Friedrich-Alexander-Universität Erlangen-Nürnberg

Preparation Course MATLAB Programming

Dr.-Ing. Heinrich Löllmann
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl für Multimediakommunikation und Signalverarbeitung
Cauerstraße 7, 91058 Erlangen
heinrich.loellmann@fau.de

Prof. Dr. Meinard Müller
Friedrich-Alexander-Universität Erlangen-Nürnberg
International Audio Laboratories Erlangen
Lehrstuhl Semantic Audio Processing
Am Wolfsmantel 33, 91058 Erlangen
meinard.mueller@audiolabs-erlangen.de

AUDIO LABS

LMS

Authors and Contributors:

Heinrich Löllmann
Meinard Müller
Christian Hümmer

Contact:

Prof. Dr. Meinard Müller
Friedrich-Alexander Universität Erlangen-Nürnberg
International Audio Laboratories Erlangen
Lehrstuhl Semantic Audio Processing
Am Wolfsmantel 33, 91058 Erlangen
meinard.mueller@audiolabs-erlangen.de

Dr.-Ing. Heinrich Löllmann
Friedrich-Alexander Universität Erlangen-Nürnberg
Lehrstuhl für Multimediakommunikation und Signalverarbeitung
Cauerstraße 7, 91058 Erlangen
heinrich.loellmann@fau.de

Preparation Course MATLAB Programming, © 2018
(Version 1.3)

Contents

Chapter 1

Introduction to MATLAB

The programming language MATLAB is a versatile and powerful tool for solving various problems in mathematics and engineering. MATLAB is an acronym for **Mat**rix **Lab**oratory which already bespeaks its suitability for matrix computations and solving tasks in linear algebra. Real and complex-valued matrices are thereby the elementary data types in MATLAB.

This experiment serves as an introduction to the programming language MATLAB. In particular, an introduction into the user interface, dealing with scalars, vectors and matrices and the execution of elementary operations will be introduced.

Note: Please create an M–file with all used commands and inputs when solving the various tasks such that you can reproduce your work-flow later on.

1.1 MATLAB–User Interface

The way to start MATLAB as well as its appearance depend on the operating system and the GUI (Graphical User Interface) that is being used:

- In **MS Windows**, MATLAB is usually started from the Start Menu or by clicking on the corresponding Desktop icon,

- in **Unix**–like operating systems, it is usually started by typing the command `matlab` in a Terminal–Window (Shell) or an Application Launcher.

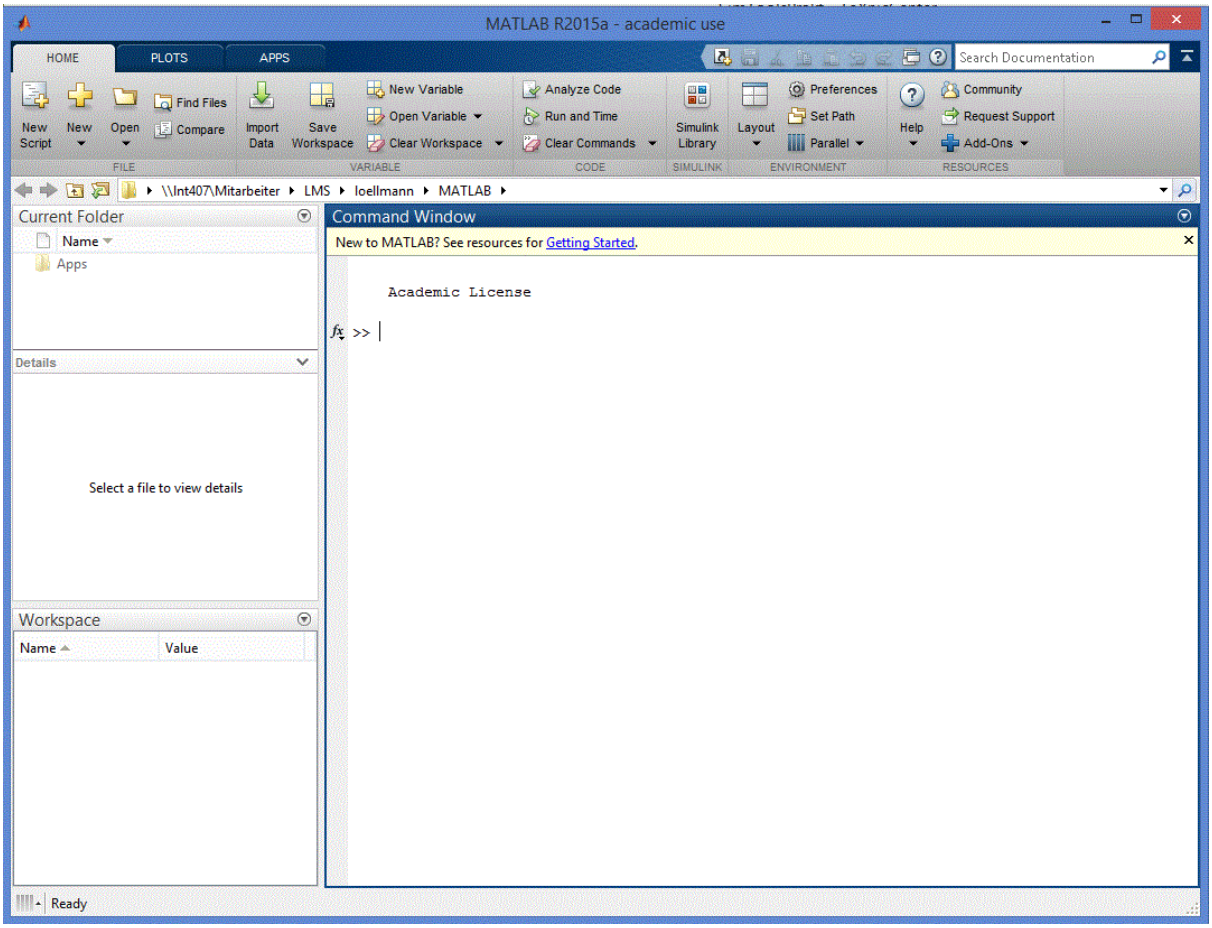


Figure 1.1: Opening window of MATLAB (Version 8).

MATLAB usually starts with a window as shown in Figure 1.1 which comprises of 3 sub-windows:

1. **Command Window**

   In this window, MATLAB commands are entered which are executed immediately. To do that, the mouse cursor must be positioned after the Prompt (double arrow) >>. The commands are entered to the right of the double arrow.

   The results and error messages will be shown in this window as well.

The commands in MATLAB are *case-sensitive* and allow similar operations as the commands in terminal windows of some operating systems, e.g., to list the current directory or to change the directory path. Appendix A provides lists with the most common MATLAB commands.

2. **Workspace**

   Here, all the variables that are used in the current session are displayed by their name, size and data type. By entering the command `clear` in the Command Window, all variables of the Workspace will be deleted.

   Table A.4 in the Appendix comprehends the most important MATLAB–commands for operations in the Workspace.

3. **Command History**

   Here, all the commands of the current session are displayed. A double click on a command will execute it again.

   Previously used commands can also be called in the Command Window by pressing the up arrow key.

MATLAB can be closed by entering the command `quit` or `exit` in the Command Window.

1.2 Data Types, Scalar Variables, Elementary Operations and Functions

MATLAB can be used as a "pocket calculator"; if a mathematical operation is entered in the Command Window, it will be evaluated. Each entry is completed by pressing the "Enter" key. MATLAB obeys the conventional precedence order in mathematics. If two operations with the same priority are put together, the evaluation follows from left to right.

1.2.1 Data Types and Variables

A variable refers to a data object which is linked with a certain data type and whose value can be changed during the program flow.

MATLAB uses the concept of implicit allocation and type declaration, i.e., a variable is allocated and its data type determined implicitly by just assigning a value to a variable; an explicit memory allocation and data type declaration as in many other programming languages like C or C++ is not needed.

If a value is assigned to an existing variable, the variable takes over the data type of the assigned value (which may differ from the original data type of the variable). If a variable does not exist before the assignment, it will be dynamically created and its data type is determined by the value assigned to it.

MATLAB has several predefined data types. With these types, it is possible to represent scalars as well as one- and multi-dimensional arrays (see Sec. 1.4). The data types `double` and `char` are of special importance.

- MATLAB executes all numeral computations in a double-precision floating-point format. The data type `double` is not only used to store real numbers, but also for complex numbers (see also Sec. 1.4), i.e., there is no dedicated data type for complex numbers. (A complex number is indicated in MATLAB by assigning the Attribute 'complex' to the variable class 'double'.) The real and, if so, complex part of a `double` variable is stored in the form of a floating-point number $\pm M\, 2^E$ that consists of a sign, exponent E and the mantissa M. In MATLAB, the exponent can take on integer values in the range of $-1024, \ldots, 1023$, the mantissa M can take on values given by $\sum_{k=1}^{52} d_k\, 2^{-k}$ with $d_k \in \{0, 1\}$.

  In MATLAB, numbers $v \cdot 10^{\pm x}$ are represented as v`e`$\pm x$, e.g., $3.5 \cdot 10^5$ as `3.5000e+5` (or short `3.5e5`). By default, only 4 positions after the decimal point are displayed. The command `format` can be used to change this number.

- The data type `char` is used to store single characters or character strings. The strings are stored as vectors and their elements are single characters counted from left to right. The number of characters is called the length of the string. Strings are bordered in single quotes (`'`) such that the space character can be an element of a string.

MATLAB offers many other data types besides the above mentioned ones such as the logical variable `logical` that takes on only the values `1` for true and `0` for false.

A unique name must be assigned to every data object by which it can be addressed later. A valid variable name starts with a letter, followed by letters, digits, or underscores. German umlauts are not allowed.
Attention: MATLAB is a *case sensitive* programming language and distinguishes between upper and lower case letters!

1.2.2 Assignment, Operations and Functions

As already mentioned, a variable is created just by assigning a certain value to it. The type of the newly defined variable is determined implicitly by the assigned value, i.e., the declaration of a data object is done with the help of an assignment of the form:

```
variable_name = expression
```

The `expression` will be evaluated and the resulting value is assigned to the variable on the left hand side that can be addressed later on by the `variable_name`. MATLAB allows arbitrarily many spaces between variables and operations and spaces can improve considerably the readability of the expressions and the whole program code, respectively.

If the result of an arithmetic operation should not be displayed, one has to add a semicolon (`;`) after the command. The value of a variable can be shown by simply entering the name of the variable in the Command Window. If a name is not assigned to the result of an operation, it will be stored under the variable name `ans`. The name of an existing variable can be changed by a value assignment `variable_name = expression`. Now, the variable that was before called `expression` is now stored with the name `variable_name`.

The content of the Workspace (available variables and their types) can be listed in the Command Window by using the commands `who` or `whos` which show their name, size and data type (see Table A.4 in the Appendix for further details).

Variables can also be concatenated with each other by various operations. MATLAB provides all well-known elementary mathematical operations such as the summation (`+`), subtraction (`-`), multiplication (`*`), division (`/`), etc. (see Table A.7 in the Appendix for more information).

Relational operations are also available such as larger than (>), less than (<), larger than or equal to (>=), less than or equal to (<=), equal to (==), not equal to (~=), etc. (see Table A.7 in the Appendix).

Besides the arithmetical and relational operators, MATLAB provides also a variety of standard functions such as sine (`sin`), cosine (`cos`), natural logarithm (`log`), exponential function (`exp`) and many more (see Table A.8 in the Appendix for more information).
Note: The arguments of trigonometric functions must be given in radiant!

Here are some examples of inputs to MATLAB and the corresponding results:

- The numbers 2 and 3 are added up and stored in the variable c:

```
>> c=2+3
c =
     5
```

- The result of the computation $3 \cdot 2^8 - \frac{1923}{3}$ is stored in the variable e:

```
>> e=3*2^8-1923/3
e =
   127
```

- A character string `Hello` is defined and stored in the variable `c`:

```
>> c='Hello'
c =
Hello
```

- The values of the variables a and b are compared and the result (True (`1`) or False (`0`)) is assigned to the variable c:

```
>> a=1; b=2;
>> c=(a>b)
c =
     0
>> b=0;
>> c=(a>b)
c =
     1
```

1.2.3 MATLAB–Help and Error Messages

Detailed information about all commands available in MATLAB is obtained by means of the command `help`. Moreover, the complete MATLAB documentation is available online. The most important help commands are listed in Table A.5 in the Appendix.

The help functionality can also be helpful when errors occur. If an input with incorrect syntax is used, MATLAB displays an error message that describes the type of the error and, if possible, also the location of the error, e.g:

```
>> a=2**3
??? a=2**3
        |
Error: Unexpected MATLAB operator.

>> a=sine(45)
??? Undefined function or method 'sine' for input arguments of type 'double'.
```

Preparation:

Exercise P-1.1: Explain which of the following variable names are not permitted in MATLAB and why:

```
determinant_of_A    A    3_D    value-1

 a    value5.8    Ü    ThisIsTheResult
```

Practical Experiments:

Exercise E-1.1: Have a look at the help texts of

- general purpose commands `help matlab/general`
- operators and special characters `help matlab/ops`

and get familiar with the elementary operations.

Exercise E-1.2: Create a floating point variable with the value $\pi/4$. After that, find the value of $\cos(\pi/4)$ and store it in a second variable.

Exercise E-1.3: Check for several pairs of positive values x and y if the equation $\ln(xy) = \ln(x) + \ln(y)$ is also valid in MATLAB. What do you observe?

1.3 Exceptions

Sometimes results may occur that cannot be interpreted or stored as a real (or complex) floating-point number. The treatment of such exceptional cases is summarized in the following.

- `Inf` (also: `inf`):
  The "value" `Inf` is assigned to the result of operations that either yield the value $\pm\infty$ (division by zero) or a value which is above the maximum value that can be represented in MATLAB (overflow). MATLAB distinguishes between `Inf` and `-Inf`.

- `0`:
  An exact zero is, among others, displayed if the result of the computation is smaller than the smallest number that can be represented in MATLAB. Thereby, MATLAB keeps the sign of the result.

- `NaN` (also: `nan`):
  "*Not a Number*" is displayed when the result of a computation is mathematically not defined. Almost all operations provide the result `NaN` if `NaN` is given as an input argument.

The aforementioned 'exceptional values' `Inf` and `NaN` can be assigned to a variable as any other value (e.g., `a=-Inf`). It can be checked with the logical operators of Table 1.1 whether the result of an operation or a variable attains an exceptional value. These functions operate element-wise if they get a matrix as input argument.

Table 1.1: Functions to check exceptional cases

Function	Result = true (logical 1), if
`isfinite(a)`	$\|a\|$ is finite
`isinf(a)`	$\|a\| = \infty$
`isnan(a)`	`a==NaN`

A variable can only attain one of the three properties `isfinite`, `isinf` or `isnan`.

Practical Experiments:

Exercise E-1.4: Which results do you obtain for the following computations:

```
1/0    0/0    Inf/Inf    Inf+Inf    Inf-Inf
       isinf(Inf-Inf)    isnan(Inf)
```

Explain the behavior of MATLAB.

Exercise E-1.5: Which results are returned:

```
1/2^1022    1/2^1024    1/(1/2^1022)    1/(1/2^1024)
sin(nan)    sin(inf)    max([-inf nan])    sum([-inf inf])
```

1.4 Vectors and Matrices

One of the strengths of MATLAB is that it treats scalars, vectors and matrices in the same manner and does not differentiate between them. Not only scalars can be stored in a variable, but also vectors (one-dimensional arrays) or matrices (two-dimensional arrays), and even multidimensional arrays in principle.

Vectors are defined by using a pair of squared brackets (`[]`). If a row vector needs to be formed, the individual vector elements are inserted into squared brackets next to each other, separated by the space character:

```
>> b=[3 4 5]
b =
     3     4     5
```

A column vector is created by listing its elements within squared brackets, separated by semicolons:

```
>> b=[3;4;5]
b =
     3
     4
     5
```

Vectors whose elements begin with a start value `Start` and can be counted with a constant step-size `Step` until the final value `End` is reached, can be created with the help of the colon (`:`) operator:

```
Start:Step:End
```

creates a row vector

```
[Start Start+Step Start+2*Step ... End]
```

If there does not exist an integer value k for which the statement `Start`$+k\cdot$`Step`$=$`End` is true, then the last element of the vector will be the largest number which is smaller than `End` but can be created by the above statement with an integer value k.

For `Step` $= 1$, the declaration of the step-size can be omitted (`Start:End`). For `Step` < 0, the values will be counted backwards.

The following examples should illustrate the explained relations:

```
>> a=1:1:6
a =
     1     2     3     4     5     6

>> a=1:6
a =
     1     2     3     4     5     6

>> a=1:2:6
a =
     1     3     5

>> a=4:-1:0
a =
     4     3     2     1     0
```

Individual elements of a vector are accessed by means of an index (subscript). It is important to note that the counting of vector components starts at "1" (and not at "0" as for some other programming language like C or C++). For example, the second element of vector `a` can be accessed by entering the command `a(2)` as demonstrated in the following:

```
>> a=1:2:10;
>> a(2)
ans =
     3
```

Several elements of a vector can be accessed by entering the corresponding indexes in a vector:

```
>> a=1:2:10;
>> a([2 3 4])
ans =
     3     5     7
```

or a shorter version:

```
>> a(2:4)
ans =
     3     5     7
```

where the index vector was created by using the increment–operator; compare to:

```
>> a(1:2:5)
ans =
     1     5     9
```

If the last element of the vector should be addressed, the generic index `end` can be used:

```
>> a=1:2:10;
>> a(end)
ans =
     9
```

The whole vector can be addressed either by the statement `a`, `a(1:end)` or `a(:)`, where the latter always creates a column vector:

```
>> a=1:2:10;
>> a
a =
     1     3     5     7     9
```

```
>> a(1:end)
ans =
     1     3     5     7     9

>> a(:)
ans =
     1
     3
     5
     7
     9
```

The length (number of elements) of a vector can be determined by using the function `length`:

```
>> a=1:2:10;
>> length(a)
ans =
     5
```

Matrices are two-dimensional arrays. Similarly to vectors, they can be generated and processed in several ways by MATLAB. Matrices are also defined by means of squared brackets (`[ ]`) and the rows of matrices are separated by semicolons.

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

In turn, matrices can be used as elements in another matrix:

```
>> A = [1 2; 3 4];
>> B = [5 6; 7 8];
```

```
>> C = [A B]
C =
     1     2     5     6
     3     4     7     8

>> D = [A; B]
D =
     1     2
     3     4
     5     6
     7     8
```

An error occurs, if the dimensions of the sub-matrices do not match properly.

Some special matrices can be created by the commands `zeros(number)` and `ones(number)`. `zeros` creates an all-zero matrix and `ones` creates a matrix where each element is equal to one, both of dimension $number \times number$. Row and column vectors containing only ones or zeros can also be created by these commands:

```
>> Z = zeros(2,3)
Z =
     0     0     0
     0     0     0

>> O = ones(3,2)
O =
     1     1
     1     1
     1     1

>> o = ones(1,5)
o =
     1     1     1     1     1
```

The command `eye` creates an identity matrix with the given dimensions:

```
>> I=eye(3)
I =
     1     0     0
     0     1     0
     0     0     1
```

Single elements of a matrix can be addressed by their row and column index, e.g.:

```
>> A=[1 2; 3 4];
>> A(1,2)   %Extract the element in row 1, column 2
ans =
     2

>> A(2,2)
ans =
     4
```

The row and column indices of the extracted value shall not exceed the dimensions of the matrix which can be obtained by the command `size(A)`:

```
>> A=[1 2 3; 4 5 6];
>> size(A)
ans =
     2     3

>> A(1,4)
??? Index exceeds matrix dimensions.
```

The colon operator is used to select the columns and rows of a matrix. A complete column/row is selected, if the column/row index is fixed and the row/column index runs through all the possible values (by means of the colon operator) as demonstrated in the following:

```
>> A=[1 2 3; 4 5 6]
A =
```

```
     1     2     3
     4     5     6

>> row=A(2,:)
row =
     4     5     6

>> column=A(:,2)
column =
     2
     5
```

All the MATLAB operators can also be applied to matrices (including vectors as special case). Thereby, it is important that the dimension of the matrices match properly according to the calculus rules of linear algebra. For example, both operands (matrices) must be of the same dimension for a summation or subtraction operation. The multiplication of variables containing matrices results in a matrix multiplication:

```
>> A = [1 2; 3 4];
>> b = [5; 6];
>> A*b
ans =
    17
    39
```

or

```
>> [1 2 3]*[1; 2; 3]
ans =
    14
```

since the product of a row and column vector is a scalar. On the other hand, the product of a column and row vector is a matrix:

```
>> [1; 2; 3]*[1 2 3]
ans =
     1     2     3
     2     4     6
     3     6     9
```

The *element-wise* multiplication and division of matrices is a frequently needed function and executed by the operators (`.*`) and (`./`). The operator (`.^`) allows to exponentiate the single elements of a matrix or vector:

```
>> x = [ 1 2 3 ] .* [ 1 2 3 ]
x =
     1     4     9

>> x = [ 1 2 3 ] ./ [ 1 2 3 ]
x =
     1     1     1

>> x = [ 1 2 3 ] .^ [ 1 2 3 ]
x =
     1     4    27
```

The largest and smallest elements of a matrix or vector can be found by the commands `max` and `min`, respectively. If $\boldsymbol{x}$ is a vector, the statement `max(x)` yields the largest element of the vector. If $\boldsymbol{A}$ is a matrix, the statement `max(A)` yields a row vector containing the largest elements of each matrix column. If two return parameters are specified, the statement `max` (`min`) returns the maximum (minimum) value as well as the index (position) of the extremal value within the array. In the case of the vector `x = [4 -2 6 1]`:

```
>> max(x)
ans =
     6
```

```
>> min(x)
ans =
    -2
```

```
>> [value,index] = max(x)
value =
```

```
     6
index =
     3
```

`sum(x)` and `prod(x)` return the sum and the product of the vector elements of $\boldsymbol{x}$, respectively. Using the previously defined vector `x`:

```
>> sum(x)
ans =
     9
```

```
>> prod(x)
ans =
     -48
```

Practical Experiments:

Exercise E-1.6: Create the variables x and y in your Workspace. The variable x shall be a row vector containing values from 0 to 10 with a step-size of 2 and the variable y shall be a row vector containing values from 0 to 100 with a step-size of 5. What are the lengths of the vectors x and y?

Exercise E-1.7: Create the following sums:

- Sum of all natural numbers from 1 to 100.
- Sum of all even numbers between 1 and 100.
- Sum of all numbers divisible by 5 between 1 and 1000.

1.5 M–Files and Creating Functions

MATLAB is not only able to process commands entered in the Command Window, but can also read in a command or a sequence of commands from a (program) file, which is essential to solve more complex tasks. Files containing MATLAB–commands are termed as M–files due to the file extension ".m".

M–files consist of a sequence of regular MATLAB–commands. *Comments* in the program code are essential to keep it understandable and to ease its maintenance. They start with the percent symbol (%) such that MATLAB does not interpret the statements that are right to the comment symbol. Accordingly, comments can be added at the end of a command line or they can be written in a separate line.

There are two types of M–files: *Script files* and *function files* commonly referred to as scripts and functions:

- **Scripts:**
  Scripts contain a sequence of MATLAB–commands that is executed consecutively as if the single commands therein would have been entered directly in the Command Window. To execute a script, its file name is entered in the Command Window without the extension (`.m`), for example, by typing `myscript` a file named `myscript.m` will be executed (if it is available in the current working directory).

  An M–file `compute.m` with the content

  ```
  A = [1 2; 3 4];
  B = [5; 6];
  A*B
  ```

  executes the 3 instructions and results in

  ```
  >> compute
  ans =
      17
      39
  ```

- **Functions:**
  A function is marked by the keyword `function` at the beginning. The main difference to a script is that they accept input arguments and can return one or more output arguments. Moreover, all the variables that are used within the function are local variables, i.e, they do not appear in the base Workspace of MATLAB and they are removed from the workspace upon the termination of a function.

  Functions help the user to tailor MATLAB according to his needs. For example, if a function is needed that calculates the sine of an angle in degree, the resulting function can look like this (saved as `sindeg.m`):

```
function y = sindeg(x)
%
% sindeg(x)  Sine values of x entered in degrees
%
y = sin(pi*x/180);
```

A call of the function returns

```
>> sindeg(90)
ans =
     1
```

The use of multiple input and output arguments can be realized as follows:

```
function [s, c] = sincos(x,A)
%
% sincos(x,A)   A*sin(x) and A*cos(x) of x entered in degrees
%
s=A*sin(pi*x/180);
c=A*cos(pi*x/180);
```

A call of this function returns

```
>> [s,c] = sincos(90,2)
s =
     2
c =
   1.2246e-16
```

Preparation:

Exercise P-1.2: Draft a function that returns the sum of all integer values from 1 to n where n serves as input argument for this function.

Exercise P-1.3: Draft a function `reverse.m` that returns the elements of an input vector in reversed order.

Practical Experiments:

Exercise E-1.8: Write the function drafted in **P-1.2** and test it for some examples.

Exercise E-1.9: Write and test the function drafted in **P-1.3**.

1.6 Conditional Statements and Flow Control

MATLAB offers several possibilities to control the program flow of an M–file. There are five constructs available for that:

- `if` conditionally execute statements
- `switch` switch among several cases based on the expression
- `for` repeat statements a specific number of times
- `while` repeat statements an indefinite number of times
- `break` terminate execution of a `while` or `for` loop

1.6.1 `if`–Statement and `switch`

It is often needed to execute commands in dependence of a variable value for which the `if`-statement comes in handy. A logical expression follows the keyword `if`. If this logical expression is true, the commands in the next block are executed until the keyword `end` is reached:

```
...
if i>0 ,    % i larger than 0?
    ...     % if true, execute commands entered here
end;
...
```

The optional `else`-statement can be used to execute a set of commands for the case that the logical expression of the `if`-statement is false:

```
...
if i>0,     % i larger than 0?
    ...     % if true, execute commands entered here
else,
    ...     % otherwise execute commands entered here
end;
...
```

A program flow that is dependent on multiple logical expressions can be realized by means of the `elseif`-statement:

```
...
if i>0,     % i larger than 0?
    ...     % if true, execute commands entered here
elseif i<0, % i smaller than 0?
    ...     % commands to be executed if i<0
else,
    ...     % otherwise (i==0) execute commands entered here
end;
...
```

Note: (==) is a logical operator that tests for equality, whereas (=) assigns a value to a variable.

The conditional execution of statements for a number of different cases can be realized with the `switch`-statement more efficiently than with the `elseif`-statement. The corresponding syntax is as follows:

```
a = ...
switch a
    case 1,     % a == 1
        ...
    case 2,     % a == 2
        ...
    case 3,     % a == 3
        ...
    otherwise,  % otherwise
        ...
end;
```

1.6.2 `for`–loop

A `for`–loop allows to execute operations repeatedly in a loop for the various values of a variable (mostly a counter). The part that needs to be executed in a loop ends

with keyword `end`. For example, the calculation of the first 10 squared numbers can be realized by means of a `for`-loop in the following way:

```
...
for i=1:10,
  i^2
end;
...
```

The expression `i=Start:Increment:End` causes that the loop will be executed for the variable `i` taking on values from `Start` in steps of `Increment` until `End` is reached. If `Increment=-1` (e.g. `i=10:-1:1`), the values are counted "backwards".

Loops can also be nested:

```
...
for i=1:10,
  ...
  for j=10:-1:i,  ...  end;
end;
...
```

1.6.3 `while`–loops and `break`–statements

A `while`–loop allows to execute a set of operations while the condition stated at the beginning of the loop is fulfilled. The keyword `end` marks again the set of operations to be executed within the loop.

For example, if a value should be doubled repeatedly until an upper limit is reached, it can be implemented in MATLAB as follows:

```
i = ...
while i<upperlimit,   % i smaller than the upper limit?
    i=i*2;            % yes
end;
...
```

Sometimes it is necessary to check after the `while`–loop, how the variable changed during the iterations. For example, a given number should be divided by 3 as long as the difference from the previous value is smaller than a given threshold.

```
i = ...
diff = Limit + 1;
while diff > Limit,  % Difference larger than the threshold?
    i_prev=i;
    i=i/3;
    diff=abs(i-i_prev);
end;
...
```

With the help of the `break`–command, `for` and `while`-loops can be terminated once the `break`-command is reached. Such a termination is usually only reasonable if a certain condition is fulfilled as exemplified in the following:

```
i = ...
while 1,              % Infinite loop
    i_prev = i;
    i = i/3
    diff = abs(i-i_prev);
    if diff <= Limit,  % condition to abort the loop
        break;
    end;
end;
...
```

An infinite loop is given, since the logical expression after the `while` command is always true, and the loop only terminates if the `break`-command within the `if`-statement is reached. It can be reasonable to terminate a loop based on a set of conditions which can be linked by logical AND and OR operators (see Table A.7).

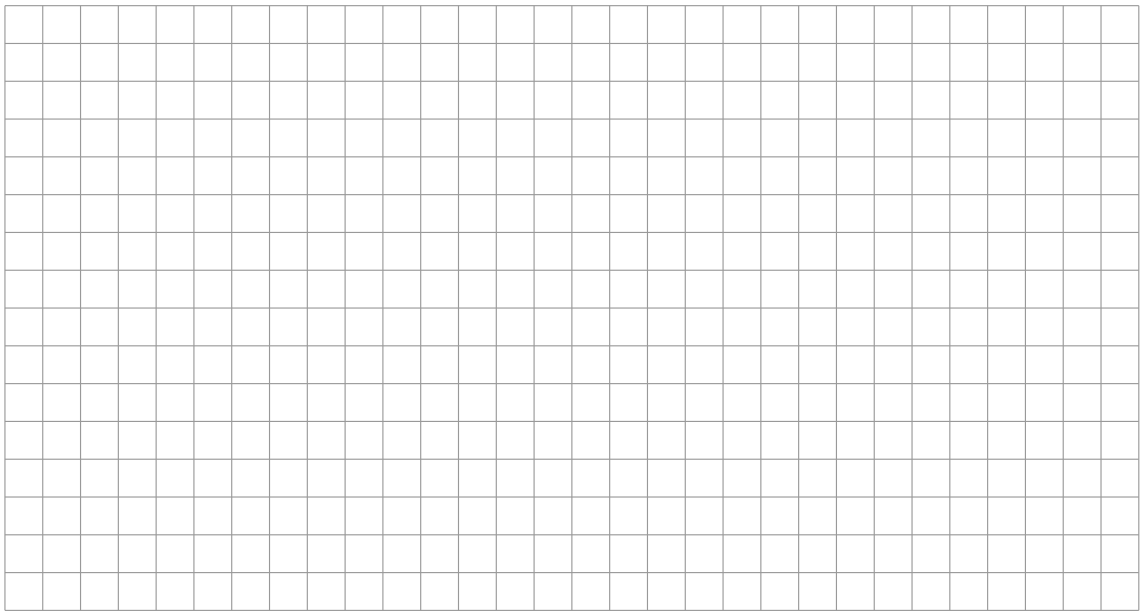
Preparation:

Exercise P-1.4: The roots of a given function $f(x)$ (available in MATLAB under the name `myfun`) should be found with the help of *nested intervals*.

If one knows that the continuous function $f(x)$ has a single root within the interval $[a, b]$, where $f(a)$ and $f(b)$ have opposite signs, the root can be localized by means of iterative interval bisection with steadily increasing precision.

Draft a function `interval.m` that performs the interval bisection for a given function and starting interval. Proceed as follows:

1. The interval $[a, b]$ whose center is given by $c = (a + b)/2$ is divided into two sub-intervals $[a, c]$ and $[c, b]$.
2. Calculate the value $f(c)$.
3. An interval is chosen that definitely contains a root: The interval $[a, c]$ contains a root of $f(x)$, if $f(a)$ and $f(c)$ have different signs, otherwise the root lies within the interval $[c, b]$. (The iteration terminates of course if $f(c) = 0$ and the root c is returned.)
4. The procedure is repeated until the found interval length is smaller than a given threshold and a sufficient approximation for the root is obtained.



Practical Experiments:

Exercise E-1.10: Implement the function `interval.m` and test it for the following functions:

- $f_1(x) = \mathrm{e}^{-x} - 5$, root search within the interval $[-2, -1]$
- $f_2(x) = -0.25\,x^3 + 1.25\,x$, root search within the interval $[-3, -1]$

Thereby, the function `myfun` can be defined as a so-called *anoymous function*, e.g., `myfun= @(x)(exp(-x)-5);`, which can be used like any ordinary MATLAB function, e.g., `y= myfun(0);`. Moreover, such a function can be passed as argument to another function (similar as for variables).

Note: In order to check the found root you can plot the function as follows:
`x=-3:.01:3; plot(x,myfun(x)); grid on;`

Literature to Experiment 1

[1] The MathWorks Inc. *MATLAB 7.0 (R14SP2)*, The MathWorks Inc., 2005

[2] A. Gilat. *MATLAB: An introduction with Applications*, John Wiley & Sons, 2004.

[3] J. Michael Fitzpatrick and J. D. Crocetti *Introduction to Programming with Matlab*, Vanderbilt University Nashville (TN),2010

[4] D. Houcque *Introduction to Matlab for Engineering Students*, Northwestern University, 2005

[5] R. L. Spencer *Introduction into Matlab*, Brigham Young University, 2000

Chapter 2

Graphics in MATLAB

The graphical representation of results is, besides the numerical computation, an essential feature of mathematical software tools. This experiment provides an introduction into the basic functionality of MATLAB to plot functions.

2.1 Two-dimensional Representation

2.1.1 Representation of Functions with One Variable

This kind of representation is used to represent the values of a function

$$y = f(x) \tag{2.1}$$

for an argument x within the interval

$$x \in [x_1,\, x_2]. \tag{2.2}$$

In order to represent this function by MATLAB, it needs to be evaluated for a discrete set of data points which are usually (but not necessarily) uniformly spaced.

This shall be illustrated for the following function

$$y = 3\sin(x) \tag{2.3}$$

with

$$x \in [0,\ 4\pi]. \tag{2.4}$$

In a first step, a vector x is needed which contains the data points at which the function shall be represented and plotted, respectively. The function values are also arranged in a corresponding vector.

Preparation:

Exercise P-2.1: Specify the MATLAB–command that creates a vector of data points x. The distance of the data points should be $\frac{\pi}{4}$ and the values in the vector should be given in an ascending order.

Exercise P-2.2: How can the values of the function $y = 3\sin(x)$ be determined from the given vector x?

The desired graphical representation of the function can be obtained by using the MATLAB–command `plot`. This is done by using the vectors x and y as input arguments of the `plot`-function:

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

The representation is done such that the function values for the discrete data points are connected via a line (linear interpolation). Accordingly, it is important that the data points are arranged in the correct order to produce the desired result. The figure is scaled automatically such that all function values are visible.

Practical Experiments:

Exercise E-2.1: Visualize the function $y = 3\sin(x)$ for the interval $x \in [0,\, 4\pi]$. The distance of data points should be $\frac{\pi}{4}$.
Note: All the used commands and inputs should be stored in an M–File such that the work-flow can be easily reproduced later on.

Exercise E-2.2: Interpret the result and propose an approach to improve the representation.

Figure 2.1 provides an example for the plot of a function using the `plot`–command. There is a possibility to activate the zoom function by clicking on the symbol  . If

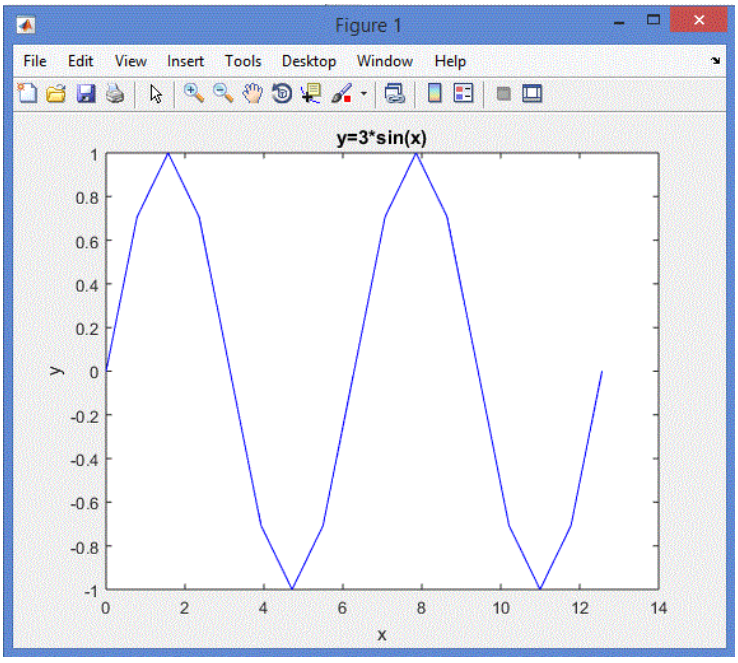


Figure 2.1: Example for a graphical output.

the zoom-mode is activated, one can view certain parts of the figure in greater detail by selecting the zoom area with the mouse pointer.

When working with 3-dimensional graphs, one can use the button  to change the view, which will be explained later on in more detail.

A correct graphical representation comprises the function values as well as a correct labeling of axes. For this, the commands `xlabel` and `ylabel` are provided and a title can be added to a figure by the command `title`. All these commands require a string as input argument, e.g.:

```
xlabel('x-name');
```

Practical Experiments:

Exercise E-2.3: Label the figure created within **E-2.2**.

Frequently, functions need to be visualized whose input and output arguments have units. The correct display of such functions should be exemplified by means of a sine wave.

Practical Experiments:

Exercise E-2.4: Generate a figure that depicts a sine wave with a frequency of 1 kHz and an amplitude of 1 V. The time axis shall be in the range of $[-1\ \mathrm{ms},\ 3\ \mathrm{ms}]$. Pay attention to the correct labeling (x-axis in ms, y-axis in V).

2.1.2 Representation of Loci (X-Y Representation)

Besides representing a single function over its argument, it is also possible to plot two functions that have a common argument (e.g., t) in a single figure. The function values of one function are used for the abscissa (x-axis) and those of the second function for the ordinate (y-axis).

$$x = f_1(t) \tag{2.5}$$

$$y = f_2(t) \tag{2.6}$$

The graph created by such a function is termed as locus. To illustrate this, the representation of a circle is used as an example in the following.

Preparation:

Exercise P-2.3: Specify the functions $f_1(t)$, $f_2(t)$ such that a circle is created by varying the variable t. What is the range of values that needs to be taken for t?

Practical Experiments:

Exercise E-2.5: Create a variable vector t and also the vectors x and y corresponding to your considerations in **P-2.3**.

The representation is now done in the same way as in the case of a simple function with the only difference that the vector x contains now a set of function values and not a set of data points as before. As before, the order of the value pairs is also important here such that the connecting line is properly plotted. Thereby, the correct order has to be insured by an appropriate choice for the variable vector t.

Practical Experiments:

Exercise E-2.6: Plot a figure with the already created vectors x and y.

Exercise E-2.7: Interpret the result. Why is the desired result not obtained?

In addition to the labeling of plots, the scaling of the coordinate axis is also of great importance and can be changed by using the command `axis`. By entering

```
help axis
```

in the Command Window, a detailed overview about the different options is given.

Practical Experiments:

Exercise E-2.8: Which options offers the command `axis` for a correct (undistorted) representation of the circle and how do they differ?

Exercise E-2.9: Correct the previously generated figure accordingly.

2.1.3 Representation of Multiple Functions in a Single Plot

The different methods to plot multiple functions in a single figure are treated in the following. The easiest option in MATLAB is to use the command `plot`. Representing the two functions

$$y_1 = f_1(x)\ , \qquad y_2 = f_2(x)$$

in a single figure can be done the by the statement

```
plot(x,y1,x,y2);
```

It is also possible to use different variable vectors x_1 and x_2:

$$y_1 = f_1(x_1)\ , \qquad y_2 = f_2(x_2)$$

as follows

```
plot(x1,y1,x2,y2);
```

The extension to more than two functions is simply done by adding more (x, y)–pairs. In such a way, it is also possible to plot multiple loci in a single figure.

MATLAB assigns automatically different colors to the different curves within a plot. However, the differentiation between curves by means of different line colors can be insufficient, e.g., if a figure is plotted in a black-white or gray color scheme. For such cases, additional markers for the curves are needed.

The choice of color and other line properties can be specified by additional parameters within the `plot`-function. The available colors, line styles and marker symbols are listed below (obtained by the command `help plot`):

Color		Marker		Line style	
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dashdot
r	red	+	plus	--	dashed
g	green	*	star		
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

An attribute can be chosen from every column. If one of the columns is not used, MATLAB takes a default property. The combination of the wanted attributes is connected to a string. For example, `'co--'` draws a dashed cyan line with circles at the positions of the data points and `'b-'` results a continuous blue line. Such parameters can be added to any (x,y)–pair as a third parameter, e.g., `plot(x,y,'b-')` or `plot(x1,y1,'b-',x2,y2,'c--')`.

This shall be exemplified for the previously discussed example $y = 3\sin(x)$.

Practical Experiments:

Exercise E-2.10: Plot the function $y = 3\sin(x)$ for data points at spacing of $\frac{\pi}{8}$. Use different combinations for the attributes and observe the results.

Exercise E-2.11: Plot the functions $3\sin(x)$ and $2\cos(x)$ in the same figure. Use a distance of $\frac{\pi}{8}$ for the data points. Play around with different combinations for the line attributes.

The previously discussed options to display multiple curves in a single figure do not allow to label them individually. This problem can be solved by creating a legend with the help of the command `legend`. Its input arguments are strings containing the description (label) for each line, e.g., `legend('Function 1','Function 2')`.

Practical Experiments:

Exercise E-2.12: Choose a combination of attributes that is suitable to print out the figure of **E-2.11** in black-and-white with both lines being distinguishable. A possible solution is illustrated in Figure 2.2.

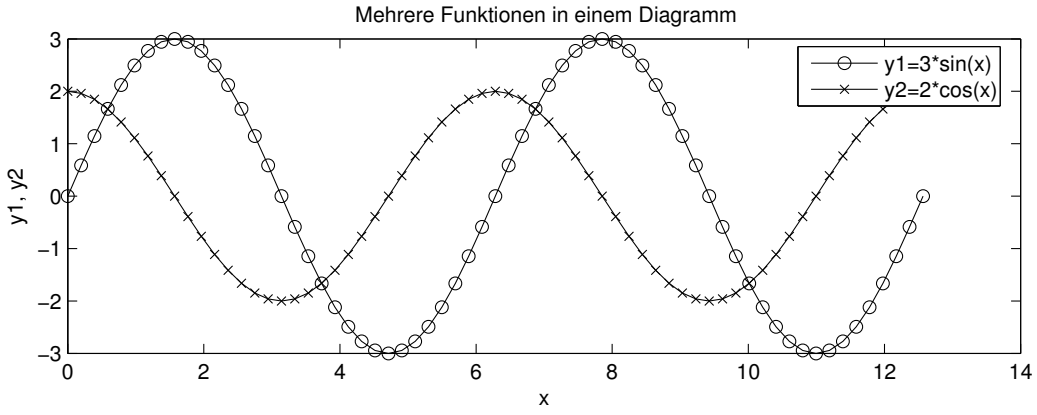


Figure 2.2: A possible solution of **E-2.12**.

2.1.4 Variants of `plot` and Auxiliary Functions

In addition to the previously discussed methods for plotting a figure with MATLAB, there is a variety of further commands and options available for this purpose. In the following, some of them are briefly introduced.

Practical Experiments:

Exercise E-2.13: Create the vectors x and y as done in Tasks **E-2.10**. Compare the results obtained by using either `plot`, `stem`, `stairs` and `bar`. For these functions, only x and y are needed as input parameters (as for the introduced `plot` function).

For which application scenarios would you prefer `plot`, `stem`, `stairs` or `bar`?

A variety of auxiliary functions is provided to change the appearance of a figure.

First, the creation and hiding of the coordinate grid is introduced.

Practical Experiments:

Exercise E-2.14: Create a figure of your choice and use the commands `grid on` and `grid off`. Observe the results.

It is often desirable to have an axis scaling that is different to the default setting of MATLAB. This can be realized with the help of the previously mentioned `axis` function, where the desired limits of the axes are provided as a vector.

Practical Experiments:

Exercise E-2.15: Create a plot of your choice and try out different scalings by using the function

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

Thereby, the values $x_{\min}$, $x_{\max}$, $y_{\min}$ and $y_{\max}$ need to be replaced with your desired values.

For the representation of functions whose domain and/or codomain span over multiple powers of ten, it is often beneficial to use a logarithmic scaling for the coordinate axes instead of a linear one. Different methods for this are examined in the following.

The functions `semilogy`, `semilogx` and `loglog` allow for a logarithmic scaling of the axis. Their input arguments are the same as for `plot`. In the case of `semilogy` and `semilogx`, one axis has a linear scaling and the second one a logarithmic scaling, whereby both axes feature a logarithmic scaling if the function `loglog` is used.

Practical Experiments:

Exercise E-2.16: Plot the function `y=exp(x)` within the range $x \in [-5, 10]$ by using `plot`, `semilogy`, `semilogx` and `loglog`. What do you observe? Which representation would you prefer?

So far, the call of the `plot` function has cleared all existing lines and plotted the new one in the current figure window. The behavior can be changed by using the command `hold on` such that new lines can be added to the existing ones. The original behavior can be reestablished by the command `hold off`.

MATLAB offers also the functionality to open several figure windows. A new empty figure window can be created by the command `figure`. The active window, where the plot commands are executed, can be selected in two ways. The most recently created window remains active until one clicks with the mouse pointer on another one. A window can be directly selected by using its so-called figure handle `figure(h)`, e.g., `figure(2)` activates the figure window with handle 2. (The figure handle is displayed by default in the top of the figure window.) A specific figure windows can be closed by the command `close(h)` and the command `close all` closes all figure windows that are currently open.

Practical Experiments:

Exercise E-2.17: Create several figure windows and experiment with the explained

options to activate them. Note that commands like `plot`, `axis`, `grid` etc. only apply to the currently activated figure.

It is also possible to display multiple subplots in a single figure window. For this, the function `subplot(m,n,p)` creates $m \times n$ subplots in the current figure and activates the subplot at position p. The subplots are thereby counted line-by-line.

Practical Experiments:

Exercise E-2.18: Create a new figure window with 6 subplots in 2 columns and 3 rows and experiment with the activation of the single subplots. Add titles and, if needed, a legend to the subplots.

2.2 Three-dimensional Representation

In the following, a function $f(x, y)$ shall be visualized where f is here dependent on *two* argument variables x and y which shall lie within the interval $x, y \in [-1, 1]$. For this, data points (x, y) are needed which are available by two matrices of the same dimensions. For example, a matrix `X` contains all the x-coordinates of the data points and a matrix `Y` comprises the corresponding y-coordinates.

For a representation of the function at equidistant data points $\Delta x = \Delta y = 0.1$, the

following matrices would be suitable:

$$\boldsymbol{X} = \begin{bmatrix} -1 & -0.9 & -0.8 & \cdots & 0.8 & 0.9 & 1 \\ -1 & -0.9 & -0.8 & \cdots & 0.8 & 0.9 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & -0.9 & -0.8 & \cdots & 0.8 & 0.9 & 1 \\ -1 & -0.9 & -0.8 & \cdots & 0.8 & 0.9 & 1 \end{bmatrix}$$

$$\boldsymbol{Y} = \begin{bmatrix} -1 & -1 & -1 & \cdots & -1 & -1 & -1 \\ -0.9 & -0.9 & -0.9 & \cdots & -0.9 & -0.9 & -0.9 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0.9 & 0.9 & 0.9 & \cdots & 0.9 & 0.9 & 0.9 \\ 1 & 1 & 1 & \cdots & 1 & 1 & 1 \end{bmatrix}$$

The matrices `X` and `Y` can be created very efficiently in MATLAB as follows:

```
>> [X, Y] = meshgrid(-1:.1:1, -1:.1:1);
```

With the help of these matrices, the matrix `F` containing all function values can be computed in the following way:

```
F = function(X,Y);
```

A three-dimensional surface plot of this function can be created by the command

```
surf(X,Y,F);
```

Practical Experiments:

Exercise E-2.19: The function

$$f(x,y) = \operatorname{sinc}(2x) + \operatorname{sinc}(2y) \qquad \text{with} \qquad \operatorname{sinc}(x) = \frac{\sin(\pi\, x)}{\pi\, x}$$

shall be plotted.

Create the matrices X and Y that have data points with a spacing of $\Delta x = \Delta y = 0.1$. Compute with these matrices and the MATLAB function `sinc` the matrix `F` containing the function values $f(x, y)$. (Why is the direct use $\sin(\pi x)/(\pi x)$ not feasible?)
Note: Use the element-wise matrix multiplication instead of the regular matrix multiplication.

Visualize the function by means of the `surf` function. As before, you can change the viewing angle by clicking on the button  and dragging the plot with the mouse pointer.

Visualize the functions

$$g(x, y) = \operatorname{sinc}(2\sqrt{x^2 + y^2}) \,, \qquad h(x, y) = \operatorname{sinc}(2x) \cdot \operatorname{sinc}(2y)$$

by the same procedure as before.

Which conceptual difference do you notice between the graphs for $f(x, y)$, $g(x, y)$ and $h(x, y)$? (Pay attention to the symmetry of the functions!)

MATLAB allows to choose from several built-in colormaps that determine the color scheme of a figure. This is done with the command

`colormap('`*name of the colormap*`');`

for which the following *colormaps* are available:

```
default, hsv, hot, gray, bone, copper, pink, white, flag, lines,
colorcube, vga, jet, prism, cool, autumn, spring, winter, summer.
```

Practical Experiments:

Exercise E-2.20: Use different colormaps for the previous figure.

The introduced command `surf` visualizes a function as a three-dimensional surface. The individual points are connected with black lines and the grid created by this is filled with colored surfaces where the color value depends on the corresponding function value.

Sometimes it is desirable to color only the grid which connects the function values and to leave the space in-between the grid lines white. This behavior is achieved by the command

```
mesh(X,Y,F);
```

which has essentially the same syntax as the function `surf`.

Practical Experiments:

Exercise E-2.21: Visualize the function $f(x, y)$ by a `mesh` plot and try out different colormaps.

2.3 Representation of Images

The representation and manipulation of images by means of MATLAB is now treated. For this, it is exploited that a digital image can be represented by a matrix. For the representation of a digital color image, 3 matrices are needed for its RGB-values (red, green and blue). A digital gray image can be represented by a single matrix containing values between 0 (black) and 1 (white), where the values in between correspond to gray values. In the next exercises, the representation and manipulation of black-and-white images is treated first, followed by the treatment of color images.

Practical Experiments:

Exercise E-2.22: Load with the command `load('some_images.mat')` the following matrices into the Workspace: `lena`, `einstein`, `noise`, `peppers`, `A`, `B`, `C`.

a) Display the matrices `lena` and `einstein` as images by means of the commands `imshow(lena)` and `imshow(einstein)`.

b) Transpose the matrices `lena` and `einstein` and display them. What do you observe?

c) Multiply the matrices `lena` and `einstein` with a scalar and display the scaled matrices as images. Experiment with different scalar values in the range from 0.1 to 2.0. How does the scaling effect the images?

d) Add a scalar to the matrices `lena` and `einstein`. Experiment with different scalars in the range from -1.0 to 1.0. How does adding a scalar influence the images?

e) Add up the two matrices `lena` and `einstein`, and visualize the result. What does it look like?

f) Visualize the matrices `einstein`, `noise` and `einstein+noise` as images. How does the addition of `noise` change the image of `einstein`?

g) Visualize the matrices `A`, `B`, `C`, `B` $\cdot$ `A` and `A` $\cdot$ `C` as images. Explain the observed results? *Hint:* Since the matrices `A`, `B` and `C` have values that are considerably larger than 1.0, the values must be normalized to represent them as an image. This can be done by using `imshow(A/max(max(A)))`, where `max(max(A))` returns the maximal value of `A`.

Color images can be represented in the RGB (red, green, blue) color space by means of 3 matrices: one for red, a second one for green and a third one for blue. These matrices can be concatenated to a three-dimensional array. A color image with 256×256 pixels can be represented by an array of dimensions $256 \times 256 \times 3$. The first plane in the third dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities.

The command `imshow(B)` checks if `B` is a two-dimensional array (matrix) or a three-dimensional array. In the first case, `imshow(B)` displays the image in black-and-white, while a color image is displayed in the second case. The following exercises should teach you the treatment and manipulation of color images in MATLAB.

Practical Experiments:

Exercise E-2.23: The 3-dimensional array `peppers` (that was loaded in the Workspace in the previous exercise) contains the RGB-representation of an image.

a) Visualize `peppers` as a color image by using `imshow(peppers)`.

b) Scale the color channels by using the following commands and visualize the modified 3-dimensional array as a color image. Experiment with different values. For example, display only the green parts of an image.

```
% multiply individual color channels
mod_peppers = peppers;
value_red = 0.8;
value_green = 1.2;
value_blue = 0.2;
%Red
mod_peppers(:,:,1) = peppers(:,:,1)*value_red;
%Green
mod_peppers(:,:,2) = peppers(:,:,2)*value_green;
%Blue
mod_peppers(:,:,3) = peppers(:,:,3)*value_blue;
%Display image
figure
imshow(mod_peppers)
```

c) In a similar way, add various constants to each of the three color channels and display the modified 3-dimensional array as a color image. Experiment with different values and explain your observations.

Chapter 3

Matrices and Systems of Linear Equations

In this experiment, you should apply some basic knowledge about linear algebra obtained in your undergraduate studies. First, some basic operations with vectors and matrices are recapped. Subsequently, efficient methods for solving a system of linear equations are treated. In the end, the Gaussian elimination algorithm needs to be implemented.

3.1 MATLAB–Commands in Linear Algebra

This section reviews some basic MATLAB operations to solve tasks in linear algebra.

3.1.1 Generating Vectors

In order to create a row vector $\boldsymbol{v} = [1, 2, 3, 4, 5]$, the command

```
v = [1 2 3 4 5]
```

can be used.

A column vector is obtained by transposing ($\boldsymbol{v}^{\mathsf{T}}$):

```
u = v.'
```

It is important to note that in case of a complex vector, the command `u = v'` (without the dot) results the complex conjugate transpose of a vector, i.e., the *Hermitian vector* of `v`, usually denoted as $\boldsymbol{v}^{\mathsf{H}}$.

Certain elements of a vector can be accessed, e.g., with the command `v(1:3)`. In this case, the first three elements of the given vector are selected: `ans = [1 2 3]`. The command `v(1:2:5)` selects every second element of the vector `v` starting with the first and ending with the fifth element: `ans = [1 3 5]`. If only one element (in this case the second) should be chosen, the command `v(2)` can be used.

3.1.2 Generating Matrices

Besides the previously discussed vectors, matrices can also be efficiently created and manipulated. A new row of the matrix is started with a semicolon (`;`). The statement `A=[1 2 3; 4 5 6; 7 8 9]` creates a (3×3) matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{3.1}$$

In order to select certain rows or columns in a matrix, the colon (`:`) operator can be employed: `B=A(1:2,2:3)` generates the matrix

$$\boldsymbol{B} = \begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix} \tag{3.2}$$

that consists of the first 2 rows and the columns 2 and 3 of the matrix $\boldsymbol{A}$. Single elements of a matrix can be addressed by the statement

```
A(1,3)
```

In addition to the possibilities to define matrices as described above, there are also some other built-in operations for special matrices:

- `rand(p)` generates a $(p \times p)$-matrix of random numbers with uniform distribution in the range of $(0, 1)$.
- `eye(p)` generates a $(p \times p)$-identity matrix with ones on the main diagonal and zeros elsewhere.
- `zeros(p)` generates a $(p \times p)$-matrix of zeros.
- `ones(p)` generates a $(p \times p)$-matrix of ones.

3.2 Basic Arithmetic Operations for Matrices

Operators like `+`, `-`, `*` (multiplication), `/` (division) and `^` (exponentiation) can be used to build arithmetical expressions as for conventional programming languages. The operators have the usual precedence and the parentheses have accordingly the highest precedence.

When dealing with vectors and matrices, one has to pay attention that the dimensions of the operands match properly, i.e., for a summation or subtraction, the operands must be of same dimensions or one of them needs to be a scalar. The multiplication of vectors and matrices refers to a *matrix multiplication*.

For a $(m \times p)$–matrix $\boldsymbol{A} = [\, a_{ij} \,]$ (m rows and p columns) and a $(p \times n)$–matrix $\boldsymbol{B} = [\, b_{jk} \,]$, the elements c_{ik} of the product matrix $\boldsymbol{C} = [\, c_{ik} \,] \stackrel{\text{def}}{=} \boldsymbol{AB}$ $(m \times n)$ are defined as $c_{ik} = \sum_{j=1}^{p} a_{ij} b_{jk}$. This calculation rule is illustrated by the sketch of the matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ in Figure 3.1. This illustration also helps to track the dimensions of the involved matrices. The elements c_{ik} are the scalar products of the i-th row of $\boldsymbol{A}$ and the k-th column of $\boldsymbol{B}$; easy to remember as *'row times column'* (element-wise multiplication followed by a summation).

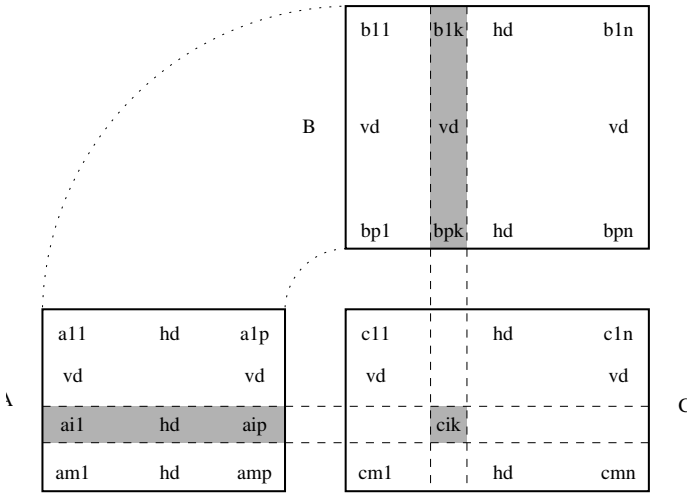


Figure 3.1: Multiplication of matrices.

Preparation:

Exercise P-3.1: The row vector $\boldsymbol{a} = [3 \;\; 2 \;\; 1]$ and column vector $\boldsymbol{b} = [6 \;\; 5 \;\; 4]^{\mathsf{T}}$ are given. Compute

$$\boldsymbol{ab} \qquad \text{and} \qquad \boldsymbol{ba}$$

What are the dimensions of the results?

Practical Experiments:

Exercise E-3.1: Confirm your results of Task **P-3.1**.

The division operation has a special meaning in the context of vectors and matrices: The operators "/" and "\" perform *different* division operations:

`x = A \ b` is the solution of `A * x = b`, where `x` and `b` are column vectors

and

`y = c / A` is a solution of `y * A = c`, where `y` and `c` are row vectors.

Thus, `A\b` computes the expression $\boldsymbol{A}^{-1} \cdot \boldsymbol{b}$ and `c/A` evaluates the expression $\boldsymbol{c} \cdot \boldsymbol{A}^{-1}$. The division operators allows to solve a systems of linear equations directly, i.e., without an explicit matrix inversion. This is advantageous in terms of precision and efficiency (see also the exercise in Sec. 3.3).

Applying the introduced division operators to

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} \qquad \text{and} \qquad \boldsymbol{b} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}, \quad \boldsymbol{c} = \boldsymbol{b}^{\mathsf{T}}$$

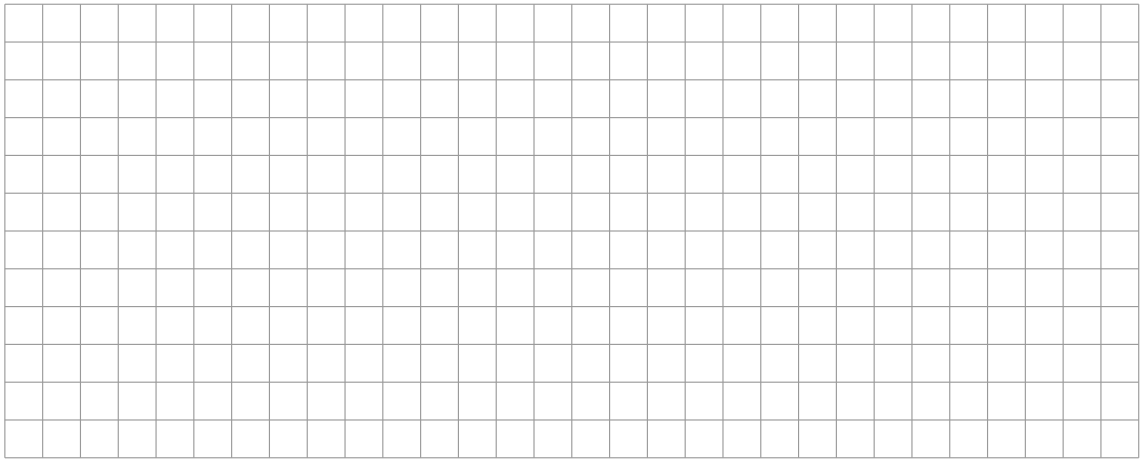
results in

```
>> A\b
ans =
    3.0000
   -1.0000

>> c/A
ans =
     7    -2
```

Preparation:

Exercise P-3.2: Verify the results of the previous example.



The frequently needed *element-wise* multiplication and division of vectors and matrices can be performed by the operators "`.*`" and "`./`", respectively. The operator "`.^`" allows to exponentiate each element of a vector or matrix (see also Sec. 1).

As mentioned before, the transpose of matrices and vectors is obtained by the operator "`.'`" ($\boldsymbol{a}^{\mathsf{T}}$) and the complex conjugate transpose by the operator "`'`" ($\boldsymbol{a}^{\mathsf{H}}$, Hermitian matrix). These operators turn column vectors into row vectors and vice versa:

```
>> a=[1 2 i]
a =
   1.0000            2.0000                 0 + 1.0000i

>> a'
ans =
   1.0000
   2.0000
        0 - 1.0000i

>> a.'
ans =
   1.0000
   2.0000
        0 + 1.0000i
```

or

```
>> A=[1 2 3; 4 5 6]
A =
     1     2     3
     4     5     6

>> A'
ans =
     1     4
     2     5
     3     6
```

Table A.10 and Table A.11 in the Appendix list the most important MATLAB operators for vectors and matrices.

The relational operators >, <, >=, <=, ~= (inequality) and == (equality) perform an element-wise comparison. Hence, they can only be applied if the operands have the same dimensions (as for the summation and subtraction of matrices). The resulting matrix contains ones for the elements where the comparison is true and zeros elsewhere.

MATLAB provides several functions to perform basic arithmetic operations used in linear algebra. A small selection is given in the following:

- `det(A)` returns the determinant of the square matrix $\boldsymbol{A}$.
- `inv(A)` returns the inverse of the square matrix $\boldsymbol{A}$.
- `rank(A)` returns the rank of the matrix $\boldsymbol{A}$, i.e., returns the number of linearly independent rows or columns of $\boldsymbol{A}$.
- `[V,D]=eig(A)` returns the diagonal matrix $\boldsymbol{D}$ with the eigenvalues of $\boldsymbol{A}$ and the matrix $\boldsymbol{V}$ whose columns are the corresponding eigenvectors.
- `norm(A,p)` returns the norm of $\boldsymbol{A}$ (dependent on the choice of p), where $\boldsymbol{A}$ can be a vector or a matrix.

The command `help matlab/elmat` provides an overview of all operations for creating and manipulating matrices.

Practical Experiments:

Exercise E-3.2: Use the MATLAB–command `rand` to generate (4×4)–random matrices `A` and `B`. Calculate the following expressions for the matrices `A1`, `A2`, `A3`, `A4` and check (by calculating the difference) which matrices are identical. Explain the observed result!

- ```
  A1 = A*B                A2 = B*A
  A3 = (A'*B')'           A4 = (B'*A')'
  ```
- ```
  A1 = A'*B'              A2 = (A*B)'
  A3 = B'*A'              A4 = (B*A)'
  ```
- ```
  A1 = inv(A*B)           A2 = inv(A)*inv(B)
  A3 = inv(B*A)           A4 = inv(B)*inv(A)
  ```
  ```

3.3 Systems of Linear Equations and Gaussian Elimination

A system of linear equations with n equations and n unknowns
$x_1, x_2, \ldots, x_n$

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{3.3}$$

can be expressed in the more compact matrix form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{3.4}$$

Such a system is termed as regular, if its coefficient matrix $\boldsymbol{A} = [a_{ij}]$ is invertible. A regular system has a unique solution $\boldsymbol{x} = [x_i]$ which can be found in MATLAB with either the array left division

```
x=A\b
```

or by a multiplication with the inverted matrix

```
x=inv(A)*b
```

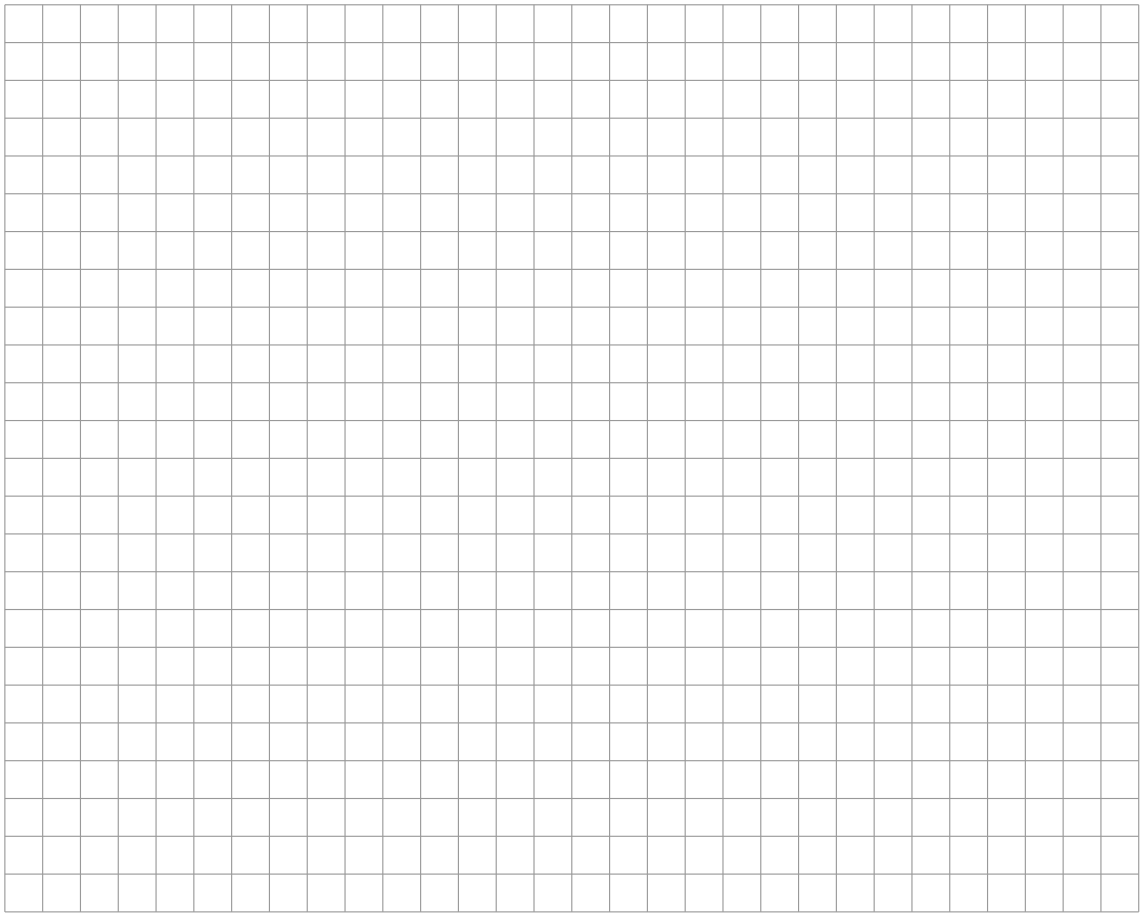
Essentially, both operations achieve the same results but the first method should be preferred since its calculation is more efficient. In the following, the methods to solve a linear system of equations should be explored by looking at a simple version of the Gaussian elimination procedure. First of all, the solution for a system of linear equations should be illustrated graphically.

Preparation:

Exercise P-3.3: Recapitulate the principles of the Gaussian elimination method as known from basic math courses.

Exercise P-3.4: Use this method to find the solutions to the following system of linear equations:

$$\begin{aligned} 2x_1 + 3x_2 + x_3 &= -4 \\ 4x_1 + x_2 + 4x_3 &= 9 \\ 3x_1 + 4x_2 + 6x_3 &= 0 \end{aligned} \tag{3.5}$$



Practical Experiments:

Exercise E-3.3: Consider the following system of linear equations ($a \in \mathbb{R}$)

$$\begin{aligned} 2x_1 + & + x_3 &= -1 \\ \tfrac{1}{4}x_1 + & \tfrac{15}{4}x_2 + x_3 &= -3 \\ (3+6a^2)x_1 + & (1-16a^2)x_2 + x_3 &= 10a - 3 \end{aligned} \tag{3.6}$$

whose solution(s) should now be illustrated graphically .

- Consider only the first equation of the system. Solve it for x_3 and create a 3D–plot of x_1 and x_2, e.g., using

```
>> [x1, x2] = meshgrid( -2:.1:2, -2:.1:2 );
>> x3_Eq1 = ...
>> surf( x1, x2, x3_Eq1, 1*ones(size(x3_Eq1)) ); shading flat;
>> hold on; axis(2*[-1 1 -1 1 -1 1]); caxis([1 3]);
```

  Which geometric figure can you see?

- Consider now the second equation of the system. Solve it for x_3 and plot x_3 into the previously created figure:

```
>> x3_Eq2 = ...
>> surf( x1, x2, x3_Eq2, 2*ones(size(x3_Eq2)) ); shading flat;
```

  Where are the points $[x_1, x_2, x_3]^\mathsf{T}$ that fulfill both equations simultaneously?

- Consider now only the third equation of the system. Solve it for x_3 and add the plot x_3 for $a = 0$ into the previously created figure. Where is the solution of the system of equations?

- Change the value of a in small steps from 0 to 1 and display the result in the 3D-plot from above. What do you observe? Where lies the solution? Which special case is obtained for $a = 1$?

- Change the value of a in small steps from 0 to -1 and display the results for all three equations of the system in a 3D plot. What do you observe? Where lies the solution? Which special case is obtained for $a = -1$?

3.3.1 Implementation of an Elimination Step

For the Gaussian elimination, the system of linear equations of (3.3) is rearranged such that the coefficient a_{ii} does not disappear ($a_{ii} \neq 0$).

By dividing the i-th row with a_{ii}, the coefficient of the i-th row and i-th column is normalized to one. Finally, the i-th row multiplied with $-a_{ji}$ will be added to the j-th row for $j = 1, \ldots, i-1, i+1, \ldots, n$. After these steps, the i-th column is filled with zeros, only the coefficient of the i-th row is 1. The set of equations has now the following form:

$$\begin{array}{ccccccccccc}
a'_{11}x_1 + & \ldots + & a'_{1,i-1}x_{i-1} + & & 0 + & a'_{1,i+1}x_{i+1} + & \ldots + & a'_{1n}x_n & = & b'_1 \\
\vdots & & & & & & & \vdots & & \\
a'_{i1}x_1 + & \ldots + & a'_{i,i-1}x_{i-1} + & & x_i + & a'_{i,i+1}x_{i+1} + & \ldots + & a'_{in}x_n & = & b'_i \\
\vdots & & & & & & & \vdots & & \\
a'_{n1}x_1 + & \ldots + & a'_{n,i-1}x_{i-1} + & & 0 + & a'_{n,i+1}x_{i+1} + & \ldots + & a'_{nn}x_n & = & b'_n
\end{array} \tag{3.7}$$

Practical Experiments:

Exercise E-3.4: Implement a function `[As, bs] = elim(A,b,i)` that performs the elimination step described above for the i-th row/column of the matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$. It is assumed here that $a_{ii} \neq 0$. Proceed as follows:

1. Create a matrix $[\boldsymbol{A}|\boldsymbol{b}]$ that contains the matrix $\boldsymbol{A}$ and the column vector $\boldsymbol{b}$ in its $(n+1)$-th column. .
2. Divide the i-th row of the matrix $[\boldsymbol{A}|\boldsymbol{b}]$ by the element a_{ii}.
3. Add the i-th row multiplied by $-a_{ji}$ to the j-th row ($j = 1, \ldots, n$, $j \neq i$) of the matrix $[\boldsymbol{A}|\boldsymbol{b}]$.
4. This results the new matrix $[\tilde{\boldsymbol{A}}|\tilde{\boldsymbol{b}}]$. Extract the matrix $\tilde{\boldsymbol{A}}$ and the vector $\tilde{\boldsymbol{b}}$ and return them as output of your function `elim`.

Test your function using the matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ of the system of linear equations (Eq. 3.5) mentioned in Exercise **P-3.4** as inputs of `elim`. Vary the column number i.

3.3.2 Elimination through Pivoting

A system of linear equations in the form of Eq. 3.3 can be solved by applying the above treated elimination step successively to the rows/columns from 1 to n. At the end of the elimination process, an identity matrix is obtained on the left hand side and the vector on the right contains the solution of the system of linear equations.

It can also happen that the coefficient in the i-th row and i-th column is equal to zero. The function `elim` encounters a division by zero in this case. This can be avoided by using the so-called *row-pivoting*. In this case, it is recommended to exchange the rows i to n in the i-th elimination step such that the coefficient with the largest magnitude value of the i-th column appears in the i-th row. This procedure is also useful to improve the accuracy of the result.

Practical Experiments:

Exercise E-3.5: Implement the function `x=gausselim(A,b)` that solves the system of linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ by means of the Gaussian elimination method. Proceed as follows:

1. Create a loop with the column index i as counter.

2. Find the row $j = i, \ldots, n$ that contains the element with the largest magnitude a_{ji}. Swap this row with the i-th row. (*Note:* The elements of the column vector $\boldsymbol{b}$ need to be rearranged accordingly).
   If the magnitude of the largest element is below a threshold (e.g., `1e-14`), the system of equations is nearly singular and the function `gausselim` should abort with the MATLAB–command `return`) and the value `NaN` should be returned.

3. Execute the elimination in the i-th column by using the function `elim`.

4. Display the solution $\boldsymbol{x}$ after the n-th step (i.e., the last value of the loop counter).

The following core code is available for creating the function `gausselim`:

```
function x = gausselim(A,b)
%
[m,n] = size(A);
% Loop through all the rows/columns
for i=...,
  % Find the absolute maximum in the i-th column
  % going through rows i,..,m
  [value, index] = ...;
  % Abort if < 1e-14
  ...
  % Swap rows i and index (of both A and b)
  ...
  % Carry out the elimination
  [...] = elim(...);
end;
% Return value
x = ...;
```

Test your function with the system of linear equations introduced in **P-3.4**

Exercise E-3.6: Solve the system of linear equations using `gausselim`

$$\begin{aligned} 2x_1 + 3x_2 - 8x_3 &= 10 \\ -4x_1 + 5x_2 + x_3 &= -8 \\ 5x_1 + x_2 &= 11 \end{aligned} \tag{3.8}$$

Compare the result with that obtained by a matrix left division `x=A\b`.

Exercise E-3.7: Solve the system of linear equations using `gausselim`

$$\begin{aligned} 2x_1 + 3x_2 + 1x_3 &= 1 \\ -4x_1 + 4x_2 + 2x_3 &= 2 \\ 4x_1 + x_2 &= 3 \end{aligned} \tag{3.9}$$

Interpret the result!

Literature to Experiment 3

[1] S.J. Leon: *Linear Algebra with Applications*, 6th edition, Prentice Hall, Englewood Cliffs, New Jersey, 2001.

Chapter 4

Complex Numbers

Complex numbers are a vital mathematical tool for engineers. For example, they are of fundamental importance for solving equations as well as for the analysis of oscillation processes. In electrical engineering, these numbers are needed, among others, for the complex analysis of AC circuits as well as the complex representation of baseband signals.

4.1 Complex Numbers in MATLAB

Computations by MATLAB are done without differentiating between real and complex numbers and all arithmetical operators and standard functions are defined for real and complex numbers alike. The imaginary unit is given in MATLAB by either `i` or `j`. For example, the variable `a` is assigned to a complex number $1 + 2\mathrm{j}$ as follows:

```
>> a=1+2i
a =
   1.0000 + 2.0000i
```

and i^2 is given by

```
>> a=i^2
a =
   -1
```

Alternatively, the symbol j can also be used.[1] MATLAB always uses the symbol `i` to display complex numbers:

```
>> a=1+2j
a =
   1.0000 + 2.0000i
```

The real and imaginary part of a complex number is obtained by the functions `real` and `imag`. For the example of `a=1+2i`, this looks as follows:

```
>> real(a)
ans =
     1

>> imag(a)
ans =
     2
```

Magnitude and phase (in radiant) of a complex number (in its polar form) can be obtained by the functions `abs` and `angle`:

```
>> abs(a)
ans =
    2.2361

>> angle(a)
ans =
    1.1071
```

[1] As common in electrical engineering, the symbol j instead of i is used here to denote the imaginary unit since the variable i is usually used to describe the current.

The complex conjugated number is obtained by the function `conj`:

```
>> conj(a)
ans =
   1.0000 - 2.0000i
```

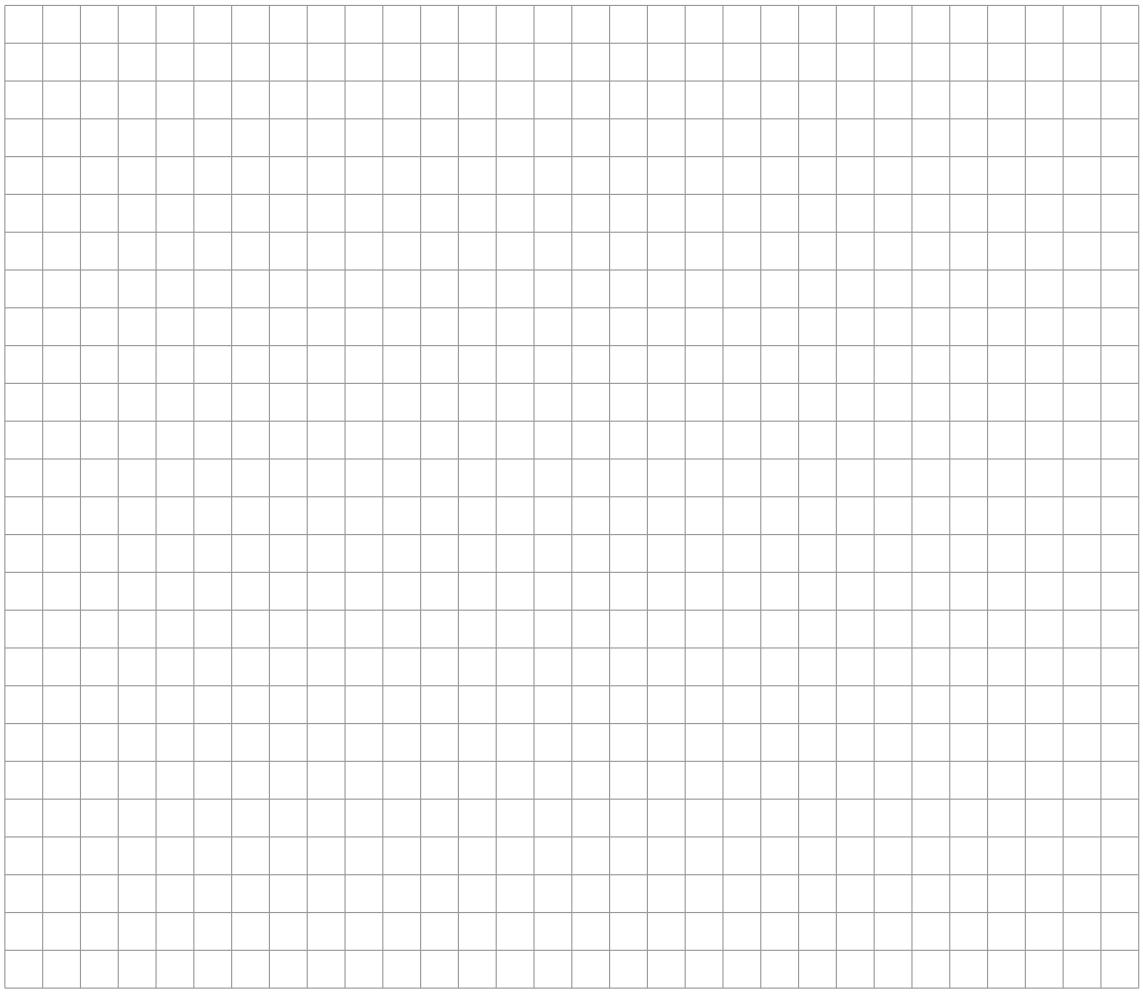
The most important functions related to complex numbers in MATLAB are summarized in Table A.12 in the Appendix.

Preparation:

Exercise P-4.1: Recap the main calculation rules for complex numbers taught in your undergraduate math course.

Let $z_1 = a_1 + \mathrm{j}b_1$ and $z_2 = a_2 + \mathrm{j}b_2$, a_i, $b_i \in \mathbb{R}$, be complex numbers. What are the real and imaginary parts

- of a) $z_1 + z_2$, b) $z_1 - z_2$, c) $z_1 \cdot z_2$ and d) z_1/z_2,
- of the complex conjugate of z_1 (find z_1^*),
- and of $1/z_1^*$?

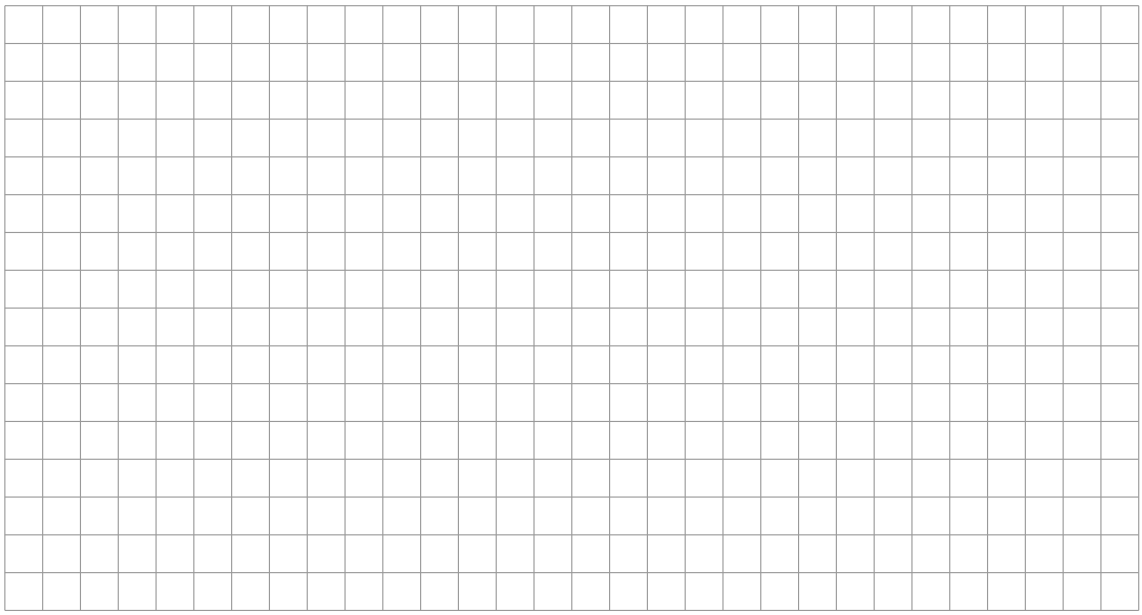


Exercise P-4.2: Determine the real and imaginary part of

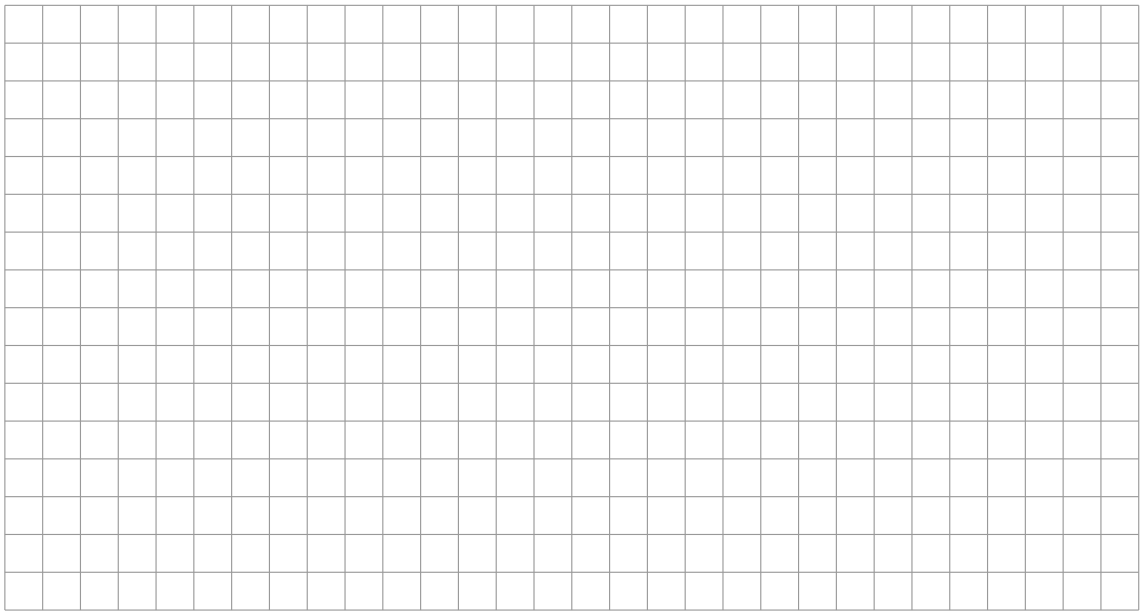
$$z = \frac{1}{\frac{\sqrt{2}}{2} + \mathrm{j}\frac{\sqrt{2}}{2}}$$

Exercise P-4.3: How is the complex number $z = a + \mathrm{j}b$ represented in its polar form? Express z as a complex exponential function (exponential form)!

Write the numbers $z = 1+\mathrm{j}$, $z = \mathrm{j}$ and $z = -1$ in its polar form (exponential form).



Let z be given in a polar form. How is $1/z^*$ represented in a polar form? Interpret the result on the complex plane.



Practical Experiments:

Exercise E-4.1: Calculate with the help of MATLAB the real and imaginary parts of

$$\text{a)}\quad z = \frac{1}{\frac{\sqrt{2}}{2} + \mathrm{j}\frac{\sqrt{2}}{2}} \qquad \text{b)}\quad z = \frac{2+3\mathrm{j}}{4-\mathrm{j}} + \frac{2-3\mathrm{j}}{1-4\mathrm{j}} \qquad \text{c)}\quad z = \mathrm{j}^{\mathrm{j}} \qquad \text{d)}\quad z = \log(-1).$$

Exercise E-4.2: Determine with MATLAB the polar form of the following complex numbers:

$$\text{a)}\quad z = (1-3\mathrm{j})^3 + (1+3\mathrm{j})^3 \qquad \text{b)}\quad z = \frac{(2-\mathrm{j})^7}{|-1+\mathrm{j}| \cdot (-1+\mathrm{j})}$$

$$\text{c)}\quad z = \left(\frac{1}{2} + \mathrm{j}\frac{\sqrt{3}}{2}\right)^{50} \qquad \text{d)}\quad z = 5^{-1+2\mathrm{j}} + (-1+2\mathrm{j})^5.$$

Exercise E-4.3: Create a MATLAB–function that converts complex numbers from a Cartesian form to a polar form and vice versa. The input argument for this function shall be a vector with two numerical values, which are either the real and imaginary part or the magnitude and phase of the complex number, and a distinguishing letter `'C'` (Cartesian form) or `'P'` (polar form). The output of the function is the converted complex number.

If, for example, the complex number $z = 1 + \mathrm{j}$ should be converted to its polar form the function call could be as follows:

```
>> complex( [1, 1], 'C' )
ans =
    1.4142    0.7854
```

Test your function with the help of the following examples and explain the results:

$$z = 1 + \mathrm{j}, \qquad z = -5\mathrm{j}, \qquad z = 3\mathrm{e}^{\mathrm{j}3\pi/4}, \qquad z = \sqrt{2}\mathrm{e}^{\mathrm{j}5\pi/4}.$$

What is the result of your function for a call `complex(complex([1 p], 'P'), 'C')`, where p takes values between 0 and 2π? Explain the result!

Exercise E-4.4: Consider the function

$$f(x) = \frac{\mathrm{e}^{\mathrm{j}x} + \mathrm{e}^{-\mathrm{j}x}}{2}\,, \qquad x \in \mathbb{R}.$$

Plot the real and imaginary part of $f(x)$ over x! What do you observe? Which function is given by $f(x)$?
Hint: Which part of a complex number z is in general given by $(z + z^*)/2$?

4.2 Visualization of Complex Numbers

Complex numbers can be graphically represented in the *complex plane*. The function `cplot` (which is only available in the CIP–Pool for this lab course) visualizes complex numbers as pointers in the complex plane.

Practical Experiments:

Exercise E-4.5: Enter the following MATLAB–commands and explain the results:

```
>> a = 1+2i;
>> b = 1+i;
>> c = a+b;
>> plot( 0,0 ); axis([-3 3 -3 3]); grid; axis square; hold on;
>> cplot( a, 'r' );
>> cplot( b, 'b' );
>> cplot( c, 'g' );
```

Repeat this for

- `c = a-b;`
- `c = a*b;`
- `c = a/b;`
- `c = conj(a);`
- `c = a*conj(a);`
- `c = a*conj(b);`
- `c = 1/conj(b);`
- `c = (a+conj(a))/2;`
- `c = (a-conj(a))/2;`

Preparation:

Exercise P-4.4: State all complex solutions of the following equation:

$$z^3 = -2 + 2\mathrm{j} \qquad (\textit{Hint: } -2 + 2\mathrm{j} = \sqrt{2}^3 \mathrm{e}^{\mathrm{j}3\pi/4})$$

Practical Experiments:

Exercise E-4.6: Represent all the solutions determined in **P-4.4** graphically! Pay attention to the fact that MATLAB only returns one possible solution for the roots.

Moreover, give the solutions for the following equations:

$$\text{a)} \quad z^2 = \mathrm{j} \qquad \text{b)} \quad z^4 = 1 + 2\mathrm{j} \qquad \text{c)} \quad z^6 = -1$$

If a complex function of a real variable (e.g., time) should be represented, a three-dimensional representation is needed. This can be done with the help of the function `plot3(x,y,z)` which displays a three-dimensional plot for a set of data points given by the vectors `x`, `y` and `z`.

Preparation:

Exercise P-4.5: Figure 4.1 shows a helix that can be described by its function values (y and z values) in dependence of time t (x values).

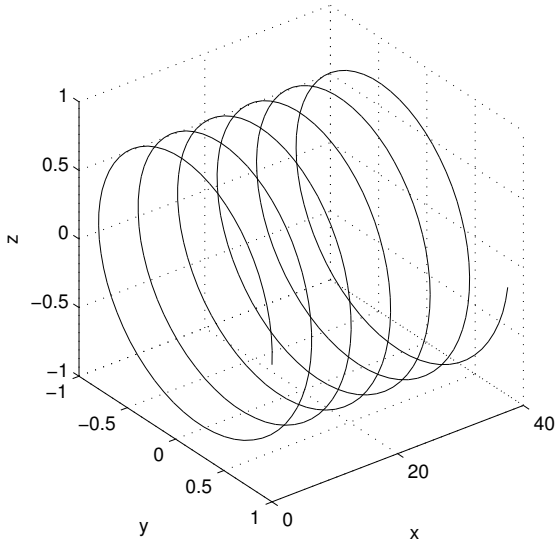


Figure 4.1: Example of a helix function.

Write down the real coordinate functions $y(t)$ and $z(t)$ of the helix in dependence of time $t = x$, $t \in [0,\, 12\pi]$. (*Hint:* The pitch of the helix is $\Delta t = 2\pi$.)

Consider now the y-coordinate as the real part and the z-coordinate as the imaginary part of a complex number $c(t)$ that depends on the time t (x-coordinate). Write down the resulting function $c(t)$ in a compact form! Which mathematical function describes the helix that is depicted on Figure 4.1?

Exercise P-4.6: Consider the following complex function that depends on a (real) time variable t and another real parameter $d \geq 0$:

$$c(t) = \mathrm{e}^{(\mathrm{j}-d)t}\,, \qquad d \geq 0$$

How do you expect the graph to look if (as seen in Figure 4.1) the time runs along the x-axis, and the real and imaginary parts of the function run along the y- and z-axis, respectively?

Practical Experiments:

Exercise E-4.7: Create a plot of the helix as seen in Figure 4.1 by using the function `plot3`. Pay attention to the proper scaling of the axes (for which the command `axis ij` can be useful).

After that, use the button  in order to vary the viewing angle of the plot. Especially, the following views are of interest

- x-y-plane (real part over time),
- x-z-plane (imaginary part over time) and
- y-z-plane (imaginary part over real part).

Explain the different representations!

Exercise E-4.8: Represent the complex function $c(t)$ from Exercise **P-4.6** with various parameters $0 \leq d \leq 1$. How can the parameter d be interpreted? What happens if d takes on a value $d < 0$ contrary to its definition?

4.3 Complex Analysis of AC Circuits

Now we will have a look at another example that exemplifies the use of complex numbers. For this purpose, the course of an *impedance* of an AC current in dependence on the frequency is analyzed. In *AC circuit analysis*, sine- or cosine-shaped current and voltage waveforms are easily represented with the help of complex numbers. This allows for an elegant representation of both, oscillation amplitude/RMS value (magnitude value of the complex number), and also the phase (phase of the complex number). Formally, the oscillation frequency f_0, the RMS-value V_{rms} and the phase φ

$$\sqrt{2}V_{\mathrm{rms}} \cdot \cos(2\pi f_0 t + \varphi) = \sqrt{2}V_{\mathrm{rms}} \cdot \mathrm{Re}\{\mathrm{e}^{\mathrm{j}(2\pi f_0 t+\varphi)}\} = \sqrt{2} \cdot \mathrm{Re}\{V_{\mathrm{rms}}\,\mathrm{e}^{\mathrm{j}\varphi} \cdot \mathrm{e}^{\mathrm{j}2\pi f_0 t}\} \qquad (4.1)$$

are represented by the complex pointer (phasor) $V = V_{\mathrm{rms}}\,\mathrm{e}^{\mathrm{j}\varphi}$.

Similar to Ohm's law, the complex ratio of the voltage and the current is specified as *complex resistance* or *impedance*, i.e.,

$$Z = \frac{U}{I}, \quad \text{where } Z \text{ is a complex number.} \qquad (4.2)$$

The complex resistance Z does not only describe the ratio of amplitudes (via $|Z|$), but also the phase shift between current and voltage (via $\arg(Z)$). An ideal resistor has the (real valued) impedance that is equal to its resistance value $Z_{\mathrm{R}} = R$, whereas an ideal capacitor with capacity C has an (frequency-dependent) impedance of $Z_{\mathrm{C}} = \frac{1}{\mathrm{j}\omega C}$, and an ideal inductor with inductance L has an impedance of $Z_{\mathrm{L}} = \mathrm{j}\omega L$. Thereby, $\omega = 2\pi f_0$ is the angular frequency of the voltages and currents.

Networks of (common) impedances are analyzed in the same way as networks of resistors, with the only difference that complex impedance values must be used instead of real-valued resistance values.

Practical Experiments:

Exercise E-4.9: A parallel circuit consisting of a resistor $R = 1000\ \Omega$ and a capacity $C = 2\ \mu\mathrm{F}$ is considered.

Represent the real and imaginary part of the parallel circuit over its angular frequency ω in a 3D–plot with ω covering a range $[0\ \mathrm{s}^{-1},\ 10^4\ \mathrm{s}^{-1}]$. Rotate the diagram

such that you can follow the course of the imaginary part over the real part of the signal (locuse). How would you interpret the behavior of this impedance?

Repeat this procedure for a parallel circuit with an inductance of $L = 0,1\ \mathrm{H}$ and a capacity of $C = 2\ \mu\mathrm{F}$.

Exercise E-4.10: Now a series circuit of a resistance $R = 100\ \Omega$ with capacity $C = 400\ \mathrm{nF}$ and an inductance $L = 0,1\ \mathrm{H}$ is considered. The serial RLC circuit is excited with an AC voltage having an amplitude of $1\ \mathrm{V}$ and an angular frequency ω. We are interested in the voltage and capacity.

Display the pointer (real and imaginary part) of the voltage over the capacitor over the angular frequency in a 3D–plot for the same range of frequencies as before. Moreover, create a 2D–plot showing the magnitude and phase of the voltage over the angular frequency. What kind of behavior do you observe?

Create another graph by using `cplot` which shows the complex pointers for the overall voltage as well as the voltage for the resistance and the capacity for a given angular frequency. (Select suitable steps for the angular frequency between $3000\ \mathrm{s}^{-1}$ and $7000\ \mathrm{s}^{-1}$). Interpret the result!

Chapter 5

Fourier Transform

The Fourier transform, which is used to convert a time-dependent signal to a frequency-dependent signal, is one of the most important mathematical tools in signal processing. Applying the Fourier transform to local sections of a time-dependent signal (e.g., an audio signal), one obtains the short-time Fourier transform (STFT). In this course, we study a discrete version of the STFT. To work with the discrete STFT in practice, one needs to correctly interpret the discrete time and frequency parameters. Using basic MATLAB functions, we generate discrete-time signals, compute a discrete version of the Fourier transform and the STFT, and visualize the various representations. Furthermore, we study how to convert the discrete time and frequency axes to obtain physically meaning units given in seconds and Hertz.

5.1 Discrete Signal

When using digital technology, only a finite number of parameters can be stored and processed. To this end, analog signals need to be converted into finite representations—a process commonly referred to as *digitization*. One step that is often applied in an analog-to-digital conversion is known as *equidistant sampling*. Given an analog signal $f : \mathbb{R} \to \mathbb{R}$ and a positive real number $T > 0$, one defines a function $x : \mathbb{Z} \to \mathbb{R}$ by

setting

$$x(n) := f(n \cdot T) \tag{5.1}$$

for $n \in \mathbb{Z}$. Since x is only defined on a discrete set of time points, it is also referred to as a *discrete-time* (DT) signal. The value $x(n)$ for some $n \in \mathbb{Z}$ is called a *sample* taken at time $t = n \cdot T$ of the original analog signal f. This procedure is also known as T*-sampling*, where the number T is referred to as the *sampling period*. The inverse

$$F_{\mathrm{s}} := 1/T \tag{5.2}$$

of the sampling period is also called the *sampling rate* of the process. It specifies the number of samples per second and is measured in Hertz (Hz).

Practical Experiments:

Exercise E-5.1: Let $[0, D] := \{t \in \mathbb{R} \colon 0 \leq t \leq D\}$ be a time interval of duration $D \in \mathbb{R}_{>0}$ (given in seconds). Furthermore, let us consider the analog signal $f \colon \mathbb{R} \to \mathbb{R}$ defined by

$$f(t) := A \sin(2\pi(\omega t - \varphi)) \tag{5.3}$$

for $t \in [0, D]$. This signal is a *sinusoid*, where the parameter A corresponds to the *amplitude*, the parameter ω to the *frequency* (measured in Hz), and the parameter φ to the *phase* (measured in normalized radians with 1 corresponding to an angle of $360°$). Write a MATLAB script to do the following:

- Based on a sampling rate of $F_{\mathrm{s}} = 22050\ \mathrm{Hz}$, generate a DT-signal x by sampling f, where the following parameters are to be used: $A = 0.5$, $\varphi = 0$, $\omega = 440$, $D = 2$.
- Plot the signal x once using a time axis based on samples and once using a time axis based on seconds. Explain what you see in the plots.
- Use the command `xlim` to plot the signal only for the time interval $[0.05, 0.06]$ (given in seconds).
- Save the resulting plots as an image file in `png` format using the MATLAB command `print`.

- Familiarize yourself with the MATLAB command `sound` and use it to listen to the signal x. Explain what happens if you do not specify the the sampling rate F_{s}.

5.2 Practical Issues

In the above definition, we allowed a DT-signal to have infinite length (by considering samples indexed by $n \in \mathbb{Z}$). In practice, one typically considers only a finite number of samples $x(n)$ for $n \in [0 : L-1]$ for some parameter $L \in \mathbb{N}$ (often assuming that all other samples are zero). The parameter L is also called the *length* of the signal x.

Using a programming language such as MATLAB, a DT-signal is usually represented as a *vector*, where each entry of the vector corresponds to one sample. Note that there are two kinds of vectors in MATLAB, namely *row* and *column* vectors. Often, it does not matter whether one represents a DT-signal as a row or a column vector. However, for certain MATLAB operations, this may become important—often a confusion between row and column vectors leads to fatal errors. In this lab, to avoid confusions, we always represent a DT-signal as a row vector. Note that it is easy to transform a column vector into a row vector by applying a *transposition*. For example, in MATLAB, the vector `x.'` is the transposed version of the vector `x`.

There is another issue that is a frequent source of confusion. The indexing schemes used in theory and in programming languages such as MATLAB often differs. For example, let us consider the DT-signal $x : \mathbb{Z} \to \mathbb{R}$ defined by

$$x(n) := \begin{cases} 3 & \text{for } n = 0 \\ 2 & \text{for } n = 1 \\ 1 & \text{for } n = 2 \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

for $n \in \mathbb{Z}$. When representing this signal in MATLAB, one typically defines a vector

```
x = [3 2 1];
```

Note that in this case the first entry `x(1)` of the vector corresponds to the sample $x(0)$. In other words, the index of the vector differs from the actual argument of the signal. As a second example, let us consider the *autocorrelation* R_{xx} of x defined by

$$R_{xx}(n) = \sum_{k=-\infty}^{\infty} x(k) \cdot x(k-n) \tag{5.5}$$

for $k \in \mathbb{Z}$. Using the values from (5.4), we obtain

$$R_{xx}(n) = \begin{cases} 3 & \text{for } n = -2 \\ 8 & \text{for } n = -1 \\ 14 & \text{for } n = 0 \\ 8 & \text{for } n = 1 \\ 3 & \text{for } n = 2 \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

for $n \in \mathbb{Z}$. When computing the autocorrelation in MATLAB of the vector `x` (using the command `xcorr`), we obtain

```
>> Rxx = xcorr(x,x); disp(Rxx);
     3     8    14     8     3
```

Note that, in this case, the first entry `Rxx(1)` of the result vector `Rxx` corresponds to $R_{xx}(-2)$. In most programming languages, vectors have a fixed indexing scheme. In MATLAB, a vector `x` is always indexed starting with the index `1`. As shown above, this may not correspond to the indexing scheme as in the mathematical definitions. To account for this fact, it is useful to define an additional *index vector* along with a DT-signal that encodes the relation between the "mathematical" and the"practical" world. For the above cross correlation example, we can define

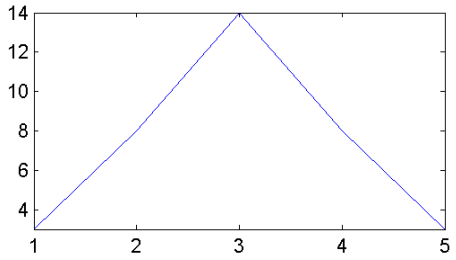
```
index_Rxx = [-2 -1 0 1 2];
```

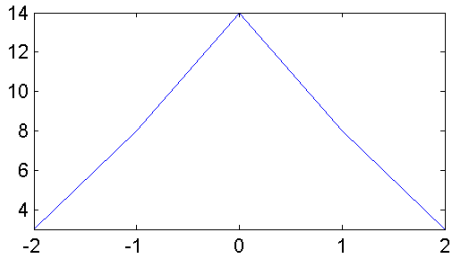
The index vector `index_Rxx` indicates that `Rxx(1)` corresponds to R_{xx}(`index_Rxx(1)`), and so on. Such index vectors are also useful when it comes to plotting DT-signals. As an example, compare the two plots obtained by the following MATLAB code:

```
x = [3 2 1]
Rxx = xcorr(x,x);
index_Rxx = [-2 -1 0 1 2];

figure;
plot(Rxx);
axis tight;
set(gcf,'PaperPosition',[0 0 10 5]);
print('FT_Indexing_1','-dpng');

figure;
plot(index_Rxx,Rxx);
axis tight;
set(gcf,'PaperPosition',[0 0 10 5]);
print('FT_Indexing_2','-dpng');
```





5.3 Discrete Fourier Transform (DFT)

Audio signals can be complex mixtures consisting of a multitude of different sound components. A first step in better understanding a given signal is to decompose it into building blocks that are better accessible for the subsequent processing steps. In the case that these building blocks consist of exponential functions, such a process is also called Fourier analysis. The Fourier transform maps a time-dependent signal to a frequency-dependent function which reveals the spectrum of frequency components of the original signal. Loosely speaking, a signal and its Fourier transform are two sides

of the same coin. On the one side, the signal displays the time information and hides the information about frequencies. On the other side, the Fourier transform reveals information about frequencies and hides the time information.

For discrete time signals, one often computes the Fourier transform only for a finite number of samples $x(0), x(1), \ldots, x(N-1)$ for some suitable number $N \in \mathbb{N}$. Furthermore, similar to the sampling of the time axis, one also samples the frequency axis only considering a finite number of frequencies. This leads to a discrete version of the Fourier transform also referred to as *discrete Fourier transform* (DFT). The DFT of the samples $x(0), x(1), \ldots, x(N-1)$ yields Fourier coefficients $X(k)$ for $k \in [0:K] := \{0, 1, \ldots, K\}$ with $K := N/2$ (assuming that N is even), which are defined by

$$X(k) = \sum_{n=0}^{N-1} x(n) \exp(-\mathrm{j}\, 2\, \pi k\, n/N). \tag{5.7}$$

Given the sampling rate F_{s} of the DT-signal x, the index k of the coefficient $X(k)$ corresponds to the physical frequency

$$F_{\mathrm{coef}}(k) := \frac{k \cdot F_{\mathrm{s}}}{N} \tag{5.8}$$

given in Hertz. Note that the largest index $K = N/2$ corresponds to the *Nyquist frequency*, which is half of the sampling rate F_{s} of the discrete signal.

Practical Experiments:

Exercise E-5.2: Write a MATLAB script that determines the physical frequencies $F_{\mathrm{coef}}(k)$ (given in Hertz) of the DFT coefficient $X(k)$ for $k \in [0:K]$ as well as the Nyquist frequencies when using the following parameter settings:

(A) $F_{\mathrm{s}} = 22050$, $N = 1024$

(B) $F_{\mathrm{s}} = 48000$, $N = 1024$

(C) $F_{\mathrm{s}} = 4000$, $N = 4096$

To this end, compute three vectors `FcoefA`, `FcoefB`, and `FcoefC` (for the three parameter settings), where each vector contains the frequency values (in Hertz) of

the frequency indices for the respective setting. Then, plot the vectors in one figure using an index–frequency representation. To this end, use the MATLAB command `hold on` and use a different color for each of the vector's representation. Explain the figure in words.

Hint: In the above indexing of the frequency values, the indexing starts with the index $k = 0$. When programming in MATLAB, indexing typically starts with the index 1. Be aware of this fact (see also Section 5.2).

The DFT can be computed efficiently by using an algorithm known as the *fast Fourier transform* (FFT). The FFT algorithm, which was discovered by Gauss and Fourier two hundred years ago, has changed whole industries and is now being used in billions of telecommunication and other devices. The FFT exploits redundancies across sinusoids of different frequencies to jointly compute all Fourier coefficients by a recursion. This recursion works particularly well in the case that N is a power of two. As a result, the FFT reduces the overall number of operations from the order of N^2 to the order of $N \log_2 N$.

Practical Experiments:

Exercise E-5.3: In this task, we experiment with the FFT in MATLAB. First, based on (5.3) and using a sampling rate $F_\mathrm{s} = 22050$, generate the following two DT-signals x and y (see Task 1):

- signal x with $A = 1$, $\varphi = 0$, $\omega = 1$, $D = 5$
- signal y with $A = 0.7$, $\varphi = 0$, $\omega = 5$, $D = 5$

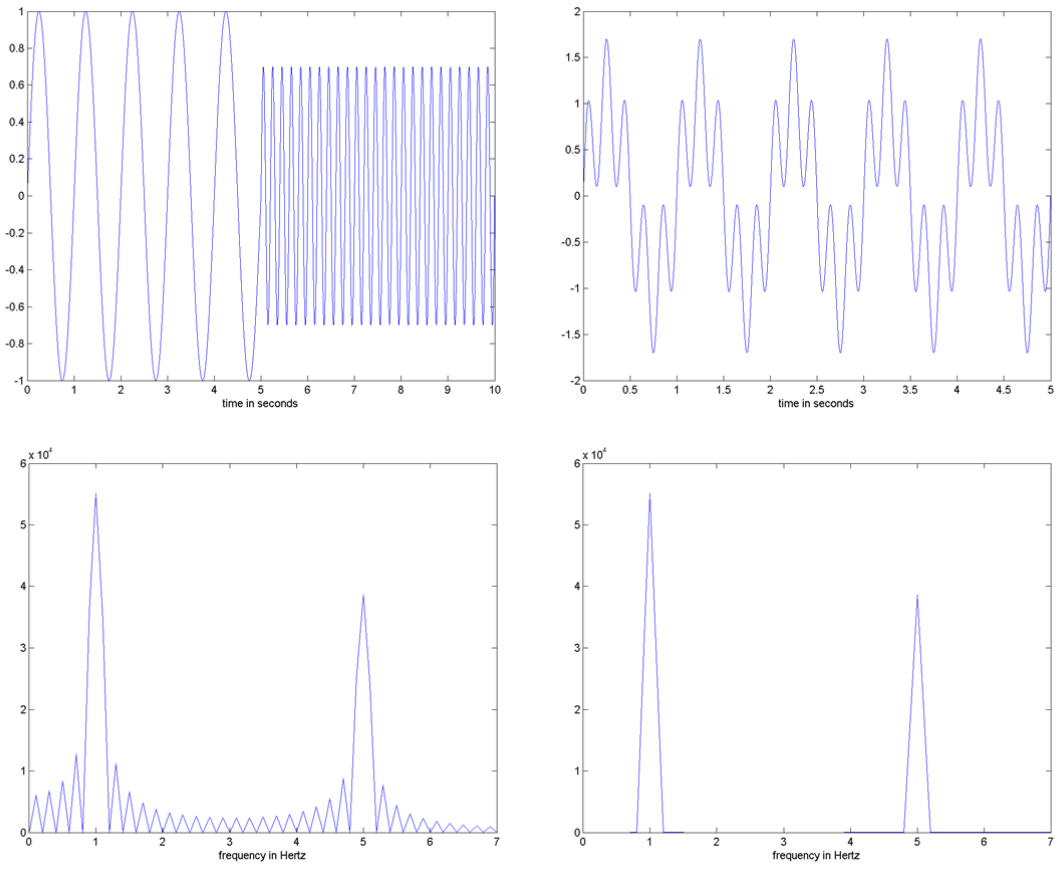
Next, create the DT-signals u and v as follows:

- signal u by concatenating x and y (i.e., `v=[x y]`)
- signal v by superimposing x and y (i.e., `v=x+y`)

Plot both signals on a time axis based on seconds.

Now, we compute the Fourier transform of the signals u and v. Familiarize yourself with the MATLAB command `fft` and use it to compute the Fourier transforms of u and v, respectively. Plot the two resulting spectra using a frequency axis given in Hertz. To compute the frequency axis, you need (5.8) (see also Task 2). Note that applying the FFT yields a complex-valued vector of length N. (What is N for u and v, respectively?) We are only interested in the first $K + 1$ entries of this vector. (Why?) When visualizing a frequency spectrum, we only consider the magnitude of the complex-valued Fourier coefficients. To get the magnitude spectrum from a complex spectrum, you can use the MATLAB function `abs`.

Plot only the frequency range from zero to seven Hertz of the two spectra using the MATLAB command `xlim`. As result, your plots should look like as follows:



What are your observations when comparing the two spectra? Explain your find-

ings in words.

Practical Experiments:

Exercise E-5.4: In this task, you are given a rather long audio signal (around eight minutes) which is zero almost everywhere (silence). At some point in the signal, there is a very short tone of a certain frequency. Your task is to estimate both the time of occurrence (in seconds) as well as the frequency of this tone (in Hertz). You can load the signal in MATLAB by calling `[x,Fs]=wavread('FT_HiddenTone.wav')`.

5.4 Short-Time Fourier Transform (STFT)

As mentioned before, the Fourier transform reveals the frequency content of a signal while hiding the time information. To obtain back the hidden time information, Dennis Gabor introduced in the year 1946 the modified Fourier transform, now known as *short-time Fourier transform* (STFT). This transform is a compromise between a time- and a frequency-based representation by determining the sinusoidal frequency and phase content of local sections of a signal as it changes over time. In this way, the STFT does not only tell which frequencies are "contained" in the signal but also at which points of time or, to be more precise, in which time intervals these frequencies appear.

We now review the definition of the discrete STFT. Let $x : \mathbb{Z} \to \mathbb{R}$ be a real-valued discrete signal obtained by equidistant sampling with respect to a fixed sampling rate F_{s} given in Hertz. Furthermore, let $w : [0 : N-1] := \{0, 1, \ldots, N-1\} \to \mathbb{R}$ be a discrete-time window of length $N \in \mathbb{N}$ and let $H \in \mathbb{N}$ be a hop size parameter. With regard to these parameters, the discrete STFT $\mathcal{X}$ of the signal x is given by

$$\mathcal{X}(m,k) := \sum_{n=0}^{N-1} x(n+mH)w(n)\exp(-\mathrm{j}\,2\,\pi\,k\,n/N) \tag{5.9}$$

with $m \in \mathbb{Z}$ and $k \in [0 : K]$ and $K = N/2$ (assuming that N is even). The complex number $\mathcal{X}(m,k)$ denotes the k^{th} Fourier coefficient for the m^{th} time frame. Each Fourier

coefficient $\mathcal{X}(m,k)$ is associated with the physical time position

$$T_{\text{coef}}(m) := \frac{m \cdot H}{F_{\text{s}}} \tag{5.10}$$

given in seconds and with the physical frequency

$$F_{\text{coef}}(k) := \frac{k \cdot F_{\text{s}}}{N} \tag{5.11}$$

given in Hertz (Hz). For example, using $F_{\text{s}} = 44100$ Hz as for a CD recording, a window length of $N = 4096$, and a hop size of $H = N/2$, we obtain a time resolution of $H/F_{\text{s}} \approx 46.4$ ms and a frequency resolution of $F_{\text{s}}/N \approx 10.8$ Hz.

Practical Experiments:

Exercise E-5.5: Using $F_{\text{s}} = 22050$, $N = 2048$ and $H = 1024$, write a short MATLAB script to compute the physical meaning of the Fourier coefficients $\mathcal{X}(1000, 1000)$, $\mathcal{X}(17, 0)$, and $\mathcal{X}(56, 1024)$?

When computing the STFT, there are many choices for the window function w. For example, one often uses the so-called *Hamming window* of length N, which is defined by

$$w(n) = 0.56 - 0.46 \cdot \cos\left(\frac{2\pi n}{N-1}\right) \tag{5.12}$$

for $n \in [0 : N-1]$. In the following, we assume that the DT-signal x has finite length $L \in \mathbb{N}$ only considering the samples $x(n)$ for $n \in [0 : L-1]$. Furthermore, let

$$x_m(n) := \begin{cases} x(n+mH) & \text{for } n \in [0 : N-1], \\ 0 & \text{otherwise.} \end{cases} \tag{5.13}$$

denote the m^{th} *frame* of the signal. Note that the m^{th} frame is only defined for $m \in [0 : M-1]$ for

$$M := \lfloor (L-N)/H \rfloor + 1 \tag{5.14}$$

(Why? Give an explanation!), where $\lfloor \cdot \rfloor$ denotes the floor operator, which yields the largest integer that is smaller or equal to its argument. Furthermore, one often uses the *window overlap* O (instead of a hop size H), where

$$O := N - H. \tag{5.15}$$

Based on this notation, the STFT of a signal x of length L can be computed by the following procedure:

Algorithm: STFT

Input: Samples $x(n)$ for $n \in [0 : L-1]$
Window function w of length N (assuming N is even)
Window overlap O

Output: STFT coefficients $\mathcal{X}(m,k)$ for $m \in [0 : M-1]$ and $k \in [0 : K]$

Procedure: Compute $H = N - O$, $M = \lfloor (L-N)/H \rfloor + 1$, and $K = N/2$. Then perform in a loop for $m = 0, \ldots, M-1$ the following steps:

(1) Compute the frame x_m.
(2) Compute the pointwise product $w \cdot x_m$.
(3) Compute the DFT of the pointwise product $w \cdot x_m$.
(4) Consider the first $K+1$ entries of the result to define $\mathcal{X}(m,k)$ for $k \in [0 : K]$.

Practical Experiments:

Exercise E-5.6: Following the above procedure, write a MATLAB function `X=FT_STFT(x,w,O)`. Given a vector `x` that encodes a DT-signal x of length L, a vector `w` that encodes a window function of length N and a window overlap `O`, your function should compute a matrix `X` that encodes the STFT coefficients $\mathcal{X}$.
Hint: Again be aware of indexing issues (see Section 5.2). The MATLAB function should output a $((K+1) \times M)$ matrix `X`, where the first column of `X` (having length $K+1$) corresponds the Fourier coefficients $\mathcal{X}(m,k)$ for $m = 0$ and $k \in [0 : K]$.

The STFT is often visualized by means of a *spectrogram*, which is a two-dimensional representation of the squared magnitude:

$$\mathcal{Y}(m,k) = |X(m,k)|^2. \tag{5.16}$$

When generating an image of a spectrogram, the horizontal axis represents time, the vertical axis is frequency, and the spectrogram value at a particular time–frequency point is represented by the intensity or color in the image.

Practical Experiments:

Exercise E-5.7: In this task, we test the STFT implementation from Task 6.

- Use the function `wavread` to read the file `FT_TwoSineTwoImpulse.wav`. This defines a vector `x` that represents the audio signal as well as the sampling rate `Fs`. In the case that the signal is stereo, only use the first channel. Following our conventions, make sure that the DT-signal x is represent as a row vector `x` (possibly by applying the MATLAB transposition operator).
- Listen to `x` by using the `sound` command.
- Initialize a window length parameter `N=1024`, a hop size parameter `H=256` and derive the window overlap `O`.
- Compute a vector `w` that represents a Hamming window function of length `N` as defined in (5.12).
- Compute the matrix `X` encoding the discrete STFT $\mathcal{X}$ using the function `FT_STFT` from Task 6.
- Alternatively, you can also compute `X` using the predefined MATLAB function `X=spectrogram(x,w,O)`.
- Compute a matrix `Y` that encodes the spectrogram $\mathcal{Y}(m,k)$ as in (5.16).
- Using (5.10), compute a vector `Tcoeff` that contains the physical time positions (in seconds) of the time indices.
- Using (5.11), compute a vector `Fcoeff` that contains the frequency values (in Hertz) of the frequency indices.
- Visualize the spectrogram in various ways using the functions `image`, `imagesc`, `axis xy`, `colorbar`, and so on. Doing so, use the various visualization parameters and tools of MATLAB treated in the previous experiments.

- Plot the spectrogram using a time and a frequency axis based on indices.
- Plot the spectrogram using a time and a frequency axis given in seconds and Hertz, respectively. This can be done by applying the functions `image` or `imagesc` using `Tcoeff` and `Fcoeff` as additional parameters.
- Next, use a logarithmic decibel-scale for visualizing the values $\mathcal{Y}(m,k)$. (Recall that, given a value $v \in \mathbb{R}$, the decibel value is $10\log_{10}(v)$.) When plotting the spectrogram, restrict the visible frequency range from 300 Hz to 1000 Hz using the MATLAB command `ylim`.
- Compute a spectrogram of the same recording using `N=8192` and `H=2048`. Note, that you also need to recompute `O` and `w`! Again, restrict the plot of the spectrogram to the frequency range from 300 Hz to 1000 Hz. Discuss the trade-off between time resolution and frequency resolution.
- Try out other audio files.

Literature to Experiment 5

[1] Meinard Müller. *Fundamentals of Music Processing*. Springer Verlag, 2015.

Appendix A

MATLAB–Reference

In the following some important commands, operations, characters etc. are categorized and summarized.

Table A.1: Data input and output in Command Window.

Command	Meaning
`clc`	Clear Command Window
`disp`	Display value of a variable
`input`	Request user input
`pause`	Halt execution temporarily
`diary`	creates a log of keyboard input and the resulting text output (saves Command Window text to file)

Table A.2: Special characters in MATLAB.

Character (sequence)	Meaning
`( )`	Parentheses are used to enclose arguments of functions
`[ ]`	Brackets are used to form vectors and matrices
`.`	Decimal point
`...`	Continuation (often used to continue the current function on the next line)
`,`	Used to separate matrix subscripts and function arguments
`;`	Used inside brackets to end rows of a matrix
`;`	Suppress printing on the display
`%`	Denotes a comment; indicates a logical end of line
`=`	Used in assignment statements

Table A.3: System Commands.

Command	Meaning
`quit, exit`	Terminate MATLAB
`cd`	Change current folder
`pwd`	Identify current folder (print working directory)
`dir`	List current folder contents
`..`	Change to parent folder
`delete`	Delete files or objects
`path`	View or change search path

Table A.4: Workspace operations.

Command	Meaning
`who`	List variables in workspace
`whos`	List variables in workspace, with sizes and types
`clear`	Remove variables from current workspace
`load`	Load variables from file into workspace (`mat`–file in special MATLAB–format)
`load -ascii`	Treats filename as an ASCII file, load data from file
`save`	Save Workspace variables to a `mat`–file
`save -ascii`	Save data to an ASCII file
`save -append`	Append a variable to an existing (`mat`–file

Table A.5: Help functions.

Command	Meaning
`help`	Online Help
`help fun`	Help text for the functionality specified by `fun`
`help general`	Help text for general commands
`help ops`	Help text for operators
`help elmat`	Help text for matrices and their manipulation
`help elfun`	Help text for elementary functions
`help matfun`	Help text for special functions in linear algebra
`helpwin`	Online–Help in a separate Help Window
`helpdesk`	Detailed Hypertext–Documentation
`demo`	MATLAB Demo–Programs

Table A.6: Special Variables and Constants.

Command	Meaning
`ans`	Default name for most recent answer
`eps`	Floating-point relative accuracy, `eps = 2.2204e-16`
`realmax`	Largest positive floating-point number, `realmax = 1.7977e308`
`realmin`	Smallest positive normalized floating-point number, `realmin = 2.2251e-308`
`pi`	Floating-point number nearest the value of π
`i, j`	$\mathrm{j} = \sqrt{-1}$ (imaginary unit)
`Inf`	Infinity
`NaN`	Not a Number

Table A.7: Arithmetic, relational and logical operators.

Command	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`a^b`	a^b
`>`	Greater than
`<`	Less than
`==`	Equal to
`>=`	Greater than or equal to
`<=`	Less than or equal to
`~=`	Not equal to
`&`	Logical AND
`|`	Logical OR

Table A.8: Standard Functions.

Command	Meaning
`sin`	Sine
`cos`	Cosine
`tan`	Tangent
`cot`	Cotangent
`exp`	Exponential function
`log`	Natural logarithm
`log10`	Common logarithm (base 10)
`sqrt`	Square root
`fix`	Round toward zero
`floor`	Round toward negative infinity
`ceil`	Round toward positive infinity
`round`	Round to nearest decimal or integer
`sign`	Indicate sign of input

Table A.9: Operations with matrices (column by column).

Command	Meaning
`min`	Minimum
`max`	Maximum
`sum`	Sum of array elements
`prod`	Product of array elements

Table A.10: Elementary matrix manipulations and definitions.

Command	Meaning
`zeros`	Create array of all zeros
`ones`	Create array of all ones
`eye`	Identity matrix
`diag`	Create diagonal matrix or get diagonal elements of matrix
`rand`	Uniformly distributed random numbers (range $(0, 1)$)
`randn`	Normally distributed random numbers (mean 0, variance 1)
`triu`	Upper triangular part of matrix
`tril`	Lower triangular part of matrix

Table A.11: Basic operations with matrices.

Command	Meaning
`.*`	Element-wise multiplication
`/ \`	Matrix left and right division
`.'`	Non-conjugate transpose
`'`	Complex conjugate transpose (Hermitian transpose)
`size`	Matrix dimensions
`length`	Vector dimension
`find`	Find indices and values of nonzero elements
`inv`	Invert matrix
`det`	Determinant of matrix
`trace`	Trace of matrix
`norm`	Norm of matrix/vector
`rank`	Rank of matrix
`eig`	Eigenvalues and eigenvectors of matrix
`svd`	Singular value decomposition
`cond`	Condition number of matrix
`orth`	Orthonormal basis for range of matrix

Table A.12: Operations with complex numbers.

Command	Meaning
`real`	Real part
`imag`	Imaginary part
`abs`	Absolute value
`angle`	Phase angle
`conj`	Complex conjugate

Table A.13: MATLAB–Functions for searching and editing.

Command	Meaning
`edit`	Opens a new M–file called Untitled in the Editor
`lookfor`	Search for keyword in all help entries
`which`	Locate functions and M–files

Table A.14: Special keyboard shortcuts.

Key combination	Assignment
`Ctrl-P`	Move up in Command History
`Ctrl-N`	Move down Command History
`Ctrl-B`	Move one position to the left
`Ctrl-F`	Move one position to the right
`Ctrl-A`	Move to the beginning of the row
`Ctrl-E`	Move to the end of the row
`Ctrl-U`, `Ctrl-Z`	Delete one row
`Ctrl-C`	Terminate the currently running script

(The assignment of the shortcuts can be seen and altered in MATLAB via Preferences $\rightarrow$ Keyboard $\rightarrow$ Shortcuts.)