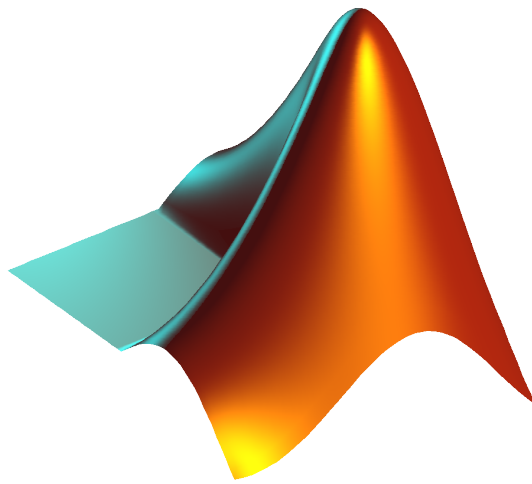


Programming with MATLAB



Elżbieta Pękalska (Delft University of Technology)
Gjerrit Meinsma (University of Twente)
Hans Zwart (University of Twente)

December
2018

This manual was prepared in the years 2001–2005 by Elżbieta Pękalska. At the TU Delft it was used to teach students of Physics both programming (at least some concepts of it) and MATLAB. At that time, it was based on the book "The Student Edition of MATLAB". The manual is, however, written to be self-contained. Marjolein van der Glas took part in the preparation of the first version. In the winter of 2005–2006 Hans Zwart added an index and a chapter on symbolic computation. This version was used for a MATLAB course for Technical Medicine students at the University of Twente. In September 2006 some typos were removed. In 2014 Gjerrit Meinsma revamped the material and partly rewrote it and he replaced the by now outdated inline functions with the anonymous functions. In 2015 only minor changes were made.

Contents

Introduction	v
1 Getting started with MATLAB	1
1.1 Input via the command-line	2
1.2 Help-facilities	4
1.3 Interrupting a command or program	5
1.4 Path	5
1.5 Workspace issues	5
1.6 Saving and loading data	6
1.7 Script m-files	7
2 Basic syntax and variables	9
2.1 MATLAB as a calculator	9
2.2 An introduction to floating-point numbers (optional)	10
2.3 Assignments and variables	11
3 Vectors and matrices	15
3.1 Vectors	15
3.1.1 Colon notation and retrieving parts of a vector	16
3.1.2 Column vectors, complex vectors and the transpose	16
3.1.3 Element-wise operations	17
3.2 Matrices	21
3.2.1 Special matrices	22
3.2.2 Building matrices and retrieving parts of matrices	23
3.2.3 Operations on matrices	26
3.3 Multi-dimensional arrays	31
4 Visualization	33
4.1 Simple 2D plots	33
4.2 Several functions in one figure	35
4.3 Plotting in the complex plane	37
4.4 Other 2D plotting features	37
4.5 Printing & saving figures	38
4.6 3D line plots	38
4.7 Plotting surfaces in 3D	39
4.8 Animations	42
5 Logicals and loops	43
5.1 Logical and relational operators	43
5.2 The command find	46
5.3 Conditional code execution	47
5.4 Loops	50
5.5 Evaluation of logical and relational expressions in the control flow structures . .	53

6	Functions	55
6.1	Anonymous functions	55
6.2	Function m-file	57
6.3	Subfunctions	59
6.4	Special function variables	60
6.5	Local and global variables	61
6.6	Passing functions to functions	61
6.7	Scripts vs. functions vs. anonymous functions	63
6.8	Recursion	65
7	Numerical analysis	67
7.1	Curve fitting	67
7.2	Interpolation	68
7.3	Evaluation of functions	69
7.4	Integration and differentiation	69
7.5	Numerical computations and the control flow structures	70
7.6	Numerical solution of differential equations	71
8	Text	73
8.1	Character strings	73
8.2	Text input and output	75
9	Cell arrays and structures	79
9.1	Cell arrays	79
9.2	Structures	80
9.3	Classes and object oriented programming	82
10	Symbolic computation	83
10.1	Symbolic objects	83
10.2	Solving symbolic expressions	84
10.3	Solving ordinary differential equations	85
10.4	From symbolic function to function handle	86
11	Optimizing the performance of MATLAB code	87
11.1	Vectorization — speed-up of computations	87
11.2	Array preallocation	88
11.3	MATLAB tricks and tips	89
12	File input/output operations	95
12.1	Text files	96
12.2	Binary files	97
13	Writing and debugging MATLAB programs	101
13.1	Structural programming	101
13.2	Debugging	103
13.3	Recommended programming style	104

Introduction

In this course you will learn how to use MATLAB, to design, and to perform mathematical computations. You will also get acquainted with basic programming. If you learn to use this program well, you will find it very useful in the future, since many technical and mathematical problems can be solved using MATLAB.

This text includes all material (with some additional information) that you need for this course, however, many things are treated briefly. Keep in mind that MATLAB has been around for several decades now and over the years many users have contributed specialized routines and toolboxes. To name but a few, one can nowadays analyze and listen to sound files and play with pictures and movies. Such applications go beyond this introductory course, but this course should put you on the right track, and perhaps one day you will develop your own professional MATLAB toolbox.

Please test after each section whether you have sufficient understanding of the issues discussed. Use the lists provided below.

Chapters 1–2. You should be able to:

- recognize built-in variables;
- define variables and perform computations using them;
- perform basic mathematical operations;
- know how to suppress display with ; (semicolon);
- use the format command to adjust the Command window output;
- add and remove variables from the workspace; check which variables are currently present in the workspace;
- use on-line help to get more information on a command and know how to use the lookfor command;
- use the load and save commands to read/save data to a file;
- access files at different directories (manipulate path-changing commands);
- create and edit script m-files

Chapter 3. You should be able to:

- create vectors and matrices with direct assignment (using []);
- 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;

- apply transpose operators to vectors and matrices;
- perform legal addition, subtraction, and multiplication operations on vectors and matrices;
- understand the use of element-wise operators, such as `.*`, `./` and know how they differ from the regular `*`, `/` operators;
- delete elements from vectors and matrices;
- compute inner products and the Euclidean length of vectors;
- create and manipulate complex vectors and matrices.

Chapter 4. You should be able to:

- 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 a legend, describe the axes, change the range of axes;
- use logarithmic axes;
- make simple 3D line plots;
- plot surfaces, contours, change colors;
- save figures to files;
- *optional*: make some fancy plots;
- *optional*: create MATLAB animations.

Chapter 5. You should be able to:

- use relational operators: `<`, `<=`, `>`, `>=`, `==`, `~=` and logical operators: `&`, `|`, `~` and `&&` and `||`;
- understand the logical addressing;
- fully understand how to use the command `find`, both on vectors and matrices;
- 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.

Chapter 6. You should be able to:

- create anonymous functions;
- edit and run an function m-files;
- identify the differences between scripts, functions and anonymous 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 functions;
- know how to pass a functions as an input argument to another function via pointers @;
- understand what recursion is and know when to use it and when not.

Chapter 7. You should be able to:

- create and manipulate MATLAB polynomials;
- fit a polynomial to data;
- interpolate the data;
- evaluate a function;
- integrate and differentiate a function;
- solve ordinary differential equations;

Chapter 8. You should be able to:

- 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 etc;
- use freely and with understanding the text input/output commands: input, disp and fprintf;

Chapter 9. You should be able to:

- operate on cell arrays and structures;

Chapter 10. You should be able to:

- Create symbolic objects and expressions;
- Solve expressions symbolically;
- Solve differential equations symbolically and plot them.

Chapter 11. You should be able to:

- *optional*: preallocate 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.

Chapter 12. You should be able to:

- perform low level input and output with `fopen`, `fscanf` and `fclose`;
- understand how to operate on text files (input/output operations);
- get more understanding on the use of `fprintf` while writing to a file;
- *optional*: understand how to operate on binary files (input/output operations);

Chapter 13. You should be able to:

- 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.

Preliminaries

Below you find a few basic definitions on computers and programming. Please get acquainted with them since they introduce key concepts needed in the coming sections:

- A *bit* (short for *binary digit*) is the smallest unit of information on a computer. A single bit can hold only one of two values: 0 or 1. More meaningful information is obtained by combining consecutive bits into larger units, such as byte.
- A *byte* – a unit of 8 bits, being capable of holding a single character. Large amounts of memory are indicated in terms of kilobytes (1024 bytes), megabytes (1024 kilobytes), and gigabytes (1024 megabytes).
- *Binary system* – a number system that has two unique digits: 0 and 1. Computers are based on such a system, because of its electrical nature (charged versus uncharged). Each digit position represents a different power of 2. The powers of 2 increase while moving from the right most to the left most position, starting from $2^0 = 1$. Here is an example of a binary number and its representation in the decimal system:

$$10110 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0 = 16 + 0 + 4 + 2 + 0 = 24.$$

Because computers use the binary number system, powers of 2 play an important role, e.g. 8 (= 2^3), 64 (= 2^6), 128 (= 2^7), or 256 (= 2^8).

- *Data* is information represented with symbols, e.g. numbers, words, signals or images.
- A *command* is a instruction to do a specific task.
- An *algorithm* is a sequence of instructions for the solution of a specific task in a finite number of steps.
- A *program* is the implementation of an algorithm suitable for execution by a computer.
- A *variable* is a container that can hold a value. For example, in the expression: $x+y$ both x and y are variables. They can represent numeric values, like 25.5 but also characters, like 'c' or character strings, like 'hello'. Variables make programs more flexible. When a program is executed, the variables are then *replaced* with real data. That is why the same program can process different sets of data.

Every variable has a name (called the *variable name*) and a *data type*. A variable's data type indicates the sort of value that the variable represents (see below).

- A *constant* is a value that never changes. That makes it the opposite of a variable.
- A *data type* is a classification of a particular type of information. The most basic data types are:
 - *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 1.2×10^{-16} (this stands for 1.2×10^{-16}).
 - *character*: readable text character, e.g. 'p'.
- A *bug* is an error in a program, causing the program to stop running, not to run at all or to provide wrong results. Some bugs can be very subtle and hard to find. The process of finding and removing bugs is called *debugging*.
- A *file* is a collection of data or information that has a name, stored in a computer. There are many different types of files: data files, program files, text files etc.
- An *ASCII* is a widely used character encoding scheme. It is an abbreviation of *American Standard Code for Information Interchange* and initially it favored English characters. ASCII files are plain text files that are readable and editable by text editors and by many other programs including WORD.
- A *binary file* is a file stored in a format, which is computer-readable but not human-readable. Most numeric data and all executable programs are stored in binary files. MATLAB binary files are those with the extension '*.mat'.
- A *directory* is a group of files and other directories. Operating systems with graphical user interface, such as Mac OS X and Microsoft Windows, call it “map” or “folder”.

Chapter 1

Getting started with MATLAB

MATLAB is a tool for mathematical calculations. It can be used as a *scientific calculator*, and it allows you to *visualize* data in many different ways, perform *matrix* algebra, work with polynomials and integrate functions and much, much more. In MATLAB you can create, execute and save a sequence of commands. It is also a full-fledged yet user-friendly *programming language*, which gives you the possibility to perform intricate mathematical calculations in a structured fashion. MATLAB especially facilitates easy manipulation of arrays of data, such as vectors, matrices and images.

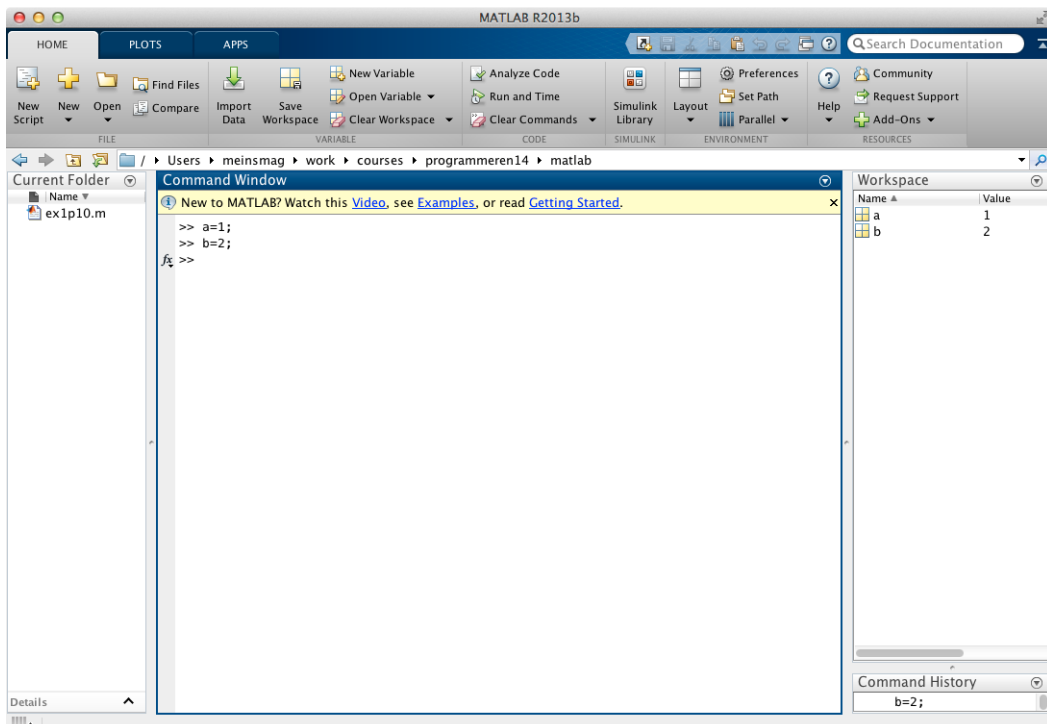


FIGURE 1.1: A MATLAB desktop. Here the *current folder* is on the left. The *command window* is in the middle, and on the right is the *workspace* and *command history*

Opening MATLAB creates a desktop with one or more windows, see Fig. 1.1. The most important is the *Command Window*, which is where you interact with MATLAB, i.e. where you enter commands and where MATLAB displays results. The string `>>` is the MATLAB prompt. When the Command Window is active, a cursor appears after the prompt, indicating that MATLAB is waiting for your input.

To exit MATLAB use the command `exit` or `quit`.

TABLE 1.1: Some mathematical notation and its MATLAB equivalent

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

1.1 Input via the command-line

MATLAB is an interactive system; commands followed by Enter are executed immediately. The results are, if desired, displayed on screen. Table 1.1 contains a list of some elementary MATLAB commands and their mathematical notation (a, b, x and y are numbers). Below you find basic information to help getting started with MATLAB.

- Commands in MATLAB are executed by pressing Enter or Return. The output will be displayed on screen. Try the following (hit Enter after the end of line):

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

You see that spaces between numbers are *not* important in MATLAB.

- The result of the last un-assigned computation is stored in the variable ans. This ans is an example of a MATLAB built-in variable. It can be used in the subsequent command. For instance:

```
>> 14/4
ans =
    3.5000
>> ans^(-6)
ans =
    5.4399e-04
```

5.4399e-04 is a computer notation of 5.4399×10^{-4} . Note that ans is overwritten by the last command.

- You can define your own variables. Look how the information is stored in the variables a and b:

```
>> a = 14/4
a =
```

```

3.5000
>> b = a^(-6)
b =
5.4399e-04

```

Read **Preliminaries** to better understand the concept of variables. You will learn more on MATLAB variables in Section 2.3. The outcome of these two commands are *not* stored in ans because they are already stored (in a and in b).

- If the command is followed by a semicolon ;, the output is suppressed. Check the difference between the following expressions:

```

>> 3 + 7.5
>> 3 + 7.5;

```

- It is possible to execute more than one command at the same time. The commands should then be separated by commas (to display the output) or by semicolons (to suppress the output display), e.g.:

```

>> sin(pi/4), cos(pi); sin(0)
ans =
    0.7071
ans =
    0

```

Here the value of $\cos(\pi)$ is not printed.

- By default, MATLAB displays only 5 digits even though it computes to some 15 digits. The command `format long` increases the display to 15, and `format short` returns it to 5. For instance:

```

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

```

- 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 statement that is too long to be typed on one line, 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 =
    1.0357

```

- MATLAB is case sensitive, so a and A are two different variables in MATLAB;
- All text after a percent sign % until the end of a line is treated as a comment. Enter e.g. the following:

```
>> sin(3.14159)           % this is an approximation of sin(pi)
```

You will notice that some examples in this text are followed by comments. They are meant for you, and you should not type them while entering the commands.

- Previous commands can be fetched back with the `↑`-key. The command can also be changed, the `←` and `→`-keys may be used to move around in a line and edit it. In case of a long line, `Ctrl-a` and `Ctrl-e` might be useful; they move the cursor to the beginning and the end of the line, respectively.
- To recall the most recent command starting with, say, `c`, type `c` at the prompt followed by the `↑`-key. Similarly, `cos` followed by the `↑`-key will bring you to the last command starting with `cos`.
- If you type `cos` and hit the `Tab` key then MATLAB will give you a list of all variables *and all commands* it knows that begin with `cos`. This is useful if you wonder about the existence of certain commands. Now MATLAB has thousands of commands so you should be careful here.

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 1000×1000 matrix!

1.2 Help-facilities

MATLAB provides assistance through extensive online 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 specified help. For instance:

```
>> help ops
```

gives information on the operators and special characters in MATLAB. The topic 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`. Besides 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 1.1. Use `help` or `lookfor` to find out the following:

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


Exercise 1.2. Type `help` to see which toolboxes are installed along with your copy of MATLAB. What command is needed to get help on the toolbox `matlab/matfun`? □

Fancier help is available with `doc`.

1.3 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-C` (or `Ctrl-Break` on PC) forces MATLAB to stop the process. Sometimes, however, you may need to press it a few times. After this the MATLAB prompt (`>>`) re-appears. This may take a while, though.

1.4 Path

In MATLAB, commands and programs are contained in “m-files”, which are plain text files that have extension `.m`. The m-file must be located in one of the directories that MATLAB searches. The list of these directories can be obtained 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`. By clicking the icon  it is easy to change the current folder. Use `path`, `addpath` and `rmpath` functions to modify the path. Instead it is also possible to access the path browser from `Set Path` in the menu bar.

Exercise 1.3. Type `path` to check which directories are placed on your path. For this MATLAB course you probably want to create a new directory called, say, `MatlabCourse`. Create such a directory and then add this directory to the path. □

1.5 Workspace issues

If you work in the Command Window, MATLAB retains all commands that you entered and all variables that you created. These commands and variables are said to reside in the MATLAB *workspace*. To recall a previous command use the `↑`-key. Variables can be verified with the commands `who`, which gives a list of variables present in the workspace, and `whos`, which 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, while `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
```

Note that you cannot use comma after a variable, i.e. `clear a, x`, as it will be interpreted as `clear a` followed by the command `x` (which prints the value of `x` on screen). See what the result is when you do:

```
>> clear all
>> a = 1;
>> b = 2;
>> c = 3;
>> clear a, b, c
```

1.6 Saving and loading data

The easiest way to save or load MATLAB variables is by selecting **Save Workspace** or **Import Data** from the menu-bar. This can also be done from the Command Window with the commands `save` and `load`. The command `save` allows for saving your workspace variables either into a binary file or an ASCII file (check **Preliminaries** on page ix on binary and ASCII files). Binary files automatically get the `.mat` extension, which is not true for ASCII files. However, it is recommended to add a `.txt` or `.dat` extension to ASCII files to emphasise that they are readable by plain text editors and Word and many other programs.

Exercise 1.4. Learn how to use the `save` command by exercising:

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

□

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

Exercise 1.5. Assuming that you have 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 only specified variables from the file data.mat
>> load strdata            % loads all variables from the file strdata.mat
```

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

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

□

1.7 Script m-files

The easiest way to save your MATLAB commands is to store them in an external file called *script m-file*. These files must have extension `.m`. To create a script you need to open an editor, enter all commands needed, save it with the extension `.m` (e.g. `mytask.m`) and then run it from the Command Window, by typing `mytask`. To open the MATLAB editor to make a new script select `New Script` from the menu bar or simply type `edit` in the Command Window. To open an existing script use `Open` from the menu bar or type `edit mytask` (assuming the script is called `mytask.m`). All commands in the script will be executed in MATLAB when you enter the name of the file without the extension (e.g. `mytask`). The m-script file must be saved in one of the directories in MATLAB's path so that MATLAB can find it.

As an example, say you created the file `short.m` with these two commands:

```
x = 0:0.5:4;
plot(x,sin(x),'*-');
```

and that you saved the file somewhere in your MATLAB path. Typing `short` will then execute the two commands. Another example: open the editor and type in the editor:

```
disp('A very nice day to you')
```

Next save this file under the name `goodbye.m`. Now type `goodbye` in the command window. If everything goes well, then MATLAB answers with a friendly `A very nice day to you`. If Matlab complains with

```
Undefined function or variable 'goodbye'.
```

then MATLAB cannot find the file. You can fix it by changing the current folder.

M-files are *very* useful when the number of commands increases or when you want to change values of some variables and re-evaluate them quickly.

Exercise 1.6. type `edit sinplot.m`. Then MATLAB's Editor Window will appear. Enter the lines listed below and save the file as `sinplot.m`:

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

and then run it by typing:

```
>> sinplot
```

It makes a plot with title. The `sinplot` script affects the workspace. Check:

```
>> clear           % all variables are removed from the workspace
>> who            % no variables present
>> sinplot
>> who
Your variables are:
   x   y
```

□

Be aware that all commands in a script have access to all variables in the workspace and all variables created in this script become a part of the workspace. Scripts create and change

variables in the workspace without warning. You therefore have to be careful when using generic single-letter variables, such as x and y .

Exercise 1.7. Write a script `secondsage` that asks the user for her age in years and then displays her age in seconds. Hint: do `help input` and do not worry about leap years and such. □

Bugs are inevitable, no matter how clever you are.

Exercise 1.8. Write a script `ex1p8.m` with an illegal command in line 3:

```
a=1;
b=2;
c=3/; % yes, this is wrong
d=4
```

Execute the code. MATLAB will complain that something is wrong. Now the good part: move the mouse over to the underlined text of the complaint and click on it. □

A final remark: you must add comments/explanations to the scripts that you write. Without comments your code will be virtually useless within a month because by then you will have forgotten most of the details that seemed so obvious to you when you wrote the code.

For the remaining exercises in this course follow this naming convention: save all commands for, say, Exercise 3.10 in an m-file called `ex3p10.m` and add comments to the m-file. These m-files are needed for the assessment of your work.

Chapter 2

Basic syntax and variables

2.1 MATLAB as a calculator

There are three kinds of numbers in MATLAB: integers, real numbers and complex numbers. In addition, MATLAB has representations for some non-numbers:

- Inf which denotes positive infinity, generated e.g. by $1/0$,
- NaN which means Not-a-Number, obtained as a result of mathematically undefined operations such as $0/0$ or $\text{Inf} - \text{Inf}$.

You have already a bit of experience with MATLAB and you know that it can be used as a calculator. For example, if you type

```
>> (23*17)/7
```

then MATLAB returns

```
ans =  
55.8571
```

MATLAB has six basic arithmetic operations, such as: +, -, *, / or \ (left division) and ^ (exponentiation). Note that the two division operators are different:

```
>> 19/3          % mathematically: 19/3  
ans =  
6.3333  
>> 19\3         % mathematically: 3/19  
ans =  
0.1579
```

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

Exercise 2.1. Evaluate the following expressions by hand and use MATLAB to check the answers. Note the difference between the left and right divisors. Use help to learn more on commands rounding numbers, such as: `round`, `floor`, `ceil`, etc.

1. $2/2*3$

4. $7-5*49$

2. $8*54$

5. $6-2/5+7^2-1$

3. $8*(54)$

6. $10/25-3+2*4$

- | | |
|----------------------------------|--|
| 7. $3^{2/4}$ | 12. $x=\pi/3; x=x-1; x=x+5; x=\text{abs}(x)/x$ |
| 8. 3^{2^3} | 13. $1/\text{Inf}$ |
| 9. $2 + \text{round}(6/9+3*2)/2$ | 14. $0*\text{Inf}$ |
| 10. $2+\text{floor}(6/9+3*2)/2$ | 15. $\text{Inf}*\text{Inf}$ |
| 11. $2+\text{ceil}(6/9+3*2)/2$ | |

□

Exercise 2.2. Define 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. Use the command `format` to put MATLAB in standard-format. □

2.2 An introduction to floating-point numbers (optional)

In a computer, numbers can be represented only in a *discrete* form. It means that numbers are stored within a limited range and with a finite precision. Integers can be represented *exactly* with the base of 2 (read **Preliminaries** on page viii on bits and the binary system). The typical size of an integer is 32 bits, so the largest positive integer, which can be stored, is $2^{32} = 4294967296$. If negative integers are permitted, then 32 bits allow for representing integers between $\pm 2^{15} = \pm 2147483648$. Within this range, operations defined on the set of integers can be performed exactly. The maximal integer value that MATLAB can handle is `intmax`.

However, this is not valid for other real numbers. In practice, computers are integer machines and are capable of representing real numbers only by using complicated codes. The most popular code is the *floating point* standard. The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point, meaning that the decimal point can float. Note that most floating-point numbers that a computer can represent are just approximations. Therefore, care should be taken that these approximations lead to reasonable results. If a programmer is not careful, small discrepancies in the approximations can cause meaningless results. Note the difference between e.g. the integer arithmetic and floating-point arithmetic:

Integer arithmetic:

$$2 + 4 = 6$$

$$3 * 4 = 12$$

$$25/11 = 2$$

Floating-point arithmetic:

$$18/7 = 2.5714$$

$$2.5714 * 7 = 17.9998$$

$$10000/3 = 3.3333e+03$$

When describing floating-point numbers, precision refers to the number of bits used for the fractional part. The larger the precision, the more exact fractional quantities can be represented. Floating-point numbers are often classified as single precision or double precision. A double-precision number uses twice as many bits as a single-precision value, so it can represent fractional values much better. However, the precision itself is not double. The extra bits are also used to increase the range of magnitudes that can be represented.

MATLAB relies on a computer's floating point arithmetic. You will have noticed that in the last exercise the value of $\sin(\pi)$ was close to zero, but not exactly zero. This is due to the fact

that the value of π is represented with a finite precision and that the sine function is evaluated with finite precision.

The fundamental type in MATLAB is `double`, which stands for a representation with a double precision. It uses 64 bits. The single precision obtained by using the `single` type offers 32 bits. Since most numeric operations require high accuracy the `double` type is used by default. This means, that when the user is inputting integer values in MATLAB (for instance, $k = 4$), the data is still stored in `double` format.

The smallest and largest floating-point number that MATLAB can handle are `realmin` and `realmax`. They are about $2 \times 10^{\pm 308}$. The *relative* accuracy can be defined as the smallest positive number ϵ that added to 1, creates the result larger than 1, i.e. $1 + \epsilon > 1$. It means that in floating-point arithmetic, for positive values smaller than ϵ , the result equals to 1 (in exact arithmetic, of course, the result is always larger than 1). In MATLAB, ϵ is stored in the built-in variable `eps` $\approx 2.2204 \times 10^{-16}$. This means that the relative accuracy of individual arithmetic operations is about 15 digits.

2.3 Assignments and variables

Working with complex numbers is easily done with MATLAB.

Exercise 2.3. Choose two complex numbers, for example $-3 + 2i$ and $5 - 7i$. Add, subtract, multiply, and divide these two numbers. \square

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 created by an assignment (e.g. `z = -3 + 2*i`), i.e. values are assigned to variables. Each newly created numerical variable is *always* of the `double` type, i.e. real numbers are approximated with the highest possible precision. You can change this type by converting it into e.g. the `single` type¹. 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` representations into e.g. `int32` integer type¹. Note that integer numbers are represented exactly, 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.

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)
```

It becomes possible by:

```
>> x = pi / 3;
```

¹a variable `a` is converted into a different type by performing e.g. `a = single(a)`, `a = int32(a)` etc.

```
>> f = x^2 + 4 * sin(x)
```

Variable names begin with a letter, followed by letters, numbers or underscores. MATLAB recognizes only first 63 characters of the name.

Exercise 2.4. 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_simple_variable_which_I_typed_in_after_a_first_course_on_typing = 5
                                     % check what happens; the name is very long
>> 2t = 8                             % what is the problem with this command?
>> t2 = 8                             % This is allowed
>> 2*t = 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 using the command `whos`. Use `clear <name>` to remove a variable from the workspace. □

As you already know, MATLAB variables can be created by an assignment. There is also a number of built-in variables, e.g. `pi`, `eps` or `i`, summarized in Table 2.1. In addition to creating variables by assigning values to them, another possibility is to copy one variable, e.g. `b` into another, e.g. `a`. In this way, the variable `a` is automatically created (if `a` already existed, its previous value is lost):

```
>> b = 10.5;
>> a = b;
```

A variable can also be created as a result of the evaluated expression:

```
>> a = 10.5;
>> c = a^2 + sin(pi*a)/4;
```

or by loading data from text or `*.mat` files.

If `min` is the name of a function (see `help min`), then `a` defined, e.g. as:

```
>> b = 5;
>> c = 7;
>> a = min(b,c);                       % create a as the minimum of b and c
```

will call that function, with the values `b` and `c` as parameters. The result of this function (its return value) will be written (assigned) into `a`. So, variables can be created as results of the execution of built-in or user-defined functions. You will learn more about functions in Section 6.2.

Important: do not use variable names that are defined as function names (for instance `mean` or `error`)²! If you intend to use a suspicious variable name, use `help <name>` to find out if the function already exists.

²There is always one exception of the rule: variable `i` is often used as counter in a loop, while it is also used as $i = \sqrt{-1}$.

TABLE 2.1: Built-in variables in MATLAB

Variable name	Value/meaning
ans	the default variable name used for storing the last unassigned result
pi	$\pi = 3.141592653589793..$
eps	the smallest positive number that added to 1 makes a result larger than 1
Inf	representation for positive infinity, e.g. $1/0$
nan or NaN	representation for not-a-number, e.g. $0/0$
i or j	$i = j = \sqrt{-1}$
nargin/nargout	number of function input/output arguments used
realmin/realmax	the smallest/largest usable positive real number: 2.2251×10^{-308} / 1.7977×10^{308}

Chapter 3

Vectors and matrices

The principal data type in MATLAB is the *array*, and among these the one-dimensional arrays (aka *vectors*) and two-dimensional arrays (aka *matrices*) and numbers are the most common.

3.1 Vectors

Row vectors are easily defined in MATLAB by listing the entries separated by commas or spaces such as $v=[11\ 12\ 13]$ or $v=[11,12,13]$. The number of entries of a vector is known as the *length* of the vector. Entries of a vector are also referred to as *elements* or *components*.

```
>> v = [-1 sin(3) 7]
v =
   -1.0000    0.1411    7.0000
>> length(v)
ans =
     3
```

A vector can be multiplied by a scalar (a number) or added/subtracted to/from another vector of *the same* length, or a number can be added/subtracted to/from a vector. All these operations are carried out element-by-element. Vectors can also be assembled from 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]
t =
    -2     4    14    -2    -3    -4
```

Using indices one can retrieve, change or display entries:

```
>> v(2) = -3         % change the 2nd element of v, and display the new v
v =
    -1    -3     7
>> w(2)             % display the 2nd element of w
ans =
     3
```

3.1.1 Colon notation and retrieving parts of a vector

With the colon notation we can define row vectors of equidistantly spaced numbers (see Table 3.1 and do `help colon` to learn more):

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

More general, `first:step:last` is the row vector whose first entry is `first`, whose second entry is `first+step`, et cetera, until it reaches `last`:

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

Segments of vectors are easily retrieved using the colon notation:

```
>> r = [-1:2:6, 2, 3, -2]
r =
    -1     1     3     5     2     3    -2
>> r(3:6)           % the elements of r at positions 3,4,5,6
ans =
     3     5     2     3
>> r(1:2:5)         % the elements of r at positions 1, 3 and 5
ans =
    -1     3     2
>> r([1,1,6,7])     % yes, you can also list the indices one by one
ans =
    -1    -1     3    -2
```

3.1.2 Column vectors, complex vectors and the transpose

To create *column* vectors, simply separate the entries by a new line or semicolon `'`;':

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

The operations that we introduced for row vectors, such as addition and indexing, works the same for column vectors. It is also possible to turn a row vector into a column vector, and the other way round. In mathematics this operation is called *transposition*. In MATLAB we simply include an ' at the end of the expression:

```
>> u'                % if u is a column vector then u' is a row vector
ans =
    -1     3     5
>> v = [-1 2 7];    % v is a row vector
>> u' + v           % row + row = row
ans =
    -2     5    12
>> u + v'          % column + column = column
ans =
    -2
     5
    12
```

If z is a complex vector, then z' is the *complex conjugate transpose* of z . Pay particular attention to the signs of the imaginary parts. For instance:

```
>> z = [1+2i, -1+i]
z =
    1.0000 + 2.0000i   -1.0000 + 1.0000i
>> z'                % this is the complex 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
```

3.1.3 Element-wise operations

MATLAB has the very useful *element-wise* product. For two row vectors x and y of equal length, the element-wise product $x.*y$ is the row vector $[x_1y_1 \ x_2y_2 \ \dots \ x_ny_n]$. Likewise for column vectors:

```
>> u = [-1; 3; 5]    % a column vector
>> v = [-1; 2; 7]    % a column vector
>> u .* v           % this is an element-by-element multiplication
     1
     6
    35
```

You can now easily tabulate the values of a function for a given array 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
```

Similarly, $./$ means *element-wise* division, and $.^$ element-wise exponentiation.

```

>> 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
Warning: Divide by zero.
ans =
   -2.0000         Inf    6.0000    4.0000    3.3333
% division 4/0, resulting in Inf
>> z./z
Warning: Divide by zero.
ans =
     1   NaN     1     1     1
% division 0/0, resulting in NaN
>> z.^2
ans =
     1     0     1     4     9
>> z.^z
ans =
    -1     1     1     4    27
% remember that z=[-1 0 1 2 3]
%
% yes: 0^0 is considered 1!

```

The operator `./` can also be used to divide a scalar by a vector:

```

>> x=1:5;
>> 2/x
??? Error using ==> /
Matrix dimensions must agree.
% this is not possible
>> 2./x
ans =
    2.0000    1.0000    0.6667    0.5000    0.4000
% but this is!

```

Yet another useful property is that we can add a number to vector. It adds the number to every element of the vector:

```

>> u = [-1 3 5]
ans =
    -1     3     5
% a row vector
>> u+10
ans =
     9    13    15

```

Exercise 3.1.

1. Create a vector consisting of the even numbers between 21 and 99.
2. Given $x = [2 \ 1 \ 3 \ 7 \ 9 \ 4 \ 6]$, explain what the following commands do (we have not explained what `end` means but you can guess it):

- `x(3)`
- `x(1:7)`
- `x(1:end)`
- `x(1:end-1)`
- `x(2:2:6)`
- `x(6:-2:1)`
- `x(end-2:-3:2)`
- `sum(x)`
- `mean(x)`
- `min(x)`

3. Given a vector t , determine the MATLAB expression that computes the vector defined by the formula

- $\ln(2 + t + t^2)$
- $\cos(t)^2 - \sin(t)^2$
- $e^t(1 + \cos(3t))$
- $\tan^{-1}(t)$.

4. Create a vector x with the elements:

- $\{2, 4, 6, 8, \dots, 16\}$
- $\{9, 7, 5, 3, 1, -1, -3, -5\}$
- $\{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}\}$
- $\{0, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \dots, \frac{99}{100}\}$

5. Create a vector x with the elements: $x_n = \frac{(-1)^n}{2n-1}$ for $n = 1, 2, 3, \dots, 200$. Find the sum of the first 50 elements x_1, \dots, x_{50} .

6. Let $x = 20:10:200$. Create a vector y of the same length as x such that

- $y_i = x_i - 3$ for each entry.
- $y_i = x_i$ for every even index i and $y_i = x_i + 11$ for every odd index i .
- $y_i = \sqrt{x_i}$ for every index i .
- $y_i = x_i^3$ for $i = 1, 4, 9, 16$ and for all other indices $y_i = x_i$.

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

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

□

TABLE 3.1: Manipulation of (groups of) matrix elements. Here i, j, k, l, m, n are positive integers

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

TABLE 3.2: Frequently used matrix operations and functions

Command	Result
$C = A + B$	sum of two matrices
$C = A - B$	subtraction of two matrices
$C = A * B$	multiplication of two matrices
$C = A .* B$	'element-by-element' multiplication (A and B are of equal size)
$C = A^k$	power of a matrix ($k \in \mathbb{Z}$; can also be used for A^{-1})
$C = A.^k$	'element-by-element' power of a matrix
$C = A'$	the (complex conjugate) transpose of a matrix; (A^T if real)
$C = A ./ B$	'element-by-element' division (A and B are of equal size)
$X = A \setminus B$	the solution in the least squares sense to the equation $AX = B$
$X = B / A$	the solution of $XA = B$, analogous to the previous command
$C = \text{inv}(A)$	C becomes the inverse of A
$n = \text{rank}(A)$	n becomes the rank of matrix A
$x = \text{det}(A)$	x becomes the determinant of matrix A
$x = \text{size}(A)$ $[m,n] = \text{size}(A)$	x becomes a row-vector of 2 elements: the number of rows and columns of A m and n are the number of rows and columns of A
$x = \text{trace}(A)$	x becomes the trace (sum of diagonal elements) of matrix A
$A = \text{eye}(n)$	A is an $n \times n$ identity matrix
$A = \text{zeros}(n,m)$	A is an $n \times m$ matrix with zeros (default $m = n$)
$A = \text{ones}(n,m)$	A is an $n \times m$ matrix with ones (default $m = n$)
$A = \text{diag}(v)$	results in a diagonal matrix with the elements v_1, v_2, \dots, v_n on the diagonal
$v = \text{diag}(A)$	results in a vector equivalent to the diagonal of A
$X = \text{tril}(A)$	X is lower triangular part of A
$X = \text{triu}(A)$	X is upper triangular part of A
$A = \text{rand}(n,m)$	A is an $n \times m$ matrix of elements drawn from a uniform distribution on $[0, 1]$
$A = \text{randn}(n,m)$	A is an $n \times m$ matrix of elements drawn from a standard normal distribution
$v = \text{max}(A)$ $v = \text{max}(A, [], \text{dim})$	v is a vector of the maximum values of the columns in A v is a vector of the maximum values along the dimension dim in A
$v = \text{min}(A)$ $v = \text{min}(A, [], \text{dim})$	ditto - with minimum
$v = \text{sum}(A)$ $v = \text{sum}(A, \text{dim})$	ditto - with sum
$v = \text{mean}(A)$ $v = \text{mean}(A, \text{dim})$	ditto - with mean

TABLE 3.3: Advanced matrix operations and functions

Command	Result
<code>C = null(A)</code>	C is an orthonormal basis for the null space of A obtained from the singular value decomposition
<code>C = orth(A)</code>	C is an orthonormal basis for the range of A
<code>C = rref(A)</code>	C is the reduced row echelon form of A
<code>L = eig(A)</code>	L is the vector of the (possibly complex) eigenvalues of matrix A
<code>[Q,L] = eig(A)</code>	produces a diagonal matrix L of eigenvalues and a full matrix Q whose columns are the corresponding eigenvectors of a square matrix A
<code>S = svd(A)</code>	S is a vector containing the singular values of a rectangular matrix A
<code>[U,S,V] = svd(A)</code>	S is a diagonal matrix with the singular values of A on the diagonal (in decreasing order); the columns of U and V are the corresponding singular vectors of A
<code>x = norm(v)</code>	x is the Euclidean length of vector v
<code>x = linspace(a,b,n)</code>	x is the vector of n equally spaced points from a to b
<code>x = logspace(a,b,n)</code>	x is a vector starting at 10^a , ending at 10^b containing n values

3.2 Matrices

An $n \times k$ matrix is a two-dimensional array of numbers having n rows and k columns. Entering a matrix element-by-element is similar to that of vectors. Commas or spaces are used to separate elements in a row, and semicolons are used to switch to the next row. For example, the matrix

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

can be entered as follows

```
>> 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:

```
>> 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
```

To MATLAB a row vector is a $1 \times n$ 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 round. Similarly, the transpose of a matrix turns *all* the rows into columns, in following way:

```
>> A2
```

```

A2 =
     1     2     3     4
    -1     1     3     5
>> A2'
ans =
     1    -1
     2     1
     3     3
     4     5
>> size(A2)
ans =
     2     4
>> size(A2')
ans =
     4     2

```

3.2.1 Special matrices

There are several special matrices in MATLAB (see Table 3.2). A few examples are given below.

```

>> E = []
E =
     []
>> size(E)
ans =
     0     0
>> I = eye(3);
I =
     1     0     0
     0     1     0
     0     0     1
>> x = [2; -1; 7];
>> I*x
ans =
     2
    -1
     7
>> r = [1 3 -2];
>> R = diag(r)
R =
     1     0     0
     0     3     0
     0     0    -2
>> A = [1 2 3; 4 5 6; 7 8 9];
>> diag(A)
ans =
     1
     5
     9
>> B = ones(3,2)
B =
     1     1
     1     1
     1     1
>> C = zeros(size(B'))
C =
     0     0     0

```



```

0 0 0
>> D = rand(2,3)           % a matrix of random numbers
D =
    0.0227    0.9101    0.9222
    0.0299    0.0640    0.3309
>> v = linspace(1,2,4)    % a vector from 1 to 2 with 4 entries
v =
    1.0000    1.3333    1.6667    2.0000

```

3.2.2 Building matrices and retrieving parts of matrices

We sometimes need to build a matrix from a number of smaller matrices. That is a piece of cake in MATLAB:

```

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

```

Parts of a matrix can be retrieved in a similar way as it is done for vectors. An element in a matrix is indexed by the row and the column to which it belongs. Mathematically, the element from the i -th row and the j -th column of the matrix A is denoted as A_{ij} ; in MATLAB this is $A(i, j)$.

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

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

```
>> A(5,2) = 5 % assign 5 to the position (5,2); all un-initialized
A = % elements of A are set to zero:
     1     2     3
     4     5     8
     7     8     9
     0     0     0
     0     5     0
```

If needed, the other zero elements of the matrix A can 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?
```

Rectangular parts of the matrix A can be retrieved:

```
>> A(3,:) % the 3rd row of A
ans =
     7     8     9
>> A(:,2) % the 2nd column of A
ans =
     2
     5
     8
     1
     5
>> A(1:2,:) % the 1st and 2nd row of A
ans =
     1     2     3
```

```

    4    5    8
>> A([3,1],1:2)           % rows 3 and 1, columns 1 and 2
ans =
    7    8
    1    2

```

As you have seen in the examples, MATLAB is efficient in manipulating (groups of) matrix-elements. An overview is given in Table 3.1. The concept of an empty matrix [] is also very useful. For instance, a group of columns or rows can be removed from a matrix by assigning an empty matrix to it.

```

>> C = [1 2 3 4; 5 6 7 8; 1 1 1 1];
>> D = C;           % now a copy of C is stored in D
>> D(:,2) = []     % remove the 2nd column from D (this does not affect C)
>> C ([1,3],:) = [] % remove the rows 1 and 3 from C

```

Exercise 3.2. Clear all variables (use `clear`). Define the matrix $A=[1:4; 5:8; 1\ 1\ 1\ 1]$. Predict and check the result of the following operations:

1. $x = A(:,3)$
2. $B = A(1:3,2:2)$
3. $A(1,1) = 9 + A(2,3)$
4. $A(2:3,1:3) = [0\ 0\ 0; 0\ 0\ 0]$
5. $A(2:3,1:2) = [1\ 1; 3\ 3]$
6. $y = A(3:3,1:4)$
7. $A = [A; 2\ 1\ 7\ 7; 7\ 7\ 4\ 5]$
8. $C = A([1,3],2)$
9. $D = A([2,3,5],[1,3,4])$
10. $D(2,:) = []$

□

Exercise 3.3. Let $T = [3\ 4; 1\ 8; -4\ 3]$ and $A = [\text{diag}(-1:2:3)\ T; -4\ 4\ 1\ 2\ 1]$. Perform the following operations on A:

1. retrieve a vector consisting of the 2nd and 4th elements of the 3rd row.
2. find the minimum of the 3rd column.
3. find the maximum of the 2nd row.
4. compute the sum of the 2nd column
5. compute the mean of the row 1 and the mean of row 4
6. retrieve the submatrix consisting of the 1st and 3rd rows and all columns
7. retrieve the submatrix consisting of the 1st and 2nd rows and the 3rd, 4th and 5th columns
8. compute the total sum of the 1st and 2nd rows
9. add 3 to all elements of the 2nd and 3rd columns

□

Exercise 3.4. Let $A = \text{rand}(5,6)$. It creates a 5×6 random matrix. Provide the commands that

1. assign the first row of A to a vector x;

2. assign the last 2 rows of A to a vector y;
3. create the column vector that equals the sum of all the columns of A;
4. create the row vector that equals the sum of all the rows of A;
5. compute the standard error of the mean of each column of A (i.e. the standard deviation divided by the square root of the number of elements used to compute the mean).

□

Exercise 3.5. Let $A = \begin{bmatrix} 2 & 7 & 9 & 7 \\ 3 & 1 & 5 & 6 \\ 8 & 1 & 2 & 5 \end{bmatrix}$. Explain the results or perform the following commands:

- | | |
|------------------------------|---|
| 1. A' | 18. $\max(A')$ |
| 2. $A(1, :)'$ | 19. $\min(A(:, 4))$ |
| 3. $A(:, [1 \ 4])$ | 20. $[\min(A)' \ \max(A)']$ |
| 4. $A([2 \ 3], [3 \ 1])$ | 21. $[[A; \text{sum}(A)] \ [\text{sum}(A, 2); \text{sum}(A(:))]]$ |
| 5. $\text{reshape}(A, 2, 6)$ | 22. $\max(\min(A))$ |
| 6. $A(:)$ | 23. assign the even-numbered columns of A to an array B |
| 7. $\text{flipud}(A)$ | 24. assign the odd-numbered rows to an array C |
| 8. $\text{fliplr}(A)$ | 25. convert A into a 4-by-3 array |
| 9. $[A; A(\text{end}, :)]$ | 26. compute the reciprocal of each element of A |
| 10. $[A; A(1:2, :)]$ | 27. compute the square-root of each element of A |
| 11. $\text{sum}(A)$ | 28. remove the second column of A |
| 12. $\text{sum}(A')$ | 29. add a row of all 1's at the beginning and at the end |
| 13. $\text{mean}(A)$ | 30. swap the 2nd row and the last row |
| 14. $\text{mean}(A')$ | |
| 15. $\text{sum}(A, 2)$ | |
| 16. $\text{mean}(A, 2)$ | |
| 17. $\min(A)$ | |

□

3.2.3 Operations on matrices

Table 3.2 on page 20 lists some frequently used matrix operations and functions. The important ones are the element-wise operations, matrix-vector products and matrix-matrix addition and multiplication. In the class of the element-wise operations — also called *dot* operations because they are preceded by a dot — there are the element-wise product, element-wise division and element-wise exponentiation (power). Those operations work the same as they do for vectors: they address matrices element-by-element, therefore they can be performed on matrices of the same size. Some examples of basic operations are given below:

```

>> B = [1 -1 3; 4 0 7]
B =
     1     -1     3
     4      0     7
>> B2 = [1 2; 5 1; 5 6];
>> B = B + B2'           % add two matrices; why B2' is needed instead of B2?
B =
     2     4     8
     6     1    13
>> B-2                   % subtract 2 from all elements of B
ans =
     0     2     6
     4    -1    11
>> ans = B./4           % divide all elements of the matrix B by 4
ans =
    0.5000    1.0000    2.0000
    1.5000    0.2500    3.2500
>> 4/B                   % this is not possible
??? Error using ==> /
Matrix dimensions must agree.

>> 4./B                  % this is possible; equivalent to: 4.*ones(size(B))./B
ans =
    2.0000    1.0000    0.5000
    0.6667    4.0000    0.3077
>> C = [1 -1 4; 7 0 -1];
>> B .* C                % multiply element-by-element
ans =
     2    -4    32
    42     0   -13
>> ans.^3 - 2           % do for all elements: raise to the power 3, subtract 2
ans =
     6    -66   32766
   74086     -2   -2199
>> ans ./ B.^2         % element-by-element division
ans =
   1.0e+03 *
    0.0015   -0.0041    0.5120
    2.0579   -0.0020   -0.0130
>> r = [1 3 -2];
>> r * B2               % allowed because r has as many rows as B2 has columns
ans =
     6    -7

```

A new type of element-wise matrix / vector addition. In mathematics addition of two matrices $A+B$ is defined only if A and B have the same number of rows and the same number of columns. However, since MATLAB version R2017b addition is also defined for certain differently dimensioned matrices. Here is an overview of all possibilities, old and new:

- If A and B have the same dimensions then $A+B$ is defined as in mathematics:

$$\begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array}$$

This rule works in older version of MATLAB as well.

- If one of the two is a number and the other is a matrix then $A+B$ adds the number to every entry of the matrix:

$$\begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix} + \square = \square + \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix}$$

This rule works in older version of MATLAB as well.

- If one is a column vector and the other is a matrix with equally many rows as the column vector, then $A+B$ adds the column to every column of the matrix:

$$\begin{bmatrix} \square \\ \square \end{bmatrix} + \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix} + \begin{bmatrix} \square \\ \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix}$$

- Similarly, if one is a row vector and the other is a matrix with equally many columns as the row vector, then $A+B$ adds the row to every row of the matrix:

$$\begin{bmatrix} \square & \square & \square \end{bmatrix} + \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix} + \begin{bmatrix} \square & \square & \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix}$$

- If one is a column vector and the other is a row vector then $A+B$ returns the matrix of all possible sums $a_i + b_j$:

$$\begin{bmatrix} \square & \square & \square \end{bmatrix} + \begin{bmatrix} \square \\ \square \end{bmatrix} = \begin{bmatrix} \square \\ \square \end{bmatrix} + \begin{bmatrix} \square & \square & \square \end{bmatrix} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \end{bmatrix}$$

The above explains the new rules for addition. Similar rules apply to element-wise division, multiplication and exponentiation. For example

```
>> [1 2 3; 4 5 6].*[-10; 100]      % in newer versions of Matlab this should work
ans =
   -10   -20   -30
   400   500   600
>> [1 2 3; 4 5 6].*[1 -10 100]    % ... and this as well
ans =
     1   -20   300
     4   -50   600
```

Concerning the standard matrix-vector and matrix-matrix products, two things should be reminded from linear algebra. First, a matrix A with n rows and k columns can be multiplied from the right by a column vector x with k entries. The product Ax is then a column vector with n entries. More general, given two matrices A and B , the product $C = AB$ is well defined if and only if A has as many columns as B has rows. To envision this product $C = AB$ it may be useful to place A to the left of C and B above C , see Fig. 3.1. The product C has as many rows as A and as many columns as B . The entry c_{ij} in the i th row and j th column of C is determined by row i of A and column j of B : assuming A has k columns (and hence B has k rows) this is $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{ik}b_{kj}$.

```
>> b = [1 3 -2];
>> B = [1 -1 3; 4 0 7]
B =
     1     -1     3
     4     0     7
>> b * B
??? Error using ==> *           % not possible: b is 1-by-3 and B is 2-by-3
```

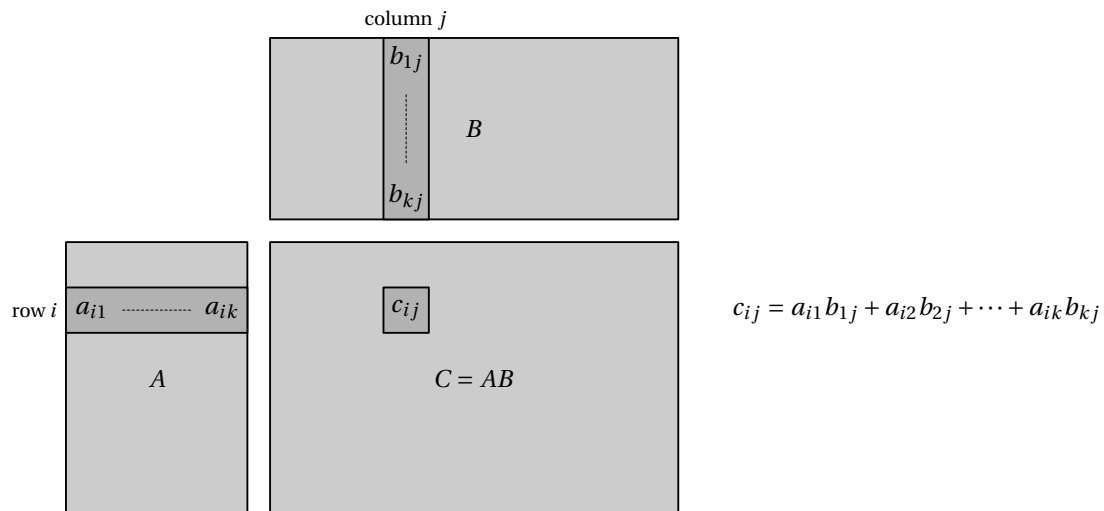


FIGURE 3.1: Matrix product $C = AB$: place A to left of C and B above C . Then entry c_{ij} follows from the corresponding row of A and corresponding column of B . The product AB is defined only if A has as many columns as B has rows

```

Inner matrix dimensions must agree.

>> b * B'                % this is allowed: length(b) == numbers of rows of B'
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 an 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 3.6. Perform all operations from Table 3.2, using some matrices A and B , vector v and scalars k , a , b , n , and m . □

Exercise 3.7. Find two 2×2 matrices A and B for which $A .* B$ does not equal $A * B$. □

Exercise 3.8. Let A be a square matrix.

1. Create a diagonal matrix, which has the same diagonal as A . *Hint:* you may use the command `diag`.
2. Create a matrix B , whose elements are the same as those of A except the entries on the main diagonal. The diagonal of B should consist of 1s.
3. Create a tridiagonal matrix T , whose three diagonal are taken from the matrix A . *Hint:* you may use the commands `triu` and `tril`. □

Exercise 3.9. Given the vectors $x = [1 \ 3 \ 7]$, $y = [2 \ 4 \ 2]$ and matrices $A = [3 \ 1 \ 6; 5 \ 2 \ 7]$ and $B = [1 \ 4; 7 \ 8; 2 \ 2]$, determine which of the following statements can be correctly executed (and if not, try to understand why) and provide the result:

- | | | |
|--------------------|---------------|--------------------------|
| 1. $x + y$ | 8. $A - 3$ | 15. $2 .* B$ |
| 2. $x + A$ | 9. $A + B$ | 16. $B ./ x'$ |
| 3. $x' + y$ | 10. $B' + A$ | 17. $B ./ [x' \ x']$ |
| 4. $A - [x' \ y']$ | 11. $B * A$ | 18. $2 / A$ |
| 5. $A + [0; 10]$ | 12. $A .* B$ | 19. $\text{ones}(1,3)*A$ |
| 6. $[x; y] + A$ | 13. $A' .* B$ | 20. $\text{ones}(1,3)*B$ |
| 7. $[x; y']$ | 14. $2 * B$ | |

□

Exercise 3.10. Consider the following problem. John and Pete are together 90 years old. John is 10 years older than Pete. These are two linear equations:

$$\text{John} + \text{Pete} = 90, \quad \text{John} - \text{Pete} = 10.$$

Write this in the form

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

with x_1 the age of John and x_2 the age of Pete and $a_{11}, a_{12}, a_{21}, a_{22}, b_1, b_2$ certain numbers that you have to find. Once you found matrix A and vector b compute x in MATLAB and check the outcome. (Hint: look at Table 3.2.) □

Exercise 3.11. Let A be a random 5×5 matrix and let b be a random 5×1 vector. Given that $Ax = b$, try to find x (look at Table 3.2). Explain the difference between the operators `\` and `/` and the command `inv`. Having found x , check whether $Ax - b$ is close to a zero vector. □

Exercise 3.12. Consider a rectangular plate of size 1 by 2 which is held at the temperature of zero degrees at three of its borders and at 20 degrees at the fourth border, see Fig. 3.2. We want to know the temperature distribution inside the plate.

We assume the following model. We take n points along the horizontal axis and m points along the vertical axis. All these points are equally distributed and the corner points are included. (In Fig. 3.2 we took $n = 4$ and $m = 8$.) The temperature at point (n, m) we assume to be the average of the temperature at its four neighboring points $(n - 1, m)$, $(n + 1, m)$, $(n, m + 1)$, and $(n, m - 1)$. Write this model in the form $Ax = b$ with x the vector of unknown temperatures and take $n = 4$ and $m = 8$. Obtain the temperature profile (that is, determine x) and check if the profile makes sense given the 20 and 0 degrees at the borders. □

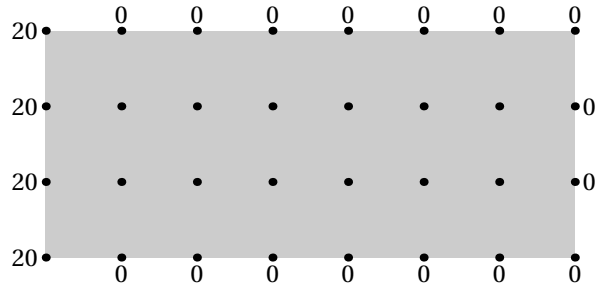


FIGURE 3.2: Heat distribution inside a plate. The temperature at the left is 20. At the other borders it is zero. See Exercise 3.12

Exercise 3.13. Let $A = \text{ones}(6) + \text{eye}(6)$. Normalize the columns of the matrix A so that all columns of the resulting matrix, say B , have the Euclidean norm¹ (length) equal to 1. \square

Exercise 3.14. Let $H = \text{hilb}(4)$. This computes the 4×4 Hilbert matrix. Determine in MATLAB the inverse H^{-1} of this matrix and verify in MATLAB that $H^{-1}H$ is the identity matrix. \square

3.3 Multi-dimensional arrays

In MATLAB one can do $A(1, 2, 2) = 1$ as well! This defines a 3-dimensional array. Yes, 1-dimensional arrays we call vectors, 2-dimensional arrays we call matrices, but MATLAB can handle higher dimensional arrays. We do not explore this any further.

¹The Euclidean norm of a vector (x_1, \dots, x_p) is defined as $\sqrt{x_1^2 + \dots + x_p^2}$. It can be computed with `norm(x)`.

Chapter 4

Visualization

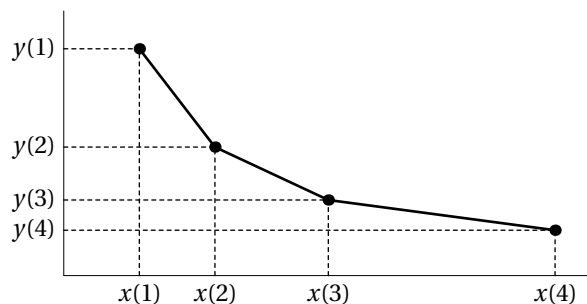
MATLAB can visualize data and export it in various formats. We introduce 2D and 3D plots.

TABLE 4.1: Plot colors and styles

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

4.1 Simple 2D plots

With the command `plot`, a graphical display in the 2D plane can be made. For a real vector y of n entries, the command `plot(y)` draws the points $[1, y(1)]$, $[2, y(2)]$, \dots , $[n, y(n)]$ and connects the consecutive points with straight lines. The command `plot(x, y)` does the same for the points $[x(1), y(1)]$, $[x(2), y(2)]$, \dots , $[x(n), y(n)]$. Here x and y have to be vectors of the same length. For example,



The commands `loglog`, `semilogx` and `semilogy` are similar to `plot`, except that one or both axes are then in logarithmic scale.

Exercise 4.1 (Power laws). It is a curious fact that many relations approximately satisfy a power law, $y = cx^\alpha$. For instance the number of gas stations y in a city of population size x satisfies a power law with $\alpha \approx 0.77$. Also, the calories y needed per day of a mammal of weight x behaves like a power law and now $\alpha \approx 0.74$ — irrespective of the species of mammal. Let us set up some artificial data:

```
alpha=3/4;
x=100*rand(1,500); % 500 random numbers between 0 and 100
y=1234*x.^alpha;
```

Given x and y , which plot command would be best to figure out if x, y satisfy a power law? Is it `plot`, or `semilogx`, or `semilogy`, or `loglog`? □

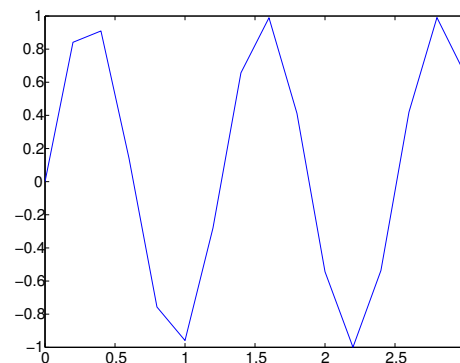
Exercise 4.2. Type the following commands after predicting the result:

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

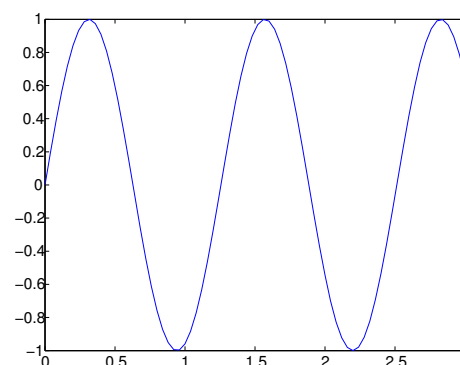
As you can see, the same figure is used for both plot commands. The previous function is removed as soon as the next is displayed. The command `figure` creates a new figure window. Repeat the previous commands, but create a new figure before plotting the second function, so that you can see both functions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

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

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



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



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

```
x = 0:.04:3;
y = sin(5*x);
plot(x,y,'r--')
```

This produces the dashed red line. Here the third argument of `plot` specifies the color (optional) and the line style. Additionally, the line width can be specified. You can also mark the points by a selected marker and choose its size:

TABLE 4.2: Useful commands to make plots

Command	Result
grid on/off	adds a grid to the plot at the tick marks or removes it
box off/on	removes the axes box or shows it
axis([xmin xmax ymin ymax])	sets the minimum and maximum values of the axes
axis square	makes the axis box square
axis equal	makes the increments on the axis equal
axis image	like axis equal, but with tighter bounding box
xlabel('text')	plots the label text on the x-axis
ylabel('text')	plots the label text on the y-axis
zlabel('text')	plots the label text on the z-axis
title('text')	plots a title above the graph
text(x,y,'text')	adds text at the point (x,y)
gtext('text')	adds text at a manually (with a mouse) indicated point
legend('fun1','fun2')	plots a legend box to name your functions (move the box with your mouse)
legend off	deletes the legend box
clf	clear the current figure
figure(n)	make figure n the current figure
subplot	creates a subplot in the current figure

```
x = 0:.04:3; y = sin(5*x);
plot(x,y,'r*--','linewidth',1,'markersize',6)
```

To add a title, grid and to label the axes, do

```
xlabel('x-axis'); % default fontsize is rather small
ylabel('y-axis','FontSize',14); % larger fontsize
title('Function y = sin(5*x)','FontSize',14);
set(gca,'FontSize',12); % set fontsize of axes text in current plot to 12
grid % you can remove grid again by calling "grid off"
```

Table 4.1 documents several possibilities of the plot command. Do help plot to see them all.

Exercise 4.3. 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 4.4. 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 axes description. □

Exercise 4.5. Plot a circle with the radius $r = 2$, knowing that the parametric equation of a circle is $x(t) = r \cos(t)$, $y(t) = r \sin(t)$ for $t \in [0, 2\pi]$. □

4.2 Several functions in one figure

There are different ways to draw several functions in the same figure. The first one is with the command hold on. Then all subsequent plots are added to the current figure. To stop it do hold off. When a number of functions is plotted in a single figure, it is useful to use different symbols and colors. An example is:

```
x1 = 1:.1:3.1;
y1 = sin(x1);
plot(x1,y1,'md');
x2 = 1:.3:3.1;
y2 = sin(-x2+pi/3);
hold on
plot(x2,y2,'k*-.','linewidth',1)
plot(x1,y1,'m-','linewidth',2)
hold off
```

A second method to display a few functions in one figure is to plot several functions at the same time. The next commands produces 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 adjust the axes to better fit the plots one might add

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

or

```
axis tight
```

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

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

It is also possible to produce subplots in one figure window. With the command subplot(p, r, n), the figure window is divided horizontally and vertically into $p \times r$ subfigures, and subfigure n is selected as current subfigure. The commands plot, title, grid, etc. affect only in the current subfigure. So, if you want to change something in another subfigure, you must first switch to that subfigure:

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

Exercise 4.6. Plot the functions $f(x) = x$, $g(x) = x^3$, $h(x) = e^x$ and $z(x) = e^{x^2}$ over the interval $[0, 4]$ on the normal scale and on the log-log scale. Use an appropriate sampling to get smooth curves. Describe your plots by using the functions: xlabel, ylabel, title and legend. □

Exercise 4.7. Make a plot of the functions: $f(x) = \sin(1/x)$ and $f(x) = \cos(1/x)$ over the interval $[0.01, 0.1]$. How do you create x so that the plots look sufficiently smooth? □

4.3 Plotting in the complex plane

As explained in the beginning of this chapter, for a real vector y of length n the command `plot(y)` draws the points

$$[1, y(1)], [2, y(2)], \dots, [n, y(n)]$$

and connects the consecutive points with straight lines. If however the vector y is complex, then the command `plot(y)` draws the points

$$[\operatorname{Re}(y(1)), \operatorname{Im}(y(1))], [\operatorname{Re}(y(2)), \operatorname{Im}(y(2))], \dots, [\operatorname{Re}(y(n)), \operatorname{Im}(y(n))],$$

and (again) connects them with straight lines. Thus for a complex vector y the command `plot(y)` is the same as `plot(real(y), imag(y))`.

Exercise 4.8. Make a plot of the functions $f(t) = e^{it}$ and $f(t) = e^{it} + \frac{1}{2}e^{10it}$. Choose a sensible range for t . □

4.4 Other 2D plotting features

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

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

Exercise 4.9. To get an impression of other visualizations, type the following commands and describe the result (note that the command `figure` creates a new figure window):

```
figure
x = -2.9:0.2:2.9;
bar(x,exp(-x.*x));        % bar plot of a bell shaped curve
figure
stem(x,exp(-x.*x));       % stem plot (aka "lollipop" plot)
figure
x = 0:0.25:10;
stairs(x,sin(x));         % staircase plot of a sine wave
figure
x = -2:0.1:2;
y = erf(x);               % error function; check help if you are interested
e = rand(size(x)) / 10;
errorbar(x,y,e);          % errorbar plot
figure
r = rand(5,3);
subplot(1,2,1);
bar(r,'grouped')          % bar plot
subplot(1,2,2);
bar(r,'stacked')
```

```
figure
x = randn(200,1);           % normally distributed random numbers
hist(x,15)                 % histogram plot with 15 bins
```

□

4.5 Printing & saving figures

Before printing a figure, you might want to add some information, such as a title, or change the lay-out somewhat. Table 4.2 shows some of the commands that can be used.

Exercise 4.10. Plot the functions $y_1 = \sin(4x)$, $y_2 = x \cos(x)$, $y_3 = (x + 1)^{-1} \sqrt{x}$ for $x = 1:0.25:10$; and a single point $(x, y) = (4, 5)$ in one figure. Use different colors and styles. Add a legend, labels for both axes and a title. Add also a text to the single point saying: 'single point'. Change the minimum and maximum values of the axes such that one can look at the function y_3 in more detail. □

When you like the displayed figure, you can print it to paper. The easiest way is to click on File in the menu-bar of the figure window and to choose Print. If you click OK in the print window, your figure will be sent to the printer indicated there.

There exists also a print command, which can be used to send a figure to a printer or output it to a file. You can optionally specify a print device (i.e. an output format such as *jpeg* or *png*) and options that control various characteristics of the printed file (i.e., which figure to print etc). You can also print to a file if you specify the file name. If you do not provide an extension, print adds one. Since they are many parameters they will not be explained here (check help print to learn more). Instead, try to understand the examples:

```
print -dwincc             % print current Figure to current printer in color
print -f1 -deps myfile.eps % print Figure 1 to myfile.eps in black/white
print -f1 -depsec myfilec.eps % print Figure 1 to myfilec.eps in color
print -dtiff myfile1.tiff % print current Figure to myfile1.tiff
print -dpng myfile1.png  % print current Figure to myfile1.png
print -f2 -djpeg myfile2  % print Figure 2 to myfile2.jpg
```

Choose EPS or PDF if you want to include it in \LaTeX -documents. EPS and PDF are usually not bitmaps but vector graphics, meaning that they are error-free and small in size.

Exercise 4.11. Practise with printing, especially to a file. Try to print figures from the previous exercises. □

4.6 3D line plots

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

```
t = linspace(-10*pi,10*pi,200);
plot3(t.*sin(t), t.*cos(t), t, 'md-'); % plot the curve in magenta, indicate the
                                        % points with diamonds, and connect them
                                        % with a straight line

title('Curve r(t) = [t sin(t), t cos(t), t]');
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
grid
```


Exercise 4.12. Make a 3D smooth plot of the curve defined parametrically as:

$$(x(t), y(t), z(t)) = (\sin(t), \cos(t), \sin^2(t))$$

for $t = [0, 2\pi]$. Plot the curve in green, with the points marked by circles. Add a title, description of axes and the grid. You can rotate the image by clicking Tools at the Figure window and choosing the Rotate 3D option or by typing `rotate3D` at the prompt. Then by clicking at the image and dragging your mouse you can rotate the axes. Exercise with this option. \square

4.7 Plotting surfaces in 3D

MATLAB provides a number of commands to plot 2D surfaces in 3D. We begin with surfaces defined by a function $z = f(x, y)$ with (x, y) living in some rectangular domain. Plotting such $z = f(x, y)$ requires in MATLAB a gridded rectangular domain (x, y) . The command `meshgrid` helps in constructing this grid.

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

The domain $[-1, 1] \times [0, 2]$ is now gridded with a grid-spacing 0.5 in both directions, and the grid-points are $[X(i, j), Y(i, j)]$. To obtain a smooth surface $z = f(x, y)$, the grid-spacing should be small enough. The surface can be plotted with the commands `mesh` or `surf` and variations:

```
[X,Y] = meshgrid(-1:.05:1, 0:.05:2);
Z = sin(5*X) .* cos(2*Y);
mesh(X,Y,Z); % "fishnet" plot with color
surf(X,Y,Z); % colored surface plot
surfl(X,Y,Z); % colored surface plot with lamp
shading interp % interpolate (smooth)
colormap(copper) % change colormap to "copper"
title ('Function z=sin(5x)*cos(2y)')
xlabel('x')
ylabel('y')
zlabel('z')
```

See Fig. 4.1. You can also try the command `waterfall` instead of `mesh`.

Exercise 4.13. Produce a nice graph that demonstrates as clearly as possible the behavior of the function $f(x, y) = \frac{xy^2}{x^2+y^4}$ near the point $(0, 0)$. Note that the grid-spacing around this point should be small. \square

Surfaces need not necessarily be defined on a rectangular grid (x, y) . It is possible to draw surfaces where all three components (x, y, z) are parameterized. For instance, to plot the sur-

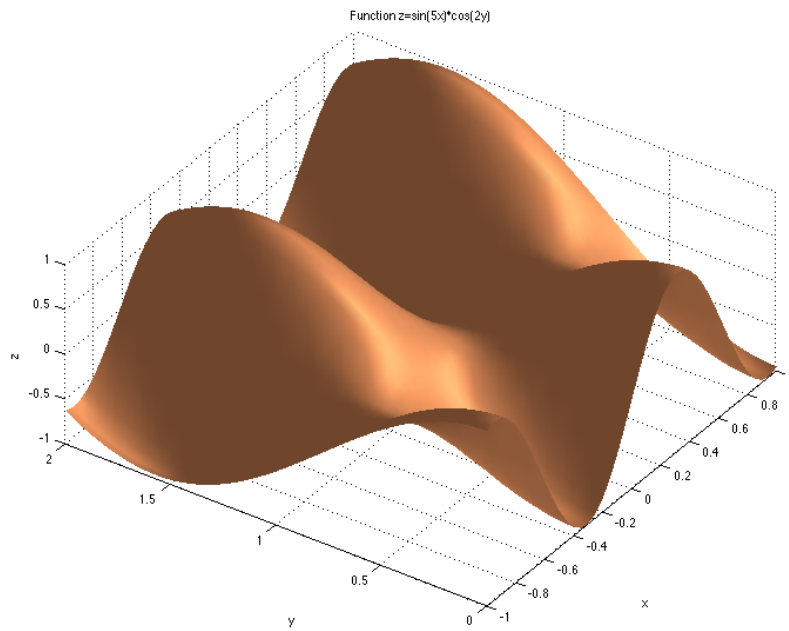
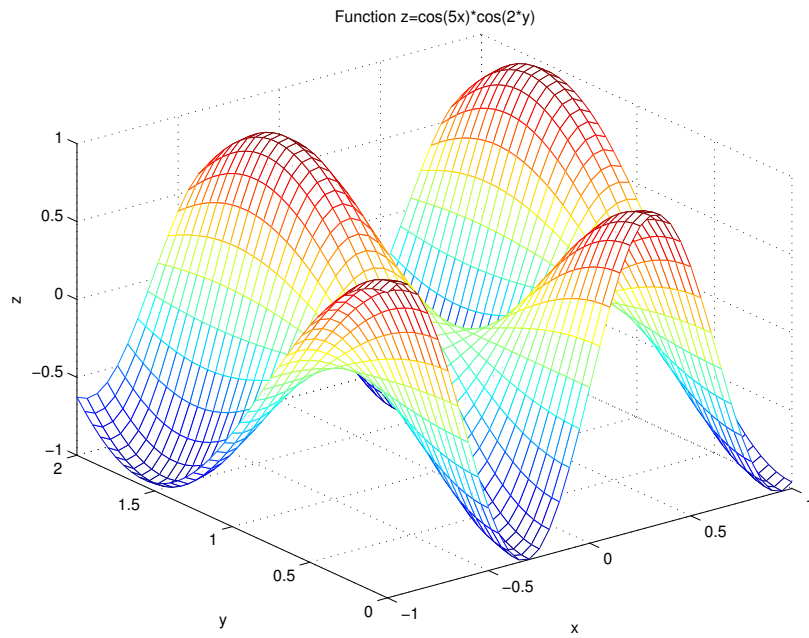


FIGURE 4.1: Example of mesh and surf plots

face of the unit sphere you might want to use the parameterization

$$(x(t, s), y(t, s), z(t, s)) = (\cos(t) \cos(s), \cos(t) \sin(s), \sin(t)) \quad t \in [-\pi/2, \pi/2], s \in [0, 2\pi].$$

Since (t, s) lives in rectangular domain we can now plot the unit sphere as follows.

```
[T,S] = meshgrid(-pi/2:.05:pi/2, 0:.05:(2*pi));
X=cos(T).*cos(S);
Y=cos(T).*sin(S);
Z=sin(T);
surf(X,Y,Z);
```

In fact the sphere is predefined in MATLAB, do type `sphere.m` to find out how it is defined.

Exercise 4.14. Plot the parametric function of r and θ :

$$(x(r, \theta), y(r, \theta), z(r, \theta)) = (r \cos(\theta), r \sin(\theta), \sin(6 \cos(r) - n\theta)).$$

Choose n to be constant. Observe, how the graph changes depending on different $n \in \mathbb{N}$. \square

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

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

Use `close all` to close all figures and start a new task (or use `close 1` to close Figure no. 1 etc). Use `colormap` to define different colors for plotting.

To locate e.g. the minimum value of the `peaks` function on the grid, you can proceed as follows:

```
[mm,I] = min(Z);          % a row vector of the min elements of each column
                           % I is a vector of corresponding indices
[Zmin, j] = min (mm);     % Zmin is the minimum, j is the index where attained
                           % So minimum of matrix Z is attained at index (I(j),j)
xpos = X(I(j),j);         %
ypos = Y(I(j),j);         % position of the minimum value
contour (X,Y,Z,25);
xlabel('x-axis');
ylabel('y-axis');
hold on
plot(xpos(1),ypos,'*');
text(xpos(1)+0.1,ypos,'Minimum');
hold off
```

It is also possible to combine two or more plots into one figure. For instance:

```
surf(peaks(25)+6);        % move the z-values with the vector [0,0,6]
hold on
pcolor(peaks(25));
```

Exercise 4.15. Plot the surface $f(x, y) = xye^{-x^2-y^2}$ over the domain $[-2, 2] \times [-2, 2]$. Find the values and the locations of the minima and maxima of this function and explain how the code works. \square

4.8 Animations

A sequence of graphs can be put in motion in MATLAB, i.e. you can make a movie using MATLAB graphics tools. To learn how to create a movie, analyze first the script below which shows the plots of $f(x) = \sin(nx)$ over the interval $[0, 2\pi]$ and $n = 1, \dots, 5$:

```
N = 5;
M = moviein(N);
x = linspace(0, 2*pi);
for n=1:N
    plot(x, cos(n*x), 'r-');
    xlabel('x-axis')
    if n > 1,
        ss = strcat('cos(', num2str(n), 'x)');
    else
        ss = 'cos(x)';
    end
    ylabel(ss)
    title('Cosine functions cos(nx)', 'FontSize', 12)
    axis tight
    grid
    M(:, n) = getframe;
    pause(1.8)
end
movie(M) % this plays a quick movie
```

Here, a for-loop construction has been used to create the movie frames. You will learn more on loops in Section 5.4. Also the command `strcat` has been used to concatenate strings. Use `help` to understand or learn more from Section 8.1.

Play this movie to get acquainted. Five frames are first displayed and at the end, the same frames are played again faster. Command `moviein`, with an integral parameter, tells MATLAB that a movie consisting of N frames is going to be created. Consecutive frames are generated inside the loop. Via the command `getframe` each frame is stored in the column of the matrix M . The command `movie(M)` plays the movie just created and saved in columns of the matrix M . Note that to create a movie requires quite some memory. It might be useful to clear M from the workspace later on.

Exercise 4.16. Write a script that makes a movie consisting of 5 frames of the surface $f(x, y) = \sin(nx) \sin(ny)$ over the domain $[0, 2\pi] \times [0, 2\pi]$ and $n = 1 : 5$. Add a title, description of axes and shading. \square

Chapter 5

Logicals and loops

With loops we can efficiently perform *repetitive* tasks, and with logicals (or *booleans*) we can choose *which* tasks to perform, and which to skip.

5.1 Logical and relational operators

The two logical values are TRUE and FALSE. In MATLAB the result of a logical operation is 1 if it is true and 0 if it is false. Table 5.1 shows the relational and logical operations. Read it carefully. Another way to get to know more about them is to type `help relop`. The relational operators `<`, `<=`, `>`, `>=`, `==` and `~=` can be used to compare two arrays of the same size or an array to a scalar. The logical operators `&`, `|` and `~` allow for the logical combination or negation of relational operators. In addition, three functions are also available: `xor`, `any` and `all` (use `help` to find out more).

TABLE 5.1: Relational and logical operations

Command	Result
<code>a = (b > c)</code>	a is 1 if b is larger than c. Similar are: <code><</code> , <code>>=</code> and <code><=</code>
<code>a = (b == c)</code>	a is 1 if b is equal to c
<code>a = (b ~= c)</code>	a is 1 if b is not equal c
<code>a = ~b</code>	logical complement: a is TRUE if b is FALSE, and the other way around
<code>a = (b & c)</code>	logical AND: a is 1 if b = TRUE AND c = TRUE
<code>a = (b c)</code>	logical OR: a is 1 if b = TRUE OR c = TRUE
<code>a = (b && c)</code>	Short-circuit AND: like <code>a=b&c</code> but does not evaluate c if the result is already determined by b (if b is FALSE)
<code>a = (b c)</code>	Short-circuit OR: like <code>a=b c</code> but does not evaluate c if the result is already determined by b (if b is TRUE)

Important: MATLAB treats every non-zero number as true. Thus

```
>> b = 10;  
>> 1 & b  
ans =  
    1
```

This show that you should be careful in the distinction between numbers and logical variables.

Exercise 5.1. Predict and explain the outcome of following commands.

```
b = 10;
whos
b = 1 | b
whos
```

□

The logical & precedes | in MATLAB. Both $0 \& 1 | 1$ and $1 | 1 \& 0$ hence are true. A common situation is:

```
>> b = 10;
>> 1 | b > 0 & 0
ans =
     1
>> 1 | (b > 0 & 0)           % this indicates the same as above
ans =
     1
>> (1 | b > 0) & 0
ans =
     0
>> b=0;
>> a=(b~=0&&1/b)           % the double && suppresses evaluation of 1/0
a =
     0
```

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

The introduction of the logical data type has forced some changes in the use of non-logical 0-1 vectors as indices for subscripting. You can see the differences by executing the following commands that attempt to extract the elements of y that correspond to either the odd or even elements of x , assuming that x and y are two vectors of the same length:

```
x=1:6;
y=11:16;
y(rem(x,2))           % is this what we want?
y(logical(rem(x,2))) % or this?

y(~rem(x,2))         % this?
y(~logical(rem(x,2))) % or this, or both? (for the even elements)
```

Use whos to understand the differences.

Exercise 5.2. Exercise with logical and relational operators:

1. Predict and check the result of each of the operations of Table 5.1 for $b = 0$ and $c = -1$.
2. Predict and check the result of each logical operator for $b = [2 \ 31 \ -40 \ 0]$ and $c = 0$.
3. Define two random vectors `logical(round(rand(1,7)))` and perform all logical operations, including xor, any and all.

□

Exercise 5.3. Exercise with logical and relational operators:

1. Let $x = [1 \ 5 \ 2 \ 8 \ 9 \ 0 \ 1]$ and $y = [5 \ 2 \ 2 \ 6 \ 0 \ 0 \ 2]$. Execute and explain the results of the following commands:

- $x > y$
- $x \leq y$
- $x \& (\sim y)$
- $y < x$
- $y \geq x$
- $(x > y) \mid (y < x)$
- $x == y$
- $x \mid y$
- $(x > y) \& (y < x)$

2. Let $x = 1:10$ 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:

- $(x > 3) \& (x < 8)$
- $y(x \leq 4)$
- $y((x < 2) \mid (x \geq 8))$
- $x(x > 5)$
- $x((x < 2) \mid (x \geq 8))$
- $x(y < 0)$

□

Exercise 5.4. Let $x = [3 \ 16 \ 9 \ 12 \ -1 \ 0 \ -12 \ 9 \ 6 \ 1]$. Provide the command(s) that will:

1. set the positive values of x to zero;
2. set values that are multiples of 3 to 3 (make use of `rem`);
3. multiply the even values of x by 5;
4. extract the values of x that are greater than 10 into a vector called y ;
5. set the values in x that are less than the mean to 0;
6. set the values in x that are above the mean to their difference from the mean.

□

Exercise 5.5. Execute the following commands and try to understand how z is defined.

```
x = -3:0.05:3;
y = sin(3*x);
subplot(1,2,1);
plot(x,y);
axis tight
z = (y < 0.5) .* y;
subplot(1,2,2);
hold on
plot(x,y,'r:');
plot(x,z,'r');
axis tight
hold off
```

□

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

```
a = randperm(10);    % random permutation
b = 1:10;
b - (a <= 7)         % subtract from b a 0-1 vector (1 if a<=7 otherwise 0)
(a >= 2) & (a < 4)  % return ones at positions where 2 <= a < 4
~(b > 4)            % return ones at positions where b <= 4
(a == b) | b == 3   % return ones at positions where a equals b or b equals 3
any(a > 5)          % return 1 if ANY of the a elements are larger than 5
any(b < 5 & a > 8)  % return 1 if there in the evaluated expression
                    % (b < 5 & a > 8) appears at least one 1
all(b > 2)          % returns 1 when ALL b elements are larger than 2
```

5.2 The command find

You can extract all elements from the vector or the matrix satisfying a given condition, e.g. equal to 1 or larger than 5, by using logical addressing. The same result can be obtained via the command `find`, which returns the positions (indices) of such elements. For instance:

```
>> x = [1 1 3 4 1];
>> i = (x == 1)
i =
     1     1     0     0     1
>> y = x(i)
y =
     1     1     1
>> j = find(x == 1)    % j holds indices of those elements satisfying x == 1
j =
     1     2     5
>> z = x(j)
z =
     1     1     1
```

Another example is:

```
>> x = -1:0.05:1;
>> y = sin(x) .* sin(3*pi*x);
>> plot(x,y, '-'); hold on
k = find(y <= -0.1)
k =
     9    10    11    12    13    29    30    31    32    33
>> plot(x(k), y(k), 'ro');
>> r = find(x > 0.5 & y > 0)
r =
    35    36    37    38    39    40    41
>> plot(x(r), y(r), 'r*');
```

The command `find` operates in a similar way on matrices:

```
>> A = [1 3 -3 -5; -1 2 -1 0; 3 -7 2 7];
>> k = find(A >= 2.5)
k =
     3
     4
    12
>> A(k)
ans =
     3
     3
     7
```

In this way, `find` reshapes first the matrix `A` into a column vector, i.e. it operates on `A(:)`, i.e. all columns are concatenated one after another. Therefore, `k` is a list of indices of elements larger than or equal to 2.5 and `A(k)` gives the values of the selected elements. Also the row and column indices can be returned, as shown below:

```
>> [I,J] = find(A >= 2.5)
I =
     3
     1
```



```

3
J =
1
2
4
>> [A(I(1),J(1)), A(I(2),J(2)), A(I(3),J(3))] % lists the values
ans =
3 3 7

```

Exercise 5.6. Let $A = \text{ceil}(5 \cdot \text{randn}(6,6))$. Perform the following:

1. find the indices and list all elements of A that are smaller than -3 ;
2. find the indices and list all elements of A that are smaller than 5 and larger than -1 ;

Exercise with both: logical indexing and the command `find`. □

5.3 Conditional code execution

With `if`-blocks we can choose which command to execute next, depending whether a logical (boolean) expression is `TRUE` or not. The general description is given below. In the following examples the command `disp` is frequently used. This command displays on the screen the text between the quotes.

The simplest `if`-block has the general syntax

```

if logical_expression
    statement1
    statement2
    ....
end
other commands

```

If the logical expression is `TRUE` then the statements are executed and after that the other commands. If the expression is `FALSE` then the statements are skipped and `MATLAB` jumps to the other commands. For example:

```

if (a > 0)
    b = a; % executed only if a>0
    disp('a is positive'); % displayed only if a>0
end
disp('end of code'); % always displayed

```

Often we need to execute one set of statements if the logical expression is `TRUE`, and another set if it is `FALSE`. This is accommodated for by the `else`-block. The general syntax is

```

if logical_expression
    block of statements
    (evaluated if TRUE)
else
    block of statements
    (evaluated if FALSE)
end

```

For example:

```

if (temperature > 100)
    disp ('Above boiling. ');
    toohigh = 1;
else
    disp ('Temperature is OK. ');
    toohigh = 0;
end

```

With the elseif-block we can combine more conditions. The general syntax is:

```

if logical_expression1
    statements % if logical_expression1 is TRUE
elseif logical_expression2
    statements % if logical_expression1 is FALSE
    statements % and logical_expression2 is TRUE
elseif logical_expression3
    statements % if logical_expression1 and
    statements % logical_expression2 are FALSE,
    statements % and logical_expression3 is TRUE
else
    block of statements evaluated % in all other cases
end

```

An example is:

```

if (height > 190)
    disp ('very tall'); % above 190
elseif (height > 170)
    disp ('tall'); % in between 170 and 190
elseif (height < 150)
    disp ('small'); % less than 150
else
    disp ('average'); % in between 150 and 170
end

```

Exercise 5.7. In each of the following questions, evaluate the given code fragments. Investigate each of the fragments for the various starting values given on the right. Use MATLAB to check your answers (be careful, since the fragments are not always the proper MATLAB expressions):

1.

```

if n > 1
    m = n + 2
else
    m = n - 2
end

```

a) n = 7 m = ?
b) n = 0 m = ?
c) n = -7 m = ?

2.

```

if s <= 1
    t = 2z
elseif s < 10
    t = 9 - z
elseif s < 100
    t = sqrt(s)

```

a) s = 1 t = ?
b) s = 7 t = ?
c) s = 57 t = ?
d) s = 300 t = ?

```

else
  t = s
end

```

3.

```

if t >= 24      a) t = 50   h = ?
  z = 3t + 1    b) t = 19   h = ?
elseif t < 9    c) t = -6   h = ?
  z = t^2/3 - 2t d) t = 0    h = ?
else
  z = -t
end

```

4.

```

if 0 < x < 7    a) x = -1   y = ?
  y = 4x        b) x = 5    y = ?
elseif 7 < x < 55 c) x = 30   y = ?
  y = -10x      d) x = 56   y = ?
else
  y = 333
end

```

□

Exercise 5.8. Write a script that checks whether an integer 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. [Hint: use the command rem]. □

Another selection structure is switch, which switches between several cases depending on an expression, which is either a scalar or a string. An example is

```

method = 2;
switch method
  case {1,2}
    disp('Method is linear. '); % if method==1 or method==2
  case 3
    disp('Method is cubic. '); % if method==3
  case 4
    disp('Method is nearest. '); % if method==4
  otherwise
    disp('Unknown method. '); % if method not equal to 1 or 2 or 3 or 4
end

```

The statements following the *first* case where the expression matches the choice are executed. This construction can be very handy to avoid long if .. elseif ... else ... end constructions. The expression can be a scalar or a string. A scalar expression matches a choice if expression == choice. A string expression matches a choice if strcmp(expression, choice) returns 1 (is true) (strcmp compares two strings).

Important: Note that the switch-construction only allows the execution of one group of commands, namely the *first* case that matches.

Exercise 5.9. Assume that the months are represented by numbers from 1 to 12. Write a script that asks you to provide a month number and returns the number of days in that particular month. (You may assume it is not a leap year.) Alternatively, write a script that asks you to provide a month name (e.g. 'June') instead of a number. Use the switch-construction. (Hint: type `help input`.) □

5.4 Loops

Iteration control structures, *loops*, are used to repeat a block of statements until some condition is met. Two types of loops exist: the *for*-loop and the *while*-loop.

For loop

The *for*-loop repeats a group of statements a *fixed* number of times. The standard *for*-loop has general syntax

```
for index = firstvalue:step:lastvalue
    block of statements
end
```

The block of statements is first executed with *index* equal to *firstvalue*. After completing the block of statements, *index* is increased with *step* and the block of statements is executed again. Then *index* is increased once again with *step* and the block of statements is executed again, et cetera. This process stops if *index+step* exceeds *lastvalue*. For example,

```
x=1:10;                % whatever vector
sumx = 0;
for i=1:length(x)
    sumx = sumx + x(i);
end
```

This adds up all elements of the vector *x*. Incidentally, you can specify any step, including a negative value. Some possible variations:

```
for i=1:2:10
    % loops over i=1,3,5,7 and 9
end
disp(i); % i=9
```

and

```
for zz=5:-1:10
    % not executed at all because the loop array is empty!
end
zz % variable zz not defined
```

Two more examples:

```
for x=0:0.5:1
    % should loop over x=0,0.5,1
end
```

Due to the finite numerical precision it may be better to replace `x=0:0.5:1` with `x=(0:2)/2` or `x=linspace(0,1,3)`.

```
for x=[25 9 81]
    disp(sqrt(x)); % first displays 5, then 3 and then 9
end
```

MATLAB has the nice feature that it can loop over matrices as well. If A is some matrix then `for c=A` makes c loop over the columns of A .

```
>> for c=[1 2; 3 4]
c
end

c =
    1
    3

c =
    2
    4
```

A possibly confusing side effect of this choice is that looping over a *column* vector means the loop is executed just once. Probably not what you intended.

An example how to use the loop construct to draw graphs of $f(x) = \cos(nx)$ for $n = 1, \dots, 9$ in different subplots is:

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

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

```
x = [1 2 -1 5 -7 2 4]; % whatever
y = [3 1 -5 7]; % whatever
n = length(x);
m = length(y);
for i=1:n
    for j=1:m
        A(i,j) = x(i) * y(j);
    end
end
```

While loop

In a `for`-loop the number of times the loop is executed is fixed upon entering the loop. Quite often we want this number of times to depend on the outcome of the commands within the loop. In such cases we use the `while`-loop. A `while` loop evaluates a group of commands as long as a logical expression is `TRUE`.

```
N = 100;
k = 1;
```

```

while (k*(k+1)) <= N
    k = k + 1;
end;

```

It computes the smallest positive integer k for which $k(k+1)$ exceeds N . We can create the matrix A with entries $A_{ij} = x_i y_j$ also with the while-loop, although for-loops are more suitable here:

```

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

```

Exercise 5.10. Determine the sum of the first 50 squared numbers with a loop. □

Exercise 5.11. Write a script that determines the largest integer n for which $\sqrt{1^3} + \sqrt{2^3} + \dots + \sqrt{n^3}$ is less than 1000. □

Exercise 5.12. Use a loop construction to carry out the computations. Write short scripts.

1. Create a script with just one loop that calculates the sum of all entries of a vector x and also the vector of running sums. (The running sum of a vector x of n entries is the vector of n entries defined as $[x_1 \ x_1 + x_2 \ x_1 + x_2 + x_3 \ \dots \ \sum_{i=1}^n x_i]$.) You are not allowed to use the built-in functions `sum` and `cumsum`. Test your code for $x = [1 \ 9 \ 1 \ 0 \ 4]$.
2. 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:
 - $a_{ij} = y_i / x_j$;
 - $b_i = x_i y_i$ and add up the elements of b ;
 - $c_{ij} = x_i / (2 + x_i + y_j)$;
 - $d_{ij} = 1 / \max(x_i, y_j)$.
3. Write a script that transposes a matrix A . Check its correctness with the MATLAB operation: A' .
4. Create an m -by- n array of random numbers (use `rand`). Move through the array, element by element, and set any value that is less than 0.5 to 0 and any value that is greater than (or equal to) 0.5 to 1.
5. Write a script that will use the random-number generator `rand` to determine:
 - the number of random numbers it takes to add up to 10 (or more);
 - the number of random numbers it takes before a number between 0.8 and 0.85 occurs;
 - the number of random numbers it takes before the mean of those numbers is within 0.01 and 0.5.

It will be worthwhile to run your script several times because you are dealing with random numbers. Can you predict any of the results that are described above?

□

Exercise 5.13. Write a script that asks for a temperature in degrees Celsius t_c and computes the equivalent temperature in degrees Fahrenheit t_f (use the formula $t_f=32+9/5*t_c$). The script should keep on asking for a temperature until you provide an empty temperature. The functions `input` and `isempty` (use `help` to learn more) should be useful here. □

5.5 Evaluation of logical and relational expressions in the control flow structures

The relational and logical expressions may become more complicated. It is not difficult to operate on them if you understand how they are evaluated. To explain more details, 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 gives 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. The second expression is checked only if `data` is a non-empty matrix. Remember to put logical expression units between brackets to avoid wrong evaluations!

Important: The fact that computers make use of floating-point arithmetic means that often you should be careful when comparing two floating-point numbers just by:

```
if (x == y)
    ....
end
```

(Of course, such a construction is allowed e.g. when you know that x and y represent integers.) Instead of the above construction, you may try using this:

```
if (abs (x - y) < tolerance)           % e.g. tolerance = 1e-10
    ....
end
```

Exercise 5.14. Consider the following example:

```
max_iter = 50;
tolerance = 1e-4;
iter = 0;
x = 0.2;           % some initial try
xold = cos(x);
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 $\cos(x) = x$.

This code is motivated by the pattern of the red curve in Fig. 5.1. Explain in words why you expect that this code will find a solution x that satisfies $\cos(x) = x$ with small error.

Run the code for different tolerance parameters, e.g.: $1e-4$ and $1e-10$. Use `format long` to check more precisely how much the found x is really different from $\cos(x)$.

For each tolerance values check the number of performed iterations (the value of `iter`) and explain why, in general, it is a good idea to use

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

and not just `while (abs(x-xold) > tolerance)`. □

Exercise 5.15. Create the script `solve_cos2`, which is equal to the one 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`? □

Exercise 5.16. It is known that the Taylor series of $\sin(t)$ around zero is given by $t - \frac{t^3}{3!} + \frac{t^5}{5!} + \dots$. Create a script which uses a while-loop to find the largest t_f such that $|\sin(t) - t + \frac{t^3}{3!}| < \text{tolerance}$ on the interval $[0, t_f]$, where `tolerance` is the tolerance parameter. You may only use a maximum of `max_iter` iterations. □

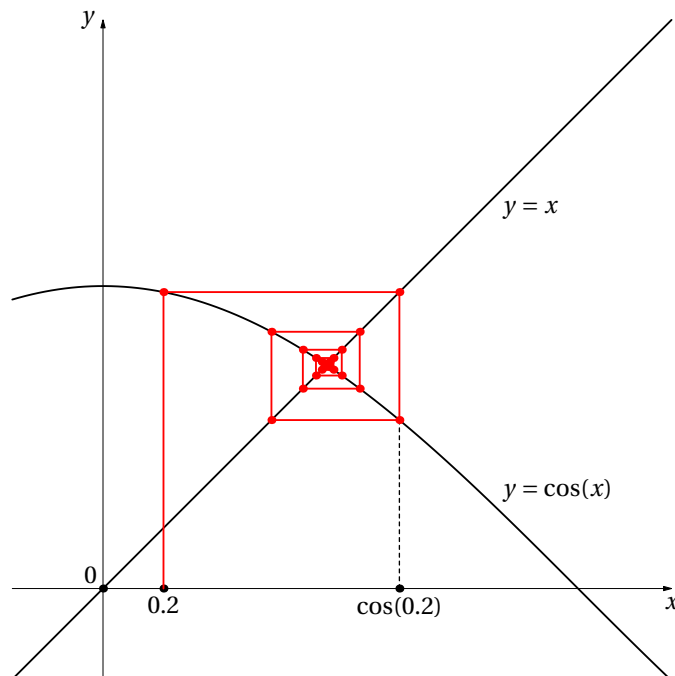


FIGURE 5.1: Finding a solution x of the equation $\cos(x) = x$, see Exercise 5.14

Chapter 6

Functions

As we have seen, in MATLAB the commands can be entered at the MATLAB prompt and can be entered into an external file called *script m-file*. If a problem is complicated then these two ways become unmanageable. In that case you will want to use subprograms. In MATLAB these are called *functions*. There are two ways of entering functions. Simple ones can be defined using *anonymous functions*. More involved subprograms are defined in what MATLAB calls *function m-files*.

6.1 Anonymous functions

An example of an anonymous function is

```
>> f = @(x) cos(x).*sin(2*x)
f =
    @(x)cos(x).*sin(2*x)
```

The @ means the *pointer* to (or *function handle* of) the otherwise unnamed function that maps x to $\cos(x)\sin(2x)$. Formally then, f is not a function but a function handle. Nevertheless, you can evaluate the function the usual way:

```
>> f(-2)
ans =
   -0.3149
```

Anonymous functions can have more than one input argument but the output must be a single expression. For example

```
>> g = @(x,y) [x+y, x.*y]
g =
    @(x, y) [x + y, x .* y]
```

This function has two inputs and one output.

```
>> g(3,4)                                % the function works for numbers
ans =
     7     12
>> A = [1 2; 3 4];
>> B = [1 0; 0 1];
>> g(A,B)                                % the function also works for matrices
ans =
     2     2     1     0
     3     5     0     4
```

TABLE 6.1: Several commands that operate on functions. Here f is the function handle of the function

Command	Result
<code>fplot(f, [minx maxx])</code>	plots the function $f(x)$ on $[\text{minx}, \text{maxx}]$
<code>fminbnd(f, minx, maxx)</code>	returns the x for which $f(x)$ is minimal on $[\text{minx}, \text{maxx}]$
<code>fsolve(f, x0)</code>	returns the x for which $f(x)$ is zero in the neighborhood of x_0
<code>integral(f, x0, x1)</code>	returns the integral of $f(x)$ from x_0 to x_1

Now suppose we need to find a zero around $x = 3$ of our function $f = @(x) \cos(x) .* \sin(2*x)$. In MATLAB this can be done with `fsolve(f, 3)`. The command `fsolve` requires the function handle and not the entire function definition. For the same reason, if we need to find the zero of the standard function named $\sin(x)$ near $x = 3$ then we should do `fsolve(@sin, 3)` and not `fsolve(sin, 3)`. The distinction between function and function handle plays a role in the next section as well.

Let

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

The point at which f takes its minimum is found using `fminbnd`. By default, the relative error is of $1e-4$. 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 = @(x) 1./((x-0.1).^2 + 0.1) + 1./((x-1).^2 + 0.1);
>> fplot(f, [0 2]);
>> xm1 = fminbnd(f, 0.3, 1);
>> fm1 = f(xm1);
>> xm2 = fminbnd(f, 0.3, 1, optimset('TolX', 1e-8));
>> fm2 = f(xm2);
>> [xm1, xm2] % compare the two answers
```

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

Exercise 6.1. Use `fminbnd` to find the maximum of $f(x) = \frac{1}{(x-0.1)^2 + 0.1} + \frac{1}{(x-1)^2 + 0.1}$ in the interval $[0, 0.5]$. Choose an error tolerance such that the maximum is correct to $\pm 10^{-6}$. If x_m denotes the computed solution, check the answer by evaluating f at $x_m + 10^{-6}$ and $x_m - 10^{-6}$. The values should be smaller at both these neighboring points. *Hint:* there is no function `fmaxbnd`. Find the minimum of $-f(x)$, instead. \square

Exercise 6.2. Determine the maximum of the function $f(x) = x \cos(x)$ over the interval $10 < x < 15$, and use `fplot` to plot the function. \square

Exercise 6.3. Find the zero, i.e x_0 , of the functions $f(x) = \cos(x)$ and $g(x) = \sin(2x)$ around the point 2. Use a command from Table 6.1. Check that the values are approximately zero for x_0 and the signs of f at $x_0 \pm 10^{-4}$. \square

Statements like

```
a=10;
g=@(x,y)sin(a.*x.*y);
a=pi;
```

are allowed and it defines $g(x, y) = \sin(10xy)$. The variable a used in the definition of g is not in the argument list (x, y) . In that case the current value of $a=10$ is substituted. Changing a to $a=\pi$ afterwards has no effect on $g(x, y)$.

Exercise 6.4. Explain the outcome of the following command `f=@(x)(@(y)y.^2.*x)`; and test it with `h=f(2)`, `h(10)`, `x=5`; `h(10)`. □

For the function f defined in the last exercise, the command `f(2)(10)` generates an error in MATLAB. In the MATLAB look-alike GNU OCTAVE it is allowed and it does what you expect.

Exercise 6.5. Define a function f such that $f(n)$ is the integral from 0 to 1 of the function t^n . (You must use the command `integral` and your definition of f must use only one line of MATLAB.) □

6.2 Function m-file

Functions m-files are true subprograms, and they can take input parameters and/or return output parameters. They can call other functions as well. Here is an example of a function m-file:

```
function [avr,sd] = stat(x)
%STAT Simple statistics.
%   Computes the average value and the standard deviation of a vector

%Version December 2001. Author: St. Nicholas
n = length(x);
avr = sum(x)/n;
sd = sqrt(sum((x - avr).^2)/n);
return; % this is optional
```

Once saved as `stat.m` somewhere in your path, you can call this function:

```
>> x=1:5;
>> [a,s]=stat(x) % compute average and standard deviation of x
a =
    3
s =
    1.4142
>> b=stat(x) % if we only need the first output (average)
b =
    3
```

The general syntax of a function is:

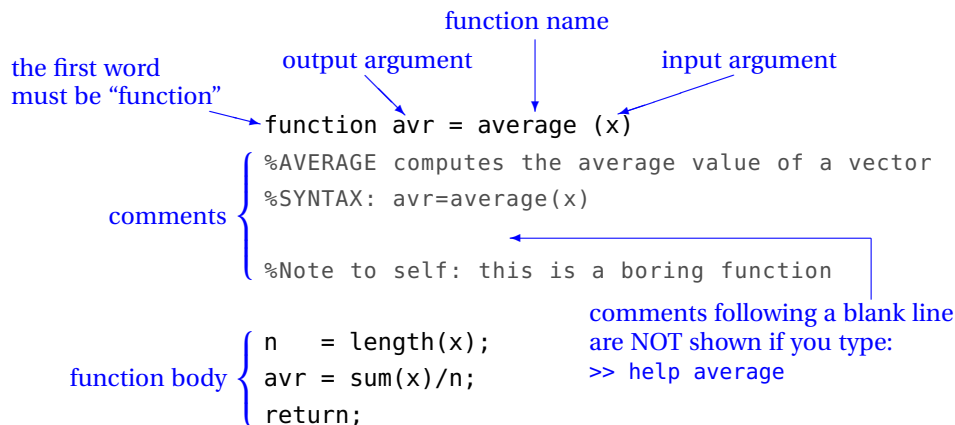
```
function [outputArgs] = function_name(inputArgs)
```

Here `outputArgs` are enclosed in `[]` and they are a comma-separated list of variable names. The brackets `[]` are optional if there is only one output argument. Functions without output arguments are also allowed, as in

```
function function_name(inputArgs)
```

In other programming languages, functions without output arguments are called *procedures*. The `inputArgs` are enclosed in parentheses `()` and they are a comma-separated list of variable names. Functions without `inputArgs` are also allowed.

The first line of the m-file should be the definition of the function (also called a header) that we talked about: `function [outputArgs] = function_name (inputArgs)`. After that, a continuous sequence of comment lines should appear. In this part it is explained what the function does. Not only a general description, but also the expected input parameters, returned output parameters and synopsis should be documented there. These comment lines (counted up to the first non-comment line) are important since they are displayed in response to the `help` command. Finally, the remainder of the function is called *the body*. Function m-files terminate execution and return when they reached the end of the file or, alternatively, when the command `return` is encountered. As an example, the function `average` is defined as follows:



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

Exercise 6.6. Create the function `average` and save it on disk as `average.m`. Remember about the comment lines. Check its usability by calling `help average` and then test the function by typing `avr1=average(1:4)`. Verify the result. □

Warning: The functions `mean` and `std` already exist in `MATLAB`. As long as a function name is used as variable name, `MATLAB` can not perform the function. Many other, easily appealing names, such as `sum` or `prod` are reserved by `MATLAB` functions, so be careful when choosing your names (see Section 13.3).

The `return` statement can be used to force an early return. An exemplary use of the `return` is given below:

```

function d = determinant(A)
%DETERMINANT Computes the determinant of a matrix
[m,n] = size(A);
if (m ~= n)
    disp ('Error. Matrix should be square. ');
    return;
else
    d = det(A);           % standard Matlab function
end
return;                 % this is optional

```

Analyze the use of the `return` command in the function `checkarg`, which is presented in Section 6.4 as well.

When controlling the proper use of parameters, the function `error` may become useful. It displays an error message, aborts function execution, and returns to the command environment. Here is an example:

```
if (a >= 1)
    error ('a must be smaller than 1');
end
```

Change some scripts that you created into functions, e.g. create the function `drag`, computing the drag coefficient (see Section 5.3), or `solve_cos` (see Section 5.5) or `cubic_roots` (see Section 1.7).

Exercise 6.7. Write the function `[elems, mns] = nonzero(A)` that takes as input a matrix `A` and returns all nonzero elements of `A` in the vector `elems` and returns the means of all columns in the vector `mns`. Test your function with the commands below and explain the outcome:

```
>> A=[1 0 0; 2 3 4]
>> [eee,mmm]=nonzero(A)
>> nonzero(A)
>> ee=nonzero(A)
>> [~,mm]=nonzero(A)
```

□

Exercise 6.8. Create the function `[A,B,C] = sides(a,b,c)` that takes three positive numbers `a`, `b` and `c`. If they are sides of a triangle, then the function returns its angles `A`, `B` and `C`, measured in degrees. Display an error when necessary.

□

Exercise 6.9. The area of a triangle with sides of length a , b , and c is $A = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$. Write a function that accepts a , b and c as inputs and returns the area A as output. Note that the sides should be non-negative and should fulfil the triangle inequalities: $a \leq b+c$; $b \leq a+c$; $c \leq a+b$. Make use of the error command.

□

Exercise 6.10. Create the function `myrandint` that randomly generates a matrix of integer numbers (use the command `rand`). These integers should come from the interval $[a, b]$. Exercise in documenting the function. Use the following function header:

```
function r = myrandint(m,n,a,b) % a m-by-n matrix
```

If this seems too difficult, start first with the fixed interval, e.g. $[a, b] = [0, 5]$ (and remove `a` and `b` from the function definition) and then try to make it work for an arbitrary interval.

□

6.3 Subfunctions

A function m-file may contain more than one function. The function appearing first in the m-file is the primary function and only this one can be called directly by the user. Other functions are *subfunctions* and can be called from within the m-file. So, they are invisible from outside the file. A subfunction is created by defining a new function with the function statement after the body of the preceding function. The use of subfunctions is recommended to keep the function readable when it becomes too long or too complicated. For example, in the code below, `average` is now a subfunction within the file `stat.m`:

```
function [a,sd] = stat(x)
%STAT Simple statistics.
```

```

% Computes the average and standard deviation of a vector
n = length(x);
a = average(x,n);
sd = sqrt(sum((x - avr).^2)/n);
return;

function a = average (x,n)
%AVERAGE subfunction
a = sum(x)/n;
return;

```

6.4 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,out2] = checkarg (in1,in2,in3)
%CHECKARG Demo on using the nargin and nargout variables.
if (nargin == 0)
    disp('no input arguments');
    return;
elseif (nargin == 1)
    s = in1;
    p = in1;
    disp('1 input argument');
elseif (nargin == 2)
    s = in1+in2;
    p = in1*in2;
    disp('2 input arguments');
elseif (nargin == 3)
    s = in1+in2+in3;
    p = in1*in2*in3;
    disp('3 input arguments');
else
    error('Too many inputs.');
```

```

end

if (nargout == 0)
    return;
elseif (nargout == 1)
    out1 = s;
else
    out1 = s;
    out2 = p;
end

```

Exercise 6.11. Read the above function `checkarg` and explain what will happen if you type

```

>> checkarg
>> s = checkarg(-6)
>> s = checkarg(23,7)
>> [s,p] = checkarg(3,4,5)

```

□

6.5 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 variables in the workspace. To understand it better consider a function `myfun`

```
function z = myfun (x,y)
x = 2*x;
z = x+y;
```

and that we call this function from the command window with

```
>> x = 100;
>> a = -1;
>> b = 20;
>> c = myfun(a,b) % c = 2a + b
```

Now in the MATLAB workspace, variables `a`, `b`, `c` and `x` are available, and `x` equals 100. The variables `y` and `z` are visible only within the function `myfun` and the same is true for the variable `x` used in `myfun`. It is completely separate from the `x` defined in the Command Window. The fact that they share the same name is irrelevant. Upon executing `myfun(a,b)` the current values of `a`, `b` are copied to the local variables `x`, `y`. Doubling the local variable, `x=2*x`, in the function then has no effect on `a`.

An definite advantage of local variables is that we, as a user of functions, need not worry about the variable names defined in the function. They are completely independent. Sometimes this is a drawback, though, because each time we call a function, the values of the input arguments are copied to the local variables. Now imagine copying a 10000×10000 matrix a number of times. Then it might be an idea to declare the data as `global`. It should be declared `global` in every function or script that needs the data. See `help global`. Any assignment to a global variable is available to all other functions and/or the workspace. However, you should be careful when using global variables. It is very easy to get confused and end up with serious errors.

Exercise 6.12. Make the function `iproduct` as follows

```
function z = iprod(n)
i=10;
z=i*n;
```

Then in the command window do

```
>> z=-1000;
>> x=iproduct(3)
```

and afterwards display the values of `x`, `z`, `n` and `i` from the Command Window. Explain the outcome. □

6.6 Passing functions to functions

We can pass functions to functions, that is, the input argument of a function may itself be a function. A function handle to be precise. We have seen examples of this already when we used `fsolve` to find a zero of a function.

Exercise 6.13. Download the function `funplot` from the course website (it is the function given below) and test it and try to understand how it works.

```
function funplot (F, xstart, xend, col);
%FUNPLOT makes an embellished plot of a function on an interval.
%SYNTAX: funplot(@F,xstart,xend,col)
%       It plots a function F on interval [xstart,xend] in color 'col'.
%       'col' is one of the following: 'b','k','m','g','w','y' or 'r'.
%Default values:
%       [xstart,xend] = [0,10]
%       col = 'b'

% Note: illustrates the use of passing function (handles) to functions

if (nargin == 0)
    error ('No function is provided.');
```

```
end
if (nargin < 2)
    xstart = 0;
    xend   = 10;
end
if (nargin == 2)
    error ('Wrong number of arguments. You should provide xstart and xend.');
```

```
end
if (nargin < 4)
    col = 'b';
end

if (xstart == xend),
    error ('The [xstart, xend] should be a non-zero range.');
```

```
elseif (xstart > xend),
    exchange = xend;
    xend     = xstart;
    xstart   = exchange;
end

switch col
    case {'b','k','m','g','w','y','r'}
        ; % do nothing; the right color choice
    otherwise
        error ('Wrong col value provided.')
```

```
end

x = linspace(xstart, xend,1000); % choice of 1000 is arbitrary
plot (x,F(x),col);
fstring=func2str(F);
description = ['Plot of ',fstring];
title (description);
```

Note the use of comments, the `nargin` variable and the `switch`-construction. Call `funplot` for different built-in functions, like `sin`, `exp`, etc. Test it for your own functions as well. Write for example a function `myfun` that computes $\sin(x \cos(x))$ or $\log(|x \sin(x)|)$. Explain why it would be wrong to use the fragment given below instead of its equivalent part in `funplot`.

```
if nargin < 2
    xstart = 0;
    xend   = 10;
```



```
elseif nargin < 3
    error ('Wrong number of arguments. You should provide xstart and xend.');
```

```
elseif nargin < 4
    col = 'b';
end
```

□

6.7 Scripts vs. functions vs. anonymous functions

The most important difference between a *script* m-file and a *function* m-file is that *all* parameters and variables in a script are externally accessible (i.e. are available in the workspace), while function variables are not. Therefore, a script is a good tool for documenting work, designing experiments and testing for *given* parameters. But you will want to create function m-files to solve a problem for *arbitrary* parameters. Use a script to run functions for specific parameters required by the assignment. Anonymous functions should only be used for simple functions. Recall that anonymous functions are actually function *handles*. The function handle of a function m-file, say, `stat.m` is `@stat`.

Exercise 6.14. Create the function `binom` that computes the value of the binomial symbol $\binom{n}{k}$. Make the function header: `function b = binom (n,k)`. Note that in order to write this function, you will have to create the factorial function, which computes the value of $n! = 1 \times 2 \times \dots \times n$. This may become a separate function (enclosed in a new file) or a subfunction in the `binom.m` file.

Try to implement both cases if you got acquainted with the notion of a subfunction. Having created the `binom` function, write a script that displays on screen all the binomial symbols for $n = 8$ and $k = 1, 2, \dots, 8$ or write a script that displays on screen the following 'triangle':

$$\begin{array}{cccc} \binom{1}{1} & & & \\ \binom{2}{1} & \binom{2}{2} & & \\ \binom{3}{1} & \binom{3}{2} & \binom{3}{3} & \\ \binom{4}{1} & \binom{4}{2} & \binom{4}{3} & \binom{4}{4} \end{array}$$

□

Exercise 6.15. Solve the following exercises by using either a script or a function. The nature of the input/output and display options is left to you. Problems are presented with the increasing difficulty; start simple and add complexity. If possible, try to solve all of them.

1. Write a function with as input a vector x and as output the cumulative product of this x . The cumulative product of a vector x of n elements is the vector of n elements defined as

$$[x_1 \quad x_1 x_2 \quad x_1 x_2 x_3 \quad \dots \quad \prod_{i=1}^n x_i].$$

Make your code as simple as possible. You are not allowed to use the built-in function `cumprod` but it might be useful to check your code.

2. Write a MATLAB function `wmean` with syntax `m=wmean(x,w)` that returns the weighted arithmetic mean $m = \frac{\sum_{i=1}^n x_i w_i}{\sum_{i=1}^n w_i}$ of two vectors x and w of length n . Add error messages to terminate execution of the function if:

- x and w have different length,

- at least one element of w is negative,
- all weights w_i are equal to zero.

3. Compute the value of π using the following series:

$$\frac{\pi^2 - 8}{16} = \sum_{n=1}^{\infty} \frac{1}{(2n-1)^2(2n+1)^2}$$

How many terms are needed to obtain an accuracy of $1e-12$? How accurate is the sum of 100 terms?

4. Write a program that approximates π by computing the sum:

$$\frac{\pi}{4} \approx \sum_{n=0}^m \frac{(-1)^n}{2n+1}$$

The more terms in summation the larger the accuracy (although this is not an efficient formula, since you add and subtract numbers). How many terms are needed to approximate π with 5 decimals? Use the sum to approximate π using 10, 100, 1e3, 1e4, 5e4, 1e5, 5e5 and 1e6 terms. For each of these numbers compute the approximation error. Plot the error as a function of the term numbers used in a summation.

5. The Fibonacci numbers are computed according to the following relation:

$$F_n = F_{n-1} + F_{n-2}, \quad \text{with } F_0 = F_1 = 1$$

- Compute the first 10 Fibonacci numbers.
 - For the first 50 Fibonacci numbers, compute the ratio $\frac{F_n}{F_{n-1}}$. It is claimed that this ratio approaches the value of the golden mean $\frac{1+\sqrt{5}}{2}$. What do your results show?
6. Consider a problem of computing the n -th Fibonacci number. Find three different ways to implement this and construct three different functions, say `fib1`, `fib2` and `fib3`. Measure the execution time of each function (use the commands `tic` and `toc`) for, say, $n = 20, 40$ or 100 .
7. *Collatz conjecture*. Write a script that asks for a positive integer n (or a function that has n as the input argument) and then computes the following: if n is even then divide it by two: $n := n/2$ and if n is odd, then triple it and add one: $n := 3n + 1$ and repeat this procedure until n equals 1. Make provision to count the number of values in (or the length of) the sequence that results.

Example calculation: If $n = 10$, the sequence of integers is 5, 16, 8, 4, 2, 1, so the length is 6.

Make a plot of the length of the sequence that occurs as a function of the integers from 2 to 30. For example, when $n = 10$, the length is 6 while for $n = 15$, the length is 17. Is there any pattern? Try larger numbers to see if any pattern occurs. Is there any integer for which the sequence does not terminate?

The *Collatz conjecture* is that for every n we eventually reach 1 so eventually the procedure ends. This is quite remarkable since even numbers are halved while all odd numbers are more than tripled. The conjecture is open for more than 70 years now. You will be instantly famous if you can prove or falsify the conjecture.

8. Provide all prime numbers that are smaller than the given number n .

□

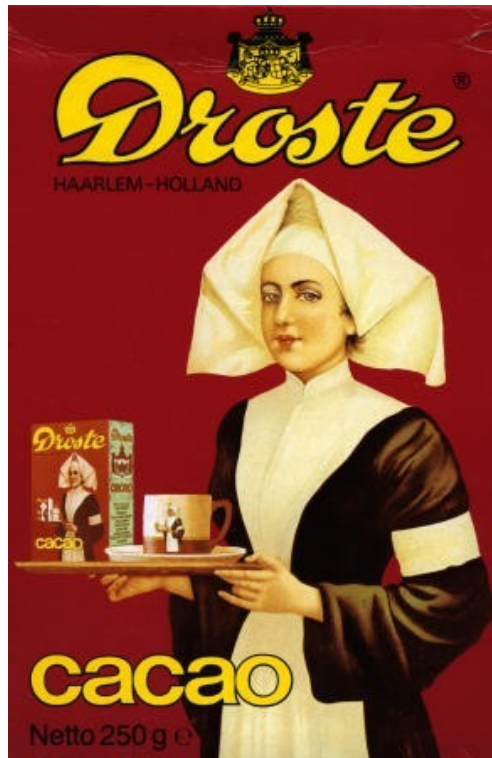


FIGURE 6.1: The *Droste effect*. The woman holds an object bearing a smaller image of her holding the same object, which in turn bears a smaller image of her holding the same object, and so on [wikipedia]. This is an example of recursion, see § 6.8

6.8 Recursion

Recursion is best explained on an example. Consider the function below

```
function m=myfactorial(n)
% myfactorial(n) computes the factorial of nonnegative integer n
if n==0
    m=1;
else
    m=n*myfactorial(n-1);
end
```

This function calls itself! What happens if we do `myfactorial(4)`. This:

- `myfactorial(4)` calls `myfactorial(3)`
- then `myfactorial(3)` calls `myfactorial(2)`
- then `myfactorial(2)` calls `myfactorial(1)`
- then `myfactorial(1)` calls `myfactorial(0)`
- then `myfactorial(0)` returns the value $m = 1$
- now `myfactorial(1)` receives this value 1 and returns $m = 1 \times 1 = 1$
- now `myfactorial(2)` receives this value 1 and returns $m = 2 \times 1 = 2$
- now `myfactorial(3)` receives this value 2 and returns $m = 3 \times 2 = 6$

- finally `myfactorial(4)` receives this value 6 and returns $m = 4 \times 6 = 24$. Ready.

We computed the factorial of 4. The process of functions calling themselves is known as *recursion*. For factorials recursion is overkill, but many problems can be neatly and elegantly solved with recursion. A famous application of recursion is the quicksort algorithm to sort a list of items, say an array of numbers. A **pseudo-code** of this algorithm is:

```

ALGORITHM quicksort
  INPUT: vector S
  OUTPUT: sorted vector
  IF S contains just one element or is empty
    return S
  ELSE
    choose whatever element A of S
    let S1 be the subset of S of elements less than A
    let S2 be the subset of S of elements equal to A
    let S3 be the subset of S of elements larger than A
    return the vector [quicksort(S1), S2, quicksort(S3)]
  END

```

Exercise 6.16. Write a function `quicksort.m` that sorts an arbitrary vector of real numbers, based on the above pseudo-code.

Test your quicksort on `quicksort([5 3 3 4 -1 100])`, and explain why your quicksort routine always returns an answer (i.e. that the number of recursions for each input is finite). □

Exercise 6.17. The Legendre polynomials $P_n(x)$ are defined by the following recurrence relation:

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

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

Chapter 7

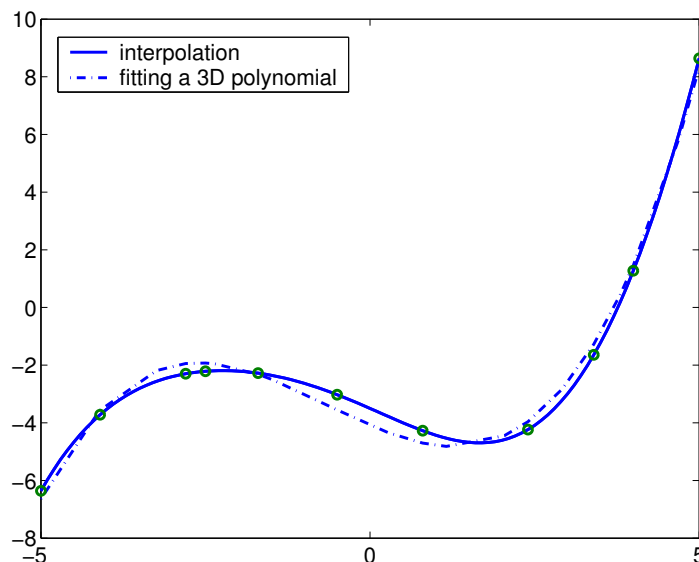
Numerical analysis

Numerical analysis can be used whenever it is impossible or difficult to determine the analytical solution. MATLAB can be used to find, for example, the minimum, maximum or integral of a function. In this chapter we briefly introduce some of the many numerical routines in MATLAB.

There are two basic ways to model data with analytical functions:

- *curve fitting or regression*; finding a smooth curve that best fits (approximates) the data according to a criterion (e.g. best least square fit¹). The curve does not have to go through any of the data points.
- *interpolation*; the data are assumed to be correct, desired in 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:



7.1 Curve fitting

MATLAB interprets the best fit of a curve by the “least squares curve fitting”. The curve used is restricted to polynomials. With the command `polyfit` any polynomial can be fitted to the

¹That is a famous application in linear algebra. For sure you will learn about this method one day.

data. `pc=polyfit(x,y,n)` finds the coefficients of a polynomial $p(t)$ of degree n that fits the data (x_i, y_i) best in least-squares sense. Let's start with finding a linear regression (first order polynomial) of some data:

```
>> x = 0:10;
>> y = [-.10 .24 1.02 1.58 2.84 2.76 2.99 4.05 4.83 5.22 7.51];
>> pc = polyfit(x,y,1); % find the fitting [pc(1) pc(2)] of order 1
>> p=@(x) polyval(pc,x); % define polynomial p(x)=pc(1)x+pc(2)
```

The output of `polyfit` is a row vector of the polynomial coefficients `[.67718 -.39136]`, the solution of this example is $p(x) = .67718x - .39136$.

Exercise 7.1. 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:* with the command `x1 = linspace(xmin,xmax,n)`; you make a vector with n elements evenly distributed from `xmin` till `xmax`. `y1 = polyval(p,x1)`; is the vector of values of the polynomial evaluated at the elements of `x1`. □

Exercise 7.2. 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 $p(x)$ fitting `y` (note that because of added noise the coefficients are different than originally chosen). Try some other polynomials as well. Plot the results in a figure. □

7.2 Interpolation

The simplest way to examine an interpolation is by displaying the function plots with the command `plot`. The neighboring data points are connected with straight lines. This is called a linear interpolation. When more data-points are taken into account the effect of interpolation becomes less visible. Analyze:

```
>> 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 exist also 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 p
>> 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 neighbor interpolation on all x points
>> plot(x,yn,'g')
>> axis tight
>> legend('Original','Crude version','Nearest neighbor interpolation')
>>
>> subplot(2,1,2); plot(xx,yy,'ro-');
>> hold on
>> yc = interp1(xx,yy,x,'linear');
                                % linear interpolation on all x points
>> plot(x,yc,'g')
>> ys = spline(xx,yy,x);        % spline interpolation
>> plot(x,ys,'k')
>> axis tight
>> legend('Crude version','Linear interpolation','Spline interpolation')

```

You can also see a coarse approximation of the function peaks:

```

>> [X,Y,Z] = peaks(10);
>> [Xi,Yi] = meshgrid(-3:.25:3,-3:.25:3);
>> Zi = interp2(X,Y,Z,Xi,Yi);
>> mesh(Xi,Yi,Zi);

```

7.3 Evaluation of functions

As said before, the command `plot` plots a function by connecting defined data points. If a function is constant and not very interesting over some range and then changes abruptly over another, then it might be misinterpreted by using `plot`. The command `fplot` is then more useful. An example is:

```

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

```

Execution of these commands results in a figure with an almost straight blue line and a red sine.

7.4 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 the 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);
>> integral1 = trapz(x,y)
integral1 =
    18.1655

```

```
>> x = 0:0.05:10;
>> y = 0.5 * sqrt(x) + x .* sin(x);
>> integral2 = trapz(x,y)
integral2 =
    18.3846
```

A more accurate result can be obtained by using the command `integral`, which also numerically evaluate an integral of the function f . Let $f = \frac{1}{(x-0.1)^2+0.1} + \frac{1}{(x-1)^2+0.1}$.

```
>> f = @(x) 1./((x-0.1).^2 + 0.1) + 1./((x-1).^2 + 0.1);
>> integral1 = integral(f,0,2)
>> integral2 = integral(f,0,2)
```

You can also add an extra parameter to `integral` specifying the accuracy.

Exercise 7.3. Find the integral of the function $f(x) = e^{-x^2/2}$ over the interval $[-3,3]$. Exercise with different MATLAB commands and different accuracy. \square

Differentiation is done by determining the slope in each data point. A somewhat simplistic way to do this numerically is by first fitting a polynomial, followed by differentiating this polynomial:

```
>> x = 0:10;
>> y = [-.10 .24 1.02 -1.58 2.84 2.76 7.75 .35 4.83 5.22 7.51];
>> p = polyfit(x,y,5)
p =
    0.0026    -0.0554     0.3634    -0.6888     0.2747     0.0877
>> dp = polyder(p)
dp =
    0.0132    -0.2216     1.0903    -1.3776     0.2747
>> 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 7.4. Determine the integral and derivative of the degree 4 polynomial fitted through (x,y) with $y=[12.84 \ 12.00 \ 7.24 \ -0.96 \ -11.16 \ -20.96 \ -27.00 \ -24.96 \ -9.56 \ 25.44]$ and $x=4:13$. \square

7.5 Numerical computations and the control 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

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots$$

with $x = 0.42$:

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


The approximation of $\frac{1}{1-x}$ is returned in s (for 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 7.5. Find the Taylor expansion of $\frac{1}{1-x}$ and $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$ with the absolute precision of $1e-5$ for $x = 0.4$ and $x = -0.4$. \square

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

```

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

```

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

7.6 Numerical solution of differential equations

Suppose that $x: \mathbb{R} \rightarrow \mathbb{R}$ is a function that satisfies the differential equation

$$x'(t) = -x(t)/3$$

with initial condition $x(0) = 2$. It is easy to verify that the solution of this differential equation is $x(t) = 2e^{-t/3}$. In most cases we can not explicitly solve the differential equation. In MATLAB however we can simulate (approximate) the solution using Euler's method and fancier variations. To simulate the solution $x(t)$ of

$$x'(t) = f(t, x(t))$$

given $x(0)$ we can do the following.

```

f=@(t,x) -t*x;      % here we took f(t,x(t)) = -tx(t)
tspan=[0 10];      % time interval over which to solve x(t)
x0=1;              % an initial condition
[t,x]=ode45(f,tspan,x0); % do the simulation (determine t,x(t))
plot(t,x);         % x is of the same length as t

```

Exercise 7.6. Determine the solution of $x'(t) = -x(t)/3$, $x(0) = 2$ using the above ode45 and compare the outcome with the exact solution $x(t) = 2e^{-t/3}$. \square

In the above examples the order of the differential equation (the highest order derivative) is one. In that case only one number $x(0)$ is required to uniquely determine the solution of the differential equation. In applications the order is often higher. In that case you should rewrite the differential equation in vector form

$$x'(t) = f(t, x(t)).$$

For example, the pendulum—see Fig. 7.1 on page 72—is described by the second order differential equation

$$m\ell\phi''(t) + mg\sin(\phi(t)) = 0$$

in the angle $\phi(t)$. With the choice

$$x(t) := \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} := \begin{bmatrix} \phi(t) \\ \phi'(t) \end{bmatrix}$$

this second order pendulum equation becomes a vector valued first order equation,

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ -\frac{g}{\ell} \sin(x_1(t)) \end{bmatrix}.$$

In this form we can simulate the solution. All we need is an initial condition of the vector x at time zero, $x(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix}$. For the rest the method is the same as for the previous case:

```
g=9.8;
el=1;
f=@(t,x) [x(2); -g/el*sin(x(1))]; % define f(t,x) (a column with 2 entries)
x0=[1;0]; % initial condition (column)
tspan=[0 10]; % as before
[t,x]=ode45(f,tspan,x0); % as before
plot(t,x); % x is of the same length as t
```

Exercise 7.7. Solve the coupled differential equation

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} -0.1x_1(t) - x_2(t) \\ x_1(t) - 0.1x_2(t) \end{bmatrix}$$

with initial condition $x_1(0) = 1$ and $x_2(0) = 0$ over time $t \in [0, 50]$. Plot both x_1, x_2 as a function of time, but also plot x_2 against x_1 . (Hint: do whos t x to see what t and x are.) \square

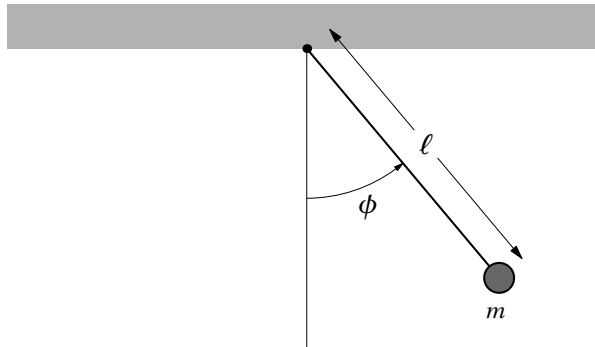


FIGURE 7.1: The pendulum

Chapter 8

Text

Although MATLAB is mainly designed to be efficient with numbers, it may be necessary to operate on text, as well. Text is saved as a character string.

8.1 Character strings

The string is 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. An example:

```
>> t = 'This is a character string'
t =
This is a character string
>> size(t)
ans =
     1     26
>> whos
  Name      Size      Bytes  Class
  t         1x26         52   char array
>> t(10:19)
ans =
  character
>> t([2,3,10,17])
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           % it is transformed to ASCII code
ans =                     % if mathematical operations are used
    102    119
>> t([16:17])-3         % ASCII codes again
ans =
```

```

    96  113
>> char(t([16:17])-2)      % transform ASCII codes to characters
ans =
ar

```

Exercise 8.1. Use string `t = 'This is a character string'` 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'`. □

Using, e.g. `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')        % finds a 'c' in string t; positions are returned
ans =
    11 16
>> findstr(t, 'racter')  % finds the beginning of a string 'racter' in t
ans =
    14
>> findstr(t,u)          % finds string u in string t; returns the position
ans =
    11
>> strcat(u,u1)          % concatenates multiple strings: u and u1
ans =
    characterretcarahc
>> strcmp(u,u1)          % comparison of two strings;
ans =                    % 1 for identical strings; 0 for different strings
    0
>> q = num2str(34.35)     % converts a number into a string
q =
34.35
>> z = str2num('7.6')    % converts a string into a number
z =
    7.6
>> whos q z              % q is a string (a character array); z is a number
  Name      Size      Bytes  Class
  q         1x5         10    char array
  z         1x1         8     double array
>> t = str2num('1 -7 2') % converts a string into a vector of number
t =
     1    -7     2
>> t = str2num('1 - 7 2') % here spaces around the sign - or + are important!
t =                    % performs: [1-7, 2]
    -6     2
>> A = round(4*rand(3,3))+0.5;
>> ss = num2str(A)       % A is random: you will get different numbers here
ss =
-3.5    -3.5     6.5
-2.5    -1.5     0.5
 5.5    -1.5    -3.5
>> whos ss
  Name      Size      Bytes  Class
  ss       3x28       168    char array
>> ss(2,1), ss(3,15:28) % ss is a char array
ans =
-
ans =
.5    -3.5

```

```
>> ss(1:2,1:3)
ans =
-3.
-2.
```

You can get acquainted with other functions operating on character strings. These are listed by calling `help strfun`.

Exercise 8.2. Perform the commands shown in the example above. Define another string, such as `s = 'Nothing wastes more energy than worrying'` and exercise with `findstr`. □

Exercise 8.3. Become familiar with the commands: `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` to get more information). □

8.2 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: ');
```

You have learned in Section 6.2 that inputs can be passed on to functions. This is a recommended approach, since using the `input` command for more complex programs makes automatic computation impossible.

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))                         % a matrix
    0.2221    0.0129    0.8519
    0.4885    0.0538    0.5039
    0.2290    0.3949    0.4239
```

The `fprintf` function (familiar to C programmers) is useful for writing data to a file (see Section 12.1) or printing on 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.
>> str = 'beginning';
>> fprintf('Every %s is difficult.\n',str);
Every beginning is difficult.
```

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 a 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 decimal point of a single or double type.

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

	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.nnnnnE[±]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;

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!\n', 1000*sqrt(2))
look at           1.414214e+3!
>> fprintf('look at %-20.6f!', 1000*sqrt(2))
look at 1414.213562      ! >>
```

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. The difference in the presentation is caused by the conversion characters `e` and `f`.

Exercise 8.4. Try to exercise 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 8.5. 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 euro.\n',name, age, salary);
My name is John. I am 30. My salary is 6130.50 euro.
>>
>> x = [0, 0.5, 1];
>> y = [x; exp(x)]                % y has two rows and three columns
y =
      0      0.5000      1.0000
  1.0000      1.6487      2.7183
>> fprintf('%6.2f %12.8f\n',y);    % but now it is transposed! (see below)
  0.00      1.00000000
  0.50      1.64872127
  1.00      2.71828183
>>
>> fprintf('%6.1e %12.4e\n',y);
  0.0e+00      1.0000e+00
  5.0e-01      1.6487e+00
  1.0e+00      2.7183e+00
>>
>> 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 answers. 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 euro.\n',name,salary);
My name is John. I am 6.130500e+03. My salary is >>

```

The command `fprintf` when given a matrix as argument reads the elements of this matrix *columnwise!* Invoking `fprintf('%6.2f %12.8fn', y)` as above keeps on reading groups of 2 elements of this matrix until all elements are read. Since here `y` is a 2×3 matrix it can do this three times. This is why the result has three rows instead of two.

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 euro.\n',name,age,salary)
str =
My name is John. I am 30. My salary is 6130.50 euro.

```

Exercise 8.6. Define a string

`s='How much wood could a wood-chuck chuck if a wood-chuck could chuck wood?'`.

Exercise with `findstr`, i.e. find all appearances of the substrings 'wood', 'o', 'uc' or 'could'.

Try to build `ss` by concatenation of separate words. Try to do this in a few ways, e.g. make use of `strcat`, `disp`, `sprintf` or `fprintf`. □

Exercise 8.7.

1. Write a script/function that converts a Roman numeral to its decimal equivalent. There are two distinct situations. The 'old' style where the order of the symbols does not matter. In this case, IX and XI both mean 10 + 1 or 11. You should be able to handle the following conversion table:

Roman	Decimal
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

The 'new' style where the order of the symbols does matter. For example, IX is 9 (10 - 1), XC is 90 (100 - 10). The conversion table given above still holds and you may assume for this case that the only instances of order you will encounter are: IV (4), IX (9), XL (40), XC (90), CD (400) and CM (900). The function input will be useful here. *Hint:* try the case of the 'old' style first.

2. Write a function that will do the inverse of the previous problem — convert a decimal number into a Roman number.

□

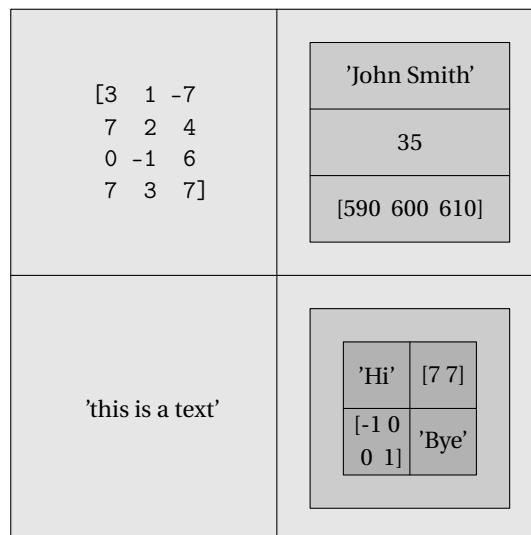
Exercise 8.8. Write a coder (and a decoder) based on a Caesar encoding scheme. This scheme is based on a fixed shift of all letters of the alphabet to the right, e.g., say with a shift 1, 'abcde' becomes 'bcdef' and 'alphabet' becomes 'bmqibcfu'. Of course, 'z' becomes then 'a'. You can imagine a shift with 2, 3, etc, but the shift 26 gives again the same alphabet. Write two functions `coder` and `decoder` whose one of the input arguments is `shift`. The second input argument should be a string or a text file where the message is written. In case of coding, the string or the text file should contain the original message, and in case of decoding, you should provide an encoded string or file. □

Chapter 9

Cell arrays and structures

9.1 Cell arrays

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



This particular cell array is defined in the following code. Cell arrays can be built up by assigning data to each cell. Cells are defined and accessed by brackets `{}`. For example:

```
>> 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} = [590 600 610];
>> A{1,2} = B;           % A{1,2} is the cell B

>> C{1,1} = 'Hi';
>> C{2,1} = [-1 0;0 -1];
>> C{1,2} = [7,7];
>> C{2,2} = 'Bye';
>> A{2,2} = {C};       % A{2,2} is the CELL THAT CONTAINS cell C

>> A
```

```

A =
    [4x3 double]    {3x1 cell}
    'this is a text' {1x1 cell}
>> A{2,1}
ans =
this is a text
>> A{2,1}(1:4)
ans =
this
>> A{2,2}
% is itself a cell
ans =
    {2x2 cell}
>> A{2,2}{1}
% A{2,2}{1} is our cell C
ans =
    'Hi'          [1x2 double]
    [2x2 double] 'Bye'
>> A{2,2}{1}{2,1}
% the {2,1} element of cell A{2,2}{1}
ans =
    -1     0
     0    -1
>> A{2,2}{1}{2,1}(1,1)
ans =
    -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:

```

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

```

Exercise 9.1. 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. □

9.2 Structures

Structures are MATLAB arrays with data objects composed of *fields*. Each field contains one item of information. For example, one field might include a string representing a name, another a scalar representing age or an array of the last few salaries. Structures are especially useful for creating and handling a database. One of the possible ways to create a structure is by assigning data to individual fields. Imagine that you want to construct a database with some information on workers of a company:

```

>> worker.name = 'John Smith';
>> worker.age = 35;
>> worker.salary = [5900, 6000, 6100];
>> worker =
    name: 'John Smith'
    age: 35

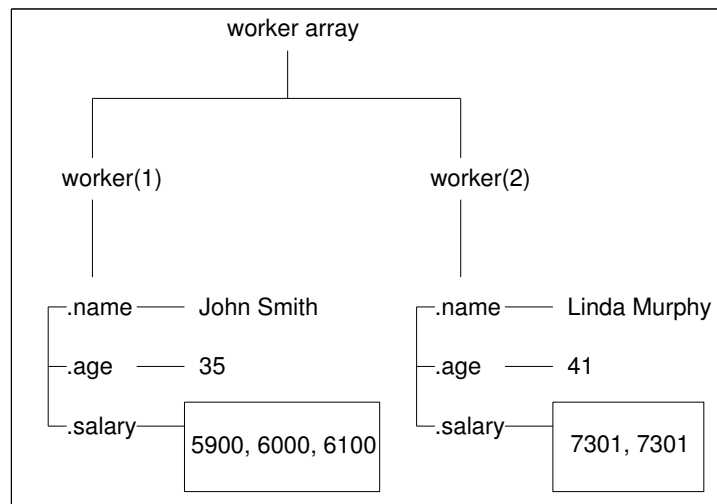
```

```
salary: [5900 6000 6100]
```

In this way, a 1×1 structure array worker with three fields: name, age and salary is constructed. To expand the structure, add a subscript after the structure name:

```
>> worker(2).name = 'Linda Murphy'; % after this, a 2nd subarray is created
>> worker(2).age = 41;
>> worker(2).salary = [7301, 7301]; % field sizes do not need to match!
>> worker
1x2 struct array with fields:
    name
    age
    salary
```

Since this structure has now the size of 1×2 , MATLAB does not display the contents of all fields. The data are now organized as follows:



Structures can also be build by using the struct function. For example:

```
>> employee=struct('name','John Smith','age',35,'salary',[5900, 6000, 6100]);
```

To access an entire field, include a field name after a period. To access a subarray, follow the standard way of using subscripts:

```
>> worker(1).age
    35
>> worker(2).salary(2)
    7301
>> worker(2)
    name: 'Linda Murphy'
    age: 41
    salary: [7301 7301]
```

An alternative way is to use the getfield function:

```
>> getfield(worker,{2},'salary')
    7301    7301
```

There exists also a function setfield, which assigns values to a given field. New fields can be added or deleted from every structure. To delete a field, use the command rmfield.

Analyze the following example:

```
>> worker2 = worker;
>> worker2 = rmfield (worker, 'age');
>> worker2(2).street = 'Bakerstreet 5';
>> worker2(2)
    name: 'Linda Murphy'
    salary: [7301 7301]
    street: 'Bakerstreet 5'
>> worker2(1)
    name: 'John Smith'
    salary: [5900 6000 6100] % in all other substructures address field is empty
    street: []                % you should assign new values for each substructure
```

Operating on fields and field elements is done in the same way as on any other array. Consider the following example, where the average salary for each worker is computed.

```
avr_salary(1) = mean (worker(1).salary);
avr_salary(2) = mean (worker(2).salary);
```

Remark: structures as a concept are common organizations of data in other programming languages. They are created in different ways, but the intention remains the same.

Exercise 9.2. Construct a structure friend, with the fields: name, address, age, birthday. Insert a few friends with related information. Remove e.g. the field age and add the field phone. □

Exercise 9.3. Suppose you saved some data x in a mat-file and your friend also saved some data x. With structures you can load both your data and your friend's data: try and explain the following code:

```
>> x=1;
>> save mydata
>> x=10;
>> save frienddata
>> clear
>> mine = load('mydata')
>> his = load('frienddata')
```

□

Concerning the use of lists and parentheses in MATLAB, see help lists and help paren.

9.3 Classes and object oriented programming

Classes and *objects* allow for adding new data types and new operations to MATLAB. For instance, the class of a variable describes the structure of the variables and the operations permitted as well as functions to be used. An object is an instance of a particular class. The use of classes and objects is the basis of *object-oriented programming*, which is also possible in MATLAB. It is outside the scope of this course.

Chapter 10

Symbolic computation

Even though MATLAB was created to facilitate numerical calculations, nowadays it features some symbolic computations as well. The syntax resembles that of MAPLE. For instance it is possible to solve the equation $x^2 + 3x + 2 = 0$ exactly, and to solve the differential equation $y^{(2)}(t) + 3y^{(1)}(t) + 2y(t) = \sin(t)$ for $t \in \mathbb{R}$ analytically. In this chapter we explain the basics only.

10.1 Symbolic objects

A symbolic object can be a variable, a number or an expression made of symbolic variables and numbers. These objects can be defined using `sym` or `syms`. A common situation is:

```
>> a = sym('a')
a =
a
>> syms b c      % this is equivalent ...
>> b = sym('b') % .. to this and
b =
b
>> c=sym('c')   % .. this: now b c are symbolic variables
c =
c
>> d = a+5
d =
a+5
>> e = a+b+5;
```

As usual MATLAB displays the result of the expressions if not suppressed by `;`. A difference is that displayed symbolic objects are not indented. As we shall see the syntax and command of the symbolic toolbox is similar to that of MAPLE, but some MATLAB conventions are maintained like the `;`.

Exercise 10.1. Clear your workspace, and define the symbolic variables x , y , and z , where z is $x + y + \sqrt{5}$. Furthermore, define the (normal) variable a as $\sqrt{5}$. Finally, type `who` and `whos` and note how the symbolic variables are displayed differently from the non-symbolic ones. \square

Creating a symbolic expression is also easy.

```
>> syms x y
>> f = x^2 + y^2 + 3/2      % A symbolic expression
f =
x^2 + y^2 + 3/2
>> g = (1/3)*x^2 - 5/4;    % Another symbolic expression (not displayed)
```

```

>> F = f+g
F =
4/3*x^2+y^2+1/4
>> G=subs(F,1)           % substitute x=1
G =
19/12+y^2

```

So we see that it does the calculations exact and directly.

Exercise 10.2. Define the symbolic expressions $f = x^3 + 3x^2 + 3x + 1$ and $g = (x + 1)^2$. Next calculate $f + g$, $f - g$, $f * g$, and f / g . □

As you might have noticed in the last exercise, the symbolic expression are not simplified, only simple calculations are done.

Exercise 10.3. Use `simplify` to simplify the answers of the previous exercise. □

Exercise 10.4. Define the symbolic expression $f = \sin(x) + x^2$. Next perform the commands `int` and `diff` on this expression and guess what these commands do. You can use `help` to find more about it. □

10.2 Solving symbolic expressions

Many times symbolic expression have to be solved. With the `solve` command this is easy. Standard the command `solve('eq')`, solves the equation $eq=0$.

```

>> syms x y
>> solve(exp(x)-5)           % This solves exp(x)=5
ans =
log(5)
>> solve(x-5+y,x)          % This solves in x+y=5 for x
ans =
5-y
>> f = x^2 + 3*x + 2;
>> xzero= solve(f)
xzero =
-1
-2
>> xzero = solve('x^2+3*x+2=0'); % Same as above
>> yzero = solve('y^2+3*y+3-9')
yzero =
-3/2+1/2*i*3^(1/2)
-3/2-1/2*i*3^(1/2)
>>whos
Name          Size          Bytes  Class

ans           1x1           136   sym object
f             1x1           142   sym object
x             1x1           126   sym object
xzero        2x1           192   sym object
y            1x1           126   sym object
yzero        2x1           256   sym object

Grand total is 65 elements using 978 bytes

```

We see that the zeros of the expression are treated as symbolic objects. It is easy to see which numerical value `xzero` have, but of `yzero` this is not immediately clear. With the command `double` one can convert a symbolic number to a MATLAB number.

Exercise 10.5. Find the zeros of x^2+4x+2 and that of x^2+2x+4 . Calculate the absolute value of all the zeros. Which of the (four) zeros has the largest absolute value? \square

One can also solve a system of equations.

```
>> syms x y
>> [x1,y1]=solve('y+x=5','x-y=3',x,y)
x1 =
4
y1 =
1
```

Exercise 10.6. Determine the intersection points of the line $x+3y=1$ with the circle $x^2+y^2=2$. \square

10.3 Solving ordinary differential equations

An ordinary differential equation can be solved symbolically with the command `dsolve`. An ordinary differential equation can also be solved numerically using the command `quad`. However, in this section we only show how it can be done symbolically. The derivative of the function f is denote Df , and the second derivative by $D2f$.

```
>> syms f g t
>> dsolve('Df+f=5')
ans =
5 + exp(-t)*C1
>> dsolve('Df+f=5','f(0)=4')
ans =
5 - exp(-t)
>> dsolve('D2g+3*Dg+2*g=0')
ans =
C1*exp(-t)+C2*exp(-2*t)
>> pretty(ans)

          C1 exp(-2 t) + C2 exp(-t)
```

Exercise 10.7. Solve the differential $y^{(2)}(t)+3y^{(1)}(t)+2y(t)=t$ with $y(0)=0$, and $y^{(1)}(0)=3$. \square

Exercise 10.8. Solve the differential equation $y^{(2)}(t)+3y^{(1)}(t)+3y(t)=e^{-t}$ with $y(0)=y(1)=0$. \square

The answer to differential equation can be quite long and complicated. Thus it would be nice if one can plot the answer. Since the answer is symbolic expression, one cannot use the `plot`-command. The `ezplot`-command plots symbolic expressions.

```
>> syms f t
>> f1=dsolve('Df+f=5','f(0)=4')
f1 =
5 - exp(-t)
>> ezplot(f1)
>> ezplot(f1,[0,6])           % Plots the function on the range 0 < t < 6
```

Exercise 10.9. Plot the solution of the differential equation of the previous exercise. \square

Exercise 10.10. Type the following commands and see what happens.

```
>> syms x y t
>> S=x^2 + 3*x*y + y^2 - 10
>> ezplot(S)
>> f=cos(2*t)
>> g=sin(4*t)
>> ezplot(f,g)
```

□

10.4 From symbolic function to function handle

```
>> syms f,t;
>> f=sin(t)/t;           % symbolic function
>> fh=matlabFunction(f) % function handle
fh =
    @(t) sin(t)./t
```


Chapter 11

Optimizing the performance of MATLAB code

MATLAB means “Matrix Laboratory”. It is optimized for matrix operations. For best performance of your code, you should always try to take advantage of this fact.

11.1 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 than loops. Most built-in functions support vectorized operations. So, when possible, try to replace loops with vector operations. For instance, instead of using:

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

try this:

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

Copying or other operations on 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);           % A and B have the same size
end
```

and the vectorized code of the last loop:

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

or 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 a proper use of the colon notation make the 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 if possible, by removing loops.*

Exercise 11.1. Look at your script or function m-files already written, especially your loop exercises. Try to vectorize their codes. □

Exercise 11.2. Try to vectorize the following codes:

```
% code 1
n = 20;
m = 10;
A = rand(n,m);
for i=1:n
    for j=1:m
        if (A(i,j) > 0.5)
            A(i,j) = 1;
        else
            A(i,j) = 0;
        end
    end
end
end
```

```
% code 2
n = 20;
m = 10;
A = randn(n,m);
x = randn(n,1);
p = zeros(1,m);
for i=1:m
    p(i) = sum (x .* A(:,i)); % what is p?
end
%
%
```

□

11.2 Array preallocation

In 'real' programming languages the allocation of memory is necessary. It is not needed in MATLAB, but it might improve execution speed. Moreover, practicing it, is a good habit. Pre-allocate the arrays, which store your output results. This prevents MATLAB from resizing an array each time you enlarge it. Preallocation also helps to reduce memory fragmentation if you work with large matrices. During a MATLAB session, memory can be fragmented. As a result, there may be plenty of free memory, but insufficient in continuous blocks to store a large variable.

Exercise 11.3. Create two scripts with the listings given below. Run them and compare the results, i.e. the measured times of their performances. The pair of commands `tic` and `toc` is used to measure the execution time of the operations:

```
% script 1
clear s x;
tic;
x = -250:0.1:250;

for i=1:length(x)
    if (x(i) > 0)
        s(i) = sqrt(x(i));
    else
        s(i) = 0;
    end
end
```

```

end
end
toc;

```

```

% script 2
clear s x;
tic;
x = -250:0.1:250;
s = zeros (size(x)); % preallocate memory
for i=1:length(x)
    if (x(i) > 0)
        s(i) = sqrt(x(i));
    end
end
end
toc;

```

Remember that scripts work in the workspace, interfering 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. □

Exercise 11.4. Consider the following sequence of commands:

```

d = pi/50;
n = round(2 + pi/d);
m = round(n/2);
for j = 1:m
    x(j) = (j+3)*d;
    y(j) = cos(2*x(j));
end
for j = m+1:n
    x(j) = (j-1)*d;
    y(j) = cos(4*x(j));
end

```

Improve the performance of the code above. Pre-allocate memory for the vectors `x` and `y`. Vectorize the calculation of these vectors. □

11.3 MATLAB tricks and tips

Many of the MATLAB's tricks use the fact that there are two ways of addressing matrix elements using a vector as an index:

1. 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

```

2. If x and y are vectors of the same size and y only consists of 0s and 1s 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 x . 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 to decide which solution is faster.

- 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 elements
```

- Initialize a vector with a constant.

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

- Create an $n \times n$ matrix of

```
>> n = 1000;
>> A = 3 * ones(n,n); % 1st solution
>> B = repmat(3,n,n); % 2nd solution - should be much faster!
```

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

```

>> m = 10;
>> x = 1:5;
>> A = ones(m,1) * x;           % 1st solution
>> B = x(ones(m,1),:);        % 2nd solution - should be fast

```

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

```

>> n = 5;
>> x = (1:5)';
>> A = x * ones(1,n);         % 1st solution
>> B = x(:,ones(n,1));       % 2nd solution - should be faster

```

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

```

>> n = 5;
>> 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:

```

>> x = [3 -1 4 2 7 2 3 5 4];
>> n = length(x);
>> x(n:-1:1)           % 1st solution
>> fliplr(x)          % 2nd solution - should be faster

```

Reverse one column of a matrix:

```

>> A = round(5*rand(4,5));
>> c = 2;
>> A(:,c) = flipud(A(:,c));

```

- Interchange rows or columns of a matrix:

```

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

```

- Make a column vector from a matrix A by concatenating its columns, $A(:)$.
- Reshape an $(m*n)$ -by-1 vector x into an m -by- n matrix whose elements are taken column-wise from x .

```

>> A = round(5*rand(4,5))
>> x = A(:); % x is now a 20-by-1 vector
>> B = reshape(x,4,5); % B is now a 4-by-5 matrix; B = A;
>> C = reshape(x,5,4); % B is now a 5-by-4 matrix; B is NOT A'!

```

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

```
>> A = round (5*rand(4,5));
>> B = round (6*rand(3,6));
>> 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)
```

- Find the elements in a vector x which are different from a vector y:

```
>> x = [1 5 4 1 7 -1 2 2 6 1 1];
>> y = [5 2 2 7 8 4 4];
>> setdiff(x,y)
```

- Derive cumulative sums of a vector x:

```
>> x = [1 5 4 1 7 1 2 2 6 1 1];
>> z = cumsum(x);    % z is a vector of cumulative sums
```

- Given x, determine a vector of differences $[x(2) - x(1) \quad x(3) - x(2) \quad \dots \quad x(n) - x(n-1)]$:

```
>> x = [1 5 4 1 7 1 2 2 6 1 1];
>> z = diff(x);
```

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

```
z = zeros(n,1);
for i=1:n
    z(i) = A(i,:) * B(i,:)' ;
end
```

The solutions:

```
>> z = diag(A * B');    % 1st solution
>> z = sum (A .* B, 2);    % 2nd solution solution - should be faster
```

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

```
>> A = round (7*rand(4,5));
>> [n,m] = size(A);
>> x = (1:n)';    % x is an n-by-1 vector
>> B = A ./ x(:,ones(m,1));
```

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

```
>> A = round (7*rand(4,5));
>> [n,m] = size(A);
>> x = 1:m; % x is 1-by-m
>> B = A ./ x(ones(n,1),:);
```

Actually, since 2017 MATLAB does this very efficiently with $B=A ./ x$ which works for both of the above to cases!

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
```


Chapter 12

File input/output operations

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

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

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

- 'r' - for reading only
- 'w' - for writing only
- 'wt' - for writing only, in text mode
- 'a' - for appending only
- 'r+' - both for reading and writing

Here is an example:

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

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

It is a good habit to test the file identifier when you open a file, especially for reading because perhaps the file does not exist.

Exercise 12.1. Create a script with the code given below and check its behavior when you give a name of a non-existing file (e.g. `noname.txt`) and a readable file (e.g. one of your functions). □

```
fid = 0;
while fid < 1
    fname = input ('Open file: ', 's');
    [fid, message] = fopen (fname, 'r');
    if (fid == -1)
        disp('couldn't find the file');
    else
        disp('yes, I found the file and opened it');
        fclose(fid)
```

```
end
end
```

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

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

Type also help `fileformats` to find out which are readable file formats in MATLAB.

12.1 Text files

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

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

For more detailed description, see Section 8.2. Consider the following example:

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

Exercise 12.2. Prepare a script that creates the `sintab.txt` file, containing a short table of the sine function. □

The `fscanf` command is used to read a formatted text file. The general function definition is:

```
[A,count] = fscanf (fid, format, size)
```

This function reads text from a file specified by file identifier `fid`, converts it according to the given format (the same rules apply as in case of the `fprintf` command) and returns it in a matrix `A`. `count` is an optional output argument standing for the number of elements successfully read. The optional argument `size` says how many elements should be read from the file. If it is not given, then the entire file is considered. The following specifications can be used:

- `n` - read at most `n` elements into a column vector;
- `inf` - read at most to the end of the file;
- `[m,n]` - read at most `m,n` elements filling at least an `m`-by-`n` matrix, in column order; `n` can be `inf`.

Here is an example:

```
>> a = fscanf (fid, '%5d', 25);      % read 25 integers into a vector a
>> A = fscanf (fid, '%5d', [5 5]);  % read 25 integers into a 5 x 5 matrix A
```

MATLAB can also read lines from a formatted text and store it in a string. Two functions can be used for this purpose, `fgets` and `fgetl`. The only difference is that `fgetl` copies the newline character while `fgets` does not.

Exercise 12.3. Create the following script and try to understand how it works (use the `help` command to learn more on the `feof` function):

```
fid = fopen('sintab.txt','r');
mytitle = fgetl(fid);
k = 0;
while ~feof(fid)
    k = k+1;
    line = fgetl(fid);
    tab(k,:) = str2num(line);
end
fclose(fid);
```

Look at the matrix `tab`. How does it differ from the originally created matrix? □

Reading lines from a formatted text file may especially be useful when you want to modify an existing file. Then, you may read a specified number of lines of a file and add something at the found spot.

Exercise 12.4. Create a script that reads the `exptab.txt` file and at the end of the file adds new exponential values, say, for $x = 1.1 : 0.1 : 3$. Note that you should open the `exptab.txt` file both for reading and writing. □

A pair of useful commands to read and write ASCII delimited file (i.e. columns are separated by a specified delimiter such as space ' ' or tab, 't') is `dlmread` and `dlmwrite`. A more general command is `textread`, which reads formatted data from a text file into a set of variables. Not only numeric data are read, but also characters and strings.

Exercise 12.5. Write the function `countlet` that: opens a file specified by a name, reads this file line by line and counts the number of appearance of the specified letter (so, there are two input arguments: the name and the letter). Remember to return an error message when there is a problem with opening the file. Create a text file `test.txt` with some text, e.g. by writing random words or retyping a few sentences of a book. Test your function by calling:

```
>> c = countlet('test.txt','d');
>> c = countlet('countlet.m','a'); % YES, you can do this!
```

□

12.2 Binary files

There are two important variables to write and read from a binary file: `fread` and `fwrite`. The definition of the function `fwrite` is given below:

```
count = fwrite(fid, A, precision)
```

This command writes the elements of matrix `A` to the provided file, while converting values to the specified precision. The data is written in column order. `count` is the number of successfully written elements.

The definition of the function `fread` is given below:

```
[A, count] = fread(fid, size, precision)
```

This reads binary data from the specified file and writes it into matrix A. `count` is an optional parameter, which returns the number of elements successfully read. The size argument is optional and its specification is the same as in case of the `fscanf` function (see Section 12.1). The precision argument controls the number of bits read for each value and their interpretation as character, integer or floating-point values, e.g. `'uchar'`, `'int16'` or `'single'` (learn more from `help`). By default, numeric values are returned in double precision.

Exercise 12.6. Exercise with the following scripts:

```
% Write the 5-by-5 magic square into binary file
M = magic(5); % creates a magic square
fid1 = fopen ('magic5.bin','w');
count = fwrite (fid1, M, 'int16'); % count should be 25
% fwrite reads M columnwise!
fclose(fid1);
```

Note that a 50-byte binary file should appear (25 integer numbers, each stored in 2 bytes).

```
% Read the 5-by-5 magic square into binary file
fid2 = fopen ('magic5.bin','rb');
[A, count] = fread (fid2, [5,5], 'int16'); % count should be 25
fclose(fid2);
```

Check what will happen if you skip the specification of either size or type, e.g.

```
[B, count] = fread (fid2);
```

or

```
[B, count] = fread (fid2, [5,5]);
```

Note that each time you should open and close the `magic5.bin` file. □

A particular location within a file is specified by a *file position indicator*. This indicator is used to determine where in the file the next read or write operation will begin. The MATLAB functions operating on the file position indicator are summarized below:

Function	Purpose
<code>feof</code>	determines if file position indicator reached the end-of-file
<code>fseek</code>	sets file position indicator to the specified byte with respect to the given origin
<code>ftell</code>	returns the location of file position indicator
<code>frewind</code>	resets file position indicator to beginning of file

To understand how `fseek` and `ftell` work, consider this script (you can use on-line help to learn more about these function's specifications):

```
a = 1:5;
fid = fopen ('five.bin','w');
count = fwrite (fid, a, 'single');
fclose(fid);
```

Five integers are written to the file `five.bin` with a single precision, so each number uses 4 bytes. Try to understand what is happening here:

```
>> fid = fopen ('five.bin','r'); % open for reading
>> status = fseek (fid, 12, 'bof'); % move the file position indicator forward
```

```
>> % 12 bytes from the beginning of file 'bof'
>> % status is 0 if operation was successful
>> four = fread (fid, 1, 'single'); % read one element at the current position,
>> % starting from byte 13 (the fourth number)
>> pos = ftell (fid); % check the position; the number 4 is read,
>> % so it should be 16 now
>> status = fseek (fid, -8, 'cof'); % move the file position indicator backward
>> % 8 bytes from the current position 'cof'
>> pos = ftell (fid); % check the position; it should be 8
>> three = fread (fid, 1, 'single'); % read one element, so it should be 3
>> fclose(fid);
```


Chapter 13

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, where structured programming is indispensable.

13.1 Structural programming

Never write all code at once; program in small steps and make sure that each of these small steps work as expected, before proceeding to the next one. Each step should be devoted to only one task. Do not solve too many tasks in one module. This is called *structured* or *modular* 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 benefit of structural programming are: easier error detection and correction, modifiability, extensibility and portability. A general approach to a program development is presented below:

1. **Specification.** Read and understand the problem. The computer cannot do anything itself: 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 this problem?
 - What are the results/outputs for this problem?
 - What form should the inputs/outputs be provided in?
 - What sort of algorithms is needed to find the outputs from the inputs?
2. **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, for example, 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.

As an example, 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:

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

```
[d1, d2] = read_data ('data1.dat', 'data2.dat');
[res1, err1] = solve_problem (d1);
[res2, err2] = solve_problem (d2);
compare_results (res1, res2, err1, err2);
```

- This is the second level. The following functions `read_data`, `solve_problem` and `compare_results` belong here. Each of them has to be defined in a separate file:

```
function [d1, d2] = read_data (fname1, fname2)
% Here should be some description.
%
fid1 = fopen (fname1,'r');
.... % check whether the file fname1 exists
fclose(fid1);
fid2 = fopen (fname2,'r');
.... % check whether the file fname2 exists
fclose(fid2);
....
d1 = ...
d2 = ...
return;
```

```
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;
```

3. **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 13.3).

4. **Running and debugging (see also Section 13.2).** Bugs will often exist in a newly written program. Never, ever, believe or assume that the code you just created, works as intended. *Always check the correctness of each function or script: Twice.* You may add some extra lines to your code which will present the intermediate results (screen displays, plots, writes to files) to help you controlling what is going on. Those lines can be removed later.
5. **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 that good tests are those for which the answers are known. Your program should produce correct results for normal test conditions as well as boundary conditions.
6. **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 problem, you will be able to easily modify it after some time. Take a responsibility to improve your solutions or correct your errors when found later.

13.2 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 by 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 *defensive programming*:

- Do not assume that input is correct, simply check.
- 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 a part of the early debugging process. Another important part is modularity, breaking large task into small subtasks, which allows for developing tests for each of them more easily. You should always remember to run the tests again after the changes have been made. To make this easy, provide extra print statements that can be turned on or off.

MATLAB provides 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. To use the debugging facility to find out what is happening, you start with the `dbstop` command. This command provides a number of options for stopping execution of a function. A particularly 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 variables or step through execution from that point. The command `dbstep` allows you to step through execution one line at a time. You can continue execution with the `dbcont`. To exit debug mode, type `dbquit`. For more information, use `help` for the following topics: `dbstop`, `dbc clear`, `dbcont`, `dbstep`, `dbtype`, `dbup` and `dbquit`.

13.3 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. This will also allow you to quickly understand what you did in your program when you look at it weeks or months from now. The style conventions are for the reader only, but *you* will become that reader one day.

Some style requirements and style guidelines are presented below. These are recommendations, and some personal variations in style are acceptable, but you should not ignore them. It is important to organize your programs properly since it will improve the readability, make the debugging task easier and save time of the potential readers.

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 i=1:length(x)
    if x(i) > 0
        s(i) = sqrt(x(i));
    else
        s(i) = 0;
    end
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:

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

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.
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 you noticing it!
8. Variables should have meaningful names. You may also use the standard notation, e.g. x , y are real-valued, i , j , k are indices and n is an integer. This will reduce the number of comments and makes 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 like `mean`, `end`, `sum` etc (check in 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 white spaces; both horizontally and vertically, since it will greatly improve the readability of your program. Blank lines should separate larger blocks of the code.

References

- MATLAB. <http://www.mathworks.com/>.
- All sorts of tips and tricks in MATLAB:
http://www.ee.columbia.edu/~marios/matlab/matlab_tricks.html
- D.C. Lay, *Linear algebra and its applications*, Addison-Wesley, 1994.
- A. Gilat, *Matlab An introduction with Applications*, second edition, John Wiley & Sons, 2005.

Index

- `*`, 9, 20
- `+`, 9, 20
- `<=`, logical, 43
- `<`, logical, 43
- `==`, logical, 43
- `>=`, logical, 43
- `>`, logical, 43
- `\`, 9
- `^`, 9
- `-`, 9, 20
- `.*`, 20
- `>>`, 1
- `./`, 20
- `./`, 17, 18
- `.\`, 20
- `.^`, 20
- `.^`, 17
- `.dat`, 6
- `.m`, 5
- `.mat`, ix, 6
- `.txt`, 6
- `/`, 9
- `|`, logical, 43
- `~`, logical, 43
- `~=`, logical, 43
- `&`, logical, 43
- `[]`, 22
- `%`, 3
- `^`, 20

- `@`, 55
- algorithm, ix
- anonymous function, 55
- ans, 2
- array, 15
- ASCII, ix
- axis, 35, 36
- axis equal, 35, 36
- axis image, 35, 36
- axis normal, 36
- axis off, 36
- axis on, 36
- axis square, 35

- axis tight, 36

- binary file, ix
- binary system, viii
- bit, viii
- box off, 35
- box on, 35
- bug, ix
- byte, viii

- character, ix
- clear, 5
- clear all, 5
- clf, 35
- colon notation, 16
- command, ix
- complex conjugate transpose, 17
- complex numbers, 11
- constant, ix
- cos, 3

- D2f, 85
- data, ix
- data type, ix
- debugging, ix
- Df, 85
- diag, 20
- diff, 84
- directory, ix
- disp, 47
- double, 84
- dsolve, 85

- eig, 21
- element-wise
 - division, 17
 - product, 17
- empty matrix, 22
- end, 18
- enter, 2
- eps, 13
- exit, 1
- eye, 20, 22
- ezplot, 85

- figure, 34
- figure(n), 34
- file, ix
 - ASCII, ix
 - binary, ix
 - dat-, 6
 - function, 57
 - m-, 5, 7
 - mat-, ix, 6
 - txt-, 6
- find, 46
- floating point, ix
- fminbnd, 56
- for, 50
- format compact, 3
- format long, 3
- format loose, 3
- format short, 3
- fplot, 69
- fsolve, 56
- function
 - anonymous, 55
 - handle, 55
- function, 57
- function m-file, 57

- grid, 36
- grid off, 35
- grid on, 35
- gtext, 35

- help, 4
- hold off, 35
- hold on, 35

- i, 13
- if, 47
- Inf, 9, 13
- input, 53
- int, 84
- integer, ix
- integral, 56, 70
- inv, 20
- isempty, 53

- j, 13

- legend, 35
- legend off, 35
- linewidth, 35
- linspace, 21
- load, 6
- logical operation, 43
 - |, 43
 - ~, 43
 - ~=, 43
 - <=, 43
 - <, 43
 - ==, 43
 - >=, 43
 - >, 43
 - &, 43
- loglog, 33
- logspace, 21
- lookfor, 4
- loops, 50

- m-files, 5, 7
- matrix, 21
 - transpose, 20
- matrix manipulation, 19
- matrix operation, 20, 21
- max, 20
- mesh, 39
- meshgrid, 39
- min, 20

- NaN, 9, 13
- nargin, 13
- nargout, 13
- null, 21

- ones, 20

- path, 5
- pi, 13
- plot
 - color, 33
 - complex, 37
 - function plot, 69
 - styles, 33
 - surfaces, 39
 - symbolic expression, 85
 - three dimensional, 38
- plot, 33
- plot3, 38
- pointer
 - to function, 55
- print, 38
- program, ix
- prompt >>, 1

- quad, 85
- quit, 1

- rand, 20

- randn, 20
- realmax, 13
- realmin, 13

- save, 6
- semilogx, 33
- semilogy, 33
- simplify, 84
- sin, 3
- size, 22
- solve, 84
- subplot, 35, 36
- sum, 20
- surf, 39
- svd, 21
- switch, 49
- sym, 83
- syms, 83

- text, 35
- title, 35, 36
- transpose, 17, 21
 - matrix, 20
- tril, 20
- triu, 20

- variable, ix
 - global, 61
 - local, 61
- vector
 - colon notation, 16
 - column, 16
 - row, 15

- while, 51
- who, 5
- whos, 5
- window
 - command, 1

- xlabel, 35

- ylabel, 35

- zeros, 20
- zlabel, 35