Contents

Python Primer

Python Primer

The following pages provide the fundamentals to learn and use Python in digital signal processing. After the syntactic basics have been introduced, the most important packages for the use of Python in science and engineering are presented. It is beneficial to have a Python interpreter at your side in order to reproduce and comprehend the examples that are made. In case that you do not understand certain phenomenons at first instance, it is good practice to think of new test examples.

P.1 Installation and Set-up

As always when you are up to learn a new programming language, a few steps are required to set up everything on your machine. For Python, the minimum you need is a Python interpreter. However things are easier with a proper development environment. The next pages explain how you can set everything up at your machine and lead you to a successfully interpreted `Hello, World!`. Albeit this tutorial is for a Windows machine, we assume that it is transferable to MacOs and Unix.

P.1.1 Install Python and pip

At `https://www.python.org/downloads/` select your operating system and select the latest stable release for Python 3.8. In February 2021 this is Python 3.8.8. Then select the installer that corresponds to your machine. Run the installer. Select *Add Python 3.8 to PATH*. If you choose *Customize installation* make sure that at least *Documentation*, *pip* and *py launcher* are checked. In the advanced options on the next page check *add Python to environment variables*. After the installation it is recommended to check *Disable path length limit*.

At the creation time of this introduction, Python 3.9 has been released but is not yet compatible with some of the libraries used in the lab course. Therefore, do *not* install Python 3.9 but Python 3.8.8.

P.1.2 Use Python without IDE

Up to here you have installed the Python interpreter. Basically you are now ready to code in Python using the terminal and text editors. This skill is necessary now and then when you are working on a remote machine and have only access via terminal.

Interactive Use

When you press Windows+R and enter `cmd`, a command window opens. On a computer with macOS, find the terminal application. If you type `python` you should see the following:

```
C:\any\path\on\your\machine>python
Python 3.8.8 (default, Nov 16 2020, 16:55:22)
[MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The three arrows in the last line show that you are using Python in interactive mode, meaning that any command you type is interpreted by Python directly. In order to ensure a great Python career, your first command should be nothing else than

```
>>> print('Hello, World!')
Hello, World!
```

You can leave Python and turn back to your operating system's shell using

```
>>> quit()
```

Use with Scripts

When you want to execute larger projects, you do not want to type every single line into the console. Instead, you have prepared your code in `.py` files. You can also test this case by creating a text file that contains some code like.

```
# this is just a comment
print("hello world, but from a script!")
```

Save the file as `script.py` or anything else with a `.py` extension. In your operating system's terminal navigate to the folder where you placed the script and call

```
C:\Users\iksuser\Desktop>python script.py
hello world, but from a script!
```

See that your code was executed without use of the interactive mode.

Load Packages using pip

In order to check whether `pip` was installed correctly, you can try

```
C:\Users\iksuser\Desktop>pip list
```

And you should see

```
Package    Version
---------- -------
pip        21.0.1
setuptools 53.0.0
```

Pip is the package installer for Python. It is used to maintain your installed Python packages. By typing `pip list` you see that there are currently only two packages installed namely pip itself and setuptools.
You can try to install a new package (`numpy`) using

```
C:\Users\iksuser\Desktop>pip install numpy
```

A few progress bars should appear and once the installation is completed you should see numpy when calling `pip list`.
You can test the freshly installed package by opening Python and typing

```
>>> import numpy as np
>>> np.arange(5)
array([0, 1, 2, 3, 4])
```

P.1.3 Install PyCharm

As soon as you plan to create larger projects, you do not want to stick to text files and the command window unless you are a hacker from an 80s movie. Usually, an IDE (integrated development environment) is preferred, that allows for code highlighting, code completion, syntax checking and much more. For this lab course we have chosen PyCharm, which is the choice of most Python programmers. In addition, RWTH students can obtain the professional version for free. If you prefer a more lightweight alternative, you can try Visual Studio Code[1] with Python plug-in[2].

Get PyCharm Professional

In order to get PyCharm Professional, you need to apply for it at JetBrains. At the JetBrain website[3] select 'For students and teachers' and then 'Apply for free products'. Here you should use your RWTH-address. You will then receive a first email to check whether it was actually you, followed by a second mail to activate the license. When you follow the second link, you are asked to create an account. Once you are logged in, you see that you purchased the 'JetBrains Product Pack for Students'. From the list select PyCharm. If you do not see anything visit the PyCharm page[4] and make sure that you are logged in. Download PyCharm for your operating system, run the installer, launch PyCharm and log in with your account.

Create a Project and an Environment

In order to test, you can create a new project (*File* → *new Project*). The name of the project is chosen by specifying the project path. You can name it `path\to\project\hello_world` if you like. Next you are asked to create a virtual environment.

Virtual environments are used to make sure that the right versions of the right packages are used in a project. Since many packages like `sklearn` and `tensorflow` are still subject to API-changes, some code loses its compatibility with new versions. In addition, some packages make requirements about the version of other packages. Apart from that, each virtual environment contains an instance of the Python executable.
Some common tools to create virtual environments are virtualenv, pipenv and conda. Since we have neither installed pipenv nor conda, we use a virtualenv ( ). As soon as you created the project with the desired environment, an example script should open.

[1] `https://code.visualstudio.com/`
[2] `https://code.visualstudio.com/docs/languages/python`
[3] `https://www.jetbrains.com/de-de/community/education/`
[4] `https://www.jetbrains.com/pycharm/`

Run Scripts

You can run the script by pressing ▶ (run) in the top bar, or in the gutter. The gutter is the space between the line numbers and the code. When you hit run, the code is executed and its output is printed in the Python Console.
In the default project, there is also a breakpoint (●) set in the gutter. To make the program execution stop at this point, you need to run it in debug mode. This is done by hitting the debug symbol next to the play icon. As soon as the program stops at the breakpoint, you can investigate and even modify local variables.
One disadvantage of running programmes in default mode is that all local variables disappear after the program is executed. As a solution, it is preferable to right click in the program code and select *Run File in Python Console*. This will run the code in a Python Console and keep variables after the execution. You can imagine the Python console like a system console in which we called `python`. In addition it offers highlighting and keeps track of all variables that we defined in the current session.

Pip and the Terminal

In the lower bar, next to the Python Console you can select the Terminal. In this terminal you can execute commands as in your operating systems command line. Mostly, this is used to update packages with pip. Make sure that the line in the terminal begins with `(venv)` or whatever you named the virtual environment. If it does not, the environment is not active. Activate it by calling

```
C:\Users\iksuser\PycharmProjects\hello_world> venv\Scripts\activate
```

on Windows or

```
[iksuser hello_world]\$ source venv/bin/activate
```

on Unix or mac.
When you type `pip list` you can see all packages that are installed in the currently active environment.

```
(venv) C:\Users\iksuser\PycharmProjects\hello_world>pip list
Package    Version
---------- -------
pip        21.0.1
setuptools 53.0.0
```

If you do not want to maintain the virtual environment via keyboard commands, the PyCharm IDE is at your service. At *File* → *Settings* select *Project* → *Python Interpreter*. Again, you see a list of all packages that are currently installed. You can install new packages when you hit the small plus sign at the bottom.

If the environment is contained in folder `venv`, the Python interpreter is located at `venv/bin/python.exe` or `/venv/Scripts/python.exe`. If you want to run your project without the IDE you can activate the environment in a terminal like described before and invoke the interpreter from the environment folder. The folder also contains an executable for pip and a script to deactivate the environment.

When the virtual environment is set up, you can install packages with pip and use them in the IDE.

Open an Existing Project with Requirements

In order to open an existing project you can either create an empty project and paste the source code in it or create a project in a folder where the source code already exists. To do the latter go to File → Open Project and select the desired folder. You should now see the source code. Before you run it, you probably need to create an environment corresponding to the project requirements. To do so, find *Python 3.8* in the lower right corner of the IDE, click it and select *Add Interpreter*. Here again, you can create a yet empty virtual environment. Go to the terminal to activate the environment as described in the last section.

```
C:\Users\iksuser\APP\prerequisites>venv\scripts\activate
```

If you were given the project requirements as a text file, you can easily install all packages in one fell swoop by calling.

```
(venv) C:\Users\iksuser\APP\prerequisites>pip install -r requirements.txt
```

If everything is installed without errors, you can run your project. Probably, you must right click a file and select *Run* or *Run in Terminal*.

P.2 Getting Started with Python

Now that everything is set up, we can start to learn Python. For the first steps we stick to the console. When the code examples get more complicated, feel free to create and run script files.

P.2.1 Using the Python Console

A Mighty Calculator

In general, we could consider the Python Console as a mighty calculator. See that Python does not require a semicolon at the end of a command.

```
>>> 3+3
6
>>> 10-2
8
>>> (6+2)/4
2
>>> 6+2/4
6.5
>>> 5**3
125
>>> 7//2
3
>>> 7 % 2
1
```

Unlike in most other programming languages, powers are denoted by `**`. The double slash `//` denotes a division without remainder and the percentage sign % denotes the modulo operation.

Define and Access Variables

The calculator can simply define variables. Again (and this will not change), no semicolon is needed at the end of a command. If the name of a (defined) variable is typed in the command window, its value is displayed. However, this does not work within scripts.

```
>>> x1 = 3*2
>>> x1
6
>>> x2 = x1+1
>>> y = x1*x2
>>> y
42
```

Like all cool programming languages, Python supports the combined evaluation and assignment operators `+=`, `-=`, `*=`, `/=`, `//=`, `**=` and `%=`. If you have not heard of these before: `x += 5` is a shortcut for assigning `x = x + 5` which increments x by 5. Similarly, `x *= 3` multiplies the value of x by 3. However, there is no additional shortcut for the increment or decrement by one, as in C and C++.

It is even possible to assign multiple variables at once:

```
>>> a, b, c = 6, 7, 42
>>> b
7
```

Often we wish to continue a computation, without typing the whole expression again. We can use the variable `_` to do so. It is a placeholder for the last value that was a result but has not been assigned to a variable.

```
>>> 6*8
48
>>> _+2
50
```

Functions and the `math` Module

We can upgrade our yet primitive calculator by including so called packages or modules. This is done with the keyword `import`. For instance we can import and use the `math` module as follows:

```
>>> import math
>>> math.sqrt(42)
6.48074069840786
>>> math.sin(90)
0.8939966636005579
>>> math.sin(math.pi/2)
1.0
```

After the import, the functions contained in the math module can be accessed by use of the module name (`math`) and a point. We can distinguish functions from variables by the fact that functions are called with parentheses after the function name. What is contained in these brackets is the *argument* of the function. If a function has more than one argument, these are separated by commas. If a function has no arguments, the brackets remain empty. We will learn how to define functions in Section P.5.

If you forgot what a function does, type `help` and the name of the function (without parentheses).

```
>>> help(math.sqrt)
Help on built-in function sqrt in module math:
sqrt(x, /)
    Return the square root of x.
```

Moreover, `help(math)` displays a list of all functions that are implemented in `math`.

As you see, trigonometric functions use radians and not degrees. Below is a brief overview about implemented functions:
Powers and Logs: `exp`, `log`, `log2`, `log10`, `sqrt`
Trigonometry: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `degrees`, `radians`, `sinh`, `cosh`, `tanh`, `acosh`, `asinh`, `atanh`
Constants: `pi`, `e`, `inf`, `nan`
A complete list can be found in the documentation[5].

Next, you can upgrade your calculator to cope with complex numbers. These can either be defined by use of the imaginary unit `j` (not i) or by use of the constructor `complex(real, imag)`. Note that the imaginary unit can be accessed by `1j`, `2j`, `3j` but not by `j` without a number. With this notation you can still use `j` as a variable.

```
>>> z1 = 2 + 2j
>>> z2 = complex(2, -2)
```

Real and imaginary part of a complex number as well as its conjugate can be accessed using the point notation

```
>>> z2.real
2.0
>>> z2.conjugate()
(2-2j)
```

Here we can see a special case of a function, namely a *method*. A method is a function that is bounded to an object (or type). In this case, the type is `z2` and the method is `conjugate`.

Some basic computations from native Python are also applicable to complex numbers

[5] https://docs.python.org/3/library/math.html

```
>>> z1 * z2
(8+0j)
>>> abs(z1)
2.8284271247461903
>>> (z2/4) ** 2
-0.5j
```

The commands from the `math` package however will not work. Instead, the `cmath` package is the module of choice. In general, it supports all functions from the math package that are applicable to complex numbers. In addition,

`cmath.phase(z)` returns the phase of `z` in radians.

`cmath.polar(z)` returns a representation of z in polar coordinates. The representation is equivalent to `(abs(z), cmath.phase(z))`. Note that the result does not have the type `complex` but `tuple`. We will deal with tuples in Section P.3.6.

`cmath.rect(abs, phase)` returns the cartesian representation of a complex number with the given absolute value and phase. Note that the function receives two arguments.

For a complete overview, please refer to the API[6].

Finally, our calculator supports comments. These are lines that are only to annotate the code but are not executed. They help us to understand the code when we look at it a few days later or show it to someone else (for instance the lab supervisor). In Python, comments begin with a `#`.

```
>>> # create two complex numbers and perform computations
>>> z1, z2 = 1 + 2j, 3 - 4j
>>> z3 = z1 * z2.conjugate()
>>> import cmath # provides functionality
>>> z3_pol = cmath.polar(z3)
>>> # convert back to cartesian
>>> cmath.rect(z3[0], z3[1]) # real and imag are accessed with squared
↪  brackets
```

P.2.2 Jupyter Notebooks

Jupyter notebooks are a useful mixture of Python in the interactive mode and Python in script mode. On top of that, you can integrate a formatted explanation in your document instead of a million comments.

In PyCharm Professional you can create Notebooks (.ipynb files) in the same way as usual Python scripts (.py files). First you need to install the package `jupyter` either using pip or in the PyCharm settings. Right click on your project directory → New → Jupyter Notebook or simply hit File → New → Jupyter Notebook. Below the tab bar, a new menu bar appears that contains among others a doubled green play icon. At the very right of this bar, you can select whether you want to see the source code, the formatted notebook or both next to each other.
A Jupyter Notebook consists of consecutive *cells*. There are code cells that contain

[6] `https://docs.python.org/3/library/cmath.html`

interpretable Python code, markdown cells that contain formatted text and raw cells whose content is neither formatted nor executed. Code cells start with a line that contains only `#%%`. Markdown cells start with `#%% md` and raw cells start with `#%% raw`. Make sure that there are no empty cells after `md` or `raw`. When you begin a cell, a green start icon should appear in the gutter. In the last line, you can see a plus sign in the gutter with which you can add cells as well. The source code of a notebook could look as follows:

```
#%%
import numpy as np
import matplotlib.pyplot as plt

#%% md
# Explanation
This notebook shows how to create a time series in numpy and plot it with
↪  matplotlib.
The sine can be obtained from a complex phasor as follows:
$\text{sin}(x) = \frac{e^{jx} - e^{-jx}}{2j}$
The following bullet points
* demonstrate
* how to make bullet points
 * or even sub points
  * in markdown

#%%
phi = np.linspace(0, 4*np.pi, 200)
y = np.sin(phi)
line = plt.plot(phi, y)
title = plt.title("Sine Wave")
```

You can run each cell one after another by hitting the small play icons right to each cell. Alternatively, you can use Shift + Enter to execute a cell and switch to the next. If you want to run the whole notebook at once, hit the double play icon in the new menu bar.
For a more comfortable view without the PyCharm IDE, you can visit `localhost:8888` in your browser. The file tree of the current folder should appear, including the notebook. When you open the notebook, you can edit and execute the cells. The result should look as in Figure P.1. If the page does not open, go back to PyCharm, find the Jupyter-Terminal next to the OS Terminal and the Python Console and find a similar link there. Many examples for jupyter notebooks, among others from the lecture 'Machine Learning for Speech and Audio Processing' can be found at JupyterLab[7] of the RWTH.

P.3 Data Types

Among the default data types in Python are numbers, logical values, strings and a few compound data types that can store collections of data. In particular, these are lists, tuples, sets and dictionaries.

[7] `https://jupyter.rwth-aachen.de`

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Explanation

Sine waves

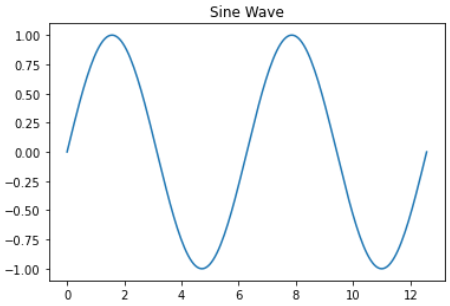
This notebook shows how to create a time series in numpy and plot it with matplotlib. The sine can be obtained from a complex phasor as follows: $\sin(x) = \frac{e^{jx} - e^{-jx}}{2j}$

Bullet Points

The following bullet points

- demonstrate
- how to make bullet points
  - or even sub points
    - in markdown

In [2]:
```
phi = np.linspace(0, 4*np.pi, 200)
y = np.sin(phi)
line = plt.plot(phi, y)
title = plt.title("Sine Wave")
```



In [ ]:
```
```

Figure P.1: The formatted jupyter Notebook

P.3.1 Numbers

In the last examples we already encountered a few data types, namely integer numbers, floating point numbers and even complex numbers. Unlike in many other programming languages, we do not need to specify the type of a variable when we define it. We can even change the type of a variable after it has been defined. The type of a variable can be assessed by the `type` command.

```
>>> x = 5
>>> type(x)
<class 'int'>
>>> x = x/2
>>> type(x)
<class 'float'>
>>> type(x + 2j)
<class 'complex'>
>>> type(type(x))
<class 'type'>
```

A number can be casted to another type by calling the other type's constructor:

```
>>> i = int(2.8)  # i is 2
>>> f = float(2)  # f is 2.0
>>> s = str(2)    # s is '2'
```

P.3.2 Booleans

Booleans are logical values, which are either `True` or `False`. Unlike in many other programming languages, `True` and `False` have to be written in upper-case. In most cases we do not define a boolean variable explicitly, but retrieve it as a result of a logic expression, e.g. a comparison.

```
>>> 7 > 3
True
>>> 5 == 9
False
```

The following logic expressions (with booleans `x` and `y`) are supported in Python:

`x or y`	`True` if either `x`, `y` or both are `True`. `False` if both are `False`
`x and y`	`True` only if both `x` and `y` are `True`
`not x`	`True` if `x` is `False`. `False` if `x` is `True`.

Boolean variables also arise, when other variables are compared. In most cases, the variables being compared are numbers but comparisons are also applicable to other data types. The following comparisons of the variables `a` and `b` are supported:

`a < b`	`a` strictly less than `b`.
`a <= b`	`a` is less than `b` or they are equal.
`a > b`	`a` is strictly greater than `b`.
`a >= b`	`a` is greater than `b` or they are equal.
`a == b`	`a` and `b` are equal.
`a != b`	`a` and `b` are not equal.
`a is b`	`a` and `b` are identical.
`a is not b`	`a` and `b` are not identical.

It is possible to combine multiple binary comparison operators to more complex expressions:

```
>>> 3 < 4 <= 5
True
```

For programmers who are new in Python, the comparisons `a == b` and `a is b` seem identical. And indeed, if we compare numbers they seem to behave equally.

```
>>> 2+2 == 4
True
>>> 2+2 is 4
True
```

The next example however, shades a different light on things:

```
>>> 2 == 2.0
True
>>> 2 is 2.0
False
```

Here we compared the integer 2 with the floating point number 2.0. Even though they are equal, they have a different data type, which is why they are not *identical* and the second comparison yields `False`.

The next example reveals another difference between the comparisons. We create two objects (lists) that contain the same values. Hence the test for equality is `True`, but the test for identity returns `False`.

```
>>> a = [1, 2, 3, 4]
>>> b = [1, 2, 3, 4]
>>> a == b
True
>>> a is b
False
```

The conceptual difference between `is` and == can be found in the German language as well. The relational operator `is` can be explained by the German expression *dasselbe*, whereas the comparison == checks whether the operands are *das gleiche*.

For now it is not important to understand what these list objects are (we will come to them in a moment). Just keep in mind that the test for equality is true if the objects are equal but the test for identity is only true if they are the same object. We will come back to classes and objects in Section P.6.

P.3.3 Strings, Output and Input

For completeness, here is what you've been waiting for the whole time:

```
>>> print('Hello, World!')
Hello, World!
```

Creation and Access

Strings can be treated like the variables we saw until now but unlike numbers, the `+` operator concatenates two strings. Unlike typing a variables name in the console, the `print` command works in Python scripts as well. Strings can be delimited by both `"double quotes"` and `'single quotes'`. Only if a string contains a single quote, it must be delimited by double quotes.

```
>>> s = 'Hello'
>>> s + ", is it me you're looking for?"
'Hello, is it me you're looking for?'
```

Formatted Output

When we want to print numbers or strings from within a script, we need the `print` command. It accepts any number of input arguments that are then displayed sequentially in the console. Assume we want to print the progress during the training of a neural network.

```
>>> loss = 0.092392837
>>> it_epoch, max_iter = 54, 100
>>> print("Loss is", loss, "after epoch", it_epoch, "of", max_iter)
Loss is 0.092392837 after epoch 54 of 100
```

With the optional argument `sep=` you can specify the separator between the printed arguments. By default this is a space `' '`. With the optional argument `end=` you can specify what is printed after the last argument. By default, this is a new line.

When you want to format the output more specific, the `format` function is needed. This routine can be called on a string that contains several `{ }` placeholders and receives several strings as arguments. In the output, the pairs of curly braces are replaced by the arguments.

```
>>> "Loss is {} after epoch {} of {}".format(loss, it_epoch, max_iter)
'loss is 0.092392837 after epoch 54 of 100'
```

When an argument is a floating point number, we can determine its precision by inserting a colon, a point the precision and an f in the corresponding curly braces.

```
>>> s = "Loss is {:.3f} after epoch {} of {}"
>>> s.format(loss, it_epoch, max_iter)
'loss is 0.092 after epoch 54 of 100'
```

In the brackets you can specify by which argument they are to be replaced. This can be either by position or by keyword. As a result, the arguments can have a different order in the function call.

```
>>> # Access by position
>>> s = "Loss is {1:.3f} after epoch {2} of {0}"
>>> s.format(max_iter, loss, it_epoch)
'loss is 0.092 after epoch 54 of 100'
>>> # Access by keyword
>>> s = "Loss is {loss:.3f} after epoch {epoch} of {max_epoch}"
>>> s.format(max_epoch=max_iter, epoch=it_epoch, loss=loss)
```

If the last example was yet unclear, don't worry. We will deal with keyword and positional arguments later in Section P.5.6. The same holds for the next example, where we use an unpacked dictionary as input. You don't need to understand it now, but as soon as you finished Section P.5.8 you can come back here and enjoy its beauty.

```
>>> training_params = {loss: 0.092392837, it_epoch: 54, max_iter: 100}
>>> s = "Loss is {:.3f} after epoch {it_epoch} of {max_iter}"
>>> s.format(**training_params)
```

Access

We can access substrings from a string using indexing and slicing. This will be covered with more detail in Section P.3.4.

```
>>> str = 'Bananas'
>>> str[0]
'B'
>>> str[1:]
'ananas'
>>> str[2:6]
'nana'
>>> str[-1]
's'
```

Functional API

Below is an excerpt of the functions, you can call on a string. The full list can be found here `https://docs.python.org/3/library/stdtypes.html#string-methods`. Squared brackets denote optional arguments.

`capitalize()` returns a copy of the string, where only the first character is capitalized and the rest is lowercase.

`count(sub[, start[, end]])` returns the numbers of non-overlapping occurrences of the string pattern `sub` in the string.

`endswith(suffix[, start[, end]]` returns `True` if the string (or the substring between start and end) ends with `suffix`.

`find(sub[, start[, end]])` returns the lowest index of `sub`. Return -1 if sub is not found. If you only want to check if the substring is contained at all, you should prefer `in` (See example below).

`index(sub[, start[, end]])` same as `find`, but raises an error if the substring is not found.

`join(iterable)` returns the concatenation of all strings in `iterable`.

`lower()` returns a copy of the string where all characters are in lowercase.

`replace(old, new[, count])` returns a copy of the string with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.

`split(sep)` returns a list of substrings that are separated by `sep`.

`startswith(prefix[, start[, end]])` See `endswith`.

`upper()` returns a copy of the string where all characters are capitalized.

These functions are also used with the point notation. Assume we created a string `s` and we need a capitalized version `s_c`, we call `s_c = s.capitalize()`. After that, the string `s_c` is a capitalized version of `s`. See that all functions in the API that modify a string only return a modified copy of the string. They do not act *in place*. This is due to the fact that strings are *immutable*, which means that after creation a string cannot be changed or modified.

```
>>> s = 'Spam, Spam, Spam, egg and Spam'
>>> s.count('Spam')
4
>>> s_r = s.replace('egg', 'Spam')
>>> s  # stays unchanged
'Spam, Spam, Spam, egg and Spam'
>>> s_r
'Spam, Spam, Spam, Spam and Spam'
>>> filename = "cool_song_by_the_foo_fighters.mp3"
>>> filename.endswith(".mp3")
True
>>> filename.split('_')
['cool', 'song', 'by', 'the', 'foo', 'fighters.mp3']
>>> 'foo' in filename
True
>>> filename.index('foo')
17
```

User Input

The user can be asked to input content into the command prompt using the `input()` command. This command returns a string containing what the user entered. The

function takes a string as an optional argument, that is displayed prior to the input. In the following example we request three inputs by the user and seperate them using `split`.

```
>>> s = input("Name, Age and Profession please")
Name, Age and Profession please
>? Gallahad the Pure, 42, Knight
>>> s
'Gallahad the Pure, 42, Knight'
>>> name, age, profession = s.split(', ')
>>> print(name)
Gallahad the Pure
```

Note that `type(age)` is `<class 'str'>` albeit we entered a number. A numeric value can be obtained by `int(age)`.

P.3.4 Lists

A list is a data type that contains a sequence of usually homogeneous data. Albeit it is possible to store elements of different (heterogeneous) types, this is not typical. A property that separates lists from other compound data types is the list being *mutable*. This means that the content of a list can be modified without the need to make a copy.

Construction

When you create an empty list, you do not need to specify which type(s) of data you are up to store in it. An empty list can be created as follows

```
>>> empty_list = []
>>> also_empty_list = list()
```

If the content is known at creation time, separate it by commas and enclose it with square brackets. The following example shows the basic commands to create a list and to add elements.

```
>>> l = [2, 4, 6, 8, 10]
>>> l
[2, 4, 6, 8, 10]
>>> l.append(12)
>>> l
[2, 4, 6, 8, 10, 12]
>>> l.insert(2, 5)
>>> l
[2, 4, 5, 6, 8, 10, 12]
```

Like in the previous section about strings, lists can perform methods using the dot notation after the variable name. See that the commands worked in place. The content of the list was modified directly and no copy was returned.
For completeness, there is another way to create a list, namely using the constructor of type `l = list(iterable)`. Since we haven't covered iterables yet, let's leave it for

now that sets, ranges, dictionaries and tuples are so called iterables. So after you have read the next section dealing with ranges, you are able to create a list from a range.

Accessing Elements: Indexing and Slicing

We can access single elements in the list by denoting the index of interest in squared brackets behind the list's name. Note that the first element has the index 0, where the last element has the index `len(l) - 1`, where `len(l)` is the length of the list. However, accessing the last element can be shortened by writing just `l[-1]` and returns the last element in the list. `l[-2]` is the second to last element and so on.

```
>>> l = [3, 1, 4, 1, 5, 9]
>>> l[2]
4
>>> l[-1]
9
>>> l[-2]
5
```

If a list contains objects, indexing returns a reference of the actual object and does not create a copy. See the following example, where the list `l` contains objects, namely other lists.

```
>>> l = [[2, 3, 4], [4, 9, 16], [8, 27, 63]]
>>> cubes = l[2]
>>> cubes
[8, 27, 63]
>>> cubes[2] = 64
>>> l
[[2, 3, 4], [4, 9, 16], [8, 27, 64]]
```

When `cubes` was accessed by indexing, a reference to the actual element was returned. Thus, modifying this reference, also changed the actual content of `l`. If you want to leave the original list unchanged, you need to copy the desired element.

```
>>> squares = l[1].copy()
>>> squares
[4, 9, 16]
>>> squares[1] = 10
>>> l
[[2, 3, 4], [4, 9, 16], [8, 27, 64]]
```

Since only a copy of `l[1]` was modified, `l` remains unchanged. For now, the only mutable objects we know are lists themselves, but we will get in touch with more object types in the next subsections and deal with objects in general in section P.6.

We can also use the colon operator `start:stop` to access a sublist. This is denoted as *slicing* and returns a list again. If `start` and or `stop` are not given, the first or last index are used instead.

```
>>> l = [3, 1, 4, 1, 5, 9]
>>> l[1:4]
[1, 4, 1]
>>> l[:3]
[3, 1, 4]
>>> l[3:]
[1, 5, 9]
```

See, that when we queried `l[:3]` we were given the first three elements, but not the element with the actual index 3. As a general rule, keep in mind that when you use the colon operator `start:stop` to access elements in a list, the returned list has the length `stop - start` and the element with index `stop-1` is the last index in the returned list.
With an additional colon, we can also specify a step size. If you already know this from Matlab, be aware that the order in Python is `start:stop:step`.

```
>>> l[1:6:2]
[1, 1, 9]
>>> l[::2]
[3, 4, 5]
```

Slicing a list with objects constitutes the same risk as indexing. The list being returned contains references to the actual objects. Changes that are made on them also affect the list where the selection was taken from.

If you have used Matlab before, also be aware that the colon operator can not be used to create lists as in `x = 1:10`. In Matlab this would return an array with elements ranging from 1 to 10. In Python however, this will raise a Syntax error. In a moment, we will encounter the `Range` object with which we can create lists like these.

API Reference

Below is a brief overview on operations on a list. Again, see `https://docs.python.org/3/library/stdtypes.html` for a more detailed description.

`x in l` returns `True` if the list `l` contains the element `x`

`x not in l` returns `True` if the list `l` does not contain the element `x`

`l1 + l2` returns the concatenation of the lists `l1` and `l2`.

`l * n or n * l` returns a list consisting of `l` repeated `n` times. However note that when l contains objects, these are referenced and not copied. So once you perform changes in `l`, its references in `l * n` are affected as well. A common case is when the elements in `l` are lists themselves. If the elements of `l` are simply numbers, you are save.

`l *= n` replaces `l` by its contents repeated `n` times

`l.append(x)` adds x as the last element to the list. If `x` is a list itself, it will be added as a single element. If you wish to add the elements in `x` one by one, use `l.extend(x)`.

`l.clear()` removes all items from `l`.

`l.copy()` returns a copy of `l`

`l.count(x)` returns the number of occurrences of `x` in `l`.

`l.extend(l2)` adds all elements in the list `l2` to `l`. Generally, `l2` needs to be an `iterable`, which we will encounter later.

`l.index(x[, i[, j]]` return the index of the first occurence of `x`. If indices `i` and `j` are given, the search is restricted to this range.

`l.insert(i, x)` inserts `x` at index `i`.

`len(l)` returns the number of elements in `l`

`min(l)` returns the smallest element in `l`

`max(l)` returns the largest element in `l`

`l.pop()` returns the last element in the list and removes it from the list, as in a stack. If an index is provided as argument, the element at this position is returned and removed.

`l.remove(x)` removes the first occurrence of `x`.

`l.sort()` sorts the element in the list. If the elements are not comparable, like for instance a string and a number, an error is raised and the list stays in a partially modified state. If sorting in descending order is desired, use `l.sort(reverse=True)`

P.3.5 Ranges

Even though it is a distinct data type we can deem it a special case of a list that contains only numbers (integers or floats) in ascending or descending order with a fixed step size. A range object can only be created with the range constructor. `range(length)` creates a range of length `length` where the first element is `0` and the last element is `length-1`. If two arguments are provided, the range created by `range(start, stop)` ranges from `start` to `stop-1`. An optional third argument determines the step size, which can also be negative.

If we want to access the numbers themselves, the range needs to be converted to a list.

```
>>> range(4)
range(0, 4)
>>> range(20, 40, 5)
range(20, 40, 5)
>>> list(range(20, 40, 5))
[20, 25, 30, 35]
```

By now, this distinct data type appears to be of limited use. However, we will see it in a different light when we are dealing with for loops, comprehensions and iterables in the next section.

P.3.6 Tuples

Since tuples are confused with lists from time to time, we start with their differences.

- Tuples are intended to store heterogeneous data.
- Tuples are immutable, which means that their content can only be manipulated by creating a copy. This includes that their length cannot be changed without creating a copy either.
- Tuples are created with round brackets instead of square brackets.

A typical example for the use of tuples is to determine the dimensions of multi dimensional arrays.

Construction

In the following example, two tuples are created, that contain coordinates of cities and their altidude in meters. For this task, tuples are perfectly suited, since we do not expect the coordinates to change or to be extended by a fourth dimension. Finally we can argue that even though all elements in the tuples are floating point numbers, they are still heterogeneous, since the level above see is something entirely different than the north latitude.

```
>>> new_york = ("40° 43' N ", "74° 0' W", 10)
>>> mumbai = tuple(["18° 58' N", "72° 50' O", 11])
```

In the second case, the tuple was created from a list. Like lists, tuples also have the ominous constructor `tuple(iterable)`. Albeit we still haven't talked about iterables, keep in mind that you can create tuples from existing lists, sets, dictionaries or from other existing tuples.

When you need to define a tuple that contains only one number, a comma needs to be added. Otherwise, the expression is recognized in a numerical way. `(3)` is an integer in brackets, whereas `(3,)` is a tuple.

Access

A tuple's contents are accessed the same way as the contents in a list, using the slicing notation introduced in the last subsection. For the access, squared brackets are used again. The round brackets are only used for construction.

```
>>> new_york[1]
"74° 0' W"
>>> mumbai[:3:2]
("18° 58' N", 11)
```

As you see, indexing returns the element itsself, while slicing returns a tuple.

For access to tuples that contain objects, the same rules apply as for lists in the last section. Modification on accessed elements or subtuples also affect the original tuple.

API Reference

Since both lists and tuples are so called sequence types that have a defined order, their API shares a lot with the list's API.

`x in t` returns `True` if the tuple `t` contains the element `x`

`x not in l` returns `True` if the tuple `t` does not contain the element `x`

`t1 + t2` returns the concatenation of the tuples `t1` and `t2`.

`t * n or n * t` returns a tuple consisting of `t` repeated `n` times. However, note that when `t` contains objects, these are referenced and not copied. So once you perform changes in `t`, its references in `l * n` are affected as well.

`t.count(x)` returns the number of occurrences of `x` in `t`.

`t.index(x[, i[, j]]` return the index of the first occurence of `x`. If indices `i` and `j` are given, the search is restricted to this range.

`len(t)` returns the number of elements in `t`

`min(t)` returns the smallest element in `t`

`max(t)` returns the largest element in `t`

P.3.7 Dictionaries

In the previous coordinates example it might happen, that we forget which entry in a tuple corresponds to which property. We might query `new_york[2]` and wonder what is the meaning of the `10` being displayed. A solution is provided by the dictionary type. Instead of integer indices, the elements in the dictionary are accessed by arbitrary keys.

```
>>> d = dict()
>>> d['the_key'] = 42
>>> d['the_key']
42
```

These keys can be integers as well, but also strings, characters or tuples (and all immutable types).

When you intend to use floating point numbers as keys you should take care, since these are prone to numeric errors.

Construction

There are several ways to create a dictionary on given data, all of which are useful in different situations.

```
>>> moscow = {'latitude': "55° 45'N", 'longitude': "37° 37'", 'altitude': 156}
>>> berlin = dict(latitude="52° 31' N", longitude="13° 24' O", altitude=30)
>>> shanghai = dict([('latitude', "31° 14' N"), ('longitude', "121° 28' O"),
↪  ('altitude', 4)])
>>> laurensberg = dict()
```

```
>>> laurensberg['latitude'] = "50° 48' N"
>>> laurensberg['longitude'] = "6° 4' O"
>>> laurensberg['altitude'] = 163
>>> laurensberg
{'latitude': "50° 47' N", 'longitude': "6° 3' O", 'altitude': 163}
```

In the moscow dict, pairs of `key: value` were separated by commas and enclosed by curly braces. The berlin dict was created by use of the constructor with arguments of type `key=value`. Instead, the constructor can also receive a list of 2-tuples `(key, value)` as in the shanghai dict. Since dictionaries are mutable, we can create an empty dict and add the elements afterwards, as was done in the laurensberg example.

API Reference

Below is an incomplete list of functions and commands related to dictionaries.

`k in d` returns `True` if the dict d has a key named `k`

`k not in d` returns `True` if the key `k` is not in the dict.

`del d[k]` removes `d[k]` from `d` or raises an error if `k` is not in the `d`.

`d.clear()` removes all items from the dictionary.

`d.copy()` returns a copy of the dictionary.

`d.get(k [, default])` same as `d[k]` but returns the optional argument `default` if key `k` is not present in `d`.

`list(d)` returns a list of all keys, used in `d`.

`len(d)` returns the number of items in the `d`.

`d.pop(k[, default])` return `d[k]` if the key `k` is existent and removes the entry from the dict. Returns the optional argument `default`, if there is no entry with key `k`.

`d.popitem()` returns the last key value pair as a tuple and removes the entry from the dict.

P.3.8 Sets

Sets are containers for unordered data, which seperates them from sequential containers like lists, ranges and tuples. As a consequence, sets do not support access by indexing or slicing.

Construction

The creation is similar to the creation of lists and tuples but uses curly braces, or the constructor `set()`. Again, there is a constructor `set(iterable)` that receives an ominous iterable, which we know can be a list, a tuple or a dict. In the case of sets, this constructor has a useful property:

```
>>> menu = ["Spam", "Spam", "Spam", "Eggs", "Spam"]
>>> set(menu)
{"Spam", "Eggs"}
```

```
>>> list(set(menu))
["Spam", "Eggs"]
```

As you see, we can use the set constructor to remove obsolete elements in an iterable structure.

There exists a special case of sets, namely the `frozenset()`. While the usual set is mutuable and allows for its content to be changed, the frozenset is not. As a result, the frozenset is hashable and can be used as a dictionary key.

API Reference (Sets and frozen sets)

The following commands can be used on both sets and frozen sets:

`len(s)` returns the number of elements in `s` (aka its cardinality)

`x in s` returns `True` if item `x` is contained in the set `s`

`x not in s` returns `True` if item `x` is not contained in the set `s`.

`s.isdisjoint(other)` returns `True` if the sets `s` and `other` do not share a single member.

`s.isssubset(other)` returns `True` if every element in set is contained in `other`

`set <= other` same as `s.issubset(other)`

`s.issuperset(other)` returns `True` if every element of `other` is a member of `s`.

`s >= other` same as `s.issuperset(other)`

`s > other` returns `True` if `s` is a proper superset of other, meaning that every element of `other` is contained in s but as contains at least one element that is not contained in `other`

`s.union(others)` returns a new set that contains all members from `s` and `other`.

`s | other1 | other2 | ...` same as `s.union(other1, other2, ...)`

`s.intersection(other)` returns a new set with the elements that are contained in `set` and (every set in) `other`

`set & other1 & other2 & ...` same es `s.intersection(other1, other2, ...)`

`s.difference(other)` returns a set containing elements that are in `s` but not in (any set in) `other`. The returned set does not contain the elements that are in `other` but not in `s`.

`s - other1 - other2 - ...` same as `s.difference(other1, other2, ...)`

`s.symmetric_difference(other)` returns a set containing the elements that are either `s` or `other` but not in both. Unlike the aforementioned commands, `symmetric_difference` accepts only a single argument.

`s ^other1 ^ other2` same as `(s.symmetric_difference(other1)).symmetric_difference(other2)`

`s.copy()` returns a copy of `s`.

API Reference for Mutable Sets

As they modify the set members, the following operations are only applicable to sets but not to frozen sets.

`s.update(other)` adds all elements from `other` to `s`.

`set |= other1 | other2 | ...` same as `set.update(other1, other2, ...)`

`s.intersection_update(other)` keeps only elements that are in both `s` and `other` and deletes the remaining elements.

`s &= other1 & other2 & ...` Same as `s.intersection_update(other1, other2, ...)`.

`s.difference_update(other)` removes all elements from `s` that are also contained in `other`.

`s -= other1 - other2 - ...` same as `s.difference_update(other1, other2, ...)`.

`s.symmetric_difference_update(other)` updates `s` by keeping (or adding) only elements that are contained in either set but not in both.

`s ^= other` same as `s.symmetric_difference_update(other)`.

`s.add(x)` adds the element `x` to `s`.

`s.remove(x)` removes `x` from `s` but raises an error if `x` is not contained in `s`.

`s.discard(x)` removes `x` from `s` if x is contained.

`s.pop()` removes and returns an element from `s` but raises an error if the set is empty.

`s.clear()` removes all elements from `s`.

P.4 Control Flows

Until here you learned a lot about data types and commands that we can use on them. Some of them were boolean commands that give us information about the current state of the data type, e.g. we can check whether a set contains a certain element. However until now, we did not encounter any way to make use of this kind of information. In this section we deal with control flows that steer our program depending on conditions that we define.

P.4.1 The if-Statement

In the `if` statement, a command or a block of commands is executed only if a condition is fulfilled. If the condition is not fulfilled, the block is skipped and the program is continued after the block. As an example, we might want to process an audiofile of which we know the filename. To make sure that it is a proper audio file we could use the `endswith()` method on the file name, to check if it ends with `'.wav'`. If it does not, we can skip the processing.

If you know if statements from other programming languages you might expect something like this:

```
if(i >= 2){
   do_stuff_in_C++();
}

if(i == 2)
   do_stuff_in_matlab();
end
```

In both cases, the boolean statement is enclosed by parentheses and the actual body is either enclosed by curly braces or delimited by an end statement. In Python, things are somewhat different and take some getting used to for new programmers. The basic principle is illustrated in the following example:

```
>>> if 2 + 2 == 4:
...     print('You were right all the time')
...     print('2 + 2 equals 4')
...
You were right all the time
2 + 2 equals 4
```

Apart from missing semicolons, we observe two differences to the previous snippets. Instead of parentheses the condition is enclosed by the keyword `if` and a colon. The following sequence of commands is indented equally by four white spaces. The body of the condition ends with the first line that is not indented. This principle is called *indentation* and is used to group lines of code in Python. All consecutive lines with the same number of preceeding whitespaces belongs together. As a result, the following two examples raise an indentation error:

```
>>> if 2+2==4:
... print("Yeah, maths!")
  File "<stdin>", line 2
    print("Yeah, maths")
        ^
IndentationError: expected an indented block

>>> if "on" in "Python":
...     print("Hello World")
...        print("I love Python")
  File "<stdin>", line 3
    print("I love Python")
    ^
IndentationError: unexpected indent
```

In the first example, the interpreter expected an indent but there was none. In the second attempt, the third line was indented more than the second line but without a corresponding keyword.

When you are coding in an IDE it is convenient to use the tab key for indentation. However, in some IDEs one tab is converted to four white spaces. For the interpreter, these are not interchangeable. So once you change your IDE or want to do quick changes in an editor like vim, make sure to use the same amount of spaces as in the existing code. If the tab key is used without conversion, an indentation error arises, even though the lines appear to be indented correctly. Therefore, better always use spaces!

The `if` block can be expanded by an `else` block. The commands in the `else` block are only executed if the prior condition is `False`. In the last example, we can process our audio file if the filename ends with `'.wav'`. But if it does not we could display a warning.

```
>>> if file.name().endswith('.wav'):
...     do_audio_process_stuff(file)
... else:
...     print("Unknown file format! Please convert!")
...
```

If these options are not enough, we can add else-if-blocks, specified by `elif`. These are executed if the first condition is false, but another condition is true.

```
>>> if file.name().endswith('.wav'):
...     do_audio_process_stuff(file)
... elif file.name().endswith('.mp3'):
...     converted = mp3_to_wav(file)
...     do_audio_process_stuff(converted)
... elif file.name().endswith('.ogg'):
...     converted = ogg_to_wav(file)
...     do_audio_process_stuff(converted)
... else:
...     print("Unknown file format! Please convert!")
```

As soon as the first condition is true and the corresponding block is executed, none of the following conditions is checked and the program continues after the last block of the whole if-else-block.

P.4.2 The Keyword pass

In some situation we need to ask our program to do a particular difficult task: We want it to do nothing. A common use case is when we first want to set up a code skeleton, but want to implement the functionality later. In other programming languages this is not difficult at all. If we want the program to do nothing we write nothing, e.g. an empty line. In Python however, the indentation rules throw a spanner in the works. Lets go back, to the previous example, but we are waiting for a colleague to implement the `do_audio_process_stuff(file)` and conversion routines. Hence we decide to skip these lines until then:

```
>>> if file.name().endswith('.wav'):
... elif file.name().endswith('.mp3'):
  File "<input>", line 2
    elif name.endswith('.mp3'):
       ^
IndentationError: expected an indented block
```

After the `if` keyword, followed by a boolean condition and a colon, the interpreter expects a proper command. For this purpose, Python supports the `pass` keyword, which does exactly what we need: nothing. To the interpreter however, it looks like a proper command.

```
>>> if file.name().endswith('.wav'):
...     pass
... elif file.name().endswith('.mp3'):
...     pass
```

```
... elif file.name().endswith('.ogg')
...     pass
... else:
...     print("Unknown file format! Please Convert!")
```

P.4.3 While Loops

Basic Usage

In the `if`-construction, a code block was executed only if a condition was `True`. In `while`-loops, this block is executed as long as the condition is true. After the block has been executed (given that the condition was True), the condition is checked again. If it is still `True`, the block of commands is executed again. This repeats until the condition is not fulfilled anymore. One execution of the loop's body is called an *iteration*. The loop's syntax is the same as for the `if` construct, only with a `while` instead of an `if`. Further, there is no such thing as `else` or `elif`. The following example illustrates the basic usage of a while loop:

```
>>> l = list(range(5))
>>> listsum = 0
>>> while len(l) > 0:
...     listsum += list.pop()
...     print("Now the list is", l, "and the sum is", listsum)
...
Now the list is [0, 1, 2, 3] and the sum is 4
Now the list is [0, 1, 2] and the sum is 7
Now the list is [0, 1] and the sum is 9
Now the list is [0] and the sum is 10
Now the list is [] and the sum is 10
```

In each iteration, the list is modified so that its length decreases. After the last element was taken from the list, the condition is `False` and no further iteration follows.

Nested Statements

If you need to use an if statement within a while loop (or a loop within a loop or anything else) you need to indent the inner construction twice. In the following example, we want to collect only fruits that have at least one a in their name.

```
>>> fruits = {'Apple', 'Banana', 'Cherry', 'Lemon'}
>>> fruits_with_a = set()
>>> while len(fruits) > 0:
...     f = fruits.pop()
...     if 'a' in f.lower():
...         print("Found", f.count('a'), "a in", f)
...         fruits_with_a.add(f)
...
Found 1 a in apple
Found 3 a in banana
>>> fruits_with_a
{'Banana', 'Apple'}
```

The nested if block follows the known indentation rules with regard to the already indented commands in the while loop.

The Keywords break and continue

From time to time, when we deal with more complex loops, we need the additional keywords `break` and `continue` to control the flow of the loop.
As soon as the interpreter encounters a `break` within a loop, the loop is left immediately and no command after the `break` is executed. In the following example, the numbers from 0 to 19 are added but the addition is stopped as soon as the cumulative sum exceeds 42 for the first time.

```
>>> l = list(range(20))
>>> s = 0
>>> while len(l) > 0:
...     s += l.pop(0)
...     print(s, end=', ')
...     if s > 42:
...         print("Exceeded 42")
...         break
...
0, 1, 3, 6, 10, 15, 21, 28, 36, 45, Exceeded 42
```

The `continue` statement does a similar thing but instead of leaving the whole loop, only the current iteration is left. After a `continue` statement, the interpreter jumps back to check the loop condition for the next iteration. In the following example we want to multiply a sequence of numbers but discard values that are smaller or equal to zero.

```
>>> prod = 1
>>> numbers = [2, 3, 0, -6, 7]
>>> while len(numbers) > 0:
...     x = numbers.pop()
...     if x <= 0:
...         print("Skipped", x)
...         continue
...     prod *= x
...     print("The product is", prod)
...
The product is 7
Skipped -6
Skipped 0
The product is 21
The product is 42
```

P.4.4 for Loops

You may know from other programming languages that `for` loops are used to execute a certain block of commands for a given number of times in a row. E.g. if you want to get the five last elements in a list, you need to call `l.pop()` five times. Surely you could write `l.pop()` five times in a row, however this would not look very nice.
In Python, `for` loops are used in a more general way. In a moment we will see that they can be used for much more than repeating the same command over and over.

Simple Example

Lets say we want to know the squares from the first five integers.

```
>>> for i in range(5):
...     print("The square of", i, "is", i**2)
...
The square of 0 is 0
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
```

As you see, we can use the `range` keyword, to specify the number of repetitions. Moreover, the number of the current iteration (starting with 0) is stored in the local variable `i`. During the execution of the loop, we have access to i. After the execution it stays in the workspace with its last value. If you remember the `range` keyword from subsection P.3.5 you know that we can also specify the start and a step size.

```
>>> for i in range(10, 20, 2):
...     print("The square of", i, "is", i**2)
...
The square of 10 is 100
The square of 12 is 144
The square of 14 is 196
The square of 16 is 256
The square of 18 is 324
```

General Usage with Iterables

In the last examples, the range command was used to control how often the loop's body is executed and to specify the value of `i` in each iteration. However, in Python the use of `for` loops is way more flexible. In fact, we can use any *iterable* instead of a `range` object. From Section P.3 you know, that lists, tuples, ranges and sets belong to this group of data types. Furthermore, strings are also iterable. But what does that mean?
An object is iterable if we can iterate over it. In other terms, an object is iterable if it is able to return one member at a time. For now, it is enough to know that the aforementioned data types are iterable. Later on in Section P.6.3, we will see how we can implement iterables on our own. The following examples show that we can use any iterable data type to control the loop.

```
>>> l = [3, 1, 4, 1, 5, 9]
... for element in l:
...     print(element, sep=', ')
...
3, 1, 4, 1, 5, 9

>>> s = {"Spam", "Eggs", "Beans"}
>>> for member in s:
...     print(member)
...
Spam
```

```
Beans
Eggs
```

When you iterate over a dictionary, only the key is returned:

```
>>> person = {'Name': 'Alice', 'Age': 42, 'Job': 'Developer'}
>>> for key in person:
...     print(key)
...
Name
Age
Job
```

If you need both keys and values, you can call the dictionary with the corresponding key. Otherwise you can use `dict.items()` which returns a list of tuples.

```
>>> for entry in person.items():
...     print(entry[0], "->", entry[1])
...
Name -> Alice
Age -> 42
Job -> Developer
```

Since `dict.items()` returns a list of tuples, you can also access the tuple entries directly. In the following example this is demonstrated with a set of tuples:

```
>>> s = {(0, 1), ("Hello", 2), ("Bob", "Banana"), ("Spam", "Eggs")}
>>> for x, y in s:
...     print(x, "->", y)
...
0 -> 1
Spam -> Eggs
Bob -> Banana
Hello -> 2
```

Finally, iterating over a string will return all characters one by one:

```
>>> str = "Spam and Eggs"
>>> for c in str:
...     print(c, sep="-")
...
S-p-a-m- -a-n-d- -E-g-g-s
```

The Keywords `enumerate` and `zip`

When you call the `enumerate` function on an iterable object, it will return a list of tuples. In fact, it returns an `enumerate object`, which is iterable but not subscriptable. We can treat it like a list of tuples. The first element in the tuples is the index ranging from 0 to the number of elements in the iterable minus one. The second elements in the tuples are the actual elements of the original iterable. As shown in the previous example, a list of tuples can easily be used in a `for` loop:

```
>>> persons = {"Bob", "Alice", "Frank"}
>>> for idx, name in enumerate(persons):
...     print("Person no", idx+1, "is called", name)
...
Person no 1 is called Frank
Person no 2 is called Alice
Person no 3 is called Bob
```

The keyword `zip` is used to merge two (or more) iterables to a list of tuples. It returns a `zip object` that is comparable to the `enumerate object`. It is iterable but not subscriptable using squared brackets.

```
>>> val = [3, 4, 5]
>>> sq = [9, 16, 25]
>>> for it_val, it_sq in zip(val, sq):
...    print("The square of", it_val, "is", it_sq)
...
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

If you need to access the individual elements in an enumerate or zip object, remember that lists (as well as sets and tuples) can be constructed from an iterable:

```
>>> list(zip(val, sq))
[(3, 9), (4, 16), (5, 25)]
```

P.4.5 List Comprehensions

Basic Usage

In many cases we use `for` loops only to fill a list, as in the following example:

```
>>> val = [3, 1, 4, 1, 5, 9]
>>> sq = []
>>> for it_val in val:
...     sq.append(it_val ** 2)
...
>>> sq
[9, 1, 16, 1, 25, 81]
```

However, the variable `it_val` stays in the workspace even after the last iteration is executed. Even worse: If a variable named `it_val` existed before, it is overwritten without warning.

In Python there exists an elegant shortcut called list comprehension, which is closely related to the for loop. In general, it has the form `[` *expression* `for` *variable* `in` *iterable* `]`

```
>>> val = [3, 1, 4, 1, 5, 9]
>>> sq = [it**2 for it in val]
>>> sq
[9, 1, 16, 1, 25, 81]
```

Here, the expression is `it**2` and the variable `it` is drawn from the iterable `val`. This expression is then evaluated `for` (each) `it in val`, yielding as many elements as there are elements in val. Then all these elements are returned in a list.

As well as in `for` loops, you can use any iterable in a list comprehension. For instance `[x for x in range(5)]` gives you `[0, 1, 2, 3, 4]`.

Filtering

We can also use list comprehensions to filter the content of an iterable. In the following example, we create a list that contains only the strictly positive entries from a previous list. Of course this could be solved using a for loop:

```
>>> l = [2, 4, 0, -1, -2, 3, 5]
>>> l_pos = []
>>> for x in l:
...     if x > 0:
...         l_pos.append(x)
...
>>> l_pos
[2, 4, 3, 5]
```

In list comprehensions, we can add a condition that depends on the variable after the iterable.

```
>>> l = [2, 4, 0, -1, -2, 3, 5]
>>> l_pos = [x for x in l if x > 0]
>>> l_pos
[2, 4, 3, 5]
```

There is a second way to combine a list comprehension with an if statement. In the following example the signs of all negative numbers in a list are flipped, to ensure that all elements in the resulting list are positive. First, see the equivalent for loop:

```
>>> l = [2, 4, 0, -1, -2, 3, 5]
>>> l_pos = []
>>> for x in l:
...     if x < 0:
...         l_pos.append(-x)
...     else:
...         l_pos.append(x)
...
>>> l_pos
[2, 4, 0, 1, 2, 3, 5]
```

Again, the list comprehension gives us a more handy expression:

```
>>> l= [2, 4, 0, -1, -2, 3, 5]
>>> l_pos = [(-x if x < 0 else x) for x in l]
>>> l_pos
[2, 4, 0, 1, 2, 3, 5]
```

At the first sight, this syntax might be confusing. First, see that in this example the `if` construction is written before the `for`, other than in the previous example. In the previous example, the elements were only appended to a list if they satisfied a certain condition. Now, the elements are appended anyway but if they satisfy a condition they are modified. The structure of the if statement is different than in a usual if construction. Here it is: [*value* `if` *condition* `else` *else_value*]. If the *condition* holds, *value* is appended. Otherwise, *else_value* is appended. The parentheses are not needed but they improve the readability of the expression.

Of course you can combine these methods. Lets say we have all numbers from -10 to 10 and want to flip the sign of all negative numbers but only keep even numbers. Using a loop this would be:

```
>>> l = []
>>> for x in range(-10, 11):
...     if x % 2 == 0:
...         if x < 0:
...             l.append(-x)
...         else:
...             l.append(x)
...
>>> l
[10, 8, 6, 4, 2, 0, 2, 4, 6, 8, 10]
```

Using list comprehensions and ifs:

```
>>> l = [-x if x < 0 else x for x in range(-10, 11) if x % 2 == 0]
>>> l
[10, 8, 6, 4, 2, 0, 2, 4, 6, 8, 10]
```

At this point we may state that readability does not improve in any case when list comprehensions get longer.

Set comprehensions

Similar so list comprehensions, there exist set comprehensions that have a similar syntax but yield sets. You only need to replace the squared brackets by curly braces:

```
>>> s = {-x if x < 0 else x for x in range(-10, 11) if x % 2 == 0}
>>> s
{10, 8, 6, 2, 0}
```

Dictionary Comprehensions

In addition to list and set comprehensions, there exist dict comprehensions. They differ from set comprehensions in that the expression contains a colon to separate key and value of each entry. In the following example we use an `enumerate` object and use the tuple's first element as key and the second element as value.

```
>>> names = ["John", "Eric", "Michael", "Terry"]
>>> d = {idx: name for idx, name in enumerate(names)}
>>> d
{0: 'John', 1: 'Eric', 2: 'Michael', 3: 'Terry'}
```

Nested Comprehensions

Similar to nested loops we can also nest list comprehensions into each other. There are no new rules since you can just use a comprehension as an expression in another comprehension. The result will be a list of lists.

```
>>> pows = [ [x ** p for x in range(1, 6)] for p in range(2, 5)]
>>> pows
[[1, 4, 9, 16, 25], [1, 8, 27, 64, 125], [1, 16, 81, 256, 625]]
```

If the inner brackets are left out, all values are in a single list. Be aware that in this case the latter `for` loops are the inner loops whereas the first `for` denotes the outer loop. As a result, the order of the resulting list differs from the previous example.

```
>>> pows = [ x ** p for x in range(1, 6) for p in range(2, 5)]
>>> pows
[1, 1, 1, 4, 8, 16, 9, 27, 81, 16, 64, 256, 25, 125, 625]
```

When you are not entirely sure, you might use usual `for`-loops as well.

P.5 Functions

P.5.1 Function Definition

In the simplest case, functions are shortcuts for pieces of code that you need to run more than one time. Lets say, we want to print the first four lines of a Shakespeare sonnet:

```
>>> def print_sonnet():
...     print("Shall I compare thee to a summer's day?")
...     print("Thou art more lovely and more temperate:")
...     print("Rough winds do shake the darling buds of May,")
...     print("And summer's lease hath all too short a date:")
...
>>> print_sonnet()
Shall I compare thee to a summer's day?
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date:
```

Now, whenever we type `print_sonnet()` in the console, the same four lines of code are executed. As you see, functions are defined using the keyword `def` followed by the function name, in this case `print_sonnet`. Normally, function names are written in lower case and if they consist of multiple words, these are separated by underscores. After the function name there are parentheses and a colon. Similar to loops and if-statements the colon marks the begin of the function body which is also indented. The function ends with the first line of code that is not indented.

P.5.2 Input Arguments

Despite its literary value, the preceding example is only of limited use for programming, since the function behaves identical in each call. In most cases, we need functions to behave according to an input that we define somewhere else in the code. Assume you want to compute the double and the triple of an arbitrary value.

```
>>> def print_double_triple(val):
...     print("Single:", val, "Double:", val*2, "Triple:", val*3)
...
>>> for field in range(16, 21):
...     print_double_triple(field)
...
Single: 16 Double: 32 Triple: 48
Single: 17 Double: 34 Triple: 51
Single: 18 Double: 36 Triple: 54
Single: 19 Double: 38 Triple: 57
Single: 20 Double: 40 Triple: 60
```

In this example, the function depends on `val` which can be any value later on. In the function definition, we denoted this with the arguments name in the parentheses after the function name. During the function execution, `val` is treated as a variable in the workspace but is deleted as soon as the function finishes. The value that `val` takes during the execution is specified when the function is called, e.g. when you call `print_double_triple(13)`, the variable `val` has the value 13 during the function execution.

Note, that we did not need to specify the data type of `val` because Python does not require it. Since the `*` operator is also defined for strings and lists, we can also call

```
>>> print_double_triple("Spam")
Single: Spam Double: SpamSpam Triple: SpamSpamSpam
```

P.5.3 Output Values

In the last example the results of the computation were only printed in the console. In most cases we need the results of a function in the rest of the program. Hence it would be convenient to save the results in a variable. The following example shows how that can be done.

```
>>> def comp_double_triple(single):
...     return single*2, single*3
...
```

```
>>> d, t = comp_double_triple(19)
>>> t
57
```

Like in C or java, we use the `return` statement, to pass the results from the function to the caller. As shown in this example, it is even possible to return multiple values at once of these are comma separated. When we call the function, the returned values are assigned to `d` and `t`.

When a function returns multiple values but you assign the results to only one (or no) value, a tuple is returned that contains all output values. For instance, `comp_double_triple(14)` returns `(28, 42)`. This differs from Matlab, where in this case only the first value is returned.

P.5.4 Docstrings and Help

When you write code, you know what each function does, which arguments it receives and what it returns. However, this might not be the case as soon as your colleague needs to work with your code or vice versa. Therefore, you should always give a short description what a function does or even better, what it receives and returns. Unlike comments within your code, docstrings are delimited by `"""triple double quotes"""`. While comments are entirely ignored by the interpreter, the docstring is still used after the function definition. In case you do not know what a function does, you can call `help(func)` and you see the docstring. Note that the function `help` receives a function pointer, thus you need to leave the parentheses.

```
>>> def do_nothing(whatever):
...     """Do precisely nothing, whatever argument you pass to it."""
...     pass
>>> help(do_nothing)
Help on function do_nothing in module __main__:

do_nothing(whatever)
    Do precisely nothing, whatever argument you pass to it.
```

The docstring can be accessed using the functions `__doc__` property.

```
>>> do_nothing.__doc__
Do precisely nothing, whatever argument you pass to it.
```

This is especially useful when you forgot the exact behaviour of a specific function during coding and you want to avoid using a well known search engine in the internet.

```
>>> s = set([1, 2, 3])
>>> help(s.remove)
Help on built-in function remove:

remove(...) method of builtins.set instance
```

```
    Remove an element from a set; it must be a member.

    If the element is not a member, raise a KeyError.
```

P.5.5 Default Arguments

You can specify default arguments using an equal sign after the argument name in the function definition. If the argument is not specified during run time, the default argument is used instead.

```
>>> def compute_pow(x, p=2):
...     print(x, "to the power of", p, "equals", x**p)
...
>>> compute_pow(5)
5 to the power of 2 equals 25
>>> compute_pow(5, 3)
5 to the power of 3 equals 125
```

The default argument is evaluated only once. If it is an object it will not be reconstructed after multiple function calls. If this default object is modified during the function execution, in the next function call it will start with the modified value and not with the default value. See the following example.

```
>>> def append(x, L=[]):
...     L.append(x)
...     return L
...
>>> append(10)
[10]
>>> append(11)
[10, 11]
>>> append(4, [1, 2, 3])
[1, 2, 3, 4]
>>> append(12)
[10, 11, 12]
```

When the function was called the second time, `L` already had the value `[10]` but not `[]`. Even after the function was called with a specified `L`, the modified default argument was kept (`[10, 11]`).

You can avoid this behaviour with the following workaround:

```
>>> def append(x, L=None):
...     if L is None:
...         L = []
...     L.append(x)
...     return L
...
```

`None` is a Python keyword to define no value at all. It is to some extent comparable to a null-pointer in C++ or Java.

P.5.6 Positional and Keyword Arguments

In the examples we saw so far, the order of function arguments was determined by the function definition. If `x` is the first argument in the function definition, we need to pass `x` to the function first.

```
>>> def print_values(x, y, z):
...     print("x equals %d, y equals %d and z equals %d" % (x, y, z))
...
>>> print_values(3, 4, 5)
x equals 3, y equals 4 and z equals 5
```

Arguments that are mapped by their position are *positional arguments*. However we can also specify arguments by their name. This makes code more understandable in many places.

```
>>> print_values(z=5, x=3, y=4)
x equals 3, y equals 4 and z equals 5
```

Albeit the arguments were passed in another order, they were mapped correctly. When you call a function and use a keyword argument, all following arguments must be keyword arguments as well. For instance `print_values(3, z=5, 4)` would raise a syntax error.

P.5.7 Argument Lists

Argument Lists are used when it is unclear how many arguments a function will receive. They allow the function to be called with a flexible number of positional arguments.

```
>>> def print_greets(name, *titles):
...     pass
```

This function requires one positional argument and allows arbitrary more arguments. E.g. we can call `print_welcome('Bob')`, `print_welcome('Bob', 'Mr.')` or `print_welcome('Ruediger', 'Mr.', 'Univ.-Prof.', 'Dr.rer.nat.', 'Dr.h.c.mult.')`. In the function body, all these arbitrary many arguments are wrapped into a single tuple, named `titles`. The order within the tuple is the same order with which the arguments were passed. If no argument other than `name` is given, the tuple is empty.

```
>>> def print_welcome(name, *titles):
...     print("Hello ", end='')
...     for t in titles:
...         print(t, end=' ')
...     print(name + "!")
...
>>> print_welcome('Bob')
Hello Bob!
>>> print_welcome('Ruediger', 'Mr.', 'Univ.-Prof.', 'Dr. rer. nat.', 'Dr. h.
↪  c. mult.')
Hello Mr. Univ.-Prof. Dr. rer. nat. Dr. h. c. mult. Ruediger!
```

In this example, the individual arguments represented by `titles` do not have keywords and are hence positional. Nonetheless, argument lists are also possible for arbitrary many keyword arguments. To do so, two asterisks are used and the function receives a dictionary instead of a tuple.

```
>>> def make_menu(**kwargs):
...     print("Today we offer you", len(kwargs), "courses:")
...     for key in kwargs:
...         print("As", key, "we serve", kwargs[key])
...
>>> make_menu()
Today we offer you 0 courses:
>>> make_menu(starter="mushroom soup", main_course="Eggs and Spam",
↪  dessert="strawberry ice cream with spam", wine="Chateauneuf-du-Pape")
Today we offer you 4 courses:
As starter we serve mushroom soup
As main_course we serve Eggs and Spam
As dessert we serve strawberry ice cream with spam
As wine we serve Chateauneuf-du-Pape
```

You can also combine positional argument lists with keyword argument lists. You just have to make sure that the positional argument list comes first in the function definition. For instance you could define `print_welcome(name, *titles, **courses)` to welcome your guests and list the menu of the day.

In the Python documentations argument lists are mostly denoted by `*args` and `**kwargs`.

P.5.8 Unpacking Argument Lists

In the last section about argument lists, many arguments were passed to a function. The multitude of these arguments is represented by `*args` or `**kwargs`. When the asterisks are removed these separate arguments are packed into a tuple or a dictionary. The very same process can be done in reverse which is named *unpacking*. Instead of packing multiple arguments into a tuple, a tuple or a list can be unpacked to several arguments.
Of course you can remember the command `range(start, stop, end)`. Using the unpacking operator, you can call the function with a single list or tuple.

```
>>> params = [12, 19, 2]
>>> r = range(*params)
>>> list(r)
[12, 14, 16, 18]
>>> print(r)  # one argument r
[12, 14, 16, 18]
>>> print(*r)  # for arguments 12, 14, 16, 18
12 14 16 18
```

The same is possible for dictionaries. If the function supports keyword arguments but no argument lists, you can pass an unpacked dictionary and the function will receive its entries as keyword arguments.

```
>>> param_dict = {'start': 5, 'step': 5, 'end': 30}
>>> r = range(**param_dict)
>>> list(r)
[5, 10, 15, 20, 25]
```

When `print(l)` is called, the function receives a list. In the next line, the list is unpacked so that the print command receives three arguments that are printed one after another.

Apart from function definitions, the unpacking operator can be useful in other places. In the following example, two lists are to be concatenated:

```
>>> l = [1, 2, 3]
>>> [l, 6, 7, 8]
[[1, 2, 3], 6, 7, 8]
>>> [*l, 6, 7, 8]
```

With the same principle, dictionaries can be merged:

```
>>> d1 = {'a': 123, 'b': 456}
>>> d2 = {'c': 789, 'd': 012}
>>> {**a, **b}
{'a': 123, 'b': 456, 'c': 789, 'd': 012}
```

If two dictionaries are merged and a value occurs twice, later occurrences override earlier values.

P.5.9 Global and Local Scope

During the execution of a function, the interpreter has access to all variables that were defined in the function before the current line. As soon as the function finishes, these variables are deleted. The totality of variables is referred to as *scope* and in this particular case *local scope* since it contains the local variables in the function. Apart from that, there is the *global scope* containing all variables that were defined in the program but not in the current function or within any other function. If a function is called by another function, the scope of the calling function is referred to as *nonlocal scope*.
When a variable in a function is accessed, it is first searched in the innermost scope. If it does not exist there, the next outer scope is searched until the global scope is reached.

```
>>> x = 5
>>> y = 6
>>> def func():
...     x = 7
...     print("x = %d, y = %d" %(x, y))
...
>>> func()
x = 7, y = 6
>>> print(x)
5
```

Since `x` was defined in the local scope this value was used, whereas `y` only exists in the global scope.
Even though we can read variables from the outer scope we can not modify them. To do so, we first need to declare that to the interpreter, using the `global` keyword.

```
>>> x = 5
>>> def inc_x_by(val):
...     global x
...     x += val
...
>>> inc_x_by(3)
>>> x
8
```

P.5.10 Function Objects and Lambdas

In some situations it happens that a function needs to be passed to another function. For instance, when you want to build a calculator app and the function `compute(a, b, op)` receives two operands `a` and `b` as well as an operation `op` to perform on those. In this case, `op` is a *function object*.

```
>>> def plus(a, b):
...     return a + b
...
>>> def minus(a, b):
...     return a - b
...
>>> def compute(a, b, op):
...     return op(a, b)
...
>>> compute(8, 2, plus)  # plus is a function object
10
>>> compute(8, 2, minus)  # minus is a function object
6
```

The function object is accessed by writing the name of the operation without parentheses. When the function object is called with parentheses in the first line of `compute(a, b, op)` the actual function corresponding to the function object is executed.
Using function handles, it is possible to make copies of a function. The actual name of the original function can be accessed using its `__name__` property.

```
>>> def a(x):
...     return x**2
...
>>> b = a
>>> b.__name__
'a'
```

When you type a name of a function without arguments in your code and forget the parentheses, the interpreter does not execute the function. Instead, the function object is displayed when you are in terminal mode. In a Python script, nothing happens and nothing will indicate a possible error.

By use of the `lambda` keyword, it is possible to define function objects without the usual `def` syntax. The lambda syntax is `lambda` *variables* : *body*. With the use of lambdas, the previous example reads:

```
>>> def compute(a, b, op):
...     return op(a, b)
...
>>> compute(8, 2, lambda a, b: a+b)
10
>>> compute(8, 2, lambda x, y: x-y)
6
```

Like usual functions, lambdas also accept argument lists:

```
>>> f = lambda *args: print(args[1])
>>> f("Hello", "How", "Are", "You")
How
```

P.5.11 Decorators

A decorator is a syntactic feature in Python that allows to modify functions by adding more functionality to it. The particular functionality that is added, is provided by the decorator. For instance, there exist decorators that measure and display the execution time of a function each time that it is called. In another scenario, you might only want to execute a function when there is enough memory available. The next examples explain, how decorators work and how they can be implemented.

In the last section we encountered that function objects simply arise when we call a function name without parentheses. Decorators are a type of function that always receive a function object, wrap functionality around it and return a function object that performs the received function with the added functionality.

```
>>> def the_decorator(func):
...     def wrapper(*args, **kwargs):
...         print("Buckle up, now I execute", func.__name__)
...         print("args:", *args, "kwargs:", kwargs)
...         out = func(*args, **kwargs)
...         print("That was a blast, the output is", out)
...         return out
...
...     return wrapper
...
>>> def f(x):
...     return x**2
...
>>> f = the_decorator(f)
```

Let's look carefully at what happened here. `the_decorator` is a function that receives a function handle `func`. Then, within `the_decorator` a new function `wrapper` is defined. The expression `*args, **kwargs` is a common pattern with which a function can receive any combination of arguments. The wrapper prints these arguments, executes `func` with them and returns the output that is produced by `func`. What is then returned by `the_decorator` is a function object that contains `wrapper`. More specific, the function that is returned by `the_decorator` contains the function that was passed to it in the first place, but with three lines of additional code before and after it. With the line `f = the_decorator(f)` the behaviour of `f` is overridden and at each call of `f`, the additional lines are executed before and after it.

```
>>> y = f(5)
Buckle up, now I execute f
args: 5 kwargs: {}
That was a blast, the output is 25
```

This process is referred to as *decoration*, where `the_decorator()` is the actual decorator. The function `f` is decorated with additional code.

Lets say, we have a function `load_train_data()` that must not be executed when there is not enough free space in memory. In addition, we want to know the free memory after the data has been loaded.

```
>>> import psutils  # checks current memory status
>>> def mem_checker(func):
...     def wrapper(*args, **kwargs):
...         mem = psutil.virtual_memory()
...         if mem.available > 3E9:
...             out = func(*args, **kwargs)
...             mem = psutil.virtual_memory()
...             print("Loaded Data, remaining memory is %2.1f GB" %
↪  (mem.available / 1E9))
...             return out
...         else:
...             print("Cannot run %s with %2.1f GB Memory" % (func.__name__,
↪  mem.available / 1E9))
...             return None
...
...     return wrapper
...
```

The wrapper checks whether there is enough free memory and if so, the wrapper calls the function that was initially passed to `mem_checker` with the arguments that were passed to `wrapper`. If there is not enough memory, a warning is printed.

Since decorators are widely used in advanced Python, there exist a shortened syntax to apply an existing decorator to a new function:

```
>>> @mem_checker
... def load_train_data():
...     return [[1, 2, 3, 4]], [42]
...
>>> X, y = load_train_data()
Loaded Data, remaining memory is 3.5 GB
```

As a result, you can easily add decorators to a function by adding the name of the decorator before the function definition. Typical use cases for decorators are debugging and timing. With regard to timing, you can measure the execution time of a function or inhibit too many calls in a row. When it comes to object oriented programming, decorators are used to give class methods specific properties.

P.6 Classes

Classes come into play when we need to store data and want to perform routines based on this data. In Section P.3 we already encountered lists, sets, dictionaries and others. These objects could store data and provided functionality to either access, add or remove content or to check whether some data is contained. A more practical example is a neural net that also contains data, namely its weights. In addition, it provides functions that either adapt these weights or make a prediction based on these weights. Naturally, we can create many instances of one class that store different data but provide the same functionality.

P.6.1 Class Definition

Basic Usage

Like with the definition of functions, classes can be defined anywhere in your code as well as in the console. Similar to the definition of functions using `def`, classes are defined using the keyword `class`. The next example illustrates the class definition syntax:

```
>>> class Counter:
...     counts = 1
...
...     def inc(self, steps=1, verbose=True):
...         self.counts += steps
...         if verbose:
...             print("Count:", self.counts)
...
>>> c = Counter()
>>> c.counts
1
>>> c.inc(5)
Count: 6
```

For no surprise, the class definition consists of an indented block and is announced by its name and a colon. Usually, class names start with a capital letter. If they consist of multiple words, they are written in camel case, e.g. `FrequencyDomainAdaptiveFilter`. In the next line, a public attribute variable is defined and assigned. Once an object of type `Counter` was created, the attribute can be accessed and assigned using the point notation.

Another type of attribute that a class can have are *methods*. These are functions that belong to a class. They are defined the same way as ordinary functions with the only difference that methods always receive the calling class instance as a first function argument. This argument is named `self` by convention but can have any other name.

When the object instance `c` calls the method `inc` in the previous example, the method has access to the member (`counts`) of `c` via the `self` argument.
As soon as the class definition has been interpreted without errors, instances of the class can be created. This is called *instantiation* and can be done with the same syntax that is used to call functions. Usually, you want to assign the created instance to a local variable (here `c`).
Since member functions always receive the calling instance as a first argument, you have to 'skip' this argument when you call a method. When `c.inc(5)` is called, the first argument (`self`) is already given by `c`. The first argument in the parentheses is the second argument `steps` in the function definition.

If you have studied classes in C++ or java you might wonder how we defined a class without a constructor. If not, you might now wonder what a constructor is. In the last example, all objects of type `Counter` would be instantiated with `counter=1`. However, we might want to be able to create individual counters with a specific start value. Moreover, we might want to set the `verbose` property as soon as the counter is created. Such a sequence of commands that is executed immediately after the object has been created can be implemented by means of an `__init__` method. It must always be named `__init__` with two underscores at the beginning and at the end of the function name. Its first argument is the object instance `self` and there can be any number of additional arguments.

```
>>> class Counter:
...     def __init__(self, count_init=1, verbose=True):
...         self.counts = count_init
...         self.verbose = verbose
...         if verbose:
...             print("Counts:", self.counts)
...
...     def inc(self, step=1):
...         self.counts += step
...         if self.verbose:
...             print("Counts:", self.counts)
>>> c = Counter(5)
Counts: 5
```

When you are about to define a member variable within the class scope, like in the first example, you should think twice since these are shared by all class instances. This leads to unusual behaviour, when the member variable is mutual (see next example). In contrast, all attributes that are defined in the `__init__` method are individual for each instance.

```
>>> class ShoppingList:
...     items = []  # shared by all ShoppingLists
...
...     def add_item(self, *items):
...         self.items.extend(items)
...
...     def print(self):
...         print(*self.items, sep=', ')
```

```
...
>>> list_groceries = ShoppingList()
>>> list_groceries.add_item("Eggs", "Spam")
>>> list_groceries.print()
Eggs, Spam
>>> list_gardening = ShoppingList()
>>> list_gardening.add_item("Sunflowers")
>>> list_gardening.print()
Eggs, Spam, Sunflowers
>>> list_groceries.print()
Eggs, Spam, Sunflowers
```

This behaviour is circumvented by defining `items` in `__init__`.

Private Variables

Private variables are variables that are not accessible from the outside of a class. In Python, this concept is not embedded in the native language. However, there exist a convention that private variables in a class begin with a single or double underscore. If you work with someone elses API and you see that a class variable begins with an underscore, it is supposed to make you understand 'Stay away!'. Surely you can investigate the variables as is possible with ordinary debuggers, but your code should in no way call these variables or rely on them. In most IDEs this will give you a condemning warning.
With two underscores, this idea is taken even one step further, as the following example demonstrates:

```
>>> class ClassTest:
...     def __init__(self, a, b)  # not these underscores
...         self._a = a
...         self.__b = b  # these
...
>>> c = ClassTest(3, 5)
>>> c._a  # just look, don't touch
3
>>> c.__b
AttributeError: 'ClassTest' object has no attribute '__b'
>>> c._ClassTest__b
5
```

As you see, the variable `__b` is internally replaced by `_ClassTest__b` so that there is no attribute `__b` in class `ClassTest`. This mechanism is referred to as *name mangling*.

Static Methods and Factory Methods

Static methods are methods that do belong to a class but are not bound to an instance of this class. Lets say we have a class `TFSignal` that represents a short time spectral analysis of an audio signal. A function that maps a frequency to the corresponding bark band is useful for all instances of `TFSignal`. However, sometimes we need this function independent of any signal. A sign, that the method should rather be static is given by the fact, that the function `hz2mel` does not rely on any property of a class instance.

In Python, static methods can be created using the `@staticmethod` decorator. Naturally, they do not receive a class instance as first argument.

```
>>> from math import atan
>>> class TFSignal:
...     @staticmethod
...     def hz2bark(fhz):
...         return 13*atan(0.00076*fhz) + 3.5*atan((fhz/7500)**2)
...
>>> TFSignal.hz2bark(100)
0.9867265581717046
>>> tfs = TFSignal()
>>> tfs.hz2bark(200)
1.9634785895415792
```

A special kind of static methods are so called factory methods. These are static methods that have the aim to construct an instance of a class. Assume you have a class `Signal` that represents a signal in time domain. It can be created from a pair of data and a sampling frequency. For convenience, we provide functions to create typical signals, as for instance a sweep or white noise. These functions, that create an object instance but are not the `__init__` method are called factory methods. They use the `@classmethod` decorator and always receive a class object as first argument. Class objects are comparable to function objects and must not be confused with class instances. Like class instances are usually denoted by `self`, class objects are denoted by `cls`.

```
import numpy as np
import matplotlib
from matplotlib import pyplot as plt
matplotlib.use("Qt5Agg")

class Signal:
    """Represents a Signal consisting of data and sampling frequency"""

    def __init__(self, data, fs):
        self.data = data
        self.fs = fs

    @classmethod
    def make_sweep(cls, f_start, f_end, fs, duration=5):
        """Factory method for sweeps"""
        t = np.arange(duration*fs) / fs
        f = np.linspace(f_start, f_end, len(t))
        data = np.sin(2*np.pi*f*t)
        return cls(data, fs)

    @classmethod
    def make_white_noise(cls, fs, v_max=1, duration=5):
        """Factory method for white noise"""
        data = np.random.rand(fs*duration)
        data -= np.mean(data)
        data *= 2*v_max
        return cls(data, fs)
```

```
    def plot(self):
        """Plot the signal"""
        t = np.arange(len(self.data)) / self.fs
        plt.plot(t, self.data)
        plt.xlabel('t [s]')
        plt.ylabel('Amplitude')
        plt.show()
```

Then we can easily create specific `Signal` instances. The result looks as depicted in Fig. P.2. The matplotlib.pyplot module will be covered in more detail in Section P.9.

```
>>> s = Signal.make_sweep(2, 5, fs=200, duration=5)
>>> s.plot
```

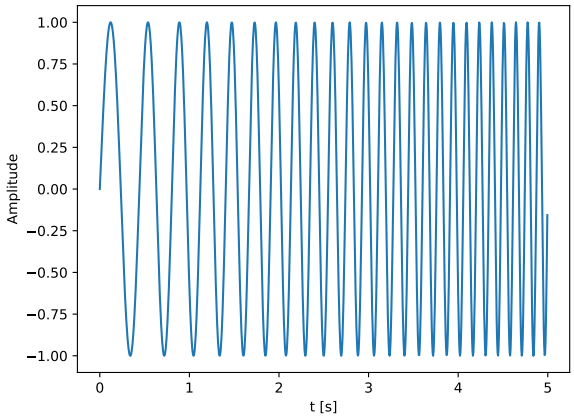


Figure P.2: A plot of a sweep signal

Property Decorators

From time to time we encounter the situation that a property of a class needs to be computed at the time that it is accessed. From another point of view it might be too expensive to compute all properties in advance if they might not be needed at all. This means that we need an option to perform computations or invoke commands as soon as the user requests a class attribute using the dot notation. In other situations, we want to perform safety checks, when the user tries to set a member variable. Again, we need to invoke commands as soon as the user tries to touch the attribute. One way to achieve this behaviour is the `@property` decorator. If a function `var(self)` is decorated with `@property`, this function is invoked, every time the user wants to access `obj.var`. What is returned to the user, depends on what is returned by `var(self)` When another function `var(self, value)` is decorated with `@var.setter`, this function is invoked every time the user tries to set `obj.var`.

In the following example, an FIR-Filter class is implemented. For efficiency, the filter output $y = f * h$ shall only be computed on demand and only be recomputed when

either x or h were modified. To do so, we implement setter and getter for the class attributes `y`, `x` and `h` using the `@property` decorator.

```
from scipy.signal import convolve
import warnings

class Filter:
    def __init__(self, x, h):
        self._x = x
        self._h = h
        self._y = None
        self.recompute = True

    @property
    def y(self):  # only recompute if something changed
        if self.recompute:
            print("Recompute output")
            self._y = convolve(self.x, self.h)
            self.recompute = False
        else:
            print("Can reuse stored y")
        return self._y

    @y.setter  # forbid to change output
    def y(self, value):
        warnings.warn("Filter output can not be set")

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, new):
        self._x = new
        self.recompute = True

    @property
    def h(self):
        return self._h

    @h.setter
    def h(self, new):
        self._h = new
        self.recompute = True
```

In the getter for `y`, the computation is only performed when `self.recompute` is true. Otherwise, the private variable `self._y` is returned. In the setter for `y`, we forbid to change the output, since we did not write `self.y = value`. Thus, the setter does not modify `y`. In the setter for `x` and `h`, the `recompute` flag is set to `True`.

```
>>> import numpy as np
>>> x = np.random.rand(256)
>>> h = np.random.rand(32)
>>> f = Filter(x, h)
```

```
>>> y1 = f.y
Recompute output
>>> y2 = f.y
Can reuse stored y
>>> f.h = np.random.rand(64)
>>> y3 = f.y
Recompute output
```

P.6.2 Inheritance

An important concept in object oriented programming is inheritance and fortunately it is also implemented in Python. Inheritance is a mechanism with which so called *child classes* can be derived from *base classes*. These child classes offer the same attributes and functions like their base classes but are able to implement additional functionality and parameters, the base class does not have. This will save many lines of redundant code, when multiple classes have functionality in common.

Basic Usage

The following example illustrates the basic scheme for inheritance.

```
>>> class Animal:
...     """Base class for animals"""
...     def __init__(self, name):
...         self.name = name
...         print("I am an animal and my name is", name)
...
>>> class Dog(Animal):
...     """Derived class for good girls and boys"""
...     def __init__(self, name, breed):
...         super().__init__(name),
...         self.breed = breed
...         print("I am a", breed)
...
...     def protect_home(self):
...         print("Bark Bark!")
...
>>> wesley = Dog('Wesley', 'Golden Retriever')
I am an animal and my name is Wesley
I am a Golden Retriever
```

The first of two key points is that the base class is denoted in parentheses after the class name. This is not to be confused with class arguments. In the `__init__` method of class `Dog` we made use of the `super()` command. Within a class, the use of `super()` is equivalent to a call to the base class with `self` as first argument. In this case, `super().__init__(name)` equals `Animal.__init__(self, name)`.

A class can inherit methods from multiple base classes. The syntax to do so is `class Derived(Base1, Base2, Base3):`. Then, `super().__init__()` calls `__init__()` for all base classes in a row.

Abstract Classes

Abstract classes are classes that serve as base class but can not be instantiated themselves. We already encountered a popular example in the last section, namely class `Animal`. Any animal that exists on our planet is not only an animal but a specific kind of animal, for instance a dog, Python, koala bear or anything else. But no animal is only an animal. In programming terms, we can derive any class from `Animal` but we can not create instances of type `Animal` itself. In Python, this concept is not included in the native language itself but nonetheless possible. All we need is a base class and a decorator.

```
>>> from abc import ABC, abstractmethod
>>> class Shape(ABC):  # since python 3.4
...     @abstractmethod
...     def area(self):
...         pass
...
>>> class Square(Shape):
...     def __init__(self, a):
...         self.a = a
...
...     def area(self):
...         return self.a ** 2
...
>>> s = Shape()
TypeError: Can't instantiate abstract class Shape with abstract methods area
```

A class is abstract as soon as it inherits from `abc.ABC` (short for Abstract Base Class) and has at least one method that is decorated with `@abstractmethod`.

P.6.3 Iterators and Generators

A few sections ago we encountered list comprehensions with the structure `[` *expression* `for` *variable* `in` *iterable* `]`. Albeit they make the code slim and elegant, a few things happen under the hood. The iterable is converted into an *iterator* object. It was said earlier that a datatype or a container is iterable if it is able to return its elements sequentially, one at a time. This is exactly what an iterator object does. Iterator objects provide a `__next__` method that for no surprise returns the next element. Such an object with such a method can be created using the function `iter(iterable)`. As a shortcut to `i.__next__()` you can call the built-in function `next(i)`.

```
>>> i = iter(range(2))
>>> next(i)
0
>>> next(i)
1
>>> next(i)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

When the iterator is finished, it raises a `StopIteration` exception that causes the for loop or list comprehension to stop.
The useful thing at this point is that you can easily use this API to create iterators on your own. The iterator is defined like a usual class and does not need to be derived from a iterator class or anything like this. It just needs to provide a `__next__` method. The following example implements an iterator that returns all files in a folder, one after another.

```
from os import listdir
from os.path import isfile, join, abspath

class DirIterator:
    def __init__(self, path, include_dirs=False):
        self.path = abspath(path)
        self.include_dirs = include_dirs
        self.files = listdir(path)

    def __next__(self):
        while True:
            if len(self.files) == 0:
                raise StopIteration
                return
            file = self.files.pop()
            if isfile(join(self.path, file)) or self.include_dirs:
                return join(self.path, file)
```

```
>>> path = 'C:\\Users\\iksuser\\pictures\\employees\\'
>>> i = DirIterator(path)
>>> next(i)
'C:\\Users\\iksuser\\pictures\\employees\\Assistant.jpg'
>>> next(i)
'C:\\Users\\iksuser\\pictures\\employees\\Professor.jpg'
>>> next(i)
'C:\\Users\\iksuser\\pictures\\employees\\Secretary.jpg'
```

This would continue until a `StopIteration` is thrown. Next, we could try to load all pictures in a for loop (given that the Iterator has not ended).

```
>>> import matplotlib.image
>>> for file in i:
...     img=matplotlib.image.imread(file)
...
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'DirIterator' object is not iterable
```

The problem here is that, as said before, the iterable in a for loop will be converted to an iterator. This holds even if it is already an iterator. To allow for that, the iterable must provide an `__iter__` method that simply returns itsself. So we have to add

```
    def __iter__(self):
        return self
```

to the class definition.

Honestly, this procedure seems to be a little cumbersome, given that we only want to iterate over a folder. In fact, it is preferable in more complex scenarios where a class exists that you can enrich with the `__iter__` and `__next__` methods. For simpler use cases like before, there exists a short cut, namely *generator objects*. You can think of the concept as implementing the `__next__` method directly without a class around it. The smart thing that makes a class less necessary is that local variables are *persistent*, meaning that they keep their value between subsequent calls.
From an implementation point of view, generator objects look just like normal functions but return values with the keyword `yield` instead of `return`. The last example would read as follows:

```
def dir_generator(path, include_dirs=False):
        path = abspath(path)
        files = listdir(path)
        while True:
            if len(files) == 0:
                return
            file = files.pop()
            if isfile(join(path, file)) or include_dirs:
                yield join(path, file)
```

The creation of `__iter__` and `__next__` happens automatically and `StopIteration` is raised as soon as the function terminates. The generator can be used as easy as

```
>>> files = [file for file in dir_generator(path)]
```

Admittedly, the files-in-dir example can be solved way more easy using a list comprehension with if clause directly. However, iterators and generators are a knight in shining armour when we need to load more data than fits in memory as it can occur in deep learning. Then, the respective chunks of data are not loaded before they are needed.

P.7 Modules

At the beginning of the section about functions it was said that once you defined a function and call it, it will repeat the same lines of code over and over. That was not entirely true: As soon as you restart the Python Console all variables and function definitions disappear. If you need to run functions in future sessions, they need to be stored in a separate .py file. This file can contain many function definitions and is called a *module*. The following module computes first and second order statistics of a list (or an iterable).

```
def mean(l):
    m = 0
    for x in l:
        m += x
    m /= len(l)
    return m

def var(l):
    m = mean(l)
    v = 0
    for x in l:
        v += (x - m) ** 2
    v /= len(l)
    return v
```

In order to use these functions, the module needs to be imported in our session (or in another .py file that contains a program). For the import, the file extension `.py` does not need to be added. Then, the functions from the module can be used by the name of the module and point notation. The `import` command is equal to running all definitions from the module in the current session.

```
>>> import second_order_stats
>>> L = [20, 35, 30, 40, 25]
>>> second_order_stats.var(L)
50
```

Since this long module name can become unhandy you can give any module an alias name in your program, using the keyword `as`.

```
>>> import second_order_stats as sos
>>> sos.var(L)
50
```

If even that is too long, you can import specific functions to be used without dot notation, using the keyword `from`.

```
>>> from second_order_stats import mean, var
>>> mean(L)
30
>>> var(L)
50
```

The following construct can often be found in python modules:

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

This allows modules to be executed as a script as well. The value of the variable `__name__` is different, depending on whether the module is being imported or executed itself. In the first case, `__name__` equals the name of the module. If you run a module, `__name__` is equal to `'__main__'` and the optional code is executed. This can be useful to run tests or a small main routine.

Apart from functions you can as well define variables in a module file. Inside the module these are global variables. From outside (where you load the module) they can be accessed in the same way as functions.

In each interpreter or jupyter session, every module is loaded only once. So if you do changes at your module file and import it again, no changes will be made. For the changes to take effect, the interpreter or jupyter kernel needs to be started again.

Often, many modules are grouped to a package. The import of a module from a package is similar to the import of a function from a module.

```
import mypackage
x = mypackage.mymodule.myfunction()

from mypackage import mymodule
x = mymodule.myfunction()

from mypackage.mymodule import myfunction
x = myfunction()

from mypackage.mymodule import myfunction as mfc
x = mfc()
```

The next sections will guide you through some of the most important packages in python.

P.8 NumPy

NumPy, short for Numerical Python is one of the most important packages for the use of Python in data science. In most Python projects using numpy the alias `np` is used so that you will read the line `import numpy as np` from time to time.

P.8.1 Numpy Arrays

Numpys core is a new data type named `numpy.ndarray` which presents an n-dimensional array. Other than lists in native Python, `ndarrays` are stored in consecutive cells in the computer memory similar to arrays in C and C++. As a result, they are better suited for iterations. Albeit they are intended to contain numbers, they are not restricted to that.

Construction

Like lists or tuples, ndarrays can be constructed from any iterable (lists, tuples...). To do so, the alias `array` can be used for ndarrays.

```
>>> import numpy as np
>>> x = np.array([1, 2, 3, 4])
>>> x
array([1, 2, 3, 4])
>>> print(x)
[1, 2, 3, 4]
>>> print(*x)
1, 2, 3, 4
```

From the last statement you can guess a very useful property of numpy ndarrays. Like lists and tuples, they are as well iterable and can be used for list comprehensions or can be unpacked.

Apart from `np.array(iterable)` there exists the constructor `np.ndarray(dims)` which has a different behaviour. In fact, both accept an iterable as input argument. But while the `np.array(iterable)` creates an array with the elements from the iterable, `np.ndarray(dims)` creates an array with the dimensions specified by `dims`. See the following example.

```
>>> np.array([2, 2, 3]) # creates an array containing [2, 2, 3]
array([2, 2, 3])
>>> np.ndarray([2, 2, 3]) # creates an 2 x 2 x 3 array
array([[[1.10039802e-311, 1.10041209e-311, 1.10041862e-311],
        [1.10042250e-311, 1.10041209e-311, 1.10041862e-311]],

       [[1.10042249e-311, 1.10041209e-311, 1.10042310e-311],
        [6.95189900e-310, 1.10041505e-311, 1.10042248e-311]]])
```

See that the second command creates a three dimensional $2 \times 2 \times 3$ array with floating point numbers close to zero. When numpy arrays are displayed to the console, the numbers in a line belong to the last (innermost) dimension. Subsequent lines belong to the second-to last dimension and subsequent blocks represent the first (outermost) dimension.

To create sequence of numbers, numpy supports the commands `np.arange(start, stop, step)` and `np.linspace(start, stop, num)`. The first behaves similar to the `range` command in native Python but returns a numpy array. If you do not

know the `range` command, yet: `np.arange(end)` returns an array with the numbers from `0` to `end-1`. `np.arange(start, end)` returns an array from `start` to `end-1`. `np.arange(start, end, step)` returns an array from `start`, increasing by `step` until `end`. In `linspace(start, stop, num)`, the third element specifies the number of elements in the array so that the actual step is computed based on that.

```
>>> np.arange(10, 50, 5)
array([10, 15, 20, 25, 30, 35, 40, 45])
>>> np.linspace(10, 50, 5)
array([10., 20., 30., 40., 50.])
```

In many cases, arrays are needed that contain only zeros and ones. These can be created with `np.zeros(dims)` and `np.ones(dims)`. If you need an array that has the same shape as another array, there are `np.zeros_like(other)` and `np.ones_like(other)`.

Vectors vs Matrices

One thing that is puzzling, especially for experienced Matlab users and mathematicians, is the difference between vectors and matrices with one column. Usually, one would expect, that an array of length N is the same as an $N \times 1$ matrix, but not so in numpy.

```
>>> a = np.array([1, 2, 3, 4]) # create a 1D-array
>>> a.shape
(4,)
>>> a.ndim
1
```

The result has only one dimension, hence it is neither a column, nor row vector. The attempt to append this as the last row or the last column onto an existing matrix would raise a `ValueError`. Instead, we would have to expand its dimension. This can be done with `np.newaxis` (which is an alias for `None`).

```
>>> a_row = a[np.newaxis, :]
>>> a_col = a[:, np.newaxis]
>>> a_row.shape
(1, 4)
>>> a_row.ndim
2
>>> a_col.shape
(4, 1)
```

The results are still vectors, only that they have two dimensions and are able to be concatenated with existing matrices.

Datatypes

All numpy arrays have a certain data-type. It can be accessed using the arrays `.dtype` property. When an element is added to or inserted into an existing array the new element is typecasted to fit the existing arrays data-type. The type can be casted using the `.astype` function on an array. At the construction time, the data-type can

be specified by passing the desired type as an argument with the keyword `dtype`. This holds for all creation methods from the last section.

```
>>> x1 = np.arange(10)
>>> x1.dtype
dtype('int32')
>>> x1[-1] = 3.333
>>> print(x1)
[0 1 2 3 4 5 6 7 8 3]
>>> x2 = x1.astype(np.float64) # Convert using astype
>>> print(x2)
>>> x3 = np.arange(10, dtype = np.float64) # Create with keyword argument
```

Read Access

Basically, all indexing rules that apply to lists also apply to np.arrays.

```
>>> x = np.arange(0, 14, 2)
>>> print(*x)
0 2 4 6 8 10 12
>>> x[:3]
[0, 2, 4]
>>> x[1:5]
array([2, 4, 6, 8])
>>> x[-1]
12
```

When it comes to arrays with more than one dimension, the indices for each dimension are separated by commas. In case of 3D arrays, the last index denotes the column, the second to last denotes the row and the first index denotes the slice (depth). With the slicing syntax submatrices can be acessed.

In the following example `np.reshape(array, dim)` is used to create a two-dimensional array from a one-dimensional array.

```
>>> x = np.reshape(np.arange(25), (5, 5))
>>> x
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
>>> x[0, 3]  # First row, fourth element
3
>>> x[:, -1] # The last column
array([4, 8, 14, 19, 24])
>>> x[::2, :2] # Every second row, first two columns
array([[ 0,  1],
       [10, 11],
       [20, 21]])
```

If only one argument `i` is given, but `x` has more than one dimension, this is interpreted as `x[i, :]` and returns the i^{th} row. Generally speaking, this accesses the array in the outermost dimension.

```
>>> x[0]
>>> x[-1] # The last row. Same as x[-1, :]
```

If you want to access an array in the innermost dimension, but the number of dimensions is not yet available, you can use the `...` operator. It expands to as many `:`, as needed in order to obtain a valid expression. For instance if `x` has four dimensions, `x[i, ...]` equals `x[i,:,:,:]` and `x[i,...,j]` equals `x[i,:,:,j]`.

Instead of the colon notation `start:end:stop` you can use arbitrary integer list to access the elements in arrays.

```
>>> x = np.reshape(np.arange(25), (5, 5))
>>> x[[0, 1, 3]] # first, second and fourth row
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [15, 16, 17, 18, 19]])
>>> x[1, [0, 2, 3]] # second row, first third fourth column
array([5, 7, 8])
>>> x[[0, 1], [3, 4]]
array([15, 21])
```

The last command, where two index lists were used, is equivalent to `[x[0, 3], x[1, 4]]`. In other terms, the two index lists (or arrays) `[0, 3]` and `[1, 4]` are zipped to index pairs. As a result, they need to have the same length. In the second example `x[1, [0, 1, 2]]` the first index 1 was broadcasted to `[1, 1, 1]`.

It is as well possible to access the elements of an n-dimensional array using tuples.

```
>>> x = np.reshape(np.arange(25), (5, 5))
>>> t = (1, 3) # second row, fourth column
>>> x[t]  # x[1, 3]
8
>>> l = [1, 3]
>>> x[l]
array([[ 5,  6,  7,  8,  9],
       [15, 16, 17, 18, 19]])
```

See that acessing an array with a list leads to a different behaviour than access using a tuple. The numbers in the list are interpreted as elements in the first dimension. The command `x[l]` is similar to `x[[1, 3]]`. In a one-dimensional array this would return the second and fourth element of the array. In the two dimensional case it is interpreted as `x[1, :]` which is `x[[1, 3], ...]`.

The syntax to access array elements can as well be used to modify the array at these places. You can either assign a scalar, or another array with the correct shape.

```
>>> x = np.reshape(np.arange(16), (4, 4))
>>> x[[0, 1], :] = -1 # set the first two rows to -1
>>> x[:, -1] = -np.arange(4) # set the last column to [0, 1, 2, 3]
```

Views and Copys

The difference between views and copies is not visible at first sight but can be quite costly if it is overseen. Assume you have to hand in your university diploma when you are applying for a job. The recruiter is very pleased that you have taken the course *Audio Processing using Python* and encircles it with a red pen. Later it turns out, that you accidentally handed in the very original that you received from university and you did not make a copy, so for the rest of your life *Audio Processing using Python* will be encircled in red.
In the Python analogue, the handed in document would be the *view*, albeit a *copy* would have been more reasonable. A view of a numpy array arises, when an array is indexed or sliced. The selection being returned points onto the same place in your memory. This relationship can be checked for, using the `base` command.

```
>>> grades = np.array([1.3, 1.7, 1.3, 2.0, 5.0])
>>> good_grades = grades[:4]
>>> good_grades.base
array([1.3, 1.7, 1.3, 2. , 5.0])
>>> good_grades.base is grades
True
>>> good_grades[0] = 6.0
>>> grades
array([6. , 1.7, 1.3, 2. , 5.0])
```

As you remember, the keyword `is` checks if two objects are the very same object. In this example, `good_grades` is only a view of `grades` and any changes that are made on `good_grades` would affect `grades` since it is the same place in memory. What's nasty about it is that the behaviour differs depending on the type of indexing. When you use a list or logical indexing (next section), this will create a copy.

```
>>> grades_passed = grades[grades < 5] # logical indexing
>>> grades_passed.base is grades
False
>>> grades_first_term = grades[[0, 1, 2]] # indexing using a list
>>> grades_first_term.base is grades
False
```

You can make a copy explicitely, using `array.copy()`. When you are not sure, whether a command returns a view or a copy it is a good advice to have a second look in the documentation or check it yourself using `base`.

Operations on Arrays and Broadcasting

When binary arithmetic operators are applied to numpy arrays of the same shape, the operation is applied element by element. This can significantly shorten your code since it avoids for loops or list comprehensions in many places.

```
>>> a = np.arange(3, 6) # [3, 4, 5]
>>> b = np.arange(1, 4) # [1, 2, 3]
>>> a + b
array([4, 6, 8])
>>> a - b
array([1, 2, 3])
```

```
>>> a ** b
array([3, 16, 125])
```

Under given circumstances it is possible to apply binary operations on arrays of different shapes. One situation where this is particularly useful is when one operand is a scalar.

```
>>> x = np.array([3, 5, 8, 13])
>>> x + 2
array([5, 7, 10, 15])
>>> 2 ** x
array([   8,   32,  256, 8192], dtype=int32)
>>> x % 2
array([1, 1, 0, 1], dtype=int32)
```

This principle, where the shape of one operand is expanded to fit to another operand is called *broadcasting*. It is applicable when one of the operands is a scalar. However it is not applicable, when both are `ndarrays` of different shapes. It is also applicable to logical operations.

```
>>> x = np.array([1, 2, 3, 4])
>>> x > 2
array([False, False,  True,  True])
>>> x == 3
array([False, False, True, False])
```

The comparison between an array and the scalar yields boolean arrays. This allows a particularly elegant way of indexing the array called *logical indexing* or sometimes *fancy indexing*. When an array is accessed using a boolean array, an array is returned containing only the elements where the index-array is `True`.

```
>>> x = np.arange(10)
>>> x[x > 5]
array([6, 7, 8, 9])
>>> x[x % 2 == 0]
array([0, 2, 4, 6, 8])
```

The principle of broadcasting can also be useful to modify multiple values at once.

```
>>> x = np.array([3, 4, 15, 2, 5, 2, 20, 2, 1])
>>> x[x > 10] = 10
>>> x
array([3, 4, 10, 2, 5, 2, 10, 2, 1])
```

In the second line of the preceding example, the principle of broadcasting was even used twice. First, `[x > 10]` expands to an array that is true where x is larger than 10 and false everywhere else. Then, the vector `x` is set to 10, at all positions where this logical array is `True`.

P.8.2 Mathematical Functions and Constants

All common mathematical functions are implemented in numpy. Usually they accept `np.arrays` as arguments so that the function is applied to each element in the array. For instance, if `t` is an array consisting of time steps, `np.cos(t)` returns a cosine oscillation that is evaluated at the given time steps.

Trigonometric and hyperbolic

The following trigonometric and hyperbolic functions are supported:

`sin`	`arctan`	`cosh`
`cos`	`arctan2`	`tanh`
`tan`	`deg2rad`	`arcsinh`
`arcsin`	`rad2deg`	`arccosh`
`arccos`	`sinh`	`arctanh`

Rounding

`around(x[, decimals])` rounds all elements in `x` to the given number of decimals (default 0).

`floor(x)` returns the floor of all elements in `x`.

`ceil(x)` returns the ceil of all elements in `x`.

Sums and Products

`prod(x)` returns the product of all elements in `x`.

`sum(x)` returns the sum of all elements in `x`.

`diff(x)` returns the differences (discrete derivative) between consecutive elements in `x`.

`cumprod(x)` returns the cumulative product of all elements in `x`.

`cumsum(x)` returns the cumulative sum of all elements in `x`.

The functions `nansum`, `nanprod`, `nancumsum`, and `nancumprod` act similarly but ignore nan values.

Exponents and Logarithms

`exp(x)` returns `e**p` for all `p` in `x`

`exp2(x)` returns `2**p` for all `p` in `x`.

`log(x)` returns the natural logarithm for all elements in `x`.

`log2(x)` returns the base-2 logarithm for all elements in `x`.

`log10(x)` returns the base-10 logarithm for all elements in `x`.

Routines for complex numbers

In Numpy, complex numbers arise in the same way as in native python. For instance, `np.exp(1j*0.25*np.pi)` creates a complex number with an absolute value of 1 and an angle of $\frac{\pi}{4}$. The following functions can process arrays of complex numbers.

`np.real(c)` produces the real part of the complex number or array of complex numbers c.

`np.imag(c)` produces the imaginary part of the complex number or array of complex numbers c.

`np.abs(c)` produces the absolute value of the complex number or array of complex numbers c.

`np.angle(c)` produces the phase in radian of the complex number or array of complex numbers c in the range from $-\pi$ to π.

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

Similarly, these commands can be used as attributes of an array `c` e. g. `c.real()` or `c.conj()`.

Constants

The following constants are implemented in Numpy:
`e`, `pi`, `inf`, `nan`
A more complete collection of scientific constants is contained in the `scipy` library (section. P.11.1).

P.8.3 Array Manipulation Routines

Shape Manipulations

`a.shape or np.shape(a)` returns the shape (or the dimensions) of array `a`.

`a.reshape(newshape) or np.reshape(a, dims)` changes the shape of `a` as specified by the tuple `newshape`.

`a.flatten()` returns a copy of the array where all elements are collapsed into one dimension.

`a.T` returns a transposed version of a but does not make a copy.

`a.squeeze() or np.squeeze(a)` removes singleton dimensions that contain no data. If `a.shape()` is `(3, 4, 1, 5)` the results shape is `(3, 4, 5)`.

`np.expand_dims(a, pos)` inserts an empty axis at position `pos`. This is the opposite of squeeze. The same can be achieved with `np.newaxis`. For instance `a[:, np.newaxis, ...]` is the same as `np.expand_dims(a, 1)`.

Note that these functions do not work in place. This means that when you call functions to manipulate an array, they return a manipulated copy of the array but leave the array unchanged. For instance, calling `a.flatten()` on an n-dimensional array will return a one-dimensional array, whereas a remains n-dimensional. In order to apply the operation on `a` you need to assign `a = a.flatten()`.

Join and Repeat Arrays

`np.concatenate(in, axis=0)` concatenates all arrays in the iterable `in` along an existing axis. For instance when each element in `in` is a 3x3 array and `in` has 5 elements, the result will be a 15x3 array. If the argument `axis` is given, it specifies along which axis the elements from `in` are concatenated, e.g. `axis=1` would return a 3x15 axis. The shape of the arrays in `in` must be equal except for the dimension along which the arrays are concatenated.

`np.stack(in, axis=0)` does a similar thing like `np.concatenate` but at a new dimension. If `in` contains 5 3x3 arrays, the result will be a 5x3x3 array. If the argument `axis` it specifies which axis will be the new one. `axis=-1` indicates that the last axis is the new one.

`np.vstack(in), np.hstack(in), np.dstack(in)` are shortcuts to concatenate or stack along the first three dimensions (depth, vertical, horizontal). Given that `in` contains 5 3x3 arrays, `vstack` returns a 15x3 array, `hstack` returns a 3x15 array and `dstack` returns a 3x3x15 array.

`np.tile(a, reps)` returns an array, where `a` is repeated as specified by `reps` and reps is a tuple of integers. The first entry of `reps` denotes how often `a` is repeated along the first axis. If `reps` has more entries than `a` has dimensions, new dimensions are appended.

Remove and Add Elements

Squared brackets denote optional arguments.

`np.delete(a, pos [, axis])` deletes the element from `arr` that has the index `pos`. For instance, `np.delete(a, 1)` will delete the element with index one. If you wish to delete all elements *equal to* one, call `x = x[x != 1]`. If the axis argument is specified, the entire row, column or slice is removed. Otherwise, the result is flattened to one dimension.

`np.insert(a, pos, values[, axis])` inserts a specific value before a specific position. If an iterable of positions is specified, the value is inserted before all positions in the iterable. Otherwise if an iterable of values is specified but only one position, all values are inserted before this position. If more than one value and more than one position are provided, the first value is inserted before the first position, the second value is inserted before the second position and so on. If `axis` is not specified or `None`, the resulting array is flattened. Otherwise, the values to be inserted are broadcasted to fit the desired dimension.

`np.append(a, values[, axis])` appends `values` to `a`. If `axis` is specified, the values are appended in this dimension, given that the shapes fit. If no axis is given, the result is flattened to one dimension.

`np.unique(a)` returns a copy of `a` where each value occurs only once. For instance, `np.unique([2, 1, 2, 1, 2])` returns an array containing `[1, 2]`.

Rearranging Elements

`np.flip(a[, axis])` reverses the order of elements along the dimension specified by `axis`. If no second argument is given, the order is reversed so that the first element of the output is the last element of `a`. The commands `np.flipud(a)` and `np.fliplr(a)` are shortcuts for flips in the first and last dimension respectively.

`roll(a, shift[, axis])` shifts the array elements circularly from left to right. For instance `np.roll(range(5), 2)` yields `[3, 4, 0, 1, 2]`. When the specified shift is negative, the elements are shifted from right to left. If no axis is specified or the axis is `None`, the elements are shifted along all dimensions, starting in the last dimension.

P.8.4 Indexing Routines

Find Elements

`np.nonzero(a)` or `a.nonzero()` returns the indices where the array `a` is not zero. This is particularly useful when `a` contains logic values. The function always returns a tuple of arrays, where the first array contains the indices along the first dimension and so on. Even if `a` is one dimensional, a tuple with one array is returned. This tuple can be used to access `a` or other arrays of the same shape. If applicable, logical indexing should be preferred, e.g. `a[a > 5]` instead of `a[np.nonzero(a>5)]`.

`np.count_nonzero(a)` returns the number of non zero elements in a.

`np.where(condition, x, y)` returns the elements of `x` where the condition applies and `y` otherwise. If `x` and `y` are not specified, the behaviour is similar to `np.nonzero`.

```
>>> # 1D case
>>> a = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5])
>>> np.nonzero(a == 1)  # returns a tuple
(array([1, 3], dtype=int64),)
>>> np.nonzero(a == 1)[0]  # array
array([1, 3], dtype=int64)
>>> a[np.nonzero(a==1)]  # use tuple for indexing
array([1, 1])
>>> a[a==1]  # logical indexing (recommended)
array([1, 1])
```

```
>>> a = np.random.randint(1, 10, size=(5, 5))
>>> a
array([[7, 9, 3, 6, 9],
       [5, 9, 3, 4, 7],
       [1, 1, 4, 8, 1],
       [1, 4, 1, 7, 7],
       [8, 9, 5, 2, 8]])
>>> idx = np.nonzero(a==9)  # tuple with both row and column indices
>>> idx
(array([0, 0, 1, 4], dtype=int64), array([1, 4, 1, 1], dtype=int64))
>>> a[idx]
array([9, 9, 9, 9])
>>> a[a==9]  # recommended
array([9, 9, 9, 9])
```

Create Index Grids

`np.indices(shape)` returns an ndarray `grid` that contains grid indices. `grid[0,...]` contains the indices along the first dimension and the `grid[1,...]` contains the indices along the second dimension and so on. For instance `np.indices((5, 5))` returns a `2x5x5` array, where `array[0, :, :]` contains the row indices and `array[1, :, :]` contains the column indices. You can provide multiple output arguments, to unpack the result to individual coordinate grids. For instance, `gx, gy = np.indices((5, 5))` creates two `5x5` arrays as described before.

`np.meshgrid(*ix)` does a similar thing as `np.indices` but receives vectors with the individual coordinates per dimension. Thus, the coordinate grid can cover an arbitrary range and the coordinates are not forced to start at zero.

P.8.5 Random Numbers

In order to create random numbers, it is recommended to create a Generator object. This can be done using `np.random.default_rng()`. The returned object then provides functions to generate random data. When an additional argument `seed` is provided at the construction of the generator object, the generator behaves reproducible. For instance, every generator created by `np.random.default_rng(42)` will create the same series of random number, whereas a generator created by `np.random.default_rng()` is entirely unpredictable.

Below is a list of member functions to create random numbers or perform random actions, once the generator object was created. Apart from that, there are many more distributions that can be found at `https://numpy.org/doc/stable/reference/random/generator.html#numpy.random.default_rng`. If applicable, the optional argument `size` specifies the shape of the returned array.

General Random Operations

`integers(high[, size])` or `(low, high[, size])` returns random integers in the range from low to high, including low and excluding high. If an additional argument `endpoint=True` is provided, the end is included.

`random([size])` returns random floating point values in the range $[0, 1)$.

`choice(a[, size])` returns a random sample from `a`.

`shuffle(a[, axis])` shuffles the elements in `a` in place (no copy is returned).

`permutation(a[, axis])` returns a permutation of `a`.

`binomial(n, b[, size])` draws samples from a binomial distribution.

`laplace([loc, scale, size])` draws samples from a laplace distribution.

`lognormal([mean, sigma, size]` draws samples from a log normal distribution.

`multivariate_normal(mean, cov[, size])` draws samples from a multivariate gaussian distribution.

`normal(loc=0, shape=1, size=1)` draws samples from a univariate gaussian distribution.

`triangular(left, mode, right[, size])` draws samples from a triangular distribution over the interval `[left, right]` with the maximum at `mode` where `left <= mode <= right]`

`uniform([low, high, size])` draws samples from a uniform distribution.

If you do not intend to call the random generator multiple times in a row, `numpy.random` provides convenience methods to generate random numbers with the same function signature. For instance you can replace `rng.random()` by `np.random.random()`.

P.8.6 Statistics

`np.amin(a[, axis])` or `np.min(a[, axis])` returns the minimum of `a`. If `axis` is specified, the minimum for each row, column, slice, etc. is returned.

`np.amax(a[, axis])` or `np.max(a[, axis])` returns the maximum of `a`, similar to `np.amin`.

`np.median(a[, axis])` returns the median value of `a`.

`np.mean(a[, axis])` returns the mean value of `a`.

`np.std(a[, axis])` returns the standard deviation of `a`.

`np.var(a[, axis])` returns the variance of `a`.

`np.corrcoef(x[, y, rowvar])` returns the correlation matrix between the rows in `x`, given that each row corresponds to a variable and columns correspond to observation. If `rowvar` is set to `False` (default: `True`), columns are treated as variables and rows are observations. If an additional array `y` of the same shape as `x` is given, the correlation between x and y is computed in addition.

`np.cov(x[, y, rowvar )]` returns the covariance matrix between the rows in `x`, given that each row corresponds to a variable and columns correspond to observation. If `rowvar` is set to `False` (default: `True`), columns are treated as variables and rows are observation. If an additional array `y` of the same shape as `x` is given, the covariance between `x` and `y` is computed in addition.

`np.correlate(a, v)` returns the cross-correlation sequence of the sequences `a` and `v`.

`np.histogram(a[, bins, density )]` computes the histogram of `a`. If `bins` is an integer, it represents the number of bins. If it is a monotonically increasing sequence, it specifies the bin-edges starting with the rightmost edge. If `density=True` the output is normalized to be a probability density function instead of counts.

In addition, the routines `nanmin`, `nanmax`, `nanmedian`, `nanmean`, `nanstd` and `nanvar` provide the same functionality as their namesakes but ignore all `nan` values in `a`.

P.8.7 Matrices and Linear Algebra

Products of Real and Complex Matrices

Many operations in digital signal processing can be expressed by means of inner products and matrix multiplications. For the following cases, assume that we want to compute a product between the matrices or vectors $\boldsymbol{x_1}$ (`x1`) and $\boldsymbol{x_2}$ (`x2`) and their dimensions allow to do so.

`np.dot(a, b)` When both `x1` and `x2` are 1-D arrays, the function `dot(x1, x2)` computes the inner product, yielding a scalar.

`np.matmul(a, b)` When `x1` and `x2` are 2-D arrays, the rules for matrix multiplications apply. The multiplication of a row vector with a column vector is identical to the inner product, but returns a 2-D-array with shape `(1, 1)`. The multiplication of a column vector with a row vector, however, returns an $N \times N$ matrix. If the dimensions are incompatible, a `ValueError` is returned. For matrix multiplications `dot(x1, x2)` can be used as well but it is recommended to use `np.matmul(x1, x2)` or `x1 @ x2`.

`np.vdot(a, b)` When `x1` and `x2` are 1-D arrays and `x1` contains complex numbers, the function `vdot` returns the inner product between the complex conjugation of `x1` and `x2`.

`a.T` The transpose of a 2-D array can be obtained by its attribute `T`. Unlike in Matlab, this does *not* perform a conjugate transposition. The latter can be obtained by `a.conj().T`.

Solving Linear Equation Systems

`np.linalg.qr(a)` computes the qr decomposition of a matrix and returns them as seperate output arguments

`np.svd(a)` computes the singular value decomposition $A = USV^H$ and returns `u`, `s` and `vh` as seperate outputs.

`np.linalg.eig(a)` computes an eigenvalue decomposition $A = V\text{diag}(W)V^{-1}$. Returns the eigenvalues `w` as first and the transformation matrix `v` as second argument.

`np.linalg.norm(a[, ord, axis ])]` computes the vector or matrix norm. If `a` is a matrix and `axis` is not specified, the norm is computed over all elements. Otherwise, the norm is computed row- or columnwise.

`np.linalg.det(a)` computes the determinant of matrix `a`.

`np.linalg.matrix_rank(M)` computes the rank of matrix M using a singular value decomposition.

`np.linalg.solve(a, b)` solves the set of equations $ax = b$ for x.

`np.lstsq(a, b)` returns the least square solution `x` for an overdetermined set of equations $ax = b$.

`np.linalg.inv(a)` returns the inverse of `a`.

`np.linalg.pinv(a)` returns the pseudoinverse of `a`.

P.9 Matplotlib

As soon as we exceed a certain amount of data, we need graphical tools to visualize it. This is where the matplotlib package comes into play. It provides the functionality that we need to create plots and figures. In the first part, we will deal with the matplotlib.pyplot API, which can be used to generate plots really quick. After that, we will deal with the more flexible object oriented API, which is advantageous when you are dealing with multiple plots and figures at once.
Before we do anything else, we:

```
>>> import matplotlib.pyplot as plt
>>> # and
>>> import numpy as np
```

If you do not want the plots to be shown inside the PyCharm IDE, you should switch to the Qt5 backend as follows:

```
>>> import matplotlib
>>> from matplotlib import pyplot as plt
>>> matplotlib.use("Qt5Agg")
```

P.9.1 Matplotlib.Pyplot

The `matplotlib.pyplot` API is useful when we quickly want to generate figures with as few lines of code as possible. The following sections give an overview over the most common commands from the matplotlib API.

Simple Plots

In the most simple case, we just call `plt.plot(x, y)` or even `plt.plot(y)` in order to create a simple line plot. The result of the following example is depicted in Fig. P.3.

```
>>> x = np.linspace(0, 4*np.pi, 500)
>>> y = 0.5 + 0.63*np.cos(x) -0.21*np.cos(3*x) + 0.12*np.cos(5*x) -
↪  0.09*np.cos(7*x)
>>> plt.plot(x, y)
>>> plt.show()
```

For not having to type `plt.show()` for each plot, you can turn on the interactive mode with `plt.ion()`. It is turned off with `plt.ioff()`. For the following examples, we assume that the interactive mode is on.

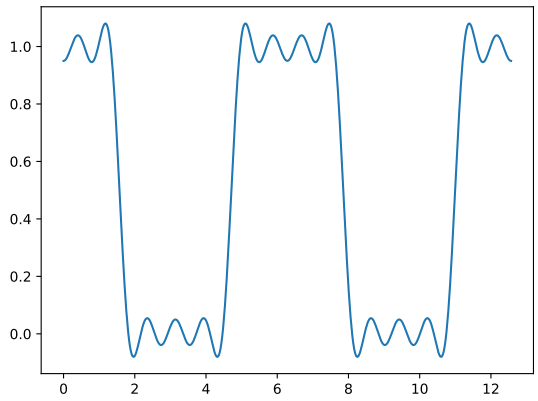


Figure P.3: A simple line plot

If only one argument is given, e.g. `plt.plot(y)` the x-data is created as `np.arange(len(y))`. The x-data does not necessarily need to be monotonically increasing, which is useful to create locus curves or lissajous figures as in the next example. The result looks as depicted in Fig. P.4.

```
>>> def lissajous(n, m, delta, N=500):
...     phi = np.linspace(0, 2*np.pi, N)
...     x = np.sin(n*phi)
...     y = np.sin(m*phi - delta)
...     return x, y
...
>>> x, y = lissajous(3, 5, np.pi/4)
>>> plt.plot(x, y)
```

Every further time that we call `plt.plot` a new line is drawn in the same coordinate system like the most recent line. We can clear the current coordinate system by `plt.cla()` which is short for 'clear axis', or clear the whole figure by `plt.clf()`. Once we are done, we can call `plt.close()` in order to close the current figure.

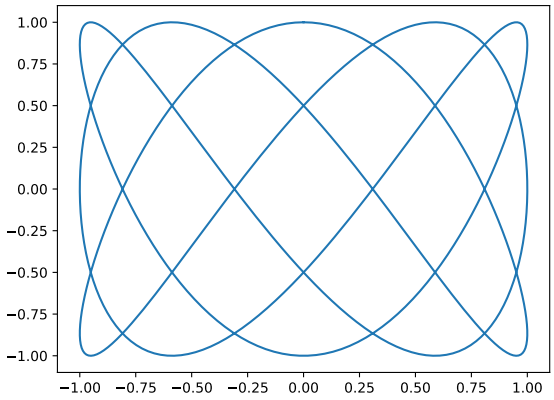


Figure P.4: Lissajous Figures

Customize Line Plots

Lets continue on the lissajous example. As soon as we want to show multiple lines in a single plot, they might be difficult to distinguish. By default, matplotlib uses different colours for plots that are placed in the same axis, however, this might only be of limited use if we want to publish our findings in a paper that is printed in black and white. In addition to colours, we can make plots more distinguishable by markers and line styles. Markers are symbols that are printed at each data point of the plot. We can combine the definition of colour, marker and linestyle in a single format string which is provided as a third argument after `x` and `y`. It has the structure `'marker line color'` where marker, line and colour are from the tables P.1 to P.3

Line string	description	Line string	description
`-`	solid	`-.`	dash dot
`--`	dashed	`:`	dotted

Table P.1: Line Style Options

Colour string	colour	Colour string	colour
`b`	blue	`m`	magenta
`g`	green	`y`	yellow
`r`	red	`k`	black
`c`	cyan	`w`	white

Table P.2: Line Colour Options

For example, the format string `'^g--'` would yield a dashed green line with triangles as markers. You can also provide only one or two arguments. For instance `'b'` would result in a blue solid line without markers and '*-' would become a solid line with star markers and default colour. In the following example, we use a for loop to try different combinations which generates the result depicted in Fig. P.5.

Marker string	Description	Marker String	Description
`.`	point	`+`	plus
`,`	pixel	`x`	x marker
`o`	circle	`h, H`	hexagon
`^, v, <, >`	triangles (rotated)	`d, D`	(thin) diamond
`s`	square	`*`	star
`|, _`	lines		

Table P.3: Marker Style Options

```
>>> colors = ['b', 'g', 'r', 'y']
>>> lines = ['-', '--', '-.', ':']
>>> markers = ['o', 's', 'D', 'H']
>>> phi = 0
>>> for it in range(4):
...     plt.plot(*lissajous(1, 2, phi, 30), markers[it] + colors[it] +
↪ lines[it])
...     phi += np.pi/8
...
```

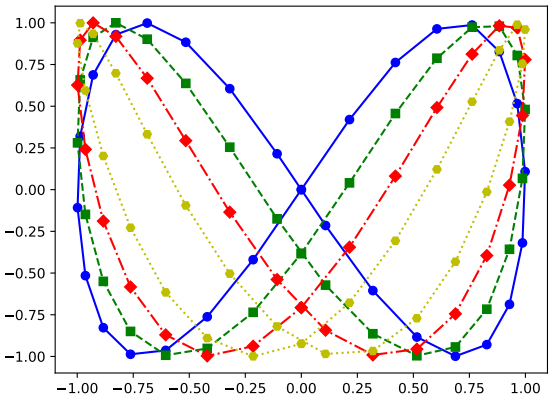


Figure P.5: Lissajous figures in different styles

Albeit being very handy, the latter syntax for formatting is of limited flexibility. Instead, there is a multitude of keyword arguments that can be passed to `plot` in order to design the plots appearance.

Keyword	Type	Explanation
`alpha`	float	describes the curves opacity in a range from 0 to 1, where 0 is invisible.
`color` or `c`	color	the line colour
`data`	tuple	This keyword can be used to pass a dictionary or a `pandas.dataframe`. Then only the keys need to be passed as `x` and `y`.
`figure`	figure	assigns the line to an existing figure handle.
`label`	string	assigns a label to the line that is displayed when a legend is displayed.
`linestyle` or `ls`	string	as described before
`linewidth` or `lw`	float	the with in pts
`marker`	string	as described before
`markeredgecolor` or `mec`	colour	the colour of the markers edge
`markerfacecolor` or `mfc`	color	the colour of the markers faces
`markersize` or `ms`	float	the markers' size
`markevery`	int	marks only every n^{th} point
`xdata`	array	the x-data (usually the first argument)
`ydata`	array	the y-data (usually the second argument)

Some of the properties in the list were colours. These can specified in multiple ways:

- As an RGB or RGBA tuple with values between 0 and 1 specifying red, green, blue (and the alpha value). Example: `(1, 0.25, 0)` for orange or `(0, 0.2, 1, 0.5)` for transparent blue.
- As a HEX RGB or RGBA string, e. g. `'00549F'` for the RWTH colour.
- A single string as described before, e. g. `'c'` for cyan.
- A X11/CSS4 colour name, e. g. `'cornflowerblue'`, `'lavender'`, `'tomato'`, `'red'`
- Matplotlibs default colours can be specified by `'tab:color'` where `color` is one of `'blue'`, `'orange'`, `'green'`, `'red'`, `'purple'`, `'brown'`, `'pink'`, `'gray'`, `'olive'` or `'cyan'`

Customize the Axis

As you know, a good plot always has labeled axises. To get those, we use the commands `plt.xlabel`, `plt.ylabel` and `plt.title` for a title. With `plt.text`, text can be placed at an arbitrary point in the plot. The axis labels can be specified by `plt.axis(lims)` where `lims` is a list (or an array) containing the axis limits as `[xmin, xmax, ymin, ymax]`. Below is an exemplary piece of code where a spectral analysis of a signal is performed, the result of which is depicted in Fig. P.6.

```
>>> from scipy.fft import fft, fftfreq
>>> # generate a signal of two sines and noise
>>> fs = 16000
>>> f1, f2 = 1000, 3000
>>> t = np.arange(0, 1, 1/fs)
```

```
>>> s = np.sin(2*np.pi*f1*t) + 0.5*np.sin(2*np.pi*f2*t) + 0.1*np.randn(len(t))
>>> # fourier transform
>>> S = fft(s)
>>> f = fftfreq(len(t), 1/fs)
>>> # visualize
>>> plt.plot(f, abs(S) / (len(t)/2))
>>> plt.xlabel('frequency')
>>> plt.ylabel('amplitude')
>>> plt.title('Spectral analysis')
>>> plt.axis([0, fs/2, 0, 1.1])
```

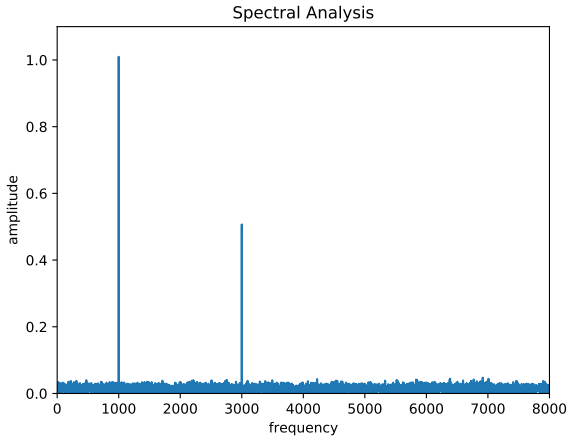


Figure P.6: A pyplot figure with adequate labels

In many cases from system theory, we need one or two axes with logarithmic scale. For convenience, we can use `plt.semilogx(x, y)`, `plt.semilogy(x, y)` for semilogarithmic plots and `plt.loglog(x, y)` for a plot with two logarithmic scales.

Scatter Plots

Apart from lines, there are many more ways to visualize data. An important one is the scatter plot, where the data points are simply drawn as points without connections. Especially in machine learning it is often use to investigate trends in data. In the following example, we use a scatter plot to investigate the data points drawn from a bivariate gaussian distribution. The colour is coded as the mahalanobis distance[8], which measures the distance to a distribution's expectated value, with respect to its covariance.

```
>>> from scipy.spatial.distance import mahalanobis
>>> # generate bivariate gaussian data
>>> mu = np.array([1, 2])
>>> sigma = [[0.9, 0.4], [0.4, 0.5]]
>>> rng = np.random.default_rng(42)
>>> X = rng.multivariate_normal(mu, sigma, 10000)
>>> # compute Mahalanobis distance
>>> beta = np.linalg.inv(sigma)
>>> mahal = np.array([mahalanobis(x, mu, beta) for x in X])
```

[8] https://en.wikipedia.org/wiki/Mahalanobis_distance

```
>>> plt.scatter(X[:,0], X[:,1], s=1, c=mahal, alpha=0.9)
>>> plt.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> # Choose cool colormap and add colorbar
>>> plt.magma()
>>> plt.colorbar()
```

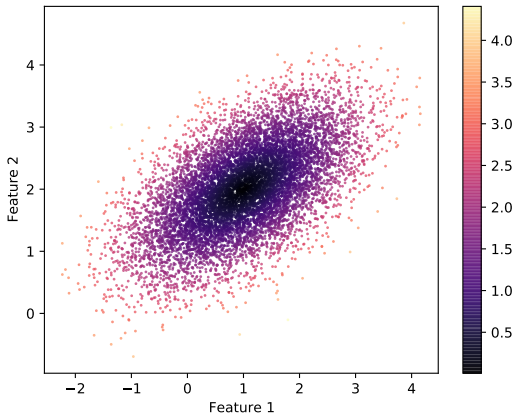


Figure P.7: A scatter plot of a bivariate gaussian with the Mahalanobis distance as colour coding

See that by assigning an array of distances as `c` argument, we could display a third variable. And there is more to it than that. We could even specify an array as size `s`, instead of a scalar. This would define the size for each data point (in pts) individually.
By calling `plt.magma()` we chose one of matplotlibs numerous colormaps[9]. These represent the mappings from numeric scalar values to colour values. While `plt.magma()` sets the colormap for all following plots, you can also specify a colormap only for the current plot by providing a `cmap` argument as a string, e. g. `cmap='viridis'`.

Stem and Step Plots

These kind of plots play an important role in digital signal processing as it is often used to visualize data that is time discrete or even frequency discrete, like a fourier series.

[9] `https://matplotlib.org/tutorials/colors/colormaps.html`

```
>>> from scipy.fft import rfft
>>> # Create symmetric rect sequence
>>> x = np.zeros(31)
>>> x[8:24] = 1
>>> # compute and plot real ft
>>> X = rfft(x)
>>> # Check if result is real
>>> assert(np.all(np.isreal(X)))
>>> plt.stem(X)
>>> plt.xlabel('k')
>>> plt.ylabel('X(k)')
```

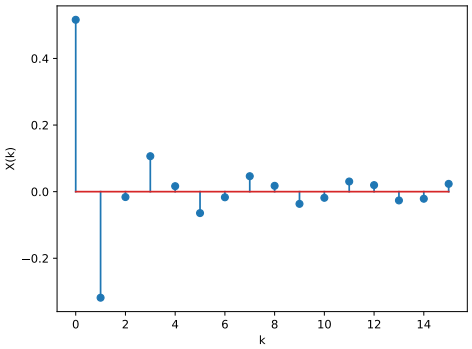


Figure P.8: Discrete Fourier transform of a symmetric sequence

In contrast to stem plots, step plots (sometimes refered to as stairs) require both `x` and `y` data as arguments and it is required that `x` is monotonically increasing.

Histograms

When you are dealing with statistical signal processing, histograms are a useful tool to assess the statistical property of a signal. One way to get a histogram is to use `np.hist` and plot the result. However, there is also a routine in `pyplot` that performs the numpy call for you. For no suprise, it takes the same arguments as `np.hist`. Be aware, that by default the number of bins is set to 10, which is hardly sufficient for signal processing. Below is an example of a histogram, and the result is depicted in Fig. P.9.

```
>>> # Generate random signal and quantization error
>>> X = np.random.laplace(0, 0.5, 100000)
>>> e = X - np.round(X)
>>> # Generate hist
>>> plt.hist(X, 100, histtype='step', density=True, range=(-4, 4),
↪  label='Signal')
>>> plt.hist(e, 100, histtype='step', density=True, range=(-4, 4),
↪  label='Quantization Error')
>>> plt.xlabel('x')
>>> plt.ylabel('p(x)')
>>> plt.legend()
>>> plt.xlim(-3, 3)
```

In this example, the `range` keyword was used to restrict the histogram to a range of interest. This is particularly useful when a signal has a possibly infinite range. Moreover, the `label` keyword was used to assign proper names to the legend.

P.9.2 The Object Oriented API

In the last section you learned how to generate plots quickly. This might be useful when we are exploring new data 5 minutes before lunch or when we quickly want to show something to a colleague. However, once we created the plots we have only limited control over them. The better practice is to create *objects* for figures and plots, to which we have access ever after. This makes things easier, especially when we deal with more than one plot.

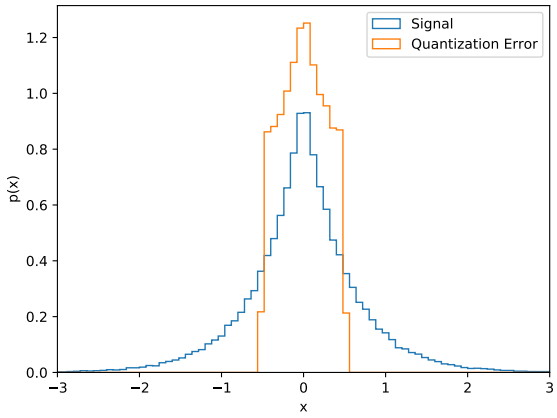


Figure P.9: Histogram of a random signal and its quantization error

Lines

When you tried the last examples yourself, you might have noticed that the code snippets where not entirely honest. Actually, whenever you type `plt.plot(...)` in the console, you receive something like `[<matplotlib.lines.Line2D at 0x14f19a66108>]` as an output. This is due to the fact, that `plt.plot(...)` returns a `Line2D` object. If you assign this object to a local variable, you keep access to the graphic handle and can modify or even delete it after creation time. To be precise, a list is returned that contains as many lines as you plotted. In the following example, the propagation of a wave package is plotted for 10 time steps. For clarity, earlier states should fade out whereas the most current line shall be drawn thicker. The result is shown in Fig. P.10.

```
>>> t = np.arange(10)[np.newaxis, :]  # 1 x 10 row vector
>>> x = np.linspace(0, 10, 200)[:, np.newaxis]  # 200 x 1 column vector
>>> x_0 = np.linspace(3, 7, t.shape[1])[np.newaxis, :]
>>> k, w = 6, 2
>>> y = np.exp(- (x - x_0)**2) * np.sin(k*x - w*t) # broadcasts to 200 x 10
>>> pls = plt.plot(x, y, color='tab:blue', lw=1)
>>> for it, pl in enumerate(pls):
...     pl.set(alpha=(it+1) / 10)
...     if it < 3:
...         pl.set(ls=':')
...     elif it < 6:
...         pl.set(ls='-.')
...     elif it < 9:
...         pl.set(ls='--')
...     else:
...         pl.set(ls='-')
...
>>> pls[-1].set(lw=2)
>>> plt.xlabel('Place')
>>> plt.ylabel('Elongation')
```

Here, `plot` returned a list of 10 `Line2D` objects that could be modified selectively. A line can be deleted by calling its `remove()` function. A line's current properties

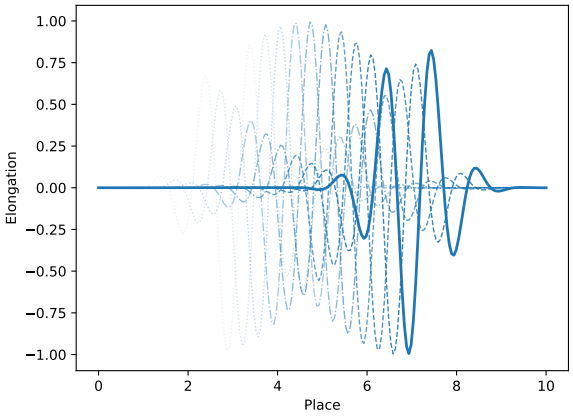


Figure P.10: The propagation of a wave package, where the solid line is the most current line

can be obtained using `get_ + <property>`, e. g. `get_ls()` returns the line style and `get_xdata()` returns the line's x data property. Similarly, `set_ + <property>` modifies a line's appearance. The aaccessible properties correspond to the keyword arguments that can be passed to `plot`. The x- and y-data can be accessed and modified with the keywords `xdata` and `ydata`.

In this example, we made use of broadcasting by exploiting that the addition of a 200×1 and a 1×10 array yields a 200×10 array.

Axes

The term `Axes` is often confused with `Axis`. Usually, by `Axes` we refer to the graphic's container, which has an x-axis and a y-axis. In addition, an `Axes` object contains a white area onto which our line, stem or scatter objects can be drawn. The `Axes` itself is always contained in a `Figure` which is a class that describes a window in your operating system.
The usual way to obtain both `Axes` and `Figure` is the `subplots()` command from the `pyplot` module. The name might be confusing, since without arguments it returns just one `Figure` and one `Axes` object. If the arguments `nrows` and `ncols` are provided, the figure and an `nrows` $\times$ `ncols` array of axis handles is returned. On your screen you will see a grid of `nrows` $\times$ `ncols` `Axes`. If you specify, `sharex=True` or `sharey=True`, all `Axes` in the Figure will share their x- and y-range.
The freshly obtained `Axes` handles can then be used to create plots explicitly. In fact, the functions from `matplotlib.pyplot`, like `plot`, `stem`, `semilogy`, etc. can be called as a method of an `Axes` object directly. The desired plot will be displayed in the calling `Axes`.
In the following example, we load the famous Iris Dataset[10]. The data set contains 150 observations of 4 features and each observation is assigned to one of three classes. Using subplots, we can create a scatter matrix that scatterplots the features against each other and shows histograms on its diagonal. The result is shown in Fig. P.11

[10] `https://en.wikipedia.org/wiki/Iris_flower_data_set`

```
>>> from sklearn.datasets import load_iris
>>> X, y = load_iris(return_X_y=True)
>>> fig, axs = plt.subplots(4, 4)
>>> fig.subplots_adjust(hspace=0.05, wspace=0.05)
>>> for it1 in range(4):
...     for it2 in range(4):
...         if it1 != it2:
...             axs[it1, it2].scatter(X[:, it1], X[:, it2], c=y, s=10)
...         else:
...             axs[it1, it2].hist(X[:, it1], bins=20, histtype='step')
...
...         if it1 != 3:
...             axs[it1, it2].xaxis.set(visible=False)
...         if it2 != 0:
...             axs[it1, it2].yaxis.set(visible=False)
...
```

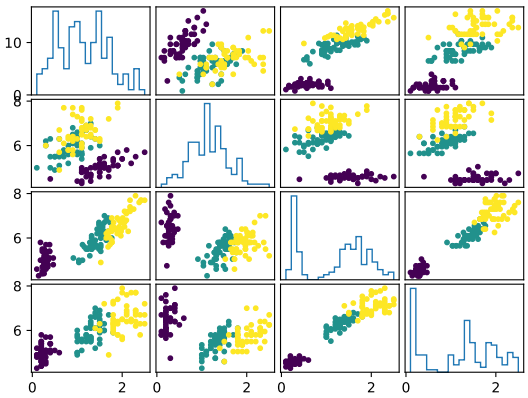


Figure P.11: The scatter matrix corresponding to the iris dataset

As you see, the `Axes` x- and y-axis can be accessed and modified by its `xaxis` and `yaxis` property. If you only need to specify x and y-label like in `plt.xlabel()`, you can use `ax.set_xlabel()` equivalently. The complete api of the `Axes` class can be found at the documentation[11].

P.9.3 Matplotlib in Jupyter Notebooks

In jupyter notebooks, `matplotlib` behaves slightly different than in a 'pure' python session. There exist several modes in which `matplotlib` can be used. The mode is chosen by the line `%matplotlib <option>`. This line needs to be placed *before* `matplotlib` is imported and changes only take place, when the kernel is restarted.

`%matplotlib inline` With this option, plots are displayed below the code cells but are not interactive. This means that you can not zoom or pan within the figure. Thus, you may choose this option if you do not want the view to be changed.

[11] `https://matplotlib.org/stable/api/axes_api.html`

`%matplotlib notebook` In most cases, this option should be preferred. The figures are again displayed below the code cells but are interactive. Apart from zooming and panning, the figure size can be modified with the mouse (lower right corner of the figure). With the on-off button in the upper right corner of a figure, you can turn the interactive mode on and off. This should be done before you send a notebook to someone else (e. g. to the lab supervisor).

`%maplotlib qt` With this option, figures open in a new window. This option is similar to the Qt5Agg backend in scripted python.

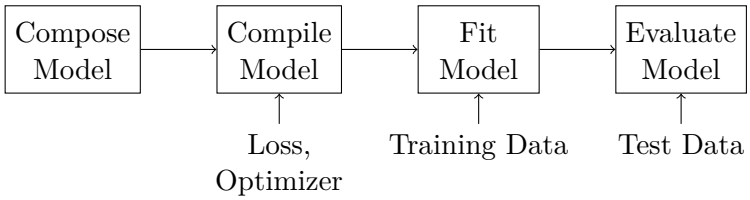
P.10 Keras and Tensorflow

In the last section we encountered `sklearn` which provides a vast landscape of estimators and transformers that can be trained to fit to data. `tensorflow` and `keras` follow the same purpose, only that the estimators here are (deep) neural networks. Keras and Tensorflow are related in that keras is a submodule of tensorflow. Since the release of tensorflow 2.3 in 2019, their relation is comparable to the one between `matplotlib` and `pyplot`. While tensorflow contains the functionality to build any thinkable deep learning model, keras contains the most frequent usage patterns (Dense Layers, Convolutive Layers, LSTM, activations...).

The concept of tensorflow is to express computations as computational graphs. Such a graph consists of computational nodes, that perform a specified operation on a specified number of inputs and pass the result to other nodes.

P.10.1 Train Models in Keras

A simplified workflow to train models in keras can be given as follows:



Lets go through the workflow step by step. First of all:

```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
```

Since keras has been fully integrated as a submodule into tensorflow, the use of

```
>>> import keras
>>> import keras.backend as K
```

is deprecated. You might find this syntax from time to time in older code. If you need any backend functionality, simply

```
>>> import tensorflow as tf
```

To be accurate, the data (or at least the problem) should be known before you tackle the first step. The well known MNIST dataset is a good way to get started. It consists of small images of handwritten digits. To each image, the written digit is known as a label. Thus, the dataset can be used to train a model for written digit recognition.
Note that this example has the purpose to demonstrate how to use keras rather than to explain neural networks. For more details on machine learning, please refer to the literature.[12] [13]

```
# load data
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

As the data is scaled from 0 to 255, it should be normalized prior to training.

```
mean_x = np.mean(x_train)
x_train = x_train.astype(np.float32) - mean_x
x_test = x_test.astype(np.float32) - mean_x

std_x = np.std(x_train)
x_train /= std_x
x_test /= std_x
```

Convolutive Layers expect the input shape to be `(instances, rows, cols, channels)` where `channels` indicates the number of colour channels in an image. However, the input data has only three dimensions and needs to be expanded.

```
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
input_shape = (x_train.shape[1:])  # (28, 28, 1)

num_classes = len(np.unique(y_train))  # 10

y_train_one_hot = keras.utils.to_categorical(y_train, num_classes)
```

[12] https://www.deeplearningbook.org/
[13] http://neuralnetworksanddeeplearning.com/

With the last command the training targets were transformed to the one-hot-representation.
Next, the actual model can be composed, for which exist several ways. Here, we create a list of layers which is passed to a `keras.Sequential` object. `Sequential` indicates that the layers are ordered one after another.

```
# build model
layers = [
    keras.Input(shape=input_shape),
    layers.Conv2D(32, kernel_size=(3, 3)),
    layers.MaxPool2D((2, 2)),
    layers.BatchNormalization(),
    layers.ELU(alpha=0.1),

    layers.Conv2D(32, kernel_size=(3, 3)),
    layers.MaxPool2D((2, 2)),
    layers.BatchNormalization(),
    layers.ELU(alpha=0.1),

    layers.Flatten(),
    layers.Dropout(0.25),
    layers.Dense(num_classes, activation="softmax")
]
model = keras.Sequential(layers)
```

Instead of a distinct layer, activations be passed to convolutional or dense layers by providing a keyword argument, e. g. `activation='relu'` for a rectified linear unit. An overview about supported layers can be found in the keras API[14].
Next, the model needs to be compiled. To do so, we need to specify both the loss and the desired optimizer. For multinomial classification problems the categorical crossentropy between the predicted posteriors and the true targets is usually preferred. Choosing an optimizer, Adam is a good choice for most problems.

```
loss = keras.losses.CategoricalCrossentropy()
optimizer = keras.optimizers.Adam(learning_rate=0.001)
model.compile(loss=loss, optimizer=optimizer, metrics=["accuracy"])
```

If loss and or optimizer are used with default arguments like in the previous lines, there exists the following shortcut:

```
model.compile(loss="categorical_crossentropy", optimizer="adam",
↪  metrics["accuracy"])
```

[14] https://keras.io/api/layers/

Possible choices for losses are (among others):

`BinaryCrossentropy`

`CategroicalCrossentropy`

`KLDivergence`

`MeanSquaredError`

`MeanAbsoluteError`

Possible choices for optimizers are (among others):

`SGD`

`Adam`

`Adagrad`

`Adadelta`

For most problems, `Adam` is the first choice.

The next step is to fit the compiled model.

```
model.fit(x_train, y_train_one_hot, batch_size=128, epochs=10,
↪  validation_split=0.15)
```

In your terminal you will see a progress bar for each epoch, next to the current loss and accuracy for training and validation split. Apparently, the model already achieves 97% accuracy after the first epoch. After roughly 5 minutes the training finishes with a validation loss of 99%. The next step is to investigate the models performance on the test data.

```
post_test = model.predict(x_test)
y_pred = np.argmax(post_test, axis=1)
acc = np.count_nonzero(y_pred == y_test) / len(y_test)  # 0.9916
```

P.10.2 Custom Layers

From time to time the predefined layers from keras are not enough and we have advanced needs. The following example shows how an *Radial Basis Function* (RBF) Layer can be implemented with a few lines. Instead of a linear operation, an RBF Layer outputs

$$a_{ij} = \mathrm{e}^{-\gamma \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|}$$

where $\boldsymbol{x}_i$ is the i^{th} input and $\boldsymbol{\mu}_j$ is the centroid of the j^{th} kernel. γ is the kernel bandwidth of all kernels.
The following code shows the implementation of such a layer where `units` is the number of kernels.

```
import tensorflow as tf  # here, backend functionality is needed
class RBFLayer(layers.Layer):
    def __init__(self, num_kernels, gamma_init):
        super(RBFLayer, self).__init__()
```

```
        self.num_kernels = num_kernels
        self.gamma = tf.Variable(gamma_init, trainable=False,
        ↪  dtype=tf.float32)

    def build(self, input_shape):
        self.mu = self.add_weight(name='mu',
                                  shape=(1, input_shape[1], self.num_kernels),
                                initializer=keras.initializers.random_uniform(0,
                                    ↪  1),
                                  trainable=True)

    def call(self, inputs, **kwargs):
        diff = tf.expand_dims(inputs, -1) - self.mu
        norm = tf.reduce_mean(tf.pow(diff, 2), axis=1)
        res = tf.math.exp(tf.negative(tf.multiply(self.gamma, norm)))
        return res

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.num_kernels)
```

The first thing to see is that custom layers inherit from `keras.layers.Layer`. From this base class we override 4 functions. The first is `__init__` which only calls the constructor of the base class and sets constant variables that do not depend on the input shape. Variables that depend on the input shape are defined in `build(self, input_shape)`. This function is called when the layer is confronted with an input the first time. This way you can make sure, that the dimension of the layers' weights always fits the input. The `call(self, inputs, **kwargs)` is always called when the layer processes an input and implements the actual functionality, which is the aforementioned equation in our case. Note that the inputs' first dimension is the number of examples in the presented batch and not a single example. The last function `compute_output_shape(self, input_shape)` states the shape of the data that is passed to the next layer.

In order to test the layer, we can replace the convolutional layers in the previous example. Since the RBF-Layer expects row vectors as examples, the Flatten Layer is moved to the beginning.

```
layers = [
    keras.Input(shape=input_shape),
    layers.Flatten(),

    RBFLayer(32, 0.5),
    layers.BatchNormalization(),
    layers.ELU(alpha=0.1),

    RBFLayer(32, 1),
    layers.BatchNormalization(),
    layers.ELU(alpha=0.1),

    layers.Dropout(0.25),
    layers.Dense(num_classes, activation="softmax")
```

```
]
model = keras.Sequential(layers)
```

When the model is trained and tested as before it still achieves 92 % accuracy on the test set.

Apart from custom layers, it is possible to implement custom accuracy metrics and loss functions.

P.11 Other Packages You Should Know

This section presents packages that every Python programmer should know, but whose API is too extensive for this introduction.

P.11.1 Scipy

The package scipy[15], which is short for Scientific Python, provides routines for scientific data processing. Below is an excerpt of its modules.

`cluster` routines for clustering and vector quantization such as the k-means algorithm.

`constants` mathematical and physical constans such as pi, the speed of light or the neutron-proton mass difference energy equivalent in MeV.

`fft` optimized routines to perform the fast fourier transform on real and complex signals as well as routines for the discrete sine and cosine transforms.

`integrate` routines for numerical integration of function statements.

`io` routines to read and write data in specialized formats such as `.mat` or `.wav` files.

`linalg` is similar to `np.linalg` but differs in the very detail.

`ndimage` a variety of routines for n-dimensional image processing.

`optimize` routines for the numerical minimization or maximization of objective functions, e. g. gradient descent, least squares, curve fitting, root finding and a lot more.

`signal` many routines for digital signal processing. Among others are convolution, filtering, filter design and tools for spectral analysis.

`sparse` sparse matrix computations.

`special` contains many implementations of various mathematical functions, i.a. bessel functions or spherical harmonics.

`stats` a variety of continuous and discrete probability distributions as well as statistical moments and tests.

[15] `https://docs.scipy.org/doc/scipy/reference/api.html`

Example 1: Fourier transform of a rectangular pulse

```
from scipy.fft import rfft, rfftfreq
import matplotlib.pyplot as plt
x = np.zeros((1024,))
x[:10] = 1
X = rfft(x)
f = rfftfreq(len(x))
fig, ax = plt.subplots(1, 2)
ax[0].plot(f, np.real(X))
ax[1].plot(f, np.abs(X))
```

Here, a rectangular pulse is generated and transformed. Real part and magnitude of the Fourier transform of this pulse are then displayed. Since the signal is real, `rfft` can be used, returning the right sided spectrum of length 513. `rfftfreq` returns an array with frequencies belonging to the frequency bins. The result is shown in Figure P.12.

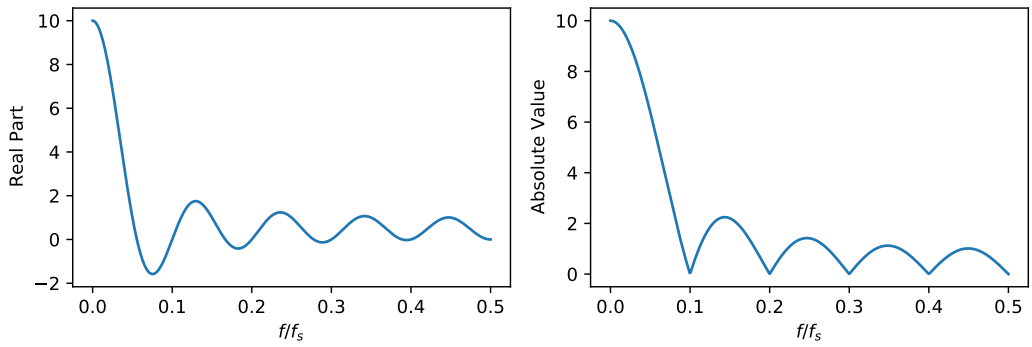


Figure P.12: Real part and magnitude of the Fourier transform of a rectangular pulse

Example 2: Fast Fourier Transform with Windowing

This example shows the influence of an analysis window applied prior to a frequency analysis.

```
from scipy.fft import rfft, rfftfreq
from scipy.signal.windows import hamming
import numpy as np
import matplotlib
matplotlib.use('Qt5Agg')
from matplotlib import pyplot as plt

fs = 16000
L = 320
t = np.arange(L) / fs

f1, f2 = 2000, 4000.5
x = np.sin(2*np.pi*t*f1) + np.cos(2*np.pi*t*f2)
f = rfftfreq(L, 1/fs)
```

```
X = rfft(x, norm='forward')
X_win = rfft(x*hamming(L), norm='forward')

plt.plot(f, 20*np.log10(abs(X)), label='No Window')
plt.plot(f, 20*np.log10(abs(X_win)), ls=':', label='Hamming')
plt.xlabel('f[Hz]')
plt.ylabel('X(f) [dB]')
plt.legend()
plt.show()
```

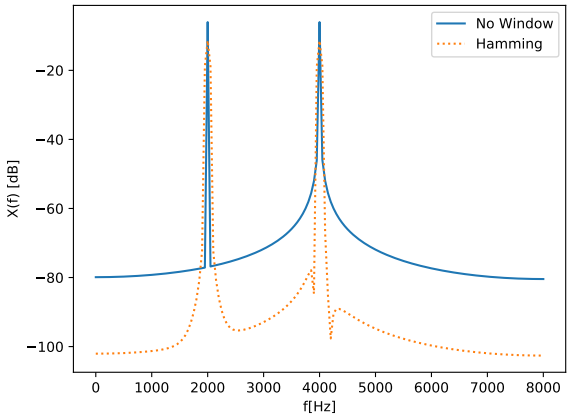


Figure P.13: Frequency analysis of two oszillations with and without analysis window

Example 3: Filtering

In this example, an electrocardiogram is processed, which contains strong baseline fluctuations. To remove them, an IIR highpass filter is designed.

```
import numpy as np
from scipy.signal import buttord, butter, lfilter
from scipy.misc import electrocardiogram
import matplotlib.pyplot as plt

# load data
ecg = electrocardiogram()
fs = 360
t = np.arange(len(ecg)) / fs

# plot raw data
plt.plot(t, ecg, lw=1, label='unfiltered')
plt.xlim(102, 107)
plt.ylim(-2, 3)
plt.xlabel('t [s]')
plt.ylabel('ecg')

# filter design
N, Wn = buttord(0.5, 0.05, gpass=0.1, gstop=60, fs=fs)
b, a = butter(N, Wn, fs=fs)
```

```
# apply filter
y = lfilter(b, a, ecg)
plt.plot(t, y, lw=1, label='filtered')
```

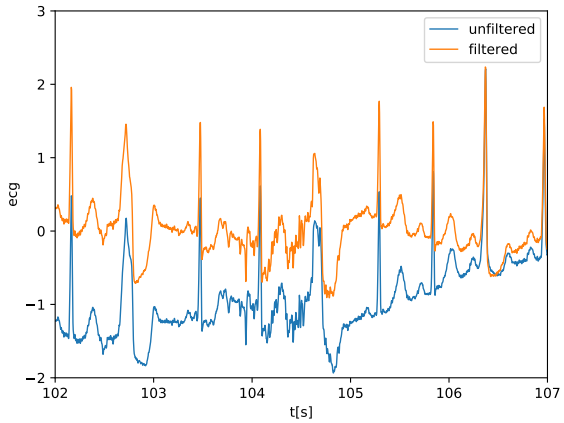


Figure P.14: An ECG signal before and after the application of a baseline filter

P.11.2 Librosa

As well as `scikit.signal`, `librosa`[16] also deals with signal processing, however, it is specialized to audio signals, such as speech and music. Next to general routines for audio signal processing, it provides common methods for feature extraction and signal analysis. Below is an excerpt of its modules (there are many more).

`core` provides routines for loading and resampling of audio files, autocorrelation, linear prediction, A- or µ-law compression, generation of sweeps, spectral transforms such as STFT, Griffin Lim phase recovery and many more.

`feature` provides routines for feature extraction. Most of them are spectral features, such as the mel spectrogram, MFCCs, spectral centroid, bandwidth, contrast, or flatness.

`utils` contains several utilities, e. g. to divide a signal into time frames or non-negative least squares.

```
import librosa
import librosa.display

# load example signal
path = librosa.ex('trumpet')
y, fs = librosa.load(path)

# compute mfccs and deltas
mfcc  = librosa.feature.mfcc(y, sr=fs, win_length=400, n_fft=1600,
↪  hop_length=100)
```

[16] https://librosa.org/doc/latest/

```
delta = librosa.feature.delta(mfcc, width=9, order=1)

# display
fig, ax = plt.subplots(2, 1, sharex=True, sharey=True)
im1 = librosa.display.specshow(mfcc, sr=fs, hop_length=100, x_axis='time',
↪  ax=ax[0])
im2 = librosa.display.specshow(delta, sr=fs, hop_length=100, x_axis='time',
↪  ax=ax[1])
ax[0].set(title='MFCC')
ax[0].label_outer()
ax[1].set(title=r'$\Delta$-MFCC')
fig.colorbar(im1, ax=[ax[0]])
fig.colorbar(im2, ax=[ax[1]])
```

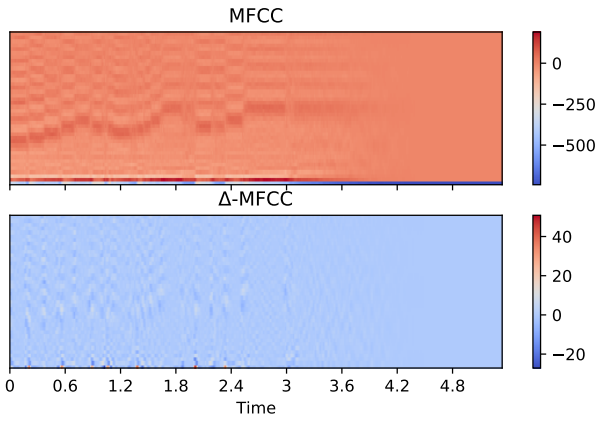


Figure P.15: MFCCs and their Δ-features

P.11.3 Scikit-Learn

Scikit-Learn[17], often referred to as `sklearn` provides all important algorithms for machine learning except for deep neural networks. It contains algorithms for supervised classification and regression tasks as well as for unsupervised clustering procedures. Prior to that, you find help during dimensionality reduction, and feature selection. Its flexible API allows you to build your own models and transformers easily, by deriving them from defined base classes. Below is a brief overview about its modules to give you an overview of sklearn's capabilities.

`base` contains base classes for estimators, transformers and many more that can easily be subclassed.

`cluster` provides classes and routines for clustering, such as kmeans or dbscan.

`datasets` loads famous datasets from the internet or lets you generate toy data to test your estimators.

`decomposition` contains routines for (kernel) principal component analysis, independent component analysis, nonnegative matrix factorization, dictionary learning and more.

[17] `https://scikit-learn.org/stable/`

`discriminant_analysis` provides linear and quadratic discriminant analysis.

`dummy` provides very minimalistic estimators, to test your pipeline.

`ensemble` contains ensemble methods, like random forests or bootstrap aggregating.

`feature_extraction` provides feature extraction methods for text and images. For audio input, `librosa` should be preferred.

`feature_selection` helps you to select features based on mutual information or other metrics.

`impute` this module helps, when your dataset contains corrupted values.

`linear_model` contains a large variety of linear models for classification and regression. Examples are logistic regression, the perceptron classifier, linear regression or orthogonal matching pursuit.

`manifold` implements techniques to deal with especially high dimensional data that are located on a lower dimensional manifold. These techniques include Isomap Embeddings, Locally Linear Embeddings and t-distributed Stochastic Neighbor Embedding.

`metrics` implements score functions and performance metrics for regressors and classifiers.

`mixture` contains Gaussian Mixture Models.

`model_selection` contains techniques that help you to find the best model or the best hyperparameter for your model. These include routines to divide data in train and test splits as well as grid search algorithms.

`neighbors` provides nearest neighbor algorithms, such as the k-nearest-neighbor algorithm or neighborhood component analysis.

`neural_network` provides methods for shallow neural networks. For more sophisticated models, `keras` or `tensorflow` should be preferred.

`pipeline` contains the `Pipeline` class which is a concatenation of many transformers and one estimator.

`preprocessing` contains normalizer, scaler, encoder and many more techniques to preprocess your data.

`svm` implements support vector machines.

`tree` implements decision trees.

Apparently, a lot of examples[18] are needed to cover at least a part of sklearn's functionality. In the following example, the basic principle is demonstrated. First, a dataset is loaded that contains 13 descriptors of houses and the corresponding areas. The target which we want to predict is the median value of occupied homes in multiples of $1000. Each column of `X` contains a feature (descriptor) and each row contains an observation. Accordingly, the same row in `y` contains the housing price.

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.model_selection import train_test_split
>>> X, y = load_boston(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

[18] https://scikit-learn.org/stable/auto_examples/index.html

A linear regression model can be fitted with a few lines:

```
>>> from sklearn.linear_model import LinearRegression
>>> linreg = LinearRegression()
>>> linreg.fit(X_train, y_train)
>>> y_pred_linear = linreg.predict(X_test)
```

A more sophisticated model can be obtained by arranging transformers and a k-nearest-neighbor estimator into a pipeline.

```
>>> from sklearn.preprocessing import RobustScaler
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.feature_selection import mutual_info_regression as mireg
>>> from sklearn.neighbors import KNeighborsRegressor
>>> from sklearn.pipeline import  Pipeline, make_pipeline

>>> steps = [RobustScaler(), SelectKBest(mireg, 6),
↪  KNeighborsRegressor(n_neighbors=3)]
>>> pipe = make_pipeline(*steps)
>>> pipe.fit(X_train, y_train)
>>> y_pred_knn = pipe.predict(X_test)
```

First, all input features are scaled equally, then only the 6 features with the highest mutual information to the price are kept. These 6 informative and scaled features are then fed to the estimator. The pipeline can be treated like a single estimator, albeit it contains three subsequent objects. Finally, the results can be compared:

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(y_test, y_pred_linear, s=10, label='linear model')
>>> plt.scatter(y_test, y_pred_knn, s=10, label='k nearest neighbors')
>>> plt.plot([0, 50], [0, 50], color='k', ls=':', lw=1)
>>> plt.xlabel('median value')
>>> plt.ylabel('predicted')
>>> plt.legend()
```

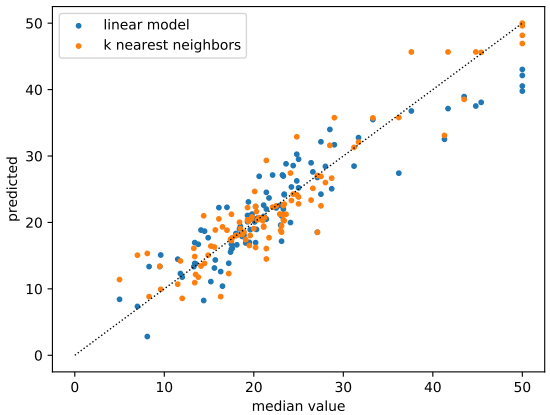


Figure P.16: The predictions on the test set for linear regression and a pipeline estimator

P.12 Exceptions and Assertions

Usually, when you write programs that are intended to be used by someone else, you have to be prepared for the worst. It can always happen that the user is eager to experiment and confronts your program with unseen challenges. Assume that you developed a real time audio framework with sophisticated algorithms that are able to de-noise .mp3 files, .wav files, or an input channel from your sound card. The user, however, is smart and wants to use the software to de-noise a picture. If you are not prepared for this case, your algorithm and maybe the program will crash at some point. For us to get ahead of the user, there are exceptions and assertions that can make our program more stable.

P.12.1 Exceptions

Expecting Exceptions

Let's try to load an audio stream, given that we have any `process` routine.

```
import audioread
def process_stream(file_name):
    with audioread.audio_open(filename) as f:
        for chunk in f:
            process(chunk)
```

If we try to process a picture, the following happens, since the audioread module has no backend to read .jpg files.

```
>>> path = 'C:\\Users\\iksuser\\Desktop\\DSC0049.jpg'
>>> process_stream(path)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File
  ↪  "C:\Users\iksuser\project\venv\lib\site-packages\audioread\__init__.py",
  ↪  line 116, in audio_open
    raise NoBackendError()
audioread.exceptions.NoBackendError
```

The error message says that an *exception* has been raised. While we can continue working in the interactive console, a program would crash at this point. To circumvent this situation, we can already expect the exception with a `try-except`-construction as follows:

```
def process_stream(filename):
                try:
                    f = audioread.audio_open(filename)
                except audioread.exceptions.NoBackendError:
                    print('No backend for file format .%s, please load audio
                    ↪  files only.' % filename.split('.')[-1])
                              return
```

```
    for chunk in f:
                    process(chunk)
```

Now the attempt will proceed as follows:

```
>>> process_stream(path)
No backend for file format .jpg, please load audio files only.
```

Let's look at what happened. First the interpreter tries to execute the commands in the try clause, between try and except. If no exceptions occur in this block, the execution continues after the end of the except-block. If the specified exception occurs, the interpreter jumps immediately to the `except` block and executes the commands that are listed there. The remarkable thing is that no error message is printed in the console and the program can continue.
It is possible to expect multiple exceptions at once, as follows:

```
try:
    f = audioread.audio_open(filename)
except (audioread.exceptions.NoBackendError, FileNotFoundError):
    print('Wrong file format or file does not exist)
    return
```

As you see, this attempt is improvable since all exceptions lead to the same error message. In the following solutions, each exception is handled individually:

```
    try:
    f = audioread.audio_open(filename)
    except audioread.exceptions.NoBackendError:
        print('No backend for file format .%s,' % filename.split('.')[-1],
        ↪  end='')
    except FileNotFoundError:
        print('File %s does not exist,' % filename)
    else:
        print('please try again!')
                    return
```

The `else` branch is executed if one of the except branches was executed. Hence, it should contain code that has to be executed after an error, independent of the error itself.

Raising Exceptions

At some points, you and your code are confronted with situations that you can not cover or solve locally. At this point, the best you can do is to raise an exception yourself. As a rule of thumb, exceptions should be raised when something occurs that violates the assumptions of your code. For instance the `audioread.audio_open` function assumes that you pass a proper audio file to it. Since an image or a file that

does not exist hurt this assumption, an exception is raised. In Python this is as easy as

```
>>> raise TypeError
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError
>>> raise TypeError('Never seen such an erroneous type')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: Never seen such an erroneous type
```

A list of all built-in exceptions can be found in the Python documentation[19]. For your programs however, it might be more appropriate to define custom exceptions that match the situation better.

Creating Custom Exceptions

If you want to create your own individual exception, all you need to do is to derive it from the `Exception` base class. For instance:

```
>>> class TooLateError(Exception):
...     """Exception raised when commands are executed after 6pm"""
...     pass
...
>>> raise TooLateError
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TooLateError
```

Usually, you want to equip the exception with at least an error message. On top of that, you can specify any parameters that make the error more meaningful.

```
>>> from datetime import datetime
>>> class TooLateError(Exception):
...     """Exception raised when commands are executed too late.
...
...     Attributes:
...        time -- the time when the command was inteded to be executed
...     """
...     def __init__(self):
...         self.time = datetime.now()
...
>>> try:
...     raise TooLateError
... except TooLateError as e:  # assigns the exception to e
...     print(e.time, 'is too late. Go Home.)
...
19:20:20.935872 is too late. Go Home.
```

If you are dealing with several exceptions that are related, exceptions can inherit from each other like any other classes if the base class inherits from `Exception`. If you intercept a base class (or even `Exception` itself), all child classes are caught as well.

[19] `https://docs.python.org/3/library/exceptions.html`

P.12.2 Assertions

In the last section about exceptions, we learned how we can make the program more robust against erroneous use. But to be honest, we do not always have a clean slate ourselves. As we do not have 20 years of coding experience yet, it may happen that we make little mistakes from time to time or we are just not sure whether the code behaves as we wanted it to. Therefore, it is good practice to check the assumptions either in the code itself or in a separate test. This is the grand entrance of the `assert` command in Python.

Software testing is a whole discipline on its own and in larger companies, there are separate positions for software-system-testers. For the inner modules (units) of a software architecture, the developers use *test first development* where the test routines are written before the code. In the best case, these tests run automatically and you can immediately check if all units work when you added or modified a feature.

Assertions in the Code

Assertions within your code usually serve a debugging purpose. Especially when you are not sure if the program will behave according your idea. In the following code, a column vector and a row vector are multiplied and we are 98% sure that the result is a scalar. In order to cope for the remaining 2% we make an assertion about the result's size. The syntax is `assert` *condition, message*, where the message is optional.

```
>>> import numpy as np
>>> x1 = np.random.rand((1, 20))
>>> x2 = np.random.rand((20, 1))
>>> res = np.matmul(x1, x2)
>>> assert res.shape == (1, 1), "Result is not a scalar"
>>>
```

Apparently nothing happened. If an assertion is fulfilled, the assert command behaves as if it wasn't there. Otherwise, an `AssertionError` with the given message is raised.

```
>>> assert np.matmul(x2, x1).shape == (1, 1), "Result is not a scalar"
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AssertionError: Result is not a scalar
```

Assertions in a Test

Below is a piece of code to test an `fftconv` routine that is yet to be implemented. The code checks, whether the result produced by `fftconv` has the right length, is real and if it is identical to the output produced by the reference `scipy.signal.convolve`. Usually, one would test many more scenarios but the principle should become clear in this example.

```
from scipy.signal import convolve
import numpy as np

def fftconv():
    pass

def test_fftconv():
    l1, l2 = 32, 1024
    x1 = np.random.random((l1,))
    x2 = np.random.random((l2,))

    res = fftconv(x1, x2)
    assert np.all(np.isreal(res)), "Result is imaginary"

    assert res.shape == (l1 + l2 - 1,), "Shape mismatch"

    ground_truth = convolve(x1, x2)
    assert np.mean(np.abs(res - ground_truth)) < 1E-12, "Accuracy above
    ↪  tolerance!"

    print("Tests completed sucessfully!")
```

After that, we can implement `fftconv`.

```
from scipy.fft import fft, ifft

def fftconv(x1, x2):
    l1, l2 = len(x1), len(x2)
    L = l1 + l2 - 1
    x1f = fft(x1, L)
    x2f = fft(x2, L)
    yf = x1f * x2f
    return np.real(ifft(yf))
```

```
>>> test_fftconv()
Tests completed sucessfully!
```

P.13 Debugger and Profiler

The following description is specific to PyCharm Professional and might not be applicable to other IDEs.

P.13.1 Debugger

The debugger comes into play when we are about to find or prevent bugs in our code. In the first place it helps us to pause the execution of code at certain lines. This can either be at a place where we suspect an error or when we want to make sure that the program behaves as we intended it to. These lines are denoted by *breakpoints*.

Such a breakpoint can be set by clicking in the gutter (the space between the line numbering and the acutal line). When the point ( ) is set and the program is started in debug mode ( ), the execution stops before the line. By right-clicking on the break point you can set a condition that has to hold for the breakpoint to stop the program. If the breakpoint is reached and the condition is not met, the execution continues without effect. This is for instance useful when you want to investigate what happens in the i^{th} iteration of a loop but do not want to pause in every iteration before.

When the execution has stopped, the debugger can be used to investigate local and global variables at this instant of execution. One example is given in Fig. P.17. In the right section (`Frames`), you see that the breakpoint is set in the function `build_model` which is called from a module named `solution_deep_learning`. Since the function is selected in the left section, the right section (`Variables`) shows the local variables that existed in the local scope before the breakpoint was triggered. By use of the symbols in the top line, you can walk through the code line by line ( ), step into functions that are called ( ) or step out of the current function ( ). In addition, you can also step to the next line of code that has been written by yourself, ignoring library code ( ) or continue until the current position of the mouse cursor ( ).

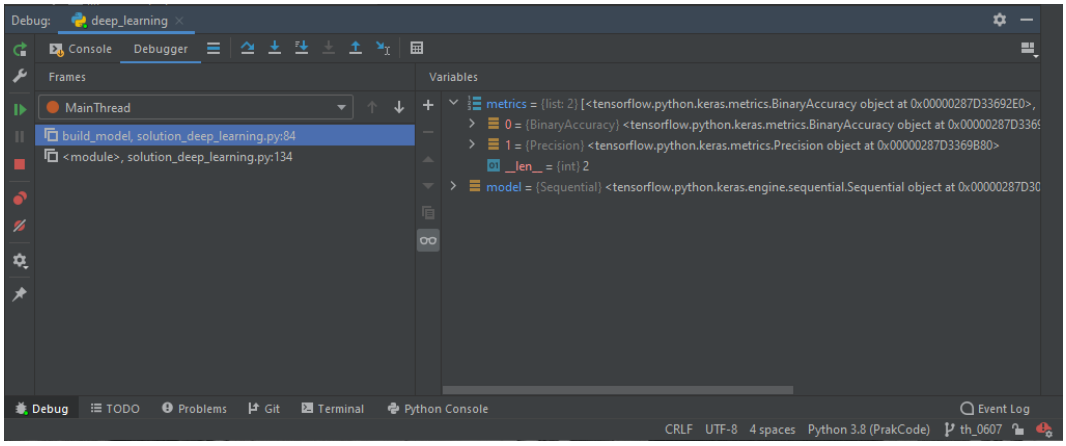


Figure P.17: The debugger view during a breakpoint

In interpreted languages like Python, commands can be executed during the execution pause. For instance, you may want to visualize the state of a matrix or listen to an intermediate signal. Furthermore it is possible to manipulate local or global variables during the execution pause. Both can be done by hitting `Console` in the upper left corner of Fig. P.17. When the execution continues, the program uses the modified values.

P.13.2 Profiler

The profiler ( ) helps us to identify time consuming passages (bottlenecks) in our code. This is useful when code is required to run in real time, when it is resource consuming or when it runs slower than expected. When the profiler is attached to an execution, it measures the processor time it takes for every code line during the execution of your program.

As soon as the execution is completed, the results appear in a well-arranged table. From this table, you can easily deduce at what positions your program does in fact

require a lot of processor time and where it is worth making optimizations to your code. The third column *Own Time* refers to the time that is spent within a function, excluding time that is spent in functions called from this function.

In addition you can visualize a call graph. The cells in the graph correspond to the functions in your program and the functions that are called under the hood. The colour corresponds to the run time of the functions. A connection from cell `A` to `B` indicates that function `B` has been called from `A`.

P.14 File Handling

This section describes the functions that are necessary to read and save files from your disk.

P.14.1 Sound Files

There are several ways to load sound files into your workspace. One of the easiest ways is using the `soundfile` package. This package provides a function `read(path)` that returns the actual samples from the file as well as the sample rate.
In addition, the `soundfile` package contains a function `write` to create sound files and save them on your disk. As the first argument it receives the file path, the second argument is the actual data as a numpy array and the third argument is the sampling rate. The next example demonstrates the basic usage of `read` and `write`.

```
>>> import soundfile as sf
>>> data, sr = sf.read('some\\audio\\file.wav')
>>> data_low = data*0.1
>>> sf.write('some\\audio\\file_low.wav', data_low, sr)
```

`sf.read` returns two output arguments, namely the actual data and the sampling rate. You need to assign these to two variables, like `data` and `sr` in the previous example. Otherwise, both variables are packed into one tuple.

In order to play back sound, you need the module `sounddevice`. Then, listening to audio is as easy as

```
>>> import sounddevice as sd
>>> sd.play(data_low, sr)
>>> status = sd.wait()  # Wait until playback has ended
```

P.14.2 Text Files

Often, we need to read text files when they contain metadata related to data sets. In other situations, writing data to text files provides a quick way to share content with other languages or even operating systems.

Basic usage with `open`

In Python, text files can be accessed with the keyword `open`. As a first argument it receives a file path that is either absolute or relative to the current working directory. If you are not sure, the current path can be accessed as follows:

```
>>> import os.path
>>> os.path.abspath('')
```

As a second argument, the function `open` receives a `char` that specifies the access mode:

`'r'` Open for reading. Raises an error if the file does not exist.

`'w'` Open for writing. Creates the file if it does not exist. Overrides all content if the file exists!

`'a'` Open for appending. Creates the file if it does not exist.

`'x'` Creates the file. Raises an error if the file exists.

If successful, `open` returns a `TextIOWrapper`, which you can see as a representation of the file in your program. It provides the following methods to read and write to the file:

`read([size])` returns the entire content of the file as a string. A line break is indicated by each `\n` in the string. If an optional argument `size` is provided, only `size` characters are returned.

`readline([size])` returns the content of the current line, e. g. until the next end-of-line indicator. If an optional argument `size` is provided, only `size` characters are returned.

`write(s)` writes the string `s` to the stream. `\n` indicates a new line. The content does not appear in the file until `flush` or `close` are called.

`flush()` flushes the stream content to the file.

`close()` calls `flush` and closes the connection to the file. After that, an attempt to write to the closed file will raise an error.

In the following example, three lines are written to a file and read out afterwards.

```
>>> f = open('testfile.txt', 'x')  # create file
>>> f.write('This is the first line \nfollowed by the second one')
>>> f.close()
>>> f = open('testfile.txt', 'r')  # open to read
>>> f.readline()
'This is the first line \n'
>>> f.readline()
'followed by the second one'
```

The Keyword `with`

A common syntax to read and write files is using the keyword `with`. The usage is illustrated in the following example:

```
>>> with open('testfile.txt', 'r') as fl:
...     fl.read()
...
'This is the first line \nfollowed by the second one'
>>> fl.closed
True
```

The difference to the previous usage is that by use of the `with` syntax, `fl.close()` was called automatically at the end of the indented block. The key advantage of this construct is that `fl.close()` is called automatically after the with-block, even if an exception occurs in the indented block.

P.14.3 Save Variables to Disk

As our variables become more complex, storing them as text files can be cumbersome. Imagine you have a neural net consisting of many weights and biases. In such cases, it is easier and more efficient to store the data in a binary way, like it already occurs in the memory. Generally, this can be done with the `pickle` module. For numpy arrays, special functions are provided by the `numpy` library.

Pickle

Pickle allows you to save and load any variables of (nearly) any type. Saving can be done with the commands `dump` and `open`.

```
>>> import pickle
>>> x = list(range(10))
>>> pickle.dump(x, open('any_file.p', 'wb'))
```

`pickle.dump` receives two arguments: First, the data you want to store, and second, an `IOStream`, that is created with `open`. In contrast to the last section, `open` is now used with an additional flag `'b'` to indicate that the data is to be written in binary mode and not as a text. You do not need to close the stream. This is done by the `dump` function.
In the next Python session you can call

```
>>> import pickle
>>> x_load = pickle.load(open('any_file.p', 'wb'))
>>> x_load
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Save and Load Numpy Arrays

Albeit numpy arrays can be saved and loaded with `pickle`, `numpy` offers its own routines `np.save` and `np.load` to do so. `np.save` receives two positional arguments. The first argument is either a relative file path, or a file object (e. g. created with

open). The second argument is the actual array to be stored. In the simplest way, an array could be stored to the current folder as follows:

```
>>> import numpy as np
>>> x = np.arange(5)
>>> np.save('my_range.npy', x)
```

Since PyCharm does not refresh the project directory every second, the new variable might not appear there instantly. In another interpreter session you can load the array as follows:

```
>>> import numpy as np
>>> x = np.load('my_range.npy')
>>> x
array([0, 1, 2, 3, 4])
```

When you decide to open a file object using a `with` block, it is possible to store multiple arrays in a single file as long as the file is open.

```
>>> import numpy as np
>>> with open('my_ranges.npy', 'wb') as f:
...     np.save(f, np.arange(5))
...     np.save(f, np.arange(5, 10))
...
```

The order in which the arrays are loaded in the next session is the same order with which the arrays have been stored (first in, first out).

```
>>> import numpy as np
>>> with open('my_ranges.npy', 'rb') as f:
...     x1 = np.load(f)
...     x2 = np.load(f)
...
>>> x1
array([0, 1, 2, 3, 4])
>>> x2
array([5, 6, 7, 8, 9])
```

One drawback of this method is that the programmer has to remember the order in which the data is stored. This can be circumvented by using `np.loadez`. This function stores the variables in a dictionary-like manner. The variables to be stored are passed to the function as keyword arguments. The keywords are then stored together with the actual variables and can be used to access and load the data. To indicate the difference to files that have been saved using `np.save`, files that are saved with `np.savez` should be in the `.npz` format.

```
>>> import numpy as np
>>> np.savez('my_ranges.npz', x1=np.arange(5), x2=np.arange(5, 10))
```

The file that has just been created can as well be loaded with `np.load`, which will return an `NpzFile` object that can be used like a dictionary.

```
>>> import numpy as np
>>> ff = np.load('my_ranges.npz)
>>> ff['x1']
array([0, 1, 2, 3, 4])
>>> ff['x2']
array([5, 6, 7, 8, 9])
```

P.14.4 Folder and Path Operations

The following routines are useful when you need to interact with files that are saved on your disk.

`os.path.abspath(path)` returns the absolute path, given that `path` is a relative path to the current directory. When `''` is passed as argument, the path of the current directory is returned.

`os.path.exists(path)` returns `True` if the directory or the file with the given path exists.

`os.path.isdir(path)` returns `True` if the given path points to a directory. `False` is returned if the path points to a file or if it does not exist.

`os.path.isfile(path)` returns `True` if the given path points to a file. `False` is returned if the path points to a folder or if it does not exist.

`os.path.join(path, *paths)` concatenates the components of a file path, depending on the operating system. For instance, this can be used to generate the absolute path of a file, if only the file name and the path to the containing folder are known.

`os.path.split(path)` is used to separate a file path into directory and file name. The directory (head) is returned as first output and the file name (tail) is returned as second output. When `path` ends with a backslash, the second output will be empty.

`os.listdir(path)` returns a `list` containing all file names in the directory located at `path`. This is particularly useful to iterate over all files in a folder.

`os.mkdir(path)` creates a directory with the given `path` if it does not exist. Raises an error if the directory exists. This function should be used together with `os.path.isdir(path)`. If the directory to be created is a subdirectory of a directory that does not exist, use `os.makedirs`.

`os.makedirs(path)` behaves like `os.mkdir` but can create subdirectories.

`os.rmdir(path)` removes the directory with the given `path` but raises an error if it is not empty or does not exist.

`shutil.rmtree(path)` removes the directory at `path` including the contained files and subfolders.

When you are about to type a string containing a file path, this is recognized by PyCharm and you can use the tabulator key for autocompletion.

In Python strings, `\n` is interpreted as a new line and `\t` is expanded to a tab. These expansions might make strings that contain a path invalid. This effect can be circumvented by using double backlashes in all strings containing file paths.

Exercises

Chapter 1

Introduction to Python I

The laboratory 'Audio Processing using Python' serves as an introduction to working with the programming language Python with applications from the field of digital signal processing. In the process, theoretical fundamentals of signal processing will be covered as well. There are various reasons why we have selected Python as programming language:

- Python makes it possible to realize highly complex algorithms in a structured and elegant way
- Python is very popular in industry because it does not require expensive licences.
- Learning Python is easy. Even after a short period of time, complex tasks can be approached and solved.
- Python can easily be expanded using so-called modules. Among others, there are very powerful modules for the development of machine learning algorithms.

1.1 Introductory Reading

In order to prepare for this week's lab cource, you are asked to familiarise yourself with the basics of Python. Please read the following sections from the primer in Chapter I. In some sections you will encounter lists of functions. You do not need to learn all functions and arguments by heart but you should know, which functions exist, what they do and where you can look them up.

- Section P.1 explains how to install Python and Pycharm on your computer. Please follow the tutorial carefully.
- Section P.2 helps you to make your first steps in python and to understand the basic syntax.
- For data types read Section P.3 up to and including P.3.4 about lists.
- For many commands in numpy and matplotlib you need to know the difference between positional and keyword arguments, which is explained in Section P.5.6.
- When working with arrays, Numpy is indispensable. Get started with Numpy by reading the Sections P.8.1 (numpy arrays), P.8.2 (functions in numpy), P.8.4 (indexing routines), P.8.5 (random numbers), and P.8.6 (statistics).
- In order to generate simple plots, please read Section P.9.

- In some tasks you need to load and listen to a sound file, which is described in Section P.14.1.

1.2 Exercises

Download all code prerequisites from moodle and set up a virtual environment with the given requirements. In the requirements for the first lab course, you find a sound file and a jupyter notebook file with partially empty cells.

Exercise 1.1 Properties of speech signals

1. Import the file `speech.wav` as a numpy array. What sampling rate does the signal have?

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

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

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

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

5. Generate random signals with the same length as the speech signal with each of the functions `np.random.rand`, `np.random.randn` and the unknown function `randg` that has been imported. Note that `randg` takes a few seconds.

   Generate histograms with the same settings as for the speech signal for all of these random signals. Use `plt.subplots` in order to display them in seperate figures. Modify the value range of the histogram so that it covers the entire signal by making use of the commands `np.min` and `np.max`. What is the distribution of the different noise signals? What distribution can the speech signal best be described by?

Exercise 1.2 Quantization

In this exercise, you will deal with a quantizer or, more precisely, a uniform quantizer. A quantizer maps an input value $x(n)$ to an output value $\hat{x}(n)$ where the set of possible output values is smaller than the set of possible input values. A uniform quantizer divides the input range up to some maximum value into equally sized sections and maps all signal values within one of these sections to exactly one corresponding output value. This mapping relation can be described by a characteristic curve. Figure 1.1 shows the characteristic of a uniform 3-bit quantizer which "rounds" the input values to the closest quantization level. For larger input values, the quantizer is saturated.

The quantization error is defined as

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

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

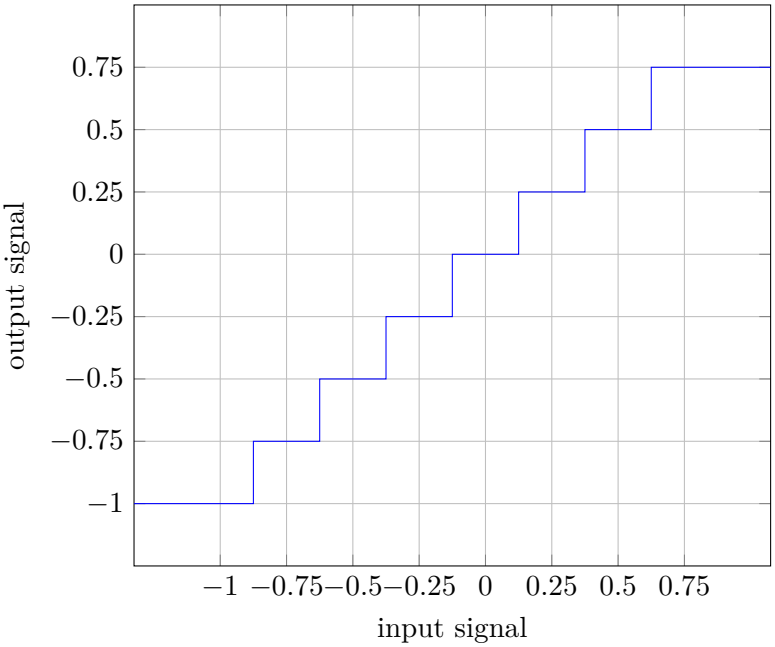


Figure 1.1: Characteristic of a uniform 3-bit quantizer

1. Normalize the input signal to the value range $[-1,1]$ and perform a uniform quantization of the speech signal from the previous exercise with 8, 6 and 4 bit precision, i. e. 256, 64 and 16 valid steps. The input values are rounded to the closest quantization level in the process. The value range of the quantizer is assumed to be limited to $[-1,1]$. The quantizer characteristic should be analogous to Figure 1.1.

   `Note:` The signal must be scaled first, then rounded to integers and then scaled back. To do so, you might find the function `np.round` helpful.

2. Listen to the quantized signals.

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

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

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

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

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

Exercise 1.3

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

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

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

Exercise 1.4 Amplitude modulation

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

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

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

The corresponding parameters are the amplitude factor a_0, the modulation index m and the initial phase of the cosine signal φ_0. It is assumed that the modulating signal $x(t)$ has a limited amplitude $|x(t)| < 1$. Furthermore, the bandwidth of the signal $x(t)$ must be limited to a maximum bandwidth of $2\omega_0$ (i. e. cut-off frequency $\omega_g = 2\pi f_g < \omega_0$)

In the frequency domain, an amplitude modulation simply implies a shift of the signal to the left and right to a carrier frequency $-\omega_0$ or ω_0 respectively. Here, we will employ the amplitude modulation with a carrier where the cosine with the carrier frequency is added additionally. Figure 1.2 illustrates this more a modulation index of $m = 1$.

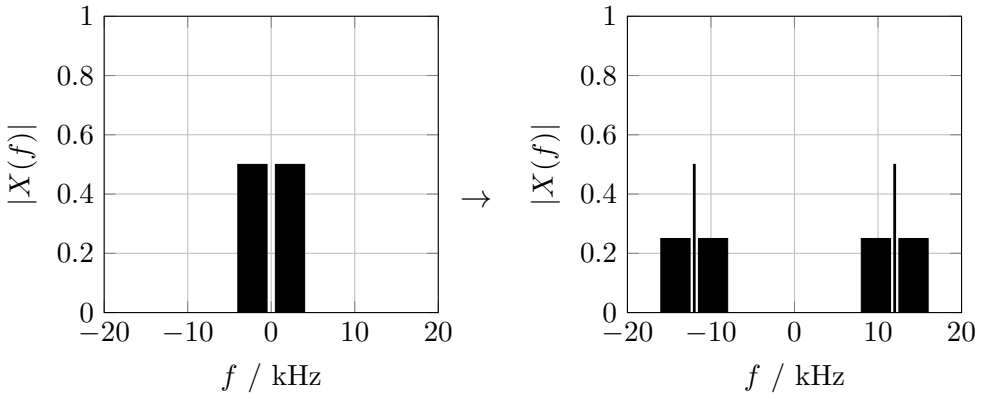


Figure 1.2: Amplitude modulation

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

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

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

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

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

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

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

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

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

2. Generate a cosine of the same length and sampling rate with the frequency $f = 12$ kHz and a phase of $\varphi_0 = 0$.

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

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

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

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

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

Chapter 2

Introduction to Python II

In the last chapter, we used a lot of commands in a procedural way. In this chapter, things are getting more exciting since we are exploiting control flows and functions.

2.1 Introductory reading

For this week's lab course, please read the following Sections:

- Expand your knowledge about data types and read the Sections P.3.5, P.3.6 and P.3.7.
- Read Section P.4 about control flows. Pay special attention to the section on list comprehensions, as this feature is unique in Python.
- Get started with functions by reading Section P.5 up to and including Subsection P.5.6 about keyword arguments.
- Your functions will be stored in a module, hence read Section P.7 about modules.
- We will dive deeper into numpy. To do so, read the remaining Subsections P.8.3 (array manipulation) and P.8.7 (linear algebra).
- Finally, read Section P.12 about exceptions and assertions.

2.2 Exercises

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

Channel coding with linear block codes

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

Addition:

+	0	1
0	0	1
1	1	0

Multiplication:

·	0	1
0	0	0
1	0	1

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

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

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

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

- Exchanging two rows
- linear combination of two rows

Gaussian normal form

A systematic generator matrix in Gaussian normal form satisfies the equation

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

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

On the receiving end, the so-called parity-check matrix is needed to test a received code word for validity. This allows to detect errors that occurred during the transmission

to some extent and, if applicable, even to correct them. From the Gaussian normal form, the parity-check matrix H is given as

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

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

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

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

Hamming distance

weight

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

For the correction of the up to e bit errors, their position within the received code word is required. This position can be determined with the help of the syndrome s. For up to e bit errors, the syndrome is unambiguous and corresponds to a specific error pattern with which the correction can be conducted. This correspondence is usually stored in a syndrome table. You get the syndrome table by computing the syndromes for all possible error patterns. One possible syndrome table for single errors is shown in table 2.1.

Error vector $\boldsymbol{e}_i$	Syndrome $\boldsymbol{e}_i \cdot \boldsymbol{H}^T$
0 0 0 0 0 0 0	0 0 0
0 0 0 0 0 0 1	0 0 1
0 0 0 0 0 1 0	0 1 0
0 0 0 0 1 0 0	1 0 0
0 0 0 1 0 0 0	1 0 1
0 0 1 0 0 0 0	0 1 1
0 1 0 0 0 0 0	1 1 0
1 0 0 0 0 0 0	1 1 1

Table 2.1: Syndrome table

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

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

a cyclic shift of the bits yields a code word

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

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

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

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

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

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

Exercise 2.1 Channel coding – Linear block codes

In this weeks prerequisites you will find a `ChannelCodingLib.py`. Please use this file to implement the functions. The functions that you will implement in this and the subsequent exercises should all comprise a docstring. Moreover, intercept possible invalid inputs and inform the user with appropriate error messages.

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

   In a test script, test your function with the generator polynomials

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

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

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

`Note:` If the modulo operation using `%` is applied to a numpy array, it is applied element-wise.

3. Extend the function `cyclgenmat` so that it also returns the parity-check matrix. Test the function with the generator polynomials given above.

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

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

   In a second step, extend the function so that it can not only process a single but also $w \in \mathbb{N}$ information words at a time. The information and code words should be represented in matrix form.

   `Note:` The individual positions of the information and code words should be arranged in ascending order in the numpy arrays from the LSB to the MSB. For example, an information word x is then `x[0]` $= x_0$, `x[1]` $= x_1$, etc.

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

   `Note:` In this function, you should first generate all possible code words and then determine their weights. The first can be elegantly done with help of the provided function `de2bi` that converts decimal numbers to binary vectors. The weights can be computed elegantly using `np.sum` with an `axis` argument. The minimum weight is equal to the minimum Hamming distance.

6. Write a function `syntab` that returns the syndrome table to a given parity-check matrix and a number of correctable errors e. The syndrome table should be a dictionary that has syndromes as key and the error patterns as value, such that `tab[syn] = pattern`. Which data type must the keys have and why?

   `Note:` Think about how to generate a matrix with all relevant error patterns first. For $e > 1$ this is *not* the unity matrix!

   `Note:` It is possible to obtain the syndrome table with a dict comprehension. You can iterate over the result of `de2bi` as it can return one row at a time.

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

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

8. Extend the function `decoder` so that errors in the code word can be corrected by use of the syndrome table and test your function.

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

Chapter 3

Signal Analysis

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

This laboratory experiment deals with the possibilities for signal analysis that Python has to offer. The given statistical signals are discrete in both time and value here. In the context of this experiment, operators and functions are introduced that are provided by the packages `numpy` and `scipy`.

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

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

Before you continue reading here, please read the (short) Section P.11.1 about scipy. Also read the Sections P.5.7 about argument lists and P.5.8 about the unpacking operation.

3.1 Statistical Properties

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

3.1.1 Generation of random signals

In digital signal processing, random signals play a central role. An important class of random sequences are noise sequences for the modeling of disturbances such as

quantization noise, channel noise or the like in digital systems. For the generation of random signals, you allready encountered `numpy.random`. To map a random signal to an arbitrary range of values, the signal can be shifted (i. e. the mean value can be changed) and/or scaled (i. e. the variance can be changed) accordingly:

```
>>> import numpy as np
>>> rng = np.random.default_rng(42)
>>> x = rng.random(8)
>>> print(x)
[0.77395605 0.43887844 0.85859792 0.69736803 0.09417735 0.97562235
 0.7611397  0.78606431]
>>> x2 = 2*(x - 0.5)
>>> print(x2)
[0.5479121  -0.12224312  0.71719584  0.39473606 -0.8116453   0.9512447
 0.5222794   0.57212861]
```

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

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

`np.abs(x)` Produces the absolute value of all elements of the array $\mathbf{x}$.

`np.min(x)` Returns the smallest values of all column vectors of the array $\mathbf{x}$.

`np.max(x)` Returns the largest values of all column vectors of the array $\mathbf{x}$.

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

```
>>> x2 /= np.max( np.abs(x2) )
>>> print(x2)
[ 0.5759949  -0.1285086   0.75395515  0.41496794 -0.85324554  1.
  0.54904842  0.60145261]
```

3.1.2 Mean Value, Standard Deviation and Variance

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

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

The variance σ^2 – also called the dispersion – is the so-called second-order central moment and it is a measure for how far the values x_n of a signal $\mathbf{x}$ deviate from their

mean value $\bar{x}$ on average. The standard deviation σ is then the square root of the variance. In the literature, two different definitions can be found:

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

or

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

These two definitions differ only in the normalization term. If you want to compute the standard deviation or the variance with `np.std` and `np.var`, you will have to pay attention to which of the definitions should be used for the calculation of the values. Numpy uses the definition from equation (3.2-b), unless you specify the keyword argument `ddof`. In the following, the functions for the computation of the named quantities are described:

With the help of `np.mean` and `np.std`, it is for example possible to check the behavior of the functions for the generation of random numbers:

```
>>> x = rng.standard_normal(1000000)
>>> np.mean(x)
0.00010406282530330993
>>> np.std(x, ddof=1)
1.0004846978498307
```

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

3.1.3 Histograms

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

The `h = np.histogram(x,bins)` function provided by Python allows the easy creation of histograms. For the signal **x** (vector), a histogram is returned to the vector **h**. The edges of the value ranges can be passed as the vector **bins**. This makes it possible to vary the size of the value ranges. Alternatively, it is also possible to specify the desired number of value ranges by passing a scalar as *bins*. The range between the maximum and the minimum of the signal is then divided into equidistant bins. The normalized pdf can be obtained by passing `density=True` to the function.

An example should illustrate how simple histograms can be created:

```
>>> x1 = rng.random(500000) - 0.5
>>> x2 = rng.standard_normal(1000000)
>>> edges = np.arange(-5, 5, 0.2);  # bin edges
>>> hx1, _ = np.histogram(x1, edges)
>>> hx2, _ = np.histogram(x2, edges)
>>> centers = 0.5*(edges[1:] + edges[:-1])  # bin centers
>>> import matplotlib.pyplot as plt
>>> plt.plot(centers, hx1)
>>> plt.plot(centers, hx2)
```

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

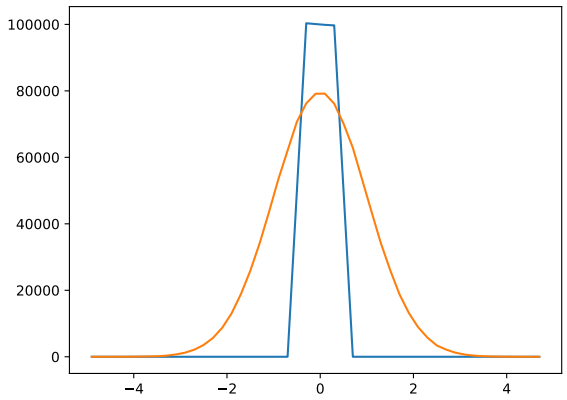


Figure 3.1: Histograms of the values of a uniformly and a normally distributed signal.

When only interested in the graphical output, it is sufficient to use `plt.hist(x,bins)` without return vector $\mathbf{h}$. The result is a bar chart or step plot of the histogram.

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

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

The procedure is illustrated in the following example:

```
>>> px1 = hx1/np.sum(hx1)  # same as histogram with density=True
>>> px2 = hx2/np.sum(hx2)
>>> cx1 = np.cumsum(px1)
>>> cx2 = np.cumsum(px2)
>>> plt.plot(centers, cx1)
>>> plt.plot(centers, cx2)
```

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

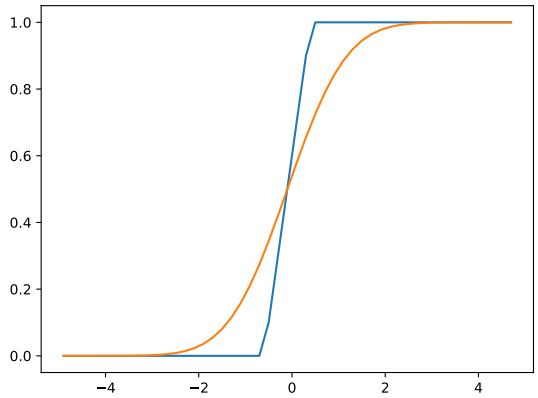


Figure 3.2: Approximated cumulative distribution function of a uniformly and a normally distributed signal.

3.1.4 Cross- and Autocorrelation

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

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

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

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

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

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

For the calculation of cross- or autocorrelation sequences, we use the `scipy.signal` module which provides the function `correlate(x, y)`. In order to get the autocorrelation sequence of a sequence $\mathbf{x}$, the same argument must be passed twice. For convenience, the function `correlation_lags` in the same module returns the corresponding time lags λ to the autocorrelation sequence, ranging from $-N+1$ to $N-1$. If the vectors $\mathbf{x}$ and $\mathbf{y}$ are different in length, the shorter of the two is filled up with zeros (*zero-padding*) so that it is ensured that both have the same length. The returned cross-correlation sequence will then also exhibit corresponding zeros in its first or last elements.

The following example should demonstrate the use of the aformentioned functions for the calculation of an autocorrelation sequence:

```
>>> from scipy.signal import correlate, correlation_lags
>>> fs = 1000
>>> t = np.arange(0, 1, 1/fs)
>>> f = 5
>>> x = np.sin(2*np.pi*f*t)
>>> phi_xx = correlate(x, x)
>>> lags = correlation_lags(len(x), len(x))
>>> plt.plot(lags, phi_xx)
```

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

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

3.2 Discrete Fourier Transform (DFT)

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

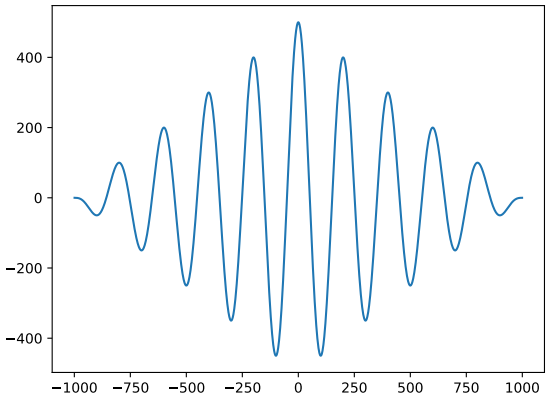


Figure 3.3: Autocorrelation sequence of a 5 Hz sine signal of length 1 second (sampling rate: 1 kHz).

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

3.2.1 Definition and Properties of the DFT

The discrete Fourier transform is the essential foundation of digital spectral analysis. The Fourier integral transformation is closely related to the z-transform.

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

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

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

IDFT denotes the inverse DFT.

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

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

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

This property in the time and frequency domain can be explained with the discretization (sampling) in both domains. For time-continuous signals, the DFT is generally

afflicted with errors due to the discretization. Only in the special case of a periodic, band-limited signal $x(k)$ with period length N, the Fourier integral can be exactly calculated with the DFT.

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

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

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

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

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

The spectral resolution is then

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

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

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

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

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

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

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

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

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

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

We can use the implementation of the fast fourier transform from the `scipy.fft` module. When this module was imported, the FFT of the vector **x** can be acquired by calling `fft(x)`. The length of the returned transform matched the length of the vector **x**. By additionally passing a scalar n (`fft(x, n)`), the number of returned spectral values can be specified. If the number of signal values of **x** is smaller than n, the signal is artificially lengthened by means of zero padding. The calculation of the IDFT with the help of the functions `ifft(x)` or `ifft(x,n)` from the same module works analogously.
Since the fourier spectrum of a real signal is conjugated symmetric it is convenient to compute only the right sided spectrum. This can be done, using `rfft(x[,n])`. When the time signal (after zero padding or truncation) has the length N where N

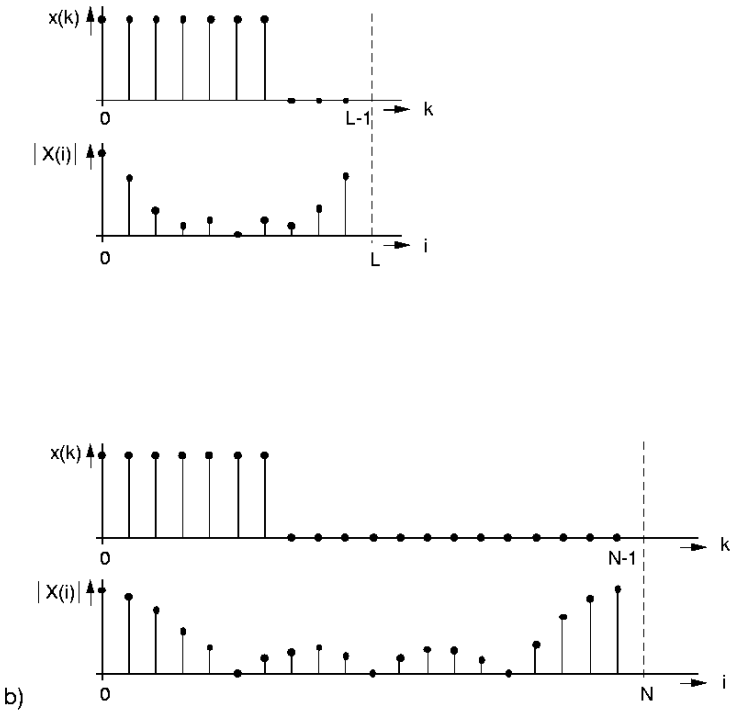


Figure 3.4: Interpolation of the spectrum by adding zeros a) $N = L = 10 \Rightarrow \Delta\Omega = \frac{\pi}{5}$ b) $N = L + 10 = 20 \Rightarrow \Delta\Omega = \frac{\pi}{10}$

is even, `rfft` returns only the $\frac{N}{2} + 1$ first frequency components. The first component corresponds to the DC-bin and the last component corresponds to the Nyquist-Bin. The real valued time signal to a right sided spectrum can be obtained by `irfft`.
When the result of `fft` or `rfft` is to be visualized, the functions `fftfreq` and `rfftfreq` are helpful. They return vectors of the same length like `fft` or `rfft` which obtain the frequency corresponding to each bin. As an argument they receive the length of the time signal and as an optional argument they receive the time between two samples (i. e. the inverse of the sampling frequency).

3.2.2 Window functions

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

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

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

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

  For the periodic repetition of the time window, discontinuities (signal jumps) may occur at the window edges depending on the shape of the signal which affect the result of the DFT more or less significantly. As a result of these discontinuities, the spectrum will exhibit frequency components that are not present in the signal to be analyzed (with respect to the long-time spectrum). This effect is called spectral leakage.

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

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

$w(k)$	:	window function
$W(i)$	:	spectrum of the window function
$x(k)$	:	analysis signal
$X(i)$	:	spectrum of the analysis signal

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

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

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

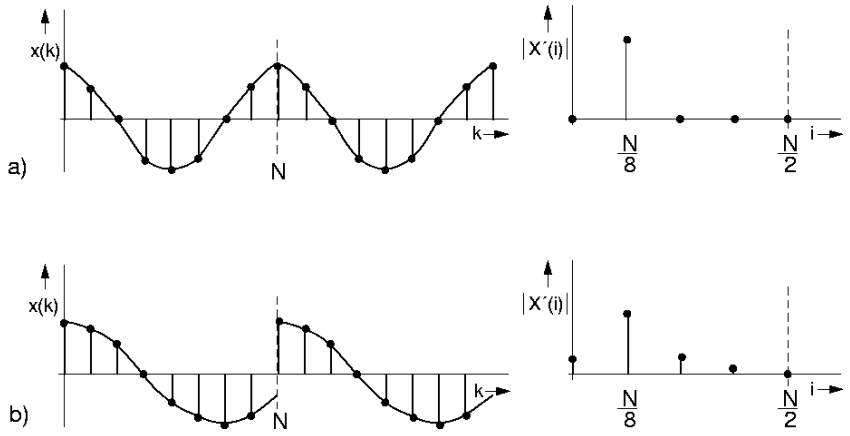


Figure 3.5: Explanation of spectral leakage in the time and frequency domain for a sine signal a) signal period $P = N$, b) signal period $N < P < 2N$

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

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

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

Sidelobe attenuation

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

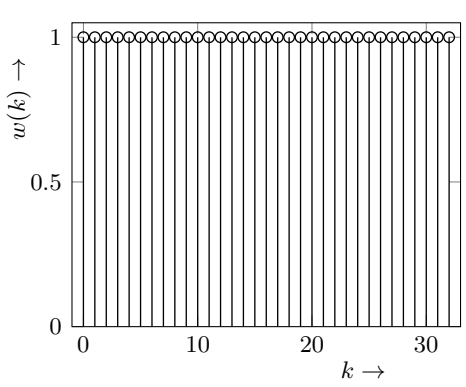
3 dB cutoff

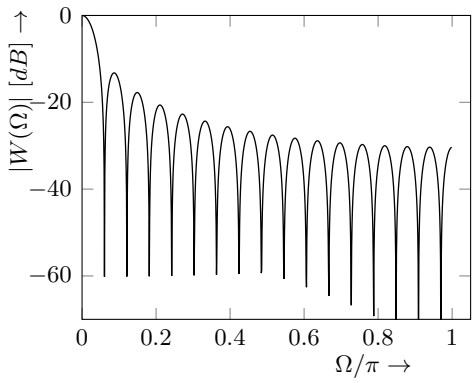
The 3 dB cutoff, also called the spectral width of a window function, describes the width of the main lobe in the spectrum of the window function. For the analysis of signals with several harmonic components, the 3 dB cutoff is a measure for the frequency selectivity, i. e. for the resolution of harmonic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

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

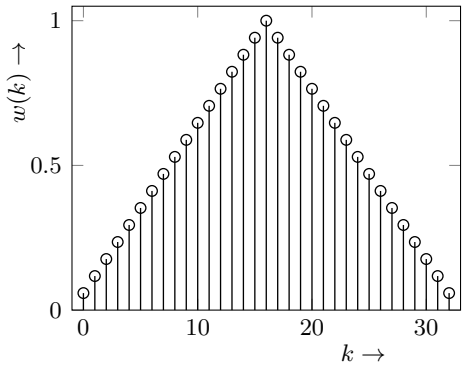
Between the frequency selectivity (3 dB cutoff) and the sidelobe attenuation, there is an uncertainty principle. Either the spectral function of the window has a narrow mainlobe or the sidelobes are strongly attenuated. As a result, it is not possible to design a window that is equally suitable for all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

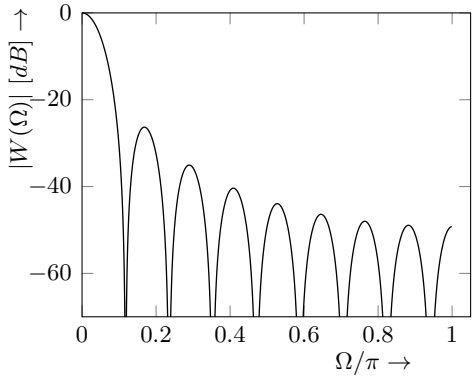
Rectangular window: $w(k) = \begin{cases} 1 & ; k = 0,1,\ldots,N-1 \\ 0 & ; \text{else} \end{cases}$



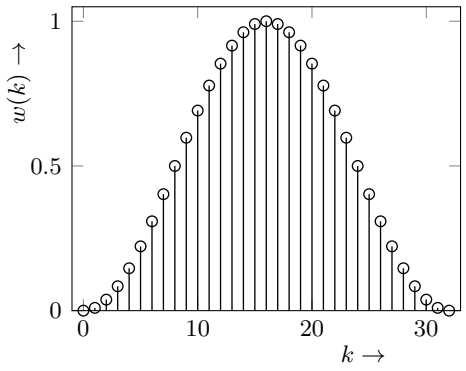


Triangular window: $w(k) = \begin{cases} 2k/(N-1) & ; k = 0,1,\ldots,(N-1)/2 \\ w(N-1-k) & ; k = (N-1)/2,\ldots,(N-1) \end{cases}$, N odd





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



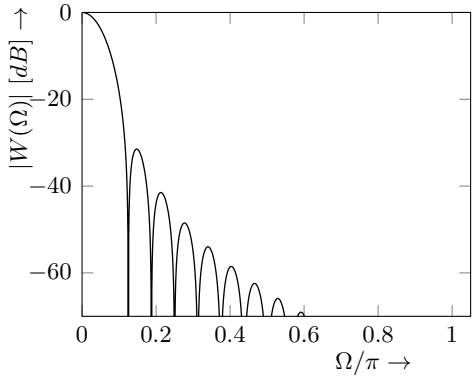
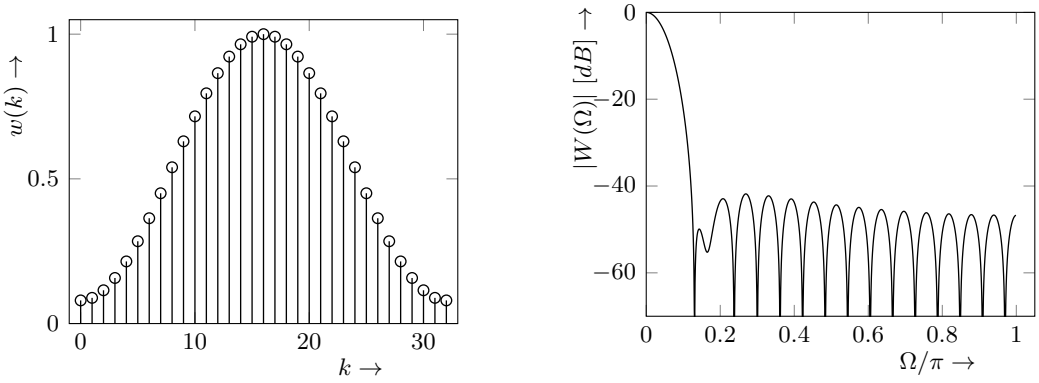
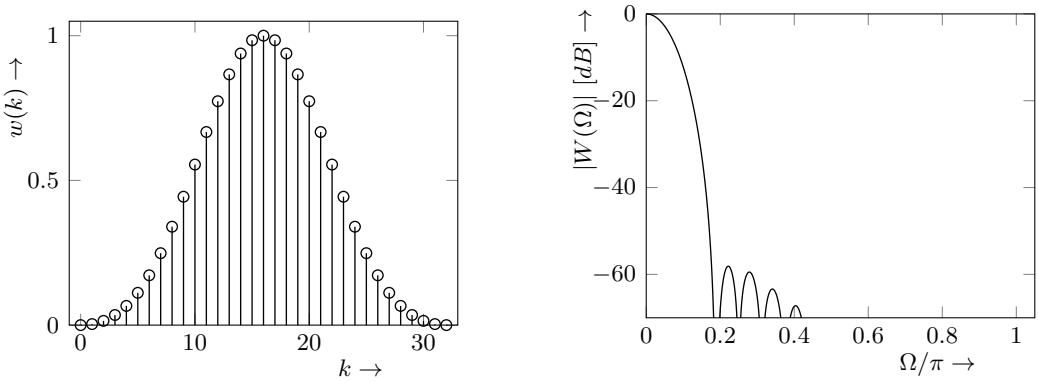


Figure 3.6: Comparison of several window functions

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



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



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

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

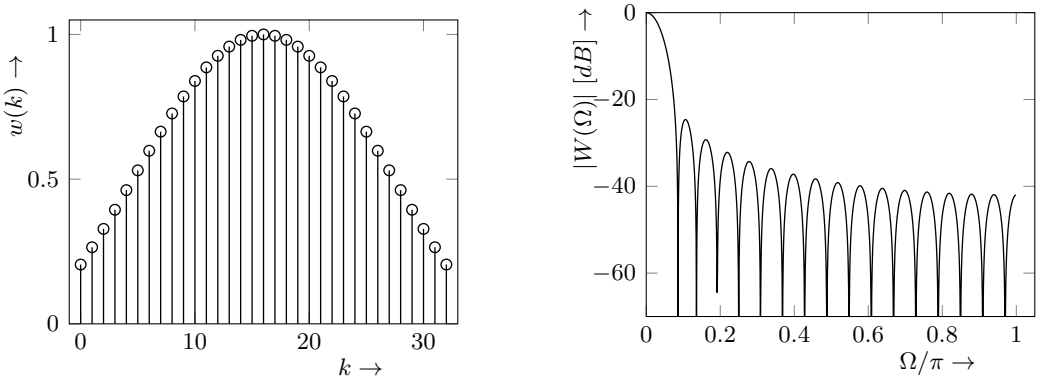


Figure 3.7: Comparison of several window functions (continued)

All of the window functions presented here are implemented within `scipy.signal`:

`boxcar(n)` creates a rectangular window of length n.

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

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

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

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

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

3.2.3 Spectrograms

A commonly used way to visualize the frequency domain transform of a signal is the so-called *short time fourier transform* (stft). For this, a time signal is divided in segments by applying a "sliding" window. Then a DFT is applied to each segment individually yielding a complex time-frequency-representation.

In python, several options exist to compute or visualize a spectrogram. Amongst others, these are `librosa.stft`, `scipy.signal.spectrogram` and `matplotlib.pyplot.specgram`. The inputs and outputs of these functions are similar but differ in detail, e. g. the padding length is specified differently in all approaches. To be sure, use the `help` routine in python. For instance, the function from the `scipy.signal` module receives the following input arguments. Except for the first argument `x`, it is recommended to specify arguments by keyword or by a keyword-dictionary.

`x` represents the signal to be analyzed.

`fs` represents the sampling rate (usually denoted by fs).

`window` is usually a string containing a window name, e. g. `boxcar`, `blackman`, `hamming`, `hann`, etc. If the window needs additional parameters, `window` must be a tuple of a the string and the parameter, e. g. `('kaiser', 0.5)`. Otherwise you can pass a window sequence with length `nperseg` directly.

`nperseg` is the number of samples per segments before padding. If `window` is a sequence, its length is used. If `window` is a string or tuple it defaults to 256.

`nfft` sets the fft length if zero-padding or truncation is desired. If unspecified, this value defaults to `nperseg`.

`noverlap` determines the overlap in samples between two consecutive frames. The difference between window length and overlap is called the frame shift (time difference between two frames).

`scaling` defines the normalization procedure. If set to `'density'`, a spectral density of the signal power is returned (default). If set to `'spectrum'`, the values in the output correspond to the actual signal powers per frequency bin.

The function returns the following outputs:

`f` is an array containing the frequencies corresponding to each frequency bin.

`t` is an array containing the time (in seconds) at the beginning of the signal segments.

`Sxx` contains the actual complex spectrogram. where each column is the one sided Fourier transform of one signal segment.

In order to display a matrix or image (`Sxx`) with known coordinate vectors (`f` and `t`) `matplotlib` provides the function `matplotlib.pyplot.pcolormesh`. It receives the following arguments:

`X` contains the x values to each column.

`Y` contains the y values to each row. `X` and `Y` do not necessarily have to be distributed uniformly, e. g. the frequencies can also increase in exponential steps.

`C` contains the actual data matrix.

`vmin`, `vmax` denote the lower and upper color limit. Every value that is lower (higher) than `vmin` (`vmax`) is mapped onto these values.

`shading` if set to `'nearest'`, all rectangles in the mesh have a constant color. If `'goraud'` the color in the rectangles is interpolated linearly.

Like all plotting routines in `matplotlib.pyplot`, `pcolormesh` can also be called from an `axes` object directly. The following example demonstrates how `spectrogram` and `pcolormesh` act together.

```
>>> from scipy.signal import spectrogram
>>> fs = 1000
>>> t = np.arange(0, 10, 1/fs)
>>> f1, f2 = 350, 400
>>> x = np.random.randn(len(t)) + np.sin(2*np.pi*f1*t) + np.cos(2*np.pi*f2*t)
>>> fbin, tframe, S = spectrogram(x, fs, window='blackman', nperseg=200,
↪  noverlap=100, nfft=2048, scaling='spectrum')
>>> plt.pcolormesh(tframe, fbin, 20*np.log10(S), vmin=-40)
>>> plt.colorbar()
>>> plt.xlabel('t [s]')
>>> plt.ylabel('f [Hz])
```

First, a sampling rate `fs` of 1 kHz is specified. With the help of the time vector **t** which represents a time duration of 10 s, a signal **x** consisting of two sinusoidal components and a noise component is generated. This signal is then windowed with a Blackman window with a length of 200 samples and transformed to the frequency domain with an FFT of length 2048. The used frame shift is 100 samples long.

The complex stft matrix is transformed to decibels. With the given normalization, a signal with amplitude 1 is converted to 0 dB (if no window was applied). The result of `pcolormesh` is depicted in Figure 3.8. The strong signal components at the frequencies `f1` and `f2` are clearly visible here. Furthermore, the white characteristic (equally strong signal contributions at all frequencies) of the Gaussian noise can be seen.

When functions that receive many arguments are called several times in a row, it is useful to use dictionaries that contain all arguments. In the following example, `spectrogram` is called two times with only one differing parameter. Instead of providing each argument separately, the unpacked argument dictionary is passed to the function. Similarly, a dictionary is passed to `pcolormesh` that contains the correct display settings.

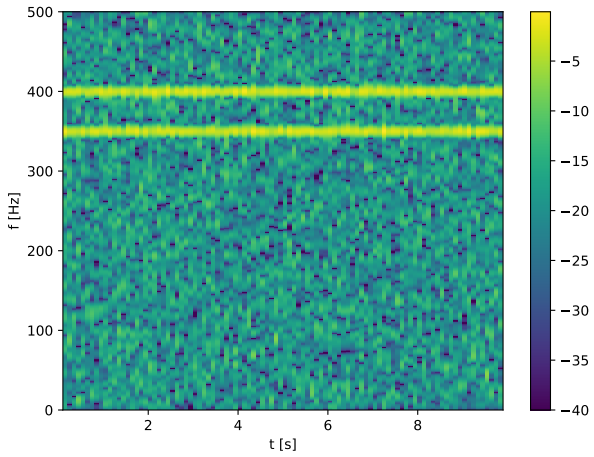


Figure 3.8: Spectrogram of an artificially generated noisy sinusoidal signal with two frequency components.

```
>>> from scipy.signal import chirp
>>> x = chirp(t, f0=100, t1=t[-1], f1=110) + 0.2*np.random.randn(len(t))
>>> spec_dict = dict(nperseg=512, noverlap=400, nfft=2048, window='rect',
↪   scaling='spectrum')
>>> plot_dict = dict(shading='goraud', vmin=-40)
>>> fig, ax = plt.subplots(1, 2)
>>> fbin, tframe, S1 = spectrogram(x, fs, **spec_dict)
>>> ax[0].pcolormesh(tframe, fbin, 20*np.log10(S1), **plot_dict)
>>> spec_dict['window'] = 'hamming'
>>> fbin, tframe, S2 = spectrogram(x, fs, **spec_dict)
>>> ax[1].pcolormesh(tframe, fbin, 20*np.log10(S2), **plot_dict)
```

First, a time signal is generated that contains a frequency sweep from 100 to 110 Hz and a noise component. Then, dictionaries are defined that contain the parameters for `spectrogram` and `pcolormesh`. When spectrogram is called , the unpacked `**spec_dict` expands to `nperseg=512, noverlap=400, ...` and so on. Similarly, the `plot_dict` ensures, that both spectrograms are displayed with the same settings. After the first spectrogram was generated, only the entry with the key `'window'` was exchanged. The result in Figure 3.9 clearly illustrates the effect of the hamming window on the signal analysis.

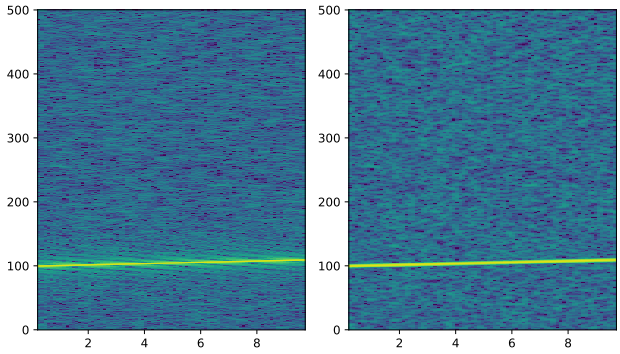


Figure 3.9: Spectrogram of a frequency sweep without (left) and with window function (right).

3.3 Exercises

In this weeks lab experiment you will again complete a jupyter notebook. An empty template can be found in this weeks folder. Since you will have to read values from plots, it is important that `matplotlib` is used with Qt backend. As a result, plots will open in external windows that allow you to zoom in.

Exercise 3.1

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

all sequences have a mean value of 2,

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

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

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

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

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

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

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

Exercise 3.2

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

Calculate the auto-correlation sequence of the signal and display it graphically!

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

Now load the file *sequences.npz* into the Workspace! This file contains four numpy arrays. `np.load('sequences.npz')` returns a file object that can be used like a dictionary. The sequences can be acessed with the keys `'x'`, `'y1'`, `'y2'` and `'y3'`. You will get the random sequences **x**, **y1**, **y2** and **y3**. Determine which of the three sequences **y1**, **y2** and **y3** is a shifted or scaled version of **x** and which sequence is uncorrelated!

How large is the shift and the scaling factor?

Exercise 3.3

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

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

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

`Note:` You can save time by implementing a function `show_fft(x)` that displays real part, imaginary part, absolute value and angle of the absolute value of signal `x` in one figure.

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

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

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

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

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

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

Exercise 3.4

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

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

Exercise 3.5

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

`Note:` A systematic and efficient way to investigate the parameters is as follows: Implement a function `compare_specs(x, fs, params_default, **kwargs)` that receives a time signal `x`, its sampling frequency `fs`, a parameter dictionary `params_default` and arbitrary many keyword arguments. In this function first calculate and plot the spectrogram with the unpacked `params_default` dictionary as arguments. Then replace the entries in `params_default` that are also specified in `**kwargs` and plot a new spectrogram next to the first. This way you can specify each parameter that you want to vary as a keyword argument.

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

Chapter 4

Adaptive Filtering

Digital filters in non-recursive and recursive form are the most commonly used tool for the processing of digital signals. Beside digital filters with fixed and time-invariant filter coefficients, so-called *adaptive* filters are also commonly used in digital signal processing. The filter coefficients of these filters are time-variant and are readjusted (adapted) during the runtime. Applications of such filters are for example noise reduction, signal coding and acoustic echo cancellation, the latter of which will be the focus of this laboratory experiment. The goal is to introduce this important class of filters and to convey how they can be realized and inspected with the help of Python. In the process, the knowledge about the techniques and functions presented in the previous experiments should be further improved.

4.1 Description of Linear Discrete-Time Filters

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

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

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

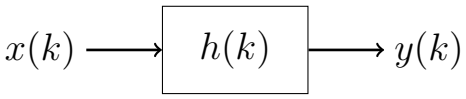


Figure 4.1: Block diagram of a digital filter with the impulse response $h(k)$.

An alternative description is given by the difference equation

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

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

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

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

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

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

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

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

It follows for the transfer function of an FIR filter:

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

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

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

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

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

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

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

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

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

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

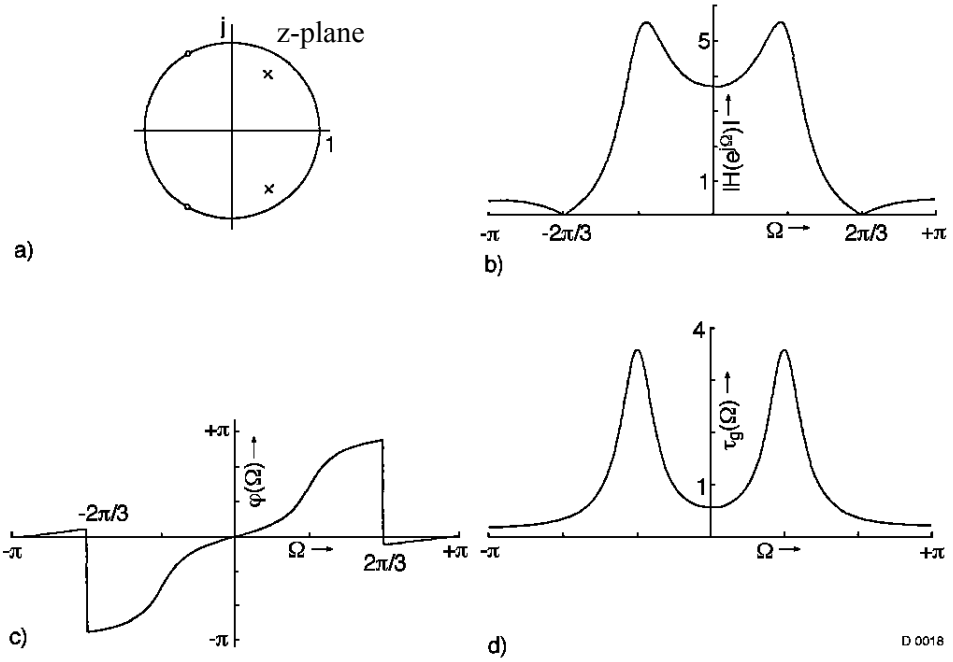


Figure 4.2: Example of a second order system (from: Schüssler 1988): a) pole-zero plot, b) magnitude frequency response, c) phase response and d) group delay

4.2 Digital Filters in scipy

All common commands that are needed to implement, apply and analyze digital filters are implemented in the `signal` module of the `scipy` package.
The function `convolve(in1, in2)` receives two arrays as input arguments and returns their convolution product. This can be used to apply simple FIR-Filters. By default, the output has the length `len(in1)+ len(in2)- 1`, meaning that the output includes transients at the beginning and at the end. If a keyword argument `mode='same'` is passed, transients are omitted and the output has the same length as `in1`. The `method` argument determines whether the convolution should be performed in time domain (`'direct'`) or in frequency domain (`'fft'`) by exploitation of the convolution theorem (Tab 3.1). By default (`method='auto'`, the optimal method is estimated. The function `fftconvolve(in1, in2)` always performs the convolution in frequency domain.
In the following example, a rectangular sequence of length 512 is convolved with a Gauss window of length 64 (with a standard deviation of 8). The result is depicted in Fig. 4.3.

```
>>> from scipy import signal
>>> import matplotlib.pyplot as plt
>>> x = np.zeros((512,))
>>> x[128:-128] = 1
>>> h = signal.window.gaussian(64, 8)
>>> h /= np.sum(h)
>>> y = signal.convolve(x, h, mode='same')
>>> plt.plot(x, label = 'x')
>>> plt.plot(y, label = 'y')
>>> plt.legend()
```

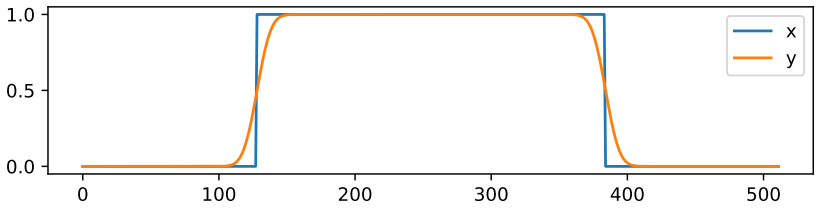


Figure 4.3: Convolution of a rectangular sequence with a gauss curve.

When we deal with recursive filters, the `convolve` command is not sufficient any more. Instead, the function `lfilter` is of interest. It receives 3 positional arguments `b, a, x` where `b` and `a` are the numerator and denominator of a digital filter as described by eq. (4.3) and eq. (4.6). Note that scipy's `a` corresponds to c in the aforementioned equations. The third argument `x` is the actual signal that is to be filtered. The following example shows how exponential smoothing can be performed with an IIR filter. Assume we want to smooth a noisy signal by application of the following difference equation:

$$y(k) = 0.25x(k) + 0.75y(k-1) \tag{4.11}$$

In a scipy implementation this would look as follows.

```
>>> x = np.sin(np.linspace(0, 6*np.pi, 512)) + 0.1*np.random.randn(512)
>>> b = [0.7]
>>> a = [1, -0.3]
>>> y = signal.lfilter(b, a, x)
>>> plt.plot(x, label='raw')
>>> plt.plot(y, label='filtered')
>>> plt.legend()
```

In the result in Fig. 4.4, it is observable that the filter's phase delay causes a slight delay between input and output. This delay can be compensated by using the function `filtfilt` instead. However, due to the delay compensation, this operation is non-causal.

Several options exist to analyze the behaviour of digital filters. For instance, we can use `lfilter` to compute the impulse response and step response of the filter. For convenience, `scipy.signal` offers the function `unit_impulse(shape[, idx])` that returns a sequence with shape `shape` with a 1 at index `idx`.

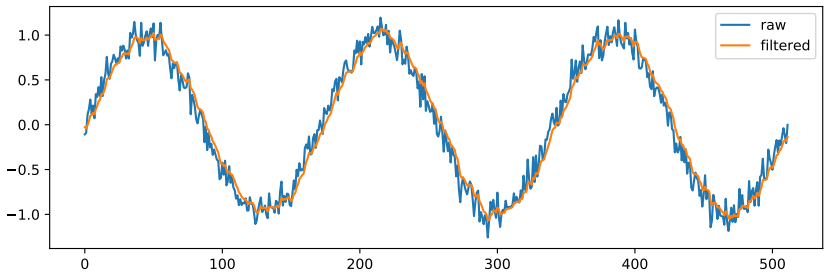


Figure 4.4: Application of a simple IIR-Lowpass filter

```
>>> d = signal.unit_impulse(64)
>>> h = signal.lfilter(b, a, d)
```

For an analysis in the frequency domain, the function `freqz` helps to compute and evaluate the filter's z-transform, given its coefficient vectors `b` and `a`. It returns a frequency vector Ω in a range from 0 to π as a first argument and the complex frequency response itself as a second argument. As an optional argument, `freqz` can also receive the number of frequency bins at which the output shall be evaluated. The following example shows how to generate a Bode plot using freqz. The result is depicted in Fig. 4.5.

```
>>> H, f = signal.freqz(b, a, 2048)
>>> logmag = 20*np.log10(np.abs(H))
>>> phase = np.angle(H)
>>> fig, ax = plt.subplots(2, 1, sharex=True)
>>> ax[0].semilogx(f, logmag)
>>> ax[1].semilogx(f, phase)
```

Additionally, eq. (4.9) could be implemented as follows:

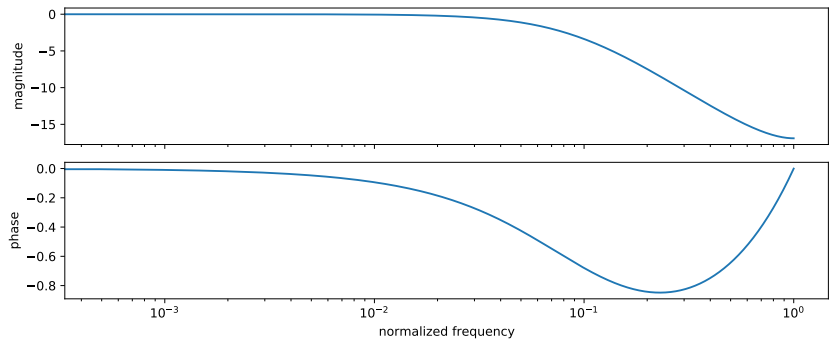


Figure 4.5: Magnitude and phase response of a recursive filter for smoothing

```
>>> delay = - np.diff(phase) / np.diff(f)
```

Finally, a more mathematical analysis of digital filters can be done by means of their poles and zeros. The function `tf2zpk` receives the filter coefficients `a` and `b` and returns a list of complex zeros, a list of complex poles and the scalar system gain.

4.3 Adaptive Filters for the Acoustics Echo Cancellation

Many devices for speech communication, like smart phones, are equipped so that they can be used hands-free nowadays. To increase the user comfort or for safety reasons (for example in a car), a loudspeaker-microphone setup is used instead of the telephone handset.

The acoustic coupling between loudspeaker and microphone can cause that beside the (desired) speech signal of the near speaker, the signal of the far speaker is transmitted as well. This undesired transmission leads to disruptive 'echos' which should be compensated by means of adaptive filtering[1].

The fundamental task will be explained with the help of Figure 4.6. The problem is

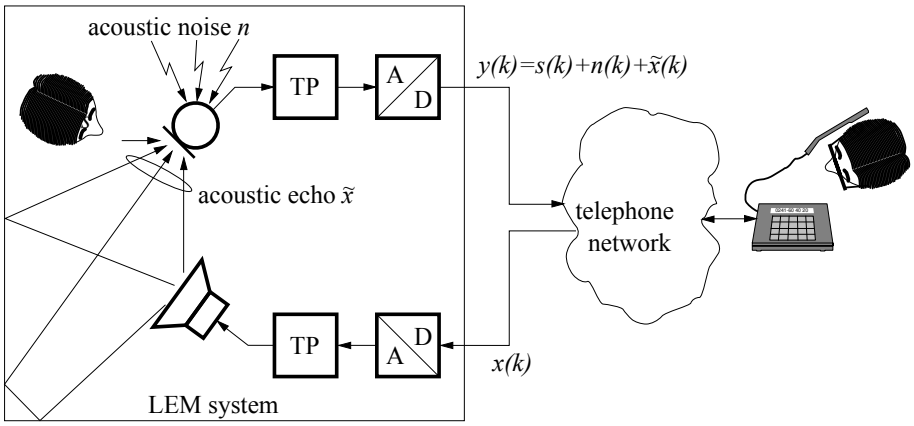


Figure 4.6: Loudspeaker-enclosure-microphone (LEM) system of hands-free equipment.

the acoustic coupling between the loudspeaker and the microphone. For simplification, it will be assumed in the following that the transmitted signal of the far speaker $\tilde{x}(k)$ and the signal $y(k)$ are given in digitized form. The microphone records not only the desired signal $s(k)$ of the near speaker but also the undesired background noise $n(k)$ and particularly the signal from the far speaker $\tilde{x}(k)$ which is modified by the acoustic transmission from the loudspeaker to the microphone. The signal component $\tilde{x}$ is created by numerous acoustic reflections. It is generally referred to as the acoustic echo to distinguish it from the electric line echo of the telephone network. It therefore holds for the digitized transmitted signal that

$$y(k) = s(k) + n(k) + \tilde{x}(k). \tag{4.12}$$

The aim of the acoustic echo cancellation is to prevent the echo signal $\tilde{x}(k)$ from being transmitted to the far speaker. This should ensure the stability of the electroacoustic loop if the far speaker uses a hands-free capability as well. It should further be prevented that the far speaker perceives his own speech signal with a delay caused by the telephone or mobile network. Such a feedback would have an impact on the conversation between the speakers.

[1]The following presentation of these techniques is based on Cha. 13, Vary P., Martin R. (2006) *Digital Speech Transmission: Enhancement, Coding and Error Concealment*, Wiley, where references to further literature can also be found. The book is accessible online for RWTH students.

4.4 Possible Solutions and Evaluation Criteria

In many (simple) telephone devices and cell phones, a so-called *voice-controlled echo suppressor* can be found for echo cancellation, as depicted in Figure 4.7. With one

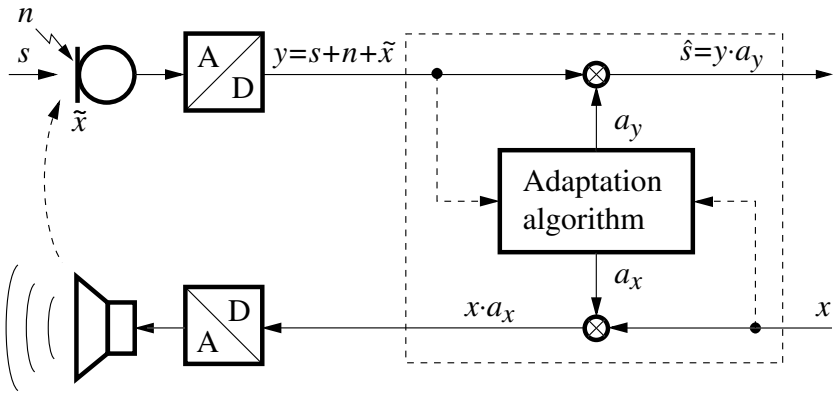


Figure 4.7: Loudspeaker telephone with voice-controlled echo suppressor.

variable damping element in both the transmitting and receiving branch, the branches are attenuated differently depending on the speech activity of the two speakers, with the total attenuation of the loop never being below some minimum value, e. g. 40 dB. As the damping factors should satisfy the condition

$$-(20\log a_x + 20\log a_y) = 40\text{dB} \tag{4.13}$$

it is possible only to a limited extent for both the near and the far speaker to speak simultaneously. This situation is referred to as double talk. Techniques that generally allow double talk will be considered in the following.

The basic principle of these methods is illustrated in Figure 4.8. The loudspeaker-

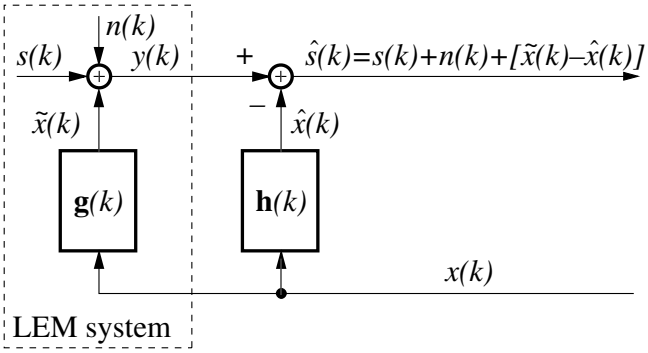


Figure 4.8: Discrete-time model of a hands-free setup with echo cancellation.

enclosure-microphone (LEM) system depicted in Figure 4.6 is shown as a discrete-time model. The transmission of the far speaker $x(k)$ over the LEM system should be described by means of a causal, linear filter. The impulse response of this time-variant filter is assumed to be limited to m' coefficients and is represented by the coefficient vector

$$\mathbf{g}(k) = [g_0(k), g_1(k), \ldots, g_{m'-1}(k)]^T \tag{4.14}$$

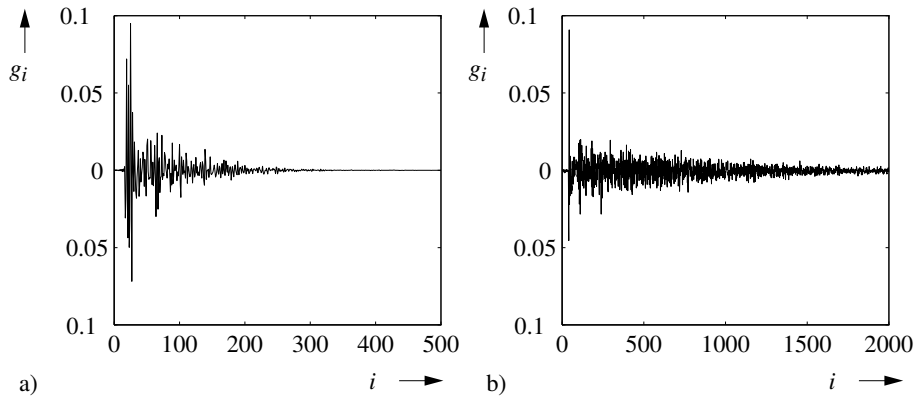


Figure 4.9: Impulse response of an LEM system measured in a car (left) and in an office room (right).

that is time-variant in k. Examples for measured impulse responses of LEM systems are shown in Figure 4.9.

To compensate the undesired echo signal $\tilde{x}(k)$, a time-variant transversal filter of length m with the coefficient vector

$$\mathbf{h}(k) = [h_0(k), h_1(k), \ldots, h_{m-1}(k)]^T \tag{4.15}$$

is employed. In the following, this transversal filter will alternatively be called the *compensation filter* as its task is to 'compensate' the undesired echo signal $\tilde{x}$ in the transmitting branch. If the coefficients of the impulse responses $\mathbf{g}(k)$ and $\mathbf{h}(k)$ match exactly, the echo signal is eliminated (compensated) completely, i. e. the difference vector

$$\mathbf{d}(k) = \mathbf{g}(k) - \mathbf{h}(k) \tag{4.16}$$

is a zero vector. The aim of the (acoustic) echo cancellation is therefore to replicate the unknown and time-variant coefficients of the impulse response of the LEM system $\mathbf{g}(k)$ as well as possible with $\mathbf{h}(k)$.

A measure for the quality of the echo cancellation is the *relative system distance*

$$D(k) = \frac{||\mathbf{d}(k)||^2}{||\mathbf{g}(k)||^2} \tag{4.17}$$

where the system distance is given by $||\mathbf{d}||^2$ and $||\mathbf{d}||^2 = \mathbf{d}^T\mathbf{d}$ is the squared (Euclidean) vector norm. Typically, the logarithmic distance $10 \log D(k)$ is expressed in dB. When taking the difference of the vectors according to eq. (4.16), the shorter vector is filled up with zeros if the two lengths of $\mathbf{g}(k)$ and $\mathbf{h}(k)$ are not the same. In real systems, the impulse response $\mathbf{g}(k)$ is usually not known which is why the relative system distance according to eq. (4.17) can only be evaluated in simulations.

A measure that correlates with the subjective auditory impression is the achievable reduction of the power of the echo signal $\tilde{x}(k)$. The corresponding measure is referred to as the *echo return loss enhancement (ERLE)* in the literature. It is defined as

$$\frac{ERLE}{\text{dB}}(k) = 10 \log_{10} \frac{\mathrm{E}\{\tilde{x}^2(k)\}}{\mathrm{E}\{(\tilde{x}(k) - \hat{x}(k))^2\}}. \tag{4.18}$$

The required difference signal

$$e(k) = \tilde{x}(k) - \hat{x}(k) \tag{4.19}$$

is also called the *residual echo*. Unlike the system distance according to eq. (4.17), the ERLE measure depends on the signals $x(k)$ or equivalently $\tilde{x}(k)$.[2] To compute the ERLE, the expected value in terms of an ensemble average over many experiments is replaced by the (time-dependent) short-term expected value, i. e. the expected value from the last values. Due to the limited window length of the short-term expected value, this value suffers from an estimation error.

A low system distance $D(k)$ implies a high echo attenuation $ERLE(k)$. The converse is not true, as for example in case of a narrow-band signal $x(k)$ a high echo cancellation (ERLE) can be achieved if the frequency response of the compensation filter "matches" the frequency response of the LEM system only in the relevant bandpass interval (which is possible even in case of a large system distance).

In real systems, the residual echo from eq. (4.19) is available only for $s(k) = 0$, i. e. in speech pauses of the near speaker (single talk) and in the noiseless case ($n(k) = 0$, see Figure 4.8). Still, in the (Python) simulation, it is possible to compute $e(k)$ first and then superimpose the components $s(k)$ and $n(k)$ accordingly. This allows for an inspection of the time-dependent echo attenuation in case of double talk and additive interference (background noise) as well.

4.5 Adaptation Algorithm

The principle of the *Least-Mean Square* (LMS) algorithm is to estimate an echo signal by convolving the input signal with the estimated filter response. By use of an inner product, the convolution operation can be expressed as follows:

$$\hat{x} = \mathbf{h}^T(k)\mathbf{x}(k) \tag{4.20}$$

The aim is the minimization of the mean squared compensation error

$$\mathrm{E}\{e^2(k)\} = \mathrm{E}\{(\tilde{x}(k) - \mathbf{h}^T(k)\mathbf{x}(k))^2\} \tag{4.21}$$

with

$$\mathbf{x}(k) = [x(k), x(k-1), \ldots, x(k-m+1)]^T \tag{4.22}$$

and

$$\mathbf{h}(k) = [h_0(k), h_1(k), \ldots, h_{m-1}(k)]^T. \tag{4.23}$$

The gradient of the compensation error function can be expressed in vector form as

$$\nabla(k) = \frac{\partial \mathrm{E}\{e^2(k)\}}{\partial \mathbf{h}(k)} = 2 \cdot \mathrm{E}\left\{e(k)\frac{\partial e(k)}{\partial \mathbf{h}(k)}\right\} = -2 \cdot \mathrm{E}\{e(k)\mathbf{x}(k)\}. \tag{4.24}$$

The gradient is proportional to the deviation of the current coefficient vector $\mathbf{h}(k)$ from the optimum coefficient vector $\mathbf{h}_{\mathrm{opt}}(k)$. Consequently, the current coefficient

[2] As the compensation filter is adapted with the help of the signal $x(k)$, the system distance does, as shown in the following Section 4.5, in fact also depend on $x(k)$.

vector is to be changed in the direction of the negative gradient. To do so, the mean gradient is approximated by the current gradient

$$\hat{\nabla}(k) = -2e(k)\mathbf{x}(k). \tag{4.25}$$

This yields the following adaptation rule

$$\mathbf{h}(k+1) = \mathbf{h}(k) + \beta(k)e(k)\mathbf{x}(k) \tag{4.26}$$

where the time-variant stepsize factor

$$\beta(k) = \frac{\alpha}{||\mathbf{x}(k)||^2} \tag{4.27}$$

must satisfy the stability condition

$$0 < \alpha < 2. \tag{4.28}$$

As already pointed out, the residual echo $e(k)$ is not available in an isolated form in real systems. Therefore, the output signal

$$\hat{s}(k) = s(k) + n(k) + e(k) \tag{4.29}$$

is used, so that the adaptation rule

$$\mathbf{h}(k+1) = \mathbf{h}(k) + \beta(k)\hat{s}(k)\mathbf{x}(k) \tag{4.30}$$

is acquired. This adaptation rule however, does not minimize the power of the residual echo $e(k)$, but the power of the output signal $\hat{s}(k)$. Because of eq. (4.29), this leads to an undesired distortion of the useful signal $s(k)$. The adaptation would therefore need to be stopped when the near speaker becomes active. In practice, this problem is addressed indirectly by an adaptive stepsize.

The derived algorithm for the adaptive identification of the filter coefficients $\mathbf{h}(k)$ is referred to as *Normalized Least-Mean-Square (NLMS)* algorithm. This principle of time-variant filtering is of major significance for (adaptive) digital signal processing and is used for a large variety of fairly different algorithms.[3]

The stepsize factor α must in practice be controlled adaptively. A possible optimization criterion is the average change of the system distance according to equation (4.16) in the transition from the time k to $k+1$

$$||\mathbf{\Delta}_E(k)||^2 \doteq \mathrm{E}\{||\mathbf{d}(k)||^2\} - \mathrm{E}\{||\mathbf{d}(k+1)||^2\} \tag{4.31}$$

to be maximized, i. e.

$$\frac{\partial||\mathbf{\Delta}_E(k)||^2}{\partial\alpha(k)} = 0. \tag{4.32}$$

This yields, with several intermediate steps, the solution

$$\alpha_{\mathrm{opt}}(k) = \frac{\mathrm{E}\{e^2(k)\}}{\mathrm{E}\{\hat{s}^2(k)\}}. \tag{4.33}$$

[3] A detailed description of these filters can for example be found in *Adaptive Filter Theory*, Prentice Hall, 1996.

In the case without any interferences $e(k) = \hat{s}(k)$, this results in $\alpha(k) = 1$. Granted that the signals $s(k)$, $n(k)$ and $e(k)$ are statistically independent, the expected value from eq. (4.29) is

$$\mathrm{E}\{\hat{s}^2(k)\} = \mathrm{E}\{s^2(k)\} + \mathrm{E}\{n^2(k)\} + \mathrm{E}\{e^2(k)\}. \tag{4.34}$$

With this assumption and with eq. (4.33), it follows for the *optimum stepsize factor*

$$\begin{aligned} \alpha_{\mathrm{opt}} &= \frac{\mathrm{E}\{e^2(k)\}}{\mathrm{E}\{s^2(k)\} + \mathrm{E}\{n^2(k)\} + \mathrm{E}\{e^2(k)\}} \\ &= \frac{1}{1 + \dfrac{\mathrm{E}\{s^2(k)\} + \mathrm{E}\{n^2(k)\}}{\mathrm{E}\{e^2(k)\}}} \leq 1. \end{aligned} \tag{4.35}$$

If successful at setting the stepsize according to this expression, the stepsize is automatically reduced in case of double talk. The distortion of the speech signal $s(k)$ is prevented or at least reduced and the adaptation must not explicitly be paused during double talk situations. Moreover, the stepsize $\alpha(k)$ has small values if the power of the residual echo $\mathrm{E}\{e^2(k)\}$ becomes small. Usually, the power of the residual signal in double talk situations is smaller than the power of the near speaker $E\{s^2(k)\}$.

In real systems, the residual echo $e(k)$ is not available in isolated form and must be estimated. Determining the residual echo (or its power spectral density) is an important problem in the development of algorithms for echo cancellation. A second problem is the possibly slow speed of convergence of the NLMS algorithm for large filter lengths or correlated speech signals which will be further inspected in the following experiments. In the literature, numerous approaches to solve these problems can be found. However, this is beyond the scope of this introduction.

4.6 Additional Measures for Echo Attenuation

In the practical operation of an echo canceller, the echo suppression achieved by the transversal filter may be insufficient in many situations. This can for example be due to a slow adaptation of the compensation filter in response to a fast change of the room impulse response. The length of the compensation filter might also be considerably shorter than the room impulse response, which likewise leads to a residual echo. In some systems, this is accepted deliberately to improve the convergence of the transversal filter and reduce its complexity. Residual echos can become audible particularly in speech pauses of the near speaker and in case of large transmission delays.

A simple measure for their suppression is provided by the so-called *center-clipper* as shown in Figure 4.10. The output signal $\hat{s}(k)$ of the echo canceller is (partly) free from residual echos by the rule

$$\tilde{s}(k) = \begin{cases} \hat{s}(k) - A & ; \ \hat{s}(k) > +A \\ 0 & ; \ |\hat{s}(k)| \leq +A \\ \hat{s}(k) + A & ; \ \hat{s}(k) < -A. \end{cases} \tag{4.36}$$

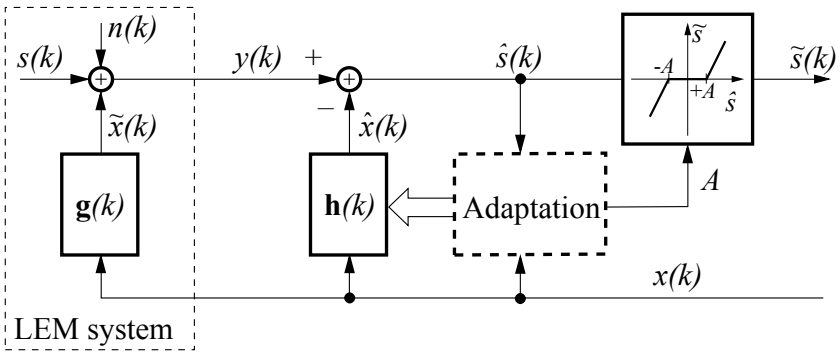


Figure 4.10: Echo canceller with center-clipper for the suppression of residual echos.

A large value of A leads to a large residual echo suppression as well as to distortion of the speech signal and vice versa. This value should therefore not be chosen too large.

Newer techniques use (instead of the center-clipper) a so-called postfilter in the transmitting branch, the coefficients of which are based on signal statistical modeling of the residual echo signal. Just like the compensation filter, the postfilter can be efficiently adapted and realized by means of block-based *frequency domain techniques*. The post filter can be used not only for the reduction of the residual echo but simultaneously for the reduction of disturbing background noise. Refer to the literature for a detailed description of these methods.[4]

[4] e. g. Stefan Gustafsson *Enhancement of Audio Signals by Combined Acoustic Echo Cancellation and Noise Reduction*, dissertation, IND, RWTH Aachen, 1999.

4.7 Exercises

In the following experiments, you should get to known the properties of the presented NLMS algorithm and learn how it operates. The file `echocomp.py` serves as a template for all of the following tasks. In the beginning of this file, you can find the function `nlms4echokomp` which is the core of this experiment. In it, a framework is given within which you will implement the NLMS algorithm in the first exercise.

After the function definition, you find a variable `exercise` that allows you to set for which of the subtasks the code should be run. In the `echokomp.npz` file, there are 4 numpy arrays. The array `s` is the speech signal $s(k)$ of the far speaker that has been sampled with 8 kHz. The arrays `g1` `g2` and `g3` are three artifically generated room impulse responses with a length of 200, 1000, and 2000 samples respectively.

Exercise 4.1 Construction of the echo canceller

The echo cancellation system according to Figure 4.8 will be simulated and analyzed. The filter $\mathbf{h}(k)$ will be adapted by means of the NLMS algorithms as described in Section 4.5. The (fixed) adaptation stepsize for the NLMS algorithms is chosen as $\alpha = 0.1$ and there is no background noise present, i. e. $n(k) = 0$. The excitation signal $x(k)$ is white noise with the variance $\sigma^2 = 0.16$.[5] For $\mathbf{g}$, the room impulse response `g1` with 200 values is used. *After the function definition of* `nlms4echocomp` *a default configuration is set. This configuration should also be used for the subsequent experiments unless stated otherwise.*

Complete the `nlms4echokomp` function and add code to plot the echo signal $\tilde{x}(k)$ and the residual echo $e(k)$ in the section for exercise 1. The two signals should be plotted in a single diagram. Plot the relative system distance $D(k)$ and the ERLE measure - both in dB - over the time. Ensure that the axes are labeled correctly!

`Note:` For better comparability, draw all three plots of this experiment in a single figure using the `subplot` command.

`Note:` Compute the ERLE measure as a function of the discrete time according to eq. (4.18) by taking the average over the previous 200 samples for the expected value in each time step k.

Exercise 4.2 Influence of the signal properties on the adaptation

The properties of the signal $x(k)$ of the far speaker have a major influence on the convergence behavior of the NLMS algorithms! To take a closer looks at this, the following three excitation signals $x(k)$ will be considered:

[5] The computation of the variance was covered in experiment 3.

a) speech $s(k)$ (given by the variable `s`)

b) white, Gaussian distributed noise $r_w(k)$ with the variance $\sigma^2 = 0.16$

c) colored noise $r_c(k)$.

The colored noise $r_c(k)$ is acquired from the white noise $r_w(k)$ by filtering with a filter that is described by the transfer function

$$H(z) = \frac{1}{1 - 0.5 \cdot z^{-1}}. \tag{4.37}$$

In this and the following experiments, only the relative system distance (in dB) will be considered. Plot it for all three cases in a single diagram. Add axes labeling and a legend to the plot to make it as self-explanatory as possible.

`Note:` The variables `x`, `g`, `noise`, `alphas` and `mh` are lists where each entry should represent a choice of the same parameter. If a parameter is not varied in an experiment, the corresponding list should have three identical entries.

What signal property is important for a good convergence behavior?

Exercise 4.3 Adaptation with background interference and noise as excitation

The transmitted excitation signal $\tilde{x}(k)$ now suffers from a background noise $n(k)$. The latter is modeled by additive white noise with a variance of $\sigma^2 = 0$ (no interference), $\sigma^2 = 0.001$ and $\sigma^2 = 0.01$ respectively. Draw the relative system distances for these three cases (in one diagram).

Exercise 4.4 Adaptation with background interference and speech as excitation

Repeat the previous experiment with speech as excitation signal. What are the prominent differences compared to the previous experiment?

Exercise 4.5 Influence of the stepsize

The background interference $n(k)$ is now white noise with the variance $\sigma^2 = 0.01$. (The excitation signal is again white noise with the variance $\sigma^2 = 0.16$.) How do the different stepsizes $\alpha \in \{0.1, 0.5, 1.0\}$ affect the relative system distance under these circumstances?

Exercise 4.6 Influence of the compensation filter length

The length m of the transversal filter is usually smaller than the length m' of the room impulse response in practice, i. e. $m < m'$. Compute the relative system distance for the cases $m = m' - 10$, $m = m' - 30$, and $m = m' - 60$. What do you notice?

Exercise 4.7 Room impulse responses of different lengths

The three different (fixed) room impulse responses for this experiment are given by the cell array `g`. The length of the compensation filter is assumed to be always equal to the length of the room impulse response, i. e. $m = m'$. (The excitation signal is white noise and no background interference is present.) Plot the relative system distance for these three cases! What do you observe?

Chapter 5

Digit Recognition using a Linear Model

In this lab we will use machine learning to perform a classification of audio data into distinct categories. Moreover, we will walk through the typical phases of developing a machine learning algorithm, which are data acquisition, feature selection, model selection, hyperparameter tuning and evaluation. With `sklearn`, a library is introduced that covers important functionality to develop machine learning algorithms. Moreover, we will use `librosa` for audio feature extraction. First of all, a compact introduction into the basics of machine learning and two popular classifiers is given.

5.1 Introduction to Machine Learning

As you might guess, this section can only cover a mere fraction of this huge (and growing) topic. If you want to dive in deeper take a look into the literature, e. g. Bishop[1]. There are many ways to divide and classify the field of machine learning. One of them is the distinction between *supervised* and *unsupervised* learning (and also *reinforcement learning* which is not covered here). Supervised learning refers to tasks, where a set of labeled training data is known. For instance, consider many images that show either cats and dogs or many sound snippets that contain either a piano or a saxophone. The term *labeled* refers to the situation that we know for each snippet whether it shows a cat or a dog, respectively. What were pictures or sound snippets in the previous example is generally called *instances*, *samples* or *examples*. In the following, we denote the number of instances as N, where $\boldsymbol{x}^{(n)}$ is the instance n and $t^{(n)}$ is the corresponding label. The task of supervised learning is then to present all pairs $\left(\boldsymbol{x}^{(n)}, t^{(n)}\right)$ to a *model*. For now, this model is not more than a black box with a memory that can learn a relationship between $\boldsymbol{x}$ and t. This procedure is mostly called *training* or *fitting*. If successful, we can present unseen instances to the trained model and it will make a meaningful prediction on whether the instance contains a cat or a dog playing saxophone.

In contrast, unsupervised learning is the task of learning something from the instances without knowledge of any targets. In other terms, the model does not learn a relation between instances and targets, but relations within the data itself. In most cases this results either in the detection of distinct groups in the data (so called *clusters*), or in another representation of the data.

Supervised machine learning tasks can be divided into *regression* and *classification* tasks. An example for a classification task has been given in the previous section.

[1] Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

In such tasks, the model makes a decision to which of the K classes (e.g. cats and dogs) the sample belongs. In regression, the target data cannot be divided in K possible results. Instead, we want to predict continuous variables like the pitch of an instrument or a stock price in the next 10 minutes. In this lab course, we will focus on classifiers.

5.2 Classifiers

5.2.1 Problem Statement

Let us start with the example depicted in Fig. 5.1. We have a set of points (vectors) $\boldsymbol{x}^{(n)} = [x_1^{(n)}, x_2^{(n)}]^T$ of which we know if they belong to the orange or the blue class. Since every vector consists of a x_1 and x_2 coordinate, we have $F = 2$ *input features*. Since there are blue and orange points, we also have $K = 2$ *classes*.

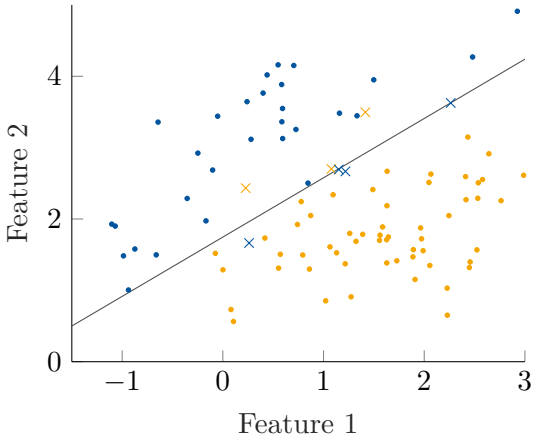


Figure 5.1: A simple labeled dataset

The goal of a classifier is to provide a clear decision rule that assesses, whether a new unseen point belongs to the blue or orange class. A special class of classifiers is the group of *linear classifiers* or *linear models*. These models predict the class membership, by evaluating whether the points are above or below a hyperplane. In a two dimensional feature space like in the example, the hyperplane is just a (one dimensional) line. In numbers, an instance $\boldsymbol{x}$ is predicted to belong to class blue if $w_2 x_2 > w_1 x_1 + w_b$. Here, $\boldsymbol{w} = [w_1, w_2]^T$ and w_b are the parameters, the model learned during training. More generally, we may state this condition as

$$x_1 w_1 + x_1 w_2 > w_b \tag{5.1}$$

which is nothing else than an inner product between the training instance and a vector containing the learned parameters. If we append the negative threshold $-w_b$ to the feature vector and append a 1 to every instance vector, we may more elegantly write

$$a = \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x} > 0 \tag{5.2}$$

where a is the *score*. This equation is central to all linear models for classification.

5.2.2 Probabilities

Before we continue, let us quickly recap on probabilities. For convenience, C_k is the name of both the k$^{\text{th}}$ class and the hypothesis that a sample belongs to this class. The true class of sample n is denoted as $t^{(n)}$, i. e. $t^{(n)} = C_k$ means that instance n belongs to class k.

$p(\boldsymbol{x})$ is the probability density function or *prior distribution* of $\boldsymbol{x}$. Precisely it is $p(x_1, x_2, \ldots, x_F)$, where F is the number of features. $p(\boldsymbol{x}_0)$ states the probability that a random sample will be located in a small interval around $\boldsymbol{x}_0$. For the example above, $p(\boldsymbol{x})$ is depicted in Fig. 5.2a. Note that when we are given a new dataset, we do not have access to $p(\boldsymbol{x})$ because there exist infinitely many distributions that could have yielded the same data points. What is depicted in Fig. 5.2a is the distribution that has been used to generate the example dataset. Luckily, there exist many techniques to estimate the a distribution from a number of points [2] or Chapter 2 in [1].

$p(C_k)$ is the probability that a randomly chosen sample belongs to class C_k. It is also called *class priors*. If $p(C_k)$ is equal for all k, we call the dataset *balanced*. Since in the example above exist more blue than orange points, it is unbalanced. Albeit not being very complex, the class priors for the example data are plotted in Fig. 5.2c

$p(\boldsymbol{x}|C_k)$ are the *class conditional prior distribution*. It is related to the prior disribution $p(\boldsymbol{x})$, with the difference that we have two (or K) probabilities, one for each class. So for instance $p(\boldsymbol{x}|C_1)$ is the probability density function of all samples that belong to class C_1. For the example, these distributions are depicted in Fig. 5.2b.

$p(C_k|\boldsymbol{x})$ in turn, is the probability that a given instance $\boldsymbol{x}$ belongs to class C_k. It is called the *posterior probability*. If we have access to this quantity, we can use it directly to make a prediction for C. If $p(C_2|\boldsymbol{x}) > p(C_1|\boldsymbol{x}_n)$ we can predict that $\boldsymbol{x}_n$ belongs to class 2, i. e. $t^{(n)} = C_2$. For the example it is depicted in Fig. 5.2d. Obviously, in the area where most blue points are located, it is likely that an unlabeled datapoint belongs to the class of blue points as well. In the region, where the probability surfaces intersect we could at least reject the decision.

For these probabilities, the following rules hold:

$$\sum_k p(C_k) = 1. \tag{5.3}$$

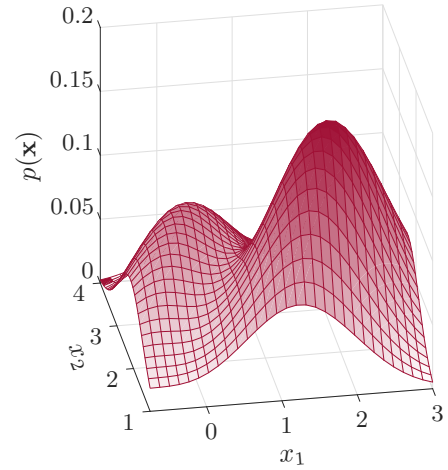
The latter might be intuitive.

$$p(\boldsymbol{x}) = \sum_k p(\boldsymbol{x}|C_k)p(C_k) \tag{5.4}$$

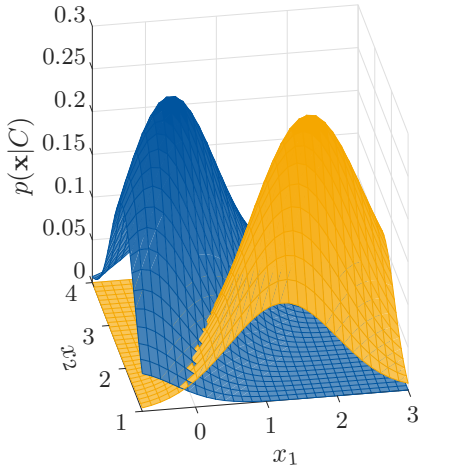
The class priors can be composed of (or decomposed into) the class conditional priors, weighted by the class priors. The procedure (5.4) is also called *marginalization*.

$$p(C_k) = \int_{\mathbb{R}^F} p(C_k|\boldsymbol{x})p(\boldsymbol{x})\mathrm{d}\boldsymbol{x} \tag{5.5}$$

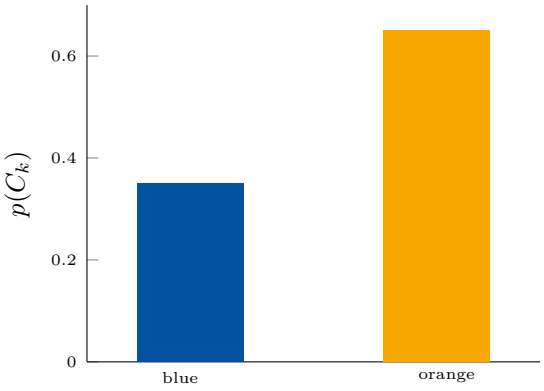
[2] https://scikit-learn.org/stable/modules/density.html



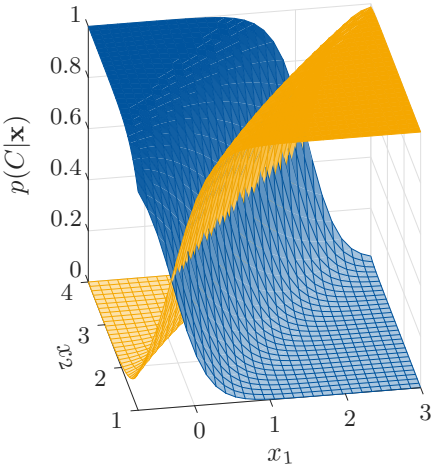
(a) Prior Distribution



(b) Class Conditional Prior Distributions



(c) Class Priors



(d) Posterior Distributions

Figure 5.2: Prior and Posterior Distributions for the example dataset

The relation eq. (5.5) is complementary to the previous point and is called marginalization as well.

$$p(C_k|\boldsymbol{x}) = p(C_k)\frac{p(\boldsymbol{x}|C_k)}{p(\boldsymbol{x})} \tag{5.6}$$

Eq. (5.6) follows from Bayes' Rule. Inserting eq. (5.4) into eq. (5.6) yields

$$p(C_k|\boldsymbol{x}) = \frac{p(C_k)p(\boldsymbol{x}|C_k)}{\sum_j p(\boldsymbol{x}|C_j)p(C_j)}. \tag{5.7}$$

This states that the posteriors can be obtained by the ratio of class conditional priors to each other. If $p(\boldsymbol{x}|C_1) \gg p(\boldsymbol{x}|C_2)$, then $p(C_1|\boldsymbol{x})$ gets close to 1. Finally, from eq. (5.7) it follows that

$$\sum_k p(C_k|\boldsymbol{x}) = 1 \tag{5.8}$$

5.2.3 Logistic Regression

Logistic Regression is a popular linear model for classification. In fact, we use a regression model to predict the posterior probability $p(C_k|\boldsymbol{x})$ that a presented instance belongs to the respective classes. This likelihood is used to make a prediction on the instances class.

The principle of logistic regression is often used as the last stage of deep neural networks for classification. It arises naturally, when the last layer uses a softmax activation function and the net uses a cross-entropy loss.

Principle

In Fig. 5.2d you see that the posterior probability $p(C_k|\boldsymbol{x})$ is somewhat s-shaped. This shape arises when the prior distributions $p(\boldsymbol{x}|C_k)$ are Gaussian. You can easily show that if they have the same covariance matrix $\boldsymbol{\Sigma}$ and individual expectation values $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, inserting in eq. (5.7) yields that the result has the form

$$p(C_k|\boldsymbol{x}) = \frac{1}{1-\exp^{-\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}} \tag{5.9-a}$$

$$= \frac{1}{1-\exp^{-a_k}} \tag{5.9-b}$$

$$= \sigma(a_k) \tag{5.9-c}$$

The function $\sigma(a)$ is called *sigmoid*, which means 's-shaped' or *logistic* function which gives the classifier its name. The idea of logistic regression is to use eq. (5.9-a) to make a prediction about the class of a (new) point. As a result, only vector $\boldsymbol{w}$ has to be stored to describe the model. Once we have fitted the logistic function onto our data, vector $\boldsymbol{w}$ is fixed. We can assign a new point to class C_1 when $p(C_1|\boldsymbol{x}) > 1 - p(C_1|\boldsymbol{x}) = p(C_2|\boldsymbol{x})$ and vice versa.

When there are more than two classes, one classifier is fitted for each class. The classifier belonging to class k only makes an estimation $\hat{p}(C_k|\boldsymbol{x})$ of the probability whether a point belongs to class k or not. The normalized posterior probabilities for all classes are obtained by applying eq. (5.8). When each class has a weight vector $\boldsymbol{w}_k$, this procedure can be expressed analogously by

$$p(C_k|\boldsymbol{x}) = \frac{\exp^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}}{\sum_\kappa \exp^{\boldsymbol{w}_\kappa^{\mathrm{T}}\boldsymbol{x}}} \tag{5.10}$$

When we define $\hat{\boldsymbol{y}} = [\hat{p}(C_1|\boldsymbol{x}), \hat{p}(C_2|\boldsymbol{x}), \ldots, \hat{p}(C_K|\boldsymbol{x})]^{\mathrm{T}}$ and $\boldsymbol{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots \boldsymbol{w}_K]^{\mathrm{T}}$ we can state this equation as

$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{W}\boldsymbol{x}) \tag{5.11}$$

The softmax function combines the evaluation of the logistic function and subsequent normalization. The fact that the maximum value in $\boldsymbol{W}\boldsymbol{x}$ is dragged close to one and all other values are dragged close to zero, gives this function its name.

Training and Cost Function

Albeit the use of the sigmoid function is motivated by possibly gauss-distributed priors, it is not required that the data follows a Gaussian distribution. Hence, no knowledge of ($\boldsymbol{\mu}_1$, $\boldsymbol{\mu}_2$ and $\boldsymbol{\Sigma}$) is required for training. During training, only matrix $\boldsymbol{W}$ is updated until the estimated posterior probability fits the dataset. The update, as well as the assessment whether the fit is acceptable, is done by defining a cost function that depends on $\boldsymbol{W}$ and the dataset. For regression tasks, the squared loss is widely used. When we want to predict probabilities as we do in logistic regression, there is a more elegant solution called *cross-entropy loss*. This loss states the deviation between probability densities. In our case, this is the deviation between the predicted posteriors and the true posterior probability.

Assume, the model predicts the posterior probabilities for one example n as $\hat{\boldsymbol{y}}^{(n)} = [0.25, 0.1, 0.6, 0.05]^{\mathrm{T}}$ and we know that the example (or instance) belongs to class 3, i. e. $t^{(n)} = C_3$. Then, the (true) posterior probabilities are known, namely $\boldsymbol{y}^{(n)} = [0, 0, 1, 0]^{\mathrm{T}}$. This is often referred to as *one-hot-encoding*. With this notation, the cross-entropy loss for any pair of predicted and true posteriors $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ is

$$\mathcal{C}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_k y_k \log \hat{y}_k \tag{5.12}$$

Due to the sparsity of $\boldsymbol{y}$, the loss for this instance is determined by $-\log \hat{y}_3$. It follows, that the loss is smallest when $\hat{y}_3$ gets close to one.

In order to adapt $\boldsymbol{W}$, the cost function must be derived with regard to the rows $\boldsymbol{w}_k$ of $\boldsymbol{W}$. To do so, the forward rule eq. (5.11) is inserted into eq. 5.12. Then, we obtain the gradient

$$\nabla \mathcal{C}(\boldsymbol{w}_k) = (\hat{y}_k^{(n)} - y_k^{(n)})\boldsymbol{x} \tag{5.13}$$

In the mathematical notation and in the literature[1], all $\boldsymbol{x}^{(n)}$ are column vectors in a data matrix $\boldsymbol{X}$ so that the rows of this matrix correspond to features. In python, the data is arranged differently so that each row of $\boldsymbol{X}$ is a feature vector. Precisely, `X[0]` (or `X[0, :]`) returns the first instance and `X[:, 0]` returns the first feature for all instances.

We can make use of this gradient to find a weight matrix $\boldsymbol{W}_{\text{opt}}$ that is optimal in the sense that

$$\boldsymbol{W}_{\text{opt}} = \arg\min_{\boldsymbol{W}} \mathcal{C}(\boldsymbol{W}) \tag{5.14}$$

In some cases we might find a closed form solution by solving the set of equations $\nabla \mathcal{C}(\boldsymbol{W}) = \boldsymbol{0}$. However, this often requires an inversion of the (large) data matrix $\boldsymbol{X}$. Instead, we can use that the gradient shows in the direction of the steepest ascent. If we drag $\boldsymbol{w}$ in the opposite direction we obtain the update rule

$$\boldsymbol{w}_k \leftarrow\!\shortmid \boldsymbol{w}_k - \eta \nabla \mathcal{C}(\boldsymbol{w}_k) \tag{5.15}$$

where η is the *learning rate*. This update is called *gradient descent* and it should be nothing new, since you used the exact same approach for the LMS algorithm in Lab Course 4, minimizing the quadratic cost between a desired signal and the output of an adaptive filter.

Logistic regression in python

We can generate a dataset similar to the example, using `sklearn.datasets.make_blobs`.

```
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split

X_blob, t_blob = make_blobs(n_samples=500, n_features=2, centers=[[-0.5,
↪  -0.3], [0, .5], [0.5, -0.5]], cluster_std=0.2, random_state=42)
X_train, X_test, t_train, t_test = train_test_split(X_blob, t_blob,
↪  train_size=0.85)
```

```
>>> X_train.shape
(425, 2)
```

When you are about to plot multiple similar plots, it is useful to define parameter dictionaries with the common settings. Here, the training data is plotted transparent and the test data is plotted solid and slightly larger. The result is depicted in Fig. 5.3.

```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
import numpy as np

cmap = ListedColormap(np.array([[0, 84, 159], [204, 7, 30], [246, 168,
↪  0]])/255)

plot_dict_train = dict(cmap=cmap, s=10, alpha=0.55, marker='.')
plot_dict_test = dict(cmap=cmap, s=20, alpha=1, marker='o')

fig, ax = plt.subplots()
ax.scatter(X_train[:,0], X_train[:,1], c=t_train, **plot_dict_train)
ax.scatter(X_test[:,0], X_test[:,1], c=t_test, **plot_dict_test)
```

Training a logistic regression model is as easy as follows:

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(random_state=42)
lr.fit(X_train, t_train)
t_test_pred = lr.predict(X_test)
```

After the code has been executed, `lr` is an object containing the estimator. The matrix $\boldsymbol{W}$ is stored in the `lr.coef_` property. `lr.fit()` returns an array with the predicted class labels, here 0, 1 and 2. We can compute the accuracy by counting how many points in the test set were classified correctly.

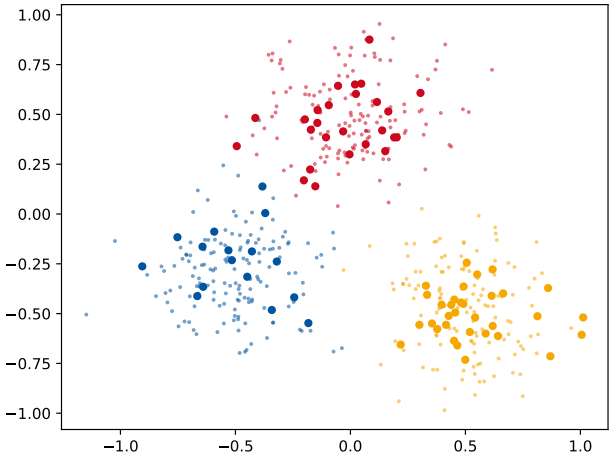


Figure 5.3: The 'blob' data set

```
acc = np.count_nonzero(t_test_pred==t_test) / len(t_test)
```

For convenience, there exists the function `sklearn.metrics.accuracy_score` which computes the accuracy for given `t_test` and `t_test_pred`.

Apart from hard decisions obtained by `lr.predict`, the method `lr.predict_proba` predicts the posterior probabilities according to eq. (5.11). We can define a uniform coordinate grid to evaluate the results systematically.

```
x1, x2 = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
X = np.array(list(zip(x1.flat, x2.flat)))
p_pred = lr.predict_proba(X)
```

This data can be visualized using `pcolormesh`. This function receives two coordinate grids and the actual data.

```
fig, ax = plt.subplots(1, 3, sharey=False)
for it in range(3):
    ax[it].pcolormesh(x1, x2, p_pred[:,it].reshape(100, 100),
    ↪  shading='nearest', cmap='bone')
```

As desired, the regions where the posterior probabilities are one correspond to the regions where the respective point clouds are located. The result is depicted in Fig. 5.4.

5.2.4 K Nearest Neighbours

The *k nearest neighbours* (k-NN) algorithm follows an entirely different approach than logistic regression. In logistic regression it is assumed that the class conditional

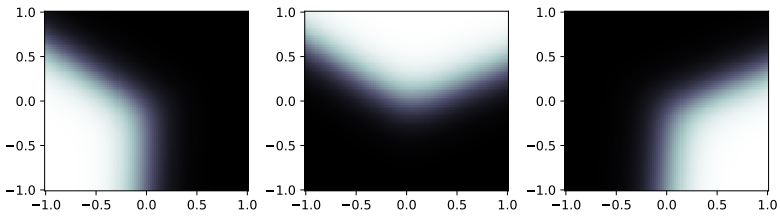


Figure 5.4: Posterior Probabilities for the 'blob' data set

priors are linearly separable so that one weight matrix is sufficient to describe the model and to make predictions. This matrix is obtained by gradient descent. With a k-NN estimator the entire training set is saved and employed for the prediction. When a new point is presented to the estimator, its k nearest neighbours are searched in the data set. Their classes are known and can be used to make a prediction on the new point's class. This is usually done by a majority vote. Unless stated otherwise, the term 'nearest' refers to the least Euclidean distance.

In `sklearn`, all estimators follow the same API. Thus, creating a k nearest neighbour model is similar to the previous example with logistic regression.

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, t_train)
t_test_pred = knn.predict(X_test)

acc = np.count_nonzero(t_test_pred==t_test) / len(t_test)
```

With the same data set as before, this estimator achieves an accuracy of 100%. Like the logistic regression model, the k nearest neighbour estimator can also predict posterior probabilities. Here, the posterior probability of class C is the number of neighbours with this class divided by the total number of neighbours. As a result, the posterior probability has discrete values.

One fundamental benefit of the k nearest neighbour classifier becomes apparent, when the data set is not linearly separable. In the following example, such a data set is created and tackled with linear regression, and k nearest neighbours. Then, the estimated posterior probability for one of the classes is evaluated on a uniform grid.

```
from sklearn.datasets import make_moons

X_moon, t_moon = make_moons(n_samples=500,noise=0.15, random_state=42)

lr.fit(X_moon, t_moon)
knn.fit(X_moon, t_moon)

# create uniform grid
x1, x2 = np.meshgrid(np.linspace(min(X_moon[:, 0]), max(X_moon[:, 0]), 100),
                     np.linspace(min(X_moon[:, 1]), max(X_moon[:, 1]), 100))
X = np.array(list(zip(x1.flat, x2.flat)))
```

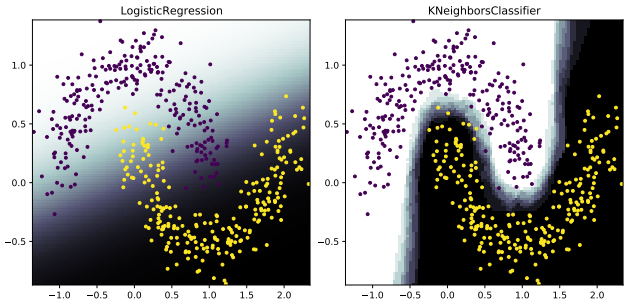


Figure 5.5: The predicted posterior probabilities for the 'moons' data set

```
# predict posteriors for grid
p_pred_lr = lr.predict_proba(X)
p_pred_knn = knn.predict_proba(X)

fig, ax = plt.subplots(1, 2)
ax[0].pcolormesh(x1, x2, p_pred_lr[:, 0].reshape(100, 100), shading='nearest',
↪  cmap='bone')
ax[0].scatter(X_moon[:, 0], X_moon[:, 1], c=t_moon, s=10)
ax[0].set_title('LogisticRegression')

ax[1].pcolormesh(x1, x2, p_pred_knn[:, 0].reshape(100, 100),
↪  shading='nearest', cmap='bone')
ax[1].scatter(X_moon[:, 0], X_moon[:, 1], c=t_moon, s=10)
ax[1].set_title('KNeighborsClassifier')
```

The result in Fig. 5.5 makes it apparent that the logistic regression classifier is not able to make meaningful predictions on the test grid, since the posterior probability does not follow the logistic model. The k-NN approach matches the distribution better, since neighbouring data points mostly belong to the same class.

Up to here, you learned about two opposing approaches. The logistic regression model parametrizes the posterior probability by means of a single matrix $\boldsymbol{W}$. It is only able to separate the data linearly. In contrast, the k-NN approach is able to model any prior distribution of the data, but has to memorize the complete data set. A compromise is given by so called *support vector machines* (SVMs). With this approach, only a few support vectors are stored, that are sufficient to model the prior distribution. In its simplest form, the SVM can only separate the data linearly. However, by application of the *kernel trick* the separation of data like in the previous example is possible. The (kernel) SVM only needs to store a fraction of data, compared to the k-NN approach. For a detailed explanation see Chapter 6 and 7 in [1] or try `sklearn.svm.SVC`.

5.3 Classification on High Dimensional Data

With only two features x_1 and x_2, the previous examples were well suited to demonstrate the basic classification problem and the challenge of linear separability. When

the input vectors have more features (i. e. the feature space has more dimensions) we may jump from the frying pan and into the fire. With every feature that is added to the input vectors, it gets more likely that a linear classifier finds a hyperplane that can separate classes in the training data. At some point, however, there are too many options which all separate the training data. The training will converge to the option that minimizes the cost criterion for the training data. As a result, the model fits the training data perfectly, but might loose its capability to generalize on unseen data. This phenomenon is called *overfitting* and is likely to occur when there are too many features, too few training samples and when the data is noisy.

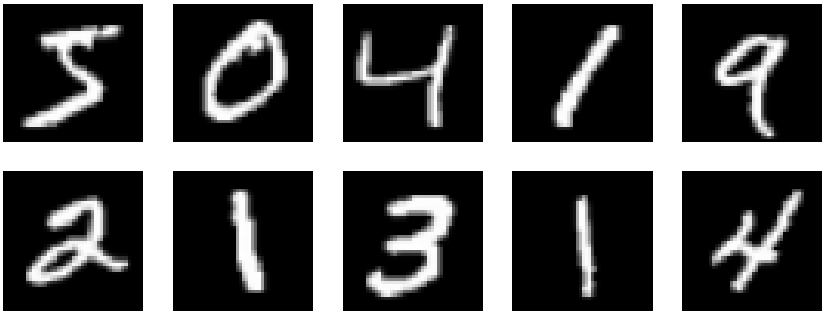


Figure 5.6: The first 10 samples from the MNIST training set

We can illustrate this situation with a slightly advanced and more popular dataset, namely MNIST. It consists of 70.000 images of handwritten digits as depicted in Fig. 5.6. The data set can be used to develop a model that detects written digits. We can load it as follows.

```
from tensorflow.keras.datasets import mnist
(x_train, t_train), (x_test, t_test) = mnist.load_data()
```

In order to demonstrate the effect of regularization, only a subset of training and test data is kept and corrupted with Gaussian noise. The function `np.random.permutation(L)` returns a sequence with all numbers from 0 to L-1 in a random order.

```
# select random subset
idx_train = np.random.permutation(len(x_train))[:6000]
x_train = x_train[idx_train].astype(np.float64)
t_train = t_train[idx_train]


idx_test = np.random.permutation(len(x_test))[:1000]
x_test = x_test[idx_test].astype(np.float64)
t_test = t_test[idx_test]

# add noise
x_train += np.std(x_train)*np.random.randn(*x_train.shape)
x_test += np.std(x_train)*np.random.randn(*x_test.shape)
```

Each picture consists of 28×28 pixels. If the columns of each pictures are stacked into one column, a vector with 784 features is obtained for each training instance. This process is often referred to as *Flattening* and can be done with the `reshape` command.

```
x_train = x_train.reshape((x_train.shape[0], -1))
x_test = x_test.reshape((x_test.shape[0], -1))
```

Again, we can train a linear model and a k nearest neighbour classifier.

```
lr = LogisticRegression()
lr.fit(x_train, t_train)
acc_lr = accuracy_score(t_test, lr.predict(x_test))

knn = KNeighborsClassifier()
knn.fit(x_train, t_train)
acc_knn = accuracy_score(t_test, knn.predict(x_test))
```

The achieved test accuracies are only 72.00% for logistic regression and 88.20% for the k-NN estimator. It can be observed, that the estimators take more time to fit and predict. The logistic regression model even throws a warning that the training did not converge.

In the next sections, techniques are presented that help to deal with non ideal data.

5.3.1 Scaling and Normalization

For many algorithms that rely on gradient descent, such as logistic regression, it is beneficial if all features are on the same scale. For this, there exist (among others) two solutions in `sklearn`, namely `MinMaxScaler` and `StandardScaler`. The first transforms the training data in a way, that the user can specify the minimum and the maximum of each feature. The second transforms the data, such that each feature is zero mean and/or has unit variance.

```
from sklearn.preprocessing import StandardScaler

scale = StandardScaler()
x_train_scale = scale.fit_transform(x_train)
x_test_scale = scale.transform(x_test)
```

When we call `fit_transform`, the scaler computes the mean and the variance of the training set and saves them as object properties. When `transform` is called on the test set, these values are used to scale the test data in a similar manner. As a result, the mean and variance of each feature in the transformed test data might not be exactly 0 and 1, respectively. Again, the test data is only to test our models and should never be used to change its properties.

For the k-NN classifier, scaling should be treated with caution. The classifier performs best, when informative features contribute more to the distance metric than less informative features. In the current example, the standard scaler would increase the influence of less informative pixels and decrease the accuracy of the k-NN estimator. An alternative will be shown soon, by means of Neighbourhood Component Analysis.

5.3.2 Regularization

This section only holds for logistic regression. When you suspect that a model fitted too well on the training data so that it looses its ability to generalize, you can penalize the use of too many input features. This is done by adding a term to the cost function that penalizes too large values in the weight matrix. The cost function then reads

$$\mathcal{C}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_k y_k \log \hat{y}_k + \lambda \|\boldsymbol{W}\|_L \tag{5.16}$$

where L is the selected matrix norm, which is 2 by default. The regularization strength is controlled with parameter λ. In `sklearn`, however, its inverse `C` is used, such that a small value of `C` corresponds to a high regularization. By default, this value is set to 1. The value of `C` highly affects the test accuracy and it is difficult to guess which value is optimal. Thus, the optimal value can be found by trial and error. If you do not know where to start, you should first try different orders of magnitude. Since the training did not converge in the first try, we increase the maximum number of iterations from 100 to 1000. Usually, the training terminates earlier, when the cost criterion does not decrease any further.

```
all_C = np.logspace(-5, 0, 11)
all_acc = np.zeros_like(all_C)
for it, C in enumerate(all_C):
    lr  = LogisticRegression(C=C, max_iter=1000)
    lr.fit(x_train, t_train)
    all_acc[it] = accuracy_score(t_test, lr.predict(x_test))
    print("C: {:1.0e}\tacc:{:4.2f}%".format(C, all_acc[it]*100))
```

We obtain:

```
C: 1.00e-05        acc: 56.70%
C: 3.16e-05        acc: 71.70%
C: 1.00e-04        acc: 79.90%
C: 3.16e-04        acc: 83.10%
C: 1.00e-03        acc: 83.70%
C: 3.16e-03        acc: 82.60%
C: 1.00e-02        acc: 80.20%
C: 3.16e-02        acc: 76.70%
C: 1.00e-01        acc: 73.60%
C: 3.16e-01        acc: 72.30%
C: 1.00e+00        acc: 70.90%
```

Apparently, $C \approx 10^{-3}$ is best suited to regularize the estimator for the given problem.

5.3.3 Feature Selection

If we have a look at Fig. 5.6, we see that the pixels in the corner contain no valuable information, as they are always black in every sample. For our training time it would be favorable to discard these features. We can use `sklearn.features_selection.SelectKBest` to select a certain number of best features. Additionally, we have to specify a metric to assess what is 'best'. In the following example, we use the mutual information from information theory and continue only with the 300 best features.

```
from sklearn.feature_selection import SelectKBest, mutual_info_classif
sel = SelectKBest(mutual_info_classif, k=300)
x_train_kbest = sel.fit_transform(x_train, t_train)
x_test_kbest = sel.transform(x_test)
```

This procedure is beneficial for the k-NN estimator, as it mitigates the influence of irrelevant features on the distance metric.

```
knn = KNeighborsClassifier()
knn.fit(x_train_kbest, t_train)
acc_knn = accuracy_score(t_test, knn.predict(x_test_kbest))
```

Compared to the first try, the test accuracy is slightly increased (89.40 %). The effect is stronger, when the data is more corrupted.

5.3.4 Unsupervised Dimensionality Reduction: PCA

Although we removed unnecessary features in the first step, our dataset still contains redundant information. As neighbouring pixels tend to have a similar intensity value, they are correlated. This redundancy can be used to further decrease the number of features. Instead of eliminating features, we compute a few descriptive features as a linear combinations from all features. This leads us to the principle of *principal component analysis* (PCA). A comprehensive view on PCA can be found in Section 12.1 of [1].

One possible application of PCA is to find an under complete orthonormal basis $\boldsymbol{A}$, so that the projections on this basis $\boldsymbol{X}' = \boldsymbol{A}\boldsymbol{X}$ can be used to reconstruct the original values $\boldsymbol{x}$ as good as possible. This reconstruction is performed again with matrix $\boldsymbol{A}$ and can be described by

$$\hat{\boldsymbol{X}} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{X} \tag{5.17}$$

By use of the $F \times F$ covariance matrix

$$\boldsymbol{\Psi} = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} \tag{5.18}$$

an optimal solution for $\boldsymbol{A}$ can be found by the eigenvalue decomposition

$$\boldsymbol{\Psi} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{\Lambda}\boldsymbol{A} \tag{5.19}$$

where $\boldsymbol{\Lambda}$ is the covariance matrix of $\boldsymbol{X}'$. From the fact that $\boldsymbol{\Lambda}$ is diagonal, it follows that all features from $\boldsymbol{X}'$ are uncorrelated. Usually, the diagonal entries of $\boldsymbol{\Lambda}$ are ordered in descending order so that the first column vector of $\boldsymbol{A}$ explains the most variance of $\boldsymbol{X}$.

One practical drawback of this procedure is that eq. (5.18) requires the whole data set at once. For large data sets, it is more efficient to compute the PCA on small subsets of the data, so called *mini batches*. This procedure is referred to as *incremental PCA*.

Since PCA does not consider class labels, it is an unsupervised procedure. Prior to PCA, a `StandardScaler` should be used. Hence, we continue with `x_train_scale` and `x_test_scale`.

```
from sklearn.decomposition import IncrementalPCA as PCA

all_n = np.array([16, 32, 64, 128, 200, 300, 400, 500, 600, 700])

for it, n_PCA in enumerate(all_n):
    pca = PCA(n_components=n_PCA)
    x_train_pca = pca.fit_transform(x_train_scale)
    x_test_pca = pca.transform(x_test_scale)

    lr = LogisticRegression(max_iter=1000)
    lr.fit(x_train_pca, t_train)
    acc_lr = 100*accuracy_score(t_test, lr.predict(x_test_pca))

    knn = KNeighborsClassifier()
    knn.fit(x_train_pca, t_train)
    acc_knn = 100*accuracy_score(t_test, knn.predict(x_test_pca[:, :n_PCA]))

    print("n: {}\tlr:{:4.2f}%\tknn: {:4.2f}".format(n_PCA, acc_lr, acc_knn))
```

This results in the following output:

```
n: 16   lr:82.10%        knn: 89.80
n: 32   lr:83.80%        knn: 92.00
n: 64   lr:82.60%        knn: 90.20
n: 128  lr:78.20%        knn: 87.70
n: 200  lr:74.90%        knn: 84.30
n: 300  lr:70.50%        knn: 81.90
n: 400  lr:69.60%        knn: 81.20
n: 500  lr:71.50%        knn: 80.50
n: 600  lr:71.50%        knn: 79.30
n: 700  lr:71.20%        knn: 78.00
```

The results reveal that both estimators perform best with only 32 principal components. In this case, both estimators perform better than without PCA.

The inverse transform $\hat{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}'$ is implemented by `PCA.inverse_transform()`.

5.3.5 Supervised Dimensionality Reduction: NCA

The transformation obtained by principal component analysis is well suited to find the directions in which the data varies most, i. e., the directions that are best suited

to describe the data. However, this procedure does not involve the class labels. In contrast, the *Neighbourhood Component Analysis* (NCA) tries to find a linear transformation into a subspace that maps instances from the same class next to each other and keeps instances from distinct classes apart.

The concept of NCA is based on the idea of a *stochastic nearest neighbour* classifier. This classifier is related to a k-NN estimator, but chooses the class label of a near but *random* neighbour in the training set. In the context of NCA, this classifier operates on feature vectors $\boldsymbol{x}'$ that are projected onto a lower dimensional subspace by

$$\boldsymbol{x}' = \boldsymbol{A}\boldsymbol{x} \tag{5.20}$$

The matrix $\boldsymbol{A}$ is learned by the NCA so that the classification accuracy in the subspace is maximized. As a result, instances of the same class are close to each other in the subspace. The idea of the stochastic nearest neighbour classifier is only considered in order to obtain a differentiable cost criterion $\mathcal{C}(\boldsymbol{A})$. In `sklearn`, NCA is only used as a transformer that maps a vector $\boldsymbol{x}$ onto a vector $\boldsymbol{x}'$ that contains only features that are useful for the classification task.

```
from sklearn.neighbors import NeighborhoodComponentsAnalysis as NCA

all_n = np.array([16, 32, 64, 128, 256, 512])
acc_lr = np.zeros_like(all_n)

for it, n_NCA in enumerate(all_n):
    nca = NCA(n_components=n_NCA)

    nca.fit(x_train_scale, t_train)
    x_test_nca = nca.transform(x_test_scale)
    x_train_nca = nca.transform(x_train_scale)

    lr = LogisticRegression(max_iter=1000)
    lr.fit(x_train_nca, t_train)
    acc_lr = 100*accuracy_score(t_test, lr.predict(x_test_nca))

    knn = KNeighborsClassifier()
    knn.fit(x_train_nca, t_train)
    acc_knn = 100*accuracy_score(t_test, knn.predict(x_test_nca))

    print("n: {}\tlr:{:4.2f}%\tknn:{:4.2f}".format(n_NCA, acc_lr, acc_knn))
```

We obtain:

```
n: 16   lr:79.40%       knn:87.80
n: 32   lr:82.20%       knn:90.60
n: 64   lr:83.10%       knn:90.70
n: 128  lr:78.30%       knn:88.50
n: 256  lr:73.70%       knn:86.60
n: 512  lr:72.20%       knn:85.90
```

Like with PCA, the best result is obtained with 32 components. Since NCA is a supervised technique and based on gradient descent, it may overfit. As a result, PCA performs better in this example.

5.4 Model Selection

5.4.1 Pipelines and Transformer

In the last section, a few objects from the `sklearn` library were introduced. Most of them had the purpose to transform the data. All of these transformers have in common that they implement `fit(X, y)`, `transform(X)` and `fit_transform(X, y)`. `fit` can either learn a relationship between the training data and the targets in a supervised manner (as in `SelectKBest`) or perform an unsupervised transformation on the data regardless of any targets (as in `PCA` or `StandardScaler`). Again, it is important to note that `fit` should never be called on test data. Once these transformers were fitted, we can use `transform` to transform both seen and unseen data. In the case of the PCA-Transformer, this command would project the data onto the principal components. In the case of the `SelectKBest` it would remove useless features. In many cases it is handy to call `fit_transform` on the training data, which performs both sequentially.

In contrast, the `LogisticRegression` object is not a transformer but a classifier. It also implements `fit`, but instead of a `transform` function it has a `predict` function that outputs class estimates based on the input data. Alternatively, we can call `predict_posteriors` to get the posterior probabilities of all classes.

In the previous examples, we used several transformers and one estimator, each of which needed to be fitted separately. In order to get a class prediction on an unseen test sample, we needed to transform the sample with all transformers and pass the result to the estimator. However, `sklearn` offers a more convenient way, called `Pipeline`. The construction of the `Pipeline` needs a list of tuples, where the first entry in each tuple is a string and the second entry is the transformer or estimator object. By calling `Pipeline.fit`, all objects in the `Pipeline` are fitted at once.

```
from sklearn.pipeline import Pipeline

pipe = Pipeline(steps=[('scaler', StandardScaler()),
                       ('pca', PCA(n_components=32)),
                      ('estimator', LogisticRegression(C=1E-3, max_iter=1000))
                       ])
pipe.fit(x_train, t_train)
```

The fitted `Pipeline` can be used like any other estimator.

```
t_test_pred_pipe = pipe.predict(x_test)
acc_pipe = accuracy_score(t_test, t_test_pred_pipe)
print("Accuracy: {:4.2f} %".format(100*acc_pipe))
```

```
Accuracy: 83.90 %
```

5.4.2 Grid Search and Cross Validation

Grid Search

In the last examples, some parameters were studied, e.g. the regularization strength of the classifier, while the remaining parameters were kept constant. In most cases, we do not know beforehand whether the parameters depend on each other. Maybe a high number of principal components needs a stronger regularization, who can tell? The solution to this problem is the *grid-search* approach, where all combinations of possible parameters are investigated. A drawback of this approach is its complexity. Imagine we want to investigate 6 parameters, each of which can have 5 values this results in $5^6 = 15625$ combinations. If the training takes 8 seconds on average, the grid search will take more than 17 hours. If there are only 5 parameters, that can take 4 values, the search will only take 1 hour and 20 minutes.

Validation Data

When performing a grid search, it is not recommended to use the test data to asses, which model is the best. If you did so, the test data would still influence the fitting process of your model. Instead, the training data is split into training and *validation* data. The validation data is not used to train the model, but only to asses the training progress and to select the best model. Since this decreases the amount of available training data, a technique called *k-fold-cross-validation* is often used. The training set is divided in mostly 3 to 5 subsets and each subset is once left out and used as validation data. When a best configuration was found using grid search, the estimator is trained again, using the best hyper parameters and the entire training data.

In the following example, the number of principal components and the regularization strength are examined. The `GridSearchCV` object combines Grid Search and Cross Validation. The parameter grid is a dictionary, where each key consists of the transformer (or estimator) identifier and a list of the arrays to inspect, separated by a double underscore.

```
from sklearn.model_selection import GridSearchCV

parameter_grid = {
 'pca__n_components': [30, 40, 50, 60],
 'estimator__C': np.logspace(-5, 0, 6)
 }

search = GridSearchCV(estimator=pipe, param_grid=parameter_grid, cv=5,
↪  verbose=True)
search.fit(x_train, t_train)
```

After a few minutes the search is completed and we can investigate the best results:

```
>>> search.best_params_
{'estimator__C': 0.1, 'pca__n_components': 40}
>>> search.best_score_
0.8651666666666668
```

Note that for each parameter configuration the score is computed as the average over all 5 validation splits. Here, the best score is 86.5%. For comparability, the accuracy should be computed on the test data. The fitted grid search object can be treated as an estimator.

```
t_test_pred_grid = search.predict(x_test)
acc_grid = accuracy_score(t_test, t_test_pred_grid)
print("Accuracy: {:4.2f} %".format(100*acc_grid))
```

We obtain:

```
Accuracy: 84.70 %
```

Parallelization

In order to decrease the training time, the grid search can be parallelized on multiple CPU cores. This is done by passing an `n_jobs` argument to `fit`. When `n_jobs=-1` is passed, all available cores are used.

Caching

If no action is taken, every transformer in the pipeline is fitted again in each iteration. This means that every transformer is fitted several times with the same input data and the same parameters. A significant amount of time can be saved when the fitted transformer objects are stored, especially when their training is time consuming (e. g. `NCA`, `SelectKBest`). The estimators can be stored by passing a `memory` argument to the pipeline, or by modifying its `memory` attribute directly. It has to be a string, containing a relative path where the cached objects can be stored.

```
pipe = Pipeline(steps=[('sel', SelectKBest(mutual_info_classif))
                       ('estimator', KNeighborsClassifier())
                      ],
                memory='.temp/')
```

When this Pipeline is fitted using grid search, the fitted `SelectKBest` transformers are stored in the folder `'.temp/'`. You can as well choose any other name.

Other ways to decrease training time is `RandomizedSearchCV` which investigates only a representative but random subset of the parameter grid. In addition, there is the `HalvingGridSearchCV`, which trains all configurations on a small subset of the training data and then continues with the best 50% of combinations and more data.

5.4.3 Evaluation

Apart from the accuracy score, we can evaluate our model using the confusion matrix. It shows, which classes are classified correctly and which tend to be confused. The element m_{ij} is equal to the probability that class j was estimated when class i was known to be true.

$$m_{ij} = p(\hat{t} = C_j | t = C_i)$$

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
M = confusion_matrix(t_test, y_test_pred_grid, normalize='true')
```

When the matrix is to be visualized, the following shortcut exists:

```
from sklearn.metrics import plot_confusion_matrix
plot_confusion_matrix(search, x_test, t_test, values_format='.2f',
↪  normalize='true')
```

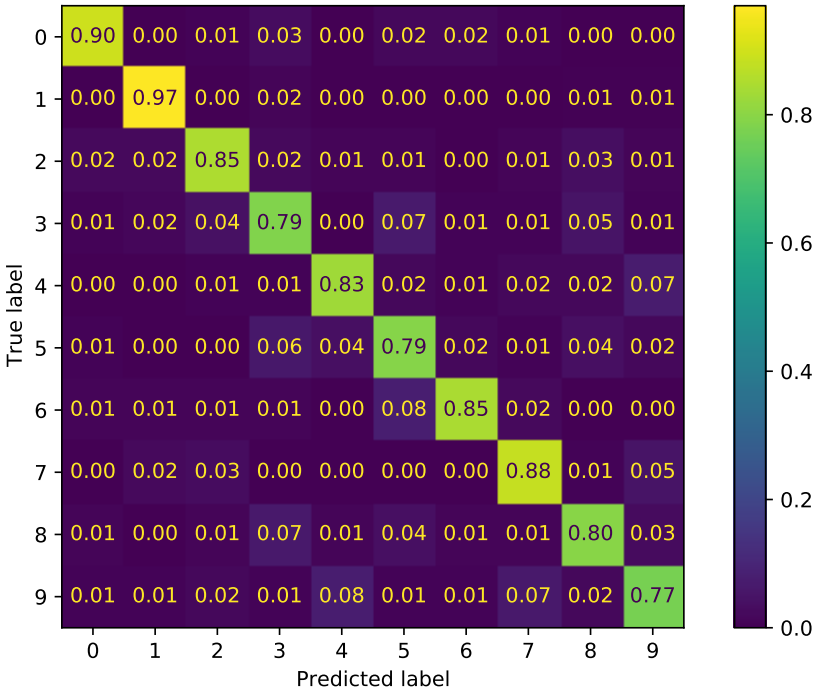


Figure 5.7: The confusion matrix for the digit classification problem

5.5 Feature Selection for Speech Data

This section presents the most commonly used features for speech processing in machine learning.

5.5.1 Log Mel Spectrogram

From Lab Course 3 you know the short time Fourier transform, which is a discrete Fourier transform of short, possibly overlapping segments of an audio signal. For the use in speech recognition, this representation is impractical, since the most information is contained in a few low frequency bins. As a consequence, the spectra

for each frame are mapped onto the *mel scale*. This is a logarithmic frequency scale which considers the human perception of pitch. One popular formula to convert a frequency f in Hz into the perceived pitch m in mel is

$$\frac{m}{\text{mel}} = 2595 \log_{10}\left(1 + \frac{f}{700\,\text{Hz}}\right) \tag{5.21}$$

The mapping from frequency bins to mel bins is usually performed with triangular windows. When $\boldsymbol{s}_{\text{ft}}$ is a column vector containing the magnitude spectrum of one signal frame and $\boldsymbol{w}_i$ is a row vector containing the window for the i^{th} mel component, the component can be computed as

$$m_i = \boldsymbol{w}_i \boldsymbol{s}_{\text{ft}} \tag{5.22}$$

When all windows are given by one transformation matrix $\boldsymbol{W} = [\boldsymbol{w}_0, \boldsymbol{w}_1, \ldots \boldsymbol{w}_{n_{\text{mel}}-1}]^{\text{T}}$ the transformation can be notated as

$$\boldsymbol{s}_{\text{mel}} = \boldsymbol{W} \boldsymbol{s}_{\text{ft}} \tag{5.23}$$

where $\boldsymbol{s}_{\text{mel}}$ is the mel scale representation of the linear spectrum $\boldsymbol{s}_{\text{ft}}$. This matrix can be obtained by the function `librosa.filters.mel`. The mel spectrogram can also be generated from a time series or an absolute spectrum. This is done with the function `librosa.feature.melspectrogram`. When the input is a time series, you need to provide the arguments that specify how to generate the magnitude spectrogram. As an example, we use the first seconds of the speech sample from lab course 3.

```
s, sr = librosa.load('spam.wav', sr=None)
s = s[:55000]
plt.plot(s)
s_mel = librosa.feature.melspectrogram(y=s, sr=sr, n_mels=128, win_length=320,
↪  n_fft=1024, hop_length=80)
```

Instead of 320 frequency bins in the magnitude spectrum, the mel spectrogram has only 128 bins.

In most of its functions, librosa defaults a sampling frequency of 22.5 kHz which is only rarely used in speech processing. If you do not specify the true sampling frequency, it might happen that the wrong rate is used without notice. The function `librosa.load` even resamples the input data to 22.5 kHz unless you specify `sr=None`.

One drawback of the mel spectrogram is that it contains a few components with high values while most values are close to zero. For many estimators this can be detrimental. A common solution is to compute the logarithm of the spectrogram. This transformation can be motivated well by the human perception of sound pressure. The function `librosa.display.specshow` is useful to visualize spectrograms. It needs the hop length and the sampling rate in order to display the right axis descriptions.

```
import numpy as np
import librosa.display
log_mel = 20*np.log10(s_mel)
librosa.display.specshow(log_mel,  sr=sr, hop_length=80,
                         x_axis='time', y_axis='mel', fmax=8000)
plt.colorbar()
```

In Fig. 5.8, it is observable that the lower frequency components take most place in the mel spectrogram, while the higher frequency components are mapped to a small range.

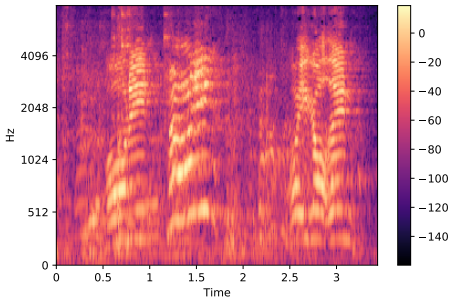


Figure 5.8: A mel spectrogram

5.5.2 Mel-Frequency Cepstrum

In Fig. 5.8, it can be seen that many phonemes consist of a fundamental tone and a series of its harmonics. In the section from 2.5 s to 3.3 s it can be seen that this spike train is spectrally shaped, depending on the current vowel. This effect can be explained with the speech generation in the vocal tract. The following explanation covers the bare minimum to motivate the Mel-Frequency Cepstrum. For a detailed explanation of speech production refer to the literature [3]

For voiced sounds, the vocal cords produce a periodic excitation signal which determines the pitch and creates the visible series of harmonics. This excitation signal then resonates in the vocal tract. Depending on the position of the vocal tract, frequency ranges are amplified or attenuated. This can be described by means of a filter operation, so that for voiced sounds, the signal at the mouth can be described by

$$s(t) = u(t) * h(t) \tag{5.24}$$

where $u(t)$ is the periodic exitation signal at the glottis and $h(t)$ is the impulse response of the vocal tract filter. After analog digital conversion, a (mel) frequency representation $S(m)$ of $s(t)$ would be obtained by

$$S(m) = U(m) \cdot H(m) \tag{5.25}$$

[3] Cha. 2, Vary P., Martin R. (2006) *Digital Speech Transmission: Enhancement, Coding and Error Concealment*, Wiley. The book is accessible online for RWTH students.

where $U(m)$ and $H(m)$ are defined similarly. From the convolution theorem (Tab. 3.1) it follows that in a spectral representation the convolution is transformed into a multiplication. When the logarithm is computed, like we did for the log mel spectrum, we obtain

$$\log|S(\mu)| = \log|U(\mu)| + \log|H(\mu)| \tag{5.26}$$

This means that the log mel spectrum can be decomposed into a excitation component and a filter component. Along the frequency axis, the excitation component (i. e. the spike train) varies fast, while the filter component (the spectral envelope) varies slowly. Therefore, they can be separated from each by a frequency analysis of the log spectrum. The resulting representation is called *cepstrum* and its independent variable is called *quefrency*. Components with a low quefrency correspond to the slow variations from the vocal tract filter. At high quefrencies, a peak can be observed, which is caused by the harmonic excitation. When the pitch frequency is higher, this peak is shifted to lower quefrencies. The components of the cepstrum are called *Mel-Frequency Cepstral Coefficients* (MFCCs).

A routine to compute MFCCs is provided by `librosa`. They can be created from the time series or from the log mel spectrum. In the first case, all parameters have to be specified that are neccessary for the generation of the spectrogram and the mel scale. In the latter case, only the number of coefficients has to be specified. For instance, `n_mfcc=22` returns the first 22 coefficients (i. e. the components with the lowest quefrencies). It is useful to remove the mean value from the log mel spectrum. Otherwise, the first cepstral coefficient has a high absolute value, compared to the rest of the cepstrum.

```
mfcc = librosa.feature.mfcc(S=log_mel-np.mean(log_mel), n_mfcc=40)
librosa.display.specshow(mfcc, sr=sr, hop_length=80, x_axis='time')
plt.colorbar()
```

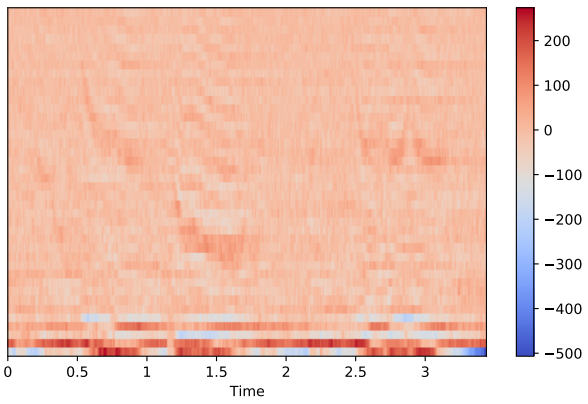


Figure 5.9: A mel-frequency cepstrum

In the output in Fig. 5.9, it can be seen that, as the pitch in the mel spectrum increases, the corresponding peak in the cepstrum moves to lower quefrencies. Furthermore, strong variations in the first five coefficients can be observed, which depend on the current vowel.

5.5.3 Other Features

This section briefly presents other features that are used in machine learning for speech recognition.

Linear Predictive Coding

This technique was originally developed for speech transmission but has useful properties for speech recognition. The vocal tract filter can be modeled by an by an autoregressive (AR) filter. This means that, in the model, one sample from the speech signal is influenced by the last L samples, where L is the length of the filter model. In turn, the current sample can be predicted by the last L samples. To do so, an analysis filter is fitted to small stationary segments of the signal. The filter coefficients depend on the position of the vocal tract. They can be obtained by the function `librosa.lpc(y, n)` where `y` is the speech segment and `n` is the length of the analysis filter.

Zero Crossing Rate

This quantity can be used to distinguish between harmonic and percussive sound. It is defined as the number of zero crossings in a signal segment, i. e. the number of samples that have a different sign than the previous sample. The zero crossing rate is low for voiced sounds and high for unvoiced sounds. It can be computed by `librosa.feature.zero_crossing_rate(y, frame_length, hop_length)`. The arguments `frame_length` and `hop_length` define how the signal `y` is divided into segments.

Spectral Flatness

This quantity is close to one, when a signal segment has a flat magnitude spectrum, as is the case for many unvoiced sounds. Otherwise it is close to zero. It can be computed by `librosa.feature.spectral_flatness`. The function either receives a magnitude spectrum as parameter `S`, or a time signal `y` together with the parameters to create the spectrum.

Spectral Centroid and Bandwidth

These features determine, where the energy in a signal frame is located. If the magnitude spectrum was a probability density function, these values would correspond to mean and standard deviation. When S is the discrete spectrum of a signal frame and f is a mapping from bins to frequencies, they are computed by

$$centroid = \frac{\sum_i |S(i)| \cdot f(i)}{\sum_j |S(j)|} \tag{5.27}$$

$$bandwidth = \left(\sum_i |S(i)| (f(i) - centroid)^p \right)^{\frac{1}{p}} \tag{5.28}$$

In python they can be computed by `librosa.feature.spectral_centroid` and `librosa.feature.spectral_bandwidth`. Both can either receive a time series or a magnitude spectrum. The function for the spectral bandwidth also receives an argument `p`.

5.6 Exercises

In the introduction to this lab, you learned about the MNIST dataset, consisting of pictures of written digits. In this task, we will cover a dataset consisting of the *spoken* digits from 0 to 9. The digits are taken from a larger data set, that contains many more spoken keywords like 'On', 'Off', 'Go', 'Stop', 'Wow' and many more. [4] , [5]

The code in this lab is divided into two files, namely one library file `lib_digit.py` and a jupyter notebook `solution_digit.ipynb` . The library contains a framework that does the preprocessing for you and the notebook should contain your solution.

Before you start, download the training data from the link that is specified in moodle. The size of the data set is 70 MB per digit. If you have limited memory or internet connectivity, it is sufficient to load the folders `four`, `five`, `six`, `seven`, and the text files. Otherwise, we recommend to load all digits if you intend to do experiments that go beyond the lab. After lab course 7, you can use it for deep learning experiments as well.

Exercise 5.1 Data Acquisition

First, we need to prepare the wav files so that they are suited for a machine learning application.

- Familiarize yourself with the functions `load` and `wav2instance` in the library file. Figure out, how the training instances are organized. Look up unknown functions.
- Load the training data for the digits `['four', 'five', 'six', 'seven']`.
- Select one instance from the training set and display the dimensions of all contained features by iterating over the keys.

In the library file you find a `FeatureSelector` that allows you to select the desired features from the training instances. It follows the `sklearn` API, meaning that the data is transformed by `FeatureSelector.transform()` and that it can be used in a `Pipeline`.

- Familiarize yourself with the implementation in the library file
- Create a FeatureSelector object and use it to generate a matrix `mel_train` where one row corresponds to the complete flattened log mel spectrum of one training instance.
- How many features does one instance have?
- Use the same object to generate a matrix `mel_test` containing the test data.

Exercise 5.2 Logistic Regression

Now that the data has an appropriate shape, we can use it to train a machine learning model.

- Create a logistic regression object with `max_iter=1000`.

[4] Warden, P. (2018) *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*
[5] `https://www.tensorflow.org/datasets/catalog/speech_commands`

- Train the model with the extracted training data (`mel_train`). Probably you will receive a warning that the training did not converge.

Exercise 5.3 Model Evaluation

With the model being trained, we can analyze its performance.

- Compute and display the accuracy on the test set.
- Create a diagram that shows $\hat{p}(\texttt{'four'}|\boldsymbol{x}^{(n)})$ for all $\boldsymbol{x}$ in the test set `mel_test`. On the x-axis show the sample index n.
- In the same diagram add a line with the true posterior probability $p(\texttt{'four'}|\boldsymbol{x})$. This probability is 1 if it is known that an instance is a `'four'` and zero otherwise.
- In another diagram, generate the same curves for the training set. Is the training data linearly separable?
- Visualize the normalized confusion matrix for the test set. From the output, read which misclassification happens most often.

Exercise 5.4 Pipelines

- Create a `Pipeline` with a `FeatureSelector` and a `LogisticRegression` object. Let the `FeatureSelector` select only the log-mel-spectra and allow 1000 iterations for the logistic regression object. Set the `Pipeline`'s `verbose` attribute to `True` in order to see the fitting time.
- Fit The `Pipeline` with the training data and compute the accuracy score on the test set.
- Then add a `StandardScaler` to the `Pipeline` and compare accuracy and fitting time.
- Repeat these steps but let the `FeatureSelector` select all cepstral coefficients.
- Denote accuracy and fitting times in the table given in the notebook. How do scaler and the reduced number of features impact the pipeline?

Exercise 5.5 Grid Search

In this experiment we investigate how certain parameters influence the estimator's performance by use of the `GridSearchCV` class. For time reasons we do not exploit k-fold cross validation but a fixed validation set. To make use of it, pass `cv=val_split` to all `GridSearchCV` objects that are created in the following.

- Create a `Pipeline` with cepstral coefficiens as input features, standard scaling and logistic regression with a maximum of 1000 iterations. Pass `verbose=False` to the Pipeline.
- Define a parameter grid where you vary the regularization parameter `C` of the estimator in an exponential range from 10^{-5} to 10^{5} and the number of selected cepstral coefficients between 2 and 30 in steps of 4.
- Create and fit a `GridSearchCV` object with the `Pipeline`, the parameter grid and the predefined validation split.

- The provided function `lib.analyze_grid_search` receives a fitted grid search object and visualizes the validation accuracy in dependency of each parameter. Use this function to determine which range of parameters is best suited.
- Refine the grid in this region and again investigate the results.
- Compute and display the accuracy on the test set.

Exercise 5.6 Dimensionality Reduction

- Create a `Pipeline` with a `FeatureSelector` and a `KNeighborsClassifier` (no Standard Scaling). The `FeatureSelector` should select the mel-frequency spectrum. Fit the `Pipeline` and display the accuracy on the test set.
- Now create a second `Pipeline`, where a `NeighborhoodComponentAnalysis` is added before the classifier, and provide a `memory` argument, so that the `Pipeline` is able to cache fitted estimators. The pipeline should receive the argument `verbose=0`.
- Create a parameter grid where you vary `nca__n_components` (20, 50, 100) and `knn__n_neighbors` (1, 2, 5, 10, 20, 50, 100).
- Create a `GridSearchCV` with the `Pipeline` and the parameter grid. The grid search should have the argument `verbose=3`.
- Fit the grid search and investigate the fitting time for the parameter configurations. How can you observe the effect of caching? You may receive a `joblib warning` that can be ignored.

Exercise 5.7 How to go on? (Optional)

- If you like, you can reload the data set with all ten digits from zero to nine.
- The `sklearn` API supports many more estimators and transformers. You can try support vector machines, shallow neural networks, and many more.
- The function `wav2inst` computes several signal features and appends them to a feature dictionary for each instance. You can easily add more signal features. The corresponding keys must be added to the `FeatureSelector` class. Check if the additional features improve logistic regression, by means of `GridSearchCV`.

Exercise 5.8 The Ultimate Test (Optional)

The function `lib.test_utter` receives an estimator and then starts a recording for two seconds. The recording is preprocessed like the training data and presented to the estimator. The recording and the predicted posteriors (if applicable) will be displayed.

- Retrain your favourite estimator that you developed during this lab course and use this function to test it.