# Contents

# Chapter 1

# Introduction to MATLAB I

The laboratory "MATLAB in digital signal processing" serves as an introduction to working with the mathematical high performance software MATLAB[1] with applications from the field of digital signal processing. In the process, theoretical fundamentals of signal processing will be covered as well. There are various reasons why we have selected MATLAB as programming language and development environment:

- MATLAB makes it possible to realize highly complex algorithms in a structured and elegant way
- MATLAB offers enormous flexibility: applications range from a scientific calculator for numbers and matrices up to the solving of challenging technical problems
- Learning MATLAB is easy. Even after a short period of time, complex tasks can be approached and solved.
- MATLAB provides powerful tools for visualization of complex data structures.
- The development environment that comes with MATLAB offers excellent help.
- MATLAB can easily be expanded using so-called toolboxes.[2]

These properties make MATLAB popular among both mathematicians as well as engineers of different fields. The strength of MATLAB lies in the numerical methods. Although it is also possible to perform symbolical operations such as with Maple or Mathematica if an adequate toolbox is used, this shall not be part of this laboratory.

MATLAB has established itself as one of the most popular tools in the field of digital signal processing. The name MATLAB is derived from MATrix LABoratory and refers to the two main goals that are addressed in the development. On the one hand, matrices which can be manipulated in different ways are the basic elements in MATLAB. This makes MATLAB very well-suited for the processing of digital signals and for numerical methods. The second part of the name (LABoratory) indicates that the main field of application is in technical and scientific research.

[1] MATLAB is a registered trademark of the company The MathWorks Inc. In this laboratory documentation, the name MATLAB refers to both the software and the programming language. It is usually apparent from the context which of the two is meant.

[2] The MathWorks Inc., the creator of MATLAB, as well as several third parties offer such toolboxes for a variety of special applications.

## 1.1 MATLAB Desktop

In this laboratory, you will work with MATLAB 2016a. The MATLAB software is an Integrated Development Enviroment (IDE) which includes an editor with integrated debugger, a command line interpreter, an extensive help system and various other useful features. You can start MATLAB on Windows either by double clicking the MATLAB icon or via the Windows start menu. The MATLAB main window – also called the MATLAB desktop – as shown in Figure 1.1 will then open up.

### 1.1.1 User Interface Elements

The MATLAB desktop comprises several subwindows. These can either be docked (default) or they can be opened as separate windows. The four most important so-called desktop tools – Command Window, Command History, Workspace and Current Folder – are opened by default and arranged as shown in Figure 1.1. Two further desktop tools are the help and the Profiler. All desktop tools can be turned on and off via the **Desktop** menu. Furthermore, they can be undocked or hidden by clicking respectively on the small arrow directed to the upper right or the cross in the top right corner of the MATLAB desktop. An undocked window can be docked again by clicking on the arrow directed to the bottom right.

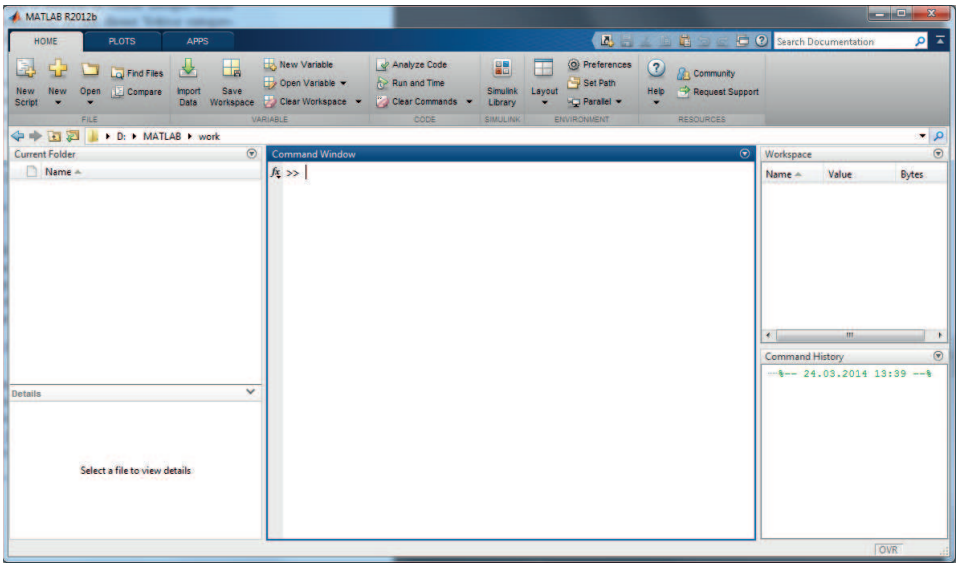



**Figure 1.1:** The MATLAB desktop.

The MATLAB Command Window is your "control center" for MATLAB. Here you can create variables, start function for the manipulation of data and enter commands for visualization. The prompt – which can be identified by >> and a blinking cursor – indicates that MATLAB is ready to receive instructions. If calculations are currently in progress, the text *Busy* appears in the status line next to the start button. Usually, ongoing calculations can be aborted using the key combination **Ctrl-C**. Occasionally, MATLAB may be so busy that it does not react instantly to this input and proceeds with its calculations for another while.

Beside the command input, the MATLAB Command Window is also used for the output of warning and error messages as well as – unless suppressed – results of calculations.

The window Workspace lists all variables which have been created in the current context. Here, you can see name, data type, value, size and memory requirement of the variables. By default, not all columns are turned on. You can select which columns should be displayed via the context menu that opens upon right-clicking the table header. From the Workspace, you can rename or delete variables. The context menu of the individual variables provides you with a selection of different options which can be applied on this particular variable.

A double click on the variable name opens the array editor which can be used to inspect and modify the contents of a variable. However, the size of the processed variable is limited. Large variables and variables with more than two dimension can only be processed with MATLAB commands.

The subwindow Command History shows the most recent commands that have been entered in the command line. This history is retained even when MATLAB is closed. The entries are grouped by date and the groups can be expanded and collapsed by clicking on the small plus sign next to the date. You can easily double click a command to run it again or insert it to the command line by drag and drop. In the latter case, it is not immediately run so that you can modify the command first.

To run multiple entries of the history at a time, mark the desired entries – there is no need for them to be consecutive entries - and proceed as with the single entries.

The window Current Directory lists the contents of the currently selected directory. Here, you can move, rename, copy, create and delete files as with a file manager. Additionally, actions specific to MATLAB can be selected in the context menu.

Finally, there are also the desktop tools help and Profiler. You can use the Profiler to analyze the runtime behavior of your programs. You will learn more about it in chapter 6. The desktop tool help provides access to the good and extensive MATLAB help system. This will be further elaborated in the following section.

As indicated in the introduction, the MATLAB development environment also comprises an editor and a debugger. A lot of functions are provided that facilitate the editing and the debugging of MATLAB code. These windows can also be docked to the MATLAB desktop.

Beside the graphical user interface, a command line interface is additionally available for UNIX/Linux and Mac OS. This can be particularly useful on slow computers as the MATLAB interface is written in Java and can therefore be a little slow sometimes. You can access the command line interface using the options `-nojvm` or `-nodesktop` when opening MATLAB. The first of these options completely turns off the Java Virtual Machine so that you will for example no longer be able to edit plots. The alternative option will disable Java only for the dekstop.

### 1.1.2 Help System

MATLAB includes an excellent integrated help system in which all available command and functions of MATLAB are documented in detail and where a lot of addi-

tional assistance is provided. This system is useful especially for newcomers but it is equally indispensable even for MATLAB experts.

To access the documentation of a specific function from the command line, you can choose from three commands: `help`, `helpwin` and `doc`. With the two commands `help` and `helpwin`, you will in either case get the same information for a given function where in case of `help`, this information will be displayed as output in the Command Window while `helpwin` will instead open the help browser to display it. With `doc`, the help browser will also be opened, only this time the information will be formatted in HTML and it will offer a more detailed help, usually including examples.

The command and the output to receive further information for example on the function `disp`, looks as follows.

```
>> help disp
 DISP Display array.
    DISP(X) displays the array, without printing the array name.  In
    all other ways it's the same as leaving the semicolon off an
    expression except that empty arrays don't display.

    If X is a string, the text is displayed.

    See also int2str, num2str, sprintf, rats, format.

    Overloaded functions or methods (ones with the same name in other directories)
       help opaque/disp.m
       help inline/disp.m
       help cdfepoch/disp.m
       help timer/disp.m
       help serial/disp.m
       help ftp/disp.m
       help memmapfile/disp.m
       help audiorecorder/disp.m
       help audioplayer/disp.m

    Reference page in help browser
       doc disp
```

As you can see, you will also get references to related functions. In this example, this includes the functions `int2str`, `num2str` amongst others. Note that in the help system, command names are almost always given in capitals – most likely to emphasize them. However, the real command names of built-in MATLAB functions will always exclusively consist of lowercase letters. This must be taken into account as MATLAB distinguishes between uppercase and lowercase letters.

The input

```
>> helpwin disp
```

opens the help browser as shown in Figure 1.2. The text matches that of the command line output. Compared to the simple `help`, `helpwin` has two considerable advantages:

- The help is in a separate window so that the most recent command remain visible and the current command line will not disappear from sight even when scrolling through the help text.

- In the top right, there is a link “Default Topics” which can be used to access a list of the most important categories. This way, you can easily get an overview of all the functions of a particular category. If for example, you wish to find out what functions can be used to create a 2D plot, just click on `matlab\graph2d` so that an overview of the corresponding commands will be displayed.

The third command to access the help system is used as follows.

```
>> doc disp
```

This will open the MATLAB help browser (see Figure 1.3) where a detailed description of the requested function formatted in HTML will be displayed. The mostly extensive examples and references to related topics are particularly useful. You can also access the MATLAB help browser from the menu **help** or the button with the question mark in the toolbar of the MATLAB desktop. Within the MATLAB help browser, you can also create bookmarks for more frequently used information.

From the left part of the MATLAB help browser, you can navigate through all of the locally available help. By selecting the respective tab in the upper part of the navigation, you can also perform a keyword or full text search.

## 1.2 Basic Functionality

In the simplest case, you can use MATLAB as an – admittedly a little oversized – calculator. To do so, just enter the calculating operations, as done with any other calculator, directly in the command line. You can for example enter the following (the rows without >> mark the output).

```
>> 12 + 9
ans =
```

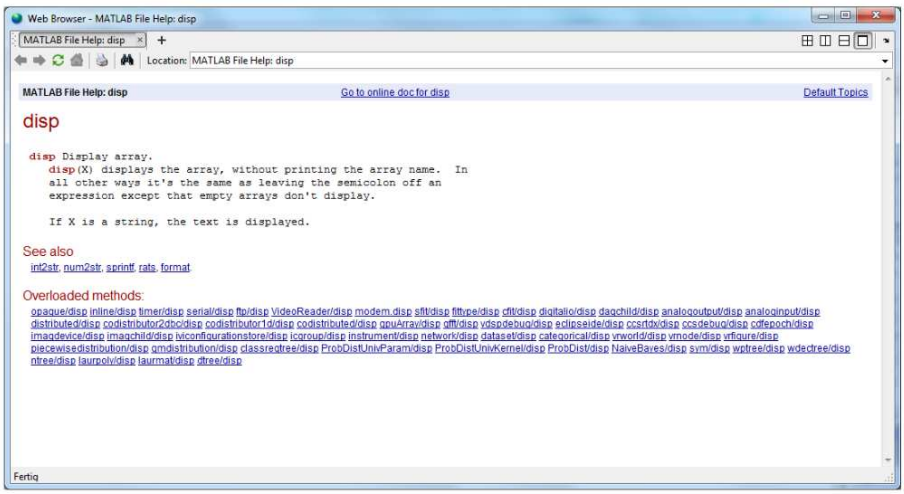



**Figure 1.2:** Output window after calling `helpwin disp`.

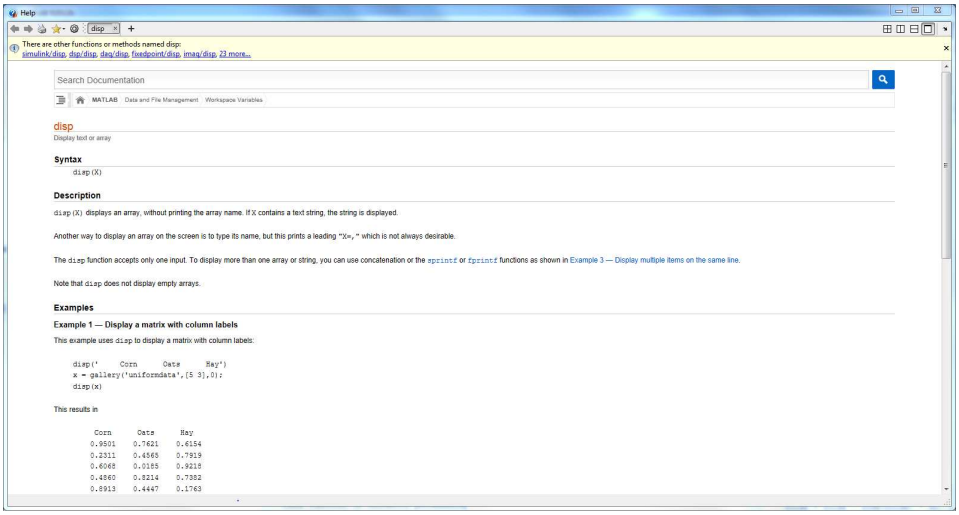



**Figure 1.3:** Output window after calling `doc disp`.

```
   21
>> (3 * 15 + 7) / 2 - 6
ans =
   20
>> 2^7
ans =
   128
```

As you can see, MATLAB can deal with the basic arithmetic operations `+`, `-`, `*`, `/`, `^` etc. in the expected precedence. As usual, you can arbitrarily group the operations using parentheses.

### 1.2.1 Variables

As with other programming languages, there are variables in MATLAB which can be used to store values and results of calculations in order to access these values again at a later time. The type of a variable does not need to be pre-determined. It is also possible to change the type of a variable simply by assigning data of a different type to the variable.

#### The variable `ans`

In the previous example, we have already encountered the variable `ans` which is short for "answer". This variable is used by MATLAB to store results by default unless a different variable is stated explicitly. As with any other variable, it is possible to reference the contents of `ans` simply by using the variable name.

```
>> 3 + 4
ans =
   7
>> ans
ans =
   7
>> ans + 8
```

```
ans =
   15
```

### Defining custom variables

You can also define your own variables by assigning the result of a calculation explicitly to a variable using an equals sign. You are free to select any variable name as long as you do not violate any of the following rules.

- Variable names may only consist of upper and lower case letters (`a-z`, `A-Z`), digits (`0-9`) and underscores (`_`). The first character must always be a letter.
- The name may not comprise any space characters.
- MATLAB distinguishes between upper and lower case letters. Therefore, the names `some_variable` and `Some_Variable` refer to two different variables.
- There is no need to declare variables in advance. They are automatically created by MATLAB upon their first usage.

Here are some examples on how to assign values to your own variables and use them again at a later time:

```
>> a = 15
a =
   15
>> b = 25
b =
   25
>> sum = a + b
sum =
   40
```

### Listing and deleting variables

Once a large number of variables has been created, it is easy to lose track over the already existing variables. You can let MATLAB display a list of the variables that currently exist in the memory using the command `whos`. It shows the name, the dimension, the memory usage and the data type of the individual variables.

```
>> whos
  Name        Size                    Bytes  Class

  a           1x1                         8  double array
  b           1x1                         8  double array
  sum         1x1                         8  double array

Grand total is 3 elements using 24 bytes
```

In order to delete a variable from the Workspace and thereby freeing the memory once more, use the command `clear`. Without further parameters, the command will delete all variables currently existing in the Workspace. To specifically delete single variables, add their names behind the `clear` command.

```
>> whos
  Name        Size                    Bytes  Class
```

```
  a           1x1                         8  double array
  b           1x1                         8  double array
  sum         1x1                         8  double array

Grand total is 3 elements using 24 bytes
>> clear a
>> whos
  Name        Size                    Bytes  Class

  b           1x1                         8  double array
  sum         1x1                         8  double array

Grand total is 2 elements using 16 bytes
```

### Assigning a string to a variable

In MATLAB, you can not only assign numerical values but also strings to variables. MATLAB are always surrounded by simple quotation marks in MATLAB. The assignment of a string to a variable might for example look as follows.

```
>> string = 'Text that is assigned to a variable!'
string =
Text that is assigned to a variable!
```

**Double quotation marks are invalid characters in MATLAB and serve no further purpse. They cannot be used to enclose a string!**

### Suppressing the output of the result variable

In case of extensive calculations with several temporary or very large variables, the output of the result of each of the expressions can get disturbing and unclear. It is often better to graphically visualize long arrays instead of numerically on the console. You can suppress the output of a result by adding a semicolon at the end of the expression.

```
>> a = 10;
>> b = 25;
>> c = a / b;
>> c = a / b
c =
    0.4000
>> c
c =
    0.4000
```

## 1.2.2 Operators

As you will see later, almost all data structures in MATLAB are actually matrices. In order to be able to properly work with numerical matrices in particular, there is need for a means to apply mathematical operations on them. At this point, the full strength of MATLAB can be exploited as it is possible to apply various matrix

operations on matrices in a simple way unlike C, where it is necessary to iterate over each of the matrix elements first.

You have already learned that MATLAB can cope with the basic arithmetic operations. In MATLAB, these can not only be applied on scalar values, but also on matrices (more on matrices and other data structures in chapter 1.3). In the following, the most important operators in MATLAB are briefly described.

### Arithmetic operators

Among the arithmetic operators that are provided by MATLAB are addition, subtraction, multiplication, division and exponentiation. In the last three cases, the operation that is actually executed depends fundamentally on the operands. Moreover, there is an extension: By adding a point in front of the operator, you can indicate MATLAB that the operation should be executed element-wise. Here is an example for a multiplication:

```
>> a = [1, 2; 3, 4]
a =
     1     2
     3     4
>> b = [5, 6; 7, 8]
b =
     5     6
     7     8
>> a * b
ans =
    19    22
    43    50
>> a .* b
ans =
     5    12
    21    32
```

| | |
|---|---|
| `A + B` or `A - B` | Two matrices of the same dimension are added or subtracted element-wisely. As this operation is element-wise anyway, the point in front of the operator can be omitted. |
| `A * B` | Matrix multiplication of two matrices. |
| `A .* B` | Element-wise multiplication of two matrices. |
| `A ^ x` | Exponentiation of $A$ with $x$ if $A$ is a square matrix and $x$ a scalar. If $x$ is an integer number larger than one, the result is calculated by repetitive multiplication of the matrix with itself. |
| `A .^ x` | Element-wise exponentiation of $A$ with $x$. |
| `A / B` | Matrix division (from the right), i. e. the matrix $A$ is multiplied with the inverse matrix of $B$ (`A * inv(B)`). |
| `A ./ B` | Element-wise division of the two matrices. |

| | |
|---|---|
| `A \ B` | Matrix division (from the left), so basically `inv(A) * B` except for a peculiarity in the calculation. If $A$ is a square $(N \times N)$ matrix and $B$ is a vector of length $N$, the result `X = A \ B` is the solution of the system of equations $A*X = B$. If $A$ is an $(M \times N)$ matrix with $M \neq N$, $X$ is the optimum solution of the underdetermined or overdetermined system of equations with respect to the least square error. |
| `A .\ B` | Element-wise division (from the left) of the two matrices $A$ and $B$, i. e. the elements of the matrix $A$ are inverted individually (reciprocal value) and then multplied with the elements of the matrix $B$. |

If MATLAB detects that the dimensions of the matrices for a particular operator do not match, it returns an error message. The last two operator might seem a little peculiar at first. As it turns out however, this proves to be a very compact notation, particularly for solving systems of equations.

### Relational operators

With the relational operators, you can check for two numbers, matrices or strings whether the elements of one of the operands are smaller, larger or equal to the elements of the other operand. The comparison is always conducted element-wise where the dimension of the matrices to be compared must always match. The output is again a matrix of the same size with a one at each position where the condition is fulfilled and a zero at each position where this is not the case;

The available operators are:

| | | | |
|---|---|---|---|
| `==` | equal | `<` | less than |
| `~=` | not equal | `>=` | greather than or equal |
| `>` | greater than | `<=` | less than or equal |

```
>> a = [1, 2; 13, 14]
a =
     1     2
    13    14
>> b = [5, 6; 7, 14]
b =
     5     6
     7    14
>> a == b
ans =
     0     0
     0     1
>> a > b
ans =
     0     0
     1     0
>> a >= b
ans =
     0     0
     1     1
```

### Logical operators

In MATLAB, you can connect matrices of the same size element-wise with logical operators. When doing so, values that are not equal to zero are interpreted as true, zero-values are interpreted as false. As for the relational operators, the result is a matrix consisting of ones and zeros. You can choose from the operators `&` (and), `|` (or), `~` (not) and the function `xor`. Furthermore, there are the functions `any` and `all` that return one if at least one of the elements of a matrix or all of the elements respectively are unequal to zero.

To learn more about operators, you can enter `doc +` on the MATLAB console (also works with the other operators). MATLAB will then list all available operators and their function equivalents. If you call `doc` with the respective name of the function, you will get detailed information.

## 1.2.3 Functions and their Particularities

You would not get far with MATLAB if you were only able to use the basic operations. Because of this, MATLAB has a large pool of built-in functions to offer. Among these are functions like `sin`, `log10` etc., that return a function value to a given input value. Beside this, there is also a large number of help functions that can aid you when it comes to dealing with and manipulation of complex data structures. Listing and explaining all of the available functions is beyond the scope of this laboratory documentation, but you can always use the MATLAB help.

A function call in MATLAB looks as follows.

```
>> [out1, out2, ...] = funcname(in1, in2, ...);
```

Depending on the function, input and/or output parameters may not be required. In this case, the parentheses and/or the square brackets and the equals sign can be omitted. Input parameters can be both data written down explicitly as well as variables of the required type. Functions can also be used as parameters as long as they return the correct data type with the needed dimension.

There is also another possibility to call a function which is used by some of the built-in MATLAB functions as a syntax simplification. Alternatively to the functional form described before, the parameters can be listed without the surrounding parentheses when calling one of these functions. However, there are two considerable restrictions for this form.

- Parameters are always interpreted as strings. Therefore, it is impossible to pass variables.
- The return value of a function called like this cannot be assigned to a variable. Usually, functions that explicitly support this calling convention, do not return values.

Functions that are called in this non-functional form according to the documentation, can always be called in the functional form as well by enclosing the individual parameters by simple quotation marks and separating them with commas. If your own functions process string inputs only, you can also call them in the non-functional form without the need for any special modifications.

What makes functions in MATLAB so special is that most of them can not only be applied on scalars, but also directly on vectors or matrices. Assume for example that you wish to create a vector `x` of length 100 with values between 0 and $2 \cdot \pi$ and then to compute the sine of these values. With the following code lines, you can create `x` and compute the sine of all of its elements with just a single command. In this example, the result is assigned to the vector `y` the size of which matches that of `x`.

```
>> x = (0:99) * 2 * pi / 99;
>> y = sin(x);
>> plot(x, y);
>> whos
  Name      Size                    Bytes  Class

  x         1x100                     800  double array
  y         1x100                     800  double array

Grand total is 200 elements using 1600 bytes
```

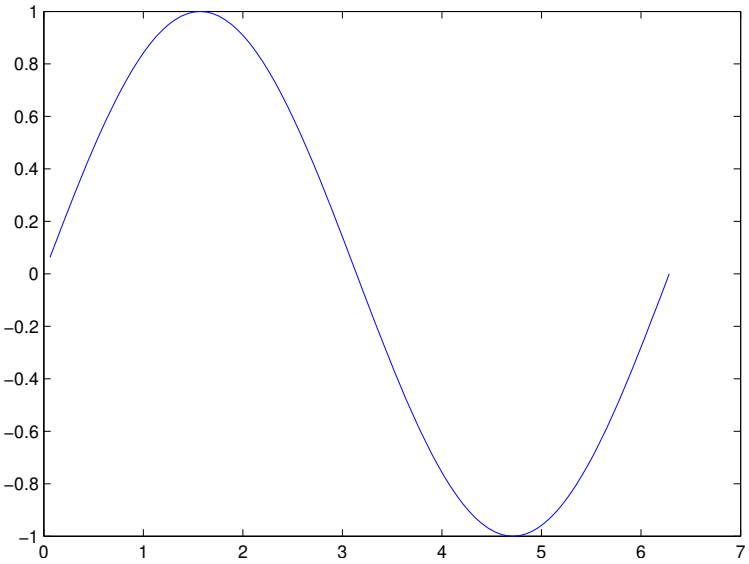



**Figure 1.4:** Output of the function `plot(x, y)` where `x` goes from zero to $2 \cdot \pi$ and `y` contains the corresponding sine values.

For the same operation, you would have to program a loop in C. In MATLAB, you could also use such a loop although this would require a lot more typing and, as you will learn later on, would be considerably slower in the execution.

You will learn more about the command `plot` which is used in the example above at a later time. In this case, it plots the values of the vector `x` against the values of the vector `y`. The result is shown in Figure 1.4. The call of the sine function is here basically the same as before, except that in this case, both `x` and `y` are vectors. MATLAB computes the sine of each of the elements of `x` and stores the result in `y`. `x` and `y` have the same length after the execution of the function. In case of the `sin`, this even works for matrices of arbitrary dimension. The result is always a matrix of the same size as the input matrix.

Many functions like for example `fft`, which determines the Fourier transform to the given input values, expect a signal vector as input. If you wish to process several vectors at a time, this is also possible as such functions process input data separately

column by column. The function `fft` computes the FFT column-wise by default in case of a matrix and stores the results in the columns of the output matrix.

You can refer to the respective `doc` to get information on the valid dimensions of the input matrix for various functions and how these matrices are then processed.

### 1.2.4 Effective Use of the Command Line

If you spend some time with MATLAB, you will notice that you need certain commands very frequently and that you commonly wish to make only minor modifications to commands that you have used shortly before. When it comes to this, MATLAB provides a lot of support to significantly reduce the required typing effort. As explained before, all commands that you enter in the command line are recorded in the History and displayed in a separate subwindow. It has already been outlined how you can use the History to call one of the previous commands again.

Moreover, it is also possible to access previously entered commands from the command line using only the keyboard. Using the (up and down) arrow keys, you can browse through the most recent input commands which will appear on the console in the process. Once you have found the right one, you are free to modify it and then execute the command.

You can navigate through the History entries even faster if you know what characters the desired command line starts with. Assume that it start with `plot`. If you enter this in the command line and then scroll through the History using the arrow keys, MATLAB will exclusively show you the entries that start with the characters specified by you.

Searching in the History

Finally, there is also a third option to find recent commands, the normal or the incremental search in the Command Window. More specifically, this will not only search the command lines, but the entire output in the Command Window. You can access the normal search from the menu **Edit → Find ...**, the incremental search with **Ctrl-S** or **Ctrl-R** for the forward and backward search respectively. When repetitively using the same keyboard shortcut, MATLAB will jump to the next occurrence of the desired character sequence and mark it.

When entering function and variable names, these can be completed by pressing the tabulator key provided that are no ambiguities. If there is for example a variable with the name `examplevariable` in the Workspace and you enter `examplev` followed by the tabulator key, the input will be complemented to the full variable name as neither a function nor a different variable exists that start with `examplev`. If the name still is ambiguous, MATLAB will only add characters until ambiguities arise and then present you a selection of the names that are still possible.

### 1.2.5 Saving and Loading Variables

Variables in MATLAB's Workspace are only available in the memory and are lost once MATLAB is closed. There are various possibilities to save them to a file. The two functions to save and restore variables are `save` and `load`. If you call `save` without parameters, this will save all variables of the current Workspace to the file `matlab.mat` in the current directory.

`save`

`load`

Equivalently, the function `load` loads `matlab.mat` from the current directory and creates the saved variable on the desktop with their original name. If there are any name collisions, variables that already exist will be overwritten. If no file `matlab.mat` is found, MATLAB will return an error message.

To get control over what variables are saved to what file or what variables to import from a file, you can add further parameters when calling either of the two functions. Furthermore, you can not only store the variables in MATLAB's own binary format but also as ASCII text file. Another interesting option is to not directly load the variables from a MAT file to the current Workspace but as elements of a structure. This ensures that variables with the same name are not overwritten. You can refer to the MATLAB help system to get further information on the exact calling conventions of both functions.

## 1.3 Data Types and Data Structures

In order to be able to program properly with MATLAB, you have to know what data types and data structures there are and how to handle them. Beside simple matrices, there are so-called structures, cell arrays and classes or objects in MATLAB. The latter will not be covered as it goes beyond the scope of this laboratory.

You can find an overview of the data type related functions in the help at **MATLAB → Programming → Data Types → Numeric Types → Function Summary**.

### 1.3.1 Numerical Data Types

MATLAB distinguishes between integer data types and floating point data types. Integers are further divided into those that are signed and those that are unsigned and they differ in size, namely 8, 16, 32 or 64 bit per value. There are two floating point data types, one with single precision (float) and one with double precision (double). The default data type for numerical data is double but it can also be specified by explicitly stating the type upon create the variable. The corresponding functions are `int8`, `uint16` and `double` amongst others.

Complex numbers

Beside real numbers, MATLAB can also handle complex numbers that consist of a real and an imaginary part. The data type of the values can be an arbitrary numerical data type but it must be the same for both components. You can either create complex numbers directly by marking the imaginary part by an `i` or a `j`. You can use the notation with the multiplication sign provided that you have not created a variable `i` or `j`:

```
>> c = 1 + 2i
c =
   1.0000 + 2.0000i
>> c = 1 + 2j
c =
   1.0000 + 2.0000i
>> c = 1 + 2*i
c =
   1.0000 + 2.0000i
```

The first syntax works only for the direct input of numbers, not for variables. A simple way to set the real and imaginary parts of a variable without risking that `i` or `j` may have accidentally set differently before is the function `complex`, the first argument of which is the real part and the second argument is the imaginary part of the complex number that it returns. This also works for matrices of arbitrary dimension:

```
>> a = 1;
>> b = 2;
>> c = complex(a, b)
c =
   1.0000 + 2.0000i
```

### 1.3.2 Strings

As you have already learned in the chapter on variables, MATLAB can process strings as well. You can enter them directly by enclosing the string with simple quotation marks. If the string itself should contain a simple quotation mark, the quotation mark must be doubled. As with vectors and matrices, it is also possible to concatenate string.

```
>> s = 'String'
s =
String
>> s = 'String with ''quotation marks'''
s =
String with 'quotation marks'
>> s2 = [s ' and supplement.']
s2 =
String with 'quotation marks' and supplement.
```

If you wish to combine several strings in a matrix, so that each there is a string in each of the lines of the matrix, you have to ensure that all lines have the same length when creating the matrix. If you do not want to take care of that yourself, you can use the function `char` which fills short strings up appropriately.

```
>> s = ['Here we go!'; 'That was it']
s =
Here we go!
That was it
>> s = char('Just a word', 'To cut a long story short')
s =
Just a word
To cut a long story short
```

### 1.3.3 Matrices

As indicated in the introductions, MATLAB is primarily based on matrices and MATLAB's special skill is the manipulation of these. Scalars and vectors are merely special cases of matrices. A scalar is a matrix of dimension $(1 \times 1)$. A vector can be either a row vector or a column vector which corresponds to matrices of dimension $(1 \times n)$ or $(n \times 1)$ respectively. Aside from these special cases, a matrix in MATLAB can have an almost arbitrarily large number of dimensions of any size, limited only

by the available memory space. However, we will only consider matrices with up to three dimensions in the rest of this laboratory.

### Creating matrices

In MATLAB, matrices are defined with square brackets (`[...]`). The matrix elements are separated by commas or space characters as well as semicolons. Commas and spaces separate the elements within a single line while semicolons separate one line from the next. This allows for the easy creation of small vectors and matrices.

```
>> rowvector = [1, 2, 3]
rowvector =
     1    2    3
>> columnvector = [1; 2; 3]
columnvector =
     1
     2
     3
>> matrix2d = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10, 11, 12]
     1    2    3
     4    5    6
     7    8    9
    10   11   12
```

In the previous example, a $(4 \times 3)$ matrix was created, i. e. it consists of four rows and three columns. You now have the option to combine matrices and vectors to create new matrices. When doing so, you have to ensure that horizontally combined matrices must always have the same number of rows, vertically combined matrices must always have the same number of columns.

```
>> rowvector1 = [1, 2, 3];
>> rowvector2 = [4, 5, 6];
>> combination = [rowvector1, rowvector2]
combination =
     1    2    3    4    5    6
>> combination = [rowvector1; rowvector2]
combination =
     1    2    3
     4    5    6
>> rowvector3 = [4, 5, 6, 7];
>> combination = [rowvector1, rowvector3]
combination =
     1    2    3    4    5    6    7
>> combination = [rowvector1; rowvector3]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

Explicitly writing out the individual elements to create a matrix is still quite handy for smaller matrices. However, for larger matrices this procedure can get very tedious. Consider the creation of a vector of the form

```
>> vector = [0, 2, 4, 6, 8, ..., 1996, 1998];
```

in the previously introduced way. This would result in a lot of typing effort and subject to errors. Luckily, MATLAB provides various functions and operators that allow you to easily create commonly used matrices.

**zeros(n,m)** Generates a $(n \times m)$ matrix filled with zeros. If a zero matrix with more dimensions is required, the size of the following dimensions is specified as well, separated by commas (`zeros(n,m,o,p,...)`). If only one parameter is used, a two-dimensional square $(n \times n)$ matrix is generated.

**ones(n,m)** Generates a $(n \times m)$ matrix filled with ones. The generation of matrices with more dimensions and square matrices works exactly as with `zeros(...)`.

*start*:*increment*:*stop* The so-called colon operator creates a row vector that starts with *start* where each element is incremented by *increment* as long as the elements are no larger. in case of a positive increment, or no smaller, in case of a negative increment, than *stop*.If :*increment* is omitted such as in *start*:*stop*, 1 is used as increment instead.

There are numerous other commands for the generation of generic matrices some of which you will encounter in the course of this laboratory. For further information, you can search for "Elementary Matrices and Arrays" in the MATLAB help documentations. In the following, several examples are given:

```
>> zeros(2)
ans =
     0     0
     0     0
>> zeros(2,3)
ans =
     0     0     0
     0     0     0
>> ones(3,1)
ans =
     1
     1
     1
>> 1:10
ans =
     1     2     3     4     5     6     7     8     9    10
>> 1:2:10
ans =
     1     3     5     7     9
>> 10:-2:0
ans =
    10     8     6     4     2     0
```

If you encounter a matrix in a program that you do not know the dimension of, you can use the commands `length`, `ndims` and `size`.

**length** This command receives one argument and returns the length if this parameter is a vector. For matrices, the length of the largest dimension is returned.

```
>> x = ones(1,8);
>> n = length(x)
n =
     8
>> x = ones(2,10,3);
>> n = length(x)
n =
    10
```

**size** This command can be called with one or two arguments. In case of a single argument, it returns a vector the length of which matches the number of di-

mensions of the input matrix while the elements of the output vector represent the size of the individual dimensions.

```
>> x = ones(1, 8);
>> n = size(x)
n =
     1     8
>> x = ones(2, 10, 3);
>> n = size(x)
n =
    2    10    3
```

If `size` is called with two arguments, the size of the dimension specified in the second argument is returned.

```
>> x = ones(2, 10, 3);
>> n = size(x, 2)
n =
    10
```

Refer to the documentation for further properties of this command.

**`ndims`** This command returns the number of dimensions of a vector. In case of a matrix `x`, `ndims(x)` is equivalent to `length(size(x))`.

### Accessing matrix elements – Indexing

Now you know how to generate a matrix and use it as a whole. But how to access the individual elements or a specific range? This is achieved using the index of an element. Unlike C/C++, the numbering in MATLAB starts with 1, i. e. the first element can be accessed with the index 1, the second element with the index 2 and so forth. The index is specified in parentheses behind the variable name. There is also a special index identifier `end` which always allows to access the last element of a vector. This identifier can be used like a variable within the indexing, i. e. you can include it in the index calculations.

```
>> vector = [2, 5, 9, 14]
vector =
     2    5    9    14
>> vector(3)
ans =
     9
>> vector(end)
ans =
     14
>> vector(end - 1)
ans =
     9
>> vector(end + 1) = 19
vector =
     2    5    9    14    19
```

Just as simply and intuitively, you can also address the elements of a two-dimensional matrix with `matrix(i,j)` where `i` specifies the row and `j` the column of the desired element. In case of dimensions with three or more dimensions, the indices of the higher dimensions are appended to the other indices separated by commas (`matrix(i,j,k,...)`). The identifier `end` can be used here as well and represents the length of the corresponding dimension.

```
>> matrix2d = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10, 11, 12]
     1    2    3
     4    5    6
     7    8    9
    10   11   12
>> matrix2d(1,1)
ans =
     1
>> matrix2d(4,2)
ans =
     11
>> matrix2d(end,end)
ans =
     12
>> matrix2d(2,end)
ans =
     6
```

Occasionally, you may not only wish to access individual elements but entire blocks within a matrix. This can also be done very easily in MATLAB as it is possible to use a vector for the index of each of the dimensions, i. e. you can access multiple rows or columns at a time. There is no need for these blocks to be connected.

```
>> matrix2d([1,2],[1,2])
ans =
     1    2
     4    5
>> matrix2d([1,4],[2,3])
ans =
      2     3
     11    12
```

Instead of explicitly writing out vectors, arbitrary expressions can also be used as index as long as they return a valid index as a result. For example, you can use the colon operator to do so.

```
>> matrix2d = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12; 13, 14, 15, 16]
     1    2    3    4
     5    6    7    8
     9   10   11   12
    13   14   15   16
>> matrix2d(2:end, 1:3)
ans =
     5    6    7
     9   10   11
    13   14   15
>> matrix2d(1:2:end, 1:2:end)
ans =
     1   3
     9  11
```

In MATLAB, it is not only possible to address the elements of an $n$-dimensional matrix using the indices of all $n$ dimensions (subscripts), such a matrix can also be interpreted as a vector in which all matrix elements are simply concatenated. These elements can also be accessed using a single (linear) index. To do so, it is important to know that MATLAB stores the elements of two-dimensional matrices column-wise in the memory. Analogously, this applies to more-dimensional matrices in a similar way. This can best be illustrated by means of an example.

```
>> a = [1 2 3; 4 5 6; 7 8 9]
a =
     1     2     3
     4     5     6
     7     8     9
>> a(:)
ans =
     1
     4
     7
     2
     5
     8
     3
     6
     9
>> b = rand(2,2,2)
b(:,:,1) =
    0.6038    0.1988
    0.2722    0.0153

b(:,:,2) =
    0.7468    0.9318
    0.4451    0.4660

>> b(:)
ans =
    0.6038
    0.2722
    0.1988
    0.0153
    0.7468
    0.4451
    0.9318
    0.4660
```

Switching between these two methods of indexing can easily be done with the two functions `ind2sub` and `sub2ind`.

**`ind2sub`** Computes the index to a given linear index for a matrix of the specified size.

```
[i1, i2, ..., in] = ind2sub(size(A), ind);
```

**`sub2ind`** Computes the linear index to a given index for a matrix of the specified size.

```
[ind] = sub2ind(size(A), i1, i2, ..., in);
```

The indexing methods presented above are called more-dimensional indexing and linear indexing respectively. Another kind of indexing is the use of logical operators. For example, relational operators can be used to generate a matrix filled with logical values which has the same dimension as the matrix that the comparison was applied on. Most importantly, when processing large matrices, the logical indexing can save computation time as further computation steps are required for the transformation into linear indices. Only if the order of the extracted values matters, it is reasonable to perform the transformation into linear indices. In this example, the functions

`find` and `flipud` are used, the exact purpose and handling of which you can find in the MATLAB help.

```
>> % Defining a 2x2 matrix
>> matrix2d = [5, -6; -7, 8]
     5   -6
    -7    8
>> % Logical indexing
>> ind_logical = matrix2d > 0
     1    0
     0    1
>> % Value output
>> matrix2d(ind_logical)
     5
     8
>> % Transformation into linear indices
>> ind_linear = find(ind_logical)
     1
     4
>> % Reversing the order of the index column vector
>> ind_linear = flipud(ind_linear)
     4
     1
>> % Value output now in reverse order
>> matrix2d(ind_linear)
     8
     5
>> % Of course, you can also combine it all in a single command
>> matrix2d(flipud(find(matrix2d>0)))
     8
     5
```

With more complex programs, it may be possible, that indexing and reindexing is performed multiple times. To maintain the legibility of the code and to prevent redundant calculations, it is often advisable to declare more rather than less help variables - as in the example above.

### 1.3.4 Cell Arrays

In a normal MATLAB matrix, you can store data of the same type only. It is possible to bypass this constraint by using cell arrays instead. While basically being a matrix, any data type supported by MATLAB can be stored in each of the elements of the cell array independently from the other elements. You can for example store a $(2 \times 3)$ matrix of double values as one element of the cell array and a simple string as another element. You can create a multi-dimensional empty cell array with the function `cell`.

**`cell(m,n)`** Generates an empty cell array of the size $(m \times n)$. If you wish to create higher-dimensional cell arrays, you can proceed as with `ones` or `zeros`.

However, it is far more common to create cell arrays by surrounding the elements with braces instead of square brackets as for normal arrays.

```
>> matrix2d = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10, 11, 12];
>> C = {matrix2d, 'string' , sum(matrix2d)}
C =
    [4x3 double]    'string'    [1x3 double]
```

You can access the elements using the index exactly as it is the case for normal arrays. While using parentheses when trying to access individual elements of cell arrays results in a valid operation, the result will certainly not be what you have tried to achieve in most cases. Consider the following example:

```
>> A = C(1)
A =
    [4x3 double]
>> A(1,2)
??? Index exceeds matrix dimensions.
```

What happened? Take a look at the third line. When accessing the first element of C, this does not only return the $(4 \times 3)$ matrix as expected, but a cell which contains this matrix. Otherwise, the entire matrix would have had to be displayed at this point. This makes A a cell array as well, more specifically one of dimension $(1 \times 1)$. Therefore, the element (1,2) does not exist which explains the error message from the example above.

If you wish to address the contents of the cells rather than the cell itself, you have to use braces instead of parentheses. This even allows you to access the contents in the same step if, in case of the matrix, you add the desired index in parentheses behind the expression. One application of cell arrays is to combine strings of different lengths in a single data structure. As the individual cells are independent from each other, this removes the constraint from Section 1.3.2 that the strings must all have the same length.

```
>> A = C{1}
A =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
>> A(1,2)
ans =
     2
>> C{1}(1,2)
ans =
     2
```

### 1.3.5 Structures

Structures are basically cell arrays of dimension $(1 \times n)$ where $n$ represents the number of fields. Unlike cell arrays, the elements of a structure are not addressed by their index but by a name which is specified behind the variable name, separated by a point. This makes them similar to structs in C/C++. As typically the case for MATLAB, there is no need to declare the individual fields in advance, you can just go ahead and use them. MATLAB takes care of the creation of the respective fields and the reservation of memory space upon the first usage.

```
>> st.x = [1, 2, 3];
>> st.y = [11, 12, 13];
>> st.description = 'A string';
>> st
st =
               x: [1 2 3]
```

```
              y: [11 12 13]
    description: 'A string'
```

## 1.4 Simple Visualization

What comes next is one of the major strengths of MATLAB, the graphical visualization of numerical data. In order to properly interpret big vectors and matrices, the output as a long series of numbers on the console is hardly helpful at all, but rather requires a visualization. MATLAB provides a variety of possibilities to present data graphically. In this section, we will cover the basics of this matter and we will elaborate further on it in the subsequent chapters.

At first, several remarks. A plot command opens a new figure window – provided that none of the kind is open yet – and creates an empty axes element inside the figure window in which it can then draw the graph. If one or more figure windows are already open, the plot command draws the graph in the most recently used axes element of the figure window that was active before. By manually activating a specific window, you can influence where the graph is drawn.

`figure`

You can open an additional figure window with the command `figure`. The following plot commands will draw inside this window. You can also activate a figure window using the figure command by adding its handle as a parameter. If no figure window of the specified handle exists, a new one is created. You can get the handle of a figure window by storing the return value of `figure` in a variable when the figure window is created. You can also get the handle of the current figure window with the function `gcf` (get current figure).

```
f = figure;   % creates a new figure window and stores the
              % corresponding handle in f
figure        % creates a new figure window
figure(1)     % activates the figure with handle 1 or creates
              % it if it does not exist yet
figure(f)     % actives the figure window created in the first line
```

### 1.4.1 Two-dimensional Plots

The simplest and most commonly used command for graphical visualization is `plot`. When two vectors $x$ and $y$ are passed as parameters, MATLAB will use them to generate a simple line chart where the x- and y-coordinates of the plotted points are determined by the elements of the vector $x$ and the corresponding elements of the vector $y$. These points are connected by straight lines in the original order in which the points are extracted from the vectors.

```
>> x = [1,    2, 3, 4, 5];
>> y = [12, 0.5, 1, 3, 0];
>> plot(x, y);
```

After the plot command is called, a new window (which will be referred to as the figure window in the following) opens that shows the corresponding diagram. The output of the code lines above is shown in Figure 1.5.

More simply, the command

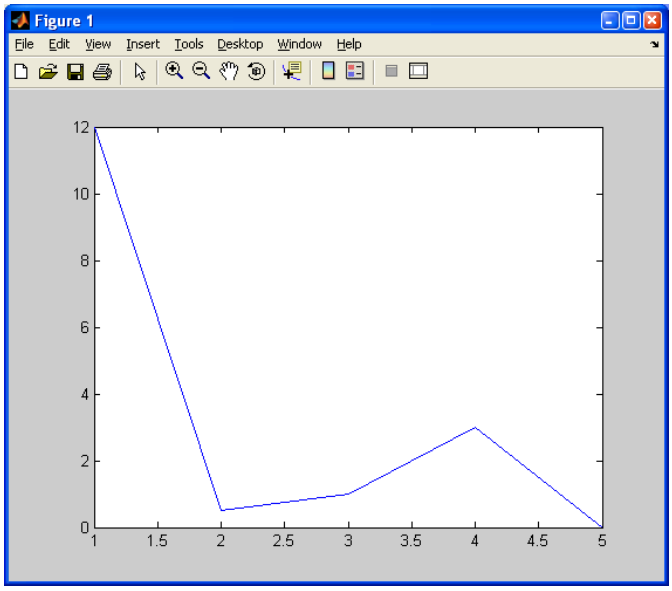



**Figure 1.5:** Output of `plot(x, y)`

```
>> plot(y);
```

would have produced the same result. This is possible because MATLAB interprets the values of the vector as y-coordinates if only one argument is passed and assumes the corresponding indices of the elements as x-coordinates. This means that the values of the vector are plotted equidistantly with distance 1 on the x-axis.

In case of multiple arguments, the order of the arguments matters. You can for example easily exchange the x and y-axis by inverting the order of the arguments when calling `plot`. The result is shown in Figure 1.6.

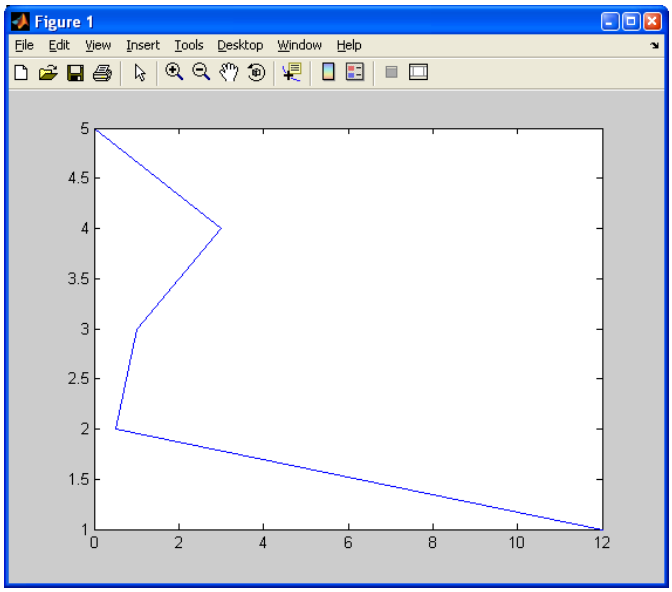



**Figure 1.6:** Output of `plot(y, x)`

Surely, you will soon happen to have the desire to visualize multiple curves in a single chart, for example so that you can compare them more easily. This can be

achieved by the use of the `hold` command which allows you to set whether the plot command clears the contents of the window prior to drawing the new curve (`hold off`, the default setting) or whether the curve to be plotted should be added to the existing ones (`hold on`). The output of the following command lines is depicted in Figure 1.7.

```
>> x = 0.1:0.1:10;
>> plot(x, sqrt(x));
>> hold on;
>> plot(x, sqrt(2*x));
>> plot(x, sqrt(3*x));
>> plot(x, sqrt(4*x));
```

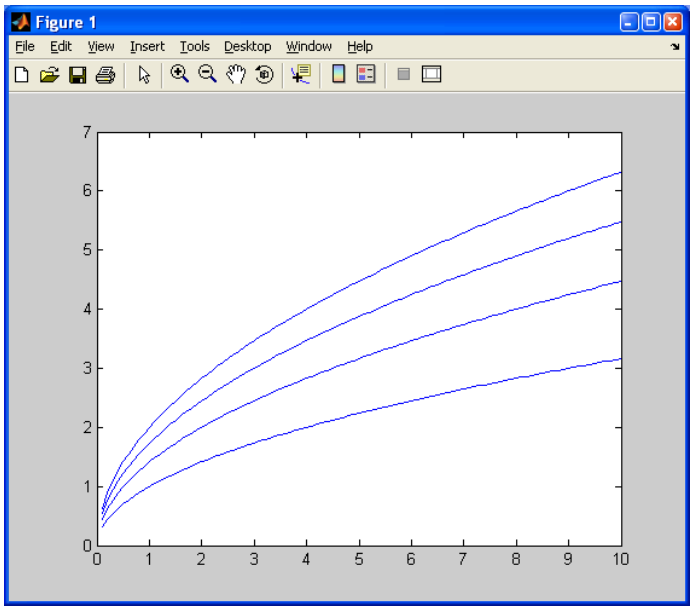



**Figure 1.7:** Output of multiple plot command with the setting hold on

The downside of this method is that the same line type is used for every plot. In the current version of MATLAB (from version 7.0) on, this is addressed with the alternative command `hold all` instead of `hold on`. When this command is used instead, MATLAB will – up to a certain number – use a different color for every newly added line.

Instead of using `hold on` or `hold all`, you can also pass all vector pairs to the plot command at the same time by concatenating them in the plot command as shown in the following example. When doing so, the curves are drawn in different colors, exactly as when `hold all` is used.

```
>> x = 0.1:0.1:10;
>> plot(x, sqrt(x), x, sqrt(2*x), x, sqrt(3*x), x, sqrt(4*x));
```

Moreover, there are several possibilities to pass the coordinate pairs as matrices or as a combination of vectors and matrices. This would go to far at this point. For further information, you can refer to the `doc` for the command `plot`.

### 1.4.2 Three-dimensional Plots

In MATLAB, you can create two different types of three-dimensional plots: line plots and surface plots. You can generate three-dimensional line plots analogously to their two-dimensional equivalent except that you have to use the command `plot3` and that you need an additional third vector which contains the z-coordinates for each of the points. The following example, the output of which is shown in Figure 1.8, shows how this works. In this example, you will also encounter a new command, `grid`. This allows you to turn the grid lines of a plot (not only in 3D plots) on and off with the switches `on` and `off` respectively.

```
>> Z = [0 : pi/50 : 10*pi];
>> X = exp(-.1 .* Z) .* cos(Z);
>> Y = exp(-.1 .* Z) .* sin(Z);
>> plot3(X, Y, Z); grid on;
```

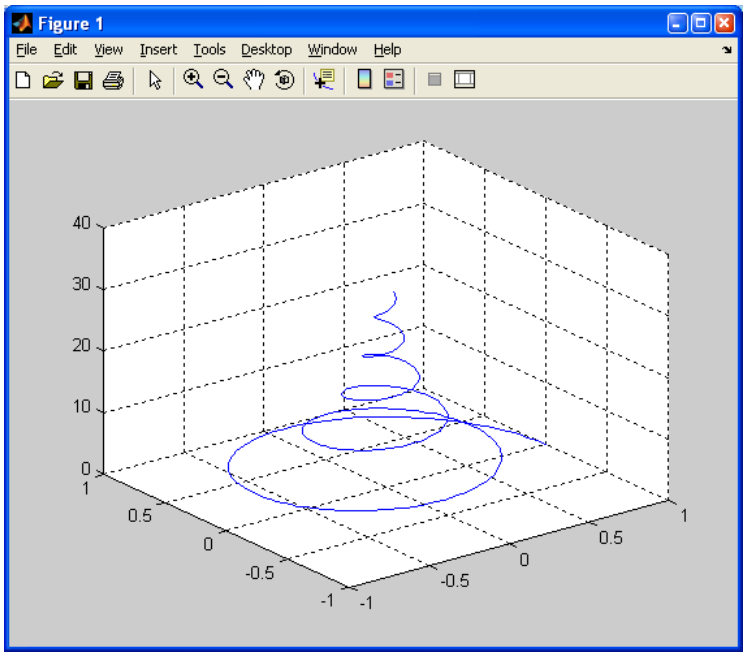



**Figure 1.8:** Three-dimensional line plot created with the command `plot3`

The second type of three-dimensional plots, the surface plot, can be created with the commands `mesh` and `meshgrid`. The procedure is usually as follows:

1. Generation of the grid points in the xy-plane with the command `meshgrid`.
2. Evaluation of a three-dimensional function at these grid points.
3. Generation of the surface plot with the command `mesh`.

For the generation of the grid points, first create vectors for the x- and y-axis that cover the desired value range equidistantly. Call `meshgrid` with these vectors to get two matrices that contain all the x- and corresponding y-coordinates of the grid that is created when all possible combinations of x and y values are interpreted as grid points. Afterwards, use these matrices to evaluate a three-dimensional function at these grid points. You can display the result, as shown in the following example for a three-dimensional paraboloid, with the command `mesh`.

```
>> x = [-10 : .5 : 10];
>> y = [-10 : 2 : 10];
>> [X, Y] = meshgrid(x, y);
>> Z = X.^2 + Y.^2;
>> mesh(X,Y,Z);
```

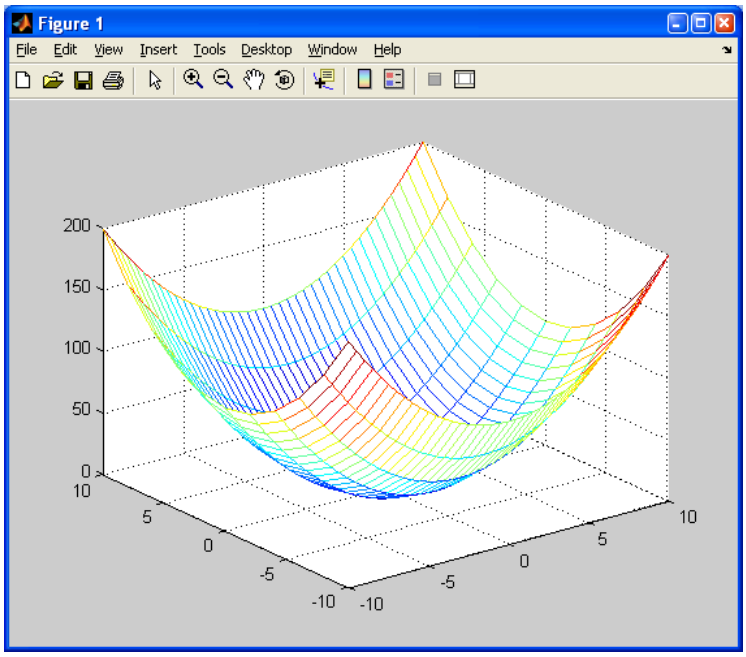



**Figure 1.9:** Three-dimensional surface plot creates using the command `mesh`

### 1.4.3 Labels and Legends

In order for graphs to be meaningful, the axes should be labeled, a title and, if more than one line is shown, a legend should added. The corresponding commands are `xlabel`, `ylabel`, `zlabel`, `title` and `legend`. The usage of these commands is very intuitive and is briefly illustrated in the following example. You can refer to the `doc` for further information.

```
>> Z = [0 : pi/50 : 10*pi];
>> X = exp(-.1 .* Z) .* cos(Z);
>> Y = exp(-.1 .* Z) .* sin(Z);
>> plot3(X, Y, Z); grid on; hold on;
>> X = exp(-.2 .* Z) .* cos(Z);
>> Y = exp(-.2 .* Z) .* sin(Z);
>> plot3(X, Y, Z, '.'); hold off;
>> xlabel('x-Achse');
>> ylabel('y-Achse');
>> zlabel('z-Achse');
>> title('Schlangenbeschwörung')
>> legend('Schlange 1', 'Schlange 2');
```

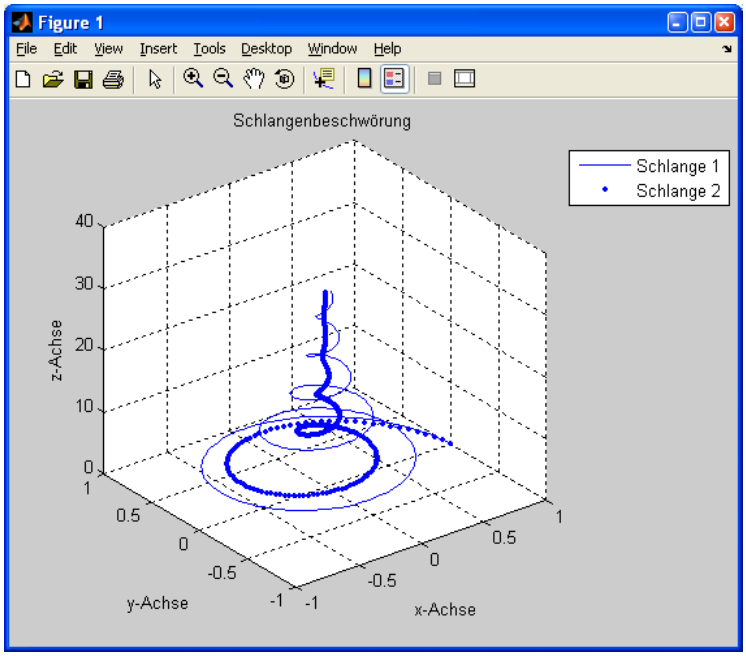



**Figure 1.10:** Three-dimensional line plot with title, axes labeling and legend

## 1.5 Importing and Playing Audio Files

MATLAB can handle a variety of different audio file formats, including e. g. Microsoft WAVE (.wav). Audio files can be imported with the command `audioread` and can likewise be exported with the command `audiowrite`.

A function call to import a WAVE file then looks as follows:

```
y = audioread('filename');
[y, Fs] = audioread('filename');
```

The first option merely returns the audio data and stores it in `y` while the second one additionally returns the sampling rate `Fs`. The command `audioinfo` provides additional information such as the number of bits (`bits`) that each of the samples of the WAVE file is encoded with. There are yet other ways to call `audioread`, for example to import only a section of the data or to determine the number of samples without actually importing the file.

The function `audiowrite` allows you to save audio data back in a WAVE file. The function call looks as follows:

```
audiowrite('filename', y);
audiowrite('filename', y, Fs);
audiowrite('filename', y, Fs, 'BitsPerSample', Value_BitsPerSample);
```

The meaning of the parameters is the same as for `audioread`.

Sometimes, you may want to listen to some audio data, for example to check for certain effects. To remove the need to always save the data in WAVE files and then play them with an external player, MATLAB provides the functions `sound` and `soundsc` that allow you to play audio data directly from MATLAB.

To play audio data with the function `sound`, you have to pass the audio data and the sampling rate to the function.

```
sound(y, Fs);
```

`soundsc` additionally performs an automatic scaling of the input data.

These functions are well-suited to quickly listen to short pieces of audio. A far more flexible alternative is the audioplayer object. When creating such an object, you need to specify the audio data to be reproduced and the sampling frequency. The playback is not started immediately at this point. You can start and manage the playback with the following functions, all of which expect the audioplayer object as the first argument.

`play`
: Starts the playback. You can additionally specify what data range should be played.

```
play(ap)
play(ap, start)
play(ap, [start stop])
```

`playblocking`
: Starts the playback in blocking mode. Otherwise like `play`.

`stop`
: Terminates the playback.

`pause`
: Interrupts the playback at the current position.

`resume`
: Resumes the playback.

`isplaying`
: Returns whether the data of the specified audioplayer object is currently played.

```
ap = audioplayer(y, Fs);  % create audiplayer object
play(ap);                 % initiate playback,
pause(ap);                % ... pause,
resume(ap);               % ... resume again
stop(ap);                 % and finally stop.
```

If the given audio data is neither in the WAVE format nor in the format of NeXT/-SUN or equivalently if you wish to store audio data in a different file type, you will have to use the binary input and output functions for reading and writing respectively. To do so, read the `doc` for the functions `fread` and `fwrite`.

## 1.6 Exercises

To work on the exercises, open MATLAB from the icon on the desktop. Only then, the paths required for the laboratory are all set correctly. MATLAB then opens in the directory `H:\` where a subdirectory exists for each laboratory date. Switch to the respective folder and save your own files in there. The given files that you need for each week's exercises can also be found in this folder. If you need the original again after having made some modifications to the given files, you can find all of them at `X:\Vorgaben\`.

### Exercise 1.1 Properties of speech signals

1. Import the file `speech.wav`. What sampling rate does the signal have?

   Play it with the correct sampling rate via the sound card.

2. Extract a section of 8000 samples starting from sample 5500 and plot it so that the x-axis shows the time instead of the index.

3. Determine minimum and maximum as well as arithmetic mean $\bar{x}$ and squared mean(mean value of $x^2$) of the entire speech signal.

4. Generate a histogram of the speech signal using the function `histogram`. Set the parameters so that you receive a plot that covers the value range from $-0.25$ to $+0.25$ with a resolution of $1/500$.

5. Generate a random signal with the same length as the speech signal with each of the functions `rand`, `randn` and `randg` (you can find the latter in this week's folder, it must first be compiled once using the command "mex randg.c") and generate a separate histogram with the same settings as for the speech signal for all of these random signals. Modify the value range of the histogram so that it covers the entire signal by making use of the command `min` and `max`. What is the distribution of the different noise signals? What distribution can the speech signal best be described by?

### Exercise 1.2 Quantization

In this exercise, you will deal with a quantizer or, more precisely, a uniform quantizer. A quantizer maps an input value $x(n)$ to an output value $\hat{x}(n)$ where the set of possible output values is smaller than the set of possible input values. A uniform quantizer divides the input range up to some maximum value into equally sized sections and maps all signal values within one of these sections to exactly one corresponding output value. This mapping relation can be described by a characteristic curve. Figure 1.11 shows the characteristic of a uniform 3-bit quantizer which "rounds" the input values to the closest quantization level. For larger input values, the quantizer is saturated.

The quantization error is defined as

$$e(n) = \hat{x}(n) - x(n), \tag{1.1}$$

where the error is limited to the values $-\Delta/2 \leq e(n) \leq \Delta/2$, when $\Delta$ is the stepsize of the quantizer.

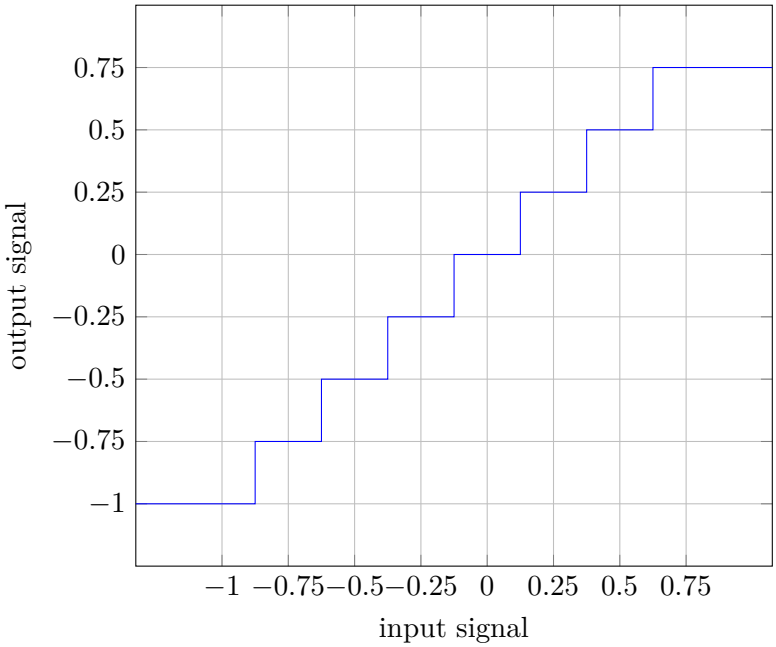



**Figure 1.11:** Characteristic of a uniform 3-bit quantizer

1. Normalize the input signal so that it does not exceed the value range $[-1,1]$ and perform a uniform quanitzation of the speech signal from the previous exercise with 8, 6 and 4 bit precision, i. e. 256, 64 and 16 valid steps. The input values are rounded to the closest quantization level in the process. The value range of the quantizer is assumed to be limited to $[-1,1]$. The quantizer characteristic should be analogous to Figure 1.11.

   **Note:** You might find the functions `find`, `round` and the relational operators helpful. The signal must be scaled first and rounded to integers. Then the signal must be limited to the correct minimum and maximum values. Then it must be scaled back.

2. Listen to the quantized signals.

3. Plot a section of 500 samples from the original signal and the three quantized signals in a diagram. The selected signal segment should be one where there is "a lot going on". Use the function `stairs` for the quantized signal segments.

4. Determine the error signal for the three quantized signals for the entire length of the signals.

5. Generate histograms for the error signals. Plot them together with the histogram of the original signal in a single diagram. What do you notice?

6. The signal is normalized to the value range $[-1,1]$. Set the value range of the quantizer to $[-0.01, 0.01]$ and apply the quantization with 8 bit once more. Keep in mind that the stepsize changes accordingly. Listen to the result.

7. Determine the indices of the samples that are in the saturation region of the quantizer. Think about how you can highlight these positions in a plot of the original signal and create such a plot.

## Exercise 1.3

A measure for the quality of a quantizer is the relation of the energies of the original signal and the error signal. This so-called SNR (Signal-to-Noise-Ratio) is defined as

$$\mathrm{SNR} = 10\log_{10}\left(\frac{\sum_{n=0}^{L-1}(x(n))^2}{\sum_{n=0}^{L-1}(\hat{x}(n)-x(n))^2}\right) \tag{1.2}$$

Compute the SNR for the four quantizers used in the previous exercise. Again, use the speech signal `speech.wav` as an input signal.

## Exercise 1.4 Amplitude modulation

In this exercise, you will perform an amplitude modulation and demodulation. First, a short introduction.

For the "regular" amplitude modulation, a cosine signal is modulated in its amplitude with the signal to be transmitted $x(t)$ according to equation (1.3).

$$x_{AM}(t) = a_0 \cdot (1 + m \cdot x(t)) \cdot \cos(\omega_0 t + \varphi_0) \tag{1.3}$$

The corresponding parameters are the amplitude factor $a_0$, the modulation index $m$ and the initial phase of the cosine signal $\varphi_0$. It is assumed that the modulating signal $x(t)$ has a limited amplitude $|x(t)| < 1$. Furthermore, the bandwidth of the signal $x(t)$ must be limited to a maximum bandwidth of $2\omega_0$ (i. e. cut-off frequency $\omega_g = 2\pi f_g < \omega_0$)

In the frequency domain, an amplitude modulation simply implies a shift of the signal to the left and right to a carrier frequency $-\omega_0$ or $\omega_0$ respectively. Here, we will employ the amplitude modulation with a carrier where the cosine with the carrier frequency is added additionally. Figure 1.12 illustrates this more a modulation index of $m = 1$.

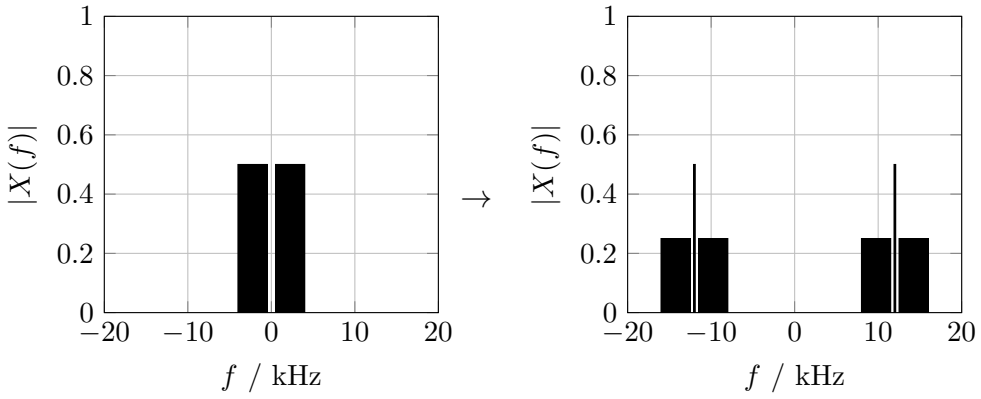



**Figure 1.12:** Amplitude modulation

In this experiment, the incoherent demodulation, also referred to as envelope demodulation, is used for the reception. Formally, the demodulation then looks as follows:

$$x_{demod}(t) = \frac{1}{a_0 \cdot m}\left(|x_{AM\_env.}(t)| - a_0\right) \tag{1.4}$$

where $x_{AM\_env.}(t)$ denotes the complex envelope of the modulated signal $x_{AM}(t)$:

$$x_{AM\_env.}(t) = x_{AM}^{+}(t) \cdot e^{-j\omega_0 t} = (x_{AM}(t) + j\mathcal{H}\{x_{AM}(t)\}) \cdot e^{-j\omega_0 t}. \tag{1.5}$$

The signal $x_{AM}^{+}(t)$ is the so-called analytical signal of $x_{AM}(t)$ and $\mathcal{H}\{\cdot\}$ is the Hilbert transform. Thus, the analytical signal is required to calculate the complex envelope of the corresponding bandpass signal. It is composed of the original signal as real part and its Hilbert transform as imaginary part. What makes the analytical signal special is that its spectrum is equal to zero for negative frequencies. It holds for the spectrum of the analytical signal of the function $x(t)$ that:

$$X^{+}(j\omega) = \begin{cases} 2X(j\omega) & \text{für } \omega > 0 \\ 0 & \text{für } \omega \leq 0. \end{cases} \tag{1.6}$$

As the signal has a component only for positive frequencies, it can easily be shifted to the baseband by multiplication of $e^{-j\omega_0 t}$. The result is then the envelope of the bandpass signal. Because the signal is complex in general, this is referred to as the complex envelope.

For the incoherent demodulation of an amplitude modulated signal, the absolute value of the complex envelope must be taken. In the following, it is $a_0 = 1$.

1. Generate a signal of length $t = 1$ s with a sampling rate of $f_s = 32$ kHz. The signal should be composed of four sine components of the frequencies 50 Hz, 110 Hz, 230 Hz and 600 Hz with the weights 1, 0.4, 0.2 and 0.05. Normalize the signal to the maximum amplitude 1.0, store it in a variable and display it graphically.

2. Generate a carrier signal with the same length and sampling rate as the composed signal. It should be a cosine with the frequency $f = 12$ kHz and a phase of $\varphi_0 = 0$.

3. Perform an amplitude modulation of the signal from point 1 according to equation (1.3). Use each of the modulation indices $m = 0.8$ and $m = 1.8$ once. Display the modulated signal, its envelope and the signal to be modulated graphically. Use `abs(hilbert(amsignal))` for the calculation of the envelope where `amsignal` is the modulated signal.

   Can the signal be received without errors for both modulation indices?

   The case $m > 1$ is called overmodulation. Why?

4. Demodulate the signals generated in the point before. To do so, first compute the corresponding analytical signal using the Hilbert transform (see function `hilbert`).Then, perform a frequency shift to the baseband by multiplying with $e^{-j\omega_0 t}$ in the time domain. Finally, take the absolute value of the signal, subtract the direct component $\frac{1}{a_0}$ and scale it with $\frac{1}{a_0 m}$.

   Keep in mind that the function `hilbert` returns the analytical signal and not just the Hilbert transform. Plot the result.

# Chapter 2

# Introduction to MATLAB II

In the preceding chapter, you have learned the fundamentals of handling MATLAB. However, it can get very tedious to always enter all of these commands on the console, particularly if you need a specific sequence of commands more often.

Just like there are shell scripts under UNIX/Linux and batch files under Windows, you can write so-called M-files in MATLAB. Similar to shell scripts, these contain a sequence of commands that you can execute by calling the filename from the command line (or from a different M-file) as if it was a regular MATLAB command. Parameters can also be passed when the M-file satisfies a specific format.

Beside scripts and functions, you will also get to know the MATLAB debugger in this chapter which helps you with the tracking of programming errors. Also, we will familiarize you with further aspects of graphical visualization.

## 2.1 Scripts and Functions

Rather than entering long code repeatedly on the MATLAB console, you can also write it in an M-file and run it via the name of this file – without the extension `.m`. A function call is possible only if the M-file is stored in the current directory or in a directory that is included in MATLAB's search path. You can assess what directories are part of MATLAB's path by using the command `path`. By default, these are the toolbox directories which must however not be used for self-written scripts and functions. You can extend the search path by additional directories in which your own functions are saved by using the command `addpath` in the following way.

```
>> addpath('c:\path\to\your\M-files\');  % oder
>> addpath c:\path\to\your\M-files\
```

This adds the specified directory at the front of the current MATLAB path. Afterwards, you are free to run all M-files from this directory independently from the current directory. The directories added with `addpath` are no longer included in the path after restarting MATLAB. You can use the `pathtool`, which can be started with the command of the same name, to permanently add directories to the path.

We distinguish two kinds of M-files: simple scripts and functions. Scripts are executed line by line in the global Workspace as if you had entered the command directly on the console. This also implies that variables that existed prior to calling

the script are available within the script as well. Changes to these variables and newly created variables persist even when the execution of the script is terminated.

Other than the filename, which is subject to the rules for variable names for all M-files and which must always end on `.m`, there are no special requirement for scripts. The same rules as for the input on the Command Window apply.

In contrast, functions must always start with the keyword `function`. The program code is executed in a Workspace of its own by MATLAB. The data transfer is usually carried out directly via the input and output parameters.

Line breaks

Before we get to the details of the function declaration, a note on the formatting of long command lines. You can spread such lines, which can easily get very confusing, over multiple lines by putting three dots at the end of the line. MATLAB then knows that this line is continued in the next one.

```
t = 1:1000;               % This text is ignored by MATLAB
r = sin(2*pi*f1/fa*t) ... % This line is continued in the next one
    + cos(2*pi*f2/fa*t);
```

### 2.1.1 Function Declaration and Passing of Arguments

If an M-file should behave like a function, it must contain a function declaration as first MATLAB expression. The function declaration starts with the keyword `function`. Then come the output parameters in square brackets followed by an equals sign. Multiple parameters are separated by commas or space characters. The function name, which should match the filename without the ending `.m`, is specified behind the equals sign. The input parameters, which are separated by commas as well and enclosed by parentheses, are the final part of the function declaration. The program code that follows the function headed is executed when the M-file is called.

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

In the program code, you can access the passed parameters using the names you have assigned to them. Note that these variables are local. Any changes to these variables do not affect the variables outside of your function.

It is a little tricky in MATLAB that the user does not receive an error message by default if he passes fewer arguments to the function than available in the list of parameters. The function is called as usual except that several parameter variables are not defined. However, if the number of arguments is too large, MATLAB returns an error message and terminates the execution of the function immediately.

To handle the variable number of input parameters or to enforce that a specific number of arguments is passed, you can check with the function `exist` whether a specific variable is defined. Alternatively, you can use the functions `nargin`, `nargout`, `nargchk` and `nargoutchk` for this purpose.

`nargin`

Within a function, the number of received input parameters is returned. For a function name as an argument, the number of arguments declared in the header of this function is returned.

```
function [x, y] = myfunc(a, b, c, d)
```

```
if nargin == 4
  % do something useful
else
  % do something else
end
```

```
>> nargin('fft')
ans =
     3
```

`nargout`
: Similar to `nargin` but for the output parameters.

`nargchk`
: Returns an error text if the number of passed arguments is not in the specified range. With the optional argument `'struct'`, the return value is a structure that contains an identifier additionally to the error text. The fields are `message` and `identifier`.

```
msg = nargchk(low, high, n)
msgstruct = nargchk(low, high, n, 'struct')
```

The argument *low* represents the smallest, *high* the largest number of accepted input arguments.

`nargoutchk`
: Similar to `nargchk` but for the output parameters.

It is also possible to write functions that can handle a variable number of input and output parameters. This can be achieved using the special variables `varargin` and `varargout`. If you use these as variable names in the list of parameters for the input or output parameters, MATLAB will generate a cell array from all parameters following the fixed ones. The order of the cells matches the order of the parameters in the function call.

You can return a variable number of output values by creating a cell array in the variable `varargout` and filling it with data. A compact example:

```
function [varargout] = nonsense(a, b, varargin)
s = [mat2str(a) mat2str(b)];

for k = 1:length(varargin)
  if ischar(varargin{k})
    s = [s varargin{k}];
  elseif isnumeric(varargin{k})
    s = [s mat2str(varargin{k});
  end
end

idx = regexp(s, ' *');

if idx(1) ~= 1
  idx = [1 idx];
end

idx = [idx, (length(idx) + 1)];

for k = 1:length(idx)-1
  varargout{k} = s(idx(k):idk(k+1)-1);
end
```

To maintain a clear structure within your M-file, you can move parts of it to sub-functions within the same file. These sub-functions start with the same keyword as the main function but they are visible within the M-file only. They are declared in the same way as the main function.

### 2.1.2 Comments and Help Texts

If you move complex code to a script, it is important to add comments so that others – but most importantly yourself – are able to reconstruct what the M-file is good for later on. In MATLAB, you start a comment with a percent sign (`%`). The effect is the same as a double slash (//) in C++, i. e. if a percent sign appears in a line, the remainder of the line is ignored unless the percent sign is part of a string surrounded by simple quotation marks. Empty lines are ignored as well.

Since MATLAB version 7.0, it is also possible to comment out entire blocks just as in C/C++. Such a comment block starts with `%{` and ends with `%}`. Such a construct is equivalent to the effect of `/* ... */` in C/C++. If your MATLAB programs should also work on older versions of MATLAB, you need to avoid this kind of commenting as it is not detected by MATLAB 6.5 or older. For the convenient commenting or the removal of commenting signs, MATLAB provides a simple keyboard shortcut. These shortcuts are explained in more detail in Section 2.2.1.

Providing help

You have already learned that the command `help` followed by the name of a function displays help for the respective function. This functionality can also be established for your own M-files as the function `help` does nothing but returning the first coherent comment block. So if you introduce the file with a help text in the form of such a comment block, you and every other user of your M-file can get this help text displayed without having to look into the source code of the file every time. The procedure is the same for both simple scripts and for functions. It is advisable to stick with the style of the MATLAB functions with respect to the format and structure when writing your own help texts.

If you wish to specify references to further functions, write the names of the functions in a line of its own that starts with `% See also`. If the listed functions exist in the search path, they are displayed as hyperlinks via which you can then directly get to the help of the respective function.

```
function [x, y] = foo (a, b)
% FOO -- A sample function
%
%   [X, Y] = FOO(A, B) takes two arguments A and B and returns two
%        arguments X and Y.  A and B have to be twodimensional
%        matrices. X and Y then have the same dimensions
%
%   [X] = FOO  returns some matrix X
%
%   See also BAR, ARB, RBA.
```

### 2.1.3 Control Structures

To write reasonable programs in MATLAB, you will need control structures that allow you to direct the program flow. The syntax of the different structures that are

available in MATLAB is presented in the following.

### if, else and elseif

```
if condition
  code
elseif condition
  code
else
  code
end
```

For an if structure, MATLAB first evaluates the expression *condition* behind the keyword `if`. If this expression is true or unequal to zero, the code is run until the next `elseif`, `else` or `end`, whichever of these keywords comes first. If the condition is not met, it is proceeded with the next `elseif` in the same manner. If neither of the conditions is satisfied, the code delimited by the keyword `else` and the end of the if structure is executed. Simple expressions can be combined with logical operators to construct more complex conditions.

Caution is advised when working with matrices and logical operators in the condition part of the if structure. Assume you write the following for two matrices `A` and `B`:

```
if A > B
   disp('A is greater than B')
end
```

While this condition is valid, it may not have the expected effect. As outlined in the previous chapter, MATLAB does not check for the entire matrix whether a condition is fulfilled but it processes each element individually and returns *where* and where not the condition is fulfilled.

If the condition in an if structure is applied to a matrix, *all* of the elements must be true or unequal to zero for the rest of the if structure to be executed. To test whether two matrices are equal, you can use the command `isequal(A,B)` instead of the double equals sign.

`isequal(A,B)`

### switch and case

If a lot of different cases should be handled for a specific expression, it is often more compact to use a switch structure rather than an if structure with lot of elseif branches.

```
switch switch_expression
  case case_expression
    code
  case {case_expression1, case_expression2, ...}
    code
  otherwise
    code
end
```

The result of the expression *switch_expression* must be either a scalar or a string for a switch structure. It is then compared with the case expressions one after another. If it matches one of the case expressions, the corresponding code is executed.

In MATLAB, only the code corresponding to one applicable case expression is run. This implies that unlike in C, there is no need to use some of break command to prevent that the code for all of the subsequent case expressions is run as well.

To still provide a possibility to run the same code for multiple case expressions without having to duplicate code, MATLAB allows the use of cell arrays consisting of scalars or strings additionally to pure scalars and string as case expressions. During the execution, *switch_expression* is then compared with each of the elements of the cell array and the corresponding code is run if there is a match for a at least one of the elements.

Finally, you can add some code that is executed if there is no match for any of the case expressions by introducing it with the specific keyword `otherwise`. The procedure is the same as for `default` in C/C++.

### for

```
for variable = expression
    code
end
```

In a for loop, the result of the expression *expression* behind the equals sign is written in the variable *variable* column by column. For each of the columns, the for loop is run once. This allows to construct powerful for loops in a very compact way. For example, you can use the colon operator (`for k = 1:N`) for a simple iteration over integer numbers.

### while

```
while expression
    code
end
```

A while loop is executed repeatedly until the expression *expression* is false or equal to 0. As with the if structure, it is possible to combine multiple conditions with logical operators. For these conditions, the same rules as for if structures apply.

### continue, break and return

With these three command, you can stop individual loop iterations or you can terminate loops and functions completely. The command `continue` interrupts the current iteration of a while or for structure and MATLAB will then continue with the following iteration. The command `break` stops not only the current iteration but also the surrounding while or for structure. To terminate a function before the end of the code is reached, you can use the command `return` which passes the control back to the calling function or the console.

### try and catch

You can use the try-catch structure to intercept function errors and to be able to appropriately react to them. This allows you to prevent MATLAB from exiting a

function early and presenting you some default error message due to some error in your function although you might have been able to react to the error and remove the error source in the function yourself. Another useful application is to provide additional information about what might have caused the problem once an error occurs.

```
try
  code
catch
  code
end
```

The program code between the keywords `try` and `catch` is executed as usual first. If no error occurs in the progress, the code between `catch` and `end` is skipped and MATLAB proceeds with the remainder of the program. Once an error occurs, MATLAB immediately jumps to the catch instruction and runs the corresponding code. At this point, you can add code that analyzes the error and either displays an informative error message or a warning. You will learn more about warnings and error messages in the following section.

### 2.1.4 Warnings and Error Messages

If you notice that something in your function is not right, e. g. the input data has a format that your function does not support, it may be advisable to return the function caller a helpful warning or even error message. The available functions to do so are `warning` and `error`. Calling `warning` does not affect the running program. The specified text is merely displayed. In addition to that, the M-file is terminated at this position when `error` is called instead.

Beside the error or warning message itself, a message ID can be specified that unambiguously identifies this message. There is no need to specifically code the message to be displayed in the source code. The text may comprise format instructions just as with the functions `sprintf` and `fprintf`, e. g. to dynamically include numbers or strings in the message text. The various possibilities to call the functions are listed in the following.

```
% Displaying a warning message
warning('message')
warning('msgid','message')
warning('msgid','message', A, B, ...)

% Displaying an error message and terminating the execution of the M-file
error('message')
error('msgid','message')
error('msgid','message', A, B, ...)
```

The parameter *msgid* should have the form `<component>[:<component>]:<mnemonic>` where `<component>` and `<mnemonic>` are alphanumeric strings with the additional constraint that the first character must be a letter in both cases. Examples for *msgid* would be the identifiers `MATLAB:divideByZero` or `Simulink:actionNotTaken`. As `<component>`, a rough categorization such as the toolbox name or the company name is normally used. To further divide messages into categories, multiple `<component>` elements can be concatenated using colon characters. The `<mnemonic>` identifies

the individual error or warning messages. If for example, you work on some toolbox ABC for the company XYZ and some error "fatalError" can occur in the current function, you could select `XYZ:ABC:fatalError` as the corresponding error message. Thus, the purpose of the `<component>` field is to ensure the uniqueness of the *msgid*.

There are two main applications of such an identifier. On the one hand, you can instruct MATLAB to suppress specific warning messages. This can be controlled through the command `warning` *state msgid*. To turn a specific message on or off, select `on` or `off` for *state* respectively. To check the current status, use `query` for *state*.

On the other hand, both the text message and the identifier can be requested programmatically with the commands `lasterr` and `lastwarn`. This allows for example to determine what error was triggered in the try part of a try-catch structure. Depending on the error, you can then take appropriate action in your M-file.

```
function matrix_multiply(A, B)
try
   Z = A * B;
catch
   [errmsg, errid] = lasterr;
   if(isequal(errid, 'MATLAB:innerdim'))
      disp('** Wrong dimensions for matrix multiply')
   elseif (strfind(errmsg, 'not defined for variables of class'))
      disp('** Both arguments must be double matrices')
   end
end
```

## 2.2 Editing and Debugging M-files

In MATLAB, you have two options to create and edit M-files. You can either use the included MATLAB editor or you can use an arbitrary external editor. To use an external editor directly from the development environment, you must tell this to MATLAB in the settings menu. Then, if for example you call the `edit` command on the console, the respective editor is started. In the following, we will asumme that you use the built-in MATLAB editor. As a debugger is also integrated in this editor, it is particularly suitable for a comfortable development-test cycle.

### 2.2.1 MATLAB Editor

Initially, the MATLAB editor is a completely normal editor in which you can edit arbitrary text files. When doing so, you can use the well-known standard functions and navigational possibilities such as search, search and replace, mark text, cut, copy, paste and more. When – which you are most likely to do most of the time in the MATLAB editor – editing M-files, you can choose from a wide range of additional features.

As can be seen immediately, the editor supports syntax highlighting. Keywords and comments are displayed in color. Moreover, the editor supports you with reasonable indention of both simple and more complex control structures. Like this, the editor makes it very easy for you to write clearly arranged code.

ments

As already indicated above, you can comment out or remove comment characters from multiple lines at once in the MATLAB editor. This can be done with the mouse by marking the lines and selecting **Text➔Comment** in the menu bar or **Comment** in the context menu that appears when right-clicking the marked text. Likewise, you can remove the comment characters from all marked lines by selecting the menu entry **Uncomment** instead of **Comment**.

Alternatively, you can use the keyboard shortcuts to get the same result even faster. You can comment out a marked a marked code block with **Ctrl-R** and remove the comment characters again with **Ctrl-T**. When nothing is marked, MATLAB applies the respective operation on the current line, no matter where in this line the cursor is.

Code sections

A feature of the MATLAB editor that is particularly interesting in the development stage are code sections. These allow you to run only specific blocks of code when running an M-file from within the editor. This is very useful when testing and comparing individual sections. The related options can be found in the **Run** part of the menu bar of the editor, in the **Editor** tab.

You can now arrange your code in sections. A new section starts with a comment line that is introduced with two percent signs. Behind these, you can give the section a title which is marked in bold. Moreover, you can use this title for easy navigation as all titles are listed upon clicking on the button with the two percent signs and the small arrow or 'Go To' in newer MATLAB vesions on the toolbar. By selecting the respective title, you jump to the chosen section.

You can explore in the **Run** part of the menu bar what other functionality code sections offer and how you can for example run code only of individual sections. Refer to the MATLAB help for further information.

### 2.2.2 Debugger

For convenient error search, MATLAB offers a debugger which can be opened both from the command line and from the MATLAB editor. First, we will deal with how to debug your program from the console. You can choose from the following commands:

| | |
|---|---|
| `dbstop` | Set a breakpoints. |
| `dbclear` | Remove a breakpoint. |
| `dbcont` | Resume program. |
| `dbdown` | Switch local Workspace context. |
| `dbstack` | List which function was called by which. |
| `dbstatus` | List all breakpoints. |
| `dbstep` | Run current code line. |
| `dbtype` | Return M-file with line numbers as output. |
| `dbup` | Switch local Workspace context. |
| `dbquit` | End debug mode. |

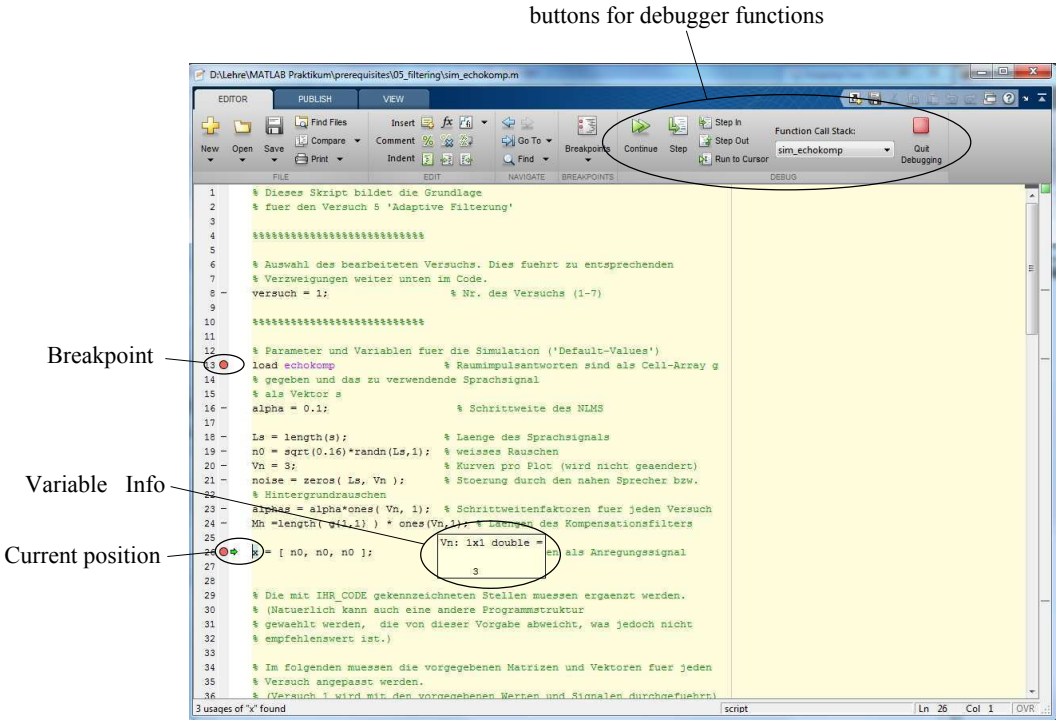



**Figure 2.1:** Editor window with active debugger

The typical procedure is to first set several breakpoints with commands like

```
dbstop in mfile at funcname
dbstop in mfile at lineno
dbstop in mfile
```

(refer to the corresponding help for further possibilities). You can get a list of all breakpoints that already set by using the function `dbstatus`. If you now call a function that you have set a breakpoint in, MATLAB interrupts the execution at the specified poisition and returns the control to the command line. The prompt changes from » to `K>` to indicate that MATLAB is in debug mode. You are in the local Workspace of the halted function and you can run any MATLAB function from there to inspect or even modify variables. With the command `dbstep`, you can go through your code line by line to observe whether or not the data is processed correctly. With `dbcont`, the execution of the function will be resumed until the next breakpoint is reached or until the end if there are no further breakpoints. To end the debug mode and return to the main Workspace, enter `dbquit` on the console. To find out what the other commands do, you can use the respective documentation.

Debugging from the editor

More conveniently, you can debug directly from the MATLAB editor as the debugging functionality has been integrated directly in its graphical user interface. Here, you can easily set breakpoints, get variable values displayed, run the code step by step and much more using the mouse and/or keyboard shortcuts.You can access the debugger functions for setting breakpoints, starting and stopping the debugger and running the code step by step via the buttons in the toolbar (see Figure 2.1) or via the keyboard shortcuts listed in the following table.

| | |
|---|---|
| **F5** | Start M-file |
| **F6** | Run a single line before returning to debug mode again |
| **F7** | Jump into a function, then interrupt in the first line |
| **Shift-F7** | Jump from the current function to the calling one. |

In the editor, lines in which a breakpoint can be set, are marked by a minus sign behind the line number. When a breakpoint is set in some line, a red circle appears at this position. You can easily turn on and off a breakpoint by clicking on the minus sign or the red circle respectively.

You can inspect the contents of a variable from the editor by hovering over the respective variable with the mouse. After a short time, a pop-up window will appear that shows the contents of the variable (or only a part thereof in case of bigger data structures).

During the execution of an M-file in debug mode, the line that will be run next is marked by a small green arrow at the left side.

## 2.3 Advanced Visualization

In chapter 1, you have learned how to create simple 2D and 3D plots. In the following, we will familiarize you with advances techniques of visualization under MATLAB. We will present two interfaces to you, namely the functional interface that you use from the command line and in M-files as well as the interactive interface that allows you to make modifications directly from the figure window. Due to the abundance of possibilities, we cannot cover all of them in this laboratory and therefore advise you to use the MATLAB help for further possibilities and information.

### 2.3.1 Customizing and Designing Graphs

All properties of a graph can be modified with the function `set` and appropriate parameters. It has the following calling convention: `set`

```
set(handle, 'propertyName', propertyValue);
set(handle, 'propertyName', propertyValue, 'properyName', propertyValue, ...);
```

The argument *handle* is a so-called handle – similar to a pointer – to an object the properties of which you wish to modify, *propertyName* represents the name of a property of the specified object and *propertyValue* the new value of this property. Within one call, it is also possible to change an entire series of properties of arbitrary length so that there is no need to call `set` once for each of the properties to be modified.

Just like you can set properties with the function `set`, you can also inquire the current value with the function `get`. The only difference in the calling procedure is that you do not specify values for the property and that you can only request for one property at a time. `get`

```
get(handle, 'propertyName');
```

get

You can get an overview over all available properties and their current values by calling the function `get` with the handle as the only function parameter.

```
get(handle);
```

The functions `get` and `set` are very powerful as they allow you to access almost all properties of an object. All you have to know it the object and the name of the property. This however is exactly what makes using the two functions a little tricky because how can you get the handle of the desired object and how are you supposed to know what properties the object has at all and what the name of the property is that you wish to change?

As it can get very tedious to collect this information again and again – you will get to know several commands for this purpose further below –, there are special functions for a few of the most commonly set or modified properties to save you a lot of typing effort and detective work. Five of these, you have already seen in the first chapter, namely `xlabel`, `ylabel`, `zlabel`, `title` and `legend`. For the example of the `title` function, it is shown here what kind of typing effort can be avoided by using this special function rather than `set`/`get`.

```
ax = gca;
h = get(ax,'title');
set(h, 'FontAngle',  get(ax, 'FontAngle'), ...
       'FontName',   get(ax, 'FontName'), ...
       'FontSize',   get(ax, 'FontSize'), ...
       'FontWeight', get(ax, 'FontWeight'), ...
       'Rotation',   0, ...
       'string',     string, varargin{:});
```

Im this, the variable `string` contains the title that you entered and the expression `varargin{:}` passes the optional parameters on to `set`. The command `set` sets a few of the text properties to the default values of the surrounding axes object that are acquired with the `get` command.

gca

For you, particularly the first two lines will be interesting. In these, the required object handles are determined. The command `gca`, which stands for "get current axes", returns a handle for the currently active axes object. In the second line, the title property of the axes object is read out which contains a handle to a text element that in turn contains the title itself. With the subsequent `set`, the string property – the text that actually appears – and a few properties of the text element are set.

gcf

Beside `gca`, `gcf` is an important function in this context as it returns the handle to the active figure window. To get an overview over the different properties of an object or to look up the name of a property again, you can call `get` with a handle only. This returns a list of all properties of the specified object with the corresponding currently set values.

There is an entire series of other useful functions to get the objects of a graph and their handles. Look for "Handle Graphics" and function names like `allchild` and `findall` in the help.

In the following, we will go into the most important properties of the axes object which contains the actual graph and we will describe how you can set them.

### Axes object and its properties

The axes object is a kind of container for the various graphics objects like lines, shapes and text. Through the properties of axes, you can customize the orientation and scaling of your graph to achieve the desired effect. Moreover, you can access the elements included in the axes objects via the `Children` property.

As you have already learned, you can add axes labeling and a title to a graph or, more specifically, to the axes object. These elements can be addressed via the properties of the axes object, but can best be set by means of the already known special functions.

Furthermore, MATLAB automatically determines properties like the value range of the axes, the placement of the tick marks and their labeling upon creating a new graph. To inspect the results within MATLAB, this is usually sufficient but, particularly if you wish to optimize your graphics for a report or a documentation, you will need to access these parameters.

Value range

The commands adjusting the value range are `xlim`, `ylim` and `zlim` (corresponding properties of the axes object: `XLim`, `YLim` and `ZLim`). All three are called with one parameter which is a vector of two elements. The first element represents the minimum value, the second argument the maximum value to be displayed.

```
ylim([-10 10]);
% or equivalently ...
a = [-10 10];
ylim(a);
```

Another command that has an influence on the scaling of a graph is `axis` (not to be confused with `axes` which creates a new axes object). You can again set the limits of the x, y and z axis via the parameters of the function. Furthermore, you can influence the aspect ratio of the axes. This makes it possible to e. g. display a circle so that it really has a circular shape rather than looking like an ellipse (`axis equal`). To have an even better control over the aspect ratio, you can modify the axes property `DataAspectRatio` which is a vector of the exact ratio between the scaling of the axes.

Direction of ascending values

You can influence the direction in which ascending values are plotted on an axis by means of the axes properties `XDir`, `YDir` and `ZDir`. The property can take on the two values `'normal'` and `'reverse'`. There are no special functions for setting these values so that you will have to explicitly set them using the function `set`. If you want to e. g. invert the direction of the x axis in the current graph, use the commands from the following example. It also shows an application of the function `axis` with the parameter `tight` which automatically makes the value range of the axes as small as possible. The result is shown in Figure 2.2.

```
plot((0:100).^3);
axis tight;
set(gca, 'XDir', 'reverse');
```

Logarithmic scale

To display the graph's value range logarithmically, there are two possibilities. On the one hand, you can use one of the commands `loglog` (both axes), `semilogx` (x axis) or `semilogy` (y axis) instead of `plot` for 2D plots. These commands work exactly like `plot` except for the logarithmic scale instead of the linear one.

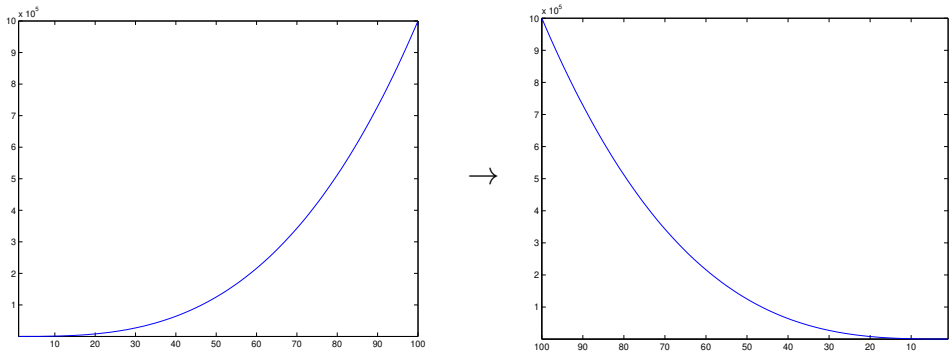



**Figure 2.2:** Inverting the direction in which the values are plotted on the x axis.

On the other hand, you can set the properties `XScale`, `YScale` and `ZScale` of the axes object differently, with the possible values being `'linear'` and `'log'`, when the plot already exists.

Subplots

Until now, you have always used exactly one axes object per figure object. Commonly, it is also reasonable to include multiple plots in a single figure, for example to display magnitude and phase of the Fourier transform of a time domain signal separately. Use the command `subplot` to create multiple axes objects within one figure window.

```
subplot(m, n, p);
```

This command divides the figure window into a grid of *m* rows and *n* columns. `subplot` creates a new axes object in the grid field specified by the third parameter `p`. The fields are – unfortunately unlike the counting method that is otherwise used by MATLAB – numbered line by line. Thus, the command `subplot(3,2,4)` creates an axes object in the second grid field of the second row. Once `subplot` is called, the newly created axes object becomes the active one and all following plot commands will create their graph in it.

A special property of the `subplot` command is that already created axes objects are not modified unless they overlap with a new object. If such a collision occurs, the affected axes objects are deleted and the new object is created at the specified position. This makes it possible to arrange axes objects so that they effectively extend across multiple grid fields. This is illustrated in the following example and the resulting Figure 2.3.

```
% Grid of 3x2 fields, new axes at the top left
subplot(3,2,1); plot([0:10],[0:10]);

% Grid of 3x2 fields, new axes at the top right
subplot(3,2,2); plot([-10:10].^2);

% Grid of 3x1 fields, new axes in the middle
subplot(3,1,2); plot(freq,mag);
ylabel('Betrag'); grid on; xlabel('Frequenz [Hertz]');

% Grid of 3x1 fields, new axes at the bottom
subplot(3,1,3); plot(freq,phase);
ylabel('Phase'); grid on; xlabel('Frequenz [Hertz]');
```

Tick marks

Normally, MATLAB chooses the position of the tick marks and their labels automatically so that the displayed value range is evenly divided and so that the marks are not too close to each other. By changing the axes properties `XTick`, `YTick` and `ZTick` for the position of the marks and the properties `XTickLabel`, `YTickLabel` and `ZTickLabel` for the labels, you can also specify these yourself.

Grid lines

Often, it improves the readability of a graph when grid lines are added. You can control this using the command `grid`. The command accepts the parameter `'on'`, `'off'` and `'minor'`. If it is called without parameters, the grid lines for all axes are turned on or off again. The parameters `'on'` and `'off'` explicitly control turning on and off the grid. The grid lines are always set to the positions of the axes' tick marks. Moreover, a so-called minor grid can be turned on via the parameter `'minor'` so that grid lines are also drawn between the ticks marks.

To turn the grid on and off separately for the different axes, you must set the axes properties `XGrid`, `YGrid` and `ZGrid` or `XMinorGrid`, `YMinorGrid` and `ZMinorGrid`. The lines style can be set independently for the broad and the fine grid via the axes properties `GridLineStyle` and `MinorGridLineStyle`. An example which illustrates the use of tick marks and grid lines is given in the following.

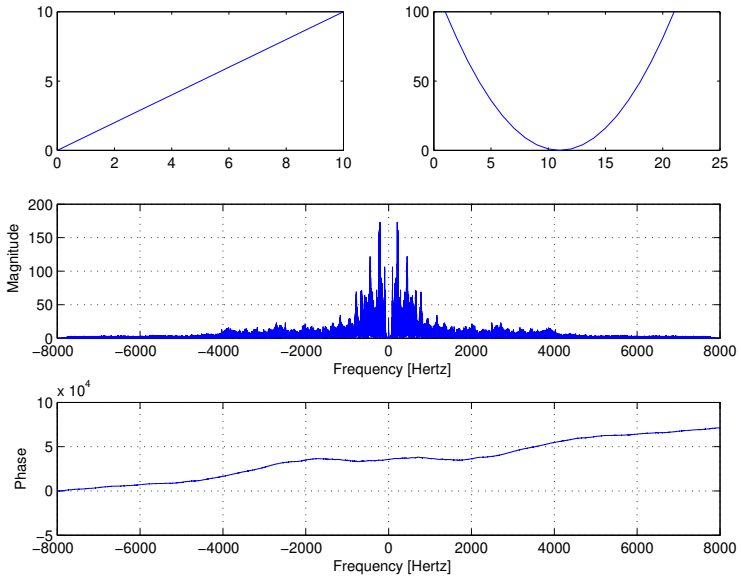



**Figure 2.3:** Use of the command `subplot` for the creation of multiple graphs in one diagram.

```
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
set(gca, 'XTick',-pi:pi/2:pi)
set(gca, 'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
grid on
set(gca, 'XMinorGrid', 'on')
set(gca, 'GridLineStyle', '--')
set(gca, 'MinorGridLineStyle', ':')
```

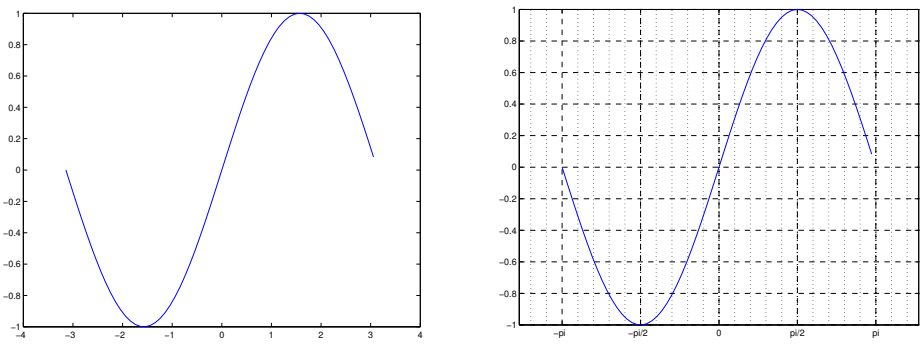



**Figure 2.4:** Changing ticks marks and turning on grid lines.

## Lines with style

As you have already learned in the first chapter, MATLAB automatically uses different colors for the individual curves when drawing them in a graph. However, you can also set the color and various other line properties yourself. Table 2.2 lists the properties that you can choose from.

**Table 2.2:** The most important line properties

| Property | Description and possible values |
|---|---|
| `Color` | Color of the line: ColorSpec, i. e. RGB values in a vector of length 3 (e. g. [1 0 0] for red) or a pre-defined name in short or long form (`'y'\|'yellow'`, `'m'\|'margenta'`, `'c'\|'cyan'`, `'r'\|'red'`, `'g'\|'green'`, `'b'\|'blue'`, `'w'\|'white'`, `'k'\|'black'`) |
| `LineStyle` | Line type: `'-'` (solid), `'--'` (dashed), `':'` (dotted), `'-.'` (dash-dotted), `'none'` (no line) |
| `LineWidth` | Line width: number (measuring unit: points; 1 point = 1/72 inch) |
| `Marker` | symbols at data points: `'+'` (plus sign), `'o'` (circle), `'*'` (asterisk), `'.'` (point), `'x'` (cross), `'^'`, `'v'`, `'<'`, `'>'` (triangle pointing up, down, left or right), `'s'\|'square'` (square), `'d'\|'diamond'` (lozenge), `'p'\|'pentagram'` (five-pointed star), `'h'\|'hexagram'` (six-pointed star), `'none'` (none) |
| `MarkerEdgeColor` | Edge color of markers: ColorSpec (see Color) |
| `MarkerFaceColor` | Fill color of markers: ColorSpec (see Color) |
| `MarkerSize` | Size of the markers: number (measuring unit: points) |

It is easiest to specify the line properties directly upon plotting with the `plot` command. For this, there are two possibilities for calling the plot command.

```
plot(x, y, 'PropertyName', 'PropertyValue', ...)
plot(x, y, 'LineSpec', 'PropertyName', 'PropertyValue', ...)
plot(x1, y1, 'LineSpec', x2, y2, 'LineSpec', ... , 'PropName', 'PropValue', ...)
```

The property/value pairs set the specified properties of *all* of the data sets plotted in this command. You can set the properties `Color`, `LineStyle` and `Marker` separately for each data set via the value of *`LineSpec`*. *`LineSpec`* is a string in which you can set the values for the three properties – without space characters in between – in arbitrary order. There is no need to always include all three, you can also set only part of the properties.

You can still set the line style differently after plotting using `set` and `get` as well as the respective properties. To do so, you need the handles to the line objects which you can acquire in two ways: either directly as a return value when calling `plot`, which will give you a vector containing the handles, or later on, via the `Children` property of the axes object. The property `Children` contains a vector of all the objects in the axes object, which means that these are not necessarily line objects only. You can identify the line objects by their `Type` property which then has the value `line`. You can use the command `findall` to get all objects of type `line` within one axes object.

```
>> x = [(1:10)', (2:11)'];
>> plot(x)
>> h = plot(x)
```

```
h =
  163.0021
  164.0011
>> findall(gcf, 'Type', 'line')
ans =
  164.0011
  163.0021
```

Finally, there are commands for adding texts, arrows, lines and more to plots. There are two different types of "annotation objects": those for which the position is specified in *data coordinates* and which move when the graph is scaled and those which are positioned relative to the dimensions of the figure window. The former are created and positioned with commands like `line`, `text` and `rectangle`. You can create and place the latter with the function `annotate`.

As particularly adding such annotation objects can much more conveniently be done from the graphical tools, we will not go into more depth on this matter at this point.

### 2.3.2 Interactive Visualization

Now, you have learned how to create graphics and to customize them according to your wishes by means of adding labels and similar things the traditional way via the console. Additionally, MATLAB offers an edit mode in the figure window that allows you to zoom into the graph, add and edit labels, change line parameters and much more. From this interface, you can customize a graph in every detail and add annotations.

Plotting tools

From version 7.0 on, this interface is extended by so-called plotting tools. These complement the former edit mode by additional GUI elements that can be turned on and off and from which you have more direct access to the different parameters. With the command `plottools on`, you turn on all available plotting tools in the current figure window. They can be turned off with `plottools off`. The plotting tools can also be turned on and off from the corresponding buttons on the toolbar. Individual plotting tools can also be added or removed in the **View** menu via the menu points **Figure Palette**, **Plot Browser** and **Property Editor**. You can see a figure window with plotting tools turned on in Figure 2.5. In the following, the three main elements are presented in more detail.

#### Figure Palette

The subwindow commandFigure Palette is further divided into the sections `New Subplots`, `Variables` and `Annotations`. The section `New Subplots` provides you with tools for adding new 2D or 3D axes objects to your figure window. You can choose whether you want to add the axes objects separately or multiple at once in the order that you specified. The former is achieved by clicking on the field `2D Axes` or `3D Axes`, the latter by clicking on the icon with the square grid on the right which opens a small window in which you can select the desired number of rows and columns.

The section `Variables` lists all Workspace variables. From here, you can transfer data to axes objects that should then be plotted. Right-clicking a variable yields

the same context menu that you would have received upon clicking on it in the Workspace. Several commonly used plot commands are available directly on the first level. You can also click on **More plots** to get an overview of all available plot commands from which you can then choose one. The data is then plotted in the currently marked axes object.

Additionally, you can also plot data per drag and drop. To do so, drag the desired variable – or multiple variables – on an axes object whereupon the contents of this variable will be added to the current axes object with the default plotting method.

Finally, the section `Annotations` provides you with all tools for adding any kind of annotation such as arrows, texts or similar things. After having clicked on the desired tool, the mouse pointer turns into a cross within the figure area and you can add the selected elements.

### Plot Browser

The subwindow `Plot Browser` shows for each axes object what data sets it currently comprises. When an axes object is selected, you can add more data sets to this object via the button **Add Data ...**. The purpose of the checkbox next to each of the entries is to allow you to show and hide the corresponding curves in the graph. A data set can be deleted completely from the context menu (after a right-click) or by pressing the **Delete** key on the keyboard.

### Property Editor and Inspector

From the Property Editor, you can access the most important properties of the selected object. When no object is selected, the Property Editor shows the properties of the figures. From here, you can modify the properties according to your wishes.

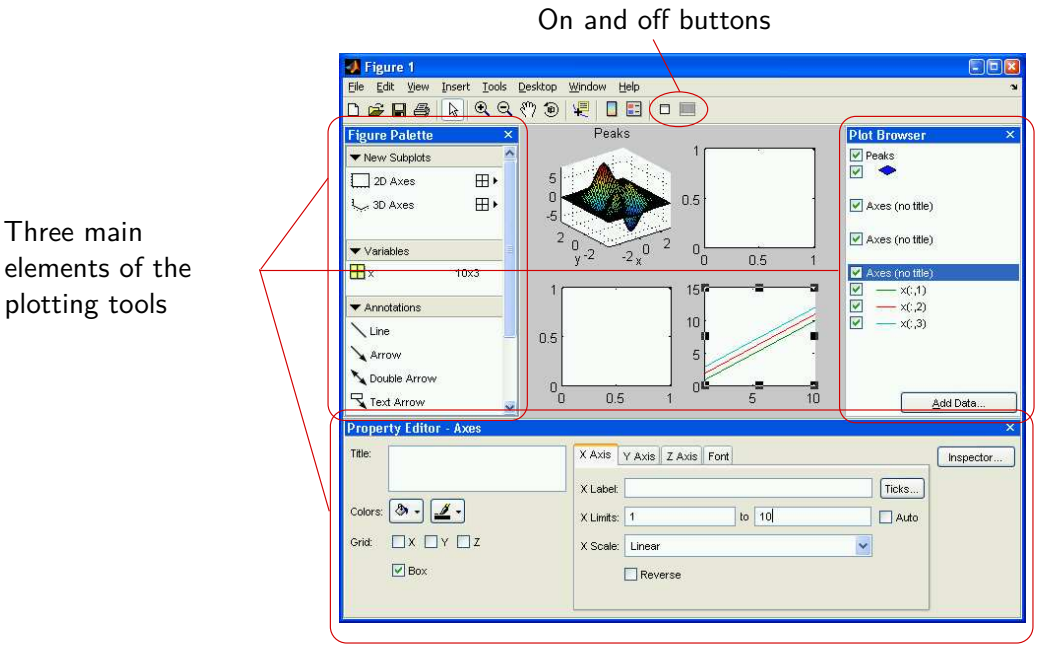



**Figure 2.5:** A figure window with plotting tools turned on

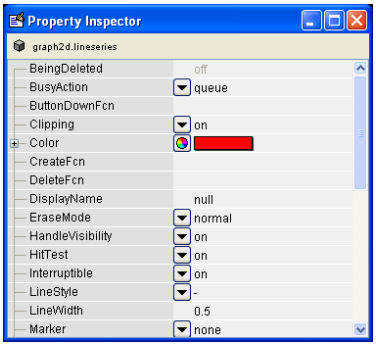



**Figure 2.6:** Property Editor of the Plotting Tools and Property Inspector

All changes are applied immediately so that you can inspect the result. To get access to all object properties, you can click on the **Inspector** button. The Property Inspector will then open which lists all of the object's properties and their values in a table. An example for a Property Inspector window is shown in Figure 2.6.

### 2.3.3 Printing, Saving, Exporting

You can not only admire the graphs you created with MATLAB on the screen, you can also print them. You can open the print dialog via the menu item **File➔ Print...** or the respective button on the toolbar. The available options are mostly self-explanatory.

FIG file

To avoid having to generate important graphs again and again or to further process them in other programs, it is important that you can also save them. MATLAB offers a large variety of settings and output formats to do so. If a graph should later again be displayed and processed in MATLAB, it is advisable to store it as a FIG file. This binary format which is native to MATLAB stores all the information about the data and exact formatting so that MATLAB can reconstruct it later and you will be able to make changes.

However, this format is not suitable for the use in programs of different developers. Instead, you can make use of MATLAB's export function which makes it possible to convert your graphs into common formats like e. g. TIFF or EPS. In particular, formats that store the image as a vector graphics – like EPS or EMF – are very useful as you will then be able to arbitrarily scale the image without suffering from a loss of quality.

Save As...

You now have several options to export a MATLAB graphics to an image file. From the figure window, you can click on the menu item **File➔Save As...** and select the desired file format in the following dialog. However, you then have no access to

Export Setup

further parameters. The second option is the `Export Setup` dialog which can be reached from the menu entry **File➔Export Setup...** This dialog lets you access all parameters that are important for exporting. Among these are the size, the aspect ratio, color models and much more. In the lower part, you can choose from various sets of settings, so-called export styles, or define and save some yourself. This is particularly useful if you wish to use very specific settings more often.

`print`

Finally, the third option is to save the graph in a file using the `print` function. This makes sense particularly when you generate a lot of graphs with scripts and you

then wish to automatically save them as a graphics files. For the selection of the file format and the adjustment of the large diversity of options, there is a great number of possible parameters. For more detailed information, refer to the `print` help.

M file

Beside the saving in the MATLAB FIG format and the export to various graphics formats, there is also a very interesting alternative which allows you to, effectively, save the settings of the plot. In the **File** menu, there is an entry **Generate M-File...** When creating an M-file to a graph this way, all settings that deviate from the default settings are added here as program code. You can then save the generated file under an arbitrary file name. Subsequently, you can use this file as a kind of modified plot function.

This functionality is particularly useful if you want to plot many data sets of the same type using the same settings. You can also use this function to find out how to apply specific effects via MATLAB functions rather than the graphical user interface.

## 2.4 Exercises

In the following exercises, you will build a channel encoder and the corresponding decoder. First some theoretical fundamentals.

### Channel coding with linear block codes

A generator matrix is needed for the encoding. In this laboratory, we will only cover binary codes, i. e. codes in the $GF(2)$ (Galois field). In the literature, you can also find the notation $\mathbb{F}_2$. This denotes the so-called *residue field modulo 2*. We will not elaborate further on the exact definition of a residue field modulo 2 here but merely present the most important properties. This *fields* consists of exactly two elements which are called 0 and 1. Note that these do *not* correspond to the real number 0 and 1. The following rules apply:

Addition:

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Multiplication:

| · | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Linear block codes can always be described as systematic codes so that the code word $\mathbf{y}$ of length $m$ corresponding to an information word $\mathbf{x}$ of length $n$ is composed of the information word and $k = m - n$ check bits. The information word is thus contained in the code word.

A linear block code is fully described by a so-called generator matrix $\mathbf{G}$. For an $(m,n)$ block code, it comprises $n$ rows and $m$ columns. The mapping rule of an information word $\mathbf{x}$ of length $n$ to a code word of length $m$ is

$$\mathbf{y} = \mathbf{x} \cdot \mathbf{G}. \tag{2.1}$$

The code word $\mathbf{y}$ can be interpreted as a linear combination of the rows of the generator matrix $\mathbf{G}$. If a generator matrix is non-systematic, it can be converted to the so-called *Gaussian normal form* by applying elementary row operations. The valid row operations are:

- Exchanging two rows
- linear combination of two rows

Gaussian normal form

A systematic generator matrix in Gaussian normal form satisfies the equation

$$\mathbf{G} = (\mathbf{E}_{n,n} | \mathbf{P}_{n,m-n}). \tag{2.2}$$

In this expression, $\mathbf{E}_{n,n}$ denotes the identity matrix with $n$ lines and $n$ columns, $\mathbf{P}_{n,m-n}$ a parity matrix with $n$ lines and $m-n$ columns. Such a generator matrix always leads to a systematic code as the identity matrix in the left part first reproduces the information word and the parity matrix then generates check symbols.

On the receiving end, the so-called parity-check matrix is needed to test a received code word for validity. This allows to detect errors that occurred during the transmission to some extent and, if applicable, even to correct them. From the Gaussian normal form, the parity-check matrix $H$ is given as

$$\mathbf{H} = \left(\mathbf{P}^T_{n,m-n} | \mathbf{E}_{m-n,m-n}\right). \tag{2.3}$$

A received code word $\mathbf{y}$ is valid only if the condition

$$\mathbf{y} \cdot \mathbf{H}^T = \mathbf{s} = \mathbf{0} \tag{2.4}$$

is met. $\mathbf{s}$ is the so-called syndrome. If $\mathbf{s}$ is unequal to zero, it can be used for the detection and, if applicable, for the correction of errors. The number of bits errors that can be detected or corrected is determined by the so-called minimum Hamming distance $d_{min}$ of the code. The Hamming distance between two code words is the number of bit positions where the two code words are different. The minimum Hamming distance is therefore the smallest Hamming distance that results from comparing all valid code words. However, for linear block codes, there is no need to compare all code words with each other, as it holds that the minimum Hamming distance is equal to the minimum of the weights of all code words except the all-zero code word in this case. The weight is the number of ones. The number of detectable bit errors is then $d_{min} - 1$, the number of correctable bit errors $e$ is

Hamming distance

weight

$$e = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor. \tag{2.5}$$

For the correction of the up to $e$ bit errors, their position within the received code word is required. This position can be determined with the help of the syndrome $s$. For up to $e$ bit errors, the syndrome is unambiguous and corresponds to a specific error pattern with which the correction can be conducted. This correspondence is usually stored in a syndrome table. You get the syndrome table by computing the syndromes for all possible error patterns. One possible syndrome table for single errors is shown in table 2.3.

| Error vector $\mathbf{e}_i$ | Syndrome $\mathbf{e}_i \cdot \mathbf{H}^T$ |
|---|---|
| 0 0 0 0 0 0 0 | 0 0 0 |
| 0 0 0 0 0 0 1 | 0 0 1 |
| 0 0 0 0 0 1 0 | 0 1 0 |
| 0 0 0 0 1 0 0 | 1 0 0 |
| 0 0 0 1 0 0 0 | 1 0 1 |
| 0 0 1 0 0 0 0 | 0 1 1 |
| 0 1 0 0 0 0 0 | 1 1 0 |
| 1 0 0 0 0 0 0 | 1 1 1 |

**Table 2.3:** Syndrome table

An important subclass of the linear block codes are the cyclic block codes. A block code is called cyclic if from every code word

$$\mathbf{y} = (y_{m-1}, y_{m-2}, \ldots, y_1, y_0), \tag{2.6}$$

a cyclic shift of the bits yields a code word

$$\mathbf{y}^{(1)} = (y_0, y_{m-1}, y_{m-2}, \ldots, y_1) \tag{2.7}$$

which is again valid. Such an ($m$,$n$) code is fully described by a so-called generator polynomial

$$G(z) = g_0 + g_1 \cdot z + \ldots + g_{m-n-1} \cdot z^{m-n-1} + g_{m-n} \cdot z^{m-n}. \tag{2.8}$$

The coefficients $g_i$ are elements of the $GF(2)$. It will not be further considered here how a code is generated from this polynomial. At this point, we are only interested in the representation as a generator matrix. For every generator polynomial of a cyclic code, this generator matrix can easily be created from the coefficients of the generator polynomial.

$$\mathbf{G} = \begin{bmatrix} g_0 & g_1 & \cdots & \cdots & g_{m-n} & 0 & 0 & \cdots & 0 \\ 0 & g_0 & g_1 & \cdots & g_{m-n-1} & g_{m-n} & 0 & \cdots & 0 \\ 0 & 0 & g_0 & \cdots & \cdots & \cdots & g_{m-n} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & g_0 & \cdots & \cdots & \cdots & g_{m-n-1} & g_{m-n} \end{bmatrix} \tag{2.9}$$

However, this generator matrix is not systematic. If a systematic code should be generated, the non-systematic generator matrix can be converted to the systematic form with the row operations listed further above.

## Exercise 2.1 Channel coding – Linear block codes

The M-files that you will implement in this and the subsequent exercises should all comprise an informative help text. Moreover, intercept possible invalid inputs and inform the user with appropriate error messages.

1. Write a function `zyklgenmat` that returns the non-systematic generator matrix to a given generator polynomial of a cyclic ($m$,$n$) block code. As first input parameter, this function receives a vector of length $m - n + 1$ which contains the coefficients of the generator polynomial. The first element of the vector corresponds to the lowest-order coefficient, the last element corresponds to the highest-order coefficient. The second argument specifies the code word length $m$.

   Test your function with the generator polynomials

   $$G_1(z) = 1 + z + z^3, \qquad m = 7, \qquad n = 4$$
   $$G_2(z) = 1 + z + z^2 + z^4 + z^5 + z^8 + z^{10}, \qquad m = 15, \qquad n = 5$$

2. Write a function `genmatsys` that converts a given generator matrix to the systematic form. Extend the function `zyklgenmat` from the first part of the exercise so that it returns the generator matrix in the systematic form if desired (can be specified through an optional parameter).

**Note:** In `genmatsys`, you can assume that the received matrix was created by `zyklgenmat` (triangular shape according to equation (2.9)). To convert the matrix to the systematic form, produce an $(n \times n)$ identity matrix in the left part of the matrix using row operations. Use only modulo 2 additions as row operations.

3. Extend the function `zyklgenmat` so that it also returns the parity-check matrix if a second return parameter is specified. Test the function with the generator polynomials given above.

   **Note:** The parity-check matrix can easily be constructed from equation (2.3).

4. Write a function `encoder` which computes the code word $\mathbf{y}$ to an information word $\mathbf{x}$ and a generator matrix $\mathbf{G}$. Take the rules of the $GF(2)$ into account and use vector and matrix operations if possible.

   In the second step, you extend the function so that it can not only process a single but also multiple information words at a time. The information and code words are represented in matrix form

   **Note:** The individual positions of the information and code words should be arranged in ascending order in the MATLAB vectors from the LSB to the MSB. For example, the information word $x$ is then `x(1)` $= x_0$, `x(2)` $= x_1$, etc.

5. Write a function `codefeatures` that determines the minimum Hamming distance and the number of correctable bit errors of a code with a given generator matrix.

   **Note:** In this function, you should first generate all code words and determine the weight of these. The minimum weight is equal to the minimum Hamming distance. For the conversion of natural numbers to their binary representation (vector of ones and zeros) and vice versa, use the functions `de2bi` and `bi2de` respectively.

6. Write a function `syntab` that returns the syndrome table to a given parity-check matrix and a number of correctable errors $e$. Test this function with the parity-check matrices you have computed with the function `zyklgenmat`.

   **Note:** Think about how to generate a matrix with all error patterns first.

7. Write a function `decoder` that determines the information word $\mathbf{x}$ corresponding to a valid code word $\mathbf{y}$ of a systematic $(m,n)$ code.

   In the next step, extend the function so that multiple code words in a matrix can be processed at the same time, just like the `encoder` function.

8. Extend the function `decoder` so that errors in the code word can be corrected and test your function.

   **Note:** Your function needs to support the correction of multiple bit errors, if the used block code, i.e. the used generator matrix, allows for it.

9. Write a script with the help of which you can run the solutions of this experiment in separate code sections.

# Chapter 3

# Signal Analysis

An aspect of digital signal processing that is particularly important is the analysis of statistical signals. These are for example speech and video signals, interferences and noise signals. First of all, the statistical analysis of such signals allows the characterization of their statistical properties such as the mean value, variance, correlation or the like. This characterization then serves as a basis for algorithms, e. g. for noise reduction, echo cancellation, compression or signal coding.

This laboratory experiment deals with the possibilities for signal analysis that MATLAB has to offer. The given statistical signals are discrete in both time and value here. In the context of this experiment, operators and functions are introduced that are provided by the *Signal Processing Toolbox* used in this laboratory.

It will first be dealt with the statistical properties of random signals. Their generation and handling under MATLAB will be explained and important operators and functions for the generation and evaluation of histograms will be introduced. Furthermore, functions for the analysis of the correlation of signals are considered.

The second part of this experiment deals with the frequency domain transformation of time signals. For this purpose, functions for the discrete Fourier transform (DFT) of time domain signals are presented. Another important aspect in the context of frequency domain transformation is windowing. For this as well, MATLAB offers numerous functions which will be described here. Finally, we will deal with the visual representation of the transformed signals in so-called spectrograms.

The explanations of the presented operators and functions as well as the given examples merely serve as a basis for the fundamental comprehension. For further possibilities of the presented operators and functions and references to thematically related functions, using the MATLAB help system is advised.

## 3.1 Statistical Properties

In this section, we will consider important properties of discrete signals. You will learn what possibilities MATLAB has to offer for the analysis of these properties. There is no claim to be exhaustive for what is presented in this chapter as a further elaboration is beyond the scope of this laboratory experiment.

### 3.1.1 Generation of random signals

In digital signal processing, random signals play a central role. An important class of random sequences are noise sequences for the modeling of disturbances such as quantization noise, channel noise or the like in digital systems. For the generation of random signals, MATLAB provides two important functions amongst others.

**`rand(n,m)`** Generation of a $(n \times m)$ matrix with **uniformly distributed** random numbers $x_{i,j}$ where $0 \leq x_{i,j} \leq 1, i \in \{1, \dots n\}, j \in \{1, \dots m\}$.

**`randn(n,m)`** Generation of a $(n \times m)$ matrix with **normally (Gaussian) distributed** random numbers $x_{i,j}$ where $i \in \{1, \dots n\}, j \in \{1, \dots m\}$, the mean value $\bar{x} = 0$, the standard deviation $\sigma_x = 1$ and variance $\sigma_x^2 = 1$. Mean value, standard deviation and variance will be discussed in more detail in the following section.

To map a random signal to an arbitrary range of values, the signal can be shifted (i. e. the mean value can be changed) and/or scaled (i. e. the variance can be changed) accordingly:

```
>> x = rand(1,8)

x =

    0.0153    0.7468    0.4451    0.9318    0.4660    0.4186    0.8462    0.5252

>> x2 = 2 * x - 1

x2 =

   -0.9695    0.4936   -0.1098    0.8636   -0.0680   -0.1627    0.6924    0.0503
```

In this example, an eight element row vector `x` with uniformly distributed random number between 0 and 1 is generated. From this, a vector `x2` is computed which also contains random numbers, now between -1 and 1.

Commonly, signals must be normalized before further processing. This requires the knowledge of the largest or the smallest value of the signal. The following operators are helpful to find them:

**`abs(x)`** Produces the absolute value of all elements of the matrix **`x`**.

**`min(x)`** Returns the smallest values of all column vectors of the matrix **`x`**.

**`max(x)`** Returns the largest values of all column vectors of the matrix **`x`**.

Using these functions, the vector `x2` can for example be normalized to its largest element in absolute value in the following way:

```
>> x2 = x2 / max( abs(x2) )

x2 =

-1.0000 0.5091 -0.1133 0.8908 -0.0702 -0.1678 0.7143 0.0519
```

### 3.1.2 Mean Value, Standard Deviation and Variance

When considering random signals, mean value, standard deviation and variance play an important role. The time average $\bar{x}$ of a finite signal with $n$ elements is also referred to as the expectancy value or first-order moment.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{3.1}$$

The variance $\sigma^2$ – also called the dispersion – is the so-called second-order central moment and it is a measure for how far the values $x_n$ of a signal $\mathbf{x}$ deviate from their mean value $\bar{x}$ on average. The standard deviation $\sigma$ is then the square root of the variance. In the literature, two different definitions can be found:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x-\bar{x})^2} \tag{3.2-a}$$

or

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x-\bar{x})^2} \tag{3.2-b}$$

These two definitions differ only in the normalization term. If you want to compute the standard deviation or the variance with one of the built-in MATLAB functions `std` and `var`, you will have to pay attention to which of the definitions should be used for the calculation of the values. By default, MATLAB uses the definition from equation (3.2-a). In the following, the functions for the computation of the named quantities are described:

**`mean(x)`** Returns the mean value of the elements of the vector **x** or the mean values of the column vectors of the matrix **x** according to equation 3.1.

**`std(x)`** Returns the standard deviation of the elements of the vector **x** or the standard deviations of the column vectors of the matrix **x** according to equation 3.2-a. Via another parameter *`flag`* which can be either 0 or 1, you can directly set that the standard deviation should be calculated using equation (3.2-a) or equation (3.2-b) respectively. Calling `std(x)` is therefore equivalent to calling `std(x,0)`, where `flag` is set to 0.

**`var(x)`** Returns the variance of the elements of the vector **x** or the variances of the columns vectors of the matrix **x**. The remarks on `std` apply here in the same way.

With the help of these functions, it is for example possible to check the behavior of the functions for the generation of random numbers:

```
>> x = randn(1e6,1);
>> mean(x)

ans =

  -6.9405e-04
```

```
>> std(x)

ans =

    1.0002

>> var(x)

ans =

    1.0005
```

A column vector **x** with one million normally distributed values is generated, the mean value of which is approximately 0 and the variance is approximately 1.

### 3.1.3 Histograms

For the further characterization of signals, particularly the kind of distribution of the values of the signal is of interest. One way to identify the distribution is to – as already indicated in chapter 1 – create histograms, from which it can be deduced how often certain values occur. As the signals are usually continuous in value, it is not possible to consider discrete values but value ranges, so-called *bins*. A histogram is created by checking for each of the values of the signal what value range it corresponds to. A counter for this value range is then incremented by 1. The value ranges at both ends of the histogram are open so that all values of the signal can be allocated. In the end, the sum of all value range counters should match the number of individual values of the signal. When the counters are normalized to this number, an approximation of the *probability density function (PDF)* of the signal is acquired.

`hist(x,bins)`

The `h = hist(x,bins)` function provided by MATLAB allows the easy creation of histograms. For the signal **x** (vector), a histogram is returned to the vector **h**. Its centers (midpoints between value range edges) are passed to the vector **bins**. This makes it possible to vary the size of the value ranges. Alternatively, it is also possible to specify the desired number of value ranges by passing it as a scalar *nbins* to the function `hist(x,nbins)`. The range between the maximum and the minimum of the signal is then divided into equidistant bins.

An example should illustrate how simple histograms can be created:

```
>> x = rand(0.5e6,1) - 0.5;
>> y = randn(1e6,1);
>> bins = -5:0.2:5;
>> hx = hist(x,bins);
>> hy = hist(y,bins);
>> plot(bins,hx,bins,hy,'--')
```

In this example, a zero-mean ($\bar{x} = 0$) uniformly distributed signal **x** with half a million values and a zero-mean normally distributed signal **y** are generated. The respective histograms are recorded for value ranges with a width of 0.2 with centers between -5 and 5 and displayed graphically. The result is shown in Figure 3.1.

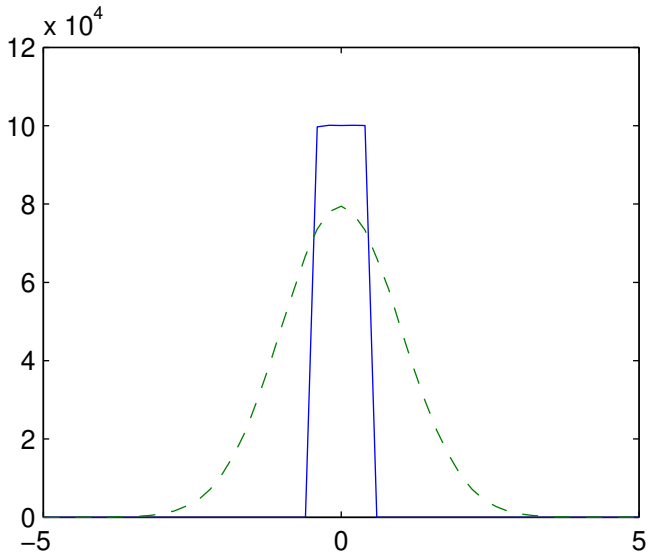



**Figure 3.1:** Histograms of the values of a uniformly and a normally distributed signal.

When only interested in the graphical output, it is sufficient to call `hist(x,bins)` without return vector **h**. The result is a bar chart of the histogram in which the centers are marked.

With the help of the approximated of the probability density function resulting from the histogram, it is also possible to get an approximation of the *cumulative distribution function (CDF)*. For a given probability density function, the corresponding CDF could be acquired by means of integration. Here, it is analogously computed by cumulatively adding up the histogram values.

`cumsum(p)`

For this purpose, MATLAB provides the function `c = cumsum(p)`. The returned vector **c** has the same length as the input vector **p** and contains the cumulative sum of the elements of the vector **p**. If **p** is a histogram normalized to the number of signal values, i. e. an approximation of the PDF, the result will be the desired approximation of the CDF. With the help of this, it is for example possible to get the probability that any signal value is smaller or equal to a given value.

The procedure is illustrated in the following example:

```
>> px = hx/sum(hx);
>> py = hy/sum(hy);
>> cx = cumsum(px);
>> cy = cumsum(py);
>> plot(bins,cx,bins,cy,'--')
```

Here, the approximated probability density functions **px** and **py** are first acquired by normalizing the histograms **hx** and **hy** to the respective number of signal values. From the result, the approximated cumulative density functions **cx** and **cy** are then calculated. The result is shown in Figure 3.2. Here, it can for example be said that for both signals, the probability that any signal value is smaller than 0 is 0.5.

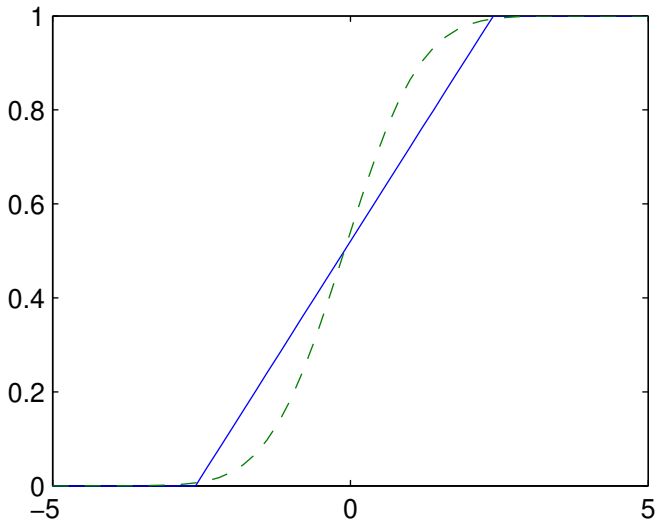



**Figure 3.2:** Approximated cumulative distribution function of a uniformly and a normally distributed signal.

### 3.1.4 Cross- and Autocorrelation

Another important role in the analysis of signals plays the similarity, more precisely the correlation, of signals. In statistics, the *cross-correlation sequence* is used as a measure for the similarity of two real signals $\mathbf{x}$ and $\mathbf{y}$ with:

$$\varphi_{xy}(\lambda) = \sum_{n=-\infty}^{\infty} x(n) \cdot y(n+\lambda) \tag{3.3}$$

The cross-correlation can be interpreted as an overlapping of the values $x_n$ and the shifted values $y(n+\lambda)$ where the overlapping values are multiplied and added up. Large values of the cross-correlation sequence indicate a large similarity of the signals $\mathbf{x}$ and $\mathbf{y}$ for the respective shift of $\lambda$ values.

Of particular importance – for example for the calculation of prediction filter coefficients or the decorrelation of CDMA (Code Division Multiple Access) signals – is the comparison of signals with themselves. It follows for $\mathbf{x} = \mathbf{y}$ that:

$$\varphi_{xy}(\lambda) = \varphi_{xx}(\lambda) = \sum_{n=-\infty}^{\infty} x(n) \cdot x(n+\lambda) \tag{3.4}$$

This is referred to as the *autocorrelation sequence* $\varphi_{xx}(\lambda)$ of the signal $\mathbf{x}$. It serves as a measure for the self-similarity of a signal. It can immediately be seen that the maximum of the autocorrelation sequence can be found at the position $\lambda = 0$, i. e. for an overlapping without shift. However, for periodic or approximately periodic signals of period $T$, further local maxima at positions $\lambda = iT$ with $i \in \mathbb{N}$ can be found.

`xcorr(x,y)`

For the calculation of cross- or autocorrelation sequences, MATLAB provides the function `xcorr(x,y)` or `xcorr(x)`. The passed vectors $\mathbf{x}$ and $\mathbf{y}$ should have the

same length $N$. Up to two vectors can be returned. When calling `[phi_xx, lags] = xcorr(x,y)`, the cross- or autocorrelation sequence with a length of $2N-1$ is returned in the vector **phi_xx**, the vector **lags** contains the shift values $\lambda$. If the vectors **x** and **y** are different in length, the shorter of the two is filled up with zeros (*zero-padding*) so that it is ensured that both have the same length. The returned cross-correlation sequence will then also exhibit corresponding zeros in its first or last elements.

The following example should demonstrate the use of the function `xcorr` for the calculation of an autocorrelation sequence:

```
>> fs = 1000;
>> t = 1/fs : 1/fs : 1;
>> f = 5;
>> x = sin(2*pi*f*t);
>> [phi_xx,lags] = xcorr(x);
>> plot(lags,phi_xx);
```

First, a sampling frequency $fs$ of 1 kHz is specified and with it a sampling time vector **t**, the length of which corresponds to a time of one second. From this, a 5 Hz sine signal **x** is generated, for which the autocorrelation sequence is returned to *phi_xx* and the lags to *lags*. The result can be found in Figure 3.3.

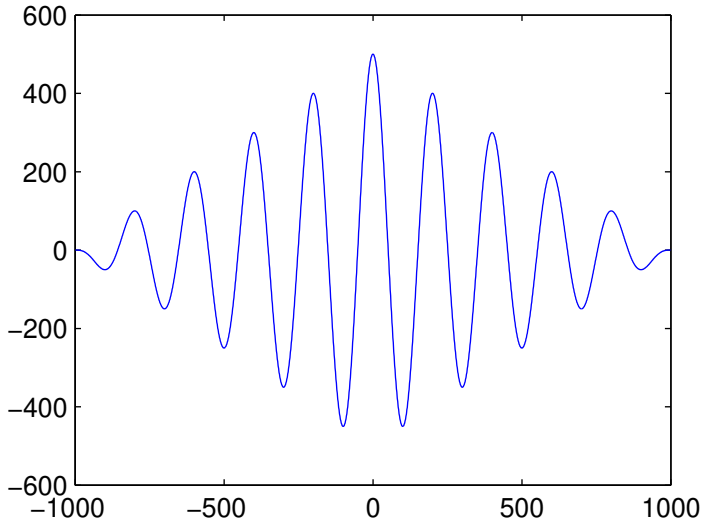



**Figure 3.3:** Autocorrelation sequence of a 5 Hz sine signal of length 1 second (sampling rate: 1 kHz).

Beside the global maximum at the position $\lambda = 0$, the first sidelobes can be found at the positions $\lambda = \pm 200$. With the help of the specified sampling rate of 1 kHz, it can be deduced that that the frequency of the original signal is $f = (200/1000)^{-1} = 5\,\text{Hz}$.

## 3.2 Discrete Fourier Transform (DFT)

In this section, the Discrete Fourier transform (DFT) and its special properties will be introduced. For this purpose, the theoretical fundamentals of the DFT are compactly summarized and the influence of the so-called windowing will be described in the following. Additionally, the functions will be presented that MATLAB provides

for the realization of the DFT and for windowing. Among these are also functions for visualizing the Fourier transforms of signals.

### 3.2.1 Definition and Properties of the DFT

The discrete Fourier transform is the essential foundation of digital spectral analysis. The Fourier integral transformation is closely related to the z-transform. However, the DFT does not necessarily need to be interpreted as an approximation of the aforementioned transformation. Rather than that, it can be interpreted as a mapping rule of its own that maps $N$ possibly complex samples $x(k)$ with $k = 0,1,\ldots N-1$ to $N$ complex spectral values $X(i)$ with $i = 0,1,\ldots N-1$. The uniquely invertible transformation is:

$$\text{DFT:} \quad X(i) = \sum_{k=0}^{N-1} x(k) \cdot e^{-j\frac{2\pi}{N}ik} \quad ; i = 0,1,\ldots N-1 \tag{3.5}$$

$$\text{IDFT:} \quad x(k) = \frac{1}{N} \sum_{i=0}^{N-1} X(i) \cdot e^{+j\frac{2\pi}{N}ik} \quad ; k = 0,1,\ldots N-1 \tag{3.6}$$

IDFT denotes the inverse DFT.

An elementary property of this transformation is that both the forward and backward transformations yield periodic sequences if the indexing is extended to $\pm\infty$. It holds that

$$\text{DFT:} \quad X(i + l \cdot N) = X(i) \quad ; l = 0, \pm 1, \pm 2, \ldots \tag{3.7}$$

$$\text{IDFT:} \quad x(k + l \cdot N) = x(k) \quad ; l = 0, \pm 1, \pm 2, \ldots \tag{3.8}$$

This property in the time and frequency domain can be explained with the discretization (sampling) in both domains. For time-continuous signals, the DFT is generally afflicted with errors due to the discretization. Only in the special case of a periodic, band-limited signal $x(k)$ with period length $N$, the Fourier integral can be exactly calculated with the DFT.

If the DFT is now used for the spectral analysis of a time signal $x(k)$, which was generated by sampling with $f_S = 1/T$, a normalization of the frequency axis by $2\pi$ according to

$$\Omega = 2\pi \cdot f / f_S \tag{3.9}$$

is common. The DFT yields N equidistant spectral values for the normalized frequencies

$$\Omega_i = \frac{2\pi}{N} i \quad ; i = 0,1,\ldots N-1 \tag{3.10}$$

in the range $0 \leq \Omega < 2\pi$ which corresponds to the non-normalized scale $0 \leq f < f_S$.

The spectral resolution is then

$$\Delta\Omega = \frac{2\pi}{N}. \tag{3.11}$$

It can be deduced from this that by increasing the block length $N$, the frequency resolution can be improved. It must however be noted that the temporal resolution is reduced in the process.

If the time signal $x(k)$ is purely real, a complex spectrum with the following symmetry properties is acquired:

$$\begin{aligned} X(0) &= \text{purely real} \\ X\left(\frac{N}{2}\right) &= \text{purely real} \\ X(i) &= X^*(N-i) \end{aligned}$$

Because of this, the spectrum can be represented by two real and $\frac{N}{2} - 1$ complex values. The number of values to be stored or processed remains unchanged by the transformation.

Further essential properties of the DFT are summarized in Table 3.1.

| Operation | Property |
|---|---|
| DFT | $X(i) = \sum_{k=0}^{N-1} x(k) \cdot e^{-j\frac{2\pi}{N}ik} \quad ; i = 0,1,\ldots,N-1$ |
| IDFT | $x(k) = \frac{1}{N} \sum_{i=0}^{N-1} X(i) \cdot e^{+j\frac{2\pi}{N}ik} \quad ; k = 0,1,\ldots,N-1$ |
| Linearity | $\mathrm{DFT}\{a \cdot x(k) + b \cdot y(k)\} = a \cdot X(i) + b \cdot Y(i)$ |
| Time shift | $\mathrm{DFT}\{\tilde{x}(k+m)\} = X(i) \cdot e^{+j\frac{2\pi}{N}im} \quad ; m \in \{0, \pm 1, \ldots\}$ |
| Modulation | $\mathrm{DFT}\{x(k) \cdot e^{+j\frac{2\pi}{N}mk}\} = \tilde{X}(i-m)$ |
| Multiplication | $\mathrm{DFT}\{x(k) \cdot y(k)\} = \frac{1}{N} \sum_{\rho=0}^{N-1} X(\rho) \cdot \tilde{Y}(i-\rho)$ |
| Convolution | $\mathrm{IDFT}\{X(i) \cdot Y(i)\} = \sum_{\rho=0}^{N-1} x(\rho) \cdot \tilde{y}(k-\rho)$ |

**Table 3.1:** Properties of the discrete Fourier transform (DFT) for sequences $x(k)$, $y(k)$ of length $N$ ($\tilde{x}, \tilde{X}, \tilde{y}, \tilde{Y}$ = periodic repetitions of $x, X, y, Y$)

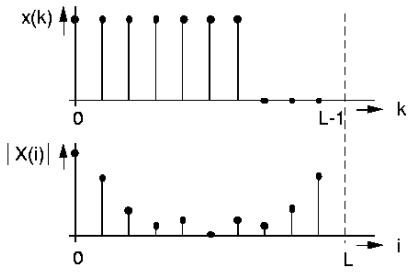
The number of spectral values of the DFT is not necessarily coupled with the length of the signal or the signal section. For a given signal length $L$, additional values in the frequency domain can be acquired by artificially extending the time signal with zeros. This method known as *zero padding* or *zoom-DFT* is shown in Figure 3.4. By means of *zero padding*, the original spectrum is interpolated but the resolution of harmonic components in the time signal is not improved.

For the calculation of the DFT or the IDFT of a signal, the so-called *fast Fourier transform (FFT)* is used in MATLAB. The FFT is an algorithm that realizes the DFT in a particularly efficient way, i. e. with minimum computational effort.

`fft(x)`

The FFT of the vector **x** can be acquired by calling `fft(x)`. The length of the returned transform matched the length of the vector **x**. By additionally passing a scalar $n$ (`fft(x,n)`), the number of returned spectral values can be specified. If the number of signal values of **x** is smaller than $n$, the signal is artificially lengthened by means of zero padding. The calculation of the IDFT with the help of the functions `ifft(x)` or `ifft(x,n)` works analogously.

`ifft(x)`



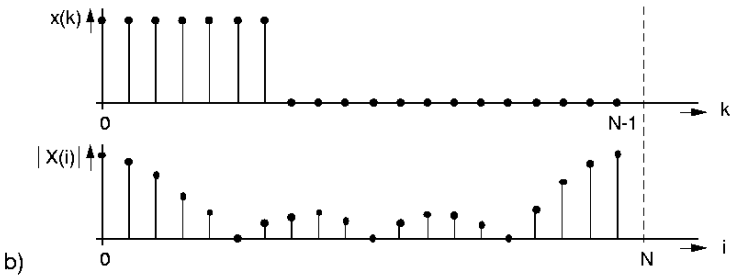




**Figure 3.4:** Interpolation of the spectrum by adding zeros a) $N = L = 10 \Rightarrow \Delta\Omega = \frac{\pi}{5}$ b) $N = L + 10 = 20 \Rightarrow \Delta\Omega = \frac{\pi}{10}$

### 3.2.2 Window functions

For the spectral analysis, the DFT is applied on signal sections of $N$ samples each. This is also referred to as short-time spectral analysis. The section comprising $N$ samples represents the time window of the DFT. The resulting (short-time) spectrum allows different but equivalent interpretations:

- Uncertainty principle:
  Due to the time section of finite extent, the spectral resolution is limited as well. It is therefore only in special cases possible that the DFT short-time spectrum and the spectrum of the entire signal coincide.

- z-transform of a sequence of finite length
  Formally, the DFT coincides with the z-transform of a signal sequence of the finite duration $N$ for $z = e^{j\frac{2\pi}{N}i}$ with $i = 0,1,\ldots N-1$. In consequence, the DFT yields the "true" spectrum of the time-limited or windowed signal. This spectrum is therefore "sampled". Between the sampling positions, this spectrum is different from zero in general.

- Periodic repetition
  The back transformation of the DFT spectrum yields a periodically repeated time signal. According to this interpretation, the DFT spectrum coincides with the "true" line spectrum of the periodically repeated signal. The line spectrum is zero between the spectral samples.

  For the periodic repetition of the time window, discontinuities (signal jumps) may occur at the window edges depending on the shape of the signal which affect the result of the DFT more or less significantly. As a result of these

discontinuities, the spectrum will exhibit frequency components that are not present in the signal to be analyzed (with respect to the long-time spectrum). This effect is called spectral leakage.

All three interpretations are valid and non-contradictory. All in all, it can be concluded that the windowing effect must be taken into account when the DFT is used for the estimation of the long-time spectrum. Only in special cases of a signal of length $L \leq N$ or a signal with period P according to $l \cdot P = N$ (l integer), the estimate is exact.

The multiplication of the analysis signal with a window function corresponds to a convolution of the discrete Fourier transform of the analysis signal with the spectrum of the window function in the frequency domain:

$$\begin{aligned} w(k) &: \text{window function} \\ W(i) &: \text{spectrum of the window function} \\ x(k) &: \text{analysis signal} \\ X(i) &: \text{spectrum of the analysis signal} \end{aligned}$$

$$x'(k) = x(k) \cdot w(k) \tag{3.12}$$

$$X'(i) = X(i) * W(i), \tag{3.13}$$

where $x'(k)$ denotes the analysis signal weighted with the window function. The effect of spectral leakage is illustrated in Figure 3.5 for a sine signal. In the first case, the signal period matches the length of the transformation ($P = N$) while in the second case, it holds that $N < P < 2N$.

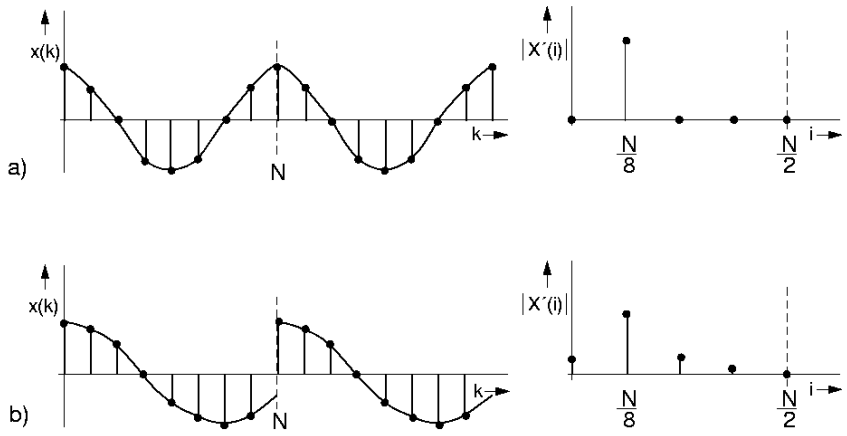



**Figure 3.5:** Explanation of spectral leakage in the time and frequency domain for a sine signal a) signal period $P = N$, b) signal period $N < P < 2N$

Despite the sinusoidal shape of the signal in both cases, only the first spectrum comprises a single spectral line only. In the second spectrum, the influence of the rectangular window is clearly visible. The effect of spectral leakage can be reduced by multiplying the signal extract with a more suitable window function.

The most important window functions are shown in Figure 3.6 and Figure 3.7. It should be noted that the spectral shapes in Figure 3.6 have been calculated with a significantly increased resolution compared to the signal length $N$ (*zero padding*, $N' = 512$).

Window functions can be described by a series of characteristic parameters. The most important parameters are the sidelobe attenuation and the 3 dB cutoff.

### Sidelobe attenuation

This parameter refers to the difference in amplitude in dB between the main lobe and the next sidelobe in the spectrum of the window function. The sidelobe attenuation is a criterion for the attenuation in the stopband.

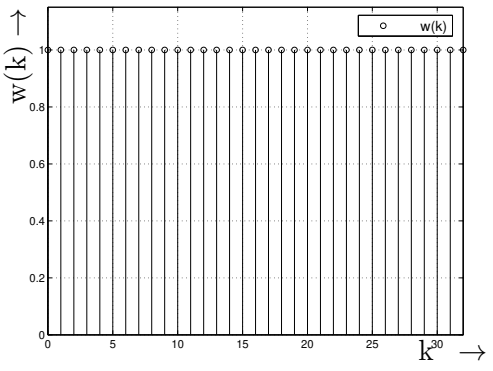
### 3 dB cutoff

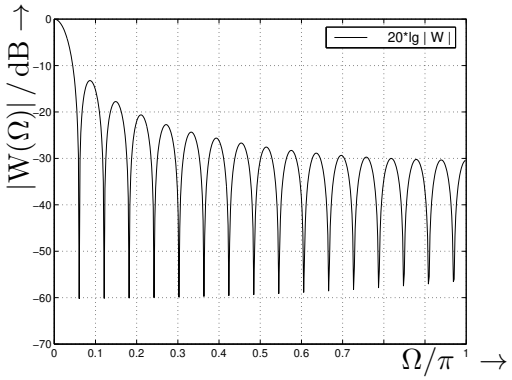
The 3 dB cutoff, also called the spectral width of a window function, describes the width of the main lobe in the spectrum of the window function. For the analysis of signals with several harmonic components, the 3 dB cutoff is a measure for the frequency selectivity, i. e. for the resolution of harmonic components in close proximity. This parameter is specified in multiples of the reciprocal window width $1/N$.

| window function | next sidelobe in dB | 3 dB cutoff $/(2\pi/N)$ with $N = 33$ |
|---|---:|---:|
| Rectangle | $-13$ | 0.89 |
| Bartlett (Triangle) | $-27$ | 1.28 |
| Hann | $-32$ | 1.30 |
| Hamming | $-43$ | 1.44 |
| Blackman | $-58$ | 1.68 |
| Kaiser ($\alpha = 3$) | $-25$ | 1.17 |

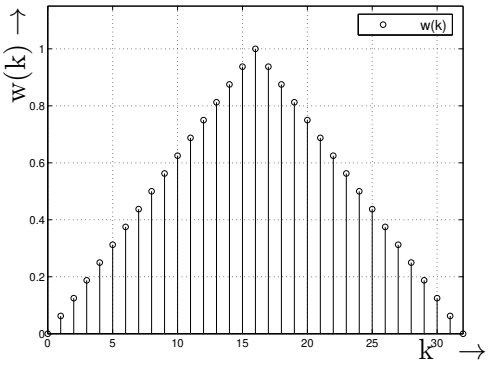
Between the frequency selectivity (3 dB cutoff) and the sidelobe attenuation, there is an uncertainty principle. Either the spectral function of the window has a narrow mainlobe or the sidelobes are strongly attenuated. As a result, it is not possible to design a window that is equally suitable for all applications of the DFT. It is therefore advisable to find the best possible compromise for the respective application when designing a window.

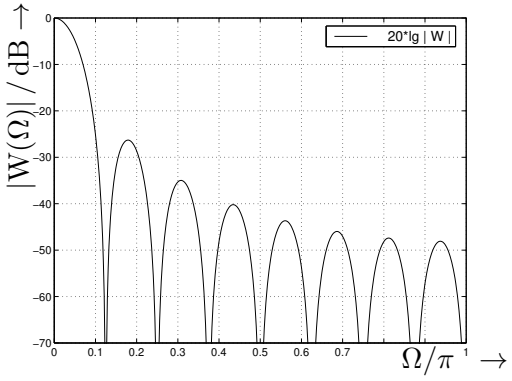
Rectangular window: $$w(k) = \begin{cases} 1 & ; k = 0,1,\ldots,N-1 \\ 0 & ; \text{else} \end{cases}$$



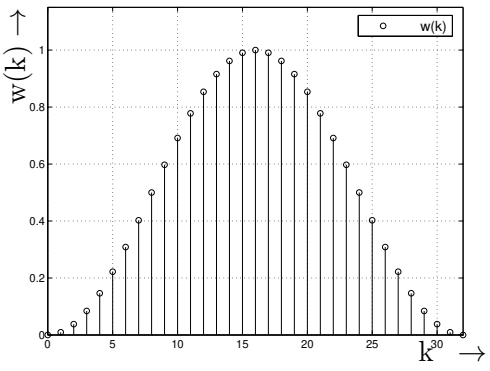




Triangular window: $$w(k) = \begin{cases} 2k/(N-1) & ; k = 0,1,\ldots,(N-1)/2 \\ w(N-1-k) & ; k = (N-1)/2,\ldots,(N-1) \end{cases}$$ , N odd




Hann window: $$w(k) = 0.5 - 0.5\cos(2\pi k/(N-1))$$



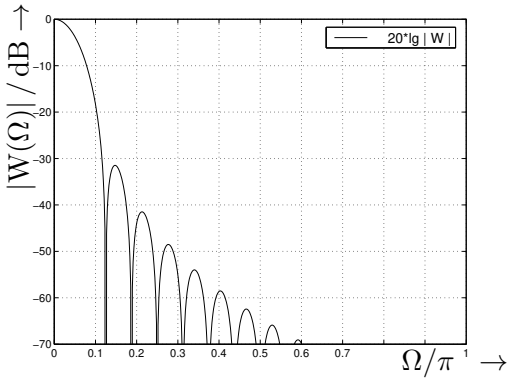
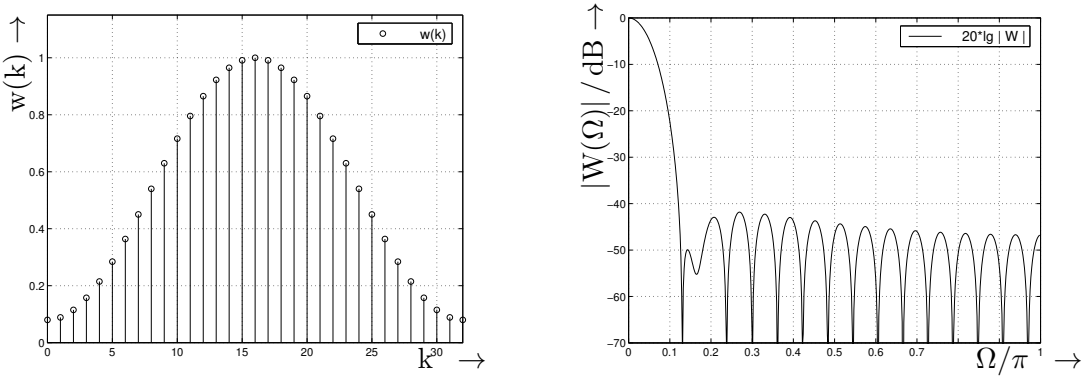
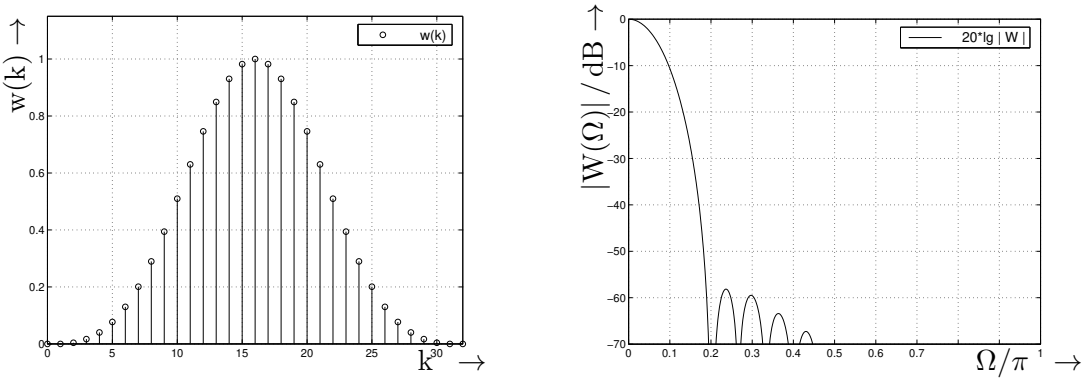




**Figure 3.6:** Comparison of several window functions

Hamming window: $w(k) = 0.54 - 0.46\cos(2\pi k/(N-1))$




Blackman window: $w(k) = 0.42 - 0.5\cos(2\pi k/N - 1)) + 0.08\cos(4\pi k/(N-1))$




Kaiser window: $w(k) = \dfrac{I_0\left(\alpha\sqrt{1-(1-\frac{k}{\beta})^2}\right)}{I_0(\alpha)}$

$I_0$: Modified Bessel function first kind of order zero, $\alpha$: form parameter, $\beta = (N-1)/2$

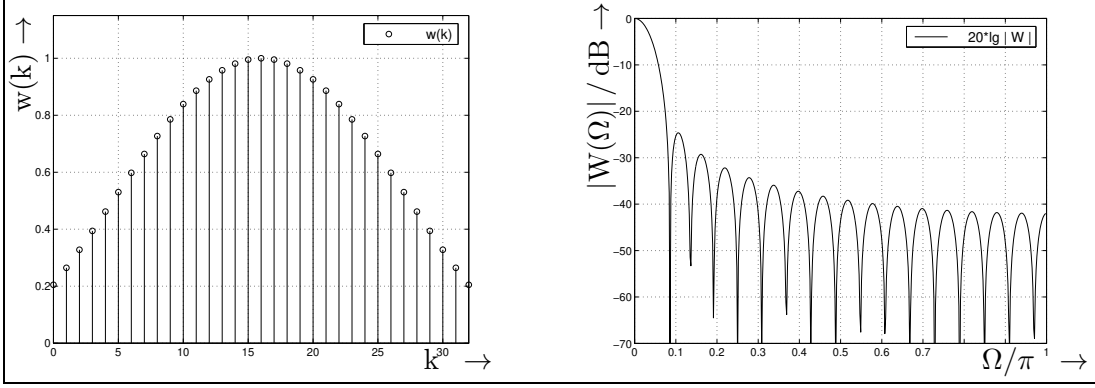



**Figure 3.7:** Comparison of several window functions (continued)

In MATLAB, all of the window functions presented here can easily be generated with special functions:

**`rectwin(n)`** creates a rectangular window of length $n$.

**`triang(n)`** creates a triangular window of length $n$.

**`hann(n)`** creates a Hann window of length $n$.

**`hamming(n)`** creates a Hamming window of length $n$.

**`blackman(n)`** creates a Blackman window of length $n$.

**`kaiser(n,alpha)`** creates a rectangular window of length $n$ $n$ with the form parameter *alpha*.

These and further window functions and their descriptions can be found in the MATLAB help under the keyword `window`.

### 3.2.3 Representation of Complex Numbers in MATLAB

Fourier transforms of real signals are generally complex. This makes their visualization more complicated than the visualization of real time signals.

MATLAB treats Matrices – including vectors and scalars – with complex values just like real matrices with respect to standard operators. The individual values of a complex matrix are represented component-wise which means that they consist of the sum of real and imaginary part, the latter of which is indicated by a small `i`. This is illustrated in the following example:

```
>> x = 1 + i

x =

   1.0000 + 1.0000i

>> y = [1 , 2 + 2j ; -2 - 0.5*j , 3j]

y =

   1.0000             2.0000 + 2.0000i
  -2.0000 - 0.5000i        0 + 3.0000i
```

`i`
`j`

Obviously, both characters `i` and `j` may be used to generate the complex number $i$. As shown in the example above, these can either directly be connected with the imaginary part (`3i`, `3j`) or by explicit multiplication (`3*i`, `3*j`).

**Careful: The characters `i` and `j` should not be used as variables if possible because they are overloaded in the process:**

```
>> i = 3;
>> x = 3*i

x =

     9
```

But:

```
>> i = 3;
>> x = 3i

x =

0 + 3.0000i

>> y = 1i * x

y =

-3
```

MATLAB offers various functions and operators for handling complex numbers. The most important are listed in the following:

**`real(c)`** produces the real part of the complex number $c$.

**`imag(c)`** produces the imaginary part of the complex number $c$.

**`abs(c)`** produces the absolute value of the complex number $c$.

**`angle(c)`** produces the phase in radian of the complex number $c$ in the range from $-\pi$ to $\pi$.

**`conj(c)`** produces the complex conjugate $c^*$ of the complex number $c$.

The following example should illustrate how the aforementioned functions operate:

```
>> x = [ones(1,10) zeros(1,990)];
>> X = fft(x);
>> figure; subplot(1,2,1); plot(real(X(1:500))); title('Realteil');
>> subplot(1,2,2); plot(abs(X(1:500))); title('Betrag');
```

Here, a rectangular pulse is first generated and transformed. Real part and magnitude of the Fourier transform of this pulse are then displayed. Because the spectrum is symmetric, merely the first 500 values are considered. The result is shown in Figure 3.8.

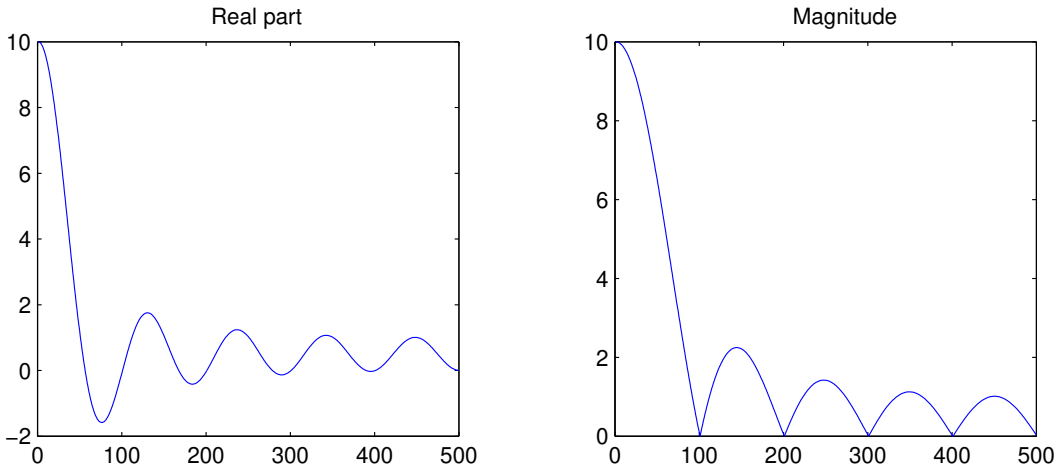



**Figure 3.8:** Real part and magnitude of the Fourier transform of a rectangular pulse.

A commonly used way to visualize the frequency domain transform of a signal is the so-called *spectrogram*. For such spectrogram, the DFT of a time signal is calculated

by applying a "sliding" window, this is consequently also referred to as short-time DFT. As only a (depending on the window size and the DFT length) temporally short section of the signal is analyzed like this, this is equivalent to the presented short-time analysis.

`spectrogram(x)`

In the *Signal Processing Toolbox*, MATLAB provides the function `spectrogram(x)` for this purpose. In addition to the signal to be analyzed, all important parameters can be passed to this function:

`X = spectrogram(x,window,noverlap,nfft,fs)`.

In this,

**x** represents the signal to be analyzed,

**window** represents the window to be used,

*noverlap* the overlapping range of the analysis window.

*nfft* represents the FFT length,

*fs* represents the sampling frequency of the analyzed signal.

The generally complex-valued spectrogram is passed to the vector **X**. If some of the named parameters are not passed, MATLAB uses the preset parameters, the values of which are listed in the MATLAB help. If the function

```
spectrogram(x,window,noverlap,nfft,fs)
```

is called without a return vector, MATLAB realizes a two-dimensional visualization of the spectrogram where the abscissa represents the frequency and the ordinate represents the time. The absolute value of the respective frequency components is indicated by different colors:

```
>> fs = 1000;
>> t = 1/fs : 1/fs : 10;
>> f1 = 350;
>> f2 = 400;
>> x = 0.2*randn(1,length(t)) + sin(2*pi*f1*t) + cos(2*pi*f2*t);
>> spectrogram(x,blackman(200),100,200,fs,'yaxis');
```

Firstly, a sampling rate `fs` of 1 kHz is again specified. With the help of the time vector **t** which represents a time duration of 10 s, a signal **x** consisting of two sinusoidal components and a noise component is generated. This signal is then windowed with a Blackman window with a length of 200 samples and transformed to the frequency domain with an FFT of length 200. The used overlapping is 100 samples long.

The spectrogram is depicted in Figure 3.9. The strong signal components at the frequencies `f1` and `f2` are clearly visible here. Furthermore, the white characteristic (equally strong signal contributions at all frequencies) of the Gaussian noise can be seen.

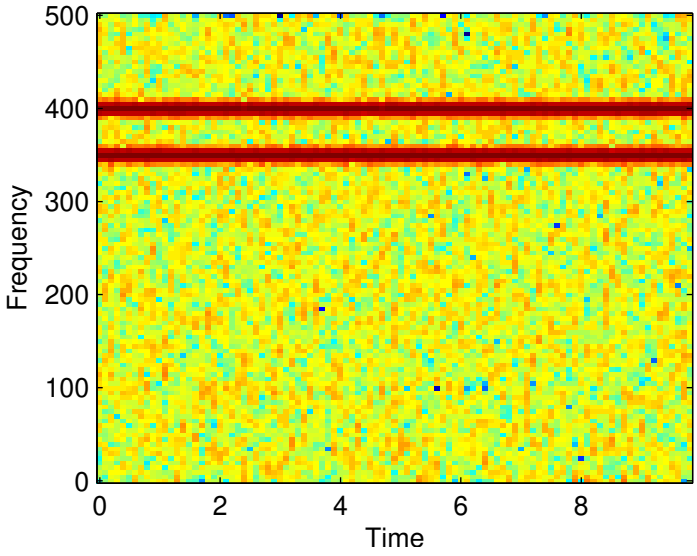



**Figure 3.9:** Spectrogram of an artificially generated noisy sinusoidal signal with two frequency components.

## 3.3 Exercises

### Exercise 3.1

Generate a uniformly distributed noise sequence **x1** and two Gaussian distributed noise sequences **x2** and **x3** of length $1 \cdot 10^6$ with the help of the functions `rand` and `randn`! Manipulate the sequences so that

all sequences have a mean value of 2,

the uniformly distributed noise has a maximum amplitude 2 around the mean value 2

and the two Gaussian distributed noise sequences have the variances 0.5 and 1.5 respectively!

Confirm these properties by applying `mean` and `var`!

Create histograms **h1**, **h2** and **h3** of the three distributions and plot them in a single graph! Use a resolution (width) of 0.1 for the *bins*. How large must the considered value range be chosen to get a proper (no peaks at the edges) distribution?

Approximate the probability density functions of the distributions **p1**, **p2** and **p3** from the histograms and calculate the resulting cumulative distributions functions **c1**, **c2** and **c3** from them!

Display **c1**, **c2** and **c3** in a single graph!

How large are the three probabilities $p_i(x \geq 1), i = 1,2,3$ that an arbitrary value $x$ of the noise sequences is greater or equal to 1? (Read from graphs!)

### Exercise 3.2

Load the file *voice1.mat* into the Workspace! The variable *voice*1 contains a section of a speech sample that has been digitized with a sampling rate of 8 kHz. How long (in ms) is the section?

Calculate the autocorrelation sequence of the signal and display it graphically!

Determine the pitch period (or the fundamental frequency) of the speaker from the first sidelobe!

Now load the file *sequences.mat* into the Workspace! You will get the random sequences **x**, **y1**, **y2** and **y3**. Determine which of the three sequences **y1**, **y2** and **y3** are a shifted or scaled version of **x** and which sequence is uncorrelated!

How large is the shift and the scaling factor?

### Exercise 3.3

Generate the three signals $x_i(k)$ $(i = 0, 1, 2)$ of length $N = 64$ with

$$x_i(k) = \gamma_0(k - k_i) = \begin{cases} 1 & k = k_i \\ 0 & \text{else} \end{cases} \qquad (\text{with} \quad \gamma_0 = \text{unit impulse})$$

for $k_i = i$ with $i = 0, 1, 2$! Compute the Fourier transforms of the signals and get both the magnitude $|X(\Omega)|$ and phase response $arg\{X(\Omega)\}$ as well as the real and imaginary parts displayed graphically!

Repeat these steps with the following input vectors where N denotes the length of a vector to be transformed (select a single, arbitrary $N$):

$$x(k) = \cos(\Omega_0 k) \quad \text{with} \quad \Omega_0 = \frac{2\pi n}{N}\ ,\ n = 1,2,...$$

$$\text{i. e. } \Omega_0 \text{ is an integer multiple of } \frac{2\pi}{N}$$

$$x(k) = \cos(\Omega_1 k) \quad \text{with} \quad \Omega_1 \neq \frac{2\pi n}{N}\ ,\ n = 1,2,...$$

$$\text{i. e. } \Omega_1 \text{ is not an integer multiple of } \frac{2\pi}{N}$$

Explain the amplitude and phase spectra as well as the real and imaginary parts!

## Exercise 3.4

Load the file *distorted.mat* into the Workspace! You will get a noise signal **x** that has been sampled with 8 kHz and which comprises three sinusoidal components of different strengths and frequencies.

Identify the frequency components using an adequate spectral representation of the signal.

## Exercise 3.5

Load the file *voice2.mat* into the Workspace. The variable *voice*2 contains a speech signal sampled with 8 kHz. Create proper spectrograms in which you vary the FFT length, the window length, the window type and the overlapping range! What do you notice?

Listen to the speech sample with the function `sound` and try to follow where the consonants, vowels and speech breaks are. Which characteristics can you find?

# Chapter 4

# Filter Design

## 4.1 Introduction

Digital filters in non-recursive and recursive form are the most commonly used tool for the processing of digital signals. The goal of this experiment is to understand how they operate and to get to know different filter design techniques in MATLAB. For this purpose, filters with temporally constant coefficients are considered first.

A further elaboration of the matter can be found in the literature. [1]

## 4.2 Description of Linear Discrete-Time Filters

This experiment deals with digital filters that are to be considered as linear discrete-time systems. The relation between the input signal $x(k)$, the output signal $y(k)$ and the impulse response $h(k)$ of the filter (see Figure 4.1) is described by the convolution operation denoted with the symbol "$*$". It holds that

$$y(k) = \sum_{i=-\infty}^{\infty} h(i) \cdot x(k-i) \tag{4.1}$$

$$= h(k) * x(k). \tag{4.2}$$

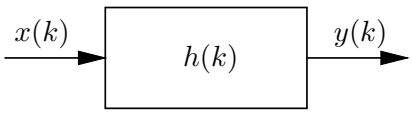



**Figure 4.1:** Block diagram of a digital filter with the impulse response $h(k)$.

An alternative description is given by the difference equation

$$y(k) = \sum_{i=0}^{M} b_i \cdot x(k-i) - \sum_{i=1}^{N} c_i \cdot y(k-i) \tag{4.3}$$

with the constant coefficients $b_i$ and $c_i$. The greater of the two numbers $N$ and $M$ is called the order $G$ of the difference equation or the filter. The first sum describes the *non-recursive* part, the second equation the *recursive* part of the filter that is described by the difference equation (4.3).

[1] e. g. Jackson: *Digital Filters and Signal Processing*, Springer, 1996

The impulse response $h(k)$ can be determined from equation (4.3) when the input signal $x(k)$ is set to the unit impulse $\delta(k)$.

$$x(k) = \delta(k) = \begin{cases} 1 & ; \quad k = 0 \\ 0 & ; \quad \text{else} \end{cases} \tag{4.4}$$

If the length of $h(k)$ is infinite, which is generally the case for real-world systems, the filter is referred to as an IIR filter (IIR = Infinite Impulse Response). An impulse response of finite length results from the special case of a purely non-recursive system, i. e. for $c_i = 0$ with $i = 1, \ldots N$. Such a system is called an FIR filter (FIR = Finite Impulse Response). The impulse response has a length of $L = M + 1$ and the difference equation (4.3) is simplified to

$$y(k) = \sum_{i=0}^{M} h(i) \cdot x(k-i) = \sum_{i=0}^{M} b_i \cdot x(k-i). \tag{4.5}$$

Beside the convolution product and the difference equation, the representation of the filter by a transfer function $H(z)$, i. e. the $z$-transform of the difference equation, is important. Consequently, the transfer function of an IIR filter is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum\limits_{i=0}^{M} b_i \cdot z^{-i}}{\sum\limits_{i=0}^{N} c_i \cdot z^{-i}} \quad \text{with } c_0 = 1. \tag{4.6}$$

It follows for the transfer function of an FIR filter

$$H(z) = \sum_{i=0}^{M} b_i \cdot z^{-i}. \tag{4.7}$$

The frequency response is obtained for $z = e^{j\Omega}$ with $\Omega = 2\pi f / f_S$ and the sampling frequency $f_S$. In this expression, $H(e^{j\Omega})$ is a function periodic with $2\pi$. It is commonly separated by magnitude $|H(e^{j\Omega})|$ and phase $\varphi(\Omega)$:

$$H(e^{j\Omega}) = |H(e^{j\Omega})| \cdot e^{j\varphi(\Omega)}. \tag{4.8}$$

The derivative of the phase response is called group delay $\tau_g(\Omega)$:

$$\tau_g(\Omega) = \frac{d\varphi(\Omega)}{d\Omega}. \tag{4.9}$$

For certain considerations, it is advantageous to use the factorized form of the transfer function according to equation (4.6), i. e. to characterize the transfer function by its poles $z_{\infty_i}$ and zeros $z_{0_i}$:

$$H(z) = \frac{\prod\limits_{i=1}^{M} (z - z_{0_i})}{\prod\limits_{i=1}^{N} (z - z_{\infty_i})}. \tag{4.10}$$

From the positions of the poles and zeros, it is possible to draw conclusions about the properties of the filter. An example is shown in Figure 4.2 for a second-order system with

$$z_{0_{1,2}} = e^{\pm j2\pi/3}$$
$$z_{\infty_{1,2}} = 0.75 \cdot e^{\pm j\pi/3}.$$

Poles and zeros of the transfer function can clearly be seen in the frequency response.

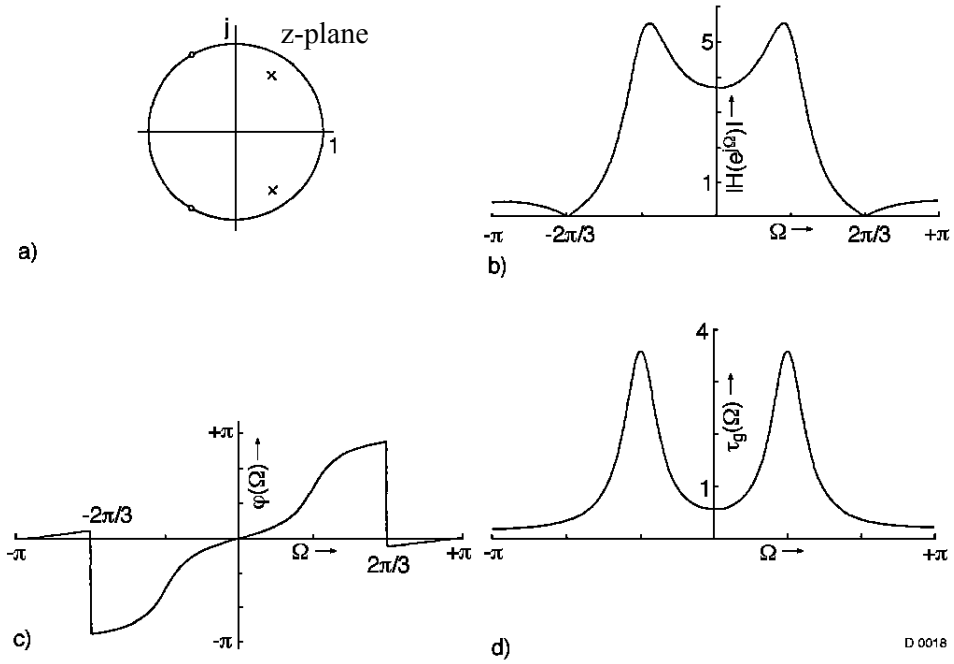



**Figure 4.2:** Example of a second order system (from: Schüssler 1988): a) pole-zero plot, b) magnitude frequency response, c) phase response and d) group delay

## 4.3 FIR Filters

### 4.3.1 FIR Filters with Linear Phase

Filters with linear phase that in consequence show a constant group delay are of particular interest. These filters affect only the magnitude spectrum of a signal. The phase spectrum remains unchanged except for a linear term (constant delay).

As a result of the constant group delay requirement, restrictions with respect to the possible zero locations follow. This in turn leads to a coefficient symmetry of the form

$$h(k) = \pm h(M-k) \quad ; \quad k = 0,1,\ldots M. \tag{4.11}$$

The relations are explained in more detail in the additional literature.

For the further discussion, it is distinguished between four filter types depending on whether $L$ is even or odd and whether the positive or the negative sign is used

in equation (4.11). In all cases, the frequency response can be described in the following way:

$$H(e^{j\Omega}) = e^{-j\frac{M}{2}\Omega} \cdot H_0(e^{j\Omega}), \tag{4.12}$$

where $H_0$ is either purely real or purely imaginary.

**Type 1:** Even coefficient symmetry, odd filter length $L = M + 1$

$$\begin{aligned} h(k) &= h(M-k) \quad ; k = 0,1,\dots M \\ M &= 2l \\ H_{0_1}(e^{j\Omega}) &= h(l) + 2\sum_{k=0}^{l-1} h(k)\cos((l-k)\Omega) \end{aligned} \tag{4.13}$$

**Type 2:** Even coefficient symmetry, even filter length $L = M + 1$

$$\begin{aligned} h(k) &= h(M-k) \quad ; k = 0,1,\dots M \\ M &= 2l-1 \\ H_{0_2}(e^{j\Omega}) &= 2\sum_{k=0}^{l-1} h(k)\cos((M-2k)\Omega/2) \end{aligned} \tag{4.14}$$

**Type 3:** Odd coefficient symmetry, odd filter length $L = M + 1$

$$\begin{aligned} h(k) &= -h(M-k) \quad , \quad h(l) = 0 \quad ; k = 0,1,\dots M \\ M &= 2l \\ H_{0_3}(e^{j\Omega}) &= 2j\sum_{k=0}^{l-1} h(k)\sin((l-k)\Omega) \end{aligned} \tag{4.15}$$

**Type 4:** Odd coefficient symmetry, even filter length $L = M + 1$

$$\begin{aligned} h(k) &= -h(M-k) \quad ; k = 0,1,\dots M \\ M &= 2l-1 \\ H_{0_4}(e^{j\Omega}) &= 2j\sum_{k=0}^{l-1} h(k)\sin((M-2k)\Omega/2) \end{aligned} \tag{4.16}$$

For each of the different filter types, examples are depicted in Figure 4.3. They generally differ in their behavior at $\Omega = 0$ and $\Omega = \pi$ as well as the number of zeros at $z = \pm 1$.

It follows for example that a lowpass filter cannot be realized with a filter of type 3 or 4 as the frequency response is always zero at $\Omega = 0$. These boundary conditions are to be taken into account when designing a filter.

### 4.3.2 FIR Filter Design

For the design of FIR filters, two standard techniques will be outlined for the example of the selective lowpass.

Given is for example the tolerance scheme according to Figure 4.4. It defines the edge frequencies $\Omega_P$ and $\Omega_S$ as well as the maximum approximation error in the passband $\delta_P$ and in the stopband $\delta_S$.

A good lowpass filter is characterized by small approximation errors $\delta_P$ and $\delta_S$, a small transition region $\Delta\Omega = \Omega_S - \Omega_P$ and a filter length which is as short as possible.

The filter design process usually consists of multiple steps:

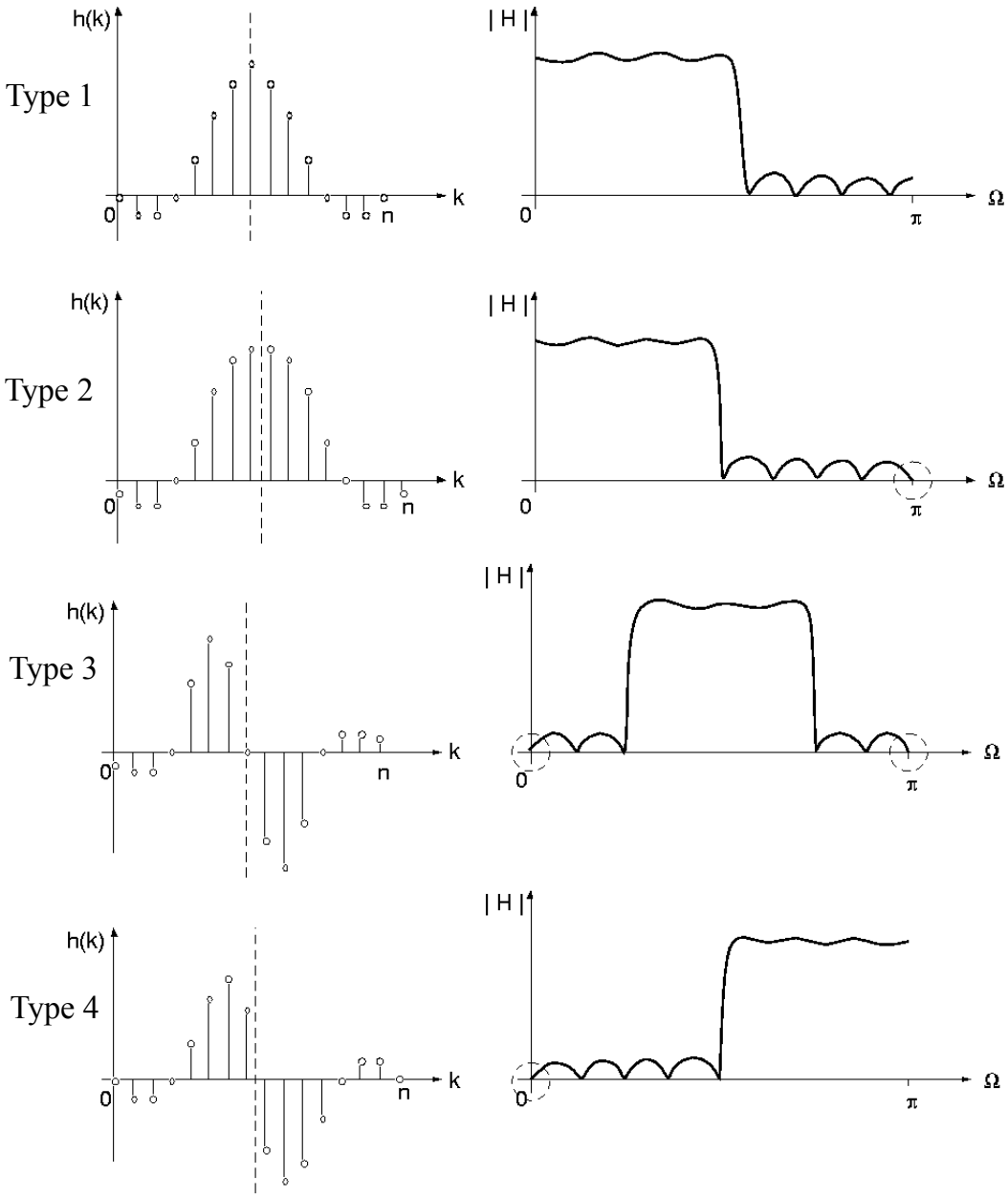



**Figure 4.3:** Types of linear-phase filters

1. Estimation of the filter length $L$
2. Optimization of the filter coefficients with respect to some error criterion
3. Comparison of the calculated filter with the tolerance scheme. If the tolerance scheme is violated or not effectively used, the process must be repeated once more from the first step.

### Modified Fourier Approximation

The impulse response of the non-causal ideal discrete-time lowpass can directly be obtained by sampling the corresponding ideal continuous-time filter. With the usual

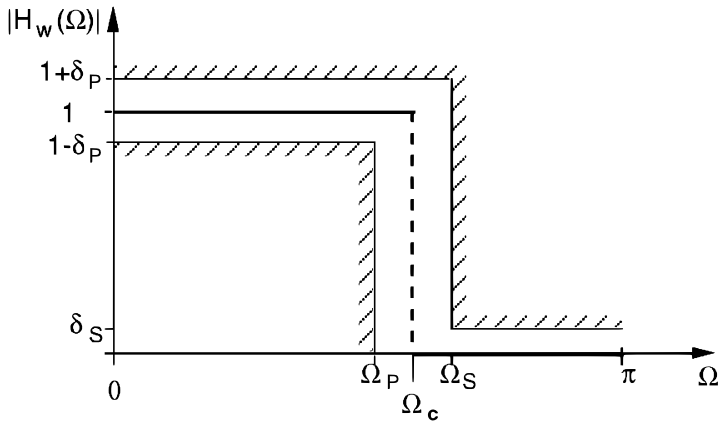



**Figure 4.4:** Tolerance scheme and desired frequency response $H_W(\Omega)$ for the LP design

frequency normalization, it holds that

$$h_W(k) = \frac{\Omega_c}{\pi} \frac{\sin(\Omega_c k)}{\Omega_c k} \tag{4.17}$$

with the cutoff frequency $\Omega_c$ of the filter. The frequency response of the discrete-time filter is given by

$$H_W(e^{j\Omega}) = \sum_{k=-\infty}^{\infty} h_W(k) \cdot e^{-jk\Omega}. \tag{4.18}$$

This expression can formally also be interpreted as the Fourier series expansion of the $\Omega = 2\pi$-periodic function $H_W(e^{j\Omega})$. It is well-known that such a series yields an optimum approximation with respect to the least mean squared error when only a finite number of the first terms is taken. Therefore, a solution of the approximation problem with respect to the aforementioned error criterion is now available.

The frequency response then has the form

$$H_0(e^{j\Omega}) = \sum_{k=-l}^{l} h_W(k) \cdot e^{-jk\Omega} \tag{4.19}$$

for a finite series. The still non-causal impulse response of finite length is

$$h_0(k) = \begin{cases} h_W(k) & ; \quad -l \leq k \leq +l \\ 0 & ; \quad \text{else.} \end{cases} \tag{4.20}$$

The filter design hence merely incorporates setting the desired impulse response to zero outside of an interval of finite length. This process can be described by the multiplication with a rectangular window $w_R(k)$:

$$h_0(k) = h_W(k) \cdot w_R(k). \tag{4.21}$$

This operation corresponds to a convolution in the frequency domain according to

$$\begin{aligned} H_0(e^{j\Omega}) &= H_W(e^{j\Omega}) * W_R(e^{j\Omega}) \qquad (4.22) \\ &= \frac{1}{2\pi} \int_0^{2\pi} H_W(e^{j\Theta}) \cdot W_R(e^{j(\Omega-\Theta)}) d\Theta \end{aligned}$$

$$\text{with} \quad W_R(e^{j\Omega}) = \frac{\sin(\frac{\Omega}{2}(2l+1))}{\sin(\frac{\Omega}{2})}. \tag{4.23}$$

It becomes apparent from the convolution process that the width of the transition region of the frequency response $H_0$ is determined by the width of the main lobe of the window function. An improvement to the slope in the transition region can therefore only be achieved with longer windows (or equivalently, longer filters).

However, due to the overshoot (9% at jump discontinuity) known as the "Gibbs phenomenon" and due to the related low stopband attenuation (21 dB, corresponds to a factor of about 100), this approximation is unsatisfactory.

Better attenuation properties and a lower ripple in the passband are achieved by modification of the window function. The following windows are commonly used (see experiment 3):

- Bartlett window (triangle function)
- Hann window (cosine function)
- Hamming window (raised cosine)
- Blackman window (two superimposed cosine functions)
- Kaiser window (modified Bessel functions).

However, the improved attenuation in the stopband and the reduction of the ripple in the passband of these windows is only achieved at the cost of a broadening of the transition region which can principally only be counteracted with a longer window.

The filter design with the window functions named above is also referred to as the modified Fourier approximation because of the similarity to the use of rectangular windows.

The Kaiser window (see p. 80f.) is of particular interest as it allows the filter design with a freely adjustable stopband attenuation.

It is defined as follows:

$$w(k) = I_0\left(\alpha\sqrt{1-\left(1-\frac{2k}{G}\right)^2}\right)/I_0(\alpha) \quad ; \quad 0 \le k \le G \tag{4.24}$$

where $I_0$ is the modified Bessel function first kind of order zero. The form parameter $\alpha$ allows to trade the slope in the transition region against stopband attenuation to a very large extent.

For the filter design, the following estimation formulas are used to determine the window parameters $G$ and $\alpha$ for the desired transition width $\Delta\Omega$ and the requested stopband attenuation $a = -20\log\delta_s$:

$$G \ge \frac{a/\mathrm{dB} - 7.95}{2.285 \cdot \Delta\Omega} \tag{4.25}$$

$$\alpha = \begin{cases} 0 & ; \quad a < 21 \\ C_1 \cdot (a/\mathrm{dB} - 21)^{0.4} + C_2 \cdot (a/\mathrm{dB} - 21) & ; \quad 21 \le a \le 50 \\ C_3 \cdot (a/\mathrm{dB} - 8.7) & ; \quad a \ge 50 \end{cases} \tag{4.26}$$

with the following constants

$$C_1 = 0.5842 \tag{4.27}$$

$$C_2 = 0.07886 \tag{4.28}$$

$$C_3 = 0.1102. \tag{4.29}$$

### Chebyshev Approximation

A better exploitation of the tolerance scheme can be achieved by means of a Chebyshev approximation in the passband and in the stopband.

The tolerance scheme is, as illustrated in Figure 4.5, approximated by the minimization of the maximum absolute error. Unlike the Fourier approximation, the error does not drop towards higher frequencies but oscillates with constant amplitude ("equi-ripple" approximation).

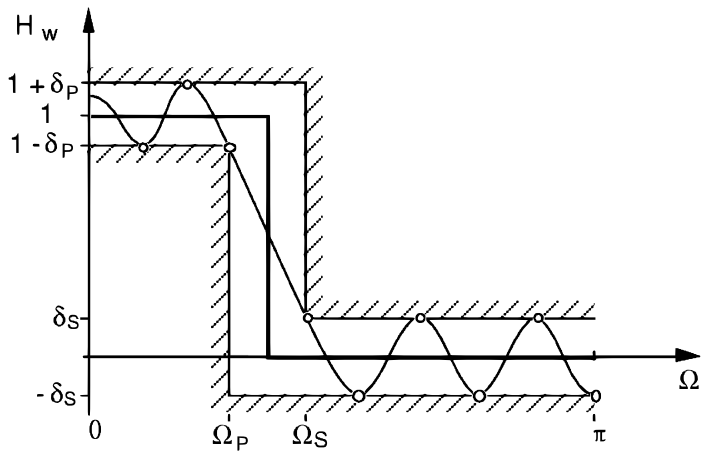



**Figure 4.5:** Concerning the Chebyshev approximation

The problem of the Chebyshev approximation can only be solved iteratively. An efficient algorithm for the numerical computation of the coefficients has been developed by Parks and McClellan. This algorithm is based on the Chebyshev Alternation Theorem and the Remez exchange algorithm. The approximation errors can be specified freely for the Parks-McClellan algorithm.

A design example is shown in Figure 4.6. It shows the lowpass frequency response when using the Chebyshev approximation and the modified Fourier approximation with a Hamming window in direct comparison. The stopband attenuation of the filter designed with a Hamming window is $a = -53$ dB while it is about $a = -63$ dB for the Chebyshev design.

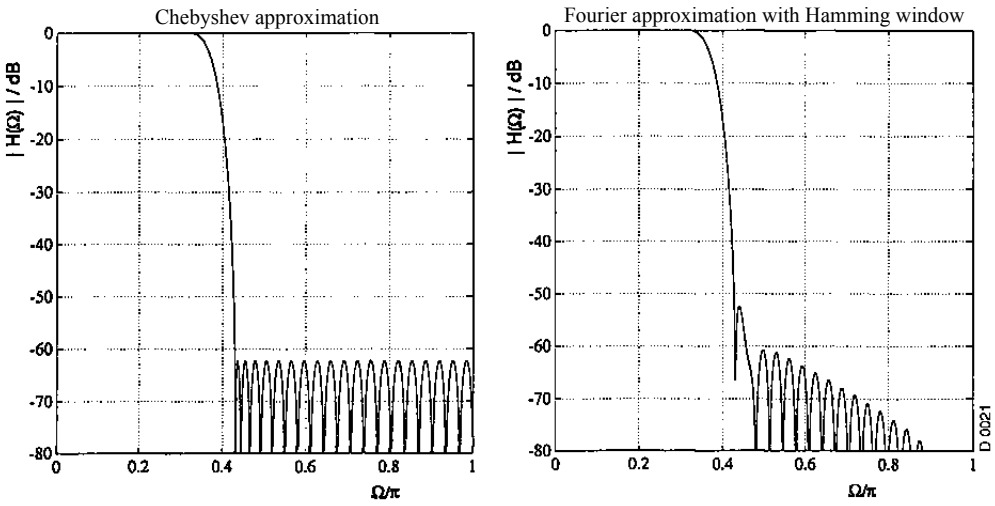



**Figure 4.6:** Comparison between Chebyshev approximation and modified Fourier approximation with Hamming window ($G = 64, \Omega_P = 0.32\pi, \Omega_c = 0.375\pi, \Omega_S = 0.43\pi$)

To conclude, it should be added that the filter order can also be estimated in case of the Chebyshev approximation. A rough estimate of filter order is given as

$$G \geq \frac{-10\log(\delta_P \delta_S) - 13}{2.324 \cdot \Delta\Omega}. \tag{4.30}$$

## 4.4 IIR Filters

### 4.4.1 Stability

Due to their recursive construction, the stability is not always ensured for IIR filters. A recursive filter is stable if all poles of the transfer function, i. e. the zeros of the denominator polynomial, lie within the unit circle. Unlike non-recursive filters, recursive filters generally do not have a linear phase response as a result. It holds for the impulse response of a stable, causal and time-invariant filter that

$$\sum_{k=0}^{\infty} |h(k)| < \infty. \tag{4.31}$$

The systems responds to a bounded input sequence with a bounded output sequence. From this requirement, the relation (4.31) can be deduced as a both necessary and sufficient condition for stability.

### 4.4.2 Filter Structures

A variety of different structures, which can be derived from different but equivalent representations of the transfer function, the difference equation or the state description, exist for the realization of digital filters.

These structures are completely identical with respect to the functional relation between $y(k)$ and $x(k)$ for non-quantized parameters and state values. Differences in the filter behavior result only from the required quantization or the finite calculation accuracy which is inevitable for the practical execution.

In the following, only the most important basic structures are outlined. The so-called canonical forms, which lead to a realization of the filter with a minimum number of delay elements, are of particular importance.

### First canonical structure (direct-form 1):

The structure illustrated in Figure 4.7 follows directly from the representation

$$Y(z) = b_0 X(z) + \sum_{i=1}^{N} (b_i X(z) - c_i Y(z)) \cdot z^{-i} \tag{4.32}$$

for $N = M$.

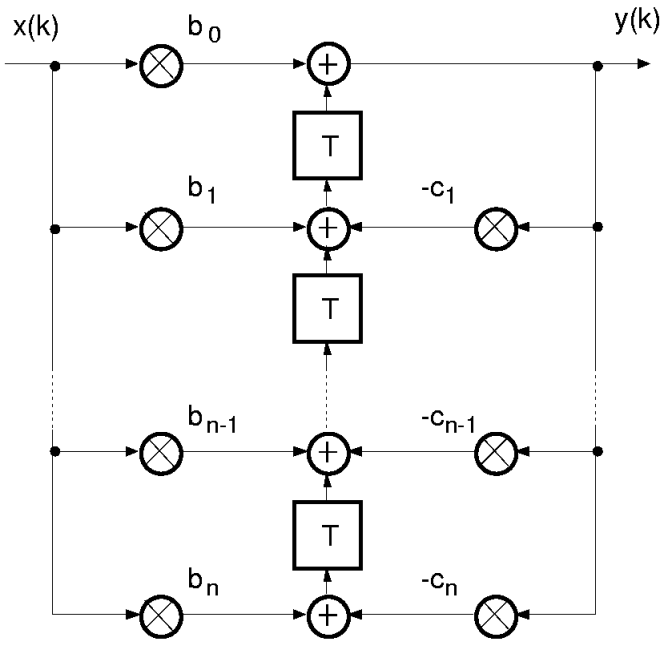



**Figure 4.7:** First canonical structure (direct-form 1)

The first canonical structure is particularly well-suited for the fixed-point realization of a filter. In the literature, the first canonical structure is also referred to as the transpose of the second canonical structure (DF2 transposed).

### Second canonical structure (direct-form 2):

For the derivation of this structure, the transfer function according to equation (4.6) is described as a cascade of a purely recursive and a non-recursive filter.

$$H(z) = \frac{1}{1 + \sum\limits_{i=1}^{N} c_i z^{-i}} \cdot \sum_{i=0}^{N} b_i z^{-i}. \tag{4.33}$$

In the signal flow diagram according to Figure 4.8, two parallel chains of delay elements which can be combined to a single chain, are obtained.

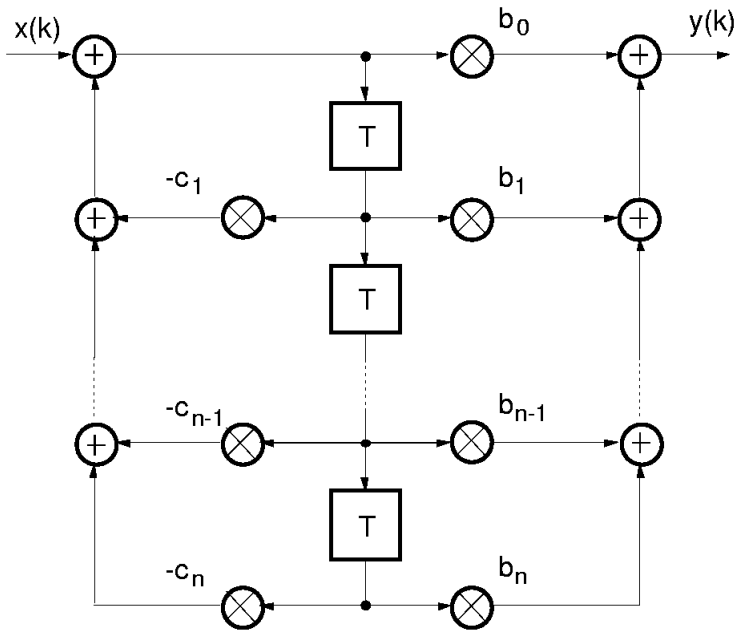



**Figure 4.8:** Signal flow diagram of the second canonical form (direct-form 2)

### Third canonical structure (cascade form):

From the pole-zero representation of the transfer function

$$H(z) = b_0 \frac{\prod_{i=1}^{M} (z - z_{0_i})}{\prod_{i=1}^{N} (z - z_{\infty_i})}, \tag{4.34}$$

a cascade of $p$ subsystems of order $G = 1$ and $G = 2$ (complex conjugates) is obtained by combining complex conjugate poles and zeros:

$$H(z) = \prod_{i=1}^{p} H_i(z). \tag{4.35}$$

The subsystems can be realized in either the first or the second canonical structure (Figure 4.9).

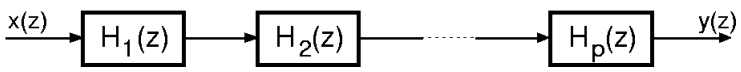



**Figure 4.9:** Third canonical structure (cascade form)

### Fourth canonical structure (parallel form):

This structure (Figure 4.10) is obtained from the partial fraction expansion of the transfer function

$$H(z) = b_0 + \sum_{i=1}^{G} \frac{B_i}{(z - z_{\infty_i})}. \tag{4.36}$$

For complex conjugate poles, second-order subsystems can be formed that are to be realized in the first or second canonical form. Assuming simple poles, the number of subsystems is the same as for the third canonical structure.

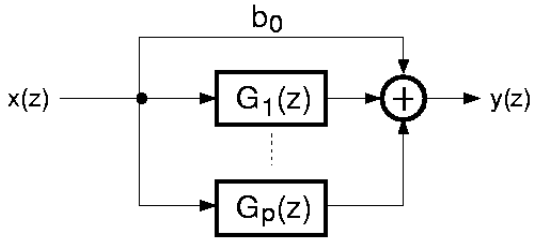



**Figure 4.10:** Fourth canonical structure (parallel form)

Beside these canonical basic structures, numerous further structures exist that are beyond the scope of this experiment. Among these are most notably the wave digital filters, the structures of which are not derived from the transfer functions but systematically with the help of the wave parameter theory from the network structures of analog reactance filters. This approach preserves certain advantageous properties of these network structures such as the stability or the insensitivity towards inaccuracies (quantization) of the parameters. A special form of these structures that is important for speech coding is the so-called lattice filter. This filter models the wave propagation in the acoustic tube model of speech production.

## 4.4.3 IIR Filter Design

Extensive literature exists for the design of discrete-time IIR filters. In this experiment, only the most important technique, the impulse invariance method, will be outlined.

The requirements for the discrete-time filter to be designed are assumed to be given in the form of a tolerance scheme for the frequency response $|H(\Omega)|$ like the one shown in Figure 4.11a.

For the design of discrete-time filters, it is exploited that numerous methods are available for the design of continuous-time filters. Therefore, the tolerance scheme is first transformed to a corresponding scheme for the frequency response $|H_C(f)|$ of a continuous-time filter (see Figure 4.11b) which can be achieved by a simple rescaling of the frequency axis according to

$$f = f_s \frac{\Omega}{2\pi}, \tag{4.37}$$

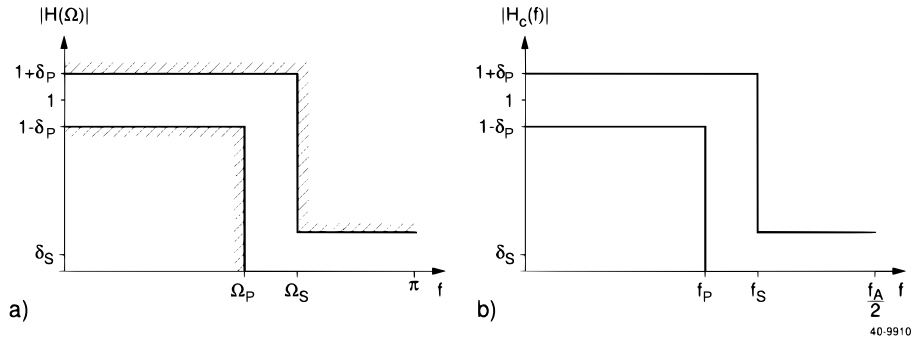



**Figure 4.11:** Tolerance scheme: a) discrete-time IIR filter, b) corresponding continuous-time filter (sampling rate is denoted here with $f_A$)

where $f_s$ with lower case '$s$' is the sampling rate. Now, any technique (e. g. Chebyshev approximation, Butterworth approximation) can be used to design a continuous-time filter that satisfies the specifications. The design techniques for continuous-time filters will not be presented in more detail here, you can refer to the literature for more information.

From the transfer function $H_C(f)$ or the impulse response $h_C(t)$ of the continuous-time filter, the discrete-time filter is obtained by applying the inverse transform. In this step, the behavior of the filter in the frequency or time domain should be preserved as well as possible. Most importantly, it is generally required that the imaginary axis of the $s$ plane is mapped to the unit circle of the $z$ plane and that the poles in the left half-plane of the $s$ plane are mapped to poles inside of the unit circle of the $z$ plane. The second requirement ensures that a stable continuous-time filter also leads to a stable discrete-time filter.

For the impulse invariance method, the discrete-time IIR filter is defined so that its impulse response is the impulse response of the corresponding continuous-time filter sampled with an arbitrary period $T_d$, so

$$h(k) = T_d h_c(kT_d). \tag{4.38}$$

The frequency response $H(\Omega)$ of the discrete-time filter

$$H(\Omega) = \sum_{i=-\infty}^{\infty} H_C\left(\frac{\Omega}{2\pi T_d} - \frac{i}{T_d}\right) \tag{4.39}$$

is equal to the periodically repeated frequency response $H_C(f)$ of the continuous-time filter (see Figure 4.12). If $H_C(f)$ is band-limited so that $H_C(f) = 0$ for $|f| > 1/(2T_d)$, the frequency responses are exactly identical for $-\pi \leq \Omega \leq \pi$. Since a real continuous-time filter can never be ideally band-limited, distortions caused by aliasing as shown in Figure 4.12 arise.

If the frequency response of a continuous-time filter converges sufficiently fast to zero for high frequencies, the impulse invariance method can provide a discrete-time filter which can serve as a sufficiently good approximation. However, this also implies that this method only allows the realization of filters that feature a stopband at high frequencies. If for example a highpass or a band-stop should be realized, an additional band limitation for high frequencies is inevitable.

In summary, the impulse invariance method successfully manages to replicate the time behavior of the continuous-filter it is based on, but it leads to nonlinear distortions due to aliasing so that the requirements concerning the frequency response

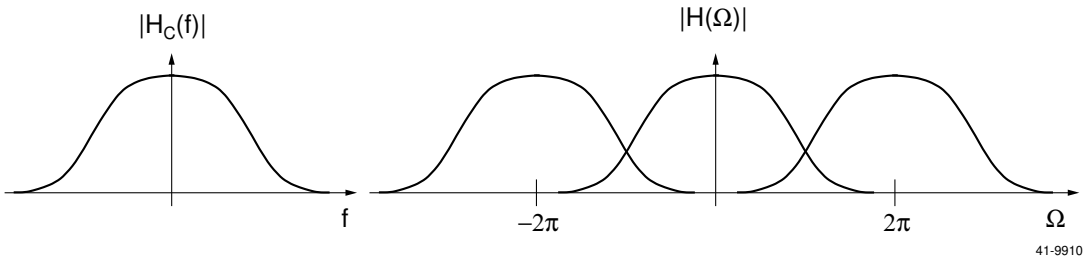



**Figure 4.12:** Aliasing when using the impulse invariance method (according to Oppenheim 1999)

are not necessarily fulfilled anymore by the discrete-time filter. This can only be handled by choosing stricter tolerance scheme requirements than actually necessary.

## 4.5 Special Filters

### 4.5.1 Differentiation

The ideal discrete-time differentiator has, analogously to the continuous-time differentiator, the periodic frequency response (see Figure 4.13).

$$H_{D_0}(e^{j\Omega}) = j\Omega \quad ; \quad -\pi \leq \Omega \leq \pi \tag{4.40}$$

The corresponding non-causal impulse response can be obtained by inverse Fourier transform

$$\begin{aligned} h_{D_0}(k) &= \frac{1}{2\pi}\int_{-\pi}^{\pi} j\Omega e^{j\Omega k} d\Omega \\ &= \frac{\cos(\pi k)}{k} \\ &= \begin{cases} \frac{(-1)^k}{k} & ; \quad k = \pm 1, \pm 2, \ldots \\ 0 & ; \quad k = 0 \end{cases} \end{aligned} \tag{4.41}$$

The differentiating behavior can only approximately be achieved with a finite filter order.

The simplest approximation is to replace the differential quotient by the difference quotient according to

$$y(k) = x(k) - x(k-1). \tag{4.42}$$

A better approximation is achieved by applying Stirling's interpolation formula [2]:

$$y(k) = \frac{1}{12}(x(k-4) - 8x(k-3) + 8x(k-1) - x(k)). \tag{4.43}$$

Very small approximation errors can be reached by the Chebyshev approximation using the Parks-McClellan algorithm presented in the previous section.

[2] see e. g. Bronstein 1983

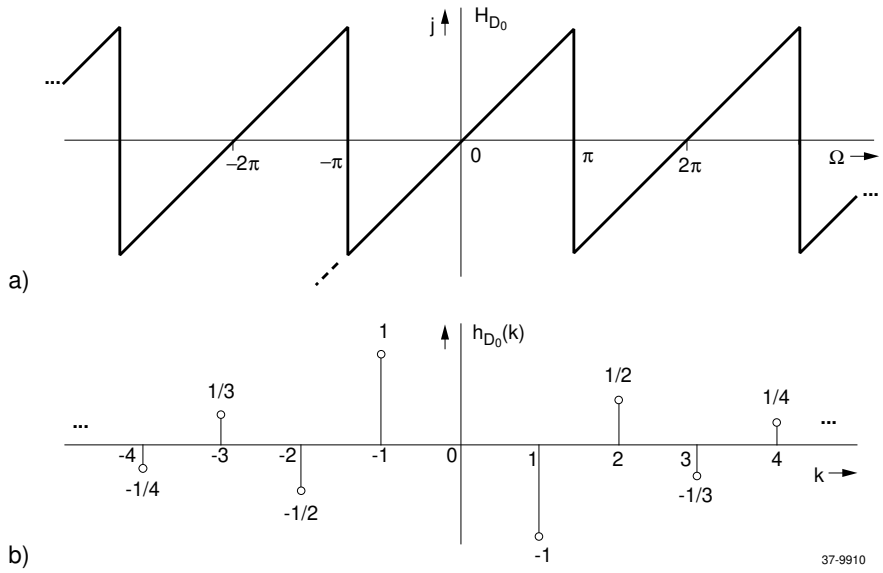



**Figure 4.13:** Ideal differentiator: frequency response (a), impulse response (b)

### 4.5.2 Integration

The ideally integrating systems has the frequency response

$$H_0(e^{j\Omega}) = \frac{1}{j\Omega} \tag{4.44}$$

For the approximation, the various integration formulas of numerical mathematics can be considered which can be represented as recursive difference equations:

- Rectangle rule: $y(k) = y(k-1) + x(k)$
- Trapezoid rule: $y(k) = y(k-1) + \frac{1}{2}x(k) + \frac{1}{2}x(k-1)$
- Simpson's rule: $y(k) = y(k-2) + \frac{1}{3}x(k) + \frac{4}{3}x(k-1) + \frac{1}{3}x(k-2)$

All systems are conditionally stable because of the pole at $z = 1$ and approximate the ideal behavior in the low frequency range. The best approximation is provided by Simpson's rule.

### 4.5.3 Hilbert transform

The Hilbert transform causes a phase shift by $\Delta_\phi = 90$ degrees. Such systems are applied in the context of quadrature modulation, single-sideband modulation and the determination of the envelope (analytical signal). The ideal discrete-time (non-causal) Hilbert transformer is described by

$$H_{H_0}(e^{j\Omega}) = -j\mathrm{sgn}(\Omega) \qquad -\pi \leq \Omega \leq \pi \tag{4.45}$$

$$h_{H_0}(k) = \frac{1}{2\pi}\int_{-\pi}^{\pi} -j\mathrm{sgn}(\Omega)e^{j\Omega k}d\Omega \tag{4.46}$$

$$= \frac{1}{\pi k}(1-\cos(\pi k)) \tag{4.47}$$

$$h_{H_0}(k) = \begin{cases} 0 & ; \quad k = 0, \pm 2, \pm 4 \ldots \\ \frac{2}{\pi k} & ; \quad k \,\mathrm{odd} \end{cases} \tag{4.48}$$

A causal approximation with even filter order $G$ (type 3) is of the form

$$h_{H_2}(k) = \begin{cases} h_{H_0}(k - G/2) & ; \quad 0 \leq k \leq G \\ 0 & ; \quad \text{else.} \end{cases} \tag{4.49}$$

Better results are achieved by filters with odd order (type 4) as these do not have a zero at $\Omega = \pi$.

Good results are obtained with the design programs of Parks and McClellan with the requirement of an odd filter order.

### 4.5.4 Interpolation

Provided that the sampling theorem is satisfied, discrete-time signals can be ideally interpolated with an integer factor $r$ by ideal lowpass filtering. To do so, $r-1$ zero values are first inserted between each pair of two adjacent samples as shown in Figure 4.14 before a lowpass filter is applied. The design problem reduces to the design of a lowpass.

## 4.6 Properties of Real Filters

### 4.6.1 FIR Filters

When realizing digital filters, inaccuracies arise due to the finite calculation precision (quantization), particularly for fixed-point arithmetics.

The following effects are discriminated:

- Finite precision of the filter coefficients
- Finite precision of arithmetic operations
- Restrictions to the numerical range

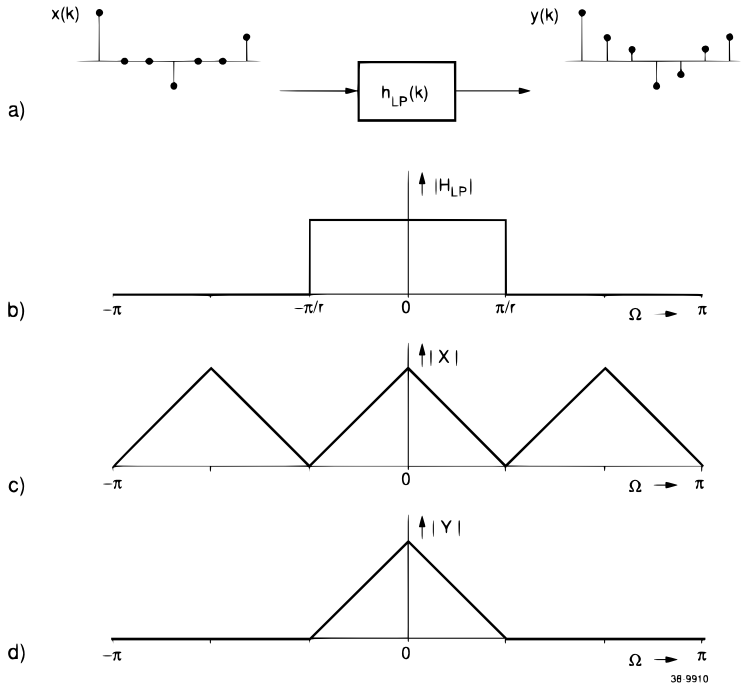



**Figure 4.14:** Ideal interpolation with the factor $r = 3$: a) time domain, b) frequency response of the interpolation lowpass, c) spectrum of the input signal, d) spectrum of the interpolated signal

- Quantization of input and output signal (not covered in more detail here)

The nonlinear nature of these effects makes their analysis very difficult. Therefore, linear analogous models are introduced that describe the relations with sufficient accuracy.

### Finite precision of the filter coefficients

After the quantization of the filter coefficients, the zeros of the transfer function can no longer lie at arbitrary positions in the complex $z$ plane. Rather than that, they are restricted to discrete positions (see Figure 4.15). The changed zero locations result in a change of the frequency response.

The error caused by the quantization of the coefficients can be described as follows. It follows with the quantized coefficients

$$\hat{h}(k) = h(k) + \Delta h(k) \tag{4.50}$$

that

$$\begin{aligned} \hat{H}(e^{j\Omega}) &= \sum_{k=0}^{G} \hat{h}(k) \cdot e^{-jk\Omega} && (4.51)\\ &= \sum_{k=0}^{G} h(k) \cdot e^{-jk\Omega} + \sum_{k=0}^{G} \Delta h(k) \cdot e^{-jk\Omega} && (4.52)\\ &= H(e^{j\Omega}) + \Delta H(e^{j\Omega}) && (4.53) \end{aligned}$$

As a result of the frequency response error $\Delta H$, a tolerance scheme that was satisfied prior to the quantization (filter design in floating-point arithmetics), can be violated

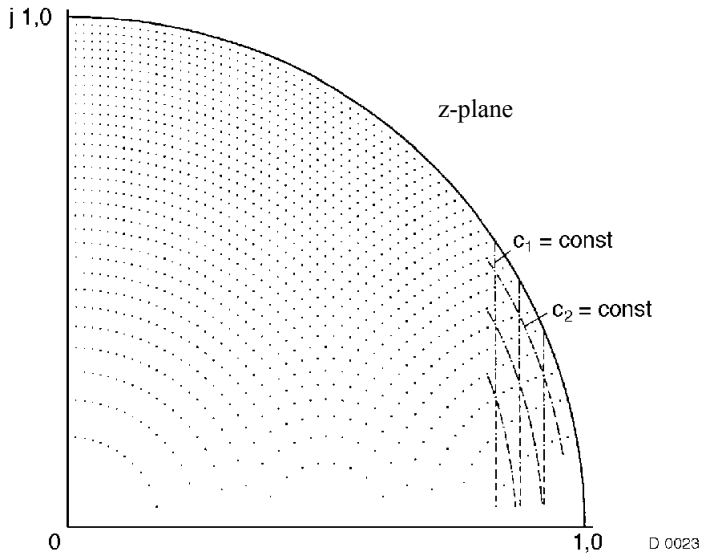



**Figure 4.15:** Grid of the possible pole locations of a second-order filter (direct form) and quantization of the coefficients with 5 bit each

thereafter. Still, the linear phase property is preserved as the coefficient symmetry remains unaffected by the quantization.

### Finite precision of arithmetic operations

When using fixed-point arithmetics, the multiplication of two signed numbers of word length $w$ leads to an increase to $2w + 1$ binary digits. The increased word length must again be shorted to $w$ bits, for example by means of rounding. The quantization noise caused in the process can well be approximated with the model of an additive, uniformly distributed noise source with the variance

$$\sigma_r^2 = \frac{Q^2}{12} \qquad ; Q = 2^{-w+1} \text{ for } |x| \leq 1 \tag{4.54}$$

where $Q$ denotes the quantization step size. For the calculation of each of the filter output values $y(k)$, $n$ subproducts $\hat{h}(i) \cdot x(k-i)$ must be added up. When rounding after each multiplication, i. e. before the accumulation, the produced noise variance is $\sigma_y^2 = n \cdot Q^2/12$ in total.

If on the other hand, the unrounded products with appropriately increased word length are added up first and the rounding is applied only after the accumulation, the resulting noise variance is merely $\sigma_y^2 = Q^2/12$. For this reason, signal processors possess accumulators with increased word width.

### Restrictions to the numerical range

Yet another realization problem occurs, particularly for fixed-point arithmetics, when the numerical range is limited to $|x| \leq 1$. To prevent overflow, the input signal must

be scaled so that the output signal will always stay within the valid numerical range. The "secure" scaling with the factor $K$ follows from the requirement

$$|y(k)| = \left|\sum_{i=0}^{N} h(i) \cdot K \cdot x(k-i)\right| \tag{4.55}$$

$$\leq K \cdot |x_{\max}| \cdot \sum_{i=0}^{N} |h(i)| \overset{!}{<} 1. \tag{4.56}$$

This yields for $|x_{\max}| = 1$

$$K < \frac{1}{\sum_{i=0}^{N} |h(i)|}. \tag{4.57}$$

### 4.6.2 IIR Filters

For digital IIR filters, the same quantization effects generally occur as for FIR filters. However, due to the recursive structure of the IIR Filters, periodic error patterns, so-called limit cycles, can arise which cause the filter to become unstable. In this case, it is no longer sufficient to describe the quantization effect with additive noise sources.

#### Limit cycles

Because the quantization is a nonlinear operation, the theoretical analysis of the limit cycles is difficult and only possible in simple special cases. To illustrate the issue, the simple system according to Figure 4.16 will be considered.

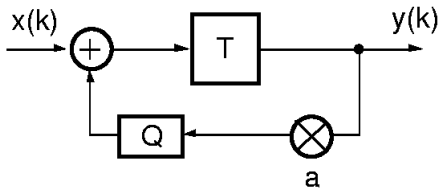



**Figure 4.16:** First-order system with rounding

Without a quantization in the feedback path, the impulse response of the system is:

$$y(k) = a^{k-1} \text{ for } k \geq 1.$$

The system is thus stable for $|a| < 1$.

Now, the result of the multiplication with the constant $a$ is rounded in such a way that it is an integer multiple of $Q$. Figure 4.17 shows the impulse response of the system for different values of $a$ and $Q$. Three possible cases can be distinguished: For $|a| < 1/2$, the output sequence $y(k)$ will be zero after a finite number of cycles. For $1/2 \leq a < 1$, a stationary final value will be reached which is different from zero; this is referred to as a limit cycle of period 1. Finally, for $-1 < a \leq -1/2$, the result will be an alternating output sequence, i. e. a limit cycle of period 2. This means

that the system may not necessarily still be asymptotically stable, i. e. the output sequence $y(k)$ will not decay completely for $k \to \infty$ although the input sequence $x(k)$ disappears for $k \geq 1$.

In case of higher-order feedback systems with quantization, limit cycles with larger periods can occur.

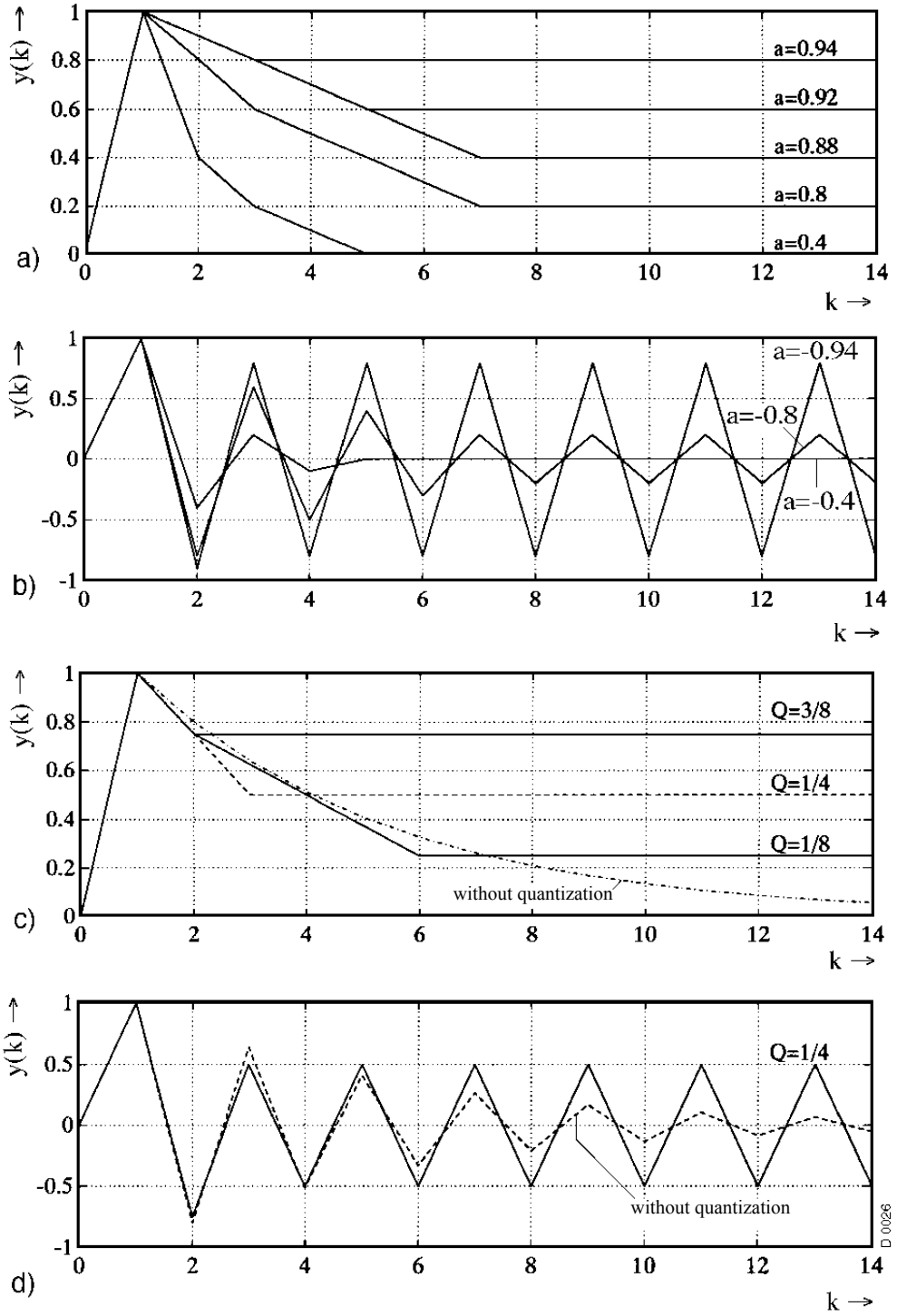



**Figure 4.17:** Impulse response of the system depicted in Figure 4.16
a) for different values of $a$ ($a > 0$) and $Q = 0.1$
b) for different values of $a$ ($a < 0$) and $Q = 0.1$
c) for $a = 0.8$ and different values of $Q$
d) for $a = -0.8$ and $Q = 1/4$ and without quantization

### Overflow oscillations (large limit cycles)

When realizing filters in two's complement arithmetics with the numerical range

$$-1 \leq x \leq 1 - 2^{-w+1}, \tag{4.58}$$

an overflow can occur when adding two numbers. This overflow results in an incorrect sum value with the opposite sign according to the overflow characteristic of the two's complement arithmetics (see Figure 4.18). This signal jump can cause an instable system behavior via the signal feedback. The oscillations that occur as a result are called overflow oscillations. They can persist even when input signal is turned off.

In a second-order filter block, these limit cycles can be avoided if the two's complement characteristic is replaced by the saturation characteristic (Figure 4.19).

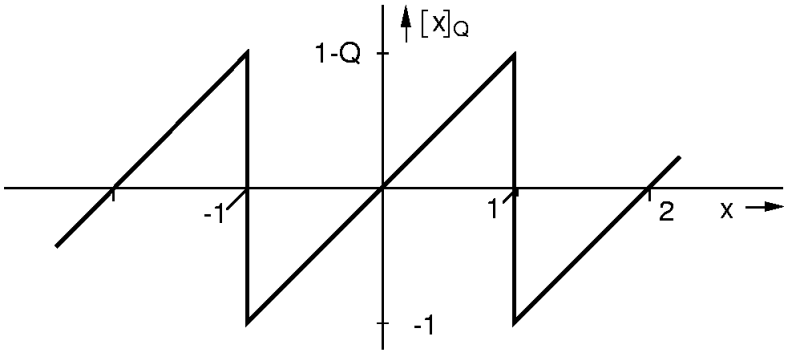



**Figure 4.18:** Saturation characteristic of the two's complement arithmetics ($Q = 2^{-w+1}$).

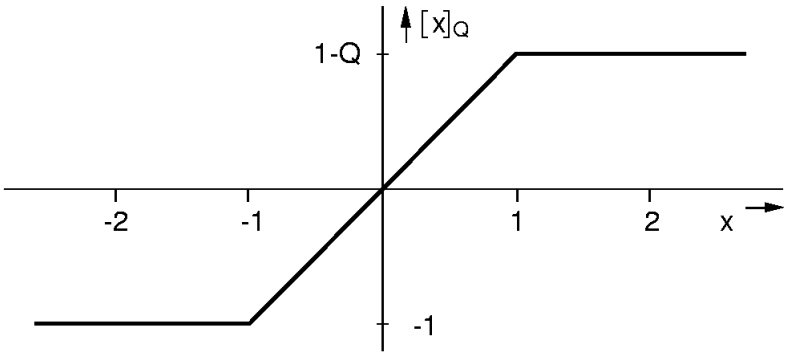



**Figure 4.19:** Saturation characteristic

## 4.7 Signal Analyzing and Digital Filter Design in MATLAB

The *Signal Processing Toolbox* provides an extensive collection of functions for digital signal processing. This includes a variety of functions that can be used for analyzing signals and designing filters.The documentation of this toolbox can be inspected by

typing `doc` in command window. Under **Contents →Signal Processing Toolbox** you could see explicit explanations and examples to help you to use this toolbox.

*Signal Processing Toolbox* offers two convenient graphical user interfaces for analyzing signals and designing digital filters, respectively. In the following sections, we will present a brief guidance for you to use these two tools. For a more thorough description, you can refer to the MATLAB Helpdesk.

Beside the convenient option to design and display signals, filters and spectra from the graphical user interface, the fundamental MATLAB commands should also be known. These can be used for example in your own MATLAB scripts and functions to solve specific problems.

### 4.7.1 Signal Analyzer App.

The user interface of Signal Analyzer could be opened with the `signalAnalyzer` command under MATLAB command window.

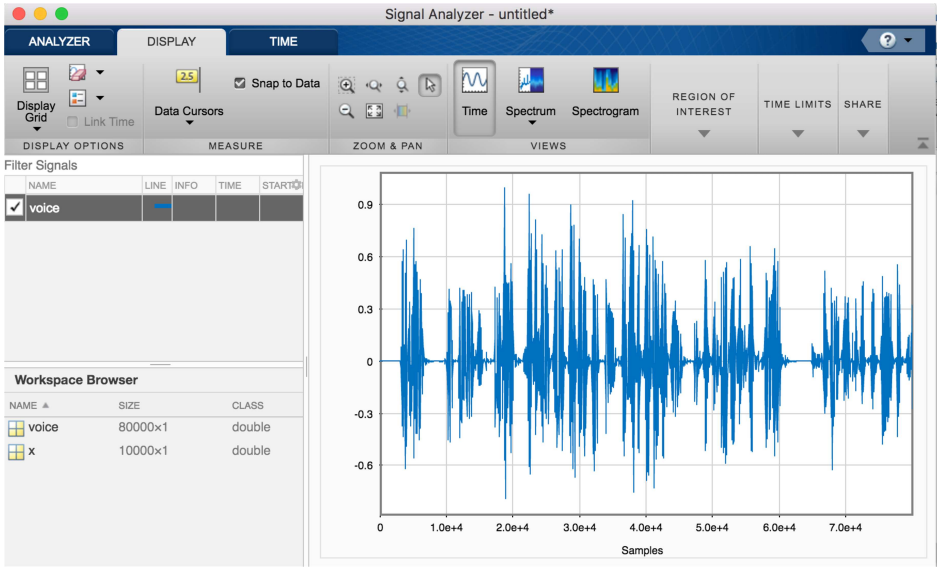



**Figure 4.20:** Main window of Signal Analyzer

The main window (see Figure 4.20) of Signal Analyzer is divided into three areas. The first part in the bottom right is the plotting area, where the selected signal is plotted. According to your choice **Time/Spectrum/Spectrogram** under the **DISPLAY** selection menu on the top of Figure 4.20, signals can be plotted by time, by frequency(spectrum), or by time and frequency(spectrogram). The area in the bottom left of Figure 4.20 is a copy of the current MATLAB workspace, where all available data are presented here. The last part in the middle left is the signal table which contains all signals that are imported into Signal Analyzer. The easiest way to import a signal in workspace into Signal Analyzer is left-clicking on a signal in workspace browser, then hold and drag it into the other two areas. If you hold and drag a signal into the plotting area, the signal will be imported and plotted automatically. In contrast, if you drag a signal into the signal table, the signal will be imported but not plotted.

Under the selection menu **ANALYZER** you can apply different kinds of filters to selected signals (only available in MATLAB 2018a or later versions), set sampling frequencies or export signals to workspace according to you requirement.

### 4.7.2 Filter Designer App.

The Filter Designer app. is another extremely powerful tool for digital filter design. The filter designer can be summoned by running `filterDesigner` under the command window.

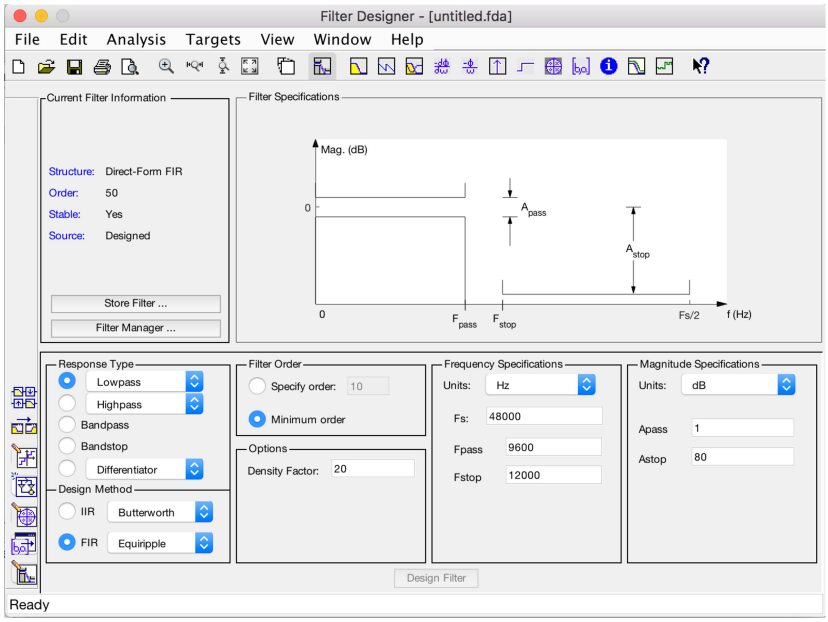



**Figure 4.21:** Main window of Filter Designer

You can either import an existing filter to filter designer or create a new filter. For the purpose of designing a new filter, an appropriate design algorithm must be chosen from the different filter types by pressing the corresponding button. Besides the design algorithm, other parameters like the sampling frequency, cutoff frequency etc. must also be specified in order to meet the designing requirement.

The button **Minimum Order** is used to specify the order of the filter. If it is activated, an optimal filter of variable order will always be created. If it is inactive, a filter of fixed order will be created, the order of which might exceed the minimum required order. The filter itself can now be created by moving the limits in the editor. In the pole-zero design mode (➔Pole/Zero Editor), see symbol in the bottom left menu area of the filter designer, the filter is created by moving the poles and zeros. It should be ensured that when selecting this mode, the order of the current filter should not be too large as otherwise, it requires a considerable time to build the window. The effect of the filter on the amplitude and phase response as well as the step and impulse response and the pole-zero diagram can be inspected by clicking the menubar on the top of Figure 4.21 during the filter design process.

When the filter meets the requirements, it can be exported to the MATLAB Workspace. To do so, the subitem **Export** is selected in the **File** menu of the main window. Here, the filter to be exported is selected and then confirmed with **Export to workspace**.

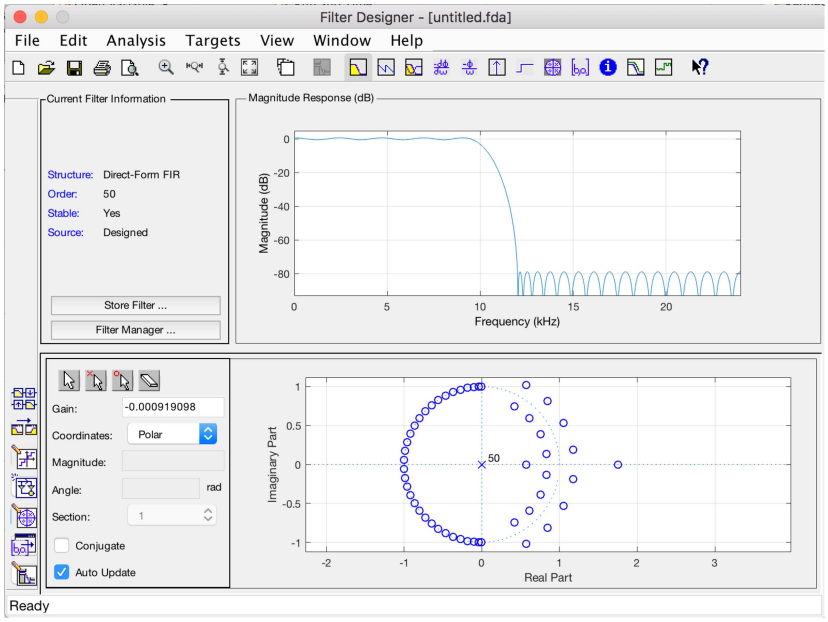



**Figure 4.22:** Filter Designer - Frequency spectrum and zero/pole diagram of a FIR low pass filter with order 50

The filter can either be exported in the format of a vector and a matrix containing the coefficients in case of IIR-filter or one single array containing the coefficients of numerator in case of FIR-filter, or in the format of an object. Here is an example for a FIR lowpass filter, where the coefficients are exported in the format of an object.

In the MATLAB workspace, an object with the name of the `filter` now exists. The filter parameters are stored under `filter.Numerator` as numerator of the filter terms.

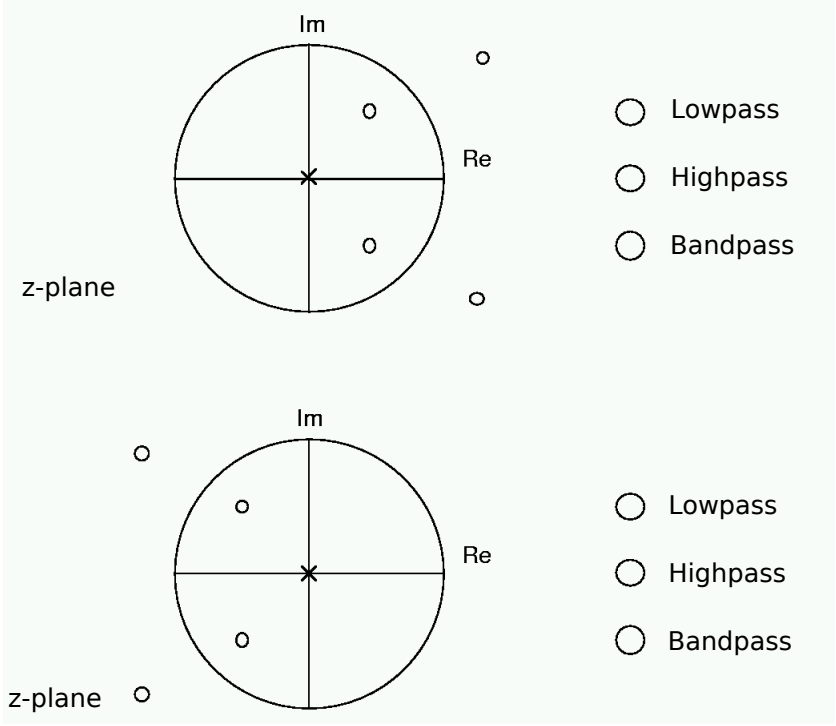
```
>> filter

filter =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x21 double]
    PersistentMemory: false

>> filter.Numerator

ans =

  Columns 1 through 7

    0.0389    0.0026   -0.0302   -0.0181    0.0357    0.0394   -0.0450

  Columns 8 through 14

   -0.0923    0.0472    0.3119    0.4488    0.3119    0.0472   -0.0923
  Columns 15 through 21

   -0.0450    0.0394    0.0357   -0.0181   -0.0302    0.0026    0.0389
```

## 4.8 Exercises

**For the following experiments, a sampling frequency of $f_s = 8000$ Hz should be assumed unless stated otherwise.**

### Exercise 4.1

Two filters characterized by their pole-zero diagrams are given. What kind of filters are they? Use the pole-zero editor of filter designer to answer this question.




### Exercise 4.2 FIR experiment: rectangular, Hamming- and Blackman windows

A lowpass filter of order $G = 16$ with the impulse response

$$y(k) = \frac{\sin(\frac{\pi}{2}(k-8.5))}{\frac{\pi}{2}(k-8.5)}, \qquad k = 1,2,\ldots,16 \tag{4.59}$$

is considered.

Transform the impules response to the frequency domain using a rectangular window function. Use the MATLAB function `freqz` for this purpose. Normalize the transfer function to an amplification factor of 0 dB (amplification = 1) at the frequency 0 Hz. Display the normalized transfer function graphically.

What cutoff frequency does this lowpass have?

Generate another lowpass filter of order $G = 64$. Note that you have to choose the appropriate temporal shift for a symmetrical impulse response! Transform this impulse response to the frequency domain using a rectangular window function as well and display the normalized transfer function graphically.

Fill in the first two lines of the tables. The width of the transition region of the lowpass filter is defined as $f_S - f_P$ where $f_P$ denotes the frequency where the minimum ripple is reached for the last time and $f_S$ the frequency at which the minimum attenuation in the stopband is observed for the first time. See Fig. 4.5 for reference.

Transform the lowpass impulse response of order $G = 64$ to the frequency domain with a Hamming and a Blackman window. To do so, use the MATLAB functions `hamming` and `blackman` respectively.

Compare the different lowpass filters by completing the table.

| | Minimum attenuation in the stopband [dB] | width of the transition region [Hz] | Maximum overshoot in in the passband. [dB] |
|---|---|---|---|
| Rectangle 16 | | | |
| Rectangle 64 | | | |
| Hamming 64 | | | |
| Blackman 64 | | | |

The influence of the window function:

## Exercise 4.3 FIR experiment B: Kaiser window

Consider the tolerance scheme from Figure 4.23. Construct a lowpass filter that satisfies these tolerances with minimum filter length using the filter designer and the algorithm option **Kaiser Window FIR**. Export the filter coefficients to the MATLAB Workspace as you will still need these for the subsequent experiment.

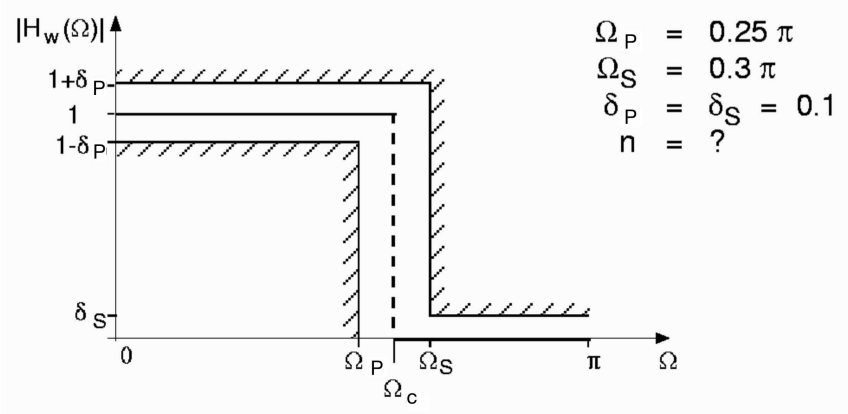



**Figure 4.23:** Tolerance scheme for the filter design

Use the **Filter Viewer** to inspect the magnitude, phase, group delay, pole-zero diagram, impulse response and step response.

Familiarize yourself with the other functions and parameters of the filter designer.

## Exercise 4.4 FIR experiment C: coefficient quantization

Simulate a coefficient quantization with the help of MATLAB by rounding the filter coefficients exported from FIR experiment B to 64 possible values. Import the rounded coefficients to the filter designer and check whether the filter still complies with the given tolerance scheme. For the quantization, the minimum and the

maximum of the filter coefficients must be determined first. Afterwards, each of the coefficients is shifted by the minimum so that the value range's lower bound is 0. The resulting values are rounded to integer numbers according to the number of quantization steps (maximum is mapped to the maximum integer value) and then converted back to the original value range. The quantization is applied in the process of rounding to the integer numbers. Repeat the same investigations for 4 quantization steps as well.

## Exercise 4.5 FIR experiment D: Differentiation

Generate the impulse responses of the following two differentiators:

1. Difference: $y(k) = x(k) - x(k-1)$
2. Stirling's formula: $y(k) = \frac{1}{12}(x(k-4) - 8x(k-3) + 8x(k-1) - x(k))$

You can use for example the MATLAB function `impz` or import the transfer function $H(z)$ to the filter designer and display it there.

Use the Parks-McClellan algorithm to design a differentiator with the following parameters (see Figure 4.24):

filter length $n = 32$, $\Omega_u = 0$, $\Omega_o = \pi$.

Use the MATLAB function `firpm` for this purpose.

**Note:** The frequencies passed to `firpm` must be normalized to 1.

Compare the transfer functions of the different approximations (linear scale in the frequency domain) and identify the frequency ranges in which the filters can approximately be used as differentiators.

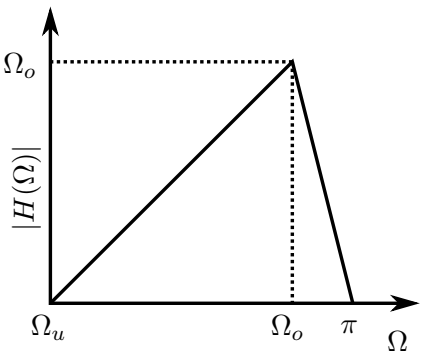



**Figure 4.24:** Magnitude frequency response of a differentiator

## Exercise 4.6 IIR experiment A: Different approximation techniques

The choice of the design technique affects the filter order (and thus the required computation time), the frequency and phase behavior and the time behavior of a filter. In this experiment, these filter properties will be compared with the help of the filter designer. To do so, design a bandpass filter of order 6 with the help of the filter designer using the different IIR algorithm options. This allows a comparison of the filter slope in the transition regions for identical filter computational effort.

Use the following parameters for the filter design:

1. $f_s = 8000$ Hz

2. $F_{c1} = 1000$ Hz
3. $F_{c2} = 2000$ Hz

Compare the frequency behavior of the filters designed with the different approximation techniques. In which regions is the frequency response smooth or rippled? Is the slope between stopband and passband flat or steep? This is only about a qualitative description, not about numerical values!

| Appr. Technique | Passband | Stopband | Slope |
|---|---|---|---|
| Butterworth | | | |
| Chebyshev I | | | |
| Chebyshev II | | | |

Which filter offers the *steepest* and which filter the *flattest* descent in the transition from the passband to the stopband?

Now inspect the different phase behavior of the filters.

When evaluating the phase responses, keep in mind that most of all, the phase behavior in the passband is of interest.

Take a look at the positions of the poles and zeros of the different filters.

Which filter has poles close to the unit circle in the $z$ plane? Which filter's poles are farthest away from the unit circle? How can you know that these filters must be bandpasses?

Finally, compare the time behavior of the filters. Which filter takes the least time to approach a stable condition? Which impulse response has the shortest length, which filter has the longest impulse response? What is the relation between the time behavior and the locations of their poles (distance to the unit circle)?

### Exercise 4.7 IIR experiment B: Recursive difference equations

First, a simple recursive filter will be realized by its difference equation and the influence of the sampling frequency on the cutoff frequency of the filter will be investigated.

Generate the impulse response of the recursive filter with the difference equation

$$y(k) = x(k) + x(k-1) + 0.16y(k-1). \tag{4.60}$$

The sampling frequency should be $f_s = 500$ Hz.

What is the transfer function $H(z)$ of this filter?

Apply the Fourier transform on the impulse response and inspect the frequency response. What kind of filter is this?

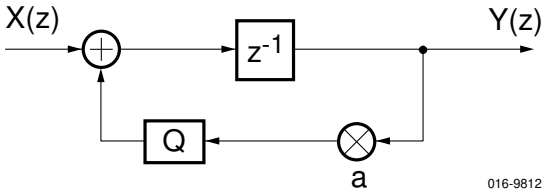
Determine the cutoff frequency of the filter ($-3$ dB point) and enter it in the table below. Now generate the impulse responses of the filter with the same different equation for the sampling frequencies $f_s = 1000$ Hz und $f_s = 1500$ Hz. Transform both impulse responses into the frequency domain and determine the cutoff frequencies ($-3$ dB points) for both filters.

| Sampling frequency | Cutoff frequency |
|---|---|
| 500 Hz | |
| 1000 Hz | |
| 1500 Hz | |

What is the relation between the cutoff frequency and the sampling frequency of a digital filter?

## Exercise 4.8 IIR experiment C: Word length reduction

The effect of a word length reduction will be analyzed for the example of a recursive first-order filter.




Create a simple MATLAB function for the realization of this filter. In it, you can use the MATLAB functions `round` or `floor` to model the quantization effect which can for example be regulated via a function parameter.

It holds for the input sequence that

$$x(k) = \begin{cases} 10 & ; \quad k = 0 \\ 0 & ; \quad \text{else.} \end{cases} \tag{4.61}$$

It is further assumed that $y(k) = 0$ for $k < 0$.

For the following cases, the output sequences should now be determined and displayed graphically:

1. $a = -0.92$ without quantization
2. $a = -0.92$ and rounding to the closest integer value
3. $a = -0.92$ and word length reduction by cutting off the second decimal place
4. $a = +0.92$ with and without rounding. What effect occurs now?

# Chapter 5

# Adaptive Filtering

Beside the digital filters with fixed and time-invariant filter coefficients presented in the previous experiment, so-called *adaptive* filters are also commonly used in digital signal processing. The filter coefficients of these filters are time-variant and are readjusted (adapted) during the runtime. Applications of such filters are for example noise reduction, signal coding and acoustic echo cancellation, the latter of which will be the focus of this laboratory experiment. The goal is to introduce this important class of filters and to convey how they can be realized and inspected with the help of MATLAB. In the process, the knowledge about the techniques and functions presented in the previous experiments should be further improved.

## 5.1 Adaptive Filters for the Acoustics Echo Cancellation

Many devices for speech communication, like e. g. cell phones or convenience phones, are equipped so that they can be used hands-free nowadays. To increase the user comfort or for safety reasons (for example in the car), a loudspeaker-microphone setup is used instead of the telephone handset.

The acoustic coupling between loudspeaker and microphone can cause that beside the (desired) speech signal of the near speaker, the signal of the far speaker is transmitted as well. This undesired transmission leads to disruptive 'echos' which should be compensated by means of adaptive filtering[1].

## 5.2 Problem Description

The fundamental task will be explained with the help of Figure 5.1. The problem is the acoustic coupling between the loudspeaker and the microphone. For simplification, it will be assumed in the following that the transmitted signal of the far speaker $\tilde{x}(k)$ and the signal $y(k)$ are given in digitized form. The microphone records not only the desired signal $s(k)$ of the near speaker but also the undesired background noise $n(k)$ and particularly the signal from the far speaker $\tilde{x}(k)$ which is modified by the acoustic transmission from the loudspeaker to the microphone. The signal component $\tilde{x}$ is created by numerous acoustic reflections. It is generally referred

[1]The following presentation of these techniques is based on Vary, Heute, Hess *Digitale Signalverarbeitung*, Teubner, 1998 where references to further literature can also be found.

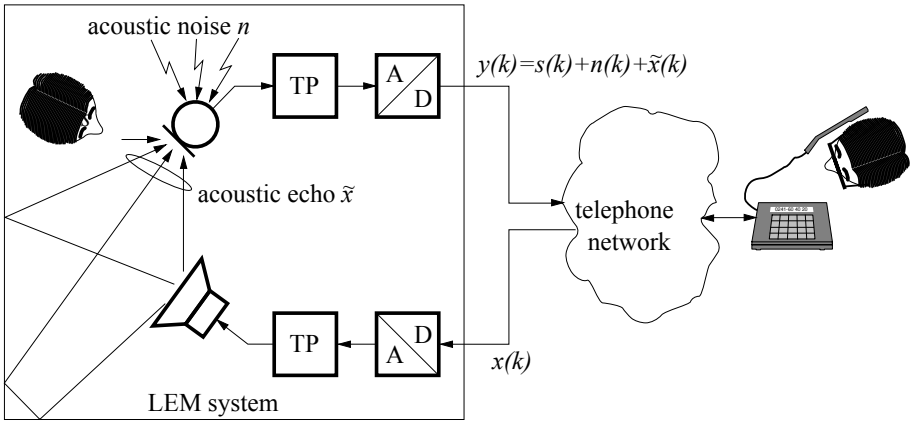



**Figure 5.1:** Loudspeaker-enclosure-microphone (LEM) system of hands-free equipment.

to as the acoustic echo to distinguish it from the electric line echo of the telephone network. It therefore holds fort the digitized transmitted signal that

$$y(k) = s(k) + n(k) + \tilde{x}(k). \tag{5.1}$$

The aim of the acoustic echo cancellation is to prevent the echo signal $\tilde{x}(k)$ from being transmitted to the far speaker. This should ensure the stability of the electroacoustic loop if the far speaker uses a hands-free capability as well. It should further be prevented that the far speaker perceives his own speech signal with a delay caused by the telephone or mobile network. Such a feedback would to some extent have a considerable impact on the conversation between the speakers.

## 5.3 Possible Solutions and Evaluation Criteria

In many (simple) telephone devices and cell phones, a so-called *voice-controlled echo suppressor* can be found for echo cancellation, as depicted in Figure 5.2. With

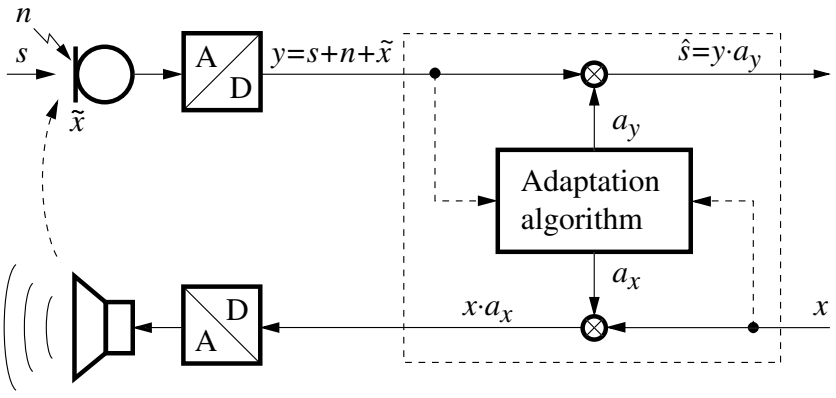



**Figure 5.2:** Loudspeaker telephone with voice-controlled echo suppressor.

one variable damping element in both the transmitting and receiving branch, the branches are attenuated differently depending on the speech activity of the two

speakers, with the total attenuation of the loop never being below some minimum value, e. g. 40 dB. As the damping factors should satisfy the condition

$$-(20\log a_x + 20\log a_y) = 40\text{dB} \tag{5.2}$$

it is possible only to a limited extent for both the near and the far speaker to speak simultaneously. This situation is referred to as double talk. Techniques that generally allow double talk will be considered in the following.

The basic principle of these methods is illustrated in Figure 5.3. The loudspeaker-

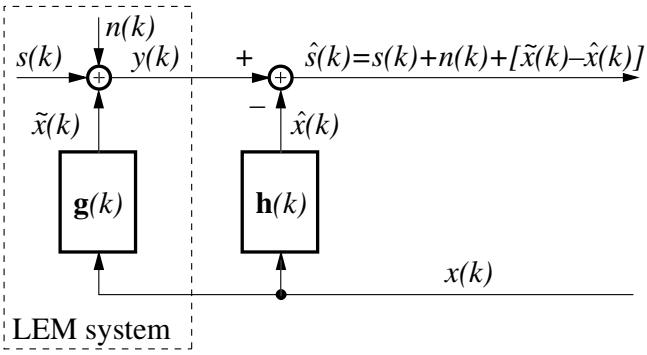



**Figure 5.3:** Discrete-time model of a hands-free setup with echo cancellation.

enclosure-microphone (LEM) system depicted in Figure 5.1 is shown as discrete-time model. The transmission of the far speaker $x(k)$ over the LEM system should be described by means of a causal, linear filter. The impulse response of this time-variant filter is assumed to be limited to $m'$ coefficients and is represented by the coefficient vector

$$\mathbf{g}(k) = [g_0(k), g_1(k), \ldots, g_{m'-1}(k)]^T \tag{5.3}$$

that is time-variant in $k$. Examples for measured impulse responses of LEM systems are shown in Figure 5.4.

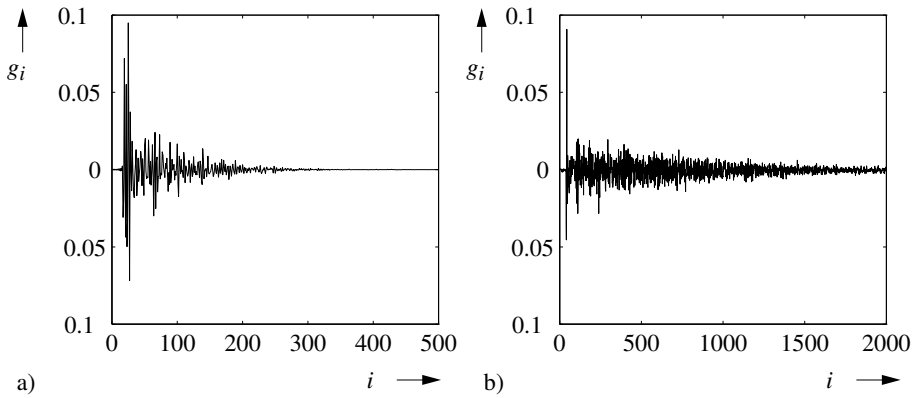



**Figure 5.4:** Impulse response of an LEM system measured in a car (left) and in an office room (right).

To compensate the undesired echo signal $\tilde{x}(k)$, a time-variant transversal filter of length $m$ with the coefficient vector

$$\mathbf{h}(k) = [h_0(k), h_1(k), \ldots, h_{m-1}(k)]^T \tag{5.4}$$

is employed. In the following, this transversal filter will alternatively be called the *compensation filter* as its task is to 'compensate' the undesired echo signal $\tilde{x}$ in the transmitting branch. If the coefficients of the impulse responses $\mathbf{g}(k)$ and $\mathbf{h}(k)$ match exactly, the echo signal is eliminated (compensated) completely, i. e. the difference vector

$$\mathbf{d}(k) = \mathbf{g}(k) - \mathbf{h}(k) \tag{5.5}$$

is a zero vector. The aim of the (acoustic) echo cancellation is therefore to replicate the unknown and time-variant coefficients of the impulse response of the LEM system $\mathbf{g}(k)$ as well as possible with $\mathbf{h}(k)$.

A measure for the quality of the echo cancellation is the *relative system distance*

$$D(k) = \frac{||\mathbf{d}(k)||^2}{||\mathbf{g}(k)||^2} \tag{5.6}$$

where the system distance is given by $||\mathbf{d}||^2$ and $||\mathbf{d}||^2 = \mathbf{d}^T\mathbf{d}$ is the squared (Euclidean) vector norm. Typically, the logarithmic distance $10\log D(k)$ is expressed in dB. When taking the difference of the vectors according to equation (5.5), the shorter vector is filled up with zeros if the two lengths of $\mathbf{g}(k)$ and $\mathbf{h}(k)$ are not the same. In real systems, the impulse response $\mathbf{g}(k)$ is usually not known which is why the relative system distance according to equation (5.6) can only be evaluated in simulations.

A measure that correlates with the subjective auditory impression is the achievable reduction of the power of the echo signal $\tilde{x}(k)$. The corresponding measure is referred to as the *echo return loss enhancement (ERLE)* in the literature. It is defined as

$$\frac{ERLE}{\text{dB}}(k) = 10\log_{10}\frac{\mathrm{E}\{\tilde{x}^2(k)\}}{\mathrm{E}\{(\tilde{x}(k) - \hat{x}(k))^2\}}. \tag{5.7}$$

The required difference signal

$$e(k) = \tilde{x}(k) - \hat{x}(k) \tag{5.8}$$

is also called the *residual echo*. Unlike the system distance according to equation (5.6), the ERLE measure depends on the signals $x(k)$ or equivalently $\tilde{x}(k)$.[2] To compute the ERLE, the expectancy value in terms of an ensemble average over many experiments is replaced by the (time-dependent) short-term expectancy value, as explained in chapter 3. However, due to the limited window length of the short-term expectancy value, this value suffers from an estimation error.

A low system distance $D(k)$ implies a high echo attenuation $ERLE(k)$. The converse is not true, as for example in case of a narrow-band signal $x(k)$ a high echo cancellation (ERLE) can be achieved if the frequency response of the compensation filter "matches" the frequency response of the LEM system only in the relevant bandpass interval (which is possible even in case of a large system distance).

In real systems, the residual echo from equation (5.8) is available only for $s(k) = 0$, i. e. in speech pauses of the near speaker (single talk) and in the noiseless case ($n(k) = 0$, see Figure 5.3). Still, in the (MATLAB) simulation, it is possible to compute $e(k)$ first and then superimpose the components $s(k)$ and $n(k)$ accordingly. This allows for an inspection of the time-dependent echo attenuation in case of double talk and additive interference (background noise) as well.

[2] As the compensation filter is adapted with the help of the signal $x(k)$, the system distance does, as shown in the following Section 5.4, in fact also depend on $x(k)$.

## 5.4 Adaptation Algorithm

The aim of the *Least-Mean Square* (LMS) algorithm is the minimization of the mean squared compensation error

$$\mathrm{E}\{e^2(k)\} = \mathrm{E}\{(\tilde{x}(k) - \mathbf{h}^T(k)\mathbf{x}(k))^2\} \tag{5.9}$$

with

$$\mathbf{x}(k) = [x(k), x(k-1), \ldots, x(k-m+1)]^T \tag{5.10}$$

and

$$\mathbf{h}(k) = [h_0(k), h_1(k), \ldots, h_{m-1}(k)]^T. \tag{5.11}$$

The gradient of the compensation error function can be expressed in vector form as

$$\nabla(k) = \frac{\partial \mathrm{E}\{e^2(k)\}}{\partial \mathbf{h}(k)} = 2 \cdot \mathrm{E}\left\{e(k)\frac{\partial e(k)}{\partial \mathbf{h}(k)}\right\} = -2 \cdot \mathrm{E}\{e(k)\mathbf{x}(k)\}. \tag{5.12}$$

The gradient is proportional to the deviation of the current coefficient vector $\mathbf{h}(k)$ from the optimum coefficient vector $\mathbf{h}_{\mathrm{opt}}(k)$. Consequently, the current coefficient vector is to be changed in the direction of the negative gradient. To do so, the mean gradient is approximated by the current gradient

$$\hat{\nabla}(k) = -2e(k)\mathbf{x}(k). \tag{5.13}$$

This yields the following adaptation rule

$$\mathbf{h}(k+1) = \mathbf{h}(k) + \beta(k)e(k)\mathbf{x}(k) \tag{5.14}$$

where the time-variant stepsize factor

$$\beta(k) = \frac{\alpha}{||\mathbf{x}(k)||^2} \tag{5.15}$$

must satisfy the stability condition

$$0 < \alpha < 2. \tag{5.16}$$

As already pointed out, the residual echo $e(k)$ is not available in an isolated form in real systems. Therefore, the output signal

$$\hat{s}(k) = s(k) + n(k) + e(k) \tag{5.17}$$

is used, so that the adaptation rule

$$\mathbf{h}(k+1) = \mathbf{h}(k) + \beta(k)\hat{s}(k)\mathbf{x}(k) \tag{5.18}$$

is acquired. This adaptation rule however, does not minimize the power of the residual echo $e(k)$, but the power of the output signal $\hat{s}(k)$. Because of equation (5.17), this leads to an undesired distortion of the useful signal $s(k)$. The adaptation would therefore need to be stopped when the near speaker becomes active. In practice, this problem is addressed indirectly by an adaptive stepsize.

The derived algorithm for the adaptive identification of the filter coefficients $\mathbf{h}(k)$ is referred to as *Normalized Least-Mean-Square (NLMS)* algorithm. This principle of time-variant filtering is of major significance for (adaptive) digital signal processing and is used for a large variety of fairly different algorithms.[3]

The stepsize factor $\alpha$ must in practice be controlled adaptively. A possible optimization criterion is the average change of the system distance according to equation (5.5) in the transition from the time $k$ to $k+1$

$$||\mathbf{\Delta}_E(k)||^2 \doteq \mathrm{E}\{||\mathbf{d}(k)||^2\} - \mathrm{E}\{||\mathbf{d}(k+1)||^2\} \tag{5.19}$$

to be maximized, i. e.

$$\frac{\partial ||\mathbf{\Delta}_E(k)||^2}{\partial \alpha(k)} = 0. \tag{5.20}$$

This yields, with several intermediate steps, the solution

$$\alpha_{\mathrm{opt}}(k) = \frac{\mathrm{E}\{e^2(k)\}}{\mathrm{E}\{\hat{s}^2(k)\}}. \tag{5.21}$$

In the case without any interferences $e(k) = \hat{s}(k)$, this results in $\alpha(k) = 1$. Granted that the signals $s(k)$, $n(k)$ and $e(k)$ are statistically independent, the expectancy value from equation (5.17) is

$$\mathrm{E}\{\hat{s}^2(k)\} = \mathrm{E}\{s^2(k)\} + \mathrm{E}\{n^2(k)\} + \mathrm{E}\{e^2(k)\}. \tag{5.22}$$

With this assumption and with equation (5.21), it follows for the *optimum stepsize factor*

$$\begin{aligned} \alpha_{\mathrm{opt}} &= \frac{\mathrm{E}\{e^2(k)\}}{\mathrm{E}\{s^2(k)\} + \mathrm{E}\{n^2(k)\} + \mathrm{E}\{e^2(k)\}} \\ &= \frac{1}{1 + \dfrac{\mathrm{E}\{s^2(k)\} + \mathrm{E}\{n^2(k)\}}{\mathrm{E}\{e^2(k)\}}} \leq 1. \end{aligned} \tag{5.23}$$

If successful at setting the stepsize according to this expression, the stepsize is automatically reduced in case of double talk. The distortion of the speech signal $s(k)$ is prevented or at least reduced and the adaptation must not explicitly be paused during double talk situations. Moreover, the stepsize $\alpha(k)$ has small values if the power of the residual echo $\mathrm{E}\{e^2(k)\}$ becomes small. Usually, the power of the residual signal in double talk situations is smaller than the power of the near speaker $E\{s^2(k)\}$.

In real systems, the residual echo $e(k)$ is not available in isolated form and must be estimated. Determining the residual echo (or its power spectral density) is an important problem in the development of algorithms for echo cancellation. A second problem is the possibly slow speed of convergence of the NLMS algorithm for large filter lengths or correlated speech signals which will be further inspected in the following experiments. In the literature, numerous approaches to solve these problems can be found. However, this is beyond the scope of this introduction.

[3] A detailed description of these filters can for example be found in *Adaptive Filter Theory*, Prentice Hall, 1996.

## 5.5 Additional Measures for Echo Attenuation

In the practical operation of an echo canceller, the echo suppression achieved by the transversal filter may be insufficient in many situations. This can for example be due to a slow adaptation of the compensation filter in response to a fast change of the room impulse response. The length of the compensation filter might also be considerably shorter than the room impulse response, which likewise leads to a residual echo. In some systems, this is accepted deliberately to improve the convergence of the transversal filter and reduce its complexity. Residual echos can become audible particularly in speech pauses of the near speaker and in case of large transmission delays.

A simple measure for their suppression is provided by the so-called *center-clipper* as shown in Figure 5.5. The output signal $\hat{s}(k)$ of the echo canceller is (partly) free

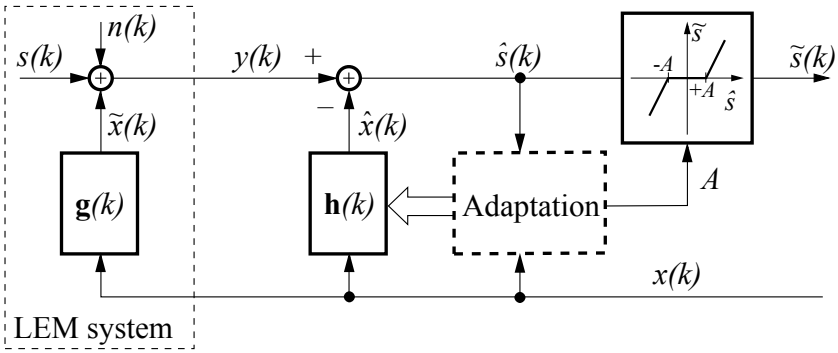



**Figure 5.5:** Echo canceller with center-clipper for the suppression of residual echos.

from residual echos by the rule

$$\tilde{s}(k) = \begin{cases} \hat{s}(k) - A & ; \ \hat{s}(k) > +A \\ 0 & ; \ |\hat{s}(k)| \leq +A \\ \hat{s}(k) + A & ; \ \hat{s}(k) < -A. \end{cases} \tag{5.24}$$

A large value of $A$ leads to a large residual echo suppression as well as to distortion of the speech signal and vice versa. This value should therefore not be chosen too large.

Newer techniques use (instead of the center-clipper) a so-called postfilter in the transmitting branch, the coefficients of which are based on signal statistical modeling of the residual echo signal. Just like the compensation filter, the postfilter can be efficiently adapted and realized by means of block-based *frequency domain techniques*. The post filter can be used not only for the reduction of the residual echo but simultaneously for the reduction of disturbing background noise. Refer to the literature for a detailed description of these methods.[4]

[4] e. g. Stefan Gustafsson *Enhancement of Audio Signals by Combined Acoustic Echo Cancellation and Noise Reduction*, dissertation, IND, RWTH Aachen, 1999.

## 5.6 Exercises

In the following experiments, you should get to known the properties of the presented NLMS algorithm and learn how it operates. The M-files `sim_echokomp.m` and `nlms4echokomp.m` serve as a template for all of the following tasks.

The MATLAB script `sim_echokomp.m` is the main script that you will run for each of the exercises. In the beginning of this file, you can find the variable `versuch` that allows you to set for which of the subtasks the code should be run. The MATLAB function `nlms4echokomp` is the core of this experiment. In it, a framework is given within which you will implement the NLMS algorithm in the first exercise.

In the `echokomp.mat` file, there are 2 variables. The variable `x` is the speech signal $x(k)$ of the far speaker that has been sampled with 8 kHz. The variable `g` is a cell array and contains three artifically generated room impulse responses with a length of 200, 1000 and 2000 samples respectively.

**Exercise 5.1 Construction of the echo canceller**

The echo cancellation system according to Figure 5.3 will be simulated and analyzed. The filter $\mathbf{h}(k)$ will be adapted by means of the NLMS algorithms as described in Section 5.4. The (fixed) adaptation stepsize for the NLMS algorithms is chosen as $\alpha = 0.1$ and there is no background noise present, i. e. $n(k) = 0$. The excitation signal $x(k)$ is white noise with the variance $\sigma^2 = 0.16$.[5] For $\mathbf{g}$, the room impulse response with 200 values is used. *This configuration should also be used for the subsequent experiments unless stated otherwise.*

Complete the `nlms4echokomp.m` file and add code to plot the echo signal $\tilde{x}(k)$ and the residual echo $e(k)$ in `sim_echokomp.m`. The two signals should be plotted in a single diagram. Plot the relative system distance $D(k)$ and the ERLE measure - both in dB - over the time. Ensure that the axes are labeled correctly!

**Note:** For better comparability, draw all three plots of this experiment in a single figure using the `subplot` command.

**Note:** Compute the ERLE measure as a function of the discrete time according to equation (5.7) by taking the average over the previous 200 samples for the expectancy value in each time step $k$.

**Exercise 5.2 Influence of the signal properties on the adaptation**

The properties of the signal $x(k)$ of the far speaker have a major influence on the convergence behavior of the NLMS algorithms! To take a closer looks at this, the following three excitation signals $x(k)$ will be considered:

[5] The computation of the variance was covered in experiment 3.

a) speech $s(k)$ (given by the variable `s`)

b) white, Gaussian distributed noise $r_w(k)$ with the variance $\sigma^2 = 0.16$

c) colored noise $r_c(k)$.

The colored noise $r_c(k)$ is acquired from the white noise $r_w(k)$ by filtering with a filter that is described by the transfer function

$$H(z) = \frac{1}{1 - 0.5 \cdot z^{-1}}. \tag{5.25}$$

In this and the following experiments, only the relative system distance (in dB) will be considered. Plot it for all three cases in a single diagram. Add axes labeling and a legend to the plot to make it as self-explanatory as possible.

What signal property is important for a good convergence behavior?

### Exercise 5.3 Adaptation with background interference and noise as excitation

The transmitted excitation signal $\tilde{x}(k)$ now suffers from a background noise $n(k)$. The latter is modeled by additive white noise with a variance of $\sigma^2 = 0$ (no interference), $\sigma^2 = 0.001$ and $\sigma^2 = 0.01$ respectively. Draw the relative system distances for these three cases (in one diagram).

### Exercise 5.4 Adaptation with background interference and speech as excitation

Repeat the previous experiment with speech as excitation signal. What are the prominent differences compared to the previous experiment?

### Exercise 5.5 Influence of the stepsize

The background interference $n(k)$ is now white noise with the variance $\sigma^2 = 0.01$. (The excitation signal is again white noise with the variance $\sigma^2 = 0.16$.) How do the different stepsizes $\alpha \in \{0.1, 0.5, 1.0\}$ affect the relative system distance under these circumstances?

### Exercise 5.6 Influence of the compensation filter length

The length $m$ of the transversal filter is usually smaller than the length $m'$ of the room impulse response in practice, i. e. $m < m'$. Compute the relative system distance for the cases $m = m' - 10$, $m = m' - 30$, and $m = m' - 60$. What do you notice?

**Note:** To compute the system distance of filters with different lengths, the shorter filter is usually zero padded.

### Exercise 5.7 Room impulse responses of different lengths

The three different (fixed) room impulse responses for this experiment are given by the cell array `g`. The length of the compensation filter is assumed to be always equal to the length of the room impulse response, i. e. $m = m'$. (The excitation signal is white noise and no background interference is present.) Plot the relative system distance for these three cases! What do you observe?

# Chapter 6

# Efficient Programming in MATLAB

Whoever uses MATLAB will quickly realize that some self-written functions are quite fast while others are very, very slow, some require a lot of memory while others require only a little even though the amount of data that used for calculations is comparable. In this chapter, you will learn how to find the parts of your MATLAB program that cause it to be slow, and how you can optimize your code with respect to the resource usage (in terms of both memory and computation time).

## 6.1 Identifying Slow Code Sections

The first imminent challenge when you want to speed up your MATLAB program, consists of identifying the slow parts of your program first. This will allow you to concentrate on optimizing the actual bottlenecks of your application afterwards. MATLAB comes with two tools that support you with tracking down slow code: a Profiler and a simple stopwatch.

### 6.1.1 MATLAB Profiler

With the help of the M-file Profiler, you can conveniently analyze your self-written functions. You get a good overview over what program sections MATLAB spends the most time on. For this purpose, MATLAB determines the processor time it takes for every code line during the execution of your M-file. The results are then listed in a well-arranged table.

From this table, you can easily deduce at what positions your program does in fact require a lot of processor time and where it is worth making optimizations to your code. The output of the Profiler can also be used for debugging the code. The Profiler lists the code of the analyzed M-file and assigns a number to every code line that indicates how often this line was run. You can therefore easily detect what code line was for example not run at all which might suggest that there is an error.

There are different ways to start the Profiler:

- Select **Home➔Run and Time** from the menu line of the MATLAB desktop
- Select **Editor➔Run and Time** from the menu line of the MATLAB editor
- Enter the following line in the MATLAB Command Window

```
>> profile viewer
```

In the started Profiler window, you can then enter a code line that should be analyzed in the designated field. This can for example be a function call with a lot of intput parameters. This is then run in the current Workspace, i. e. all currently existing variables are available. An example for a Profiler output is shown in Figure 6.1.

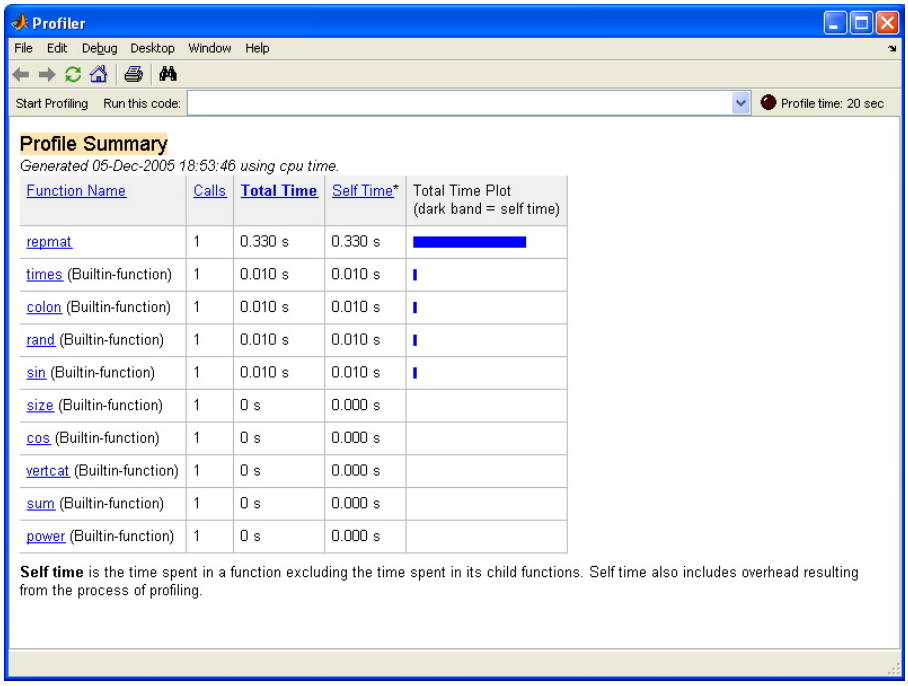



**Figure 6.1:** The Profiler window after the analysis of the program

As is usually the case in MATLAB, you can also use the Profiler directly from the command line. To activate the Profiler mode, enter the following:

```
>> profile on
```

All code that is entered and started thereupon, will be analyzed. When starting the Profiler mode like this, all the Profiler data from previous sessions will be overwritten. The Profiler mode is exited once either

```
>> profile viewer
```

or

```
>> profile off
```

is entered. The former of these commands (`profile viewer`) ends the Profiler Mode and displays the analysis data in the usual Profiler window. The latter merely terminates the Profiler mode so that following commands are no longer analyzed. It is now possible to enter

```
>> profile resume
```

to reactivate the Profiler and resume the current session without losing the previously collected data.

If you wish to use the Profiler to analyze one or more code lines that appear in the history, there is a convenient way to do so. Just mark the desired lines and call the Profiler from the corresponding context menu that appears upon right-clicking the marked command lines.

Please refer to the provided documentation for further features that the MATLAB Profiler offers.

### 6.1.2 MATLAB's Timer

If only a rough estimation of the execution time of specific blocks is of interest, you can use the functions `tic` and `toc`. A timer is started with `tic`. As soon as `toc` is called, MATLAB stops the timer and returns the elapsed time. The main difference to the Profiler is that this only measures real-time. Switching to processor time, as possible with the Profiler, is not intended here. If the computer is busy with other, computationally intensive tasks during the execution of your MATLAB program, the data provided by this method is usually not very reliable.

Here a little example:

```
x = 0;
tic
for k = 1:1000
  x = x + k;
end;
toc
```

Let's assume that the code given above is stored in the file `mytimer.m`. Then the output of the program might look as follows:

```
>> mytimer
elapsed_time =

   9.7000e-05
```

It is also possible to use several timers at once. If for example you wish to compute the total runtime and the runtime of a code section, you can store the return value of `tic` in different variables and pass it to `toc` at the end of the section. Here a little example:

```
tall=tic;
x=0;
for k=1:10
tpart=tic;
    x=x+k;
    toc(tpart);
end
disp('Total duration:');
toc(tall);
```

Let's assume that the code given above is stored in the file `mytimer2.m`. Then the output of the program might look as follows:

```
>> mytimer2
Elapsed time is 0.000165 seconds.
Elapsed time is 0.000007 seconds.
```

```
Elapsed time is 0.000005 seconds.
Elapsed time is 0.000004 seconds.
Elapsed time is 0.000004 seconds.
Elapsed time is 0.000004 seconds.
Elapsed time is 0.000004 seconds.
Elapsed time is 0.000004 seconds.
Elapsed time is 0.000004 seconds.
Elapsed time is 0.000004 seconds.
Total duration:
Elapsed time is 0.002664 seconds.
```

## 6.2 Code Optimization

Once you have found the critical code section with the help of the Profiler, you can start optimizing them. In the following sections, we will introduce several techniques that can help you with accomplishing this.

### 6.2.1 Saving MATLAB Time

#### Vectorization of Loops

MATLAB is a programming language that is optimized for matrix operations. Vector and matrix operations can therefore be evaluated very quickly and efficiently. This suggests that by vectorizing your algorithm, i. e. by transforming `for` and `while` loops into vector and matrix operations, you may be able to achieve a significant increase in calculation speed.

As an example, consider having a signal `s` that you wish to multiply with a sine and a cosine, square both components and then take the root of the two components. When being used to C/C++, one might perhaps proceed like this:

```
t = 0:0.01:100;
s = rand(size(t));   % row vector

for k = 1:length(t)
  A(1,k) = sin(t(k));
  A(2,k) = cos(t(k));
end

% multiplication
for k = 1:size(A,1)
  for l = 1:size(A,2)
    r(k,l) = A(k,l) * s(l);
  end
end

% square, add and calculate root
for l = 1:size(r,2)
    result(l) = sqrt(r(1,l)^2 + r(2,l)^2);
end
```

This implementation is mainly based on `for` loops, i. e. MATLAB's matrix computation capabilities are not exploited. Because MATLAB is a script language, the

commands within the for loops must be interpreted anew in each iteration, which takes up a lot of time. You can avoid this by vectorizing the implementation above, i. e. by converting it to vector and matrix operations.

```
% Initialization of the variables t and s as before

A = [sin(t); cos(t)];
S = repmat(s, 2, 1);
result = sqrt(sum((A .* S).^2));
```

Here, the vector `s` is replicated with the command `repmat` (`S` consists of two lines, both of which are equal to `s`). Afterwards, `A` is multiplied with `S` element by element. and the result is squared, again element-wise. The command `sum` is used to then get the sum of all of the elements in the same column. The increase in calculation speed immediately becomes apparent in this case, particularly for longer signals.

Beside `repmat`, there still is a large variety of other functions that are helpful for the vectorization of your algorithms. Commonly used functions can be found in table 6.1. You can look into the MATLAB help to get more details about the functionality of the listed functions.

**Table 6.1:** Functions that are commonly helpful for the vectorization of algorithms

| Function | Brief description |
|---|---|
| `all` | Test whether all elements are unequal to zero |
| `any` | Test whether any element is unequal to zero |
| `cumsum` | Column-wise computation of the cumulative sum |
| `find` | Determines indices of the elements that are unequal to zero[1] |
| `ind2sub` | Converts from linear index to component-wise index |
| `ipermute` | Inverse permutation of the dimensions of a multi-dimensional matrix (see `permute` further below) |
| `logical` | Converts numerical values into logicals |
| `permute` | Permutes the dimensions of a multi-dimensional matrix according to a specified scheme |
| `prod` | Column-wise multiplication of all of the elements of a matrix |
| `repmat` | Replicates and “tiles” a matrix |
| `reshape` | Reshapes a matrix |
| `shiftdim` | Shifts the dimensions of a matrix |
| `sort` | Sorts the elements of a matrix in ascending or descending order |
| `squeeze` | Removes all dimensions that contain only one element |
| `sub2ind` | Converts from component-wise index to linear index |
| `sum` | Column-wise computation of the sum of a matrix |

[1] Most usecases of `find` can be replaced by so called logical indexing. For instance, the query `A(A>B)` safes computation time, compared to `A(find(A>B))`.

### Critical Use of Data Types

In general, there is no need for you to worry about the type of a variable in MATLAB. You as a developer can easily specify whether a variable stores real or imaginary numbers, strings or cell arrays by the corresponding allocation. Moreover, you can assign a string to a variable several lines after having assigned a number to the very same variable. MATLAB will take care of the correct type of variable.

However, this "taking care" always requires some time. You should therefore avoid code like the following for applications in which a good performance is required.

```
a = 45;
... % some code
a = 'Awesome string';
... % more code
```

It is not just the change between numbers and strings that is critical here. If you have stored a real number in a variable, you should also avoid assigning a complex number to this variable and vice versa.

If you know exactly that you will only have to deal with real numbers, you can save MATLAB a lot of computational effort by using special functions. Among these are the funtions in the following table.

| Function | Brief description |
|---|---|
| `reallog` | Computes the natural logarithm for non-negative real numbers |
| `realpow` | Computes the power with a purely real output |
| `realsqrt` | Computes the square rot for non-negative real numbers |

### Logical Operators

As you have already learned in chapter 1, there are two different types of logical operators, namely

| Function | Brief description |
|---|---|
| `&, \|` | Element-wise application of the logical operators AND and OR on vectors and matrices |
| `&&, \|\|` | Element-wise application of the logical operators AND and OR on scalars with so-called "short-circuiting" |

In `if` and `while` expressions, it is more efficient to use the operators with "short-circuiting" as this will abort the computation as soon as the result of the entire expression is known. For example, MATLAB only evaluates the first part of the expression in the following example if the number of input parameters is less than 3:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

### Logical Indexing

In logical indexing, the elements of an array/matrix is designated by a indexing array/matrix based on their position, not their value. Every `true` element in the indexing array/matrix serves as a position index. In the following example,

```
>> A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> B = logical([1,1,0,0;0,1,0,0;0,0,0,1;0,0,1,0])
B =
  4 by 4 logical array
   1   1   0   0
   0   1   0   0
   0   0   0   1
   0   0   1   0

>> A(B)
ans =
    16
     2
    11
    15
    12
```

`B` is the indexing matrix with the same size as `A`. `A(B)` gives an array containing all the elements in `A` whose positions correspond to all the 1s in `B`.

Logical indexing is extremely useful when you want to, for example, find the matrix elements that satisfy certain condition. Suppose you have a random matrix, and you want to set all the values that are less than 0 to 0. Instead of using loops, this operation can be very easily done using logical indexing.

```
>> C = randn(4)
C =
   -0.2028    1.3085    0.5252    0.4825
   -0.6894   -0.0103   -0.9623    0.0705
   -0.4274   -0.3376   -0.3584   -1.4408
    0.4030    0.8248   -0.5896   -0.8714

>> IND = C<0
IND =
  4 by 4 logical array
   1   0   0   0
   1   1   1   0
   1   1   1   1
   0   0   1   1

>> C(IND) = 0
C =
         0    1.3085    0.5252    0.4825
         0         0         0    0.0705
         0         0         0         0
    0.4030    0.8248         0         0
```

or in a more concise manner

```
>> C = randn(4)

C =

   -0.2028    1.3085    0.5252    0.4825
   -0.6894   -0.0103   -0.9623    0.0705
   -0.4274   -0.3376   -0.3584   -1.4408
    0.4030    0.8248   -0.5896   -0.8714

>> C(C<0)=0

C =

         0    1.3085    0.5252    0.4825
         0         0         0    0.0705
         0         0         0         0
    0.4030    0.8248         0         0
```

`find` is a frequently used function for logical indexing. It returns the indices of all non-zero elements of the input array. For example,

```
>> X = [1 0 2; 0 1 1; 0 0 4]
X =
     1     0     2
     0     1     1
     0     0     4

>> index1 = find(X)  % find linear indices of non-zero elements in X
index1 =
     1
     5
     7
     8
     9

>> index2 = find(~X) % find linear indices of zero elements in X
index2 =
     2
     3
     4
     6
```

However, `find` should only be used if the indices themselves are needed. Otherwise use the approaches described previously in this paragraph, because they save execution time.

### 6.2.2 Accelerating Element-wise Operations

Although MATLAB is able to calculate matrix operations very efficiently, there is still something that user can do to make it even faster. `bsxfun` is a built-in Matlab function that implements element-wise operations. Compared with the normal element-wise operations, `bsxfun` is more efficient. You are recommended to use `bsxfun` when the running time is critical. Please call `doc` and read the detailed documentation in case that you need more help. A simple example is given below.

Suppose you need to calculate a function $f(x,y) = cos(x) + e^y$ for all values of $x$ and $y$ each from 0 to 1 with a step size of 0.0001. The result should be a 10000 by 10000 matrix. There are two possible solutions for this problem.

```
fun = @(x,y)cos(x)+exp(y) %create a function handle

x = 0.0001:0.0001:1;
y = 0.0001:0.0001:1;

[X,Y] = meshgrid(x,y);  % generate 2-D grid suitable for 2-D calculating.
                        % refer to Matlab help if you need more details.
tic;
result1 = bsxfun(fun,x,y'); %Element-wise calculation using bsxfun
toc;

tic;
result2 = fun(X,Y);  %Normal element-wise operation
toc;
```

The result after executing the `example.m` file above is:

```
>>example

Elapsed time is 1.388926 seconds.
Elapsed time is 5.573585 seconds.

>> isequal(result1,result2)
ans =
  logical
   1
```

As you can see, the execution time is considerably reduced by using `bsxfun`.

### Overloading Functions

It is possible to overload functions in MATLAB, i. e. to implement an own function under the name of a built-in function. You might for example want to implement an addition function `plus` that treats the different integer data types differently and that contains a runtime-optimized code for the respective data type.

Even beside the fact that you need to be careful when using the names of built-in functions and constants, this also means that you bypass optimizations that are used in the built-in functions of MATLAB.

## 6.2.3 Efficient Memory Usage

When using MATLAB for larger simulations, e. g. of transmission lines, the memory capacity may be exhausted even for well-equipped computers as the data amount can grow immensely. Under these circumstances, it is vital to properly take care of the memory. This however is only possible if you know how MATLAB uses the memory so that you can systematically apply memory-saving programming techniques.

### Reserving Memory for Vectors in Advance

As you have learned in the preceding chapters, there is not need to explicitly reserve memory for variables, even if they grow during the execution. This is particularly useful in `while` loops if it is not known exactly how large a matrix or a vector will become during the execution.

Still, in long `for` and `while` loops, this can lead to significant performance issues. This is because MATLAB will reserve a certain amount of memory for a variable first. If the variable becomes larger than the reserved memory during runtime, new memory will have to be reserved and all of the contents have to be copied to the new position. If this happens very often, there are two consequences. On the one hand, memory blocks of continuously growing size need to be reserved and also copied, i. e. the longer the loop runs, the slower the program gets. On the other hand, a significant memory fragmentation occurs, i. e. a large number of more or less small, separated memory blocks are generated. This makes it increasingly difficult to reserve a large coherent block of memory, which significantly complicates the memory management and in turn leads to a loss of performance.

In the following code, the variable `x` is first set to zero and the memory for a single element is reserved. In the loop, one more element is added in every iteration.

```
x = 0;
tic;
for k = 2:100000
   x(k) = x(k-1) + 10;
end
toc
```

Output:

```
Elapsed time is 0.014223 seconds.
```

To avoid problems that arise due to the dynamic enlargement of the vector, you can explicitly reserve a sufficient amount of memory before entering the loop. To do so, you can for example use the function `zeros`. This creates a zero matrix of the desired size and all of the required memory is reserved. In the code example above, only the first line must be modified.

```
x = zeros(1,100000);
tic;
for k = 2:100000
   x(k) = x(k-1) + 10;
end
toc
```

Output:

```
Elapsed time is 0.000799 seconds.
```

### Avoiding Creating Temporary Arrays

Avoiding creating unnecessary temporary variables is another handy strategy to save memory. This can be achieved by cascading multiple functions or by specifying optional input arguments of some functions.

```
a = randn(1000); %Create a 1000 by 1000 gaussian distributed random matrix
                 %with standard deviation 1 and mean value 0.
b = sum(a, 1);   %Columnwise summation
c = std(b);      %Standard deviation
```

In this example, the variables `a` and `b` are not necessary, so we could alternatively write

```
c = std(sum(randn(1000),1));
```

by cascading three functions into one single line.

Many functions also provide optional input arguments, which makes programming much easier and resource saving. For example, the following code

```
a = zeros(100);
b = single(a);
```

generates an all-zero matrix `b` of data type `single`. This could also be done by

```
b = zeros(100,'single');
```

,where the unnecessary temporary variable `a` is left out.

### Lazy Copy

When copying a variable in MATLAB, i. e. when assigning the contents of a variable to another, MATLAB does not immediately copy the data, but it creates a second variable so that it points to the same data as the original variable. As long as you access both variables read-only, MATLAB will not change anything about this. It is not until you change the data of one of the two variables that MATLAB copies the data so that the unchanged variable will still have the original data available.

It should be noted that in such as case, MATLAB always copies the entire variable, i. e. the entire matrix in case of a matrix, independently from the number of elements that were actually changed. When sequentially processing a larger matrix or another data structure, it may be helpful to only copy individual sections to another variable so that merely a limited amount of additional memory is required.

Caution is also advised when passing large variables to functions. As soon as just one field of this data structure is changed within the function, the entire variable needs to be copied as changes may not affect the external variable.

### Sparse Matrices

If the majority of the elements of a matrix is zero, these can be managed particularly efficiently in MATLAB by storing only those elements in the memory that are unequal to zero. To make use of this efficient storing in MATLAB, call the function `sparse` with the respective variable. This makes MATLAB attempt to store the matrix as a sparse matrix.

Afterwards, such a sparse matrix can be used just like any other matrix. It is possible to perform operations both among sparse matrices and among a mix of normal and

sparse matrices. It depends on the used operation whether the result is a normal matrix or again a sparse matrix. If for example an element-wise operation like `.*` is applied, the zero-elements remain zero so that the resulting matrix is guaranteed to be a sparse matrix.

### Header Information of Vectors, Structures and Cells

Beside the data, MATLAB stores additional information like the data type and the dimension in the corresponding data structure for every matrix and every vector. Usually, the contribution of this additional information is negligible compared to the size of the useful data. It is therefore more favorable with respect to the memory usage to store large amounts of data in a small number of matrices rather than using a large number of small matrices. Still, you should have always find the right balance between keeping the memory usage to a minimum and maintaining a good readability of the code, the latter of which might be reduced when attempting to store different kinds of information in the same matrix to save a bit of space.

If you use structures of cell arrays, MATLAB does not only generate such a header for the entire data structure, but additionally for each individual field. It therefore has a major influence on the required memory space, how you construct such data structures.

Consider saving the RGB values of an image with a resolution of 640x480 dots in a data structure with the fields R, G and B for the different colors. You now have two possibilities. One option would be to create a structure array with the three fields R, G and B, all of which have a size of 640x480.

```
% Structure with three fields with matrices
image1.R(1:640, 1:480)
image1.G(1:640, 1:480)
image1.B(1:640, 1:480)
```

For this data structure, a header is required for the structure array `image1` and one header for each of the three matrices `R`, `G` and `B`. This results in a total of four headers for the entire data structure. The second option would be to create a matrix of size 640x480 where each element has the scalar fields R, G and B.

```
% Matrix with three fields per element
image2(1:640, 1:480).R
image2(1:640, 1:480).G
image2(1:640, 1:480).B
```

In this case, a header is required for the matrix `image2` and additionally for each of the three fields of every single element as well. For 307200 elements, this yields 921601 in total, which leads to a significant increase in memory usage.

Note that the function `whos` shows only the memory usage of the actual data. The header information is not included.

### MATLAB Recycled Storage

Under Unix operating systems, MATLAB takes care of the further memory management itself once it has reserved some memory because it cannot return the storage

to the operating system just like that. So if you delete a variable in the Workspace, MATLAB will continue to try placing new variables at this exact, released memory range. However, this is possible only if the variable completely fits into the free space.

For this reason, it makes sense to reserve large variable early in the program. When releasing these variables later on, there is a bigger chance that smaller variables completely fit into the free memory blocks. In the following example, the first two commands require about 15.25 MB of memory on their own, the final line 16.02 MB. Although the variables `a` und `b` are released, a total of about 31.28 MB is required as `c` does not completely fit into the space of `a` and `b`.

```
a = rand(1e6,1);
b = rand(1e6,1);
clear
c = rand(2.1e6,1);
```

When inverting this example, the space that is first reserved and then released is large enough to take in both `a` and `b` after `c` has been cleared. This reduced the total amount of used space to about 16.02 MB.

```
c = rand(2.1e6,1);
clear
a = rand(1e6,1);
b = rand(1e6,1);
```

When using MATLAB under Windows, the aforementioned phenomena are not as critical as the memory management is handled differently here.

### Tools for Memory Management

MATLAB provides you as a user with several commands that allow you to influence the memory management of MATLAB. The most important commands are listed in the following.

| Function | Brief description |
|---|---|
| `whos` | Display all of the variables that exist in the Workspace and how much space they use |
| `pack` | Save all variables on the hard drive, delete them in the Workspace and load them again. This helps in resolving e. g. issues with the memory fragmentation. |
| `clear` | Delete variables and functions from the memory (also separately) |
| `save` | Save variables on the hard drive. If you expect vast amounts of data, you can use this to externally store data that is currently not in use and release the space in the process. |
| `load` | Restore data that has been saved with save before. |

| Function | Brief description |
|---|---|
| `quit` | Ends MATLAB and releases all allocated memory. UNIX does not release memory that has been allocated by an application until the application is closed. So if MATLAB is not used at the moment and you wonder why other programs are very slow although MATLAB has nothing to calculate at that time, it may very well be helpful to close and restart MATLAB. |

The function `whos` is very frequently used to monitor variables, including its size, memory space, class and attributes. If you run `whos` in command window with no input, all the variables in workspace will be listed in alphabetical order. If you call `whos variablename`, only the designated variable will be listed.

```
>> whos
  Name         Size              Bytes  Class     Attributes

  a            5x1                 920  cell
  b          100x100             80000  double

>> whos a
  Name       Size             Bytes  Class    Attributes

  a          5x1                 920  cell
```

### 6.2.4 Moving loops to MEX files

If a `for`- or `while` loop cannot be vectorized, you still have the option to move the code to a so-called MEX file. MEX files are the MATLAB interface to external Fortran and C/C++ code. As this allows MATLAB to access precompiled code, there is no need to reinterpret instructions within a loop every time that they are executed. This makes some significant runtime improvement possible. You will learn about the possibilities of this interface and how to use it in the following experiment.

## 6.3 Exercises

### Exercise 6.1

Go over the examples in this chapter and analyze them with tools that are suitable in the respective case.

### Exercise 6.2

1. Analyze the M-file `decimal2binary.m` that you can find in the experiment folder and identify the slow code sections. Test it several times with small and with large amounts of data.
2. Save the M-file under a different name, e. g. `decimal2binaryfast.m`. Now optimize the M-file step by step. Check after each change with the Profiler to what extent this accelerates the function.
3. Which code sections are particularly critical? How is this affected by the amount of data?

### Exercise 6.3 Data transmission over an erroneous channel

1. Write a function `channelsim` that expects a bit vector as input value, adds statistically distributed errors and returns the result again as a bit vector. In this context, a bit error means that the bit is inverted, so that it has a value of 1 instead of 0 or vice versa.

   To do so, first map the binary values 0 and 1 to 1 and -1 respectively (bipolar transmission; Binary Phase Shift Keying, BPSK). Then add normally distributed white noise with a mean value of $m = 0$ and a variance of $\sigma^2 = 1$ (function `randn`) to this signal. For the conversion back to binary values, implement a decision maker with the decision threshold $s = 0$ and map all values that are larger or equal to $s$ to 0 and all smaller values to 1.

2. Extend the function `channelsim` so that you can set the ratio $\frac{E_b}{N_0}$ between the energy per transmitted information bit ($E_b$) and the noise power density on the channel ($N_0$) via a parameter in dB.

   **Note:** It holds for the relation between the variance and the noise power density that

   $$\sigma^2 = \frac{N_0}{2}. \tag{6.1}$$

   The energy per bit is

   $$E_b = 1 \tag{6.2}$$

   for bipolar transmission with amplitude 1. A dB value can be calculated according to

   $$10 \cdot \log_{10}\left(\frac{E_b}{N_0}\right) = 10 \cdot \log_{10}\left(\frac{1}{2\sigma^2}\right) \tag{6.3}$$

   here.

3. Now load the file `speech.wav`. The data in the WAVE-file has a precision of 16 bit per sample and the value range is limited to $[-1, +1]$. Write a function that converts the samples to their binary 16-bit representation. Add $+1$ to the values to receive positive samples in the range of $[0, +2]$, then multiply them with $2^{15}$ and round – just in case, as the rounding should actually not be necessary – to integer values. These can then be converted into their binary representation. The output value of the function is then supposed to be a $(16 \times M)$ matrix where $M$ is the number of samples of the audio signal.

4. Perform a transmission without encoding for different $\frac{E_b}{N_0}$ and listen to the respective results. From around which $\frac{E_b}{N_0}$ are you no longer able to hear the errors? Identify the bit error rates for $\frac{E_b}{N_0}$ between 0 and 24 dB in steps of 1 dB and plot them over $\frac{E_b}{N_0}$ in a diagram with the bit error rate on a logarithmic scale. Determine the lowest $\frac{E_b}{N_0}$ for which you do not hear any errors. What is the bit error rate of the system for this $\frac{E_b}{N_0}$?

5. Encode the data of the speech signal with the (31,16) block code to the generator polynomial

$$G_3(z) = 1 + z + z^2 + z^3 + z^5 + z^7 + z^8 + z^9 + z^{10} + z^{11} + z^{15}. \tag{6.4}$$

   **Note:** For the generation of the generator matrix, the parity-check matrix and the syndrome table, use the corresponding functions from the Communcations Toolbox (`cyclgen`, `syndtable`). Complete functions are also available for the encoding and decoding (`encode`, `decode`). Computing the syndrome table can be pretty time-consuming so that it may be worthwhile saving it to a file and then only reading it when needed.

   **Note:** Before simulating the transmission, you have to keep in mind that $E_b$ denotes the energy per information bit. For a real-time transmission with channel coding however, during the time period in which you effectively get 16 information bits across the channel, you must actually transmit a total of 31 bits in the case considered here. The energy per bit on the channel is thus reduced by a factor of $\frac{k}{n}$. You can account for this in the simulation of the channel by mapping the bits not to $\pm 1$ but to $\pm\sqrt{\frac{k}{n}}$. To keep the function `channelsim` flexible, introduce another, optional parameter for the so-called code rate $\frac{k}{n}$.

   Now simulate the transmission channel with the modified `channelsim` and then decode the data again. Listen to the results for different $\frac{E_b}{N_0}$ between 0 dB and 24 dB. From about what $\frac{E_b}{N_0}$ are you no longer able to hear the errors? Compare this value with that of the previous subtask. Identify the rest bit error rate after decoding for $\frac{E_b}{N_0}$ between 0 dB and 24 dB in 1 dB steps and plot this over $\frac{E_b}{N_0}$ in the same diagram as the curve from the previous subtask.

6. Add a title and axes labeling to the diagram with the help of the corresponding functions. Set the value range so that the curves cover the entire display range. Set the tick marks of the x-axis in 3 dB intervals and turn on the rough grid lines, for the x-axis additionally the minor grid lines.

7. What do you notice when comparing the curves? Is it always beneficial to use channel coding? If not, mark the position in the diagram, for example with an arrow, from which on it is no longer advantageous to use channel coding.

# Chapter 7

# MATLAB C API

Despite MATLAB being a very powerful tool and providing plentiful special functions via toolboxes, it is not uncommon to encounter the issue that a (self-written) function does not reach the required execution speed. A possibility to accelerate individual functionalities is to realize them in a different programming language, that does not interpret, but compile in machine code, rather than MATLAB.

In some cases, special functions and algorithms may already be available in a different programming language. Particularly for complex and highly optimized algorithms, porting these to MATLAB may require an immense effort. Moreover, significant losses in execution speed would be inevitable because MATLAB is an interpreting language.

For both cases, MATLAB offers a solution for integrating external program code with the programming interface MEX. The interface supports the programming languages C/C++ and Fortran to make it possible to call functions that are implemented in these programming languages directly from MATLAB just like any other MATLAB function.

MATLAB also provides an interface to call MATLAB functions from C/C++ programs. The module that makes this possible is called the MATLAB Engine.

In this laboratory, we will only cover the C/C++ interface. The concepts for Fortran are almost identical so that you can easily transfer the mechanisms presented here to Fortran programs provided that you are familiar with Fortran.

## 7.1 Calling C functions from MATLAB

We will first deal with how to call C/C++ code from MATLAB. The basic principle of MEX functions is that the functions that are implemented in C/C++ are dynamically embedded into MATLAB as a library during runtime. This is achieved by means of a Dynamic Link Library (.dll) under Microsoft Windows or with the help of a Shared Object (.so) under Unix/Linux.

An entry point with a fixed name (`mexFunction`) must exist so that MATLAB knows what to do with such a DLL. This is the so-called Gateway Routine. As this name is the same for every DLL that conforms with MEX conventions, MATLAB can determine the initial address from the name and call the function. Such DLLs as well as the corresponding source code file will also be referred to as MEX-files in the following.

The functionality in MEX-files can be implemented in C but also in C++. A necessary requirement for this is that the compiler used for the creation can compile C++. Also, the entry point of a MEX-file must always be a pure C function. To ensure this, the function must be enclosed as follows:

```
#ifdef __cplusplus
extern "C"
{
#endif

...

#ifdef __cplusplus
}
#endif
```

Within the function that is enclosed like this, it is perfectly possible to call C++ functions.

### 7.1.1 Basic Structure

A MEX-file that has been compiled with the `mex`-script – which we will deal with at a later point – can be called just like a normal M-file or built-in MATLAB function from within MATLAB. Just as for MATLAB-functions, arbitrary MATLAB data structures can also be passed to a MEX-file and it can return any kind of data structure that is supported by MATLAB as a result.

To make this work, the MEX-file must comprise the following elements:

- a Gateway Routine that serves as an interface between MATLAB and your calculation routine as well as
- one or more functions that perform the special calculations that you want MATLAB to have access to

You are free to choose what to do in the calculation part. However, the gateway routine must be implemented exactly as the following prototype so that MATLAB can find the entry point:

```
void mexFunction (int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]);
```

The parameters have the following meaning in this:

**nrhs** The number of passed arguments and thus the number of elements of the vector that `prhs` points to. The name is derived from “right-hand-side” since the arguments are listed to the right of the function name.

**prhs** A vector of pointers to mxArray data structures in which the data of the passed arguments is stored. For example, the first argument can be accessed with the notation `prhs[0]` that is typical for C

**nlhs** The number of result variables and thus the number of elements that `plhs` points to. The name is derived from “left-hand-side” since the result variables are listed to the left of the function name.

**plhs** A vector of pointers to mxArray data structures. The pointers of this vector are first set to NULL. If you wish to provide an output, you have to generate mxArray objects and store pointers to these in this vector.

Figure 7.1 illustrates the parameter passing relations upon calling the MEX gateway routine. Here, the variables `a` and `b` are passed while the variables `c` and `d` are created in the MEX-file and passed to MATLAB as output value.

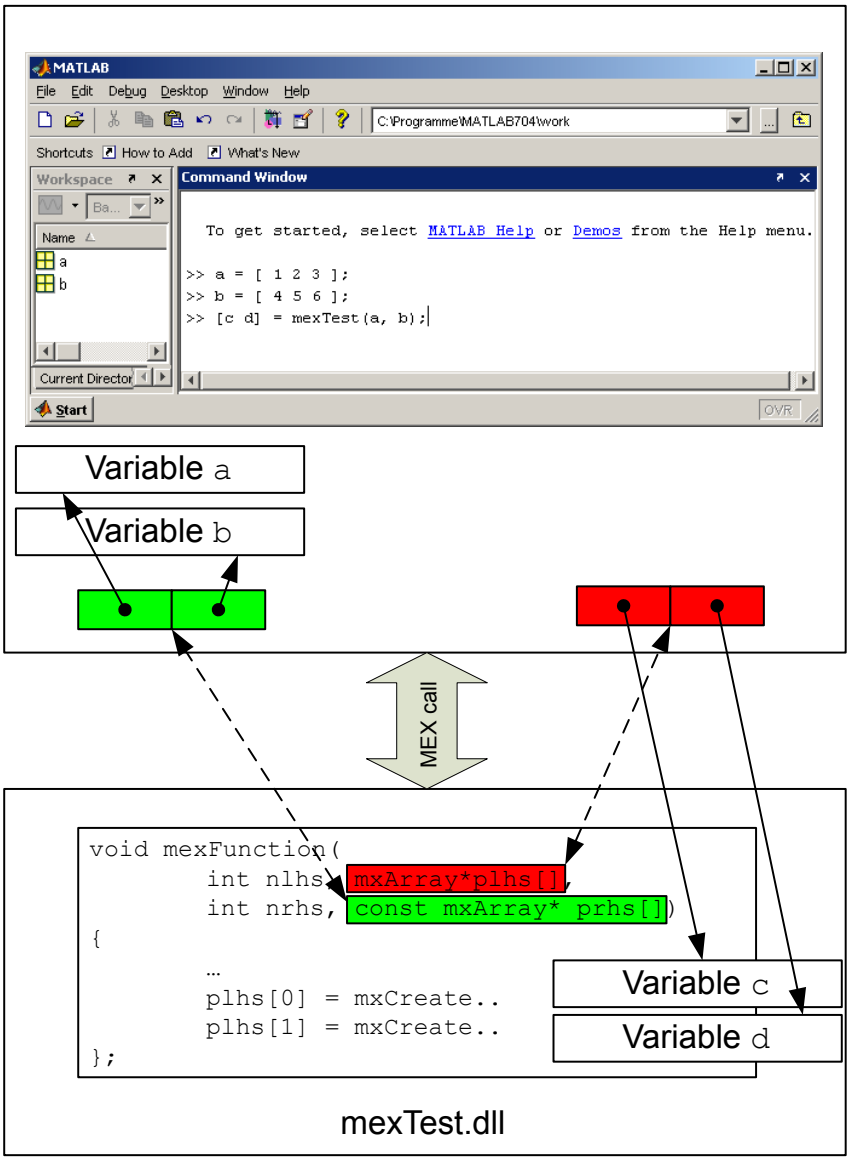



**Figure 7.1:** Call of a MEX-file.

Objects that are created as output variables are passed to MATLAB when exiting the MEX-file so that they are then available in the Workspace. Therefore, MATLAB has the responsibility for releasing the created storage. You should not release the return variables' memory by yourself for this reason as this would otherwise cause a memory access error.

To access the data in mxArray objects or to create new mxArray objects, you need the functions from the MATLAB C API. You can integrate them into your source code files with the line

```
#include "mex.h"
```

The MATLAB C API consists of two parts that are discriminated by the prefix in the function name.

**mx functions** Functions starting with the prefix `mx` make it possible to manipulate MATLAB arrays (mxArray).

**mex functions** Functions starting with the prefix `mex` perform operations in the MATLAB environment. This allows you to access all MATLAB functions and the Workspace from within your MEX-files. For example, you can evaluate a string in the MATLAB Workspace with the function `mexEvalString`. Anything that is permitted on the MATLAB console is also permitted here.

All of these functions are completely documented in the MATLAB help for the MATLAB C API. While you can use mx functions in any software project, mex functions may exclusively be used in MEX-files.

### 7.1.2 Data Types

Before you start programming MEX-files, it is important to understand how the various data types are stored in MATLAB and in what form they are passed to MEX-files. Generally, it can be said that all data types in MATLAB can be traced back to a single, generic data type: the MATLAB array. All variables that exist in MATLAB are saved as such MATLAB arrays. The corresponding C data structure is the `mxArray`. To be able to distinguish the higher data types from each other, various metadata like variable type, dimension etc. are stored beside the data itself in the `mxArray` structure.

An important difference to using C data types is that matrices – as is the case in Fortran – are stored column-wise. If for example the matrix is A is given by

```
A = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]
A =
     1    2    3    4
     5    6    7    8
     9   10   11   12
```

then the data is arranged in the memory in the following way

| 1 | 5 | 9 | 2 | 6 | 10 | 3 | 7 | 11 | 4 | 8 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Note that in C++, the data is arranged in a row-wise fashion in the memory:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|

In the following, the various MATLAB data types and their C representations are listed.

**Complex double precision matrices** This is the data type that is most commonly used in MATLAB. The data is stored in two vectors of the C type `double` where one contains the real and the other the imaginary part. The vectors can be accessed with the pointer `pr` (pointer to the real data) and `pi` (pointer to the imaginary data). If the matrix is purely real, the pointer `pi` is `NULL`. The matrix has the dimension $(m \times n)$ with $m$ denoting the number of rows and $n$ denoting the number of columns. The values of $m$ and $n$ are

stored in an integer vector. If the number of dimensions exceeds two, the number of elements of this vector is accordingly higher with each of the elements specifying the size of the corresponding dimension.

**Numerical matrices** Beside `double` type matrices, MATLAB also supports the use of single precision floating point numbers as well as 8-, 16- and 32-bit integer numbers, each both signed and unsigned. Aside from this, the data is stored exactly as for complex double-precision matrices.

**Logical matrices** The statuses `true` and `false` are represented by the numbers 1 and 0 respectively and are stored in the data structure as such.

**Empty matrices** A matrix is empty if at least one of the dimensions has a size of zero.

**Sparse matrices** These are matrices where the majority of the elements is zero. Unlike normal matrices, only elements that are unequal to zero are stored in the vectors `pi` and `pr` in this case. The number of elements that is unequal to zero is stored in `nnz`. The indices of the elements are stored in the vectors `ir` and `jc` in the following way

- `ir` points to a vector that contains the row index for each corresponding element in `pr` und `pi`.
- `jc` points to a vector of length $N+1$ that contains the information about the column index. For $0 \leq j \leq N-1$, `jc[j]` is the index in the vector `ir` (as well as `pr` and `pi`) of the first element in the j-th column that is different from zero. The final non-zero element in the same column is then `jc[j+1]-1`. This makes `jc[N]` the index of the last valid element in `ir` and thus the total number of elements `nnz` in the matrix that are different from zero.

**Strings** These contain the C type `char` and are otherwise stored like 16-bit integer matrices without imaginary part. Unlike it is the case in C, the end of MATLAB strings is not marked with a zero-value.

**Cell arrays** Principally, a cell array is nothing but a matrix of MATLAB arrays where each MATLAB array corresponds to one row. The individual MATLAB arrays may store any kind of data type.

**Structures** A structure is stored just like a cell array that consists of one line only. In contrast to cell arrays, the individual elements, which are called fields, have a name that they are addressed by.

**Objects** MATLAB objects are stored and addressed like MATLAB structures. Additionally, it is possible to register methods.

### 7.1.3 An Example

The implementation procedure of a MEX-file can best be explained with an example. The following is based on the example "Passing Two or More Inputs or Outputs" from the MATLAB documentation. For more special techniques, particularly about how to handle data structures that are more complicated compared to simple MATLAB matrices, take a look at other example programs in the MATLAB documentation and consult the documentation of the MATLAB C API.

The following is based on this call of the example function.

```
>> z = xtimesy(x,y);
```

Behind this simple function call is the following C code.

```
/*
 * xtimesy.c - example found in API guide
 *
 * multiplies an input scalar times an input matrix and outputs a
 * matrix
 *
 * This is a MEX-file for MATLAB.
 * Copyright 1984-2000 The MathWorks, Inc.
 */

#include "mex.h"

void xtimesy(double x, double *y, double *z, int m, int n)
{
  int i,j,count=0;

  for (i=0; i<n; i++) {
    for (j=0; j<m; j++) {
      *(z+count) = x * *(y+count);
      count++;
    }
  }
}

/* the gateway function */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
  double *y,*z;
  double  x;
  int     status,mrows,ncols;

  /************ Argument Checking ***************************************/
  /*  check for proper number of arguments */
  /* NOTE: You do not need an else statement when using mexErrMsgTxt
     within an if statement, because it will never get to the else
     statement if mexErrMsgTxt is executed. (mexErrMsgTxt breaks you out of
     the MEX-file) */
  if(nrhs!=2)
    mexErrMsgTxt("Two inputs required.");
  if(nlhs!=1)
    mexErrMsgTxt("One output required.");

  /* check to make sure the first input argument is a scalar */
  if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]) ||
      mxGetN(prhs[0])*mxGetM(prhs[0])!=1 ) {
    mexErrMsgTxt("Input x must be a scalar.");
  }

  /************ Argument Data Extraction ********************************/
  /*  get the scalar input x */
  x = mxGetScalar(prhs[0]);

  /*  create a pointer to the input matrix y */
  y = mxGetPr(prhs[1]);
```

```
  /*  get the dimensions of the matrix input y */
  mrows = mxGetM(prhs[1]);
  ncols = mxGetN(prhs[1]);

  /************ Creation of Output Argument Data ************************/
  /*  set the output pointer to the output matrix */
  plhs[0] = mxCreateDoubleMatrix(mrows,ncols, mxREAL);

  /*  create a C pointer to a copy of the output matrix */
  z = mxGetPr(plhs[0]);

  /************ Call to custom function(s) ******************************/
  /*  call the C subroutine */
  xtimesy(x,y,z,mrows,ncols);

}
```

The gateway routine can be divided into four sections:

1. Validating and possibly returning error messages to MATLAB
2. Extracting parameter data with the corresponding mx functions
3. Generating MATLAB data structures for the output parameters and reserving the memory
4. Calling the own calculation routine

### 7.1.4 Compiling the C Code

For the C code to be usable in MATLAB, a special compilation is required in addition to the gateway routine that provides the interface to MATLAB. With `mex`, MATLAB has a function of its own that uses available compilers and configures them accordingly during the creation process. To select the compiler, the function must be called with the parameter `-setup`. Once confirmed, all available compilers are displayed and a selection can be made. This can look as follows:

```
>> mex -setup

...

Select a compiler:
[1] Microsoft Visual C++ 2010 in C:\Program Files (x86)\Microsoft Visual Studio 10.0

[0] None
```

Depending on what version of Windows is used and what programming tools are installed, different options are offered. Typically, MATLAB comes with the gcc compiler in a 32-bit version. A 64-bit variant is not directly included in the MATLAB installation so that a compiler will have to be installed manually in this case. The various compilers differ mainly in their execution speed. Typically, the Visual C++ compiler yields very good results.

In the next step, the created C code can be compiled under specification of all required *.c source files. This generally looks as follows:

```
mex <gateway file> <additional source files>
```

In our example, the compilation command is therefore:

```
>> mex xtimesy.c
```

The most important flags that can be passed as additional parameters for the compilation are listed in the following. These enforce a specific, desired behavior and allow the programmer to have an influence on the compilation.

-O (Optimization) Activates additional optimization flags of the C compiler.

-v (Verbose) Generates additional outputs that might be helpful for debugging.

-g (Include debug information) Generates symbolic debug information in a *.pdb file that is required for debugging with external development tools. More information can for example be found in the MATLAB documentation under the keyword "Debug C/C++ Language MEX-Files".

### 7.1.5 Debugging on Microsoft Windows Platforms

With the help of an external compiler(Microsoft Visual C++ etc.), we could make full use of any commands the external debugger provides to examine variables, display memory, or inspect registers when we are running the `.c` file in MATLAB. A simple example is demonstrated below.

Matlab offers plentiful examples to help you understand how it works. Example routine `xtimesy.c` can be found in Matlab installation folder.

1. Make sure that a suitable C/C++ compiler is installed on your computer. Microsoft Visual Studio 2015 is installed on lab-computer.
2. Copy the file `xtimesy.c` into your current folder. This file can be found under `C:\Program Files\matlab\R2017a\extern\examples\refbook`.
3. Compile the file `mex -g xtimesy.c`.
4. Start Visual Studio and keep MATLAB running.
5. From the Visual Studio **Debug** menu, select **Attach to Process**
6. In the Attach to Process dialog box, select the MATLAB process and click Attach. Visual Studio loads data then displays an empty code pane.
7. Open the source file `xtimesy.c` by selecting **File➔Open➔File**. Locate `xtimesy.c` in the folder into which you just copied the file.
8. In Visual Studio interface, set a breakpoint by right-clicking the desired line of code and following `Breakpoint➔Insert Breakpoint` on the context menu. It is often convenient to set a breakpoint at `mexFunction` to stop at the beginning of the gateway routine.
9. Run `xtimesy` with appropriate input and output arguments in MATLAB.

The program will be paused at the code line where you just set the break point in Visual Studio. Now, all the debug tools provided by Visual Studio can be applied. For computers running Mac OS and Xcode, the debugging procedure is much more complicated. All the documentations can be accessed in Matlab documentation by executing `doc` and searching "Debug on Mac Platforms".

### 7.1.6 Important Functions

To remove the necessity to look into the MATLAB C API documentation all the time during the programming process, the most important and most commonly used mx and mex functions are presented here. All functions with the prefix mx are declared in the header file `matrix.h`, the functions with the prefix `mex` in the header file `mex.h`. Since `mex.h` embeds `matrix.h` as well, there is no need for you to explicitly embed the latter of these yourself.

An overview can be found under *MATLAB -> Advanced Software Development -> External Programming Language Interfaces -> Application Programming Interfaces to MATLAB -> C/C++ Matrix Library* in the MATLAB documentation.

#### MEX help functions

`mexCallMATLAB`
Calling a MATLAB function, an M-file or a MEX-file

```
int mexCallMATLAB (int nlhs, mxArray *plhs[],
                   int nrhs, mxArray *prhs[], const char *command_name);
```

`nlhs` Number of required output arguments.

`plhs` Pointer to the array in which the output arguments should be stored. The called command stores pointers to the results in this array.

`nrhs` The number of input arguments.

`prhs` Pointer to the array of input arguments.

**Return value:** 0 if successful; unequal to 0 if an error has occurred

`mexErrMsgIdAndTxt`
Providing an error message with identifier and returning to MATLAB

```
void mexErrMsgIdAndTxt (const char *identifier,
                        const char *error_msg, ...);
```

`identifier` String that contains a MATLAB message identifier.

`error_msg` String that contains an error message that should be displayed. It may include format commands like the ANSI C function sprintf.

`...` Additional arguments that are required due to the format commands in `error_msg`.

`mexErrMsgTxt`
Providing an error message and returning to MATLAB

```
void mexErrMsgTxt (const char *error_msg);
```

`error_msg` String that contains an error message that should be displayed.

`mexEvalString`
Running MATLAB expressions in the Workspace of the caller

```
int mexEvalString (const char *command);
```

`command` String that contains the MATLAB command to be called.

**Return value:** 0 if successful; unequal to 0 if an error has occurred

Like `mexCallMATLAB`, this also allows to run any kind of MATLAB command. The difference is that this commands is run in the Workspace of the caller. Input parameters must exist in the Workspace and output parameters are saved in the Workspace if applicable. The results are therefore not directly available inside the MEX-file.

`mexPrintf`

Displaying some output on the MATLAB console in ANSI C printf style.

```
int mexPrintf (const char *format, ...);
```

`format, ...` ANSI C printf format string and the corresponding optional arguments.

**Return value:** Number of displayed characters including characters like `\n` and `\b`

`mexWarnMsgIdAndTxt`

Displaying a warning message with identifier (the MEX-file is not terminated)

```
void mexWarnMsgIdAndTxt (const char *identifier,
                         const char *warning_msg, ...);
```

`identifier` String that contains a MATLAB message identifier.

`warning_msg` String that contains a warning message that should be displayed. It may include format commands like the ANSI C function sprintf.

`...` Additional arguments that are required due to the format commands in `warning_msg`

`mexWarnMsgTxt`

Displaying a warning message (the MEX-file is not terminated)

```
void mexWarnMsgTxt (const char *warning_msg);
```

`warning_msg` String that contains a warning message that should be displayed.

### Creating and deleting mxArrays

`mxCreateCharArray`

Creating an empty N-dimensional string mxArray

```
mxArray *mxCreateCharArray (int ndim, const in *dims);
```

`ndim` The desired number of dimensions in the string mxArray. The number must be positive. For 0, 1 or 2, the produced mxArray will be two-dimensional.

`dims` Dimension array. Each element specifies the size of the respective dimension.

**Return value:** If successful, the pointer to the created mxArray is returned. If unsuccessful, NULL is returned in stand-alone applications. Within MEX-files, the MEX-file is terminated and an error message is passed to MATLAB.

`mxCreateDoubleMatrix`

Creating an empty two-dimensional mxArray with double type values

```
mxArray *mxCreateDoubleMatrix (int m, int n,
                               mxComplexity ComplexFlag);
```

`m` Number of requested lines

`n` Number of requested columns

`ComplexFlag` Either `mxREAL` or `mxCOMPLEX` depending on whether the data to be stored has an imaginary component or not.

**Return value:** If successful, the pointer to the created mxArray is returned. If unsuccessful, NULL is returned in stand-alone applications. Within MEX-files, the MEX-file is terminated and an error message is passed to MATLAB.

The elements of the generated matrix are initialized with zero.

`mxCreateDoubleScalar`

Creating a double type scalar and initializing it with the specified value

```
mxArray *mxCreateDoubleScalar (double value);
```

`value` The value that the "Array" is initialized with.

**Return value:** If successful, the pointer to the created mxArray is returned. If unsuccessful, NULL is returned in stand-alone applications. Within MEX-files, the MEX-file is terminated and an error message is passed to MATLAB.

`mxCreateDoubleScalar` is nothing but the abbreviated form of

```
pa = mxCreateDoubleMatrix (1, 1, mxREAL);
*mxGetPr(pa) = value;
```

`mxCreateString`

Creating a string of length $n$ that is initialized with the specified string

```
mxArray *mxCreateString (const char *str);
```

`str` The C string that the string mxArray should be initialized with.

**Return value:** If successful, the pointer to the created mxArray is returned. If unsuccessful, NULL is returned in stand-alone applications. Within MEX-files, the MEX-file is terminated and an error message is passed to MATLAB.

`mxDestroyArray`

Dynamically releasing memory that has been reserved with a mxCreate... routine

```
void mxDestroyArray (mxArray *array_ptr);
```

`array_ptr` The pointer to the mxArray, the memory of which should be freed.

This function frees all of the space that used to be occupied by an mxArray. Do not use this function on mxArrays that you wish to use as return parameters.

`mxDuplicateArray`

Creating a copy of an array

```
mxArray *mxDuplicateArray (const mxArray *in);
```

`in` Pointer to the mxArray that should be copied.

**Return value:** Pointer to the copy

This function generates a so-called deep copy of the specified mxArray, i. e. all of the data is actually copied, not just pointers to the data.

### Reading data from mxArrays

`mxGetDimensions`

Returns a vector with the size of the individual dimensions

```
const int *mxGetDimensions (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Address of the first element of the dimension vector. Each element specifies the size of the respective dimension.

Use `mxGetNumberOfDimensions` to determine the number of dimensions.

`mxGetM`

Returns the number of lines

```
int mxGetM (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Number of lines in the specified mxArray.

This function always returns the size of the first dimension. For a two-dimensional matrix, this corresponds to the number of lines.

`mxGetN`

Returns the number of columns or the number of elements

```
int mxGetN (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Number of columns in the specified mxArray.

This function returns the number of columns in case of a two-dimensional matrix. For a matrix of dimension N, the return value is the product of the sizes of the dimensions 2 to N. For a matrix of dimension $(2 \times 5 \times 4 \times 3)$ for example, the function would return 60.

`mxGetNumberOfDimensions`

Returns the number of dimensions

```
int mxGetNumberOfDimensions (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Number of dimensions of the specified mxArray. The returned number is always greater or equal to 2.

`mxGetNumberOfElements`

Returns the number of elements of an mxArray

```
int mxGetNumberOfElements (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Number of elements of the specified mxArray.

`mxGetPi`

Returns the imaginary data of an mxArray

```
double *mxGetPi (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Pointer to the imaginary data of the specified mxArray. If there is no imaginary data or if an error occurred, NULL is returned.

`mxGetPr`

Returns the real data of an mxArray

```
double *mxGetPr (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Pointer to the real data of the specified mxArray. If there is no real data or if an error occurred, NULL is returned.

`mxGetScalar`

Returns the real component of the first data element of an mxArray.

```
double mxGetScalar (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object which may not be a cell or structure mxArray though.

**Return value:** The value of the first real element of the specified mxArray. If the data is anything else than a double, the respective data format is converted to double.

`mxGetString`

Copies an mxArray string to a C string

```
int mxGetString (const mxArray *array_ptr, char *buf, int buflen);
```

`array_ptr` Pointer to a string mxArray object, i. e. it must be of the `mxCHAR_CLASS` class.

`buf` Pointer to the begin of the memory range in which the string should be written. The result is an ANSI C conform string terminated with NULL.

`buflen` Maximum number of characters that may be written in `buf`, including the terminating NULL.

**Return value:** 0 if successful; 1 if an error has occurred

If the specified mxArray is a string matrix, the strings are concatenated and written in `buf` so that a longer string is formed.

### Writing in mxArrays

`mxSetDimensions`

Sets the number and size of the dimensions

```
int mxSetDimensions (mxArray *array_ptr, const int *dims, ind ndim);
```

`array_ptr` Pointer to an mxArray object

`dims` Array with the dimensions of the individual dimensions

`ndims` Number of dimensions

**Return value:** 0 if successful; 1 if an error has occurred

`mxSetM`

Sets the number of rows

```
void mxSetM (mxArray *array_ptr, int m);
```

`array_ptr` Pointer to an mxArray object

`m` Number of rows

`mxSetN`

Sets the number of columns

```
void mxSetN (mxArray *array_ptr, int n);
```

`array_ptr` Pointer to an mxArray object

`n` Number of columns

`mxSetPi`

Sets the imaginary data of an mxArray differently

```
void mxSetPi (mxArray *array_ptr, double *pi);
```

`array_ptr` Pointer to an mxArray object

`pi` Pointer to the first element of an array. Each element represents the imaginary component of a value.

`mxSetPr`

Sets the real data of an mxArray differently

```
void mxSetPr (mxArray *array_ptr, double *pr);
```

`array_ptr` Pointer to an mxArray object

`pr` Pointer to the first element of an array. Each element represents the real component of a value.

### Tests

`mxIsChar`

Indicates whether it is a string

```
bool mxIsChar (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Logical 1 (true) if `array_ptr` points to an mxArray of the class `mxCHAR_CLASS`; logical 0 otherwise.

`mxIsComplex`
Indicates whether the data is complex

```
bool mxIsComplex (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Logical 1 (true) if `array_ptr` points to a numerical mxArray that contains complex data; logical 0 otherwise.

`mxIsDouble`
Indicates whether the stored data has the data type double

```
bool mxIsDouble (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Logical 1 (true) if `array_ptr` points to an mxArray with data of type double; logical 0 otherwise.

`mxIsEmpty`
Indicates whether an mxArray is empty

```
bool mxIsEmpty (const mxArray *array_ptr);
```

`array_ptr` Pointer to an mxArray object

**Return value:** Logical 1 (true) if `array_ptr` points to an empty mxArray; logical 0 otherwise.

### Memory management

`mxCalloc`
Reserving memory that is dynamically initialized with zero with the MATLAB memory manager.

```
void *mxCalloc (size_t n, size_t size);
```

`n` Number of elements for which the memory should be reserved.

`size` Number of bytes per element.

**Return value:** Pointer to the beginning of the reserved storage block that was initialized with NULL. Within a MEX-file, the execution is terminated if no storage could be reserved. Within a stand-alone application, NULL is returned in this case.

`mxMalloc`
Dynamically reserving memory with the MATLAB memory manager

```
void *mxMalloc (size_t size);
```

`size` Size of the storage to be reserved in bytes.

**Return value:** Pointer to the beginning of the reserved storage block. Within a MEX-file, the execution is terminated if no storage could be reserved. Within a stand-alone application, NULL is returned in this case.

`mxRealloc`
Reserving memory anew with the MATLAB memory manager

```
void *mxRealloc (void *ptr, size_t size);
```

`ptr` Pointer to a storage block that has been reserved by `mxCalloc`, command-mxMalloc or `mxRealloc`.

`size` Size of the storage to be reserved in bytes.

**Return value:** Pointer to the beginning of the newly reserved storage block. Within a MEX-file, the execution is terminated if the available space is insufficient. Within a stand-alone application, NULL is returned in this case.

`mxFree`

Space is released again that has been reserved with the MATLAB memory manager

```
void mxFree(void *ptr);
```

`ptr` Pointer to a storage block that has been reserved by `mxCalloc`, command-mxMalloc or `mxRealloc`.

### 7.1.7 Pointers in C

In C, there is a variable type for which there is no equivalent in MATLAB: pointers.

A pointer is a variable that contains a memory address. At this memory address, there can be for example a variable of the `integer`, `double` or `mxArray` type or yet another pointer can be stored there.

```
int a;
a=5;
int *PointerInteger;

PointerInteger=&a;
cout << "Value of a: " << *PointerInteger;
```

A pointer is declared like a variable except that a * is put in front. The variable `a` is an integer and `PointerInteger` is a pointer to an integer. With the ampersand (`&`), the address of a variable is returned, i. e. by calling `PointerInteger=&a;`, the address of `a` is stored in `PointerInteger`.

To access the value of a variable that a pointer points to, the star (`*` - indirection operator) is used to "dereference" the pointer. In this example, the integer value 5 is returned when `*PointerInteger` is called.

Things are getting interesting when the pointer points to yet another pointer:

```
int a=5;
int *PointerInteger;
int **PointerToPointerInteger;

PointerInteger=&a;
PointerToPointerInteger=&PointerInteger;
cout << "Value of a: " << **PointerToPointerInteger;
```

Here, `PointerToPointerInteger` is a pointer to a pointer to an integer. The role of `&` and `*` remains the same.

It is also possible to define pointer fields. With `int *PointerIntField[2];`, a field that consists of two pointers to integers is defined and enough working space is reserved to save both pointers. Technically, `PointerIntField` is also a pointer, i. e. `**PointerIntField` returns the integer value at the address that the first element of the field points to.

Example:

```
int mexCallMATLAB (int nlhs, mxArray *plhs[],
                   int nrhs, mxArray *prhs[], const char *command_name);
```

The passed parameter `plhs` of the interface definition, e. g. of `mexCallMATLAB` is thus a pointer to a pointer to an `mxArray`.

## 7.2 Calling MATLAB from C Programs

Beside the possibility to call C/C++ functions from MATLAB, it is also possible to use and control MATLAB out of C/C++ programs. This is done with the help of the so-called MATLAB Engine. It is a good idea to use this if established algorithms that are provided by MATLAB should be included into existing C programs where the implementation in C would require an enormous effort.

### 7.2.1 Basic Principle

To call a MATLAB function from C/C++, a new instance of MATLAB must be opened first. Then, a communication channel to this instance is installed via which commands and data can be transferred.

What happens first usually is that data from C/C++ is placed in the MATLAB Workspace. During this progress, data fields are copied and given a name. To run a function, a command in text form is sent to MATLAB in the next step. This command is executed and the text of a possible output is returned to the calling program. Finally, it is also possible to return contents of data fields in the MATLAB Workspace to the calling C/C++ program. To do so, the C/C++ application copies a data field of a predefined name from the MATLAB Workspace into the address space of the C/C++ application.

All objects that are moved from C/C++ to MATLAB or in the opposite direction, follow the conventions of the `mxArray` data type. You are already familiar with this data type from the MEX-files. Refer to Section 7.1.6 for further information. Objects that are generated in C/C++ and transferred to MATLAB and such objects that are requested by MATLAB from C/C++ via the Engine must be released again in the C/C++ program with `mxDestroy`.

To use the interface of the MATLAB Engine, the following line must be inserted:

```
#include "engine.h"
```

Under the operating system Microsoft Windows, for the purpose of communication with MATLAB, an instance with a corresponding `MATLAB Command Window` is

opened. While a C/C++ program is running, you can also insert commands in this Command Window.

Figure 7.2 once more summarizes how this all works.

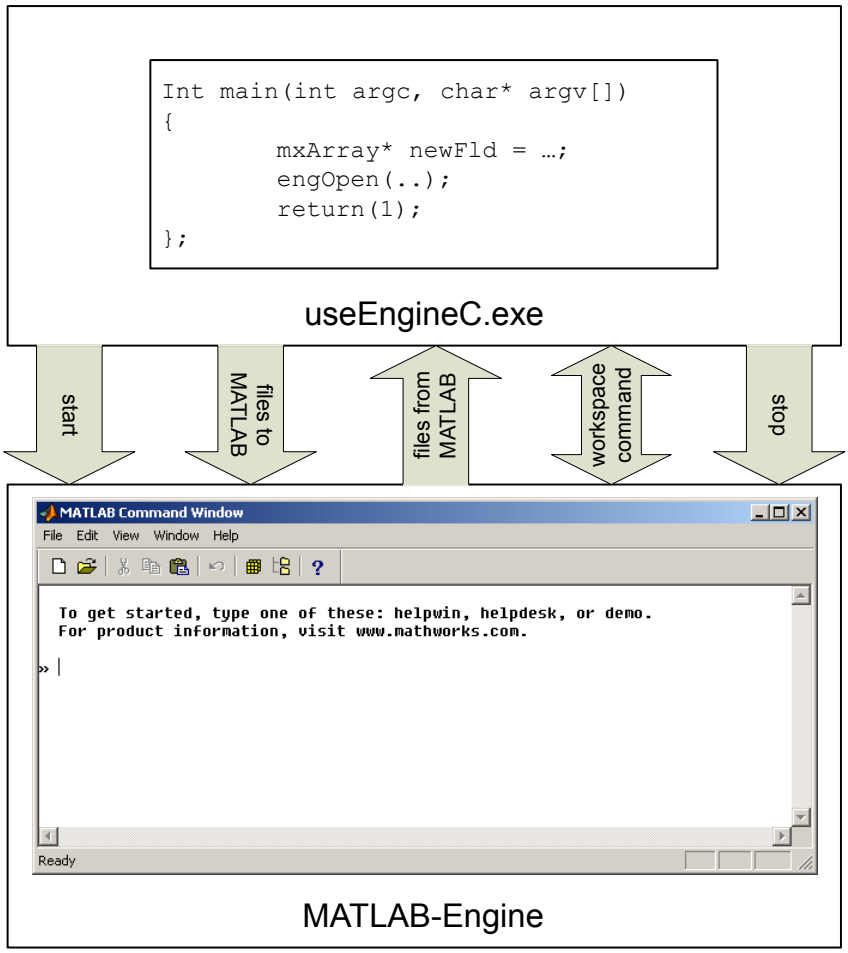



**Figure 7.2:** Data transfer between C/C++ and the MATLAB Engine.

Mex-functions and MATLAB Engine both allow you to access MATLAB functions from C/C++ code file. However, they belong to different API sets and can not be used interchangeably.

**Attention!**

1. If you need to execute a `.c` file in **MATLAB**, while calling MATLAB functions in this C program, please use **mex functions**.
2. Conversely, if you run a `.c` file in a **C compiler** (Visual C++ etc.), **MATLAB Engine** is suitable for the purpose of calling MATLAB functions

### 7.2.2 Important Functions

In this section, the most important interface functions for the use of the MATLAB Engine are listed:

`Engine`
: (Data type) Pointer to a variable of type `Engine`:

```
Engine*
```

This data type serves as a communication reference to the opened MATLAB instance and must be obtained once at the beginning. All commands always require this argument as a reference.

`engOpen`

With this command, the MATLAB instance is opened from a C/C++ program:

```
Engine* engOpen(const char* startcmd);
```

The return value is the communication reference that is required for all of the following communication calls. The argument `startcmd` allows to specify a MATLAB start command.

`engPutVariable`

With this command, a field in form of a created `mxArray` data type can be copied to the MATLAB Workspace:

```
int engPutVariable(Engine* ep, const char* name, const mxArray* mp);
```

In the Workspace, the name according to the argument `name` is assigned to the object. If a field with that name already exists, the contents of this field are overwritten.[1] To specify the communication link, the MATLAB communication reference is also passed as the `ep` argument.

`engOutputBuffer`

With this command, a character field can be associated with the output of the MATLAB Engine:

```
int engOutputBuffer(Engine* ep, char* p, int n);
```

Subsequently, the output is placed in the specified character field when a command is executed in the MATLAB Workspace and can thus be displayed in C/C++. The argument `n` determines how many characters can be stored in the field. The communication reference is again the first calling argument.

`engEvalString`

With this function, MATLAB commands can be passed from C/C++ to the MATLAB Workspace so that they can be executed there:

```
int engEvalString(Engine* ep, const char* command);
```

The command is specified as the string `command`. The text that is produced during the execution is stored in the associated text field if applicable.

`engGetVariable`

With this command, the contents of a field in the MATLAB Workspace can be passed to C/C++:

```
mxArray *engGetVariable(Engine* ep, const char* name);
```

The argument `name` identifies the field in the MATLAB Workspace. The returned fields must be released in the C/C++ program (`mxDestroy`...).

[1] Note: It is therefore not possible to perform a call by reference variable transfer.

`engClose`

This command closes the MATLAB instance:

```
int engClose(Engine* ep);
```

Subsequent function calls with the communication reference are not successful.

Refer to the MATLAB documentation for a detailed description of the functions listed above and others.

## 7.3 Exercises

### Exercise 7.1 Embedding C code into MATLAB - flanger

In the experiment directory, you can find the C-file flanger.c with the corresponding header file that you are supposed to embed into MATLAB. Inspect the function and determine the required parameters and their format.

**Note:** The flanger is an effect that is used in electronical music. It duplicates the input signal. One track is mixed with the other with a delay of about 1 to 20 milliseconds. You can find furter information at Wikipedia amongst others.

Now create a C-file and implement a corresponding MEX interface function. (You can do all of this in the MATLAB editor!) Make sure that you catch invalid inputs. This is even more important in MEX-files than in M-files since MATLAB has no option to give you information about the error source here. Rather than that, the best that can happen is that the function simply crashes. Things get even worse if the function happens to work even when the input is invalid but the results that are returned make no sense whatsoever. Also create an M-file that contains the corresponding help.

**Note:** Examine the given files properly!

Afterwards, apply the flanger effect on the given source file "music.wav".

### Exercise 7.2 Embedding C code into MATLAB - NLMS

In the following, you are supposed to move the NLMS algorithm that you got to know in experiment 5 to a MEX-file. Unlike it has been the case in experiment 5, this MEX-file will be used for system identification only rather than for echo cancellation.

In the experiment directory, a test signal and the system response are provided in the files `testsig.mat` and `response.mat` respectively. As test signal, a so-called odd-perfect sequence of period length 500 was used. Your task is now to find the analyzed system.

To do so, extend the MEX-file `mexNlmsVorgabe.c` that receives the test signal (vector), the period length of the odd-perfect sequence (scalar), the system response (vector) and the factor $\alpha$ (scalar) as input parameters. Now, in the function, compute the impulse response of the system for each sample according to the NLMS algorithm. In the end, return all impulse responses in a large matrix to MATLAB.

Finally, add the functionality that a progress bar is displayed in MATLAB while the function runs. You can do this by calling the respective MATLAB functions from within the MEX-file (keywords: waitbar, mexCallMATLAB, mexEvalString)

**Note:** Use the help to find out how to call the function `waitbar` from the MEX-file and how to pass the corresponding parameters to it.

# Chapter 8

# Multi-channel Signal Processing

In the preceding experiments, digital filters with fixed and time-variant coefficients have been covered amongst other things. In all cases, the filtering of a single signal was considered. In the last two experiments of this laboratory,the multi-channel filtering of signals will be introduced. The goal is to build and analyze a simple, multi-channel system for improving a noisy speech signal with the help of MATLAB. In the process, you will also learn about graphically displaying more complex, multi-dimensional functions.

## 8.1 Signal Processing with Microphone Arrays

The filtering of signals that are recorded with multiple microphones (sensors) - so-called *microphone arrays* - provides the possibility to exploit the *spatial* properties of the signal sources in addition to the temporal properties. For example, the makes it possible to detect the direction of incidence and / or separate one signal source from another, spatially separated source. While a multitude of approaches for the realization of such *space-time filters* exist; we only consider the class of so-called *beamformers* here which is presented in the following.[1]

[1]The reader finds an extensive presentation of the application of microphone arrays for noise reduction in numerous publications like e. g. J. Bitzer *Mehrkanalige Geräuschunterdrückungssysteme - eine vergleichende Analyse*, dissertation, Bremen univ., 2002; M. Dörbecker *Mehrkanalige Signalverarbeitung zur Verbesserung akustisch gestörter Sprachsignale am Beispiel elektronischer Hörhilfen*, dissertation, RWTH Aachen, 1998; S. Fischer *Adaptive Mehrkanalgeräuschunterdrückung bei gestörter Sprachsignalen unter Berücksichtigung der räumlichen Kohärenz des Geräuschfeldes*, dissertation, Bremen univ., 1996; M. Brandstein et al. *Microphone Arrays*, Springer Verlag, 2001.

### 8.1.1 Structure of a Beamformer

The fundamental principle of a digital beamformer is depicted in Figure 8.1.

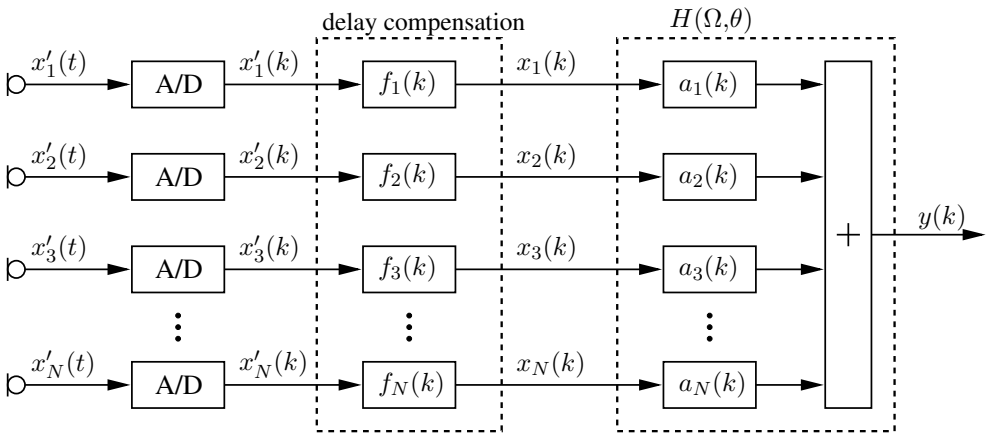



**Figure 8.1:** Concept of a beamformer with $N$ channels.

The signals $x'_n(t)$ from $N$ separate microphones are converted to digital signals $x'_n(k)$, $n = 1,2,\ldots N$ by means of analog-digital conversion. It is assumed that the microphones exhibit an ideal, omni-directional directivity, i. e. the transmission behavior of the microphones is independent from the direction of incidence of the sound wave and the frequency, which is only approximately true in practice. The signals $x'_n(k)$ therefore yield a temporal and spatial sampling of a wave field and the subsequent filtering and summation of the signals $x'_n(k)$ leads to a 'space-time filtering'.

Firstly, the discrete microphone signals $x'_n(k)$ are filtered with the filter $f_n(k)$ to achieve a *delay compensation*[2]. This directs the spatial direction of preference (beam-steering). The computation of the propagation time differences will now be discussed for the microphone setup shown in Figure 8.2. What is characteristic about the de-

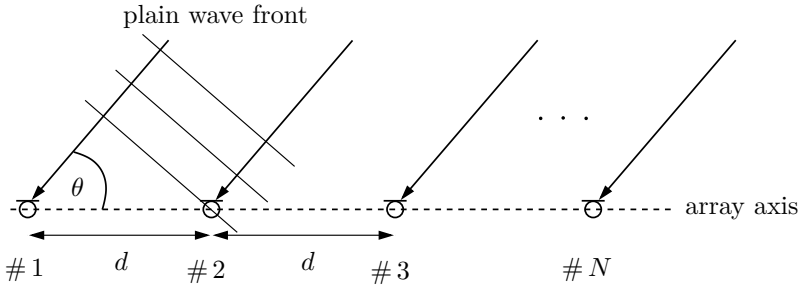



**Figure 8.2:** Microphone setup for a (linear) line array. The $N$ microphones are positioned on the same axis and are arranged equidistantly. The microphone 1 is assumed to be the reference microphone.

picted (linear) *line array* is that its sensors lie on the same axis and that they are arranged equidistantly. For the computation of the delay differences between the sensors, the incident angle $\theta$ of the incoming sound wave (with respect to the specified axis) must be selected. In the following, it is assumed that a plane sound wave hits the microphone array (far-field approximation).

[2]When a signal has a prime, this indicates that this is the signal before the delay compensation.

The delay $\tau$ between two adjacent sensors for the incident angle $\theta$ can then be described by the relation

$$\tau(\theta) = \frac{d \cdot \cos\theta}{c} \tag{8.1}$$

according to Figure 8.2 with the propagation velocity of sound in air $c = 340\,\mathrm{m/s}$. This delay only depends on one angle as the considered line array's directivity is rotationally symmetric with respect to the array axis for direction-independent (omnidirectional) microphones. The maximum delay difference between two neighboring sensor is consequently

$$\tau_{\max} = \tau(\theta = 0) = \frac{d}{c}. \tag{8.2}$$

The incident angle of the useful signal source $\theta_0$ that the array is aligned for is of particular interest. Here, it is assumed that this direction of incidence is known a priori or that it can be estimated reliably. When the array is aligned so that the useful signal hits the array perpendicular to the array axis ($\theta_0 = \pm\pi/2$), this is referred to as a *Broadside* Array (see also Figure 8.2). In this case, there is thus no need for a delay compensation. Arrays where the direction of incidence coincides with the array axis ($\theta_0 = 0$) are called *Endfire* arrays. The useful signal delay between adjacent sensors is given by equation (8.2) in this case.

The aim of the *delay compensation* is to make the useful signal source arrive at all microphones simultaneously. A delay of the $n$th microphone signal $x'_n(k)$ by

$$k_n = (n-1) \cdot \tau(\theta_0) \cdot f_s \quad ; \quad n = 1,2,\ldots,N \tag{8.3}$$

samples compensates the difference in travel times where $f_s$ denotes the sampling frequency of the A/D conversion. The delay values $k_n$ may possibly not be integer so that it is not possible to compensate the delay by simple delay elements.

The realization of a non-integer delay is possible with the help of a so-called *fractional-delay filter*. Such a filter is an ideal lowpass with the cut-off frequency $\Omega = \pi$ and a linear phase response $\varphi(\Omega) = -k_n\Omega$, expressed by the normalized frequency $\Omega = 2\pi f/f_s$. The infinitely extended impulse response of this non-causal filter is given by

$$f_n(k) = \mathrm{sinc}(k - k_n). \tag{8.4}$$

The sinc function is defined as

$$\mathrm{sinc}(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x} & ; \quad x \neq 0 \\ 1 & ; \quad x = 0. \end{cases} \tag{8.5}$$

A causal FIR filter (i. e. $f_n(k) = 0$ for $k < 0$) is obtained by temporally shifting and windowing the impulse response, e. g. with a rectangular window. (This is equivalent to the 'modified Fourier approximation' covered in Section 4.3.2.) This approximation introduces - depending on the window length - a slight signal distortion that is neglected in the following however.

The $N$ delay compensated signals $x_n(k)$ are filtered by $N$ FIR filters with the impulse responses $a_n(k)$ of length $L$ and are subsequently summed up. By the selection of

these filters, the frequency-dependent, spatial *directivity* of the beamformer is defined which will be discussed in more detail in the following. Depending on whether the filter coefficients are adapted or time-invariant, this is referred to as an adaptive or non-adaptive (fixed) beamformer respectively. Here, only non-adaptive beamformers will be considered. In this case, the beamformer in Figure 8.1 is also called a *Filter-and-Sum Beamformer*. It holds for the output signal $y(k)$

$$y(k) = \sum_{n=1}^{N} \sum_{l=0}^{L-1} x'_n(k-l) \cdot \underbrace{f_n(l) \star a_n(l)}_{\doteq b_n(l)} = \sum_{n=1}^{N} \sum_{l=0}^{L-1} x'_n(k-l) \cdot b_n(l) \tag{8.6}$$

where the symbol $\star$ marks a discrete convolution. It follows in the frequency domain

$$Y(\Omega,\theta) = \sum_{n=1}^{N} X'_n(\Omega) \cdot \underbrace{\exp(-j\, k_n\, \Omega) \cdot A_n(\Omega)}_{\doteq B_n(\Omega)} = \sum_{n=1}^{N} X'_n(\Omega) \cdot B_n(\Omega). \tag{8.7}$$

If the signal of an arbitrary reference microphone is called $x'_r(k)$ and the angle-dependent travel time difference between the reference microphone $r$ and microphone $n$ of the array $\tau_{n,r}(\theta)$, it holds

$$X'_n(\Omega) = X'_r(\Omega) \cdot \exp(j\, \Omega f_s\, \tau_{n,r}(\theta)) \quad ; \quad n = 1,2,\ldots,N\,. \tag{8.8}$$

This means for equation (8.7) that

$$Y(\Omega,\theta) = X'_r(\Omega) \sum_{n=1}^{N} \exp(j\, \Omega f_s\, \tau_{n,r}(\theta)) \cdot B_n(\Omega) \tag{8.9}$$

$$= X'_r(\Omega) \sum_{n=1}^{N} A_n(\Omega) \cdot \exp\left(j\, \Omega(f_s\, \tau_{n,r}(\theta) - k_n)\right) \tag{8.10}$$

$$\doteq X'_r(\Omega) \cdot H(\Omega,\theta). \tag{8.11}$$

The function $H(\Omega,\theta)$ is the *transfer function* of the beamfrmer. It depends on the frequency $\Omega$, the incident angle of the wave front $\theta$ as well as the resulting delays $\tau_{n,r}(\theta)$. It hence describes the frequency-dependent behavior of the beamformer as a function of the direction of incidence of the wave front. The compact vector representation of equation (8.9) is

$$Y(\Omega,\theta) = X'_r(\Omega) \cdot \mathbf{B}(\Omega)^T \cdot \mathbf{E}(\Omega,\theta) \tag{8.12}$$

mit den $N \times 1$ Vektoren

$$\mathbf{B}(\Omega) = [\, B_1(\Omega,), B_2(\Omega), \ldots, B_N(\Omega)\,]^T \tag{8.13}$$

$$\mathbf{E}(\Omega,\theta) = [\, \exp(j\, \Omega f_s\, \tau_{1,r}(\theta)), \exp(j\, \Omega f_s\, \tau_{2,r}(\theta)), \ldots \exp(j\, \Omega f_s\, \tau_{N,r}(\theta))\,]^T. \tag{8.14}$$

The vector $\mathbf{E}$ is also referred to as the *array steering vector* as it describes the spatial alignment of the array as a function of the angle $\theta$[3]. (This vector representation is

[3] In the literature, a more general vector representation is typically found that describes the delay by the inner product of the wave number vector and the vector for the respective microphone position. This general representation was not used here in favor of a more simple and clear description that depends only on the incident angle $\theta$.

also used in the MATLAB programs that are given for this experiment!) With the Wiener-Lee relation, it follows from equation (8.9) that

$$\Phi_{yy}(\Omega,\theta) = |H(\Omega,\theta)|^2 \cdot \Phi_{x'_r x'_r}(\Omega). \tag{8.15}$$

$\Phi_{yy}(\Omega,\theta)$ is the power spectral density (PSD) of the output signal $y(k)$ and $\Phi_{x'_r x'_r}(\Omega)$ is the PSD of the reference signal $x_r(k)$. It should be noted that the transfer function $H(\Omega,\theta)$ according to equation (8.9) depends on the choice of the reference microphone. Here, the line array in Figure (8.2) is considered with microphone 1 as reference microphone. In this case, the simple relation

$$\tau_{n,1}(\theta) = (n-1)\frac{d \cdot \cos\theta}{c} \tag{8.16}$$

holds for the delay between the reference microphone and microphone $n$.

Self-explanatorily, a different convention is also possible and there is no need for the reference microphone to necessarily be part of the microphone array.

A simple realization of the filter-and-sum beamformer is the *delay-and-sum beamformer* (DS BF) which is also known as the conventional beamformer. The name already indicates how it works: The *delay-compensated* signals $x_n(k)$ are only multiplied with the factor $1/N$ and summed up, i. e.

$$a_n(k) = \frac{1}{N} \cdot \delta(k) \quad ; \quad n = 1,2,\dots N \tag{8.17}$$

with the Fourier transform

$$A_n(\Omega) = \frac{1}{N}. \tag{8.18}$$

Due to the in-phase superposition, an output signal is generated that is maximal for the main direction of incidence while the microphone signals for all other directions more or less cancel out as a result of destructive interference. An advantage of the delay-and-sum beamformer is its simple implementation and robustness against microphones that are afflicted with errors (e. g. sensor noise).

### 8.1.2 Measures

The evaluation of systems requires the introduction of appropriate measures. These allow an analysis of the system and commonly serve as a basis for its optimization.

A helpful quantity for the description of the spatial directivity of a beamformer is its *beam pattern*. This beam pattern follows from the transfer function according to equations (8.9) and (8.15) as

$$\Psi(\Omega,\theta) \doteq \left|\sum_{n=1}^{N} \exp(j\,\Omega f_s\,\tau_{n,r}(\theta)) \cdot B_n(\Omega)\right|^2 = |H(\Omega,\theta)|^2 = \frac{\Phi_{yy}(\Omega,\theta)}{\Phi_{x'_r x'_r}(\Omega)}. \tag{8.19}$$

The beam pattern is a measure for the sensitivity of the beamformer as a function of the direction of incidence $\theta$ of the sound wave as well as the frequency $\Omega$. The sensitivity of the beamformer is described by the ratio of the PSD of the output signal $\Phi_{yy}(\Omega,\theta)$ and the PSD of the reference signal $\Phi_{x'_r x'_r}(\Omega)$.

A more compact measure for the description of the directivity of an array is the so-called *gain* (of the microphone array) that - unlike the beam pattern - does not depend on the direction of incidence $\theta$. Generally, the gain is defined as the sensitivity of the array in the main direction of incidence compared to the sensitivity of sound with omni-directional incidence[4]:

$$G(\Omega) = \frac{\Psi_{\text{gen}}(\Omega,\theta_0,\phi_0)}{\frac{1}{4\pi}\int_0^{2\pi}\int_0^{\pi}\Psi_{\text{gen}}(\Omega,\theta,\phi)\sin\theta\,\mathrm{d}\theta\,\mathrm{d}\phi} \tag{8.20}$$

where the beam pattern $\Psi(\Omega,\theta)$ of the line array considered here is independent of the angle $\phi$. After several steps, the representation

$$G(\Omega) = \frac{\left|\sum_{n=1}^{N} B_n(\Omega)\exp(j\Omega f_s \tau_{n,r}(\theta_0))\right|^2}{\sum_{n=1}^{N}\sum_{m=1}^{N} B_n(\Omega)B_m^*(\Omega)R_{m,n}(\Omega)} \tag{8.21}$$

of equation (8.20) is acquired. The time difference $\tau_{n,r}(\theta_0)$ is given by equation (8.16) with the reference microphone $r = 1$. The *coherence function* of the diffuse sound field between sensors $n$ and $m$ is represented by the function

$$R_{n,m}(\Omega) = \operatorname{sinc}\left(\frac{\Omega f_s}{\pi c} d_{n,m}\right). \tag{8.22}$$

In this expression, $d_{n,m}$ denotes the distance between the sensors $n$ and $m$ which is given by

$$d_{n,m} = d \cdot |n - m| \tag{8.23}$$

in this case. By inserting the coherence function of an *uncorrelated* sound field

$$R_{n,m}(\Omega)\Big|_{\text{uncorr}} = \delta(m - n) \tag{8.24}$$

into equation (8.21) with $\delta(n)$ being the discrete dirac function, the *white noise gain*

$$G_{\text{WN}}(\Omega) = \frac{\left|\sum_{n=1}^{N} B_n(\Omega)\ \exp(j\,\Omega\,f_s\,\tau_{n,r}(\theta_0))\right|^2}{\sum_{n=1}^{N}|B_n(\Omega)|^2} \tag{8.25}$$

is obtained. An uncorrelated sound field can be caused by sensor noise which is why the reciprocal of the white noise gain is also considered the susceptibility of the beamformer towards sensor noise.

### 8.1.3 Properties of the Delay-and-Sum Beamformer

The beam pattern of a delay-and-sum beamformer in broadside configuration is depicted in Figure 8.3. For the considerations in this section, a line array with $N = 8$

[4]The array gain refers to sound fields with arbitrary coherence properties in some publications. The gain of the array for the special case of omni-directional, i. e. diffuse sound field is then termed the directivity factor or directivity index if given in dB.

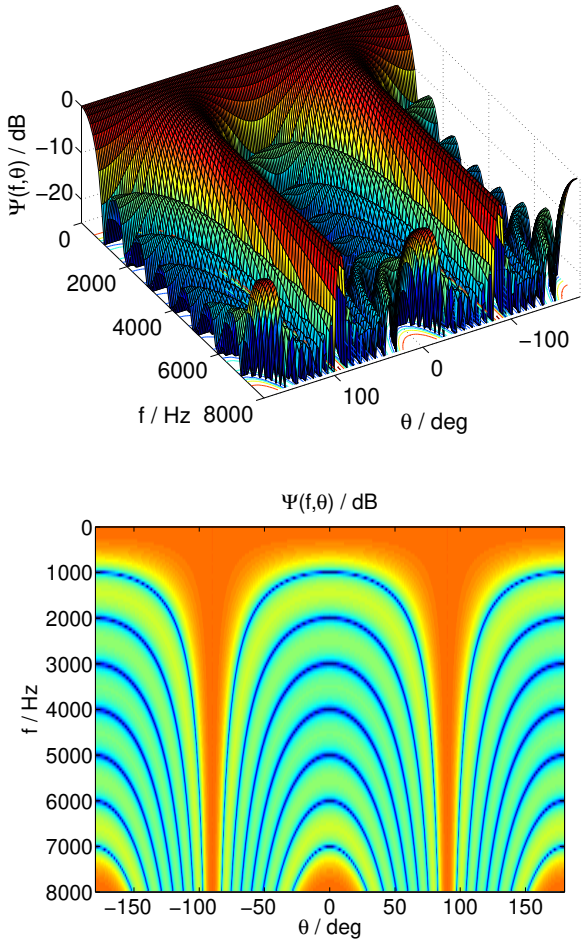



**Figure 8.3:** Different graphical representations of the beam pattern of a delay-and-sum beamformer in broadside configuration ($\theta_0 = 90°$) for N=8 microphones in a distance of 4.25 cm.

microphones and a microphone distance of $d = 4.25\,\text{cm}$ is used. The sampling frequency is $f_s = 16\,\text{kHz}$. In the beam pattern, a frequency-independent sensitivity can be observed for the incident angle $\theta_0$. A signal from this main direction of incidence is thus transmitted over the system *free of any distortions*. Beside the main lobe, there are also sidelobes, so-called *grating lobes* that occur at a frequency of 8 kHz. The microphone array performs, as indicated in the beginning, a spatial and temporal sampling of a wave field. Analogously to the temporal sampling theorem (Nyquist condition), a 'spatial sampling theorem' must be fulfilled to prevent ambiguities in the beam pattern. Spatial *aliasing* can be avoided as long as the condition

$$\frac{d}{\lambda} \leq \frac{1}{1 + |\cos\theta_0|} \tag{8.26}$$

is met with the wavelength being determined by the wave equation $c = \lambda f$. For low frequencies, the delay-and-sum beamformer features no directivity. This omnidirectionality and the occurrence of grating lobes are reflected in the array gain which is shown in Figure 8.4. For low frequencies, hardly any array gain is achieved. At 8 kHz, the gain slightly drops due to the grating lobes.

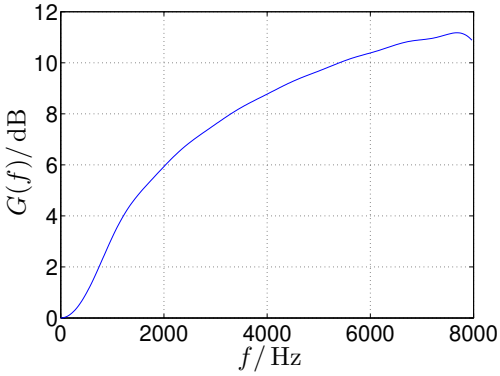



**Figure 8.4:** Gain of a delay-and-sum beamformer in broadside configuration ($\theta_0 = 90°$).

An illustrative description of the directivity of a beamformer is obtained by visualizing the beam pattern in polar coordinates like it is shown in Figure 8.5.

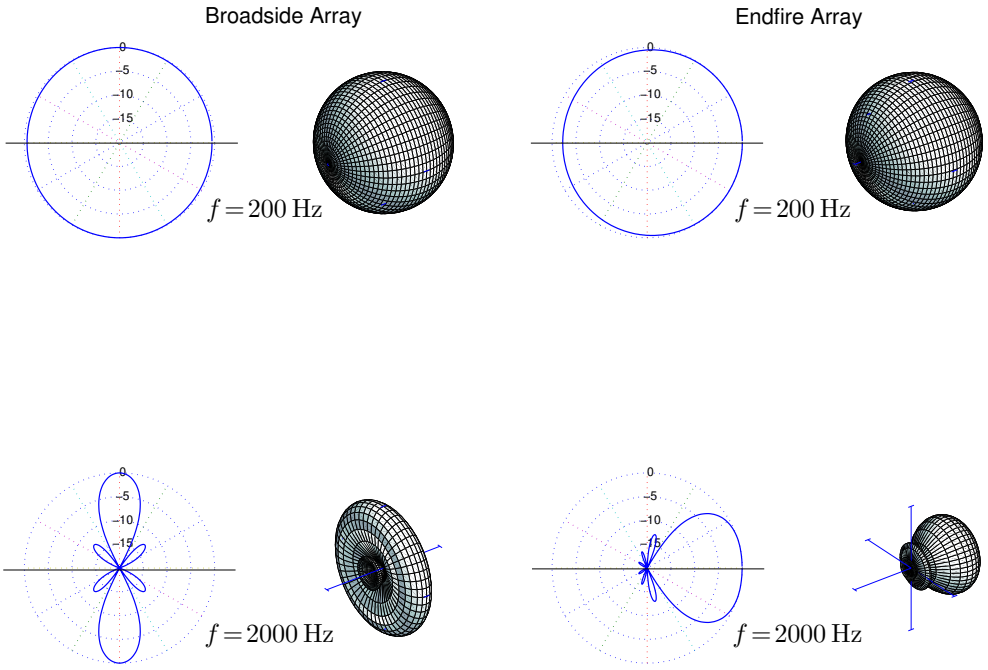



**Figure 8.5:** Beam pattern in logarithmic polar coordinate representation of a delay-and-sum beamformer in broadside configuration ($\theta_0 = 90°$) and endfire configuration ($\theta_0 = 0°$).

It can be observed that the frequency-dependent directivity also strongly depends on how the array is arranged. The broadside array features two main lobes while the endfire array has only a single main lobe. However, this main lobe is wider than it is the case for the broadside array. The rotational symmetry of the directivity is caused, as mentioned previously, by the use of line arrays with omni-directional microphones.

Finally, it should be noted that a variety of approaches exist to compensate for the presented disadvantages of the delay-and-sum beamformer. This can be achieved

by a different selection of the filter $a_n(k)$ (see Figure 8.1). So-called superdirective beamformers maximize the gain of the array. This is done by not - as it is the case for the delay-and-sum beamformer - adding up the in-phase signals. Superdirective beamformers can provide a significantly improved directivity for lower frequencies compared to the delay-and-sum beamformer. However, they also suffer from a higher susceptibility to sensor noise. This issue can be addressed by using beamformers with limited superdirectivity. The delay-and-sum-beamformer presents the limiting case as its susceptibility (to sensor noise) is minimal.

A disadvantage of the discussed beamformer is its frequency dependency as can for example be seen in Figure 8.3. Beamformers with (almost) constant directivity are employed to counteract this although these, on the other hand, require a large number of microphones.

Beside the fixed beamformer, there are also *adaptive* beamformers. These are used for example to systematically suppress noise sources that are not a fixed position. One representative of this class is the so-called generalized sidelobe canceler the transfer function of which features an adaptive zero in the direction of the noise source that it attempts to suppress. For a more detailed discussion of this and other beamformers, refer to the aforementioned literature.

Für eine detaillierte Besprechung dieser und anderer Beamformer muss auf die (eingangs erwähnte) Literatur verwiesen werden.

## 8.2 Exercises

The *delay-and-sum beamformer* (DS BF) is also used for the improvement of signals with acoustic interferences amongst other things due to its simple implementation and high robustness. Before this application is considered in the final laboratory experiment, the properties of the DS BF are analyzed in more detail in this experiment. In particular, the influence of the microphone setup as well as the beamformer configuration (broadside or endfire) will be analyzed. For this purpose, MATLAB offers a large selection of functions to (graphically) evaluate multi-dimensional transfer functions of beamformers.

### Exercise 8.1 Beam pattern of a beamformer

The beam patterns in Figure 8.3 have been calculated with the script `beampattern.m` which you can find in the experiment directory. However, the calculation of the array steering vector is incomplete. Insert the missing argument for the `???`.

Generate the two representations of the beam patterns that are depicted in Figure 8.3. Do so by using the command `surfc` for the 3-dimensional representation and the command `imagesc` for the 2-dimensional visualization. You can modify the properties of the axes with the `set(gca, ...)` command. (More details on these commands are provided by the MATLAB help.)

Find the position in the program where the array configuration is specified. Introduce the variable `theta_0` to the list of simulation parameters to be able to flexibly set the main direction of incidence. Then use the extended script for the calculation of the beam pattern to generate the corresponding plots for an endfire array. Use different figures so that you can compare the plots more easily!

What is the striking difference compared to the broadside array? What effect does the increase of the sensor distance from $d = 4.25\,\text{cm}$ to $d = 20\,\text{cm}$ have?

### Exercise 8.2 Array gain

The gain of the DS BF in broadside configuration is depicted in Figure 8.4. Write a script for the generation of this curve. The script `arraygain.m` should serve you as a template.

Compare this gain with that of an endfire array. How does the gain change upon an increase of the sensor distance from $d = 4.25\,\text{cm}$ to $d = 20\,\text{cm}$?

Directly compare the plots for the gain with the respective beam pattern plots! How can the shape of the gain curve be explained with the beam pattern?

**Note:** The calculation of the beam pattern can be adopted from the `beampattern.m` script. The denominator in equation (8.21) can be calculated in a similar fashion for the identification of the gain.

**Exercise 8.3 Comparison of the beamformers**

Which configuration of the beamformer would you recommend for a noise reduction system? Discuss the advantages and disadvantages of your proposal!

# Chapter 9

# Noise Reduction by Beamforming

In this experiment, the application of the previously examined delay-and-sum beamformer (DS BF) to noise reduction will be assessed. Multi-channel noise reduction systems are used for example in hands-free setups in cars or telephone conference systems. Their use usually allows for improved results compared to single-channel techniques at the cost of increased hardware requirements and computational effort.

The aim of this experiment is to convey the fundamental principal of multi-channel noise reduction systems. By means of instrumental measures, the properties of the DS BF for different sound fields will be analyzed. MATLAB allows the simple realization of a corresponding simulation environment and the visualization of the results.

## 9.1 Principle

Nowadays, many cell phones as well as wireless telephones possess a hands-free system, for example to make it possible for the driver in a car to make a call during the ride. An inherent problem is that the useful signal is commonly superimposed by other, interfering sounds. The aim of the *noise reduction system* is to (partly) free the useful signal of interferences. This facilitates the communication with the far speaker and proves to be advantageous for speech encoders that are employed in mobile phones.

Interferences can occur in countless ways. Engine, tire and wind noise generated outside of the vehicle interior can approximately be considered as diffuse (undirected) interferences. Examples for nearly coherent (directed) interferences are loudspeaker sounds.

A technique to at least partly free the useful signal of interferences is to use a beamformer like it is shown in Figure 9.1. The noise reduction of the beamformer is caused by its spatial directivity whereby signals from directions other than that of the useful signal source are attenuated. It immediately follows from the discussion of the DS BF in Section 8.1.3 that a full suppression of broadband noise sources cannot be expected.

In this experiment, it will be analyzed more closely how good the noise reduction that is achieved by the DS BF still is. For this purpose, the introduction of an appropriate signal model as well as suitable quality measures is necessary. Based on this, a simulation program can be realized with MATLAB.

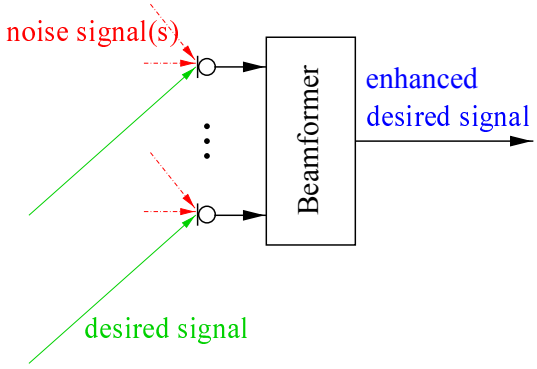



**Figure 9.1:** Principle of noise reduction by beamforming.

## 9.2 Signal Model

To analyze a multi-channel noise reduction system, a signal model will be established first. This model defines how the microphone signals will have to be generated for the simulation later on.

Beside the useful signal $s(t)$, $M$ coherent, interfering signals $e_m(t)$ might arrive at a microphone of the array. Moreover, diffuse noise $d_n(t)$ can occur at each of the microphones. Thus, for the respective received microphone signal, an additive superposition of these signals according to

$$x'_n(t) = s'(t - t_n) + \sum_{m=1}^{M} e_m(t - t_{m,n}) + d_n(t) \quad ; \quad n = 1,2,\ldots,N \tag{9.1}$$

follows.

Here, $t_n$ denotes the signal delay between the signal source and the $n$-th microphone and $t_{m,n}$ the corresponding signal delay for the $m$-th coherent noise source. For the following considerations, a simpler model will be used for which the interfering signals at one sensor are combined to a single signal $v'(t)$

$$x'_n(t) = s'(t - t_n) + v'_n(t). \tag{9.2}$$

After A/D conversion and ideal delay compensation, the signals are given in the form

$$x_n(k) = s(k) + v_n(k - k_n). \tag{9.3}$$

The used notation and applied assumptions are consistent with those of the previous Chapter 8.) The DS BF covered in Section 8.1.1, the filter of which is given by equation (8.17), yields the output signal

$$y(k) = s(k) + \frac{1}{N} \sum_{n=1}^{N} v_n(k - k_n). \tag{9.4}$$

The filtering of the DS BF, in the ideal case considered here, does not result in any distortion of the useful signal. This conclusion could already be drawn from the beam patterns in Figure 8.3. The reduction of the noise power is a result of

the interfering signals that are not in-phase when they are summed up (destructive interference).

To describe the achieved noise reduction analytically, the *coherence function* of the sound field must be known. For a series of sound fields, an analytical description exists. In Section 8.1.2, the coherence function of sound fields was already used to determine the gain of the array.

It should be noted that for the presented *modeling*, a series of assumptions have been made. Among these is that an ideal estimation of the incident angle of the useful signal source $\theta_0$ has been taken for granted and the microphones have been considered ideal (see also Section 8.1.1). Additionally, signal distortions due to sound reflections (room reverberation) have been neglected. These assumptions are (at most) approximately valid in practice.

The reader can find a more detailed discussion of these aspects in the aforementioned literature amongst others.

## 9.3 Instrumental Quality Measures

A very important criterion for the assessment of the quality of a speech improvement system is the auditory impression. The auditive evaluation of speech samples by as many test subjects as possible is therefore desirable. However, since such tests usually require a lot of effort, instrumental quality measures are required. The results of these should obviously correlate with the subjective auditory impression as well as possible. Still, the evaluation of a system for speech improvement should always include an auditory evaluation using of audio samples beside the instrumental evaluation.

The principle for the instrumental evaluation of noise reduction systems is shown in Figure 9.2. The following consideration refers to multi-channel noise reduction

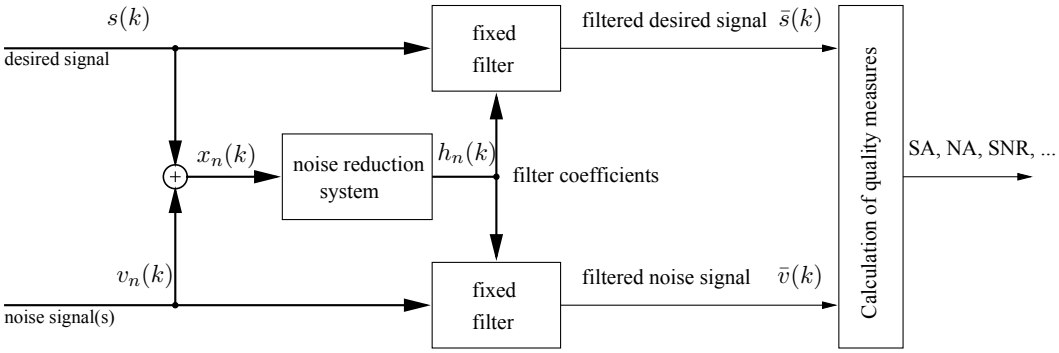



**Figure 9.2:** Calculation of instrumental quality measures for noise reduction systems.

systems and includes single-channel processing as a special case. For the unfiltered interference, the signal from an arbitrary reference microphone $v_r(k)$ is assumed.

In the simulation, the interfering signal $v_n(k)$ that arrives at one sensor as well as the speech signal $s(k)$ are separately available. The noisy signals

$$x_n(k) = s(k) + v_n(k) \quad ; \quad n = 1,2,\ldots,N \tag{9.5}$$

are obtained from the additive superposition in case of an ideal delay compensation for the useful signal. The filtering with the $N$-channel noise reduction system yields the output signal

$$\begin{aligned} y(k) &= \sum_{n=1}^{N} x(k) \star h_n(k) = \sum_{n=1}^{N} \big(s(k) + v_n(k)\big) \star h_n(k) \\ &= \underbrace{\sum_{n=1}^{N} s(k) \star h_n(k)}_{\bar{s}(k)} + \underbrace{\sum_{n=1}^{N} v_n(k) \star h_n(k)}_{\bar{v}(k)} . \end{aligned} \tag{9.6}$$

The signals $\bar{s}(k)$ and $\bar{v}(k)$ are the filtered speech and noise signals respectively. In the simulation, these signals are obtained by separately filtering the speech and interference signals only. This principle also applies to systems with *adaptive* coefficients.[1] Obviously, the used filter coefficients $h_n(k)$ are always those that are also used for the filtering of the disturbed signals $x_n(k)$. With the signal $\bar{s}(k)$, the evaluation of the *useful signal distortion* is performed while the signal $\bar{v}(k)$ is used to assess the *noise suppression*. The instrumental evaluation can be made in the frequency or in the time domain.

An example for an instrumental evaluation in the time domain is the (undesired) attenuation of the useful signal, the *Speech Attenuation (SA)*. It can be determined with

$$\mathrm{SA} \,/\, \mathrm{dB} = 10 \log_{10} \left( \frac{1}{\mathcal{C}(F_s)} \sum_{\lambda \in F_s} \frac{\sum_{i=0}^{M-1} s^2(\lambda M + i)}{\sum_{i=0}^{M-1} \bar{s}^2(\lambda M + i)} \right) . \tag{9.7}$$

The calculation is performed for frames of $M$ values that are shifted by $\lambda\, M$ samples. For integer values of $\lambda$, there is no overlapping of the frames. Only those frames are are included in the calculation in which a so-called voice activity detection (VAD) has identified that the speaker is active. In principle, the short-term energy of each frame of the speech signal $s(k)$ is considered. If this value is above some threshold, speech activity is indicated and vice versa. The received ratios of the short-term energies are then averaged over all frames with speech activity ($\lambda \in F_s$). The expression $\mathcal{C}(F_s)$ denotes the number of frames with speech activity. $F_s$ represents the set of all frame indices for segments with speech activity. A distortion of the speech signal is present if SA$\neq 0\,$dB.

Analogously to the speech attenuation, the attenuation of the noise signal level, the *Noise Attenuation (NA)* can be defined

$$\mathrm{NA} \,/\, \mathrm{dB} = 10 \log_{10} \left( \frac{1}{\mathcal{C}(F)} \sum_{\lambda \in F} \frac{\sum_{i=0}^{M-1} v_r^2(\lambda M + i)}{\sum_{i=0}^{M-1} \bar{v}^2(\lambda M + i)} \right) . \tag{9.8}$$

Here, $v_r(k)$ is the noise at the reference microphone. Typical parameters - *that should also be used in the experiment to come* - are $M = 256$ and $\lambda = 0{,}1{,}2, \ldots$, i. e.

[1] For example, the DF BF has variable coefficients for the delay compensation of moving useful signal sources. However, this case will *not* be considered in the upcoming experiments.

non-overlapping frames. In contrast to the SA, all frames are now included in the calculation which is expressed by the set $F$. A reduction of the noise level is achieved on average if $NA > 0\,\mathrm{dB}$; for negative dB-values, the noise signal is amplified rather than attenuated.

Noise reduction systems underlie the general trade-off between the useful signal distortion and the noise power suppression: A large reduction of the interfering signals can only be achieved at the cost of an increased distortion of the useful signal and vice versa. This relation is partly accounted for in the *effective noise attenuation (ENA)*

$$\mathrm{ENA}\,/\mathrm{dB} = (\mathrm{NA} - \mathrm{SA})/\mathrm{dB}. \tag{9.9}$$

However, the speech attenuation (SA) is of no relevance for the considered *ideal* DS BF since, as shown before, there is no speech distortion here.[2]

Another, very universal measure is the *signal-to-noise-ratio* (SNR). It can be calculated in different ways. The *global SNR* is acquired from the ratio of the powers of the speech and the noise signal

$$\mathrm{SNR}\,/\,\mathrm{dB} = 10\log_{10}\left(\frac{\sum\limits_i s^2(i)}{\sum\limits_i v_r^2(i)}\right). \tag{9.10}$$

Beside this, the *segmental SNR*

$$\mathrm{SNR}_{\mathrm{seg}}\,/\,\mathrm{dB} = \frac{1}{\mathcal{C}(F_s)}\sum_{\lambda\in F_s}\left(10\log_{10}\left(\frac{\sum\limits_{i=0}^{M-1} s^2(\lambda M+i)}{\sum\limits_{i=0}^{M-1} v_r^2(\lambda M+i)}\right)\right) \tag{9.11}$$

is also commonly applied. In the calculation of it, only the frames (segments) with speech activity are included, because of which the value might possibly significantly deviate from that of the global SNR calculation which takes all frames into account.

It should be noted that the introduced measures can also be applied to single-channel systems. Beside the instrumental measures presented here, a large variety of other evaluation measures exist that will not be covered in more detail here however. In the following exercises, we will restrict ourselves to the global SNR and the noise attenuation to merely illustrate the *principle* of the instrumental evaluation.

[2] This does not apply to real systems or special beamformers like e. g. the Griffith-Jim beamformer.

## 9.4 Exercises

In this experiment, the delay-and-sum-beamformer (DS BF) will be used for noise reduction. The achieved performance will be evaluated with instrumental measures as well as the auditory impression. Different scenarios will be considered and compared with each other.

### Exercise 9.1 Setup of the simulation environment

In your working directory, you will find the MATLAB script `noisered_sim.m` which should serve as a framework for the following exercises.

In the first part of the script, all required simulation parameters, such as the number of microphones, the microphone distance etc., should be set. Afterwards, the $N$ input signals for the beamformer are generated. In the third section, the filtering of the signals is performed. Beside the actual output signal $y(k)$, the signals $\bar{s}(k)$ and $\bar{v}(k)$ must also be generated as pointed out in Section 9.3. With these sequences, the employed instrumental measures can be evaluated and the results can be displayed graphically.

The sampling frequency of the audio signals is 8 kHz. The number of microphones for the *broadside array* is $N = 8$ with a distance of $d = 4.25\,\text{cm}$ between them.

The matrix `Noise` contains the coherent interference signals that arrive at the microphones.

Calculate the angle at which the coherent noise arrives at the array!

The variable `S` contains the speech signal that is recorded at the 8 microphones. The input signals of the microphone array are obtained by the matrix addition `X=S+Noise`.

Calculate the angle at which the speech arrives at the array!

Perform the noise reduction with the DS BF. **In this and in the following experiments, there is *no* need for a delay compensation. Why is no delay compensation required for a DS BF in this setup?**

Plot the spectrogram for the input signal at the reference microphone and the spectrogram for the output signal in the same figure. The time axis should be in seconds and the frequency axis in Hz! Listen to the signals. How would you subjectively evaluate the auditory impression and the achieved noise reduction?

### Exercise 9.2 Coherent interference with different power

Now, the global SNR of the input signals will be varied. To do so, you need the auxiliary function `scale4snr.m`. This function receives 3 input arguments: the useful signal sequence $s(k)$, the noise signal sequence at the reference microphone $v_r(k)$ and the desired global SNR in dB. The output value of the function is a scaling factor $a$ for the *noise signal* so that the signals $s$ and $a \cdot v(k)$ have the specified global SNR (in dB). (This MATLAB function consists of no more than 2 to 3 command lines!)

Use this function to scale the input matrix of the noise signals `Noise` in such a way that the global input SNR per microphone (according to equation (9.10)) respectively

is [-10, -5, 0, 5, 10, 15] dB. This scaling can be applied with the value obtained for one microphone as all given noise signals have almost the same power.

Compute the *noise attenuation* (without frame overlapping) according to equation (9.8) and the *global SNR* at the output of the beamformer after the filtering. Plot both curves in the same diagram as a function of the (global) input SNR values. Ensure that there is a meaningful legend and that the axes are labeled correctly!

### Exercise 9.3 Uncorrelated interference with different power

Repeat the preceding experiment once more with white Gaussian noise as interference signal. Scale it to reach the specified input SNR.

What differences do you observe compared to the instrumental curves from the previous experiment?

### Exercise 9.4 Coherent interference with varying incident angle

The incident angle of the coherent interference should now be $\theta = 0°, 30°, 60°, 90°$ and the input SNR is 0 dB. Load the WAV-file `autonoise.wav`. This signal will now serve as the interference and arrive at the beamformer with the angle $\theta$ (see also Figure 8.2).

Write a function `frac_del.m` (see Section 8.1.1). The input parameters are the input signal that should be delayed and the possibly non-integer delay that corresponds to the incident angle of the interference. The output variable is the delayed sequence. The fractional-delay filter should have 256 coefficients. (The MATLAB function has one to two lines!) First, test your function by taking a look at the input and output signal of your fractional-delay filter to estimate the delay.

Like in the preceding experiments, determine the noise attenuation and the global SNR at the output for the different incident angles of the noise.

What is the difference compared to the curves from the second experiment and how can this be explained?

# Chapter 10

# Spatial Signal Processing

Acoustic events in our everyday-life like music or speech often take place in enclosures (e.g. rooms, concert halls,...). Therefore, the generated soundfield exhibits a complicated three dimensional structure due to reflections. Usually, speech and music are picked up by linear or planar microphone arrays and thereby a large part of the 3D information is lost. Recently, 3D array geometries gained a lot of interest and one specific type, the Spherical Microphone Array, will be investigated in this lab experiment. Furthermore, an elegant mathematical formulation is introduced that allows for efficient signal processing of the multi-channel spatial audio signals.

Spherical Microphone Arrays as the one illustrated in Figure 10.1 and the corresponding mathematics are a versatile tool used in different disciplines in acoustics and audio technology. In the beamforming community, they are popular because they enable rotational invariant beams which can be designed analytically. Decoupling between space and time is achieved via a spherical harmonics transform, which allows for an abstraction of the microphone array. In room acoustics, the technology is popular because it allows for an accurate representation of a sound field. The inverse problem, the synthesis of the sound field using multiple loudspeakers, is in the focus of the spatial audio community. The latter community coined the term *Higher Order Ambisonics* (HOA).

For the realization of the spacial signal processing in MATLAB we will use the concept of object-oriented programming (OOP) in this lab. Therefore, a short introduction to OOP in MATLAB will be given in the Section 10.3. Please refer to the MATLAB help for more information.[1]

[1] https://de.mathworks.com/help/matlab/object-oriented-design-with-matlab.html

**Figure 10.1:** Eigenmike® with 32 microphone capsules uniformly distributed on a rigid sphere

## 10.1 Spherical Harmonics Description of Sound Fields

The goal of this section is to introduce a mathematical description of the soundfield that facilitates applications such as beamforming or spatial rendering. This is achieved by separating dependencies of the soundfield on time (or frequency) from those that depend on space. In Section 10.1.1 we start with a compact and mathematically dense introduction of the spherical harmonics definition in the continuous time and space domain. After that, in Section 10.1.2, we translate this continuous representation to a discrete one by sampling the unit sphere in space. The discrete representation will be used in the exercises in Section 10.4.

Note that the following introduction to spherical harmonics is very short. For a thorough understanding consider the corresponding literature, for example the book by Rafaely et al.[2]

### 10.1.1 Continuous Representation

With the inclination angle $\theta$, the azimuth angle $\phi$ and the Euclidian distance from the origin $r$ we define the spherical coordinate system shown in Figure 10.2. Using the wavenumber $k = \frac{\omega}{c}$ with angular frequency $\omega$ and the speed of sound $c$, the pressure of a sound field $p(k,\theta,\phi,r)$ in the vicinity of the coordinate system's origin can be approximated with the series expansion[3]

$$p(k,\theta,\phi,r) \approx \sum_{n=0}^{N} \sum_{m=-n}^{n} 4\pi \mathrm{i}^n \cdot \underbrace{j_n(kr)}_{\substack{\text{basis function} \\ \text{for frequency} \\ \text{and sphere radius}}} \cdot \overbrace{a_{nm}(k)}^{\substack{\text{expansion} \\ \text{coefficient}}} \cdot \underbrace{Y_n^m(\theta,\phi)}_{\substack{\text{basis function} \\ \text{for sphere surface}}} . \tag{10.1}$$

Here, $j_n(\cdot)$ is the spherical Bessel function of the first kind, i is the imaginary unit. $j_n(\cdot)$ models the influence of the array geometry that is used for the acquisition of the soundpressure and is assumed to be constant for this experiment. $Y_n^m(\theta,\phi)$ are the complex-valued spherical harmonics functions[4]

$$Y_n^m(\theta,\phi) = \sqrt{\frac{2n+1}{4\pi}\frac{(n-m)!}{(n+m)!}} \cdot P_n^m(\cos\theta) \cdot \mathrm{e}^{\mathrm{i}m\phi} \tag{10.2}$$

with the associated Legendre functions $P_n^m(\cdot)$ and factorial $(\cdot)!$. The spherical harmonics $Y_n^m(\theta,\phi)$ of orders $n = 0,1,2,3$ are illustrated in Figure 10.3. The choice of truncation order $N$ and wavenumber $k$ (i.e., the frequency) determines the region where the approximation is accurate.

The actual frequency domain (or strictly speaking: wave domain) coefficients $a_{nm}(k)$ with order $n = 0,\ldots,N$ and degree $m = -n,\ldots,n$ do not depend on $r$ but only on the wavenumber $k$. The magnitude of $a_{nm}(k)$ reflects the amount of energy that

[2] B. Rafaely, Fundamentals of Spherical Array Processing, vol. 8. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[3] See footnote 2

[4] All considerations presented here are also valid for the real-valued N3D normalized SH definition commonly used in the Ambisonics community.

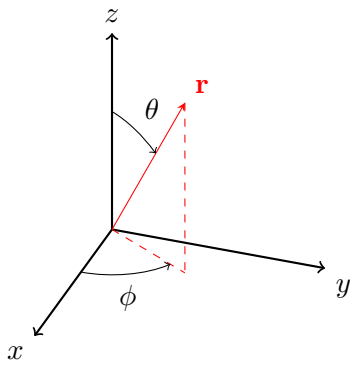



**Figure 10.2:** Coordinate system

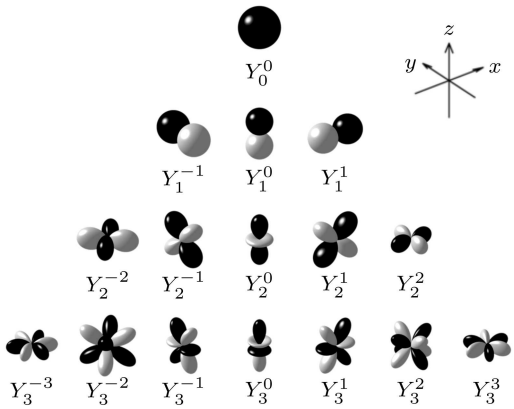



**Figure 10.3:** Baloon plots of spherical harmonics $Y_n^m(\theta,\phi)$ of orders $n = 0,1,2,3$. The radius encodes the absolute value, the brightness indicates the sign (Black positive, gray negative)

comes from the directions specified by the corresponding spherical harmonics function $Y_n^m(\theta,\phi)$ shown in Figure 10.3. The coefficients $a_{nm}(k)$ can also be expressed in time domain, i.e.,

$$a_{nm}(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} a_{nm}(k=\frac{\omega}{c})\cdot \mathrm{e}^{\mathrm{i}\omega t}\ \mathrm{d}\omega, \tag{10.3}$$

and are commonly known as *Higher Order Ambisonics* signals. The signals $a_{nm}(t)$ will be used in the exercises in Section 10.4. In the following, a simplified notation is being used neglecting temporal and frequency dependencies, i.e., $t$ and $k$ are omitted.

The coefficients $a_{nm}$ stand in a direct relation to the directional amplitude density $a(\theta,\phi)$ of the continuum of plane waves impinging into the coordinate system's origin[5],

$$a(\theta,\phi) \approx \sum_{n=0}^{N}\sum_{m=-n}^{n} a_{nm}\cdot Y_n^m(\theta,\phi). \tag{10.4}$$

This relation is enabled by the fact that both, the coefficients $a_{nm}$ and the directional amplitude density $a(\theta,\phi)$, do not depend on $r$.
The inverse relation is given by

$$a_{nm} = \int_{\mathrm{S}^2} a(\theta,\phi)\cdot [Y_n^m(\theta,\phi)]^*\ \mathrm{d}A, \tag{10.5}$$

where $\mathrm{S}^2$ denotes the surface of the unit sphere with the corresponding surface element $\mathrm{d}A = \sin\theta\ \mathrm{d}\theta\,\mathrm{d}\phi$. Eqs. (10.4) and (10.5) are sometimes referred to as the *inverse spherical Fourier transform* and the *spherical Fourier transform*, respectively. It does not matter whether the transformation is done on the time or the frequency domain representation of $a_{nm}$.

### 10.1.2 Discrete representation

In this laboratory we want to use the spherical harmonics decomposition for digital signal processing. Therefore, we have to discretize the actually continuous angles $\theta$ and $\phi$. A spatially discrete representation can be found by sampling the unit sphere with $Q$ sampling points denoted as $\{\boldsymbol{\Theta}_1,\boldsymbol{\Theta}_2,\ldots,\boldsymbol{\Theta}_Q\}$ where $\boldsymbol{\Theta}_q = (\theta_q,\phi_q)$. One example for $Q = 121$ is shown in Figure 10.4.

We can then evaluate the inverse spherical Fourier transform in Eq. (10.4) for the finite set of directions $\boldsymbol{\Theta}_q$. This results in the matrix notation,

$$\underbrace{\begin{pmatrix} a(\boldsymbol{\Theta}_1)\\ a(\boldsymbol{\Theta}_2)\\ \vdots\\ a(\boldsymbol{\Theta}_Q)\end{pmatrix}}_{=\boldsymbol{a}} = \underbrace{\begin{pmatrix} Y_0^0(\boldsymbol{\Theta}_1) & Y_1^{-1}(\boldsymbol{\Theta}_1) & Y_1^0(\boldsymbol{\Theta}_1) & \cdots & Y_N^N(\boldsymbol{\Theta}_1)\\ Y_0^0(\boldsymbol{\Theta}_2) & Y_1^{-1}(\boldsymbol{\Theta}_2) & Y_1^0(\boldsymbol{\Theta}_2) & \cdots & Y_N^N(\boldsymbol{\Theta}_2)\\ \vdots & \vdots & \vdots & & \vdots\\ Y_0^0(\boldsymbol{\Theta}_Q) & Y_1^{-1}(\boldsymbol{\Theta}_Q) & Y_1^0(\boldsymbol{\Theta}_Q) & \cdots & Y_N^N(\boldsymbol{\Theta}_Q)\end{pmatrix}}_{=\boldsymbol{Y}} \cdot \underbrace{\begin{pmatrix} a_{0,0}\\ a_{1,-1}\\ a_{1,0}\\ a_{1,1}\\ \vdots\\ a_{N,N}\end{pmatrix}}_{=\boldsymbol{a_{nm}}}. \tag{10.6}$$

[5] B. Rafaely, Fundamentals of Spherical Array Processing, vol. 8. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

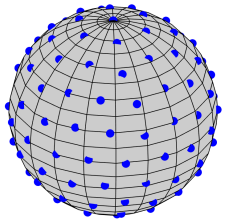


**Figure 10.4:** $Q = 121$ sampling points (blue) on the unit sphere (gray).

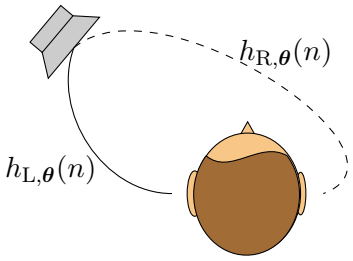



**Figure 10.5:** Illustration of head-related impulse response from a sound source in direction $\boldsymbol{\Theta}$ to left and right ear.

The spherical harmonics coefficients in $\boldsymbol{a_{nm}}$ are arranged in a sequential order such that each $a_{nm}$ is stored as the $\chi$-th element with

$$\chi = n^2 + n + m + 1. \tag{10.7}$$

Eq. (10.6) can be referred to as the *discrete inverse spherical Fourier transform*. The spherical harmonics matrix $\boldsymbol{Y}$ can be used to easily design a beamformer, that looks in a specific direction.

## 10.2 Binaural Rendering

Humans are able to localize sound events by processing sounds received through their two ears. Sound from a source positioned to the left of the listener's head will arrive at the left ear before it will arrive at the right ear (cf. Figure 10.5). Additionally, the sound pressure at the left ear will be higher than the sound pressure at the right ear. These phenomena are known as binaural cues and are referred to as *Interaural Time Difference* (ITD) and *Interaural Level Difference* (ILD), respectively.

These binaural cues (and other cues) are captured by the *Head-Related Transfer Function* (HRTF). It describes how the sound is modified when it travels from a source position to an ear of the listener. The HRTF can be transformed to the time domain to obtain the corresponding *Head-Related Impulse Response* (HRIR). The HRIR will be used in this experiment and we define the set of HRIRs $h_{l,\boldsymbol{\Theta}}(n), l \in \{\mathrm{L}, \mathrm{R}\}$ from direction $\boldsymbol{\Theta}$ to the left and right ear, respectively.

HRIRs can be used to synthesize headphone signals that will create the illusion of a sound from direction $\boldsymbol{\Theta}$. These so-called binaural signals are obtained separately for

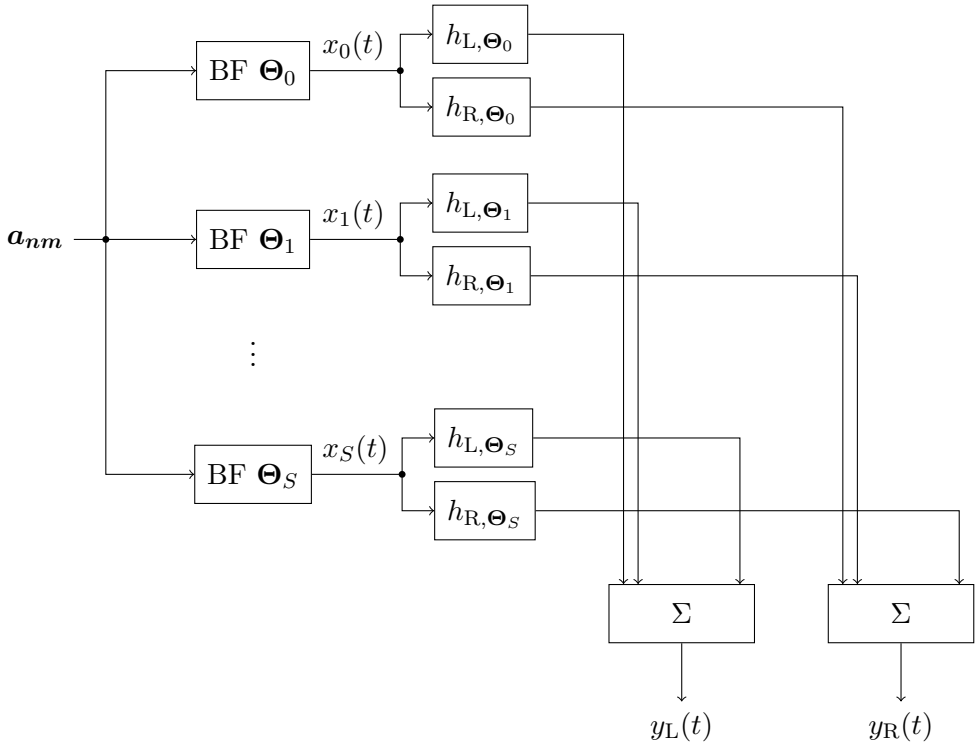



**Figure 10.6:** Block diagram of HOA-to-binaural renderer for $S + 1$ loudspeakers.

each ear by convolving the source signal $x(n)$ with the HRIR for the corresponding ear $l$ and direction $\boldsymbol{\Theta}$:

$$y_l(n) = x(n) * h_{l,\boldsymbol{\Theta}}(n). \tag{10.8}$$

Listening to theses signals using headphones will create the illusion of an actual sound source from direction $\boldsymbol{\Theta}$.

The same principle can be used to reproduce spatial audio signals in *Higher Order Ambisonics* representation. Virtual loudspeakers are quasi-uniformly placed on a sphere and their signals are calculated using a beamformer for each direction where a virtual loudspeaker is placed. Then, each loudspeaker signal is filtered with the HRIR that corresponds to the direction of the virtual loudspeaker. Finally, the filtered loudspeaker signals are summed up separately for the left and right ear to obtain the binaural signal. This process is illustred in Figure 10.6.

## 10.3 Object-Oriented Programming in MATLAB

A straight-forward approach to writing applications is by means of procedural programs. These programs pass data to functions, the functions modify the data and return results. When the scope of the application grows, code quickly gets messy by having to define similar data structures and/or functions.

Object-oriented programming (OOP) is based on the concept of objects that encapsulate data as well as methods. For a number of applications, thinking in objects rather than procedures is more natural. By encapsulating the data together with

methods, only relevant methods can be applied to the data. This improves the readability of code and is less error-prone.

Objects can interact with each other by interface functions. These interface functions are well defined and segregate the important information, that is required to use a certain object, from unimportant information, that may internally be required but is of no need for the user. This again increases the readability of code by locking unimportant information in the object.

Class-oriented programming is a style of OOP, where a class serves as a blueprint from which objects are created. An object that is created from a class blueprint is called an instance of the class. A class can define properties and methods as well as respective access modifiers that every instance of the class obtains. Access modifiers are useful to protect data from misuse. For example, by making a class property `private` the user is not allowed to change the value of this property of existing class instance.

Classes can inherit properties and methods from base classes. This increases the reusability of code by sharing base class features across different sub classes. Inheritance can also be used to implement the same function differently for different sub classes. Therefore, the function interface, defined by its inputs and outputs, does not change but the function body differs for every sub class. This can be useful, if modularity and flexibility is desired.

When approaching a problem with OOP, the following consecutive steps my be useful as a guideline:

1. Identify the components of a system
2. Analyze similarities and differences of components
3. Define a class hierarchy
4. Define the class interfaces

### 10.3.1 Class Definition

In MATLAB, a class definition starts with the `classdef` keyword followed by the class name and enclosed with the `end` keyword. The following code snippet defines the class `BasicClass` with a numeric property called `value`, a constructor and a method `showValue(obj)` that prints the current value to the console.

```
classdef BasicClass
    properties
        value {mustBeNumeric}
    end

    methods
        function obj = BasicClass(value)
            if nargin > 0
                fprintf('My value is %f, nice to meet you!\n', value);
                obj.value = value;
            else
                obj.value = 1;
            end
        end

        function showValue(obj)
            fprintf('My value is %f\n', obj.value);
        end
    end
end
```

#### Constructor and Destructor

We instantiate an object from a class blueprint by calling the class constructor. The constructor is a `public` function that has the same name as the class. For our `BasicClass` the constructor is defined by:

```
function obj = BasicClass(value)
    if nargin > 0
        fprintf('My value is %f, nice to meet you!\n', value);
        obj.value = value;
    else
        obj.value = 1;
    end
end
```

In this example, the constructor receives a `value`, prints out a message, assigns `value` to the object's property `obj.value` and returns the object. Note that the constructor assigns a default value of `1` to `obj.value` if no `value` is specified. It

is generally considered good practice to implement a check for zero arguments, e.g., with `if nargin > 0`. When populating an array with class instances, the constructor is called with no arguments. The constructor can return multiple arguments, but the object needs to be the first argument.

```
>> obj = BasicClass(1.0)
My value is 1.000000, nice to meet you!

obj =

  BasicClass with properties:

      value: 1
```

The destructor is called when an object is deleted, e.g., by calling `clear obj` in MATLAB. A destructor only exists for handle classes and is implemented via the `delete` function. A handle class inherits from the MATLAB `handle` class. To implement a destructor for our `BasicClass` we have to rewrite the class definition to:

```
classdef BasicClass < handle
    ...
end
```

We then add the destructor by implementing the `delete` function.

```
function delete(obj)
    fprintf('My value was %f, goodbye!\n', obj.value);
end
```

Now, if we create and clear an instance of the `BasicClass` we first call the constructor and then the destructor.

```
>> obj = BasicClass(1);
My value is 1.000000, nice to meet you!
>> clear obj
My value was 1.000000, goodbye!
```

### Access to Properties and Methods

We can access properties and call methods of an instance with the dot operator, provided that these properties and methods are `public`.

```
>> obj = BasicClass();
>> obj.value = 2;
>> obj.showValue()
My value is 2.000000
```

### Properties

Class properties can be defined in conjunction with a size, a data type, validation functions and default values. In the following example, we create a property `coord3D` that represents a 3D coordinate.

```
properties
    % Name,   size,  data type, validation functions,      default value
    coord3D   (1, 3) double     {mustBeReal, mustBeFinite} = [0, 0, 0]
end
```

It is a `1x3` vector of doubles that is initialized as `coord3D = [0, 0, 0]`. The validation functions validate if the values that we assign to the coordinate are real, finite numbers. The following expression for an object `obj` with the property `coord3D` would therefore lead to an error.

```
>> obj.coord3D = [1, 2, 3 + 1i];
Error setting property 'coord3D' of class
'BasicClass':
Value must be real.
```

### Access Modifiers

Access modifiers are useful to protect data from misuse. In contrast to programming languages such as *C++* or *Java*, MATLAB has a variety of access modifiers with again different options. Some of the most important modifiers, similar to those in, e.g., *C++*, are `Access, GetAccess, SetAccess`. The options for these modifiers are `private, protected` and `public`. Properties or methods with `private` access can only be accessed from class member functions, `protected` properties or methods can only be accessed from class or sub class member functions and the access for `public` properties or methods is unrestricted. We define the access modifiers right after the beginning of the `properties` (or `methods`) section.

```
properties (Access = private)
    password
    creditCardNumber
end

properties (Access = public)
    bankName
end
```

## 10.3.2 Methods

Methods play a substantial role in object oriented programming as they provide an interface to the object and allow for manipulation of data. Methods that are callable by instances of an object should always have the object as their first argument. Methods are callable using the function syntax or dot notation as the following example highlights:

```
>> obj.showValue()
My value is 2.000000
>> showValue(obj)
My value is 2.000000
```

The next sections introduce some of the most important types of methods.

### Get and Set Methods

In MATLAB we can define explicit getter and setter functions for properties. This is useful to verify the values that we try to assign to a property.
We define getters and setters by implementing the function `get.propertyName()` and `set.propertyName(newValue)`.

The getter function for a property `value` might look like the following.

```
function currentValue = get.value(obj)
    fprintf('Getting value %f\n', obj.value);
    currentValue = obj.value;
end
```

Similarly we define the setter function.

```
function set.value(obj, newValue)
    fprintf('Setting value to %f\n', newValue);
    obj.value = newValue;
end
```

During runtime, these functions are called automatically as soon as a property of the class is read or changed. However, it is not necessary to implement these methods for every property, if we are only interested in reading and writing the properties.

```
>> obj.value = 2;
Setting value to 2.000000
>> x = obj.value;
Getting value 2.000000
```

Note that we can only call the getter or setter functions if the property is `public`. Otherwise, we would have to write an explicit `public` setter function. Remember that `private` properties can only be changed by functions of class members.

```
function setValue(obj, newValue)
    fprintf('Explicitly setting value to %f\n', newValue);
    obj.value = newValue;
end
```

### Static Methods

Static methods are associated with a class, but not with a specific instance of a class. Therefore, you can call a static method without creating an object of the class. Also, static methods don't feature the object as their first argument. A static method can for example be used to calculate values that are required to initialize a class instance. We define static methods by adding the `Static` specifier right after the beginning of the `methods` section. The following static function returns the value of $\pi$.

```
methods (Static)
    function y = getPi()
        y = pi;
    end
end
```

We call the static function `getPi` of the class `BasicClass` by writing:

```
>> p = BasicClass.getPi()

p =

    3.1416
```

### Operator Overloading

MATLAB classes allow for operator overloading. Operators are generally non-alphanumeric characters such as `+, -, *, /, <, &` that define arithmetic opera-

tions. In order to apply these arithmetic operators to instances of a class, we need to attach a meaning to the operators for the respective class. MATLAB defines methods that we need to implemented in order to overload operators. The following code example overloads the binary addition operator + for our `BasicClass` by implementing the `plus` method.

```
function objOut = plus(objIn1, objIn2)
    objOut = BasicClass(objIn1.value + objIn2.value);
end
```

By applying the + operator, we create a new class instance and initialize its `value` as the sum of the values of right-hand side operands.

```
>> a = BasicClass(1);
>> b = BasicClass(2);
>> c = a + b

c =

  BasicClass with properties:

      value: 3
```

### 10.3.3 Abstract Class

An abstract class defines properties and methods that are common to a group of classes. The implementation of these methods is unique to each sub class, as the abstract class does not define its methods. Abstract classes can not be instantiated and exclusively serve as a base for a group of classes. In MATLAB `classdef`, `properties` and `methods` can come with the `Abstract` specifier. Properties and methods that are declared as abstract need to be redefined by all sub classes. If only the class definition is abstract neither its properties or methods need to be redefined. This is illustrated in the following example:

```
classdef (Abstract) Animal
    properties
        age (1, 1) double {mustBeNonnegative}
    end

    methods (Abstract)
        move(obj)
    end

    methods
        function be(obj)
            fprintf('I am a %d year old ', obj.age);
        end
    end
end
```

The abstract class `Animal` serves as a base class. It has a non-abstract property `age` that is non-negative, an abstract method `move` as well as a non-abstract method `be`. A sub class has to implement the method `move` but does not need to redefine `age` or implement `be`. In this context, it makes sense that we can not instantiate an object of type `Animal`.

### 10.3.4 Inheritance

Classes can inherit functionality from base classes. Base classes do not necessarily have to be abstract classes. This increases the reusability of code and implicitly performs a meaningful classification. The following example introduces the class `Dog` that inherits from the abstract class `Animal` from earlier:

```
classdef Dog < Animal
    properties
        name
    end

    methods
        function obj = Dog(age, name)
            obj.age  = age;
            obj.name = name;
        end

        function move(obj)
            if obj.age > 10
                fprintf('%s is pacing.\n', obj.name);
            else
                fprintf('%s is running.\n', obj.name);
            end
        end

        function be(obj)
            be@Animal(obj);
            fprintf('dog named %s.\n', obj.name);
        end
    end
end
```

The class dog adds a property `name`, implements the method `move` and overloads the method `be`. Note that in the function `be` of the class `Dog` we call the function `be` of the base class `Animal` by using the `@` operator with the respective class name. This creates the following behavior:

```
>> susi = Dog(1, 'Susi');
>> susi.be
I am a 1 year old dog named Susi.
```

The classes we covered so far are called value classes. If a variable associated with an instance of an object is reassigned, MATLAB creates an independent copy of the original object. Another type of class is the handle class. If a class inherits from the class `handle`, we can call it by reference rather than value. A variable associated with an instance of a handle class stores a reference to the instance rather than the instance itself. The handle can be assigned to multiple variables which can then modify the same underlying data. The following example illustrates this behavior. If we use the simple value class `BasicClass`, make a copy `b = a` and change the value of `b`, we see that the value of `a` is unchanged.

```
>> a = BasicClass(1) % Not inherited from handle

a =

  BasicClass with properties:

    value: 1

>> b = a;       % Create copy of a
>> b.value = 2; % Change value of b
>> a

a =

  BasicClass with properties:

    value: 1    % Value of a did not change
```

Now, if `BasicClass` inherits from `handle`

```
classdef BasicClass < handle
    ...
end
```

and we perform the same commands, we see that the value of `a` changed mutually with the value of `b`.

```
>> a = BasicClass(1) % Inherited from handle

a =

  BasicClass with properties:

    value: 1

>> b = a;       % Create copy of the handle a
>> b.value = 2; % Change value of b
>> a

a =

  BasicClass with properties:

    value: 2    % Value of a did change
```

## 10.4 Exercises

The code in the main file `hoaMain.m` guides you through all exercises. Please make yourself familiar with this file. In all experiments, a Higher Order Ambisonics (HOA) signal $\boldsymbol{a_{nm}}(n)$ is loaded from a WAV-file `signal_HOA_O4_N3D_12k.wav`, where $n$ is the sample index.

**Note:** MATLAB's function `audioread()` arranges the time in columns and channels/coefficients in rows, while the theory, Eq. (10.6), uses column vectors for the coefficients.

### Exercise 10.1

1. Create a class `Direction` which has two scalar member variables `theta` and `phi` in order to store the inclination angle $\theta$ and the azimuth angle $\phi$ (in radians). The constructor shall accept two parameters `theta` and `phi` in order to initialize the class object.
2. Implement the following member functions:
   - `function thetaRad = getThetaInRadians(obj)`
   - `function phiRad = getPhiInRadians(obj)`
   - `function thetaDeg = getThetaInDegrees(obj)`
   - `function phiDeg = getPhiInDegrees(obj)`
3. Also implement `function plot(obj)` which should generate a 2D scatter plot of the given direction with $\phi$ on the x-axis and $\theta$ on the y-axis, i. e. one direction corresponds to one point in the scatter plot.

### Exercise 10.2

1. How can the HOA order $N$ be calculated from the number of channels in the wave file that contains $\boldsymbol{a_{nm}}(n)$ according to Eqs. (10.6) and (10.7)?
2. Fill in the gaps in the constructor of the `HoaSignal` class. The constructor should load the WAV-file containing the HOA signal and set all properties specified in the `HoaSignal` class.
3. What is the easiest way to realize an omni-directional beamformer given a HOA signal? Implement it and listen to its output!

   **Hint:** How does the 3D directivity pattern of this beamformer look like?

### Exercise 10.3

In the following, we would like to store multiple directions $\boldsymbol{\Theta}_q$ with $q = 1,2,\ldots,Q$ by extending the class `Direction` accordingly while maintaining compatibility with scalar directions. Therefore, we create an array of objects where each object stores a single theta and phi. MATLAB class constructors not only allow to construct scalar objects but also to construct whole arrays with an aggregate constructor.

**Note:** A function can be applied to each element of an array with the MATLAB function `arrayfun`.

1. Implement the extension by modifying the constructor of the class `Direction` in a way that it creates an array of `Direction` objects!
2. Make sure that all member functions are still working!

## Exercise 10.4

In this exercise we want to implement a beamformer, that can be steered to arbitrary directions. For this purpose the `HoaSignal` class already has a member method `getBeamformer`, that creates an object of the new class `Beamformer`. Please take into account the code of the new class `Beamformer` that is already given! A class diagram is given in Figure 10.7.

1. Implement the method `beamformSignal` in the `Beamformer` class. The method returns the output signal of a beamformer for a given `HoaSignal` and `Direction` $\boldsymbol{\Theta}$ by calculating $a(\boldsymbol{\Theta})$ according to Eq. (10.6). Make sure that the `Beamformer` object can also process multiple directions at once!
2. Try different steering directions including $\boldsymbol{\Theta} = (90°, 45°)$, $\boldsymbol{\Theta} = (90°, -45°)$. Plot the signals and listen to them.

## Exercise 10.5

In this exercise, a simple acoustic camera shall be created by calculating a steered response power (SRP) map of the given HOA signal. This is done by steering a beamformer to every direction the acoustic camera should look into and applying the beamformer on the HOA signal. The SRP map is then generated by calculating and plotting the power of the Beamformer output over all look directions. For this exercise the code skeleton of the new class `SteeredResponsePowerMap` has already been created and the `HoaSignal` class already has a member method `getSteeredResponsePowerMap`, that creates an object of this new class. The SRP map will be calculated for the HOA signal on a frame-by-frame basis. To understand the frame-wise processing, read the comments in the code of `hoaMain.m` corresponding to exercise 1.5.

**Note:** Make sure the class `SteeredResponsePowerMap` inherits from `Beamformer`.

1. Extend the method `beamformSignal`, that was implemented in the class `Beamformer`, to support beamforming of only one frame of a given `HoaSignal`.
2. Implement the method `generateSrpMap` in the class `SteeredResponsePowerMap` that calculates the signal power of one frame for a grid of directions. Make also use of the `beamformSignal` method.
3. Visualize the result by completing the code skeletons of the methods `initPlot` and `updatePlot`.

Additional task: track the maximum in the SRP map.

## Exercise 10.6

Upto now all played back signals only had one channel. For generating a spatial impression while listening with headphones we need to render the Ambisonics signal to a binaural signal that has two channels. Therefore, a binaural renderer for an Ambisonics signal shall be implemented in the following. Again, a code skeleton for the new class `BinauralRenderer`, which inherits from `Beamformer`, is already given.

1. Implement the constructor, that loads the MAT-File `hrirs_12k.mat`. The MAT-File provides a set of HRIRs together with the corresponding directions $\boldsymbol{\Theta} = (\texttt{theta}, \texttt{phi})$ for $Q = 25$ quasi-uniformly distributed directions on the sphere.
2. Implement the method `renderSignal` which renders a binaural signal from the HOA signal **as described in Section 10.2**. Again, make use of the `beamformSignal` method implemented in the class `Beamformer`.
3. Listen to the binaurally rendered signals with headphones!

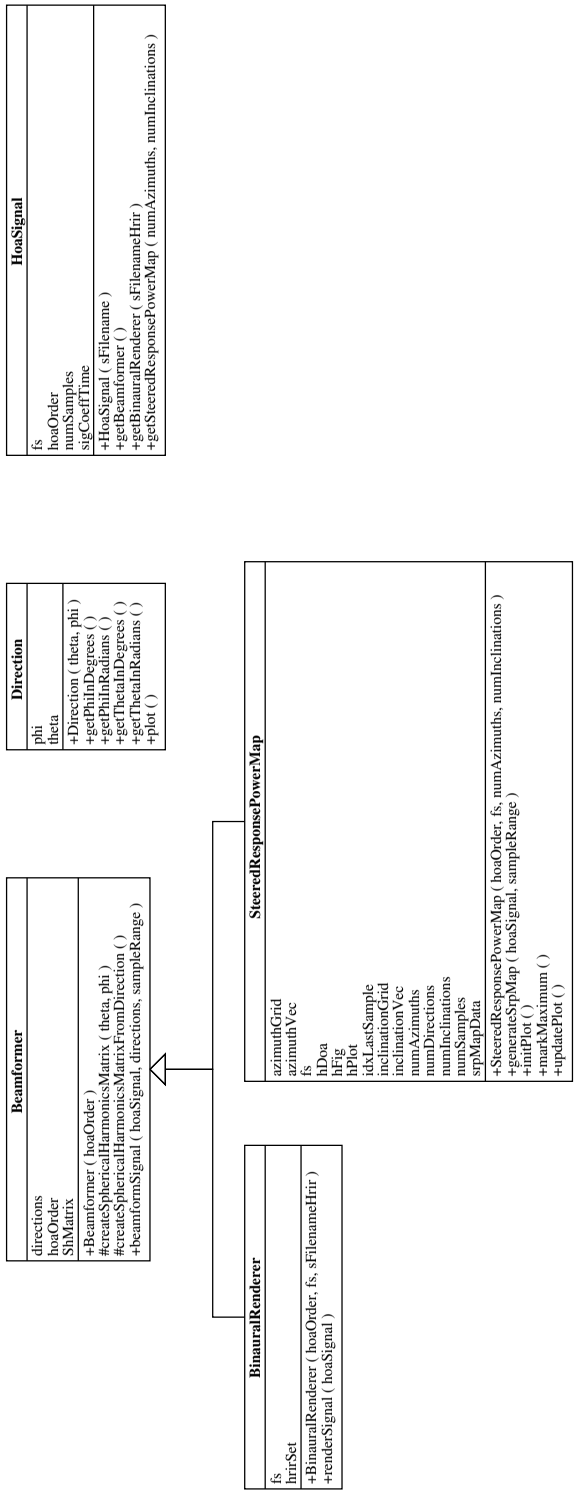



**Figure 10.7:** UML-like class diagram for classes in exercise.