

MATLAB Course

R.P.J. Kunnen

revised by E.J.D. Vredenburg

October 19, 2014

Preamble

The aim of this course is to learn how to use MATLAB. Concurrently, basic programming skills that will also be useful outside of this course, will be practiced. A basic knowledge of MATLAB is required for follow-up courses such as *OGO Instrumentele Fysica* but will turn out to be generally useful both during college years and afterwards.

Some parts of this manual, mainly problems, have been marked with an asterisk (*) to identify more challenging problems. Problems marked in this way are intended for advanced students; they are not required for the final test.

This version of the manual was revised in 2014 to correspond to Matlab version R2014a.

Schedule

The course is divided into six half-day sessions. The material to be treated per session is shown in the table below. The exercises are located in the text, numbered with Roman numbers, and are meant to learn new commands. The problems, numbered with Arabic numbers, are collected in the problem sections at the end of each chapter and serve to put the new knowledge into practice on a more advanced level.

NB: The course is finalized by a two hour long test (“tussentoets”) that will be held during the last two hours of the last session.

NB: Attendance is mandatory for this computer laboratory. You cannot complete the course *Experimentele Fysica 2* if you have been absent during one or more of the computer laboratory sessions. Absence for a valid reason must be reported via e-mail to `onderwijsn@tue.nl` with a CC to `phys.exp.fys@tue.nl`

Session:	Read the following sections and do the following assignments:
1	§1.1 - §1.5 — exercises 1.i - 1.x; problems 1.1 - 1.6 - introduction to MATLAB - using variables - order of operations §1.6 - §1.8.5 — exercises 1.xi - 1.xix; problems 1.7 - 1.24 - defining functions - programming in MATLAB: if, for, and while, comments
2	whatever remains of Chapter 1 that was not finished §2.1 — introduction §2.2 — exercises 2.i - 2.v; problems 2.1 - 2.8 - graphical representation §2.3 — exercises 2.vi - 2.xxii; - matrices, how to manipulate matrices
3	whatever remains of the previous session that was not finished §2.4.1, §2.4.2 — exercises 2.xxiii - 2.xxv; problems 2.16 - 2.19 - matrix product §2.4.3 — exercises 2.xxvi - 2.xxix; problems 2.20 - 2.25 - solving a set of linear equations
4	whatever remains of the previous session that was not finished §3.1 - §3.4 — exercises 3.i - 3.xi; problems 3.1 - 3.12 - zeroes and extremes of functions - numerical integration
5	whatever remains of the previous session that was not finished §4.1 - §4.3 — exercises 4.i - 4.viii; problems 4.1 - 4.9 - solving differential equations numerically example MATLAB test
6	whatever remains of the previous session that was not finished Q&A session — ask anything you want test — 2 hour long “tussentoets”

Contents

1	MATLAB: the basics	1
1.1	What is MATLAB?	1
1.2	Start-up screen	1
1.3	Help!	2
1.4	Data types	3
1.5	Variables	3
1.6	Order of operations and mathematical functions	6
1.7	Defining functions and creating scripts: m-files	7
1.8	Programming in MATLAB	9
1.8.1	Conditional expression: if	9
1.8.2	The for-loop	11
1.8.3	The while loop	12
1.8.4	Tips for programming	13
1.8.5	Differences between MATLAB and other programming languages	13
1.9	Problems	14
2	Matrices	19
2.1	Introduction	19
2.2	Arrays and graphical output	19
2.2.1	Graphs, continued	21
2.3	Tables of numbers: matrices	24
2.3.1	Simple matrices	24
2.3.2	Functions of multiple variables	26
2.3.3	Manipulating matrices	27
2.3.4	Algebraic operations for matrices	28
2.3.5	Reading and writing data	31
2.4	Linear algebra	32
2.4.1	Systems of linear equations	32
2.4.2	Matrix product	32
2.4.3	Solving systems of linear equations	34
2.5	Problems	36
3	Numerical evaluation of functions	43
3.1	Introduction	43
3.2	Zeroes	45
3.3	Extrema	45

3.4	Numerical integration	46
3.5	Problems	47
4	Differential equations	49
4.1	Introduction	49
4.2	A first-order differential equation	49
4.3	Systems of ordinary differential equations	51
4.3.1	* <i>Stiff differential equations</i>	53
4.4	Problems	55
A	* <i>Gaussian elimination</i>	59
	List of Tables	61
	Index	62

Chapter 1

MATLAB: the basics

1.1 What is MATLAB?

MATLAB is a computing environment that is widely used in academic and industrial research institutions. Comparing MATLAB and MATHEMATICA, the former is geared toward numerical computations and the latter toward symbolic mathematics, although MATLAB also allows for symbolic computing and MATHEMATICA for numerical processing.

The name MATLAB is an abbreviation of Matrix Laboratory, signifying that MATLAB is particularly good at manipulating matrices: tables filled with numbers (treated in more detail in chapter 2). The overwhelming majority of experimental as well as simulated data consists of such tables of numbers; examples are images, a voltage sampled during a time interval by an oscilloscope, and the current-voltage characteristic of a diode. Therefore, analyzing data from experiments or producing data with simulations usually requires numerical processing of matrices. Many functions that can manipulate matrices are predefined in MATLAB, and it is also possible to define new functions. MATLAB can also visualize the results of computations in many different kinds of graphs. The user defines how data is processed by writing (a set of) programs called *scripts*, which contain symbolic code that the computer's processing unit can execute.

The purpose of this course is to learn these basic numerical computing skills using an introductory set of MATLAB capabilities. MATLAB has many more possibilities than covered in this manual, including a large number of advanced toolboxes. Anyone that has completed this course should be able to learn the more advanced techniques by further independent study.

1.2 Start-up screen

After you have installed MATLAB an icon will appear in the start menu, and perhaps shortcut has been added to the computer's desktop as well. After starting up MATLAB, the screen shown in figure 1.1 should appear. Within MATLAB this screen is also called the desktop. When MATLAB is started the cursor will be located in the central window A. This is the command window, and commands can be inserted after the MATLAB prompt:

```
>>
```

After a command has been input (possibly after the output), another MATLAB prompt will appear.

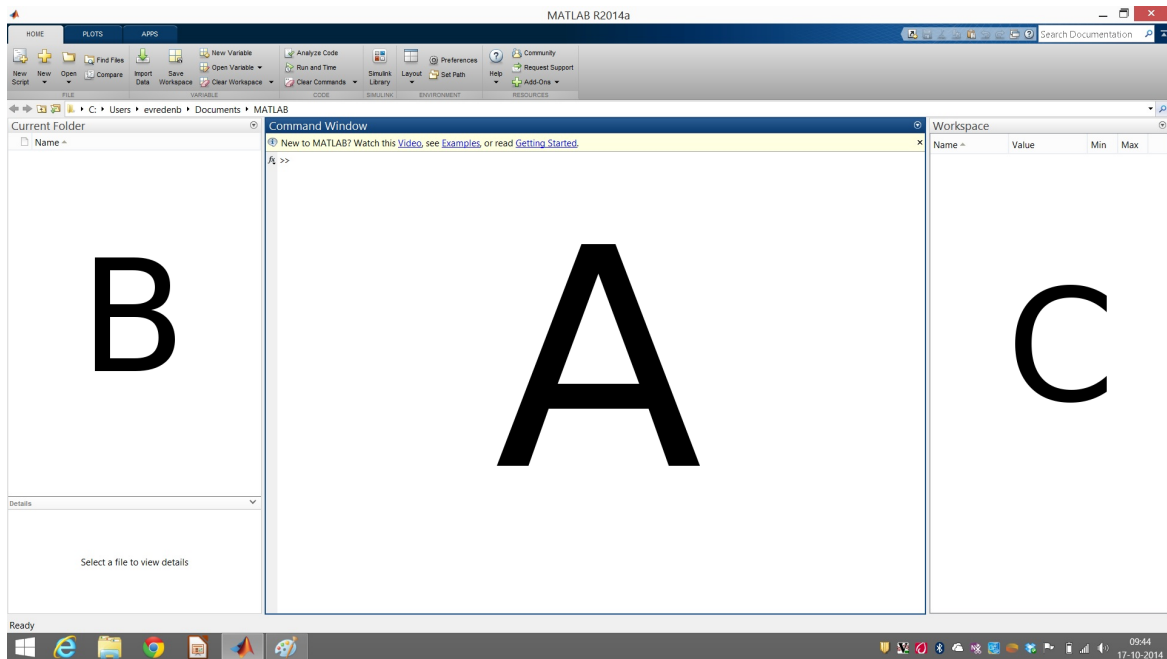


Figure 1.1: The start-up screen of MATLAB. A is the ‘Command Window’ used to input and execute commands; B is the ‘Current Folder’, the contents of the current folder; C is the ‘Workspace’ that displays all currently defined variables and their corresponding value(s).

Exercise 1.i — Type:

```
>> 1+2
```

What happens?

Next to the Command Window there is Current Folder window that shows the files and folders located in the current folder. This is window B in Figure 1.1. The Workspace window shows all variables currently in use (see also 1.5). A history of executed commands can be displayed in an additional Command History window.

1.3 Help!

The Help documentation in MATLAB is very extensive. This option is invoked by clicking the option ‘Help’ or ‘?’ in the menu bar on the top of the screen. It is often useful to directly go to the correct page in the Help documentation by using the command prompt. For example, the documentation for the built-in function `sqrt` can be found directly by typing into the Command Window:

```
>> help sqrt
```

After typing this and pressing Enter, a short description of the command `sqrt` will be shown. If a more elaborate description is needed one can use `doc`. For example, again for `sqrt`, enter

```
>> doc sqrt
```


to open a new window, showing more detailed information and some examples using `sqrt`. In this Help window links to the descriptions of related functions are also displayed.

1.4 Data types

Data to be processed normally takes the form of numbers or text, and MATLAB contains built-in types that can represent such data. Most numbers can be stored using the type `double`, which represents 64-bit floating-point (loosely: real) values, including:

- real numbers (e.g., 6.63×10^{-34}) with absolute value between 2.22507×10^{-308} and 1.79769×10^{308} ; numbers smaller than the first of these are stored as 0, numbers larger than the second as `Inf` (for infinite),
- integer numbers ($\dots, -1, 0, 1, \dots$) with absolute value smaller than 2^{53} ; larger integers cannot be represented exactly by a `double`,
- complex numbers (e.g., $1.0 + 2.0i$) are stored in two `doubles`, one for the real and one for the imaginary part,
- boolean numbers that represent true (1) or false (0).

Textual data is stored using the type `string`, which is a sequence of characters. For example, 'Hello, world!' is a string. There are many functions available to manipulate strings. Numbers can be converted to strings using `num2string`, and vice versa using `str2double`.

Several other data types are available that are not covered here, including additional numeric types that offer advantages for specific applications.

1.5 Variables

Exercise 1.ii — Type

```
>> a=1+2
```

and press Enter. What do you see?

We see that $1 + 2$ is 3. The result of this calculation is now stored in the variable named `a`. This variable is now also displayed in the Workspace (window C in Figure 1.1). Variables can be used in further calculations.

Exercise 1.iii — What are the results of the following commands? First deduce them yourself, then execute them in MATLAB. Were your initial thoughts correct?

```
>> a=a+1
>> b=a*3
```

If the result is not explicitly assigned to a variable (the command does not start with `variable=`) then the result will be assigned to `ans`.

Exercise 1.iv — Describe what happens when the following commands are executed:

```
>> a*b
>> ans/3
>> b=a*b
>> a*b
```

The variable `ans` always contains the answer to the *last* command for which the result was *not* assigned to another variable. `ans` can be used in further calculations, but be careful! It is very easy to overwrite the variable `ans` by executing a subsequent command.

In long calculations some in-between results do not need to be displayed. This can be done by using the semicolon symbol:

```
>> c=4;
>>
```

No result of the command `c=4` is shown in the Command Window. The variable `c`, however, is set and equal to 4. This can be seen in the Workspace, but can also be checked by entering

```
>> c
c =
    4
```

This shows the value assigned to `c`.

In MATLAB the full stop (`.`) is used as the decimal mark. If very large or very small numbers need to be entered, it is often useful to use the notation with `e`.

Exercise 1.v — Execute the following code:

```
>> a=0.5
>> b=a*4
>> c=1000000
>> d=1e6
>> c-d
>> f=1.34e-8
```

What do you see?

As you can see, `e6` actually means $\times 10^6$; `e-8` stands for $\times 10^{-8}$.

When a line of input is very long, it is often helpful to use multiple rows for one command. This can be achieved by typing three full stops (`...`), and hitting the Enter key. The input line will continue on the next row. Extra spaces can also be added freely.

Exercise 1.vi — Execute the following commands. Can you predict the results?

```
>> d=100*2+50*3+...
75*4+12
>> b = 2 + 3
>> c = 3 + 4
```

Variables can be named using all letters, numbers, and the ‘underscore’ `_`. However, names of variables need to start with a letter and can be at most 31 characters long. They are also case sensitive: `a` and `A` are treated as two separate variables. Some variable names have a predefined value, such as `pi`, or represent a function, i.e. `sin`. It is allowed to use names like these for your variables, but it is not advised for obvious reasons. For an overview of predefined variables, see table 1.1.

To reset the value of a variable the command `clear` can be used.

Exercise 1.vii — Application of the command `clear`:

```
>> x=10
>> clear x
>> x
```

Why does an error message appear?

Using `clear` multiple variables can be cleared. To clear variables `a` and `b` use

```
>> clear a b
```

To clear *all* variables, simply use:

```
>> clear
```

In order to use variables later, they can be saved on the hard disk. This can be done using the `save` command. MATLAB will save the variables in a `.mat` file.

```
>> save file
```

will save *all* variables in the workspace to a file named `file.mat`. To save specific variables they can be added as an argument:

```
>> save file a b c
```

Now *only* the variables `a`, `b`, and `c` are saved. In order to load these variables again the `load` command is used:

Table 1.1: Overview of the predefined variables in MATLAB.

Symbol	Meaning
<code>ans</code>	Result of last calculation not assigned to a variable..
<code>eps</code>	$\text{eps} = 2^{-52} \approx 2.2 \times 10^{-16}$, the accuracy of floating-point numbers.
<code>i</code> and <code>j</code>	$i = j = \sqrt{-1}$, imaginary unit.
<code>Inf</code>	Result of: $1/0$.
<code>NaN</code>	Result of: $0/0$.
<code>pi</code>	$\text{pi} = 3.14159\dots$
<code>realmin</code>	Smallest positive number, close to 2.2×10^{-308} .
<code>realmax</code>	Largest positive number, about 1.8×10^{308} .

Table 1.2: Notation of some predefined mathematical formulas in MATLAB.

Mathematical notation	MATLAB notation
$ x $	<code>abs(x)</code>
\sqrt{x}	<code>sqrt(x)</code>
$\exp(x) = e^x$	<code>exp(x)</code>
$\ln(x)$	<code>log(x)</code>
$\log_{10}(x)$	<code>log10(x)</code>
$\sin(x)$	<code>sin(x)</code>
$\cos(x)$	<code>cos(x)</code>
$\tan(x)$	<code>tan(x)</code>
$\arcsin(x)$	<code>asin(x)</code>
$\arccos(x)$	<code>acos(x)</code>
$\arctan(x)$	<code>atan(x)</code>

```
>> load file
```

Exercise 1.viii — Define three variables `a`, `b` and `c`, and assign a numerical value to them. Save the variables to a file named `abc.mat`. Clear all variables in the workspace. Load the saved file to see whether the variables are saved and loaded correctly.

1.6 Order of operations and mathematical functions

MATLAB uses the following notation for algebraic operations:

- Addition: `a+b`,
- Subtraction: `a-b`,
- Multiplication: `a*b`,
- Division: `a/b`,
- Exponentiation: `a^b`.

MATLAB uses the standard order of operators: first exponentiation, then multiplication and division, then addition and subtraction. Parentheses (and) can be used to force MATLAB to first execute commands and/or operations in between. A short overview of predefined mathematical functions is shown in table 1.2.

Exercise 1.ix — What is the difference?

```
>> 1/2+3
>> 1/(2+3)
```

Exercise 1.x — Write the following statements in MATLAB code:

$$\begin{aligned}x &= \cos(2\pi) \\y &= e \\z &= \ln y \\a &= (-2 + 4)^{\frac{1}{6-4}} \\b &= a^2\end{aligned}$$

Validate your answers by entering them in MATLAB!

1.7 Defining functions and creating scripts: m-files

So far, we have used Matlab mostly as a glorified calculator by entering commands in the command window and observing the result. However, the power of programming lies in the ability to repeat an intricate list of commands multiple times. This requires that commands can be grouped and stored in files which may then be executed as a whole.

When one particular chain of commands needs to be repeated multiple times it is typically more practical to define a *function*. Functions are separate files that are saved with the extension `.m`. Therefore, they are also referred to as ‘m-files’. As an example, let’s say that we want to evaluate the function $f(x) = \exp(x^{1/3} - \sin(\pi x))$ for arbitrary values of x . We need to make a new function (or m-file). This can be done by choosing ‘File’ → ‘New’ → ‘Script’ in the menu on the top of your screen. A new window will open, called the Editor .

Exercise 1.xi — Insert the following lines into the Editor:

```
function y = func(x)
y = exp(x^(1/3)-sin(pi*x));
```

Save this file as `func.m` via ‘File’ → ‘Save’. In general, it is best to give the file the exact same name as the function (`func` → `func.m`) to avoid confusion. For the same reason, avoid using names for your own functions that are already predefined in Matlab. As an example, it is a bad idea to define a function named `cos` since Matlab already has a function by that name representing the cosine.

In Current Folder (B in figure 1.1) you should now see our new file. The command `function` indicates that `func.m` can be used as a function: if an input value `x` is given, the function will calculate an output `y`. It easily to see that $f(0) = 1$. Let’s check this:

```
>> func(0)
ans =
    1
```

The function `func` has used the value `x=0` to calculate `y`. This value of `y` is returned as the result.

Exercise 1.xii — Try the function `func` for different input values. What happens when you hand `func` a negative value as input?

Functions can also handle multiple input values. For example: the function that calculates the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Where $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$ is the factorial. We will use the predefined MATLAB function `factorial(n)` to calculate $n!$.

Exercise 1.xiii — In the Editor, enter the following lines into a new m-file.

```
function y = binom(n, k)
y = factorial(n)/(factorial(k)*factorial(n-k));
```

Save this file as `binom.m`, and execute the following statements using the new function

```
>> binom(10, 4)
>> binom(6, 3)
```

Of course, MATLAB has a predefined function to calculate binominal coefficients: `nchoosek`. Do the results of our function `binom` match with those obtained by using `nchoosek`?

An important aspect of functions in MATLAB is that variables defined in the Workspace are *separate* from variables defined within a function. Variables from the Workspace cannot be used inside the function. At the same time, variables defined within a function are not available in the Workspace. It is even allowed to use the same name for variables within a function and within the Workspace! Exchange of variables between Workspace and functions (and functions between themselves) is exclusively done through function arguments and return variables.

Next to functions, it is also possible to save lists of commands that have no specific input or output but are executed frequently. Files containing such command lists are called *scripts*. Unlike functions, scripts do have access to the same variables as the Workspace. Variables declared in a script are added to the declared variables in the Workspace. The m-file does not contain a special command such as `function`. An example of a script that declares a number of variables is:

```
squareroot2 = sqrt(2);
squareroot3 = sqrt(3);
goldenratio = (1+sqrt(5))/2;
esquare = exp(1)^2;
```

Exercise 1.xiv — Write this script, save it as `constants.m` and execute it. What happens to the Workspace?

Note that the Editor signals that a script contains an error by tagging the line where the error occurs using a red or orange mark in the right margin. Hovering above the colored mark produces a hint about what is wrong. This is generally very helpful in debugging a script.

1.8 Programming in MATLAB

In the previous section we have already made some small programs, or m-files, in MATLAB. So far these were fairly simple. In this section we will cover the building blocks that are used to write programs that are larger and more complex: the conditional expression and loops. Using these constructs it is possible to control the order in which commands are executed depending on conditions encountered during the computations.

In what follows, always create scripts (m-files) when solving exercises and problems.

1.8.1 Conditional expression: if

The `if` expression is used to test a condition. Depending on whether the condition is met, different parts of the program will be executed. The overall structure of the conditional expression is:

```
if [condition]
    [lines of code to be executed if the condition is met]
else
    [lines of code to be executed if the condition is not met]
end
```

The condition is a so-called *relational expression* that can either be *true* or *false*. In most cases the condition is tested using a relational operator.

Exercise 1.xv — For example:

```
a = input('Enter a number lower than 10: ');
if a < 10
    disp('True!')
else
    disp('False!')
end
```

Type this script in the Editor and save it as `testiflower.m`, then execute it. Using the command `input` a number can be entered using the keyboard. This input value is assigned to the variable `a`. Then we check whether `a` is indeed smaller than 10. This is the same as checking whether the statement '`a<10`' is true. Try the script with some numbers i.e.: 3, 11, -12.5, `-3*exp(1.5)`. Does the script work correctly?

Some relational operators, and their corresponding MATLAB syntax, are listed in table [1.3](#). A few examples:

- `1 == 1` is *true*;
- `1 == 2` is *false*;
- `1 ~= 2` is *true*;
- `1 < 2` is *true*;
- `1 >= 2` is *false*.

Table 1.3: Relational operators in MATLAB.

Syntax	Meaning
<code>==</code>	equal to
<code>~=</code>	not equal to
<code><</code>	smaller than
<code><=</code>	smaller than or equal to
<code>></code>	larger than
<code>>=</code>	larger than or equal to

It is important to realize that there is a difference between `=` and `==`! With `=` we can *assign* a value to a variable, whereas `==` is used to *compare* two variables.

Relational operators can also be combined. There are two ways to do this. One is to use *nested if-blocks* (an `if` inside an `if`). The other way is to use logical operators. An example of the first option:

```
a = input('Insert a number larger than 1, but smaller than 10 ');

if a > 1
    if a < 10
        disp('True!')
    else
        disp('False!')
    end
else
    disp('False!')
end
```

First, it is checked whether variable `a` is larger than 1. If not, the program will jump to the last `else` and write `False!`. If `a` is larger than 1, a second `if` follows to check whether `a` is smaller than 10. If this is the case, then the text `True!` will be displayed, as both conditions are met. The extra spaces are a commonly applied way to visually distinguish the different `if` blocks. The `if`, `else`, and `end` commands that belong to each other all have the same distance to the start of the line. Lines in between have a larger indent. Especially in larger programs, with a lot of nested `if` blocks, it can be hard to identify the structure of the program when indentations like these are not inserted.

This program could also use a *logical operator* instead of two nested `if` conditions:

```
a = input('Insert a number larger than 1, but smaller than 10: ');
if a > 1 && a < 10
    disp('True!')
else
    disp('False!')
end
```

The conditions `a>1` and `a<10` are combined with the operator `&&`, the *and*-operator. The combination `a>1` and `a<10` is only true if both `a>1` and `a<10` are true. Otherwise the combi-

nation is false. The other logical operator is `||`, the *or*-operator.. Hold ‘Shift’ and press key `\` to get `|`. `[condition 1] || [condition 2]` is false if both condition 1 and 2 are false. If one of the conditions (or both) is true, then the combination will also be true. There is also a *not*-operator `~`. This operator inverts the result of a condition : `~(1>2)` is *true*.

Exercise 1.xvi — An example with the or-operator `||`:

```
b=1;
c=2;
a = input('Insert a number: ');
if a == b || a == c
    disp('a is equal to b or c')
else
    disp('a is not equal to b and not equal to c')
end
```

Try this script. What does it do?

1.8.2 The for-loop

Often it is necessary for some part of a program to be repeated a couple of times, or a variable needs to be incremented in regular steps. To achieve this a **for** loop can be used. A **for** loop has the following structure:

```
for counter = start:step:stop
    [lines of code to be executed for each value of counter]
end
```

When the program arrives at the **for** loop, the variable **counter** is assigned the value **start**. The following lines of code are executed until **end** is reached. Then we return to the **for** statement, the value of **counter** is increased with **step**, and the following lines of code are repeated with the new value for **counter**. This process is repeated until **counter** would become larger than **stop**. It is also allowed to use a negative value for **step**: the **for** loop will then be repeated until **counter** would become smaller than **stop**. In case the value for **step** is omitted it takes the default value 1.

In this example a **for** loop is used to calculate the factorial $a!$:

```
function b = fact(a)
b=1;
for n=1:a
    b=b*n;
end
```

Since no step size for the **for** loop is given it will take the default value 1. This function does not check whether the input is a positive integer number, which is a requirement for a mathematically well-defined factorial. However, when the input is a positive integer the functions works perfectly! Try this script out and compare the results with that of the built-in function **factorial**.

```
>> fact(8)
ans =
40320
>> factorial(8)
ans =
40320
>> fact(100)-factorial(100)
ans =
0
```

Exercise 1.xvii — Write a script that writes to the screen all the integer numbers between (and including) 2 and 100 using a **for** loop. Validate that it functions properly. Can you modify the script to make it show only even numbers? Can you modify the script to make it count backwards from 10 to 1?

1.8.3 The while loop

Sometimes a part of a program needs to be repeated until a condition is satisfied, without knowing in advance how many repetitions are required. In those cases a **while** loop should be used. The **while** loop has the following structure:

```
while [condition]
    [lines of code]
end
```

When the program arrives at the command **while** and the condition is true, then the lines of code within the **while** loop are executed. When the program then arrives at the command **end**, the condition is checked again. If the condition is still true, the lines of code will be executed again. This process is repeated until the condition is no longer true.

Exercise 1.xviii — Write the following script:

```
a=1;
b=1;
while a ~= 10
    a=a+b
end
```

Save the script and run it. What happens, and why? What if you change the value of **b** to 2? Hint: you can use Ctrl+C to abort the script.

It is very important to make sure that the condition for a **while** loop will actually become false. If not, the program will get stuck inside the loop. In such cases the program can be aborted using Ctrl+C. If **a** *never* reaches the value of 10, the loop will continue running until there is an overflow. This will happen when **a** is larger than the largest number MATLAB can cope with (**realmax**). Probably, the intention of this program is

```
a=1;
b=2;
```

```
while a <= 10
    a=a+b
end
```

Another example:

```
secretnumber=round(rand(1)*10+0.5);
stop=0;
while stop == 0
    a=input('Guess the secret number (between 1 and 10): ');
    if a==secretnumber
        stop=1;
        disp('Correct!')
    else
        disp('Try again.')
    end
end
```

In the first line of this example script a random number between 1 and 10 is generated. `rand(1)` is a command that gives a real number x in the interval $(0,1)$. `round` rounds the number to the closest integer. The program will continue running the code in the `while` loop until the value of `stop` does not have the value of 0. This is the case when the secret number has been correctly guessed.

Exercise 1.xix — Develop a script that asks for a number as input, and then displays all odd numbers larger than 0, but smaller than the input. Use a `while` loop.

1.8.4 Tips for programming

We advise you to use a structured approach when writing a program. Before starting to write, think about what problem the program needs to solve, what steps the program should take, and how the result is going to be presented. This will benefit the clarity of the program code. During the actual programming it is *essential* to add comments to the code. Comments are lines of text in the program preceded by a `%` sign. It is also possible to append a comment like this after a command on the same line. MATLAB ignores these lines of text when executing the program. The only use for comments is to clarify the program for others, but also for yourself when you return to your program after a few months. Use comments to explain the use of the variables that have been defined, explain what happens in loops and clearly state the meaning of the input and output of functions. In the MATLAB Editor, comments are depicted in green.

1.8.5 Differences between MATLAB and other programming languages

There are a lot of different languages for programming. Each language has its own advantages, but also disadvantages, meaning that the perfect language does not exist. One must consider the advantages and disadvantages before choosing to program in a certain language (such as MATLAB). Fortunately, the structure of programming is always the same. Switching between languages is mainly switching between commands and notation, rather than switching between structures of programming.

MATLAB is a so-called *interpreted language*. This means that code is executed by a part of MATLAB that is called the interpreter. The interpreter converts the MATLAB code, one line at a time, in such a way that the computer can execute it. This is different from *compiler languages*, like Fortran and C++, where the code as a whole is converted into an executable file. The main advantages of using the interpreter language MATLAB are: errors are located more easily, a larger number of functions is readily available and there are a lot of options for graphical output. Besides this, MATLAB is optimized for the use of matrices, which are common in many problems. The primary disadvantage of an interpreted language is that it is slower than a compiler language since every line of code has to be translated individually during run-time. This also means that MATLAB programs cannot run without running MATLAB, while programs that are written in compiler language can be run independently after compilation, even if there is no compiler available.

MATLAB makes programming easier because it handles a lot of tasks automatically. Variables, for instance, do not need to be declared. In other programming languages variables need to be assigned to a part of the available computer memory. Because there are different representations for integer and floating-point numbers in computer memory, there is also a difference in the accuracy of calculations carried out with those types. For example: when an integer is divided by an integer, the result typically is stored as an integer as well, even if the actual result is a floating-point number. Think $7/3 = 2$ even though 2.333333 would be a better representation of the result. MATLAB automatically converts variables if necessary.

1.9 Problems

Problem 1.1 — Entering numbers

Which of the following commands assigns the value $\frac{1}{1000}$ to variable **a**? There can be multiple correct answers.

- a) `>> a=1e-3`
- b) `>> a=1/1000`
- c) `>> a=1e-5/0.001`
- d) `>> a=0.001`
- e) `>> 0.001=a`

Problem 1.2 — Variables 1

Predict the value that variable **c** will get after execution of the following commands. Then, execute the commands yourself and check your predictions.

```
>> 7+5;
>> a=8;
>> b=ans+a;
>> c=b/(2+a);
```

Problem 1.3 — Variables 2

What happens in each step?

```
>> a=2^5
>> b=7/(a+3)
>> c=a^b
```

```
>> c=b*c
>> a=b
```

Problem 1.4 — Variables 3

Define the variables x and y and assign the following values to them:

$$x = (4\frac{2}{7})^{4/9} \quad \text{and} \quad y = \frac{\frac{2}{11}}{5 + 3^{-1.4}}.$$

Let their value be output to the screen. Now type

```
>> format long
```

and let their values be shown on the screen once again. Save the two variables in the file `xandy.mat`. Now type

```
>> clear
```

and load the variables x and y from the file again.

Problem 1.5 — Maths 1

Evaluate the following expressions in MATLAB.

- a) $\frac{\pi^2}{4}$
- b) $(3 + \sqrt{3})^3$
- c) $e^2 + \ln 2$
- d) $\frac{\cos(\pi - e)}{1 - \sqrt{3/11}}$
- e) $\log_{10}(10^{\sin(\pi/2)})$

Problem 1.6 — Maths 2

Correct the following MATLAB commands to calculate the given expressions.

- a) $\arctan(3^5)$ `>> arctan(3^5)`
- b) $(1 + \sqrt{3})(1 - \sqrt{3})^2$ `>> (1+sqrt(3))(1-sqrt(3))^2`
- c) $e^2 - \ln 3$ `>> e^2-ln(3)`
- d) $[(1 - 6^{1/3})(1 + 6^{1/4})]^2$ `>> [(1 - 6^(1/3)) (1 + 6^(1/4))]^2`

Problem 1.7 — Functions 1

Define a function `f` that returns the function value $f(x) = x^2 \tan(x)$.

Problem 1.8 — Functions 2

Define a function `g` that returns the function value $g(x, y) = \sqrt{x^2 + y^2}$.

Problem 1.9 — Conditional expression 1

Define a function that returns the largest number of two arguments.

Problem 1.10 — Conditional expression 2

Define a function `h` that returns the function value $h(x) = |x|$ *without* using the built-in function `abs`.

Problem 1.11 — Conditional expression 3

Consider the circle centered on $(0,0)$ with radius 1. Define a function that takes two arguments, representing the x and y coordinates of a point. The function returns a value $+1$ if the point is found *within* the circle, returns -1 if the point is *outside* of the circle, and returns 0 if the point is exactly *on* the circle. How many `ifs` do you need?

Problem 1.12 — For and while 1

- Write a script that displays the integer numbers from 0 to 10 on the screen.
- Adapt your script; let it count backwards.
- Now count in steps of $\frac{1}{2}$.

Problem 1.13 — For and while 2

Write a script that displays all *odd* numbers from 1 to 21 on the screen. Use a `for` loop.

Problem 1.14 — For and while 3

Write a script that asks for a number as input. The script then shows on the screen all the integer numbers larger than 0, but smaller than the entered number. Add comments in your code!

Problem 1.15 — For and while 4

- Write a script that adds the numbers from 1 to N . Make a version that uses a `for` loop and a version that uses a `while` loop. As a check use:

$$\sum_{n=0}^N n = \frac{1}{2}N(N+1).$$

- Alter the script such that it asks the input of a value for N first.

Problem 1.16 — For and while 5

Write a script that asks for an integer number as input. The script then checks for all the integer numbers between 1 and 100 whether they can be divided by that number. If yes, the script shows that number on the screen. Use comments!

Tip: Use the MATLAB function `mod(a,b)`. This command returns the *modulo* (remainder of division) of a and b . Example: `mod(5,2)=1` while `mod(6,2)=0`.

Problem 1.17 — For and while 6

Write a script using `while` that displays on screen a temperature conversion table from degrees Fahrenheit to degrees Celsius. Use the values 0, 10, 20, ..., 300°F and convert using

$$c = \frac{5}{9}(f - 32),$$

where f represents temperature in °F and c is temperature in °C. Do the same using `for` instead of `while`.

Problem 1.18 — Fibonacci numbers 1

Write a script that generates the Fibonacci numbers $(0, 1, 1, 2, 3, 5, 8, \dots)$. The first two numbers are $F_0 = 0$ and $F_1 = 1$. The subsequent numbers are given by $F_n = F_{n-1} + F_{n-2}$ (see http://en.wikipedia.org/wiki/Fibonacci_number for more information). Introduce a condition to stop the procedure after N numbers.

Problem 1.19 — Fibonacci numbers 2

- a) Rewrite the script from exercise 1.18 such that it calculates the N th number of the Fibonacci series.
- b) Write a script that generates the sum of the Fibonacci numbers F_N and F_{N+2} . The value of N should be input on the keyboard. Devise a test in the script that validates the value of N ($N \geq 0$).

Problem 1.20 — Series 1

- a) Develop a script that generates the number $e = \exp(1)$ by using the series:

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

Try doing this without constantly recalculating $x^n/n!$ for each n (i.e., use the result from a previous step, $r_{n-1} = x^{n-1}/(n-1)!$, to calculate $r_n = x^n/n!$). Start with **format long!**

- b) Improve the script such that it will stop automatically once it is within a certain accuracy of the 'real' value **exp(1)**.
- c) Change the script to calculate $\exp(x)$ for any x .

Problem 1.21 — Series 2

- a) Develop a script that generates the number π using the following series:

$$\arctan(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{2n+1}.$$

Use **format long**. For which value of n is π accurate up to 5 digits?

- b) Change the script so that it will stop once it has reached a certain accuracy with respect to the 'real' value of π as predefined in variable **pi**.

Problem 1.22 — Multiplication tables

Write a script that generates and displays the multiplication tables ('tafeltjes') up to and including 10. Let each table be preceded by a text 'Multiplication table of x ', with $1 \leq x \leq 10$. Use comments. To put multiple numbers on a line, use **sprintf** with **disp** or **fprintf** (look these up in help)

Problem 1.23 — Binary numbers

Develop a script that asks for an integer number as input. This number must be converted into the binary format. Let the binary digits appear in such a way that the least significant bit is displayed first. Use comments! Example: if the input would be 18, the binary format is 10010. The output of the script is

```
0
1
0
0
1
```

Problem 1.24 — Leap year

Write a script which tells you whether a certain year (ask for input) is a leap year. More information can be found at http://en.wikipedia.org/wiki/Leap_year.

Chapter 2

Matrices

2.1 Introduction

Chapter 1 covered basic Matlab skills, including controlling program flow. There we mostly dealt with individual numbers stored in individual variables. However, the majority of applications involves lists of numbers in the form of one-dimensional or multi-dimensional arrays of numbers or text. Examples are the processing of images recorded with a CCD camera, voltages recorded as a function of time with an oscilloscope, or simulation results developed for multiple values of the important model parameters. Therefore, numerical processing and visualization often requires dealing with arrays and matrices. Fortunately, MATLAB (Matrix Laboratory) is particularly suitable for such computing and visualization tasks, and includes many built-in commands and functions that can be applied not only to individual numbers, but equally well to arrays of numbers. This is the subject of the current Chapter.

2.2 Arrays and graphical output

Let's imagine that we want to create a graph of a function $f(x)$ using MATLAB. One way to do this is to define a function `f.m` as done before in section 1.7. We choose a large number of different values for x which we assign to variables `x1`, `x2`, `x3`, etc. After that we can calculate the function values for each x : $f(x)$: `y1=f(x1)`, `y2=f(x2)`, `y3=f(x3)`, etc. These values can then be plotted. It is clear that this approach would require a lot of coding.

Thankfully there is a faster way to do this. First, it is possible to collect all the values of x into a single variable `x`, containing an *array* of multiple elements. The function `f` can be written such that it can handle entire arrays of x values as input and returns an array as output. The graphical representation is then done with a single command.

As an example we will create a graph of $f(x) = \sin(x)$. First we define some x values, which we assign to the variable `x`:

Exercise 2.i — Execute the following commands. Explain what happens!

```
>> x = [-5 -4 -3 -2 -1 0 1 2 3 4 5]
>> x(3)
>> x(11)
>> x(0)
>> x(25)
```

The brackets [and] indicate an array of numbers. The individual elements are separated by a space or a comma. This array has 11 elements. Individual elements can be evaluated by entering $x(\#)$, with $\#$ the index. Indices smaller than 1, or larger than the total number of elements, generate an error.

Sometimes the required number of elements is large, so that manually typing them is not practical. We can also define the entire array x at once:

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

The notation using `:` has already been explained in section 1.8.2. Here, the step size is 1, its default value.

Exercise 2.ii — Now that the x values are known, we need to calculate $\sin(x)$ for each element of x . The MATLAB function `sin` can do this. Type:

```
>> y=sin(x)
```

With a *single* command, the function values y for *all* elements of x are calculated! This means that it is no longer necessary to write a `for` loop that calls the function `sin` for each x . y is now also an array, as can be seen in the Workspace. Type the following:

```
>> plot(x,y)
```

The command `plot(x,y)` is used to graphically display the points (x,y) . After executing this command, a new window called Figure 1 appears, containing a graph as in figure 2.1. We can see that the points are connected with a blue line. We can also see that the graph is very inaccurate, it does not resemble the graph of the sine function. This is due to the large step size in x . In order to get a smoother curve we need to use a smaller step size. Additionally, we know that the extrema and zeroes of the sine function are found at the ‘special’ points $x = n\pi/2$ (with n an integer) where the function is zero or ± 1 . For the most accurate graphical representation it is wise to explicitly use these values.

Exercise 2.iii — To obtain a better resolution we now choose the step size for x such that there are 100 steps between $-\pi$ and π . To achieve this we multiply the array `-1:0.02:1` with `pi`.

```
>> x=(-1:0.02:1)*pi;
```

Use the semicolon (there is no need to see all the elements of x on the screen)! The array x of 11 elements is now replaced by a much longer array of 101 elements (check the Workspace). Note that multiplication of an array does not require a loop. This type of multiplication (multiplying each value of an array by the same number) is already defined in MATLAB, just like the sine function. The same holds for dividing by a constant, and adding or subtracting a constant.

Calculate the new function values y , and display the result in a new graph.

Figure 2.2 shows a curve that already looks a lot more like a real graph of the sine. The step size could be made even smaller if necessary.

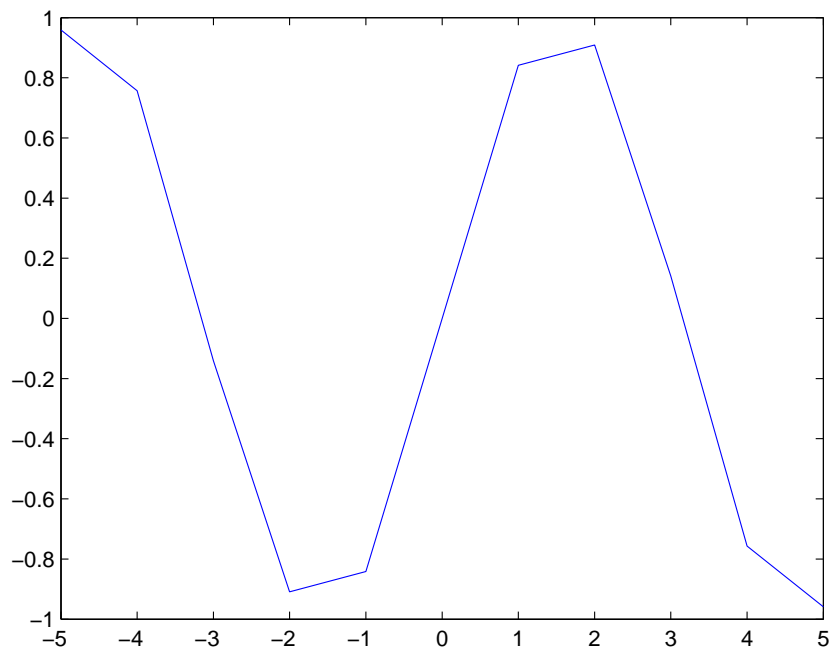


Figure 2.1: Graph of $\sin(x)$ using x values with steps of size 1.

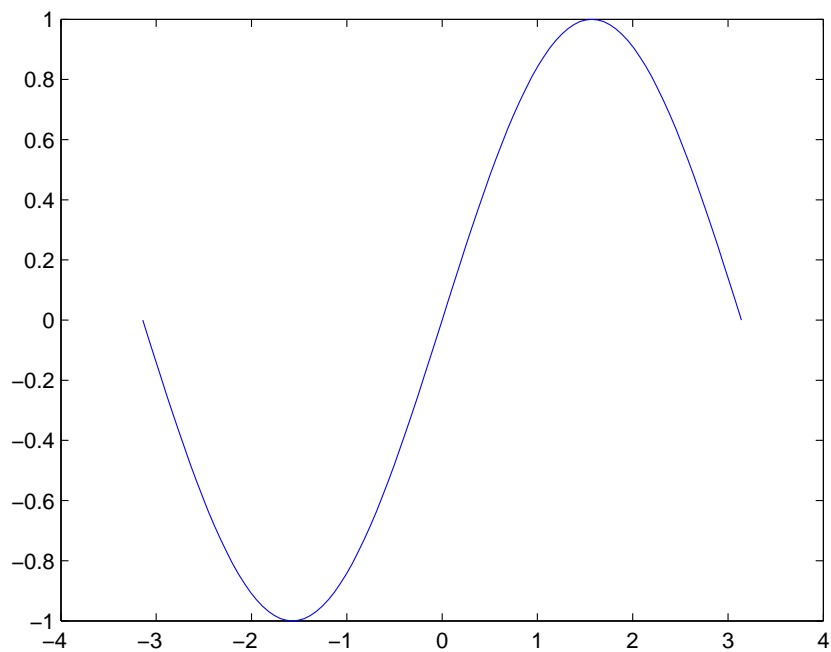


Figure 2.2: Graph of $\sin(x)$ with 101 x values in steps of 0.02π between $-\pi$ and π .

2.2.1 Graphs, continued

Graphs are plotted with blue lines by default. This can be changed by adding an extra argument, for example:

```
>> plot(x,y,'go-.')
```

Table 2.1: Properties for plots.

Colour code	Colour	Line code	Line style
r	red	-	solid
g	green	-.	dash-dotted
b	blue	--	dashed
c	cyan	:	dotted
m	magenta		
y	yellow		
k	black		
w	white		

Symbol code	Symbol
+	plus sign
o	circle
*	asterisk
.	point
x	cross
s	square
d	diamond
^	up triangle
v	down triangle
<	left triangle
>	right triangle
p	pentagram
h	hexagram

This will plot a green dash-dotted line with circles on the data points. The argument between the '' consists of (up to) 3 parts: colour, symbol shape, line style. Each of the three parts can also be left out. By default the colour is blue, the line is solid and no symbols are added. In table 2.1 an overview of these properties is shown.

As you executed the second `plot` command you probably noticed that the first graph has been replaced by the new graph. To prevent this and plot multiple graphs in one figure, we can enter the following command after the first `plot` command:

```
>> hold on
```

By executing the command `hold off` we return to the default behaviour, old graphs are replaced by new ones. Multiple graphs can also be plotted with one command:

```
>> plot(x1,y1,'b-',x2,y2,'rs')
```

The graph of y_1 as a function of x_1 is a solid blue line, y_2 as a function of x_2 is drawn using red squares.

Besides the command `plot` there are also commands that plot on a logarithmic scale.

- `semilogx(x,y)` logarithmic x axis and a linear y axis;



Figure 2.3: Buttons in the MATLAB Figure window.

- `semilogy(x,y)` linear x axis and a logarithmic y axis;
- `loglog(x,y)` both axes are logarithmic.

All plotting options for `plot` can also be applied in these commands.

Exercise 2.iv — We define the following functions:

$$f(x) = x^3 \quad g(x) = 2^x \quad h(x) = \log_{10}(x).$$

Plot these functions, both using a for-loop and not using a for loop by defining an array of number for x instead, to fill arrays for f , g and h . For each function, determine which of the plot commands results in a straight line. For x use a range of 1 to 100 with some reasonable step size (or try something else).

Additional annotation for graphs can be inserted by using the following commands.

- Axis labels: `xlabel('x axis label')` and `ylabel('y axis label')`;
- Title: `title('title')`;
- Additional graph annotation: `text(x,y,'text')` adds text on the coordinates (x, y) within the graph.

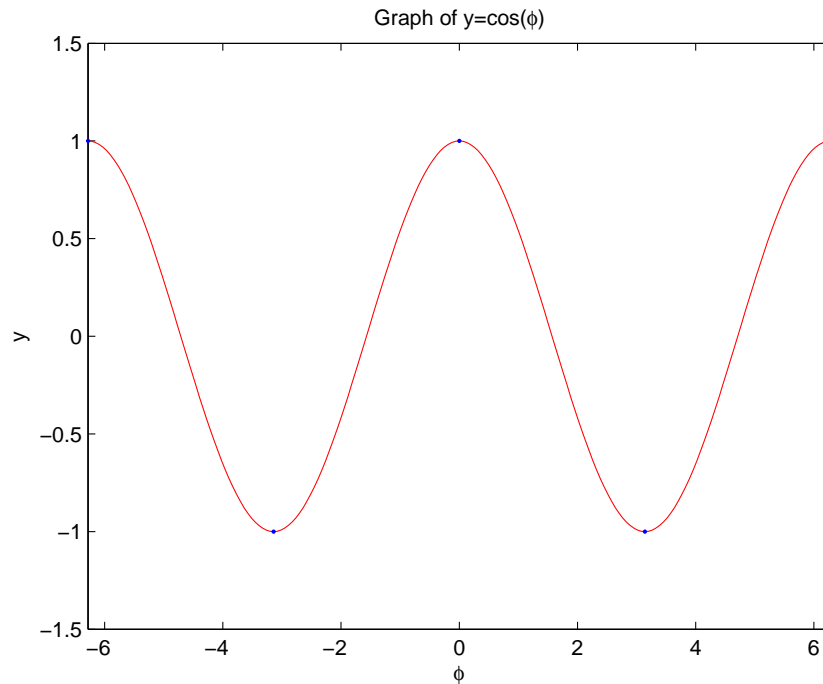
In all these text annotations, Greek characters can be inserted by using `\alpha`, `\beta`, `\gamma`, etc. Uppercase Greek characters follow the same procedure, i.e. `\Gamma`, etc. It is also possible to choose from more additions using the option Insert in the Figure window. In particular, the option 'Insert' \rightarrow 'Legend' can be used to insert a figure legend. Other options are available using the buttons in the Figure window, shown in figure 2.3. From left to right, the first five buttons are the most important: open a new Figure window, open a previously saved figure, save the current figure on the hard disk, print the figure, tool to select a curve. When the last option (the mouse arrow) is selected, the properties of a curve (colour, style, symbol, etc.) can be changed by clicking on a curve with the right mouse button. This is true for practically all parts of the graph, including the text in the figure legend. These options are also available via 'Edit' \rightarrow 'Axes Properties' where the lengths of the displayed axes can be changed, text fonts can be changed, axes can be switched between linear and logarithmic, etc. A complete overview of the options can be found at 'Help' \rightarrow 'Graphics Help'.

Figures can be saved as MATLAB figures with the extension `.fig`, or as regular pictures (JPG, BMP, PDF etc) via 'File' \rightarrow 'Save As...'.

Exercise 2.v — Write a script that carries out the following actions:

Draw a graph of the function $y = \cos(\phi)$ for $-2\pi \leq \phi \leq 2\pi$ with a red line. Add blue points at the minima and maxima. Give the graph the title: Graph of $y=\cos(\phi)$. Add the correct axis labels.

Compare your result with figure 2.4. Now change the range of the axes by adding the following line to your code:

Figure 2.4: Graph of $y = \cos(\phi)$.

```
axis([-2*pi 2*pi -1.5 1.5])
```

Rerun the script. What has changed?

The command `axis` sets the ranges of the axes:

```
>> axis([x_min x_max y_min y_max])
```

It sets the range of the x axis from `x_min` to `x_max` and the range of the y axis from `y_min` tot `y_max`. Try it out.

2.3 Tables of numbers: matrices

2.3.1 Simple matrices

All of the arrays that we have seen so far are actually simple matrices (tables filled with numbers) with only one row of numbers. Matrices are a commonly used tool to solve problems in mathematics and physics. Therefore it is useful to be able to work with matrices. One of the main advantages of MATLAB is that it is intended to work with matrices, making it the number one choice when solving matrix problems.

Let's define an array `x`:

```
>> x=-5:5
```

The following command shows the number of elements

```
>> size(x)
ans =
1 11
```

This result shows that **x** consists of 1 row and 11 columns.

Exercise 2.vi — Variables can also consist of multiple rows and/or columns:

```
>> y = [1 2 3; 4 5 6; 7 8 9]
>> size(y)
```

With this definition we see that variable **y** consists of 3 rows and 3 columns. A row is filled until the semicolon, after which a new row is started. Row 2 is also filled with 3 elements, until the next semicolon is found. MATLAB now continues with row 3, which is also filled with 3 elements.

Exercise 2.vii — What is the result of the following commands?

```
>> y(3,2)
>> y(1,1)
>> y(5,3)
>> y(2,1)
```

Can you figure out what the numbers between parentheses mean?

The first index between parentheses defines the row number, the second the column number. This is the mathematical convention that is always used for matrices. Elements outside the boundaries do not exist!

Within a matrix, all rows, and all columns, must contain an equal number of elements.

```
>> a=[1 2; 3]
Error using vertcat
CAT arguments dimensions are not consistent.
```

In the previous example, the first row would contain 2 elements whereas the second row would only contain a single element. This results in an error.

There are several functions that can be used to quickly fill a matrix:

```
>> a=zeros(2,3)
>> b=ones(4,5)
```

Variable **a** is now a matrix consisting of 2 rows and 3 columns; all elements are 0. Variable **b** is also a matrix, now consisting of 4 rows and 5 columns. All elements of **b** are 1. The command **diag** can also be useful - try it out.

Exercise 2.viii — Execute the following commands:

```
>> d=1:10
>> c=diag(d)
```

What does **diag** do?

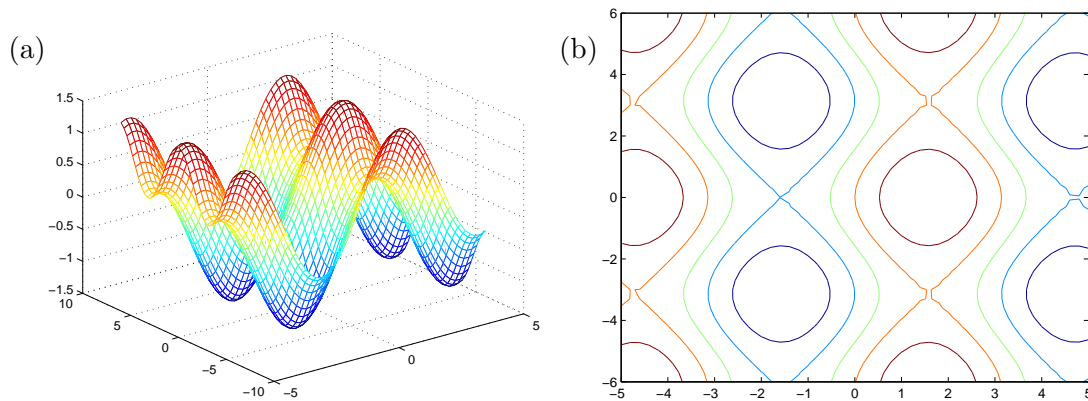


Figure 2.5: Two examples of plots of a function of two variables: (a) `mesh`, (b) `contour`.

2.3.2 Functions of multiple variables

One application of matrices can be found in the graphical representation of functions of two variables. For instance, $f(x, y) = \sin(x) - \frac{1}{2} \cos(y)$ has a function value $f(x, y)$ in each point (x, y) . We want to plot $f(x, y)$. First, we need to gather many points (x, y) for the evaluation of $f(x, y)$. The function `meshgrid` is useful for this step.

Exercise 2.ix — Enter the following commands:

```
>> [x,y] = meshgrid(-5:0.2:5, -6:0.3:6);
>> size(x)
>> size(y)
>> f=sin(x)+0.5*cos(y);
```

What do you notice?

The function `meshgrid` returns *two* matrices, `x` and `y`. Each *row* of `x` is equal to `-5:0.2:5`. Each *column* of `y` is equal to `-6:0.3:6`. These are the x and y coordinates at which the function is evaluated. The MATLAB functions `sin` and `cos` are, like almost all MATLAB functions, capable of handling matrices as input variables. The command `g=sin(x)` with `x` a matrix results in a matrix `g` with the same dimensions as `x`.

Plotting functions of two variables can for example be done using a three-dimensional mesh plot (figure 2.5a)

```
>> mesh(x,y,f)
```

or with contours: (figure 2.5b)

```
>> contour(x,y,f)
```

See the Help pages for extra information.

Exercise 2.x — Make a plot of the function `f` using filled contours. The command is `contourf`. Then enter the command `colorbar`.


Exercise 2.xi — Develop a script that plots the following function:

$$f(x, y) = 2 \exp(-\frac{1}{10}x) + \cos(3y)$$

Use the command `surf(x,y,f)`. Use -10 and 10 as the boundaries for the x and y coordinates and use appropriate step sizes. How can the axis ranges for the three axes be changed? Add annotation for the three axes. (*hint*: use `zlabel`). To set the relative scaling of the three axes use `daspect([a b c])`. Distances a on the x axis, b on the y axis and c on the z axis will be plotted with equal on-screen length. Also add a `colorbar`. Save the figure as a JPG file.

Exercise 2.xii — We will embellish the plot of the previous assignment. Use the following commands in your script:

```
surf1(x,y,f)
shading interp
```

It is also possible to change the viewing angle. Select the button  in the Figure window. Now click on the figure with the left mouse button and hold, then move the mouse. Two numbers are shown: **Az** and **E1**. These are the two angles (azimuth and elevation) that describe the viewing. They are given in degrees. What values can **Az** and **E1** attain? We can also set the viewing angle in our script: `view(Az,E1)`. Save the figure using a nice viewing angle.

2.3.3 Manipulating matrices

We can easily retrieve entire rows or columns from a matrix using the colon operator (`:`).

Exercise 2.xiii — Define the matrix `y` and retrieve some elements:

```
>> y = [1 2 3; 4 5 6; 7 8 9];
>> y(2,:)
>> y(:,3)
```

What is the result? Pay close attention to the *orientation* (row or column) of the result!

It might come in handy to diagonally mirror a matrix, which transforms a matrix with 3 rows and 2 columns into a matrix with 2 rows and 3 columns. This new matrix is then called the matrix transpose.

Exercise 2.xiv — Transposing a matrix can be done by:

```
>> a=[1 2 3; 4 5 6]
>> b=transpose(a)
>> c=[1 2 3]
>> d=transpose(c)
```

Look at the number of rows and columns for each result. Can you describe what `transpose` does?

Table 2.2: Manipulation of matrices.

MATLAB command	Result
<code>a(3,4)=10</code>	The element in row 3 and column 4 of a is equal to 10
<code>b=a(2,:)</code>	b contains the second row of a
<code>b=a(:,3)</code>	b contain the third column of a
<code>a(3,:)=0:2:8</code>	Row 3 of a becomes [0 2 4 6 8]. This can only be done if a is an $n \times 5$ matrix with $n \geq 3$.
<code>a(:,2)=transpose(1:3)</code>	Column 2 of a becomes [1; 2; 3]. Can only be done if a is a $3 \times n$ matrix with $n \geq 2$.
<code>b(4)</code>	The fourth element of row or column b . Can only be done if b is an $n \times 1$ or $1 \times n$ matrix with $n \geq 4$.
<code>b=a([1 4 6],:)</code>	b contains the first, fourth, and sixth row of a . If a is an $n \times m$ matrix, then b is a $3 \times m$ matrix.
<code>a(5:8,3)=transpose(1:4)</code>	Elements 5 to 8 of column 3 in a become [1; 2; 3; 4].

Some matrix-related commands are listed in table 2.2. We will encounter several of these commands in the problems, where their function is illustrated. Of course the commands can also be combined.

Matrices can be merged using similar commands. See the following assignment.

Exercise 2.xv — Write a script with the following commands.

```

a=[1 2 3; 4 5 6]
b=[7 8 9; 10 11 12]
c=[a b]
d=[a; b]
g=[1 2 3]
h=[4 5; 6 7]
k=[g; h]

```

Can you explain what happens in each step?

Exercise 2.xvi — Exercises for manipulating matrices using `:`. See table 2.2. Use only *one* command in each step!

Define a 4×4 matrix **A** filled with ones.

Fill the third column with twos.

Set the elements of row 4 equal to 3, 4, 5, 6.

Set the elements 2,3 and 4 of comlumn 1 equal to 7, 8, 9.

Fill the elements **A**(3,3), **A**(3,4), **A**(4,3), and **A**(4,4) (this is a 2×2 matrix in **A**) with tens.

Check whether the sum of all elements is equal to 78.

2.3.4 Algebraic operations for matrices

Now we are going to use matrices in calculations.

Exercise 2.xvii — What happens?

Subtraction and addition of matrices and numbers.

```
>> a=[1 2 3; 4 5 6]
>> c=0.5
>> a+2
>> a-c
```

Subtraction and addition of matrices.

```
>> b=[12 11 10; 9 8 7]
>> a+b
>> b-a
>> b-transpose(a)
```

Performing calculations with matrices is similar to performing calculations with variables consisting of only one element. When a number is added or subtracted from a matrix, this manipulation is performed on all the elements of the matrix. Adding or subtracting matrices is defined only if the two matrices have the same dimensions.

When a matrix is multiplied or divided by a number, this action is performed on all elements:

```
>> a*c
ans =
    0.5000    1.0000    1.5000
    2.0000    2.5000    3.0000
>> b/3
ans =
    4.0000    3.6667    3.3333
    3.0000    2.6667    2.3333
```

The multiplication, division or exponentiation of matrices is a different story. There is a special multiplication for matrices. This will be explained in section 2.4. An element-by-element multiplication/division/exponentiation per element can be done by including a full stop (.) *before* the operators *, / or ^.

Exercise 2.xviii — Enter the following commands. Do you understand what happens during each step?

```
>> a=[1 2; 3 4];
>> b=[7 8; 9 10];
>> c=a.*b
>> d=a./b
>> f=a.^2
>> g=2.^a
>> h=a.^b
```

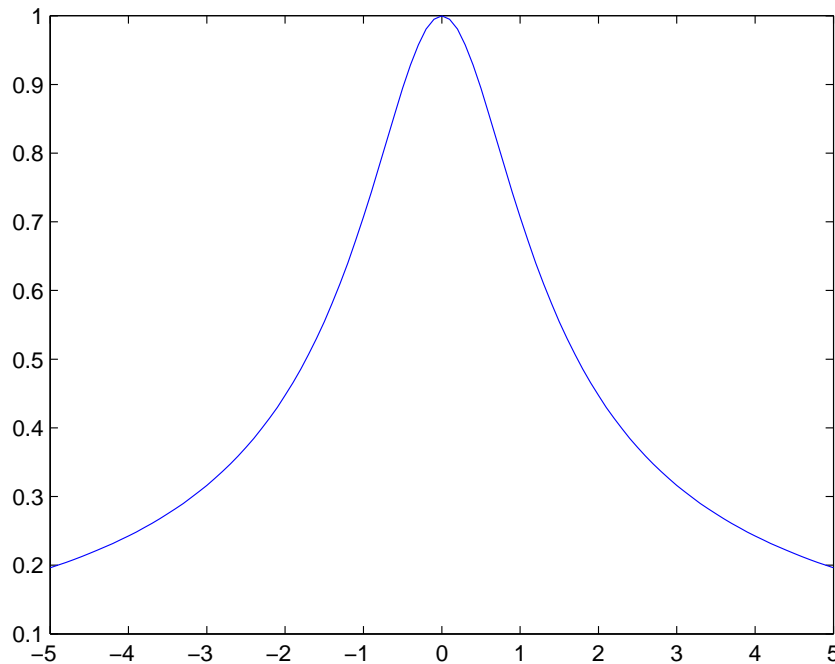


Figure 2.6: Graph of the function $f(x) = 1/\sqrt{1+x^2}$.

The operations are executed element-by-element. For example, `c=a.*b` means that `c(m,n)=a(m,n)*b(m,n)` for all involved `m` and `n`. The other manipulations are carried out in a similar fashion.

Example: a script that plots the function $f(x)$:

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

First we define the x values.

```
x=-5:0.1:5;
```

There are some points of attention for the determination of the function values. It is important to use a full stop before the operations for division and exponentiation. The correct way to write this function is:

```
y=1./sqrt(1+x.^2);
```

If we then use

```
plot(x,y)
```

a graph like figure 2.6 should appear.

Exercise 2.xix — Write a script that plots the function:

$$f(x) = \frac{e^{-x^2}}{\cos x}$$

Table 2.3: Some useful matrix operations. **x** is a matrix. **n** is a variable used to define the direction of operation, **n=1** executes the operation on the columns, **n=2** executes the operation on the rows.

Command	Meaning
<code>sum(x,n)</code>	sum
<code>prod(x,n)</code>	product
<code>max(x,n)</code>	maximum
<code>min(x,n)</code>	minimum
<code>mean(x,n)</code>	mean value
<code>std(x,0,n)</code>	standard deviation (for a description of 0 see <code>doc std</code>)

There are many commands that can be used to calculate a matrix sum, product, average, standard deviation, etc. Several are listed in table 2.3.

Exercise 2.xx — Define an array of numbers **a** that contains the integer numbers from 1 to 10. Calculate the average and the sum using the commands in table 2.3.

Exercise 2.xxi — It takes some caution to apply the commands in table 2.3. To see why, try the following commands.

```
>> b=[1:5; 6:10];
>> sum(b)
>> sum(b,2)
>> sum(sum(b))
```

The operations work as you would naively expect as long as the matrix consists of only *one* row or column. If a matrix consists of more than one row and one column the operations are applied to the columns first, e.g. we get a row matrix containing the sum of the columns. If the operation should be applied to the rows first, a second argument stating the direction should be added. This is done as `sum(b,2)`. The part `,2` states that the operation should be applied to the rows (`,1` is used for columns, and is the default value). If the operation should be applied to both the rows and the columns, the operation can be executed twice `sum(sum(b))`. The first `sum(b)` gives the sum of the columns in a row matrix, the second `sum` sums these values thus giving the total sum of all elements of matrix **b**.

2.3.5 Reading and writing data

Experiments often yield data files with large amounts of numerical values. These values can be imported into MATLAB, so that they can be further manipulated. If each row of such a file contains an equal amount of numbers in text format, importing the data is easy. For example, if `measurement.txt` is such a file, the data can be imported by

```
>> load measurement.txt
```

It is important to give the full file name including extension, otherwise MATLAB will automatically look for a `.mat` file. Using the code above, a new variable `measurement` is created,

containing all the data from the file. If the file contains a header containing explanations of the data, then it might be easier to use the import-tool which can be found under ‘File’ → ‘Import Data’. Using this option a column delimiter (often a comma, semicolon or space character) can be chosen and a header can be skipped when importing data.

Exporting/storing data can be done in a similar fashion. If data has to be used by programs other than MATLAB, it is often best to save it as a text file. The default output of MATLAB is `.mat`, which is binary and not readable for other programs. Thus we add the option `ascii`:

```
>> save data.txt a -ascii
```

This command creates a text file `data.txt` containing the data stored in variable `a`.

Exercise 2.xxii — Create a matrix `F` consisting of two columns. Fill this matrix with the function values of $f(x) = \sin(x)$ and the corresponding values for x . The x values should be placed in the first column, the values $f(x) = \sin(x)$ should be placed in the second column. Save the matrix `F` in a text file `sin.txt`. Import the file in Origin and plot the data.

2.4 Linear algebra

2.4.1 Systems of linear equations

Many problems encountered in mathematics or physics can be written as a system of linear equations. A simple example is a linear fit to two data points $P(2, 3)$ and $Q(5, 1)$. The equation describing the line is $y = kx + l$; we just need to determine the constants k and l . In point P we know that $x = 2$ and $y = 3$, so that $2k + l = 3$. Doing the same for point Q we find $5k + l = 1$. The system of linear equations is

$$\begin{cases} 2k + l = 3 \\ 5k + l = 1 \end{cases}$$

For this simple example, the solution is easily found. Rewrite the first equation as $l = 3 - 2k$ and substitute it into the second equation. We immediately find that $l = \frac{13}{3}$, and thus $k = -\frac{2}{3}$. However, as the number of equations in the system increases, it becomes increasingly impractical to solve it manually. MATLAB can help here. First, we need to rewrite the system of linear equations as a product of a matrix and a vector.

2.4.2 Matrix product

The matrix product is defined as follows. Suppose we have an $m \times n$ matrix A and an $n \times p$ matrix B . We use subscript indices to designate individual matrix elements: A_{ij} is the element of A in row i and column j . This correspond with the MATLAB code `A(i,j)`. The matrix product is now defined as $C = AB$, where each element of C is defined as

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}.$$

The element in row i and column j of C is in effect the vector dot product of row i of matrix A and column j of matrix B . Obviously, the matrix product is only defined for matrices for

which the number of *columns* of the first matrix A is equal to the number of *rows* of the second matrix B . The resulting matrix C is an $m \times p$ matrix. We illustrate the matrix product in figure 2.7, where we multiply the matrices

$$A = \begin{pmatrix} 5 & -3 & 8 & 2 \\ 4 & -6 & 0 & 3 \\ -1 & 3 & 6 & -3 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 2 & 4 \\ -5 & -2 \\ 3 & -6 \\ 9 & -4 \end{pmatrix}.$$

$$\begin{pmatrix} 5 & -3 & 8 & 2 \\ 4 & -6 & 0 & 3 \\ -1 & 3 & 6 & -3 \end{pmatrix} \begin{pmatrix} 2 & 4 \\ -5 & -2 \\ 3 & -6 \\ 9 & -4 \end{pmatrix} = \begin{pmatrix} 67 & -30 \\ 65 & 16 \\ -26 & -34 \end{pmatrix}$$

Figure 2.7: A matrix product example. The purple element on the right-hand side is located in the *second* row and the *first* column of the matrix. It is therefore obtained from the vector dot product of the *second* row (red) of matrix one on the left-hand side and the *first* column (blue) of matrix two on the left-hand side. Indeed, the dot product of the red and blue vectors is $4 \cdot 2 + (-6) \cdot (-5) + 0 \cdot 3 + 3 \cdot 9 = 65$.

The matrix product AB is valid: A is a 3×4 matrix and B is a 4×2 matrix, so the number of rows of matrix A is equal to the number of columns of matrix B . This means that $C = AB$ is a 3×2 matrix. Be careful with the order of the matrices to be multiplied. The matrix product BA is *not valid*. Unlike the multiplication of two numbers, changing the order of the matrices in a matrix multiplication is not allowed! Even if both products exist, in general $AB \neq BA$. For example, given a 3×2 matrix D and a 2×3 matrix E , $F = DE$ is a 3×3 matrix while $G = ED$ is a 2×2 matrix. F and G cannot be equal! In conclusion:

- an $m \times n$ matrix multiplied by an $n \times p$ matrix gives an $m \times p$ matrix;
- even if both AB and BA satisfy the condition above, generally $AB \neq BA$. Only in rare, special circumstances $AB = BA$.

The aforementioned matrix product can be calculated in MATLAB as follows

```
>> A=[5 -3 8 2; 4 -6 0 3; -1 3 6 -3];
>> B=[2 4; -5 -2; 3 -6; 9 -4];
>> A*B
ans =
    67    -30
    65     16
   -26   -34
>> B*A
Error using *
Inner matrix dimensions must agree.
```

The matrix product of A and B can be calculated with $A*B$. We also see that MATLAB generates an error when trying to calculate $B*A$; the dimensions of A and B do not satisfy the criteria for a valid matrix product $B*A$.

Exercise 2.xxiii — We define two square matrices (equal number of rows and columns).

```
>> A=[1 0; 1 1];
>> B=[0 1; 1 2];
```

Now evaluate $C = AB$ and $D = BA$. Compare the results. As is evident, even for simple matrices $AB \neq BA$!

Exercise 2.xxiv — What do the following commands do? Why are the results the same this time?

```
>> E=A.*B
>> F=B.*A
```

Matrices can also be raised to a power, but this operation is only defined for square matrices. The notation A^k means that A is multiplied $k - 1$ times by itself.

$$A^1 = A \qquad A^2 = AA \qquad A^3 = AAA$$

Exercise 2.xxv — Let's try to raise a matrix to a power:

```
>> A=[1 0; 1 1];
>> A^2
>> A*A
```

2.4.3 Solving systems of linear equations

We can rewrite the system of linear equations of section 2.4.1 in a different form using the matrix product. We started off with

$$\begin{cases} 2k + l = 3 \\ 5k + l = 1 \end{cases}$$

We can also write this as a matrix product:

$$\begin{pmatrix} 2 & 1 \\ 5 & 1 \end{pmatrix} \begin{pmatrix} k \\ l \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}.$$

Let's evaluate the left-hand side:

$$\begin{pmatrix} 2 \cdot k + 1 \cdot l \\ 5 \cdot k + 1 \cdot l \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}.$$

These are our two original equations! We now define $A = \begin{pmatrix} 2 & 1 \\ 5 & 1 \end{pmatrix}$ and $x = \begin{pmatrix} k \\ l \end{pmatrix}$ and $b = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$. The system can now be concisely written as

$$Ax = b.$$

Exercise 2.xxvi — Determine the unique parabola passing through the points $P(-2, 4)$, $Q(1, 1)$ and $R(2, -4)$. The equation for the parabola has the general form $y = kx^2 + lx + m$. This problem is described by three linear equations. Which ones? Write the equations in matrix form.

One way to solve these kind of problems *by hand* can be found in appendix A. MATLAB has an operator for solving these kinds of problems.

Exercise 2.xxvii — We use the same matrix A and vector b as in the previous assignment. We find the solution x of the equation $Ax = b$ with

```
>> x=A\b
```

What are the values of the constants k , l , and m ? Check the solution. What should be the result of $A*x$?

Some problems may arise when solving systems of linear equations.

Exercise 2.xxviii — *Problem 1:* A matrix C and a vector d :

$$C = \begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix}; \quad d = \begin{pmatrix} 3 \\ 9 \end{pmatrix}.$$

Find the solution x to the equation $Cx = d$. What is wrong?

If we look closely to which linear equations are defined, we find the problem. The system is:

$$\begin{cases} k + 2l = 3 \\ 3k + 6l = 9 \end{cases}$$

The second equation is actually just three times the first equation. Actually, the same condition is stated twice. Because there are two unknown parameters k and l , and only *one* independent condition, there is an infinite number of solutions. For each arbitrary value of k , $l = \frac{3}{2} - \frac{1}{2}k$ is a solution. This system is *underdetermined*.

Exercise 2.xxix — *Problem 2:* Define matrix E and vector f :

$$E = \begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix}; \quad f = \begin{pmatrix} 3 \\ 3 \end{pmatrix}.$$

Find the solution x to the equation $Ex = f$. What is wrong?

Again we recognize the problem from the system of linear equations. The system is now:

$$\begin{cases} k + 2l = 3 \\ 3k + 6l = 3 \end{cases}$$

The second equation divided by 3 gives $k + 2l = 1$, which is in direct contradiction with the first equation. The system cannot fulfill both requirements at the same time, the system is *overdetermined*.

Systems of linear equations have already been used in the method of least squares, introduced in the course *Exerpimentele Fysica 1*. When fitting a polynomial

$$p(x) = \sum_{n=0}^N a_n x^n$$

of order N through a set of measurement data (x_m, y_m) , the sum \mathcal{S} of the quadratic deviations is introduced as

$$\mathcal{S} = \sum_{m=0}^M (y_m - p(x_m))^2.$$

The deviation is minimised by setting the derivatives of \mathcal{S} to the coefficients a_n equal to zero.

$$\frac{\partial \mathcal{S}}{\partial a_n} = 0, \quad 0 \leq n \leq N.$$

This gives a system of N linear equations, see the reader *Experimentele Fysica 1*, equation (5.47). Systems like these can be solved easily using MATLAB.

2.5 Problems

Problem 2.1 — Arrays

Which MATLAB command should be used to assign the following arrays to \mathbf{x} ? In each case, how many elements does the array contain?

- From 0 to 9, step size 1.
- From -6 to -10, step size -2.
- From 0.5 to 9.0, step size 1.5.
- From 0 to 3π , step size $\pi/6$.
- From $10e$ to $-2e$ step size $-e/13$.

Problem 2.2 — Bubble sort

Bubble sort is a simple sorting algorithm for an array of numbers. It works as follows. Starting from the first element, each element is compared with the next element in the array. If the next element is smaller than the current element, they are swapped. The loop from begin to end of the array is repeated until no swaps are made anymore.

a) Make a design (e.g. on paper) for a function that carries out the bubble sort algorithm on an input array of arbitrary size. What command(s) do you intend to use for the various steps?

b) Write the function and test it! Start by filling an array of length N with random numbers using `rand(1)` (look up `rand` in help) and some kind of loop.

Bubble sort is not efficient. For large arrays the continuous looping makes it run very slowly. Typically, for arrays containing n elements, bubble sort requires a time that scales as n^2 . More efficient sorting algorithms require only a time that scales as $n \log n$. For very large n this makes quite a difference!

Problem 2.3 — Graph of a function 1

Write a script that plots the function $f(x) = \sqrt{1 - x^2}$ for $-2 \leq x \leq 2$. Why does MATLAB give a warning? Add a title and axis labels.

Problem 2.4 — Graph of a function 2

Develop a script that plots the following functions for $-3 \leq x \leq 3$ all together in a single graph. Define your own x -range. Try all 3 types of logarithmic axes.

- a) $f(x) = 2^x$
- b) $g(x) = \frac{1}{3}e^{-x}$
- c) $h(x) = f(x) + g(x)$

Problem 2.5 — Graph of a function 3

Write a script that plots $f(x) = \cos(x)$ with a red dotted line and $g(x) = \sin(x)$ with black diamonds in the same graph. Choose a suitable range and step size for the x axis. Add a legend (look up legend in help).

Problem 2.6 — Special graphs 1

Define an array representing an angle θ with many steps between 0 and 2π . Plot the function $f(\theta) = \cos^2(\theta)$ with the command `polar(theta,f)`. Add axis labels. How should one interpret this graph?

Problem 2.7 — Special graphs 2

The physical property y and its uncertainty Δy have been measured as a function of x :

x	y	Δy
5	6.38	1.06
10	13.01	2.03
15	18.93	2.78
20	25.10	3.89

Define the vectors `x`, `y`, and `deltay` and plot using the command `errorbar`. If you need help with the order of arguments, type `help errorbar` to get more information. For the data points, use red circles. Add axis labels and a title.

Problem 2.8 — Special graphs 3

The results of a die roll can be simulated with `o=floor(rand(1)*6)+1`, where `o` is the amount of pips. Create a script that generates an array `sum_of_two_dice` consisting of 100 numbers, each containing the sum of a roll with two dice. Draw a histogram using `hist(sum_of_two_dice,2:12)`. Create arrays consisting of 1000, 10000, 100000 numbers as well. How many rolls are needed to accurately represent the theoretically expected histogram?

Problem 2.9 — Matrix operations 1

Calculate:

- a) the sum $1^2 + 2^2 + 3^2 + \dots + 50^2$.
- b) the sum $2^0 + 2^1 + 2^2 + \dots + 2^{20}$.
- c) the average value of the row $e^0, e^{1/10}, e^{2/10}, \dots, e^2$.
- d) the maximum value of the row $\ln(\sin(\frac{1}{7}\pi)), \ln(\sin(\frac{2}{7}\pi)), \dots, \ln(\sin(\frac{6}{7}\pi))$.

Problem 2.10 — Series 2

Write a script that asks for a number N , then calculates the following sums using only *one* line of code per sum.

- a) $\sum_{n=1}^N n$

- b) $\sum_{n=0}^N \frac{1}{n!}$
 c) $4 \sum_{n=0}^N \frac{(-1)^n}{2n+1}$

Problem 2.11 — Matrix operations 2

Write a script that executes the following operations:

- Define a 5×5 matrix **A** filled with zeroes.
- Fill **A** with two nested **for** loops in such a way that each element becomes its row number minus its column number.
- What is the average value of the elements of **A**?
- Make the element in row 2 and column 4 equal to 2. Is the sum of all elements of **A** equal to 4?
- Fill the third column with threes. Use only one line of code!

Problem 2.12 — Matrix operations 3

Write a script that defines the following matrix **B**.

$$B = \begin{pmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{pmatrix}.$$

Add lines to the script to

- calculate the sum of each row of **B**. Use only one command.
- calculate the sum of each column of **B**. Use only one command.
- calculate the sum of each of the two diagonals of **B** using a loop.

Problem 2.13 — Matrix operations 4

Define a function that returns the sum of the elements along the diagonals **X**(1,1), **X**(2,2), etc., where **X** is a random matrix used as input for the function. Test this function for matrices that are not square, in other words: **X** has dimensions $n \times m$ with $n \neq m$.

Problem 2.14 — Contours

Define the function $f(x, y) = \exp(-x^2 - y^2)$.

- Draw a contour plot of this function for $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. After that, type **colorbar**. What does this command do?
- It is possible to add an extra argument to **contour** to set the number of contours. Try this.
- * *Extra*: It is also possible give a row/column of function values for which contours are drawn. Draw contours for $f(x, y) = \frac{1}{11}$, $e^{1/3}$, and $\sqrt{\pi/4}$. Use the help function if necessary.

Problem 2.15 — Importing and exporting data

We consider the function $f(x) = x^{2.5}$.

- Plot $f(x)$ for $0 \leq x \leq 3$ with x values in steps of 0.1.
- Export the data to a text file with two columns. The first column contains the x values, the second column contains the corresponding function values.
- Import the text file and add the data to the plot. Are the two data sets the same? Even if you zoom in?

Now consider $g(x) = 1/x$.

d) Export in a text file, for $-1 \leq x \leq 1$, and x step size 0.1, the x values in the first column, and the function values in the second column. What function value is written for $x = 0$?

Problem 2.16 — Matrix product 1

What is wrong with the matrix product $A*B$ of the following matrices? Does the product $B*A$ exist?

a) $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ en $B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix}$.

b) $A = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$ en $B = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$.

c) $A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$ en $B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$.

Problem 2.17 — Matrix product 2

Define the following matrices:

$$A = \begin{pmatrix} -2 & 5 & 3 \\ 6 & -4 & -2 \\ 3 & 8 & -1 \end{pmatrix} \quad \text{en} \quad B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

- Calculate the product $C=A*B$. Compare A with C . What do you see?
- Calculate the product $D=B*A$ and compare A with D . What do you see?
- Check if these observations hold in general for square matrices with ones on the diagonal and zeroes on the other positions.

Problem 2.18 — Matrix product 3

The following matrix is a *rotation matrix*. It represents rotation around the x axis about an angle θ .

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix}.$$

- Create a matrix R for $\theta = \frac{1}{2}\pi$.
- Define a vector $x = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$. What does $y=R*x$ look like?
- Execute the command $z=R^n*x$ for several integer values of n . Explain your results.

Problem 2.19 — * Matrix product 4

Use the MATLAB functions `rand` to fill a square matrix A of dimensions 10×10 with random numbers between 0 and 1. Then calculate A^0 . Run the script several times. Can you explain the results with your knowledge of x^0 , with x a random number?

Problem 2.20 — Linear algebra 1

We have the following set of equations:

$$\begin{cases} 2x & -5y & +3z & = & 1 \\ -x & +4y & -2z & = & -3 \\ & -y & +2z & = & 4 \end{cases}$$

- a) Rewrite this system in matrix form.
- b) Solve the system in MATLAB.
- c) * *Extra*: Solve the system with Gaussian elimination (appendix A). Is your solution correct?

Problem 2.21 — Linear algebra 2

Consider the following three planes in \mathbb{R}^3 :

$$\begin{array}{rrrrcl} 3x & -3y & +5z & -6 & = & 0 \\ -2x & & +5z & -8 & = & 0 \\ x & -y & +2z & -2 & = & 0 \end{array}$$

Create a script that locates the point of intersection of the three planes.

Problem 2.22 — Linear algebra 3

We want to fit a polynomial through the following points $A(-2, 3)$, $B(0, 1)$, $C(3, 5)$, $D(4, 2)$.

- a) How many independent conditions/equations do we obtain from these points?
- b) What is the lowest order that a polynomial fitted through these points can have? (Higher-order polynomials will also fit, but are not completely defined)

Develop a script that

- c) calculates the coefficients of this polynomial.
- d) plots the polynomial and the points A , B , C , D in a graph.

Problem 2.23 — Linear algebra 4

Solve the following system of linear equations:

$$\begin{pmatrix} 1 & 2 & 3 \\ -2 & -1 & 0 \\ 0 & 2 & 4 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 5 \\ -1 \\ 6 \end{pmatrix}$$

Problem 2.24 — * Linear algebra 5

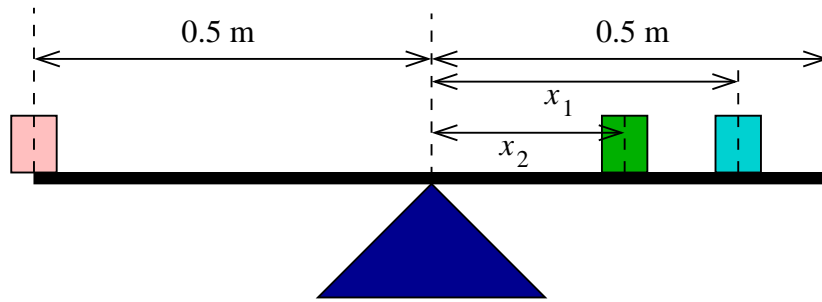
The Taylor series of $f(x) = \sin x$ around $x = 0$ is given by:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

How would you check this formula with a set of linear equations? Create a script that finds the first three terms of the approximation. Also check the accuracy of your result by using `format long`.

Problem 2.25 — * Linear algebra 6

A block of mass 1 kg is attached to the leftmost end of a weightless plank of length 1 m. The plank is balanced on a pointy pillar by two counterweights. One of these weights has a mass of 0.6 kg and is situated at x_1 , the other has a mass of 0.8 kg and is positioned at x_2 . All positions are relative to the tipping point, see the figure.



What is the relation between x_1 and x_2 ? Use a MATLAB script to find the answer! Also look at the boundaries of x_1 and x_2 .

Chapter 3

Numerical evaluation of functions

3.1 Introduction

There exist a large number of physics problems in which it is known that the state of a system can be described by a certain function, and we are interested in finding out particular characteristics of the behavior of the system. As an example, let's consider the motion of a projectile: a ball or bullet of mass m that is launched with some initial velocity v_0 under a certain angle α with the horizontal direction, whose motion is then influenced only by a gravitational force of magnitude $m g$. The vertical position y and horizontal position x of the ball are known functions of the time t after the ball was launched,

$$\begin{aligned}x(t) &= v_0 \cos(\alpha) t, \\y(t) &= v_0 \sin(\alpha) t - \frac{1}{2}g t^2.\end{aligned}$$

In studying the motion of the projectile, we may wish to know

- at what time does the ball reach a particular height y_1 ? Then we must solve the equation $y(t) = y_1$, which is equivalent to finding a zero of the function $y(t) - y_1$.
- what is the maximum height that the ball can reach? Then we have to find the maximum of the function $y(t)$.
- what is the largest distance from the origin that the ball reaches before it hits the ground? The answer is found by maximizing the distance $r(t) = \sqrt{x^2(t) + y^2(t)}$ in an appropriate time interval.
- how much work is done by the gravitational force in a time interval $t_1 < t < t_2$? Then we must evaluate the integral $\int_{t_1}^{t_2} m g v_y(t) dt$, where $v_y(t) = v_0 \sin(\alpha) - g t$ is the vertical velocity of the ball.

Finding zeroes of a function, maximizing or minimizing a function, and integrating a function over a particular interval are common computational tasks when solving physical problems. How to do this with Matlab is the subject of this Chapter.

Using Mathematica, such problems can often be solved analytically. MATLAB does not perform analytical calculations, however. We have seen this when plotting functions in the

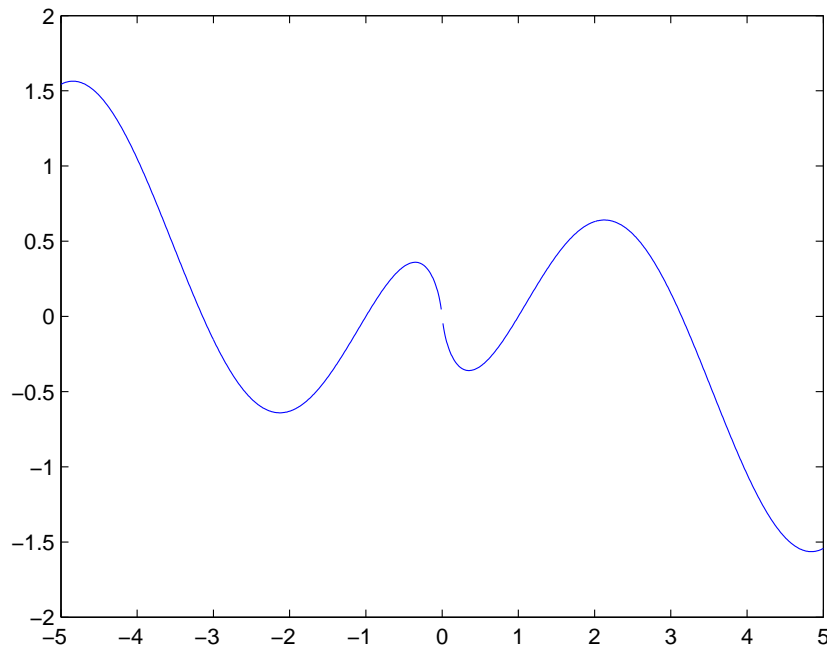


Figure 3.1: Graph of the function $f(x) = \sin(x) \ln|x|$.

previous chapters. In order to get a smooth graph it is necessary to make sure that MATLAB calculates the function value at a sufficiently large number of points. It is unclear beforehand which number of points is sufficient; this must be assessed separately for each problem by trial and error.

There are many recipes for the numerical evaluation of a function $f(x)$. There are iterative schemes for the numerical determination of extrema and zeroes of functions, as well as for numerical integration. These iterative schemes start at a predefined starting point and then ‘walk’ along $f(x)$ in small steps of x until the desired point is reached with enough accuracy. For instance, when searching for a zero, a change in sign of $f(x)$ is a clear indication that a zero has been passed. The next step in the iteration will then be in the opposite direction. By decreasing the step size the point can be found with any desired precision.

Schemes like these are well suited for application in MATLAB. We do not have to implement them ourselves because they are already predefined in Matlab. There is one requirement for the application of these predefined functions: an m-file must be defined that returns a matrix with function values $f(x)$ on input of a matrix with x values. As an example we consider the function

$$f(x) = \sin(x) \ln|x|.$$

Exercise 3.i — Develop an m-file `f.m` that returns the function values for a given input matrix `x`. Also, plot $f(x)$ for $-5 \leq x \leq 5$.

The function is plotted in figure 3.1. The gap around $x = 0$ is due to the fact that $\ln 0$ is undefined. We will use this function in the following sections 3.2–3.4.

3.2 Zeroes

MATLAB is capable of finding the zero, or root, of a function in an interval of coordinates x . However, the function must change sign between the two limits and it needs to be continuous within the interval. From figure 3.1 it is found that f has a zero around $x = 3$.

Exercise 3.ii — Find the zero of $f(x)$ in the interval $x \in [2, 4]$ using the command `fzero`:

```
x0=fzero(@f,[2 4])
```

The `@` symbol defines a so-called *function handle*. MATLAB now expects an m-file `f.m` to be present that returns function values `f(x)` for any input values `x`. If the function is saved as `function.m` in stead of `f.m`, then the function handle is `@function`.

The analytical value for the zero is $x = \pi$, because $\sin \pi = 0$. Let's see how accurate MATLAB can approximate this value.

Exercise 3.iii — Calculate the function value `f` in the point `x0`. Is `x0` a good approximation of π ? Remember that `x0` is a *numerical approximation* of the zero, and will never be *exact*.

`fzero` only finds one zero at a time, even if there are more zeros in the given interval. If f becomes zero, but does not change sign (example: $f(x) = x^2$ at $x = 0$), `fzero` will not find the zero.

Exercise 3.iv — Find the zeroes around $x = -1$ and $x = -3$. What intervals did you use? What happens if the interval contains more than one zero? Is $x = 0$ a zero according to the MATLAB function `fzero`?

3.3 Extrema

Extrema can also be found numerically. MATLAB only has a function `fminbnd` to find *minima*. In figure 3.1 we see that there is a local minimum in the interval $[0, 1]$.

Exercise 3.v — Find this minimum using

```
x1=fminbnd(@f,0,1)
```

Check if the answer is correct by evaluating `f(x1)`.

The first argument in `fminbnd` is again the function handle. The second argument states the lower bound of the interval which will be evaluated, in this case 0. The third argument states the upper bound, in this case 1. The value `x1` for the location of the minimum appears to be a good approximation, judging by the graph. Again, we stress that `fminbnd` only returns one minimum.

Exercise 3.vi — Find the minimum in the interval $[0, 6]$. Which minimum is found? How can we find the second minimum?

In order to find *maxima* of functions we need to apply a trick:

Exercise 3.vii — Define a new function `g.m`, that calls the function `f` with a minus sign. Find the *minima* of $g(x)$. Do the *minima* of $g(x)$ correspond to the *maxima* of $f(x)$?

3.4 Numerical integration

MATLAB can calculate definite integrals (integrals with given limits) numerically. The command is `quad`. `quad` also uses the function handle `@`.

Exercise 3.viii — Evaluate the following integral in MATLAB:

$$\int_0^1 x \, dx .$$

First, create a function `g.m` that returns the function values $g(x) = x$. This function is very simple, but necessary for MATLAB to be able to numerically evaluate the integral. Then, evaluate the integral using:

```
quad(@g,0,1)
```

Check the solution by calculating the integral by hand. Try changing the limits of the integral.

Exercise 3.ix — For the function $h(x) = \sin(x)$, use MATLAB to calculate the following integrals. Check the answers by hand.

$$\int_0^{\pi} h(x) \, dx , \quad \int_{-1}^1 h(x) \, dx , \quad \int_0^{\pi/4} h(x) \, dx .$$

We could have easily calculated these simple integral by hand. However, as function such as $f(x) = \sin(x) \ln |x|$ does not have an antiderivative, so we cannot analytically calculate the integrals

$$\int_{x_0}^{x_1} f(x) \, dx$$

by simply evaluating the antiderivative in the limits of the integral. We must use numerical methods to solve this integral.

Exercise 3.x — Calculate the integral stated above using MATLAB.

It is clear that the singularity in $x = 0$ is no problem for `quad`. One can guess by looking at the graph, and it can also be proven analytically, that

$$\lim_{x \rightarrow 0} f(x) = 0 .$$

Using this additional information the integral can be evaluated numerically. The command `quadgk` can be used for the numerical integration of convergent integrals with $\pm\infty$ as one of the limits.

Exercise 3.xi — Integrate the function $h(x) = 1/x^2$ from 1 to ∞ . First try `quad`, then try `quadgk`. What is the analytical solution?

3.5 Problems

Problem 3.1 — Zeroes 1

$$f(x) = \ln x + x.$$

- For which values of x is $f(x)$ defined?
- Plot $f(x)$.
- Determine the zero of the function. Check by looking at the graph.

Problem 3.2 — Zeroes 2

Plot:

$$f(x) = \frac{\pi}{4} + \sin(x) + \frac{1}{11} \cos(2x)$$

for the interval $(-\pi, \pi)$. Determine all zeroes of $f(x)$ within this interval.

Problem 3.3 — Extremes 1

Plot, and find all minima of, the function

$$f(x) = -(x+1)^2 \exp(-x^2).$$

Problem 3.4 — Extremes 2

Plot

$$f(x) = \ln(x^2 \cdot |\sin(x)|)$$

on the interval $(-6, 6)$ and find all maxima.

Problem 3.5 — Consider the motion of a projectile as described in Sec. 3.1. Assume that $v_0 = 100$ m/s, $\alpha = 30^\circ$ and $g = 9.81$ m/s². Calculate:

- the time at which the ball reaches a height of 10 m,
- the maximum height the ball reaches,
- the largest distance from the origin ($x = y = 0$) that the ball attains before it hits the ground.

Plot the trajectory of the ball to see if the answers are correct!

Problem 3.6 — Numerical integration 1

Calculate the integral

$$\int_0^1 \sin(\pi x^3) dx.$$

Problem 3.7 — Numerical integration 2

Calculate the integral

$$\int_{\pi}^{2e} \cos^2 \left[\exp \left(-\frac{2}{x} \right) \right] dx.$$

Problem 3.8 — Numerical integration 3

Calculate the integral

$$\int_0^{\infty} e^{-x} dx.$$

Problem 3.9 — Consider a small block of mass $m = 1$ kg that can move on a frictionless track. The block is connected to a spring with spring constant $k = 2.5$ kg/s². In addition, a damper is placed with damping coefficient c . The force on the block due to the damper is $F_d = -cv$ with v the velocity of the block and $c = 0.2$ kg/s.

The velocity of the block is given by

$$v(t) = -x_0 \exp\left(-\frac{c}{2m}t\right) \left[\frac{c}{2m} \cos(\omega t) + \omega \sin(\omega t)\right] \quad (3.1)$$

where $x_0 = 0.1$ m and $\omega = \sqrt{k/m}$. The period T of the motion is $T = 2\pi/\omega$.

The work W delivered by the damper in the temporal interval $[t_1, t_2]$ is given by the integral

$$W = \int_{t_1}^{t_2} F_d v dt = \int_{t_1}^{t_2} -c v^2 dt. \quad (3.2)$$

Make a script that calculates the work delivered by the damper during the first period of the oscillation, from $t = 0$ to $t = T$. Save any required additional functions!

Problem 3.10 — Numerical integration 4

Calculate the integral

$$\int_1^{\infty} \frac{1}{x} dx.$$

What is going on?

Problem 3.11 — * Numerical integration 5

Define a function with argument \mathbf{r} that calculates the area of a circle with radius \mathbf{r} . Which function must be integrated?

Problem 3.12 — * Numerical integration 6

Calculate:

$$\int_{-\infty}^{\infty} e^{-x^2} dx.$$

Compare your numerical result to the exact result.

Chapter 4

Differential equations

4.1 Introduction

Many physical phenomena are described by differential equations (DEs): relations specifying the behavior of the derivatives of a physical quantity, for example with respect to time or position. Perhaps the most familiar example so far is Newton's second law from classical mechanics, for instance as applied to the motion of a projectile or a pendulum. The result in each case is a first order DE for the velocity of the object or a second order DE for the position of the object. Similarly, in quantum mechanics the wavefunction of a particle follows from a second order DE known as Schrödinger's equation, while the properties of electromagnetic fields follow from a set of coupled, first order DEs known as Maxwell's equations.

Since DEs describe such a multitude of phenomena, numerical solutions to DEs play an important role in simulations and computations. Therefore it is no surprise that MATLAB contains a number of functions and commands that allow us to solve a variety of DEs, and some of these are introduced in this Chapter.

4.2 A first-order differential equation

A differential equation (DE) is an equation that relates the value of a function to a derivative(s) of that function. In this chapter we will only consider *ordinary* DEs (ODEs), in which all derivatives are taken with respect to only one variable. There are also *partial* DEs with partial derivatives with respect to multiple variables.

Let's start once more with the example of projectile motion: a ball of mass m is thrown upwards with initial speed v_0 . Under the influence of gravity $F_z = mg$ the velocity $v(t)$ will change according to Newton's second law:

$$F_z = ma = m \frac{dv}{dt} \Leftrightarrow \frac{dv}{dt} = g.$$

We have chosen the upward direction as positive, so $g = -9.81 \text{ m s}^{-2}$. This problem is sufficiently simple that an analytic solution is available. For initial condition $v(t) = v_0$ this is given by

$$v(t) = v_0 + gt.$$

Note that the solution does not depend on the mass m .

Exercise 4.i — Now let us compute a numerical solution for the ODE

$$\frac{dv}{dt} = g \quad (4.1)$$

with MATLAB and compare the result to the analytical solution. First create an m-file `dv.m` containing the code:

```
function dvdt = dv(t,v)
g=-9.81; % gravity
dvdt=g; % the derivative of the velocity is equal to the gravity
```

For a given time `t` and velocity `v` this function returns the derivative `dvdt`, in this case simply the value `g`. Note that there are two function arguments: both time and velocity, even though for this particular ODE the derivative in the right hand side of Eq. 4.1 is independent of `t` and `v`. MATLAB always uses this standard function format to define an ODE because, in a more general case, the derivative that is being calculated could depend on these arguments. Note that the arguments correspond to the quantities in the denominator (`t`) and numerator (`v`) of the derivative in the ODE (left hand side of Eq. 4.1).

Solve the ODE for $0 \leq t \leq 2$ s and $v_0 = 10$ m s⁻¹ with the command

```
[t,v]=ode45('dv',[0 2],10);
```

The command `ode45` works on the m-file `dv.m` (that is why it has to have the structure as shown above; note that `dv.m` is passed as an argument to `ode45` as `'dv'`), and solves the ODE for time `t` in the interval `[0 2]` with initial condition `v=10`. The function returns two vectors `t` and `v`, that contain the values of the velocity `v` for the corresponding time values `t`. Plot the numerical result together with the analytic solution.

As a second example, we investigate the velocity of a steel sphere that is dropped into a viscous liquid. The friction force F_w scales linearly with the velocity v of the sphere, but is directed opposite to the direction of motion. Assume that $F_w = -cv$ with c a positive constant. Applying Newton's second law, we find:

$$mg + F_w = m \frac{dv}{dt} \leftrightarrow \frac{dv}{dt} = -\frac{c}{m}v + g,$$

with $g = -9.81$ m s⁻² as the gravitational acceleration. The positive direction is upwards again. Now assume the sphere is released from rest: $v(t=0) = 0$, and that $c/m = 10$ s⁻¹. Again, the analytic solution is known:

$$v(t) = A \exp\left(-\frac{c}{m}t\right) + \frac{mg}{c}.$$

From the initial condition $A = -mg/c$ it follows that

$$v(t) = \frac{mg}{c} \left[1 - \exp\left(-\frac{c}{m}t\right)\right].$$

We can see that the velocity exponentially approaches a constant velocity $v(t \rightarrow \infty) = mg/c = -0.981$ m s⁻¹. The characteristic time scale of the exponential function is $m/c = 0.1$ s.

Now let us solve the ODE numerically using MATLAB. To do this we first define a function `dv2.m` that contains the ODE:


```
function dvdt = dv2(t,v)
g=-9.81; % gravity
cbym=10; % the constant c/m
dvdt=-cbym*v+g; % the right hand side of the ODE
```

This function calculates the time derivative of the velocity \mathbf{dvdt} for a given velocity \mathbf{v} and time \mathbf{t} and returns the result. As before, the ODE is not explicitly dependent on time \mathbf{t} , but now it *is* dependent on the instantaneous velocity $\mathbf{v}=v(t)$!

Exercise 4.ii — Solve this ODE in the time interval $[0 \ 1]$ with initial condition $\mathbf{v}(0)=0$. Plot the analytical and numerical solutions in one graph.

There are other commands that can be used to solve ODEs in MATLAB. If we replace `ode45` by `ode23` we get a solution faster, but with less accuracy. In general, it is advised to start with `ode45`. The difference in calculation time is often small, especially for the cases treated in this course.

Exercise 4.iii — Solve the ODE for the sphere in a viscous fluid using `ode23`. Compare your results with those obtained using `ode45`. Which method is more accurate? Plot the three solutions (analytical, from `ode23`, from `ode45`) in a single graph.

4.3 Systems of ordinary differential equations

Let us go back to the vertically launched ball in the previous section. Besides the velocity $v(t)$, we are also interested in the vertical position $y(t)$. The following equation describes the evolution of the vertical position:

$$\frac{dy}{dt} = v. \quad (4.2)$$

Equations (4.1) and (4.2) are coupled: the velocity v at a certain time can be found with (4.1), while the position y evolves according to (4.2). It is possible (and generally necessary) to solve these ODEs together. To do this we define a vector ϕ with components y and v as in

$$\phi = \begin{pmatrix} y \\ v \end{pmatrix}.$$

The time derivative of ϕ is then

$$\frac{d\phi}{dt} = \frac{d}{dt} \begin{pmatrix} y \\ v \end{pmatrix} = \begin{pmatrix} \frac{dy}{dt} \\ \frac{dv}{dt} \end{pmatrix}.$$

We rewrite `dv.m` as follows:

```
function dphidt = dv(t,phi)
% vector phi consists of two elements:
% element phi(1) is the position y
% element phi(2) is the velocity v
% vector dphidt also consists of two elements:
```

```
% element dphidt(1) is the derivative dy/dt
% element dphidt(2) is the derivative dv/dt
g=-9.81; % gravity
dphidt=zeros(2,1); % define a column vector with two elements
dphidt(1)=phi(2); % the time derivative of position is equal to the velocity
dphidt(2)=g; % the derivative of the velocity is equal to g
```

Because there are now two variables y and v involved in the problem, both dependent on t , two initial conditions are required! We choose $v(t=0) = 10 \text{ m s}^{-1}$ and $y(t=0) = 0 \text{ m}$.

Exercise 4.iv — Now solve the system of ODEs defined above using:

```
[t,phi]=ode45('dv',[0 2],[0; 10])
```

The initial condition is now a column vector consisting of two elements $[0; 10]$. Plot the graph of $y(t)$. Also solve the ODEs by hand, and plot this solution in the same graph. When does the ball hit the ground? Plot the exact and numerical solutions for $v(t)$ in a second graph.

Exercise 4.v — Now the ball is shot from a height $y(t=0) = 20 \text{ m}$ with an initial velocity of $v(t=0) = 5 \text{ m s}^{-1}$. What changes? At what time does the ball hit the floor this time (look at the graph)?

The well-known example of the mathematical pendulum,

$$\theta'' + \frac{g}{\ell}\theta = 0,$$

is a bit more complicated. Here θ is the angular displacement, g the gravitational acceleration and ℓ the length of the pendulum. The initial conditions are $\theta(t=0) = 0.5 \text{ rad}$ and $\theta'(t=0) = 0$. We set $g/\ell = 10 \text{ s}^{-2}$. Of course we can find an exact solution: we know that $\theta(t) = A \cos(\sqrt{g/\ell} t) + B \sin(\sqrt{g/\ell} t)$ satisfies the ODE. With the correct initial conditions we find:

$$\theta'(t=0) = 0 \quad \Rightarrow \quad B = 0,$$

$$\theta(t=0) = 0.5 \quad \Rightarrow \quad A = 0.5.$$

So the correct exact solution is $\theta(t) = 0.5 \cos(\sqrt{g/\ell} t)$.

Now we will solve this ODE numerically with MATLAB. First, we need to rewrite the second-order ODE to the form

$$\phi' = f(t, \phi), \tag{4.3}$$

where ϕ can be a vector. Because there is a second-order derivative in the ODE, ϕ' must contain the derivative θ' . As such we can effectively introduce a second-order derivative on the left-hand side. Looking at the original ODE, we can deduce that the function on the right-hand side must contain both θ and θ' . We define a vector ϕ that contains both the angle θ and the angular velocity θ' of the pendulum:

$$\phi = \begin{pmatrix} \theta \\ \theta' \end{pmatrix}; \quad \phi' = \begin{pmatrix} \theta' \\ \theta'' \end{pmatrix}.$$

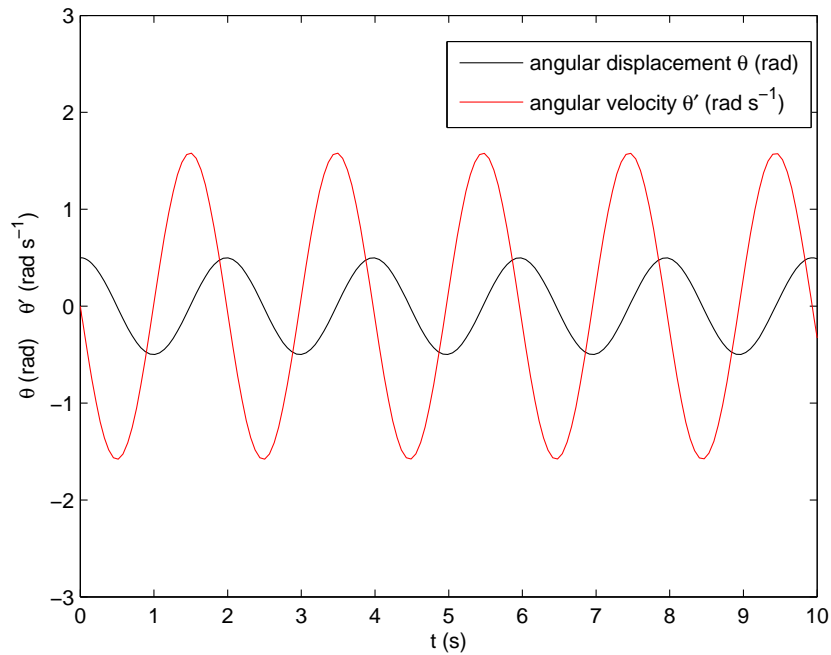


Figure 4.1: Graphs of the solution of the ODE for the pendulum displaying both θ and θ' .

We then arrive at

$$\begin{pmatrix} \theta \\ \theta' \end{pmatrix}' = \begin{pmatrix} \theta' \\ -\frac{g}{\ell}\theta \end{pmatrix},$$

where we have used the fact that the angular velocity is the time derivative of the angular displacement (first line); the second line represents the original ODE.

Exercise 4.vi — Develop a new function `dv3.m` that returns the derivative ϕ' for this vector ϕ . ϕ' is also a vector. What quantities do the elements of ϕ represent? Solve the ODE in the time interval $[0, 10]$ with initial condition $\theta(0) = 0.5$ rad, $\theta'(0) = 0$. Plot the graphs for $\theta(t)$ and $\theta'(t)$. Compare your results with Fig. 4.1.

Exercise 4.vii — Change the value of g/ℓ and compare the numerical result with the exact solution.

Rewriting higher-order ODEs to a system of the form (4.3) is always possible. The required number of equations follows directly from the highest order of derivation in the ODE. For the pendulum, due to the second-order derivative, two equations and two initial conditions are required.

4.3.1 * *Stiff differential equations*

Some ODEs are hard to solve with standard numerical procedures. If the solution presents large deviations within a small time step (large derivatives are involved), the time step size needs to be very small to accurately represent the solution. This means that it will take a lot of calculation time to solve such a system. These kinds of ODEs are also called *stiff*

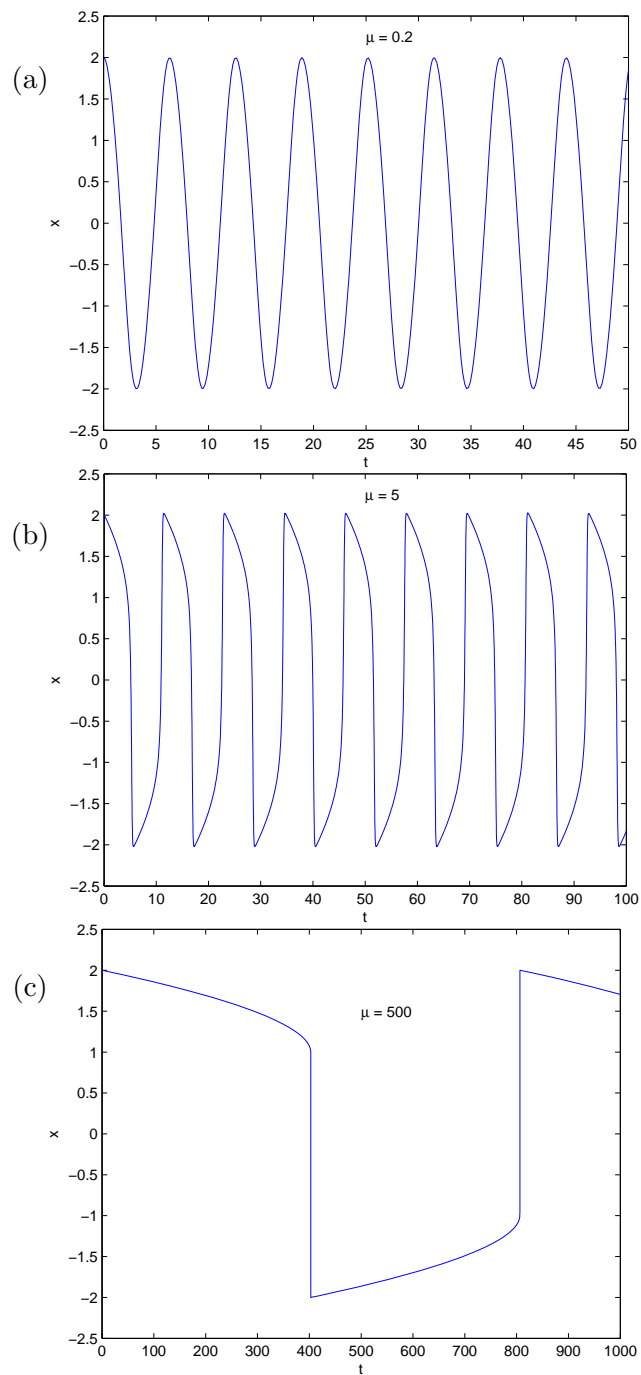


Figure 4.2: Solutions of the Van der Pol equation for different values of μ : (a) $\mu = 0.2$; (b) $\mu = 5$; (c) $\mu = 500$. Compare the shapes of the graphs and the scalings of the horizontal axis.

differential equations. One example is the Van der Pol equation. The equation was derived by Balthasar van der Pol in 1920 as a model for a certain electric system with a vacuum tube: a triode-oscillator. The equation is:

$$x'' - \mu(1 - x^2)x' + x = 0.$$

The exact meaning of this equation is beyond the scope of this course. We are only interested in the solutions of this equation.

Exercise 4.viii — Create a function `vdpol.m` for this ODE. Set $\mu = 0.2$ and use the time interval $[0, 1000]$. The initial conditions are $x(0) = 2$, and $x'(0) = 0$. Solve the ODE using methods discussed in previous sections. Compare your results with figure 4.2(a). Change μ to 500, and solve again. What do you notice?

It is clear that $\mu = 0$ is no problem for MATLAB. Even for small values $\mu < 0.5$, the solution still resembles a sine (figure 4.2a). If we increase μ (i.e. $\mu = 5$; figure 4.2b) a few things happen. The oscillation period increases. The graphs contain increasingly steeper slopes. But, perhaps most noticeably, we find that the calculation time increase. For $\mu = 500$ it is no longer practical to use `ode45` (figure 4.2c): the equation has become stiff. Conventional methods to solve ODEs numerically such as those used in `ode23` and `ode45` are no longer stable unless an extremely small time step is used. This means that many more steps are needed in the calculation, taking considerably more time. To solve a stiff differential equation it is often faster to use `ode15s`. This is a special function that uses a different numerical algorithm than `ode45` and finds a solution much faster. To find out whether an ODE is stiff, it is often best to compare calculation times for `ode45` and `ode15s` (and perhaps `ode23` in case `ode45` is too slow).

4.4 Problems

Problem 4.1 — A differential equation

Consider the following ordinary differential equation (ODE):

$$x' = ax.$$

- a) Solve the ODE analytically.
- a) Solve the ODE numerically for $a = -1$ with initial condition $x = 1$. Use a suitable time interval. Plot the exact and numerical solutions in a single graph.
- b) Now choose $a = 0.1$. What effect does this have on the solution? Plot the exact and numerical solution again.

Problem 4.2 — Radioactive decay

During radioactive decay, the amount of radioactive particles N is given by:

$$\frac{dN}{dt} = -\frac{\ln 2}{t_{1/2}}N.$$

With $t_{1/2}$ the half-life of the sample. The half-life is the time it takes for half of the radioactive particles to decay. Check this formula by solving the ODE. Choose different values for $t_{1/2}$, and look at the decay of $N(t)$ in a graph.

Problem 4.3 — The physical pendulum

The mathematical pendulum from section 4.3 is in fact a simplification of the physical pendulum. The ODE is:

$$\theta'' + \frac{g}{\ell} \sin(\theta) = 0.$$

Create a script that solves the ODEs for the mathematical and the physical pendulum. Choose initial conditions such that the difference between the two pendulums is clearly visible. Plot both solutions in one graph.

Now add a friction force proportional to θ' to the ODE. Solve the new ODE for various values of the friction constant and compare the result with the motion of the undamped pendulum.

Problem 4.4 — Predator-prey model

The animal populations of predators and their prey can be described by a coupled set of ODEs known as the Lotka-Volterra equations (see Wikipedia for more information). The ODEs are based on a very simple principle: the number of predators can grow when there are many prey. If there are many predators, the population of prey will decrease. When there are too many predators, their population will decrease because there are not enough prey. In equations:

$$p' = \alpha p - \beta pr$$

$$r' = -\gamma r + \delta pr$$

where $r(t)$ is the predator population, and $p(t)$ the population of prey; α , β , γ and δ are all positive constants.

- What processes do the individual terms in the equations represent?
- Set $\alpha = 1$, $\beta = 1.5$, $\gamma = 0.5$, $\delta = 0.5$, and solve the equation for the initial conditions: $p(0) = 1$, $r(0) = 0.5$.
- Plot $r(t)$ as a function of $p(t)$. Explain the behavior of $r(t)$ and $p(t)$. Save the figure.
- Vary the parameters and investigate under which circumstances stable and finite populations of predator and prey develop.

Problem 4.5 — The Lorenz oscillator

In 1963 Edward Lorenz presented a simple model for convection in the atmosphere (see Wikipedia for more information). This showed that small changes in initial conditions could have a huge effect on the outcome at later times, which is now known as the *butterfly effect* which signifies chaotic behavior. The model consists of a set of three coupled ODEs.

$$\begin{cases} \frac{dx}{dt} = \sigma(y - x) \\ \frac{dy}{dt} = x(\rho - z) - y \\ \frac{dz}{dt} = xy - \beta z \end{cases}$$

β , ρ and σ are all constants. For this problem set $\beta = 8/3$, $\rho = 28$ and $\sigma = 10$.

- Solve the system for $0 \leq t \leq 200$ with initial conditions $x = 0$, $y = 1$, $z = 0$.
- Plot the result with `plot3(x,y,z)`. To get a nice view use `view(-10,10)`. Save the figure.
- Change the value of ρ to 14 and try again. Under these conditions the behavior is no longer chaotic.

Problem 4.6 — Foucault's pendulum

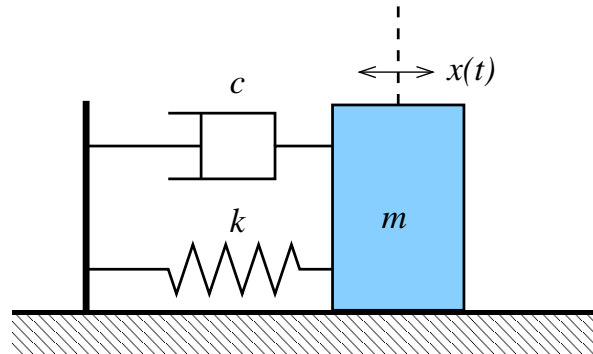
In 1851 Foucault built a large pendulum to prove that the Earth is rotating. In a rotating coordinate frame, the Coriolis effect will deflect the motion of the pendulum to the right (on the northern hemisphere). We use the equations for a pendulum with length ℓ for small amplitudes. We neglect the changes in height and approximate the position of the pendulum by only using the two horizontal coordinates (x, y) . The equations of motion are:

$$x'' + \frac{g}{\ell}x - 2\Omega_p y' = 0$$

$$y'' + \frac{g}{\ell}y + 2\Omega_p x' = 0$$

g is the gravitational acceleration, Ω_p is the component of the rotation vector that is perpendicular to Earth's surface. Ω_p will be largest at the poles, and zero at the equator. In reality Ω_p is very small (at the north pole $\Omega_p = 1$ period per day $= 7.3 \times 10^{-5} \text{ rad s}^{-1}$), for our simulation we choose a larger Ω_p .

- First set $\Omega_p = 0$, and solve the system for different g/ℓ . What is the resonant frequency ω_0 of the motion?
- Now vary Ω_p . Plot solutions in the (x, y) plane. The nicest pictures are found for $\Omega_p < \omega_0$! Save your figures.

Problem 4.7 — Damped oscillator 1

A cube of mass m is lying on a smooth horizontal table. There is no friction between the cube and the table. The cube is connected to a spring. The force of this spring increases linearly with the displacement x of the cube from the equilibrium position and is directed opposite to the displacement: $F_v = -kx$, where k is the spring constant. There is also a damping device attached to the cube. This device exerts a force that scales linearly with the velocity v but is directed in opposite direction: $F_d = -cv$.

The motion of the cube is described by the following ODE:

$$x'' = -\frac{c}{m}x' - \frac{k}{m}x.$$

- Rewrite the ODE in a suitable form for numerical solution in MATLAB.
- Is $x'(t=0) = 0$, $x(t=0) = 0$ a suitable initial condition for this system?
- What do you expect to happen when the damping coefficient c is increased?
- For $m = 1$ and $k = 4$, solve the ODE with: $c = 0$, $c = 2$, $c = 4$ and $c = 6$. Choose suitable time intervals and initial conditions. Plot the four solutions, and save the figures.

Problem 4.8 — Damped oscillator 2

We have the same situation as in exercise 4.7. Only now the mount of the spring and damping device moves as well. The mount oscillates with $x_P = a \sin(\omega_P t)$. The spring force in this situation is $F_v = -k(x - x_P)$. The damping force changes to $F_d = -c(v - v_P)$ (v_P is the velocity of the mount).

The ODE becomes

$$x'' = -\frac{c}{m}(x' - x'_P) - \frac{k}{m}(x - x_P) = 0.$$

- a) What changes must be made in the MATLAB function?
- b) Again, set $m = 1$, $k = 4$, but now set $c = 0.1$ (weak damping), $a = 0.1$ (weak forcing), and $\omega_P = 1$. Solve the equation for t between 0 and 100. Save the figure, and describe what you see.
- c) Now take $\omega_P = 2$. You will see that the effect is much larger now, even though the forcing amplitude a is the same. Forced systems can exhibit *resonance*. Save your figures.

Problem 4.9 — * Van der Pol-equation

Consider the Van der Pol equation (see also Wikipedia) in section 4.3.1. Develop a script which solves the ODE numerically for $\mu = 50$ in the time interval $[0, 100]$. The initial conditions are $x(0) = 2$, $x'(0) = 0$. Try using `ode45` and `ode15s`. Plot both solutions with different symbols so that they can be identified separately. Do you notice anything special in the graphs? Which of the methods is faster?

Appendix A

* *Gaussian elimination*

We want to find the line connecting the points $P(2, 3)$ and $Q(5, 1)$. The general equation for a line is $y = kx + l$. We need to determine the constants k and l . In point P it follows that $x = 2$ and $y = 3$, thus $2k + l = 3$. For point Q , $5k + l = 1$. The set of equations is:

$$\begin{cases} 2k + l = 3 \\ 5k + l = 1 \end{cases}$$

A systematic way of solving this system is:

1. Multiply the first equation with a constant, such that the constant in front of k becomes 1. In our example this means that we multiply by $\frac{1}{2}$; the equation reduces to $k + \frac{1}{2}l = \frac{3}{2}$ and the system is

$$\begin{cases} k + \frac{1}{2}l = \frac{3}{2} \\ 5k + l = 1 \end{cases}$$

2. Because both equations must be true, we can add or subtract them from each other without any problems. Because $k + \frac{1}{2}l = \frac{3}{2}$ and $5k + l = 1$ it follows that $k + \frac{1}{2}l + 5k + l = \frac{3}{2} + 1$. This means that we can eliminate k from the second equation! We just subtract the first equation 5 times from the second equation, leaving us with $-\frac{3}{2}l = -\frac{13}{2}$ for the second equation.
3. The second equation can be simplified just like we did in step 1: multiply both sides with $-\frac{2}{3}$ to find $l = \frac{13}{3}$.
4. Now that l is a known constant, k follows directly from the first equation: $k = -\frac{2}{3}$.

Now we solve the problem of three points that define a unique parabola passing through them. The three points are $P(-2, 4)$, $Q(1, 1)$ and $R(2, -4)$. We are looking for a parabola of the form $y = kx^2 + lx + m$. To find the parabola that passes through these three points, we use the same steps as stated above, but now we use the matrix notation. We rewrite the problem as one single matrix $C = (A | b)$:

$$C = \left(\begin{array}{ccc|c} 4 & -2 & 1 & 4 \\ 1 & 1 & 1 & 1 \\ 4 & 2 & 1 & -4 \end{array} \right).$$

The vertical line has no special meaning, it is just used to visually separate A and b .

Step 1: We make the first nonzero element of the first row equal to 1. We could do this by multiplying the entire row (this is in fact multiplying the entire first equation) by $\frac{1}{4}$. However, we already have a row which has a 1 as the first element. The order of execution is of no importance (all points are given in random order), so we can exchange rows if we want. We exchange row 1 with row 2.

$$C = \left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 4 & -2 & 1 & 4 \\ 4 & 2 & 1 & -4 \end{array} \right).$$

Step 2: We want to reduce all the rows apart from row 1 to have a 0 as the first element. To do this we must subtract row 1 four times from both row 2 and row 3.

$$C = \left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & -6 & -3 & 0 \\ 0 & -2 & -3 & -8 \end{array} \right).$$

Now we repeat step 1, but for the second row. The first element that is not zero must become 1, so we multiply by $-\frac{1}{6}$:

$$C = \left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & \frac{1}{2} & 0 \\ 0 & -2 & -3 & -8 \end{array} \right).$$

We repeat step 2 for the third row, making the second element a 0. We add the second row two times to the third row.

$$C = \left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & \frac{1}{2} & 0 \\ 0 & 0 & -2 & -8 \end{array} \right).$$

Now we make the first element in row 3 that is not zero a 1. So we multiply by $-\frac{1}{2}$. We are left with:

$$C = \left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & \frac{1}{2} & 0 \\ 0 & 0 & 1 & 4 \end{array} \right).$$

With this form we can find the solution by *substitution*. The last row tells us that $0 \cdot k + 0 \cdot l + m = 4$ or $m = 4$. The second row tells us that $0 \cdot k + 1 \cdot l + \frac{1}{2} \cdot m = 0$, or (using $m = 4$) $l = -2$. Finally, we can find k using the first equation: $1 \cdot k + 1 \cdot l + 1 \cdot m = 1$, or $k = -1$. The parabola is $y = -x^2 - 2x + 4$. This procedure is also known as *Gaussian elimination* or *row reduction*.

List of Tables

1.1	Overview of the predefined variables in MATLAB.	5
1.2	Notation of some predefined mathematical formulas in MATLAB.	6
1.3	Relational operators in MATLAB.	10
2.1	Properties for plots.	22
2.2	Manipulation of matrices.	28
2.3	Some useful matrix operations	31

Index

() , 6, 20
* , 6, 29, 33
+ , 6, 29
- , 6, 29
.
.
.* , 29
... , 4
./ , 29
.^ , 29
/ , 6, 29
: , 11, 20, 27
; , 4, 25
< , 10
<= , 10
= , 10
== , 10
> , 10
>= , 10
[] , 19
&& , 10
\ , 35
^ , 6, 34
@ , 45
~ , 11
~= , 10

abs, 6
acos, 6
algebraic operations, 6
 for matrices, 28
ans, 4, 5
asin, 6
atan, 6
axis, 24

clear, 5
colorbar, 26, 38
 Command History, 2
 Command Window, 2
comment, 13
contour, 26, 38
contourf, 26
cos, 6
 Current Folder, 2

daspect, 27
diag, 25
differential equation, 49
 system, 51
disp, 9
doc, 3

e, 4
 Editor, 7
else, *see* if
end, *see* if, *see* for, *see* while
eps, 5
errorbar, 37
exp, 6
extremum, 45

factorial, 8
 Figure, 20
figure, 20
 annotation, 23
 axis label, 23
 title, 23
fminbnd, 45
for .. end, 11
format, 15
function, 7
 handle, 45
fzero, 45

Gaussian elimination, 59
graph, *see* figure
Greek character, 23

help, 2

- hist, 37
- histogram, 37
- hold, 22
- i, 5
- if .. else .. end, 9
- Inf, 5
- input, 9
- j, 5
- load, 5, 31
- log, 6
- log10, 6
- logical operator, 10
- loglog, 23
- m-file, 7
- mathematical functions, 6
- matrix, 24
 - column, 27
 - element, 20
 - manipulation, 28
 - merger, 28
 - power, 34
 - product, 32
 - reduction, 59
 - rotation, 39
 - row, 27
- max, 31
- mean, 31
- mesh, 26
- meshgrid, 26
- min, 31
- mod, 16
- NaN, 5
- nchoosek, 8
- numerical integration, 46
- ode45, 50
- ones, 25
- pi, 5
- plot
 - colour, 22
 - line, 22
 - symbol, 22
- plot, 20, 21
- plot3, 56
- polar, 37
- predefined variables, 5
- prod, 31
- quad, 46
- quadgk, 46
- rand, 13
- realmax, 5
- realmin, 5
- relational operator, 9, 10
- rotation matrix, 39
- round, 13
- save, 5, 32
- script, 8
- semilogx, 22
- semilogy, 23
- shading, 27
- sin, 6
- size, 24
- sqrt, 6
- start-up screen, 2
- std, 31
- stiffness
 - differential equation, 53
- sum, 31
- surf, 27
- surfl, 27
- system of linear equations, 32
 - overdetermined, 35
 - underdetermined, 35
- tan, 6
- text, 23
- title, 23
- transpose, 27
- variable, 3
 - array, 19
 - in a function, 8
 - in a script, 8
 - predefined, 5
- view, 27, 56
- while .. end, 12
- Workspace, 2

xlabel, [23](#)

ylabel, [23](#)

zero, [45](#)

zeros, [25](#)

zlabel, [27](#)