University of Applied Sciences

An Introduction to

Biosignal Analysis

Author: Thomas Haslwanter
email: thomas.haslwanter@fh-linz.at

Version: 6.1
November 22, 2018

This work is licensed under a Creative Commons Attribution-NonCommercial 3.0 Unported License.
# Contents

## 1 Programming Background

1.1 Alternatives to Matlab .......................... 7  
   1.1.1 Python ........................................ 7  
   1.1.2 LabVIEW ....................................... 7  
1.2 Introduction to Matlab ......................... 8  
   1.2.1 Help on the WWW .............................. 8  
   1.2.2 Help Tools in Matlab .......................... 8  
   1.2.3 Text Editors ................................... 9  
   1.2.4 Important Matlab Concepts .................... 9  
1.3 Programming Tips ................................ 13  
   1.3.1 Matlab Documentation .......................... 14  
1.4 The Next Level .................................. 15

## 2 Introduction

2.1 Why do signal processing? ....................... 17  
2.2 What is the basic Procedure? .................... 17  
2.3 Important Topics and Questions .................. 18  
2.4 Conventions and Basics ......................... 19  
   2.4.1 Notation ...................................... 19  
   2.4.2 Mathematical Basis ............................ 19  
   2.4.3 Discrete Signals .............................. 21  
2.5 Exercises ........................................ 22

## 3 Data Filtering

3.1 Transfer Function ............................... 24  
3.2 Finite Impulse Response (FIR) filters .......... 25  
   3.2.1 Matlab Notes .................................. 25  
   3.2.2 DSP Terminology ................................ 25  
   3.2.3 Offline Analysis ............................... 26  
   3.2.4 FIR Filter 1: Moving average ................. 26  
   3.2.5 FIR Filter 2: Differentiation ................. 27  
   3.2.6 FIR Filter 3: Savitzky-Golay Filter .......... 28  
3.3 Infinite Impulse Response (IIR) filters ........ 31  
3.4 Difference between FIR and IIR filters ........ 33  
3.5 Median Filter .................................... 34  
3.6 Filtering images (2-D filtering) ................. 35  
   3.6.1 Representation of Grayscale Images .......... 35  
   3.6.2 Color images .................................. 35  
   3.6.3 2D-Filtering .................................. 35  
3.7 The Next Level .................................. 38  
   3.7.1 Lowess and Loess Smoothing ................. 38
CONTENTS

6.4.2 Line of best fit with no intercept term ....................... 77
6.4.3 Line of best fit with intercept term ....................... 78
6.4.4 Significance ........................................ 78
6.5 Correlation Coefficient and Coefficient of Determination ............... 79
  6.5.1 Correlation Coefficient ................................ 79
  6.5.2 Coefficient of Determination ........................... 79
6.6 Fitting of nonlinear functions .................................. 81
6.7 Exercises .............................................. 83

7 Spectral analysis of biological signals ............................... 85
  7.1 Introduction ........................................... 85
  7.2 Fourier integral ......................................... 87
    7.2.1 Different conventions ................................ 87
    7.2.2 Complex exponential notation ......................... 87
    7.2.3 Fourier transform of a constant ....................... 87
    7.2.4 Fourier transform of a pure sinusoid ................. 88
  7.3 Fourier Series ......................................... 89
    7.3.1 Example: Square Wave ................................ 89
  7.4 Discrete Fourier Transformation .............................. 91
    7.4.1 Fast Fourier Transformation ........................ 91
  7.5 Power Spectrum ......................................... 93
  7.6 Fourier Transform and Convolution .......................... 94
  7.7 The Next Level ......................................... 95
    7.7.1 Short Time Fourier Transform (STFT) ............... 95
  7.8 Exercises .............................................. 96

8 Mechanical Systems ............................................... 97
  8.1 Transfer Functions for Linear Time Invariant Systems ............. 98
    8.1.1 Superpositions ..................................... 99
  8.2 Laplace Transformations and Applications ........................ 101
    8.2.1 Parallel Combination of Transfer Functions: Simple Muscle Model 102
    8.2.2 Serial Combination .................................. 104
  8.3 Bode Diagram ........................................... 105
  8.4 Implementation of Simulations ................................ 106
    8.4.1 Simulink ........................................... 106
    8.4.2 Matlab - Commandline ................................ 106
    8.4.3 Python ............................................ 107

9 Graphical User Interfaces (GUIs) .................................. 109
  9.1 Callbacks .............................................. 109
  9.2 Tips for the User-Interface ................................ 110
  9.3 Matlab Graphics ......................................... 110
    9.3.1 Handles ............................................ 110
  9.4 Layout of Scientific Graphs ................................ 111

A Python ........................................................ 113
  113
  A.1 Examples ................................................. 113
    A.1.1 Example 1: Define Rotation Matrices .................. 113
    A.1.2 Example 2: Select between two choices ............... 114

B Matlab Programs ................................................ 117
C Lecture Schedule 129
Chapter 1

Programming Background

1.1 Alternatives to Matlab

1.1.1 Python

Python is a very-high-level dynamic object-oriented programming language. It is designed to be easy to program and easy to read. It was started in 1980, and has since gained popularity in a broad range of fields from web development, games, usage as a scripting language, and of course science and engineering.

With the sole exception of Simulink, Python can replace Matlab. And while Matlab licences are VERY expensive, Python is completely free - as in free beer! A pretty fair comparison of the two can be found under http://www.pyzo.org/python_vs_matlab.html.

A short introduction to Python is given in the Appendix.

1.1.2 LabVIEW

Developed by National Instruments, LabVIEW is often used for data acquisition and for real-time control systems. After C, it is the most frequently used application for data acquisition. (http://www.ni.com/labview/)
1.2 Introduction to Matlab

Matlab is a great environment for solving many practical problems you will encounter in your work-life. Since it is also very widely used in practice, I will use it consistently for the exercises accompanying this lecture. Nevertheless, don’t forget that Matlab also has disadvantages, and that there are also other good and helpful tools for data analysis. The main disadvantages of Matlab are, in my opinion: i) its high price, and ii) its somewhat outdated programming structure, which is inherently functionally oriented and not object oriented.

1.2.1 Help on the WWW

The following locations are very helpful for getting started with Matlab:

- The complete documentation of all toolboxes, including the often highly recommendable printed documentation (PDF format), can be found at http://www.mathworks.com/access/helpdesk/help/helpdesk.shtml.
- If you have not worked with Matlab before, the Matlab Introduction is a good point to start (http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml).
- At the home-site of Matlab you can also find under MatlabCentral free programs from other Matlab users, which address many problems you will encounter with your work.
- A very recommendable guide to good programming style is Matlab Programming Style Guidelines (by Richard Johnson). While it focuses on Matlab programming, most points are also valuable for programming in other languages.

In addition, on the Moodle site for this course you find ”matlab_tutorial.pdf”, a short introduction to Matlab (comparable to the Matlab Introduction mentioned above).

1.2.2 Help Tools in Matlab

These are REALLY IMPORTANT !!!

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ver</code></td>
<td>Tells you which version and what toolboxes are installed.</td>
</tr>
<tr>
<td><code>help</code></td>
<td>Like helpwin, but does not open a new window.</td>
</tr>
<tr>
<td><code>helpwin /name/</code></td>
<td>If name is empty, this lists the different help-topics. If name is the name of a toolbox, it lists the commands in that toolbox. If name is a command, it lists the first help-block for that command.</td>
</tr>
<tr>
<td><code>doc /name/</code></td>
<td>If name is empty, it starts the help-browser. If name is a command, it lists the HTML-help for that command.</td>
</tr>
<tr>
<td><code>which /name/</code></td>
<td>Tells you if name is a variable or a file (and its location)</td>
</tr>
<tr>
<td><code>name -TAB</code></td>
<td>Tells you which commands start with name (very helpful!)</td>
</tr>
</tbody>
</table>

I typically start out with `name-TAB`, to check if I might be able to guess the command. If I find a likely command, I use `doc` to find out the details; otherwise, I use `helpwin` to narrow down my search. Then I copy an example from the documentation to my command window, and modify it there as I need it.
In addition, the integrated Help-browser (opened by clicking on $fx >>$ at the beginning of the command line) offers help and information access.

In addition, to improve your code you can

- Check the warnings and errors that are indicated on the right hand border of the Matlab Editor.
- Use the M-Lint code checker to detect syntactic errors or conspicuous code segments.

Also, remember that you should not re-invent the wheel when you don’t have to. There is lots and lots of Matlab code freely available. A very good place to start is http://www.mathworks.com/matlabcentral/.

### 1.2.3 Text Editors

#### General Editors

In general, you should select one editor that you use by default for your writing and editing tasks, and make sure that you know this one editor well. Personally, I have decided to use vim, a very powerful, keyboard-oriented editor. If you prefer slower but simpler mouse-based input, check out the free http://notepad-plus-plus.org/.

Either way, make sure that you know your editor well!

#### The Matlab Editor

You are going to be using the Matlab editor a lot, so make sure that you are very comfortable with the most important keyboard shortcuts:

- **F5** Run the current program.
- **F9** Execute currently marked code.
- **F12** Inserts a break-point at the current location.
- **CTRL+A CTRL+I** Properly indents all your loops. This really helps to find mistakes in ending if, for, or while blocks.
- **CTRL+R** Comment out the marked code.
- **CTRL+T** Remove comment sign for marked code.

Also, make sure you learn how to use the debugger. It will help you massively to get your programs running more quickly.

### 1.2.4 Important Matlab Concepts

#### Workspace

The Matlab Workspace contains all the variables the Matlab currently knows, and can be checked with either the command `who` or `whos`. For example, if you type

```matlab
clear;
time = 0:0.1:10;
x = sin(time);
```

then your current workspace contains only the elements `time` and `x`:
CHAPTER 1. PROGRAMMING BACKGROUND

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>1x101</td>
<td>808</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>1x101</td>
<td>808</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

Scripts and Functions

**Scripts** When you start Matlab up, you are in the *Base Workspace*. There you can execute commands either by typing them in, or by putting them into a file and typing the filename. For example, if you put the three lines

```matlab
clear;
time = 0:0.1:10;
x = sin(time);
```

into a file called *myScript.m*, you can achieve the same result as above by simply typing *myScript* on the command line. Such a file is called a *Matlab script*, and all the commands in the script are executed just as if you would type them in, and the base workspace has the same elements as in the example above.

**Functions** Now let’s add an additional line starting with the word *function*, and save the file with the name *myFunction.m*. This file now contains the lines

```matlab
function myFunction()
clear;
time = 0:0.1:10;
x = sin(time);
```

If you now type *myFunction* you will find that - your base workspace is empty! The reason is that Matlab executes each function in its own, new workspace. All the variables that you need you have to provide as Input, and all the variables that you want to keep from this function you have to keep as Output. Note that all other variables are deleted when the function closes. For example,

```matlab
function myOutput = myFunction(myInput)
    % Display a comment
    infoText = 'Just calculating the sine for you.';
    disp(infoText);
    % Calculate the sine of the input
    myOutput = sin(myInput);
```

would take the values "myInput", and return the values "myOutput". (Lines starting with "%" are comments, and have no effect on the calculations.)

For example, you can call this function as follows:

```matlab
time = 0:0.1:10;
x = myFunction(time);
```

Important: NEVER leave a space before the brackets of a function call. This often - but not always - leads to wrong results.
Data types in Matlab

**Vectors and Matrices** Matlab is strongly matrix oriented, which often allows you to avoid for-loops. Use this notation, wherever you can!

Example:

```
>> A = randi(10,3)
A =
    7   1   1
    2   3   9
    8   1   7
>> b = randi(10,3,1)
b =
    4
   10
    1
>> A*b
ans =
   39
   47
   49
```

**Structures** are simply variables with named sub-fields. They are often used to group information that belongs together. For example, if the three variables

```
DataValues = sin(1:10);
DataInfo = 'This is a sine-wave';
DataSourceID = 247;
```

belong together, you could group them into a structure called *data*:

```
data.Values = sin(1:10);
data.Info = 'This is a sine-wave';
data.sourceID = 247;
```
Then the Matlab structure \textit{data} contains everything that you want; and you can easily access the individual elements, for example the command \textit{absValue = abs(data.Values)}; would return the absolute value of the data-values.

\textbf{Cells} are very similar to Matlab arrays. The difference is that while each element of an array has the same format, different elements of a Matlab cell can have different formats.

To address elements of arrays, you use \((...)\), but for cells you use \{\ldots\} For example:

\begin{verbatim}
myArray = 1:3;
contains 3 double values; and you address e.g. the second value by

myArray(2)
In contrast,

myCell = {'abc', 1:3, rand(2)}
contains a string, a vector, and a matrix; and you address the second element (i.e. the vector) by

myCell{2}
Cells are also commonly used when you have to deal with strings of different length:

\begin{verbatim}
>> firstNames = {'Sam', 'Peter', 'Alexander'}
>> firstNames(3)
ans =
    'Alexander'
\end{verbatim}
1.3 Programming Tips

- Before you start programming, spell out the steps you have to do, and write them down as comments. (E.g. Set the parameters – Select the input file – Read in the data – Filter the data – Show the results – Save the results to an outfile – Show the user the location of the outfile). Not only does this help you to organize your code, it also provides a first rudimentary documentation of your program.

- Data analysis is an interactive task. Make use of the very powerful Matlab environment for interactive programming, and first develop your analysis step by step on the command-line.

- Once you have your data analysis - for the one block - going, grab the Matlab history (CTRL+1), and turn it into a function. Think about what you want/need for the input, and what the output should be.

- Before you implement a mathematical algorithm, write it down on paper! This makes the implementation much quicker, because you have to spell out what you want to do.

- Use the Matlab Help - it is one of the best parts of Matlab. Especially, use the code samples from the Help and from the Demo examples.

- If possible, use some simple dummy data to test your programming steps.

- Be careful with your choice of variable names. For example, never use i as a variable use ii instead (also jj instead of j, etc.)

```matlab
for ii = 1:10
    disp(ii);
end
```

- In the Matlab help browser it is often very helpful to narrow down the search-range to a few toolboxes. This can be done in File|Preferences|Help|EnableProductFilter.

- When you write your own functions, stick to the Matlab-style of documentation: The first line starts with the name of the function/script, and contains a one-line description. The next section gives more detailed information.

- Never use generic words, such as integer, line, etc as variable name: you might hide existing Matlab functions to your program! As a specific example, test the effect of a variable named line: if you generate a variable line, then you don’t see the function any more!

```matlab
which line
>> built-in (C:\Program Files\MATLAB\R2012a\toolbox\matlab\graphics \line)
line = 'This is a line.';
which line
>> line is a variable.
```

- Know your editor well - you are going to work with it a lot! Especially, know the keyboard shortcuts!

- Make sure you know the basic Matlab syntax. Try to use matrix multiplications instead of loops wherever possible: this makes your code nicer, and your programs much faster.
Once you know the Matlab basics, get to know the Debugger. It really helps to find mistakes in your programs.

Don’t repeat code. If you have to use a piece of code more than two times, write a Function instead.

1.3.1 Matlab Documentation

A very recommendable summary of programming style tips can be found in the MATLAB Style Guidelines 2.0 by Richard Johnson.

You should ALWAYS document the code that you write even if you only hack a small program! Thereby your file-header should follow certain rules (The parts with [...] are to be filled in):

[First a 1-line description of the program]
[A more thorough description]
[An example how to call the program]
[A description of the inputs and outputs]

Author: [xxx]
Version: [xxx]
Date: [xxx]

Example

```
%addMe  Calculates the sum of two numbers
%  This is a short demo program. It only works for numbers, vectors, %  and matrices.
%  
%  Call: summed = addMe(a,b)
%  E.g.: out = addMe(1,2)
%  
%  Inputs:
%    a ... scalar, vector, or matrix
%    b ... scalar, vector, or matrix; has to match "a" in size!
%  
%  Output:
%    added ... Sum of the inputs

% Author: Thomas Haslwanter
% Version: 1.0
% Date: 12-Sept-2012
%*****************************************************************

function added = addMe(a,b)
  added = a + b;
```
1.4 The Next Level

The use of the Debugger is a tremendous help in finding errors and solving problems.

To see a debugger at work, open the following file (or copy it into the MATLAB editor):

Listing 1.1: debugDemo.m

```
function debugDemo()
%DEBUGDEMO Demonstration of the use of a debugger
% author: ThH
% date: Sept-2015
% ver: 0.1

info = 'You are now at base-level.'

for ii = -10:10
    inverted = invMe(ii);
    disp(['The inverse of ' num2str(ii) ' is ' num2str(inverted)]);
end

function out = invMe(x)
%INVME Calculatde the inverse value

subinfo = 'Now you are one level down.'
out = 1/x;
disp([subinfo num2str(out)]);
end
```

- First try to run the file as it is: the program will crash at line 21, and display the error-message:

```
Error using horzcat
Dimensions of matrices being concatenated are not consistent.
Error in debugDemo>invMe (line 21)
disp([subinfo num2str(out)]);
Error in debugDemo (line 10)
inverted = invMe(ii);
```

Note that it shows not only the line of the crash (line 21), but also the line in the calling function which called the crashing subroutine (line 10).

- You can go to that location by clicking the underlined line xxx

- Now go to Editor -> Breakpoints, activate Stop on Errors, and run the program again.

- This time the console will indicate an additional line K>>, indicating that you are now in the debugger.

- When you look at the workspace, you will see the currently available variables (e.g. subinfo), and you can use the console to figure out why your program crashed at that point.

- With the drop-down menu in Editor -> Function Call Stack: you can also step up and down in the workspace. Give it a try, and change to debugDemo, i.e. the calling function. When you do that, the workspace changes, you will now see e.g. the variable info.
Chapter 2

Introduction

2.1 Why do signal processing?

"Data is the new oil", said the mathematician Clive Humby, who with his wife made $140 mio helping UK retailer Tesco with its Clubcard system. Just as crude oil is not very useful when it comes out of the ground, it can be refined into amazing things (fertilizer, gasoline, plastic, ) when processed correctly. The same is true for biomedical signals: in their unprocessed form, they don’t really help much. But when we extract the correct parameters, we can detect diseases, measure brain waves, look into the human body, predict future health problems, etc.

A nice discussion of this topic, and a good introduction into the field from the biomedical side, can be found in:


Another very recommendable book, which approaches signal processing from the engineering point of view, is:


2.2 What is the basic Procedure?

- Define the problem
- Identify the data of interest
- Collect the data
- Import the data
- Evaluate the data quality
- Preprocess the data
- Analyse the data
- Report the results
2.3 Important Topics and Questions

Figure 2.1: This figure shows some of the main topics and questions in digital signal processing (inspired by the excellent book by R.G.Lyons *Understanding Digital Signal Processing*).
2.4 Conventions and Basics

Movements in 3-dimensional space consist of translations as well as rotations. We will use the following conventions:

2.4.1 Notation

- Scalars are indicated by plain letters (e.g. a)
- Vectors are written with an arrow above (e.g. \( \vec{r} \)) or in round brackets
  
  \[
  \begin{pmatrix}
  r_1 \\
  r_2 \\
  r_3 
  \end{pmatrix}
  \]

- Matrices are written bold (e.g. \( \mathbf{R} \)) or in square brackets
  
  \[
  \begin{bmatrix}
  R_{11} & R_{12} & R_{13} \\
  R_{21} & R_{22} & R_{23} \\
  R_{31} & R_{32} & R_{33} 
  \end{bmatrix}
  \]

- Vector- and matrix-elements are written in plain style, with indices denoted by subscripts (e.g. \( r_{1}; R_{12} \))
- Multiplications with a scalar are denoted by \( \ast \) (e.g. \( \tan(\theta/2) \ast \vec{n} \))
- Scalar-vector-products and matrix-multiplications are denoted by \( \cdot \) (e.g. \( \vec{p} \cdot \vec{q} \))
- Vector-cross-products are denoted by \( \times \) (e.g. \( \vec{p} \times \vec{q} \))

2.4.2 Mathematical Basis

Scalar Product

The scalar product of two vectors \( \vec{a} \) and \( \vec{b} \) is defined as

\[
\begin{pmatrix}
  a_1 \\
  a_2 \\
  a_3 
  \end{pmatrix} \cdot \begin{pmatrix}
  b_1 \\
  b_2 \\
  b_3 
  \end{pmatrix} = a_1 b_1 + a_2 b_2 + a_3 b_3 = |\vec{a}| \cdot |\vec{b}| \cdot \cos(\theta)
\]

(2.1)

Figure 2.2: Graphical interpretation of scalar- (left) and vector-product (right).
CHAPTER 2. INTRODUCTION

Cross Product

The cross product of two vectors $\vec{a}$ and $\vec{b}$ is defined as

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$  \hspace{1cm} (2.2)

The resulting vector is perpendicular to $\vec{a}$ and $\vec{b}$, and vanishes if $\vec{a}$ and $\vec{b}$ are parallel.

Matrix Multiplication

In general, the multiplication of two matrices $A$ and $B$ is defined as

$$A \cdot B = C$$  \hspace{1cm} (2.3)

with $C_{ik} = \sum_j A_{ij} B_{jk}$.

This equation can also be used for multiplication of a matrix with a vector, when the vector is viewed as a matrix with three rows and one column. Matrix multiplications are very important in Matlab, as they are directly implemented, and save not only a lot of coding, but also a lot of time. For example,

```matlab
A = randn(500);
B = randn(500);
tic
for ii = 1:500
    for jj = 1:500
        C(ii,jj) = A(ii,:) * B(:,jj);
    end
end
withLoops = toc

tic
C=A*B;
withoutLoops = toc
```

gives on my computer a ratio of about 65!

![Figure 2.3: Basic trigonometry.](image-url)
Basic Trigonometry

The basic elements of trigonometry are illustrated in Figure 2.3.

2.4.3 Discrete Signals

*Discrete signals* are signals that are sampled at fixed points in time. Typically, all signals we work with in signal analysis are discrete, digital signals. *Digital signals* are signals that are stored with finite precision. This can lead to artifacts in the data analysis.

The information we obtain from discrete signals is limited by the *Nyquist theorem*: The highest frequency about which we can say something is half the sampling frequency. Higher frequencies can show up as artifacts (see Fig 2.4).

![Figure 2.4: Two different sine waves (here with 1Hz and 7Hz) pass through the same discrete values.](image)

![Figure 2.5: Simple graphical user interface (GUI).](image)
2.5 Exercises

- Create two cycles of a noisy sine wave, with the following properties:
  - amplitude = 1
  - frequency = 0.3 Hz
  - sampling rate = 100 Hz
  - Gaussian random noise, with a standard deviation of 0.5.

- Plot the wave, label the x- and the y-axis, and add a title to the plot.

- When this works, take your command history, and create a Matlab function that
  - takes the number of cycles and the frequency as input
  - sets the remaining parameters as above
  - shows the resulting graph

- Set a breakpoint in the function, and inspect the workspace variables at that point. Modify the amplitude to ”2”, and continue the function.

- Generate sample data by running the script DataGen.m. (See link below: you may also want to have a look at the file, to see how you can write data-files with different formats.) Read the data from the following (generated) files into your Matlab workspace:
  - data.csv  Comma-separated data file
  - data.tab.txt  Tab-separated data file
  - data.txt  Tab-separated data file, with header
  - data.xls  Excel file

- **Hard**: create a graphical user interface (GUI, see last chapter) for this function, as shown in the Figure 2.5.

  **Code**: ”DataGen.m” (p 117) shows how to write data in different formats, and how to produce formatted output.

  **Code**: ”DataRead.m” (p 119) shows how to read in data from different file formats.
Chapter 3

Data Filtering

In the analysis of data we want to extract those signal components that are interesting to us, and eliminate artifacts that may distort our data. For example, in Fig. 3.1(top) we have a 1-dimensional data signal. The original data contain not only the sine-wave we are interested in, but also a low-frequency drift and high-frequency noise. Ideally, our Filter should eliminate all the undesired components. In other cases, we may want to use a Filter to extract the outlines from 2-dimensional images (Fig. 3.1, bottom).

![Figure 3.1: Filters applied to 1-dimensional (top) and two-dimensional (bottom) signals.](image)

The following chapter will describe how such 1-dimensional and 2-dimensional filters work.
3.1 Transfer Function

To analyse biomedical data, you have to modify them. If the incoming data are called $x$, and the resulting/outgoing data $y$, this can be indicated schematically by the diagram in Fig. 3.2.

\[
x \rightarrow G \rightarrow y
\]

Figure 3.2: Transfer function

The box modifying the signal, here labelled $G$, is often called the transfer function. A simple case is the amplification of a signal

\[y = g \ast x\]  

(3.1)

$g$ is the gain of this transfer function, and a simple amplification is commonly indicated as shown in Fig. 3.3.

\[
x \rightarrow g \rightarrow y
\]

Figure 3.3: Amplification

For discrete valued signals, the output only depends on the instantaneous input:

\[y(n) = g \cdot x(n)\]  

(3.2)

Since the gain of many systems covers many orders of magnitude, it is often expressed on a logarithmic scale:

\[\text{attenuation} = 20 \cdot \log_{10} \left( \frac{a_{out}}{a_{in}} \right) \text{ dB}.\]  

(3.3)

For example, an amplitude reduction by a factor of 2 corresponds to an attenuation of $6 \text{ dB}$, and a reduction by a factor of 10 to $20 \text{ dB}$. 
3.2 Finite Impulse Response (FIR) filters

More options become available if the output values are obtained by weighting the last \( k \) incoming data points:

\[
y(n) = \sum_{i=0}^{k-1} w_i * x(n - i) = w_0 x(n) + w_1 x(n - 1) + ... + w_{k-1} x(n - k) \tag{3.4}
\]

This can be seen as a moving window filter, which is moved over the incoming data points from beginning to end (Fig. 3.4), and is called finite impulse response (FIR) filter.

3.2.1 Matlab Notes

The choice of coefficients in Eq. 3.4 reflects conventions in the area of Digital Signal Processing (DSP). Watch out, though, because vectors in Matlab start with \( x(1) \)!

We can implement Eq.3.4 in Matlab as

\[
y = \text{filter}(w, 1, x)
\]

where \( w \) is a vector containing the weights, and \( x \) a vector containing the input. The 1 will become clear when we discuss the more general IIR-filters below.

3.2.2 DSP Terminology

In DSP, Eq. 3.4 is sometimes expressed as

\[
y(n) = w(k) * x(n) \tag{3.5}
\]

and interpreted as \( y \) of \( n \) equals the convolution of \( w \) of \( k \) and \( x \) of \( n \). So don’t get nervous about the complicated sounding term convolution: it is nothing else but the application of an FIR filter.

Another expression that is used in DSP is impulse response: If the input consists of

\[
\begin{align*}
x(i) &= 1 \text{ for } i = k \\
x(i) &= 0 \text{ for } i \neq k
\end{align*}
\]
then the output around \( y(k) \) equals the weight coefficients. The impulse response of a filter is the filter’s output time-domain sequence when the input is a single unity-valued sample (impulse) preceded and followed by zero-valued samples.

Coming from the hardware design of filters, DSP textbooks also tend to use a different depiction of FIR filters (Fig. 3.5). Keep in mind, though, that this is equivalent to our Figure 3.4.

### 3.2.3 Offline Analysis

When your data needs to be analysed in real-time and online, you only have older values of \( x \) available: \( x(m) \), with \( m \leq n \). This induces a \textit{time-delay} in your output signal. For offline analysis, it is therefore more convenient to use a window that is centred about your current position (Fig. 3.6). This reduces or eliminates the problem of time delays in the output signal \( y \) relative to the input signal \( x \).

\[
y_n = \sum_{m=-k}^{k} w_m x_{n+m}
\]

(3.6)

![Figure 3.6: Filter using a centered window (offline analysis).](image)

### 3.2.4 FIR Filter 1: Moving average

If the filter values are chosen as

\[
w_i = \frac{1}{k}, \quad i = 0 : k - 1
\]

(3.7)
the filter averages over the last $k$ data points. For example, with $k=3$ and $n=10$, we get

$$w = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

and

$$y(10) = w_0 \cdot x(10) + w_1 \cdot x(9) + w_2 \cdot x(8) = \frac{x(10) + x(9) + x(8)}{3}$$

### 3.2.5 FIR Filter 2: Differentiation

#### First-difference Differentiation

A differentiation of an incoming signal, with $\Delta(t) = 1$

$$y(n) = \frac{\Delta x}{\Delta t} = \frac{x(n) - x(n-1)}{\Delta t}$$

gives us the filter weights (Fig. 3.7, top)

$$\vec{w} = \begin{bmatrix} 1 & -1 \end{bmatrix}$$

![First-difference differentiator](image)

Figure 3.7: Top) First-difference differentiator. Bottom) Central-difference differentiator.

#### Central-difference Differentiator

For offline analysis, a centered filter may be preferable (Fig. 3.7, bottom):

$$\vec{w} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \ast \frac{1}{2 \ast \Delta t}$$

#### Cubic Differentiator

We can also differentiate a curve by taking two samples before and after each point, and taking the slope of the best cubic fit to these data points. This can be achieved with a weight vector of:

$$\vec{w} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \ast \frac{1}{2 \ast \Delta t}$$

(3.12)
CHAPTER 3. DATA FILTERING

\[ \bar{w} = [1 - 8 0 8 - 1] \ast \frac{1}{12 \ast \Delta t} \]  

(3.13)

Note: This is essentially equivalent to a Savitzky-Golay filter with a polynomialOrder \( n = 3 \), a windowSize=5, and a derivativeOrder=1. The only difference is that the Savitzky-Golay filter uses a centered window, i.e. it compensates for the time-lag of the FIR-filter.

Specialized Differentiators

To optimize the noise and the frequency response, a number of differentiation algorithms have been published, which we won’t discuss here:

- Lanczos differentiators
- Parks-McClellan differentiators
- ...

3.2.6 FIR Filter 3: Savitzky-Golay Filter

The moving average is very simple, and can induce a bias in the output. For example, it systematically underestimates the value around the peak of a signal. To overcome this problem, the idea of the Savitzky-Golay filter is to fit a polynomial of order \( q \) to the surrounding \( 2m + 1 \) data points, and use its value at the centre for data smoothing, its first derivative for calculating the derivative, etc. One of the best things about the Savitzky-Golay filter is that you only have to determine the values \( \bar{w} \) once at the beginning of your analysis; the same values can then be used to estimate the derivatives of order up to \( q - 1 \).

Figure 3.8 shows the principle of the Savitzky Golay filter:

1. For a given x-value
2. you take a symmetric window of your data set, then
3. calculate the best polynomial fit to these data, and
4. if you want to smooth the data, take the center-point of the fitted curve; or if you want to differentiate the data, take the first derivative at that location.

In addition, you also see that a moving average filter (dashed line) would underestimate peak value of the curve.

So to use the Savitzky-Golay filter, you have to define the:

- input data (e.g., x)
- order of the polynomial fit (e.g., 2 for quadratic fits)
- size of the data window (must be odd, e.g., 9)
- order of derivative (e.g., 0 for smoothing, 1 for 1st derivative)
- sampling rate (in Hz, e.g., 100)

For example, to smooth data by finding the best-fit second-order polynomial \( (q = 2) \) to the data in a five point window \( (m = 2, \text{ as } 2 \ast 2 + 1 = 5) \), you would need the coefficients

\[ \bar{w} = [ -0.086 \ 0.343 \ 0.486 \ 0.343 \ -0.086 ] \]
For general values of $q$ and $m$, you can use the Matlab command \texttt{sgolay} to find the corresponding values of $\vec{w}$. For our example ($q = 2$ and $m = 2$), the command

\[
[b, g] = \texttt{sgolay}(2, 5);
\]

gives you the smoothing coefficients in the middle line of $b$, and the derivative coefficients in the columns of $g$.

For simple smoothing, you can also use the Matlab command \texttt{sgolayfilt}:

\[
y = \texttt{sgolayfilt}(x, q, 2 * m + 1);
\]

If you want to avoid dealing with the details of applying the weights, use the function \texttt{savgol} (by Thomas Haslwanter).

\textbf{Code:} "\texttt{savgol.m}" (p 120) Utility for data smoothing, and for calculating derivatives.

\textbf{Reasons to use the Savitzky-Golay filter:}

- It is an efficient way of smoothing data.
- It is a convenient way to find higher derivatives.
- Smoothing and calculating the higher derivatives can be done simultaneously.
Reasons not to use the Savitzky-Golay filter:

- It does not have a crisp frequency response. In other words, the gain decreases only gradually as frequency increases. For example, if you know that you only need frequency components below 200 Hz, other filtering techniques are preferable.

- If you are fitting data to a parametric model, it is almost always better to use raw data than pre-smoothed data, since the smoothing may cause important information to be lost.

- If you know the ideal signal characteristic, the Wiener filter, though more complex to apply, may produce better results.

Further information


3.3 Infinite Impulse Response (IIR) filters

We have seen that the output of a FIR filter only depends on the incoming signal:

\[ y(n) = \sum_{i=0}^{k-1} w_i \cdot x(n-i) \]

In general, though, the output of a filter may also depend on the most recent values of the output signal. In this case,

\[ y(n) + a_1 y(n-1) + \ldots + a_{m-1} y(n-m) = b_0 x(n) + b_1 x(n-1) + \ldots + b_k x(n-k) \]  

(3.14)

or

\[ \sum_{j=0}^{m-1} a_j y(n-j) = \sum_{i=0}^{k-1} b_i x(n-i) \]  

(3.15)

where \( a_0 = 1 \). In other words, the coefficients \( a_i \) and \( b_j \) uniquely determine this type of filter. This type of filter is sometimes referred to as an infinite impulse response (IIR) filter. The feedback character of these filters can be made more obvious by re-shuffling the equation:

\[ y(n) = [b_0 x(n) + b_1 x(n-1) + \ldots + b_k x(n-k)] - [a_1 y(n-1) + \ldots + a_m y(n-m)] \]  

(3.16)

IIR Filter 1: Exponential Decay

When the change in a signal is proportional to the magnitude of the signal, the signal changes exponentially. For example

\[ y_n = \alpha \cdot y_{n-1} \]  

(3.17)

For \( \alpha > 1 \) the signal grows exponentially, and for \( \alpha < 1 \) it decreases exponentially. For example, for \( \alpha = 1/2 \) we get

\[ y_n = y_0 \cdot [1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots] \]
Matlab Implementation  In Matlab, IIR filters can be implemented very easily by simply specifying the coefficients $a_i$ and $b_i$ (For FIR-filters $a = 1$):

$$ y = \text{filter}(b,a,x) $$

Comparing Eq.3.16 with Eq.3.17, we see that we can implement an Exponential Decay Filter as follows:

```matlab
% Dummy inData and parameters
x = randn(1,1000);
alpha = 0.5;

% IIR filter coefficients
a = [1 -0.5];
b = [];

% Apply the filter
y = filter(b,a,x);
```

IIR Filter 2: Leaky Integrator

The exponential decay is a special case of the Exponential Averaging Filter, defined as

$$ y(n) = \alpha x(n) + (1 - \alpha)y(n - 1) $$

(3.18)

You see: the exponential decay is the impulse response of an exponential averaging filter. The nice thing about this filter is that it is a smoothing filter with a single parameter: by tuning $\alpha$, you can determine the output’s sensitivity to input noise.

IIR Filter 3: Butterworth lowpass filter

The Matlab command `butter` provides the $[b,a]$ coefficients of an IIR filter corresponding to Butterworth filters. The Butterworth filter is a type of signal processing filter designed to have as flat a frequency response as possible in the pass band. It is also referred to as a maximally flat magnitude filter.

Note: Applying a low-pass filter is equivalent to smoothing the data.

For example, if you have a sampling rate of 1 kHz (and thus a Nyquist frequency of 500 Hz), and you want to design an IIR lowpass filter with a 3dB cut-off frequency of 40 Hz and a filter of order 5, you get the corresponding $[b,a]$ coefficients as follows:

```matlab
nyq = 500;
cutoff = 40;
order_filter = 5;
[b,a] = butter(order_filter, cutoff/nyq);
```

Warning: Be careful with low filter frequencies and higher order ($n \geq 4$) Butterworth filters: there the $[b,a]$ syntax may lead to numerical problems due to round-off errors. In that case you should switch to the $[z,p,k]$ syntax, which we don’t discuss here.
3.4 Difference between FIR and IIR filters

The two names finite impulse response filter and infinite impulse response filter are derived from the differing behaviour of each type of filter to an impulse input. To demonstrate this graphically, we implement an example of each type of filter in Matlab.

```matlab
% Input spike at 5 sec:
xx = zeros(1,20);
xx(5) = 1;
tt = 1:20;

data.before = xx;
data.after_fir = filter(ones(1,5)/5 , 1, xx);
data.after_iir = filter([1], [1 -0.5], xx);

% Graph
ph = plot(tt, data.before, 'bo', ...
    tt, data.after_fir, 'rx-', ...
    tt, data.after_iir, 'b.:');
for ii = 1:3
    set(ph(ii), 'LineWidth', 2);
end
legend('input', 'FIR-filtered', 'IIR-filtered');
```

This Matlab code produces the following graph:

![Graph comparison of FIR and IIR filter behaviour](image.png)

Figure 3.10: Comparison of FIR and IIR filter behaviour.

In this graph, we can see:

- the time delay of the FIR filter
- the finite effect of an impulse on FIR-filtered data
- the instant response of the IIR filter
- the infinite effect of an impulse on IIR-filtered data

For offline analysis, the Matlab command `filtfilt` filters each signal twice, once forward and once backward. This way time delays are eliminated.
3.5 Median Filter

FIR-filters and IIR-filters are both linear filters, since the filter coefficients enter the transfer through simple multiplications. While such linear filters are good at eliminating noise that has a Gaussian distribution, it fails when you have extreme outliers. Such spikes in the data can be caused e.g. by faulty sensors, or by loose connections in the analysis setup. For such outliers, a median filter offers a better noise suppression than linear filters. In Fig. 3.11 the signal has two outliers, one at t=5, and one at t=15. The averaging filter data have been adjusted to compensate for the delay, and both averaging and median filter have a window size of 3.

![Figure 3.11: Data with extreme outliers, average filtered, and median filtered.](image)

```matlab
% Create data
x = zeros(1,20);
x(10:20) = 1;

% Add some spikes, at t=5 and t=15
x(15) = 3;
x(5) = 3;

% Median filtered data
xMed = medfilt1(x, 3);

% Average filtered data
b = ones(1,3)/3;
xFilt = filter(b,1,x);

% Plot the data
plot(xFilt(2:end), 'g', 'LineWidth', 2); % shift to compensate delay
hold on
plot(x, '-.o', 'LineWidth', 2)
plot(xMed, 'r', 'LineWidth', 2)
legend('Average', 'Rawdata', 'Median')
```
3.6 Filtering images (2-D filtering)

3.6.1 Representation of Grayscale Images

The simplest image-type is grayscale images: there each pixel is given a gray-level

\[ 0 < \text{gray level} < g_{\text{max}} \]

Figure 3.12: Example, with \(0 < \text{gray levels} < 255\). I have used the Matlab command \textit{imtool} to obtain these images.

Here it is up to you which maximum level is used. For many images, 8-bit is sufficient, giving you \(2^8 = 256\) gray levels. Note that a higher image depth only makes sense if your sensing devices can show differences at a higher level! Also keep in mind that Matlab uses by default double precision values with 64 bit. This requires 8 times as much memory, and is ok for small images. For larger images you should stick to the format with the lower memory requirements, and use the appropriate functions of the Matlab \textit{Image Processing Toolbox}.

3.6.2 Color images

A color image is nothing more than a stack of three grayscale images: one representing the "red" channel, one the "green" channel, and one the "blue" channel. If the three colors are stacked in this sequence, the image is referred to as an \textit{RGB-image}.

\begin{verbatim}
img = imread('peppers.png');
size(img)
\end{verbatim}

384 512 3

3.6.3 2D-Filtering

Filtering can also be performed on two-dimensional signals, like images. Instead of a one-dimensional input and a weight vector, we now have a two-dimensional input, and a weight matrix (Fig. 3.13). The output is still obtained by summing up the weighted input.

\begin{equation}
y(n,m) = \sum_{i=1}^{k} \sum_{j=1}^{l} w_{ij} \times x(n - 1 + i, m - 1 + j) \tag{3.19}\end{equation}

The moving window interpretation still holds, except now, the window extends in both dimensions.
CHAPTER 3. DATA FILTERING

Figure 3.13: Left) A two-dimensional filter applied to a two-dimensional window. Right) Definition of “Structural Element”. Note that a structural element does not have to be a square; it can also be a circle, a rectangle, etc.

If the image is a grayscale image, you could use the command filter2. However, it is recommendable to stick with the functions of the Matlab Image Processing Toolbox; in this case, imfilter would be the corresponding command. Using commands from the image processing toolbox has two advantages: where possible, it does not convert the data to a larger format (i.e., 8-bit unsigned integers are not converted to double); and it also works on RGB data (where filter2 would not work). For example, in order to blur an image, enhance horizontal lines, or enhance vertical lines, you can use the following lines of code:

```matlab
% Get the data
data = imread('pout.tif');

% Design the filters
Filter1 = ones(7)/7^2;
Filter2 = [1 1 1; 0 0 0; -1 -1 -1];
Filter3 = Filter2';

% Apply the filters (the "*4" is only to enhance the visibility)
subplot(221), imshow(data);
subplot(222), imshow(imfilter(data, Filter1));
subplot(223), imshow(imfilter(data, Filter2)*4);
subplot(224), imshow(imfilter(data, Filter3)*4);
```

Tip: Before you create your own filter, first check if the Matlab command fspecial already provides the filter that you need.
Figure 3.14: Original image (top-left), blurred version (top-right), horizontal edges enhanced (bottom-left), and vertical edges enhanced (bottom-right)
3.7 The Next Level

Curve smoothing is a huge topic, and depending on the requirements, a number of solutions are available. For example you may have data sampled at equal intervals, or you may have recorded them randomly. In the former case, you can use Savitzky-Golay filters or IIR-filters. In the latter case you need other approaches. Some of them are listed below.

3.7.1 Lowess and Loess Smoothing

When you have irregularly sampled data points, you cannot apply a Savitzky-Golay filter. LOESS (which can be interpreted as \textit{Local} regr\textit{ESS}ion, and LOWESS (\textit{LOcally W}eighted \textit{Scat}ter\textit{plot} Smoothing) are two strongly related non-parametric regression methods that combine multiple regression models in a k-nearest-neighbor-based meta-model. \textit{loess} is a later generalization of \textit{lowess}.

![Figure 3.15: LOWESS and LOESS smoothing of noisy data.](image)

In short, you specify the percentage of the data that you want to include. For these data, a weighted linear regression is applied. The traditional weight function used for LOESS is the tri-cube weight function,

\[
  w(x) = (1 - |x|^3)^3 I_{|x|<1}
\]  

(3.20)

$I_{...}$ is the \textit{indicator function}, indicating the range over which the function is different from 0.

The methods are just differentiated by the model used in the regression: \textit{lowess} uses a linear polynomial, while \textit{loess} uses a quadratic polynomial. An example of lowess and loess smoothing is shown in Fig.3.15.

```matlab
% Generate the data
x = 0:0.1:10;
y = sin(x)+0.2*randn(size(x));

% Eliminate some, so that we don't have equal sampling distances
curInd = (x>5) & (x<6);
x(curInd) = []; 
y(curInd) = [];

% Smooth the data
sm_lowess = smooth(x,y, 0.1, 'lowess');
```
3.7. THE NEXT LEVEL

```
sm_loess = smooth(x,y, 0.1, 'loess');

% Show the results
close
plot(x,y,'*')
hold on
lh(1) = plot(x, sm_lowess, 'r');
set(lh(1), 'LineWidth', 3);
lh(2) = plot(x,sm_loess, 'g--');
set(lh(2), 'LineWidth', 3);
legend('raw data', 'lowess', 'loess');
shg
```

3.7.2 Splines

The interpolation methods described on page 52 all go through the given data points. But this feature is not always required, and so-called splines can be used not only for interpolation, but also for data smoothing and differentiation.

The ideas of splines have their roots in the aircraft and shipbuilding industries. For example, the British aircraft industry during World War II used to construct templates for airplanes by passing thin wooden strips (called "splines") through points laid out on the floor of a large design loft, a technique borrowed from ship-hull design (Fig. 3.16). In the late 1950s and 60s, the computational use of splines was developed for modeling automobile bodies. In the computer science subfields of computer-aided design and computer graphics, splines are popular because of the simplicity of their construction, their ease and accuracy of evaluation, and their capacity to approximate complex shapes through curve fitting and interactive curve design.

A spline-function is nowadays defined as a piecewise polynomial function of degree < k in a variable x.

![Figure 3.16: A wooden spline. (From the archives of Pearson Scott Foresman, donated to the Wikimedia Foundation)](image)

B-Splines

One particularly simple and powerful option to construct smooth, piecewise polynomial 2-D and 3-D trajectories are so-called “B-splines”. The term B-splines stands for basis splines, since any spline function of given degree can be expressed as a linear combination of B-splines of that degree.

For a given trajectory, the spline-knots separate the piecewise polynomial parts of the trajectory. If the knots are equidistant, the spline is called cardinal B-spline, and the definition of B-splines then becomes remarkably simple:

With a B-spline of degree p (p ∈ N₀), the convolution operator ∗, and the indicator function \( b^p \mid_{[0,1)} \) of the half-open unit interval \([0,1)\), the corresponding cardinal B-splines is given by
CHAPTER 3. DATA FILTERING

\[ \mathbf{b}^p := \mathbf{1}_{[0,1)} * \ldots * \mathbf{1}_{[0,1)} \]  

(3.21)

Figure 3.17: The first four B-splines

Note that B-splines have what is called minimal support: linear B-splines only have an effect over two adjacent knots, quadratic B-splines over three knots, etc.

A B-spline curve \( C(u), u \in [\tau_p, \tau_{n-p-1}] \) of grade \( p \) with knot vector \( \tau \) and control points \( P_i (i = 0, \ldots, n - p - 2) \) (also called ”De-Boor-points”) is given by

\[ C(u) = \sum_{i=0}^{n-p-2} P_i N_{i,p,\tau}(u) \]  

(3.22)

In one dimension, the construction of a linear spline by three control points can be easily visualized:

A different formulation, which also works for knots that are not equidistant, as been given by DeBoor. That formulation involves recursive equations, and is implemented in the file:

Code: ”deboor.m" (p 125) Code to generate B-splines. That file is use by

"bspline.m" (p 125) to generate the 2-D cubic B-spline in Fig. 3.19. (Confusingly, a B-spline of order \( n \) contains polynomials of degree \( n-1 \)!

3.7.3 Kernel Density Estimation

Another type of smoothing can be needed when you want to know the frequency of events. Such information is often displayed with histograms. However, it is possible to smooth histogram data systematically, with a technique called Kernel Density Estimation (KDE).

For example, your friends, the Birdlife Austria, give you a list with the observations of bearded vultures ("Laemmergeier”), and ask you to represent this as a smooth curve. You could do this, by representing every observation by a Gaussian function

\[ g(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{1}{2} \left( \frac{x-\mu}{\sigma} \right)^2}. \]  

(3.23)
Figure 3.18: Explanation of B-spline curves. This example shows three control points, and the three corresponding linear B-splines. Multiplying each point with the corresponding (dotted) B-spline, and summing up the results, gives the solid line. Note that each point only contributes to a limited interval of the total curve.

Figure 3.19: B-spline with control points/control polygon, and marked component curves.

and then sum all the resulting functions.

```matlab
% Define the smoothness and the Gaussian functions
sigma = 1; % bandwidth
gauss = @(x,mu) 1/(sigma*sqrt(2*pi))*exp(-0.5*((x-mu)/sigma).^2)

% Define the input and output parameters
x = -10:0.1:10;
mu = [-5, -3, 2, 3, 3.5, 4, 6]
total = zeros(size(x));

% Calculate and plot the individual exponentials, and the sum total
hold on
set(gca, 'XTickMode', 'manual', ... 'XTick',[]);
for ii = 1:length(mu)
    out = gauss(x, mu(ii));
    plot(x, out, '-o');
end
hold off
```

Figure 3.20: **Kernel Density Estimation:** The "*" on the x-axis indicate individual events, and the thin blue lines the corresponding Gaussians. The thick red line is the sum of the thin blue lines, and provides a "density estimate" for the event rate.

```matlab
plot(x, out)
total = total + out;
end
lh(1) = plot(mu, zeros(size(mu)), '*');
set(lh(1), 'MarkerSize', 10);
lh(2) = plot(x, total, 'r');
set(lh(2), 'LineWidth', 2);
xlabel('X');
ylabel('Y');
hold off
shg```

3.7. THE NEXT LEVEL

3.7.4 Morphological Operations in Image Processing

Erosion and Dilation of Images

For linear filters as seen before, it holds that they are commutative. Cite from wikipedia: *One says that x commutes with y under "∗" if:

\[ x ∗ y = y ∗ x \] (3.24)

In other words, it does not matter how many and in which sequence different linear filters you use. E.g. if a Savitzky-Golay filter is applied to some date, and then a second Savitzky-Golay filter for calculating the first derivative, the result is the same if the sequence of filters is reversed. It even holds, that there would have been one filter, which does the same as the two applied.

In contrast morphological operations on an image are non-linear operations and the final result depends on the sequence. If we think of any image, it is defined by pixels with values \( x_{ij} \). Further this image is assumed to be a black-and-white image, so we have

\[ x_{ij} = 0 \text{ or } 1, \forall i, j \] (3.25)

To define a morphological operation we have to set a structural element \( SE \). As example, a 3x3-Matrix as a part of the image.

The definition of erosion \( E \) says:

\[ E(M) = \begin{cases} 0, & \text{if } \sum_{i,j=0}^{3} (se)_{ij} < 9 \text{, with } (se)_{ij}, M \in SE \\ 1, & \text{else} \end{cases} \] (3.26)

So in words, if any of the pixels in the structural element M has value 0, the erosion sets the value of M, a specific pixel in M, to zero. Otherwise \( E(M)=1 \).

And for the dilation \( D \) it holds, if any value in \( SE \) is 1, the dilation of M, \( D(M) \), is set to 1.

\[ D(M) = \begin{cases} 1, & \text{if } \sum_{i,j=0}^{3} (se)_{ij} >= 9 \text{, with } (se)_{ij}, M \in SE \\ 0, & \text{else} \end{cases} \] (3.27)

Opening and Closing of Images

Opening and Closing of Images

Figure 3.21: Sample image for demonstrating the effect of "opening" and "closing" of binary images.

There are two compositions of dilation and erosion. One called opening the other called closing. It holds:
Figure 3.22: Compositions of Dilation and Erosion: Opening and Closing of Images.

\[
\text{opening} = \text{dilation} \circ \text{erosion} \\
\text{closing} = \text{erosion} \circ \text{dilation}
\]  

\((3.28)\)

**Code:** "ShowMorph.m" (p 124) gives the Matlab code that generates Fig. 3.22.
3.8 Exercises

- (As in the Exercise in the previous chapter:) Create a noisy sine wave, with an amplitude of 1, a frequency of 0.3 Hz, a sampling rate of 100 Hz, a duration of 10 sec, and a Gaussian random noise with a standard deviation of 0.5.

- Using a Savitzky Golay filter, calculate the first derivative using a 2\textsuperscript{nd} order polynomial. Try different window sizes, and superpose the results in a plot. Make sure that the axes of the plot are correctly labelled.

- Generate a dummy dataset consisting of the sum of three sine-waves, with frequencies of [2, 30, 400] Hz, and amplitudes of [0.5, 1, 0.1]. The signal should have a sampling rate of 5 kHz, and a duration of 2 sec. Now generate a Butterworth low-pass filter, for the frequency band between 10 Hz and 100 Hz. When you apply that filter to your data, you should obtain a pure sine-wave with 30 Hz.

- Apply the exponential averaging filter to a step-input. Plot the response for different values of $\alpha$, to show why this filter is in some fields referred to as leaky integrator.
Chapter 4

Event- and Feature-Finding

Often it is necessary to find specific locations in a stream of data. For example, you might want to find out when a signal passes a given threshold (Fig. 4.1). Or when you analyze hand movements, you might want to find at which point the hand starts to move, and at which the hand movement ends. These locations are often referred to as features, or events if the data is a time series.

![Figure 4.1: Events can be the times when a signal passes a given Threshold.](image)

Figure 4.1: Events can be the times when a signal passes a given Threshold.
4.1 Find simple events in 1-D data

In Matlab there are two methods with which you can access single events:

- logical indexing
- the `find` function.

We will illustrate these methods with an example. Let us take the values

\[ xx = 3:8; \]
\[ yy = xx/10; \]

To check whether `xx` is larger than 5, you can simply use a logical comparison:

\[ isLarge = xx > 5; \]

The result of the comparison, which is now stored in `isLarge`, is the boolean vector

\[ [0 0 0 1 1 1] \]

To find the corresponding data in `yy`, the easiest (and often most efficient) method is to use logical indexing, which would be

\[ yy(isLarge) \]

or equivalently

\[ yy(xx > 5) \]

If it is absolutely necessary to find the indices of `xx` that are larger than 5, then the `find` function can be used:

\[ largeIndices = find(isLarge); \]

or equivalently

\[ largeIndices = find(xx > 5); \]

Mostly, though, this is unnecessary, and since `find` is currently far slower than logical indexing, `find` should only be used as a last resort.

4.1.1 Example 1: Find large signal values in data

```matlab
% Create dummy data
time = 0 : 0.01 : 20;
data = sin(time);

% Set a threshold
threshold = 0.7;

% Find the (binary) indices of all data above that threshold
isLarge = data > threshold;

% At the end, plot the data
```
4.1. FIND SIMPLE EVENTS IN 1-D DATA

Figure 4.2: Selection of large signal values

```
subplot(3,1,1)
plot(time, data)
ylabel('All data');

subplot(3,1,2)
plot(time(isLarge), data(isLarge), '.-')
ylabel('Large data')
xlabel('Time [s]')

subplot(3,1,3)
plot(data(is_large))
ylabel('Large data only')
xlabel('Points only')
```

This code produces the graphs in Fig. 4.2.

Now, if we want to eliminate all events that occur in the first 10 seconds, we could add the following code:

```
% First find the late section ...
timeThreshold = 10;
isLate = time > timeThreshold;

% ... and then combine it with the other criterion with a bit-wise AND
isLateAndLarge = isLarge & isLate;
```
4.1.2 Example 2: Find the start and end of an eye movement

Real measurement data will always contain noise (and usually a lot more than you want). To remove noise-effects from the analysis, it is therefore often helpful to go from continuous (or almost continuous) data to yes/no representations.

For example, for finding the onset of a signal such as in Fig. 4.3, we could break down the analysis into the following steps:

1. Find a threshold (i.e. a signal value that has to be reached in order to be part of our event).

2. For each data-point, calculate if the signal is above the threshold (yes/no).

   **Comment:** At this stage, you have gone from a noisy, real-valued signal, to a discrete, binary signal, which can only be 0 or 1!

3. Now the start of our event can be found easily: it is the point where $\text{diff}(\text{signal})$ is 1. Similarly, the end of our event can be found by checking where $\text{diff}(\text{signal})$ is $-1$.

Real data will probably require a few more steps. But the essence of event-finding is contained elegantly in this example of signal-analysis.

---

**Listing 4.1: EventDetection.m**

```matlab
%EventDetection Show how events can be elegantly detected using binary indexing
%
% ThH, Sept-2016, Ver 1.0
%**********************************************

% Get eye positions, sampled with 100 Hz
load HorPos; % This file has to exist in your current directory!
rate = 100;

% Select an interesting domain
myDomain = 9000 : 13500;

% Plot 1: raw data
subplot(3,2,1)
plot(HorPos(myDomain))
ylabel('Eye Position [deg]');
axis tight

% Plot 2: absolute velocity
orderPoly = 3;
winSize = 71;
deriv = 1;
eyeVelocity = savgol(HorPos, orderPoly, winSize, deriv, rate);
eyeAbsoluteVelocity = abs(eyeVelocity);
subplot(3,2,3)
plot(eyeAbsoluteVelocity(myDomain))
ylabel('Absolute Eye Velocity [deg]')
axis tight

% Set a default threshold, in case the threshold is not determined
% interactively
threshold = 6.3;

%To find the threshold interactively, use the following lines
% set(gcf, 'Name', 'Select the threshold:')
% selectedPoint = ginput(1);
```
4.1. FIND SIMPLE EVENTS IN 1-D DATA

% threshold = selectedPoint(2); % I only want the y-value
% set(gcf, 'Name', '');
line(xlim, [threshold threshold], 'Color', 'r')

% Plot3: show where the absolute velocity exceeds the threshold
isFast = eyeAbsoluteVelocity > threshold;
subplot(3,2,5)
plot(isFast(myDomain), '-x')
axis tight
ylabel('Above threshold')

% Plot4: as Plot3, but zoomed in
closeDomain = 9900 : 10600;
subplot(3,2,2)
plot(isFast(closeDomain), '-x')
axis tight
ylabel('Above threshold');

% Plot5: Find the start and end of each movement
subplot(3,2,4)
startStop = diff(isFast);
plot(startStop(closeDomain))
ylabel('Start / Stop')

% Find the start and end times for all movements (in sec)
movementStartTimes = find(startStop == 1)/rate
movementEndTimes = find(startStop == -1)/rate

The code produces the graphs in Fig. 4.3.

![Graphs](image)

Figure 4.3: Detection of the start and end of the eye movements
4.2 Interpolation

When finding the point at which a curve crosses a given threshold, we may want more accuracy than we have in the data. To do so we can find datapoints between our recorded samples through interpolation.

![Stepwise linear interpolation](image)

Figure 4.4: Stepwise linear interpolation.

What we discuss here is a simplified approach of interpolation, where we interpolate between fixed data points. In digital signal processing, interpolation is typically approached as a combination of low-pass filtering, followed by decimation. (Decimation by a factor of m is if we take every $n^{th}$ data point from our signal.) This approach allows the specification of exact frequency responses. You should look these topics up if you e.g. down-sample an audio signal from a CD, where you want good control over the way your data manipulation affects the frequency content of the original signal.

4.2.1 Linear interpolation

The simplest form is linear interpolation, where we obtain points between samples through linear interpolation between the adjacent samples. While this is a computationally quick approach, it has two disadvantages:

- It is not very accurate.
- It is discontinuous at the location of the samples.

```matlab
%% Generate the data
x = 0:6;
v = sin(x);

% Linear interpolation
xi = 0:0.01:6;
vi = interp1(x,v,xi);

%Plot the data
plot(xi, vi);
hold on;
plot(x,v, 'ro');
line([0, 6], [0, 0], 'LineStyle', '--');
```
4.2. INTERPOLATION

4.2.2 Cubic Spline Interpolation

To overcome these problems, cubic spline interpolations are often used. Thereby the datapoints between samples are derived from cubic polynomials. The expression spline indicates that the polynomial coefficients satisfy two conditions:

- They are continuous at the location of the samples.
- The second derivative at the end of each polynomial is zero.

![Figure 4.5: Cubic spline interpolation.](image)

```matlab
vic = interp1(x,v,xi, 'pchip');
vis = interp1(x,v,xi, 'spline');

% Plot polynomial interpolation
plot(xi, vic);
hold on;
plot(xi, vis);
plot(x, v, 'o');
line([0, 6], [0, 0], 'LineStyle', '--')
legend('pchip', 'spline', 'raw')
```

With the option `pchip`, the polynomial is chosen such that the first derivative is continuous. And with the – computationally slightly more demanding – option `spline`, also the second derivative is continuous.
CHAPTER 4. EVENT- AND FEATURE-FINDING

4.3 Finding simple features in 2-D data

Many methods that are used to find features in one-dimensional signals can also be applied in a modified form to two-dimensional signals. Many examples can be found in the field of image processing.

4.3.1 Find the position of bright pixels in a grayscale image

For example, to find the bright pixels in a uint8 grayscale image, all we need to do is apply a threshold:

```matlab
rawImage = imread('pout.tif');
brightThreshold = 110;
isBright = rawImage > brightThreshold;
imshow(isBright);
```

Figure 4.6: Binary image, showing the areas above threshold in white, the rest in black.

Alternatively, we could use the function `im2bw` in the Image Processing Toolbox

```matlab
isBright = im2bw(rawImage, 110/255);
```

To get the indices of the bright pixels, we can use `find` again, but now with two output variables:

```matlab
[rowBright, colBright] = find(isBright);
```

Matlab has a very powerful Image Processing Toolbox, which is – in my opinion – together with the Signal Processing Toolbox, one of the most powerful and useful tools in Matlab. If you have to work with image or video analysis, you should definitely check it out!
4.4 Cross-correlation

Sometimes the features that you need to find are more complex. For example, in an electrocardiogram, it might be necessary to locate only the peaks of the P waves because you are specifically interested in the contraction of the cardiac atria. However, such a task may be complicated in a real signal because the P wave may be very small or even non-existent.

Figure 4.7: Schematic of the electrocardiogram, taken from Wikipedia.

Figure 4.8: A real electrocardiogram, taken from Wikipedia.

There are many ways to tackle this problem. One way would be to find the peak of the R or S wave and then search backwards for the peak of the P wave. This method is fine as long as the signal is not too noisy.

However, if there is more noise in the signal, it may be useful to create a template of the ECG signal and use cross-correlation to align the template with the signal.

4.4.1 Comparing signals using cross-correlation

Let us consider how we might assess the similarity of two signals. If the reference signal is $R(t)$ and the compared signal is $C(t)$, we want some similarity function $Q(R(t), C(t))$ such that $Q$ has a maximum when $C = R$, and decreases as the difference between $C$ and $R$ increases.

It can be shown that the function $Q(t) = C(t) \cdot R(t)$ satisfies both of these properties. Thus, all we need to do to compare two signals is to multiply the two functions together. If we want to find out how much the reference signal needs to be shifted to match the compared signal, we
calculate the similarity of two signals for different relative shifts and choose the shift where the signals have the maximum amount of similarity.

For example, let us take the two signals

\[ R(t) = (0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0), \text{ and } C(t) = (0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0) \]

shown in Fig. 4.9 B). R and C have the same pattern, but are shifted by 4 time steps. If we calculate the cross-correlation function, we get Fig. 4.9 C)

We can see that the maximum value is reached when the compared signal is shifted by four time steps, which is where the two signals align.

So the cross-correlation gives us two pieces of information:

- *How similar* signal and reference are (through the maximum of the cross-correlation).
- *Where* the similarity occurs (through the location of the maximum).

In general, if \( x \) is the compared signal and \( y \) is the reference signal, the cross-correlation function can be obtained by

\[
Q_{xy}(m) = \begin{cases} 
\sum_{n=0}^{N-m-1} x_{n+m} y_n^* & 0 \leq m < N - 1 \\
Q_{yx}^*(-m) & 1 - N < m < 0
\end{cases}
\]

where \( x \) and \( y \) are length \( N \) vectors \((N > 1)\). If \( x \) and \( y \) are not the same length, the shorter vector is zero-padded to the length of the longer vector. Note that the complex conjugate of \( y \) must be taken if \( y \) is a complex function.

In practice, the cross-correlation function is not calculated directly, but rather indirectly, using the Fourier transform. If you are interested, you can take a look at the implementation.
4.4. CROSS-CORRELATION

of \texttt{xcorr} in Matlab. The concept of cross-correlation can be generalised to higher dimensions, for example to two dimensions to find a known object in an image.

To visualize cross correlations, check out the Matlab-function \texttt{Corrviz.m} on Moodle.

Auto-correlation

If the two signals being compared are the same (i.e., \(x = y\)), the result is called the auto-correlation function. Naturally, the auto-correlation function is not used to find events. Rather it can be used to detect periodicity of a noisy signal.

4.4.2 Examples

Finding a pattern in a signal

```matlab
% Create the Reference Pattern
myPattern = zeros(1,5);
myPattern(2:4)=1;

% Create the Signal
x = linspace(0,2*pi,18);
y = sin(x);
mySignal = [myPattern, zeros(1,3), y];

% Zero-pad the Reference Pattern to the same length as the Signal
myPattern = [zeros(1,length(mySignal)-length(myPattern)) myPattern];

% Plot the signals and the cross-correlation
subplot(311), plot(mySignal, '*-');
title('mySignal');
line(xlim, [0 0], 'LineStyle', '--', 'Color', [0.3 0.3 0.3]);

subplot(312), plot(myPattern, '*-');
title('myPattern');
line(xlim, [0 0], 'LineStyle', '--', 'Color', [0.3 0.3 0.3]);

subplot(313), plot(xcorr(mySignal, myPattern), '*-')
title('Cross-Correlation');
axis tight;
line(xlim, [0 0], 'LineStyle', '--', 'Color', [0.3 0.3 0.3]);
shg
```

Auto-correlation functions for simple signals

4.4.3 Features of Cross-correlation functions

Length of cross correlation function

If the signal has a length of \(n\) points, and the pattern a length of \(m\) points then the cross-correlation has a length of \(n + m - 1\) points. For example, the auto-correlation function of a signal with 10 points has a length of 19 points.

Maximum of cross correlation function

If the signal is multiplied by a factor of \(a\), and the pattern multiplied by a factor or \(b\), then the maximum of the cross-correlation function increases by a factor \(a \times b\). For example, if a signal increases by a factor of 2, the maximum of the auto-correlation function increases by a factor of 4.
Normalization of auto-correlation function

Sometimes we want to compare the shape of signals, regardless of their amplitude. For example, myo-electric prostheses use EMG-signals for their control. Now we would like the prosthesis to behave in a repeatable manner, independently of the current quality of the electrode contacts.

In order to evaluate the shape of a signal, regardless of its overall amplitude, we have to normalize the signal.

Thereby the normalization has to take three aspects of the signal into consideration:

**Offset** To normalize the offset, we can subtract the mean of the signal, or the smallest value of the signal. (The latter option ensures that our resulting signal is always positive.)

**Duration** To ensure that two signals have the same length, we can interpolate them to a fixed number of data points.

**Amplitude** One way to normalize the amplitude is to scale the signal such that the smallest value is 0 (see above), and the largest value is 1. However, a better way is to normalize it such that the maximum value of the auto-correlation function = 1. We can achieve this with

\[
\text{sig}_{\text{normalized}} = \frac{\text{sig}_{\text{raw}}}{\sqrt{\max(xcorr(\text{sig}_{\text{raw}}))}}
\]  

(4.2)

4.4.4 Cross-correlation and convolution

Cross-correlation and convolution are actually very similar:

a Convolution of \( f(t) \) with \( g(t) \) is often written as \( f * g \).

Using this notation, a Cross-correlation of the functions \( f \) and \( g \) is equivalent to the convolution of \( f^*(-t) \) with \( g(t) \): \( f^*(-t) * g \)
4.4. CROSS-CORRELATION

Figure 4.11: Examples of the auto-correlation function of some simple functions.

Figure 4.12: Comparison of Convolution, Cross-correlation and Autocorrelation.
4.5 Exercises

The code below generates a signal, containing i) steps with a length of 50, and ii) sinusoids with a length of 50. Use a cross correlation, and templates for the step and the sine-wave, and find the location where each of these signals occurs.

```matlab
% Generate a signal, containing steps and sine-waves

% Base signal
rate = 100;
dt = 1/rate;
noiseAmp = 0.1;
signalLength = 50;
t = 0:dt:signalLength;

% Step and Sine
sig = randn(1, length(t))*noiseAmp;
step = ones(1,50);
sineT = linspace(0,2*pi, 50);
sine = sin(sineT);

stepOnsets = [500, 1500, 4400];
sineOnsets = [1000, 2500, 4100];

for ii = 1:3
    onset = stepOnsets(ii);
sig(onset:onset+49) = sig(onset:onset+49) + step;
    onset = sineOnsets(ii);
sig(onset:onset+49) = sig(onset:onset+49) + sine;
end
```
Chapter 5

Statistics

5.1 Introduction to Statistics

5.1.1 Describing a Distribution

The average male American adult weighs 88.3 kg. This is a lot (actually, way too much). But how do male Austrians compare?

To answer this question, we first have to get an overview over the weight-distribution in Austrian males. We take 5 males, record their weight, and get

\[ \text{weight} = [84, 86, 88, 61, 82] \text{ kg}. \]

Based on these data, we can now estimate the average weight, and how variable the weight is between different men. Thereby we make use of the assumption that the data often have a bell-shaped distribution: the most likely value is the average value, and it is equally likely to find a low value or a high value. Mathematically, we can describe such data with the normal distribution, also called Gaussian distribution: (Fig. 5.1).

\[
f_{\mu,\sigma}(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}
\]

\[-\infty < x < \infty, \text{ and } f_{\mu,\sigma} \text{ is called the probability density function (PDF) of the normal distribution. } \mu \text{ is the mean value, and } \sigma \text{ is called standard deviation, and characterizes the variability of the data.} \]

From our data, the best estimate for the mean value for the weight of the average Austrian male is

\[
\bar{w} = \frac{\sum_{i=1}^{5} w_i}{n} = 81.2 \text{ kg}
\]

(5.2)

The best estimate we can get for the standard deviation is

\[
s = \sqrt{\frac{\sum (w_i - \bar{w})^2}{n - 1}}
\]

(5.3)

Note that we divide by \( n - 1 \) here! This is not quite intuitive, and is caused by the fact that the real mean is unknown, and our sample mean (Eq. 5.2) is chosen such that the standard deviation is minimized. This leads to an underestimation of the standard deviation, which can be compensated by dividing by \( n - 1 \) instead of \( n \).

The standard deviation as defined in Eq. 5.3 is sometimes called sample standard deviation (because it is our best guess based on the sampled data) and is typically labelled with \( s \). In contrast, the population standard deviation is commonly denoted with \( \sigma \).

The variance is the square of the standard deviation.
5.1.2 Confidence Intervals

If we know the mean and the shape of the distribution, we can determine the $\alpha$-% confidence intervals, i.e. the weight-range about the mean that contains $\alpha$-% of all data (Fig. 5.1):

$$CI = \text{mean} \pm \sigma \cdot z_{1-\alpha/2} \quad (5.4)$$

So if we want to know the 95% confidence intervals, we have to corresponding $z$-value for 0.05/2. The factor $\alpha/2$ is due to the fact that we have outliers below and above the confidence intervals (Fig. 5.2).

Figure 5.1: Normal Distribution: the unit on the horizontal axis is 1 $\sigma$. This distribution is sometimes also referred to as $z$-distribution.

Figure 5.2: If 95% of the data are within the confidence limits, 2.5% of the data lie above, and 2.5% below those limits.
5.1. INTRODUCTION TO STATISTICS

5.1.3 Boxplot and Quartiles

For distributions with extreme outliers, the mean no longer represents a “typical” data value. In that case it is better to use the median and quartiles to describe the data points:

**Upper quartile** Smallest value that is larger than 75% of all data.

**Median** Smallest value that is larger than 50% of all data.

**Lower quartile** Smallest value that is larger than 25% of all data.

Those are also the values that are indicated in a boxplot:

![Boxplot](image)

Figure 5.3: The difference between the upper and lower quartile is called the Inter-Quartile-Range (IQR). And typically, error-bars indicate the most extreme values inside 1.5*IQR outside the upper/lower quartile.

Note that this also provides a clear definition of outlier-data: data that are more than 1.5 IQR away from the closest quartile are outliers.

5.1.4 Standard Error

The more subjects we measure, the more accurate our estimation of the mean value of our population will be. The remaining standard deviation of the mean is often referred to as the standard error, and is given by

\[ se = \frac{\sigma}{\sqrt{n}} \] (5.5)

The equation for the confidence intervals for the mean value is the same as Eq. 5.4, with the standard deviation \( \sigma \) replaced by the standard error \( se \).

If you *know* the mean value and the standard deviation, you use the z-distribution and \( \sigma \). If you have to *estimate* the mean and the standard deviation, you have to use \( s \) instead of \( \sigma \), and the normal distribution has to be replaced by the so-called *t-distribution*. The T-distribution also takes the errors induces by the estimation of the mean value based on \( n \) subjects into consideration:

\[ CI_{mean} = mean \pm se \times t_{n-1,1-\alpha/2} \] (5.6)

where \( t_{n-1,1-\alpha/2} \) is the t-distribution for \( n \) subjects, at a \( \alpha \%- \) confidence level. Note that with \( n \) subjects we have \( n - 1 \) degrees of freedom, because one parameter (the mean) has to be estimated.
CHAPTER 5. STATISTICS

Figure 5.4: The Standard Deviation (blue lines) describes the uncertainty of the data. The Standard Error of the Mean (red lines) describes the uncertainty of the mean value.

The required t-value can be obtained with the Matlab command `icdf` (for inverse cumulative distribution function). In our case, where we have 5 subjects, and $\alpha = 0.05$

$$
>> tval = icdf('t', 0.975, 4)
tval = 2.7764
$$

$$
>> tval = icdf('t', 0.975, 20)
tval = 2.0860
$$

For 5 subjects we get a value of 2.77, and for 21 subjects a value of 2.09, which is already pretty close to the corresponding value from the normal distribution, 1.96.
5.2 Statistical Data Analysis

5.2.1 Null-Hypothesis

Important: First of all, you have to state explicitly which hypothesis you want to test. This has to be done before you do the experiments/analysis, and should be formulated such that the hypothesis explicitly contains the value zero or null.

In the example above, the hypothesis would be: *We hypothesize that the average weight of male Austrians minus 88.3 is equal to zero.*

5.2.2 Check Your Assumptions

In the example presented above, we have used the assumption that our data are normally distributed. Such assumptions also have to be checked - at least visually! If these assumptions don’t hold, you have to use statistical tests that don’t rely on these assumptions.

5.2.3 Hypothesis Tests

The most common statistical tests, and the only ones we are discussing here, are tests that evaluate one or two groups. We distinguish between three cases:

1. Comparison of one group to a fixed value (as in the example above).
2. Comparison between two related groups (e.g. before-after experiments, Fig. 5.5).
3. Comparison between two independent groups (see below).

These tests always return a *p-value*, which gives the probability to obtain the observed result if the null-hypothesis is true:

- If \( p < 0.05 \), we speak about a *significant difference*.
- If \( p < 0.001 \), we speak about a *highly significant difference*.
- Otherwise we state that *there is no significant difference*.

One-Sample T-Test

The first two of these cases (comparison of a group to a fixed value, or related data-sets) are tested with the *one-sample or paired-sample t-test*.\(^1\)

```matlab
>> weight = [84, 86, 88, 61, 82];
>> [h,p] = ttest(weight-88.3);
```

\begin{verbatim}
  h = 0
  p = 0.1739
\end{verbatim}

\( h = 0 \) tells us that the null hypothesis can be accepted, and \( p = 0.1739 \) tells us that the *probability that the null hypothesis is correct* is 17%. This *p-value* is typically quoted in the *Results* part of your report.

\(^1\)The T-test is sometimes also called *Student’s t-test*, because the name of the researcher who developed this test was ”Student”.
CHAPTER 5. STATISTICS

Figure 5.5: If every value in the data set "Date before" has a corresponding value in the data set "Date after", a paired t-test can detect smaller differences. Note that this is equivalent to testing if the difference between the first and the second test is significantly different from zero.

Comparison of Two Related Groups  Comparing two related groups gives the same results as comparing the difference between the groups to zero. So the following two commands give the same results (Fig.5.5):

```matlab
[h,p] = ttest(w, w_after);
[h,p] = ttest(w - w_after);
```

Note that the comparison of two related groups detects smaller differences than the comparison of two independent groups!

T-Test for Independent Samples

Two independent groups are compared with the t-test for independent samples.

Figure 5.6: ttest2 compares data sets from two different, independent groups, and provide a quantitative estimate if the data sets from the two groups are different.

For example, let us compare the weight of 10 random American males with the weight of 10 random Austrian males:

```matlab
>> weight_USA = [ 104, 86, 86, 105, 52, 80, 67, 82, 99, 102];
>> weight_A = [66, 74, 83, 77, 86, 87, 67, 78, 49, 98];
>> [h,p] = ttest2(weight_USA, weight_A);
```

h =
0

p =
0.1756

Again, the probability that the null hypothesis is true is relatively large, and the result is not significant. In order to get a significant result for a relatively small difference, we need a larger sample size:

```matlab
>> weight_USA = roundd(randn(1,100)*17 + 88.3);
>> weight_A = roundd(randn(1,100)*15 + 81);
>> [h,p] = ttest2(weight_USA, weight_A)
   h =
       1
   p =
       0.0022
```

**Hypothesis Tests with Confidence Intervals**

Instead of *p-values*, it has become recommended practice to define the significance of a test based on the confidence intervals of a tested parameter. If the null hypothesis is that the measured value is zero, the hypothesis can be accepted if the 95%-confidence interval includes zero. Otherwise, the hypothesis has to be rejected.

**One-sided T-test vs. Two-sided T-test**

In most cases that you encounter, you will use a *two-sided t-test*. However, if the data set in one group can only be larger (or only be smaller) than the data set from the other group, you use a *one-sided t-test*.

![Figure 5.7: If we do NOT know if the data set from one group is larger or smaller than the data set from the other, we have to use two-sided t-tests. But if we know in advance that the values from the data set from the second group can only be larger than those from the first group, a one-sided t-test can be used. Note that this can determine if a measured difference is significant or not!](image-url)
5.3 Exercises: Statistical analysis of data

5.3.1 Exercise 1: Analyze data
Let us assume that the weight of 100 random, 15 year old children in Tirol and in Vienna are:

\[
\begin{align*}
tirol &= 7 \times \text{randn}(100, 1) + 60; \\
viena &= 10 \times \text{randn}(100, 1) + 55;
\end{align*}
\]

Then the Tyrolian children try out a banana-diet: for one week, they eat only bananas for breakfast, and otherwise their normal food. After one week these 100 children weigh

\[
tirol\_after = tirol - 1.5 + 1.0 \times \text{randn}(\text{length}(tirol));
\]

- calculate mean, median, standard deviation (SD), and standard error for the first 10 Tyrolian children.
- calculate mean, median, standard deviation (SD), and standard error for all Tyrolian children

5.3.2 Exercise 2: Plot data
- Plot mean +/- SD , and mean +/- SE for the first 10 Tyrolian children
- Plot mean +/- SD , and mean +/- SE for all Tyrolian children

5.3.3 Exercise 3: Compare groups
- Find out if the difference between the children in Tirol and the children in Vienna is significant.
- Find out if the weight of the Tyrolian children after the banana-diet is less than the weight before the diet. Note that these data are from the same children as before the diet, in the same order.
Chapter 6

Parameter fitting

When we record data, we often obtain data that can be approximated nicely by a straight line. If this is the case, it tremendously simplifies our description of the data set: we do not need to know every single data point any more, but only the two coefficients defining the best-fit line to the data points.

$$y = k \times x + d$$

Figure 6.1: To obtain the best-fit line to noisy data, we want to find the model coefficients $k$ and $d$ that define our data-model $y = k \times x + d$.

The following chapter describes how to obtain the model coefficients for simple linear models, how accurately we know these coefficients, and how this concept can be used to describe much more complex patterns, such as polynomials, circles, etc.
6.1 Fully determined linear model

A linear model describes a linear relationship between a dependent variable and a number of independent variables. In the simplest case, there is one dependent variable $y$ and one independent variable $x$. We can write the linear model as

$$y = mx + c$$  \hspace{1cm} (6.1)

where $m$ and $c$ are the two free parameters. This is a line in a plane, with $m$ the slope, and $c$ the $y$-intercept. If we have two different data points relating the dependent and independent variables, then these two free parameters are fully determined. For example, if $P_1 = (x_1/y_1)$ and $P_2 = (x_2/y_2)$, then we can calculate that

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{and} \quad c = \frac{y_1 x_2 - x_1 y_2}{x_2 - x_1}$$  \hspace{1cm} (6.2)

In the general case, there is one dependent variable $y$ and $k$ independent variables $x_i$. The linear model then becomes

$$y = c + \sum_{d=1}^{k} m_d x_d$$  \hspace{1cm} (6.3)

where $c, m_1, ..., m_k$ are the $k + 1$ free parameters. If we have $k + 1$ different data points that lie exactly on an $k$-dimensional hyperplane, then the free parameters are fully determined.

6.1.1 Reminder: Lines in a plane

A line can either be represented by its slope and $y$-intercept, as shown in 6.1. Alternatively, it can be characterised by a point on the line, $\vec{x}_1$, and the line’s normal vector, $\vec{n}$:

$$\vec{n} \cdot \vec{x} = \vec{n} \cdot \vec{x}_1$$  \hspace{1cm} (6.4)

or

$$ax + by = c, \ \text{with} \ \vec{n} = \left( \begin{array}{c} a \\ b \end{array} \right)$$  \hspace{1cm} (6.5)
6.2 Overdetermined linear model

Mostly, we have more data points than free parameters. In this case, the model is *overdetermined* by the data, so that it is not possible to fit the model exactly to all of the data points. Instead, we try to find a best fit model, which is the model with the minimum fitting error. Often, the fitting error is taken to be the *least squares estimator*.

6.2.1 Residual

The difference between a data value and the corresponding model value is referred to as a *residual* (Figure 6.3). If we have a data point where the dependent variable has the value $y_i$ and the independent variables have value $x_{id}$, then the residual $e_i$ for the general linear model is

$$e_i = y_i - \left( c + \sum_{d=1}^{k} m_d x_{id} \right)$$

(6.6)

Figure 6.3: *Residuals* are the difference between the data and the modelled value $y = k \ast x + d$.

Note that the values along the x-axis are fixed!

6.2.2 Least squares estimators

If there are $n$ data points, the sum of the squared residuals is

$$E(x_{id}, y_i, c, m_d) = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} \left( y_i - \left( c + \sum_{d=1}^{k} m_d x_{id} \right) \right)^2$$

(6.7)

The *least squares estimators* are the values of $c$ and $m_1, ..., m_k$ that minimise $E$. To determine the value of the least squares estimators $\hat{c}$ and $\hat{m}_1, ..., \hat{m}_k$, it is necessary to locate the minimum of $E$ by finding where the following partial derivatives are zero:
0 = \frac{\partial E}{\partial \hat{c}, \hat{m}_d} = -2 \sum_{i=1}^{n} \left( y_i - \left( \hat{c} + \sum_{d=1}^{k} \hat{m}_d x_{id} \right) \right) \tag{6.8}

0 = \frac{\partial E}{\partial m_{ip}} \bigg|_{\hat{c}, \hat{m}_d} = -2 \sum_{i=1}^{n} \left( y_i - \left( \hat{c} + \sum_{d=1}^{k} \hat{m}_d x_{id} \right) \right) x_{ip} \text{ for all } p

This linear system of equations can then be solved to find the values of the least squares estimators.

### 6.2.3 Reminder: Ordinary Least Squares

The method of ordinary least squares can be used to find an approximate solution to over-determined systems. For the system \( \mathbf{A} \cdot \hat{\mathbf{p}} = \mathbf{y} \), the least squares formula is obtained from the problem

\[
\min_{\mathbf{p}} \| \mathbf{A} \cdot \mathbf{p} - \mathbf{y} \|,
\]

the solution of which can be written with the normal equations,

\[
\hat{\mathbf{p}} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \cdot \hat{\mathbf{y}}, \tag{6.9}
\]

where \( T \) indicates a matrix transpose, provided \( (\mathbf{A}^T \mathbf{A})^{-1} \) exists (that is, provided \( \mathbf{A} \) has full column rank). With this formula an approximate solution is found when no exact solution exists, and it gives an exact solution when one does exist.
6.3 Finding least squares estimators with Matlab

6.3.1 Linear model with no intercept term

Let us start with the case where there is no intercept term (i.e., \( c \) is zero). Our model is

\[
y \approx \sum_{d=1}^{k} m_d x_d
\]

\( X \) is an \( n \times k \) matrix made of the column vectors \( x_1, \ldots, x_k \), \( m \) is a \( k \times 1 \) column vector containing the free parameters \( m_1, \ldots, m_k \), and \( y \) is an \( n \times 1 \) column vector, then we can rewrite the model as

\[
X \cdot m \approx y
\]

The least squares estimator for \( m \) can be obtained easily in Matlab by reshaping the equation:

\[
m_{\text{estimator}} = X\backslash y;
\]

Note that the "\( \backslash \)" is a backslash and not a normal divide. Alternatively, one can use the command \texttt{regress}:

\[
m_{\text{estimator}} = \text{regress}(y, X);
\]

6.3.2 Linear model with intercept

Now let us address the general linear model with intercept term:

\[
y \approx c \times \sum_{d=1}^{k} m_d x_d
\]

We rewrite this in matrix form in the following way:

\[
\vec{y} \approx \vec{M} \cdot \vec{p}
\]

(6.10)

where

\[
M = \begin{bmatrix}
x_{11} & x_{12} & \cdots & x_{1k} & 1 \\
x_{21} & x_{22} & \cdots & x_{2k} & 1 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
x_{n1} & x_{n2} & \cdots & x_{nk} & 1
\end{bmatrix}, \quad p = \begin{bmatrix}
m_1 \\
m_2 \\
\vdots \\
m_k \\
c
\end{bmatrix}, \quad y = \begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_n
\end{bmatrix}
\]

(6.11)

The matrix \( M \) is sometimes called \textit{Design Matrix}. In Matlab, we can solve this in the same way as before:

\[
M = [X \ \text{ones(size(y))}];
\]

\[
p_{\text{estimator}} = M\backslash y;
\]

Again, it is possible to use the function \texttt{regress}:

\[
p_{\text{estimator}} = \text{regress}(y, M);
\]
6.3.3 Example 1: Fitting a line

For a line-fit, Eq. 6.11 is reduced to

\[
\begin{bmatrix}
    y_1 \\
    y_2 \\
    y_3 \\
    \vdots
\end{bmatrix}
= \begin{bmatrix}
    x_1 & 1 \\
    x_2 & 1 \\
    x_3 & 1 \\
    \vdots & \vdots & \vdots
\end{bmatrix}
\cdot \begin{bmatrix}
    k \\
    d
\end{bmatrix}
\]

% Generate a noisy line, with an slope of 0.5 and a y-intercept of -30
x = 1:100;
y = -30 + 0.5*(x + 5*randn(size(x)));

% Plot the line
plot(x,y);
line(xlim, [0 0], 'LineStyle', '--');

% Calculate the best-fit line
M = [x' ones(length(x),1)];
p_estimator = M\y'; % or "p_estimator = regress(y', M);

slope = p_estimator(1) % "m"
yintercept = p_estimator(2) % "c"

An alternative way to fit a line would be to see the line as a 1st order polynomial, and use the Matlab function polyfit:

\[
p = \text{polyfit}(x, y, 1);
y\_fit = \text{polyval}(p, x);
\]

6.3.4 Example 2: Fitting a polynomial

We can use the same approach to fit a polynomial curve to the data. For example, for a quadratic relationship between \(x\) and \(y\) we have

\[
y = a \cdot x^2 + b \cdot x + c \quad (6.12)
\]

Written in matrix form, this gives

\[
\begin{bmatrix}
    y_1 \\
    y_2 \\
    y_3 \\
    \vdots
\end{bmatrix}
= \begin{bmatrix}
    x_1^2 & x_1 & 1 \\
    x_2^2 & x_2 & 1 \\
    x_3^2 & x_3 & 1 \\
    \vdots & \vdots & \vdots
\end{bmatrix}
\cdot \begin{bmatrix}
    a \\
    b \\
    c
\end{bmatrix} \quad (6.13)
\]

With this we have brought the problem into the form \(\vec{y} \approx M \cdot \vec{p}\), and we can solve it in the same way as above.

6.3.5 Example 3: Fitting a sine-wave

If we have a sinusoidal oscillation, where we know the frequency \(\omega\), and where we want to know the amplitude, the phase delay, and the offset, our function can be written as

\[
x = \text{offset} \cdot 1 + \text{amplitude} \cdot \sin(\omega t + \delta) \quad (6.14)
\]

Note that here \text{offset} and \text{amplitude} appear as linear parameters - but the phase, \(\delta\), does not. But this can be solved by expressing \(\sin(\omega t + \delta)\) by the sum of a sine- and a cosine wave:
\[ x = \text{offset} \cdot 1 + a \cdot \sin(\omega t) + b \cdot \cos(\omega t) \]  
(6.15)

From these parameters, we can find the amplitude and the phase:

\[ \text{amplitude} = \sqrt{a^2 + b^2} \]
\[ \delta = \tan^{-1} \left( \frac{b}{a} \right) \]  
(6.16)

Now all the parameters that we want to know appear in the linear (!!) relationship 6.15, and we can put together our matrix \( M \) as

\[
M = \begin{bmatrix}
1 & \sin(\omega \cdot t_1) & \cos(\omega \cdot t_1) \\
1 & \sin(\omega \cdot t_2) & \cos(\omega \cdot t_2) \\
1 & \sin(\omega \cdot t_3) & \cos(\omega \cdot t_3) \\
\vdots & \vdots & \vdots \\
1 & \sin(\omega \cdot t_n) & \cos(\omega \cdot t_n)
\end{bmatrix}
\]  
(6.17)

and \( \vec{p} \), which is here given by \( \vec{p} = \begin{bmatrix} \text{offset} \\ a \\ b \end{bmatrix} \), can again be obtained from Eq. 6.10.

```matlab
% Time
t = 0:0.1:8*pi;

% Set the parameters
freq = 0.5;
offset = 3;
delta = 45*pi/180;
amplitude = 2;
omega = 2*pi*freq;

% Simulate and plot noisy sine-wave:
y = offset + amplitude * sin(omega*t + delta) + randn(size(t)); plot(t,y)

% Fit the data
M = [ones(length(t),1) sin(omega*t') cos(omega*t')]; p = M\y';

% Extract the coefficients from the fit
found.offset = p(1);
found.amp = sqrt(p(2)^2 + p(3)^2);
found.delta = atan2(p(3), p(2))*180/pi;

% Superpose the original data with the fit
hold('on');
found.y = found.offset + found.amp*sin(omega*t + found.delta*pi/180)
plot(t, found.y, 'r')
legend('original', 'fit');xlabel('Time [s]');ylabel('Data');shg
```

### 6.3.6 Example 4: Fitting a circle

The same concept can surprisingly even be extended to find the best-fit circles to data. To do so, we rearrange the general equation for a circle:
\[ (x - x_0)^2 + (y - y_0)^2 = r^2 \]
\[ x^2 - 2xx_0 + x_0^2 + y^2 - 2yy_0 + y_0^2 = r^2 \]
\[ 2x \cdot x_0 + 2y \cdot y_0 + 1 \cdot (r^2 - x_0^2 - y_0^2) = x^2 + y^2 \]

(6.18)

This gives us \( x^2 + y^2 = M \cdot \vec{p} \), with

\[
M = \begin{bmatrix}
2x_1 & 2y_1 & 1 \\
2x_2 & 2y_2 & 1 \\
... & ... & ...
\end{bmatrix}
\]

Again, this is just a linear fit! From this we get

\[
x_0 = p(1) \\
y_0 = p(2) \\
r = \sqrt{p(3) + x_0^2 + y_0^2}
\]
6.4 Confidence Interval for Linear Regressions

Let us now generalize this concept of confidence intervals to our linear regression, and add some more information about the basic assumptions underlying the calculation of confidence intervals. In the previous section, we saw that it is possible to describe a linear trend via the least squares estimators. However, measured data always have errors superimposed on the basic trend in which we are interested. This error leads to uncertainty in the estimators.

To deduce something about the uncertainty in the estimators, we have to make assumptions about the noise in the data. Some common assumptions that are made are:

- we know the independent variables $x_1, \ldots, x_k$ exactly
- the residuals are roughly normally distributed
- the noise is only in the dependent variable $y$
- the residuals are independent of the values of $x_1, \ldots, x_k$

If any of these assumptions does not hold, then the confidence interval must be calculated differently.

The Matlab function `regress` can give you both the least squares estimators and confidence intervals for the parameters:

```
[B, BINT] = regress(y, X);
```

where $B$ indicates the least squares estimators $\hat{m}$, and $BINT$ the 95% confidence interval for $\hat{m}$. Remember, though, that Matlab calculates confidence intervals as described above, so check the assumptions first.

To specify a different confidence level, use the form

```
[B, BINT] = regress(y, X, alpha);
```

Note that the confidence interval widens as the required confidence level increases. For example, the 99.9% CIs are wider than the 95% CIs.

6.4.1 Relationship to hypothesis tests

A confidence interval for a parameter can be used directly to test the corresponding null hypothesis. With the confidence intervals described above, the null hypothesis is:

*The parameter is not significantly different from zero.*

If the confidence interval contains zero, then it is not possible to reject the null hypothesis at the used confidence level; in other words, there is then no statistical evidence to suggest that the parameter is significantly different from zero.

6.4.2 Line of best fit with no intercept term

If we use `regress` to fit a linear model with no intercept term to data in Fig. 6.5, we obtain $B = 0.755$ and $BINT = [0.748, 0.761]$.

This means that the slope of the best fit line (i.e., $k$) is most likely to be 0.755, and lies with 95% probability between 0.748 and 0.761.
6.4.3 Line of best fit with intercept term

If we use `regress` to fit a linear model with intercept term to the data in Fig. 6.6,

\[
B = \begin{bmatrix} 0.751 \\ 20.228 \end{bmatrix} \quad \text{and} \quad \text{BINT is} \quad \begin{bmatrix} 0.739 & 0.764 \\ 19.483 & 20.974 \end{bmatrix}.
\]

This means that the slope of the best fit line (corresponding to \(m\)) is most likely 0.751, and lies with 95% probability between 0.739 and 0.764; and that the offset of the line (corresponding to \(c\)) is 20.228, and lies with 95% probability between 19.483 and 20.974.

6.4.4 Significance

If both confidence for the slope are larger than 0, we speak of a *significant increase* in the data. If also both 99.9% confidence intervals are above 0, we call this a *highly significant increase*. Similarly, if both confidence intervals are below zero, our data show a *significant* or a *highly significant decrease*.

On the other hand, if the confidence intervals overlap zero, we cannot claim that our data are in- or decreasing, even if the slope is positive or negative, respectively.
6.5 Correlation Coefficient and Coefficient of Determination

6.5.1 Correlation Coefficient

The correlation coefficient $r$ is a measure of the linear correlation (or dependence) between two variables $x$ and $y$, giving a value between $+1$ and $1$ inclusive, where $1$ is total positive correlation, $0$ is no correlation, and $1$ is total negative correlation. For sample data $x_i$ and $y_i$, $r$ indicates the change in $x$ multiplied by the change in $y$, normalized by the relative spread of $x$ and $y$:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}}$$ \hspace{1cm} (6.19)

Note that the correlation coefficient is symmetric in $x$ and $y$.

For linear regression, the square of the correlation coefficient $r$ equals the coefficient of determination.

![Figure 6.7: The better the linear regression (on the right) fits the data in comparison to the simple average (on the left graph), the closer the value of $r^2$ is to one. The areas of the blue squares represent the squared residuals with respect to the linear regression. The areas of the red squares represent the squared residuals with respect to the average value (from Wikipedia)](image)

6.5.2 Coefficient of Determination

The coefficient of determination tells us how well the fitted data account for the raw data. It makes use of the fact that

$$SS_{tot} = SS_{mod} + SS_{res}$$ \hspace{1cm} (6.20)

where $SS_{tot}$ is the sum of the squares of the total deviation (Fig 6.7, left), $SS_{mod}$ the sum of the squares of the model fit, and $SS_{res}$ the sum of the squares of the residuals (Fig 6.7, right):

$$SS_{tot} = \sum_{i=1}^{n}(y_i - \bar{y})^2$$ \hspace{1cm} (6.21)

$$SS_{mod} = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2$$ \hspace{1cm} (6.22)

$$SS_{res} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$ \hspace{1cm} (6.23)
\( \bar{y} \) is the average value, and \( \hat{y} \) is the value of the fitted data that corresponds to a given \( x_i \) value.

The \textit{coefficient of determination} is defined as the amount of the sum of squares that can be explained by the regression:

\[
r^2 = \frac{SS_{mod}}{SS_{tot}} = 1 - \frac{SS_{res}}{SS_{tot}}
\]

As an example, if \( r^2 = 0.7 \), this would tell us that \textit{approximately seventy percent of the variation in the response can be explained by the line fit.}

To illustrate a range of different fits, three data sets with different amounts of noise are plotted in Fig. 6.8. All three data sets have roughly the same line of best fit, but the coefficients of determination differ.

In Matlab, the correlation coefficient \( r \) can be found by using the function \textit{corr}. The code below produces a correlation coefficient of approximately \( r = 0.997 \).

```matlab
x = 1:100;
y = 10 + 0.5*x + randn(size(x));
plot(x,y,'.');
r = corr(x', y')
```

Figure 6.8: Data with different levels of noise.
6.6 Fitting of nonlinear functions

So far, we have only considered a linear relationship between the data and the parameters. Note that even tasks such as fitting a sine-function with an offset and a phase can be expressed with linear relationships (see Example 2: Fitting a sine-wave). However, often, the relationship will be visibly nonlinear, and cannot be fit with a linear model. In this case, it is necessary to select a nonlinear model for the data, and then attempt to fit the parameters of the model.

For example, for the data in Fig. 6.9, the falling part of the curve might reflect some physical process that is naturally modelled with an exponential decay.

Figure 6.9: Plot of a variable that decays over time.

Since an exponential decay is fully defined by the starting value and time, the value of the asymptotic minimum, and the half-life, one way of fitting this curve would be to estimate these parameters independently. For example, the offset is approximately the last point in time, and the decay time is approximately given by the time it takes to decay from the maxVal to offset + (maxVal - offset) * exp(-1).

Alternatively, one could use more sophisticated methods that attempt to estimate all of the model parameters simultaneously. Such methods can be found in Matlab in the Curve Fitting Toolbox or in the Optimization Toolbox. Typically, the fits are more efficient and accurate when the user provides a good estimate of the parameters as a starting point.

An important tip is to choose a model/function that is likely to fit the data well, and if you have a choice, to reduce the number of parameters that must be estimated.

Example: Exponential Decay with Offset

For fitting an exponential decay, you first have to determine where the decay starts. If the data are not too noisy, this can be automated by finding e.g. the first value clearly below the maximum. Clearly can be e.g. 10% below the maximum

\[
\text{tStart} = \min\{\text{find}(x<\text{max}(x) \times \text{threshold})\};
\]

While the quality of the fit depends on many factors, e.g. the quality of the data, the appropriateness of the model, and the starting values, the Matlab Curve Fitting Toolbox can sometimes make things remarkably easy. For example, to fit an exponential decay to an offset can be found with
% Generate dummy data
fitTime = (0:0.01:20)';
fitVal = 3 + 4*exp(-fitTime/5)+randn(size(fitTime));

% Make the fit
f = fittype('offset + amp*exp(-x/tau)');
options = fitoptions(f);
options.StartPoint = [1,1,1];
fitted = fit(fitTime,fitVal,f, options)

fitted =
General model:
fitted(x) = offset + amp*exp(-x/tau)
Coefficients (with 95% confidence bounds):
amp = 3.964 (3.79, 4.137)
offset = 3.01 (2.899, 3.122)
tau = 5.148 (4.565, 5.732)
6.7 Exercises

Exercise 1: Line Fits

- Fit a line to the last 25 years of carbon emissions from "fossil fuels and cement", using "polyfit". Thereby, choose the values for the first data point to be (0/0).
- Fit a quadratic curve to the same data, using "polyfit".
- Plot original data, line, and quadratic curve.

Exercise 2: Confidence Intervals

Use the same data as in Exercise 1, but now also determine the 95% confidence intervals. Answer the following 2 questions:

- When you fit a line, is the slope of the line significantly rising?
- When you fit a quadratic curve, is the quadratic contribution significant?

Note: When the highest order term is determined, then all lower order terms are also included. If for instance we fit a fifth order polynomial, and only the cubic term is significant, then we would omit the higher order nonsignificant terms, but retain those terms of smaller order than the cubic.

Exercise 3: Linear Circle Fit

All the data that you need for this and the next exercises are in https://github.com/thomas-haslwanter/BSA/blob/master/Data/noisyStuff.mat. "noisyCircle" contains the x/y values of a noisy circle.

Write a function that takes these data and determines the center of the best-fit circle-center (x0/y0) and the radius r.

Exercise 4: Nonlinear Exponential Decay

"noisyExp" contains the t/x values of a noisy function that exponentially decays to a constant offset.

Write a function that takes these data and calculates the best-fit Amplitude, Offset, and Decay-time for this decay.

Exercise 5: Butterworth Filter

"noisySine" contains the t/x values of a noisy sine wave.

Smooth these data with a Savitzky-Golay filter, and superpose the filtered data with the original data. The Matlab command butter provides the [b,a] coefficients of an IIR filter corresponding to Butterworth low-pass filter. Find the coefficients for a low-pass filter with a corner frequency of 0.5 Hz, and apply this IIR filter to the noisy sine wave. Compare the output with the output of the Savitzky-Golay filter.

Exercise 6: Confidence Intervals

"noisyLine" contains the t/x values of a very noisy line.

A linear regression fit indicates that the data can be fit with a line \( y = k \times x + d \), with \( k=0.1653 \).

Are the data significantly increasing?
Chapter 7

Spectral analysis of biological signals

7.1 Introduction

Have you ever watched a sculptor at work? If not, you’ve missed the opportunity to see the sculptor coax a form out of an apparently formless chunk of raw material. Generally, the sculptor will start out by creating a very coarse approximation of a shape. Then, the shape is refined until it looks roughly like what the sculptor envisages. Finally, the sculptor gives the sculpture a few finishing touches to give it the right texture. To illustrate this process, here are a few images detailing the progress of an ice sculpture:

Figure 7.1: Images of ice carving, extracted from the video found at http://www.militarychefs.com/1A/2_Media/IceVideos/IceBirdOfPrey.html (With kind permission from James Sewson)

To generate the rough shape at the beginning, the sculptor needs to use large coarse cuts, but at the end, when the fine detail needs to be added, the sculptor uses small precise cuts.

Figure 7.2: Left) If we add up two or more sinusoidal waves, we can easily calculate the resulting signal. Right) But if we have a given signal, which waves (Frequency, Amplitude, Phase) should we add up to best approximate our signal? The Fourier Transformation solves this problem.
Thus, the final shape is generated by a combination of cuts on different scales.

In signal analysis, it is possible to use a similar approach to approximate any function of interest. Here, instead of using cuts, we use sine waves of different periods. When the correct combination of sine waves are all added together, the end result is identical to the function of interest, except maybe at isolated points (for example, when the function of interest is not continuous). The period, amplitude and phase of each component sine wave can be found by using the Fourier transformation.

An excellent overview of Fourier analysis can be found in:

7.2 Fourier integral

To transform a continuous function, one uses the Fourier Integral:

\[ F(k) = \int_{-\infty}^{\infty} f(t)e^{-2\pi ikt} dt \]  \hspace{1cm} (7.1)

where \( k \) represents frequency. The inverse transform is given by

\[ f(t) = \int_{-\infty}^{\infty} F(k)e^{2\pi ikt} dk \]  \hspace{1cm} (7.2)

Typically, the time-dependent function is written in lowercase and the Fourier transform in uppercase.

7.2.1 Different conventions

It should be noted that some authors choose to define the Fourier transform (and inverse transform) in terms of the angular frequency \( \omega = 2\pi k \). In this case, there needs to be a scaling factor that is in total \( \frac{1}{2\pi} \). The \( \frac{1}{2\pi} \) may scale the forward transform or the inverse transform, or \( \frac{1}{\sqrt{2\pi}} \) may scale both (Symmetrical Fourier Transformation).

7.2.2 Complex exponential notation

Note that \( e^{2\pi ikt} \) is simply shorthand notation for the sum of phase-shifted sine waves. To understand this, one needs to understand that:

- Any combination of a sine and cosine wave with frequency \( k \) is again a sinusoid with this frequency.
- The amplitude and the phase depend on the relative components of the sine and the cosine contribution.

Specifically,

\[ a \cos(2\pi kt) + b \sin(2\pi kt) = c \sin(2\pi kt + \phi) \]

where \( c = \sqrt{a^2 + b^2} \), and \( \phi = \tan^{-1}(b/a) \). Euler’s formula states that

\[ e^{2\pi ikt} = \cos(2\pi kt) + i \sin(2\pi kt) \]  \hspace{1cm} (7.3)

From this, it follows that sine and cosine waves can be expressed in terms of \( e^{2\pi ikt} \) and \( e^{-2\pi ikt} \):

\[ \sin(2\pi kt) = \frac{1}{2i} \left( e^{2\pi ikt} - e^{-2\pi ikt} \right) \]
\[ \cos(\omega t) = \frac{1}{2} \left( e^{2\pi ikt} + e^{-2\pi ikt} \right) \]  \hspace{1cm} (7.4)

7.2.3 Fourier transform of a constant

Let us start with the constant function

\[ f(t) = 1 \]

The Fourier transform is

\[ F(k) = \int_{-\infty}^{\infty} e^{-2\pi ikt} dt \]
This integral turns out to be the Dirac delta function \(^1\) centred at 0:

\[
F(k) = \delta(0)
\]

### 7.2.4 Fourier transform of a pure sinusoid

Now, take a simple sinusoid with frequency \(\nu\):

\[
f(t) = A \sin(2\pi \nu t)
\]

The Fourier transform is

\[
F(k) = \int_{-\infty}^{\infty} A \sin(2\pi \nu t) e^{-2\pi i k t} dt
\]

Using the complex exponential representation of a sine wave, this becomes

\[
F(k) = \int_{-\infty}^{\infty} A \left( e^{2\pi i \nu t} - e^{-2\pi i \nu t} \right) e^{-2\pi i k t} dt
\]

\[
= \frac{A}{2i} \int_{-\infty}^{\infty} \left( e^{-2\pi i(k-\nu) t} - e^{-2\pi i(k+\nu) t} \right) dt
\]

Finally, we write this in terms of Dirac delta functions:

\[
F(k) = \frac{A}{2i} \left[ \delta(k - \nu) - \delta(k + \nu) \right]
\]

---

\(^1\) The \(\delta\) function is a generalized function that is zero everywhere except at zero, with an integral of one over the entire real line.
7.3 Fourier Series

In reality, measurement signals are never infinite. They always have a start and an end. Using a little trick, such signals can be turned into periodic signals, simply by repeating the signal again and again (see Fig. 7.4).

Figure 7.4: Artificially extended signal.

It can be shown that every periodic function can be decomposed into a sum of sinusoids with frequencies that are multiples of a fundamental frequency:

$$f(t) = a_0 + \sum_{n=1}^{\infty} [a_n \cos(2\pi nk_g t) + b_n \sin(2\pi nk_g t)]$$  \hspace{1cm} (7.5)

The fundamental frequency $\omega_g$ is determined by:

$$k_g = \frac{1}{T_D}$$

where $T_D$ is the length of the data set, i.e., the length of the artificial period. The Fourier series can also be written as

$$f(t) = \sum_{n=-\infty}^{\infty} F_n e^{2\pi ink_g t}$$  \hspace{1cm} (7.6)

where the values of $F_n$ are given by

$$F_n = \frac{1}{T_D} \int_{\tau}^{\tau+T_D} f(t) e^{-2\pi int/T_D} dt$$  \hspace{1cm} (7.7)

7.3.1 Example: Square Wave

Using Fourier expansion with cycle frequency $f$ over time $t$, we can represent an ideal square wave with a peak to peak amplitude of 2 as an infinite series of the form

$$x_{\text{square}}(t) = \frac{4}{\pi} \sum_{k=1}^{\infty} \frac{\sin(2\pi (2k-1)ft)}{2k-1}$$

$$= \frac{4}{\pi} \left( \sin(2\pi ft) + \frac{1}{3} \sin(6\pi ft) + \frac{1}{5} \sin(10\pi ft) + \cdots \right)$$  \hspace{1cm} (7.8)
Figure 7.5: First four Fourier approximations for a square wave. (from Wikipedia)
7.4 Discrete Fourier Transformation

In practice, real data are sampled from a continuously varying signal, and are thus discrete and time-limited. For a data set of discrete observations, we do not need an infinite number of waves to represent the signal. Specifically, if we have \(N\) discrete data points, we only require \(N\) waves to make up our signal.

If the data are sampled with a constant sampling frequency and there are \(N\) data points,

\[
f_{\tau} = \frac{1}{N} \sum_{n=0}^{N-1} F_n e^{2\pi i \frac{n\tau}{N}} \quad \text{with } \tau = 0, \ldots, N - 1
\]  

(7.9)

The coefficients \(F_n\) can be obtained by

\[
F_n = \sum_{\tau=0}^{N-1} f_{\tau} e^{-2\pi i \frac{n\tau}{N}} \quad \text{with } n = 0, \ldots, N - 1
\]  

(7.10)

Since there are a discrete, limited number of data points and with a discrete, limited number of waves, this transform is referred to as Discrete Fourier Transform (DFT).

7.4.1 Fast Fourier Transformation

The calculation of \(F_n\) can be computationally expensive, since it requires \(O(N^2)\) multiplication operations. However, if there are exactly \(2^n\) data points, the formula for the coefficients \(F_n\) becomes much simpler, and \(F_n\) can then be computed with only \(O(N \log N)\) multiplication operations by using a Fast Fourier Transform (FFT). The difference in speed can be substantial, especially when there are many data points.

In Matlab, the Discrete Fourier Transform \(F\) of data set \(f\) is found via

\[
F = \text{fft}(f);
\]

and the inverse transform with

\[
f = \text{ifft}(F);
\]

A frequent source of confusion is the question: Which frequency corresponds to \(F_n\)? If there are \(N\) data points and the sampling period is \(T_s\), the \(n^{th}\) frequency is given by

\[
f_n = \frac{n}{N \cdot T_s}, \quad 0 \leq n \leq N - 1 \text{ (in Hz)}
\]  

(7.11)

In other words, the lowest oscillating frequency is \(\frac{1}{N \cdot T_s}\) [in Hz], i.e. \(\frac{1}{\text{Signal duration}}\), while the highest independent frequency is - due to the Nyquist-Shannon theorem - half the sampling frequency, \(\frac{1}{2T_s}\).

Example

If we record the room temperature for 1 min with a sampling rate of 100 Hz, analyze the data with the Matlab command

\[
\text{my fft} = \text{fft(data)};
\]

and obtain for the first three numbers

\[
\text{my fft}(1:3);
\]

\[
\text{ans} = \\
18000 \quad 3+i \quad 4-2i
\]
What do we know about the data?

- Since data contains $60 \times 100 = 6000$ datapoints, $my_{\text{fft}}$ contains 6000 complex numbers.
- The first value is proportional to the offset, which is $\frac{18000}{6000} = 3$.
- The amplitude of the first frequency is $\sqrt{3^2 + 1^2} = \sqrt{10}$.
- The phase-shift of the first frequency is $\theta = \text{atan}(\frac{1}{3})$ (in radians).
- The first frequency is determined by the length of the recording, and is here $f_1 = \frac{1}{60}$ [Hz].
- The amplitude, phase-shift, and frequency of the second component are $\sqrt{20}$, $\text{atan}(-\frac{2}{4})$, and $\frac{2}{60}$, respectively.
- The highest frequency component which contains new information is the Nyquist frequency, which is there $\frac{100}{2}$ [Hz].
7.5 **Power Spectrum**

Most FFT functions return the complex Fourier coefficients $F_n$. If we are only interested in the size of the contribution at the corresponding frequency, we can obtain this information by taking the square of the magnitude of the coefficients:

$$P_n = F_n \cdot F_n^* = |F_n|^2$$

(7.12)

This is the *power spectrum* of our signal, because it describes the distribution of power – which is proportional to the square of the amplitude – into the frequency components composing that signal.

We will take a closer look at a modified version of the FFT example found in the Matlab documentation:

```plaintext
Fs = 1000;  % Sampling frequency
T = 1/Fs;   % Sample time
L = 1000;  % Length of signal
t = (0:L-1)*T;  % Time vector
% Sum of a 50 Hz sinusoid and a 120 Hz sinusoid, with noise
y = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t) + 2*randn(size(t));
figure(1); subplot(4,1,1)
plot(Fs*t(1:50),y(1:50))
title('Signal Corrupted with Zero-Mean Random Noise')
xlabel('time (ms)')
NFFT = 2^nextpow2(L);  % Next power of 2 from length of y
Y = fft(y,NFFT)/L;
% Plot double-sided power spectrum
subplot(4,1,2)
f_full = Fs*linspace(0,1,NFFT);
plot(f_full, log10(abs(Y).^2))
title('Double-Sided Power Spectrum')
xlabel('Frequency (Hz)')
ylabel('log Power')
% Plot single-sided power spectrum
subplot(4,1,3)
f_half = Fs/2*linspace(0,1,NFFT/2+1);
plot(f_half, log10(2*abs(Y(1:NFFT/2+1)).^2))
title('Single-Sided Power Spectrum')
xlabel('Frequency (Hz)')
ylabel('log Power')
% Periodogram
subplot(4,1,4)
[Pxx,w] = periodogram(y,[],[],Fs);
plot(w,Pxx)
title('Periodogram')
xlabel('Frequency (Hz)')
ylabel('Power Spectral Density')
```

This code produces the graphs in Fig. 7.6.

The top curve in Fig. 7.6 shows the signal. Note that by eye it is pretty much impossible to clearly discern the main frequency components in that signal. We need the Fourier transform to extract that from the data.

The second curve shows the *double-sided power spectrum*. Two prominent peaks visible. These correspond to the frequencies 50 Hz and 120 Hz found in the signal. Since the input data
(i.e. the original signal) are real, the power spectrum is symmetric about the Nyquist frequency, which here is 500 Hz.

Because the two halves of the double-sided power spectrum are identical, the normal thing to do is to collapse all of the power onto the positive frequencies, which gives rise to the single-sided power spectrum, shown in the third line.

To reduce the noise in the power spectrum, a procedure called Welsh-Periodogram can be employed, resulting in the bottom graph. Thereby the data set is broken down into a number of separate pieces, the power spectrum is calculated for each of them, and then the resulting power spectra are averaged. The resulting power signal is normalized and plotted on a linear scale (Power Spectral Density - PSD). Assuming that the frequency distribution stays approximately constant, this enhances the dominant components, and reduces random noise.

### 7.6 Fourier Transform and Convolution

For short signals, the convolutions can be calculated efficiently directly (Eq. 3.4). But for longer signals, it becomes much more efficient to calculate it through the Fourier Transformation. The basis for this is given by the convolution theorem:

\[
F\{f \ast g\} = F\{f\} \cdot F\{g\} \tag{7.13}
\]

where \(F\) denotes the Fourier transform, the left side indicates the cross-correlation, and \(\cdot\) denotes point-wise multiplication.

By applying the inverse Fourier transform \(F^{-1}\), we can write:

\[
f \ast g = F^{-1}\{F\{f\} \cdot F\{g\}\} \tag{7.14}
\]
7.7 The Next Level

7.7.1 Short Time Fourier Transform (STFT)

While the Fourier Transform provides information about the frequency content of the whole file, we often want to know when a certain frequency contribution occurs, and how strong it is at each point in time. This information can be obtained by applying temporal filters to the data, and is referred to as Short Time Fourier Transform (STFT). An example is the analysis of auditory signals, such as the one in Fig. 7.7.


Figure 7.7: Spectrogram of the German vowels "a,e,i,o,u". These correspond approximately to the vowels in the English words "hut, hat, hit, hot, put". Calculated using the Matlab command "spectrogram(data, 512,256, 512, fs)".
7.8 Exercises

Exercise 1: DFT - Frequency Domain – > Time Domain

Calculate the inverse Fourier Transform of the following data, and explain what happens:

1. \( fd = \text{zeros}(1,101); \, \text{fd}(11) = 1; \, \text{fd}(92) = 1; \)
2. \( fd = \text{zeros}(1,101); \, \text{fd}(11) = i; \, \text{fd}(92) = -i; \)
3. \( fd = \text{zeros}(1,101); \, \text{fd}(11) = i; \, \text{fd}(92) = i; \)
4. \( fd = \text{zeros}(1,101); \, \text{fd}(11) = 1; \, \text{fd}(92) = 1; \, \text{fd}(1) = 50; \)

Exercise 2: Power Spectrum

Part 1

Calculate the powerspectrum of the following two sine-waves, and explain the difference.

1. \( t = (0:99)*4*pi/100; \, x = \cos(t); \)
2. \( t = (0:70)*4*pi/100; \, x = \cos(t); \)

Part 2

Calculate the powerspectrum of

\( t = 0:0.01:10; \)
\( s = \text{RandStream}('mt19937ar','Seed',1); \)
\( \text{randNr} = s.\text{randn(size(t))}; \)
\( x = \sin(2*\pi*0.3*t) + 2*\sin(2*\pi*2*t) + \text{randNr}; \)

What happens when you replace \( x \) by \( x(1:4:end) \)?
What happens when you replace \( x \) by \( x+10 \)?

Exercise 3: DFT - Time Domain – > Frequency Domain

Take the signal described below, sampled at a frequency of 100 Hz, and calculate the power spectrum. Then increase the sampling frequency to 400 Hz, by interpolating between existing data points, and re-calculate the power spectrum. What features of the power spectrum change? And why?

\( t = 0:0.01:10; \) \hspace{1cm} \% in [sec]
\( s = \text{RandStream}('mt19937ar','Seed',1); \)
\( \text{randNr} = s.\text{randn(size(t))}; \)
\( x = 10 + \sin(2*\pi*0.3*t) + 2*\sin(2*\pi*2*t) + \text{randNr}; \)

Exercise 4: Handwork

For

\( t = 0:0.1:10; \)
\( x = \sin(t) + 3*\cos(3*t); \)

calculate the Discrete Fourier Transform by hand, and compare it to \( \text{fft}(x) \).
Chapter 8

Mechanical Systems

The principle of human motor control can be summarized with the scheme in Figure 8.1. The individual elements of this control loop can then be simulated individually, and often linear time invariant systems form a good first approximation.

Figure 8.1: Human Motor Control: The dotted lines indicate the feed-forward control pathway, and the gray lines the feedback control pathway. Elements that form part of both are shown in solid black lines.
8.1 Transfer Functions for Linear Time Invariant Systems

Linear Time Invariant Systems (LTIs) are systems that can be described by a first order differential equation.

**Linearity** The definitive test for a linear system is that if input \( x_1(t) \) produces output \( y_1(t) \) and \( x_2(t) \) produces \( y_2(t) \), then the input \( ax_1(t) + bx_2(t) \) must produce the output \( ay_1(t) + by_2(t) \). This is *superposition* and is a property of linear systems.

**Time invariance** This means that the system coefficients do not change for the period of our investigation.

These systems have the advantage that a sine input always leads to a sine output, with the same frequency.

![Figure 8.2: Amplitude and Phase of a sine-wave.](image)

Oscillations can be described elegantly with complex exponentials, because Euler’s formula tell us:

\[
e^{j\omega t} = \cos(\omega t) + j \sin(\omega t)
\]  

(8.1)

Note that the only purpose of using complex numbers is to keep the arithmetic as simple as possible. Using complex numbers, if the input to our system is \( e^{j\omega t} \), the output must have the form

![Figure 8.3: Representation of a complex number in polar coordinates.](image)

\[
\text{out}(t) = r \cdot e^{i\delta} \cdot e^{i\omega t} = G(i\omega) \cdot e^{i\omega t}
\]  

(8.2)

\( r \) quantifies the change in amplitude (or the gain), and \( \delta \) the phase shift introduced by our system. So with one complex number, \( G(i\omega) \), we can completely characterize the effect of our
8.1. TRANSFER FUNCTIONS FOR LINEAR TIME INVARIANT SYSTEMS

system on a sine input with that frequency. $G(i\omega)$ is therefore called the transfer function of our system.

\[ X(s) \xrightarrow{G(s)} Y(s) \]

Figure 8.4: Here $s$ indicates the complex frequency $i\omega$.

8.1.1 Superpositions

All linear systems obey superposition. A linear system is one described by linear equations. $y = kx$ is linear ($k$ is a constant). $Y = \sin(x)$, $y = x^2$, $y = \log(x)$ are obviously not. Even $y = x + k$ is not linear, one consequence of superposition is that if you double the input ($x$), the output ($y$) must double and, here, it doesn’t. The differential equation

\[
a\frac{d^2x}{dt^2} + b\frac{dx}{dt} + cx = y
\]

is linear.

\[
a\frac{d^2x}{dt^2} + b\frac{dx}{dt} + cx^2 = y
\]

is not, for two reasons which I hope are obvious.

So, if we can break down $f(t)$ into a bunch of sine waves, the Transfer Function $G(s)$ for a linear system can quickly give us the gain and phase for each sine wave or harmonic. This will give us all the output sine waves, and all we have to do is add them all up and voila! the desired output!

This is illustrated in Fig. 8.5. The input $f(t)$ (the square wave is only for illustration) is decomposed into the sum of a lot of harmonics on the left using the Fourier Transformation to find their amplitudes. Each is passed through $G(j\omega)$. $G(jk\omega_0)$ has a gain and a phase shift which. The resulting sinusoids $G(jk\omega_0)\sin(k\omega_0t)$ can then all be added up as on the right to produce the final desired output shown at lower right.

Fig. 8.5 illustrates the basic method of all transforms including Laplace transforms so it is important to understand the concept (if not the details). In different words, $f(t)$ is taken from the time domain by the transform into the frequency domain. There, the system’s transfer function operates on the frequency components to produce output components still in the frequency domain. The inverse transform assembles those components and converts the result back into the time domain, which is where you want your answer. Obviously you couldn’t do this without linearity and superposition.
Figure 8.5: The basic idea underlying Linear Systems Analysis.
8.2 Laplace Transformations and Applications

In the simulation of mechanical systems (or differential equations in general), this conversion from the time to the frequency domain is typically performed with the Laplace Transformation. Given any time course \( f(t) \), its Laplace transform \( F(s) \) is

\[
F(s) = \int_0^\infty f(t) \cdot e^{-st} \, dt \quad (8.3)
\]

while its inverse is

\[
f(t) = \int_S F(s) \cdot e^{st} \, ds \quad (8.4)
\]

where the integration is over the s-plane, \( S \). \( s \) is sometimes referred to as complex frequency.

Laplace Transformations are a generalization of Fourier Transforms. With Fourier Transformations we have dealt only with sine waves, \( e^{j\omega t} \). Put another way, we have restricted \( s \) to \( j\omega \) so that \( e^{st} \) was restricted to \( e^{j\omega t} \). But this is unnecessary, we can let \( s \) enjoy being fully complex or \( s = \sigma + j\omega \). This greatly expands the kinds of functions that \( e^{st} \) can represent.

![Figure 8.6: The Laplace Transform uses exponentially changing sinusoids.](image)

Fig. 8.6 is a view of the s-plane with its real axis (\( \sigma \)) and imaginary axis (\( j\omega \)). At point 1, \( \omega = 0 \) and \( -\sigma \) is negative so \( e^{st} = e^{-\sigma t} \), which is a simple decaying exponential as shown. At points 2 and 3 (we must always consider pairs of complex points recall from Eq. 7.4 that it took an \( e^{j\omega t} \) and an \( e^{-j\omega t} \) to get a real \( \sin \omega t \) or \( \cos \omega t \)) we have \( -\sigma < 0 \) and \( \omega \neq 0 \), so \( e^{-\sigma t}e^{j\omega t} \) is a damped sine wave as shown. At points 4 and 5, \( \sigma = 0 \) so we are back to simple sine waves. At points 6 and 7, \( \sigma > 0 \) so the exponential is a rising oscillation.

At \( 8, \sigma > 0, \omega = 0 \) so we have a plain rising exponential. So Fig. 8.6 shows the variety of waveforms represented by \( e^{st} \).
So 8.4 says that $f(t)$ is made up by summing an infinite number of infinitesimal wavelets of the forms shown in Fig. 8.6, $F(s)$ tells you how much of each wavelet $e^{st}$ is needed at each point on the $s$-plane. That weighting factor is given by the transform 8.3. In terms of Fig. 8.5, $f(t)$ is decomposed into an infinite number of wavelets as shown in Fig. 8.6, each weighted by the complex number $F(s)$. They are then passed through the transfer function $G$ which now is no longer $G(j\omega)$ (defined only for sine waves) but $G(s)$ defined for . The result of $F(s)G(s)$ which tells you the amount of $e^{st}$ at each point on the $s$-plane contained in the output. Using 8.4 on $F(s)G(s)$ takes you back to the time domain and gives you the output.

For us, the most important aspect of the Laplace transformation is

\[ \frac{dx(t)}{dt} \xrightarrow{\text{Laplace Transform}} s \cdot X(s) - x(0) \quad (8.5) \]

As a reminder, in your Mathematics lecture, the following notation was used to express the same equation:

\[ \mathcal{L}[y'(x)] = s\mathcal{L}[y(x)] - y(0) \]

This equation states that a Laplace transformation converts a differential equation (as a function of time) into an algebraic equation, thereby allowing us to easily solve the differential equation in the frequency domain.

### 8.2.1 Parallel Combination of Transfer Functions: Simple Muscle Model

In these systems one is concerned with force, displacement and its rate of change, velocity. Consider a simple mechanical element – a spring.

![Spring diagram](image)

**Figure 8.7:** $F$ is the force, $x$ is the length, and $k$ the spring constant.

**Hook’s Law states**

\[ F = kx \quad (8.6) \]

where $k$ is the spring constant.

Another basic mechanical element is a viscosity typical of the shock absorbers in a car’s suspension system, a hypodermic syringe, or of a muscle.

![Viscosity diagram](image)

**Figure 8.8:** Plunger in a cylinder, with a viscosity $r$. 
The relationship is

\[ F = r \frac{dx}{dt} \rightarrow rs\ddot{x} \]  

(8.7)

Where \(\ddot{x}\) indicates the Laplace-transformed variable. That is, a constant force causes the element to change its length at a constant velocity. \(r\) is the viscosity. The element is called a dashpot.

Let us put the two elements together: this combination is sometimes referred to as Voigt-Element, and is a good first approximation for a muscle model. The force \(F\) acting on the mass is divided between the two elements, and – since the two elements are arranged in parallel – the overall force is \(F = F_k + F_r\), or

\[
\tilde{F}(s) = (k + rs)\tilde{x}(s)
\]  

(8.8)

Figure 8.9: The parallel combination of a spring and a damper is called Voigt-Element.

If we take \(F\) to be the input and \(x\) the output

\[
\tilde{x}(s) = G(s) = \frac{1}{sr + k} = \frac{1/k}{sT + 1} = \frac{1/k}{sT + 1}
\]  

(8.9)

where \(T = r/k\) is the system time constant. This is called a first order lag. Fig. 8.9 is a simplified model of a muscle.

**Damped Oscillator**

The next level of realism (or complexity) is the addition of a mass \(m\) (Fig. 8.10). Compared to the Voigt element, we now also have to include the inertial force, \(F = m\frac{dx}{dt^2}\)

Figure 8.10: \(F\) is the force, \(x\) is the length.

In that case the differential equation that describes the movement of the mass is now given by
\[
\frac{d^2x}{dt^2} + \frac{dx}{dt} + kx = F(t) \quad (8.10)
\]

Applying the Laplace transformation gives us the algebraic equation

\[
ms^2\tilde{x}(s) + rs\tilde{x}(s) + k\tilde{x}(s) = \tilde{F}(s) \quad (8.11)
\]

Writing output over input, we get the transfer function for a damped oscillator

\[
\frac{\tilde{x}}{\tilde{F}} = \frac{1}{ms^2 + rs + k} \quad (8.12)
\]

### 8.2.2 Serial Combination

If you have two blocks or transfer functions in cascade, \(G(j\omega)\) and \(H(j\omega)\), the combined transfer function is just their multiplication

\[
out = H \ast G \ast in \quad (8.13)
\]

Since we are only dealing with linear systems, the sequence can be inverted \(G \ast H = H \ast G\).

For graphical representations (such as the Bode Plot below) it can be convenient to plot the logarithm of the transfer gain, since the log of the combined function just the sum of the logs:

\[
\log(G(j\omega) \times H(j\omega)) = \log(G(j\omega)) + \log(H(j\omega)) \quad (8.14)
\]
8.3  Bode Diagram

When we plot the transfer function of a first order lag on a linear-linear scale, we get

Figure 8.12: You can see that as frequency $\omega = 2\pi f$ goes up, the gain goes down, approaching zero, and the phase lag increases to 90 deg.

Bode decided to plot the gain on a log-log plot. By doing this you stretch out the low frequency part of the $\omega$ axis and get a clearer view of the system’s frequency behavior. The result will appear:

An interesting frequency is $\omega = 1/T$, since then $|G| = 1/\sqrt{2} = 0.707$ and $\angle G = -\tan^{-1}(1) = -45^\circ$. Below this frequency, $\log(|G|)$ can be closely approximated by a horizontal straight line at zero ( $\log(1)=0$ ). At high frequencies, above $\omega = 1/T$, $|G|$ falls off in another straight line with a slope of 20dB/dec, which means it falls by 10 (20dB) if $\omega$ increases by 10. The phase is a linear-log plot. The main point in Fig. 8.13 is a very simple way to portray what this lag element does to sine wave inputs ( $e^{j\omega t}$ ) of any frequency.
CHAPTER 8. MECHANICAL SYSTEMS

8.4 Implementation of Simulations

For Linear, Time-Invariant systems (LTI systems), the input and output have a simple relationship in the frequency domain:

\[ \text{Out}(s) = G(s) \ast \text{In}(s) \]

where the transfer function \( G(s) \) can be expressed by the algebraic function

\[ G(s) = \frac{\text{num}(s)}{\text{den}(s)} = \frac{n(0) \ast s^0 + n(1) \ast s^1 + n(2) \ast s^2 + \ldots}{d(0) \ast s^0 + d(1) \ast s^1 + d(2) \ast s^2 + \ldots} \]

In other words, specifying \( \vec{n} \) and \( \vec{d} \), the coefficients of the numerator and denominator, uniquely characterizes the transfer function. This notation is used by some computational tools to simulate the response of such a system to a given input.

Different tools can be used to simulate such a system. For example, the response of a low-pass filter with a time-constant of 7 sec to an input step at 1 sec has the following transfer function

\[ G(s) = \frac{1}{7s + 1} \]

and can be simulated as follows:

8.4.1 Simulink

8.4.2 Matlab - Commandline

If you work on the command line, you can use the Control System Toolbox of MATLAB.

```matlab
% Define the transfer function
num = [1];
tau = 7;
den = [tau, 1];
mySystem = tf(num, den)

% Generate an input step
t = 0:0.1:30;
```
8.4. IMPLEMENTATION OF SIMULATIONS

inSignal = zeros(size(t));
inSignal(t>=1) = 1;

% Simulate and show the output
[outSignal, tSim] = lsim(mySystem, inSignal, t);
plot(t, inSignal, tSim, outSignal);

8.4.3 Python

In Python you can do the same thing with the module signal of the Python package SciPy.

```python
# Import required packages
import numpy as np
import scipy.signal as ss
import matplotlib.pyplot as mp

# Define transfer function
num = [1]
tau = 7
den = [tau, 1]
mySystem = ss.lti(num, den)

# Generate inSignal
t = np.arange(0,30,0.1)
inSignal = np.zeros(t.size)
inSignal[t>=1] = 1

# Simulate and plot outSignal
tout, outSignal, xout = ss.lsim(mySystem, inSignal, t)
mp.plot(t, inSignal, tout, outSignal)
mp.show()
```
Chapter 9

Graphical User Interfaces (GUIs)

9.1 Callbacks

When you have a button on your figure, you must tell Matlab what it should do when that button is pressed. This is done through callbacks. Callbacks are nothing else but functions that get executed when you do something. Note that this something can be

- The push of a button
- Moving a slider
- Clicking the mouse
- Typing a key
- Closing the window
- ...

Note that this is the epitome of object-oriented programming!

You see that there are many such callbacks - and in general you just dont want to deal with them. To make is simpler for you to nevertheless make simple graphical applications, Matlab provides the program GUIDE.

![Callback diagram](image)

Figure 9.1: A Callback is a request for performing a certain task or action.
9.2 Tips for the User-Interface

- Who is using the program? Is it you (or some other expert in the field)? Or is it someone who has no experience in the application?

- Is this program used on a daily basis? In that case it should be optimized for efficiency.

- Or is it just once-in-a while? In that case the interface should be as self-explanatory as possible.

9.3 Matlab Graphics

9.3.1 Handles

When you are drawing things in Matlab, you need to have some way to control the looks of your Window (e.g. the background color, if a menu is shown, etc.), the thickness of the Lines you draw, the font of your Text labels, etc. The way this is commonly done in Matlab (and also in other graphical environments) is through Handles.

Figures

For example, the figure handle controls parameters like the

- Location of the window on the screen.
- Background color
- Title of the Window
- If it is visible
- ...

The following commands are helpful for working with the figure handle:
### 9.4. Layout of Scientific Graphs

Try to make your printouts such that when you look at the printout in one year from now, you will still know what is shown there (Fig. 9.4).

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Cur_FigHandle = gcf</code></td>
<td>Gets the current figure handle</td>
</tr>
<tr>
<td><code>get(gcf)</code></td>
<td>Gets the properties of the current figure</td>
</tr>
<tr>
<td><code>set(gcf)</code></td>
<td>Shows you the options of the properties that you can set for your current figure</td>
</tr>
<tr>
<td><code>Cur_Position = get(gcf, Position)</code></td>
<td>Get the position of the current figure</td>
</tr>
<tr>
<td><code>set(gcf, Position,[1 1 720 150])</code></td>
<td>Set the new position of the current figure</td>
</tr>
<tr>
<td><code>set(gcf, 'Tag', 'MyFigure')</code></td>
<td>Set the Tag property of the current figure</td>
</tr>
<tr>
<td><code>findobj(Tag, MyFigure)</code></td>
<td>Find something (here the figure with the Tag MyFigure)</td>
</tr>
<tr>
<td><code>set(gcf, UserData, [MyXData; MyYData])</code></td>
<td>Set the UserData property of the current figure</td>
</tr>
</tbody>
</table>

Note that most of these commands are not specific for the figure handle. For example, you can also use the command `findobj` to find a line handle. Note that you can have as many figures as you want. The current figure is the one on which you draw when you type for example `plot(1:10)`.

### Axes

Similarly, the command `gca` provides the handle for the current axis. Note that there is a hierarchy: Each figure can have many axes, and each axis belongs to exactly one figure. Each axis can have many lines, and many text objects.

### Lines

For example, the command

```matlab
lineHandle = plot([0 1 2 3 4], [0 1 4 9 16])
LineHandle_Elements = get(lineHandle);
```

produces a structure with the following elements

If you want to modify any of these elements you can do so with the set command:

```matlab
set(lineHandle, 'LineWidth', 2,
     'Color', [1 0 0]);
```

draws a thick, red line.
CHAPTER 9. GRAPHICAL USER INTERFACES (GUIS)

Figure 9.3: Elements of a Matlab line-handle. Important fields are marked. Note that no "Tag" (i.e., a name that can be used to identify the handle) has been given yet, and that the field "User Data" (which can store any element you want it to) is still empty.

Figure 9.4: Graphs should always contain enough information to be understood on their own, without any external additional information.
Python

Python is free, consistently and completely object oriented, and has a large number of (free) scientific toolboxes (e.g. http://www.scipy.org/). It is used by Google, NASA, and many others. Information can be found under http://www.python.org/. If you want to use Python for scientific applications, currently the best way to get started is WinPython, or the free Anaconda version from Continuum Analytics. These distributions are all free, and contain the complete scientific and engineering development software for numerical computations, data analysis and data visualization based on Python programming language, Qt graphical user interfaces, and the interactive scientific development environment Spyder. If you already have experience with Matlab, the article NumPy for Matlab Users provides an overview of the similarities and differences between the two languages.

A.1 Examples

The following two Python modules give an example of how Python can handle matrix multiplications and graphics simply and efficiently:

A.1.1 Example 1: Define Rotation Matrices

```python
# rot_mat.py  Definition of 3D rotation matrices

author: Thomas Haslwanter
ver:  1.1
date:  Sept-2017

from numpy import pi, sin, cos, array

def R1(psi):
    """Rotation about the 1-axis""

    # Convert from degrees into radian:
    psi = psi * pi/180;

    R = array([[1, 0, 0],
               [0, cos(psi), -sin(psi)],
               [0, sin(psi), cos(psi)]])

    return R
```
```python
def R2(phi):
    """Rotation about the 2-axis""
    # convert from degrees into radian:
    phi = phi * pi/180;
    R = array([[cos(phi), 0, sin(phi)],
                [0, 1, 0],
                [-sin(phi), 0, cos(phi)]])
    return R

def R3(theta):
    """Rotation about the 3-axis""
    # convert from degrees into radian:
    theta = theta * pi/180;
    R = array([[cos(theta), -sin(theta), 0],
                [sin(theta), cos(theta), 0],
                [0, 0, 1]])
    return R

A.1.2 Example 2: Select between two choices

```
A.1. EXAMPLES

Figure A.1: Screenshot of Spyder, a programming environment that is very similar to Matlab but leverages on IPython.

```python
plt.plot(x,y)
plt.xlabel('Time [sec]')
plt.ylabel('Values')
plt.show()

def main():
    myInput = input('Do you want to "r"otate a vector, or "p"lot a line? ')
    if myInput == 'r':
        rotateVector()
    else:
        plotLine()

if __name__ == '__main__':
    main()
```

The Figure A.1 shows a screen-shot of this code in the Spyder development environment. For interactive work, similar to Matlab, the IPython console can facilitate a smooth transition from Matlab to Python.
Appendix B

Matlab Programs

Listing B.1: DataGen.m

% DataGen Generate sample data in various formats
%
% Output:
% data.csv ....... comma-separated file
% data_tab.txt .... tab-separated file
% data.txt ....... tab-separated file, with header
% data.xls ....... Excel file
% data.mat ....... Matlab file
%
% THH, Sept-2016
% Ver 1.2
%***************************************************************************

%% Set the parameters for a noisy sine-wave:
% Note that I set all the required values at the beginning of the script.
rate = 50; % [Hz]
freq = 2; % [Hz]
duration = 4; % [sec]
amp = 5;
delta = deg2rad(90); % Phaseshift [deg]

%% Generate the data:
% Use variables, and DO NOT hardcode values. Using variables makes the code
% easier to read.
dt = 1/rate;
t = 0:dt:duration;
data = amp * sin(2*pi*freq*t + delta) + randn(size(t));
dataMat = [t', data'];

% Show how to display formatted data
outTxt = sprintf('The first time-sample is %5.3f, and the first data-value is
%5.3f.', t(1), data(1));
disp(outTxt);

%% Write them to different files:

%% Comma-separated file:
outFile = 'data.csv';
dlmwrite(outFile, dataMat, ',');

% If you generate a new file, ALWAYS tell the user the name of the file
% that has been generated, and where in the path this file is located.
% Using a variable for the name of the out-file not only makes the code
% clearer, it also makes it possible to show the user which file has been
% generated.
disp([outFile ' has been generated in ' pwd]);

%% Tab-separated file:
outFile2 = 'data_tab.txt';
dlmwrite(outFile2, dataMat, '\t');
disp([outFile2 ' has been generated in ' pwd]);

% Tab-delimited file with header
outFile3 = 'data.txt';
fh = fopen(outFile3, 'w');
fprintf(fh, '%s
', 'This file was generated on Aug 08, 2014.');
fprintf(fh, '%s%d%s

', 'Sampling rate: ', rate, ' [Hz]');
fclose(fh);
dlmwrite(outFile3, dataMat, '-append', 'delimiter', '\t');
disp([outFile3 ' has been generated in ' pwd]);

%% Excel file:
% Requires that MS Excel is installed, if you want to generate binary Excel
% files. Note that the decimal values may be indicate by a "," instead of a
% ".", depending on the Windows Region settings!
outFileXLS = 'data.xls';
xlswrite(outFileXLS, dataMat);
disp([outFileXLS ' has been generated in ' pwd]);

%% Matlab file:
% All Matlab data structures can be saved into a binary, compressed file
% with the extension ".mat". These files have a HDF5-format, and are
% very convenient as long as you stay within Matlab.
outFileMatlab = 'data.mat';
save(outFileMatlab, 'dataMat');
disp([outFileMatlab ' has been generated in ' pwd]);
Listing B.2: DataRead.m

% DataRead Read in the data generated with DataGen.m
% Input Files:
% data.csv ...... comma-separated file
% data_tab.txt .. tab-separated file
% data.txt ...... tab-separated file, with header
% data.xls ....... Excel file
% data.mat ...... Matlab file
% ThH, Sept-2016
% Ver 1.0
%*******************************************************************************

function DataRead()
% Cycles through the different data-files
% The values from the different files should be the same.

%% Read in the CSV-file
inFile = 'data.csv';
data = dlmread(inFile);
showData(inFile, data);

%% Read in the tab-separated -file
inFile = 'data_tab.txt';
data = dlmread(inFile, '	');
showData(inFile, data);

%% Read in the tab-separated -file with header
inFile = 'data.txt';
data = dlmread(inFile, '	', 3,0); % skip 3 lines, and 0 columns
showData(inFile, data);

%% Read in data from an MS Excel file
inFile = 'data.xls';
data = xlsread(inFile);
showData(inFile, data);
end % end of "DataRead"

function showData(fileName, data)
% check the first and the last data-value
disp(['The first and last data-value in ' fileName ' are: ']);
data([1, end,:]);
end % of "showData"
Listing B.3: savgol.m

%SAVGOL Savitzky-Golay filter, for data smoothing and derivatives
% SAVGOL(X, PD, WS, D, R) applies the Savitzky-Golay filter to the data in
% X vector, or column-matrix, where
% PD is the degree of the polynomial,
% WS the (uneven!) window-size,
% D the order of the derivative (0 for smoothing), and
% R the sampling rate (only required if D>0).
% smoothed = SAVGOL(x, 3, 7, 0) to smooth data, with a cubic polynomial and a
% window-size of 7
% acc = SAVGOL(pos, 3, 11, 2, 10) to calculate the second derivative of 10-Hz
% data
%
% See also: Siegmund Brandt, Datenanalyse, pp 435
% Press et al., Numerical Recipes (2. ed), pp 650
% Savitzky and Golay: Analytical Chemistry, Vol.36, No.8, Jul64, 1627 ff.
%
% Coefficients:
% C(i,k) = i! / delta_sample * SUM(j=0,...,n) { s_inv[i][j] * k^j }
% with i = 0,.., n (= degree of polynomial fitted)
% k = -m,..,m (= number of points taken for the fit)
% and S[i][j] = SUM(l= -m,..,m) { l^(i+j) }, with i,j = 0,..,n
% Thomas, Sept-2016
% Version 3.1
%*****************************************************************

function y=savgol(x, pol_degree, win_size, deriv, rate)

% Set the default values if necessary:
if (nargin==1)
    pol_degree=2
    win_size=3
    deriv=0
    rate=1;
elseif (nargin==2)
    win_size=3
    deriv=0
    rate=1;
elseif (nargin==3)
    deriv=0
    rate=1;
elseif (nargin==4)
    rate=1;
    if deriv > 0
        disp(['Default sampling rate: ' num2str(rate)]);
    end
end

% Check the input format:
[x, num_row, num_col] = check_input(x, win_size, pol_degree, deriv);
if mod(win_size, 2)==0
    warning('win_size is even, I encrease it by 1!');
    win_size = win_size+1;
end
look = (win_size-1)/2;

% Construct Vandermonde matrix:
[a, pa, p] = get_vandermonde(win_size, pol_degree, look, deriv, rate);

% Get the coefficients for the fits at the beginning and at the end of the data:
coefs = (0:pol_degree).^sign(deriv);
coef_mat = zeros(pol_degree+1);
row = 1;
for i=deriv+1:pol_degree+1
    coef = coefs(i);
    for j=1:(deriv-1)
        coef = coef * (coefs(i)-j);
    end
    coef_mat(row,row+deriv) = coef;
    row = row + 1;
end;
coef_mat = coef_mat * rate^deriv;

% Add the first and the last point "look"-times, and handle NaNs
[x_calc, nan_index] = prepare_data(x, look);

% filter the data:
% for the convolution, the filter coefficients have to be inverted
p = p(length(p):-1:1);
y_filt = [];
for i=1:num_col
    % watch out, the data are transposed!
y_filt = [y_filt; filter(p,1,x_calc(i,:))];
end

% chop away intermediate data ...
y = y_filt(:, win_size:win_size-1+num_row)';

% ...and adjust the first and the last few data:

% filtering for the first few datapoints
y(1:look,:) = a(1:look,:) * coef_mat * pa * x(1:win_size,:);

% % smoothing for the inner interval points
% for i=look+1:n-look
%     y(i,:) = p * x(i-look:i+look,:);
% end

% filtering for the last few datapoints
y(num_row-look+1:num_row,:) = a(look+2:win_size,:) * coef_mat * pa * x(num_row-win_size+1:num_row,:);

% Put back the NaNs where they were:
if ~isempty(nan_index)
    y(nan_index) = NaN;
end

% Check the form of the input data:
function [x_out, num_row, num_col] = check_input(x_in, win_size, pol_degree, deriv)
% Determine the size of the data:
[num_row, num_col] = size(x_in);

% If necessary, bring them in column form:
if num_row < num_col
    x_out = x_in';
    [num_row, num_col] = size(x_out);
else
    x_out = x_in;
end
clear x_in;

% Check input arguments:
if (win_size>num_row)
    disp('Not enough data points!');
    disp('Unless you make "win_size" smaller, the output is empty.');
    y=[];
    return;
end
if (win_size-1<pol_degree) % The "pol_degree" of the polynomial is too high!'
    disp('Unless you make the "pol_degree" smaller, or increase "look", the output
        is empty.');
    y=[];
    return;
end
if (pol_degree<=deriv) % The "=" is only to avoid non-sensible output. The
    % code would not crash.
    disp('The "deriv" of the polynomial is too high!');
    disp('Unless you make the "deriv" smaller, or increase "pol_degree", the
        output is empty.');
    y=[];
    return;
end
if ˜mod(win_size, 2)
    disp('WIN_SIZE has to be odd!');
    return;
end

function [a, pa, p] = get_vandermonde(win_size, pol_degree, look, deriv, rate)
a = zeros(win_size, pol_degree+1);
for ii=1:win_size
    for jj=1:pol_degree+1
        a(ii,jj) = (ii-1-look)^(jj-1);
    end
end
pa = pinv(a);
p = prod(1:deriv) * rate^deriv * pa(deriv+1,:); % Savitzky-Golay Coefficients

function [x_ext, nan_index] = prepare_data(x, look)
[num_row, num_col] = size(x);

% Start and End-appendices:
x_start = zeros(num_col, look);
x_end = zeros(num_col, look);

% Handle NaNs: --------------------------------------
if any(isnan(x))
    no_num.index = isnan(x);
    no_num.start = find(diff(no_num.index)== 1);
    no_num.stop = find(diff(no_num.index)==-1)+1;
    no_num.correct_start = 0;
    no_num.correct_stop = 1;
else
    no_num.correct_start = 1;
    no_num.correct_stop = 1;
end
if no_num.stop(end) > no_num.start(end)
    no_num.correct_stop = 0;
else
    no_num.correct_stop = 1;
stop_length = length(no_num.stop)-no_num.correct_start;
for ii = 1:stop_length
    start_ind = no_num.start(ii);
    stop_ind = no_num.stop(ii+no_num.correct_start);
    x(start_ind:stop_ind) = linspace(x(start_ind), x(stop_ind), stop_ind-
        start_ind+1)';
end

% Pass back the required variables
nan_index = no_num.index;
else
    nan_index = [];
end

% For start and end: --------------------------------------
for ii = 1:look
    x_start(ii) = x(1);
    x_end(ii) = x(num_row);
end

x_ext = [x_start x' x_end];

% =============================================================================
Listing B.4: ShowMorph.m

```matlab
% ShowMorph Demo-script, showing the effects of morphological image manipulation
% ThH, Jan-2007
% Ver 1.0
% *******************************************************************
% Draw a black background ...
data = zeros(99,99);
% ... and a white square in the middle:
data(34:66,:) = 1;
data(:,1:33) = 0;
data(:,67:100) = 0;
% Then make a black point in the middle ...
data(49:51,49:51) = 0;
% ... and a white point at the edge of the image
data(85:87,85:87)=1;
% Show the image
imshow(data)
shg

% Now show the effects of dilation and erosion, in a separate figure,
% with a "Square" as a structural element.
figure
se = strel('square', 5);
subplot(2,2,1);
imshow(imerode(data, se));
title('Eroded');
subplot(2,2,2);
imshow(imdilate(data, se));
title('Dilated');
subplot(2,2,3);
imshow(imopen(data, se));
title('Opened (Dilation after Erosion)');
subplot(2,2,4);
imshow(imclose(data, se));
title('Closed (Erosion after Dilation)');
shg```
Listing B.5: deboor.m

function val = deboor(knots, cp, eval_pts, order)
% INPUT: knots knot vector
% cp control points (nx2-matrix)
% eval_pts evaluation points (column vector)
% order spline-order
% OUTPUT: val value of b-spline at eval_pts (mx2-matrix)
% Author: Jonas Ballani, modified by ThH
date: Oct-2018

num_ctrlPts = size(cp,1);
num_pltPts = length(eval_pts);

X = zeros(order,order);
Y = zeros(order,order);

% Make a knot-vector such that the b-spline starts at
% the first knot vector, and ends at the last
start_pnt = knots(1);
end_pnt = knots(end);
knots = [ones(1,order-1) * start_pnt, ...
knots, ...
ones(1,order-1) * end_pnt];

% Generate the b-spline
for pnt = 1:num_pltPts
    t0 = eval_pts(pnt);
id = find(t0 >= knots);
kk = id(end);
if (kk > num_ctrlPts)
    return;
end
    X(:,1) = cp(kk-order+1:kk,1);
    Y(:,1) = cp(kk-order+1:kk,2);
    for ii = 2:order
        for jj = ii:order
            num = t0 - knots(kk-order+jj);
            if num == 0
                weight = 0;
            else
                s = knots(kk+jj-ii+1) - knots(kk-order+jj);
                weight = num/s;
            end
            X(jj,ii) = (1-weight) * X(jj-1,ii-1) + weight * X(jj,ii-1);
            Y(jj,ii) = (1-weight) * Y(jj-1,ii-1) + weight * Y(jj,ii-1);
        end
    end
    val(pnt,1) = X(order,order);
    val(pnt,2) = Y(order,order);
end
Listing B.6: bspline.m

```matlab
function spline(n_ctrlPts, spline_order)

% Plots the B-spline-curve of n control-points.
% The control points can be chosen by clicking
% with the mouse on the figure.
% COMMAND: spline(n,order)
% INPUT: n_ctrlPts  Number of Control-Points
%        order  Order of B-Splines
%        Argument is arbitrary
%        default: order = 4

% Date: Oct-2018
% Author: Stefan Heber
% cleared up and commented by Thomas Haslwanter

%% Set and check the parameters
if nargin < 1
    n_ctrlPts = 8
end
if (nargin < 2)
    spline_order = 4
end
disp(['Note that a spline of order ' num2str(spline_order) ...
     ' is made up of polynomials of degree ' num2str(spline_order-1)]);
nplot = 500; % number of plotting-points for the spline

if (n_ctrlPts < spline_order)
    display([' !!! Error: Choose n_ctrlPts >= spline_order=',num2str(spline_order),', !!!']);
    return;
end

%% Initialize figure
figure(1);
hold on; % box on;
set(gca,'Fontsize',16);

%% Get the next control point, and plot the corresponding spline
for ii = 1:n_ctrlPts
    % Get the point
    set(gcf, 'Name', ['Choose ',num2str(ii),' th. control point']);
    cp(ii,:) = ginput(1);
    hold off;

    % Plot the control points, connected by a line
    plot(cp(:,1),cp(:,2),'k-','LineWidth',2);
    axis([0 1 0 1]);
    hold on; box on;

    if (ii >= spline_order) % when we have enough points ...
        knots = linspace(0,1,ii-spline_order+2);
        eval_pts = linspace(0,1,nplot);
        p_spl = deboor(knots, cp, eval_pts, spline_order);
        % Plot the spline
```

APPENDIX B. MATLAB PROGRAMS
plot(p_spl(:,1),p_spl(:,2),'b-','LineWidth',4);
    title(['B-Spline-curve with ',num2str(ii),' control points of order ',
          num2str(spline_order))];
end
plot(cp(:,1),cp(:,2),'ro','MarkerSize',10,'MarkerFaceColor','r');
end
Appendix C

Lecture Schedule

1. Introduction to Programming
2. Tutorial 1 (Programming)
3. FIR Filters
4. Tutorial 2 (FIR)
5. IIR Filters & Image Processing
6. Tutorial 3 (IIR & Image Processing)
7. Events in Matlab
8. Tutorial 4 (Event Finding)
9. Cross correlations
10. Tutorial 5 (Pattern Recognition)
11. Statistics-Part 1
12. Statistics-Part 2
13. Tutorial 6 Statistics
14. Line Fits
15. Tutorial 7 (Line Fits)
16. Mid-term Exam: Matlab Programming
17. Linear Regression
18. Tutorial 8 (Parameter Fits)
19. Fourier Transform
20. Tutorial 9 (FFT)
21. Powerspectrum
22. Tutorial 11 (Powerspectrum)
Index

attenuation, 24
auto-correlation, 57
B-splines, 39
Bode diagram, 105
boxplot, 63

coefficient of determination, 79
confidence interval, 62
  linear regression parameters, 77
convolution, 25, 58
convolution theorem, 94
correlation coefficient, 79
cross product, 20
cross-correlation, 55

dB, see decibel
debugger, 15
decibel, 24
decimation, 52
design matrix, 73
differentiation, 27
  central-difference, 27
cubic, 27
  first-difference, 27
Savitzky-Golay, 28

euler’s formula, 87, 98
event-detection, 47
events
  1d, 48
  2d, 54

filter
  2-dimensional, 35
Butterworth, 32
  exponential averaging, 32
FIR, 25
IIR, 31
linear, 34
median, 34
morphological, 43
moving average, 26
Savitzky-Golay, 28

Finite Impulse Response (FIR), 25

fit
  circle, 75
  line, 74
  linear model, 73
  nonlinear function, 81
  parameters, 69
  polynomial, 74
  sine, 74

Fourier Transform
  short time, 95
Fourier Transformation
  convolution, 94
  Discrete Fourier Transform (DFT), 91
  Fast Fourier Transform (FFT), 91
  Fourier integral, 87
  Fourier series, 89

frequency, 40
gain, 24
Gaussian distribution, 61
graphical user interface (GUI), 109

handle
  figure, 110
histogram, 40
hypothesis test, 65, 77

image
  closing, 43
  color, 35
dilation, 43
erosion, 43
filtering, 35
grayscale, 35
opening, 43
RGB, 35

impulse response, 25
Infinite Impulse Response (IIR), 31
intercept, 73
interpolation, 52
  cubic, 53
  linear, 52
spline, 53

kernel density estimation, 40
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laplace Transformation</td>
<td>101</td>
</tr>
<tr>
<td>leaky integrator</td>
<td>32</td>
</tr>
<tr>
<td>least squares estimators</td>
<td>71</td>
</tr>
<tr>
<td>linear model</td>
<td>70</td>
</tr>
<tr>
<td>Linear Time Invariant Systems (LTI)</td>
<td>98</td>
</tr>
<tr>
<td>loess</td>
<td>38</td>
</tr>
<tr>
<td>logical indexing</td>
<td>48</td>
</tr>
<tr>
<td>lowess</td>
<td>38</td>
</tr>
<tr>
<td>Matlab</td>
<td>8</td>
</tr>
<tr>
<td>alternatives</td>
<td>7</td>
</tr>
<tr>
<td>Cells</td>
<td>12</td>
</tr>
<tr>
<td>Documentation</td>
<td>14</td>
</tr>
<tr>
<td>Function</td>
<td>10</td>
</tr>
<tr>
<td>Help</td>
<td>8</td>
</tr>
<tr>
<td>Script</td>
<td>10</td>
</tr>
<tr>
<td>Structures</td>
<td>11</td>
</tr>
<tr>
<td>Workspace</td>
<td>9</td>
</tr>
<tr>
<td>matrix multiplication</td>
<td>20</td>
</tr>
<tr>
<td>median</td>
<td>63</td>
</tr>
<tr>
<td>morphological operations</td>
<td></td>
</tr>
<tr>
<td>dilation</td>
<td>43</td>
</tr>
<tr>
<td>erosion</td>
<td>43</td>
</tr>
<tr>
<td>muscle model</td>
<td>102</td>
</tr>
<tr>
<td>normal distribution</td>
<td>61</td>
</tr>
<tr>
<td>normalization of a signal</td>
<td>58</td>
</tr>
<tr>
<td>null hypothesis</td>
<td>65</td>
</tr>
<tr>
<td>Nyquist theorem</td>
<td></td>
</tr>
<tr>
<td>OLS, ordinary least squares</td>
<td>72</td>
</tr>
<tr>
<td>oscillator</td>
<td></td>
</tr>
<tr>
<td>damped</td>
<td>103</td>
</tr>
<tr>
<td>p-value</td>
<td>65</td>
</tr>
<tr>
<td>periodogram</td>
<td>94</td>
</tr>
<tr>
<td>power spectrum</td>
<td>93</td>
</tr>
<tr>
<td>double-sided</td>
<td>93</td>
</tr>
<tr>
<td>single-sided</td>
<td>94</td>
</tr>
<tr>
<td>probability density function</td>
<td>61</td>
</tr>
<tr>
<td>quartile</td>
<td>63</td>
</tr>
<tr>
<td>quartiles</td>
<td>63</td>
</tr>
<tr>
<td>residual</td>
<td>71</td>
</tr>
<tr>
<td>scalar product</td>
<td>19</td>
</tr>
<tr>
<td>signals</td>
<td></td>
</tr>
<tr>
<td>digital</td>
<td>21</td>
</tr>
<tr>
<td>discrete</td>
<td>21</td>
</tr>
<tr>
<td>significance</td>
<td>78</td>
</tr>
<tr>
<td>Simulink</td>
<td>106</td>
</tr>
<tr>
<td>smoothing</td>
<td></td>
</tr>
<tr>
<td>Butterworth</td>
<td>32</td>
</tr>
<tr>
<td>loess</td>
<td>38</td>
</tr>
<tr>
<td>lowess</td>
<td>38</td>
</tr>
<tr>
<td>moving average</td>
<td>26</td>
</tr>
<tr>
<td>Savitzky-Golay</td>
<td>28</td>
</tr>
<tr>
<td>spectral analysis</td>
<td>85</td>
</tr>
<tr>
<td>spectrogram</td>
<td>95</td>
</tr>
<tr>
<td>Splines</td>
<td>39</td>
</tr>
<tr>
<td>splines</td>
<td>52</td>
</tr>
<tr>
<td>standard deviation</td>
<td>61</td>
</tr>
<tr>
<td>population</td>
<td>61</td>
</tr>
<tr>
<td>sample</td>
<td>61</td>
</tr>
<tr>
<td>standard error</td>
<td>63</td>
</tr>
<tr>
<td>statistics</td>
<td>61</td>
</tr>
<tr>
<td>STFT: Short Time Fourier Transform</td>
<td>95</td>
</tr>
<tr>
<td>structural element</td>
<td>43</td>
</tr>
<tr>
<td>t-test</td>
<td>65</td>
</tr>
<tr>
<td>independent sample</td>
<td>66</td>
</tr>
<tr>
<td>one-sample</td>
<td>65</td>
</tr>
<tr>
<td>one-sided</td>
<td>67</td>
</tr>
<tr>
<td>paired-sample</td>
<td>65</td>
</tr>
<tr>
<td>two-sided</td>
<td>67</td>
</tr>
<tr>
<td>time invariance</td>
<td>98</td>
</tr>
<tr>
<td>transfer function</td>
<td>24</td>
</tr>
<tr>
<td>trigonometry</td>
<td>21</td>
</tr>
<tr>
<td>vector product</td>
<td></td>
</tr>
<tr>
<td>see cross product</td>
<td></td>
</tr>
<tr>
<td>Voigt-Element</td>
<td>103</td>
</tr>
</tbody>
</table>