UNIVERSITY OF TWENTE.

Faculty of
Engineering Technology

Programming in Engineering

Anthony Thornton, Wouter den Otter, Thomas Weinhart,
Katja Bertoldi, Holger Steeb, and Stefan Luding

edition: 2013

Contents

Preface

This manuscript contains a short introduction into the object-oriented compiler language C++ and the interpreter language MATLAB, including many small and large exercises.

It consists of two parts: The first part covers MATLAB, the second part is an introduction to C++. Both parts are self-contained, but cross-references and similarities are highlighted.

Therefore, students who are interested in self-study, could begin either with the C++ or the MATLAB part. Within both parts, you will find small examples. These examples can be used for self-evaluation. At the end of the C++ and the MATLAB chapter, you will find various larger exercises similar to the exercises of the final assignment.

Acknowledgement

This work includes material from "Programming in Engineering" by Stefan Luding, Holger Steeb & Katja Bertoldi (University of Twente) and "Introduction to Matlab" by Elżbieta Pękalska (Delft University of Technology) with contributions from Hans Zwart (University of Twente). The respective authors have given us permission to re-use their work.

History

Version 1.0 The manuscript was compiled for the 10-day sumer course *Programming in Engineering,* July - 18 July 2008.

Version 1.1 Updated for self-study purposes, 3 September 2008

Version 1.2 Compiled for the 10-day summer course *Programming in Engineering*, 6 July-17 July 2009. This version also used for self-study program beginning 2 September 2009.

Version 1.3 Extended and updated by Stefan Luding and Anthony Thornton, for the 10-day summer course *Programming in Enginerring* 5 July - 18 July 2010. Short exercise added to the C++ part. This version also will be used for self-study program beginning 30 August 2010.

Version 1.4 More information on OOP added and other general improvements to both C++ and MATLAB part. Designed for use in the Summer school 2011 and for the self study for 2011/12 academic year (Anthony Thornton).

Version 1.5 Even more information on OOP to C++ , details of handles added to MATLAB part, and other general improvements to both C++ and MATLAB part. Designed for use in the Summer school 2012 and for the self study for 2012/13 academic year (by Thomas Weinhart & Anthony Thornton).

Version 1.6 Script updated to reflect change of order. Now Matlab is taught before C++ (by Anthony Thornton).

Version 1.7 C++ part updated: templates added, intro to STL added, various other sections reworded and the whole documented was reordered to reflect the way it was taught for summer 2013. This will be used for the September 2013 and 2014 version of the course (by Anthony Thornton).

June 23, 2014

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 *bi*nary digi*t*) 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`, `x` and `y` are variables. They can represent numeric values, like `25.5`, characters, like `'c'` or character strings, like `'Matlab'`. 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. It can be a numeric value, a character or a string.
- 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 1.2e-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 file* is a standardized, readable and editable plain text file.
- 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'`.

Part: A

MATLAB

Chapter 1

Getting started with MATLAB

MATLAB is a tool for mathematical (technical) calculations. Firstly, it can be used as a scientific calculator. Next, it allows you to plot or visualize data in many different ways, perform matrix algebra, work with polynomials or integrate functions. Like in a programmable calculator, you can create, execute and save a sequence of commands in order to make your computational process automatic. It can be used to store or retrieve data. In the end, MATLAB can also be treated as a user-friendly programming language, which gives the possibility to handle mathematical calculations in an easy way. In summary, as a computing/programming environment, MATLAB is especially designed to work with data sets as a whole such as vectors, matrices and images. Therefore, PRTOOLS (`http://prtools.org`), a toolbox for Pattern Recognition, and DIPIMAGE (`http://www.ph.tn.tudelft.nl/DIPlib/dipimage_1.html`), a toolbox for Image Processing, have been developed under MATLAB.

Under Windows, you can start MATLAB by double clicking on the MATLAB icon that should be on the desktop of your computer. On a unix system, type `matlab` at the command line. Running MATLAB creates one or more windows on your screen. The most important is the *Command Window*, which is the place where you interact with MATLAB, i.e. it is used to enter commands and display text results. The string `>>` is the MATLAB prompt (or `EDU>>` for the Student Edition). When the Command Window is active, a cursor appears after the prompt, indicating that MATLAB is waiting for your command. MATLAB responds by printing text in the Command Window or by creating a *Figure Window* for graphics. To exit MATLAB use the command `exit` or `quit`. `Control-C` is a local abort which kills the current execution of a command.

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. However, execution of a command will be possible if the command is typed according to the rules. Table 1.1 shows a list of commands used to solve

indicated mathematical equations (a, b, x and y are numbers). Below you find basic information to help you starting with MATLAB:

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

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

  Note that spaces are **not** important in MATLAB.
- The result of the last performed computation is ascribed to the variable `ans`, which is an example of a MATLAB built-in variable. It can be used in the subsequent command. For instance:

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

  `5.4399e-04` is a computer notation of $5.4399 * 10^{-4}$ (see **Preliminaries**). Note that `ans` is always overwritten by the last command that has no assignment.
- You can also define your own variables. Look how the information is stored in the variables `a` and `b`:

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

  Read **Preliminaries** to better understand the concept of variables. You will learn more on MATLAB variables in section **2.1.2**.
- When the command is followed by a semicolon ';', the output is suppressed. Check the difference between the following expressions:

```
>> 3 + 7.5
>> 3 + 7.5;
```

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

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

  Note that the value of `cos(pi)` is not printed.
- By default, MATLAB displays only 5 digits. The command `format long` increases this number to 15, `format short` reduces it to 5 again. For instance:

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

- 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 in 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, for example, a is written as `a` in MATLAB; `A` will result in an error in this case.
- All text after a percent sign `%` until the end of a line is treated as a comment. Enter e.g. the following:

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

  You will notice that some examples in this text are followed by comments. They are meant for you and you should skip them while typing.
- Previous commands can be fetched back with the ↑-key. The command can also be changed, the ← and →-keys may be used to move around in a line and edit it. In case of a long line, `Ctrl-a` and `Ctrl-e` might be useful; they allow to move the cursor at the beginning or the end of the line, respectively.
- To recall the most recent command starting from e.g. `c`, type `c` at the prompt followed by the ↑-key. Similarly, `cos` followed by the ↑-key will find the last command starting from `cos`.

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!

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`
$4 \cdot 10^3$	`4e3` or `4*10^3`
$i(\sqrt{-1})$	`1i` or `1j`
$3 - 4i$	`3-4*i` or `3-4*j`
e, e^x	`exp(1)`, `exp(x)`
$\ln x$, $\log x$	`log(x)`, `log10(x)`
$\sin x$, $\arctan x$, ...	`sin(x)`, `atan(x)`,...

Table 1.1: Mathematical notation in MATLAB's commands.

1.2 help-facilities

There are several ways for getting help with MATLAB. A good way to start is to type:

```
>> doc
```

The `doc` command opens an HTML document, which contains the MATLAB reference book. This is quite a large document! Useful sections for beginners include Getting Started and MATLAB Functions. You might want to browse them before beginning your first session. Its helpful to keep this browser window open while you are working with MATLAB, so that you can refer to it easily.
There are other ways of getting help with MATLAB that do not make you search through the whole reference book. If you already know the name of a MATLAB function, for example, `quit`, and you want to learn about its use, enter:

```
>> help quit
```

You will see a description of the command `quit`. You can also open the help in a browser window using `doc quit`. During your first experiences with MATLAB you can from time to time take a:

```
>> demo
```

session which will show you various features of MATLAB. Another very useful feature is the lookfor command. It looks for all the commands related to a given topic. Try:

```
>> lookfor 'help'
```

It will list all of the commands we just discussed.
Besides the inherent help of MATLAB, you will find various online courses and references in the internet and, last but not least, textbooks. Here, we just mention a view:
Online sources

- MATLAB home page (from manufacturers)
  `http://www.mathworks.com`
- MATLAB online course (TU Eindhoven)
  `http://www.imc.tue.nl`
- MATLAB online Reference Documentation
  `http://www.math.ufl.edu/help/matlab/ReferenceTOC.html`

Books and others references

- MATLAB An Introduction with Applications, Amos Gilat, 2008.
- MATLAB Programming for Engineers, Stephen Chapman, 2007.
- Essential MATLAB for Engineers and Scientists, Brian Hahn and Dan Valentine, 2007.

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 a few times. After this the MATLAB prompt (`>>`) re-appears. This may take a while, though.

1.4 Path

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

Exercise 1.1.
Type `path` to check which directories are placed on your path. Add you personal directory to the path (assuming that you created your personal directory for working with MATLAB). ■

1.5 Workspace issues

If you work in the Command Window, MATLAB memorizes all commands that you entered and all variables that you created. These commands and variables are said to reside in the MATLAB*workspace*. They might be easily recalled when needed, e.g. to recall previous commands, the ↑-key is used. Variables can be verified with the commands `who`, 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, `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 in MATLABas `clear a` and print `x` on the screen. See what is the result of:

```
>> 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 using (clicking) the **File** menu-bar, and then selecting the **Save Workspace as...** or **Load Workspace...** items respectively. Also MATLAB commands exist which save data to files and which load data from files.

The command `save` allows for saving your workspace variables either into a binary file or an ASCII file (check **Preliminaries** on binary and ASCII files). Binary files automatically get the `'.mat'` extension, which is not true for ASCII files. However, it is recommended to add a `'.txt'` or `.dat` extension.

Exercise 1.2.
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 1.3.
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 Exercises

Exercise 1.4.

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

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

Chapter 2

Basic syntax and variables

2.1 MATLAB as a calculator

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

You have already got some experience with MATLAB and you know that it can be used as a calculator. To do that you can, for example, simply type:

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

The result will be:

```
ans =
   55.8571
```

MATLAB has six basic arithmetic operations, such as: `+`, `-`, `*`, `/` or `\` (right and left divisions) and `^` (power). Note that the two division operators are different (especially for matrices, as we will see later):

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

Basic built-in functions, trigonometric, exponential, etc, are available for a user. Try `help`

`elfun` to get the list of elementary functions.

Exercise 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.

- $2/2*3$
- $8*5\backslash 4$
- $8*(5\backslash 4)$
- $7-5*4\backslash 9$
- $6-2/5+7\text{\^{}}2-1$
- $10/2\backslash 5-3+2*4$
- $3\text{\^{}}2/4$
- $3\text{\^{}}2\text{\^{}}3$
- $2+\texttt{round}\,(6/9+3*2)/2$
- $2+\texttt{floor}\,(6/9+3*2)/2$
- $2+\texttt{ceil}\,(6/9+3*2)/2$
- $\texttt{x}=\texttt{pi}/3, \texttt{x}=\texttt{x}-1, \texttt{x}=\texttt{x}+5, \texttt{x}=\texttt{abs}(\texttt{x})/\texttt{x}$

■

INTERMEZZO

2.1.1 An introduction to floating-point numbers

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 bits and the binary system). The typical size of an integer is 16 bits, so the largest positive integer, which can be stored, is $2^{16} = 65536$. If negative integers are permitted, then 16 bits allow for representing integers between -32768 and 32767. Within this range, operations defined on the set of integers can be performed exactly.

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 could have noticed that in the last exercise since the value of $\sin(\pi)$ was almost zero, and not completely zero. It came from the fact that both the value of π is represented with a finite precision and the sin function is also approximated.

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 relative accuracy might 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` $\approx$ 2.2204e-16. This means that the relative accuracy of individual arithmetic operations is about 15 digits.

END INTERMEZZO

2.1.2 Assignments and variables

Working with complex numbers is easily done with MATLAB.

Exercise 2.2.
Choose two complex numbers, for example `-3 + 2i` and `5 - 7i`. Add, subtract, multiply, and divide these two numbers. ■

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

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

Formally, there is no need to declare (i.e. define the name, size and the type of) a new variable in MATLAB. A variable is simply created 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` type[1]. 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`[1] integer type. 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.

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

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

Variable name begins with a letter, followed by letters, numbers or underscores. MATLAB recognizes only first 31 characters of the name.

Exercise 2.3.
Here are some examples of different types of MATLAB variables. You do not need to understand them all now, since you will learn more about them during the course. Create them manually in MATLAB:

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

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

As you already know, 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 how to built own functions in section **7.1**).

Important: do not use variable names which are defined as function names (for instance `mean` or `error`)[2], because from this point on you cannot access the function. For example try

```
>> mean([4 6])
ans =
     5
>> mean=3
mean =
     3
>> mean([4 6])
??? Index exceeds matrix dimensions.
```

At the second calling mean is a variable not the function mean.

If you are going to use a suspicious variable name, use `help <name>` to find out if the function already exists.

Variable name	Value/meaning
`ans`	the default variable name used for storing the last result
`pi`	$\pi = 3.14159...$
`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`	$i = j = \sqrt{-1}$ (Note: better use $1i$ in case the variable i is used)
`nargin/nargout`	number of function input/output arguments used
`realmin/realmax`	the smallest/largest usable positive real number: $1.7977e{+}308$ / $2.2251e{-}308$

Table 2.1: Built-in variables in MATLAB.

2.2 Exercises

Exercise 2.4.
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 its standard-format. ■

[2]There is always one exception of the rule: the variables `i` and `j` are often used as counter in a loop, while they are also used as $i = \sqrt{-1}$, $j = \sqrt{-1}$.

Chapter 3

Mathematics with vectors and matrices

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

- a 1×1-matrix: a scalar or a single number;
- a matrix existing only of one row or one column: a vector. [Actually this is stored as a two matrix with one dimension being length 1].

Note that MATLAB may behave differently depending on the input, whether it is a number, a vector or a two-dimensional (or more-dimensional) matrix.

3.1 Vectors

Row vectors are lists of numbers separated either by commas or by spaces. They are examples of simple *arrays*. First element has index 1. The number of entries is known as the *length* of the vector (the command `length` exists as well). Their entities are referred to as *elements* or *components*. The entries must be enclosed in `[ ]`:

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

A number of operations can be done on vectors. A vector can be multiplied by a scalar, or added/subtracted to/from another vector with *the same* length, or a number can be added/-subtracted to/from a vector. All these operations are carried out element-by-element. Vectors can also be built from the already existing ones.

```
>> v = [-1 2 7]; w = [2 3 4];
>> z = v + w                    % an element-by-element sum
z =
     1     5    11
>> vv = v + 2                   % add 2 to all elements of vector v
vv =
     1     4     9
>> t = [2*v, -w]
ans =
    -2     4    14    -2    -3    -4
```

Also, a particular value can be changed or displayed:

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

3.1.1 Colon notation

A colon notation is an important shortcut, used when producing row vectors (see Table 3.1 and `help colon`):

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

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

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

3.1.2 Extracting and appending parts of a vector

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

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

MATLAB allocates memory for all variables on the fly. This allows you to increase the size of a vector simply by assigning a value to an element that has not been previously used.

```
>> x = linspace(21,25,5)

x =

    21    22    23    24    25

>> x(7) = -9

x =

    21    22    23    24    25     0    -9
```

3.1.3 Column vectors and transposing

To create *column* vectors, you should separate entries by by new lines or by a semicolon ';':

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

The operations applied to row vectors can be applied to column vectors, as well. You cannot, however, add a column vector to a row vector. To do that, you need an operation called *transposing*, which converts a column vector into a row vector and vice versa:

```
>> u'                            % u is a column vector and u' is a row vector
ans =
    -1     3     5
```

```
>> v = [-1 2 7];                % v is a row vector
>> u + v                        % you cannot add a column vector u to a row
                                % vector v
??? Error using ==> +
Matrix dimensions must agree.
>> u' + v
ans =
    -2     5    12
>> u + v'
ans =
    -2
     5
    12
```

If `z` is a complex vector, then `z'` gives the conjugate transpose of `z`. For instance:

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

3.1.4 Product, divisions and powers of vectors

You can now compute the inner product between two vectors `x` and `y` of the same length, $\mathtt{x}^T\mathtt{y} = \sum_i \mathtt{x_i}\,\mathtt{y_i}$, in a simple way:

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

Another way to compute the inner product is by the use of the dot product, i.e. `.*`, which performs element-wise multiplication. Given two vectors `x` and `y` of the same length, an element-wise multiplication is defined as a vector $[\mathtt{x_1y_1}, \mathtt{x_2y_2}, \ldots, \mathtt{x_ny_n}]$, thus, the corresponding elements of two vectors are multiplied. For instance:

```
>> u .* v                      % this is an element-by-element multiplication
     1
     6
    35
>> sum(u.*v)                   % this is an another way to compute the inner product
ans =
```

```
    42
>> z = [4 3 1];              % z is a row vector
>> sum(u'.*z)                % this is the inner product
ans =
    10
>> u'*z'                     % since z is a row vector, u'*z' is the inner product
ans =
    10
```

You can now easily tabulate the values of a function for a given list of arguments. For instance:

```
>> x = 1:0.5:4;
>> y = sqrt(x) .* cos(x)
y =
    0.5403    0.0866   -0.5885   -1.2667   -1.7147   -1.7520   -1.3073
```

Mathematically, a division of one vector by another is an undefined operation. However, in MATLAB, the operator `./` is introduced to perform an element-by-element division. It is, therefore, defined for vectors of the same size and type:

```
>> x = 2:2:10
x =
     2     4     6     8    10
>> y = 6:10
y =
     6     7     8     9    10
>> x./y
ans =
    0.3333    0.5714    0.7500    0.8889    1.0000
>> z = -1:3
z =
    -1     0     1     2     3
>> x./z                      % division 4/0, resulting in Inf
Warning: Divide by zero.
ans =
   -2.0000       Inf    6.0000    4.0000    3.3333
>> z./z                      % division 0/0, resulting in NaN
Warning: Divide by zero.
ans =
     1   NaN     1     1     1
```

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

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

Exercise 3.1.
Get acquainted with operations on row and column vectors. Perform, for instance:

- Create a vector consisting of the even numbers between 21 and 47.

- Let `x` = [4 5 9 6].
  - Subtract 3 from each element.
  - Add 11 to the odd-index elements.
  - Compute the square root of each element.
  - Raise to the power 3 each element.
- Create a vector `x` with the elements:
  - `2, 4, 6, 8, ...,16`
  - `9, 7, 5, 3, 1, -1, -3, -5`
- Given `x` = [2 1 3 7 9 4 6], explain what the following commands do (note that `x(end)` points to the last element of `x`):
  - `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)`

■

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 $k \le i \le l$, $m \le j \le n$
`A(isnan(A))`	functions like `isnan` return a logical (boolean) array, which can be used for indexing
`v(i:j)'`	'vector-part' $(v_i, v_{i+1}, \ldots, v_j)$ of vector v

Table 3.1: Manipulation of (groups of) matrix elements.

3.2 Matrices

Row and column vectors are special types of matrices. An $n \times k$ matrix is a rectangular array of numbers having n rows and k columns. Defining a matrix in MATLAB is similar to defining a vector. The generalization is straightforward, if you see that a matrix consists of row vectors (or column vectors). Commas or spaces are used to separate elements in a row, and semicolons are used to separate individual rows. For example, the matrix $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ is defined as:

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

Command	Result
`n = rank(A)`	n becomes the rank of matrix A
`x = det(A)`	x becomes the determinant of matrix A
`x = size(A)`	x becomes a row-vector with 2 elements: the number of rows and columns of A
`x = trace(A)`	x becomes the trace (sum of diagonal elements) of matrix A
`x = norm(v)`	x becomes the Euclidean length of vector v
`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 ($k \in Z$; can also be used for A^{-1})
`C = A.^k`	'element-by-element' power of a matrix
`C = A'`	the conjugate transpose of a matrix; A^*
`C = A.'`	the transpose of a matrix; A^T
`C = A ./ B`	'element-by-element' division (A and B are of equal size)
`X = A \ B`	finds the solution in the least squares sense to the system of equations $AX = B$
`X = B / A`	finds the solution of $XA = B$, analogous to the previous command
`C = inv(A)`	C becomes the inverse of A
`C = null(A)`	C is an orthonormal basis for the null space of A obtained from the singular value decomposition
`C = orth(A)`	C is an orthonormal basis for the range of A
`L = eig(A)`	L is a vector containing the (possibly complex) eigenvalues of a square matrix A
`[Q,L] = eig(A)`	produces a diagonal matrix L of eigenvalues and a full matrix Q whose columns are the corresponding eigenvectors of a square matrix A
`S = svd(A)`	S is a vector containing the singular values of a rectangular matrix A
`[U,S,V] = svd(A)`	S is a diagonal matrix with nonnegative diagonal elements in decreasing order; columns of U and V are the accompanying singular vectors of A
`x = linspace(a,b,n)`	generates a vector x of n equally spaced points between a and b
`x = logspace(a,b,n)`	generates a vector x starting at 10^a and ended at 10^b containing n values
`A = eye(n)`	A is an $n \times n$ identity matrix
`A = zeros(n,m)`	A is an $n \times m$ matrix with zeros (default $m = n$)
`A = ones(n,m)`	A is an $n \times m$ matrix with ones (default $m = n$)
`A = diag(v)`	results in a diagonal matrix with the elements $v_1, v_2, \ldots, v_n$ on the diagonal
`v = diag(A)`	results in a vector equivalent to the diagonal of A
`X = tril(A)`	X is lower triangular part of A
`X = triu(A)`	X is upper triangular part of A
`A = rand(n,m)`	A is an $n \times m$ matrix of elements drawn from a uniform distribution on $[0, 1]$
`A = randn(n,m)`	A is an $n \times m$ matrix of elements drawn from a standard normal distribution
`v = max(A)` `v = max(A,[],dim)`	v is a vector of the maximum values of the columns in A v is a vector of the maximum values along dimension dim in A (1=column, 2=rows)
`v = min(A),` `v = min(A,[],dim)`	ditto - with minimum
`v = sum(A),` `v = sum(A,dim)`	ditto - with sum

Table 3.2: Frequently used matrix operations and functions.

Other examples are, for instance:

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

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

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

3.2.1 Special matrices

There is a number of built-in matrices of size specified by the user (see Table 3.2). A few examples are given below:

```
>> E = []                       % an empty matrix of 0-by-0 elements!
E =
     []
>> size(E)
ans =
     0     0
>> I = eye(3);                  % the 3-by-3 identity matrix
I =
     1     0     0
     0     1     0
     0     0     1
>> r = [1 3 -2]; R = diag(r)    % create a diagonal matrix with r on the diagonal
R =
     1     0     0
     0     3     0
```

```
     0     0    -2
>> A = [1 2 3; 4 5 6; 7 8 9];
>> diag(A)                      % extracts the diagonal entries of A
ans =
     1
     5
     9
>> B = ones(3,2)
B =
     1     1
     1     1
     1     1
>> C = zeros (size(B'))         % a matrix of all zeros of the size given by B'
C =
     0     0     0
     0     0     0
>> D = rand(2,3)                % a matrix of random numbers; you will get a
                                 % different one!
D =
    0.0227    0.9101    0.9222
    0.0299    0.0640    0.3309
>> v = linspace(1,2,4)          % a vector is also an example of a matrix
v =
    1.0000    1.3333    1.6667    2.0000
```

3.2.2 Building matrices and extracting parts of matrices

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

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

A part of a matrix can be extracted from a matrix in a similar way as it is done for vectors. Each element in a matrix is indexed by a row and a column to which it belongs. Mathematically, the element from the i-th row and the j-th column of the matrix A is denoted as A_{ij}; MATLAB provides the `A(i,j)` notation.

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

It is easy to 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); the uninitialized
A =                              % elements of A become zeros
     1     2     3
     4     5     8
     7     8     9
     0     0     0
     0     5     0
```

If needed, the other zero elements of the matrix `A` can 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                          % what does the matrix A look like now?
A =
     1     2     3
     4     5     8
     7     8     9
     2     1     2
     4     5     4
```

Different parts of the matrix A can now be extracted:

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

As you have seen in the examples above, it is possible to manipulate (groups of) matrix-elements. The commands are shortly explained in Table 3.1.

The concept of an empty matrix `[]` is also very useful in MATLAB. For instance, a few columns or rows can be removed from a matrix by assigning an empty matrix to it. Try for example:

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

3.2.3 Operations on matrices

Table 3.2 shows some frequently used matrix operations and functions. The important ones are dot operations on matrices, matrix-vector products and matrix-matrix products. In the class of the dot operations, there are dot product, dot division and dot power. Those operations work as for vectors: they address matrices in the element-by-element way, therefore they can be performed on matrices of the same sizes. They also allow for scalar-matrix operations. For the dot product or division, the corresponding elements are either multiplied or divided. A few 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 is B2' needed instead of B2?
B =
     2     4     8
     6     1    13
>> B-2                         % subtract 2 from all elements of B
ans =
     0     2     6
     4    -1    11
>> ans = B./4                  % divide all elements of the matrix B by 4
ans =
    0.5000    1.0000    2.0000
    1.5000    0.2500    3.2500
>> 4/B                         % this is not possible
??? Error using ==> /
Matrix dimensions must agree.

>> 4./B                        % this is possible;
                               % equivalent to: 4.*ones(size(B))./ B
ans =
    2.0000    1.0000    0.5000
    0.6667    4.0000    0.3077
>> C = [1 -1 4; 7 0 -1];
>> B .* C                      % multiply element-by-element
ans =
     2    -4    32
    42     0   -13
>> ans.^3 - 2                  % do for all elements: raise to the power 3 and
                               % subtract 2
ans =
         6    -66    32766
    74086     -2    -2199
>> ans ./ B.^2                 % element-by-element division
ans =
    0.7500   -1.0312   63.9961
  342.9907   -2.0000   -1.0009
>> r = [1 3 -2]; r * B2        % this is a legal operation:
ans =                          % r is a 1-by-3 matrix and B2 is a 3-by-2 matrix
     6    -7                   % note that B2 * r is an illegal operation
```

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

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

>> b * B'                      % this is possible: a row vector multiplied by a
```

```
                                % matrix
ans =
    -8   -10
>> B' *ones(2,1)
ans =
     5
    -1
    10
>> C = [3 1; 1 -3];
>> C * B
ans =
     7    -3    16
   -11    -1   -18
>> C.^3                         % this is 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
```

3.3 Exercises

Exercise 3.2.
Create a vector containing 5 elements such that its components are equally spaced when you take the logarith of it ■

Exercise 3.3.
Perform the following exercises:

- Create a vector `x` with the elements:
  - `1, 1/2, 1/3, 1/4, 1/5`
  - `0, 1/2, 2/3, 3/4, 4/5`

  To do this, devide a vector `y` by a vector `z`.
- Create a vector `x` with the elements: $\mathtt{x_n} = \frac{(-1)^{\mathtt{n}}}{2\mathtt{n}-1}$ for $n = 1, 2, 3, ...$ Find the sum of the 100-element vector.
- Given a vector `t`, write down the MATLAB expressions that will compute:
  - $\mathtt{ln}\,(2 + \mathtt{t} + \mathtt{t}^2)$
  - $\mathtt{cos}\,(\mathtt{t})^2 - \mathtt{sin}\,(\mathtt{t})^2$
  - $\mathtt{e}^{\mathtt{t}}\,(1 + \mathtt{cos}\,(3\mathtt{t}))$
  - $\mathtt{tan}^{-1}(\mathtt{t})$

  Test them for $\mathtt{t} = 1 : 0.2 : 2$.

■

Exercise 3.4.
Use the knowledge on computing the inner product to find:

1. the Euclidean length of the vector $\mathtt{x} = [2\ 1\ 3\ 7\ 9\ 4\ 6]$, which is defined as $||\mathtt{x}|| = \sqrt{(\Sigma \mathtt{x}_\mathtt{i}^2)}$.
2. the angle between two column vectors, which is defined as $\cos\alpha = \frac{\mathtt{x}^\mathtt{T}\mathtt{y}}{||\mathtt{x}||\,||\mathtt{y}||}$. Note that you can also use the MATLAB command norm(v), which gives you the Euclidean length of a vector v. The MATLAB command acosd(x) can be used to find the inverse cosine of x expressed in degrees. Compute the angle between two vectors:
   - $\mathtt{x} = [3\ 2\ 1]$ and $\mathtt{y} = [1\ 2\ 3]$
   - $\mathtt{x} = 1:5$ and $\mathtt{y} = 6:10$

■

Exercise 3.5.
Clear all variables (use the command `clear`). Define the matrix `A = [1:4; 5:8; 1 1 1 1]`. Predict and check the result of the following operations:

- $\mathtt{x} = \mathtt{A}(:, 3)$
- $\mathtt{B} = \mathtt{A}(1:3, 2:2)$
- $\mathtt{A}(1,1) = 9 + \mathtt{A}(2,3)$
- $\mathtt{A}(2:3, 1:3) = [0\ 0\ 0; 0\ 0\ 0]$
- $\mathtt{A}(2:3, 1:2) = [1\ 1; 3\ 3]$
- $\mathtt{y} = \mathtt{A}(3:3, 1:4)$
- $\mathtt{A} = [\mathtt{A}; 2\ 1\ 7\ 7; 7\ 7\ 4\ 5]$
- $\mathtt{C} = \mathtt{A}([1,3], 2)$
- $\mathtt{D} = \mathtt{A}([2,3,5], [1,3,4])$
- $\mathtt{D}(2,:) = [\,]$

■

Exercise 3.6.
Define the matrices `T = [ 3 4; 1 8; -4 3] and A = [diag(-1:2:3) T; -4 4 1 2 1]`. Perform the following operations on the matrix A:

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

■

Exercise 3.7.
Let $\mathtt{A} = [2\ 4\ 1;\ 6\ 7\ 2;\ 3\ 5\ 9]$. Provide the commands which:

- assign the first row of `A` to a vector `x`;
- assign the last 2 rows of `A` to a vector `y`;
- add up the columns of `A`;

- add up the rows of A;

■

Exercise 3.8.
Let A = [2 7 9 7; 3 1 5 6; 8 1 2 5]. Explain the results or perform the following commands:

- A′
- A(1, :)′
- A(:, [14])
- A([23], [31])
- reshape (A, 2, 6)
- A(:)
- flipud (A)
- fliplr (A)
- [A A(end, :)]
- [A; A(1 : 2, :)]
- sum (A)
- sum (A′)
- mean (A)
- mean (A′)
- sum (A, 2)
- mean (A, 2)
- min (A)
- max (A′)
- min (A(:, 4))
- [min(A)′max(A)′]
- max (min(A))
- [[A; sum (A)] [sum (A, 2); sum (A(:))]]
- assign the even-numbered columns of A to an array B
- assign the odd-numbered rows to an array C
- convert A into a 4-by-3 array
- compute the reciprocal of each element of A
- compute the square-root of each element of A
- remove the second column of A
- add a row of all 1's at the beginning and at the end
- swap the 2nd row and the last row

■

Exercise 3.9.
Given the vectors x = [1 3 7], y = [2 4 2] and the 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:

- x + y
- x + A
- x′ + y
- A − [x′ y′]
- [x; y] + A
- [x; y′]
- [x; y]
- A − 3
- A + B
- B′ + A
- B ∗ A
- A. ∗ B
- A′. ∗ B
- 2 ∗ B
- 2. ∗ B
- B./x′
- B./[x′ x′]
- 2/A
- ones(1, 3) ∗ A
- ones(1, 3) ∗ B

■

Exercise 3.10.
Perform all operations from Table 3.2, using some matrices A and B, vector v and scalars k, a, b, n, and m. ■

Exercise 3.11.
Let A be a square 6-by-6 matrix.

1. 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.
2. Create a tridiagonal matrix T, whose three diagonal are taken from the matrix A. *Hint:* you may use the commands triu and tril.

■

Exercise 3.12.
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 what is the difference between the operators \, / and the command `inv`. Having found `x`, check whether $Ax - b$ is close to a zero vector. ■

Exercise 3.13.
Let `A = ones(6) + eye(6)`. Normalize the columns of the matrix `A` so that all columns of the resulting matrix, say `B`, have the Euclidean norm (length) equal to 1. Next, find the angles between consecutive columns of the matrix `B`. ■

Exercise 3.14.
Find two 2×2 matrices A and B for which $\texttt{A.}*\texttt{B} \neq \texttt{A}*\texttt{B}$ holds. Make use of the following operations: `/`, \, or the command `inv`. ■

Chapter 4

Scripts

4.1 Script m-files

MATLAB commands can be entered at the MATLAB prompt. When a problem is more complicated this becomes inefficient. A solution is to use script m-files.

M-files are useful when the number of commands increases or when you want to change values of some variables and re-evaluate them quickly. Formally, a script is an external file that contains a sequence of MATLAB commands (statements). However, it is not a function, since there are no input/output parameters and the script variables remain in the workspace. So, when you run a script, the commands in it are executed as if they have been entered through the keyboard.

The use of m-files is illustrated in the following paragraph.

How to create and run a script

- Type `edit sinplot.m` in the workspace.
- Enter the lines listed below

```
%creates a plot of the sine function
 x = 0:0.2:6;
 y = sin(x);
 plot(x,y);
 title('Plot of y = sin(x)');
```

  (Starting a script with comment lines will generate response on help.)

- Save the files as sinplot.m

- Run the script by typing in the MATLAB prompt:

```
sinplot
```

Alternatively to run the script you can press F5 or you can click on the run icon.

You might have problems with this because your path is not set correctly. The path is a collection of directories (folders) where MATLAB goes to look for programs to be run. If you saved that file first in, say, `Myfolder`, you should tell this to MATLAB by typing:

```
>> cd Myfolder
```

or opening the path browser utility from your MATLAB window.

Note that 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
```

These generic names, x and y, may easily be used in further computations and this can cause side effects. Side effects occur in general when a set of commands change variables other than the input arguments. Since scripts create and change variables in the workspace (without warning), a bug, hard to track down, may easily appear. So, it is important to remember that the commands within a script have access to all variables in the workspace and all variables created in this script become a part of the workspace. Therefore, it is better to use function m-files to solve a specific problem.

4.2 Exercises

Exercise 4.1.
Write a script that for the given a, b and c returns the roots of the cubic equation: $ax^2+bx+c=0$. (Hint: the solution to the equation is given by $x = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$) ■

Chapter 5

Visualization

MATLAB includes very powerful features for easy figure creation, plotting and image display. MATLAB can be used to visualize the results of an experiment. We will start by introducing the most basic features here.

5.1 2D plots

The most important commands for basic 2-dimensional plotting are illustrated in the following example:

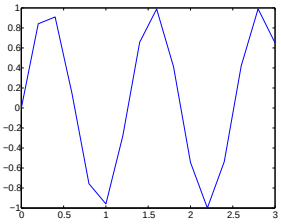
Example 5.1:
Plotting the sinus function

```
x=0:0.1:10;  % x ranges from 0 to 10 in steps of 0.1
y=sin(x);    % y contains the sin of each value of x
figure       % create an empty figure window
plot(y)      % plots all of the y values (the values on the x-axis are
             % indexes from 1 to 101)
figure       % create an empty figure window
plot(x,y)    % plots y against x (replacing the old plot on the same
             % figure, now showing the units of x on the x-axis)
close        % closes the figure window
```

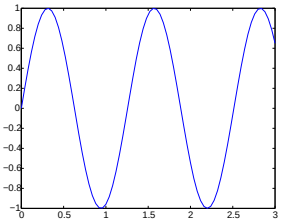
Note that `x` and `y` have to be both either row or column vectors of the same length (i.e. having the same number of elements).

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

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



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



The `plot` command has many additional possible arguments, as can be seen by typing `help plot`. There are also many other plotting commands for 2-dimensional plotting (`help graph2d`) and also for 3-D plotting (`help graph3d`). You can close a figure window by clicking on the upper right X box, using the `close` command on the figure's file menu, or by typing `close` at the command prompt. Use `close all` to close all figures and start a new task (or use `close 1` to close Figure no.1 etc).

If you do not create an extra figure using `figure`, the previous function is removed as soon as the next is displayed.

The color and point marker can be changed on a plot by adding a third parameter (in single quotes) to the plot command. For example, to plot the a function as a red, dotted line, the m-file should be changed to:

```
x = 0:0.1:100;
y = 3*x;
plot(x,y,'r:')
```

The third input consists of one to three characters which specify a color and/or a point marker type. The list of colors and styles is reported in Table 5.1.

The commands `loglog`, `semilogx` and `semilogy` are similar to `plot`, except that they use either one or two logarithmic axes.

5.2 Several functions in one figure

You can plot more than one function on the same figure. To plot a sine wave and cosine wave on the same set of axes, using a different color and point marker for each, the following m-file

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

Table 5.1: Plot colors and styles.

could be used:

```
x = 1:0.1:2*pi;
y = sin(x);
z = cos(x);
plot(x,y,'r', x,z,'gx')
```

By adding more sets of parameters to plot, you can plot as many different functions on the same figure as you want. When plotting many things on the same graph it is useful to differentiate the different functions based on color and point marker. This same effect can also be achieved using the `hold on` and `hold off` commands. The same plot shown above could be generated using the following m-file:

```
x = linspace(0,2*pi,50);
y = sin(x);
z = cos(x);
hold on
plot(x,y,'r')
plot(x,z,'gx')
hold off
```

Always remember that if you use the `hold on` command, all plots from then on will be generated on one set of axes, without erasing the previous plot, until the hold off command is issued.

It is also possible to produce a few subplots in one figure window. With the command `subplot`, the window can be horizontally and vertically divided into $p \times r$ subfigures, which are counted from 1 to pr, row-wise, starting from the top left. The commands: `plot`, `title`, `grid` etc work only in the current subfigure. So, if you want to change something in other subfigure, use the command `subplot` to switch there.

```
x = 1:.1:4;
y1 = sin(3*x);
y2 = cos(5*x);
y3 = sin(3*x).*cos(5*x);
figure
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)')
```

5.3 Adding text

Another thing that may be important for your plots is labeling. You can give your plot a title (with the `title` command), x-axis label (with the `xlabel` command), y-axis label (with the `ylabel` command), and put text on the actual plot. All of the above commands are issued after the actual plot command has been issued.

Furthermore, text can be put on the plot itself in one of two ways: the `text` command and the `gtext` command. The first command involves knowing the coordinates of where you want the text string. The command is `text(xcor,ycor,'textstring')`. To use the other command, you do not need to know the exact coordinates. The command is `gtext('textstring')`, and then you just move the cross-hair to the desired location with the mouse, and click on the position you want the text placed.

To add a title, grid and to label the axes, one uses:

```
x=0:0.1:3;
y=sin(x);
z=cos(x);
hold on;
plot(x,y);
plot(x,z);
title('Sine and Cosine')
xlabel('x')
ylabel('sin(x) and cos(x)')
gtext('unnecessary labeling')
legend('sin','cos')
```

The text "unnecessary labeling" was placed right above the position, I clicked on.

Table 5.2 shows a few possibilities of the **`plot`** command, **`help plot`** shows them all.

5.4 Editing plots

MATLAB formats a graph to provide readability, setting the scale of axes, including tick marks on the axes, and using color and line style to distinguish the plots in the graph. However, if you are creating presentation graphics, you may want to change this default formatting or add descriptive labels, titles, legends and other annotations to help explain your data. MATLAB supports two ways to edit the plots you create:

- Using the Property Editor to select and edit objects interactively.

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
`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 (move it with your mouse) to name your functions
`legend off`	deletes the legend box
`clf`	clear the current figure
`figure(n)`	creates a figure #`n` or activates existing figure #`n`.
`subplot`	creates a subplot in the current figure

Table 5.2: Useful commands to make plots.

- Using MATLAB functions at the command-line or in an M-file (look at "Line Properties" in the help for the list of properties).

```
figure
hold on
x = 1:.1:4;
h1=plot(x,cos(x))
h=plot(x,sin(x))
set(h1,'LineWidth',2,'LineStyle','.','Color','g')
set(h,'LineWidth',2,'LineStyle','^','Color','r')
hold off
```

In this above example, `h` and `h1` are handles (basically pointers to objects, see c++ part). The function `get(h)` returns all properties of the handle `h`, this can be very useful for touching up plots.

5.5 Changing the axis

An important important way to customize your plots to meet your needs is with the axis command. The axis command changes the axis of the plot shown, so only the part of the axis that is desirable is displayed. The axis command is used by entering the following command right after the plot command (or any command that has a plot as an output):

```
axis([xmin, xmax, ymin, ymax])
```

5.6 Exporting graph

To copy the plot into Word you can select directly on the figure

```
Edit → Copy Figure
```

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

With handle `h`, use `saveas(h,'name.fig')` and `load('name.fig')` to store figures in editable MATLAB format.

5.7 Plotting surfaces

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

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

The domain $[-1, 1] \times [0, 2]$ is now sampled with 0.5 in both directions and it is described by points $[X(i, j), Y(i, j)]$. To plot a smooth surface, the chosen domain should be sampled in a

dense way. To plot a surface, the command `mesh` or `surf` can be used:

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

Use `colormap` to define different colors for plotting.

5.7.1 Contour plots

A filled (2D) contour plot displays isolines calculated from matrix z and fills the areas between the isolines using constant colors. The color of the filled areas depends on the current figure's colormap.

Example 5.2:
To view a contour plot of the function $z = x \exp(-x^2 - y^2)$ *over the range* $-2 \leq x \leq 2$ *and* $-2 \leq y \leq 3$

```
[X,Y] = meshgrid(-2:.05:2,-2:.05:3);
Z = X.*exp(-X.^2-Y.^2);
figure
[C,h]=contourf(X,Y,Z,25);
colormap hsv
figure
[C,h] = contour(X,Y,Z,15);
close
```

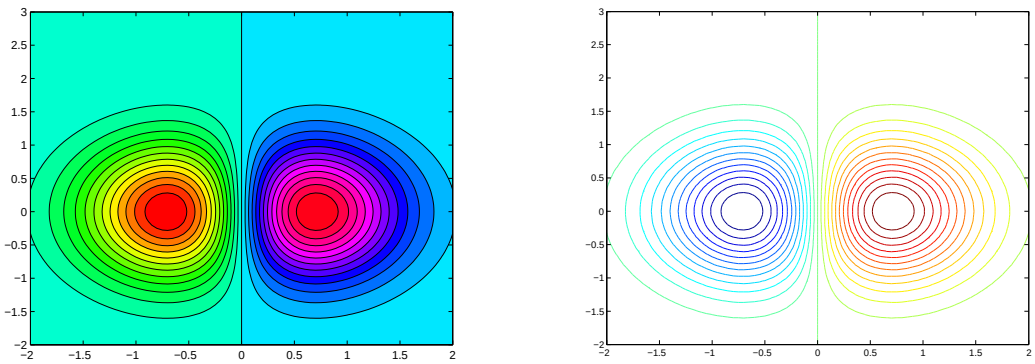


Figure 5.1: Output of the `contourf` and `contour` example

Example 5.3:
To locate e.g. the minimum value of the function the function $z = x \exp(-x^2 - y^2)$ *on the grid, you can proceed as follows:*

```
[X,Y] = meshgrid(-2:.05:2,-2:.05:3);
Z = X.*exp(-X.^2-Y.^2);
```

```
[mm,I] = min(Z);        % a row vector of the min. elements from each column
                        % I is a vector of corresponding
[~, j] = min (mm);      % j is the index of the minimum value; the tilde sign
                        % suppresses the output of the minimum value itself
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
```

5.8 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
title('Curve r(t) = [t sin(t), t cos(t), t]');
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
grid
```

5.9 Animations

A sequence of graphs can be put in motion in MATLAB (the version should be at least 5.0), i.e. you can make a movie using MATLAB graphics tools. The MATLAB command `addframe(avifile,frame)` appends the data in frame to the AVI file identified by the variable `avifile`, which was created by a previous call to the file `output.avi`.

Example 5.4:
Create a movie of an evolving sinus curve in the interval $4\,\pi$.

```
clear all
fig=figure;
mov = avifile('output.avi')
x = 0:0.1:4*pi;
xx(1) = 0;
yy(1) = 0;
for k=1:length(x)
    xx(k) = x(k);
    yy(k) = sin(x(k));
    h=plot(xx,yy);
    axis([0 4*pi -1.5 1.5])
    set(h,'color','m','linewidth',2,'marker','o')
    F=getframe(fig);
```

```
        mov = addframe(mov,F);
    end
    mov = close(mov);
```

Here, a `for`-loop construction has been used to create the movie frames. You will learn more on loops in the next classes, see section 6.3.

Via the command `getframe` each frame is stored in the column of the matrix *mov*. Note that to create a movie requires quite some memory.

5.10 Exercises

Exercise 5.1.
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 5.2.
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 5.3.
Plot a circle with the radius $r = 2$, knowing that the parametric equation of a circle is $[x(t), y(t)] = [r\,cos(t), r\,sin(t)]$ for $t = [0, 2\pi]$. ■

Exercise 5.4.
Plot an ellipse with semiaxes $a = 4$ and $b = 2$. ■

Exercise 5.5.
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 5.6.
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? ■

Exercise 5.7.
Produce a nice graph which demonstrates as clearly as possible the behavior of the function $f(x, y) = \frac{x\,y^2}{x^2+y^4}$ near the point $(0, 0)$. Note that the sampling around this points should be dense enough. ■

Exercise 5.8.
Plot a sphere, which is parametrically defined as $[x(t, s), y(t, s), z(t, s)] = [\cos(t) * \cos(s),\ \cos(t) * \sin(s),\ \sin(t)]$ for $t, s = [0, 2\,\pi]$ (use `surf`). Make first equal axes, then remove them. Use `shading interp` to remove black lines (use `shading faceted` to restore the original picture). ■

Exercise 5.9.
Plot the parametric function of r and θ: $[x(r,\theta), y(r,\theta), z(r,\theta)] = [r\ \cos(\theta),\ r\ \sin(\theta),\ \sin(6\ \cos(r) - n\theta)]$ for $\theta = [0, 2\,\pi]$ and $r = [0, 4]$. Choose n to be constant. Observe, how the graph changes depending on different n. ■

Exercise 5.10.
Plot the surface $f(x, y) = x\,y\,e^{-x^2-y^2}$ over the domain $[-2, 2] \times [-2, 2]$. Find the values and the locations of the minima and maxima of this function. ■

Exercise 5.11.
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. Experiment with this option. ■

Exercise 5.12.
Write a script that makes a movie consisting of 5 frames of the surface $f(x, y) = \sin(nx)\ \sin(ky)$ over the domain $[0, 2\pi] \times [0, 2\pi]$ and $n = 1 : 5$. Add a title, description of axes and shading. ■

Exercise 5.13.
Plot an equilateral triangle with two vertices $[a\ a]$ and $[b\ a]$. Find the third vertex. Use `fill` to paint the triangle. ■

Chapter 6

Control flow

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

6.1 Logical and relational operators

To use control flow commands, it is necessary to perform operations that result in logical values: TRUE or FALSE. In MATLAB the result of a logical operation is `1` if it is true and `0` if it is false. Table 6.1 shows the relational and logical operations. 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).

Important: The logical `&` and `|` have equal precedence in MATLAB, which means that those operators associate from *right to left* (unlike regular arithmetic). A common situation is:

```
>> b = 10;
>> 1 | b > 0  & 0          % evaluated from right to left
ans =
     1
>> (1 | b > 0) & 0         % this indicates the same as above
ans =
     0
>> 1 | (b > 0  & 0)
ans =
     1
```

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

Table 6.1: Relational and logical operations.

This shows that you should *always* 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.

- `y(rem(x,2))` vs. `y(logical(rem(x,2)))` `% odd elements`
- `y(~rem(x,2))` vs. `y(~logical(rem(x,2)))` `% even elements`

These examples show that a *numerical* 0 can not be used as an array index, i.e. the first expression will result in a MATLAB error if any of the remainders computed with the `rem` function equals zero. Converting these zeros into *logical* 0's solves the problem. Note that both expressions to select the even elements do work as the logical complement `~` implicitly converts the numerical results of `rem` into logical values.

Exercise 6.1.
Exercise with logical and relational operators:

1. Predict and check the result of each of the operations of Table 6.1 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 (`randn(1,7)`) and perform all logical operations, including `xor`, `any` and `all`.

■

Exercise 6.2.
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`
   - `y < x`
   - `x == y`
   - `x <= y`
   - `y >= x`
   - `x | y`
   - `x & (~y)`
   - `(x > y) | (y < x)`
   - `(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)`
   - `x(x > 5)`
   - `y(x <= 4)`
   - `x((x < 2) | (x >= 8))`
   - `y((x < 2) | (x >= 8))`
   - `x(y < 0)`

■

Exercise 6.3.
Let `x = [3 16 9 12 -1 0 -12 9 6 1]`. Provide the command(s) that will:

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

■

Exercise 6.4.
Execute the following commands and try to understand how `z` is defined.

```
>> hold on
>> 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); 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)          % subtracts from b a 0-1 vector, taking 1 for
>>                       %                          a <= 7 and 0 otherwise
>> (a >= 2) & (a < 4)    % returns ones at positions where 2 <= a < 4
>> ~(b > 4)              % returns ones at positions where b <= 4
>> (a == b) | b == 3     % returns ones at positions where a is equal to b or
>>                       %                                  b is equal to 3
>> any(a > 5)            % returns 1 when ANY of the a elements are larger than 5
>> any(b < 5 & a > 8)    % returns 1 when 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
```

6.1.1 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 return 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
```

An 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*');
```

`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(:)` in which 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 6.5.
Let `A` = `ceil`$(5 *$ `randn`$(6, 6))$. Perform the following:

- find the indices and list all elements of `A` which are smaller than `-3`;
- find the indices and list all elements of `A` which are smaller than `5` and larger than `-1`;
- remove those columns of `A` which contain at least one `0` element.

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

6.2 Conditional code execution

6.2.1 Using `if ... elseif ... else ... end`

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

- `if ... end`

Syntax

```
if logical_expression
   statement1
   statement2
   ....
end
```

Example

```
if (a > 0)
  b = a;
  disp ('a is positive')
end
```

- `if ... else ... end`

Syntax

```
if logical_expression
   block of statements
   evaluated if TRUE
else
   block of statements
   evaluated if FALSE
end
```

Example

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

- `if ... elseif ... else ... end`

Syntax

```
if logical_expression1
   block of statements evaluated
   if logical_expression1 is TRUE
elseif logical_expression2
   block of statements evaluated
   if logical_expression2 is TRUE
else
   block of statements evaluated
   if no other expression is TRUE
end
```

Example

```
if (height > 190)
  disp ('very tall');
elseif (height > 170)
  disp ('tall');
elseif (height < 150)
  disp ('small');
else
  disp ('average');
end
```

Exercise 6.6.
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 those fragments are not always the proper MATLAB expressions):

1.
```
if n > 1                a) n =  7   m = ?
  m = n + 2             b) n =  0   m = ?
else                    c) n = -7   m = ?
  m = n - 2
end
```

2.
```
if s <= 1               a) s =   1  t = ?
  t = 2z                b) s =   7  t = ?
elseif s < 10           c) s =  57  t = ?
  t = 9 - z             d) s = 300  t = ?
elseif s < 100
  t = sqrt(s)
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 6.7.
Create a script that asks for a number N and computes the drag coefficient C, depending on the Reynold's number N (make use of the `if ... elseif ...` construction). The command `input` might be useful here (use `help` if needed).

$$C = \begin{cases} 0, & N \leq 0 \\ 24/N, & N \in (0,\, 0.1] \\ 24/N\,(1 + 0.14\,N^{0.7}), & N \in (0.1,\, 1e3] \\ 0.43, & N \in (1e3,\, 5e5] \\ 0.19 - 8e4/N, & N > 5e5 \end{cases}$$

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

Exercise 6.8.
Write a script that asks for an integer and checks whether it can be divided by 2 or 3. Consider all possibilities, such as: divisible by both 2 and 3, divisible by 2 and not by 3 etc (use the command `rem`). ■

6.2.2 Using `switch`

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

Syntax

```
switch expression
  case choice1
     block of commands1
  case {choice2a, choice2b,...}
     block of commands2
      ...
  otherwise
     block of commands
end
```

Example

```
method = 2;
switch method
  case {1,2}
     disp('Method is linear.');
  case 3
     disp('Method is cubic.');
  case 4
     disp('Method is nearest.');
  otherwise
     disp('Unknown method.');
  end
```

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.

Exercise 6.9.
Assume that the months are represented by numbers from 1 to 12. Write a script that asks you to provide a month and returns the number of days in that particular month. Alternatively, write a script that asks you to provide a month name (e.g. 'June') instead of a number. Use the `switch`-construction. ■

6.3 Loops

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

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

Syntax

```
for index = first:step:last
  block of statements
end
```

Example

```
sumx = 0;
for i=1:length(x)
   sumx = sumx + x(i);
end
```

You can specify any `step`, including a negative value. A non-integer loop index i is also permitted. The `index` of the `for`-loop can also be a vector. See some examples of possible variations:

Example 1

```
for i=1:2:n
  ...
end
```

Example 2

```
for i=n:-1:3
  ....
end
```

Example 3

```
for x=0:0.1:1
  disp(x^2);
end
```

Example 4

```
for x=[25 9 81]
  disp(sqrt(x));
end
```

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

Syntax

```
while expression
  statement1
  statement2
  statement3
  ...
end
```

Example

```
N = 100;
iter = 1;
msum = 0;
while iter <= N
  msum = msum + iter;
  iter = iter + 1;
end;
```

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

```
figure
hold on
x = linspace(0,2*pi);
```

```
for n=1:9
  subplot(3,3,n);
  y = cos(n*x);
  plot(x,y);
  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 $\mathtt{A}_{ij} = \mathtt{x}_i\,\mathtt{y}_j$. Enter the following commands to a script:

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

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

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

Exercise 6.10.
Determine the sum of the first 50 squared numbers with a control loop. ■

Exercise 6.11.
Write a script to find the largest value n such that the sum: $\sqrt{1^3} + \sqrt{2^3} + \ldots + \sqrt{n^3}$ is less than 1000. ■

Exercise 6.12.
Use a loop construction to carry out the computations. Write short scripts.

1. Given the vector `x` = [1 8 3 9 0 1], create a short set of commands that will:
   - add up the values of the elements (check with `sum`);
   - computes the running sum (for element j, the running sum is the sum of the elements from 1 to j; check with `cumsum`);
   - computes the sine of the given x-values (should be a vector).
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:
   - $\mathtt{a}_{ij} = \mathtt{y}_i/\mathtt{x}_j$;
   - $\mathtt{b}_i = \mathtt{x}_i\,\mathtt{y}_i$ and add the elements in `btot`;
   - $\mathtt{c}_{ij} = \mathtt{x}_i/(2 + \mathtt{x}_i + \mathtt{y}_j)$;
   - $\mathtt{d}_{ij} = 1/\max(\mathtt{x}_i, \mathtt{y}_j)$.

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 6.13.
Write a script that asks for a temperature in degrees Celsius `tc` and computes the equivalent temperature in degrees Fahrenheit `tf` (use the formula $\texttt{tf} = 9/5 * \texttt{tc} + 32$). The script should keep running until no number is provided to convert. The functions `input` and `isempty` (use `help` to learn more) should be useful here. ■

6.4 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 6.14.
Consider the following example:

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

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

Exercise 6.15.
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`? ■

6.5 Exercises

Exercise 6.16.
The circumference of a unit circle (with radius 1) equals 2π. Knowing this, we can approximate π or 2π as follows. Section the circle into triangles, figure 6.1. Starting with 4 triangles the length of s_n equals $\sqrt{2}$. A first approximation for the circumference of the circle is $4\sqrt{2} \approx 5.66$.

Next the number of triangles is doubled. Length a is half of the previously determined length s_n. Then subsequently lengths b and c can be computed and also the length of the new section s_{n+1} is known from which a new (and improved) approximation for the circumference of the circle can be obtained.

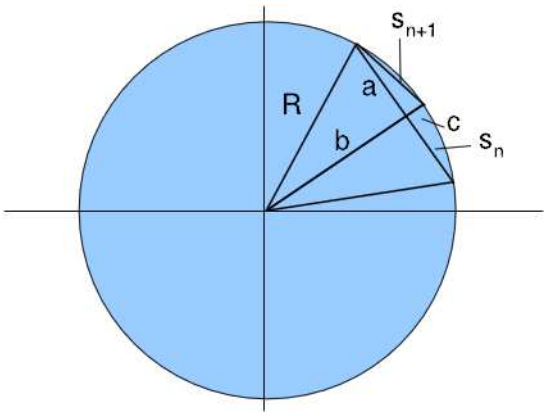


Figure 6.1: Sectioning a circle into triangle to approximate π. From the BEAM-B handout.

Write a script to evaluate the approximation for the circumference of the circle while the number of triangles is doubled at least 20 times.

■

Chapter 7

Functions

M-files have been introduced in chapter 4 to collect MATLAB commands in *scripts*. In this chapter M-files are used to define MATLAB*functions*.

7.1 Function m-file

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

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

`outputArgs` are enclosed in `[ ]`:

- a comma-separated list of variable names;
- `[ ]` is optional when only one argument is present;
- functions without `outputArgs` are legal[1].

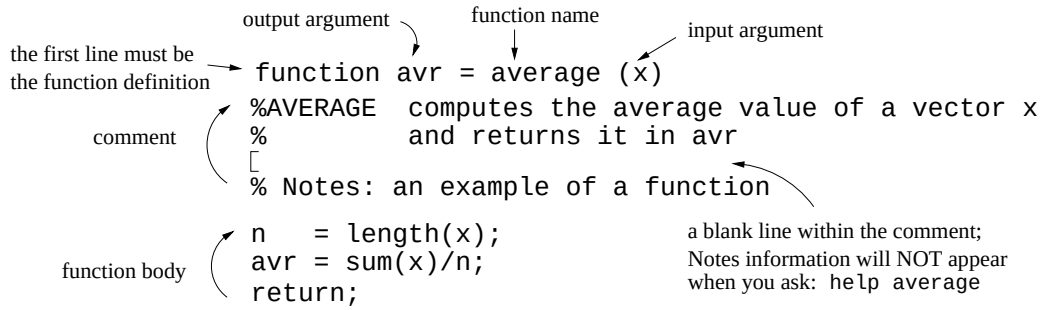
`inputArgs` are enclosed in `( )`:

- a comma-separated list of variable names;
- functions without `inputArgs` are legal.

MATLAB provides a structure for creating your own functions. The first line of the file should be a definition of a new function (also called a header). After that, a continuous sequence of comment lines should appear. Their goal is to explain what the function does, especially when this is not trivial. Not only a general description, but also the expected input parameters,

[1] In other programming languages, functions without output arguments are called *procedures*

returned output parameters and synopsis should appear there. The comment lines (counted up to the first non-comment line) are important since they are displayed in response to the `help` command. Finally, the remainder of the function is called *the body*. Function m-files terminate execution and return when they reached the end of the file or, alternatively, when the command `return` is encountered. As an example, the function `average` is defined as follows:



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

Exercise 7.1.
Create the function `average` and store it on disk as `average.m`. Remember about the comment lines. Check its usability by calling `help average`. Run `average.m` by typing `avr1 = average(1:10);` ■

Here is an another example of a function:

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

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 **8.3**).

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 A
[m,n] = size(A);
if (m ~= n)
  disp ('Error. Matrix should be square.');
  return;
else
  d = det(A);                    % standard Matlab function
end
```

```
return;
```

The use of the `return` command in combination with a check of the *number* of input arguments is given in the function `checkarg`, which is presented in section **7.1.2**.

When controlling the proper use of parameters, the function `error` may prove 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
```

Exercise 7.2.
Change some scripts that you created into functions, e.g. create the function `drag`, computing the drag coefficient (see section **6.2**), or `solve_cos` (see section **6.4**) or `cubic_roots` (see section **4**). ■

Exercise 7.3.
Write the function `[elems, mns] = nonzero(A)` that takes as the input argument a matrix `A` and returns all nonzero elements of `A` in the column vector `elems`. The output parameter `mns` holds values of the means of all columns of `A`. ■

Exercise 7.4.
Write the function `[meanrow, meancol] = allmeans(A)` that takes as the input argument a matrix `A` and returns the means of all its rows and columns respectively. ■

Exercise 7.5.
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 7.6.
The area of a triangle with sides of length a, b, and c is given by: $ar = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$, where $s = (a+b+c)/2$. Write a function that accepts a, b and c as inputs and returns the value ar as output. Note that the sides should be non-negative and should fulfill the triangle inequality. Make use of the `error` command. ■

Exercise 7.7.
Create the function `randint` 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 = randint(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. ■

7.1.1 Subfunctions

A function m-file may contain more than a single function. The function appearing first in the m-file shares the file name, as mentioned before. Other functions are *subfunctions* and can be called from the primary function only. So, they cannot be accessed from outside the file, in which they are defined (neither from the Command Window nor via other m-files). 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 and too complicated. For example, `average` is now a subfunction within the file `stat.m`:

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

Exercise 7.8.
Modify the function `stat` presented above by using the subfunction `average` for the computation of variable `sd` as well. ■

7.1.2 Special function variables

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

```
function [out1,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
```

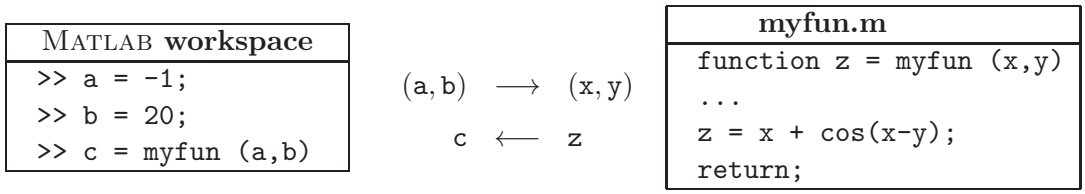
Exercise 7.9.
Construct the function `checkarg`, call it with different number of input and output arguments and try to understand its behavior. For example:

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

■

7.1.3 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 function and from the workspace. To understand it better analyze the following diagram:



In the MATLAB workspace, variables `a`, `b` and `c` are available. Variables `x`, `y` and `z` are visible only in the function `myfun`. However, if several functions and/or the workspace, **all** declare a particular variable as `global`, then they all share this variable (see `help global`). Any assignment to that 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.

OPTIONAL

7.1.4 Indirect function evaluation

Using indirect function evaluation makes programming even more general, since functions can become input arguments. The crucial MATLAB command here is `feval`, an abbreviation of

function **eval**uation. The `feval` command allows execution of a function specified by a string. The general definition is as follows:

```
[y1,..,yn] = feval (F,x1,...,xn),
```

where `F` is a name of a function defined in MATLAB, `x1,...,xn` are input arguments and `y1,...,yn` are possible output parameters. Consider an example:

```
>> x = pi; y = cos(x);
>> z = feval('cos',x);
```

The last command is also equivalent to the following two expressions:

```
>> F = 'cos';
>> z = feval(F,x)
```

Indirect function evaluation is a nice tool to build a program with a function given as an argument.

Exercise 7.10.
Create the function `funplot` and try to understand how it works:

```
function funplot (F, xstart, xend, col);
%FUNPLOT  makes a plot of the function F at the interval [xstart, xend].
%         The plot should be made in one of the standard Matlab colors, so
%         'col' is one of the following value: 'b','k','m','g','w','y' or 'r'.
% default values:
%       [xstart,xend] = [0,10]
%       col = 'b'

% Note: illustrates the use of feval command

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);
  y = feval(F,x);
  plot (x,y,col);
  description = ['Plot of ', F];
  title (description);
  return;
```

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
```

■

END OPTIONAL

7.2 Scripts vs. functions

The most important difference between a script and a function is that *all* script's parameters and variables are externally accessible (i.e. in the workspace), where function variables are not. Therefore, a script is a good tool for documenting work, designing experiments and testing. In general, create a function to solve a given problem for arbitrary parameters. Use a script to run functions for specific parameters required by the assignment.

7.3 Exercises

Exercise 7.11.
Create a MATLAB function to evaluate the expression

$$f(n) = \left(1 + \frac{1}{n}\right)^n$$

as a function of n. Next check the outcome of this expression for increasing n. Can you verify that $\lim_{n\to\infty} f(n) = e$? ■

Exercise 7.12.
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 $n! = 1 * 2 * ... * n$. 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 $k = 1, 2, ..., 8$ or write a script that displays on screen the following 'triangle' (use the `fprintf` command; try `help fprintf`. Note `fprintf` is a c-command, which is in the header `cstdio` (with the same syntax), but `c++` streams are much better for dealing with input/output, so its use is not recommended in `c++`.

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

■

Exercise 7.13.
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 which computes the cumulative product of a vector elements. The cumulative product up to x_j - the j-th element of the vector x is defined by $C_j = \prod_{k=1}^{j} x_k$ for $j = 1 : \texttt{length(x)}$. Check your results with the built-in function `cumprod`.

2. Write MATLAB function `m = wmean (x,w)` computing the weighted arithmetic mean, given the vector `x` and a vector of nonnegative weights `w`, such that $\sum_{i=1}^{n} w_i > 0$, i.e. $\frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$. Add error messages to terminate execution of the function in the case when:
   - `x` and `w` are of different lengths,
   - at least one element of `w` s negative,
   - sum of all weights is equal to zero.

3. What is the greatest value of n that can be used in the sum

   $$1^2 + 2^2 + 3^2 + \ldots + n^2$$

   and gives a total value of the sum less than 100?

4. 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?

5. 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.

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

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

   – 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?

7. 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.

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

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

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

9. Write a script that asks for an integer n (or a function that has n as the input argument) and then computes the following: while the value of n is greater than 1, replace the integer with $(n/2)$ if the integer is even. Otherwise, replace the integer with $(3n + 1)$. 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?

10. Provide all prime numbers smaller then the given N.

■

Exercise 7.14.
Declare an array with 10 001 entries. Compute the solution vector $f(x_i)$, with $x_i = 0, \ldots, 10\,000$

$$f(x_i) = \frac{x_i}{\sin(x_i) + 2}$$

In the same program, form the alternating sum

$$S_n = f(x_0) + f(x_1) - f(x_2) + f(x_3) - \ldots$$

up to the last term and print the result to the screen.

■

Chapter 8

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/C++*, where structured programming is indispensable.

8.1 Structural programming

Never write all code at once; program in small steps and make sure that each of these small steps works as expected, before proceeding to the next one. Each step should be devoted to only one task. Do not solve too many tasks in one module, because you may easily make a mess. This is called a *structured* or *modular* way of programming. Formally, modularity is the hierarchical organization of a system or a task into self-contained subtasks and subsystems, each having a prescribed input-output communication. It is an essential feature of a well designed program. The 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 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,'w');
  ....                              % check whether the file fname1 exists
  fclose(fid1);
  fid2 = fopen (fname2,'w');
  ....                              % 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 **8.3**).

4. **Running and debugging** (see also section **8.2**).
   Bugs will often exist in a newly written program. Never, ever, believe or assume that the code you just created, works. **Always check the correctness of each function or script: Twice.** You may add some extra lines 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.

8.2 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. Already while editing a script or function m-file, the MATLAB editor will indicate possible errors. Add e.g. the following line to one of your m-files:

```
x = 2pi;
```

There should be a marker on the right side of the edit window. Moving the cursor over this mark will reveal the message with the syntax error: “Parse error at ’pi’: usage appears to be invalid MATLAB syntax.”.

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`, `dbclear`, `dbcont`, `dbstep`, `dbtype`, `dbup` and `dbquit`.

8.3 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
```

The MATLAB m-file editor will introduce such spaces by default.

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

```
% A BAD code that uses
% 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.']);
```

```
% 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)]);
```

5. Avoid the use of more than *one* code statement per line in your script or function m-files.

6. No line of code should exceed 80 characters (it is a rare case when this is not possible). When necessary, use the *Line Continuation Symbol*, i.e. the characters `...`, at the end of a line of to indicate that the expression continues on the next line.

7. Avoid declaring global variables. You will hardly ever encounter a circumstance under which you will really need them. Global variables can get you into trouble without your noticing it!

8. Variables should have meaningful names. You may also use the standard notation, e.g. `x`, `y` are real-valued, `i`, `j`, `k` are indices or `n` is an integer. This will reduce the number of comments and make your code easier to read. However, here are some pitfalls when choosing variable names:
   - A meaningful variable name is good, but when it gets longer than 15 characters, it tends to obscure rather than improve the code readability.
   - Be careful with names since there might be a conflict with MATLAB's built-in functions, or reserved names 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). Note that even the obvious examples of variables `i` or `j` overwrite the built-in complex imaginary unit i.
   - 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.

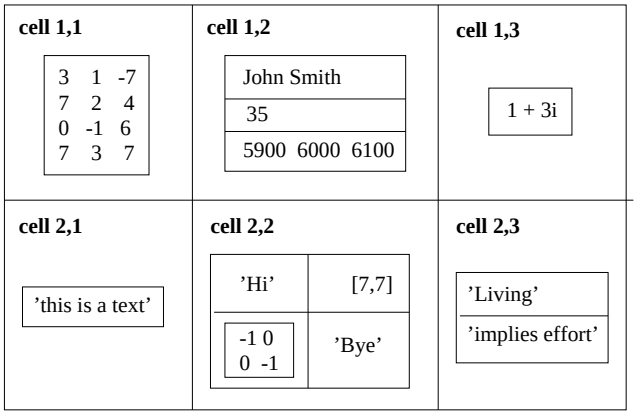
10. Test your program before submitting it. Do not just assume it works.

Chapter 9

Cell arrays and structures

9.1 Cell arrays

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



Cell arrays can be built up by assigning data to each cell. The cell contents are 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) = {[5900 6000 6100]}
>> A(1,2) = {B};      % cell array B becomes one of the cells of A
>> C(1,1) = {'Hi'}; C(2,1) = {[−1 0;0 −1]}; C(1,2) = {[7,7]}; C(2,2) = {'Bye'};
>> A(2,2) = {C};      % cell array C becomes one of the cells of A
>> A                  % A represents now the first 2 columns of the exemplar
                      % cell array
```

```
    A =
        [4x3 double]    {3x1 cell}
     'this is a text'    {2x2 cell}
>> A(2,2)            % access the cell but not its contents
ans =
    {2x2 cell}
>> A{2,2}            % use {} to display the contents
ans =
    'Hi'            [1x2 double]
    [2x2 double]    'Bye'
>> A{2,2}{2,1}       % take the (2,1) element of the cell A{2,2}
 ans =
    -1     0
     0    -1
```

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

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

```
>> 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

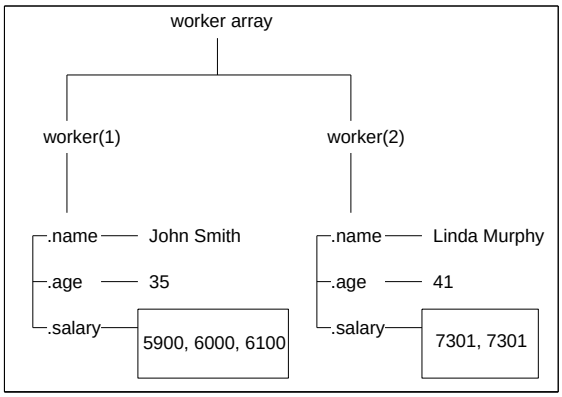
Structures are MATLAB arrays with data objects composed of *fields* and are very similar to classes in `c++`, see section 20. Each field contains one item of information. For example, one field might include a string representing a name, another a scalar representing age or an array of the last few salaries. 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 create an empty structure with fields field1, field2

```
s = struct('field1', {}, 'field2', {})
```

To create an empty structure with no fields

```
struct([])
```

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 following fields: `name`, `address`, `age`, `birthday`. Insert a few friends with related information. Remove e.g. the field `age` and add the field `phone`. ■

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

General remark: *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 classs. The use of classes and objects is the base of *object-oriented programming*, which is also possible in MATLAB.

9.3 Object handles and, the internal set and get functions

Everything MATLAB creates is an object was has children and parents very similar to OOP in C++. Understanding how this works allows very fine control over the internal working of MATALB. First of all everything in MATLAB has a unique handle (a well of identifying it). A simple handle example would be a figure handle. Type

```
>> a=figure;
a=
```

```
    1
```

The value assigned to `a` is the handle of the new figure. Note, figure are always assigned to integers starting at 1 for the first figure you create. Now you can view all the properties of this figure using the `get` command, which is the internal version of `getfield`. So typing

```
>> get(a)
    Alphamap = [ (1 by 64) double array]
    CloseRequestFcn = closereq
    Color = [0.8 0.8 0.8]
    Colormap = [ (64 by 3) double array]
    CurrentAxes = []
    CurrentCharacter =
    CurrentObject = []
    CurrentPoint = [0 0]
    DockControls = on
    DoubleBuffer = on
    FileName =
    FixedColors = [ (3 by 3) double array]
    IntegerHandle = on
    InvertHardcopy = on
    KeyPressFcn =
    KeyReleaseFcn =
    MenuBar = figure
    MinColormap = [64]
    Name =
    NextPlot = add
    NumberTitle = on
    PaperUnits = inches
    PaperOrientation = portrait
    PaperPosition = [0.25 2.5 8 6]
    PaperPositionMode = manual
    PaperSize = [8.5 11]
    PaperType = usletter
    Pointer = arrow
    PointerShapeCData = [ (16 by 16) double array]
    PointerShapeHotSpot = [1 1]
    Position = [360 280 560 420]
    Renderer = None
    RendererMode = auto
    Resize = on
    ResizeFcn =
    SelectionType = normal
    ToolBar = auto
    Units = pixels
    WindowButtonDownFcn =
    WindowButtonMotionFcn =
    WindowButtonUpFcn =
    WindowScrollWheelFcn =
    WindowStyle = normal
    XDisplay = localhost:10.0
    XVisual = [ (1 by 59) char array]
    XVisualMode = auto

    BeingDeleted = off
    ButtonDownFcn =
    Children = []
    Clipping = on
    CreateFcn =
```

```
    DeleteFcn =
    BusyAction = queue
    HandleVisibility = on
    HitTest = on
    Interruptible = on
    Parent = [0]
    Selected = off
    SelectionHighlight = on
    Tag =
    Type = figure
    UIContextMenu = []
    UserData = []
    Visible = on
```

These are all the properties of a figure in MATLAB and manipulating these allows you too fine tune many things from within your script files. This brings us to the `set` command which allows us to set properties of anything internal in MATLAB (does not have to be a figure). For example

```
>> set(a,'name','MyFigure');
```

gives your figure the name 'MyFigure'.

Now lets plot something on out figure. Type the follwing

```
>>figure(a);
>>b=ezplot('sin');
b=
   317.0021
```

The first line makes sure the figure with handle a i.e. 1 in this case is the current active figure and the second command plot the function sin on this figure. This time the returned handle is 317.0021 and this is assigned to the variable b. Now you can make the line thicker on this plot via the command

```
>> set(b,'LineWidth',10);
```

9.3.1 Parents and children

MATLAB has a hierarchal structure: A figure can have many axes and each axis can have many plots. The relationship between figure, axes and plots is maintained by the special `Parent` and `Children` fields. This will be quickly illustrated by the following example.

```
fig1=figure;
plot1=ezplot('sin');
hold on;
plot2=ezplot('cos');
```

The `hold on` makes sure the plots are on the same axes. Now notice that `get(fig1,'Children')` returns the same as `get(plot1,'Parent')` and `get(plot2,'Parent')` this is the handle of the axis the two plots are on. Now assign this axis handle to a variable; `axis1=get(fig1,'Children')` is one of three ways to get this axis handle. Notice that the axis has two children with the command `get(axis1,'Children')` and these handles are the handles of the two plots you created.

This has been a very superficial overview of the power of the set and get commands, but once you understand the basics it is a simple job to manipulate these figure properties to give you very fine control.

Exercise 9.3.
Write a function which takes a figure handles and will change the line width of all plots on the figure to 10. Hint, you need to get all axes first and then all plots on each of the found axes. ■

Chapter 10

File input/output operations

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

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

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

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

Here is an example:

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

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

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

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

Exercise 10.1.
Create a script with the code given above 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). ■

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.

10.1 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 **10.1**. Consider the following example:

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

Exercise 10.2.
Prepare a script that creates the `sintab.txt` file, containing a short table of the sinus 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 10.3.
Create the following script and try to understand how it works (use the `help` command to learn more on the `feof` function):

```
fid  = fopen ('exptab.txt','r');
title = fgetl (fid);
k = 0;
while feof(fid)~=1                    % as long as end-of-file is not reached do
  k = k+1;
  line = fgetl (fid);                % get a line from the exponential table
  tab(k,:) = str2num (line);         % convert the line into a vector from tab
                                     % matrix
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 10.4.
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.

10.1.1 Flexible reading from a file

fscanf requires the user to know exactly how a file is formatted before it can be read correctly into MATLAB. This does mean you are certain what you will get, but can require the writing of a lot of messy code to do some very standard file reading. Therefore, two extra commands have been added to MATLAB that are more flexible in their ability to read in text files; however, the downside is you do not know in advance how many lines of text will be read. These commands are textread and textscan.

textread

The basic syntax for textread is

```
my_array_variable = textread('name_of_file');
```

The reads the contexts on the file 'name_of_file' into an array called `my_array_variable`. The command can also be used to read in column into a separate variable i.e.

```
[col1,col2,col3,col4] = textread('name_of_file');
```

would place the first column of the file 'name_of_file' in a variable called `col1`, second into a varible called `col2`, etc.

`textread` also accepts many additional flags that allow advance interpretation of the text files. The most useful includes `headerlines` which ignores a specified number lines at the beginning of the file; `delimiter` which you to specify the delimiter (separator) between different elements of data; and `emptyvalue` which allows you no specify how to treat empty data elements. This three comments are illustrated in the following example. Consider a file called `grades.dat` containing

```
Student grades in APiE
Year 2011–2012
Thornton, 9,,9
Weinhart, 5,4,5
Smith 8,8,7
```

The grades could be read with the command

```
[names,assignment1,assignment2,assignment3] = ...
textread('grades.dat, '','headlines',2' 'delimiter', ',','emptyvalue', NaN);
```

This would create four new variables and example of the contents is illustrated below.

```
name=
'Thornton',
```

```
'Weinhart'
'Smith'
assignment2=NaN 4 8
```

Note, MATLAB deals with the different types of data and the missing grade is replaced with `NaN` as requested.

textscan

A sister command to `textread` is `textscan`. The main difference with `textscan` is that it operates on open files i.e. the file must first be opened with `fopen` and then the fid number not name of the file is used. So a simple example of the use of `textscan` would be

```
fid = fopen('scan1.dat');
C = textscan(fid, fid = fopen('scan1.dat');
C = textscan(fid, '%s', 'delimiter', '\n');
fclose(fid);
```

where the `%s` tells MATLAB to interpret the data as strings. There are a number of advantage of `textscan` over `textread` and these include

- it is faster;
- it remembers the point you are at in the file (because the file remains open after the command);
- it always returns a single cell array (not really an advantage, but you should be aware of this).

10.2 Working with Excel

Using `xlswrite` you write Microsoft Excel spreadsheet file (.xls)

This example writes the following mixed text and numeric data to the file tempdata.xls

```
d = {'Time', 'Temp'; 12 98; 13 99; 14 97};
```

The 4-by-2 matrix will be written to the rectangular region that starts at cell E1 in its upper left corner

```
s=xlswrite('tempdata.xls', d, 'Temperatures', 'E1')
```

Using `xlsread` you read Microsoft Excel spreadsheet file (.xls)

Exercise 10.5.
Prepare a script that creates the `costab.xls` file, containing a short table of the cosinus function.
■

Exercise 10.6.
Open the file *Mesh.inp*. Extract the coordinate of the nodes (they are introduced by the string $*Node$) and the nodal sets (they are introduced by the string $*Nset$).

Plot the nodes on a figure with circles and the nodes contained in the nodal sets with triangles.

■

Chapter 11

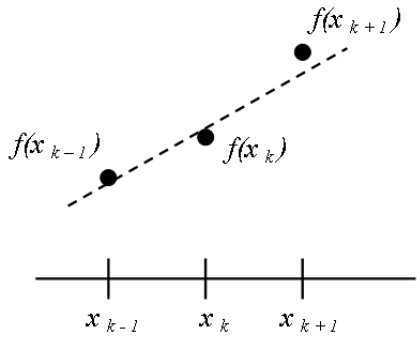
Numerical analysis

Numerical analysis can be used whenever it is impossible or very difficult to determine the analytical solution. This is often the case when analysing experimental results. MATLAB can be used to find, for example, the minimum, maximum or integral of a certain function or set of experimental data.

In this section we will discuss some issues on numerical integration and differentiation. Apart from that, the possibility to solve differential equations numerically will be introduced.

11.1 Differentiation

The derivative of a function $f(x)$ is defined as $f'(x) = \frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x)-f(x)}{(x+\Delta x)-x}$. In practice, the derivative of $f(x)$ at a is the slope of the line tangent to $f(x)$ at a. Numerical algorithms for computing the derivative of a function thus require the estimate of the slope of the function for some particular range of x values. Three common approaches to do this are the *backward difference*, *forward difference* and *central difference*. The figure below helps to illustrate the difference between these three approaches:



- **Backward difference**: $f'(x) \approx \frac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}$
- **Forward difference**: $f'(x) \approx \frac{f(x_{k+1})-f(x_k)}{x_{k+1}-x_k}$
- **Central difference**: $f'(x) \approx \frac{f(x_{k+1})-f(x_{k-1})}{x_{k+1}-x_{k-1}}$

To calculate the numerical derivative, the MATLAB function `diff(x)` can be used, which computes the differences between adjacent values of the vector `x`. If values of $f(x)$ are contained in vector `y` and the corresponding x values in vector `x`, the derivative of the function can be estimated using

```
>> deriv_y = diff(y)./diff(x);
```

The vector (or matrix) that `diff(x)` returns contains one less element (or row) than `x`. The corresponding x values are obtained from the original `x` vector by trimming either the first or last value;

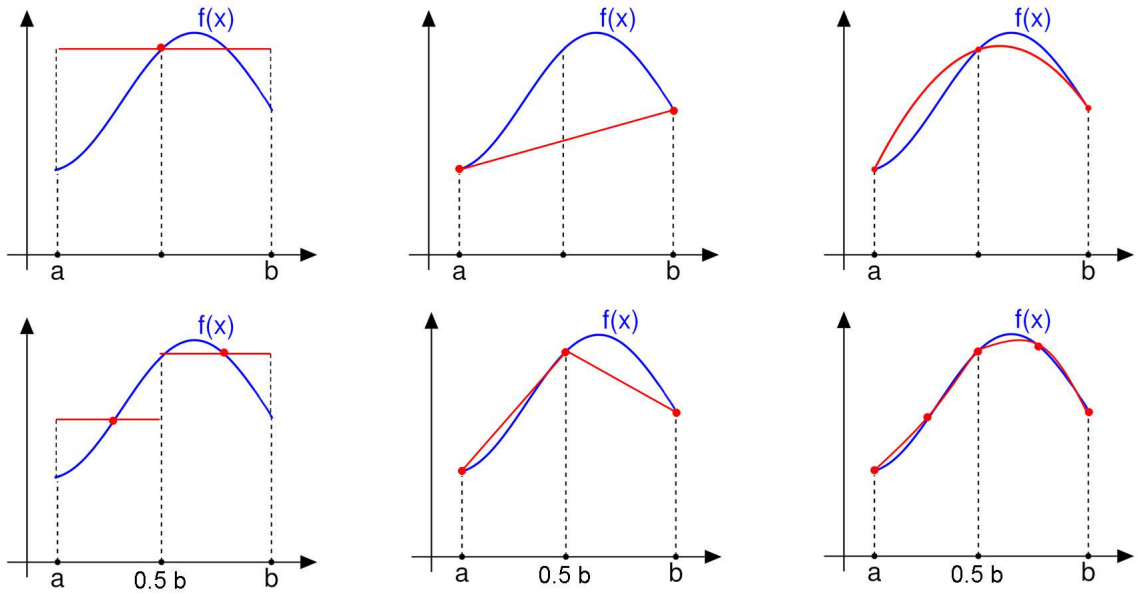
- Trimming the last value results in a forward difference estimate.
- Trimming the first value results in a backward difference estimate.

To find and plot for instance the derivative of a function $f(x) = 5\cos(10x) + x^3 - 2x^2 - 6x + 10$, we do:

```
>> x=0:.01:4;
>> y=5*cos(10*x) + x.^3 - 2*x.^2 - 6*x + 10;
>> subplot(1,2,1)
>> plot(x,y)
>> title('f(x) = 5cos(10x) + x^3 - 2x^2 - 6x +10')
>> deriv_y = diff(y)./diff(x);
>> xd = x(2:length(x));    % Backward difference x values
>> subplot(1,2,2)
>> plot(xd,deriv_y)
>> title('f`(x)')
>> hold on
>> dy=-50*sin(10*x) + 3*x.^2 - 4*x - 6;  % The exact function for f`(x)
>> plot(xd,dy(2:length(x)),'r')  % Showing that the numerical result is correct
```

11.2 Integration

The integral, or the surface underneath a 2D function, can be determined numerically. A broad family of algorithms for calculating the numerical value of a definite integral have been derived, some of which are displayed in the following figure.



- The left panel is a representation of the *Rectangle rule*. Here, the interpolating function is a constant function (a polynomial of degree zero) which passes through the point $((a+b)/2, f((a+b)/2))$. The approximation of the integral can be described as follows: $\int_b^a f(x)dx \approx (b-a)f(\frac{a+b}{2})$
- The middle panel is a representation of the *Trapezoidal rule*. Here, the interpolating function is a straight line (a polynomial of degree 1) which passes through the points $(a, f(a))$ and $(b, f(b))$. The approximation of the integral can be described as follows: $\int_b^a f(x)dx \approx (b-a)\frac{f(a)+f(b)}{2}$
- The right panel is a representation of the *Simpson's rule*. Here, the interpolating function is a polynomial of degree 2 which passes through the end points a and b, and the midpoint $m = (a+b)/2$. The approximation of the integral can be described as follows: $\int_b^a f(x)dx \approx \frac{b-a}{6}(f(a) + 4f(\frac{a+b}{2}) + f(b))$

As can be clearly seen in the figure, the accuracy of the different methods can be increased by dividing the interval (a, b) into multiple subintervals, computing an approximation for each subinterval, and then adding up all the results.

11.2.1 MATLAB commands

MATLAB has several commands to determine the integral of a function. One of these commands is `trapz`, which numerically integrates using the Trapezoidal rule. The accuracy of this method

also depends on the distance between the data points. Use `help` to learn more.

```
>> 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 `quad` or `quadl`, which numerically evaluate an integral of a function using an adapted Simpson's rule. Where the input for `trapz` is a vector or a matrix, the input for `quad` or `quadl` is specified by a **string** or by an **inline** definition (have a look at the intermezzo to learn more about inline functions).

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

```
>> format compact
>> f='1./((x-0.1).^2+0.1)+1./((x-1).^2+0.1)';
>> integral1=quad(f,0,2)
integral1 =
   13.4118
>> integral2=quadl(f,0,2)
integral2 =
   13.4118
>> whos
  Name           Size            Bytes  Class     Attributes

  f              1x37               74  char
  integral1      1x1                 8  double
  integral2      1x1                 8  double
```

Note in the previous example that `f` is defined as a character array (a string).

An extra parameter can be added to `quad` and `quadl` to specify the accuracy of the evaluation. If this parameter is not defined as in the previous example, the default error tolerance is 1.e-6.

```
>> f='1./((x-0.1).^2+0.1)+1./((x-1).^2+0.1)';
>> integral1=quad(f,0,2)
integral1 =
   13.4118
>> integral2=quad(f,0,2,1.e-4)
integral2 =
   13.4119
>> integral3=quad(f,0,2,1.e-1)
integral3 =
   13.2654
```

Apart from the value of the approximated integral, MATLAB can also return the number of function evaluations that are needed to reach the defined (or default) error tolerance.

```
>> >> f='1./((x-0.1).^2+0.1)+1./((x-1).^2+0.1)';
>> [integral1,fcnt1]=quad(f,0,2)
integral1 =
```

```
   13.4118
fcnt1 =
    97
>> [integral2,fcnt2]=quad(f,0,2,1.e-3)
integral2 =
   13.4117
fcnt2 =
    29
```

This clearly illustrates that larger values for the tolerance result in fewer function evaluations and thus faster computation.

INTERMEZZO

Inline functions

It may also be useful to define a function that will only be used during the current MATLAB session. MATLAB offers a command `inline` to define the so-called inline functions in the Command Window. For instance,

```
>> f = inline('cos(x).*sin(2*x)')
f =
     Inline function:
     f(x) = cos(x).*sin(2*x)
>> g = inline('sqrt(x.^2+y.^2)','x','y')
g =
     Inline function:
     g(x,y) = sqrt(x.^2+y.^2)
```

You can evaluate this function in a usual way:

```
>> f(-2)
ans =
   -0.3149
>> g(3,4)
ans =
     5
>> A = [1 2; 3 4];
>> B = [2 3; 4 5];
>> g(A,B)                          % the function also works with arrays
ans =
    2.2361    3.6056
    5.0000    6.4031
```

END INTERMEZZO

11.3 Solving differential equations - The ODE toolbox

Before going into mathematical details, we would like to point out that we have to be very carefull using the ODE toolbox as a *black box solver* for various ordinary differential equations (ODEs). Note that you have to analyze carefully the *type* of ODE before you use a certain numerical method to solve it. With the numerical method, it is e.g. important to distinguish so-called *stiff* and *non-stiff* ODEs.

A *stiff* equation is a differential equation for which certain numerical methods for solving the equation are numerically unstable, unless the step size is taken to be extremely small.

Here, we discuss only functions which can be used for *non-stiff* problems, such as `ode23` and `ode45`. Note that in MATLAB also solvers for *stiff* problems are included; `ode15s, ode23s, ode23t` and `ode23tb`.

11.3.1 1st order ODE

MATLAB has an extensive library of functions for solving ordinary differential equations. In this course we will only show a very basic introduction. Therefore, we will focus on the main two, the built-in functions `ode23` and `ode45`, which implement versions of Runge-Kutta 2nd/3rd-order and Runge-Kutta 4th/5th-order, respectively.

Example 11.1:
Solve the following ODE of 1st order

$$\frac{\mathrm{d}y}{\mathrm{d}x} = x\,y^2 + y, \qquad y_0 := y(x=0) = 1,$$

in the intervall $x \in [\,0, 0.5\,]$

For any differential equation in the form $\mathrm{d}y/\mathrm{d}x = y' = f(x, y)$, we begin by defining the function $f(x, y)$. Note that for single equations, we can define $f(x, y)$ as an inline function which simplifies the procedure. The basic usage for MATLAB's solver `ode45` is

```
ode45(function,domain,initial condition)
```

Thus, we can formulate the problem as

```
>> f  = inline('x*y^2+y');
>> [x,y] = ode45(f,[0 0.5],1);
>> plot(x,y)
```

MATLAB returns two column vectors, the first with values of `x` and the second with values of `y` which can be plotted with the standard command `plot(x,y)`.

Approximating this solution numerically, the algorithm `ode45` has selected a certain discretization of the interval $[\,0, 0.5\,]$, and MATLAB has returned a value of `y` at each point in the partition. It is often the case that we would like to specify the partition, i.e. the numerical effort, on which MATLAB returns an approximation. For example, we might only want to approximate the solution at 5 points: $y(.1), y(.2), \ldots, y(.5)$. We can specify this by entering the vector `[0,.1,.2,.3,.4,.5]` as the domain in `ode45`. That is, we use

```
>> ic    = 1.0;
>> disc  = 0:0.1:0.5;
>> f  = inline('x*y^2+y');
>> [x,y] = ode45(f,disc,ic);
>> plot(x,y)
```

It is important to point out here that MATLAB continues to use roughly the same partition of values that it originally chose; the only thing that has changed is the values at which it is returning a solution. In this way, no accuracy is lost.

Several options are available for MATLAB's `ode45` solver, giving the user limited control over the algorithm. Two important options are relative and absolute tolerance, respecively `RelTol` and `AbsTol` in MATLAB. At each step of the `ode45` algorithm, an error is approximated for that step. If y_k is the approximation of $y(x_k)$ at step k, and e_k is the approximate error at this step, then MATLAB chooses its partition to insure

$e_k \leq max(RelTol * y_k, AbsTol)$

where the default values are RelTol = .001 and AbsTol = .000001. As an example for when we might want to change these values, observe that if y_k becomes large, then the error e_k will be allowed to grow quite large. In this case, we can increase the value of RelTol. For the equation $\frac{dy}{dx} = xy^2 + y$ with $y(0) = 1$, the values of y get quite large as x nears 1.

```
>> f=inline('x*y^2+y');
>> [x,y]=ode45(f,[0 1],1);
Warning: Failure at t=9.999897e-001.  Unable to meet integration tolerances
without reducing the step size below the smallest value allowed (1.776357e-015)
at time t.
> In ode45 at 371
```

In order to fix this problem, we can choose a smaller value for RelTol.

```
>> f=inline('x*y^2+y');
>> options=odeset('RelTol',1e-10);
>> [x,y]=ode45(f,[0 1],1,options);
```

11.3.2 Systems of ODE

Solving a system of ODE in MATLAB is quite similar to solving a single equation, though since a system of equations cannot be defined as an inline function we must define it as an M-file. As an example, let's solve the system of Lorenz equations (The Lorenz equations have some properties of equations arising in atmospherics. Solutions of the Lorenz equations have long served as an example for chaotic behavior),

$\frac{dx}{dt} = -\sigma x + \sigma y$

$\frac{dy}{dt} = \rho x - y - xz$

$\frac{dz}{dt} = -\beta z + xy$

where for the purposes of this example, we will take $\sigma = 10$, $\beta = \frac{8}{3}$, and $\rho = 28$, as well as $x(0) = -8$, $y(0) = 8$, and $z(0) = 27$.

First, a MATLAB M-file lorenz.m is created that contains the Lorenz equations

```
function xyz = lorenz(t,x);
%LORENZ: Defines the Lorenz equations.
sig=10;
beta=8/3;
rho=28;
xyz=[-sig*x(1) + sig*x(2); rho*x(1) - x(2) - x(1)*x(3); -beta*x(3) + x(1)*x(2)];
```

Observe that x is stored as x(1), y is stored as x(2), and z is stored as x(3). Additionally, xyz is a column vector, as is evident from the semicolon following the first appearance of x(2).

Now, the solution of this system of ODE over the interval t=[0,20] can be found:

```
>> x0=[-8 8 27];
>> tspan=[0,20];
>> [t,xyz]=ode45(@lorenz,tspan,x0);
```

The output for this last command consists of a column t of times followed by a matrix xyz with three columns, the first of which corresponds with values of x at the associated times, and similarly for the second and third columns for y and z.

Exercise 11.1.
Create a figure with 4 subplots. The first three should contain the components of the lorenz equation as introduced in this chapter as a function of t over the interval t=[0,20]. The fourth should display the 'Lorenz attractor', which is a 3D plot of the three coordinates. ■

11.4 Exercises

Exercise 11.2.
Determine the first and second derivative of the line through `(x,y)` with `x = 4:13` and `y = [12.84 12.00 7.24 -0.96 -11.16 -20.96 -27.00 -24.96 -9.56 25.44]`. Make a figure containing the plot of the initial data, the first derivative, and the second derivative in separate sub-plots. ■

Exercise 11.3.
Make a figure containing 2 sub-plots, and plot the function $\sin(x)/x$ and its integral on the interval $[0, 10]$. Find the location of the roots of f(x) and mark the values corresponding to the roots in each figure. ■

Exercise 11.4.
Write a script that repeats the example displayed in the section on noise. However, add a column of plots showing the filtered data, the first derivative of the filtered data, and the second derivative of the fltered data (thus acquiring a figure with 3x3 plots). Also plot the exact derivatives of the function in the graphs as a control (you can use the MATLAB symbolic toolbox to find the derivatives). ■

Exercise 11.5.
Write scripts that compute the integral of the function $f(x) = \frac{1}{1+x^2}$ using the Rectangle rule,

the Trapezoidal rule and the Simpson's rule. Use the scripts to find the integral over the interval $[0, 4]$. Exercise with different accuracies (number of subintervals) and compare the results with the analytical value of the integral. ■

Exercise 11.6.
Adapt the scripts you prepared for the Trapezoidal rule and the Simpson's rule to determine the integral of $f(x) = sin(x)$ over the interval $[0, \frac{\pi}{2}]$. Do this with different numbers of subintervals; `n=[2 4 8 16 32 64 128 256]`. For both the Trapezoidal rule and the Simpson's rule, create a matrix with 4 rows containing the values of n, the approximation of the integral, the error of the approximation, and the ratio between the errors of consecutive approximations. What does this tell you about the accuracy of both methods? ■

Exercise 11.7.
Find the integral of the function $f(x) = e^{-x^2/2}$ over the interval $[-3, 3]$. Exercise with the MATLAB commands `trapz`, `quad` and `quadl`, and different accuracies. With `quad` and `quadl`, get an idea of the number of function evaluations that are needed to reach a certain accuracy ■

Exercise 11.8.
The commands `quad` and `quadl` can also be used to integrate a function stored in a .m file (use `help` to find out how to do this). Make a .m file defining the function $y = sin(x^5 + x^3)$. Then use `quad` to determine the integral of this function over the interval $[0, 2]$. Also try to find the exact value of the integral using the symbolic toolbox. Why can numerical integration be useful, even though it results in an approximation of the integral? ■

Exercise 11.9.
Solve the following ODE of 1st order

$$\frac{\mathrm{d}y}{\mathrm{d}x} = x\,y^2 + y, \qquad y_0 := y(x = 0) = 1,$$

in the domain $x \in [\,0, 0.9\,]$ with an equidistant discretization of $\Delta x = 0.01$. Use a m-file `firstode.m` to define the function. Plot the result. ■

Exercise 11.10.
The predator-prey model is a model that describes the number of predators and preys over time. When the prey is defined as x and the predator is defined as y, the number of predators and preys can be defined with the following system of ODE:

$\frac{dx}{dt} = \alpha x(1 - y)$

$\frac{dy}{dt} = \beta y(x - 1)$

where for the purposes of this example, we will take $\alpha = 5$ and $\beta = 1$, as well as $x(0) = 3$ and $y(0) = 1$. Solve the predator-prey model over the interval t=[0,10] and make a plot displaying the populations of preys and predators in a single graph.

■

Chapter 12

Some longer exercises

12.1 Matrices and Vectors

Exercise 12.1.
Create a vector containing 5 elements such that it's components are equally spaced when you take the logarith of it ■

Exercise 12.2.
Perform the following exercises:

- Create a vector `x` with the elements:
  - `1, 1/2, 1/3, 1/4, 1/5`
  - `0, 1/2, 2/3, 3/4, 4/5`

  To do this, divide a vector `y` by a vector `z`.
- Given a vector `t`, write down the MATLAB expressions that will compute:
  - $\texttt{ln}\,(2 + \texttt{t} + \texttt{t}^2)$
  - $\texttt{cos}\,(\texttt{t})^2 - \texttt{sin}\,(\texttt{t})^2$
  - $\texttt{e}^{\texttt{t}}\,(1 + \texttt{cos}\,(3\texttt{t}))$
  - $\texttt{tan}^{-1}(\texttt{t})$

  Test them for $\texttt{t} = 1 : 0.2 : 2$.

■

Exercise 12.3.
Use the knowledge on computing the inner product to find:

1. the Euclidean length of the vector $\texttt{x} = [2\ 1\ 3\ 7\ 9\ 4\ 6]$, which is defined as $||\texttt{x}|| = \sqrt{(\Sigma \texttt{x}_\texttt{i}^2)}$.

2. the angle between two column vectors, which is defined as $\cos\alpha = \frac{\mathtt{x}^T\mathtt{y}}{||\mathtt{x}||\,||\mathtt{y}||}$. Note that you can also use the MATLAB command norm(v), which gives you the Euclidean length of a vector v. The MATLAB command acosd(x) can be used to find the inverse cosine of x expressed in degrees. Compute the angle between two vectors:
   - x = [3 2 1] and y = [1 2 3]
   - x = 1 : 5 and y = 6 : 10

■

Exercise 12.4.
Clear all variables (use the command `clear`). Define the matrix `A = [1:4; 5:8; 1 1 1 1]`. Predict and check the result of the following operations:

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

■

Exercise 12.5.
Define the matrices `T = [ 3 4; 1 8; -4 3] and A = [diag(-1:2:3) T; -4 4 1 2 1]`. Perform the following operations on the matrix A:

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

■

Exercise 12.6.
Let A = [2 4 1; 6 7 2; 3 5 9]. Provide the commands which:

- assign the first row of A to a vector x;
- assign the last 2 rows of A to a vector y;
- add up the columns of A;
- add up the rows of A;

■

Exercise 12.7.
Let A = [2 7 9 7; 3 1 5 6; 8 1 2 5]. Explain the results or perform the following commands:

- A′
- A(1, :)′
- A(:, [14])
- A([23], [31])
- reshape (A, 2, 6)
- A(:)
- flipud (A)
- fliplr (A)
- [A A(end, :)]
- [A; A(1 : 2, :)]
- sum (A)
- sum (A′)
- mean (A)
- mean (A′)
- sum (A, 2)
- mean (A, 2)
- min (A)
- max (A′)
- min (A(:, 4))
- [min(A)′max(A)′]
- max (min(A))
- [[A; sum (A)] [sum (A, 2); sum (A(:))]]
- assign the even-numbered columns of A to an array B
- assign the odd-numbered rows to an array C
- convert A into a 4-by-3 array
- compute the reciprocal of each element of A
- compute the square-root of each element of A
- remove the second column of A
- add a row of all 1's at the beginning and at the end
- swap the 2nd row and the last row

■

Exercise 12.8.
Given the vectors x = [1 3 7], y = [2 4 2] and the 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:

- x + y
- x + A
- x′ + y
- A − [x′ y′]
- [x; y] + A
- [x; y′]
- [x; y]
- A − 3
- A + B
- B′ + A
- B ∗ A
- A. ∗ B
- A′. ∗ B
- 2 ∗ B
- 2. ∗ B
- B./x′
- B./[x′ x′]
- 2/A
- ones(1, 3) ∗ A
- ones(1, 3) ∗ B

■

12.2 Visualization

Exercise 12.9.
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 12.10.
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 12.11.
Plot a circle with the radius $r = 2$, knowing that the parametric equation of a circle is $[x(t), y(t)] = [r\,cos(t), r\,sin(t)]$ for $t = [0, 2\pi]$. ■

Exercise 12.12.
Plot an ellipse with semiaxes $a = 4$ and $b = 2$. ■

Exercise 12.13.
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 12.14.
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? ■

Exercise 12.15.
Produce a nice graph which demonstrates as clearly as possible the behavior of the function $f(x, y) = \frac{x\,y^2}{x^2+y^4}$ near the point $(0, 0)$. Note that the sampling around this points should be dense enough. ■

Exercise 12.16.
Plot a sphere, which is parametrically defined as $[x(t, s), y(t, s), z(t, s)] = [\cos(t) * \cos(s),\ \cos(t) * \sin(s),\ \sin(t)]$ for $t, s = [0, 2\,\pi]$ (use `surf`). Make first equal axes, then remove them. Use `shading interp` to remove black lines (use `shading faceted` to restore the original picture). ■

Exercise 12.17.
Plot the parametric function of r and θ: $[x(r, \theta), y(r, \theta), z(r, \theta)] = [r\ \cos(\theta),\ r\ \sin(\theta),\ \sin(6\ \cos(r) - n\theta)]$ for $\theta = [0, 2\,\pi]$ and $r = [0, 4]$. Choose n to be constant. Observe, how the graph changes depending on different n. ■

Exercise 12.18.
Plot the surface $f(x, y) = x\,y\,e^{-x^2-y^2}$ over the domain $[-2, 2] \times [-2, 2]$. Find the values and the locations of the minima and maxima of this function. ■

Exercise 12.19.
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. ■

Exercise 12.20.
Plot an equilateral triangle with two vertices $[a\ a]$ and $[b\ a]$. Find the third vertex. Use `fill` to paint the triangle. ■

Part: B

C++

Chapter 13

Introduction to C/C++

13.1 Objective of the course

The goal of this short course in C/C++ is not to become a perfect C/C++ programming guru. For this we refer to special courses and a broad range of expert literature – see below in section 23.1

The goal of this introduction to C/C++ is to better understand computer hardware and software, to perform basic tasks, operations, and programming within the framework of the programming language C/C++. The goal of this course is to first install on your computer an advanced Graphical User Interface (GUI) software toolbox like DEV-C++ and make you understand simple C/C++ programs, allow you to write own code, and to get used to error-searching/debugging of code.

First of all you need a C++ compiler. On UNIX and LINUX systems this should be installed already. (If not, install the `g++` compiler and debugger.) On MS-Windows and MAC OS you have to follow the instructions in section 13.2.

13.2 Installation of a C++ compiler/GUI

There are many commercial C++ compilers for various operating systems available. While many Unix-type operating systems like modern Linux distributions have C++ compilers and debuggers embedded (g++ and gcc), this is not the case for a MS-Windows systems like Microsoft Windows XP or Microsoft Vista. Nevertheless, you can use the following open source GUI-based C++ compiler (GNU General Public License (GPL)) on a MS-Windows system.

13.2.1 Windows : All versions

- **Download from** `http://sourceforge.net/projects/orwelldevcpp/`
- **Installation.**
  Double-click on the setup file. On the `Choose components` window,
  select the `Full` type of installation.
- **First time configuration.**
  Accept all the default configurations.

  A very useful tool is to have line numbers displayed.
  Tools → Editor options → Display → Line numbers

  In case the linker does not find some functions, you have to set the following command line options.
  Tools → Compiler options
  and insert `-lm -l .` into both fields and activate the input.
- **Ready to start.**
  First create a new C++ source file and save it under the name e.g. `test.cpp`.

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

  Alternatively, you can create a new project:
  File → New → Project → Console Application.
  At this point the project is created with a main file in it. You need to save the main file under the name e.g. `test.cpp`. Then you have the leave the project and open the source file (projects are much too complicated/overshooting for our purposes - we recommend to avoid them). Then you can compile it and run it!
  Furthermore, you can try to modify the file and compile and run it.

```
cout <<"hello!"<< endl;
system("PAUSE");
return EXIT_SUCCESS;
```

13.2.2 MAC

For the MAC you need to download and install Xcode. This comes with the `g++` compiler and a development environment called Xcode. Unfortunately you have to register with Apple to be able to download the package (only takes ten minutes). For more information
`http://developer.apple.com/technologies/tools/xcode.html`

13.3 A short history of C/C++

C was developed in the 70's by K. Thompson, B. Kernighan, and D. Ritchie, and was introduced together with the operating system UNIX. Today, LINUX is the free derivate of UNIX, and like its ancestor is based on C, i.e., the operating system is – in big parts – written in C. The original goal was a standardised language for the structured programming. However, the so-called ANSI-standard was only introduced as late as 1988 as ANSI-C.

C++ can be seen as an extension of C, developed, propagated and introduced between 1983 and 1985 by B. Stroustrup. Besides the goal to develop a higher, object oriented language, down-ward compatibility with C was desired. Therefore, C is to be seen as a fraction/part of C++. The ANSI Standard for C++ was only fixed in June 1998. In the framework of this course, we will use C++ syntax whenever this is simpler than C, e.g. for I/O (input/output).

Some reasons for the use of C/C++ are

- it is free! (e.g. g++ compiler or DevC++ environment)
- it is system-independent – applicable also on super-computers
- it is rather low level – close to the operating system
- it is extendable, re-usable, and robust, ...

Chapter 14

Programming-style

This section briefly discusses some issues related to a good style of programming. Good style means that programs are simple and easy to read. It involves some rules like indentations that make the program also optically structured and, finally, it involves comments that describe in words what the program is doing.

Note that a good style does not only help other people reading and understanding your program, it also will help you to understand your own program – after you have not looked at it for a while. Think not of the few 100 lines we will write in this course, think of several 10,000 lines of code that a programmer can write during a thesis for example.

In "real life" people do NOT spend most of the time with programming itself, but with **debugging** (error-search), updates, changes, optimisation and re-using of existing codes and code-fragments. For these tasks, a clean, proper, and consistent style is not only helpful, but absolutely necessary.

This is – by the way – true for all programming languages, including MATLAB.

14.1 Comments (see also §15.1)

There are two things a program should do: First, it tells the computer what to do and, second, it also tells the (human) reader of the program what it does.

A working program without comments is a time-bomb. Even the simplest ideas flowing into the program will be forgotten after some time ... and without comments, a program can not be re-used. Without comments, a program cannot be understood or changed so that either it has to be program anew, or a lot of time is spent re-understand the exciting program; both are a waste of time.

In C++ there are two types of comments.

The first, traditional comments begin with `/*` and end with `*/`. They can enclose single characters or many lines and are most convenient for larger comments, or for making old code invisible to the compiler.

The alternative comment begins with two slash `//` and ends automatically at the end of a line. It can start anywhere in the line and is thus convenient for commenting on single commands.

As a suggestion, each program should contain at the beginning:

- ***Header***
  Name of program and what is it supposed to do.
- ***Author***
  Author and (e-mail) contact address
- ***Date***
  Date of changes
- ***How to use the program***
  Here the mode of usage is described
- ***Files (I/O)***
  Input data file format and output data-files
- ***Restrictions***
  Are there, e.g., restrictions to the input-data-files?
- ***Revisions***
  Dates and type of changes performed
- ***Error-management***
  What happens when a program detects an error
- ***Other remarks***
  ...

14.2 Clear style

A program should be read like a thesis or a scientific paper. It should contain an introduction, like the header-comments above, and it should be clearly separated in chapters, paragraphs, etc. with clear boundaries. Comments can be used as a way to optically separate paragraphs within the program.

Chapter 15

The basics

15.1 Language Elements

The first language element are *comments*. They do not affect the function of the program or the computer – they enhance the understanding of the program when read by another (human) user. A comment is everything between '`/*`' and '`*/`' or after '`//`'.

The second essential element are *data*, i.e., counters, physical constants, variables, etc. It is necessary to first tell the computer which data are required for the program to work. This could be done anywhere in the program, however, it is neither efficient nor clean or clear programming. It is recommended to define data (only) at the beginning of a function/program. Exceptions to this rule will be discussed below.

Operators, *Control Structures*, and *Functions* are required to manipulate the data. Examples for operators are the assignment operator `=`, the basic arithmetic operations `+`, `-`, `*`, or `/`, and comparison operators (`==`, `!=`, `>`, `<`, `>=`, `<=`). However, there are many more as summarised in the following tables and discussed in more detail later in §15.3.

Operator type	Operators
Assignment	`=`
Arithmetic operators	`+, -, *, /, %`
Increase and decrease	`++, --`
Relational and equality operators	`==, !=, >, <, >=, <=`
Logical operators	`!, &&, ||`
Bitwise Operators	`~, &, |, ^, <<, >>`
Compound assignment	`+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, =`
Other operators	`(type), ? :, ,, ::, (), [], ., ->, *, &, ...`

Table 15.1: Examples of operators, sorted by type

expression	short-form for:
`x += y`	`x = x+y`
`x -= y`	`x = x-y`
`x *= y`	`x = x*y`
`x /= y`	`x = x/y`
`x %= y`	`x = x%y`

Table 15.2: Short expressions and the long form they replace

Examples for control structures are `for`- and `while`-loops, situation dependent execution with `if`, `else if`, and `else` structures, block begin- and end-brackets { ... } and many others as defined in the next subsections.

Examples for functions are the trigonometric functions `sin` and `cos`, but functions can also be user-defined groups or sequences of operations that are repeated many times. Use of functions makes the program shorter and better readable.

Note that the so-called main program `main()` is also a function – it is *always* called `main`. (That means no other function can use this name.) Begin and end of a function are marked by the curly brackets '{' and '}'.

The next class of language elements are those used for the communication between the user and the computer, the so-called I/O (input/output) functions and operators. In C/C++ standardised libraries exist already that contain the definitions of the I/O commands.

The simplest way of I/O is output on the screen and input via the keyboard. The next way – especially when a lot of information has to be input into the program or is returned – is via data-files (for both input and output).

Finally, there are *organisational* elements, that do not per-se belong to the programming language, but can be used for management of the program *before* compilation and execution, e.g., by the so-called pre-processor. This allows to shorten and clarify the program by just moving big, un-clear code-fragments out of the program, or by conditionally selecting alternative parts of the code.

The first program (DevC++ $\longrightarrow$ New Program) looks like:

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- The first lines include libraries `#include <iostream>` and `#include <cstdlib>` to get the definitions from the standard libraries for I/O and standard C-routines and functions. The former are required for input and output of text and numbers, the latter contain various helpful and necessary tools. This line must be situated at the beginning of each main-program – outside of the `main()...` function. This library contains operators/functions like '`cin`', '`cout`', '`<<`' or '`>>`'.

- The command `using namespace std` imports all symbols (like `cin` or `cout`) from namespace `std`. Such namespaces in C++ are used, to separate algorithms from different libraries. A command `cout` from another namespace, e.g., a self-defined one, cannot be used anymore. This is supposed to help against confusing names for functions: in principle, C++ allows "overloading"; event though this provides an enormous freedom, it is better to avoid it in order to avoid confusion. In general, we will use the C++ standard libraries, whereas for using C-functions, the command `using` is explicitly needed.

  Note: You can also import individual symbols, for example `using std::cout` imports the command `cout` from the `std` namespace. We will see what this command does in §16.1

- The command `main(...)` is a function of integer type `int`. Every program contains at least one such function (there can be much more functions, but only one special "main". The terms in brackets are used to transfer parameters to the function main, however, we delay the discussion of this to later.

- `return EXIT_SUCCESS;` returns the value of 0 to the calling program, i.e., to the operating system. `EXIT_SUCCESS` is defined in the standard library as integer with value zero. Zero is the common return value if the function finished without errors.

- The brackets `{...}` enclose the main-program, i.e. the commands, the main function `main(...)` consists of. In general, program blocks are enclosed by these curly brackets, and each begin-bracket `{` must be accompanied by an end-bracket `}`.

- The command `system("PAUSE");` writes the word "Pause" to the screen and then waits for a key-stroke from the user. This is a convenient way to stop the program before it completely ends and disappears from the screen.

  Note : system commands are dependent on your operating system; `system("PAUSE");` is a Windows only command; remove this line if you work with MAC or Linux. This command is only required for windows, were new temporary terminals are created for the execution of your program. For MAC and Linux you run you program in a terminal which exists before your code is run and still exists after.

- Note that *every* command has to be ended by a semicolon '`;`'. This is because a new line alone is not the beginning of a new command or program element. It is better to not use several '`;`' within a single line in order to keep the program clean and readable. Since the semicolon has the function to separate two commands, it is *not* found at the end of program-blocks, that are already clearly terminated by the end-bracket of the pair `{...}`.

15.2 Data-types, Variables, Constants

15.2.1 Numbers

Computers process numbers (and also symbols) in binary representation. The binary system was developed by G.W. Leibniz when studying Chinese letters. Based on the binary representation, also octal (3 bits) or hexa-decimal (8 bits) representations are used in the computer.

Example – if you really want to know:
The number 496 can be represented in the following ways (where the subscript indicates the type of representation, and the superscipt are mathematical powers):

$$\begin{aligned}
496_{10} &= 4\cdot 10^2 + 9\cdot 10^1 + 6\cdot 10^0 \\
&= 1\cdot 2^8 + 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 \\
&= 111110000_2 \\
&= 7\cdot 8^2 + 6\cdot 8^1 + 0\cdot 8^0 \\
&= 760_8 = \texttt{0760} \quad \text{(in C++)} \\
&= 1\cdot 16^2 + 15\cdot 16^1 + 0\cdot 16^0 \\
&= 1F0_{16} = \texttt{0x1F0} \quad \text{(in C++)}
\end{aligned} \tag{15.1}$$

The number 5 is much simpler:

$$\begin{aligned}
5_{10} &= 5\cdot 10^0 \\
&= 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 \\
&= 000000101_2 \\
&= 0\cdot 8^1 + 5\cdot 8^0 \\
&= 5_8 = \texttt{0005} \quad \text{(in C++)} \\
&= 0\cdot 16^1 + 5\cdot 16^0 \\
&= 005_{16} = \texttt{0x005} \quad \text{(in C++)}
\end{aligned} \tag{15.2}$$

Numbers x in floating-point representation (`0.496e+03`) are transformed to fixed-point notation using the formula

$$x = mb^e \tag{15.3}$$

with the "mantissa" m (here `496`), the basis b ($b = 10$ could be also $b = 2, 8, 10, 16$), and the exponent e (here `3`). The in-fact implementation of floating-point numbers is machine/processor-dependent and will not be discussed here.

15.2.2 Basic data-types and their declaration

The most basic built-in data-types in C++ are: `bool, char, int, float, double`. Historically the processors were optimised to deal with those – however, the development of processors has progressed so much that also other data-types are efficiently treated. Usually, a compiler can optimise a code such that a user does not feel anymore the difference between different types of data.

The type `bool` is a logical data-type that can be used, e.g. to deal with the result of a logical comparison using logical operators, see Sec. (15.3). It can only have two values `true` or `false`. Furthermore, its freely transformable to data-type `int`, where `true` corresponds to 1 and `false` to 0. This transformation was introduced for compatibility reasons, to old programs where instead of `bool` just `int`'s were used. Some simple declarations are:

```
bool     test;     // declare test
test = (0 < 5); // comparison with result true; test is set to true
int      i(test); // i is declared and set to 1
int      j=0;     // j is declared and set to 0
```

Initialisation/Construction of a variable can be performed as *type variable = type_constant* or equivalently as *type variable(type_constant)*. (*For experts:* The equal symbol '=' here is not the setting of the variable, since this was just created; technically, here is *not* the set-to Operator '=' called, but a copy-operator.)

The type `int` is that type that is recognised by the compiler (i.e. the compiler-programmer) as most appropriate for dealing with integer numbers. Its size is not fixed in the standard. However, with help of the operator `sizeof`, one can determine the size of a type `int` in units of a data-type `char`, which is also the smallest data-type (again, selected by the compiler as most appropriate for single letters).

Different computers also support further integer data-types, e.g., 16, 32, or 64 bits large. These are defined using `short` and `long` or (not ANSI conform) `long long`. The longer form of declaration `short int` and `long int` is not required. Note that an integer variable always has an integer value – even if one assigns a real (floating-point) value to it.

Examples for `int`eger declarations are:

```
bool     test;                    // declare test
test = (0 < 5);                   // (see above)
int      i(test);                 // i is declared and set to 1
int      j=0;                     // j is declared and set to 0
long     l, n, m;                 // l, n, and m are declared "long"
short    s1, s2;                  // s1 and s2 are declared "short"

cout << sizeof(test) << " "
     << sizeof(i) << " "
     << sizeof(l) << " "
     << sizeof(s1) << endl;       // check size of: test, i, l, s1
```

```
i = 37;
j = 38.5674;                          // ATTENTION
cout << i << " " << j << " " << i/j << endl;
                                      // this leads to output: 37 38 0
```

The data-types `float` and `double` are the single- and double-precision floating point data-types that are implemented optimally in the processor. The type-specifier `long` can be used to modify `double` in order to get quad-precision (if supported). The in-fact representation of floating-point numbers is hardware dependent, e.g., in 1998 at least the double-precision basic operations were hard-ware implemented. Nowadays, with 64-bit processors more and faster operations with larger numbers are possible. Type `float` should only be used if memory is a problem, since for the (processor) operations, often `float` has to be modified to `double`.

C/C++ does not specify how variables are represented in memory and the use of memory is not specified per type either. However, at least the relations:

`1 = sizeof(char) = sizeof(bool)` $\leq$ `sizeof(short)` $\leq$ `sizeof(int)`
$=$ `sizeof(unsigned)` $\leq$ `sizeof(long)`

and

`sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)`

are always true.

In general, on UNIX-machines, `int`-variables are 32, `short` 16, and `char` 8 bit long. Double-precision `double` requires 64 bit, and `float` only 32 bit. The values that are used are different from machine to machine, but also can be different from compiler to compiler.

One can check the range of the basic data type using the `numeric_limits` library:

```
#include <iostream>
#include <limits>
using namespace std;

int main () {
  cout << "Minimum int: " << numeric_limits<int>::min() << endl;
  cout << "Maximum int: " << numeric_limits<int>::max() << endl;
  cout << "Minimum double: " << numeric_limits<double>::min() << endl;
  cout << "Maximum double: " << numeric_limits<double>::max() << endl;
  // ..
}
```

In the following example, different data-types are declared:

```
#include <string>

int main () {
    char       c1 = 'x';          // one character: 'x'
    char      c2[] = "hello";     // text array with 5 characters
                                  // (note: length is 6 due to added '\0')
    std::string str("hello");     // the C++ way, arbitrary length strings
                                  // (note: length is 5 - no added '\0')
```

```
    char       c3 = '\n';       // the new-line character
    int        i_var;           // integer variable with name i_var
    long int   il_var = 1;      // long constant
    long int   jl_var = 1L;     // another long constant
    short      is_var;          // short integer variable (int per default)
    double     d_var = 34.2;    // real number 34.2 with double precision
    // ..
}
```

Note that for `char` variables single characters can be enclosed by two apostrophes, e.g. `'x'`, where as longer chains of characters must be enclosed by a double apostrophe, e.g. `"hello"`. Above, the variables `c1` and `c3` both have length unity (1), whereas the chain `"hello"` is accessible in the "field" `c2` and has length 6, since during its declaration, automatically, a symbol `'\0'` is added. (*Try the definition* `char c2 = 'hello';` *that will lead to an error.*) More flexible is the C++ method to use `string`, which do not need the terminator `'\0'`. Note that for strings, the standard-library `string` must be included, see above.

For a computer there are three types of single quotes, left quotes ‘, right quotes ’ and straight quotes '. Characters must be enclosed by two straight quotes, or your code will not compile. Some operating systems try and be clever and insert ‘ the first time you ask for a quote and ’ the second time. If you get this problem turn off sticky keys on your operating system and it should disappear.

Declaration of variables is not limited to the beginning of a block, but can be done everywhere at first occurrence of a variable – be it clear, useful, and efficient or not.

Important and useful is the declaration within a `for(;;)` loop, as discussed in more detail later:

```
double a;
cin >> a;                 // input a via keyboard
int    i_trunc = a;       // throw away the fractional part
for ( int i=0; i < i_trunc; i++ )
{
// ...
}
```

In the above program, the variable `i` does not exist outside of the `for(;;)` loop, neither before nor after.

Summary
(i) Declarations can be located at arbitrary position in the program
(ii) Initialisation with `()` or `=` operator
(iii) `sizeof(type)` can be used if the size of a type must be known, e.g., for portability between different computers/architectures.

15.2.3 Type conversion

It is possible it change one variable into another type

```
int p=2, q=3;
double r=p/(double)q;
```

Here this is necessary to make sure rounding does not take place in the answer of r.

Two very useful ones are for converting char arrays to double and integers; the commands are `atof` and `atoi` respectively.

```
const char* str_int = "777";
const char* str_float = "333.3";
int i = atoi(str_int);
float f = atof(str_float);
```

Please note in C++ this can be done using `stringstream`'s and this method is recommended nowadays, see §15.2.6.

15.2.4 Variable-attributes

Besides the attributes `short` or `long` integer variables can be also modified by `signed` and `unsigned`, that allow to use integers (positive or negative) or exclusively positive integers, respectively. The `int` can be dropped here.

Exercise 15.1.
Write a problem to find out the size of variables of different types on your computer. ■

15.2.5 string type

String objects are a special type of container, specifically designed to operate with sequences of characters. Each character of the string can be access using the `[]` operator i.e.

```
string a;
a="Hello";
string b;
b=a[2];
cout << b <<endl;
```

would write out the letter 'l' , recall c++ arrays start numbering at 0.

Additional with the + operator can be used to concatenate two strings together; for example

```
string a,b,c;
a="Hello ";
b="World";
c=a+b;
```

```
cout << c << endl;
```

would produce 'Hello World'.

Note: The function `c_str()` returns a const pointer to a regular C string, identical to the current string. The returned string is null-terminated. This is often required because a lot of routies require C strings not C++ strings to be passed to them.

15.2.6 String stream

`stringstream`'s provide an interface to manipulate strings as if they were input/output streams. They are declared in the header called *sstream*, as you would expect from the other stream names, but the type is called `stringstream`.

This is described here, after string for ease of finding, but iostreams are described in more detail in chapter 21, so please refer to this section for a more detailed explanation of streams.

Below is a small example code that creates a filename by passing information into a stringstream and re-extract the string at the end.

```
#include<sstream>
#include<fstream>
#include<iostream>
using namespace std;

int main()
{
    stringstream file_name;
    stringstream problem_name("my_problem");
    ofstream script_file;
    file_name << problem_name.str() <<".disp";
    script_file.open((file_name.str()).c_str());
    // ..
    script_file.close();
    return 0;
}
```

The key commands are `<<` which adds data to the stringstream this can be any type for which the `<<` operator is defined. The `.str()` method return a traditional C++ string, see section 15.2.5 were `.c_str()` is also described.

The nice thing about stringstreams is they automatically deal with type conversion. For example the code below will convert a string to int via a stringstream.

```
#include <iostream>
#include <sstream>
```

```
using namespace std;
int main()
{
    int a;
    string s = "456";
    istringstream sin(s);
    sin >> a;
    cout << a << endl;
    return 0;
}
```

15.2.7 Life-time and position in memory

`auto` If there is no further declaration, variables are `auto`matic. These variables have a finite lifetime, or *scope*. This means they are assigned to memory as soon as the program reaches the line where they are declared. As soon as the program leaves the scope where they are declared, the memory is freed again. (The scope is, in general, the inner-most enclosing pair of brackets, or like in the case of a `for(;;)` loop, the end of this object.)

For experts: Variables of this type are usually assigned by the compiler to the so-called stack memory. The value of such a variable is random (!), if not explicitly assigned or initialised. Thus *make sure to initialise* your variables!

Aside: stacks, heaps, etc.
In computer science, a **stack** *is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterised by only two fundamental operations, the push and the pop. The push operation adds to the top of the list, hiding any items already on the stack, or initialising the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list. (See Wikipedia for further reading).*
A **queue** *is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. (See Wikipedia for further reading).*
In computer science, a **heap** *is a specialised tree-based data structure that satisfies the heap property: if B is a child node of A, then key(A) ≤ key(B). This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a max-heap. (Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a min-heap.) The several variants of heaps are the prototypical most efficient implementations of the abstract data type priority queues. Priority queues are useful in many applications. In particular, heaps are crucial in several efficient graph algorithms. (See Wikipedia for further reading).*

`static` The keyword `static` makes a variable exist during the whole duration of the program. *For experts: Such variables are saved on the heap-memory.* If not initialised otherwise, they get the value 0 (zero). A `static` variable inside a function has thus the value 0, if the function is called for the first time, and keep this (or another assigned value) between subsequent calls to

this function. In contrast, an `automatic` variable is every time newly created and initialised. The following function counts how often it was called and returns this value.

```
int f_count_calls()
{
  static int count;   // 0 at first function call
  return ++count;     // increment before use as return value
                      // (i.e. returns one on the first call)
}
```

`const` Variables that are declared `const`, can only be assigned a value at initialisation and not anymore thereafter. This is used to define constants within a namespace that – by the compiler – can be optimised such that their use costs less time than using variables. Thus, if variables can be defined as `const`ants, this might lead to a faster program execution.

```
const int    Size = 100;
const double Pi   = 3.1415926;   // Self-defined value of PI
// When including cmath one can use the predefined constant M_PI
```

15.2.8 Fields

Fields are used to combine variables of the same type in a connected range of memory. *For experts: Access to variables can be faster if they are in such a connected range of memory and, on the other hand, using fields also can make a program shorter and more clear.* Access to the elements of a field is achieved by an integer index. A linear field (with one index) can be seen as a vector, while a field with two indices represents a matrix. Mathematical formulas with indices can often be represented in a program using fields.

A field is declared in C/C++ by adding brackets `[]` to the variable name. At the same time, this operator is also used to access the elements of the field, see below. In C/C++ fields always start with the index 0, so that the last valid index is the length of the field reduced by 1. Multi-dimensional fields are declared by multiple (not enfolded) pairs of brackets, see below.

Such fields receive – at declaration – a fixed size. Thus, the argument in brackets `[]`, must be either a number, a `const` variable, or a well-defined variable with a known value for the compiler (at compilation time).

```
#include <string>

int main()
{
  // array of 20 int's
  int  a[20];
  // initialisation of field a -- ERROR!
  for ( int i=0; i <= 20; i++ )
      a[i] = i;
```

```
    // initialisation of field a
    for ( int i=0; i < 20; i++ )
        a[i] = i;

    // array of three strings, initialised to some values:
    std::string  stra2[] = { "Katia", "Holger", "Stefan" };

    // two dimensional array of 20x30 elements of int type
    int  ia[20][30];
    // initialization of field ia
    for ( int i=0; i < 20; i++ )
      for ( int j=0; j < 30; j++ )
        ia[i][j] = i*j;
    return 0;
}
```

Later, we will see that fields and pointers are closely related. The use of pointers allows to define "dynamic" fields of a size that is determined during run-time, e.g., following a user-decision and -input.

C++ supports containers that can dynamically allocate memory and allows simple manipulation of fields, such as `vector` or `valarray`. The C++ vector-container is discussed in more detail 20.5 . Nowadays the use of C++ vector over fields is often recommended; however, non careful use can lead to slow code.

15.3 Operators

Operators allow us to make a program readable. Sometimes, unfortunately, they also make a program un-readable, cryptic and unclear. A term like `3 * z + 5` is easily understandable, whereas the term `plus(mult(3,z),5)` that is expressed using functions is much less clear.

There exist unary, binary, and ternary operators. Binary operators like the multiplication- and addition-operators, have two arguments, while unary operators, like the address-operator `&` require only one. Finally, ternary operators take three arguments, the discussion of these is delayed until § 16.2.

Table 15.1 gives an overview of all operators in C++. We have already been using assignment and arithmetic operator and next we will discuss these in more detail.

15.3.1 Arithmetic operators

The arithmetic operators are `+, -, *, /` and `%`. The Operator `%` (modulo) forms the integer rest of the division of the left with the right operand (both integer). If both are positive, the result is positive and strictly smaller than the right operand. For one or two negative operands, the result depends on implementation.

Combining the above operators with the '=' allows to repeat the left operand, as for example:

```
int i=3, j=7;       // initialise
i    = 3 + j % i;   // i = 3 + (j mod i),    i is set to 4
i   += 3 * 4 + j;   // i = i + (3 * 4 + j), i is set to 23
i   /= 5;           // integer division, i is now 4
i   /= 5;           // integer division, i is now 0 !
```

The type that is returned by an operator expression depends on the type of the operand. If both operands are type `T`, also the result is type `T`. For different operand-types the "better" type wins; one operand is first transformed to the "better" type, then the operation is carried out. C/C++ has the following "type promotions":

```
bool -> char -> short -> int -> long
             -> float -> double -> long double
signed -> unsigned  (!)
```

These rule of type-transformation are also valid for the following operators.

The unary operators `++` and `--` exist in postfix and prefix-form, and they lead to an increment or decrement by one, respectively. In the postfix form, the value of the operand *before* the operation is returned, whereas in the pre-fix form, the value *after* is returned.

```
int i=5, j;
j = i++; // j is 5 now (i before increment) i is incremented to 6

double a[20];
i=j;                              // i=j=5
cout << a[++j] << a[j++];   // prints a[6] twice,    j is now 7
cout << a[i++] << a[++i];   // prints a[5] and a[7], i is now 7

// j++ = i;      // ERROR: cannot assign to a temporary
// i = (j++)++; // ERROR: cannot apply ++ to a temporary
```

15.3.2 Comparison and logical operators

For operations between two integral datatypes, there exist bitwise, logical, and shift operators. (C++ also provides keywords for these). These operators are summarised in table 15.3, and they also can be combined with the `=`.

The logical operators are `&&` (`and`) and `||` (`or`) and the negation `!` (`not`) . The operators `&&` and `||` have the special property that the evaluation of this expression is terminated already if the result is clear (so-called sequencing). This property is also shared by the comma-Operator `,` – that sometimes is used to construct complicated `for`-loops. Its value is the same as the value of the right expression.

`|`	`bitor`	bitwise OR
`&`	`bitand`	bitwise AND
`^`	`xor`	bitwise XOR
`<<`		left-shift, n-times
`>>`		right-shift, n-times
`~`	`compl`	complement, bitwise NOT, unary
`|=`	`or_eq`	bitwise OR, to-left assignment
`&=`	`and_eq`	bitwise AND, with assignment
`^=`	`xor_eq`	bitwise XOR, with assignment
`<<=`		left-shift, n-times with assignment
`>>=`		right-shift, n-times with assignment

Table 15.3: Bit-manipulation-operators

```
int a=5;
int b=10;
int c=5;

bool isittrue;

isittrue=((a==c) && (a!=b)) // This is true

isittrue=((a>b) || (b>c)) // This is also true
```

The (arithmetic) comparison-operators are `==`, `!=`, `<`, `<=`, `>` and `>=`. The value of a logical operation and the result of a comparison in C++ are of type `bool`, i.e. either `true` or `false`.

15.3.3 Further operators

Here all other operators except the ternary operator, `?:`, which is used to select alternative expressions of the same type, will be discussed. See sec. 16.2 for this operator.

```
int i;
cout << (i > 0) ? i : 0;      // never prints a negative number
```

The operator `()` leads to a function-call, `[]` is the index-operator for fields and pointers, a point `'.'` allows to select an element from a structure and `->` is equivalent when a pointer to a structure is given.

```
class Person {
public:
  string name;
  int age;
};

void f(){
```

```
    Person     p;
    Person *p_p = &p;
    p.age      = 25;
    p_p->age = 25;  // equivalent
}
```

The unary operators `*` and `&` are used to find the value of the memory location where a pointer points at, and to find an address of an arbitrary variable (see sec. 19).

Operator-expressions are (like in C) treated according to their priority and associativity, see table 15.4.

15.4 Summary

So far we have introduced the basic element of a program: variables, and discussed the many different types that are available. Also a few other commands have been used with out explanation, (but you can probably work out what they do) like `cout`, `cin` and `for`.

In the rest of this part, we will cover how to make decisions in a program, repeat sections of program, interact with the users and how to store data in a permanent way.

Exercise 15.2.
Explain what each line of the following code does i.e. add the comments

```
#include <iostream>
using namespace std;

int main()
{
    int n, m;
    cout << "Enter two integers: \n";
    cin >> n >> m;

    int sum=m+n;

    cout << "Sum of the two integers is " << sum << endl;

    return 0;
}
```

■

Exercise 15.3.
Modify the program to ask for three numbers ■

Exercise 15.4.

Modify your program to ask for both a person first name and surname. Get the program to output the persons initials, along with the sum of the three numbers they give. ■

P.	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`++ --`	Suffix/postfix increment and decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
3	`++ --`	Prefix increment and decrement	Right-to-left
	`+ ?`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(type)`	Type cast	
	`*`	Value-of (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	
	`new, new[]`	Dynamic memory allocation	
	`delete, delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer to member	Left-to-right
5	`* / %`	Multiplication, division, and remainder	
6	`+ ?`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`< <=`	For relational operators $<$ and $\leq$ respectively	
	`> >=`	For relational operators $>$ and $\geq$ respectively	
9	`== !=`	For relational $=$ and $\neq$ respectively	
10	`&`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	
15	`?:`	Ternary conditional	Right-to-left
	`=`	Direct ass. (provided by default for C++ classes)	
	`+= ?=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
16	`throw`	Throw operator (for exceptions)	
17	`,`	Comma	Left-to-right

Table 15.4: Precedence (P.) and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Chapter 16

Decision structures

For implementing algorithms into a certain programming language, this language has to provide certain control-structures that allow repeated or conditional execution of parts of the program.

In C (and C++) the most-used control strucures are the `if`-statement for the conditional execution, as well as the `switch()...case:`-statement, that jumps to one of many blocks of commands, based on the value of a certain variable. Finally, there is the command `goto`, that sometimes can be used to escape from (deeply nested) loops, see §17. (Actually I would not recommend ever using a goto, deaply nested loops can be exited by making the outer loop a function and using the `return` command, see §18).

16.1 The if-command

The `if`-command is used to execute parts of the program only under certain conditions. This command can be (but does not have to be) complemented by an `else`-statement, that is visited/exectuted in case the condition in the `if`-command is `false`. The `else`-part can consist of arbitrarily many `else if` and one `else`. As well `if`, `else if`, and `else`-branches can be single lines (terminated by a semicolon) or blocks, enclosed in curly brackets `{}`. An `else`-block always relates to the previous, actual `if`-command, i.e., also if it is part of an `else if`. In the following example, the second `if`-command is a statement by its own, outside the `else`-part of the first `if`.

```
#include<iostream>
#include<cmath>
using namespace std;

int main()
{
```

```
    // solution of quadratic equations  x^2 + p x + q = 0

    double p=2., q=3.;                  // or other values
    double sol1, sol2;
    double root_arg = 0.25*p*p - q; // calculate the argument for the root

    if ( root_arg > 0. )
    {                                   // begin of if-block
        sol1 = 0.5*p + sqrt(root_arg);
        sol2 = 0.5*p - sqrt(root_arg);
    }                                   // end if-block
    else if ( root_arg < 0. )           // 'error' condition, 'else' part
        cout << "no real solution exists" << endl;
    else                                // refers to last if(root_arg < 0.)
        sol1 = sol2 = 0.5 * p;           // assignment from right to left

    cout << "Solutions 1,2: " << sol1 << ' ' << sol2 << endl;

    if ( sol1 > 0. )                    // ok, else branch missing
        cout << "graph cuts positive x-axis";
    return 0;
}
```

16.2 The ternary ?: operator

For completeness, also the ternary ?: operator is treated here. Ternary means that this operator requires three arguments and is used to conditionally evaluate parts of expressions.

```
inline int abs(int number) {
    // compute absolute value
    return  number > 0  ?  number  :  -number ;
    // Equivalent form with brackets
    //return (number > 0) ? (number) : (-number);
}
```

Is the term before the question-mark true, then the term before the colon is examined. Otherwise, the term after the colon will be used. The two alternatives must be of the same type. Note that here, the use of brackets is recommended – partly because the ternary operator is of low priority and also because the terms that belong together should be made clear.

```
int i1   =       3 > 4 ? 0 : 1;     //  i1   is 1
int i2   = 3 * ( 3 > 4 ? 0 : 1 );   //  i2   is 3
int i3   = 3 *   3 > 4 ? 0 : 1;     //  i3   is 0, since * binds
                                    //    stronger than >
```

This operator always can be avoided by using an `if`-command, however, it could be more clear or at least shorter in some situations. In any case it is a possibility to write un-readable code and therefore should be used sparsely.

(As final remark, the ternary operator is also embedded in the graphics program gnuplot.)

16.3 switch-command

A command related to the `if` and the `goto` commands is the `switch`-command. Dependent on the value of an integer variable, several `case`-blocks are visited. This construct is often used when there is a larger number of possible alternatives to be chosen from.

In principle, a `switch` can always be replaced by `if...else if`-constructs, but often the `switch` is more clear and helps understanding and structuring the program.

```
char c;
cin >> c;
switch( c )
{                                   // begin case block
case 'a': cout << "hello world!";
          break;
case 'b': cout << "HELLO ";       // fallthrough intended, comment missing brea
case 'c': cout << "WORLD!";
          break;
default:  cout << "unknown input";
          break;                    // not necessary, but good style
}                                   // end case block
```

The above program-segment returns in small letters `"hello world!"` if the letter `'a'` is typed. In case `c` the word `WORLD!` is printed, but in case `b` the full sequence `HELLO WORLD!`. This strange behaviour comes from the missing `break` in the block `b`. This behaviour, namely that a execution is performed until a `break`, is called ("fallthrough"). Since this is often not desired, using this option required a comment whenever a `break` is missing. The `default`-label ist optional and is visited if none of the other labels is valid in the `case`-list.

Exercises

Exercise 16.1.
Modify the program from Exercise 15.4, so that it asks the users if he/she wants to enter 2 or 3 numbers. Read in the correct number of inputs and compute the sum. ■

Exercise 16.2.
Write a program that reads in a date in the format dd/mm/yyyy format and prints out 12th of March 2010 etc... ■

Chapter 17

Loop control structures

Often you want to repeat sections of a program, the most common way is using `for`-loops. Alternatively, the rejecting `while()`- and the accepting `do{...}while()`-loops exist.

17.1 (do) while-loops

Loops that are constructed using `while` are executed until the condition in the condition part of the loop is not true anymore. Is the condition already false at first occurence of the `while`-loop, the loop will not be executed at all.

```
char c=0;                          // 'loop initialisation'
while( c != 'y' && c != 'n' )      // first time c==0, so we enter the loop
cin >> c;                          // can also be a block { }
```

The symbol/character variable has to be initialised such that the loop is entered. In such cases, the `do ... while(...)` loop might be more intuitive.

```
char c;
do {
    cin >> c;
} while( c != 'y' && c != 'n' );
```

17.2 for-loops

When working with fields, one often needs loops that access all elements of a field, one by one. One can program this with `while`:

```
const int Size = 20;
// ..
int a[Size];
// ..
int i=0;                  // loop initialisation
while ( i < Size )        // loop condition, rejecting if false
{
    a[i] = i;
    i++;                    // loop control, performed after each pass
}
```

A shorter alternative used the `for(;;)` loop. The following is almost equivalent to the above, but shorter by avoiding the line with `i`, and more clear, since the begin, end, and stepsize of the loop are defined at its start.

```
int a[20];
// ..
for( int i=0; i < 20; i++ )
    a[i] = i;              // work on elements 0 .. 19
```

There are two advantages: First, the loop-beginning, -ending, and stepping is nicely combined in one line and, second, the variable `i` is only defined within the loop and not used (and not needed) outside.

17.3 break and continue

In order to exit a single, not-nested `for` or `while` loop, one can use the `break` command. The loop is terminated and the program continues directly after the loop. Thus `break` can be used to terminate loops under certain conditions, earlier than the loop implies, or it can be used to exit "endless" loops.

```
// copy input to output
while( true )
{
  char c;
  cin >> c;
  if ( cin.fail() ) break;    // end of input or something wrong
  cout << c;
}
```

The command `continue` terminates the processing of the present block and continues, dependent on the type of the loop, at the corresponding control-command (`for(;;)`), or (`(do{}) while()`).

```
#include <cmath>                    // for sqrt() function

double a[20];
// ...
double sum_sqrt=0.;
for( int i=0; i < 20; i++ )
{
  if ( a[i] < 0. ) continue;    // work on next element
  sum_sqrt += sqrt( a[i] );
}
```

Note again that `continue` and `break` only exit the innermost loop-block and cannot be used to exit nested loops.

Exercises

Exercise 17.1.
Write a code which reads in N numbers and computes their sum. ■

Exercise 17.2.
Write a program to compute the squares of all integers between two numbers. ■

Exercise 17.3.
Write a program that computes the factorial of an inputed number N. Recall the factorial of $N = N * (N-1) * (N-2) * ... * 1$. ■

Chapter 18

Functions and Operators

Longer programs usually contain parts or blocks that are repeated a certain (large) number of times. These parts can then be defined elsewehere (as a function) and within the program are replaced by a short-cut or place-holder, i.e., the name of the function. It is wise to give the `function` a name that already implies its function.

A function (also called procedure or subroutine in other languages), has input and output variables/parameters. In the simplest case the transfer of input-parameters is done as argument, and the output as a return-value.

C++ enforces that the types of input and output are specified. In the following example, the function `power` computes from a double `base` value and an integer number `exponent` a new number in double precision that is returned to the calling function/program. The name already implies the algorithm used here.

```
// raise 'base' to the power 'exponent', with exponent being integer,
// base being double; accept also negative exponents and check ...
double power(double base, int exponent){
   double result  = 1.; // will multiply this base 'exponent' times
   bool   neg_exp = false;
   if (exponent < 0 )   // for negative exponents we use a single
   {                    // division at the end of the function
      neg_exp = true;
      exponent = -exponent;
   }
   for (int i=0; i < exponent; i++ )
      result *= base;   // successive multiplication

   if ( neg_exp )       // have to take the inverse
      result = 1./result;
```

```
    return result;        // return the result to the calling program
}
```

Exercise 18.1.
Search for the explanation of **Flow-Chart** in, e.g., Wikipedia, and plot the Flow-Chart of the Power-Function. ■

Note that C++ already has a function `pow` that is described by
`double pow(double base, double exponent)`,
only different in the type of the exponent.

The names of the arguments are free to be chosen. Inside the function block these names are to be used. Note that like `for` loops, there is no semicolon after the function.

The keyword `return` terminates the function at the specified position, moves the value of the variable specified after it on the stack, and returns to the calling program. In the case above, the expression consists only of a single variable `result`. Round brackets are not required here. A subroutine can contain several return keywords at different places in the function block. This can be a useful tool to deal with problems/errors. Hence the `return` command can be used to exit deeply nested `for` loop to forcing the exit of the whole function.

Finally, one has to define or declare a function before it can be used in the program. This declaration is essentially the first line of the function (with semicolon). This so-called prototype tells the compiler the name of the function and the types of it and its arguments. Actually, if you careful about the order you do not need to prototype functions, but you must place the full function definition before the first call (use) of the function. This is often tricky to keep track of and prototyping is recommended.

```
double power(double base, int exponent);
```

At first definition, the command block is replaced by the single semicolon. Note that the type is required, but the name of the arguments in the function is not needed here. However, a clear program specifies variables that tell something about their purpose.

Declarations can (in principle) be repeated arbitrarily often in a program – they are a promise to the compiler that the function will be defined later. Unlike variables, functions are always global – it is not allowed to define them inside a block with limited existence. Actually function and varibles can be wrapped together in a very neat package, called a class, see §**??** for more details, solving this global scope problem.

A program that reads a number from the keyboard and then uses the function `power` to finally write the result could look like this.

```
#include <iostream>
using namespace std;

double power(double base, int exponent);  // declaration of 'power'
```

```
int main()                              // definition of main
{
  double in;
  cin >> in;                            // read value from keyboard

  cout << "-3rd to 3rd power of " << in << ": ";
  for( int i= -3; i <= 3; i++ ){
    cout << power(in, i) <<  " "; // call power and use return value
  }
  cout <<  "\n";

  return 0;                             // return to operating system
}

double power(double base, int exponent)   // definition of power
{
  double result = 1.;
  // ... program text as above

  return result;
}
```

18.1 Transfer of arguments

The transfer of arguments to a function takes place "by value", i.e., before the function is called, the arguments are copied and the function uses these copies. This way, the variables that contain the values that are transferred to the function can not be changed. If this is desired, e.g., because one return value is not enough and many variables are to be changed in a function, then one either use pointers, as disucssed later in section 19 or pass by reference.

If you add `&` after a variable name in the declaration the actually variable, not a copy, is passed to the function, therefore changes to that variable within the function effect the original variable.

18.1.1 Example of passing by reference

```
#include <iostream>
#include <valarray>

using namespace std;

int add(int a, int b);

int add2(int &a, int &b);
```

```
int main()
{
    int a=1;
    int b=2;
    cout << a <<" "<<b << endl;
    cout << add(a,b) << endl;
    cout << a << " " <<b <<endl;

    cout << a << " " << b <<endl;
    cout << add2(a,b)<<endl;
    cout << a << " " << b << endl;
    return 0;
}

int add(int a, int b)
{
    a=a+b;
    return a;
}

int add2(int &a, int &b)
{
    a=a+b;
    return a;
}
```

Exercise 18.2.
Run this program and from the output explain the difference between the functions `add` and `add2`. ■

18.2 Functions without arguments or return value – void

The keyword `void` can be used like a type, but is not really a type. It is useful to mark functions that do not have a return value. Examples are functions that just write something to the screen or to a file and functions that modify the input-variables directly.

```
void error(string message){
  cout << "error occured: " << message << "\n";
}
```

Also functions without arguments sometimes can be useful. In C++ this is implied by an empty list of arguments like for the pseudo-randomnumber generator as declared in the C library `stdlib.h`:

```
void srand(unsigned int seed);
int  rand(void);
```

It is used by first selecting a sequence of random numbers by calling (once only) the initialisation function `srand(.)` with a positive number. Then, for each call of `rand()` a new, integer number is returned. In order to transform the integers to real numbers, one can e.g. divide by the maximum.

Exercise 18.3.
Use the random number generator of C/C++ and later the random numbers in MATLAB to generate a vector of 10 random numbers $r_{i=1,\ldots,10} \in [0,1]$. ■

18.3 Libraries

As seen in previous examples, declarations/prototypes are sufficient for the compiler to translate a program. Wherever called in the program, the function call is prepared during compilation. The function itself can be then defined and "linked" to the program later. Function-definitions are found in so-called object-files (*.o) or libraries (see section §22 for more information of code organisation into small objects, which are compiled and linked independently).

The classical example for a library is the `cmath` file that was included at the beginning and involves functions like the `sqrt()`. The declarations are found in `cmath` itself, and the translated/compiled definitions are found in `libm.a`. A program-block that uses the `sqrt()` function can look like this.

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
  double  d;
  cout << "Enter a number:" << endl;
  cin >> d;

  if ( d<0 )
    cout << "Cannot compute the root of negative numbers" << endl;
  else // compute sqrt(d) and print value
    cout << "sqrt(" << d << ") = " << sqrt(d) << endl;
  return 0;
}
```

The compiler must contain the linker-option "-lm" and must know the correct path where it can find the library (note, some compilers do this automatically). As a remark, under LINUX, the compilation would look like this:

```
g++ myprog.cc -o myprog -lm
```

In DevC++ the corresponding command can be found under "Compilation".

The short `-l` means "library" and `m` is the abbreviation for the file `libm.a`, that one gets by adding `lib`, `m` and the ending `.a`.

18.4 inline functions

In general, several things happen when a function is either called or closed. Some of this can be avoided or optimised by use of so-called `inline`-functions. However, we will not go into detail here. (Further reading if needed).

18.5 Overloading of functions

In contrast to C and some other languages, in C++ not only the name and type of a function, but also the types of the arguments are relevant. Therefore it is possible to define a function for each type `int`, `double` or `struct Date {...}`, all with the same name. The name does not have to contain type information thus. In C++, the compiler selects the right function with the appropriate type by itself. One can think of an internal function-name used by the compiler that also contains the argument-types.

```
void swap(int &a, int &b)    // internal name e.g. swap_int_int
{
  int tmp = a;
  a = b; b = tmp;
}

void swap(double &a, double &b)     // swap_double_double
{
  double tmp = a;
  a = b; b = tmp;
}

void f()
{
  double a, b;
  int    i, j;
  // ...
  swap(a, b);  // double version
  swap(i, j);  // int    version
}
```

18.6 Templated functions

In C++ it is possible to right functions were the type of the variables is free.

```
template<class T>
T add(T a, T b)
{
  return a+b;
}
```

The following call to the functions `add<int>(3,4)` would automatically create the following code.

```
int add(int a, int b)
{
  return a+b;
}
```

That is the above templated function can be used for any type T, well, for which the operator + is defined.

Multiply template arguments can be used, for example:

```
template<class T1, class T2>
T2 add(T1 a, T2 b)
{
  return a+b;
}
```

The call `add<int,double>(a,b)` would add the double b to int a and return a double. That is `add<int,double>(2.3,4.3)` would return 6.3; whereas `add<double,int>(2.3,4.3)` would return 6.

Exercises

Exercise 18.4.
Write a program that reads in ten numbers, using the swap function above sort the numbers into ascending order and write out the order list. See `http://en.wikipedia.org/wiki/Bubble_sort` for more information. ■

Exercise 18.5.
Write a function, which calls itself, to calculate the N^{th} Fibonacci number. Recall $fib(N) = fib(N-1) + fib(N-2)$ and $fib(1) = 0$, $fib(2) = 1$. ■

Chapter 19

Pointers, pointer-field duality, and references

19.1 Pointers

From the viewpoint of the computer, every memory-cell in the (virtual) memory is uniquely marked by a certain number (address).

C/C++ allows to use these **adresses** in memory-space for programming purposes. Since it is allowed to self-define names for these address-pointers, the program can also become much more clear and readable. The declarations:

```
int     i=5;
double  a=12.34;
char*    c1="hello";    // declaration of an array of char
int     j=0;
```

create the memory-entries in Fig. 19.1 (left column). Only the memory-entries (the values of the variables) occupy space in the computer memory. The names of variables do not occupy space even if one would use instead of `i` the more meaningful name `counter`.

In general, there is no guarantee that sequentially defined varibales also occupy memory sequentially (except for fields). The variable `c1` of type `char` contains a pointer to another range of memory where the values of the field are stored. The definition of `*c1` reserves some range in memory that is large enough to store the word `"hello"`. The elements `'h'`, `'e'`, `'l'`, `'l'`, and `'o'` can also be accessed using the form `c1[0]`, `c1[1]`, `c1[2]`, `c1[3]`, and `c1[4]`. Note again that numbering in C/C++ always starts with `0`.

If one has to insert or delete a range of a sorted list (quickly and efficiently), then this is not possible using a linear field like the above `*c1`. A *linear field* is special in so far that all data are stored in subsequent positions in memory. As well inserting as removing entries from this

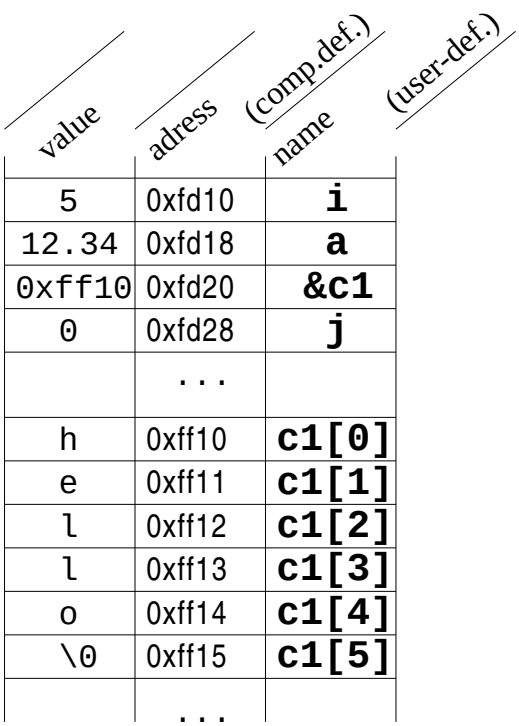


Figure 19.1: (Schematic) memory occupations (left), computer-internal memory adress (mid), and (user-defined) name of variables (right).

field requires copying the full range "above" the modified element. (In average that is half the field-length to be moved). This is acceptable for short fields but not for long ones. A possible solution to this is a *linked list* which contains not only the values of the elements but also the information where the next element can be found. The actual location does not matter, i.e., only the names of the memory-cells have to be modified. For example, in order to remove an element x from a linked list, only the name entry of the previous element has to be changed such that it does not point to x anymore, but to the following element. This information is available in the x-element and thus just has to be copied. Deleting several subsequent entries requires the same effort and entering a (part of a) linked list from somewhere else is also rather efficient. However, due to this additional options, a linked list also usually requires more memory than a normal field.

In order to allow a computer independent dealing with such objects like linked lists, C/C++ provides also the type pointer. A pointer can point to an arbitrary data-type – which actually could be a pointer itself. In order to declare a pointer, a (star) `*` is used. In order to get the address of a variable, the `&`-operator is used. The declarations in the program-fragment:

```
// Declaration part
  int      i = 5;          // initialisation
  int*   p_i = &i;         // p_i is a pointer to an int,
                           //   here the variable i
  int** pp_i = (&p_i);     // pointer to pointer to int

// How to use * and & legally in the program
```

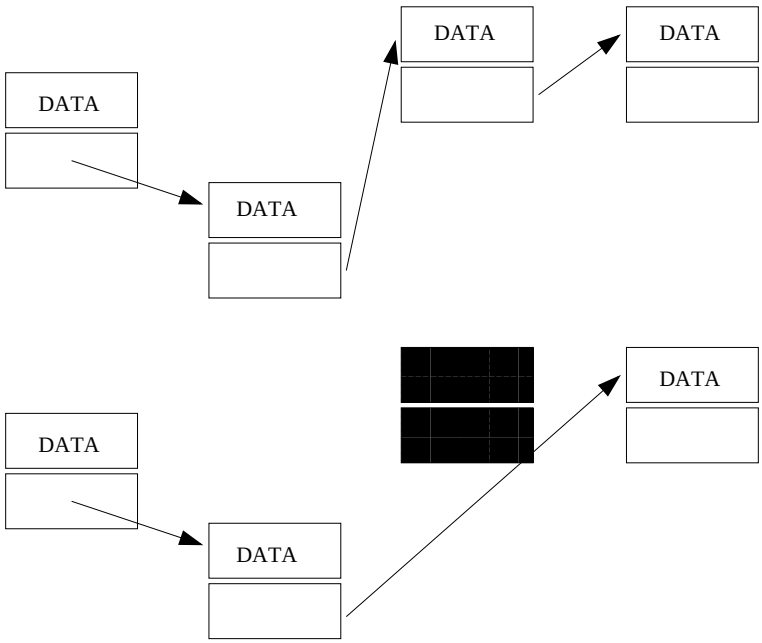


Figure 19.2: Use of pointers in a linked list. (Top) the list before an element is removed, (bottom) the list after, where the black element denotes the removed one that is not part of the list anymore.

```
    i = 0;          // set i to 0
  *p_i = 1;         // set i to 1, p_i remains unchanged
**pp_i = 2;         // set i to 2, p_i remains unchanged
 int j = 7;         // declare another integer j
 *pp_i = &j;        // change p_i to point on j, *p_i != i
   p_i = &i;        // let p_i point to i again
```

lead to memory occupation as indicated in Fig. 19.3.

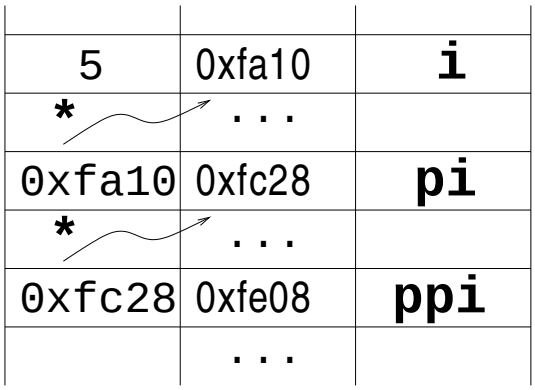


Figure 19.3: Memory occupation as in Fig. 19.1.

In order to change the value of a variable a pointer points to, or in order to use this value in an operation, also the * operator is used.

```
*p_i = 27;                  // i is 27 now
*p_i = 2 * (*p_i) + 5;      // equivalent to:  i = 2 * i + 5;
```

Constant pointers are declared by using a `const` right of the pointer-symbol `*`. They must be initialised already at declaration, since they are fixed thereafter and cannot be changed anymore.

```
const int *cp_i = &i;                       // ok, i cannot be changed
                                            //     through use of cp_i
                                            //     cp_i can be changed
const char newline = '\n';
const char * const p_nl = &newline;         // const pointer to const char
                                            // note: p_nl = '0'; -> error!
```

The `*` operator has thus two meanings: In a variable declaration, it marks a pointer, whereas in front of a variable (of type pointer) it leads to access on the memory (value) of the corresponding memory space.

Analogously, also the `&` operator has two meanings: During variable declaration, it denotes a reference, whereas in front of a variable, it is the so-called `address of` operator, i.e. it provides the address of the variable.

Note that it is possible to do arithmetics/mathematics with pointers. Above, `p_i = p_i + 5` would let the pointer point to 5 `int` memory entries further. For this, the compiler needs to know how big the corresponding data-type is. Also the difference of pointers can be formed but that makes only sense inside a field.

Navigation within a field can be done in the following way:

```
int p[30];              // declare a field of int
*(p_i + 25)  = 15;      // the value of the 25th entry (past the one
                        // pointed to by p_i) is set to 15
p_i[25]      = 15;      // equivalent ...
```

The language guarantees that 0 is never the address of a real data-element. A pointer with value 0 can thus be used to indicate an error or, e.g., the end of a list.

19.1.1 Pointer to void

The type pointer to void is generic. It is guaranteed that any pointer can be transferred into a pointer to `void`, without loss of information. In C the `void *` was often used to transfer information to functions/routines that can deal with arbitrary pointers. (e.g. sort-function like `quicksort`, or the return value of `malloc`). In C++ it is recommended to use `template`-functions that guarantee type-consistency and type-safety, see section 18.6 for more details.

19.2 pointer-field duality

Between pointers and fields exists a very close relation, see above the use of `[]` for the "indirect" de-referencing of pointers. A field `a[..]` is internally transformed to a pointer on the first element of the field. Access to the field is then dealt with via the pointer-arithmetics mentioned above. This is also the reason why C/C++-fields always start with 0 (i.e. the localisation of field entries can be done with less operations).

```
int a[20];
int* p = a;          // ok, a is name of the field and can
                     //     be used as address to the field
int* p = &(a[0]);    // ... equivalent

p[5] = 4;            // access to a through pointer arithmetic,
                     //     a[5] is set to the value of 4
a[5] = 4;            // ... equivalent

int b[20][30];       // b is now an array of 20 arrays of
                     //     30 elements of type int
                     //     int *p = b;  error! wrong type, since
                     //     b is of type pointer to 30 int here
int* r = b[0];       // ok, r points to the first int
                     //     in the first (of 20) array(s) of 30 ints
int (*q)[30] = b;    // ok, q points to the first set of 30 int's,
                     //     q and b can now be used synonymous
q[12][15]   = 26;    // ok, set element (13,16) to the value 26
(*(q+12))[15] = 26; // ... equivalent
*(*(q+12)+15) = 26; // ... equivalent
b[12][15]   = 26;    // ... equivalent
```

Exercise 19.1.
Practice the use of pointers, using the above examples as a starting point. ■

19.3 Transfer of field-variables to functions

In order to transfer a field to a function, it is sufficient to just transfer the pointer, see sec. 18. For a one-dimensional field this is just the pointer to the first element, while for two-d fields it is a pointer to a field with a compatible, fixed number of elements. Only the first dimension of the field can be left out in this case, all others have to be specified at compile-time. This makes the C-type fields rather in-practical to be used when dynamic memory management is desired.

```
void f( int b[][30] )    // function prototype that transfers b
void g( int (*q)[30] )   // function prototype that transfers

int  b[20][30];
```

```
int  c[10][30];
int  d[15][31];

f(b); g(b); f(c); g(c); // ok
f(d); g(d);             // errors, since second dimension not ==30
```

Summary: Transfer of Variables to Functions

As summary and examples for the use of fields in functions, the example in exercise 7 is re-written in order to use functions that use the field:

```
#include <cstdlib>
#include <iostream>
#include <cmath>
using namespace std;

int read_field( int flen, int *fi );
void write_field( int flen, int *fi );
void sort_field( int flen, int *fi );

int main(int argc, char *argv[])
{
   const int Mfmax=20;
   int fi[Mfmax];
   int ifmax=0;
   int ftmp;

   ifmax = read_field( ifmax, fi ); // get field from subroutine
   write_field( ifmax, fi );        // control what you got

   sort_field( ifmax, fi );         // sort the field
   write_field( ifmax, fi );        // control what you got

   system("PAUSE");
   return 0;
}

// function reads the field from another field and returns it back
int read_field( int flen, int *fi )
{
   const int field_length=17;
   int field[field_length] = {1, 0, -10, 12, -2, 5, 5, 1, -4, -2,
        -4, -8, 3, 25, 51, -4, -5};
   for( int i=0; i<field_length; i++)
   {
      fi[i]=field[i];
   }
   flen = field_length;
```

```
    return flen;
}

// function writes field to screen
void write_field( int flen, int *fi )
{
    for(int i=0; i<flen; i++)
        cout << fi[i] << " ";
    cout << endl;
    return;
}

// function sorts the field and returns it
void sort_field( int flen, int *fi )
{
    int ftmp;
    for(int i=0; i<flen; i++)
    {
        for(int j=flen-1; j>=1; j--)
        {
            if(fi[j]<fi[j-1])
            {
                ftmp=fi[j-1];
                fi[j-1]=fi[j];
                fi[j]=ftmp;
            }
        }
    }
}
```

Excerises

Exercise 19.2.
Consider the following code extract and explain the effect of each line, stating the value of a and b at all points.

```
int a=15;
int b=10;

int *p1=&a;
int *p2=&b;

int **p3=(&p1);

*p1=5;
*p2=20;
**p3=1;
```

```
*p2=*p1;
p1=p2;
*p1=10;
*p2=5
**p3=8;

cout << a << "\t" << b << endl;
```

■

19.4 Dynamical Memory Management

In order to request memory, C++ provides the operators `new` and `new []` for single variables or for fields, respectively. This memory is allocated without a scope (see section 15.2.7) and must be freed manually. In order to free the memory, `delete` and `delete []` have to be used.

```
int*    p_i  = new int; //declare new variable without scope
int        n = 40;
int*    pa_i = new int[n]; //declare new array without scope
// ...
delete   p_i;
delete[] pa_i;
```

The request for memory requires the type after `new`, and a pointer to this new element is returned. For fields, `new[]` is used and the number of memory-elements within the brackets is requested. The request returns a pointer to the first element in the field.

```
int* pa_i = new int[n];

pa_i[0]=5;
pa_i[1]=4;
```

The operator `new` returns an "exception" `bad_alloc`, if the request for memory does not succeed and then terminates. It is possible to 'catch' error and continue without ending the code, using the commands `try` and `catch` (this will not be covered by this course).

Note that in many compilers it is possible to allocate memory dynamically using VLA's (e.g., `int N=10; int a[N];` allocates a dynamic array, since N is not constant). However, we discourage you from using this, as it is not supported by the C++ standard (see `http://www.informit.com/guides/co` and can lead to segmentation faults when large dynamic arrays ($> 10^6$ bytes) are allocated.

Chapter 20

Object Oriented Programming

Object-oriented programming (OOP) is a programming style where objects - data structures that consist of both data fields and methods - are used to design computer programs. In C++ this is done using classes.

20.1 A simple OOP problem

A class can be used to declare structures which require both data storage and functions acting on that data. In the following example, a rectangular shape is defined. The shape of a rectangle is defined by its width and height; this is the data contained in the class. Further, there are many properties of a rectangle that can be evaluated from the data, for example the area or circumference of the rectangle; these class-specific functions can be implemented as part of the class.

```
#include <iostream>
using namespace std;

class Rectangle {
    double width, height;
public:
    Rectangle (double w, double h) {setValues(w, h);}
    void setValues (double, double);
    int getArea () {return (width*height);}
};

void Rectangle::setValues (double w, double h) {
    if (w>0 && h>0) {
        width = w;
```

```
        height = h;
    } else {
        cerr << "Error in Rectangle::setValues: " <<
            "height and width need to be positive" << endl;
        exit(-1);
    }
}

int main () {
    Rectangle rectA(3,4), rectB(5,6);
    cout << "rectA area: " << rectA.getArea() << endl;
    cout << "rectB area: " << rectB.getArea() << endl;
    return 0;
}
```

- The Class-Definitions appear before the `main()` program, which is especially short.

  Class definitions can be placed in separate files and this is normal the case for large projects. This is discussed further in section 22.

- A class is declared outside by the keyword `class` followed by the class name, the body in {}-parentheses, and a semicolon. It can be declared locally (i.e. inside a function), but is usually declared globally. The body contains the declaration of the member variables and member functions.

- A class like `Rectangle` is a data structure that can contain both **member variables** and **member functions**. In this case, the member variables are the height and width of the rectangle. The member functions are used to manipulate, read and write the member variables. In this case, `setValues` is used to set the dimensions of the rectangle (it also checks if the values make sense), getArea is used to evaluate its area.

- Members (variables and functions) are *declared* inside the class body. However, they can be *defined* both inside and outside the body of the class. If defined outside, the **scope operator** (`class_name::member_name`) is used to identify the function (see `setValues`).

- **Encapsulation by access specifiers**: By default members of a class are `private`, i.e., they cannot be called from outside the class (try using the variable `rectA.width` in the function `main`). You can make members public, i.e. accessible from outside the class by adding the line `public:`; all following members will then be accessible. Public members of a class are called using the dot operator (`object_name.member_name`).

- A **constructor** and a **destructor** can be defined for each class, which specifies what to do when an object of the class is created or destroyed. This is often used to set default values; the constructor is declared similar to a member function, with the same name as the class, but with no output type. A preceding tilde is used for the destructor. One can create constructors that take input variables, as in this example. Then an object of a class can be initialized with the `()` operator: `Rectangle rectA(3,4);`

20.2 Inheritance

Classes can also be inherited, which means that one can create a 'child' of a class which has all the properties of the 'parent' class, plus a few extra properties. For example, we can add colour to the definition of our rectangle:

```
#include <iostream>
#include <string>
using namespace std;

class Rectangle { /* ... */ };

class ColouredRectangle : public Rectangle {
    string Colour;
public:
    ColouredRectangle (double w, double h) : Rectangle (w, h) {
        Colour = "white";
    }
    void setColour (string c) {Colour=c;};
    string getColour () {return Colour;};
};

int main () {
    ColouredRectangle rectA(3,4), rectB(5,6);
    rectB.setColour("black");
    cout << "rectB colour: " << rectB.getColour() << endl;
    return 0;
}
```

- To inherit from an existing class, we use a colon `:` in the declaration of the derived class:

  ```
  class derived_class_name: public base_class_name { /*...*/ };
  ```

  The class has now all the member variables and functions of the parent class. Additional members are specified in the class body.

- We also use the colon if we want to inherit from the constructors or destructors of the parent class. In this example, the constructor of `ColouredRectangle` calls the constructor of Rectangle before initialising the `Colour` string.

20.3 Polymorphism

Inheritance is a relatively easy concept to understand. Polymorphism on the other hand is much harder. Polymorphism is about an object's ability to provide context when methods or operators are called on the object.

Definition of polymorphism: In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning to a particular symbol or "operator" in different contexts.

The simple example is two classes that inherit from a common parent and implement the same virtual method.

```
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;

class Shape {
public:
    virtual double getArea() {};
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle (double r) : radius(r) {};

    double getArea() {
        return M_PI*radius*radius;
    }
};

class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle (double w, double h) : width(w), height(h) {};

    double getArea() {
        return width * height;
    }
};


int main()
{
    Circle c(10);
    Rectangle r(10,5);

    vector<Shape*> ListOfShapes(2);
    ListOfShapes[0] = &r;
```

```
    ListOfShapes[1] = &c;

    for (int i=0; i<ListOfShapes.size(); i++)
        cout << "Area:" << ListOfShapes[i]->getArea() << endl;

    return 0;
}
```

In this example, we create two classes, one for rectangular and one for circular shapes. We want to be able to calculate the area for both shapes. Therefore, we create a base class `Shape` which does nothing but declare a function `getArea()`. Then we create the classes `Circle` and `Rectangle` by inheriting from `Shape`. Thus they inherit the `getArea()` function.

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. This can be used to create pointers which can point to any kind of shape and which allow access to the member variables and functions inherited from the parent class. In order for a parent class pointer to use a inherited function definition, we have to declare the parent member function as `virtual`.

Exercise 20.1.
For Experts: Think about what can be improved in the following code – for this you first will have to understand ... Further reading/learning required. ■

```
1  #include <stdlib.h>
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5  class CSample
6  {
7  public:
8      CSample(); // constructor
9      ~CSample(); // destructor
10
11     int Input(const char* fname);
12     double GetSum();
13     double GetMax();
14     int Size;
15 private:
16     int Nmax;
17     double *a;
18     bool mem_alloc;
19 };
20
21 int CSample::Input(const char* fname)
22 {
23     ifstream in(fname);
24     in >> Nmax;
25     Size = Nmax;
26
```

```
27      //memory allocation
28      a = new double[Nmax];
29
30      mem_alloc=true;
31
32      for (int i=0; i<Nmax; i++)
33          in >> a[i];
34      in.close();
35      return 1;
36  }
37
38  double CSample::GetSum()
39  {
40      double sum;
41      sum = 0;
42      for (int i=0; i<Nmax; i++)
43          sum = sum + a[i];
44      return sum;
45  }
46
47  double CSample::GetMax()
48  {
49      if (a==0) return 0;
50      double m;
51      m = a[0];
52      for (int i=1; i<Nmax; i++)
53          if (a[i] > m) m = a[i];
54      return m;
55  }
56
57  CSample::CSample()
58  {
59      mem_alloc = false;
60      a = 0;
61      Nmax = 0;
62  }
63
64
65  CSample::~CSample()
66  {
67      // free memory
68      if (mem_alloc) delete [] a;
69  }
70
71  //= MAIN PROGRAM ==========================
72  // Uses the class CSample
73  // to read in data, print the size of the data-field,
74  // compute the sum of, and the maximum.
75  int main()
76  {
```

```
77      CSample Test;
78      Test.Input("data.dat");
79      cout << "Size = " << Test.Size << "\n";
80      cout << "Sum = " << Test.GetSum() << "\n";
81      cout << "Max = " << Test.GetMax() << "\n";
82      cout << endl;
83      system("PAUSE");
84      return 0;
85  }
```

Exercise 20.2.
Below you see a simple class definition to store name of contacts. Create an inherited class that can also store the occupation of a person. ■

```
class Contact {
  //member variables
  string first_name;
  string last_name;
public:
  //member functions
  void set_name (string,string);
  string get_name ();
};

string Contact::get_name () {
  string name;
  name += first_name;
  name += " ";
  name += last_name;
  return name;
}

void Contact::set_name (string first, string last) {
  first_name = first;
  last_name = last;
}
```

20.4 C++ standard classes

All the C++ libraries contain objects and most of these are templated. We have been using these for some time without exactly understanding the syntax.

The first example we meet was `numeric_limits<int>::min()`. So here clearly it is templated with a type, in this example `int`, and we call the static member function `min()`. Note, `::` calls static member function, whose purpose we have not covered in the discussion above and will not in this short course.

A second example we have seen is

```
#include<sstream>
#include<fstream>
#include<iostream>
using namespace std;

int main()
{
    stringstream file_name;
    stringstream problem_name("my_problem");
    ofstream script_file;
    file_name << problem_name.str() <<".disp";
    script_file.open((file_name.str()).c_str());
    // ..
    script_file.close();
    return 0;
}
```

In this example `str()` is a member function of `stringstream` that return a `string`. Then `c_str()` is a member function of `string` that returns a `const char*`, i.e. a constant character array.

Full listing of all standard C++ libraries can be found at `http://www.cplusplus.com`. Note, many of the higher classes inherit from the more basic ones. For example both `isstream` and `ifstream` inherit from `istream`, this explains why both type of streams have similar methods defined and can be interchanged so easily.

20.5 STL vector

A very useful templated class in C++ is the STL vector. This is a container that can take any type of data i.e. `int`, `double` or even a user-defined class. Below we briefly introduce this very useful standard template library (STL) class.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n = 0;
    cout << "Enter how many square numbers to compute:" << endl;
    cin >> n;

    //Declay an integer array of length n
    vector<int> my_vec(n);

    //Assign the value i squared to each element
    for (int i=0;i<n;i++)
```

```
        my_vec[i]=i*i;

    //Write out each element's value to the screen
    for (int i=0;i<n;i++)
        cout << my_vec[i] << endl;

    return 0;
}
```

A major advantage of `vector` is that it supports dynamic size control, i.e. by calling `my_vec.resize(20)` you can change the size of `my_vec` *after* creation; this is useful when the size of a field is not known at compile-time.

In the above examples `my_vec` is a class which has both data (in this case integers) and functions (e.g. `resize` or `sum`).

`valarray` is a very similar container class `vector` but can only store numerical data. This is implemented in the header `<valarray>`.

Access to data stored, in both `valarray` and `vector` classes can be slower than pointer based method described in §15.2.8, but the use of iterators (not covered by this course) can massively improve access time for these classes.

Chapter 21

I/O (Input and Output)

The I/O in C++ is completely renewed as compard to C. This was mostly necessary because of the complex format in C with non-constant number of variables.

The two most important objects are I/O-channels like `istream` and `ostream`. Both are declared in the header `iostream`. In the previous chapters, the standard I/O channels '`cin`' and '`cout`' with the operators '`<<`' or '`>>`' were already used frequently. In the following alternatives and extensions are explained.

In which sequence is I/O processed?

```
#include <iostream>
//    The programmer types this
      cout << "Salary: "    << Person.mSalary;

//    ... but the compiler reads this:
   ( cout << "Salary: " ) << Person.mSalary;
```

In order to write the string `Salary:` to the screen, the operator `<<` is called from the `ostream` `cout` and returns another `ostream` with which the next `<<` is called. This time, `Person.mSalary` is the second argument. Analogously after an input with `cin`, another `istream` is returned so that one can input again.

21.1 Elementary functions of iostreams

Instead of the form as used up to now, I/O can also be performed in the form of functions – for compatibility. However, the function-form has much wider range of applications, see Tab. 21.1.

```
char c1='x';
cout << c1 << '\n';     // Syntax used up to now
cout.put( c1 );         // Equivalent form using functions
cout.put( '\n' );
```

`ostream::put (char c);` Example: `cout.put( c1 );`	writes character c to ostream
`char c='x';` `cout << c;`	For comparison ... also writes the char 'c'
`istream::get (char c);` Example: `cin.get( c1 );`	reads character c from istream
`char c;` `cin >> c;`	For comparison ... Reads character c, ignores SPACE, TAB, EOL

Table 21.1: Prototypes of some element-functions for I/O alternative to the use of `cout` und `cin`.

There are many related element-functions like this that allow a much more controlled and well-behaved input and output of data. (Further reading if needed).

21.2 Formatting

Typically it is necessary to change the format of the output, e.g., in order to obtain a better readable table, or in order to prepare the output for the transfer to another software, or to write the full precision on the screen, or ... see Tab. 21.2 for a summary.

Manipulator	Description
`int width=12;` `cout << setw(width);`	sets the minimal number of digits for the following output only.
`int prec=5;` `cout << setprecision(prec);`	Changes the number of digits behind the comma, or the max. number of digits
`char c='*';` `cout << setfill(c);`	Defines a filling-symbol as, e.g., `*`.
`boolapha`	Boolean values are written as textual version, i.e., true not 1 for a true value and false not 0 for a false value.
`noboolalpha`	Default. Has the opposite effect of boolaphla.

Table 21.2: Examples of formatting functions for the output

In order to call these functions, first use: `#include<iomanip>`. Note that there are modifications that only act on the next output, and others that remain active permanently.

```
#include<iomanip>
```

```
double x=1234.5;
cout << x << '\n';    // leads to: 1234.5
cout << setfill ('*') << setw (10) << setprecision (5);
cout << x << '\n';    // leads to: ****1234.5
cout << x << '\n';    // back to previous setting: 1234.5
```

Furthermore there exist switches (flags) for the formatted output that need no parameters, as summarised in Tab.21.3:

Flag	Group	function
left	adjustfield	left-adjust
right	adjustfield	right-adjust
internal	adjustfield	+/- left
dec	basefield	decimal numbers
hex	basefield	hexadecimal numbers
oct	basefield	octal numbers
showbase		shows the dec-basis of hex- and oct-numbers
showpos		prints '+' before number if positive
uppercase		E, X, A-F instead of: e, x, a-f
fixed	floatfield	Fixed-point notation
scientific	floatfield	scientific notation

Table 21.3: Flags for the formatted output of data

In order to activate flags one can use alternatively

```
cout.setf (ios:: 'flag', ios:: 'group')
cout << setiosflags (ios:: 'flag', ios:: 'group')
```

where the possibilities for `'flag'` and `'group'` are summarized above in Tab. 21.3, where the second parameter is not always needed. In order to change the setting, or set it to default, for a single flag or for groups of flags, one can use:

```
cout.unsetf (ios:: 'flag')
cout << resetiosflags (ios:: 'flag')

cout.unsetf (ios:: 'group')
cout << resetiosflags (ios:: 'group')
```

21.3 Files

For reading and writing on files, one can use the channels `fstream` and `ofstream`. These are derived from the std-streams `istream` and `ostream`.

```
# include <fstream>
...
double  z=1.234;
int     m=5;
ofstream outs ( "filename", ios::out );  // open the file with name
                                         // 'filename' for writing
                                         // ios::out can be skipped
outs << x << '\n';   // the use of the self-defined output-stream
outs << j << '\n';   // 'outs' is analogous to the use of 'cout'
outs.close();        // close the out-fstream 'outs'

ifstream ins  ( "filename", ios::in );   // open the file with name
                                         // 'filename' to read from
                                         // ios::in can be skipped
ins >> z;            // the use of the self-defined input-stream
ins >> m;            // 'ins' is analogous to the use of 'cin'
ins.close();         // close the in-fstream 'ins'
```

At least now, when working with files, we need information whether we have reached the end of the file (EOF) or if the opening of the file was successful. For this, the so-called status-informations can be used, which return – dependent on the status of a file – either `true` or `false`:

```
ins.good()   // true, if no error occured in stream 'ins'
ins.eof()    // true, if the end of file/stream is reached
ins.bad()    // true, if an hardware error occured
ins.fail()   // true, if a logical error or a
             // hardware error occured
ins.clear()  // set ins.good() to true
ins.clear( ios::failbit | ins.rdstate() )
             // set fail() manually to true
```

Example: How to read from a file to its end?

The following examples show which problems can occur doing this.

```
int n;

while (!cin.eof()){
       cin >> n;
       // .... this reads too far
}

while((cin >> n).good){
       // ... this reads not far enough
}

while(cin >> n){
```

```
        // ... this is the way it works
    }
```

Chapter 22

Organisation implementation and header-files

Big programs and program-packages should not be kept in a single file since this can lead to long compilation times and non-clear dealing with and navigation in the file. Furthermore it causes big trouble if more persons work on (a single file) at the same time.

Therefore, one typically splits programs into several files (projects), that can be compiled independently from each other. There are compilable (Implementation-) C++-files with the extension `.cc` or `.cpp`. Files that only contain declarations are called header-files and are used to make functions and data from one file known to another. These have the extension `.h`. In general (except for `main.cc`) an implementation-file comes together with a header-file.

header-files should contain:

1. so-called include-guards, that hinder the multiple inclusion of header-files (required by the standard)

```
#ifndef HEADER_H
#define HEADER_H
  // declarations
#endif
```

2. Declarations of functions that are used amongst several modules. For declarations of local functions, another header-file can be used.
3. Definition of `inline` functions
4. Declaration of classes and their elementfunctionen
5. `extern` declarations of global variables, and declaration of `static` varbiables
6. Declaration and definition of `template`-functions and -classes

implementation-files should contain:

1. Implementation of elementfunctions of not-template-classes
2. Implementation of regular functions
3. Declaration von file-scope, class-scope and global static variables

Exercise 22.1.
For example, the `Sample` class in exercise 9.1 is long, and we want our code to be more structured. Thus, we copy the class declaration (lines 5-19 of the code in exercise 9.1) into a header file `Sample.h`, and copy the class member declaration (line 21-69 of the code in exercise 9.1) into a implementation file `Sample.cpp`, where we have to add some include directives at the beginning of the file. We tell the compiler where to find the class declaration by adding `#include "Sample.h"` to teh implementation files. This gives our code structure. The whole programm can now be compiled with the command `g++ main.cpp Sample.cpp`. ■

Sample.h:

```
#ifndef Sample_H
#define Sample_H
class CSample
{
public:
    CSample(); // constructor
    ~CSample(); // destructor

    int Input(const char* fname);
    double GetSum();
    double GetMax();
    int Size;
private:
    int Nmax;
    double *a;
    bool mem_alloc;
};
#endif
```

Sample.cpp

```
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include "Sample.h"
using namespace std;

// ... (copy line 21-69 of the code in exercise 9.1 here)
```

main.cpp

```
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include "Sample.h"
using namespace std;

// Uses the class CSample
// to read in data, print the size of the data-field,
// compute the sum of, and the maximum.
int main()
{
    CSample Test;
    Test.Input("data.dat");
    cout << "Size = " << Test.Size << "\n";
    cout << "Sum = " << Test.GetSum() << "\n";
    cout << "Max = " << Test.GetMax() << "\n";
    cout << endl;
    system("PAUSE");
    return 0;
}
```

22.1 Compiling and linking

Whenever we use the `g++` compiler we actually do two steps: compiling and linking. Let us consider the above example. One can compile the code using the `-c` flag e.g. `g++ -c main.cpp` and `g++ -c Sample.cpp`. This will create object files called `main.o` and `Sample.o`. These separate objects are then linked into an executable by calling `g++ main.o Sample.o -o main.exe`.

Chapter 23

Literature and longer exercises

23.1 Further reading

The many new terms used in the last pages indicates that C++ has much more possibilities, rules and options to be explored. Classes, templates, over-loading, scopes, etc. can not be discussed in this short course. However, here is a list of further literature:

1. Wikipedia ...
2. Brian W. Kernighan and Dennis M. Ritchie, *The C-Programming Language*, 2nd edition, Prentice Hall, 1988.
3. B. Stroustrup, *The C++ Programming Language*, 3rd edition, Addison Wesley, 1997.
4. Steve Ouallinie, *Practical C++ Programming*, O'Reilly, 1995.
5. C. S. Horstmann, *Mastering C++*, John Wiley & Sons, New York, 1996.
6. S. B. Lippman, *C++ Einführung und Leitfaden*, 3. Auflage, Addison-Wesley, Bonn, 1998.

23.2 Longer exercises C/C++

Exercise 1

Describe with a few words what the following program is doing. Do not copy this file into your computer - solve this by reading it and write a few sentences (in 15 minutes). Then plot a flow-chart or flow-diagram.

```
#include<iostream>
#include<fstream>
#include<cmath>
using namespace std;

int main(int argc, char *argv[])
{
// Define field x(t) with length 1000
   double x[1000], t;  // output variables
// initial conditions
   double A, delta;     // A=amplitude, delta=phase-angle
   double mass, ksprng; // mass=mass, ksprng=spring-constant
   double t_max, dt;    // t_max=max-time, dt=time-intervall
// Request input of the parameters
   cout << "amplitude      "; cin >> A;
   cout << "phase-angle    "; cin >> delta;
   cout << "mass           "; cin >> mass;
   cout << "spring-const.  "; cin >> ksprng;
   cout << "max-time       "; cin >> t_max;
   cout << "time-interval  "; cin >> dt;
   double omega=sqrt(ksprng/mass);
// Loop from t=0 to t=t_max
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ )
   {
       x[i]=A*sin(omega*t+delta);  // Compute function
       t=t+dt;                     // Step to next time
   }

   ofstream outfile("plot.data");  // Output to file
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ )
   {
      outfile << t << " " << x[i] << "\n";
      t=t+dt;
   }
}
```

Exercise 2

Writing the first program.
Extend the minimal, basic C++ program by the command lines `int inum=0; cin >> inum;` where the second requests input of an integer number from the keyboard.

Write a C++ program that prints the binary representation of the integer number `inum` to the screen. You can check if the program works by, e.g., using the number 496 from the lecture-example.

Exercise 3

What is the greatest value of n that can be used in the sum

$$1^2 + 2^2 + 3^2 + \ldots + n^2$$

and gives a total value of the sum less than 100?
(a) Write a short C++ program that answers this question – like in the MATLAB exercise 7.3
Then answer the question also for an exponent of 10 and a total value of the sum of
(b) less than 10^6 and
(c) less than 10^{12} (*Advanced*).
(d) If doing both parts compare the run-time between you C++ and MATLAB implementation.

Exercise 4

Declare an array with 10 001 entries. Compute the solution vector $f(x_i)$, with $x_i = 0, \ldots, 10\,000$ (as fast as possible!)

$$f(x_i) = \frac{x_i}{\sin(x_i) + 2}$$

In the same program, form the alternating sum

$$S_n = f(x_0) + f(x_1) - f(x_2) + f(x_3) - \ldots$$

up to the last term and print the result to the screen. Finally, compare the result with the result obtained in the MATLAB section 7.3.

Exercise 5

In the following program (part of which we have discussed today) there are hidden 10 syntax and typo-errors. See how many you can find in 5-10 minutes.

```
#include <iosteam>
#include <cmath>
#include <cstdlib>
using namespce std;

int main(int argc, char *argv[])
{
    int i;
    int n=5;
    int *p;
    int a[10];

    for( i=0; i<=10; i++ }
        a[i]=10;
        p[i]=10;

    cout << a << " \n";
    cout << *a << " \n";
    cout << *(a) << " \n";
    cout << a[1] << " \n"
    cout << *(a+2) << " \n";
    cout << size of( int ) << " " << sizeoff( int* ) << "\n";
    cout << sizeof( double ) << " " << sizeoff( double* ) << "\n";
    cout << " n: " << n << " " << &n << "\n";
    i=n+1;
    cout << " i: " << i << " " << &i << "\n";
    p=&n;
    cout << " p: " << p << " " << *p << "\n";
    *p+=2
    cout << " p: " << p << " " << *p << "\n";
    p++;
    cout << " p: " << p << " " << &p << "\n";

    return 0;
}
```

Exercise 6

This is an example of pointers:

(a) Use the following program segment and play around using the pointers.

```
#include <cstdlib>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char *argv[])
{
    char c='x';
    int a[20];
    int *p = a;           // ok, a is name of the field and can
                          //     be used as address to the field
// int *p = &(a[0]);      // ... equivalent (error->redeclaration)

    p[5] = 4;             // access to a through pointer arithmetic,
                          // a[5] is set to the value of 4
    a[5] = 4;             // ... equivalent
                          //
    int b[20][30];        // b is now an array of 20 arrays of
                          //     30 elements of type int
                          //     int *p = b;  error! wrong type, since
                          //     b is of type pointer to 30 int here
    int *r = b[0];        // ok, r points to the first int
                          //     in the first (of 20) array(s) of 30 ints
    int (*q)[30] = b;     // ok, q points to the first set of 30 int's,
                          //     q and b can now be used synonymous
    q[12][15]  = 26;      // ok, set element (13,16) to the value 26
    (*(q+12))[15] = 26;   // ... equivalent
    *(*(q+12)+15) = 26;   // ... equivalent
    b[12][15]  = 26;      // ... equivalent

    cout.put(c);
    int    *p_i  = new int;
//  Person *p_p  = new Person("B.Stroustrup",45);
    int         n = 40;
    int    *pa_i = new int[n];
// ...
    delete   p_i;
//  delete   p_p;
    delete[] pa_i;
//
//
    pa_i[0]=5;
    pa_i[1]=4;
```

```
    for(int i=0; i<10; i++) cout << pa_i[i] << ' ';

    return 0;
}
```

Serious programming exercise:

Define an `int` field (vector) that contains the numbers:
`{ 1, 0, -10, 12, -2, 5, 5, 1, -4, -2, -4, -8, 3, 25, 51, -4, -5 }`

Then sort the entries of this linear field/vector.

(b) Read the field from a file.
(c) Use a linked-list to store the field.
(d) Implement a linked list (with pointers) to store the field.
(e) Use this linked list (with pointers) to sort the field.
(f) Write the sorted list to an output file, but remove double entries from the list. What is the length of the field before and after this sorting/remove operation?
(g) (Advanced) Implement a (heap) tree structure to sort the field such that the largest entry is always on top of the tree.

Exercise 7

Write a root-finding function in C/C++ and apply to general fifth order equation of your choice. (*Advanced*)