Matlab for Chemists

Theoretical Chemistry

Radboud University

The Netherlands

September 2018

PREFACE

Matlab is an interactive program package for numerical computation and visualization. It originated in the middle of the 1980s as a user interface to matrix manipulation packages written in fortran. Hence its name: "Matrix Laboratory". Over the years the system has been extended, it now includes a programming language, extensive visualization tools, and many numerical methods.

These are notes accompanying a course in Matlab for chemistry at the Radboud University Nijmegen. The course has the following objectives:

- to give a basic introduction to Matlab
- to introduce the students to programming (loops, if then else constructs, functions).
- to show how Matlab can be used as a tool for fitting and plotting data obtained in the chemical laboratory.
- to apply the acquired knowledge to some real chemical problems

The course is structured as follows: first a few basic concepts necessary to work with Matlab will be discussed, then an introduction into programming will be given and finally these concepts will be applied to the chemical concepts and examples will be given in how Matlab is used in research in several groups of the Faculty of Science. Although no new mathematical concepts are introduced in the present lectures, the mathematical knowledge necessary to do the Matlab exercises, is briefly reviewed.

This course has evolved from an early course on the use of computers in Chemistry at the Radboud University. The lecture notes are based on a syllabus by dr. ir. P.E.S. Wormer and a later version of dr. J. van Bentum. This tutorial is mainly intended to be used as material for self study. Like most modern computer programs, Matlab has a rather intuitive user friendly interface with extensive menu and help functions. This allows the user to become familiar with the basic functions without much prior knowledge. However, it is also very easy to make mistakes and one needs at least some basic understanding of the mathematics involved to use Matlab in a reliable way. One of the main advantages of programs like Matlab is their flexibility. In fact you can add library routines at will and in principle you can develop a fully personalized version of a Matlab toolbox that is specifically suited for your own kind of research.

Learning how to use a programming environment like Matlab is a useful basis even if you decide to use other programming languages such as C or Fortran. If you are interested in a freeware alternative, check out Scilab or Octave, where the latter is the most compatible to Matlab.

Contents

Part I

MATLAB basics

1. Scalars, matrices, and vectors

1.1 Matlab basics

Starting Matlab creates one or more windows on your screen. The most important is the Command Window, which is the place where you enter commands and the results of mathematical operations are displayed. When the Command Window is active, a cursor appears after the prompt (>>), indicating that Matlab is waiting for your command. Matlab responds by printing text in the Command Window or by creating a Figure Window for graphics. Matlab is an interactive system; commands followed by Enter are executed immediately. The results are, if desired, displayed on screen. However, execution of a command will be possible only if the command is typed according to the rules. Below we have listed some tips that may help you to get started with using Matlab as a scientific calculator. Try the following:

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

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

```
>> 14/4
ans =
3.5000
>> ans * ans
ans =
12.2500
```

When the command is followed by a semicolon (;), the output is suppressed. Check the difference between the following expressions:

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

1.2 Scalars and assignments

All internal operations in Matlab are performed with floating point numbers 16 digits long, *e.g.*, 3.141592653589793 or 0.3141592653589793e-1 (e-1 indicates 10^{-1}). Variables in the computer can only take a limited amount of values, it is discretized by the computer precision. When a very small number is, for instance 10^{-20}, is added to 1.0, the computer does not change anything to the value of 1.0; the representation of a floating point variable with a limited amount of bytes is not accurate enough to add such a small amount to 1.0. Off course, if one wants to add $10^{-20} + 10^{-19}$, the answer is $1.1 \cdot 10^{-19}$. The machine precision is only important if the difference between the numbers is more than a factor of `eps = 4.5e15` (for a 8-byte representation). The simplest data structure in MATLAB is the scalar. Scalars can be given names and assigned values, *e.g.*,

```
>> num_students = 25;
>> Temperature = 272.1;
>> Planck = 6.6260755e-34;
```

For most purposes, MATLAB does not make a distinction between integers (whole numbers like `num_students`) and floating points numbers (`Temperature`). In many other scripting or programming languages, this distinction is explicitly made. By default, MATLAB displays only 5 digits. The command `format long` increases this number to 15, format short reduces it to 5 again. For instance:

```
>> Planck = 6.6260755e-34
Planck =  6.6261e-34
>> format long
>> Plank
Planck =  6.62607550000000e-34
```

It is important to understand fully the difference between the mathematical equality $a = b$ (which can be written equally well as $b = a$) and the MATLAB assignment `a = b`. In the latter statement, the `=` character means "will get the value of". Therefore, `a = b` means `a` will get the value of `b` and it is necessary that the right hand side of the assignment has a value at the time that the statement is executed. The right hand side is first fully evaluated and then the result is assigned to the left hand side. Example:

```
>> a = 33.33;
>> a = a + 1
a =
34.3300
```

First the expression `a + 1` is fully evaluated. The value of `a`, which it has just before the statement, is substituted everywhere. Only at the end of the evaluation `a` is assigned a new value by means of the assignment symbol `=`. From the second line in this example, the difference between the assignment symbol `=` and the mathematical equality sign $=$ becomes immediately clear.

A variable name must start with a lower- or uppercase letter and only the first 31 characters of the name are used by MATLAB. Digits, letters and underscores are the only allowed characters in the name of a variable. MATLAB is case sensitive: the variable `temperature` is another than `Temperature`. MATLAB knows $\pi = 3.14\ldots$, it is simply the variable `pi`. *Do not use the name pi for anything else!* Scalars can be added: `a + b`, subtracted: `a - b`, multiplied: `a * b`, divided: `a/b` and taken to a power: `a^b`

Table 1.1: *Elementary mathematical scalar operations.*

mathematical notation	MATLAB command
$a+b$	`a + b`
$a-b$	`a - b`
ab	`a * b`
a/b	`a/b`
a^b	`a^b`
$\sqrt{a}$	`sqrt(a)`
$\vert a\vert$	`abs(a)`
$\exp a, e^a$	`exp(a)`

as indicated in Table 1.1. The usual priority rules hold (multiplication before addition, etc.). In case of doubt, one can force the priority by brackets and write `a/(b^3)` instead of `a/b^3`.

MATLAB can also work with complex numbers:

```
>> x = sqrt(-1)
x = 0.0000 + 1.0000i
```

For more information we refer to the appendix.

1.3 Data arrays and matrices

A matrix is a rectangular array of numbers or other mathematical objects, for which operations such as addition and multiplication are defined. The size of a matrix is defined by the number of rows and columns that it contains. A matrix with m rows and n columns is called an $m \times n$ matrix or $m - by - n$ matrix, while m and n are called its dimensions.

One way of creating a matrix is by simply typing the numbers into the command line. Rows are entered either space delimited or comma delimited and columns are entered either semicolon delimited or end of line delimited, as

```
>> A = [1 2 3; -2 -3 -4]
A =
     1     2     3
    -2    -3    -4

>> B = [1,2,3
-2,-3,-4]
B =
     1     2     3
    -2    -3    -4
```

The sequence is delimited by square brackets. Notice that the matrices `A` and `B` are indeed equal. The matrix is shown by typing its name, thus,

```
>> A = [1 2 3; -2 -3 -4]; % No output
>> A                      % echo the matrix A
A =
     1     2     3
    -2    -3    -4
```

Note that comments may be entered preceded by a % sign, they run until the end of the line. These comments are not executed by MATLAB and one can use them to indicate what the statement is doing. In this syllabus, the comments will be displayed in gray for clarity. You will later see when you write your own scripts or functions that these comments are very useful.

The rules for naming matrices are the same as for scalars: case sensitive, starting with letters, length ≤ 31, only digits, letters and underscores in the name. Single elements may be accessed by round brackets with two indexes separated by a comma where the first index refers to the specific row and the second to the specific column. For example, we can extract in the element from the second row and third column by typing `A(2,3)` from our previous example (returns `-4`). Also ranges, *e.g.*, `A(1,2:end)` (returns the row vector `[2 3]`) can be extracted. The word `end` gives the last entry in the column (or row).

The identity (or unit) matrix $\boldsymbol{I}$ plays an important role in linear algebra. It is a square matrix with off-diagonal elements zero and the number one on the diagonal. MATLAB has the phonetic name `eye` for the function that returns the identity matrix.

```
>> eye(4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

The commands `ones(n,m)` and `zeros(n,m)` return an $n \times m$ matrix containing unity or zeros in all entries, respectively. In Table 1.2 we list a few more of these commands.

Table 1.2: *Matrix operations*

eye	Identity matrix
ones	Create an array of all ones
rand	Uniformly distributed random numbers and arrays
randn	Normally distributed random numbers and arrays
zeros	Create an array of all zeros

It is good practice to allocate a matrix first before filling it piecewise

```
>> A = zeros(2,2);        % First create a 2-by-2 matrix
>> A(1,1) = 3             % Fill the (1,1) element
A =
        3    0
        0    0
>> A(3,4) = 3             % Matlab lets you assign the (3,4) element
A =
        3    0    0    0
        0    0    0    0
        0    0    0    3
```

MATLAB simply lets the matrix `A` grow to allow you to assign the (3,4) element. Try to avoid this.

Exercise 1
Read the help of `eye` and `ones`. Create the following matrices, each with one statement,

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \quad \text{and also} \quad \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

1.4 Vectors

A column vector of length n is a special case of an $n \times m$ matrix, namely one with $m = 1$. A row vector has $n = 1$. A row vector can be turned into a column vector and vice versa (transposition) by the transposition operator (')

```
>> a = [1 2 3];
>> b = a'
b =
      1
      2
      3
>> c = b'
c =
      1     2     3
```

The easiest way to enter a column vector is as a row plus immediate transposition, thus,

```
>> a=[5 4 3 2]'
a =
      5
      4
      3
      2
```

A matrix can be constructed by concatenation. This is the process of joining smaller matrices (or vectors) to make bigger ones. The pair of square brackets, `[]`, is the concatenation operator. Example:

```
>> a = [1  3 -5]';
>> b = [2  1  7]';
>> c = [-1 6  1]';
>> [a b c]
ans =
      1     2    -1
      3     1     6
     -5     7     1
```

Note

Experience shows that a common error in MATLAB *is to forget whether a vector is a row or a column vector. We adhere strictly to the convention that a vector is a column.*

The colon can also be used to generate vectors

```
>> a = [1:2:10]'
a =
     1
     3
     5
     7
     9
```

A vector is created which starts on 1, has subsequent elements with increment 2 until 10 is reached. If the increment is left out, an increment of 1 is used as default

```
>> a = [1:5]'
a =
     1
     2
     3
     4
     5
```

In mathematics and physics the *norm* and the *length* of a vector are often used as synonyms. In MATLAB the two concepts are different, `length(a)` returns the number of components of `a` (the dimension of the vector), while `norm(a)` returns the norm of `a`. Remember that the norm $|\boldsymbol{a}|$ of $\boldsymbol{a}$ is by definition $|\boldsymbol{a}| = \sqrt{\boldsymbol{a} \cdot \boldsymbol{a}}$. To obtain the dimensions of a matrix, one can use the command `size(A)`.

Exercise 2

In the USA the temperature scale of Fahrenheit is in daily use. It is named after Gabriel Daniel Fahrenheit (1686–1736), who defined it in 1714, while working in Amsterdam.

To convert to the Celsius scale use the formula $C = 5(F - 32)/9$. Prepare a table `F` starting at -20 °C, ending at 100 °C with steps of 5 °C that contains Celsius converted to Fahrenheit.

1.5 Matrix and vector indexing

Thus far we have met twice the colon (`:`) operator: once to get a section out of a matrix and once to generate a grid. In fact, these are the very same uses of this operator. To explain this, we first observe that elements from an array (= matrix) may be extracted by the use of an index array with integer elements.
Example:

```
>> A = rand(5)
A =
    0.9501    0.7621    0.6154    0.4057    0.0579
    0.2311    0.4565    0.7919    0.9355    0.3529
    0.6068    0.0185    0.9218    0.9169    0.8132
    0.4860    0.8214    0.7382    0.4103    0.0099
    0.8913    0.4447    0.1763    0.8936    0.1389
>> index = [1 3 5];
>> B = A(index,:)
B =
    0.9501    0.7621    0.6154    0.4057    0.0579
    0.6068    0.0185    0.9218    0.9169    0.8132
    0.8913    0.4447    0.1763    0.8936    0.1389
```

Explanation:
For illustration purposes we created a 5×5 matrix `A`. Then we created the integer array `index` and used it to create an array `B` that has the same columns as `A` but only row 1, 3, and 5 of `A`. We get the very same matrix `B` by the command `B = A(1:2:end,:)`, because 'on the fly' an array `[1,3,5]` is prepared and used to index `A`. Here we make use of the word `end` which gives the last entry in the column or row.

The command `[]` can be used to create an empty matrix of size 0×0. It can be used to remove rows and/or columns from a matrix. Using the same `A` and `index` as in the previous example,

```
>> C = A;
>> C(index,:) = []
C =
    0.2311    0.4565    0.7919    0.9355    0.3529
    0.4860    0.8214    0.7382    0.4103    0.0099
```

Rows 1, 3 and 5 of `C` are replaced by empty rows indicated by `[]`.

If `v` has `m` components and `w` has `n` components, then `A(v,w)` is the $m \times n$ matrix formed from the elements of `A` whose subscripts are the elements of `v` and `w`. Example:

```
A =
    0.2028    0.0153    0.4186    0.8381
    0.1987    0.7468    0.8462    0.0196
    0.6038    0.4451    0.5252    0.6813
    0.2722    0.9318    0.2026    0.3795
    0.1988    0.4660    0.6721    0.8318
```

```
v =
       1     1     2              % m = 3
w =
       3     4     3     4     2  % n = 5
>> B = A(v, w)                    % 3-by-5 matrix
B =
      0.4186    0.8381    0.4186    0.8381    0.0153
      0.4186    0.8381    0.4186    0.8381    0.0153
      0.8462    0.0196    0.8462    0.0196    0.7468
```

Explanation: The matrix `B` becomes

$$\begin{pmatrix} A_{13} & A_{14} & A_{13} & A_{14} & A_{12} \\ A_{13} & A_{14} & A_{13} & A_{14} & A_{12} \\ A_{23} & A_{24} & A_{23} & A_{24} & A_{22} \end{pmatrix}.$$

Rows are labeled by `v` and columns by `w`. Note that elements of `A` can be used more than once. Alternatively, we may use the MATLAB command `repmat`, which in fact uses the same mechanism. The following command constructs the a matrix with three of the same columns `repmat(x,1,3)`. We can make a vector out of a matrix by using `a = A(:)`. This creates a column vector with the columns of `A` stacked on top of each other.

1.6 Matrix and vector manipulations

Two very useful commands are `sum` and `sort`. `sum` returns a row vector which contains the sum of the columns in the matrix. If `sum` is applied to a vector, it returns a single value, the sum over the vector. The command `sort` returns a copy of the input vector with the elements arranged in increasing order. For matrices, `sort` orders the elements within columns.

In MATLAB the usual mathematical operations can be performed on matrices and vectors: addition and subtraction (provided they have the same dimensions) and multiplication and division by a number. For a vectors of length n this will be

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \vdots \\ a_n + b_n \end{pmatrix} \tag{1.1}$$

and

$$\lambda \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \lambda a_1 \\ \lambda a_2 \\ \vdots \\ \lambda a_n \end{pmatrix} \tag{1.2}$$

as can be seen in the following example

```
>> a = [1 2 3 4 5]';
>> b = [5 4 3 2 1]';
>> 2 * a + 3 * b
```

```
ans =
    17
    16
    15
    14
    13
```

For matrices it works analogously. For the addition operation to work, it is critical that `a` and `b` have the same dimensions; otherwise MATLAB will complain

```
Error using +
Matrix dimensions must agree.
```

A very useful command is `whos`. It will show all variables, that you have defined and their sizes.

The inner (dot) product of two real vectors consisting of n elements is mathematically defined as

$$\boldsymbol{a}\cdot\boldsymbol{b} \equiv a_1b_1 + a_2b_2 + \cdots + a_nb_n = \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \sum_{i=1}^{n} a_i\, b_i$$

In MATLAB:

```
>> a =[ 0.4225    0.8560    0.4902    0.8159    0.4608]';
>> b =[ 0.4574    0.4507    0.4122    0.9016    0.0056]';
>> a'*b
ans =
    1.5193
```

Recalling the definition of the norm of a vector, we see that `sqrt(a'*a)` gives the very same answer as `norm(a)`. Both `sqrt` and `norm` are functions known to MATLAB.

During your linear algebra course you learned the matrix-vector multiplication. We briefly repeat the rule. Suppose the matrix $\boldsymbol{A}$ consists of m columns $\boldsymbol{a}_j$, $j = 1, \dots, m$. Each vector $\boldsymbol{a}_j$ is of length n, *i.e.*, $\boldsymbol{A}$ is an $n \times m$ matrix. In order that the matrix-vector multiply $\boldsymbol{A}\,\boldsymbol{c}$ is possible, it is necessary that the column vector $\boldsymbol{c}$ is of length m. The result of the multiplication $\boldsymbol{A}\,\boldsymbol{c}$ is a column vector of length n

$$\mathbf{Ac} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix} = \begin{pmatrix} A_{11}c_1 + A_{12}c_2 + \cdots + A_{1m}c_m \\ A_{21}c_1 + A_{22}c_2 + \cdots + A_{2m}c_m \\ \vdots \\ A_{n1}c_1 + A_{n2}c_2 + \cdots + A_{nm}c_m \end{pmatrix}. \tag{1.3}$$

Back in MATLAB we note that the matrix-vector multiplication is simply given by `*`, *provided the dimensions correspond.* In order to give an example we introduce the MATLAB function `rand`; `rand(n,m)` returns a matrix with n rows and m columns of which the elements are positive random numbers in the range $0 \cdots 1$.

```
>> A = rand(4,3)
A =
    0.1934    0.1509    0.8537
    0.6822    0.6979    0.5936
    0.3028    0.3784    0.4966
    0.5417    0.8600    0.8998
>> c = rand(3,1)  % A column vector of length 3
c =
    0.8216
    0.6449
    0.8180
>> A * c          % Matrix-vector multiply
ans =             % A column vector of length 4
    0.9545
    1.4961
    0.8989
    1.7357
```

Similarly, we can multiply two matrices

```
>> C = [c c];    % Construct a 3-by-2 matrix
>> A * C         % Matrix-matrix multiplication
ans =            % A 4-by-2 matrix
    0.95454   0.95454
    1.49614   1.49614
    0.89903   0.89903
    1.73571   1.73571
```

Here we have multiplied a 4-by-3 matrix with a 3-by-2 matrix. This works since the 'inner matrix dimensions' are the same (both are 3). 4 and 2 are the 'outer matrix dimensions', they are independent. Look at the following example.

```
>> d = rand(4,1)     % A column vector of length 4
    d =
        0.6602
        0.3420
        0.2897
        0.3412
    >> A * d         % Try a matrix vector multiply
    ??? Error using ==> *
    Inner matrix dimensions must agree.
```

Exercise 3

Consider the following matrices and their dimensions: $\boldsymbol{A}$ (4×5), $\boldsymbol{B}$ (4×10), $\boldsymbol{C}$ (10×4), and $\boldsymbol{D}$ (5×5). Predict the dimensions of $\boldsymbol{ADA}^T$ and $\boldsymbol{BCAD}$. Verify your answer by creating the matrices with the MATLAB function `rand` and by explicit matrix multiplication in MATLAB.

1.7 Pointwise operations

In this section, we introduce the *dot operations*, which are not defined as such in linear algebra. It often happens that one wants to multiply all the individual elements of vectors:

```
>> [1 2 3 4]' .* [12 6 4 3]'
ans =
    12
    12
    12
    12
```

Similarly, we can use `./` and `.^` for elementwise division and taking the power of, respectively.

Suppose we have measured five numbers as a function of a certain quantity and that we also made a fit of the measured values. To compute the percentage error of the fit we proceed as follows:

```
>> % Enter the observed values:
>> obs = [ -0.3012  -0.0999   0.0527   0.1252   0.2334]';
>> % Enter the fitted values:
>> fit = [ -0.3100  -0.1009   0.0531   0.1263   0.2479]';
>> % Percentage error: (put result into vector error
>> % and transpose for typographical reasons)
>> error = [100 * (obs-fit)./obs]'
error =
   -2.9216   -1.0010   -0.7590   -0.8786   -6.2125
```

Here the operation `./`, this is pointwise division, *i.e.*, it computes

$$\mathrm{error}(i) = 100 * (\mathrm{obs}(i) - \mathrm{fit}(i))/\mathrm{obs}(i) \quad \text{for} \quad i = 1, \ldots, 5,$$

successively. *It is essential that upon any error message on the dimensions, one carefully checks all dimensions and operations to find the correct statement, instead of just adding or removing dots. The fact that the error message has disappeared does not mean that the answer is correct.*

In mathematics the following operation is *not* defined: $s + \boldsymbol{a}$, where s is a scalar (number) and $\boldsymbol{a}$ is a vector. However, in MATLAB the following happens,

```
>> a     % show the present value of vector a
a =
    0.5028
    0.7095
    0.4289
    0.3046
>> a = a + 1
a =
    1.5028
    1.7095
    1.4289
    1.3046
```

Table 1.3: *Elementary mathematical functions. All operate on the elements of matrices.*

abs	Absolute value and complex magnitude
acos, acosh	Inverse cosine and inverse hyperbolic cosine
acot, acoth	Inverse cotangent and inverse hyperbolic cotangent
acsc, acsch	Inverse cosecant and inverse hyperbolic cosecant
angle	Phase angle
asec, asech	Inverse secant and inverse hyperbolic secant
asin, asinh	Inverse sine and inverse hyperbolic sine
atan, atanh	Inverse tangent and inverse hyperbolic tangent
atan2	Four-quadrant inverse tangent
ceil	Round toward infinity
complex	Construct complex data from real and imaginary components
conj	Complex conjugate
cos, cosh	Cosine and hyperbolic cosine
cot, coth	Cotangent and hyperbolic cotangent
csc, csch	Cosecant and hyperbolic cosecant
exp	Exponential
fix	Round towards zero
floor	Round towards minus infinity
gcd	Greatest common divisor
imag	Imaginary part of a complex number
lcm	Least common multiple
log	Natural logarithm
log2	Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
log10	Common (base 10) logarithm
mod	Modulus (signed remainder after division)
nchoosek	Binomial coefficient or all combinations
real	Real part of complex number
rem	Remainder after division
round	Round to nearest integer
sec, sech	Secant and hyperbolic secant
sign	Signum function
sin, sinh	Sine and hyperbolic sine
sqrt	Square root
tan, tanh	Tangent and hyperbolic tangent

MATLAB has lots of elementary mathematical functions that all work elementwise on vectors and matrices. In Table 1.3 we show the most important mathematical functions. As a modern computer package, MATLAB has extensive built-in documentation. The command `helpdesk` gives access to a very elaborate interactive help facility. The command `help cmd` gives help on the specified command `cmd`. The command `more on` causes `help` to pause between screenfuls if the help text runs to several screens. In the online help, keywords are capitalized to make them stand out. *Always type commands in lowercase* since *all* command and function names are actually in lowercase. Recall in this context that MATLAB is case sensitive. Do not be confused by the capitals in the help. Another useful command is `lookfor` followed by a keyword. It will search through the documentation of all functions for this keyword.

Exercise 4

a. Enter the statements necessary to create the following matrices:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 4 & 6 \\ 2 & 3 & 5 \\ 1 & 0 & 4 \end{pmatrix}, \quad \boldsymbol{B} = \begin{pmatrix} 2 & 3 & 5 \\ 1 & 0 & 6 \\ 2 & 3 & 1 \end{pmatrix}$$

$$\boldsymbol{C} = \begin{pmatrix} 5 & 1 & 9 & 0 \\ 4 & 0 & 6 & 2 \\ 3 & 1 & 2 & 4 \end{pmatrix}, \quad \boldsymbol{D} = \begin{pmatrix} 3 & 2 & 5 \\ 4 & 1 & 3 \\ 0 & 2 & 1 \\ 2 & 5 & 6 \end{pmatrix}$$

b. Compute the following using MATLAB, and try to understand the results. If you receive an error message rather than a numerical result, explain the error.

a.	`A + B`	g.	`C + D`
b.	`A - 2 .* B`	h.	`C' + D`
c.	`(A - 2) .* B`	i.	`C.*D`
d.	`A.^2`	j.	`A - 2*eye(3)`
e.	`sqrt(A)`	k.	`A - ones(3)`
f.	`C'`	l.	`A^2`

Exercise 5

Type in the column vector `a` containing the six subsequent elements: 1, −2, 3, −4, 5, −6. Normalize this vector. That is, determine its norm and divide the vector by it. Check your result by computing the norm of the normalized vector.

Exercise 6

Type in the column vector `b` containing the subsequent elements: −1, 2, −3, 4, −5, 6. Determine the angle (in degrees) of this vector with the vector of the previous exercise. Explain your answer. Recall that the angle between two vectors is defined by

$$\boldsymbol{r}_1 \cdot \boldsymbol{r}_2 = r_1 \, r_2 \, \cos\phi.$$

Here $r_i \equiv |\boldsymbol{r}_i| \equiv \sqrt{\boldsymbol{r}_i \cdot \boldsymbol{r}_i}$ is the norm of vector $\boldsymbol{r}_i$ $(i = 1, 2)$ and $0 \le \phi \le \pi$ is the angle between the two vectors. We give an example of the computation of ϕ:

```
>> % Introduce two column vectors for the example:
>> a = [23 0 -21]';
>> b = [0 10 0]';
>> % Compute their lengths:
>> la = norm(a);
>> lb = norm(b);
>> % Now the cosine of the angle
>> cangle = (a' * b)/(la * lb);
>> % the angle itself
>> phi = acos(cangle)
phi =
    1.5708
>> % Radians, convert to degrees:
```

```
>> phi * 180/pi
ans =
    90
```

Exercise 7

Compute

```
[1 2 3 4]' .* [24 12 8 6]'
[1 2 3 4]' * [24 12 8 6]
[1 2 3 4] * [24 12 8 6]'
```

Explain the differences.

2. Scripts and plotting

2.1 Scripts

So far we used MATLAB as a calculator and typed in commands on the fly. When one has to type in longer sequences of MATLAB commands, it is pleasant to be able to edit and save them. Sequences of commands can be stored in *scripts*. A script is a flat (ASCII, plain) file containing MATLAB commands that are executed one after the other. A script can be prepared by any ASCII editor, such as Microsoft's NOTEPAD (but not by Microsoft's WORDPAD or WORD!) or under Linux by pico, kate, gedit, et.c. We will use the editor contained in MATLAB. This editor can be started from the MATLAB prompt by the command `edit`. The scripts must be saved under the name `anyname.m`, where `anyname` is for the user to define.
Advice:

- Use meaningful names for your scripts, because experience shows that you will forget very soon what a script is supposed to do.
- Save your scripts in a subdirectory of your network home directory, this disk is backed up regularly and can be used from different computers.
- Mind where you are with your directories and disks. MATLAB shows your current directory in the toolbar. When you are about to save a script, the editor allows you to browse and change directories. From the MATLAB session you can also browse and change directories by clicking the button to the right of the "current directory" field.

Provided your script is in your current directory, you can execute it by simply typing in `anyname`, where `anyname.m` is the meaningful name you have given to your script. Depending on the semicolons, you will see the commands being executed. If you have too much output on your screen you can toggle `more on/more off`. Press the "q" key to leave the display mode.

It is important to notice that a script simply continues your MATLAB session, all variables that you introduced 'by hand' are known to the script. After the script has finished, all variables changed or assigned by the script stay valid in your MATLAB session. Technically this is expressed by the statement: *the scope of script variables is global.*

From now on, write scripts for each of the exercises that you will make.

2.2 Plotting

MATLAB has extensive facilities for plotting. Very often one plots a function on a discrete grid. For instance, if we want to plot a sine:

```
>> x = pi/10 * [0:10]';
>> y = sin(x);
>> plot(x,y)
```

The command `plot(x,y)` plots vector `y` versus vector `x`. After this command a new window opens with the plot.

Suppose we now also want to plot the cosine in the same figure. If we would enter `plot(x, cos(x))`, then the previous plot would be overwritten. We can toggle `hold on/off` to hold the plot. Alternatively, we can create two plots in one statement: `plot(x, [sin(x) cos(x)])`, *i.e*,

```
>> plot(x, [y1 y2 y3])
```

Here we have created a matrix and all columns are plotted against `x`. If we want to use different vectors `x`

```
>> plot(x1, y1, x2, y2, x3, y3)
```

Different colors and line types can be chosen, see help plot for information on this. For example, `plot(x,cos(x), 'r:')` plots the cosine as a red dotted line.

The `xlabel` and `ylabel` functions add labels to the x and y axis of the current figure. Their parameter is a string, *e.g.*, `ylabel('sin(\phi)')` and `xlabel('\phi')`. The `title` function adds a title on top of the plot. Example:

```
phi = pi/100 * [0:200]';
plot(phi * 180/pi, [sin(phi) cos(phi)])
xlabel('\phi (degrees)')
title('Graph of the sine function')
legend('sin(\phi)', 'cos(\phi)')
```

For people who know the text editing system LaTeX this part of Matlab is easy, since it uses quite a few of the LaTeX commands. `\phi` is LaTeX to get the Greek letter ϕ. (The present lecture notes are prepared by the use of LaTeX).

Sometimes we like to zoom in or out to specific part of the plot we can then use the command `axis([xmin xmax ymin ymax])` which sets the lower and upper limits for the x and y-axis. Without any arguments, `axis` turns autoscaling on. Type `help axis` to see the other possibilities.

Exercise 8

Write a Matlab script that plots the function

$$y = \frac{1}{(x-0.3)^2+0.01} + \frac{1}{(x-0.9)^2+0.04} - 6,$$

which has two humps. Compute in the script first the grid `0:0.002:1` and then the array (= vector) y and then call `plot(x,y)`.

Exercise 9
Write a MATLAB script to plot the polynomial

$$x^4 - 11.0x^3 + 42.35x^2 - 66.550x + 35.1384.$$

Define first a grid (set of x points) with $0.6 \leq x \leq 4.9$ and plot the function on this grid (do not forget dots in the dot operations!). Try to find the roots of this polynomial graphically. When you have located the roots approximately refine your grids and try to get the roots with a two digit precision. The command `grid on` is helpful, try it!

Exercise 10
Draw a circle with radius 1 and midpoint at $(0, 0)$.

2.3 3D plots

So far we only made two-dimensional plots—values of y against x. We will now turn to displaying 3D data, i.e., $z = f(x, y)$ will be plotted against x and y. Of course, we must discretize the function parameters and value. Let $\boldsymbol{x} = (x_1, \ldots, x_m)$ and $\boldsymbol{y} = (y_1, \ldots, y_n)$ be vectors and let $\boldsymbol{Z}$ be a matrix of function values

$$Z_{ij} = f(x_j, y_i).$$

Note that x carries the column index j of the $n \times m$ matrix $\boldsymbol{Z}$ and y the row index i. The reason is that the x-axis is usually horizontal (corresponds to running within rows of $\boldsymbol{Z}$) and the y-axis is usually vertical (runs within columns of $\boldsymbol{Z}$).

A MATLAB function that is useful in the computation of $\boldsymbol{Z}$ is `meshgrid`. The statement

```
    [X,Y] = meshgrid(x,y)
```

transforms vectors (one-dimensional arrays) `x` and `y` into two-dimensional arrays `X` and `Y` that can be used with dot operations to compute `Z`. The rows of the output array `X` are simply copies of the vector `x` and the columns of the output array `Y` are copies of the vector `y`. The number of rows of `x` contained in `X` is the length n of `y` and the number of columns of `Y` is equal to the length m of `x`. Example:

```
    >> x=[1 2 3 4];  y=[5 4 3];
    >> [X Y] = meshgrid(x,y)
    X =
1      2     3     4
1      2     3     4
1      2     3     4
    Y =
5      5     5     5
4      4     4     4
3      3     3     3
```

That is, $X_{i,j} = x_j$ (independent of i) and $Y_{i,j} = y_i$ (independent of j). The two matrices `X` and `Y` created by `meshgrid` are of the same shape and can be combined with dot operations, such as `.*` or `./`.

As an example of the use of `meshgrid`, we evaluate the function

$$z = f(x, y) = x \exp(-x^2 - y^2)$$

on an x grid over the range $-2 < x < 2$ and a y grid with $-2 < y < 2$.

```
>> x = -2:0.2:2; y = -2:0.2:2;
>> [X,Y] = meshgrid(x, y);
>> Z = X .* exp(-X.^2 - Y.^2); % Dot operations everywhere
```

Now

$$Z_{i,j} \equiv X_{i,j} \exp(-X_{i,j}^2 - Y_{i,j}^2) = x_j \exp(-x_j^2 - y_i^2) = f(x_j, y_i).$$

The commands `mesh(X,Y,Z)` and `surf(X,Y,Z)` give 3D plots on the screen, while the command `contour(X,Y,Z)` gives the very same information as a contour plot. For presentations the output of `surf` is the most spectacular, but for obtaining quantitative information a contour plot is generally more useful. The difference between `mesh` and `surf` is that the former gives a colored wire frame, while the latter gives a faceted view of the surface.

We just saw that the first three parameters of the 3D plotting commands: `contour`, `mesh`, and `surf` were two-dimensional matrices, all of the same dimension. By use of `meshgrid` we created two such matrices from two vectors. However, this is not necessary, the first two parameters of the 3D plotting commands may as well be vectors. Obviously, *the third parameter* `Z`, containing the function values $Z_{ij} = f(x_j, y_i)$, *must be a* $n \times m$ *matrix* with its column index j corresponding to x values and its row index i corresponding to y values.

When the first two parameters, `x` and `y`, are vectors, their dimensions must agree with those of the third parameter `Z`. That is, if `Z` is an $n \times m$ matrix, then `x` must be of dimension m and `y` must be of dimension n. It is useful that the 3D plotting commands of MATLAB can accept vectors when one wants to plot data generated outside MATLAB (by another program or by measurements as, for instance, 2D NMR). In such a case you usually have at your disposal a vector of x values, a vector of y values and a matrix of function values that correspond to these vectors. There is then no need to blow up the vectors x and y to matrices, which we did by the use of `meshgrid`.

Exercise 11

Write a script that, by using `meshgrid`, draws the “Mexican hat”

$$z = (1 - x^2 - y^2) \exp(-0.5(x^2 + y^2))$$

on the x and y interval $[-3, 3]$. Apply one after the other: `mesh`, `surf`, and `contour`. Use matrices for all parameters.

Hint:
Between the plot commands you may enter in your script the MATLAB command `pause`. This allows you to have a look at the plot until you hit the enter key.

Exercise 12
Return to the previous exercise, change your script so that the first two parameters of `contour`, `mesh`, and `surf` are the vectors that were used in the `meshgrid` command. Verify that the figures are exactly the same as when the first two parameters were the two matrices created by `meshgrid`.

Exercise 13
A normalized hydrogen $3d_{z^2}$ orbital has the form

$$N\,r^2\,e^{-\zeta r}\,(3z^2 - r^2) \ \text{ with } \ r = \sqrt{x^2+y^2+z^2} \ \text{ and } \ N = \frac{1}{3}\sqrt{\frac{1}{2\pi}}\zeta^{7/2}.$$

Write a script that plots the intersection of this orbital with the yz-plane, that is, for $x = 0$. Choose $\zeta = 3$, and let y and z range from -4 to $+4$.
Hint:
Use for contouring something like

```
    [c h]=contour(Y, Z, dorb, [-1:0.05:1]);
    clabel(c,h, 'fontsize',6.5)
```

where `dorb` must contain the values of the $3d_{z^2}$ orbital on the y-z grid, the array `[-1:0.05:1]` gives the contour levels and the second statement puts labels on the contours.

Part II

Programming

3. Scripting, booleans, strings, and loops

3.1 Readability

It is good practice to start a script with a few lines of comments (starting with `%`) to explain the purpose of the script and if necessary how it should be used. These comments can be shown in your MATLAB session by `help anyname`, where `anyname.m` is the name of your script. What is also helpful is to add your name and the date when you wrote script. Also in the script, you should add comments to explain what the statements intend to do and choose the names of your variables wisely to describe what they contain. Both are **absolutely essential** if other people want to use your scripts, but it is also very helpful if you are just make scripts for personal use. If you want to alter the script some time later, it will help you to quickly understand how everything worked again.

Sometimes statements in a script become very long. They can be continued over more than one line by the use of three periods `...`, as for example

```
s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7 ...
      - 1/8 + 1/9 - 1/10 + 1/11 - 1/12 ...
      + 1/13 - 1/14;
```

Blanks around the $=$, $+$, and $-$ signs are optional, but they improve readability. This example also shows the use of indentation. The second and third line start a bit later to indicate that they still belong to the first line. In the examples with loops indentation was used to indicate where the loops starts and ends. Again this is used to improve the readability of the script.

For the two computer assignments that you will need to hand in during this course. One of the things that will be judged is readability (comments, indentation, variable names).

3.2 Flow control

It often happens that one wants to repeat a calculation for different values of a parameter. To this end MATLAB uses the *for loop*. The general form of a `for` statement is:

```
for var = array
  ...
  ...
end
```

Here `array` can be a rectangular array, although in practice it is usually a row vector of the form `x:y`, in which case its columns are simply scalars. The columns of `array` are assigned one at a time to `var` and then the following statements, up to `end`, are executed. For example

```
for i = 1:10
    j = 2 * i
end
i
```

will return

```
j =  2
j =  4
j =  6
j =  8
j =  10
j =  12
j =  14
j =  16
j =  18
j =  20
i =  10
```

Notice that the value of `i` after the loop is 10. For loops can be nested:

```
m = 10;
n = 5;
A = zeros(m,n);             % initialize matrix A
for i = 1:m
  for j = 1:n
    A(i,j) = 1/(i + j - 1);  % fill it piecewise
  end
end
```

Here we have initialized matrix `A` first. It now does not need to grow for every new `i` and/or `j`, which saves computer time. We have introduced variables `m` and `n` to ensure that if we want to change the dimensions of the matrix we only need to change this in one position of our script.

Exercise 14

Create a vector `y` using a `for`-loop with $y = \sin(x)$ for 100 points on the interval 0 and 2π. Plot `x` versus `y`.

Exercise 15

In exercises 11 and 12, you used matrix operations to create a matrix to plot a Mexican hat. This matrix can also be obtained by using a nested `for`-loop. Copy your previous script and change it accordingly. Is the result the same? Which script is faster? (*Hint*: use `tic(); scriptname; toc()` to get the CPU time for `scriptname`.

3.3 Booleans and if-then-else constructions

A boolean is variable that can take two values "True" and "False". MATLAB uses non-zero for true and zero for false. The comparison operator `==`, *i.e.*, `a == b` yields non-zero (1) if `a` and `b` are equal and 0 otherwise:

```
>> 1 == 2
ans =
     0
>> 1 == 1
ans =
     1
```

Table 3.1 gives an overview of the different comparison operators.

Table 3.1: Comparison operators

`==`	equal
`<`	less than
`>`	greater than
`<=`	less or equal
`>=`	greater or equal
`~=`	not equal
`&`	and
`|`	or

We can apply for instance the `<` sign to get all values smaller than 0.5 from vector `d`

```
>> d = rand(4,1)     % A column vector of length 4
d =
    0.6602
    0.3420
    0.2897
    0.3412
>> b = (d < 0.5)
b =
    0
    1
    1
    1
>> find(b)
ans =
    2
    3
    4
```

The command `find` gives the indexes of elements that are "true". We can connect comparison statements with the logical operators AND (`&`) and OR (`|`). The statement `A & B` returns true if both `A` and `B` are true; `A | B` returns true if at least one of the two is true.

We can use the concept of booleans and logical statements for if-then-else constructions. The general syntax of the `if` statement is

```
  if boolean1
    ...           % this is executed if boolean1 is true
  elseif boolean2
    ...           % this is executed if boolean1 is false
    ...           %   and if boolean2 is true
  else
    ...           % executed if both boolean1 an boolean2 are false
  end
```

The `else` and `elseif` parts are optional. Zero or more `elseif` parts can be used as well as nested `if`s. We use this in the following example where we generate a random number and then test whether it is smaller or larger than 0.5

```
  a = rand(1,1);  % generate a random number
  if a > 0.5
    disp('a is larger than 0.5')
  elseif a < 0.5
    disp('a is smaller than 0.5')
  else           % if a is not smaller or larger is must be equal to 0.5
    disp('a is equal to 0.5')
  end
```

Exercise 16
Look at the following three programs and predict what they put on the screen.

```
a=-3;              |  a=-3;              |  a=-3;
if a < -5          |  if a < -5          |  if a < -3
  disp(' < -5 ')   |    disp(' < -5 ')   |    disp(' < -3 ')
elseif a < -2      |  elseif a < -1      |  elseif a < -4
  disp(' < -2 ')   |    disp(' < -1 ')   |    disp(' < -4 ')
elseif a < -1      |  elseif a < -2      |  elseif a < -5
  disp(' < -1 ')   |    disp(' < -2 ')   |    disp(' < -5 ')
else               |  else               |  else
  disp('rest')     |    disp('rest')     |    disp('rest')
end                |  end                |  end
```

Make three script to verify your answers.

3.4 While-loops

In many cases, you would like to run a loop not a fixed number of times but until a certain criterion has been fulfilled. This can be done with a `while` loop. The `while` loop combines the functionality `for` loop and the `if` construction:

```
  while boolean
    ....
    ....
  end
```

The statements in the body of the while loop are executed as long as the `boolean` is true. This construction will work well for, for instance, an iterative optimization procedure

```
input = ....            % initialize the input parameters
y_exp = ....            % set the experimental y values
y = predicty(input);  % predict y on the basis of the input parameters
n = length(y_exp);
error = sum((y - y_exp).^2)/n;
while error > 0.1     % optimization criterion
  input = ....          % change the input parameters
  y = predicty(input);% calculate the new predicted y
  error = sum((y - y_exp).^2)/n;
end
```

Here the while loop is repeated until the `y` values are close to the experimental `y` values. How close is determined by the optimization criterion (0.1 in this case).

Exercise 17

Repeat exercise 14 but use a `while`-loop instead of `for`-loop.

Exercise 18

What is the value of `i` printed as last statement of the following script?

```
V=[rand(2,1); -rand(2,1); rand(2,1); -rand(2,1)]
i = 1;
v = V(i);
while v >= 0
  i = i+1;
  v = V(i);
end
i
```

Exercise 19

You will now write a script that numerical integrates a function $f(x)$ which value we only know on a set of points contained in vector `x` with corresponding function values in vector `y`, $y_i = f(x_i)$. Both vectors are of identical arbitrary length N. Since we do not know the exact analytic form of our function we will have to numerically integrate this function. There are several ways of doing this. For now we approximate the integral by

$$I = \int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{N}\sum_{n=1}^{N} f(x_n^*)$$

where x_n^* is any point in the nth subinterval $[x_{n-1}, x_n]$. If x_n^* is chosen as the left endpoint of the subinterval we have

$$I = \int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{N}\sum_{n=1}^{N} f(x_{n-1}.)$$

This is usually referred to as the **rectangle method**.

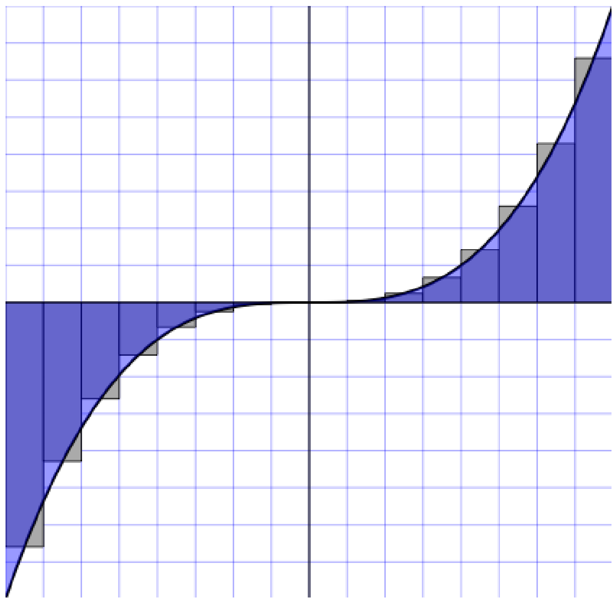

Figure 3.1: The integral of the analytical black curve is approximated by the rectangles underneath the curve, where the height of the rectangle is approximated by the midpoint.

a. Assume that `a`, `b`, `N` and a function are given. Take as a first example `a = 0; b = pi, N = 50` and the function $f(x) = \sin(x)$.
b. Write a script in which you calculate vectors `x`, and `f`.
c. Determine the value of the variable `integral` which contains the numerical integral I of this `f`.

What is the error? Try again with a finer grid ($N = 100$). Apply your script now to `a = 0; b = 2*pi, N = 50` and $f(x) = \cos(x)$.

Exercise 20
Using the left point of the subinterval is a rather crude approximation. With same number of function evaluations, the **midpoint rule** leads to a much better result

$$I = \int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{N} \sum_{n=1}^{N} f(\frac{1}{2}(x_{n-1} + x_n)).$$

This is schematically depicted in Figure 3.1. The idea is that the underestimation on one side (light purple) is compensated by the overestimation at the other side (gray).

a. Use again `a = 0; b = pi, N = 50` and the function $f(x) = \sin(x)$.
b. Write a script in which you calculate vectors `x`, and `f` where x is now at the midpoint of the interval.
c. Determine the value of the variable `integral` which contains the numerical integral I of this `f`.

What is the error? Try again with a finer grid ($N = 100$). How does your result compare to the previous exercise?

3.5 Strings

In one of the previous examples, we ran into fourth type of variable; other than floats (or doubles), integers and booleans that we have discussed so far. This is the *string*, which is a character array. We can enter text into MATLAB by using single quotes. For example,

```
s = 'Hello'
```

The result is not the same kind of numeric array as we have been dealing with up to now. Internally MATLAB stores a letter as a two byte numeric value, not as a floating point number (8 bytes). There exists a well-known convention called ASCII (American Standard Code for Information Interchange) to store letters as numbers. In ASCII the uppercase letters A,. . . , Z have code 65, . . . ,90 and the lowercase letters a,. . . , z have code 97,. . . ,122. The function `char` converts an ASCII code to an internal MATLAB code. Conversely, the statement `a = double(s)` converts the character array to a numeric matrix `a` containing the ASCII codes for each character. Example:

```
>> s = 'hello';

>> a = double(s)
a =
   104   101   108   108   111

>> b = char(a)     % array a contains numbers
b =
   hello
```

The result `a = 104 101 108 108 111` is the ASCII representation of `hello`.

Exercise 21
Write a script that returns the unique matrix elements of a square matrix `A` in a column vector. Use the following algorithm:

- Make a vector out of `A` by `a = A(:)`.
- Sort the vector `a` by the MATLAB command `sort`, now we are assured that the elements appearing more than once are adjacent.
- Loop over the sorted vector and retain of each element only one unique copy.

Apply your script to the matrix:

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 3 & 3 & 5 \end{pmatrix}.$$

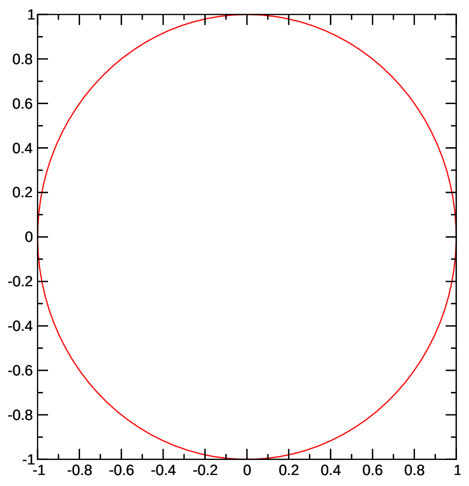

Figure 3.2: Unit circle is used to estimate π.

Exercise 22

Consider a square of length 2 which contains a circle with radius $r = 1$ (see Figure 3.2). If we would randomly place a point in the square, the probability of placing it inside the circle is proportional to the ratio in areas

$$P = \frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{r^2\pi}{4} = \frac{\pi}{4}. \tag{3.1}$$

We can use this information to numerically determine π.

a. Write a function that randomly places a point in the square and test whether it falls in the circle.

b. Use a loop to repeat this and estimate π. Repeat until some desired accuracy is reached.

c. How many repeats are needed to estimate π to an accuracy of `1e-8`? `1e-12`?

4. Reading and writing

4.1 Saving and loading data

So far we have always constructed the matrices by typing their content in the console. However, the real power of MATLAB lies in the matrix manipulations that it can do on large matrices. These matrices you do not want to type in yourself and also the resulting matrices after your manipulations you want to be able to store on disc.

File io (input-output) refers to reading data from and writing data to a file. Files can be classified in two broad types: text files and binary files. The main advantage of binary files is that they have a high information density per byte; files remain relatively small and still contain a high precision. Their main disadvantage is that they cannot simply be opened by any program. For text files the encoding of the data is not optimized. Therefore, for large data sets the file size can grow very large, but the main advantage is that you can make it very accessible for the user.

MATLAB has the commands `load` and `save`. There are two ways of calling these commands: in function form

```
save('file.mat')     % saves all variables in file file.mat
save('file.mat','X') % only save variable X
```

or in command form

```
save file.mat        % saves all variables in file file.mat
save file.mat X      % only save variable X
```

The `mat` file format is a binary data format especially created for MATLAB and it is the default format for loading and saving of data. Data can be obtained again by using the command `load`

```
load('file.mat')     % loads all variables in file file.mat
load file.mat X      % only load variable X
```

Not all programs will allow you to manipulate the `mat` output file. Files saved in ASCII format can be read or imported by any other program. It is good practice to give these ASCII data files the extension `.dat` or `.txt`. However, the number of digits that is written to file is limited and you will therefore lose some precision with respect to the `mat` file format. It depends on the application and type of data whether this is a problem. To save the variable `a` to file `mydata.dat` use

```
save -ascii mydata.dat a
save('mydata.dat','a','-ascii')
```

In principle, one can save multiple variables in one file in this way, but these are all just saved below each other and it is impossible to distinguish them later again. If the variables do not contain the same number of columns, it would even produce an error message upon loading. The free software OCTAVE, which is a MATLAB clone, has a much nicer data format, where it is possible to save multiple variables and keep their names. But back to MATLAB to load the file again use

```
load -ascii mydata.dat
load('mydata.dat','-ascii')
```

This will produce the variable `mydata` which contains matrix from the file. If you would like to put the data into a variable with a different name use `X = load('mydata.dat','-ascii')`.

Exercise 23

a. Download the file `sydrain.txt`. This file can be found on the website and contains the daily rainfall on the wettest day of the year — in millimetres — over a 47-year period in Turramurra, Sydney, Australia.
b. Use the function `load` to load in the content of this file.
c. Plot the data.
d. Determine the maximum daily rainfall in 47 years and the year that it felt (*Hint*: check the help of `maximum`).
e. Save this data to an ascii file and check outside MATLAB that it is indeed the correct data.

Exercise 24

a. Type `format long`. This ensures that all digits are displayed.
b. Calculate $a = 1/3$ and save `a` to an ascii file.
c. Load it again. How much data is lost?
d. Repeat but now in the `mat` file format.

4.2 Input and output control

In this section we will show how you cannot only save vectors and matrices to file, but also additional text information for instance. Especially, if you preform computationally expensive calculations, you do not just want to `disp` (display) the outcome to your workspace, but to save it to file where you can access it later again.

There are three basic steps to accessing data in a file: (i) open the file, (ii) read data from the file *or* write data to the file (but typically not both), and (iii) close the file. A file can be opened by calling

```
  FID = fopen(filename, accessmode)
```

The first argument to `fopen()` is a string that contains the name of the file you want to open. If the file name contains no folder path, then MATLAB will look for the file in its current working directory. You can type `pwd` (**p**resent **w**orking **d**irectory) in your workspace to find what the current working directory is. To change the working directory use the command `cd`. If the file is not found, MATLAB will search the folders defined in its "search paths" and open the first file it finds with a matching name. For example, the function call:

```
  FID = fopen('example.txt', 'r')
```

will find the file `example.txt` only if the file is in the current working folder or in a folder that MATLAB searches. If the file name contains a folder path, then MATLAB will only look in that specific folder for the file. For example,

```
FID = fopen('C:\Documents and Settings\Fred\example.txt','r');
```

or under Linux

```
FID = fopen('/home/fred/example.txt','r');
```

will find the file "example.txt" only if it exists in the folder specified directory.

The second argument to `fopen()` is a string that specifies the type of access that is desired to the file. Table 4.1 gives the possible types. Opening a file for reading will never modify the file in any way, opening a file for writing (`'w'` or `'a'`) will. If the file already exists and contains data, writing to the file will destroy all of its previous contents whereas appending to the file will add new data to the end of the file. If the file does not currently exist, writing or appending to the file will create a new file. So be very careful when writing to a file. If the file already exists, the previous contents will be replaced and destroyed by the new data.

To close a file again use `fclose(FID)` where `FID` is the file identifier that you used to open the file. Always close a file immediately after its use. Not closing a file that you are writing to can potentially cause the loss of some data. Attempts to access a file after it is closed will produce a run-time error.

So now that we know how to open and close a file, we will discuss how you read and write data to the open file. The `fprintf` function writes formatted text to a file.

Table 4.1: Possible file access types

'r'	open an existing file for reading only (default permission)
'r+'	open an existing file for reading and writing
'w'	delete/create a file and open it for writing
'w+'	delete/create a file and open it for reading and writing
'a'	open/create a file for appending (writing data after the existing contents of the file)
'a+'	open/create a file for reading and appending

The first argument is the file identifier, which identifies to which file to write. The second argument is string which can contain format specifiers. A format specifier begins with the % character and is followed by other characters that precisely describe how the corresponding output value will be formatted. Let us look at the example below.

```
   FID = fopen('demo_file.txt', 'w');

   fprintf(FID, '12345678901234567890\n');
   fprintf(FID, '%-10d\n', floor(pi));
   fprintf(FID, '%+10d\n', floor(pi));
   fprintf(FID, '%10d\n', floor(pi));
   fprintf(FID, '%9d\n',  floor(pi));
   fprintf(FID, '%8d\n',  floor(pi));
   fprintf(FID, '%08d\n',  floor(pi));
   fprintf(FID, '%g\n', pi);
   fprintf(FID, '%f\n', pi);
   fprintf(FID, '%10.2f\n', pi);
   fprintf(FID, '%0.2f\n', pi);
   fprintf(FID, '%c\n', 'Test');
   fprintf(FID, '%s%s\n', 'Test', 'text');
   fprintf(FID, '%10s%10s\n', 'Test','text');
   fprintf(FID, '%-10s%-10s\n', 'Test','text');

  fclose(FID);
```

The above commands produce the following formatted text file:

```
12345678901234567890
3
        +3
         3
        3
       3
00000003
3.14159
3.141593
      3.14
3.14
T
e
s
t
Testtext
```

```
      Test      text
Test      text
```

To understand this output we need to explain how format specifiers work. They always contain a single letter (format descriptor) that describes the data type of the output value. The most commonly used data types are listed in Table 4.2. A format specifier can contain three more parameters: a modifier, and parameters describing the field width and precision. The overall format of the format descriptor looks like

```
%modifier fieldWidth . precision formatDescriptor
```

where modifier, field width, and precision values are optional. The precision parameter only applies to floating point numbers. It specifies how many digits to display after the decimal point. The field width is an integer that specifies how many spaces to use for the displayed value. If the field is too small, the `fprintf` function will expand the field to make it large enough to display the entire value. If the field is too large, the modifier value will determine where in the field the value is placed. With a `-` sign the output is left-justified ("links uitgelijnd") in its field (no minus sign means right-justify), with `+` the sign (plus or minus) is printed with the output value and finally a `0` pads the field with leading zeros (`0003` for the integer `3` in a field with a width of four). The `fprintf` replaces each format specifier with a value from the `fprintf` function's argument list. We see for instance in line 11 of the example two format identifiers and also two additional arguments in the `fprintf` statement.

Table 4.2: The most common format descriptors.

`%d`	display the value as an integer (this only works if the number really is an integer)
`%e`	display the value in exponential format (`3.23e5`)
`%f`	display the value in floating point format (`3.451`)
`%g`	display the value showing only significant digits – use this most of the time
`%s`	display the value as a string - a series of UNICODE characters
`%c`	display the value as a single UNICODE character

Finally, we need to explain the special escape sequences, beginning with the backslash character, are used to represent "non-printable" characters. The most commonly used escape sequences are:

`\n` a "new line" character (start a new line of output)
`\t` a "tab" – spaces over to the next tab stop
`\r` a "carriage return"
`\\` output a single backslash

Exercise 25
Print the above example to command window. Use `fprintf` without the file identifier. Try to understand how the format identifiers result in this output.

Exercise 26
Write a script that loads the `wine.dat` file which you can find on the website `wine.dat`. Download this file to your own MATLAB work directory. Save the data of each of the columns in a different file. The first column in `wine01.dat`, the second in `wine02.dat`, etc. Use a `for loop` for this. All data should be right-justified and you should not lose any precision from the original data.
Hint:
Use `sprintf` to generate your filename.

5. Functions

5.1 Functions

Functions resemble scripts: both reside in an ASCII file with extension `.m`. As a matter of fact, many of the MATLAB functions that we used so far are simply `.m` files. One can inspect them by the command `type`, e.g., `type fliplr` will type the content of `fliplr.m`. The difference between scripts and functions is that all variables in scripts are global, whereas *all variables in functions are local.* This means that that the variables that are made inside functions only exist inside the function. Say, we have assigned the value -1.5 to the variable `D` in our MATLAB session and we call a function that also assigns a value to a variable called `D`. Upon return from the function the variable `D` still has the value -1.5. Conversely, `D` must be assigned a value inside the function before it can be used. The function does not know that `D` already has a value in our MATLAB session.

Scripts and functions are stored in an `.m` file. However, functions start with the word `function` which tells MATLAB that the `.m` file contains a function and not a script. If all variables in a function are local, what is the use of a function? One would like to get something useful out of a function, as for instance an array with its columns flipped. In other words, how does a function communicate with its environment? To explain this we give the syntax of the first line of a function:

```
function [out1, out2, ...] = name(in1, in2, ...)
```

The first observation is that a function accepts input arguments between round brackets. Any number (including zero) of variables can be written here. These variables can be of any type: scalars, matrices (and MATLAB objects that we have not yet met and will be explained in the appendix). The second observation is that a function returns a number of output parameters in square brackets. The values of `out1`, `out2`, etc. must be assigned within the function. These variables can also be of any type. *The square brackets here have nothing to do with the concatenation operators that made larger matrices from smaller ones.* The third observation is that the function has a name. It is common to use here the same name as of the `.m` file, but this is not compulsory. The name of an `.m` file begins with an alphabetic character, and has a filename extension of `.m`. The `.m` filename, less its extension, is what MATLAB searches for when you try to use the script or function.

For example, assume the existence of a file on disk called `stat.m` and containing:

```
function [average,stdev] = stat(x)
% STAT  Returns the average and standard deviation of vector
% average = STAT(x)
% [average,stdev] = STAT(x)
%
% HMC 06/10/2014
  n = length(x);
  average = sum(x)/n;
  stdev = sqrt(sum((x - average).^2)/n);
```

This defines a function called `stat` that calculates the average and standard deviation of the components of a vector. We emphasize again that the variables within the body of the function are all local variables. We can call this function in our MATLAB session in three different ways:

```
>> stat(x)          % no explicit assignment
ans =               % default assignment to ans
    0.0159

>> m = stat(x)      % assignment of first output parameter
m =
  0.0159

>> [m, s] = stat(x) % assignment of both output parameters
m =
    0.0159
s =
    1.0023
```

Note that `m` and `s` are *not* concatenated to a 1×2 matrix; the square brackets *do not* act as concatenation operators in this context, i.e., to the left of the assignment sign. Notice that the concatenation operators `[` and `]` only appear on the right hand sides of assignments.

You might wonder when to use a script and when a function. One typically uses scripts as the main program. Here you will define or load your data, do all data manipulations and then plot the results. Functions are used for general calculations like calculating the standard deviation or an average. The data manipulations inside the script is usually done by calling different functions.

Exercise 27

The routine from exercise 21 would be typically something one would put in a function.

Convert the script you made for this exercise into a function which needs a matrix as input and returns a column vector with unique elements. Apply your new function to

$$\boldsymbol{A} = \begin{pmatrix} -5 & 7 & -3 \\ 7 & 9 & -2 \\ 6 & -2 & 3 \end{pmatrix}$$

and

$$\boldsymbol{B} = \begin{pmatrix} 1 & 2 & 5 & 6 & 2 \\ 3 & -5 & 4 & 2 & 3 \\ -2 & 1 & 1 & 2 & 1 \\ 3 & 4 & 2 & -5 & 2 \end{pmatrix}.$$

A *subfunction*, visible only to the other functions in the same file, is created by defining a new function with the function keyword after the body of the preceding function or subfunction. For example, `avg` is a subfunction within the file `stat.m`:

```
function [average,stdev] = stat(x)
% STAT  Returns the average and standard deviation of vector
% average = STAT(x)
% [average,stdev] = STAT(x)
%
% HMC 06/10/2014
  n = length(x);
  average = avg(x,n); % function is defined at the bottom of the file
  stdev = sqrt(sum((x - avg(x,n)).^2)/n);
function [average] = avg(x,n)
   average  = sum(x)/n;
```

Subfunctions are not visible outside the file where they are defined. Also, subfunctions are not allowed in scripts, only inside other functions. Functions normally return when the end of the function is reached. We may use a `return` statement to force an early return from the function. When MATLAB does not recognize a function by name, it searches for a `.m` file of the same name on disk. If the function is found, MATLAB stores it in parsed form into memory for subsequent use. In general, if you input the name of something to MATLAB, the MATLAB interpreter does the following: (i) it checks to see whether the name is a variable, (ii) it checks to see whether the name is a built-in function (as, for instance, `sqrt` or `sin`), and finally (iii) it checks to see whether the name is a local subfunction. When you call an `.m` file function from the command line or from within another `.m` file, MATLAB parses the function, checks for syntax errors, and stores it in memory. The parsed function remains in memory until cleared with the clear command or you quit MATLAB.

5.2 Errors

So far you have probably already had several error messages when you worked on your exercises. They were most likely due to matrix dimension problems, but they could also be related to syntax errors (usually typos) or calls to variables that do not exist. Whenever you enter a statement, MATLAB does many checks on this statement to test whether it is correct and if not to give an error message to indicate what is wrong with the statement. For instance, if you type `sin(pi,0)`, it gives the error message

```
Error using sin
Too many input arguments.
```

Most large programs contain errors. We can generally classify three types of errors: (i) syntax errors, (ii) run-time errors, and (iii) logic errors. A *syntax error* is a violation of the rules of the programming language. For example, each of the lines below contain one syntax error – can you identify the errors?

```
[1 2; 3 4 5]
for count 1:10
```

The Matlab interpreter catches syntax errors and displays a red error message beginning with `???`. Matlab programs are checked for syntax errors before execution of the program begins. A single syntax error will prevent your program from executing. If you click on the syntax error "hyper-link", the cursor in the editor window will be placed in the exact spot where MATLAB recognized a syntax problem. The syntax error is on the line indicated by the cursor, or possibly on previous lines. The syntax error will never be below the indicated line. Syntax errors are relatively easy to find and fix.

A *run-time error* is an error that results from using invalid operand values for an operation. The following statement generates a run-time error if `count` has the value zero, since division by zero is not mathematically defined.

```
Average = Total/Count;
```

The Matlab interpreter usually displays a warning message when a run-time error occurs, but does not halt the running program. In most cases, any calculations or processing after a run-time error will produce incorrect results. **Never simply ignore** run-time error messages.

A *logic error* in a program is any code that causes incorrect output/results even though the program runs to completion. The following code contains a logic error because the `>` should be `>=`

```
if age > 18
  disp('You are old enough to vote!')
end
```

A program with a logic error may give the correct answer sometimes and the wrong answer other times. Logic errors typically are the most difficult type of errors to find and correct. Finding logic errors is the primary goal of testing. Some common errors for Matlab programmers are using two different names for the same variable, *e.g.*, `Count` and `count` and forgetting to initialize or modify a loop control variable.

5.3 Program debugging and testing

Debugging is the process of locating and correcting errors ("bugs") in a program. Experience teaches us that there are no "shortcuts". If you program something quickly without much thought of program design (diving it in several separate functions), readability, and comments, it will bite you in the tail and you will be much more likely to have bugs and it will be much harder to find them.

However, even very nicely written code is likely to have some bugs. The first step is then to read the error message that Matlab gives and start from there. Testing your program along the way also helps. The purpose of testing is to provide a large range in different input data to your code to find logic errors. These should be both expected and unexpected (invalid) inputs. Unexpected values may be negative, extremely large, or extremely small values. You should test your program with boundary conditions values, *i.e.*, values at and around a value for which the behavior of a program should change (`age = 17, 18, 19` in the example above). When testing loops, provide inputs that cause the loop to repeat zero, one, and many times. When testing if statements, provide inputs that test all code sections of the if statement.

Finally, a general rule for all debugging:

When in doubt, print it out

Exercise 28
Find the syntax error, the runtime error and two logic errors in the following function:

```
function printgreeting(greeting, number)
% PRINTGREETING prints a greeting a number of times with a maximum of 20
% PRINTGREETING(greeting, number)
%
% HMC 18/11/2014
  if (number >= 20 )
    disp('Your number is too large'), disp(number)
  else
    disp ('Your greeting will be printed '), disp(number-1), disp('times')
    for i = 1:number
      disp (Greeting)
  end
```

Which input values would you provide to test this function for logical and runtime errors?

5.4 Error handling

Error-free code is not necessarily good code. Good code also should be readable, maintainable and reasonably efficient. Now let us return to our initial function example `stat`. This function needs one input argument which should be a vector. This is indicated in the help of the function (first lines of comments), but it is also good to check whether this is fulfilled

```
function [average,stdev] = stat(x)
% STAT  Returns the average and standard deviation of vector
% average = STAT(x)
% [average,stdev] = STAT(x)
%
% HMC 06/10/2014
```

```
      % Test for the number input arguments
      if (nargin > 1)
        error('Too many input arguments')
      elseif (nargin < 1)
        error('Missing argument')
      end

      % Test whether x is a vector
      [n1,n2]  = size(x);
      if not(n1 ~= 1 & n2 == 1) & not(n1 == 1 & n2 ~= 1)
        error('x is not a vector')
      end

      % Calculation of average and standard deviation
      n        = max(n1,n2);
      average  = avg(x,n);
      stdev    = sqrt(sum((x - average).^2)/n);


   function [average] = avg(x,n)
   % internal averaging function
      average  = sum(x)/n;
```

Explanation:
nargin is the number of arguments that is passed to a function. The function error('message') prints the message and aborts the function.

Exercise 29
Write the function

```
   function [r, theta, phi] = cartpol(x)
```

that returns spherical polar coordinates from the 3-dimensional column vector x that contains Cartesian coordinates. Check in your function for the number of input arguments and the length of x. Remember the equations ($0 \le r < \infty,\ 0 \le \theta \le \pi,\ 0 \le \phi < 2\pi$)

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} r\sin\theta\cos\phi \\ r\sin\theta\sin\phi \\ r\cos\theta \end{pmatrix}$$

Exercise 30

a. Create the function stat as given in this chapter. Can you call the subfunction avg from the command line?

b. Create a script in which you first load the file sydrain.txt. This file can be found on the website and contains the daily rainfall on the wettest day of the year — in millimetres — over a 47-year period in Turramurra, Sydney, Australia.

c. Call the function `stat` to find the mean and standard deviation of this data and display it in such a way that it is easy to read.

d. Modify the `stat` function to accept matrices as input. Use this function to find the average and standard deviation of the data in the file `wine.txt` for each chemical individually and the total average and standard deviation.

Exercise 31

There are many situations where you would like to sort a list of values in increasing or decreasing order. MATLAB has the function `sort` to do this, which is based on a Quicksort algorithm. In this exercise, we will look at a more simple algorithm “Bubble sort”.

The idea behind Bubble sort is to make multiple passes through a list and during a pass compare each pair of adjacent elements. It then puts these two elements in the right order. An example is given in Figure 5.1. At the end of this pass the number 93 is on the correct position and it took 8 ($n-1$ for length n) comparisons to get there. In the next pass, only 7 or $n-7$ comparisons are required to put the next number in the correct place (77 in this case) and so on.

a. How many comparisons are needed to sort a list of n length?

b. Write a function that can perform a Bubble sort.

c. Test on several vectors of different lengths (`n = 3.^[2:7]`). Make a loglog-plot of n versus CPU time. Use `tic();bubblesort(a);toc()` to measure the CPU time. Does it scale as expected?

d. Perform the same test for the Quicksort algorithm of MATLAB. Which one scales better?

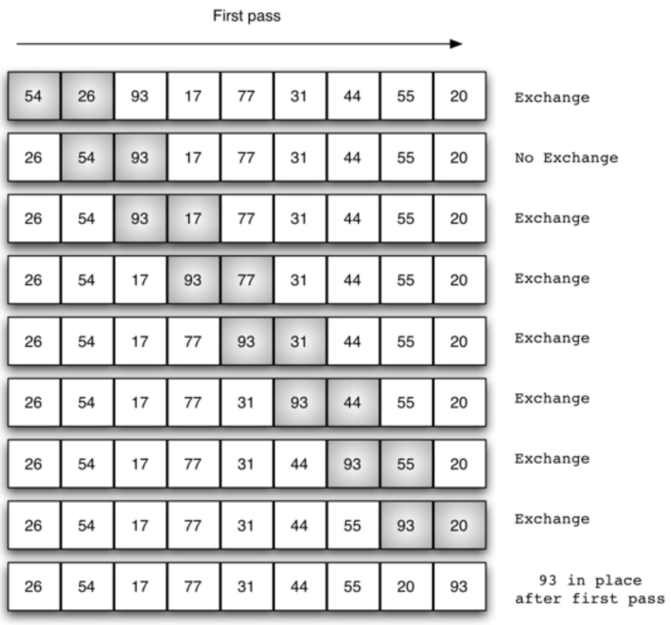

Figure 5.1: Example of a single pass in a bubble sort algorithm (taken from http://interactivepython.org)

6. Function functions

MATLAB has a number of functions that work through functions. This sounds cryptic, so let us give an example immediately. The built-in function `fminsearch` performs minimization. It finds minima of non-linear functions of one or more variables. It does not apply constraints. That is, the search for minima is on the whole real axis (or complex plane as the case maybe). We invoke this function by

```
X = fminsearch(@fun,X0)
```

The search starts at `X0` and finds a local minimum of `fun` at the point `X`. It is our duty to supply the function `fun`. The function must accept the input `X`, and must return the scalar function value `fun` evaluated at `X`. The input variable `X` can be a scalar, vector, or matrix.

Suppose we want to find the local minima of the following function:

```
function y = hump(x)
% We use dot operations so that hump may be called
% with a vector, e.g., for plotting.
y = 1 ./ ((x-0.3).^2 + 0.01) + 1 ./ ((x-0.9).^2 + 0.04) - 6;
```

which we have as `hump.m` on disk. The function call

```
x = fminsearch(@hump,0.3)
```

now returns the value of `x` for which the function has a minimum. (Or, in case of more minima, the local minimum closest to 0.3).

The function `fminsearch` belongs to the class of functions called "function functions". Such a "function function" works with (non)linear functions. That is, one function works on another function. The function functions include

- Zero finding
- Optimization
- Quadrature (numerical integration)
- Solution of ordinary differential equations

See table 6.1 for the most important function functions.

Sometimes the function to be processed is so simple that it is hardly worth the effort to create a separate `.m` file for it. For this MATLAB has the function `inline`. The function `fzero` finds zeros of polynomials, i.e., values of x for which a polynomial in x is zero. The function `quad` gives a quadrature (numerical integration). Thus, for example, enter

Table 6.1: *Function functions*

dblquad	Numerical double integration
fminbnd	Minimize a function of one variable
fminsearch	Minimize a function of several variables
fzero	Zero of a function of one variable
ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb	Solve ordinary differential equations
odefile	Define a differential equation problem for ODE solvers
odeget	Extract properties from options structure created with odeset
odeset	Create or alter options structure for input to ODE solvers
quad, quad8	Numerical evaluation of integrals
vectorize	Vectorize expression

```
>> fzero(inline('x^3-2*x-5'),1.5)
>> quad(inline('1./(1+x.^2)'),-1000, 1000)
```

to find the real zero closest to 1.5 of the third degree polynomial and to integrate numerically the Lorentz function $1/(1+x^2)$. (This integral is known from complex function theory to be π).

Exercise 32
The **bisection method** is an iterative method that finds the root or zero-point of a function on interval (x_1, x_2). It consists of the following steps

a. Calculate c, the midpoint of the interval, `c = 0.5 * (x1 + x2)`.

b. Calculate the function value at the midpoint, $f(c)$.

c. If convergence is satisfactory (that is, $f(c)$ is sufficiently small), return c and stop iterating.

d. Examine the sign of $f(c)$ and replace either $(x_1, f(x_1))$ or $(x_2, f(x_2))$ with $(c, f(c))$ so that there is a zero crossing within the new interval.

You will now write a function function

```
function x0 = bisection(myfun, x1, x2, tol)
```

that uses this method to determine the root of function `myfun` with a precision `tol` ($|f(x)| <$`tol`).

a. Which type of loop would be most appropriate for this problem? Program this in your function.

b. Off course, for this algorithm to work `x1` and `x2` should fulfill several requirements: $x_1 < x_2$ and $f(x_1) < 0$ and $f(x_2) > 0$ or $f(x_1) > 0$ and $f(x_2) < 0$. Add checks for this.

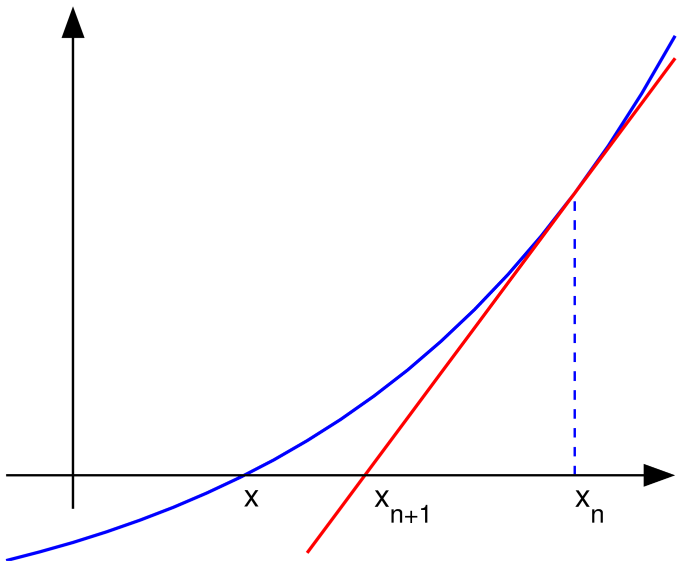

Figure 6.1: Schematic of the determination of a new point in the Newton-Raphson method.

c. Add comments where necessary.

d. Test your function to find the roots of the polynomial in exercise 9.

Exercise 33

Another root finding algorithm is the **Newton-Raphson** method. This is again an iterative method. It approximates the function at the current point with a linear curve $y_n = ax_n + b$. The slope a is calculated by taking the numerical derivative in the current point n. The zero point of this linear curve is then taken as the next point. This procedure is schematically depicted in Fig. 6.1.

a. Write a function function `newton` which input a function, an initial x value and a tolerance.

b. Do you need more or less iterations than the bisection method to obtain the root of the polynomial in exercise 9?

6.1 Optimization

Function optimization is an often recurring problem in science. Think of finding the most optimum set of atomic positions inside a molecule, for instance a protein, given an energy function. But this can also be the optimal conditions for obtaining the maximum yield of a compound, e.g. % yield as a function of reflux time and of excess of a particular reagent, or (in HPLC) the resolution of two or more chromatographic peaks as a function of the flow rate and the composition of the eluant. In maths classes, you have learned to take the first derivative and equate this zero and then solve this equation to come to the optimum. This works fine if an analytical description is known and this is differentiable and in low dimensions. In the examples that were just mentioned, we need to resort to numerical solutions. Number of variables to optimize to come to an optimal molecular geometry is simply too high ($3 \times N$ for N atoms) and in the other two cases no function is known.

There are different classes of optimization methods. Direct methods only use the function results (or the measured values); other methods use the first and/or second derivatives of the function to estimate the step direction and step size. The function function `fminsearch` uses the Simplex method to perform an optimization which is an example of direct method. It aims to find the global extreme (maximum or minimum) of any n-dimensional function $F(x_1, x_2, .., x_n)$, searching through parameter space ("search area"). A 2-dimensional simplex starts with three function evaluations obtained with three different (x_1, x_2) parameter settings ("guesses"). These three observations correspond to the vertices of a triangular constituting the first simplex. In 3-dimensional space four initial function evaluations are required defining a tetrahedral body, and so on for the impossible to be visualized spaces of higher than three dimensions.

Exercise 34
A classic test example for twodimensional minimization is the Rosenbrock banana function $f(x, y) = 100(y - x^2)^2 + (1 - x)^2$.

a. Write the MATLAB function `banana` that evaluates f for the vector $\boldsymbol{r} = (x,\, y)$.
b. Call 'by hand' `fminsearch` from your MATLAB session to minimize the banana function. Use as a start $\boldsymbol{r} = (-1.2,\, 1)$.

Sometimes input functions have parameters in addition to the independent variable. The extra parameters can be data, or can represent variables that do not change during the optimization. For example, suppose you want to minimize the function $f(x, y) = 100(y - x^2)^2 + (a - x)^2$ for different values of a. Function functions generally accept objective functions that depend only on a single variable (r in this case). We can use an anonymous function

```
>> a = 1;
>> f = @(r)banana(r,a);
```

where we give `a` a specific value and we then create an anonymous function `f` with variable r and fixed a.

c. Modify function `banana` to $f(x, y) = 100(y - x^2)^2 + (a - x)^2$ and find the minimum of the function for $a = 1, 2$, and 3.

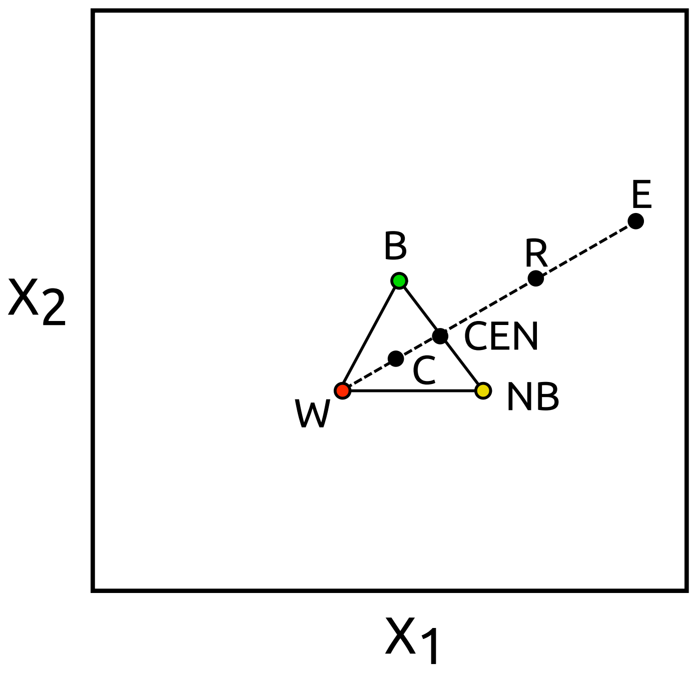

Figure 6.2: Schematic of a construction of a new simplex

Exercise 35

In this exercise, you will write your own **simplex algorithm** for optimization in two dimensions. After construction the first simplex the algorithm consists of the following steps:

a. Rank the three points as best (B), next best (NB), and worst (W). (see Figure 6.2)

b. Determine the centroid (CEN) between B and NB.

c. Determine the reflection R of W vs CEN.

d.
   - If R is within the "search area" and its function value is better than of W but not better than that of B, then a new simplex is formed by R, B, and NB.
   - Else if E is within the "search area" and the function value is better than that of B, than the new simplex is E, B, and NB (where the distance E–R is that same as CEN–R)
   - Otherwise, the new simplex is C, B, and NB (where the distance W–C is that same as C–CEN).

e. The process is repeated from step 1.

The iterations are terminated when no more significant improvement of the function values is observed on moving from one simplex to the other and/or the displacements are insignificant.

Test your function function to the banana function with $a = 1, 2$, and 3.

Part III

Applications

7. Solving differential equations

7.1 Coupled first order ordinary differential equations

One common problem in computational chemistry is to solve a set of coupled ordinary differential equations. An ordinary differential equation has only one parameter, which often is the time t. We will restrict ourselves in this section to first order equations, i.e., equations containing only first derivatives with respect to t. Usually differential equations are coupled and have the following general form,

$$\begin{pmatrix} \mathrm{d}y_1/\mathrm{d}t \\ \mathrm{d}y_2/\mathrm{d}t \\ \vdots \\ \mathrm{d}y_n/\mathrm{d}t \end{pmatrix} = \begin{pmatrix} f_1\left(t, \boldsymbol{y}(t)\right) \\ f_2\left(t, \boldsymbol{y}(t)\right) \\ \vdots \\ f_n\left(t, \boldsymbol{y}(t)\right) \end{pmatrix}, \tag{7.1}$$

with $\boldsymbol{y}(t) = \big(y_1(t), y_2(t), \ldots, y_n(t)\big)$.

One clear application of ODEs is in reaction kinetics. Remember that for a first order reaction

$$A \xrightarrow{k} 2B \tag{7.2}$$

the concentration of A, $[A]$ decreases in time according to

$$\frac{\mathrm{d}[A](t)}{\mathrm{d}t} = -k[A](t) \tag{7.3}$$

and the concentration of $[B]$ increases by

$$\frac{\mathrm{d}[B](t)}{\mathrm{d}t} = 2k[A](t). \tag{7.4}$$

For second order reaction

$$A + B \xrightarrow{k'} C \tag{7.5}$$

the concentration changes according to

$$\frac{\mathrm{d}[A](t)}{\mathrm{d}t} = -k'[A](t)[B](t). \tag{7.6}$$

By integrating this type of equations, one can follow the evolution of a chemical system in time. In the latter case, the equations for $[A]$ and $[B]$ are clearly coupled.

7.2 Euler integrator

In mathematics and computational science, the Euler method is a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value. The Euler method is named after Leonhard Euler, who treated it in his book

Institutionum calculi integralis (published 1768-70). It is the most basic explicit method for numerical integration of ordinary differential equations and often serves as the basis to construct more complex methods. Given the time derivative of a function at t_i, $(\mathrm{d}f/\mathrm{d}t)_{t=t_i}$, and the function value at t_i, $f(t_i)$, the value of f at $t_{i+1} = t_i + \Delta t$ is simply approximated by

$$f(t_{i+1}) \approx f(t_i) + \left(\frac{\mathrm{d}f}{\mathrm{d}t}\right)_{t=t_i} \Delta t. \tag{7.7}$$

Exercise 36

a. Write a **function function** `euler` that integrates a function $f(t)$ from t_0 to t with N steps given $f(t_0)$.

b. Test your function function on the following set of equations

$$\frac{\mathrm{d}y_1}{\mathrm{d}t} = -y_2 \tag{7.8}$$

$$\frac{\mathrm{d}y_2}{\mathrm{d}t} = y_1 \tag{7.9}$$

Use as initial conditions $y_1(0) = 1$ and $y_2(0) = 0$ and propagate for 400 points between $t = 0$ and $t = 2\pi$. Plot and compare against the analytical solution.

MATLAB has several ODE solvers already build in. This chapter will discuss how to use these solvers and circumvent possible numerical issues that you might run into.

The ODE solver most often used is `ode45`. It is called as

```
[t Y] = ode45(@myfun, timespan, y0)
```

`myfun` is the name of the `.m`-file that returns $f_1, f_2, \ldots, f_n$ as a column vector. The array `timespan = [t0 tf]` contains the initial and final time, i.e., the equations are integrated from `t0` to `tf`. The initial y values (for $t = t_0$) are given in the vector `y0`. The $K \times n$ matrix `Y`, returned by `ode45`, contains concentrations at K points t_i in time $t_0 \leq t_i \leq t_f$ for which MATLAB computed the vector $\boldsymbol{y}(t_i)$. The corresponding t values are in the $K \times 1$ array `t`. Before calling `ode45` the user does not know K, the number of integration points that MATLAB will generate. The time points, contained in the column vector `t`, are also known only after the call.

First `ode45` assigns the values of `y0` to the first row of `Y`. During integration `ode45` will repeatedly call `myfun.m` with the respective rows of `Y` as input parameters. The function `myfun` returns f_1, f_2, ... for given time t. Besides `Y`, also the time `t` (a scalar) is inputted. Although `t` is not always used explicitly, time must be present in the parameter list of the function, since `ode45` expects a parameter in this position. The function call of `myfun` may look as follows

```
   function  [f] = myfun(t, y).
```

Exercise 37

The Brusselator model of Oscillating Chemical Reactions was proposed by I. Pregogine and his collaborators at the Free University of Brussels. The reaction mechanism is:

$$A \xrightarrow{k_1} X \tag{7.10}$$

$$2X + Y \xrightarrow{k_2} 3X \tag{7.11}$$

$$B + X \xrightarrow{k_3} Y + C \tag{7.12}$$

$$X \xrightarrow{k_4} D \tag{7.13}$$

$$\tag{7.14}$$

which results in a net reaction of A + B $\longrightarrow$ C + D with transient appearance of intermediates X and Y.

1. Write a function that returns the two coupled differential equations for the intermediate concentrations [X] (`C(1)`) and [Y] (`C(2)`): `dCdt = diffeqs(t, C, k, A, B)`, given a vector k with the rate constants and constant concentrations [A] and [B]. We assume that they are added to the system at the same rate as they are consumed in the reactions.
2. Perform a numerical integration from time 0 to 50 (arbitrary units) using `ode45`. Pass $k_i = 1$ for each rate constant and [A] = 1 and [B] = 3, using an anonymous function.
3. Make two plots of the results: -log([X]) and -log([Y]) versus time, and -log([Y]) versus -log([X]).

7.3 Higher order differential equations

An ordinary n^{th} order differential equation can be reduced to a coupled system of n first order differential equations. To show this we introduce the notation: $y' \equiv \mathrm{d}y/\mathrm{d}t$, $y'' \equiv \mathrm{d}^2y/\mathrm{d}t^2$, $y^{(k)} \equiv \mathrm{d}^ky/\mathrm{d}t^k$. An n^{th} order differential equation can be written as

$$\frac{\mathrm{d}^n y}{\mathrm{d}t^n} = f(t, y, y', y'', \ldots, y^{(n-1)}),$$

where f indicates that the right hand side is a given function of t and the lower order derivatives. We simply put

$$\begin{aligned}
y_1 &\equiv y \\
y_2 &\equiv \frac{\mathrm{d}y_1}{\mathrm{d}t} = \frac{\mathrm{d}y}{\mathrm{d}t} \\
y_3 &\equiv \frac{\mathrm{d}y_2}{\mathrm{d}t} = \frac{\mathrm{d}^2y}{\mathrm{d}t^2} \\
\cdots &\quad \cdots \\
y_n &\equiv \frac{\mathrm{d}y_{n-1}}{\mathrm{d}t} = \frac{\mathrm{d}^{n-1}y}{\mathrm{d}t^{n-1}},
\end{aligned} \tag{7.15}$$

and get the set of n first order coupled equations

$$\begin{aligned}
\frac{\mathrm{d}y_1}{\mathrm{d}t} &= y_2 \\
\frac{\mathrm{d}y_2}{\mathrm{d}t} &= y_3 \\
\dots & \quad \dots \\
\frac{\mathrm{d}y_{n-1}}{\mathrm{d}t} &= y_{n-2} \\
\frac{\mathrm{d}y_n}{\mathrm{d}t} &\equiv \frac{\mathrm{d}^n y}{\mathrm{d}t^n} = f(t, y, y', y'', \dots, y^{(n-1)}) \\
&= f(t, y_1, y_2, y_3, \dots, y_n).
\end{aligned} \tag{7.16}$$

Example

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = t\frac{\mathrm{d}y}{\mathrm{d}t} - y^2 \tag{7.17}$$

becomes

$$f_1 \equiv \frac{\mathrm{d}y_1}{\mathrm{d}t} = y_2 \tag{7.18}$$

$$f_2 \equiv \frac{\mathrm{d}y_2}{\mathrm{d}t} = ty_2 - y_1^2. \tag{7.19}$$

In MATLAB the corresponding function called by the ode solvers would look like:

```
function [f] = difeq(t, y)
f = zeros(2,1);  % Tell matlab we return a column vector
f(1) = y(2);
f(2) = t*y(2) - y(1)^2;
```

Exercise 38

Solve Eq. (7.17) by `ode45` with initial conditions $y_1(0) = 1$ and $y_2(0) = 0$. Integrate from $t = 0$ to $t = 0.2$. Plot the resulting function $y(t) \equiv y_1(t)$.

7.4 Stiff differential equations

For many real-life situations, one runs into the problem of stiffness. Stiffness is a subtle, difficult, and important concept in the numerical solution of ordinary differential equations. It depends on the differential equation, the initial conditions, and the numerical method. An ordinary differential equation problem is stiff if the solution being sought is varying slowly, but there are nearby solutions that vary rapidly, so the numerical method must take small steps to obtain satisfactory results. Imagine you hike in the mountains. You are in a narrow canyon on a rather flat trail (solution is varying slowly) with steep walls on either side (nearby solutions that vary rapidly). An explicit algorithm, like Euler, would sample the local gradient to find the descent direction (directions of the walls), and you will be send you bouncing back and forth from wall to wall and you will eventually get home, but it will be long after dark before you arrive.

Table 7.1: ODE solvers implemented in MATLAB and when to use them

Solver	Problem Type	Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. This should be the first solver you try.
ode23	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
ode113	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
ode15s	Stiff	Low to medium	If ode45 is slow because the problem is stiff.
ode23s	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
ode23t	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
ode23tb	Stiff	Low	If using crude error tolerances to solve stiff systems.

Hence, stiffness is an efficiency issue. Nonstiff methods can solve stiff problems as long as the step size is small enough. They just take a long time to do it. Fortunately, algorithms exist that can handle stiff problems. These methods do more work per step, but can take much bigger steps. Stiff methods are implicit. At each step they use MATLAB matrix operations to solve a system of simultaneous linear equations that helps predict the evolution of the solution. Let us return to the hike example, an implicit algorithm would have you keep your eyes on the trail and anticipate where each step is taking you. It is well worth the extra concentration.

The MATLAB ode solvers that can handle stiff problems are called have an additional `s` at the end of the function name, *e.g.*, `ode15s`. An overview of the different ODE solvers in MATLAB is given in Table 7.1.

Exercise 39
Let us look at the following differential equation $\frac{\mathrm{d}r}{\mathrm{d}t} = r^2 - r^3$ with initial condition r_0, which we would like to integrate from $t = 0$ until steady state is reached. If you do not look to strictly at the units, this simple model can be used to describe the propagation of a flame. When you light a match, the ball of flame, with radius r, grows rapidly until it reaches a critical size. Then it remains at that size because the amount of oxygen being consumed by the combustion (volume $\propto r^3$) in the interior of the ball balances the amount available through the surface ($\propto r^2$). Depending on the choice of the initial radius r_0, which should be "small", this model can become stiff.

a. Plot $\frac{\mathrm{d}r}{\mathrm{d}t}$ versus r. Do you understand why we cannot use $r_0 = 0$ as initial condition? What would be the steady state radius $\frac{\mathrm{d}r}{\mathrm{d}t} = 0$?

b. Does the time it takes to reach this steady state radius depend on r_0? If yes, how?

If we would use the Euler integrator with step size Δt, the radius would propagate through

$$r_{i+1} = r_i + \frac{\mathrm{d}r}{\mathrm{d}t}\Delta t. \tag{7.20}$$

Let us look at the case where we almost reached steady state:

$$r_i = 1 - \delta. \tag{7.21}$$

c. What would r_{i+1} be?

The general solution for n steps close to steady state is

$$r_{i+n} = 1 - \delta(1 - \Delta t)^n \tag{7.22}$$

d. We want r to reach 1. What requirement gives this to Δt?

e. The time to reach steady state is $t_{\mathrm{ss}} = 2/r_0 (= t_{\max})$ (compare to your answer in (b). What is the minimum number of time steps to solve this equation starting with $r_0 = 0.01$? And for $r_0 = 0.0001$?

f. Solve the ODE using `ode45` and $r_0 = 0.01$ and request a relative error of 10^4. Check the help page on how to use `odeset` to pass options to ODE solvers. With no output arguments, ode45 automatically plots the solution as it is computed. How many steps for the integration?

g. Next use $r_0 = 0.0001$. This will take much longer to compute. How many steps for the integration? Zoom in on the steady state value. What is it deviation? Do you expect this?

h. Try several of the stiff solvers. How many steps are needed now?

Exercise 40

For this exercise we immerse a one-dimensional spring, with a point mass m attached to it, into a vessel containing a viscous liquid, e.g., syrup. We pull the spring away from equilibrium over a distance $y = 1$. At the point $t = 0$ we let the spring go, so at this point in time the velocity of m is zero: $\mathrm{d}y(0)/\mathrm{d}t = 0$. The spring will start to vibrate following Newton's equation of motion: $m\mathrm{d}^2y/\mathrm{d}t^2 = F$. The forces acting on the mass are Hooke's law: $-ky$ and the friction force: $-f\mathrm{d}y/\mathrm{d}t$ (proportional to the velocity). In total, this so-called "damped harmonic oscillator" satisfies the equation of motion[1]

$$m\frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -ky - f\frac{\mathrm{d}y}{\mathrm{d}t}$$

or

$$\frac{\mathrm{d}^2y}{\mathrm{d}t^2} + \omega^2 y + 2b\frac{\mathrm{d}y}{\mathrm{d}t} = 0, \tag{7.23}$$

where $\omega = \sqrt{k/m}$ and $b = f/2m$.

a. Write a function for solving Eq. (7.23), which has, besides t and y, also b and ω as parameters.

[1]The reader may wonder why somebody in his right mind would put a spring into a bowl of syrup. However, the damped harmonic oscillator is a useful model in many branches of science: LCR electric circuits, dispersion of light, etc.

b. Solve this equation by means of `ode45` with $\omega = 1$ and $b = 0.2$. Integrate over the time span $[0, 20]$. Take as initial conditions $y(0) = 1$ and $\mathrm{d}y(0)/\mathrm{d}t = 0$.

c. The analytic solution of Eq. (7.23) for $b < \omega$ is, with $\Omega \equiv \omega^2 - b^2$,

$$y = y(0)e^{-bt}\left(\cos\Omega t + \frac{b}{\Omega}\sin\Omega t\right). \tag{7.24}$$

Write a script that (i) numerically integrates Eq. (7.23), (ii) implements the analytic solution, Eq. (7.24), and (iii) shows both solutions in one plot.

8. Statistical analysis

In the course ‘Statistiek voor moleculaire wetenschappers’ you have learned a number of techniques to analyze large data sets. In the exercises you have done the calculations by hand. MATLAB provides excellent tools to do this analysis in an automated fashion by the computer.

8.1 Mean, variance and standard deviation

Remember the mean

$$\overline{x} = \frac{1}{N}\sum_{i}^{N} x_i, \tag{8.1}$$

the variance

$$Var(x) = \frac{1}{N-1}\sum_{i}^{N}(x_i - \overline{x})^2, \tag{8.2}$$

and the standard deviation

$$\sigma = \sqrt{Var(x)}. \tag{8.3}$$

These can be easily computed using the functions `mean`, `var`, and `std`, respectively. These functions calculate each property summing over all elements of the input vector. If a matrix is given as argument, the properties are determined per column and a row vector is given as output. **Important:** Do not use variable names that have the same name as a function, *e.g.*, `mean`,`std`, since this will cause problems.

Exercise 41
You will now apply these techniques on a set of real data. These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. In Piemonte one has analyzed three different wines with different techniques to compile a total of 13 properties to characterize the wine. The dataset contains 54 samples from a Barolo wine, 71 from a Gignolino wine and 48 from a Barbera wine (in that order). You can find the dataset on the website `wine.dat`. Download this file to your own MATLAB work directory and view it in the MATLAB editor. As you can see, the data is organized in 14 columns. The first column indicates to which of the three different classes it belongs; the others are the different characteristics as indicated in comment header. Write a script that imports the data file and then extracts the data into three separate matrices: a matrix `barolo` of size 54×13 etc. **Hint:** Use the command `find` on the first column.

Exercise 42
Make a script to calculate the average and standard deviation of the three sets of mea-

surements. Compare this with the results from the Statistics course. On the basis of this information choose two properties that appear to be different for the three different sets.

Exercise 43
Create a scatter plot between the two properties selected in the previous exercise. Use for each of the different wine classes a different color and symbol. Label the axes and add a title to the plot.

8.2 Normal distribution

For many different statistical analysis techniques, one assumes that the data follows a normal or Gaussian distribution. In this case the data can be described the following probability density function:

$$\phi(x,\mu,\sigma) = \frac{\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}{\sqrt{2\pi}\sigma} \tag{8.4}$$

where σ is the standard deviation and μ is the mean. You should realize that there can be different origins to the distribution in data values: the errors in the observations are normally distributed and/or the data itself can be have a distribution. If one would preform the same chemical analysis multiple times on one sample, one would obtain some variation because of the error. The errors are typically normally distributed. If one preforms the same chemical analysis on one time on multiple samples, one would obtain a distribution because of the error in the analysis, but more importantly because of the intrinsic differences between the samples. The latter distribution does not necessarily lead to a normal distribution. For example, variables that cannot assume negative values such as concentrations and variables that vary over orders of magnitude can be lognormally distributed, *i.e.*, the logarithm of the variable is normally distributed.

In MATLAB it is fairly easy to visual inspect whether experimental data follows a normal distribution by calling the function `ydist = normpdf(x,mu,sigma);` and then plotting this together with a histogram of the data. `normpdf` returns a vector holding $\phi(x,\mu,\sigma)$ for each entry of `x`. This vector has a maximum of one. Before plotting, we need this such that the integrated value is the same as the number of samples contained in the histogram.

Exercise 44
It appears that the total concentration of flavonoids is a relatively good in discriminating between the different classes. We will continue to visualize the data according to this characteristic (flavonoids). We would like to make a figure that shows three histograms for the three different classes using the same bins. Check the help on the functions `bar`, `hist`, and `histc` to see how to do this. We like 60 bins between 0 and 6.

Table 8.1: Typical ANOVA table

Source of variance	Sum of squares	Degrees of freedom	Mean squares	F
Class 1	SS_1	$m_1 - 1$	$SS_1/(m_1 - 1)$	
Class 2	SS_2	$m_2 - 1$	$SS_2/(m_1 - 1)$	
...				
Class n	SS_n	$m_n - 1$	$SS_n/(m_n - 1)$	
Total within	$\sum_i^n SS_i$	$N - n$	$MS_{\text{within}} = \sum_i^n SS_i/(N - n)$	
Between	SS_{between}	$n - 1$	$MS_{\text{between}} = SS_{\text{between}}/(n - 1)$	
Total	SS_{total}	$N - 1$		$\frac{MS_{\text{between}}}{MS_{\text{within}}}$

Exercise 45
Create three normal distributions overplot them on the histograms. Use the command `hold on` and `hold off`.

8.3 Analysis of variance: ANOVA

Analysis of variance (ANOVA) allows one to test the hypothesis that the mean within classes is same as the overall mean of the sample. If we have n different classes and a total of N different measurements. Let m_i be the number of measurements within class i, then

$$N = \sum_i^n m_i. \tag{8.5}$$

For each class we can determine the mean μ_i and the sum of squares *within* a class

$$SS_i = \sum_j^{m_i} (x_{i,j} - \mu_i)^2. \tag{8.6}$$

The sum of squares *between* classes is

$$SS_{\text{between}} = \sum_i^n m_i(\mu_i - \mu)^2 \tag{8.7}$$

with μ the mean of total sample including all classes. SS_{total} is the sum of squares over all measurements with respect to the total mean μ and this is equal to

$$SS_{\text{total}} = SS_{\text{between}} + \sum_i^n SS_i = \sum_i^n \sum_j^{m_i} (x_{i,j} - \mu)^2. \tag{8.8}$$

A typical ANOVA table is given in Table 8.1. The F-value tells us how far we are removed from the hypothesis, *i.e.*, the classes all have the same mean as the total mean. The probability that the hypothesis is true can be determined by `fcdf(1/F,m_n-1,n-1)`.

Exercise 46

Write a function `anovatable` that displays the table above and returns the probability that the hypothesis is true. Input arguments are a vector with the measurements (for instance a column from the wine data) and a vector which indicates to which group each data point belongs (column one from the wine data). Use `sprintf` to make your output table look nice.

Use your function to determine which type of chemical analysis gives a good discrimination between the different wines.

9. Linear equations and least squares

Linear algebra is extremely useful in science and engineering. In this chapter we first discuss the problem of solving n linear equations with n unknowns and discuss some basic applications: finding the intersection of two lines in two-dimensional space and finding the intersection of a line and a plane in three dimensions. Next, we show how to smoothly interpolate measured data—seemingly a very different problem, but mathematically it is essentially same.

In some linear problems there are more equations than unknowns and strictly speaking there is no solution. However, one can still define a "best possible" solution. This is the topic of the section on the *linear least squares* method. We will show how this method can be used to find a fit that best represents a data set.

9.1 Linear equations

Consider a set of n linear equations with n unknowns $x_1, x_2, \ldots, x_n$,

$$\begin{array}{ccccccccc} A_{11}\,x_1 & + & A_{12}\,x_2 & + & \ldots & + & A_{1n}\,x_n & = & b_1 \\ A_{21}\,x_1 & + & A_{22}\,x_2 & + & \ldots & + & A_{2n}\,x_n & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ A_{n1}\,x_1 & + & A_{n2}\,x_2 & + & \ldots & + & A_{nn}\,x_n & = & b_n. \end{array} \tag{9.1}$$

These equations may be written more compactly as

$$\sum_{j=1}^{n} A_{ij}x_j = b_i, \quad i = 1, 2, \ldots, n. \tag{9.2}$$

We will also use matrix-vector notation

$$\begin{pmatrix} A_{11} & \ldots & A_{1n} \\ \vdots & & \vdots \\ A_{n1} & \ldots & A_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \tag{9.3}$$

and write the set of equations as

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}. \tag{9.4}$$

In MATLAB the solution of this equation is found with the backslash operator (\), *e.g.*,

```
>> rng(41)              % With this command, you will get the exact same
                        % "random" numbers as in this example.
>> n = 2                % Example in two dimensional space.
>> A = rand(n, n)       % A n-by-n matrix of random numbers.
A =
    0.2509    0.6768
    0.0461    0.0435
```

```
>> b = rand(n, 1)       % A column vector of n random numbers.
b =
    0.1164
    0.6039

>> x = A\b              % Find the solution of A * x = b.
x =
    19.8929
    -7.2031
>> r = A * x - b        % Calculate the residual.
r =
     0.1110e-15         % Not exactly zero, but close enough.
     0.1110e-15
```

In MATLAB the slash (/) as well as the (\) operators are defined:

$$\begin{aligned} 5/8 &\equiv 5 \times 8^{-1} = 0.625 \\ 5\backslash 8 &\equiv 5^{-1} \times 8 = 1.6 \end{aligned} \tag{9.5}$$

Mathematically, the solution of Eq. (9.4) can be written as

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}, \tag{9.6}$$

where $\boldsymbol{A}^{-1}$ is the *inverse* of the matrix $\boldsymbol{A}$. In MATLAB the inverse of the matrix `A` is computed by `inv(A)`, so we could also solve the problem this way:

```
>> x = inv(A) * b
x =
    19.8929        % Not necessarily all digits will be the same as above.
    -7.2031        % Use "format long" to see more digits.
```

To solve linear equations it is better to use "`A\b`" instead of "`inv(A) * b`" for several reasons: (i) it is faster, (ii) it is more accurate, and (iii) it requires less memory. To illustrate the first point we can use the MATLAB functions `tic()` (start timing) and `toc()` (end timing) to determine the computing time required for an operation. Point (ii), the accuracy of the solution, is tested by calculating the *residual*

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}. \tag{9.7}$$

The norm of the residual, $|\boldsymbol{r}|$, is zero for the exact solution. In this script we check (i) and (ii):

```
>> rng(41);                      % This will make the random numbers reproducible.
>> n = 2000                      % We need a larger matrix for meaningful timings.
>> A = rand(n, n);               % Do not forget the ";", otherwise you have to
                                 % wait for the matrix to be shown on the screen.
>> b = rand(n, 1);               % This will be the right-hand-side of the equation.
>> tic(); x1 = A\b; t1 = toc()   % Start timer, solve equations, stop timer.
t1 =
     0.1687                      % The cpu time in seconds.
```

```
>> tic(); x2 = inv(A) * b; t2 = toc()
t2 =
     0.3531                    % Quite a bit slower.
>> r1 = norm(A * x1 - b)
r1 =
     5.6160e-11                % The norm of residual vector.
>> r2 = norm(A * x2 - b)
r2 =
     2.2909e-10                % A little bit larger than r1.
```

Mathematically, a set of linear equations has either zero, one, or an infinite number of solutions. In a calculation with finite precision, as in MATLAB, the situation is a little different. MATLAB will always give a single answer, but in some cases the answer will be more accurate than in others. Sometimes MATLAB will give a warning of potential problems. To understand this we need a little more theory:

It is often helpful to think of a matrix-vector product as a linear combination of the columns of the matrix:

$$\boldsymbol{Ax} = [\boldsymbol{a}_1\boldsymbol{a}_2\ldots\boldsymbol{a}_n]\,\boldsymbol{x} = \boldsymbol{a}_1x_1 + \boldsymbol{a}_2x_2 + \ldots + \boldsymbol{a}_nx_n, \tag{9.8}$$

where the components of the column vectors $\boldsymbol{a}_j$ are given by $(\boldsymbol{a}_j)_i = A_{ij}$. The set of all vectors that can be written as $\boldsymbol{Ax}$ is called the *range* (in Dutch: *bereik*) of the matrix. This set can also be written as $\text{span}\{\boldsymbol{a}_1,\ldots,\boldsymbol{a}_n\}$ (in Dutch: *het opspansel*). When the n columns of $\boldsymbol{A}$ are linearly independent, the matrix $\boldsymbol{A}$ is called *regular*, its determinant is nonzero, and the set of linear equations has a unique solution. Otherwise, the matrix is called singular, the determinant of $\boldsymbol{A}$ is zero, the columns of $\boldsymbol{A}$ are linearly dependent, and the set of linear equations has either no solution, or an infinite number of solutions. The determinant of matrix $\boldsymbol{A}$ may be calculated in MATLAB by `det(A)`.

Mathematically a matrix is either regular or singular, but numerically, when doing arithmetic with finite precision, a matrix can be "almost singular", which means that the numerical solution of the linear equations will not be accurate as a result of round-off errors in the calculation. In $\mathbb{R}^2$ this happens when the two columns of the matrix are vectors that are almost parallel. Such a matrix is called *ill conditioned.* The condition of a matrix is quantified by the so called *condition number.* In MATLAB the condition of a matrix $\boldsymbol{A}$ is given by calling the function `cond(A)`. For the identity matrix or an orthonormal matrix (i.e., a matrix where the columns are orthogonal and have length one) the condition number is one and the numerical solution of the linear equations should be very accurate. When the condition number is larger than about 10^{16} (one divided by the machine precision `eps = 2.22e-16`) the numerical solution is almost meaningless and MATLAB will give a warning, e.g.,

```
>> format long
>> A = [2 1; 2 1+4*eps]              % eps=2.2204e-16
A =
   2.000000000000000   1.000000000000000
   2.000000000000000   1.000000000000001
```

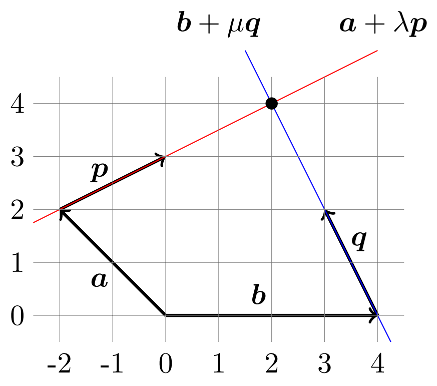

Figure 9.1: The lines $a + \lambda p$ and $b + \mu q$ intersect in the point $(2, 4)$, where $\lambda = 2$ and $\mu = 2$.

```
inv(A)
Warning: Matrix is close to singular or badly scaled. Results may be
inaccurate. RCOND =  1.480297e-16.
ans =
   1.0e+15 *
   0.562949953421312  -0.562949953421312
  -1.125899906842624   1.125899906842624
```

9.1.1 The intersection of two lines in a plane

In two dimensions a point is represented by a vector with two components,

$$\boldsymbol{r} = \begin{pmatrix} x \\ y \end{pmatrix}. \tag{9.9}$$

A line may be represented as a set of points $\boldsymbol{r}$ for which the two components satisfy the relation

$$y = \lambda x + \mu, \quad \lambda \in \mathbb{R}, \tag{9.10}$$

where λ is the slope and μ the offset. We will use an alternative parameterization of a line:

$$\boldsymbol{r} = \boldsymbol{a} + \lambda \boldsymbol{p}, \quad \lambda \in \mathbb{R}. \tag{9.11}$$

Clearly $\boldsymbol{a}$ is a point on the line (for $\lambda = 0$), and $\boldsymbol{p}$ gives the direction of the line, see Figure 9.1. This parameterization has two advantages: (i) a vertical line (e.g., the line with $x = 0$) can also be represented and (ii) the formula can be used in three- (or higher-) dimensional space, simply by taking the vectors $\boldsymbol{a}$ and $\boldsymbol{p}$ as three- (or higher-) dimensional vectors.

Figure 9.1 shows a second line which is defined by

$$\boldsymbol{x} = \boldsymbol{b} + \mu \boldsymbol{q}, \quad \text{with } \mu \in \mathbb{R}. \tag{9.12}$$

The support vectors $\boldsymbol{a}$, $\boldsymbol{b}$, and the directions $\boldsymbol{p}$, and $\boldsymbol{q}$ are given by

$$\boldsymbol{a} = \begin{pmatrix} -2 \\ 2 \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 4 \\ 0 \end{pmatrix}, \quad \boldsymbol{p} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \quad \boldsymbol{q} = \begin{pmatrix} -1 \\ 2 \end{pmatrix}.$$

The intersection is found by solving

$$\boldsymbol{a} + \lambda \boldsymbol{p} = \boldsymbol{b} + \mu \boldsymbol{q}, \tag{9.13}$$

or

$$\boldsymbol{p}\,\lambda - \boldsymbol{q}\,\mu = \boldsymbol{b} - \boldsymbol{a} \tag{9.14}$$

which may be rewritten as

$$\boldsymbol{A}\boldsymbol{c} = \boldsymbol{d}, \quad \text{with } \boldsymbol{A} = [\boldsymbol{p}\ -\boldsymbol{q}] \text{ and } \boldsymbol{c} = \begin{pmatrix} \lambda \\ \mu \end{pmatrix} \text{ and } \boldsymbol{d} = \boldsymbol{b} - \boldsymbol{a}. \tag{9.15}$$

Hence, in Matlab we can find $\boldsymbol{c}$ and the intersection with:

```
>> a = [-2; 2], b = [4; 0], p = [2; 1], q=[-1; 2]
>> A = [p -q]                         % The directions.
A =
     2     1
     1    -2
>> d = b-a                            % Difference of support vectors.
d =
     6
    -2
>> c = A\d                            % Solve A*c=d.
c =
     2
     2
>> [a+p*c(1) b+q*c(2)]                % The intersection expressed in two ways.
ans =
      2     2
      4     4
```

Exercise 47

A two-dimensional plane in three-dimensional space is parameterized by

$$\boldsymbol{x} = \boldsymbol{a} + c_1\boldsymbol{r}_1 + c_2\boldsymbol{r}_2, \quad \text{with } c_1, c_2 \in \mathbb{R}$$

and a line is given by

$$\boldsymbol{x} = \boldsymbol{p} + c_3\boldsymbol{q}, \quad \text{with } c_3 \in \mathbb{R}.$$

Find the intersection of the plane and the line for

$$\boldsymbol{a} = \begin{pmatrix} 3 \\ 7 \\ -2 \end{pmatrix}, \quad \boldsymbol{r}_1 = \begin{pmatrix} 5 \\ -1 \\ 3 \end{pmatrix}, \quad \boldsymbol{r}_2 = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, \quad \boldsymbol{p} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, \quad \boldsymbol{q} = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}.$$

Exercise 48

Repeat the previous exercise, but use instead

$$\boldsymbol{q} = 2\boldsymbol{r}_1 - \boldsymbol{r}_2 = \begin{pmatrix} 9 \\ -4 \\ 5 \end{pmatrix}.$$

Matlab should now print a warning, why?

Table 9.1: The pressures p_i of gas measured at four temperatures T_i.

i	T_i (°C)	p_i (bar)
1	0	1.0
2	20	1.1
3	40	1.3
4	80	2.0

9.1.2 Interpolation of data

Guillaume Amontons measured the pressures (p_i) of a gas at four different temperatures (T_i, $i = 1, 2, 3, 4$), the results are shown in Table 9.1. We will try to predict the pressures at other temperatures by *interpolation* of the data. This means that we have to make an assumption about the function that relates the pressure to the temperature. Let us assume that the relation is given by a polynomial:

$$p(T) = x_1 + x_2 T + x_3 T^2 + x_4 T^3. \tag{9.16}$$

In this third degree polynomial, we have have four unknown coefficients $x_i, i = 1, 2, 3, 4$. These coefficients are called *linear parameters.* We take a function with four linear parameters, since we have four data points. We can find the parameters by requiring the polynomial to reproduce the measured data:

$$p(T_i) = p_i, \quad \text{for } i = 1, \ldots, 4 \tag{9.17}$$

This gives four linear equations:

$$x_1 + T_i\, x_2 + T_i^2\, x_3 + T_i^3\, x_4 = p_i, \quad \text{for } i = 1, \ldots, 4 \tag{9.18}$$

We write the the linear parameters on the right, since it makes it easier to introduce matrix-vector notation by comparing to Eq. (9.2). Defining

$$A_{i,j} \equiv T_i^{j-1}, \quad \text{for } i, j = 1, 2, 3, 4 \tag{9.19}$$

and introducing the column vectors $\boldsymbol{x}$ and $\boldsymbol{p}$ we get

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{p}. \tag{9.20}$$

Exercise 49

Implement the matrix $\boldsymbol{A}$ (Eq. 9.19) using the data in Table 9.1 and find the linear parameters $\boldsymbol{x}$ by solving Eq. (9.20). Use the result and Eq. (9.16) to predict the pressures for $T = 0, 1, \ldots, 100$ °C. Make of plot with the predicted $p(T)$ for $T = 0, 1, \ldots, 100$, together with the original data points (T_i, p_i).

9.1.3 General linear interpolation

In the previous section we used a polynomial of degree $n-1$ to interpolate n data points. Suppose, however, that we measured the sea-level (L_i) at three different times ($t_i, i = 1, 2, 3$) on a day, and we want to represent the tidal motion $L(t)$ by interpolation. It would make more sense to try the functional form

$$L(t) = x_1 + \sin(\omega t)x_2 + \cos(\omega t)x_3, \tag{9.21}$$

where, in a much simplified model of the origin of tidal motion, $\omega = 4\pi/\text{day}$ (assuming t is expressed in days). With this value for ω, we can find the three linear parameters x_i from solving three equations with three unknowns.

In general, if we have n data points $(t_i, y_i), i = 1, 2, \ldots, n$, we can interpolate them linearly by choosing n functions $f_j(t), j = 1, \ldots, n$, and solving

$$y_i = \sum_{j=1}^{n} f_j(t_i)\, x_j, \quad i = 1, \ldots, n \tag{9.22}$$

To do this in MATLAB we define the $n \times n$ matrix $\boldsymbol{A}$ with elements

$$A_{ij} = f_j(t_i) \tag{9.23}$$

and the linear parameters $\boldsymbol{x}$ are found again with `x=A\y`. Note that the *columns* of the matrix $\boldsymbol{A}$ correspond to the functions and the rows correspond the linear parameters.

Thus, exercise 49 corresponds to choosing the four functions

$$f_j(T) = T^{j-1}, \quad j = 1, 2, 3, 4 \tag{9.24}$$

and for the tidal-motion example we would take

$$\begin{aligned} f_1(t) &= 1 && (9.25)\\ f_2(t) &= \sin(\omega t) && (9.26)\\ f_3(t) &= \cos(\omega t). && (9.27) \end{aligned}$$

Note that if we had measured the sea-level four times instead of three, we could try to determine ω, rather than assume a value. However, ω is a *nonlinear parameter*, and we will come back to that problem later.

In a MATLAB script to interpolate data, we need the set of functions $f_j(t), j = 1, \ldots, n$ twice: first to construct the matrix $\boldsymbol{A}$, and later when we want to evaluate the interpolating function at other values of t, e.g., to plot the result. Therefore, it is advantageous to write a MATLAB function to evaluate the set of functions $f_j(\boldsymbol{t})$, where $\boldsymbol{t}$ is a vector and return the result as a matrix. For instance, for the tidal-motion problem we could implement:

```
function A =  ftidal(t, omega)
% FTIDAL Evaluate the matrix A with elements A(i,j) = f_j(t(i), omega), for
%   f_1(t, omega) = 1
%   f_2(t, omega) = sin(omega*t)
%   f_t(t, omega) = cos(omega*t)
%   A = ftidal(t, omega)
%
% GCG November 2015
  t  = t(:);                           % Make sure t is a column vector
  A  = [ones(size(t)) sin(omega*t) cos(omega*t)];
```

Note that we also passed the *nonlinear* parameter ω to the function. We know that our ω value is approximate, so we do not define it inside the function. If we later have more data and try to find a better value for ω, we will not have to modify the function. With this function the script to interpolate the data becomes simple:

```
t = [0.3; 0.4; 0.5];               % We measured the sea level at
                                   % 7:12am, 9:36am, and at noon (1 day=24hour).
L = [0; 1.2; 0.15];                % Sea level in meters.
plot(t, L, 'rx')                   % Plot the data points.
xlabel('time (days)')              % x-axis label, with unit.
ylabel('Sea level (m)')            % y-axis label, with unit.
hold on                            % So the plot command below will not erase the plot.

omega = 4*pi                       % Our crude value for omega.
A     = ftidal(t, omega);          % Set up the matrix A.
x     = A\L;                       % Solve for linear parameters x.
tn    = (0:24)'/24;                % Every hour in unit of days (column vector)
Ln    = ftidal(tn, omega)*x;       % Predicted sea-levels
plot(tn, Ln)
```

Since we evaluate the sea-level for times before our first and after our last measurement, we not only *interpolated* the measurements, we also *extrapolated* the data.

Exercise 50

Rewrite the script of exercise 49 by defining and using a MATLAB function to compute the set of functions $\{T^0, T, \dots, T^{n-1}\}$ [see Eq. (9.24)]. In exercise 49 we have $n = 4$, but make sure to pass the value n to the function, so it can be defined in the main script.

9.2 Linear least squares

For an ideal gas, with the temperature expressed in Kelvin the relation between pressure and temperature is $p = \frac{nR}{V}T$ (with n the number of moles, R the gas constant and V the volume). The Van der Waals equation, which describes a real gas, is more complicated (check, e.g., Wikipedia), but the relation between p and T is still linear

$$p(T) = x_1 + x_2 T, \tag{9.28}$$

although the "constants" x_1 and x_2 depend on the size of the molecules and the density of the gas. To test the Van der Waals equation one could do three measurements and interpolate the result with a second degree polynomial

$$p(T) = x_1 + x_2 T + x_3 T^2 \tag{9.29}$$

and check whether x_3 is indeed zero. However, measurements are never exact, so instead we may want to do, e.g., $n = 101$ measurements of the pressure for $T = 0, 1, \ldots, 100$ °C and find the second degree polynomial that best fits the data. The difference between the fit and the measurement for temperature T_i, which is called the *residual* $r_i, i = 1, \ldots, n$ is given by

$$r_i = p_i - p(T_i) = p_i - \sum_{j=1}^{3} T_i^{j-1} x_j. \tag{9.30}$$

Since there are now only three linear parameters, (x_1, x_2, x_3), and 101 data points (T_i, p_i), we cannot expect a perfect fit. Instead we will try to minimize the so called *root-mean-square* error R

$$R = \sqrt{\sum_{i=1}^{n} |r_i|^2}. \tag{9.31}$$

Defining the column vector $\boldsymbol{r}$ with components r_i we see that $R = |\boldsymbol{r}|$, is the length (or *norm*) of the residual vector. Introducing matrix-vector notation as before we have

$$\boldsymbol{r} = \boldsymbol{p} - \boldsymbol{A}\boldsymbol{x}, \tag{9.32}$$

where $\boldsymbol{A}$ is again given by Eqs. (9.23) and (9.24), except that it now has 101 rows, and only 3 columns. Finding the linear parameters $\boldsymbol{x}$ for which the norm of the residual vector $R = |\boldsymbol{r}|$ is minimal is called a *linear least squares* problem. Although this problem is fundamentally different from solving n linear equations with n unknowns, in MATLAB the solution is also found with the backslash operator:

```
>> x = A\p
```

Exercise 51

Let us assume that for a give setup the exact relation between pressure (in bar) and temperature (in celcius) is given by

$$p = c_0 + c_1(T - T_0) + c_2(T - T_0)^2, \tag{9.33}$$

with $T_0 = -273$ °C, $c_0 = 4.2$ bar, $c_1 = 0.1$ bar/celcius, and, just for fun, a small quadratic contribution with $c_2 = 0.01$ bar/celcius^2. In MATLAB we can simulate a data set, including random noise in this way:

```
>> T     = (0:100)';             % A 101 temperatures from 0 to 100 celcius.
>> n     = size(T, 1);           % n = 101.
>> T0    = -273;                 % Approximately the absolute zero in celcius.
>> c0    = 4.2;                  % Just made these up.
>> c1    = 0.1;
>> c2    = 1e-2;
>> noise = 40*(rand(n,1)-0.5);  % The noise.
>> p     = c0 + c1*(T-T0) + c2*(T-T0).^2 + noise; % The simulated pressures.
```

Write a MATLAB script to determine the linear parameters x_1, x_2, x_3 in Eq. (9.29) that best fits the simulated data. Plot the data and the fit. To test your script, set the noise to zero and work out what $\boldsymbol{x}$ should be for the given values of c_0, c_1, c_2, and T_0.

9.3 Weighted linear least squares fit

In the last exercise we assumed that the noise was independent of the temperature. Let us now assume that the noise in the experiment is larger for lower temperatures:

```
>> noise = 4e11*(rand(n,1)-0.5)./(T-T0).^4;
```

Here we took the fourth power of the inverse temperature in kelvin, just to make the effect large. When we try to find the linear parameters x_1, x_2, and x_3 we may now take into account that the experiments at higher temperatures are "better" then the experiments at lower temperatures. To do this, we introduce a weighted residuals r'_i

$$r'_i = w_i[p_i - p(T_i)], \tag{9.34}$$

with the weights

$$w_i = (T_i - T_0)^4 \tag{9.35}$$

and we now try to minimize $|\boldsymbol{r}'|$ instead of $|\boldsymbol{r}|$. The new residual is related to the old one through

$$\boldsymbol{r}' = \boldsymbol{W}\boldsymbol{r} = \boldsymbol{W}\boldsymbol{p} - (\boldsymbol{W}\boldsymbol{A})\boldsymbol{x}, \tag{9.36}$$

where $\boldsymbol{W}$ is a diagonal matrix with elements

$$W_{ii} = w_i, \quad \text{(and } W_{i,j} = 0 \text{ for } i \neq j). \tag{9.37}$$

If we compare Eq. (9.36) with Eq. (9.32) we note that the problem is very similar, except that $\boldsymbol{p}$ is replaced by $\boldsymbol{W}\boldsymbol{p}$ and the matrix $\boldsymbol{A}$ is replaced by $\boldsymbol{W}\boldsymbol{A}$. Hence, the weighted linear least squares problem is solved in MATLAB with

```
>> w = (T-T0).^4;          % Define the temperature dependent weights.
>> W = diag(w);            % diag(w) turns a vector into a diagonal matrix.
>> x = (W*A)\(W*p)         % Solve the weighted linear least squares problem.
```

Exercise 52
Take the previous exercise (51) and replace the random noise (line 7) by the temperature dependent noise defined in this section. Obtain a fit with the normal linear least squares method (using the script you wrote in exercise 51) and solve the problem again using the weighted linear least squares method with the weights given in Eq. (9.35).

How would you compare the two methods to determine whether the weighted linear least squares method is better or not? Implement your test.

Appendices

A. More plotting

A.1 Polar plots

Atomic orbitals pervade all of chemistry. Invariably they are drawn as 3D polar plots and therefore we will show how MATLAB can visualize 3D polar plots. Of course, orbitals may also be drawn as contour diagrams, which is commonly done for molecular orbitals.

Atomic orbitals are products of the form $f(r)\,g(\theta,\phi)$, where we meet again spherical polar coordinates defined by

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\sin\theta\cos\phi \\ r\sin\theta\sin\phi \\ r\cos\theta \end{pmatrix}, \quad 0 \le r < \infty,\ \ 0 \le \theta \le \pi,\ \ 0 \le \phi < 2\pi. \tag{A.1}$$

In polar plots one draws only the angular part $g(\theta,\phi)$, taking a fixed radial part $f(r_0)$.

MATLAB is able to make 2D polar plots, but not 3D polar plots, so we must do this ourselves. Let us first explain how to make a 2D polar plot by hand. Say, we want to plot the angular part of the function $f(r)g(\phi)$. Figure A.1 shows polar graph paper that helps us do this. We choose a fixed value r_0 and mark points for $\phi = 0°$, $20°$, $\cdots$, $340°$ with the value of $h(\phi_i) \equiv |f(r_0)g(\phi_i)|$ as the distance from the origin. The equidistant circles help us in measuring this distance. In other words, we mark points $(x_i,\, y_i) = (r\cos\phi_i, r\sin\phi_i)$, but substitute for r the positive function values $h(\phi_i)$. After marking the points we draw a line through the points.

As an example, we make a polar plot of $g(\phi) = \cos\phi$ $[f(r_0) = 1]$ and since the computer does the work, we take a much finer grid than we would do by hand.

```
phi = [0:1:360]*pi/180;
x   = cos(phi);
y   = sin(phi);
h   = abs(cos(phi));
plot( h.*x, h.*y)
axis equal
```

The generalization to 3D is clear now. For instance, if we want to plot the $2p_z$ atomic orbital $r\exp(-r)\cos\theta$, we choose a constant value r_0 for r, but since the factor $r_0\exp(-r_0)$ does not affect the shape of the plot, we ignore it. We write $g(\theta,\phi) = \cos\theta$, $h(\theta) = |g(\theta)|$ and plot $z = \cos\theta$ against $x = h(\phi)\sin\theta\cos\phi$ and $y = h(\phi)\sin\theta\sin\phi$. In MATLAB:

```
theta = [0:5:180]*pi/180;
phi   = [0:5:360]*pi/180;
[Theta, Phi] = meshgrid(theta, phi);
H     = abs(cos(Theta));
```

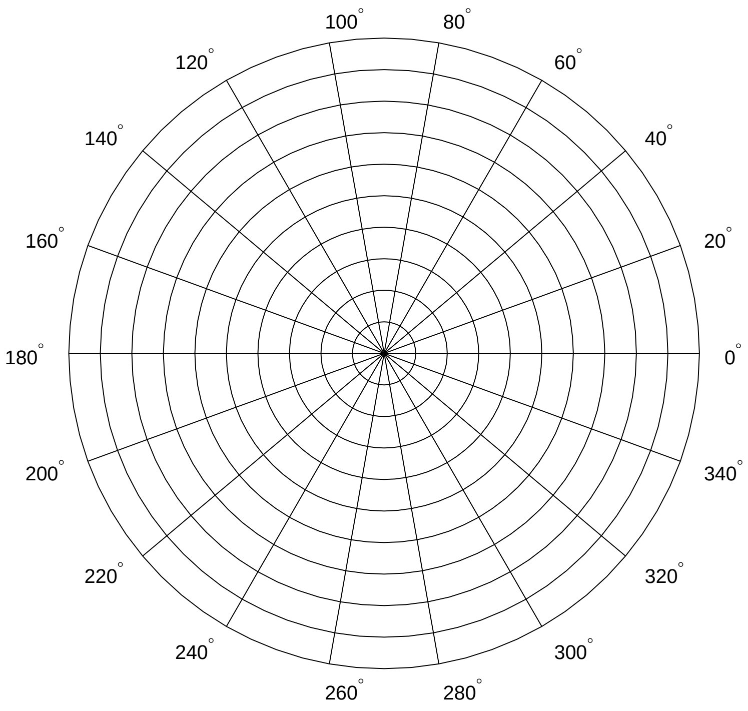

Figure A.1: Polar plot paper

```
X     = H.*sin(Theta).*cos(Phi);
Y     = H.*sin(Theta).*sin(Phi);
Z     = H.*cos(Theta);
surf(X,Y,Z)
axis equal off
```

The same can be achieved in a slightly more efficient manner

```
theta = [0:5:180]*pi/180;
phi   = [0:5:360]*pi/180;
h     = abs(cos(theta));
X     = cos(phi')*sin(theta)*diag(h);
Y     = sin(phi')*sin(theta)*diag(h);
Z     = repmat(cos(theta), length(phi),1)*diag(h);
surf(X,Y,Z)
axis equal off
```

Here `X` and `Y` are obtained from column vector times row vector multiplication (dyadic product).

A.2 Handle Graphics

You have probably noticed that you can change figures through the different menus in the figure window. We have not discussed these options, for mainly two reasons. First, these menus change with different versions of Matlab and so learning how they work in the current version might not prove very useful for future use. Secondly, we want to encourage you to make plot scripts where you can generate a complete figure just by calling one script. For instance, if you write your report and you want to add an extra data set to the graph, this can very easily be done by changing the script and all labels, titles and legends that you added in the past will be automatically created again. Generating different figures and can also be easily done by just copy-and-pasting parts of scripts.

MATLAB has a system, called Handle Graphics, by which you can directly manipulate graphics elements through the command line. This system offers unlimited possibilities to create and modify all types of graphs. The organization of a graph is hierarchical and object oriented. At the top is the Root, which simply is your MATLAB screen. This object is created when you start up MATLAB.

Figure objects are the individual windows on the Root screen, they are referred to as the *children* of the Root. There is no limit on the number of Figures. A Figure object is created automatically by the commands that we introduced earlier, namely `plot`, `mesh`, `contour` and `surf`. As for all graphical objects, there is also a separate low level command that creates the object: `figure` creates a new Figure as a child of Root. If there are multiple Figures within the Root, one Figure is always designated as the "current" figure; all subsequent graphics commands, such as `xlabel` will give output to this figure.

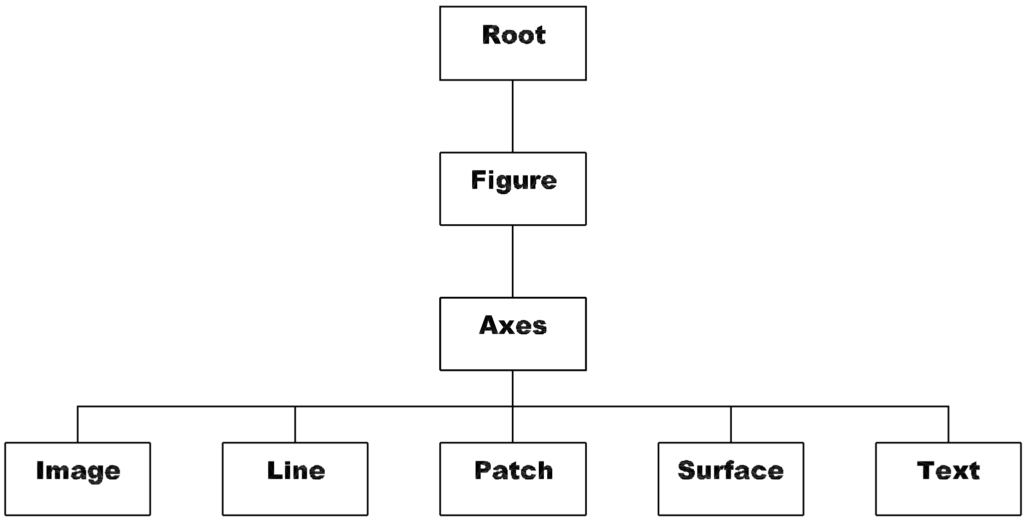

Figure A.2: *Graphics Hierarchy Tree*

A Figure object has as children the objects Axes, which define regions in a Figure window. All commands such as `plot` automatically create an Axes object if it does not exist. Only one Axes object can be current. The children of Axes are Image, Line, Patch, Surface, and Text. Image consists of pixels; see the MATLAB on-line documentation for more details. The Line object is the basic graphic primitive used for most 2D and 3D plots. High level functions such as `plot` and `contour` create these objects. The coordinate system of the parent Axes positions the Line object in the plot. Patch objects are filled polygons and are created by high level commands such as `bar`, which creates a bar graph. Surface objects represent 3D data and are created by `mesh` and `surf`.

The final objects are Text, which are also children of Axes. Text objects are character strings and are created by high level commands such as `xlabel`, `ylabel` and `title`, which we met earlier.

A.3 Handles of graphical objects

Every individual graphics object has a unique identifier, called a *handle*, that MATLAB assigns to it when the object is created. By using these handles as variables we can manipulate very easily all objects in a plot. As an example we return to `contour`. Thus far we plotted contours without numbers, which is not particularly useful. However, the MATLAB command `contour` can return values associated with each contour line. Moreover the command returns the *handle* of every contour line. Example:

```
>> x     = linspace(-1,1,20); % 20 equidistant points
>> y     = linspace(-2,2,40);
>> [X Y] = meshgrid(x,y);
>> Z     = Y.^2.*exp(-X.^2-Y.^2);
>> [c h] = contour(X,Y,Z,10);    % 10 levels
>> whos  c h
Name      Size                   Bytes  Class
  c         2x892                  14272  double array
  h        32x1                      256  double array
```

MATLAB interpolates the z values by means of an unspecified algorithm. Furthermore, the length of `c`, containing contour values, is unpredictable, as is the length of `h`. One would perhaps expect the vector of handles `h` to be of length 20 (10 levels are requested of a two-fold symmetric function). However, lines cut by borders become separate graphical objects and therefore `h` is longer than 20 (in the example 32). The important use of the arrays `c` and `h` is in the command `clabel`. This command draws the labels in the contour plot. Continuing the example, the command

```
>> clabel(c,h)
```

draws labels in the current figure (which must be drawn by `contour`). Doing this, one gets a crowded plot, the labels are often too close to each other. The command `clabel(c,h,'manual')` offers the possibility to pick the positions interactively.

As another example of the use of handles, we consider the command `plot`. When we issue `h=plot(x,Y)`, where `Y` is a matrix, then of course the plots appear, but in addition the vector `h` will appear on the screen. Its elements contain unique identification numbers of the curves: the line handles. Each curve in the figure corresponds to an element of the vector `h`. These line handles allow us to make modifications to the individual curves. (Obviously, `h` is only of length > 1 if we plot more than one curve with the same plot statement. If we plot two curves one after the other by issuing the plot command twice and the toggle `hold` in the `on` state, then we get twice a different scalar back as the line handle.)

As an example of the use of a line handle, we first plot two straight lines and then change the color of the second to red, which by default was drawn by MATLAB in faint green.

```
>> h = plot([1:10]', [[1:10]' [2:11]']) % no semicolon
h =
  103.0029   % these are the handles
    3.0063   % usually they are floating point numbers
>> set(h(2), 'color', 'red')  % second line red
```

The low level command `set` acts on the line with line handle `h(2)` and changes the color of this line to red. Note that `set` does not erase the plot and does not start a new one, it acts on the existing (current) plot. High level commands, such as `plot`, `contour`, etc., do start a new plot (unless the hold status is on).

Line handles can also be used to get information about the graphical object.

```
>> set(h(2)) % Inspect possible settings of second curve
        Color
        EraseMode: [ normal | background | xor | none ]
        LineStyle: [ - | -- | : | -. | none ]
        ....
>> get(h(2)) % Inspect actual settings of second curve
        Color = [1 0 0]
        EraseMode = normal
        LineStyle = -
        ....
```

The command `set(h(2))` returns the possible settings for the second curve, the command `get(h(2))` returns the actual choices. We see that "color" is a property of object Line, and that the actual choice of color is given by the array `[1 0 0]`.

In the "rgb" color scheme used here, this implies that the color chosen is pure red, as is expected because we changed the line color to red. (The array `[1 0 1]` would give an equal mixture of red and blue, which is magenta). We see the possible line styles (see help plot) and the actual choice: solid (the default).

Suppose now we want change the line style of the second curve to dotted. We see that `linestyle` (not case sensitive!) is a property of object with handle `h(2)`. We can change it with `set(h(2), 'linest', ':')`. This turns it into a dotted line. Note that unique abbreviations (`linest` instead of `linestyle`) of the property names of the objects are allowed.

As we said, all objects have a handle, also the Axes objects in a Figure object; by `gca` ("get current axes") we get the handle of the current Axes in the current Figure and by `get(gca)` we get the actual values of the properties of this Axes object. We see that one of the Axes properties is `FontSize`. If we want to increase to 20 points the size of the digits on the axes, we issue `set(gca, 'Fontsize', 20)`.

Fortunately, it is often not necessary to use these low level commands, MATLAB has several high level commands that make life easier. For instance the aspect ratio (ratio between x-axis and y-axis scale) can be set by a command of the type `set(gca,'DataAspectRatio',[1 1 1])`. However, much easier is the use of the high level function `axis`; see its help.

By left clicking with the mouse on an object we turn it into the "current object". Its handle is returned by the command `gco` ("get current object"). So, if we want to change the color of a line on the screen to blue and we forgot to save its handle, then we can click on it and issue the command `set(gco, 'col', 'b')`. Many

Remove an object by using the function `delete`, passing the object's handle as the argument. For example, delete the current Axes (and all of its children!) by `delete(gca)`.

We can draw lines in the coordinate system of the current Axes object by the low level command

```
line([x1 x2 .. xn], [y1 y2 .. yn])
```

This draws a line from point (x_1, y_1) to point (x_2, y_2), etc., to point (x_n, y_n). But first we want to give the plot a definite size by

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

which defines an x-axis from `xmin` to `xmax` and a y-axis from `ymin` to `ymax`. For example, the following code draws a rectangular box:

```
>> close all         % close all existing figures
>> axis([0 10 0 20])
>> h = line([2 5 5 2 2], [2 2 4 4 2])
```

We define first an x-axis from 0 to 10 and a y axis from 0 to 20. If we do not do this, the axes are generated by the line command and the boundaries of the box will coincide with the axes, which is usually not what you want. The line command draws lines from the successive points $(2, 2)$ to $(5, 2)$ to $(5, 4)$ to $(2, 4)$ back to $(2, 2)$. This is a rectangular box with width 3 and height 2 with the lower left hand corner at the point $(2, 2)$.

We can place text in the plot by the command `text(x,y,'text')`. For instance in the box that we just drew, beginning at point $(2.5, 3)$:

```
>> t = text(2.5, 3, ' Text ', 'fontsize', 20, ...
   'fontname', 'arial black')
```

Note that the fontsize has a different unit of size than the axes. Arial black is the name of a font; MATLAB can handle different fonts, enter `listfonts` to see which ones are available.

A.4 Exercises

Exercise 53

- Plot the *d*-orbital of the previous exercise in polar form. Here you may forget about the normalization constant, because only the shape matters, not the values.
- Plot $2p_x^2 = r^2 \exp(-2r) \sin^2\theta \cos^2\phi$ (unnormalized) in polar form. Do not forget the aspect ratio `axis equal`! Do you recognize this plot?

B. Cell arrays and structures

Other data structures than double arrays.

In large MATLAB programs, containing many variables and arrays, it is easy to lose track of the data. MATLAB offers two data structures that make it possible to tidy up such programs: cell arrays and structures. Collecting data in cell arrays or structures may be compared with the tidying up of large collections of disk files by storing them in tar- or zip-files. Related data are brought under a common denominator: the name of the cell array, structure, tar or zip file. Cell arrays and structures are particularly useful if parameter lists in function calls become unwieldily long, because a complete list can be generated by a single reference to a cell array or a named field of a structure.

Since structures and cell arrays are often used to store character data, we also give a short introduction to character handling in MATLAB.

Even if one does not have the intention to write extensive MATLAB programs, it pays to have some knowledge of cell arrays and structures, because MATLAB itself makes heavy use of it. To mention one example: MATLAB can handle variable length input argument lists. This allows any number of arguments to a function. The MATLAB variable `varargin` is a cell array containing the optional arguments to the function. Structures are used in setting parameters for function functions. For instance, the command `odeset` creates or alters the structure `options`, which contains parameters used in solving ordinary differential equations (ODEs). The command `optimset` creates a structure used in, among others, `fzero` and `fminsearch`.

B.1 Cell arrays

Cell arrays in MATLAB are (multidimensional) arrays whose elements are cells. A cell can be looked upon as a container with an array inside. (Remember that a number is also an array, namely a 1×1 array). Usually cell arrays are created by enclosing a miscellaneous collection of arrays in curly braces, {}. For example,

```
>> A = magic(5)       % creates magic square
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
>> C = A sum(A) prod(prod(A))    % creates cell array
C =
    [5x5 double]    [1x5 double]    [1.5511e+025]
```

These two commands produce (i) the magic square matrix `A` (all rows and columns sum to 65, elements are $1, 2, \ldots, 25$) and (ii) the 1×3 cell array `C`. The function `prod` takes the products of elements in the columns of `A` and returns a row vector. The second `prod` takes the product of the elements in this row vector. The three cells of cell array `C` contain now the following arrays, which are of different dimension:

- the square matrix `A` in `C(1,1)`,
- the row vector of column sums in `C(1,2)`,
- the product of all its elements in `C(1,3)`.

The contents of the cells in cell array `C` are not fully displayed because the first two cells are too large to print in this limited space, but the third cell contains only a single number, 25!, so there is room to print it.

Here are two important points to remember. First, to retrieve the *cell itself* (container plus content) use round brackets. Since a cell is nothing but a 1×1 cell array, MATLAB command `whos` tells us that e.g. `C(1,2)` retrieves a cell array. Second to retrieve the *content of the cell* use subscripts in curly braces. For example, `C{3}`, or equivalently `C{1,3}`, retrieves 25!, whereas `C(3)` retrieves the third cell. Notice the difference (`C` is the cell array of previous example):

```
>> c=C(1)       % Get cell (1-by-1 cell array)
c =
[5x5 double]

>> whos c
  Name      Size                   Bytes  Class
  c         1x1                      260  cell array

>> d=C1      % Get content of cell (5-by-5 magic square)
d =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> whos d
  Name      Size                   Bytes  Class
  d         5x5                      200  double array
```

The MATLAB documentation refers to this as 'cell indexing' (round brackets) and 'content indexing' (curly braces), respectively. *The curly braces peel off the walls of the container and return its contents and the round brackets return the container as a whole.*

Cell arrays contain copies of other arrays, not pointers to those arrays. If you subsequently change the array `A`, nothing happens to `C`.

As we just saw, cell arrays can be used to store a sequence of matrices of different sizes. As another example, we create first an 8×1 cell array with empty matrices and then we store magic squares of different dimensions,

```
% create cell array with a row of empty matrices
>> M = cell(8,1);
>> for n = 1:8
>>   Mn = magic(n); %content of cell n <--  magic square
>> end
>> M
M =
[ 1]
[ 2x2 double]
[ 3x3 double]
[ 4x4 double]
[ 5x5 double]
[ 6x6 double]
[ 7x7 double]
[ 8x8 double]
```

We see that M contains a sequence of magic squares. We retrieve the contents of a cell from this one-dimensional cell array by M{i}, for instance,

```
>> prod(prod(M3))  % Content of cell M(3) is 3-by-3 array
ans =
      362880

>> prod([1:9])   % 9!
ans =
      362880
```

Using round brackets, as in prod(M(3)), we get an error message because the function prod cannot take the product of a cell, only the contents of the cell can be multiplied.

As we saw above, a set of curly brackets around a set of one or more arrays converts the set into a cell array. So, in the example above we could as well have written M(n)={magic(n)} instead of M{n}=magic(n). The first form creates on the right hand side a cell containing a magic square and assigns this cell to the n^{th} cell of M. The second form adds the magic square to the content of the n^{th} cell of M

Only cells may be assigned to cells:

```
>> a    = cell(2);
>> a(1) = [1 2 3]     % Wrong, rhs is not a cell
??? Conversion to cell from double is not possible.

>> a(1) = [1 2 3]   % OK, rhs is a cell
a =
    [1x3 double]

>> b      = cell(2,2);
>> b(1,2) = a(1)   % cell a(1) assigned to cell b(1,2)

% Content of cell a(1) to content of cell b(1,2):
>> b1,2 = a1
```

When you use a range of cells in curly brackets, as in `A{i:j}`, the contents of the cells are listed one after the other separated by commas. In a situation where MATLAB expects such a *comma separated list* this is OK. For instance,

```
>> A1 = [1 2];
>> A2 = [3 4 5];
>> A3 = [6 7 8 9 10];
>> % Comma separated list between square brackets
>> [A1:3]
>> % identical to:
>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

To assign more than one element of a cell array to another, do not use curly brackets on the left hand side, because in general a comma separated list is not allowed on the left hand side of an assignment. Example:

```
>> clear a,  for i=1:20,  ai= i^2; end
>> clear A,  for i=1:20,  Ai=-i^2; end
% To assign part of cell array a to A, use cell indexing
>> a(5:10) = A(5:10); % Assignment of cells
>> a5:10            % Display part of cell array:
```

Explanation: the statement `a{5:10}=A{5:10}` is wrong because the left hand side would expand to the comma separated list `25, 36, 49, 64, 81, 100` and such a list on the left hand side of an assignment makes no sense to MATLAB. The next statement (`a{5:10}`) is OK, because it simply is equivalent to:

```
>> 25, 36, 49, 64, 81, 100
ans =
    25
ans =
    36
ans =
    49
ans =
    64
ans =
    81
ans =
   100
```

which asks MATLAB to display the six squared numbers. As we see in the example above, ranges of cell arrays can be assigned to each other by the use of cell indexing (round brackets). We could try the assignment `a(5:10)=A{5:10}`. This gives an error message for the following reason: on the left hand side we have a range of cells and on the right hand side we have numbers (the contents of the cells). We can only assign a cell to a cell, that is, on the right hand side we must also have cells. We can convert the right hand side to a cell array by enclosing it in curly brackets: `a(5:10)={A{5:10}}` is identical to `a(5:10)=A(5:10)`.

Cell arrays are useful in calling functions, where comma separated lists are expected as parameter lists. Example:

```
>> Param    = cell(1,4);     % Create cell array
>> Param(1) = 'linest';     % Assign first
>> Param(2) = ':';          %    and second cell
>> Param3 = 'color';        % Put content into third
>> Param4 = 'red';          %     and fourth cell
>> plot(sin(0:pi/10:2*pi), Param:)
```

Here `Param{:}` expands to a comma separated list, i.e., the four parameters are passed (separated by commas) to `plot`.

In summary, a cell array is a MATLAB array for which the elements are cells, containers that hold other MATLAB arrays. For example, one cell of a cell array might contain a real matrix, another an array of text strings, and another a vector of complex values. You can build cell arrays of any valid size or shape, including multidimensional structure arrays.

B.2 Strings

We have seen that strings are arrays of characters. Concatenation with square brackets joins text variables together into larger strings. The second statement joins the strings horizontally:

```
>> s = 'Hello';
>> h = [s, ' world']  % a 1-by-11 character array
h =
Hello world
```

The statement `v = [s; 'world']` joins the strings vertically. Note that both words in `v` must have the same length. The resulting array is a 2×5 character array.

To manipulate a body of text containing lines of different lengths, you can construct a character array padded with blanks by the use of `char`. We have just seen that `char` converts an array containing ASCII codes into a MATLAB character array. There are more uses of `char`. If we write `s = char(t1,t2,t3,..)` then a character array `s` is formed containing the text strings `t1,t2,t3,...` as rows. Automatically each string is padded with blanks at the end in order to form a valid matrix. The function `char` accepts any number of lines, adds blanks to each line to make them all the length of the longest line, and forms a character array with each line in a separate row. The reference to `char` in the following example produces a 6×9 character array

```
>> S = char('Raindrops', 'keep', 'falling', 'on', ...
            'my', 'head.')
S =
Raindrops
keep
```

```
falling
on
my
head.

>> whos S
  Name      Size         Bytes  Class
  S         6x9            108  char array
```

The function `char` adds enough blanks in each of the last five rows of S to make all the rows the same length, i.e., the length of `Raindrops`.

Alternatively, you can store text in a cell array. As an example we construct a column cell array,

```
>> C ='Raindrops'; 'keep'; 'falling'; 'on'; 'my'; 'head.';

>> whos C
Name      Size         Bytes  Class
C         6x1            610  cell array
```

The contents of the cells are arrays (in this example character arrays), in agreement with our definition of a cell array. You can convert a character array `S` to a cell array of strings with `C = cellstr(S)` and reverse the process with `S = char(C{:})` (comma separated list passed to `char`).

Generally speaking cell arrays are more flexible than character arrays. For instance, transposition of the 6×1 cell array `C` gives

```
>> C'   % cell array from the previous example
 ans =
   'Raindrops'   'keep'   'falling'   'on'   'my'   'head.'
```

which is a 1×6 cell array with the same strings as contents of the cells. Transposition of the 6×9 character array `S` on the other hand gives

```
>> S'   % character array
ans =
    Rkfomh
    aeanye
    iel  a
    npl  d
    d i  .
    r n
    o g
    p
    s
```

which is generally less useful (unless you are a composer of crosswords).

B.3 Structures

Structures are multidimensional MATLAB arrays with elements accessed by designators that are strings. For example, let us build a data base for a certain course. Each entry contains a student name, student number and grade,

```
>> S.name   = 'Hans Jansen';
>> S.number = '005143';
>> S.grade  = 7.5
```

creates a scalar structure with three fields:

```
S =
      name: 'Hans Jansen'
    number: '005143'
     grade: 7.5000
```

Like everything else in MATLAB, structures are arrays, so you can insert additional elements. In this case, each element of the array is a structure with several fields. The fields can be added one at a time,

```
>> S(2).name   = 'Sandy de Vries';
>> S(2).number = '995103';
>> S(2).grade  = 8;
```

The scalar structure `S` has now become a 1×2 array, with `S(1)` referring to student Hans Jansen and `S(2)` to Sandy de Vries.

An entire element can be added with a single statement.

```
>> S(3) = struct('name','Jerry Zwartwater',...
      'number','985099','grade',7)
```

The structure is large enough that only a summary is printed,

```
S =
1x3 struct array with fields:
    name
    number
    grade
```

There are several ways to reassemble the various fields into other MATLAB arrays. They are all based on the notion of a comma separated list. We have already met this notion in the discussion of the cell arrays. Typing

```
>> S.number
```

is the same as typing `S(1).number, S(2).number, S(3).number`. This is a comma separated list. It assigns the three student numbers, one at a time, to the default variable `ans` and displays these numbers. When you enclose the expression in square brackets, `[S.grade]` it is the same as `[S(1).grade, S(2).grade, S(3).grade]` which produces a row vector containing all of the grades. The statement

```
>> mean([S.grade])
ans =
    7.5000
```

gives the average grade of the three students now present in our data base. Just as in the case of comma separated lists obtained from cell arrays, we must be careful that we generate the lists only in contexts where MATLAB expects such lists. This is in six situations:

- Combination of statements on one line plus output, e.g. `>> a,b,c,d`
- inside `[ ]` for horizontal concatenation, e.g. `[a,b,c,d]`
- inside `{ }` to create a cell array, e.g. `{a,b,c,d}`
- inside `( )` for function input arguments, e.g. `test(a,b)`
- inside `( )` for array indexing, e.g. `A(k,l)`
- inside `[ ]` for multiple function output arguments, e.g. `[v,d] = eig(a)`.

Because of this expansion into a comma separated list, structures can be easily converted to cell arrays. Enclosing an expression in curly braces, as for instance `{S.name}`, creates a 1×3 cell array containing the three names.

```
>> S.name
ans =
    'Hans Jansen'    'Sandy de Vries'    'Jerry Zwartwater'
```

Since `S.name` expands to a list containing the contents of the name fields, this statement is completely equivalent to

```
ans = 'Hans Jansen', 'Sandy de Vries', 'Jerry Zwartwater'
```

which, as we saw above, creates a cell array with three cells having the three student names as their contents.

An expansion to a comma separated list is useful as a parameter list in a function call. Above we gave an example where we passed a parameter list to `plot` by expanding the cell array `Param`. As an example of expanding a structure, we call `char`, which, as we saw in subsection B.2, creates a character array with the entries padded with blanks, so that all rows are of equal length:

```
>> N=char(S.name) % expansion to comma separated list
N =
Hans Jansen
Sandy de Vries
Jerry Zwartwater

>> whos N
Name      Size         Bytes  Class
N         3x16            96  char array
```

In Table B.1 we list a few MATLAB functions that can handle text, cell arrays, and structures.

Table B.1: *Some functions useful for cell array, structure and character handling; see help files for details.*

cell	Create cell array.
cell2struct	Convert cell array into structure array.
celldisp	Display cell array contents.
cellfun	Apply a cell function to a cell array.
cellplot	Display graphical depiction of cell array.
char	Create character array (string).
deal	Deal inputs to outputs
deblank	Remove trailing blanks from a string
fieldnames	Get structure field names.
findstr	Find one string within another.
int2str	Convert integer to string.
iscell	True for cell array.
isfield	True if field is in structure array.
isstruct	True for structures.
num2cell	Convert numeric array into cell array.
num2str	Convert number to string.
rmfield	Remove structure field.
strcat	Concatenate strings
strcmp	Compare strings
strmatch	Find possible matches for a string
struct	Create or convert to structure array.
struct2cell	Convert structure array into cell array.

Cell arrays are useful for organizing data that consist of different sizes or kinds of data. Cell arrays are better than structures for applications when you don't have a fixed set of field names. Furthermore, retrieving data from a cell array can be done in one statement, whereas retrieving from different fields of a structure would take more than one statement. As an example of the latter assertion, assume that your data consist of:

- A 3×4 array with measured values (real numbers).
- A 15-character string containing the name of the student who performed the measurements.
- The date of the experiment, a 10-character string as '23-09-2003'.
- A $3 \times 4 \times 5$ array containing a record of measurements taken for the past 5 experiments.

A good data construct for these data could be a structure. But if you usually access only the first three fields, then a cell array might be more convenient for indexing purposes. To access the first three elements of the cell array `TEST` use the command `deal`. The statement

```
[newdata, name, date] = deal(TEST1:3)
```

retrieves the contents of the first three cells and assigns them to the array `newdata` and the strings `name` and `date`, respectively. The function `deal` is new. It is a general function that deals inputs to outputs:

```
[a,b,c,...] = deal(x,y,z,...)
```

simply matches up the input and output lists. It is the same as `a=x`, `b=y`, `c=z`,.... The only way to assign multiple values to a left hand side is by means of a function call. This is the reason of the existence of the function `deal`. On the other hand, to access different fields of the structure `test`, we need three statements:

```
newdata = test.measure
name    = test.name
date    = test.date
```

B.4 Exercises

Exercise 54

a. Type in

   ```
   t = 'oranges are grown in the tropics'
   ```

   Read the help of `findstr`. Parse this string into words by the aid of the output of `findstr`. That is, get six strings (character arrays) that contain the respective words. Store these strings in a cell array. Make sure that you do not have beginning or trailing blanks in the words.

b. Write a function `parse` that parses general strings.

   Hints:
   Check the input of the function by `ischar`. Make sure that the input string does not have trailing blanks by the use of `deblank`.

Exercise 55
Write a script that writes names and sizes of files in the current directory to the screen. Each line must show the filename followed by its size (in bytes). Show the files sorted with respect to size.
Hints:

a. The command `dir` returns a structure with the current filenames and sizes.

b. The command `sort` can return the sorted array together with the permutation that achieves the actual sorting.

c. MATLAB does not echo properly an array of the kind `[char num]`. Apply `int2str` to `num` to get readable output.

Exercise 56
Predict—without executing the following script—what it will put on your screen:

```
clear all;
a = magic(3);
b = 'a' 'b' 'c'         'd' 'e' 'f';
c(1,1).income = 22000;
c(2,4).age = 24;
d = rand(4,7);
e = a b ...         c d ;

u = size(e);
k = 0;
for i=1:u(1)
for j=1:u(2)
  k = k+1;
  siz(k,:) = [size(ei,j)];
end
end

siz = [siz; u]
clear u i j k siz

A = whos;
for i=1:length(A)
  siz(i,:) = A(i).size;
end
siz
```

Visit

`http:/www.theochem.kun.nl/˜pwormer/matlab/ml.html`.

where you will find this script under the name `size_struct.m`. Download it to your directory and execute it. Was your prediction correct? If not, experiment with the appropriate MATLAB statements until you feel that you understand what is going on in this script (that serves no other purpose than comparing MATLAB data structures and their dimensions).

Exercise 57
Design a MATLAB data structure for the periodic system of elements. Include the following information:

- The full English name of the element and its standard chemical abbreviation.

- The masses and abundances of the naturally occurring isotopes.
- Electron configuration, i.e., number of electrons in $n = 1, 2, \ldots.$ shells.
- Prepare the periodic system for the first 10 elements. A `.txt` file of isotopic masses can be found at the url

  `http:/www.theochem.kun.nl/~pwormer/matlab/ml.html`.

  The same site contains a periodic system as a `.pdf` file. This information originates from the USA government:

  `http://physics.nist.gov/PhysRefData/Compositions/index.html`

C. Complex numbers

Scalars, and also vectors and matrices that will be introduced later, cannot contain real numbers, but also complex numbers. Assigning complex numbers to a variable is quite straightforward in MATLAB. You can use either the symbol `i` or `j` to define the imaginary part of a complex number:

```
>> x=1+2*i;
>> x=i
x = 0.0000 + 1.0000i
>>y=x^2
y=-1
```

or you can use the following method:

```
>> x = 1;
>> y = 2;
>> z = complex(x, y)
z = 1.0000 + 2.0000i
```

To obtain the real and imaginary part of a variable, one can use the commands `real` and `imag`

```
>> z = 1 + 3*j;
>> x = real(z)
x = 1
>> y = imag(z)
y = 3
```

The command `abs` gives the complex modulus

$$|x+iy| = \sqrt{x^2+y^2} \tag{C.1}$$

of the variable

```
>> z = 1 + 3*j;
>> x = abs(z)
x = 3.1623
```

Note that in the more recent versions of MATLAB the use of `1i` (or `1j`) as the default symbol for the imaginary part of a complex number is stimulated to avoid confusion with the common counter index `i` or `j`.

For vectors containing complex numbers (`'`) gives the transpose of the complex conjugate

```
>> x=[1+8i 2+9i];
>> b=x'
b =
  1 - 8i
  2 - 9i
```

To obtain the non-conjugate transpose use `.'`

```
>> x=[1+8i 2+9i];
>> b=x.'
b =
  1 + 8i
  2 + 9i
```

C.1 Function optimization