This is the top (the most general level). A script `solve_problem` is created:

```
load data1.dat;
load data2.dat;

[res1, err1] = solve_problem (data1);
[res2, err2] = solve_problem (data2);
compare_results (res1, res2, err1, err2);
```

This is the second level. The functions `solve_problem` and `compare_result` belong here.

Each of them has to be defined in a separate file:

```
function [res, err]=solve_problem (d)
% Here should be some (possibly detailed) description.
....
res = ...    % the data d is used to compute res
err = compute_error (res);
return;

function compare_results (res1, res2, err1, err2)
% some description.
tol = 1e-6;
....
if abs (err1 - err2) > tol
    fprintf ('The difference is significant.')
else
    fprintf ('The difference is NOT significant.')
end
return;
```

In this example, this is the last level. The function `solve_problem` uses the function:

```
compute_error, which has to be defined:
function err = compute_error (res)
% here should be some (possibly detailed) description.
....
err = ...    % the variable res is used to compute err
return;
```

## Coding

Implement the algorithms sequentially (one by one). Turning your algorithm into an efficient code is not a one-shot process. You will have to try, make errors, correct them and even modify the algorithm. So, be patient. While implementing, make sure that all your outputs are computed at some point. Remember about the comments and the style requirements (see section 10.3).

## Running and debugging

Bugs will often exist in a newly written program. Never, ever, believe or assume that the code you just created, works. Always check the correctness of each function or script twice. You may add some extra lines in your code to present intermediate results (screen displays, plots, write actions to files) to help you controlling what is going on. These lines can be removed later.

## Testing and verification

After the debugging process, the testing stage starts. Prepare a number of tests to verify whether your program does what it is expected to do. Remember: the good tests are those for which the answers are known. Your program should produce correct results both for normal and extreme test values.

## Maintenance

In solving your task, new ideas or problems may appear. Some can be interesting and creative and some can help you to understand the original problem better; you may see an extent to your problem or a way to incorporate new things. If you have a well-designed program, you will be able to easily modify it after some time. Don't forget to improve your solutions or correct your errors when found later.

## Debugging

Debugging is the process by which you isolate and fix any problem with your code. Two kinds of errors may occur: syntax error and runtime error. Syntax errors can usually be easily corrected using MATLAB error messages. Runtime errors are algorithmic in nature and they occur when, e.g., you perform a calculation incorrectly. They are usually difficult to track down, but they are apparent when you notice unexpected results. Debugging is an inevitable process. The best way to reduce the possibility of making a runtime error is by **defensive programming**, given by:

- Do not assume that input is correct, simply build in checks to that end;
- Where reasonable and possible, provide a default option or value;
- Provide diagnostic error messages;
- Optionally, print intermediate results to check the correctness of your code.

Defensive programming is just a part of the early debugging process. Another important part is modularity, breaking large tasks into small subtasks, which allow to develop tests for each of them more easily. You should not forget to run the test again after the changes have been made. To make this easy, provide extra print statements that can be turned on or off.

MATLAB versions  $\geq 5.x$  provide an interactive debugger. It allows you to set and clear breakpoints, specific lines in an m-file at which the execution halts. It also allows you to change the workspace and execute the lines in an m-file one by one. The Matlab m-file editor also has a debugger. The debugging process can be also done from the command line. This command provides a number of options for stopping execution of a function. A particular useful option is:

```
dbstop if error
```

This stops any function causing an error. Then just run the MATLAB function. Execution will stop at the point where the error occurs, and you will get the MATLAB prompt back so that you can examine one line at a time. You can continue execution with the `dbcont` command. To exit debug mode, type `dbquit`. For more information, use `help` for the following topics: `dbstop`, `dbclear`, `dbstep`, `dbtype`, `dbup` and `dbquit`.

## 10.2 Recommended programming style

Programming style is a set of conventions that programmers follow to standardize their code to some degree and to make the overall program easier to read and to debug.

1. You should always **comment** difficult parts of the program! But ... do not explain the obvious.
2. Comments describing tricky parts of the code, assumptions, or design decisions are suggested to be placed above the part of the code you are attempting to document. Try to avoid big blocks of comments except for the description of the m-file header.

3. Indent a few spaces (preferably 2 or 3) before lines of the code and comments inside the control flow structures. The layout should reflect the program 'flow'. Here is an example:

```
x = [0:0.1:500];
for k = 1:length(x)
    if x(k) > 0
        s(k) = sqrt(x(k));
    else
        s(k) = 0;
    end
end
```

4. Avoid the use of magic numbers; use a constant variable instead. When you need to change the number, you will have to do it only once, rather than searching all over your code. An example:

% A BAD code that uses	% This is the way it SHOULD be
% magic numbers	
r = rand(1,50);	n = 50; % number of points
for k = 1:50	r = rand(1,n);
data(k) = k * r(k);	data = sum(1:n) .* r;
end	
y = sum(data)/50;	avr = sum(data)/n;
disp(['Number of points is 50.']);	disp(['Number of points is '... ,int2str(n)]);

5. Avoid the use of more than one code statement per line in your script or function m-files.
6. No line of code should exceed 80 characters (it is a rare case when this is impossible).
7. Avoid declaring global variables. You will hardly ever encounter a circumstance under which you will really need them. Global variables can get you into trouble without your noticing it!
8. Variables should have meaningful names. You may use the standard notation, e.g. x, y are real valued, ii, jj, k are indices and n and m are integers (because i and j are used for the imaginary part of complex numbers rather use ii and jj for indices). This will reduce the number of comments and make your code easier to read. However, here are some pitfalls when choosing variable names:
- A meaningful variable name is good, but when it gets longer than 15 characters, it tends to obscure rather than improve the code readability.
  - Be careful with names since there might be a conflict with MATLAB's built-in functions, or reserved names such as `mean`, `end`, `sum` etc. (check in the manual's index or ask which `<name>` in MATLAB - if you get the response `<name> not found` it means that you can safely use it).
  - Avoid names that look similar or differ only slightly from each other.
9. Use (blank) spaces, both horizontally and vertically, since it will greatly improve the readability of your program. Blank lines should separate larger blocks of code.
10. Test your program before submitting it. Do not just assume it works.