Introduction to Experimentation in Physics

Freek Pols

December 30, 2024

Contents

0.1 Introduction

It doesn't matter how beautiful your theory is.
It doesn't matter how smart you are.
If it doesn't agree with experiment, it is wrong.

Richard P. Feynman

There are several reasons physicists carry out experiments: to develop new knowledge, like the discovery and usage of radioactivity; to test hypotheses and theories, think of the discovery and proof of Majorana particles some 75 years after the prediction of its existence; and to explore and determine features of various systems.

One cannot simply carry out an experiment and call it a scientific experiment. There are many scientific rules that must be adhered to. One must put a lot of work and effort into the experiment and in writing the report before others (peers, other experts in the field) can be convinced that what was found is reliable and valid. Doing the work thoroughly will also help to make sure that the values of quantities can be determined accurately.

The goal of the course "Introduction to Experimentation" is to give you a first glimpse of what it means to do experiments in physics, and to demonstrate the rules that must be applied when doing such experiments. You will work with equipment that is often used by physicists in labs. You will learn how to collect, process, analyse and present experimental data. Explicitly, after following this course you are able to:

- understand what is required before you can start collecting data in a scientific, physics experiment
- collect, analyse and present empirical data for the purpose of scientific research
- write a concise report on a physics experiment

Rather than using extensive lectures, we provide all necessary information in this manual and make use of short movie clips in each of the chapters (click the movie symbol). This allows more freedom and does not require you to participate in aspects in lectures where the content is already familiar to you.

If you have ideas for improvement, let us know! You can make use of the Issue button at the top right corner.

I hope you will enjoy the physical experiments we have designed for you.

Freek Pols
Coordinator Physics Lab Courses

0.1.1 Rules and regulations

This introductory laboratory course provides you with your first experience in conducting physics experiments. This includes taking measurements, processing them, and recording the results in a short paper.

Course content & Final grade

The course consist of an introduction to **measurement and uncertainty** combined with an introduction to **Python** and **two experiments**. For each experiment, you will write a paper. You are allowed to work together, however, your partner for the second experiment should be different from the person you worked with during the first experiment. This is due to the fact that an important aspect of the minor is that you will meet new people with different (educational) backgrounds. We want to stimulate this by forming new teams.

The final grade for the course is calculated by:

$$\text{Grade} = 0.2 \cdot \text{M\&U} + 0.4 \cdot \text{IE}_1 + 0.4 \cdot \text{IE}_2 \tag{1}$$

If your grade for IE_2 is lower than a 5.5, you will be allowed to hand in a second version. The grade for this second version will not be higher than a 6.0, but in some cases that will be enough to allow you to pass the course.

What	How	Teamwork	Percentage
M&U	online test	individual	20%
IE_1	paper	student pair	40%
IE_2	paper	student pair	40%

Measurement and uncertainty Measurement and uncertainty is a chapter in this manual which you will have to read. There will be a number of exercises, some of which will involve the use of Python. These exercises will prepare you for the test. There is an afternoon scheduled where a TA is available for help if you have questions about measurement and uncertainty and the use of Python. However, you are allowed to work at home and consult the TA using MS Teams.

Your knowledge of measurement and uncertainty will be assessed in an online test using Jupyter Notebooks. You are allowed to bring this manual, premade scripts, notes etc. ,in other words, it is an open book exam. You are not allowed to consult others during the online test.

Experiment 1 The first experiment is the same for everyone and involves determining the gravitational acceleration g within 0.5%. You and your lab-partner ought to determine the fourth significant number of the gravitational acceleration. This experiment will help you get acquainted with experiments in physics. During two pre-scheduled afternoons you will work on a scientific paper using the RevTex-template. You are allowed to work at home, but it is mandatory to at least show a draft version. This will help you to adjust major mistakes. Such a paper is different in many elements from reports in other engineering subjects. The paper will be assessed.

Experiment 2 The second experiment will be done pairwise. There are five experiments in total of which one is to be chosen. You should enroll yourselves at BrightSpace for the preferred experiment. Note that we have only a limited number of equipment available and some of the groups will be quickly filled.

Please prepare the experiments for which you enrolled. This saves valuable time during the scheduled hours. Preparation means that you read the chapter, carry out the assignments and start working on labjournal.

Determining the half-life of K-40

Two different methods are used to determine the half-life of K-40. The literature value is used to determine which method yields the best value. However, one should consider the trade offs as well!

Spectral lines of Sodium or Mercury

Spectral lines can be determined to identify atoms and molecules of distant stars. In this experiment you will determine the spectral lines of Sodium or Mercury using spectroscopy.

Boltzmann constant and electronic instruments

This experiment consist of two parts. You will determine the characteristics of a low pass filter and subsequently determine the Boltzmann constant (or vice versa). In the first part of the experiment you will use an oscilloscope and in the second part a digital multi meter. The experiments focus on using accurate instruments often used in physics labs (not allowed for students with a background in electronics).

Waterdroplet

In this experiment you will investigate two aspects of a waterdroplet: the size and shape of waterdroplet and the drag coefficient.

Surface tension

In this experiment you will investigate the surface tension of water. You are free to either compare two different methods or investigate how a factor of interest (for instance temperature or sugar concentration) changes the surface tension.

Millikan

Schedule and deadlines

When you enroll in the Introductory Laboratory Course you will receive a personal timetable. This timetable shows, amongst other things, the date of the lab safety test, the time and place of the experiments and the deadlines. If you fail the online safety test, you will not be allowed to take part in the rest of the course.

The practical sessions in the IE block cover the following topics: Python, IE_1, writing a lab report, and IE_2.

Week	Topic	Deadline	Time required	Mandatory
1.1	Introduction	-	2 + 2 (reading)	V
1.2	M&U	-	4 (assignments)	X
1.3	M&U	-	4 (assignments)	X
2.1	M&U test	Test	4	V
2.2	Experiment 1	-	4+2 (preparation)	V
2.3	Report writing	Hand in draft for feedback	4	X
3.1	Report writing	Hand in report	4+2 (finishing report)	V
3.2	IE_2	-	4+2 (assignments)	V
3.3	IE_2	-	4	V
4.1	x	-	-	X
4.2	Report writing	-	4	X
4.3	Report writing	27 sept hand in report	4 + 2 (finishing report)	X

: Time schedule with deadlines and expected hours of homework.

Rules of attendance

To ensure that all students can complete the Introductory Laboratory Course in time, it is essential that everybody does what is asked of them. This has proved difficult for some students. This has forced us to make use of a penalty system. When penalties are imposed, they can have serious consequences: they could result in you failing the course and therefore failing the entire minor. (Below, the sanctions

are indicated in italics.) For this reason, we strongly advise you to take note of the penalties for infractions like late arrival and do all you can to avoid them.

Illness, etc. As stated earlier, the smooth running of the Introductory Laboratory Course relies upon your attending the sessions listed in your personal timetable. If you realize in advance that you will be absolutely unable upon your attending a particular session, contact the course office as soon as possible to try to arrange an alternative time and date. This is always subject to the availability of places, and you must have a legitimate reason for rescheduling (a holiday, for instance, is not acceptable).

By notifying us promptly, you may also enable the course office to reallocate your place to another student in a similar situation.

If you find yourself unable to attend at the last minute – due to illness, for example – you must inform the course office no later than 8.30am on the day of a missed morning session and no later than 1.30pm on the day of a missed afternoon session. To notify us, call **015 278 5886** or **015 278 2459** or e-mail adminnp-tnw@tudelft.nl.

You will need to arrange an alternative time and date as soon as possible. If you fail to do so, you will not be able to continue the course.

You can cancel and reschedule attendance at lab sessions up to three times. If you try to do so a fourth time, you are considered to have fallen so far behind that you will be unable to pass the course in the current academic year and so are excluded from it.

Absences without notification If you fail to attend a lab session or experiment without giving advance notice or informing us on time with a valid reason, you must report in person to the course office as soon as possible to arrange a new time and date.

In this case, one full point is deducted from your final grade for the course. Only one unannounced absence of this kind is permitted; if it happens a second time, you will be excluded from the course.

Lateness Lab sessions begin promptly at either 8.45am or 1.45pm.

1. If you arrive less than 15 minutes late, you are permitted to join the session.
2. If you arrive more than 15 minutes late, you will not be permitted to the session and must report in person to the course office as soon as possible to arrange a new time and date. This due to safety issues and instructions that are necessary to know how instruments work and what is expected of you during the experiments.

The worst case scenario is that a full point is deducted from that specific experiment. Whether this penalty is applied, will be discussed with the practical coordinator.

Handing in your work

The course includes a number of assignments, in the form of lab concise papers of the experiments you have conducted, that need to be handed in. As stated, you are allowed to work in pairs but have to switch partner for the second experiment.

Your paper must be handed in using Brightspace, where it is grade using a rubric. This will be scanned for plagiarism using TurnItIn. You ought to submit your digital lab notebook using Vocareum.

The deadlines for submitting both versions (and, if required, improved rewrites) are set by the course administrators. In exceptional circumstances, you may be granted an extension of up to one week; to obtain one, you must apply at least one day before the original deadline and in person or by e-mail (not by telephone), with valid reasons.

If your second lab paper is not good enough, you will be asked to submit an improved rewrite by a new deadline. In this case, you must also provide the original version again. Without that, your revised paper will not be assessed.

If you fail to hand in and upload a paper on time, it will be marked 1 out of 10. Once you have missed the deadline, there is no point in submitting the paper at all as it will not be assessed anyway.

Fraud and plagiarism To check that you have not copied somebody else's work, or parts of it, your paper is scanned for plagiarism. If one or more passages are found to be identical to another paper, that is treated as fraud and the Board of Examiners is informed. The board has the power to impose very severe penalties, not least excluding you from the course.

Software

You need to install Arduino. Furthermore, you will use LaTeX. A simple way to use LaTeX is using Overleaf.

Contact

For advice, or to make complaints, please contact:
dr. ir. C. F. J. Pols
room A005
e-mail c.f.j.pols@tudelft.nl.

0.1.2 Lab journal

Before each experiment, you will do assignments to help you prepare for and understand the experiment. During the experiment, you will collect data, write down settings of the equipment, make notes, etc. All this information should be clearly organized in a single document, your lab journal. If you have used your lab journal considerately, another researcher should be able to finish the experiment you started. Another researcher should also be able to assess your work using your lab journal as well. Keeping up a lab journal is thus an important part of scientific research and lab work. As we value it so high, as it is vital to establish the reliability and validity of a study, we only grade your report when your lab journal is handed in through Vocareum.

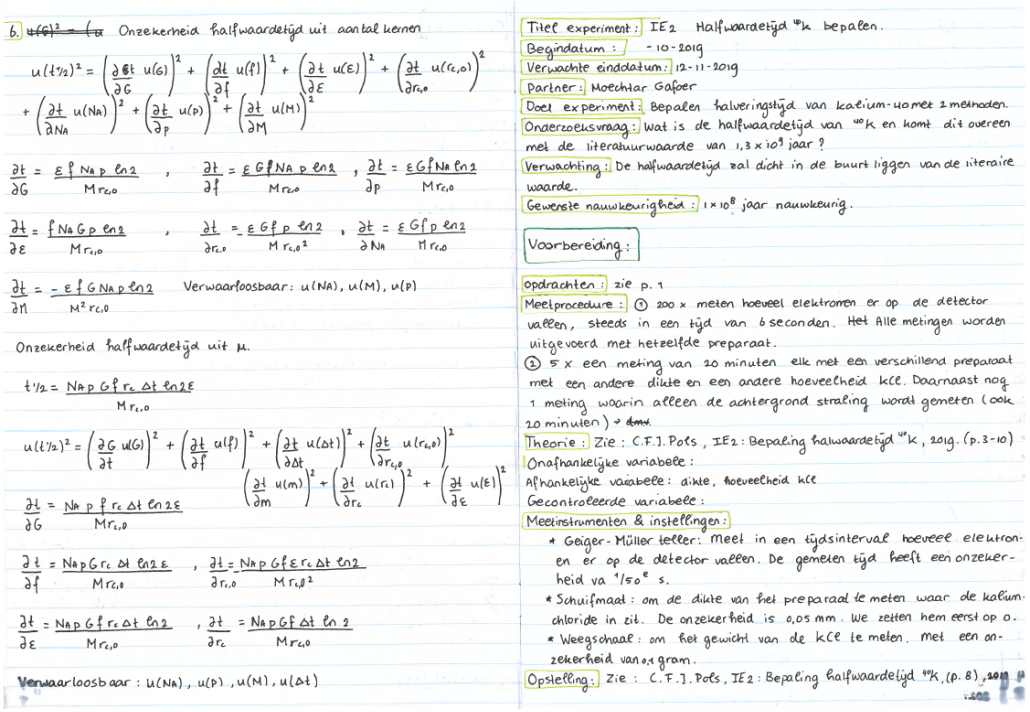


Figure 1: A scan of a lab journal from a first year physics student.

Function

Your lab journal is an archive for ideas, considerations, methods, measurements, analyses etc. Everything that is or could be important for the study should be written down in the lab journal. Unpublished experiments and materials can be published after two years on the basis of your lab journal.

Keeping track of your work in a clear, well-structured manner, making your code and analysis available and readable for others, is not easy. The following structure might help you in keeping track of your work. However, not all fields can be applied to each and every experiment. There might be fields missing as well. You can and may change the lab journal structures to better meet your demands.

Structure

The structure of the lab journal is very similar to that of a scientific report. It covers the following issues:

General

- Title of the experiment: Write down a concise, comprehensive title.
- Start date: Write the starting date of your experiment.
- Expected end date: Write down the expected end date of the experiment.
- Partner: If applicable, write the name of your partner.

- Goal: Write down the aim of the experiment.
- Research question: If applicable, write down the research question. If a prescribed experiment does not have a concrete research question, write a more elaborate goal.
- Expectations: Write down the expected outcome.
- Desired accuracy: How accurate do you want the outcome of the experiment to be, e.g. difference between result and literature value.

Preparation

- Task: Write down the tasks involved in the prescribed experiment.
- Theory: Write down the theory that is required to understand and carry out the experiment, in a few very short sentences.
- Method: Write down, in general terms, what the experiment is like. A more precise description of the single steps is noted in the procedure.
- Independent variable: The variable you change.
- Dependent variable: The variable that you want to measure.
- Controlled variable: Variables that might influence the outcomes of the experiment and thus have to be kept constant (as possible).
- Instruments & settings: The instruments used in the experiment, and their settings. These settings may change throughout the experiment. Note the accompanied accuracy of the instruments for those particular settings in this section.
- Procedure: Describe the steps that have to be carried out, or have been carried out, to do the experiment.
- Set up: An image, drawing or reference to the experimental setup.
- Comments: Any comments about things that require specific attention.
- Accuracy: Room for calculations addressing accuracy, measurement uncertainty and error propagation.

Execution

- Table: Write down all measurements in a labeled table. Values that do not change should not be inside a table.
- Observations: Observations that are not quantitative in nature *the light bulb started to glow, producing visible light for the first time.*
- Comments: Further comments that might be important for the collection of data.

Processing

- Graphs: The graphs following from the data collection and processing (described in the method section).
- Trend: The general trend that can be seen in the data, similarities with expectations following from e.g. theoretical models.
- Analysis: A further analysis of the data
- Calculations: Calculations of your final answers. Additional calculations with regard to measurement uncertainty.
- Comments: Further comments addressing the processing of data, e.g. removing data points.

Evaluation

- Discussion: Discussing your results when the description does not fit within the processing.
- Conclusion: The final conclusion, the scientific claim you draw which can be justified by your data.

Jupyter Notebook

We have made online templates in Vocareum available. For this, we made use of Jupyter Notebook. Jupyter Notebook uses various online languages (Python, LaTeX, HTML, Markdown). Often used codes can be found here. When you start the first experiment, you have to invite your partner so you can work together in the same lab journal. When finished, you can submit your lab journal and it will be graded.

0.2 Data analysis with Python

0.2.1 Introduction to Python for Physicists

These files stem from the FYPLC repository which contain the source code of the material developed for an introduction to Python for physicists that runs as a "minicourse" of 1.5 ECTS as part of the "First Year Physics Lab Course" of the Applied Physics program at TU Delft. The materials ought to introduce you to Python if you are not familiar with programming (at all).

The materials are designed for self-study and introduce one to the basics of python. The notebooks are self-contained, and include an explanation of the concepts, example code to illustrate the concepts, and exercises (with answers at the end) for testing your knowledge.

Notebook	Description	Type
Notebook 1	Python Basics	Core material
Notebook 2	Functions	Core material
Notebook 3	Program flow	Core material
Notebook 4	Scientific computing	Core material
Notebook 5	Data analysis	Core material

the end of the course, you should be familiar with:

- Basic concepts in Python: What is Python and how does it work
- Functions in Python: How to write them and how to use them
- Program flow control: How to control the flow of execution of your code
- Scientific computing in Python: Introduction to the numpy library
- Data in Python: How to load, plot and fit data
- Measurement uncertainty and error propagation: Quantify measurement uncertainties and calculate how these propagate to your final answer.

Each notebook of the first 5 Notebooks start with a pre-post test. If you have some knowledge of programming, you can make the test and see whether you already know the content. If you can make the test without any problems, skip the module. If you are not familiar with programming (in Python), go through the module, do the exercises and finish with the test to see whether you mastered the content. Each notebook includes a list of detailed learning objectives so you know what you should be learning. In addition, there is a "notebook for more advanced programmers", exploring additional programming concepts in python.

Feedback

Did you find a typo? Is there something that is not clear to you? Is there a mistake in the notebooks? We gladly welcome feedback! To give feedback, the easiest for us is for you to submit an "issue" in our repository issue tracker:

Submit an "issue"

In the message, please include:

- The notebook number
- Copy-and-paste the text from the notebook
- Optional: Describe your suggestion (with a typo not needed even)

Developers

The first five notebooks were developed by Gary Steele g.a.steele@tudelft.nl with input and feedback from Jeroen Kalkman J.Kalkman@tudelft.nl and Freek Pols c.f.j.pols@tudelft.nl. These notebooks were then developed further by Freek. All other notebooks were developed by Freek Pols

Contents

Source code

Software

Python is an interpreter programming language and thus needs an interpreter (some software). For programming it is always useful to have some software that helps you in writing code and running it. For this course we recommend you to install Mini-Conda, but provide instructions for other software as well.

VSC A popular code editor is Visual Studio Code. It allows you to program in different languages, where it recognizes the commands in that language and adjusts the FONT so that it becomes better readable. Moreover, it allows you to install various packages (such as Jupyter Notebook). It also integrates GIT and allows to code using Co-Pilot, an AI pair programmer. We advise to use VSC as it allows for multiple programming languages.

Terminal The terminal in Visual Studio Code (VSC) is a tool that lets you interact with your computer's command line directly within the editor. It's used to run commands, scripts, or programs without leaving the coding environment. For example, you can compile code, run a development server, install dependencies, or manage files. It's very helpful for developers because it allows you to code and execute commands in one place, streamlining your workflow.

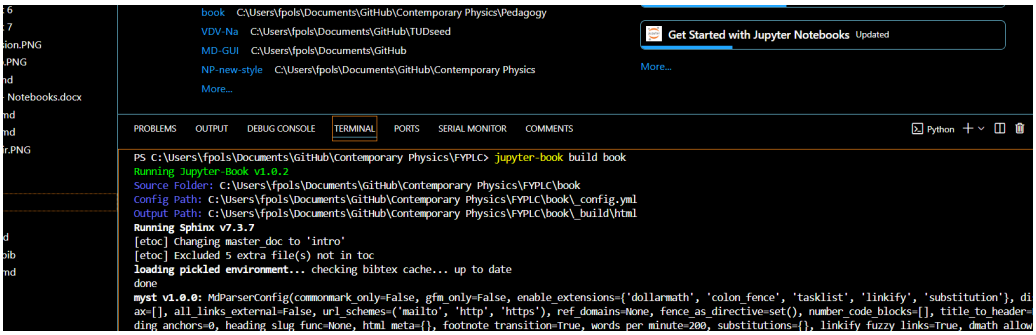


Figure 2: The VSC terminal to interact with the computer using the command line

Install packages If you're working with Python, you can install packages using pip. For example, to install the requests library, run:

```
pip install requirements
```

requirements here refers to the packages that you want to install.
For the data-analysis you at least need to install the following packages:

- numpy
- matplotlib
- scipy

Extensions Extensions in Visual Studio Code (VSC) are powerful add-ons that enhance the functionality of the editor by providing additional features, tools, and support for various programming languages, frameworks, and technologies. Extensions allow you to customize and tailor VSC to suit your specific development needs.

How to Install Extensions:

- Access Extensions View:
  - Click on the Extensions icon in the Activity Bar on the side (or press Ctrl+Shift+X / Cmd+Shift+X).
- Search for Extensions:

  - In the Extensions view, you can search for the name or keywords related to the extension you want to install.
- Install the Extension:
  - Click the Install button on the desired extension, and it will automatically be added to VSC.
- Manage Installed Extensions:
  - You can view, enable, disable, or uninstall extensions from the same Extensions view.

Popular Extensions in VSC:
- Python: Provides linting, debugging, IntelliSense, and more for Python development.
- Jupyter: Provides support for Jupyter notebooks within VSC.
- Arduino: Provides support for programming in Arduino.
- Code Spell Checker: Has a great spelling checker, also available for Dutch
- Github Copilot: Your AI pair programmer. Helps you in writing code.
- LaTeX workshop: LaTeX coding, preview, compiling.
- MyST-Markdown: The official Markdown syntax extension

Anaconda & MiniConda **Anaconda** and **Miniconda** are both distributions of the Python and R programming languages, primarily aimed at simplifying package management and deployment for data science, machine learning, and scientific computing. The main difference between these distributions is the amount of disk space required (~4GB for Anaconda, 50 MB for Miniconda). Where Anaconda has already installed all main packages for you, you control with Miniconda precisely what you want to install. Hence, Miniconda is a minimal installer for the Anaconda distribution. It only includes the `conda` package manager (to install packages) and Python, along with their dependencies. It does not come with any pre-installed packages except for these essentials. That also means that it lacks a graphical user interface which is available with Anaconda.

Install Anaconda If you want to install Anaconda, see their website. Once installed, you can open Anaconda and choose your *interactive development environments* (IDE), the interface in which you will write your code. We most often use Jupyter Notebook or Jupyter Lab.

Install Miniconda We recommend, however, to install Miniconda since it gives you a better idea how your computer works, have more control of what is being installed and takes far less disk space. To install Miniconda, download and run the .exe file as described on their website

> **Warning**
> There are multiple download options at the site of conda. Be sure you download miniconda. Furthermore, wait with continuing the instructions below until the miniconda is fully installed.

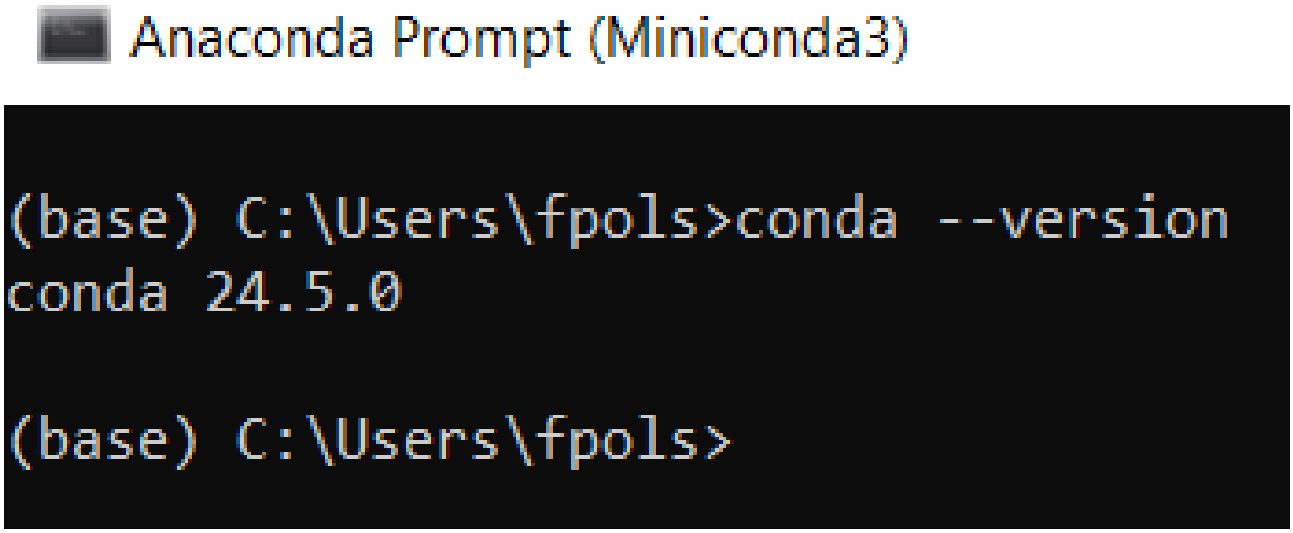
Once installed, open the anaconda prompt. This opens a terminal (a text-based interface used to interact with the operating system by executing commands, scripts, or programs). To check whether the installation is correct and which version is installed, type:

```
conda - -version
```

and press enter, the terminal will return the conda version.

Since you installed the minimal installation, we need to install the IDE's we want and the packages that we need. We, at least, need Jupyter, numpy, matplotlib and scipy. A popular library that might come in handy is Pandas.

To install the IDE Jupyter and Jupyter lab, run the commands:




```
conda install anaconda::jupyter
```

```
conda install -c conda -forge jupyterlab
```

For the packages, run the commands:

```
conda install conda -forge::matplotlib
```

```
conda install conda -forge::numpy
```

```
conda install scipy
```

```
conda install conda -forge::pandas
```

or with a single sentence:

```
conda install conda -forge::matplotlib conda -forge::numpy conda -forge::scipy conda -forge::p
```

conda-forge
conda-forge:: specifies the source from which the packages are installed. It specifies that the packages should be installed from the conda-forge channel, which is a community-driven channel that provides a large collection of packages for conda.

Note that we did not install all packages. You will regularly come across packages that needs to be installed, with the information above and the information provided by the Python community, you will manage to install these interdependencies.

Using the command line
Normally we navigate through our folders by using a graphic interface and clicking through the folders. However, there is another way to navigate through your folders, namely using the command line as we are doing with Anaconda prompt.

When you run the command `dir` it returns the folders and files in the folder you are currently in. You can go to another folder by running the command `cd NAMEFOLDER`. If you want to move to a higher folder, run the command `cd ..`

Jupyter Notebook and Jupyter Lab In this course we make use of .ipynb files which are Jupyter Notebooks. To run these notebooks we can use IDE's as *Jupyter Notebook* or *Jupyter Lab*. Jupyter Notebook is a web-based interface that allows users to create and share documents with live code, visualizations, and narrative text in a linear format. JupyterLab, on the other hand, is a more advanced interface offering a flexible and modular environment with multiple panels, including notebooks, terminals, and text editors, providing a more versatile experience for interactive computing. I prefer to use Jupyter lab.

To start Jupyter lab, open the Anaconda terminal (Anaconda Prompt), move to the folder where you want to start Jupyter Lab in (where your files are located) and run the command `jupyter lab` in the terminal. A browser is started with which you can open your Notebooks, see Figure 4. Our first notebook tells you exactly how these notebooks work and how to develop and run code.

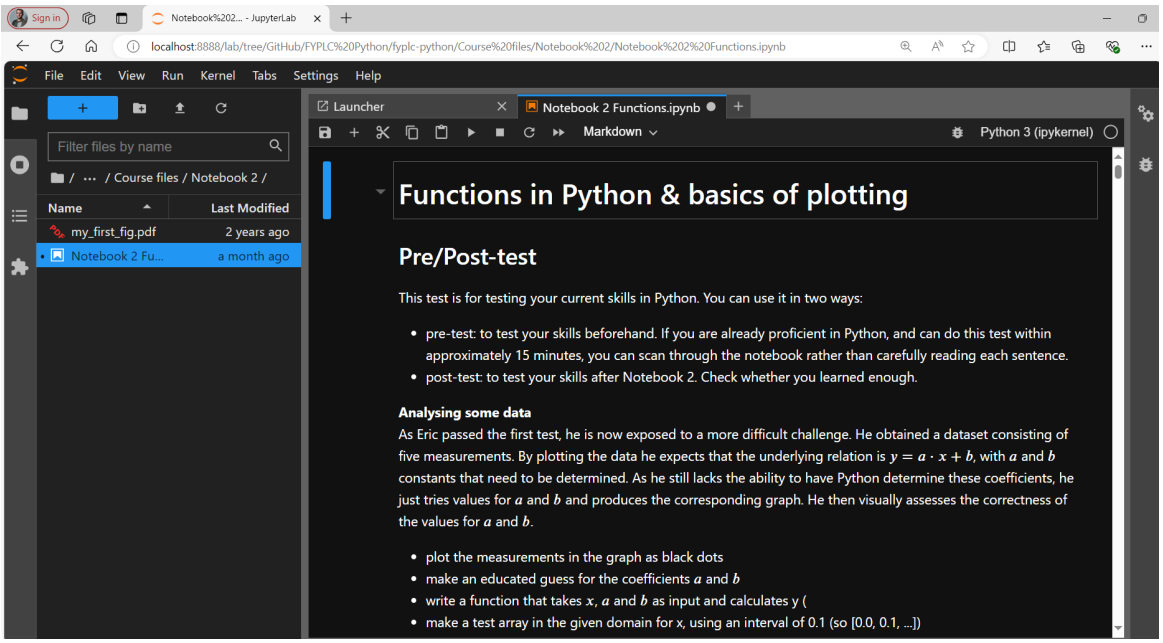


Figure 4: The Jupyter lab IDE

Git When writing code, especially larger projects, you want to keep track of your changes. Sometimes your code breaks and you want to go back to a previous version where all these troubles weren't there yet. For this purpose we have *version control software*. Most popular is **git**. Git is a distributed version control system that tracks changes in code or files, allowing multiple developers to collaborate efficiently on a project. GitHub is a web-based platform that hosts Git repositories, providing tools for collaboration, code sharing, and version management, often used for open-source and team projects. Here we don't go much deeper than this, but direct you to the manual of our colleagues at CiTG. You may want to check this out a bit later during your first year.

Vocareum We sometimes make use of **Vocareum**. Vocareum is a cloud-based platform designed for educational institutions to provide hands-on learning experiences, particularly in coding, data science, and cloud computing. It offers a virtual lab environment where students can complete assignments, receive automated feedback, and instructors can manage and grade coursework efficiently. We make use of Vocareum to grade your work and provide feedback.

If you want (online) help from the teaching assistants, you should upload your work to Vocareum. We can access your work, make changes and help you out. Moreover, you are to upload your exam(s) to Vocareum so that we can grade your work. Hence, get convenient with this platform.

Python for physicists, an introduction

Pre/Post-test This test is for testing your current skills in Python. You can use it in two ways:

- pre-test: to test your skills beforehand. If you are already proficient in Python, and can do this test within approximately 15 minutes, you can scan through this notebook rather than carefully reading each sentence.
- post-test: to test your skills after Notebook 1. Check whether you learned enough.

If you have not done any programming before, you skip this test and use it as a posttest.

Fixing Python errors in a database

Eric has recently taken up a new job at the University. His first job is to work on the TU database. As the previous programmer was a lazy basterd, the code has many flaws. Yes.... your job is to fix these.

In the TU database information on the building locations are stored. However, when printing the information, errors are reported (see below).

1) Fix the errors so that the information is printed correctly.

```
Street_name = 'Lorentzweg '
Number = 1
Full_adress = Street_name + Number

print(Full_adress)
```

In building a larger database, Eric decides to store the data in a different way, much more convenient than the use of multiple dubious variable names. Below, a first attempt is made. However, due to a typo, one address needs to be changed. But the method used to fix the problem is not working.

2) Explain why it is not working in this way.

3) Fix the error(s).

4) Explain why the chosen type, written by the former programmer, is not completely nonsense.

```
Building = (22,58)
Streetname = ('Lorentzweg ', 'Van der Maasweg ')
Number = ('1','8')
Number[1] = '9'

Full_adress = Building[1] + Streetname[1] + Number[1]
print(Full_adress)
```

Learning objectives Python has become the standard programming language in physics research. To teach you the basics of programming (in Python) we have prepared six notebooks for you, each notebook focussing on a different aspect. The first three notebooks relate to the basics of programming. Notebook 4 and 5 relate to data-analysis in physics using Python. The last notebook focusses on the statistics related to measurement uncertainty.

In this course, you will work using a platform called "Jupyter Notebooks". Jupyter notebooks are a way to combine formatted text (like the text you are reading now), Python code (which you will use below), and the result of your code and calculations all in one place. Go through these notebooks and run the examples. Try to understand their functioning and, if necessary, add code (such as print statements or variable overviews) to make it clear for you. In addition, there are exercises and practice cells where you can program for yourself. Don't be afraid to start coding yourself: writing code and making mistakes is the best way to learn Python.

In this notebook, you will learn the basic concepts of programming in Python. To get started, we will first explain the basic concepts of what Python is and how it works.

After completing this notebook, you are able to:

- start the python interpreter and run code in a notebook
- stop and start notebook kernels

- create variables
- use `%whos` to list the variables stored in the memory of the python kernel
- determine the type of a variable
- convert between different variable types (float, int, etc)
- collect user input using the `input()` command
- print variable values using the `print()` command
- use list, tuples and np.arrays and understand the differences
- combine arrays and insert data

We have embedded various YouTube clips in the first notebooks, which are **NOT** required to watch (we even recommend to not look at these videos because these are time consuming!) but might provide additional help and information when necessary. The last two notebooks include prerecorded lectures that we recommend to watch.

The solutions for the exercises in this notebook, and future ones, can be found at the bottom of the notebook, so that you can check your answer or have a peek if you are getting stuck. But don't look too quickly... thinking about how to solve a problem rather than solving it using the answers is much more educational!

5 ways I use code as an astrophysicist

Jupyter Notebook Tutorial: Introduction, Setup, and Walkthrough

NOTE When programming in the browser, some of the exercise cannot be made. For instance, stopping and rerunning the kernel is not possible. Also, it is not possible to ask input from the user. Once you have installed the software, you can return to these exercises and try to run them again.

Also, your code will be lost when programming in the browser. Write down the code you want to keep!

What is Python? And what are Jupyter Notebooks? Python is an (interpreted) computer programming language. Using Python, you can ask your computer to do specific things, like perform a calculation, draw a graph, load data from a file, or interact with the user.

Every time you run Python, either by running it on the command line, or by running it in a Jupyter notebook like you are now, a Python **"kernel"** is created. This kernel is a copy of the Python program ("interpreter") that runs continuously on your computer (until you stop it).

Jupyter notebooks are a way to interact with the Python kernel. Notebooks are divided up into "cells", which can be either a text cell (in a special text formatting language called markdown), like the one you are reading now. There are also code cell (containing your code), like the cell below it.

The selected cell is surrounded by a box. If you press "ENTER" in a text cell you can start editing the cell. If you push "Run" above, or push "Shift-Enter", the code will be "run". If it is a code cell, it will run a command (see below). If it is a markdown cell, it will "compile" the markdown text language into formatted (HTML) text.

You can give commands to this kernel by typing commands using the Python language into the code cells of the notebook. Here, you can find an example of a code cell that contains a simple Python command `print`, which prints a text string to the command line.

To send this command to the Python kernel, there are several options. First, select the cell (so that it is either blue or green), and then:

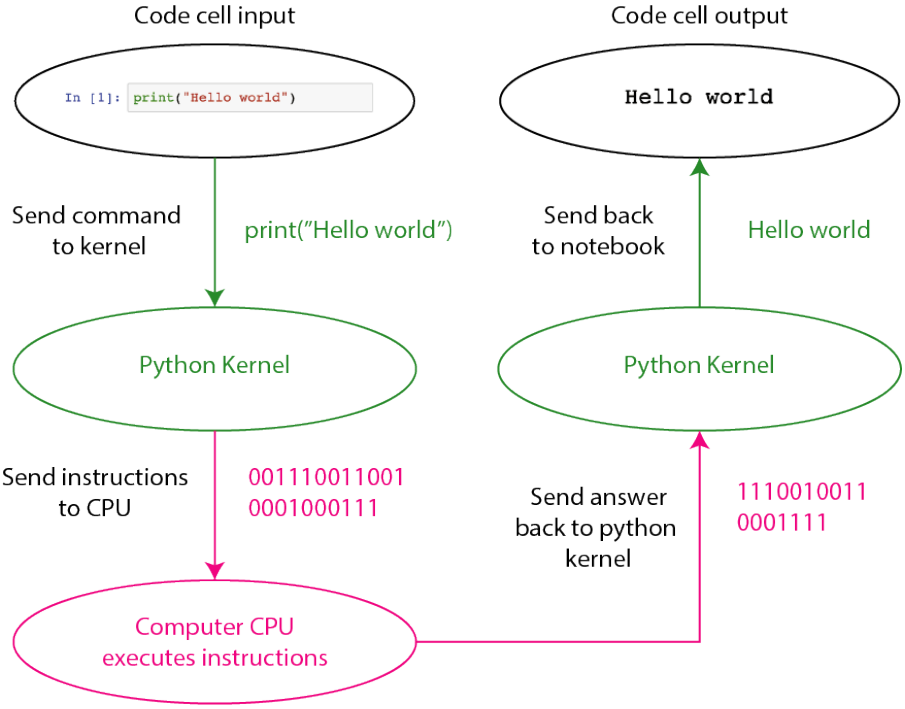
1. Click on the **Run** button above in the toolbar. This will execute the cell and move you to the next cell.

2. Push **Shift-Enter**: this will do the same thing

3. Push **Control-Enter**: this will run the cell but leave it selected (useful if you want to re-run a cell a bunch of times)

When you run the cell, the code will be sent to the Python kernel, which will translate your Python command into a binary language your computer CPU understands, send it to the CPU, and read back

the answer. If the code you run produces an "output", meaning that the kernel will send something back to you, then the output that your code produces will be displayed below the code cell in the "output" section of the code cell. This is a scheme of what this process looks like "behind the scenes":



After you have run the code cell, a number will appear beside your code cell. This number tells you in which order that piece of code was sent to the kernel. Because the kernel has a "memory", as you will see in the next section, this number can be useful so that you remember in which order the code cells in your notebook were executed.

In the example above, the code cell contains only a single line of code, but if you want, you can include as many lines as you want in your code cell:

```
print("Hello")
print("world")
print("Goodbye")
```

```
Hello
world
Goodbye
```

In the above, the text in the code cell are all Python commands. In addition, if you start a line in a code cell with a `#`, Python will ignore this line of text. This is used to add **comments** to your code. It is good programming practice to use comments to explain what the code is doing:

```
# This will print out a message
print("This is a message")
```

It might be hard at first to write proper comments. The comments should be short but still add information that is of value. One can question the value of the comment above as it hardly adds information to what the code is already saying... Some more information can be found here: https://stackabuse.com/commenting-python-code/.

Exercise 1.1

Print your own string to the command line. Can you print special characters as well?

```
# your code here
```

The Python kernel has a memory In addition to asking Python to do things for you, like the "Hello world" example above, you can also have Python remember things for you. To do this, you can use the following syntax:

```
a = 5
```

In Python, the `=` symbol represents the **assignment operator**: it is an instruction to **assign** the value of `5` to the variable `a`. If variable `a` already exists, it will be over-written with the new value (in fact, `a` is a Python object). If variable `a` does not yet exist, then Python will create a new variable for you automatically.

For you, the cell above will create a "variable" named `a` in memory of the Python kernel that has the value of 5. We can check this by printing the value of a:

```
print(a)
```

Besides numerical values variables can also be strings, which are sequences of characters. You make a string by putting the text between quotes, as seen above in "Hello World".

Note that we can also add a message if we add a string and a numerical value in the `print()` statement by combining things with commas:

```
print("The value of a is", a)
```

Exercise 1.2
Combine multiple strings and numerical values in a single `print` statement using the `,` separator.

```
# your code here
```

Exercise 1.3
Change the value of `a` to 7 by executing the following cell, and then re-run the **above** cell containing the command `print(a)` (the one with output `5`). What value gets printed now in that cell?

```
# your code here
```

As you can see in notebooks that the location of your code doesn't matter, but the order in which you execute them does!!

We can also use variables to set the values of other variables:

```
b = 0
print(b)
b = a
print(b)
```

A 'funny' thing happens with the command **b = a**. As we say that b and a are the same, rather than creating a new spot in the memory where the information is stored, the variables **b** and **a** obtain the same *memory address*. If we call upon **a**, Python searches its memory, and obtains the data stored at that unique id. We can see this using **id** function.

```
print(id(a))
print(id(b))
a = 5
b = 10
print(id(a))
print(id(b))
```

This is important to know because if we say **a = b** and we change the value of **b**, the value of **a** changes as well! We will go into more detail later on.

Sometimes, if you execute a lot of cells, or maybe even re-execute a cell after changing its contents, you might lose track of what variables are defined in the memory of your Python kernel. For this, there is a convenient built-in "magic" command called `%whos` that can list for you all the variables that have been defined in your kernel, along with their values:

```
a=5
%whos
```

(Some notes about `%whos`: `%whos` is not a "native" command of the Python language, but instead a "built-in" command that has been added by the creators of Jupyter. Because of this, you cannot use it outside of Jupyter / iPython...)

If we define some new variables, they will also appear in the list of defined variables if you execute `%whos`:

```
c = 10
d = 15.5

%whos
```

In this case the variables' names are displayed, their values, but also their type. Type defines the format in which a variable is stored in memory. In this case `int` stands for integer and `float` stands for floating point number, which is the usual way in which real numbers are stored in a computer. We will learn more about Python variable types below.

Starting and stopping the kernel When you open a notebook for the first time, a new kernel will be started for you, which will have nothing in its memory.

Important to understand: if you close the tab of your browser that has the notebook, Jupyter **will not** shut down the kernel! It will leave the kernel running and when you re-open the notebook in your browser, all the variables you defined will still be in the memory. You can test this by closing this notebook right now, clicking on the link to open it in the "tree" file browser tab of Jupyter, and then re-running the cell above with the command `%whos`.

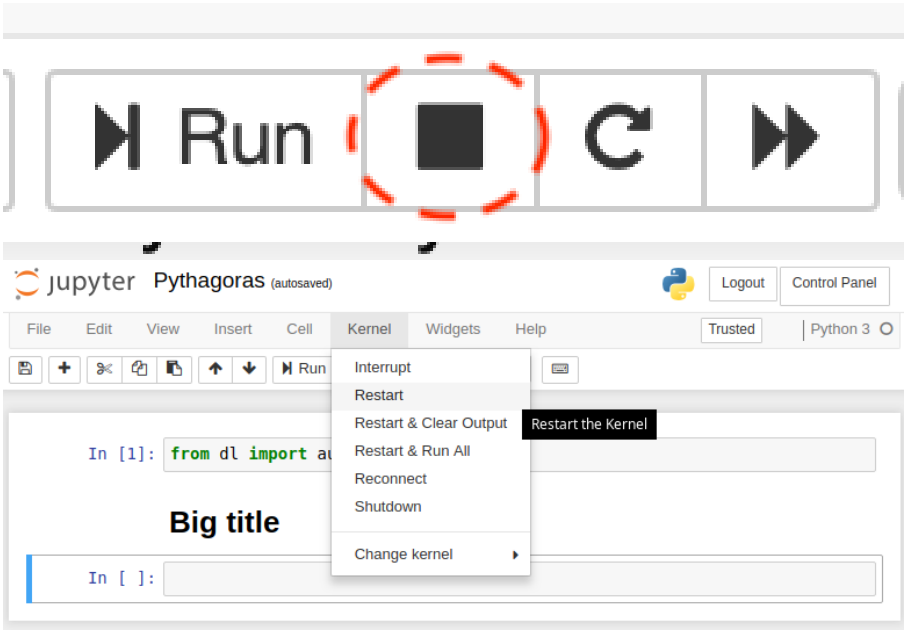
How do I shutdown a kernel? And how do I know if a notebook on my computer already has a kernel running?

- First, as you may have noticed, when you closed this notebook and went back to the "tree" file browser, the notebook icon had turned green. This is one way that Jupyter tells you that a notebook file has a running kernel.
- Second: in the "tree" view of the Jupyter interface, there is a link at the top to a tab "Running" that will show you all the running kernels and allow you to stop them manually.

Sometimes, you may want to restart the kernel of a notebook you are working on. You may want to do this to clear all the variables and run all your code again from a "fresh start" (**which you should always do before submitting an assignment!!!! Kill the kernel, run all cells and kill the kernel again. Any errors show up directly**). You may also need to do this if your kernel crashes (the "status" of your kernel can be seen in the icons at the right-hand side of the Jupyter menu bar at the top of the screen).

For this, there is both a menubar "Kernel" at the top, along with two useful buttons in the toolbar:

- "Stop": tells the kernel to abort trying to run the code it is working on, but does not erase its memory
- "Restart": "kill" the kernel (erasing its memory), and start a new one attached to the notebook.



To see this in action, you can execute the following cell, which will do nothing other than wait for one minute:

```
from time import sleep
sleep(60)
```

You will notice that while a cell is running, the text beside it shows `In [*]:`. The `*` indicates that the cell is being executed, and will change to a number when the cell is finished. You will also see that the small circle beside the `Python 3` text on the right side of the Jupyter menu bar at the top of the page will become solid. Unless you have a lot of patience, you should probably stop the kernel, using the "Stop" button, or the menu item "Kernel / Interrupt".

`Exercise 1.4`

List the stored variables using the `%whos` command. Subsequently, restart the kernel. What variables are stored in the memory of the kernel before and after the restart?

```
# your code here
```

Python variable types As we saw above, in Python, variables have a property that is called their "type". When you use the assignment operator `=` to assign a value to a variable, Python will automatically pick a variable type it thinks fits best, even changing the type of an existing variable if it thinks it is a good idea.

You have, in fact, already seen information about the types of variables in the `%whos` command again:

```
%whos
```

In the second column, you can see the **type** that Python chose for the variables we created. `int` corresponds to integer numbers, `float` corresponds to floating-point numbers. You can see that for variable `c`, Python had to choose a `float` type (because 15.5 is not an integer), but for `a` and `b`, it chose integer types.

(In general, Python tries to choose a variable type that makes calculations the fastest and uses as little memory as possible.)

If you assign a new value to a variable, it can change the variables type:

```
a = a/2
type(a)
```

Because $5/2 = 2.5$, Python decided to change the type of variable `a` from `int` to `float` after the assignment operation `a = a/2`.

When you are using floating point numbers, you can also use an "exponential" notation to specify very big or very small numbers:

```
c = 1.5e -8
```

The notation `1.5e-8` is a notation used in python to indicate the number 1.5×10^{-8}.

A third type of mathematical variable type that you may use in physics is a complex number. In Python, you can indicate a complex number by using `1j`, which is the Python notation for the complex number i:

```
d = 1+1j
type(d)
```

The notation `1j` is special, in particular because there is **no space** between the number `1` and the `j`. This is how Python knows that you are telling it to make a complex number (and not just referring to a variable named `j`...). The number in front of the `j` can be any floating point number. For example:

```
0.5j
```

In addition to the mathematical variable types listed above, there are also other types of variables in Python. A common one you may encounter is the "string" variable type `str`, which is used for pieces of text. To tell Python you want to make a string, you enclose the text of your string in either single forward quotes `'` or double forward quotes `"`:

```
e = "This is a string"
f = 'This is also a string'
type(e)

print(e)
print(f)
```

You can also make multiline strings using three single quotes:

```
multi = \
'''
This string
has
multiple lines.
'''
print(multi)
```

Note here that I have used a backslash: this a way to split Python code across multiple lines.

Although it's not obvious, Python can also do "operations" on strings, the `+` mathematical operators we saw above also works with strings.

Exercise 1.5

Discover what the `+` operator does to a string, i.e. print the output of the sum of two strings.

```
# Your code here
```

A useful variable type we will introduce here is the "boolean" type `bool`. A boolean variable can have two values: `True` or `False`. You type them in directly as `True` and `False` with no quotes (you will see them turn green).

```
g = False
type(g)
```

We will use boolean types much more later when we look at program control flow, but a simple example using the `if` statement is given below:

No panic if you don't yet understand the `if` statement, there will be another entire notebook dedicated to them. This is just an example of why boolean variables exist.

```
if True:
    print("True is always true.")

if g:
    print("g is true!")

if not g:
    print("g is not true!")
```

You can try changing the value of `g` above to `False` and see what happens if you run the above code cell again.

Also, useful to know: numbers (both `int` and `float`) can also be used in True / False statements! Python will interpret any number that is not zero as `True` and any number that is zero as `False`.

Exercise 1.6

Discover which *numbers* can be used as `True` and `False` in Python by changing the value of `g` above and re-running the cells.

YouTube video player

YouTube video player

Lists & Tuples It often happens that you want to store data that belong together (a collection of items). You have different options, we discuss lists & tuples, and subsequent (for numerical purposes) numpy arrays.

Let us think of storing personal data where we need to know: First name; Family name; Address; City. We then can make a list, see below.

```
Person_1 = ['Feek','Pols','Lorentzweg',1,'Delft']
print(Person_1)
print(type(Person_1), type(Person_1[0]), type(Person_1[3]))
```

It is interesting to see that Person_1 is a list and that within the list other types exist (Note: the first item is referred to by 0 as computers start to count at 0). If we make a mistake, we can replace an item in the list.

```
Person_1[0] = 'Freek'
print(Person_1)
```

Another way we can store data is using a tuple, see below. Note that the only difference with a list is the use of the brackets.

```
Person_2 = ('Erik','Janssen','Lorentzweg',1,'Delft')
print(Person_2)
print(type(Person_2), type(Person_2[0]), type(Person_2[3]))
```

So, what is the difference then between a list and a tuple? The most important difference is that tuples are immutable, you cannot change them:

```
Person_2[1] = 'Jansen'

a = [2,3,4]
b = a
b[0] = -10
print(a)
```

In the example above, something strange happens. We did not change **a**, did we? But remember that **b** is not a copy of **a** with a new location in the memory. Rather than that, **b** obtains the same location in the memory (you can check with id(a)). If we change **b**, we thus change **a**! We can also make tuples with only a single item stored. We have to make use of a comma, otherwise Python will not recognize it as a tuple.

```
#not a tuple
n_a_t = (1)
#a tuple
a_t = (1,)

#checking
print(type(n_a_t))
print(type(a_t))
```

Lists are mutable and are thus called variables. However, a tuple cannot be varied and is thus not a variable. It is called an object. Since it is immutable, it requires less space. We can still make effective use of tuples (and lists):

```
a = [2,3,5]
b = (2,3,5)
print(a[0]*2)
print(b[0]*2)
```

Exercise 1.7
Try to change the first value of **b**. In order to do so, make sure **b** becomes a list! You can make use of b = list(a).

Numpy Arrays Until now, the variable types we have been working with in Python represent relatively simple data types:

- `int`: integer numbers
- `float`: floating point numbers
- `complex`: complex-valued floating point numbers
- `bool`: boolean "truth" values (which can have `True` and `False`)
- `str`: strings
- `list`: mutuable list of variables
- `tuple`: immutuable list of variables

The first four are very simple data types, but actually the last one is more complicated than it looks . The `list` is a vector like-variable type. However, unlike physical vectors, it cannot be multiplied, subtracted, etc.

Here, we will introduce a new datatype that is handy for Physics calculations and that comes from the Python software package numpy called **numpy arrays**.

What are numpy arrays?

Numpy arrays are a way to work in Python with not just single numbers, but a whole bunch of numbers. With numpy arrays these numbers can be manipulated just like you do in, for example, linear algebra when one works with vectors:

Row and column vectors

For example, a (column) vector $\vec{a}$ with 5 entries might look like this:

$$\vec{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \tag{2}$$

In linear algebra you are used to manipulate these vectors, this can be done in a similar way with numpy arrays. We will use numpy arrays extensively in Python as vectors, like above, but also for storing, manipulating, and analyzing datasets (like a column of an excel spreadsheet).

To use numpy arrays, we first need to import the numpy library, which we will do using the shortened name "np" (to save typing):

```
import numpy as np
```

Now that we have imported numpy, we can use functions in numpy to create a numpy array. A simple way to do this is to use the function np.array() to make a numpy array from a comma-separated list of numbers in square brackets:

```
a = np.array([1,2,3,4,5])
print(a)
```

Note that numpy does not make a distinction between row vectors and column vectors: they are just vectors. But what if we want an array running from 0 to 100, and only the even numbers? Do we type all numbers? Certainly not! We can use for instance numpy's linspace or arange:

```
array_evennr_1 = np.linspace(0,100,51)
print(array_evennr_1)

array_evennr_2 = np.arange(0,101,2)
print(array_evennr_2)
```

Note the difference between the two ways of making the array and their output!

Exercise 1.8

Make a list and a numpy array in which the values 2, 3, 5 are stored. Multiply the list and array by the value 2 and print the outcome. What is the difference between the mathematical operation on the list and numpy array? Why do we, physicists, prefer numpy arrays?

```
#your code
```

In some cases we want to add items to our array or combine different arrays into a single one. There are different ways to do that. To combine to numpy arrays, we can make use of the concatenate function. To add items at the end of our array, we can use the append function.

```
a = np.array([1,2,3,4,5])
a = np.append(a,6)
b = np.array([6,7,8,9,10])
b = np.delete(b,0)
c = np.concatenate((a,b))
c = np.append(c,11)
print(c)
```

Exercise 1.9
Below we have made two arrays, one with even numbers and one with odd numbers. Combine these two arrays in a single array and sort them (see https://numpy.org/doc/stable/reference/generated/numpy.sort.html). Next, delete the third element in the array.

```
even_nr = np.arange(0,10,2)
print(even_nr)
odd_nr = np.arange(1,11,2)
print(odd_nr)
#your code
```

Converting variables between different types We can also convert a value from one type to another by using functions with the same name as the type that we want to convert them to. Some examples:

```
float(5)
```

```
int(7.63)
```

Note that when converting an `float` to an `int`, Python does not round off the value, but instead drops all the numbers off after the decimal point (it "truncates" it). If we want to convert to an integer and round it off, we can use the `round()` function:

```
b = round(7.63)
print(b)
```

```
a = 4.45
b = 4.55
a = round(a,1)
b = round(b,1)
print(a)
print(b)
```

There seems to be something odd happening here, as when we round to the first decimal number **a** and **b** obtain the same value. Why this happens is covered in the last two notebooks. It has to do with always rounding down or up results in a systematic error in the mean.

```
print(type(b))
print(b+0.4)
```

This works for conversions between many types. Sometimes, you will lose information in this process: for example, converting a `float` to an `int`, we lose all the numbers after the decimal point. In this example, Python makes a guess at what you probably want to do, and decides to round off the floating point number to the nearest integer.

Sometimes, Python can't decide what to do, and so it triggers an error:

```
float(1+1j)
```

A very useful feature is that Python can convert numbers into strings:

```
a = 7.54
str(a)
b = a + 1
print(b)
%whos
```

That is actually what happens when you use the `print()` commands with a numeric value.

But also very useful is that as long as your string is easily convertable to a number, Python can do this for you too! Note below that the quotation marks makes it a string!

```
float('5.74')

int('774')

complex('5+3j')
```

We even can convert immutable tuples to mutable lists, and back again.

```
a = (4,5)
print(type(a))
a = list(a)
print(type(a))
a = tuple(a)
print(type(a))
```

Exercise 1.10
Define a list of parameters with as many types as possible, i.e. all the examples you see above and maybe a few more. Use `%whos` to see how they look inside the computers' memory. Try to change their format and re-run the `%whos` command.

```
# Your parameters list
a=
b=

# Parameter formats in the computer
%whos
```

Names of variables So far we have not talked about the names of variables. Throughout the notebook we used a and b for single variables, list, numpy arrays and so on. Each time the previous value is overwritten. Furthermore, what does a refers to? Is it acceleration, is it a test variable? We have no clue.

If you want to be able to read your code next year, or if you want us to be able to read your code, you have to come up with proper names for your variables. We advice you to look at the YT-movie clip below as some conventions are clearly elaborated. Also, look at the PEP 8 – Style Guide for Python Code: https://www.python.org/dev/peps/pep-0008/ for more related tips and conventions.

```
YouTubeVideo('Uw95Uc3xgWU', width = 600, height = 450)
```

Python can do math Python has a set of math functions that are directly built in to the language. You can use Python as a calculator!

```
1+1
```

So from now on, don't use your calculator or excel when working on calculations yourselves! Use Python!

Calculations also work with variables:

```
a = 5
print(a+1)
```

Exercise 1.11
Discover what the following Python operators do by performing some math with them: `*`, `-`, `/`, `**`, `//`, `%`. Print the value of the mathematical operation to the command line in subsequent cells.

```
# Try out *

# What did it do? Add your answer here:

# Try out -

# What did it do? Add your answer here:

# Try out /

# What did it do? Add your answer here:

# Try out **

# What did it do? Add your answer here:

# Try out //

# What did it do? Add your answer here:

# Try out %

# What did it do? Add your answer here:
```

Another handy built-in function is `abs()`:

```
print(abs(10))
print(abs( -10))
print(abs(1j))
print(abs(1+1j))
```

You can find the full list of built-in math commands on the Python documentation webpage: https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Sending data to Python using input() So far, we have seen examples of "output": Python telling us stuff, like in the first "Hello world" example above.

And we have seen examples of "code": us giving instructions to Python to do things.

In addition, we can also send stuff to Python. Often in physics, we do this by having Python read data files, which we will cover later, but we can also send information to Python using the `input()` command:

```
a = input()

print()
print("The input was:")
print(a)
```

Here something seemingly odd happens: even if we type a number into the input box, it will *always* return a string variable of type `str`:

```
type(a)
```

If we want to use our input as a number, we have to convert it to a number, for example, by using the `float()` function. Moreover, it might be handy to tell the user what he or she is supposed to do. You can specify text for the label of the input box:

```
a = input("Enter a number: ")
a = float(a)
print("\nThe value of a is:", a)
print("a has the type:", type(a))
```

A very useful tool is string formatting (https://realpython.com/python-string-formatting/). We can tell Python that a number is comming, specify its type and later include the number:

```
print('The number given by the use is %.1f ' %a) #prints 1 dec float
print('The number given by the use is %d ' %a) #prints as integer
print('The number given by the use is %e ' %a) #prints with scientific notation
```

Exercise 1.12

Use the `input` function to get parameters of integer, float and string format into the computer. Have Python check whether the proper type is assigned.

```
# Your code here
```

Tab completion in Jupyter Notebooks Computer programmers often forget things, and often they forget what variables they have defined. Also, computer programmers always like to save typing if they can.

For this reason, the people who made Jupyter notebooks included a handy feature called "TAB completion" (which is actually something that has been around for a long time in unix and ms-dos command line environments).

The idea is that if you start typing part of the name of a variable or part of the name of a function, and then push the `TAB` key, Jupyter will bring up a list of the variable and function names that match what you have started to type. If only one matches, it will automatically type the rest for you. If multiple things match, it will offer you a list: you can either keep typing until it's unique and press TAB again, or you can use the cursor keys to select the one you want.

Here is an example:

```
this_is_my_very_long_variable_name = 5
this_is_another_ones = 6
```

Now click on the following code cell, go the end of the lines in this cell and try pushing `Tab`:

```
this_is_a
```

Handy! Jupyter did the typing for me!

If multiple things match, you will get a drop-down box and can select the one you want. So press `Tab` : after

```
this_is
```

You can also keep on typing: if you just type `a` after you hit tab and then hit tab again, it will finish the typing for you.

Exercise 1.13

Use tab completion on the initial letters of a few of the commands that have been presented. Along the way you will discover many more Python commands!

```
# Your code here
```

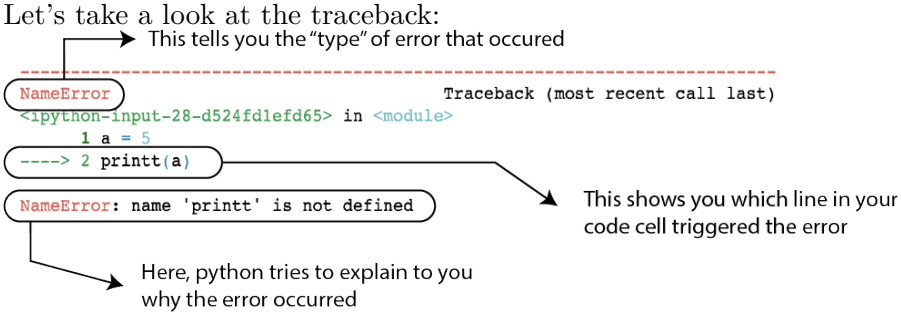
Understanding Python Errors Sometimes, the code you type into a code cell will not work. In this case, Python will not execute your code, but instead print out an error message. In this section, we will take a look at these error messages and learn how to understand them.

Let's write some code that will give an error. For example, this is a typo in the name of the `print()` command:

```
a = 5
printt(a)
```

After your code cell, you will see some colored text called a "Traceback". This "Traceback" is the way that Python tries to tell you where the error is.

Let's take a look at the traceback:



The traceback contains three important details that can help you:

1. The type of error
2. Where the error occurred in your code
3. An attempt to explain why the error happened

For 1 and 2, Python is pretty good and will communicate clearly. For 3, sometimes you need to have some experience to understand what python is trying to tell you.

In this specific case, the type was a `NameError` that occured on line 2 of our code cell. *(By the way, in the View menu, you can turn on and off line numbers in your cells.)*

A `NameError` means that Python tried to find a function or variable that you have used, but failed to find one. If you look already at the line of code, you can probably spot the problem already.

At the very end of the traceback, Python tries to explain what the problem was: in this case, it is telling you that there is no function named `printt`.

You will also get a `NameError` if you try to use a variable that doesn't exist:

```
print(non_existant_variable)
```

Another common type of error is a `SyntaxError`, which means you have typed something that Python does not understand:

```
a = a $ 5
```

You can also get errors if you try to use operators that do not work with the data type you have. For example, if you try to "divide" two strings:

```
"You cannot " / "divide strings"
```

Here, you get a `TypeError`: the division operator is a perfectly fine syntax, it just does not work with strings.

In Python, errors are also called "Exceptions", and a complete list of all error (exception) types, and what they mean, can be found here:

https://docs.python.org/3/library/exceptions.html#concrete-exceptions

Sometimes, you can learn more about what the error means by reading these documents, although they are perhaps a bit hard to understand for beginners.

In last resort, you can also always try a internet search: googling the error message can help, and there are also lots of useful posts on stack exchange (which you will also often find by google).

Exercise 1.14

Run the following code and try to understand what is going wrong by reading the error message.

```
a=10
b=0
c=(a/b)

4 + practicum*3

d='practicum is great' + 2
```

If you are not sure what to do, what the function was called or need some help, there are two handy tools. First, if you put a question mark (?id) before your variable or function, you get additional information.

The other tool is to type your variable, put a dot behind it and autocomplete using tab. This will produce a dropdown menu with all the available actions that can be carried out.

Exercise 1.15

Try **?%whos** below and see what information is given.

Exercise 1.16

Make a variable and an array. Use the second tool to find out what actions can be performed on the two different types.

Solutions to Exercises Exercise 1.1

```
# You can print special characters:
print("My first message #1")
print("Another special character: one \ two")

# However, there are some special combinations such as \n that have another meaning.
# In this case, \n starts a new line.
print("Another special character: one \n two")

# If we wish to print the text \n, we should add another \ to indicate that we do not.
print("Another special character: one \\n two")
```

Exercise 1.2

```
print('The value is', 10)
# Note that a space is added in between the string and the number.
```

Exercise 1.3

```
a = 5
print(a) # this prints 5
a = 7
print(a) # this prints 7 as the current value of a is 7
```

Exercise 1.4

```
# After restarting the kernel all variables, such as a, are no longer stored and do not show u
```

Exercise 1.5

```
print('This is the first string' + 'and this is the second string.')
# The strings are joined.
```

Exercise 1.6

```
# Only 0 is False, some examples:
if 0:
    print('0 is True')
else:
    print('0 is False')

if 1:
    print('1 is True')
else:
    print('1 is False')

if -57:
    print(' -57 is True')
else:
    print(' -57 is False')

if 6/7:
    print('6/7 is True')
else:
    print('6/7 is False')
```

Exercise 1.7

```
#
```

Exercise 1.8

```
a = [2,3,5]
b = np.array([2,3,5])

print(2*a) # Multiplying the list is like adding the two lists such that it is repeated.
print(2*b) # Multiplying the numpy array multplies each value in the array.
```

Exercise 1.9

```
even_nr = np.arange(0,10,2)
print(even_nr)
odd_nr = np.arange(1,11,2)
print(odd_nr)

total = np.append(even_nr, odd_nr)
print(total)
total = np.sort(total)
print(total)
total = np.delete(total, 2)
print(total)
```

Exercise 1.10

```
#
```

Exercise 1.11

```
print('7*3 =', 7*3)
print('7 -3 =', 7 -3)
print('7/3 =', 7/3)
print('7**3 =', 7**3)
print('7//3 =', 7//3)
print('7%3 =', 7%3)
```

Exercise 1.12

```
# You can chech that the input is always a string.
a = input('type input: ')
print(type(a))
```

Exercise 1.13

```
#
```

Exercise 1.14

```
# ZeroDivisionError: division by zero
# You cannot divide by zero, so this results in an error.

# NameError: name 'practicum' is not defined
# You cannot use an undefined variable, use a string to print text.

# TypeError: can only concatenate str (not "int") to str
# You cannot add a string and an integer. We can add two strings:
print('practicum is great' + str(2))
```

Exercise 1.15

```
#
```

Exercise 1.13

```
#
```

Good coding practices

In this notebook, we outline some example of "good coding practices": things you should do to make your code robust and understandable to others.

Learning objectives:

- Student is able to write code that has a clear and understandable structure
- Student is able to write code with descriptive variable names
- Student is able to write code with explanatory comments
- Student is able to write code that avoids "hard coding"

Writing understandable code Once you become an expert at coding, it will become very easy to write code that python can understand.

What is actually a big challenge in computer science is actually writing code that works, that is efficient, and most importantly, that **other people will understand!**

You might think: but I'm writing this code only for myself, so I don't really care about if my code is easy to understand, right?

There are many reasons why this is incorrect:

1. In this course, your teachers and your TAs will need to understand your code in order to grade it. If we can't quickly understand what you have done, **this will affect your grade!!!** (One of our grading criteria will be the clarity and readability of your code.)

2. You might, in the future, want to share your code with someone else. If they have to spend a lot of time figuring out what you've done, then your code is useless to them. (In fact, this is the power of open languages like python, and is why python is so fantastically successful: there is a huge amount of code that other people have written that you can reuse!)

3. Weeks, months, or even years later, you might want to go back and re-use your own code. If your code not written in a clear and understandable way, you yourself will waste a lot of time trying to figure out what you did!

In preparing the lecture notebooks for this course, I personally experienced 3 myself: I went back to some simple code that I wrote a few weeks earlier, and I realized I had no idea what the code did!

So how do you make sure that your code is understandable? There are two practices that can make this useful:

- **Meaningful variable names and logical structure**
- **Comments explaining what you're doing**

As a concrete example, I will take the following code below and show how by using these two techniques, an imcomprehensible piece of code that is them modified in two steps to be very easy to understand.

BAD code Here is the "bad" code:

```
foo3 = 2.0; foo1 = 10; foo2 = 0.0
def foo6(foo1):
    return foo1**2
foo5 = 0.5*foo6(foo2) + foo6(foo3); foo4 = (foo3 -foo2)/foo1
for foo7 in range(1,foo1):
    foo5 += foo6(foo2+foo7*foo4)
foo83 = (foo5*foo4)
```

Can you figure out what this code does? It would take me personally a lot of time...

This code is a mess, but is technically correct: it will give the right answer. But correct code can also be terrible code, and for me, the code above is really terrible!

Better variable names and logical structure

```
def f(x):
    return x**4 - 2*x + 1

N = 10
a = 0.0
b = 2.0
h = (b -a)/N

s = 0.5*f(a) + 0.5*f(b)
for k in range(1,N):
    s += f(a+k*h)

answer2 = (h*s)
```

Why is this better?

- Not all variables and functions are named `foo`, but have somewhat meaningful names like `f(x)`, `a`, `b`, `N`
- The definition of the variables is in a logical order (a is after b, for example)
- The functions are defined at the top (a common convention to make your code understandable)
- There are blank lines separating different logical parts of the code:
  - function definitions at the top
  - then the defining the values of the input variables
  - then the loop that does the actual work
  - then the definition of the answer / output

When you work on the Integration notebook, you may also recognise this from the trapezoidal technique. The variable names also match the formulas from a textbook we have used in the past, and if you had just read the textbook section on Integration, you will also probably almost instantly recognize all the variable names from those used in the derivation from the book, and also what it does.

However, if you come back at a later time when the integration section of the textbook is not so fresh in your head (as I did recently while updating this notebook), you may immediately think "what was I doing here?"

Even better: Descriptive variable names The above code is already better. And if I am looking a textbook where all the things above are clearly explained, then maybe the meaning of the parameters `a`, `b`, `N` etc are clear to me. However, if you haven't seen the code before, it may not be immediately obvious. Are `a` and `b` the slope and intercept of a line $y = ax + b$? Or are they something else? Is `N` the number of points in my discretisation, or is it the number of slices in my integral?

For this reason, it is better to use descriptive variable names that themselves already describe what the meaning of the variable is:

```
def f(x):
    return x**4 - 2*x + 1

N_slices = 10
start = 0.0
stop = 2.0
step_size = (stop -start)/N_slices

running_sum = 0.5*f(start) + 0.5*f(stop)

for i in range(1,N_slices):
```

```
    running_sum += f(start+i*step_size)

answer2 = (step_size*running_sum)
```

This is a bit more typing, but you can also use "tab completion" to save typing: just type the first few letters, push the "tab" key, and the Jupyter notebook will automatically type the rest out for you.

Commenting your code By taking some time to explain what you're doing and why, this code can now become instantly understandable:

```
# Code for integration with the Trapezoidal rule

# The function we want to integrate
def f(x):
    return x**4 - 2*x + 1

# The number of slices we will use, and the starting and end points of the integral
# Note that this is NOT the number of points for our discritisation: that is N_slices+1.
N_slices = 10

# The starting and stopping points of our integration range
start = 0.0
stop = 2.0

# The step size
step_size = (stop -start)/N_slices

# A running sum we will use
# We start by the half of the start and end points (trapezoidal rule)
running_sum = 0.5*f(stop) + 0.5*f(stop)

# Now a for loop that does the sum.
# range(1,N_slices) will give us 9 numbers running from 1 to N_slices -1
# For N_slices=10, we will get a list of 9 numbers: [1,2,3,4,5,6,7,8,9]
# Note that we do not need i = 0 because in the trapezoidal rule, we add it
# separately above with a factor of 0.5, together with the end point
for i in range(1,N_slices):
    running_sum += f(start+i*step_size)

# The integral is then given by the produce of the sum and the step siz
answer2 = (running_sum*step_size)
```

Here, we explain what we are doing and why! This code is likely understandable by anyone who reads it, including myself again in a year's time.

In particular, we can point out some of the sneaky things, such as the difference between number of slices and number of points, and also why our `range()` function starts at 1 and not 0.

It is always a good idea to add comments explaning what you do. But, of course, you don't want to write code that is more comments than actual code! Where do I draw the line? When do I decide if I should add a comment of if I think it is already clear enough without it?

Good guideline: if there is something that you had to fiddle around with for a while to get the code correct, add a comment in your code so that you remember this and others get to learn from your hard work!

Using descriptive variable names also helps, as then you may not have to add a comment where you otherwise would.

And, while it takes some time while you're doing it, but adding detailed comments to our code will make your life MUCH easier in the future, will make your code understandable for others, and will maximize your grade :)

Summary: Writing understandable code In summary, you can make your code much more understandable for others if you:

1. Use descriptive variable names
2. Explain things in comments

Avoid hard-coding "Hard coding" is one of the coding practices we will be discouraging in this course, and the use of "hard coding" can lose you points in your final exam.

What is "hard coding"?

Hard coding is when you fill in values repeated at multiple places in your code. For example, say your are asked to make an array `x` that is 1000 points and runs from 0 to 10, calculate array $y = \sin(x)$, and then print out the percentage of points in `y` that have a value of less than 0.3. In the code snippets below, we will take a look at a **BAD** way of doing this with hard-coding, and then also proper ways of doing this that do not involve hard-coding.

BAD code with hard-coding Here is an example of a **BAD** piece of code that does this using hard-coding of the number of points:

```
import numpy as np

# An example of hard coding (DO NOT DO!!!!)
x = np.linspace(0,10,1000)
y = np.sin(x)

num = 0
for i in range(1000):
    if (y[i]<0.3):
        num += 1

print("The percentange of points in y less than 0.3 is %.2f %%" % (num/1000*100))
```

In this example code, we say that we have "hard coded" the number of points in the array. Say we wanted to then change the number of points in array `x` from 1000 to 1500: we would then have to go through all of our code by hand and change `1000` each time into `1500`.

This time that is not so difficult since it is a short piece of code, but when you build more complex programs, this will become a lot more work, and the chance of getting a bug in your code will increase exponentially!

GOOD example with no hard-coding: replacing hard coded numbers with a variable Below, instead of typing in `1000` each time manually, we will define a variable at the top that will define how many points `x` has. This way, if we want to change the size of `x`, all we have to do is change this variable and everything will still work.

```
# An example THE PROPER WAY, using a variable
npts = 1000
x = np.linspace(0,10,npts)
y = np.sin(x)
maxval=0.3
```

```
num = 0
for i in range(npts):
    if (y[i]<maxval):
        num += 1

print("The percentange of points in y less than 0.3 is %.2f %%" % (num/npts*100))
```

The big advantage here is that you can just change `npts` and all the code works immediately with no further changes!

Another GOOD option: replace hard coded numbers with automatically calculated values Here below is also another option: instead of using a variable `npts`, we can also use the `len()` function to automatically calculate the length of array x:

```
# An example using len()
npts=1000
x = np.linspace(0,10,npts)
y = np.sin(x)
maxval=0.3
num = 0

# For for loops, this is quite handy and easy to read
for i in range(len(y)):
    if (y[i]<maxval):
        num += 1

# However, for this line, I find the example above with npts a bit more readable: it is very o
# why I would divide by npts, but maybe not so immediately obvious why dividing by len(y)
# is the right thing to do...but it's fine, in particular if you add a comment explaining
# yourself.
print("The percentange of points in y less than 0.3 is %.2f %%" % (num/len(y)*100))
```

As mentioned in the code, this is quite common and probably always a good idea for the for loop: this way, you never accidentally loop over the end of the array!

(In python, this will either give an error, or, in the case of slicing, will give strange results since slicing applies "periodic" boundary conditions when indexing instead of giving an error...)

In the case of calculating the percentage, it is maybe a bit less obvious to a non-trained programmer that this is the right thing to do, in which case it is a good idea to add a comment to your code explaining what you're doing.

Summary: How to avoid hard coding You will make your code much more robust and maintainable by avoiding hard coding of values if you:

- Replace hard-coded values with variables
- Use functions that can automatically determine the appropriate value to use

Functions in Python & basics of plotting

Pre/Post-test This test is for testing your current skills in Python. You can use it in two ways:

- pre-test: to test your skills beforehand. If you are already proficient in Python, and can do this test within approximately 15 minutes, you can scan through the notebook rather than carefully reading each sentence.
- post-test: to test your skills after Notebook 2. Check whether you learned enough.

Analysing some data

As Eric passed the first test, he is now exposed to a more difficult challenge. He obtained a dataset consisting of five measurements. By plotting the data he expects that the underlying relation is $y = a \cdot x + b$, with a and b constants that need to be determined. As he still lacks the ability to have Python determine these coefficients, he just tries values for a and b and produces the corresponding graph. He then visually assesses the correctness of the values for a and b.

- plot the measurements in the graph as black dots
- make an educated guess for the coefficients a and b
- write a function that takes x, a and b as input and calculates y (
- make a test array in the given domain for x, using an interval of 0.1 (so [0.0, 0.1, ...])
- calculate for all values of x_test the value of y
- plot these data in the same graph as the measurements as red dashed line (r–)

```
x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([1.2, 3.6, 6.0, 8.4, 10.8, 13.2])

### YOUR CODE


x_test = ...
y_test = ...
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
NameError                                 Traceback (most recent call last)
Cell In[1], line 1
 - - - -> 1 x = np.array([0, 1, 2, 3, 4, 5])
      2 y = np.array([1.2, 3.6, 6.0, 8.4, 10.8, 13.2])
      4 ### YOUR CODE

NameError: name 'np' is not defined
```

Learning objectives In this notebook, we will explore the implementation of functions in Python.

After completing this notebook, you are able to:

- define functions with input parameters to execute a piece of code (use functions as “methods”)
- create and use functions that return a value (or multiple values)
- import and use functions from libraries / modules
- use Shift-Tab to bring up the help for a function from a library
- predict if a variable name in a function refers to a local variable or a global variable
- make a plot using the matplotlib library

Python Tutorial for Beginners 8: Functions

Various functions

Functions to save typing In programming, you often want to repeat the same sequence of commands over and over again.

One way to do this is to copy and paste the same piece of code over and over again. This is actually quite easy, but runs quickly into a problem: let's say you want to change a little bit what that code will do, then you need to change it in many places. If you change it in one place but forget in another, then your program might crash (ie. give an error). Or even worse, and even harder to debug the mistake may not give an error message but give you the wrong answer!

For this reason (among others), programming languages allow programmers to define "functions". Functions are pieces of code that you can give a name and then enable you to them use over and over again, without having to retype the code text.

As an example, let's say that we want to print out the value of a variables named `a` and `b` using a long sentence:

```
a = 6
b = 4
print("The value of variable a is", a)
print("The value of variable b is", b)

a = a/2
b = 3
print("The value of variable a is", a)
print("The value of variable b is", b)

a = a+1
b = 1.5
print("The value of variable a is", a)
print("The value of variable b is", b)

a = a -20
b = -1e4
print("The value of variable a is", a)
print("The value of variable b is", b)

a = a+1j
b = 1
print("The value of variable a is", a)
print("The value of variable b is", b)
```

To save a lot of typing, one can define a simple function to do this work for us:

```
def print_status():
    print("The value of variable a is", a)
    print("The value of variable b is", b)

a = 6
b = 4
print_status()

a = a/2
b = 3
print_status()

a = a+1
b = 1.5
print_status()
```

Or even more convenient:

```
def print_status(a,b):
    print("The value of variable a is", a)
    print("The value of variable b is", b)

a = 6
print_status(a+1,1.5)
print_status(a -20, -1e4)
print_status(a+1j,1)
```

To define a function, you use the following syntax:

```
def function_name():
     ...
```

Here, you replace the `...` with the code you want to function to execute. The Python code inside the function should be indented by starting each line with a tab. By default, adding a tab will produce 4 spaces in your code. You can also "indent" your code by manually adding spaces, but you must make sure to add 4 spaces each time. The Jupyter notebook will try to detect if you make a mistake in your indentation, and will sometimes color your text in red if it detects a mistake.

Tabs in Python are VERY IMPORTANT: python uses tabs to know which code is inside the function and which is not. If you make a mistake with the tabs in such a way that Python cannot understand what you mean, it will give you an `IdentationError`.

In notebooks, you can also select a line, or multiple lines, and then use `Tab` to increase their indentation level, or use `Shift-Tab` to decrease their indentation level.

```
def test_tab_and_shift_tab():
    some code
        that is indendented
    try selecting this text
    and then pushing tab
    and shift -tab
```

In the given example, it may not be such a big deal, but you can imagine that as the code in your function becomes more and more complicated, it will save you a lot of time. Also, imagine that I wanted to change the wording of the sentence I print: in the case with the function, I would only have to do this once, while in the example without function, I would have to manually change this at 5 different places.

`Exercise 2.1`

Write your own function that contains two lines of code. The first line should make a new variable `var2` that converts `var` to an integer. The second line of your code should print the value of `var2`.

Using this code, play around with the indentation (add extra tabs and spaces for example) to see how 'critical' Python is with indentation. For example: does three spaces work instead of `Tab`? Does one space work? What about `Tab` on the first line and three spaces on the second line? Can you make Python trigger an `IdentationError`?

```
var=3.5

# Your function and out below
```

Functions with input variables Let's say that we want to print out the status of variables that we do not know the name of ahead of time, as in the example above. Say we wanted to make a function that could print out a message with the status of value of ANY variable. How could we do this?

In the example above, our function explicitly printed out variables `a` and `b`. But this only works because I know in advance that the person using my function has defined variables `a` and `b`. But what if I want to print the value of variable `c`?

To allow functions to be more generic, and therefore more "reusable" in general, Python allows you to define "input variables" for your function. The syntax for this is the following:

```
def function_name(x):
    ...
```

When you do this, for the code INSIDE your function, a variable `x` will be defined that will have the value given by the input value given to the function by the user. Let's look at a specific example:

```
def print_status2(x):
    print("The value passed to the function is", x)

a = 1.5
print_status2(a)

a = 1+1j
print_status2(a)

print_status2(1.5323)
```

How does this work?

When the function `print_status(a)` is called, Python "sends" ("passes" in computer speak) the value of `a` to the function. Inside the function, Python creates a new (temporary) variable called `x`, that is defined ONLY while the function code is running. This temporary variable `x` is then assigned the value that was sent to the function, and then the code is executed. When the function is finished, the variable `x` is destroyed. (Try adding the code `print(x)` above outside the function and see what happens!)

Note, as you can see in the third example, the things you pass to functions do not even need to be variables! This is fine because the function only needs the value of the argument that is passed to the function.

Exercise 2.2

Ask the user to type a number x, calculate its square x^2 and return the value as a one-decimal float (peak at notebook 1 how to properly do so using %f).

```
# Your code here
```

Functions with multiple inputs Functions can also take multiple input variables. To do this, you put them all in between the brackets `()`, separated by commas. For example, with 3 variables, the syntax is:

```
def function_name(variable1, variable2, variable3):
    ...
```

You would then use this function in the following way:

```
function_name(argument1, argument2, argument3)
```

When you do this, inside the function, `variable1` will get assigned the value of `argument1`, `variable2` will get assigned the value of `argument2`, and `variable3` will get assigned the value of `argument3`. This matching of the position in the list is called matching by "positional order".

Note that there are several different names used for the "input variables" of a function: often, computer scientists will also use the name "input arguments" (or just "arguments), or "input parameters" (or just "parameters").

```
def print_status3(x, y):
    print("The value of the first input variable is ", x)
    print("The value of the second input variable is ", y)

print_status3(1,2)
print_status3(2.5,1.5)
print_status3(a, 2*a)
```

Exercise 2.3
Make a new function `print_status4()` that takes three variables as arguments and prints out messages telling the user the values of each of them (as above, but with three input variables). Test it to make sure it works.

```
# Your code here
```

Functions that return a value In addition to receiving values as inputs, functions can also send back values to the person using the function. In computer programming, this is called the "return value".

When you create a function, you can use the `return` command to specify what value should be sent back to the person using the function. Let's look at an example:

```
def my_formula(x):
    y = x**2 + 3
    return y
```

In this case it could be more convenient just to type:

```
def my_formula(x):
    return x**2 + 3
```

To "capture" the value returned by the function, you can assign it to a variable, or just directly "use" the result of the function if you want:

```
result = my_formula(3.5)
print(result)
print(my_formula(4.6))
```

Note that as soon as python sees the `return` command, it stops running the function, so any code after it will not be executed:

```
def myfunction(x):
    print("This gets printed.")
    return x**2 + 3
    print("This does not.")

print(myfunction(5))
```

If you want to send back more than one result to the user of your function, you can separate the results with commas when you use the `return` command.

```
def functie(x,y):
    return x**2, y**3

s, p = functie(2,4)
print(s,p)
```

Exercise 2.4 (a)
Write a function that takes two real numbers as input and returns the sum and product of the two numbers. In your function, try to send *both* of the calculated numbers back as a return value.

```
# Your function here
def product_and_sum(...):
    ...
```

Exercise 2.4 (b)
Now USE your function to calculate the sum and product of `a` and `b`, "capturing" the sum and product in variables `s` and `p`:

```
a = 1.5
b = 2.5

#...some code that uses the return value of your function to set variable s and p...

print("Sum is:", s)
print("Product is:", p)
```

Importing functions from libraries One of the big advantages of Python is that there are huge collection of libraries that include code for doing a huge number of things for you! We will make extensive use of the library `numpy` for numerical calculations in Python, and the library `matplotlib` for generating scientific plots. Beyond this, nearly anything you want to be able to do on a computer can be found in Python libraries, which is one of the reasons Python is so popular.

In order to make use of these libraries of code, you need to "import" them into the "namespace" of your kernel.

("Namespace" is Python-speak for the list of functions and variable names that you can find in the running copy of Python that is connected to your notebook.)

Here, we will show you a few examples of different ways of importing code into your notebook from a library (also called a "module"). For this, we will take the example we used already in Notebook 1: in the module `time`, there is a function called `sleep()` that will perform the task of "pausing" for a number of seconds given by the its argument.

You can find out more about the `time` module by looking at its documentation webpage:
https://docs.python.org/3/library/time.html
and specifically about the `sleep()` function here:
https://docs.python.org/3/library/time.html#time.sleep

Importing a whole module The simplest way to be able use the `sleep` function of the `time` module is to import it using the following command:

```
import time
```

You can see it has been imported by using the `%whos` command:

```
%whos
```

Once it has been imported, you can access all the functions of the module by adding `time.` in front of the function name (from the time module) in your code:

```
print("Starting to sleep")
time.sleep(5)
print("Done!")
```

If you import the whole module, you will have access to all the functions in it. To see what functions are in the module for you to use type `dir(time)`, which will generate this list.

Sometimes, if you will be using the functions from the module a lot, you can give it a different "prefix" to save yourself some typing:

```
import time as tm
print("Starting to sleep")
tm.sleep(5)
print("Done!")
```

We will use this a lot when using the `numpy` module, shortening its name to `np` when we import it, and also for the `matplotlib.pyplot` submodule, which we will shorten to `plt`. (These are also typically used conventions in the scientific community.)

Importing a single function You can also import a single function from a library:

```
from time import sleep
from datetime import datetime
```

When you do this, the function `sleep()` will be available directly in your notebook kernel "namespace" without any prefix:

```
start_timer = datetime.now()
print("Starting to sleep")
sleep(5)
print("Done!")
end_timer = datetime.now()
print(end_timer -start_timer)
```

Using `%whos`, we can now see that we have three different ways to use the `sleep()` function. We also used a self-built stopwatch showing how long it took to run some code. This is a very convenient way to improve your own code, speeding it up is especially important when using large chunks of code.

```
%whos
```

If you look around on the internet, you will also find people that will do the following

```
from numpy import *
```

This will import all the functions from numpy directly into the namespace of your kernel with no prefix. You might think: what a great idea, this will save me loads of typing! Instead of typing `np.sqrt()` for example, to use the square-root function, I could just type `sqrt()`.

While true, it will save typing, it also comes with a risk: sometimes different modules have functions that have the same name, but do different things. A concrete example is the function `sqrt()`, which is available in both the `math` module and the `numpy` module. Unfortunately, `math.sqrt()` will give an error when using numpy arrays (which we will learn more about in later notebooks).

If you import both of them, you will overwrite these functions by the second import, and if you're not careful, you will forget which one you are using, and it could cause your code to break. It will also "crowd" your notebooks namespace: using the `whos` function, you will suddenly see hundreds or even thousands of functions, instead of only just a module.

For these reasons, it is generally advised not to use `import *`, and it is considered poor coding practice in modern Python.

Shift-Tab for getting help Like the tab completion we saw in the first notebook, Jupyter also can give you help on functions you have imported from libraries if you type `Shift-Tab`.

Say I forgot how to use the `datetime()` function. If I type the word "datetime" and then push `Shift-Tab`, Jupyter will bring up a help window for that function.

Try it: click on any part of the word `datetime` in the following code cell and push `Shift-Tab`:

```
datetime
```

You can also find the same help as the output of a code cell by using the `help()` function:

```
help(datetime)
```

There are extensive online resources for many modules. The most used modules have helpful examples.

Exercise 2.5 a

Find help for the built-in functions `abs`, `int`, and `input`. Which of the help functions are easy to read? Which one does not provide such useful information (compared to the online documentation page)? (Put each help command in a separate cell)

```
# Your code here

# Your code here

# Your code here
```

Exercise 2.5 (b)

Import the function `glob` from the library `glob` and print its help information. What does the function `glob("../*")` do?

```
# run the help here

# your code here
```

Global variables, local variables, and variable scope In our first functions above, we saw a couple of examples of using variables inside functions.

In the first example, we used the variables `a` and `b` inside our function that we created outside our function, directly in our notebook.

In the second example, we used the "temporary" variable `x` inside our function.

These were two examples of different variable "scope". In computer programming, scope define the rules Python uses when it tries to look up the value of a variable.

In the slightly simplified picture we will work with here, variables can have two different types of "scopes": **global scope** and **local scope**.

If Python looks for a variable value, it first looks in the local scope (also called "local namespace"). If it does not find it, Python will go up into the global scope (also called the "global namespace") and look for the variable there. If it does not find the variable there, it will trigger an error (a `NameError` to be precise).

How do I create a global variable? By default, if you create a variable directly in your notebook (and not in a function in your notebook), it will always be **global**. So, actually, you've already created a bunch of global variables!

Any variables you define inside a function in your code will be a **local** variable (including the input variables automatically created if your function takes any arguments).

If you want to create a global variable inside a function, or make sure the variable you are referring to is the global variable and not the local one, you can do this by the `global` qualifier, which we will look at in a minute.

Let's take a look at this in more detail by analysing a few examples.

Example 1 Accessing a global variable inside a function

```
a1 = 5

def my_func():
    print(a1)

my_func()
a1 = 6
my_func()
```

In this example, when Python is inside the function `my_func()`, it first looks to see if there is a variable `a1` in the local scope of the function. It does not find one, so it then goes and looks in the global scope. There, it finds a variable `a1`, and so it uses this one.

Example 2 An example that doesn't work (unless you've run the next cell, in which case it will only fail again after you restart your kernel)

```
def my_func():
    print(b1)

my_func()
```

This code gives a `NameError` because there is no variable `b1` yet created in the global scope. If we run the following code cell and try the code above again, it will work.

```
b1 = 6
```

Here you can see one risk of languages like Python: because of the persistent memory of the kernel, code can succeed or fail depending on what code you have run before it... This is why you kill your kernel, clear all outputs and re-run the entire script!!

If you want to see the error message above again, you can delete variable b1 using this code and run it again:

```
del b1
```

Example 3 Variables defined in the local scope of a function are not accessible outside the function

```
def my_func():
    x = 5

my_func()

print(x)
```

Example 4 Variables passed to functions cannot be modified by the function (more on this later when we look at more complicated data structures...sometimes this is different)

```
def my_func(a):
    a = 6

a=5
my_func(a)
print(a)
```

This one is a bit subtle (mega-confusing?) because we re-used the same name `a` for the local variable in the function as the global variable outside of the function. However, the operation is quite logical. When the function code starts running, it creates a `local` variable `a` to store the value it received. And now, because there is already a local variable called `a`, using `a` in the function refers to the `local` variable `a`, not the `global` variable `a` we define before calling the function.

Example 5 This one is a tricky one.

```
a = 6

def my_func():
    a = 7

print(a)
my_func()
print(a)
```

It would seem that the function would refer to the global variable `a` and therefore change it's value. However, it is tricky since we first use `a` in the function in an assignment. An assignment in python will automatically create a variable if it does not exist, and so python creates a new variable named `a` in the local scope. The name `a` inside the function now refers to this newly created local variable, and therefore the global variable will not be changed. In fact, this guarantees that you cannot change global variables inside a function, unless you use the `global` qualifier shown in the next example.

Example 6 If you want to make sure that the `a` inside your function is referring to the global variable `a`, you can include the line `global a` inside your function to tell python that you mean the global variable `a`.

```
a = 6

def my_func():
    global a
    a = 7

print(a)
my_func()
print(a)
```

Note that in general, it is considered bad programming practice to use (too many) global variables. Why? When you write longer and bigger sections of code, it is easier to understand what is going on in your function if your function uses only local variables and communicates back and forth using input parameter and return variables. Using too many global variables in a function can be confusing because they are defined in a different place in your code and so you don't have a good oversight of them. (Bigger projects can easily have 10,000+ lines of code!)

In computer science, this is a topic of often intense debate (resulting in what nerds refer to as a flame war), with global variables being branded as "dangerous" like in this stack exchange post:

https://stackoverflow.com/questions/423379/using-global-variables-in-a-function

But I personally agree with the comments in this post that "global variables have their place but should be used sparingly".

Summary of the rules for global and local variables:

- If a local variable of the same name exists or is created by Python (by assignment, for example), then python uses the local varible
- If you try to use a variable name that does not exist locally, Python checks for a global variable of the same name
- If you want to change the value of a global inside a function, then you must use the `global` statement to make it clear to Python than you want that name to refer to the global variable

`Exercise 2.6 (a)` Make a function, dependent on one variable, that may be any differentiable function. You can use any combination of standard functions (e.g. np.sin, np.exp). Use this function to print the values f(0) and f(1).

```
import numpy as np
def f(x):
    return ...
```

Exercise 2.6 (b) We want to find the tangent line at a given value of our function. For this, you may approximate the slope by $\frac{\Delta y}{\Delta x} = \frac{f(a+\epsilon)-f(a-\epsilon)}{2\epsilon}$, where epsilon is a sufficiently small number (note that Python's float type is limited to a number of digits, so ϵ should not be smaller than 1e-8) and a is the point at which we want the tangent line. Make a function that, given a certain point, returns the slope.

```
def determine_slope(a, epsilon):
    dy = ...
    dx = ...
    return dy/dx
```

Exercise 2.6 (c) We want to write the tangent line in the form $y = mx + b$. Make a function that, given a point, print outs the tangent line. Hint: use your previously defined functions.

```
def print_tangent(a, epsilon):
    m = ...
    b = ...
    print(...)

epsilon = ...
a       = ...
print_tangent(a, epsilon)
```

Plotting data and functions with Matplotlib We already encountered a plot in one of the first assignments. Physicist always want to plot the data to see what it looks like! The Matplotlib library allows us just to do that. https://matplotlib.org

Python - Matplotlib Tutorial for Beginners

Plotting basics Specifically, we will need the pyplot module of matplotlib:
https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html#module-matplotlib.pyplot
We will import it with the shortcut / "name" / "prefix" plt:

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

The routine for making line plots of your data is `plt.plot()`:
https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

We present you a plot with some additional features. Read the code, change values, colors and so on and see the result. Try to understand how each of the changes affect your plot (predict - observe - explain). Note that we included many optional elements, without these Python can perfectly create a proper graph for you.

```
x = np.arange(0,10,1)
y = 2*x**2
z = 15*x+2

plt.rcParams['figure.dpi'] = 300 #used to set proper display resolution

plt.figure(figsize=(5, 4))
plt.xlabel("$x$ (m)")
plt.ylabel("$t$ (s)")

plt.plot(x,y,'k.',markersize='4',label='quadratic function')
```

```
plt.plot(x,z,'r+',label='linear function')

plt.legend(loc='upper left')

plt.xlim(0,10)
plt.ylim(0,180)
plt.grid() #adds a grid to the plot, not always appreciated by scientists.
plt.axhline(0, ls=' -', lw=1, c='grey')
plt.axvline(0, ls=':', lw=1, c='grey')

plt.savefig('my_first_fig.pdf')
plt.show()
```

If you look in the file browser, you will now see a file my_first_fig.pdf. If you open it, you will see that is a high-quality vector graphics PDF.

Exercise 2.7

Below we have a dataset where we expect the data to be based upon the formula $y = a \cdot x + b$. To check whether this is correct, carry out the following tasks:

- plot the raw data as black dots
- write a function for the formula with input variables a and b
- make an estimated guess for the values of a and b
- use the test variable as input to calculate the corresponding values for y
- plot these values for x and y in the same graph, use a red dotted line

```
x = np.array([1.1, 3.4, 4.7, 5.3, 6.8])
y = np.array([1.05, 1.62, 1.95, 2.10, 2.47])

x_test = np.linspace(0,1.2*max(x),1000)

def func(x,a,b):
```

Exercise 2.8 (a)

Coulomb's law reads $F = k_e \frac{q_1 \cdot q_2}{r^2}$ in which k_e is Coulomb's constant $k_e = 8.988{*}10^9$ N m$^2\cdot$ C^{-2}, q denotes the magnitude of the charge (could be negative value as well!) and r the distance between the charges.

Write a function that calculates the force between a given number of (opposite) charges and the distance between them.

```
def F(q1,q2,r):
    #your code
```

Exercise 2.8 (b)

Make a plot that shows the force as function of distance between a proton and an electron in the range 0.5 - 6 ångström.

```
#your code

plt.figure()
#your code
```

Solutions to Exercises Exercise 2.1

```
var=3.5
```

```
# Example: one  space is actually enough! But is discouraged, as you can see
# by the fact the the notebook made it red colored.
def myfunction():
 var2 =  int(var)
 print(var2)

# Once you've started a code block though with a specific indentaton, then you cant change
# it anymore.
def myfunction():
    var2 =  int(var)
   print(var2)
```

Exercise 2.2

```
def myfunction(var):
    var2 =  int(var)
    print(var2)
```

Exercise 2.3

```
def print_status4(x, y, z):
    print("The value of the first input variable is ", x)
    print("The value of the second input variable is ", y)
    print("The value of the third input variable is ", z)

print_status4(1,2,3)
```

Exercise 2.4 (a)

```
# Your function here
def product_and_sum(a,b):
    return a*b, a+b
```

(b)

```
a=1.5
b=2.5

p,s = product_and_sum(a,b)

print("Sum is:", s)
print("Product is:", p)
```

Exercise 2.5 (a)

```
help(abs)

help(int)

help(input)
```

(b)

```
from glob import glob
help(glob)
```

```
glob("../*")
# It returns list of files and folders in the parent directory of that
# in which this notebook is stored
```

Exercise 2.6 (a)

```
def f(x): # All differentiable functions are correct
    return np.sin(x)

print(f(0), f(1))
```

Exercise 2.6 (b)

```
def determine_slope(a, epsilon):
    dy = f(a + epsilon) - f(a - epsilon)
    dx = 2 * epsilon
    return dy/dx
```

Exercise 2.6 (c)

```
def print_tangent(a, epsilon):
    m = determine_slope(a, epsilon)
    b = f(a) - m*a
    print("The tangent line of the function is: y = {}x + {}".format(round(m,3), round(b,3)))

epsilon = 0.01   # Depending on the function, a larger epsilon may also be sufficient or a sma
a       = 0      # Again, an arbitrary choice.

print_tangent(a, epsilon)
```

Exercise 2.7

```
x = np.array([1.1, 3.4, 4.7, 5.3, 6.8])
y = np.array([1.05, 1.62, 1.95, 2.10, 2.47])

x_test = np.linspace(0,1.2*max(x),1000)

def func(x,a,b):
    return a*x+b

y_test = func(x_test,.27,0.7)

plt.figure()

plt.plot(x,y,'k.', label='measurements')
plt.plot(x_test,y_test,'r - -', label='fitline')

plt.xlabel('$x$')
plt.ylabel('$y$')

plt.xlim(0,1.1*max(x))

plt.show()
```

Exercise 2.8 (a)

```
k = 8.988e9
def F(q1,q2,r):
    return k*q1*q2/r**2
```

Exercise 2.8 (b)

```
r = np.arange(0.5E -10,6E -10,0.1E -10)
F_q = F(1.602E -19, -1.602E -19,r)
plt.figure()
plt.plot(r,F_q,'k.')
plt.xlabel('$r$ (m)')
plt.ylabel('$F$ (N)')
plt.show()
```

Program flow control with Conditional Statements and Loops

Pre/Post-test This test is for testing your current skills in Python. You can use it in two ways:

- pre-test: to test your skills beforehand. If you are already proficient in Python, and can do this test within approximately 15 minutes, you can scan through the notebook rather than carefully reading each sentence.
- post-test: to test your skills after Notebook 3. Check whether you learned enough.

Including password protection

Although the supervisors are impressed by the work Eric is doing, his co-workers are not as happy by the quick progress Eric is making. Due to the large number of error messages in his code recently, he is convinced someone is messing with his code... Therefore he decides to write a program that asks the user to enter the password and checks whether it is correct. As a typo might occur, the user has three tries, after which the program should shut down. After every try tell the user whether the password is correct or not and how many tries are left. If the user has not provided the right passwork within the required number of attempts, provide a warning message.

Write the code.

```
password = 'practicum123'
tries = 0

### Your code
```

Learning objectives In this notebook, we will learn how to control the flow of execution of Python code using conditional statements and loops.

After completing this notebook, you are able to:

- create conditional branching statements using `if`, `elif`, and `else`
- formulate conditions using `==`, `<`, `>`
- combine logical conditions using `and` and `or`
- create a `while` loop that exits on a particular condition
- create a `for` loop that loops over a range of numbers using the `range()` command
- exit loops using the `break` command
- skip the rest of a code block in a loop using `continue`

Conditional statements A powerful concept in programming languages are pieces of code that allow you to control if segments or your code are executed or not based of the value of variables in your code.

In this section, we will look at some of the features of the Python language for conditional execution of your code.

Python Tutorial for Beginners 6: Conditionals and Booleans - If, Else, and Elif Statements

The `if` statement We have already seen an example of the `if` statement in the previous notebook. The syntax of the if statement is as follows:

```
if expression:
    code block to be executed if the expression is "True"
```

Here, `expression` is a piece of code that evaluates to either the value `True` or `False`. Here are a few simple examples:

```
if True:
    print("True is true")
```

```
if True:
    print("The code block")
    print("after 'if' statements")
    print("can have multiple lines.")

if False:
    print("False is not True (this will not be printed)")

if not False:
    print("not False is True")
```

Exercise 3.1
Try out the following conditional statements, filling in the values specified to replace the `--` in the code cells.

Check if the integer `1` is true:

```
if - - -:
    print('The expression is true')
```

Check if `103.51` is true:

```
if - - -:
    print('The expression is true')
```

Check if `-1` is true:

```
if - - -:
    print('The expression is true')
```

Check if condition statements work with boolean variables:

```
a = True
if - - -:
    print('The expression is true')
```

Check if conditional statements work with numerical variables:

```
b = -73.445
if - - -:
    print('The expression is true')
```

Comparison and test operators In the above, the `expression` in the `if` statements were all directly values that are either True or False (except for the example of `not False`).

More generally, however, the `expression` can also include comparisons. Some examples of numerical comparisons are given here below:

```
if 5 == 5:
    print("A double equal sign '==' compares the two sides of itself and gives the")
    print("result 'True' if the two sides have the same value")

a = 5
if a == 5:
    print("== also works with variables")

a = 5
if 5 == a:
    print("The variable can be on either side")
```

```
a = 5
b = 5
if a == b:
    print("and == also works with two variables")

a = 5
if a = 5:
    print("This will give an error! A single = is an assignment operator, not a conditional te
    print("To check if a is equal to 5, you need to use the double equal sign ==")

a = 5
if a == 6:
    print("This will not be printed.")

a = 5
if 2*a == 10:
    print("You can also use mathematical expressions")
```

In addition to the `==` operator, there are also several other comparison operators such as `<`, `>`, `>=`, `<=`, `!=`

Exercise 3.2
Test the operators `<`, `>`, `>=`, `<=`, `!=` by trying them in an if statement with numerical values.

```
if - - - < - - -:
    print('The expression is true')

if - - - <= - - -:
    print('The expression is true')

if - - - > - - -:
    print('The expression is true')

if - - - >= - - -:
    print('The expression is true')

if - - - != - - -:
    print('The expression is true')
```

Logical operations Python also allows you to build the `expression` out of logical combinations of several conditions using the keywords `and`, `or`, and `not`. The value of these operators is as follows

- `and` evaluates to True if both conditions are True
- `or` evaluates to True if either condition is True
- `not`evaluates to True if condition is not True

Below are a few examples.

```
if True and False:
    print("This will not be printed")

if True or False:
    print("This will be printed")

if not 5 > 6:
    print("We have already seen 'not' in previous examples. Here, we also combine with an comp

if not 5 != 5:
    print("This is a very silly example (hard to read, bad coding!)")
```

```
if 5 < 6 and 10 > 9:
    print("An example of combining conditional expressions with 'and'")
```

Exercise 3.3
Try out the following examples using the if statement form from above for the conditions
(a) Check if 5 is smaller than 6, AND 10 is smaller equal than 9:

```
# Your code here
```

(b) Do you think that the statement `not False or True` evaluates to `True` or `False`? Try it out and see:

```
# Your code here
```

To understand what happened in part (b), we have to know if Python first performs the operation `False or True` or if it performs the operation `not False` first. The rules for which order Python does things, in can be found in the documentation for operator precedence. In the example above, we can see that the `not` operator had precedence and Python performed the `not` before it performed the `or`.

What if I wanted to have Python perform the `or` first? You do this by enclosing `True or False` in brackets:

```
if not (False or True):
    print("not (False or True) is False so this will not be printed")
```

The elif and else statements In Python, you can combine the `if` statement with `elif` and `else` commands in a chain in order to allow you to take actions in the case that the starting `if` statement was false.

`elif` (else if) is a command that allows you to check another condition if the condition of the starting `if` is `False`. Note that if the `if` criterion is `True`, the `elif` statement will be skipped and not checked.

`else` is a command that allows you to execute some code if all of the `if` and all the `elif`s are are `False`.

Note that to be part of the "chain", all the `elif`s and the last `else` must follow directly after each other's code blocks with no other code in between. And a new `if` always starts a new chain. You can see how this works in the following examples:

```
a = 5
if a < 6:
    print("a is less than 6")
else:
    print("a is not less than 6")

a = 5
if a<6:
    print("the 'if' statement found that a is less than 6")
elif a<6:
    print("this will never get printed!")

a = 5
if a<6:
    print("the first if found that a is less than 6")
if a<6:
    print("unlike elif, a second if will get executed.")
```

Since the code inside your `if` code block is just regular code, you can also add another if statement inside that code block. This creates a nested `if` statement inside your first one:

```
# example of a nested if -statement
a=4
if a<6 and a>=0:
    if a>3:
        print("the value of a is 4 or 5")
    else:
        print("the value of a is 0, 1, 2, or 3")
else:
    print("none of these are the case")
```

Exercise 3.4
Practice the use of the if-elif-else statement with the following exercise by filling in the missing conditional statements. You must use all three of `if`, `elif` and `else`. You also can not use the `end` operator.

```
def check_number(a):
    ...conditional statement number 1...
        print(a, "is less than or equal to 5")
    ...conditional statement number 2...
        print(a, "is between 5 and 10")
    ...conditional statement number 3...
        print(a, "is greater than or equal to 10")

# Testing your function
check_number(1)
check_number(5)
check_number(7)
check_number(10)
check_number(15)
```

Loops Loops are a construction in a programming language that allow you to execute the same piece of code repeatedly.

In Python there are two types of loops: `while` loops and `for` loops. We will look at `while` loops first and then move on to the more common `for` loops.

The while loop Using the `while` command, you can specify that a block of code gets executed over and over again until a certain condition is satified:

```
while expression:
    code...
```

As long as `expression` is true, then the code block will be executed over and over again.

A simple example where we use a while loop to count to 10:

```
i = 1
while i <= 10:
    print(i)
    i = i+1
```

In this example here, we use a while loop to add up all the numbers between 1 and 10:

```
i = 1
s = 0

while i<= 10:
    s = s + i
    i = i+1

print("Sum is", s)
```

Note that with `while` loops, you have to be careful to make sure that your code block does something that results in your `expression` becomes false at some point, otherwise the loop will run forever.

For example, when initially I wrote the above cell to calculate the sum, I forgot the `i = i+1` line of code:

```
i = 1
s = 0

while i<= 10:
    s = s + i

print("Sum is", s)
```

This code will never finish: it will go into an infinite loop! (You can see that it is still running because the `*` beside the `In` text never gets turned into a number.) Note that for this will have to manually stop the kernel using the stop button in the toolbar or it will run forever...

When should I use a while loop? For both of the examples above, one would typically use `for` loop, which we will see in the next section. One place where `while` loops are very useful is if you do not know ahead of time how many iterations of the loop you will need.

For example, let's consider the following sum:

$$S(n) = \sum_{i=1}^{n} \sin^2(i) \tag{3}$$

Let's say we want to know: for which n does the sum exceed 30? We can easily check this with a `while` loop:

```
from numpy import sin

i = 1
s = 0

while s<30:
    s += sin(i)**2
    i += 1

print("Sum is", s)
print("i is", i)
```

Note here I have also used a new operator `+=`: this is a special assignment operator that increments the variable by the specified amount. `a += 1` is equivalent to `a = a + 1` (it just saves a bit of typing, remember: programmers are lazy...).

Exercise 3.5

Write a function to calculate the factorial of a number using a while loop. The factorial, denoted with

!, is the product of all numbers up to that number, i.e. $4! = 1 \cdot 2 \cdot 3 \cdot 4$. Note that perhaps unexpectedly for you 0!=1 for which your function also have to give an answer.

If your code doesn't seem to stop running, you may need to "debug" your code to see where your mistake is (I did). A handy way to do this, for example, is to add a `print(i)` *statement inside your while loop: it will enable you to see if the variable* `i` *is doing what you think it should be...*

```
def factorial(a):
    ...
    while(...)

print(factorial(4))
print(factorial(0))
```

The for loop The `for` loop is designed to execute a piece of code a fixed number of times. The syntax of the `for` loop is:

```
for i in (a, b, c, d, ...):
    code block
```

In each subsequent iteration of the for loop, the variable `i` is assigned next value that is supplied in the list values.

We can repeat our sum calculation above using the following `for` loop:

```
s = 0

for i in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10):
    s += i
    print(s)
```

It would, however, be much more convenient if we would not have to type all the numbers. Our could be something like this:

```
s = 0
i = np.arange(1,11)
for i in i:
    s+=i
    print(s)
```

And that brings us to the range function...

Exercise 3.6

Calculate the sum $\sum_{i=0}^{n} \sin^2(i)$ for n=10 using a `for` loop.

```
# Your code here
for...:
```

The range() function You can imagine that if we wanted to perform this sum up to 100, it would be very annoying to type out all of the numbers. For this, Python has a convenient function `range()`, than can automatically generate ranges of numbers for you!

The `range` function can be used in a couple of different ways, which we will look at here. We will see, however, that `range` does some funny things, which is related to the fact that Python "counts from zero" *(more on this later).*

But let's look just at some concrete examples for now:

`range(N)`: Print a range of N numbers starting from zero If you give `range` only one argument, it will give a range of numbers starting from 0, and a total number of numbers determined by the argument.

As an explicit example, `range(10)` is equivalent to `(0,1,2,3,4,5,6,7,8,9)`:

```
for i in range(10):
    print(i)
```

`range(N,M)`: Print a range of numbers starting from N and ending at M-1 If you give `range` two arguments `range(N,M)`, it will give a range of numbers starting from `N` and stopping at `M-1`.

```
for i in range(1,11):
    print(i)
```

You might ask: why not stop at `M`? If you say the words "range from N to M", you would think that this range should include M?

There are two reasons I can think of:

1. For programmers, it is nice that `range(0,10)` and `range(10)` do the same thing
2. It will be handy later that `range(j, j+N)` will then always give you `N` numbers in the range

Maybe there are more, I'm not sure (you'd have to ask the nerds who wrote Python...). But in any case, you just need to remember that `range(N,M)` stops at `M-1`...

`range(N,M,S)`: Print a range of numbers less than M, starting from N, with steps of S This is like the last one, but now with the chance to change the steps:

```
for i in range(1,11,2):
    print(i)
```

Note that the `range` function works only with integers: `range(1,11,0.5)` is not allowed. (For floating point ranges, you can use the `numpy` function `arange`, more on that later...)

`Exercise 3.7`

Calculate the sum of all numbers from 1 to 100 using a `for` loop with the `range()` function. Compare it to the famous value that Gauss calculated in class as an elementary school student.

```
# Your code here
```

Using for loops with things other than ranges of numbers In the examples above, we looked at using `for` loops to iterate through a list of integers.

In Python, however, `for` loops are much more flexible than only iterating over numbers: `for` loops can iterate over any iteratable object, including 1-D numpy arrays, which we will see in later notebooks.

But, as an example, here is a piece of code that uses the `numpy` random number generator to calculate the sum of 10 random integers between 0 and 100:

```
from numpy.random import randint

s = 0
for x in randint(0,100,10):
    print(x)
    s += x

print()
print("Sum is ", s)
```

Manually exiting or skipping a loop using break and continue In addition to the examples we saw above, Python offers two extra commands for controlling the flow of execution in a loop: `break` and `continue`

`break` is a command you can use to force the exiting of either a `for` loop or a `while` loop. For example, you can replace the while loop above using something like this:

```
s = 0
i = 0
while True:
    s += sin(i)**2
    i += 1
    if s > 30:
        break

print(s)
```

It looks funny at first, because `while True` looks like you will loop forever! But of course, you are saved by the `break` statement.

Using `break` statements can sometimes make your code more readable, particularly if you want to be able to exit the loop under different conditions, or have an exit condition trigger a certain piece of code. Here is an example of two conditions:

```
from numpy.random import randint

i = 0
found_five = False
max_tries = 10

while True:
    i += 1
    n = randint(0,30)
    if n == 5:
        found_five = True
        break
    if i >= max_tries:
        break

if found_five:
    print("We found a 5 after", i, "tries")
else:
    print("We didn't find a 5 in the maximum number number of tries (", max_tries, ")")
```

The statement `continue` is used if you want to skip the rest of the code in the code block and restart the next loop. This can sometimes be useful if as a way of avoiding adding an `else` statement. (Remember, programmers are lazy typers...and it can be useful if you have a big long complicated block of code...)

```
s = 0
for i in randint(0,30,100):
    if (i % 5) == 0:
        continue
    s += i

print("The sum of +/ -100 random numbers between 0 and 30 excluding those that are divisible b
```

This is probably not a great example (if you are smart you can do this with less typing), but in any case, you now know what `continue` does.

Exercise 3.8

Write code that creates a variable `i` with an initial value of zero. Use a while loop that increments `i` by one as long as `i` is less than one million. Have your loop stop if `i` satisfies the condition $i(i-10) = 257024$. Have your code print the `i` that satisfies this condition (if there is one).

```
# Your code here
```

Exercise 3.9

In order to make sure that the seasons stay aligned with our calendars, leap years of 366 days have been introduced. A leap year is a year which is an integer multiple of 4 (except for years evenly divisible by 100, but not by 400), Leap year.

Make a function to check whether a certain year is a leap year using if/else statements to return True for leap years and False for normal years.

```
def leap_year(year):

    return

# Check if your function works. The years 4, 2000, 2012 are examples of leap years,
# whereas the years 2021 and 2100 are not.
years = [1, 4, 100, 400, 2000, 2012, 2020, 2021, 2024, 2100]
for year in years:
    print(year, leap_year(year))

#autotestingscript
import numpy as np
test = []

for year in years:
    test.append(leap_year(year))

answercheck = [False, True, False, True, True, True, True, False, True, False]

#auto check
np.testing.assert_array_equal(test, answercheck, 'error', True)
```

Exercise 3.10

Sometimes we want to find the peaks in our data. One way to do this is to go through the array and check for every value if the values left and right are smaller.

Write some code that finds the peak(s) of the array and that prints the index and value of all peaks.

```
y = np.array([2, 2, 3, 4, 5, 4, 3, 8, 6, 4])    # dummy data

# your code here
```

Exercise 3.11

To keep your computer safe from others, it is probably password protected. Usually you have a few tries before the password entry is disabled for some time.

You can enter your password three times before a timeout is given. If the timeout (sleep) is over, you have three other tries. The timeout time in seconds is given by $T = 60 + i^3$, where i is the number of tries.

Write the code. 3pt

```
#your code here

password = 'practicum123'
tries = 0
```

Solutions to exercises Exercise 3.1

```
if 1:
    print('The expression is true')

if 103.51:
    print('The expression is true')

if -1:
    print('The expression is true')

a = True
if a:
    print('The expression is true')

b = -73.445
if b:
    print('The expression is true')
```

Exercise 3.2

```
if 5 < 5.1:
    print('The expression is true')

if 5 <= 5:
    print('The expression is true')
```

```
if 5.1 > 5.1:
    print('The expression is true')

if 5.1 >= 5.1:
    print('The expression is true')

# "!=" is "not -equals"
if 5 != 5.1:
    print('The expression is true')
```

Exercise 3.3 (a)

```
if 5 < 6 and 10 <= 9:
    print("i don't think this is true")
```

(b)

```
if not False or True:
    print("it is true?")
else:
    print("it wasn't true?")
```

Exercise 3.4

```
def check_number(a):
    if a <= 5:
        print(a, "is less than or equal to 5")
    elif a < 10:
        # note we don't need to test if it's greater than 5 since the first if already took
        # care of that for us
        print(a, "is between 5 and 10")
    else:
        print(a, "is greater than or equal to 10")

# Testing your function
check_number(1)
check_number(5)
check_number(7)
check_number(10)
check_number(15)
```

Exercise 3.5

```
def factorial(a):
    # f = our factorial result
    f = 1
    i = 2
    while(i<=a):
        f *= i
        i += 1
    return f

print(factorial(4))
print(factorial(2))
```

Exercise 3.6

```
# Your code here

s = 0
for i in range(11):
    s += np.sin(i)**2

print("Sum is", s)
```

Exercise 3.7

```
s = 0

# Note the range should end at 101!
# It should also start at 1: range(101) will also execute
# the loop with i=0 (which won't matter here...)
for i in range(1,101):
    s += i

print(s)
```

Exercise 3.8

```
# Solution

maxnum=1e6
i=0
while i <= maxnum:
    if i*(i -10)==257024:
        break
    i+=1

print(i)
```

Exercise 3.9

```
def leap_year(year):
    if year % 4 == 0:
        if year % 100 == 0:
            if year % 400 == 0:
                return True
            return False
        return True
    return False
```

Exercise 3.10

```
## Solution
import numpy as np
y = np.array([2, 2, 3, 4, 5, 4, 3, 8, 6, 4])

for i in range(1, len(y) -1):
    if y[i -1] < y[i] and y[i+1] < y[i]:
        print('index:', i, '\t value:', y[i])
```

Exercise 3.11

```
password = 'practicum123'
tries = 0

from time import sleep

while True:
    input_pw = input('What is the password?')
    if input_pw == password:
        break
    if (tries+1)%3==0:
        sleep(60+tries**3)
    tries += 1
```

Scientific Computing in Python with Numpy

Pre/Post-test This test is for testing your current skills in Python. You can use it in two ways:

- pre-test: to test your skills beforehand. If you are already proficient in Python, and can do this test within approximately 15 minutes, you can scan through the notebook rather than carefully reading each sentence.
- post-test: to test your skills after Notebook 4. Check whether you learned enough.

Calculating a derivative Eric is asked to develop a tool for people that need to do some mathematical calculations. He is asked to write a function that calculates and plots the derivative of a given function. The derivative of a continuous function $f(x)$ can be approximated by $f'(x) = \frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$ for some small value of ϵ. As Eric knows that the derivative of $f(x) = \sin(x)$ is $f'(x) = \cos(x)$, he uses this function to test whether his function works correct.

- Make an array in the domain [0, 2*π] with 1e4 even spaced values.
- Plot the function $f(x)$ for this domain.
- Write a function that calculates the derivative using the approach above.
- Plot the graph of the derivative in the same figure.
- Test the correctness of the function by using any other input function.

```
### Your Code
```

Learning objectives Numpy (Numerical Python) is a library designed for performing scientific computing in Python.

In this notebook, we will introduce numpy arrays, a data structure introduced in numpy for working with vectors and matrices. We will explore how to create them, how to manipulate them, and how to use them for efficient numerical calculations using numpy functions.

After completing this notebook, you are able to:

- create (multidimensional) numpy arrays from a list of numbers
- use indexing and slicing with (multidimensional) numpy arrays
- iterate over a numpy array
- perform mathematical operations on numpy arrays
- use functions for creating arrays (eg. `np.zeros()`, `np.linspace()`, `np.random.random()`
- use numpy functions for vectorized calculations
- to demonstrate the speed increase of vectorized calculations using `time()`

Learn NUMPY in 5 minutes - BEST Python Library!

Numpy Arrays You have encountered numpy arrays in previous notebooks. If you don't remember, open the first notebook and read the section where we introduce numpy arrays.

To use numpy arrays, we first need to import the numpy library, which we will do using the shortened name "np":

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

Now that we have imported numpy, we can use functions in numpy to create a numpy array. A simple way to do this is to use the function `np.array()` to make a numpy array from a comma-separated list of numbers in square brackets:

```
a = np.array([1,2,3,4,5])
print(a)
```

Note that numpy does not make a distinction between row vectors and column vectors: they are just vectors.

Look at the cell below, what is the difference with the cell above?

```
a = np.array([1,2,3,4,5],dtype=float)
print(a)
```

Indexing arrays (and counting from zero) One useful thing about arrays is that you can access the elements of the array using square brackets:

`a[n]` will give you the n-th element of the array `a`.

This process of extracting a single element from the array is called **indexing**.

Note that here we encounter for the first time what is known as the python **counting from zero** convention. What is the counting from zero convention? In the example above, we created an array:

```
a = np.array([1,2,3,4,5])
```

The first element of `a` is `1`. You might think that if you want to access the first element of `a`, you would use the notation `a[1]`. Right?

Let's try it:

```
print(a[1])
```

WRONG! Why? Because the makers of Python decided to start counting from zero: the first element of a tuple `a` is actually `a[0]`.

(This is a long-standing discussion among computer scientists, and the convention is different in many different languages. There are advantages and disadvantages of both, and even essays written about it...but in any case, Python chose to start arrays at zero.)

This also helps better understand the `range()` function: for example, to loop over all the elements in `a`, I can use this code:

```
for i in range(len(a)):
    n = a[i]
    print('a[%d] is %d' % (i,n))
```

Here the `len` function returns the length of the array `a`. As we saw before, Python has very smart `for` loops that can automatically iterate over many types of objects, which means we can also print out all the elements of our array like this:

```
for n in a:
    print(n)
```

In Python, if you try to index beyond the end of the array, you will get an error:

```
a[5]
```

(Remember: indexing starts at zero!)

Python also has a handy feature: negative indices count backwards from the end, and index `-1` corresponds to the last element in the array!

```
a[ -1]
```

```
a[ -2]
```

We can also use indexing to change the values of elements in our array:

```
print(a)
a[2] = -1
print(a)
```

Exercise 4.1

Set the first three, and the last two, entries of the following array to zero:

```
a = np.array([1,2,3,4,5,6,7,8,9,10,11,32,55,78,22,99,55,33.2,55.77,99,101.3])

#some code to set the first three and last two entries to zero

print(a)
```

Slicing numpy arrays Numpy arrays also support a special type of indexing called "slicing" that does not just return a single element of an array, but instead returns a whole part of array.

To do this, I put not just a single number inside my square brackets, but instead two numbers, separated by a colon `:`

`a[n:m]` will return a new tuple that consist of all the elements in `a`, starting at element `n` and ending at element `m-1`.

Let's look at a concrete example:

```
a = np.array(range(10))
print(a)
print(a[0:5])
```

The notation `a[0:5]` has "sliced" out the first five elements of the array.

With slicing, you can also leave off either `n` or `m` from the slice: if leave off `n` it will default to `n=0`, and if you leave off `m`, it will default to the end of the array (also the same as `m=-1` in Python indexing):

```
print(a[:5])

print(a[5:])
```

Also handy: you can can have Python slice an array with a "step" size that is more than one by adding another `:` and a number after that. Find out its operation using:

```
print(a[0:10:2])
```

Fun: you can also use negative steps:

```
print(a[ -1: -11: -1])
```

And finally, unlike indexing, Python is a bit lenient (merciful) if you slice off the end of an array:

```
print(a[0:20])
```

Exercise 4.2

Slicing can also be used to *set* multiple values in an array at the same time. Use slicing to set first 10 entries of the array below to zero in one line of code.

```
a = np.array(range(20))+1
print(a)
#some code that sets the first 10 entries to zero

print(a)
```

Exercise 4.2*

The same exercise as 4.1 but now code in a smarter way.

```
#your code here
```

Mathematical operations on arrays An advantage of using numpy arrays for scientific computing is the way they behave under mathematical operations. In particular, they very often do exactly what we would want them to do if they were a vector:

```
a = np.array([1,2,3,4,5])
print(2*a)

print(a+a)

print(a+1)

print(a -a)

print(a/2)

print(a**2)
```

What about if I multiply two vectors together?

In mathematics, if I multiply two vectors, what I get depends on if I use the "dot product" or the "outer product" for my multiplication:

Row and column vectors#Operations

The "dot product" corresponds to multiplying a column vector by a row vector to produce a single number. The "outer product" (also called the "tensor product") corresponds to multiplying the column vector by the row vector to make a matrix.

Question: If I type `a*a`, or more generally `a*b`, does Python use the inner or outer product?

It turns out: it uses **neither!** In Python, the notation `a*a` produces what is commonly called the "element-wise" product: specifically,

`a*b = [a[0]*b[0] a[1]*b[1] a[2]*b[2] ...]`

(Mathematically, this has a fancy name called the Hadamard product, but as you can see, despite the fancy name, it's actually very simple...)

We can see this in action here:

```
print(a*a)
```

What if I actually want the dot product or the outer product? For that, Python has functions `np.dot()` and `np.outer()`:

```
print(np.dot(a,a))

print(np.outer(a,a))
```

A useful mathematical operator is the cross product (Cross product) where one calculates a vector which is perpendicular to vectors $\mathbf{x}$ and $\mathbf{y}$.

```
x = np.array([1,0,0])
y = np.array([0,1,0])
z = np.cross(x,y)
```

Pretty much all operators work with numpy arrays, even comparison operators, which can sometimes be very handy:

```
print(a)
print(a>3)
```

Exercise 4.3

Generate a sequence of the first 20 powers of 2 in a numpy array (starting at 2^0).

Your output should be an array $[2^0, 2^1, 2^2, 2^3, ...]$.

(Hint: Start with a numpy array created using an appropriate range function that makes an array [0,1,2,3,...])

```
# your code that makes the desired array
```

Functions for creating numpy arrays In numpy, there are also several handy functions for automatically creating arrays, see https://numpy.org/devdocs/reference/routines.array-creation.html. We provide some examples:

```
a = np.zeros(5)
print('zeros', a)
a = np.ones(5)
print('ones', a)
a = np.eye(5)
print('diagonal')
print(a)
a = np.tri(5)
print('ones below and on diagonal')
print(a)
```

np.linspace To automatically generate an array with linearly increasing values you can use `np.linspace()`:

https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html

np.linspace takes three arguments: the starting number, the ending number, and the number of points.

This is a bit like the `range` function we saw before, but allows you to pick the total number of points, automatically calculating the (non-integer) step size you need:

```
a = np.linspace(0,20,40)
print(a)
print()
print("Length is: ", len(a))
print("Step size is: ", a[1] -a[0])
```

Note that if we wanted to have a step size of exactly 0.5, we need a total of 41 points:

```
a = np.linspace(0,20,41)
print(a)
print()
print("Length is: ", len(a))
print("Step size is: ", a[1] -a[0])
```

Exercise 4.4

Generate an array that runs from -2 to 1 with 20 points using `linspace`.

```
a = your code
print(a)
```

np.arange() If we want to have more control on the exact spacing, we can use the `np.arange()` function. It is like `range()`, asking you for the start, stop, and step size:

```
a = np.arange(0,20,0.5)
print(a)
print()
print("Length is: ", len(a))
print("Step size is: ", a[1] -a[0])
```

Here, we already see a small quirk of `arange`: the stopcondition. It does not include this specified number (rather <than <=). If we want to get a range that stops at `20.0`, we need to make the stop point any number a bit bigger than 20 (but smaller than our step size):

```
a = np.arange(0,20.00000001,0.5)
print(a)
print()
print("Length is: ", len(a))
print("Step size is: ", a[1] -a[0])
```

For this reason, I do not find myself using `np.arange()` very often, and mostly use `np.linspace()`. There are also several other useful functions, such as np.geomspace(), which produces geometrically spaced points (such that they are evenly spaced on a log scale).

Exercise 4.5

Generate a numpy array that has a first element with value 60 and last element 50 and takes steps of -0.5 between the values.

```
a = your code
print(a)
```

Random numbers Numpy can also generate arrays of random numbers:

```
a = np.random.random(40)
print(a)
```

This will generate uniform random numbers on the range of 0 to 1, but there are also several other random number generator functions that can make normally distributed random numbers, or random integers, and more.

Exercise 4.6

Generate a numpy array, `rounded_grades` that contains 300 random grades that have a distribution of a bell-shaped curve that might represent the final grades of the students in this course, with an average grade of 7.5 and a standard deviation of 1. Make sure your grades are rounded to a half point.

(You may find it useful have to look at the help of the `np.random.normal()` function.

(Because of the properties of a normal distribution a small pecentage of the grades may be above a 10, you may leave this for now.)

```
#Your code here that results in a numpy array rounded_grades
...some code...
print(rounded_grades)
```

Exercise 4.7

There are various ways in which you can analyse your grade distribution.

You can plot a histogram of the grade distribution. Make sure that the width of our histogram bars is 0.5, corresponding to our rounded grades.

```
import matplotlib.pyplot as plt

... some code ...
```

Multidimensional arrays (matrices) We have looked at 1D arrays especially. However, numpy also supports two-dimensional (or N-dimensional) numpy arrays, that can represent matrices. To make a 2D numpy array, you can use the `zeros()` function, for example, but with a two-entry list of numbers specifying the size N and M of the matrix:

```
m = np.zeros([10,10])
print(m)
```

For two dimensional matrices, the usual function `len()` is not enough to tell us about the shape of our matrix. Instead, we can use a property of the numpy matrix itself called its `shape`:

```
print(len(m))
print(m.shape)
```

Indexing two dimensional arrays works by using commas inside square brackets to specify the index of the first and second dimensions:

```
a = np.array(range(1,6))
m = np.outer(a,a)
print("Initial matrix:")
print(m)

# First index in the row number (counting from zero), second index in the column number
m[2,3] = -1
print("\nAfter changing entry [2,3] to -1:")
print(m)
```

You can also use slicing to to assign values to an array from a vector, which can be a handy way to enter a matrix by hand:

```
m = np.zeros([3,3])
m[0,:] = [1,2,3]
m[1,:] = [4,5,6]
m[2,:] = [7,8,9]
print(m)
```

Similarly, slicing also can be used to extract rows, columns, or blocks of the matrix:

```
# A row
print(m[1,:])

# A column
print(m[:,1])

# Extract a block as a sub -matrix
print(m[1:,1:])
```

There are several functions for making matrices which you may find useful someday:
https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html
including this one which is used often:

```
# The identity matrix
print(np.eye(10))

# A band diagonal matrix
print(np.eye(10,k= -1))
```

Exercise 4.8
Use Python to calculate the following matrix multiplication:

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 & 0 \\ 3 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{4}$$

To perform this multiplication, you will need to use the `matmul` or `dot` routine of numpy:
https://docs.scipy.org/doc/numpy/reference/generated/numpy.matmul.html
https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html
(For the nerds: do these two matrices commute?)
(For the real nerds: have your program check if they commute and inform the user!)

```
m1 = np.zeros([3,3])
some code to fill the matrix
m2 = np.zeros([3,3])
some code to fill the matrix

# Check the matrices
print(m1)
print()
print(m2)
print()

product = some code
print(product)

do nerd stuff if you want...
```

We can do seemingly smart things using these 'special' vectors and matrices. For instance, what if we want to take the sum of all values in an array? We can use the dotproduct:

$$\begin{bmatrix}1 & 1 & 1\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\end{bmatrix} = 6 \tag{5}$$

```
ones = np.ones(3)
array = np.linspace(1,3,3)
print(np.dot(one_diag,array))
```

Moreover, if we want to create a cummulative sum of values in an array, we can use the tri matrix:

$$\begin{bmatrix}1 & 0 & 0\\1 & 1 & 0\\1 & 1 & 1\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}1\cdot 1+0\cdot 2+0\cdot 3\\1\cdot 1+1\cdot 2+0\cdot 3\\1\cdot 1+1\cdot 2+1\cdot 3\end{bmatrix} = \begin{bmatrix}1\\3\\6\end{bmatrix} \tag{6}$$

```
tri = np.tri(3)
print(np.dot(tri,array))
```

Another way to do so is to use a for-loop:

```
a = np.linspace(1,3,3)
sum_a = 0
cumsum = np.array([])

for x in a:
    sum_a += x
    cumsum = np.append(cumsum,sum_a)

print("The sum is", sum_a)
print("The cummulative sum is", cumsum)
```

Exercise 4.9

Below we have a set of repeated measurements. We want to understand how the average value of our repeated measurements evaluates over time, that is, how does the average value change as function of N repeated measurements? To investigate this: calculate and plot the average value as function of the number of measurements.

```
measurements = np.random.normal(5,1,20)
print(measurements)

#your code here
```

But wait... is there not a more direct way to do all of the above... Sure! We can make use of Numpy functions.

Numpy functions For many common (mathematical) operations, fuctions exists. For instance, we can calculate the average value of our measurements of above using a for loop, but also using the function `np.average`:

```
sum_meas = 0

for measurement in measurements:
    sum_meas += measurement

print(sum_meas/len(measurements))
print(np.average(measurements))
```

This is very handy: it saves us loads of thinking and typing! From the function name, it is also easy to understand what you are doing, making the code clearer and easier to read. However, the purpose of numpy functions is not only to save lots of typing: they also can often perform calculations MUCH faster than if you do program the calculation yourself with a `for` loop, as we will see in the next section.

Python also has many other useful functions for performing calculations using arrays:

```
# Calculate the standard deviation
print(np.std(measurements,ddof=1))

# The square root
print(np.sqrt(a))

# Numpy also has max and min, here is an example of min
a = np.linspace( -10,10,100)
print(np.min(a**2))
```

Good question for you to think about: why is the minimum value not zero? And what would I have to change above to get the code to return zero?

In addition to finding the minimum value in a vector, the function `argmin` can tell you **where** (what index number) the minimum is:

```
# Find the index number of the minimum of the array
i = np.argmin(a**2)
print(i)
print((a**2)[i])
```

Note also here that we used round brackets `()` around the `a**2` in the `print` statement to be able to then index the resulting array `a**2` array using square brackets `[]`.

You can find the full list of mathematical numpy functions on the documentation website:
https://docs.scipy.org/doc/numpy/reference/routines.math.html
and the full list of all functions in the reference guide:
https://docs.scipy.org/doc/numpy/reference/index.html

Exercise 4.10

a) Create an array with values ranging from 14 to 42, skipping the odd numbers.

b) Make an array `x` that runs from 0 to 4 with 20 points, and calculate an array `y` whose entries are equal to the square root of the entries in `x`.

```
your code to make the requested arrays x and y
print(y)
```

Vectorisation" and fast code with numpy functions In the first example above, we showed two ways of calculating an average: one using a `for` loop, and one using the numpy function.

Functionally, they are equivalent: they do exactly the same thing.

A curious feature of Python is that if you use functions instead of coding loops yourself, often things are **MUCH MUCH** faster.

To show this quantitatively, we will use the `time` library to calculate the time it takes to find the average of a pretty big array using both techniques:

```
# The time() function from the time library will return a floating point number representing
# the number of seconds since January 1, 1970, 00:00:00 (UTC), with millisecond or even micros
# precision
#
# We will use this to make a note of the starting time and the ending time,
# and then print out the time difference
from time import time

# A pretty big array, 50 million random numbers
a = np.random.random(int(50e6))

# Set timer
t1 = time()

# Caluclate the average
avg = 0
for x in a:
    avg += x
avg /= len(a)
t2 = time()
t_forloop = t2 -t1
print("The 'for' loop took %.3f seconds" % (t2 -t1))

t1 = time()
avg = np.average(a)
t2 = time()
t_np = t2 -t1
print("Numpy took %.3f seconds" % (t_np))

# Now let's compare them
print("\nNumpy was %.1f times faster!" % (t_forloop/t_np))
```

Why is numpy so much faster? The reason is that Python is an interpreted language. In each of the steps of the `for` loop, the Python kernel reads in the next step it has to do, translates that into an instruction for your computer processor, asks the computer to perform the step, gets the result back, reads in the next step, translates that into a processor instruction, sends that as an instruction to the computer processor, etc, etc.

If we did the same test in a compiled programing language like C, there would be no difference if we used a library function or if we wrote our own `for` loop.

When you use smart functions in Python libraries, like (many of) those in numpy, numpy will actually use an external library compiled in a language like C or Fortran that is able to send all of the calculation in one step to your computer processor, and in one step, get all the data back. This

makes Python nearly as fast as a compiled language like C or Fortran, as long as you are smart in how you use it and avoid having "manual" `for` loops for large or long calculations.

(For small calculations, Python coded `for` loops are perfectly fine and very handy!)

In the language of interpreted programmers, finding smart ways of getting what you need done using "compiled library functions" is often referred to as vectorisation.

Note that even normal mathematical operators are actually "vectorized functions" when they operate:

```
# This is actually a vectorized 'for' loop, it involves multiplying 50 million numbers by 5
b = 5*a
```

Here is a nice example of a vectorized way of counting the number of times the number '5' occurs in a random sample of 100 integers between 0 and 20:

```
nums = np.random.randint(0,21,100)
print("There are %d fives" % np.sum(nums == 5))
```

To see how this works, we can look at the intermediate steps:

```
nums = np.random.randint(0,21,100)
print(nums)
print(nums == 5)
print("There are %d fives" % np.sum(nums == 5))
```

Note that in this case, `np.sum()` will convert the `bool` value `True` into `1` and `False` into `0` for calculating the sum, according the the standard convertion of `bool` types to `int` types. You can see this in action if you want using the function `astype()` that is built into numpy arrays:

```
print(nums == 5)
print((nums == 5).astype('int'))
```

Another neat feature that `numpy` has is that is can 'vectorize' normal Python functions so that they can take `numpy` functions and make them a bit faster (10-20%). This is done using the `np.frompyfunc` function. An example is given below.

```
def fac(N):
    if (N == 1) or (N == 0):
        return 1
    return N * fac(N -1)

#As you can see applying the following array does not work
N = np.arange(9)

try:
    print(fac(N))
    print("This works")
except:
    print("This does not work \n")

#Now if we make this a numpy function is will work
#(the numbers in the frompyfunc represent the input and output values)

numpy_fac = np.frompyfunc(fac,1,1)
```

```
try:
    print(numpy_fac(N))
    print("This works")
except:
    print("This does not work")
```

Monte Carlo Simulation A commonly used method for numerical calculations is a Monte Carlo simulation (named after the (in)famous casino district). Simply stated, the method consists of generating a great number of random points and checking afterwards which points satisfy the boundary conditions of the calculation.

For example: to calculate the area of a circle with radius one, one can generate a great number of random points within a square of size 1 by 1, and check for each points whether they fall within the circle $r < 1$. The area of the circle can then be obtained by multiplying the area of the square with the number of points that are within the circle divided by the total number of points.

This may sound a bit cumbersome, and of course in this example calculating $\pi \cdot r^2$ is a lot faster. There are however many applications where the Monte Carlo method proves to be very useful; for example in calculating integrals that cannot (easily) be solved analytically.

In the next exercises, you are going to calculate the integral of the function $f(x) = e^{x^2}$ in the interval [0,1] using the Monte Carlo method.

Exercise 4.11

First, make a plot to get an idea of what the function looks like.

What is the maximum value of $f(x)$ in the interval [0,1]?

Make a (rough) estimate of the value of the integral by looking a the graph

```
import matplotlib.pyplot as plt

#First we define our function
def f(x):
    return np.exp(x**2)
```

Now we are going to generate random samples in a square that has an area that is greater than the integral that we want to calculate.

For this, we will use the np.random.uniform() function, which generates random float in the interval [a,b) - notice, this is an half-open interval so b is not included.

Moreover, it is wise to define the area by four points: x_a, x_b, y_a and y_b.

Exercise 4.12

Complete the code below to generate a random sample of y-values in an appropriate range.

```
N = int(1e6)   #number of points that we are going to use in our calculation
x_a =
x_b =
y_a =
y_b =
#Generate samples:
random_x = np.random.uniform( , ,size=N)
random_y = np.random.uniform( , ,size=N)
```

Next, we want to determine the number of points in our sample that fall within the area under the line f(x).

Exercise 4.13

Determine the number of points that satisfy this condition using a for loop and calculate the value of the integral.

```
from time import time

t1 = time()

### your code ###



solution_integral =

print('The solution of the integral is %.6f' %(solution_integral))
print('Time for calculation: %.3f s' %(time() -t1))
```

```
1.464291
The solution of the integral is 1.464291
Time for calculation: 5.187 s
```

We have a solution of the integral (much easier than on paper, right?)! But how do we know how accurate our solution is? First, compare your result to your estimate from the previous exercise. Does it make sense?

To make a substantiated claim about the accuracy of our solution we have to determine the uncertainty in our result. As we are basically performing a count, Poisson statistics tells us that the uncertainty in the counted number of points that satisfy the condition is the square root of the counted number: $u(N_{counted}) = \sqrt{N_{counted}}$. We can then determine the uncertainty in the area using the calculus approach.

Let's check our solution by comparing it to the answer given by another numerical method from the scipy library.

```
from scipy.integrate import quad

Area = np.abs(x_b -x_a)*np.abs(y_b -y_a)
err_solution = Area*np.sqrt(s)/N          #estimated error of our solution, which is the area

scipy_solution = quad(f,0,1)       #calculating integral using quad function from scipy, retur

abs_difference = np.abs(solution_integral -scipy_solution[0]) #determine absolute difference b


assert abs_difference <= 2*np.sqrt(err_solution**2 + scipy_solution[1]**2), 'The results are n

print('Our solution: %.3f +/ - %.3f' %(solution_integral, err_solution))
print('Scipy solution: %.14f +/ - %.14f' %(scipy_solution[0], scipy_solution[1]))
print('Difference between the solutions: %.6f' %(abs_difference))
```

```
Our solution: 1.464 +/ - 0.002
Scipy solution: 1.46265174590718 +/ - 0.00000000000002
Difference between the solutions: 0.001639
```

Looks pretty good!

Our method gives a reasonable answer, but as you can see the uncertainty is relatively high...If we want to improve our result we should use more points to lower the uncertainty (which scales with $\sqrt{N}$). However, the for loop we used is already quite slow. More points would mean an annoyingly long calculation.

We should therefore try to avoid the use of a for loop and instead use a comparison operator to speed up the calculation.

Exercise 4.14

Again, complete the code below and calculate the integral, this time without using a for loop

```
t1 = time()

s = np.sum(  <= )

solution_integral =

print('The solution of the integral is %.6f' %(solution_integral))
print('Time for calculation: %.3f s' %(time() -t1))
```

This is much faster!

Now lastly we want to plot our random generated points that are within the area underneath f(x) and those that are not in the same graph with a different colour. For this, we need the actual values of the points that satisfy the condition.

As you have seen before, a comparison operation on a numpy array returns a type boolean array:

```
array = np.array([4,2,6,8,5,4])

boolean_array = array > 4

print(boolean_array)
```

We can subsequently use this boolean array to get the values of the array that satisfy the condition:

```
new_array = array[boolean_array]

print(new_array)
```

Or, more simply:

```
new_array = array[array > 4]

print(new_array)
```

Exercise 4.15

Make a figure in which you plot:

- the function f(x)
- the random points that are within the area under the line f(x)
- the random points that are outside this area (in a different colour)

We will generate new random samples with fewer points, otherwise we cannot distinguish them. Don't forget to label your axes!

```
random_x = np.random.uniform( , ,1000)
random_y = np.random.uniform( , ,1000)

#your code
```

Exercise 4.16

One way to 'improve' the quality of your data, making it less noisy, is the use of a moving average filter. The moving average filter calculates the average value of N elements, N often being 3 or 5. This smooth out any abrupt changes and allows a better focus on long term trends. Of course you will lose two elements of the entire dataset (think yourselves why).

The filtered data is stored in an array: $F(i) = \frac{p(i)+p(i+1)+p(i+2)}{3}$, looped over all elements $p(i)$. More precise: $F(i) = \frac{1}{k}\sum_{i}^{i+k-1} p(i)$

```
Z = np.arange(1,20)
#def moving_average_1(p, k):
#     F = np.empty(len(p) -(k -1))

print(Z)
print(moving_average_1(Z,3))
```

Solutions to exercises Exercise 4.1

```
a = np.array([1,2,3,4,5,6,7,8,9,10,11,32,55,78,22,99,55,33.2,55.77,99,101.3])

#some code to set the first three and last two entries to zero

# A bit tricky, but these work
a[0] = 0
a[1] = 0
a[2] = 0

# We could count forwards (a[19] = 0 and a[20] = 0), but is much
# easier to count backwards from the  end
a[ -2] = 0
a[ -1] = 0
print(a)
```

Exercise 4.2:

```
a = np.array(range(20))+1
print(a)
a[0:10]  =  0
print(a)
```

Exercise 4.2∗:

```
a = np.array([1,2,3,4,5,6,7,8,9,10,11,32,55,78,22,99,55,33.2,55.77,99,101.3])
a[ -2:]=0
a[:2]=0
print(a)
```

Exercise 4.3:

```
n = np.array(range(21))
out = 2**n
print(out)
```

Exercise 4.4:

```
a = np.linspace( -2,1,20)
print(a)
```

Exercise 4.5:

```
a = np.arange(60,49.9, -0.5)
print(a)
```

Exercise 4.6:

```
raw = np.random.normal(7.5,1,300)
rounded_grades = np.round(raw*2)/2
print(rounded_grades)
```

Exercise 4.7:

```
import matplotlib.pyplot as plt
bins = np.arange(np.min(rounded_grades),np.max(rounded_grades)+0.5,0.5)

plt.figure()
plt.hist(rounded_grades, bins=bins)
plt.xlabel('Grade')
plt.ylabel('#')
plt.show()
```

Exercise 4.8:

```
m1 = np.zeros([3,3])
m1[0,:] = np.array([1,1,0])
m1[1,:] = np.array([0,2,1])
m1[2,:] = np.array([1,0,1])
m2 = np.zeros([3,3])
m2[:,0] = np.array([1,3,1])
m2[:,1] = np.array([3,1,1])
m2[:,2] = np.array([0,1,1])

# Check the matrices
print(m1)
print()
print(m2)
print()

product = np.matmul(m1,m2)
print(product)
```

Exercise 4.9:

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

measurements = np.random.normal(5,1,20)
print(measurements)

average_n = np.cumsum(measurements)/np.arange(1,len(measurements)+1)

plt.figure()
plt.plot(np.arange(1,len(measurements)+1), average_n)
plt.xlabel('average')
plt.ylabel('n')
plt.show()
```

Exercise 4.10:

```
import numpy as np
```

```
array_skip = np.arange(14,44,2)
print(array_skip)


x = np.linspace(0,4,20)
y = np.sqrt(x)
print(y)
```

Excercise 4.11

```
import matplotlib.pyplot as plt

#First we define our function
def f(x):
    return np.exp(x**2)

x = np.linspace(0,1,1000)         #generate an array with x values

#make the figure
plt.figure(figsize=(6,4))
plt.plot(x,f(x))
plt.xlim(0,1)
plt.ylim(0)
plt.ylabel('f(x)')
plt.xlabel('x')
plt.show()

print('Maximum value of f(x) in the interval [0,1] is %.3f' %(np.max(f(x))))
```

The area under the graph is roughly equal to half of the total area of the square 1x3, i.e. 1.5.

Exercise 4.12

```
N = int(1e6)   #number of points that we are going to use in our calculation
x_a = 0
x_b = 1
y_a = 0
y_b = 3
#Generating our samples:
random_x = np.random.uniform(x_a,x_b,size=N)
random_y = np.random.uniform(y_a,y_b,size=N)
```

Exercise 4.13

```
from time import time

t1 = time()

s = 0

for i in range(len(random_x)):
    if random_y[i] <= np.exp(random_x[i]**2):
        s+= 1

solution_integral = (x_b -x_a)*(y_b -y_a)*s/N
```

```
print('The solution of the integral is %.6f' %(solution_integral))
print('Time for calculation: %.3f s' %(time() -t1))
```

Exercise 4.14

```
t1 = time()

s = np.sum(random_y <= np.exp(random_x**2))

solution_integral = 3*s/N

print('The solution of the integral is %.6f' %(solution_integral))
print('Time for calculation: %.3f s' %(time() -t1))
```

Exercise 4.15

```
random_x = np.random.uniform(0,1,1000)
random_y = np.random.uniform(0,3,1000)

points_within_area = np.array([random_x[random_y <= np.exp(random_x**2)],random_y[random_y <= 
points_outside_area = np.array([random_x[random_y > np.exp(random_x**2)],random_y[random_y > n

plt.figure(figsize=(6,4))
plt.plot(points_outside_area[0],points_outside_area[1], '.',c='red')
plt.plot(points_within_area[0],points_within_area[1], '.', c='green')
plt.plot(x,f(x),linewidth=3,label='f(x)')
plt.xlim(0,1)
plt.ylim(0,3)
plt.ylabel('f(x)')
plt.xlabel('x')
plt.legend()
plt.show()
```

Exercise 4.16

```
def moving_avg(p, k):

    F = np.empty(len(p) -(k -1))

    for i in range(len(p) -(k -1)):
        F[i] = (1/k)*np.sum(p[i:i+k])

    return F

Z = np.arange(4,20)
print(Z)
print(moving_average(Z, 3))
```

Data in Python: Loading, Plotting, and Fitting

Pre/Post-test This test is for testing your current skills in Python. You can use it in two ways:

- pre-test: to test your skills beforehand. If you are already proficient in Python, and can do this test within approximately 15 minutes, you can scan through the notebook rather than carfefully reading each sentence.
- post-test: test your skills after Notebook 5. Check whether you learned enough.

Doing an analysis Eric so far survived the first weeks of his employment. As he showed some true skills, he is asked to help in building an RC-filter. First, some initial data needs to be analysed. He receives a datafile called 'pretestdata.csv'. Once he retrieved (loaded) this data, he plots it first. He then uses the theoretical formula that should describe the pattern in the data: $\frac{U_C}{U_{max}} = \frac{1}{\sqrt{1+(wRC)^2}}$, where w is the angular frequency given by $w = 2\pi f$. RC can be considered a constant which can be found using least-square fitting.

- Retrieve the data and plot these using a log-scale on the x-axis. Think about the scientific rules for making plots.
- Find the best estimate of RC and plot the fitfunction on top of the dataset.
- Carry out a residual analysis.

```
# Your code
```

Learning objectives In this notebook, we will explore how to load and save datafiles in Python using Numpy, how to plot data and functions with a plotting library called Matplotlib, and how to fit a model to data and analyse the results.

After completing this notebook, you are able to:

- load data from ASCII text files
- save data to ASCII text files
- generate plots of data and functions
- fit a model to data using "fitting by hand"
- use `curve_fit` to perform a least-squares fit of a model to a dataset
- analyse the residuals to determine the quality of the fit
- calculate the statistical error in the parameters of a least-squares fit

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

Data management Here, we will explore some functions provided by numpy for loading data from files into python, and for saving data from python to files. We consider it good practices that you save a raw data (CSV) file somewhere on your computer and then copy this file for further processing. If any mistake has been made, you can always rely on your stored raw data file.

During the experiments that you encounter in this course you can write your measurements in an excel file, and store this file as a CSV file. When processing the data, you will load the data in your Jupyter Notebook file.

Loading data with Numpy Until now, we have seen how to generate data inside python, for example by assigning values to arrays, or, for example, by using functions like `np.zeros()`, `np.linspace()`, and `np.random.random()`.

However, what if we want to use Python to analyze data from an experiment? How do we get the data into python?

Sometimes, we might have just a small number of data points that we have measured by hand and wrote down on a piece of paper: say a list of measured fall times for different heights. In this case, you can just "load" the data by defining Python arrays that contain your measured values:

```
# Drop heights in meters
d = np.array([0, 0.5, 1.0, 1.5 , 2.0,  2.5])
# Measured times
t = np.array([0, 0.2, 0.3, 0.44, 0.6, 0.89])
```

In a computer controlled experiment, though, you may have, for example, a data file of voltage measured as a function of time from an acquisition card or oscilloscope which may have hundreds, thousands, or even millions of data points. Moreover, if you manually take measurements it is often more convenient to do so using excel. You can then save your data and load them using Python. If done so, your raw data is always safely stored.

If we want to look at the datafile, to get an idea of what is inside, whether there are comments included and so on, we can open the file and read it.

```
with open("v_vs_time.dat") as file:
    for i in range(5):
        print(file.readline())
```

We can also read (and print) the entire file using `print(file.read())`.

Typically (if you're lucky), the measurement software will save its data in an (ASCII) text file. (Or at least there will be software for converting the data to an ASCII text file...) For example, let's think about an experiment where we measure a voltage as a function of time. In this case, the data will often be recorded in two columns of numbers in the text file, separated by a space. The first number would represent the time of the measurement, say in seconds, and the second the measured voltage, in volts:

File: `v_vs_time.dat`:

```
0.00000000000000000e+00 -4.881968419624156397e -01
1.001001001001000992e -02 6.574038838172248100e -01
2.002002002002001985e -02 -4.868767180890223312e -01
3.003003003003002977e -02 -8.581352351530058264e -01
4.004004004004003969e -02 1.605962930324425164e+00
5.005005005005004615e -02 -5.458119271515118331e -01
6.006006006006005954e -02 -1.910121935716198927e+00
7.007007007007007293e -02 -8.213606665035044774e -01
8.008008008008007939e -02 -1.724982388182002335e+00
...
```

This is an example of a file format called Delimiter-separated values (DSV). In the example above, the "delimiter" (ie the thing that separates the values) is a space.

Another common delimiter is a comma `,` or `;`, often named a `.csv` file for "comma separated value" file. This is a common "export" format from spreadsheets like excel. A third common value is a "tab separated value" file (often saved with a `.tsv` file extension, also available as an export option from some spreadsheets), which separates the values in the file be tab characters (which in Python strings show up as special character `\t`, a bit like the newline `\n`).

For reading in CSV files, Python has a very simple routine `np.loadtxt()`:

https://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html

It can handle any type of **delimiter**: the default is any whitespace (tab or space), but this can also be configured using the `delimiter=` keyword argument. It also does not care about the file extension: for example, it is fine if a CSV file does not have a `.csv` extension (it doesn't even have to have an extension!). A typical convention is to name the files containing ASCII text data with an extension `.dat`.

Let's give it a try using the file `v_vs_time.dat`:

```
data = np.loadtxt("v_vs_time.dat")
```

If the first line would have been the names of the variables, we could have skipped the first row (see again the documentation). Here we have assigned the return value of `np.loadtxt` to a variable `data`, which is a numpy array. But what exactly is our variable `data`? We can find out more by looking at the shape of the returned `data` array:

```
data.shape
```

When `np.loadtxt()` loads a file, it will return a 2D numpy array with of shape `(n,m)`, where `n` is the number lines in the file and `m` is the number of columns (here, we have 1000 points and 2 columns).

As I mentioned above, the first column represents the time that the measurement was taken, and the second column represents the measured voltage in volts. We will typically want to extract these into two vectors `t` and `v`, which we can do using slicing:

```
t = data[:,0]
v = data[:,1]
```

We can look at the first 10 points and see that we have successfully loaded the data from the file!

```
print('t = ', t[0:10])
print('v = ', v[0:10])
```

We can check that the data is correct by opening the data file itself from the Jupyter file browser: Jupyter can also directly load and edit text files. (Another way of loading hand-recorded data into Python is just to use Jupyter to create an ASCII `.dat` file yourself.)

Exercise 5.1

Load the data from the file `exercise_data.dat`. The first column represents a measurement time in seconds, and the second represents a measurement voltage in volts. How many points are in the dataset?

(Important: make sure that you use different variable names than t and v for the data you load, since we will continue to use the data we loaded in the example code in the sections below!)

```
#your code here to load the data
```

Saving data with Numpy We can also save data using a numpy routine `np.savetxt()` (https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html). To do this, we have to "pack" the data back into a numpy array of the correct shape.

Let's take a look at an example where we calculate the square of the measured voltage and save this back into a new file:

```
v_squared = v**2
```

To "pack" them together into a matrix like the one returned by `np.loadtxt()`, we can first create a 2D matrix of the correct size and then use slicing with an assignment operator `=` to give the columns the correct values:

```
# Create empty array
to_save = np.zeros([len(v), 2])

# Fill columns with data
to_save[:,0] = t
to_save[:,1] = v_squared
```

Now we are ready to use `np.savetxt()` in the current folder:

```
np.savetxt("vsquare_vs_time.dat", to_save)
```

If you go back to your workspace in a new tab, you will can see that this file has been created, and if you click on it, you can see what is inside it.

Exercise 5.2

Make a testarray x running from 1 to 100 with an interval of 0.1. Write a function $y = 3.05 * x + 0.95$. Use this function to make a new array for y. Save your data in the file 'testfile.csv' using ; as a delimiter (NOTE: delimiters are sometimes ";" and sometimes "," Always inspect the datafile first.). If all went well, when running the subsequent cell your data are loaded and shown in a plot.

```
x =

def function

y =

imported_data = np.loadtxt('testfile.csv', delimiter=';')

plt.figure()
plt.plot(imported_data[:,0],imported_data[:,1],'k.',markersize='1')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.xlim(0,100)
plt.show()
```

Plotting data and functions with Matplotlib We loaded a dataset but haven't given it a proper look yet. You already learned how to plot data, but here we give it a more thorough look. We will use our previous data loaded, and a fake dataset.

```
# A "fake" dataset to illustrate plotting
v_fake = np.random.normal(10,0.5,1000)

plt.figure()
plt.plot(t,v,'k.',markersize=1)
plt.plot(t,v_fake,'r+')
plt.xlabel("Time (s)")
plt.ylabel("Voltage (V)")
plt.show()
```

Matplotlib will automatically change the color of the second dataset (you can also control them manually, see the documentation). But it can be hard to tell which one is which, and for this reason, it can be useful to add a legend to your plot. Furthermore, if you want a really big figure, you can also adjust the figure size:

```
plt.figure(figsize=(12,8))
plt.plot(t,v, 'k.',markersize=1, label="Voltage 1")
plt.plot(t,v_fake, 'r+',markersize=1, label="Voltage 2")
plt.xlabel("Time (s)")
plt.ylabel("Voltage (V)")
plt.legend()
plt.show()
```

Exercise 5.3

Make a plot of the data you loaded from exercise 5.1.

```
#code to plot the data you loaded in exercise 5.1
```

Exercise 5.4

As you may have noticed the data from `exercise_data.dat` does not seem like straight line, but instead looks like a power law function $V = at^p$ with power p.

One way to test for such power law relations is to make a plot with an x-axis that is not t but instead t^p: if you get the correct p, then the data plotted will form a straight line.

Write code to plot V against t^p and then put in different guesses at the power p to see if you can see what the power is.

```
#code to plot data from exercise 5.1 with t^p on the x -axis
```

Also handy to know: you can use LaTeX commands to add mathematical symbols and equations to your plot.

Important: when you do this, it is a good idea to add an `r` before the quote of your string (see below) to prevent the common backslash `\` commands of latex as things like tab characters or newline characters.

```
plt.figure(figsize=(6,3))
plt.plot(t,v, 'k.',markersize=1, label="Voltage 1")
plt.plot(t,v_fake, 'r+',markersize=1, label="Voltage 2")
plt.xlabel(r"Time $\tau$ (s)") # random greek symbols
plt.ylabel(r"Voltage $V_\alpha$ (V)") # subscripts
plt.legend()
plt.show()
```

As explained earlier on, we can plot functions as well. It is useful to plot both your data and a function describing that data. For example, with our voltage data above, if we want to plot a straight line function over over the data:

```
line = 2*t
plt.figure()
plt.plot(t,v, '.')
plt.plot(t,line, ' - -', lw=4)
plt.xlabel("Time (s)")
plt.ylabel("Voltage (V)")
plt.show()
```

Here, we also used the `linewidth` parameter (which can be shorted to `lw` to save typing) to make the line a bit fatter so it is easier to see, and used the '–' plot format specifier to make it a dashed line.

Exercise 5.5

Another way to get the exponent is to plot $\log(V)$ vs $\log(t)$ (a log-log plot). If the data really follows a power law relation, then this should give a straight line with a slope determined by the exponent. Try this, and then add a straight line $\log(V) = p * \log(t)$ to the plot. Play around with the value of p until the line has the same slope as the data. For this plot, use solid filled circle for the data and a solid line for the theory.

(Do you get a warning message? Why?)

```
#your code
```

Fitting In experimental physics, one of our primary goals is to try to figure out if our observed experimental measurements match the theoretical predictions that we can make from mathematical models. If it does, then we are happy and we have confirmed that our mathematical model is at least one way to explain our data! (We typically say that the experimental observations are consistent with the predictions of theory.) If that all works, we try to take this a step further, and extract values for our theoretical parameters from our data.

In this section of the notebook, we will discuss how to fit a mathematical model to your experimental data and extract both the best-fit parameters of your model for the given dataset, and also the estimated (statistical) uncertainties in these fitted parameters.

college 2 deel 3

Fitting by hand A first important step in any fitting is to make some plots of your experimental model and find out if there are parameter regimes where it can at least come close to the data. By "fitting by hand", you can already get an idea whether your model is a good description of your data, and approximately what the values of the model parameters are.

In the last plot above, we saw an example of plotting a function on top of some data. If we look at the data, even without looking at the plotted function, we can see that our experimental data of voltage vs. time seems to be a straight line. This suggests that we can model the data with a linear function:

$$V(t) = at + b \tag{7}$$

where a is some parameter with the units of V/s representing the slope of the line, and b (in V) is a parameter describing the value of the voltage at $t = 0$.

The fact that the the line $V = 2t$ (the orange dashed line) seems to pass through the middle of our blue data already suggests that for our data (the blue points), $a = 2$ V/s and $b = 0$ V seems like pretty good parameters for our model to describe the data.

We can also see that `a = 1.7` and `b = 1.7` are also OK (they go through a lot of the data points), but are probably not as good:

```
line1 = 1.7*t+1.7
line2 = 2*t

plt.figure()

plt.plot(t,v, '.',c='k', ms=2)
plt.plot(t,line1, ' - -', c='b', lw=4, label="y=1.7$x$+1.7")
plt.plot(t,line2, ' - -', c='r', lw=4, label="y=2.0+0.0")

plt.xlabel("Time (s)")
plt.ylabel("Voltage (V)")

plt.legend()
plt.show()
```

Now that we have a 'good' estimate of our parameters, how can we find the **best** parameters (`a` and `b` in this case) for our data? And how do I know how accurately my parameters fit the data?

For that, we can use a procedure called least squares fitting. In the remaining part of this notebook, you will learn more about least squares fitting and how it works - the video below is a very short introduction to the least squares method. Here, we will look at how to code it in Python, how to use it to get the best fit parameters of a model for you data, and how to find out what the statistical error is on the parameters of your model given the data you are using for the fit.

What is Least Squares?

Exercise 5.6

Using the data you loaded in exercise 5.1, try fitting the function $V = at^p$ "by hand" to the data by manually adjusting a and p and replotting the function until you think you have the best fit. What are the values of a and p that you find?

```
#code to fit data "by hand" to a power law
```

Least squares fitting To have Python automatically find the best parameters for fitting your data with a given function, you can use the `curve_fit()` routine from the `scipy` (scientific Python) package:

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html

Since we will use only the `curve_fit` routine, we will import it on its own:

```
from scipy.optimize import curve_fit
```

To use curve fit, we first need to create a Python function that returns the mathematical function that we want to fit. Our function is:

$$f(x) = ax + b \tag{8}$$

It has one variable, x, and two parameters, a and b. For using `curve_fit`, we will need to make a Python function that takes three arguments: the first is the variable `x` (for the above, it will be time), and the other two arguments are the fit parameters `a` and `b`, and which should return the predicted voltage:

```
def function_fit(x,a,b):
    return a*x+b
```

In the simplest case (and it will always work when you are fitting a stright line), you can just then directly send the function and your x and y data and it will do the fit for you:

```
values, covariance = curve_fit(function_fit, t, v)
```

Tada! `curve_fit` actually returns two things. The first is an array containing the optimal fit values:

```
print(values)
```

The first one corresponds to the first parameter of the function and the second to the second: in our case, `a` and `b`:

```
a_fit = values[0]
b_fit = values[1]
print(a_fit)
print(b_fit)
```

We can actually see that our fit by hand is a pretty good approximation of the values Python returns using the least squares method. We can now check what the fit looks like. To do so, it is useful to create a testarray for x first, and calculate our corresponding values for y. We will see why this is a good idea in a minute.

```
x_test = np.linspace(0.8*min(t),1.2*max(t), 1000)
y_fit = function_fit(x_test,*values)

plt.figure()

plt.plot(t, v, '.', c='k', ms=2)
plt.plot(x_test, y_fit, 'r - -', lw=2, label="Fit (y = %.2f $x$ + %.2f)" % (a_fit, b_fit))

plt.xlabel("Time (s)")
plt.ylabel("Voltage (V)")
plt.legend()

plt.xlim(0,10)
plt.ylim(min(v),max(v))
plt.show()
```

Cool! Also, it turns out that the best statistical fit of the slope to our data is not exactly 2 but slightly higher (about 2.0148)

In addition, sometimes in measurements, we also have error bars on the each of the individual black data points in the plot, and sometimes some data points could have more error than others. In these cases, the fitting routine can take these individual error bars into account when performing this fit: we will see more about this later in the course.

Exercise 5.7

Perform a least squares fit to of the data from exercise 5.1 to the function $V = at^p$. Make a plot of the function with the values of p and a from the fit on top of the data. How close are the fitted values to your attempt at fitting by hand?

```
#code that performs a least square fit to the exercise data
```

Test array and some ideas for plotting It was mentioned that it is a good idea to make a testfunction before plotting our fitline. Why is that so? If our fitfunction is not a straight line, and we have not that many measurements, the dots are connected. That is, our fit is not continous. Below we illustrate this with the red dotted lines, making sharp turns at each new datapoint. The blue dotted line shows our test variable with many more datapoints. Moreover, we can improve our plot by including values for the xlim, where our fitfunction nicely starts at the borders of the graph. To better compare our graphs, we make use of subplots. It requires some adjustments, but the manual is very helpful: https://matplotlib.org/stable/gallery/subplots_axes_and_figures/subplots_demo.html

```
x = np.arange(1,9,2)
y = 0.25*x**2

#fitting our data
def fitfunction3(x,a,b):
    return a*x**b

val, cov = curve_fit(fitfunction3,x,y)

#making a test array for our fit
x_test = np.linspace( -0.2,1.2*max(x))
y_fit = fitfunction3(x_test,*val)


#plotting of the data and fit in a subplt
fig, (ax1, ax2) = plt.subplots(1, 2)

ax1.plot(x,fitfunction3(x,*val),'r - -',label='fit')
ax1.plot(x,y,'k.',label='measurements')
ax1.set(xlabel='$x$', ylabel='$y$')
ax1.legend()
ax1.set_xlim([0, 7.5])
ax1.set_ylim([0,15])


ax2.plot(x_test,y_fit,'b - -',label='fit')
ax2.plot(x,y,'k.',label='measurements')
ax2.set(xlabel='$x$', ylabel='$y$')
ax2.legend()
ax2.set_xlim([0, 7.5])
ax2.set_ylim([0,15])
```

```
fig.subplots_adjust(wspace=0.4)
plt.show()
```

Initial guesses The curve fit routine has the option to provide initial guesses on your parameters using a keyword argument `p0`:

```
initial_guesses = (2.5,1)
values2, covariance2 = curve_fit(function_fit, t, v, p0 = initial_guesses)
print(values2)
print(values)
```

In this case it (nearly) did not make a difference: and in fact, for fitting a linear function, least squares fitting will converge to the same set of values, no matter what initial guess you provide.

This is NOT true for functions $f(x)$ that are not linear in x. In really bad cases, the parameter values that `curve_fit` finds can even be highly sensitive to the initial conditions. For this reason, it is always a good idea, and sometimes absolutely necessary, to first do some "fitting by hand", and then ideally provide these initial conditions to the `curve_fit` routine.

And even if you do not provide hand-tuned initial guesses (which you may not if you are automatically fitting a huge number of datasets), it is important that you always make a plot of the fitted curves that you find on top of the data just to make sure nothing goes wrong.

Residuals But how good is our fit? That is difficult to say by only lookig at the plot above. But we can investigate our residuals: $R(x) = D(x) - F(x)$ where $R(x)$ are our residuals as function of x, $D(x)$ are our data (measurements) as function of x, and $F(x)$ our fitfunction at x. If our functionfit perfectly matches our data, our residues are 0. But as our data often includes some noise, we expect that our residuals are normally distributed (Normal distribution).

```
R = function_fit(t,*values) -v

plt.figure()

plt.plot(t, R, '.', c='k', ms=2)

plt.xlabel("Time (s)")
plt.ylabel(r"$\Delta$ Voltage (V)")

plt.show()

plt.figure()
plt.hist(R,bins=40,density=True)
plt.xlabel(r"$\Delta$ Voltage (V)")
plt.ylabel('pdf')
plt.show()
```

Well... it sure looks likes a normal distribution. But can we visualize it even better? We can try to fit using a normal distrubtion. The details about the normal distrubution will be discussed in Notebook 6. The scipy norm library is here of good use!

```
from scipy.stats import norm

x_test = np.linspace(np.min(R),np.max(R),1000)
distribution = norm.pdf(x_test,np.mean(R),np.std(R))
```

```
plt.figure()
plt.hist(R,bins=40,density=True)
plt.plot(x_test,distribution,'r')
plt.xlabel(r"$\Delta$ Voltage (V)")
plt.ylabel('pdf')
plt.show()
```

What did we do here? We created a test dataset and for each of these values we calculated the return value from the pdf, using the average and standard deviation calculated from our residuals. We then plotted this function on top of our histogram. We can see that the distribution of the residuals can be described reasonably well with a normal distribution!

Uncertainty in the parameters We found above that the best statistical fit to our data was not our initial estimate of `a = 2`, but actually more like `a = 2.0148`. But the line with `a = 2` also looked pretty good, right? How much better is the fit with `a = 2.0148`?

To find this out, we can use the other variable that `curve_fit` returned: the covariance matrix. With two parameters, the covariance matrix is a 2x2 array:

```
values, covariance = curve_fit(function_fit, t, v)
print(covariance)
```

The most important two for us are the diagonal elements: the first diagonal element tells us the square of the standard error σ_a of parameter `a` and the second diagonal element gives us the square of σ_b:

```
a_err = np.sqrt(covariance[0,0])
b_err = np.sqrt(covariance[1,1])
print(a_err)
print(b_err)
```

Or more quickly:

```
a_rr, b_err = np.sqrt(np.diag(covariance))
print(a_rr,b_err)
```

(The standard error σ_a is also sometimes also written as Δa, and is also sometimes called the parameter "uncertainty" instead of the parameter "error").

We can now quote the full values of the parameters from our fit, including uncertainties:

```
print("The value of a is %.2f +/ - %.2f V" % (a_fit, a_err))
print("The value of b is %.2f +/ - %.2f V" % (b_fit, b_err))
```

Reminder: as earlier shown, we use a placeholder format `%.2f` designating that we want to print as float with two decimals. There are a lot of other ways to format your string.

Note that typically when you quote an uncertainty, you should pick only one significant digit. And also, when you quote the value of a parameter, you should also quote it with the same number of decimal places as your error.

So in this case, I quote the value of `a` as 2.015, and not its "exact" value of 2.0148415283012246. Why? The reason is that doesn't make sense to quote more digits on the value of `a` than a number of digits corresponding to the size of the statistical error, and so physicists and scientists truncate the accuracy of the numbers they report to a level comparable to the error.

Exercise 5.8

Below we have made a fake dataset for you. The parameters are known to you. But still, it is your task to 'deconstruct' the dataset and find the parameters as if these were unknown to you.

```
x_meas = np.arange(0,10.1,0.3)
y_meas = 0.2345*x_meas**2 + 3.045*x_meas + np.random.normal(0,1.5,len(x_meas))

#your code here
```

Exercise 5.8*
Calculate the errors on the fit parameters from your fit in exercise 5.5. Was your estimate of p from exercise 5.4 within the statistical error margins from you fit? Were the values for a and p you found from "fitting by hand" within the statistical error margins?

```
#code to calculate the parameter uncertainties (errors) and check if your earlier
#estimates agree with the least -square -fit values
#to within the error margins of the fit
```

Voorbeeld van een data analyse
Exercise 5.9
In 'dataset.csv' we have made you a dataset where you have to find the formula describing it. Find the parameters and their associated uncertainties. Don't forget to analyse your residuals to see whether you found the proper formula.

Note! This excercise consumes a lot of your time...

Solutions to exercises Exercise 5.1:

```
# You can also use the option "unpack=True" to have loadtxt send back all columns separately
t2,v2 = np.loadtxt("exercise_data.dat", unpack=True)

print("Loaded", len(v2), "points")
```

Exercise 5.2:

```
x = np.arange(1,100.1,0.1)

def function(x):
    return 3.05*x+0.95

y = function(x)
data = np.zeros([len(x),2])
data[:,0] = x
data[:,1] = y
np.savetxt('testfile.csv',data,delimiter=';')
```

Exercise 5.3:

```
plt.plot(t2,v2)
# ALL PLOTS MUST HAVE LABELS WITH UNITS!!!!
plt.xlabel("$t$ (s)")
plt.ylabel("$v$ (V)")
plt.show()
```

Exercise 5.4:

```
# p = 2 looks ok
p = 2
plt.plot(t2**p,v2)
# A decent label(we can use the  variable value automatically, and use latex for superscripts)
plt.xlabel("$t^"+ str(p) + "$ (s$^" + str(p)  +"$)")
plt.ylabel("v (V)")
plt.show()
```

Exercise 5.5:

```
# p = 2 looks ok
plt.plot(np.log(t2),np.log(v2),'o', label="Data")

# Now a straight line
# This is a bit tricky, we need the same x -range as the data
# My best estimate of the slope is 2.3
x = np.log(t2)
p = 2.3
y = p*x
plt.plot(x,y, label="Slope = " + str(p))
plt.legend()

# A decent label(we can use the  variable value automatically, and use latex for superscripts)
plt.xlabel("log(t) (log(s))")
plt.ylabel("log(v) (log(V))")
plt.show()
```

Exercise 5.6:

```
a=2
p=2.34
plt.plot(t2,v2, 'o', label='Data')
plt.plot(t2,a*t2**p, label='Model')
plt.legend()
plt.xlabel("t (s)")
plt.ylabel("v (V)")
plt.show()
```

Exercise 5.7:

```
def f2(t,a,p):
    return a*t**p

values, covariance = curve_fit(f2, t2, v2)

a_fit = values[0]
p_fit = values[1]

print(values)

plt.plot(t2,v2, 'o', label='Data')
plt.plot(t2,f2(t2,a_fit,p_fit), label='Model')
plt.legend()
plt.xlabel("t (s)")
plt.ylabel("v (V)")
plt.show()
```

Exercise 5.8

```
x_meas = np.arange(0,10.1,0.3)
y_meas = 0.2345*x_meas**2 + 3.045*x_meas + np.random.normal(0,1.5,len(x_meas))

#your code here
plt.figure(figsize=(3,3))
plt.plot(x_meas,y_meas,'k.')
plt.show()

#first guess
def function_fit(x,a):
    return a*x**2

val, cov = curve_fit(function_fit,x_meas,y_meas)
x_test = np.linspace(min(x_meas),max(x_meas),1000)

plt.figure(figsize=(3,3))
plt.plot(x_meas,y_meas,'k.')
plt.plot(x_test,function_fit(x_test,*val))
plt.show()

plt.figure(figsize=(3,3))
plt.plot(x_meas,y_meas -function_fit(x_meas,*val),'k.')
plt.show()

#there is a pattern in the residuals, so our formula does not sufffice

#second guess
def function_fit(x,a,b,c):
    return a*x**2+b*x+c

val, cov = curve_fit(function_fit,x_meas,y_meas)
x_test = np.linspace(min(x_meas),max(x_meas),1000)

plt.figure(figsize=(3,3))
plt.plot(x_meas,y_meas,'k.')
plt.plot(x_test,function_fit(x_test,*val))
plt.show()

plt.figure(figsize=(3,3))
plt.plot(x_meas,y_meas -function_fit(x_meas,*val),'k.')
plt.show()

plt.figure(figsize=(3,3))
plt.hist(y_meas -function_fit(x_meas,*val),bins=int(len(x_meas)/3))
plt.show()

#looks a bit gauss distributed
print('the formula is: a*x**2+b*x+c with parameters a = %.2f +/ - %.2f, b= %.1f+/ -%.1f, c=%.1
#Note that the number of decimal placed can differ as each time a new dataset is made!
```

Exercise 5.8*

```
a_err = np.sqrt(covariance[0,0])
```

```
p_err = np.sqrt(covariance[1,1])

# Uncomment to debug :)
#print(a_err)
#print(p_err)

# A bit nerdy but cool: automatically format the string output with the
# correct number of digits :)

# (you can also just read off the number of digits by hand and "hard code" it...)

a_dig = -(int(np.log10(a_err)) -1)
p_dig = -(int(np.log10(p_err)) -1)

# Uncomment to debug :)
#print(a_dig)
#print(p_dig)

a_fmt = "%." + str(a_dig) + "f"
p_fmt = "%." + str(p_dig) + "f"

# The units on a are a bit funny, so we'll skip the units on this one...
a_msg = "a has the value %s +/ - %s" % (a_fmt, a_fmt)
a_msg = a_msg % (a_fit, a_err)
print(a_msg)

p_msg = "p has the value %s +/ - %s" % (p_fmt, p_fmt)
p_msg = p_msg % (p_fit, p_err)
print(p_msg)
```

Exercise 5.9

```
y = 2.064 * np.cos(0.25*(2*np.pi)*x+0.45)+0.0075*x

#load our data
data = np.loadtxt('dataset.csv',delimiter=';')

x_data = data[:,0]
y_data = data[:,1]

x_test = np.linspace(min(x_data),max(x_data),1000)

#see what it looks like
plt.figure()
plt.plot(x_data,y_data,'k.')
plt.show()

#First guess:
def functie(x,a,b,c):
    return a*np.cos(b*x+c)

val, cov = curve_fit(functie,x_data,y_data,p0=(2.05,1.57,0.5))
print(val)
#see what it yields, playing with the inital guesses is required!
```

```
plt.figure()
plt.plot(x_data,y_data,'k.')
plt.plot(x_test,functie(x_test,*val))
plt.show()

#looking at the residuals
plt.figure()
plt.plot(x_data,y_data -functie(x_data,*val))
plt.show()

#Our data clearly shows a hidden pattern!!!
#second guess

def functie(x,a,b,c,d):
    return a*np.cos(b*x+c)+d*x

val, cov = curve_fit(functie,x_data,y_data,p0=(2.05,1.57,0.5,0.05)) #estimate for d from slope
print(val)


plt.figure()
plt.plot(x_data,y_data,'k.')
plt.plot(x_test,functie(x_test,*val))
plt.show()

#looking at the residuals
plt.figure()
plt.plot(x_data,y_data -functie(x_data,*val))
plt.show()

x = np.linspace(0,6*np.pi,200)
y = 2.064 * np.cos(0.25*(2*np.pi)*x+0.45)+0.0075*x

dataset = np.zeros([len(x),2])
dataset[:,0] = x
dataset[:,1] = y
np.savetxt('dataset.csv',dataset,delimiter=';')
plt.figure()
plt.plot(x,y,'k.')
plt.show()
```

0.2.2 The value of values

> **Note**
> This is meant as an interactive notebook, click the ON-buttom (enable compute) at the right to see and interact with the code and the output!

What is, in science, the value of a value? In physics, this depends on how certain you are of this value. As quantities are determined using experiments, and these experiments are subject to error and uncertainties, the value can only be determined to some degree. We are never 100% sure of the exact value. Therefore, values are given with their uncertainty: $g = 9.81 \pm 0.01$ m/s^2. So how do we determine to what extent a value is certain?

Errors and uncertainty

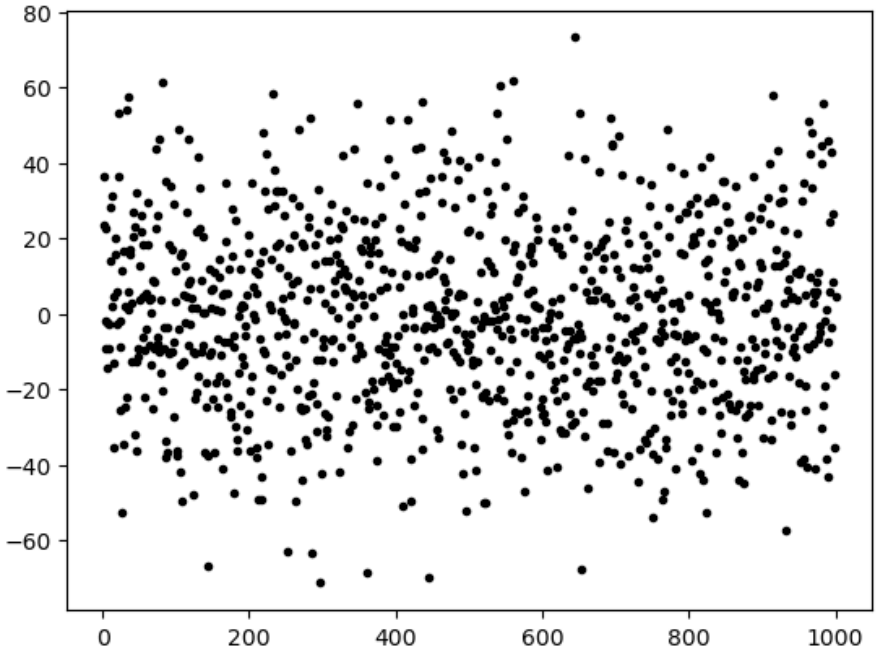
Uncertainties in values can arise from the precision of the instruments used in the experiment, errors made by the person doing the experiment, vibrations, temperature effects, and fundamental errors related to the phenomenon being studied. Some of these uncertainties can be reduced, others just have to be accepted. Whatever their cause, these effects influence experiments and their outcomes of the experiments and therefore influence the uncertainty in the quantities we want to determine.

Gaussian noise Random errors will usually conform to a Gaussian distribution. The probability of an error of some sort occurring can be calculated through:

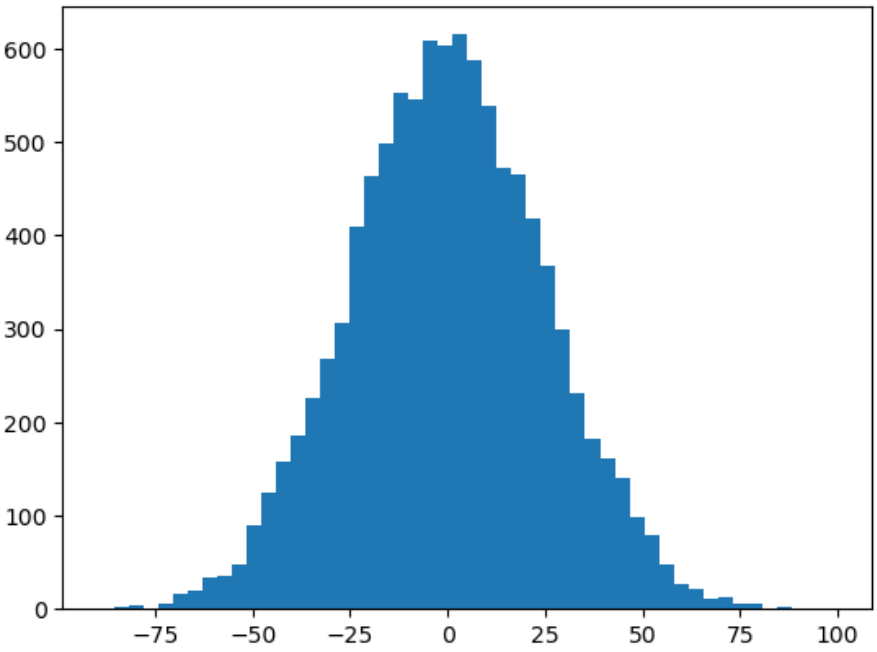
$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{9}$$

In this function, μ is the average value of the error, σ the standard deviation, a measure of the spread of the error, x is the value of the error. In section Repeating measurements we assume that the errors we have to deal with are following a Gaussian distribution. The graph below shows what Gaussian noise looks like, following the probability density function shown in the second graph.

```
import matplotlib.pyplot as plt
import numpy as np
import math
from scipy.optimize import curve_fit
from ipywidgets import interact
from scipy import special
from sympy import init_printing

mu = 0
sigma = 25
N = 10000 #number of data points

y = np.random.normal(mu,sigma,N) #noise creation
x = np.linspace(1,N,N)

plt.figure()
plt.plot(x[:1000],y[:1000],'k.')
plt.show()
print("""(a) A scatter plot of the noise. Although most data are in between -25 and 25, one ca

plt.hist(y,bins = 50) #bins chosen relatively large, but at random
plt.show()
print("""(b) A histogram of the noise. One can easily see that roughly 2/3 of the data is with
```



(a) A scatter plot of the noise. Although most data are in between -25 and 25, one can find data points with values >50.



(b) A histogram of the noise. One can easily see that roughly 2/3 of the data is within ţ \pm \sigma and 90%

Systematic error If there is, e.g., a calibration error of the instrument, a systematic error occurs with each and every measurement. If your ruler starts at 0.2 cm but you didn't notice, this will cause a systematic error.

If you suspect a systematic error, you can look for it using e.g. Python. If the to be fitted function is $F = \frac{\alpha}{r^4}$ and you suspect a systematic error in the distance, r, you can try to fit the function $F = \frac{\alpha}{(r+\Delta r)^4}$. You still have to validate whether the systematic error is within a sensible range and whether there is indeed a systematic error.

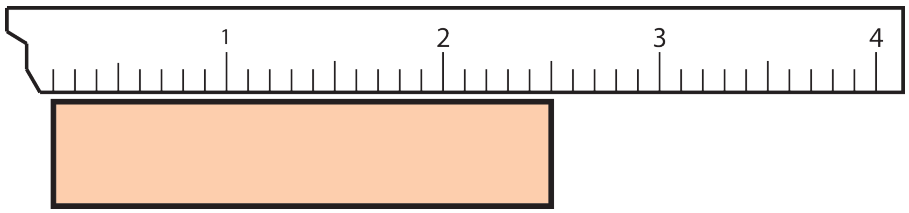
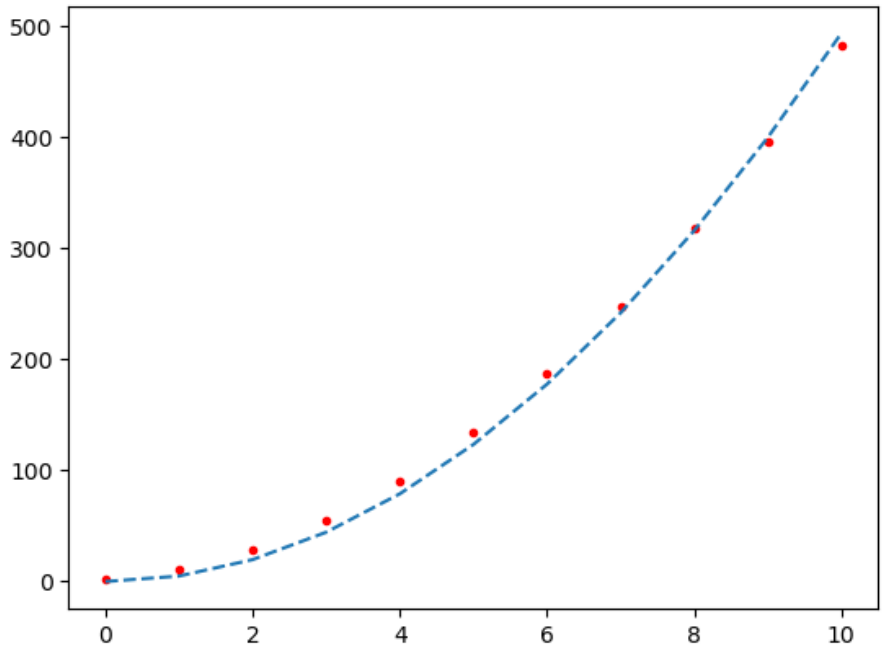


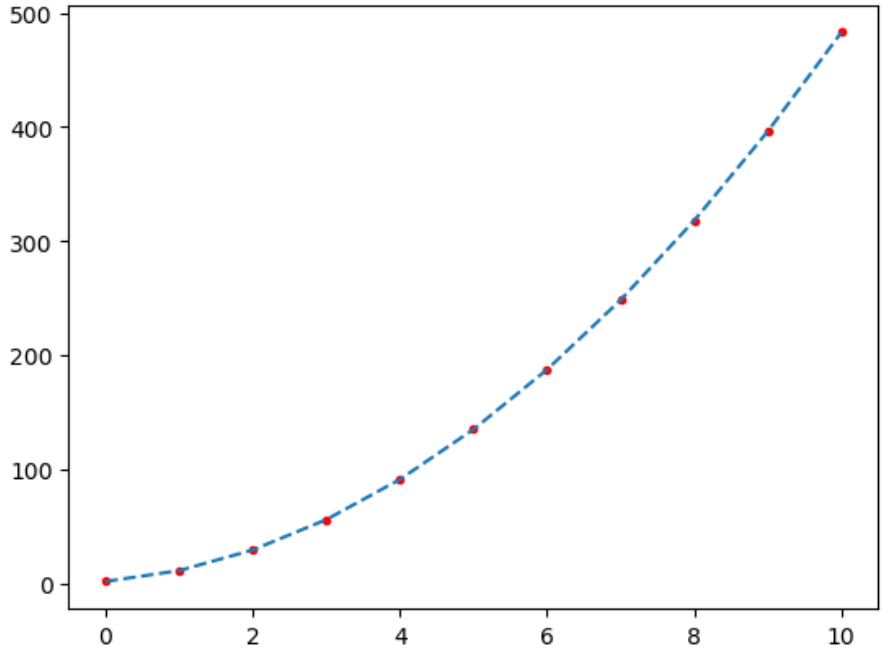
Figure 5: A not calibrated instrument will probably result in a systematic error.

Figuring out whether you have the problem of a systematic error can be done by analysing the residuals. If $M(x)$ are the values of your measurements at certain point x, and $F(x)$ is the value of the fitted function at the same point x, the residuals are defined by $R(x) = M(x) - F(x)$.

```
# Code to show the influence of a systematic error (in x)
#create data
x = np.linspace(0,10,11)

def quadratic(x):
    return 4.3*(x+.6)**2 + np.random.normal(0,1)

y= quadratic(x)

#curvefit without systematic error
def solvex1(x,a):
    return a*x**2

pval_1, pcov_1 = curve_fit(solvex1,x,y)
y2 = solvex1(x,pval_1[0])


plt.plot(x,y,'r.')
plt.plot(x,y2,' - -')
plt.show()
print(r"""(a) A curve fit using least square method without compensation for a systematic erro

#curvefit with system
def solvex2(x,a,b):
    return a*(x+b)**2

pval_2, pcov_2 = curve_fit(solvex2,x,y)
y3 = solvex2(x,pval_2[0],pval_2[1])

plt.plot(x,y,'r.')
plt.plot(x,y3,' - -')
plt.show()
print(r"""(b) A curve fit using least square method with compensation for a systematic error $
```



(a) A curve fit using least square method without compensation for a systematic error y = a $\



(b) A curve fit using least square method with compensation for a systematic error y = a $\cdo

Repeating measurements

Repeating a measurement helps us determine the 'exact' value. The best estimate of the exact value is the mean:

$$\overline{x} = \frac{\sum_{i=1}^{N} x_i}{N} \tag{10}$$

in which x_i is a measurement and N is the number of repeated measurements.

In the analysis of experimental data, an important parameter is the standard deviation, σ. If the experiment is done again, the chance that the value is between $\overline{x} \pm \sigma$ is 2/3. The standard deviation is calculated by

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N-1}} \tag{11}$$

There is another way to determine the standard deviation more quickly, but this is somewhat less accurate (the rough-and-ready approach). The standard deviation is roughly $\frac{2}{3}(x_{max} - \overline{x})$.

If the same experiment is repeated, the average value will differ slightly each time. This means that their is an uncertainty within the average. This parameter is called the standard deviation of the mean, α:

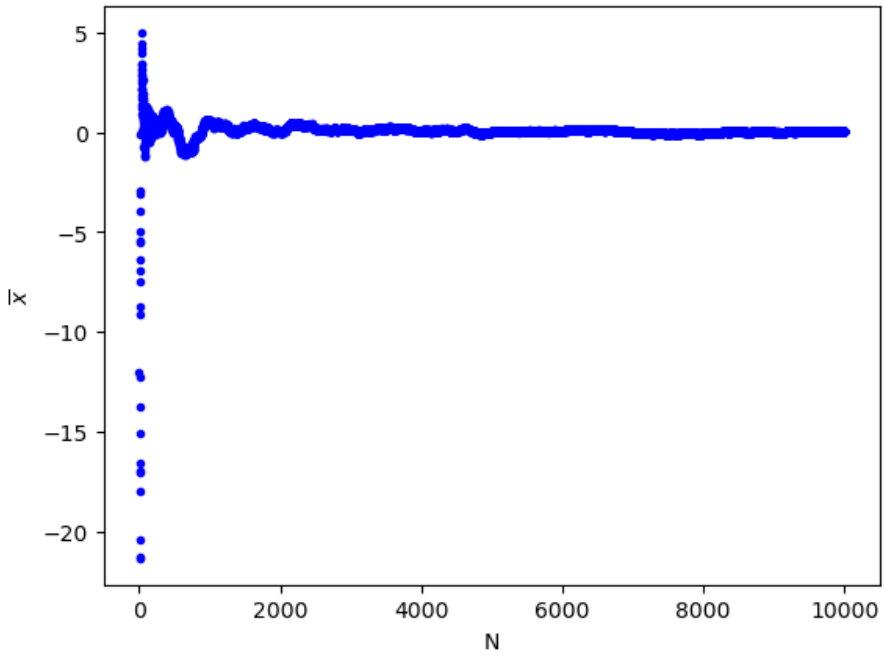
$$\alpha = \frac{\sigma}{\sqrt{N}} \tag{12}$$

This uncertainty tells you that if the entire experiment is repeated, there is a 2/3 change that the average value is within $\overline{x} \pm \alpha$.

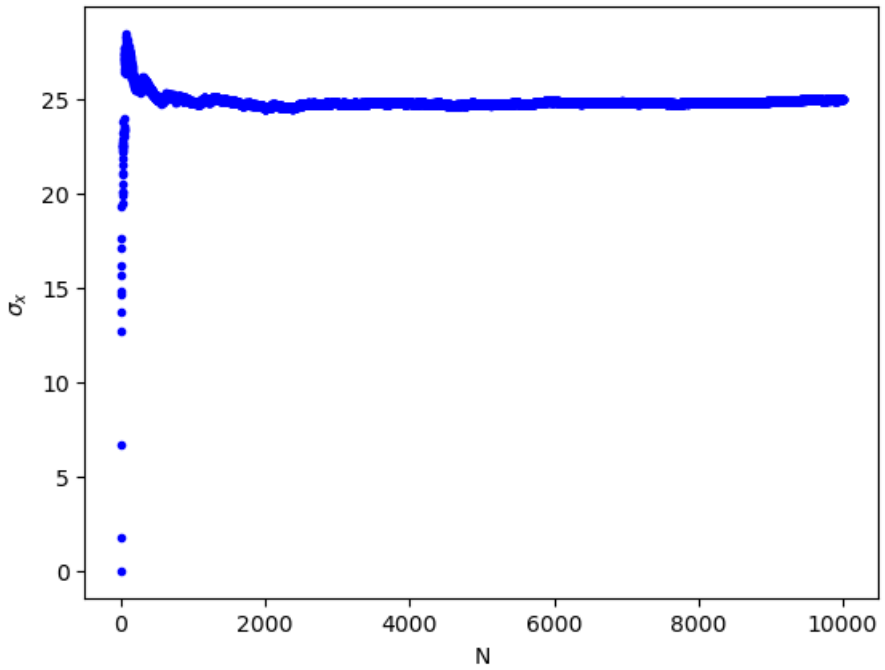
The average, standard deviation and standard error as function of N noise samples. It can be seen that the average and standard deviation do not change much and only get better determined. The standard error decreases with $\sqrt{N}$.

```
mu = 0
sigma = 25
N = 10000 #number of data points

y = np.random.normal(mu,sigma,N) #noise creation
x = np.linspace(1,N,N)

#creates and fills list for average,std, and error against the number of data points used
av_n = []
std_n = []
error_n = []
for n in range(1,N+1):
    av_n.append(np.mean(y[:n]))
    std_n.append(np.std(y[:n]))
    error_n.append(std_n[ -1]/np.sqrt(n))

plt.figure()

plt.xlabel('N')
plt.ylabel('$\overline{x}$')

plt.plot(x, av_n, "b.")
plt.show()
print("""(a) The average value of N noise samples. It can be clearly seen that the value conve


plt.figure()

plt.xlabel('N')
plt.ylabel('$\sigma_x$')

plt.plot(x, std_n, "b.")
plt.show()
print("""(b) The standard deviation of N noise samples. It can be clearly seen that the value


plt.figure()
```

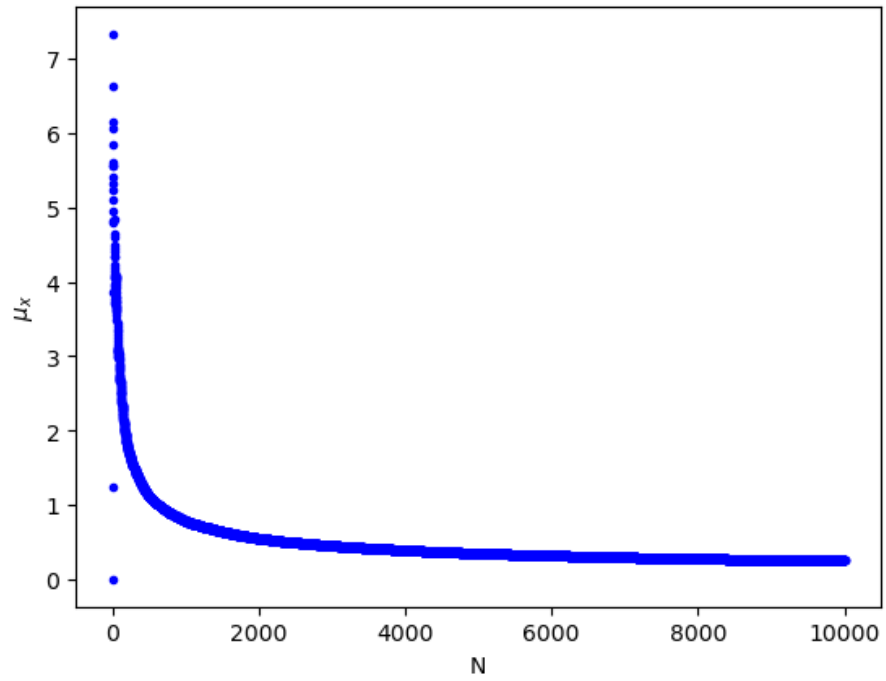
```
plt.xlabel('N')
plt.ylabel('$\mu_x$')

plt.plot(x, error_n, "b.")
plt.show()
print(""" (c) The standard error of N noise samples. It can be seen that the uncertainty decre
```



(a) The average value of N noise samples. It can be clearly seen that the value converges to 0



(b) The standard deviation of N noise samples. It can be clearly seen that the value converges



(c) The standard error of N noise samples. It can be seen that the uncertainty decreases.

Chauvenet's criterion What if a measurement is repeated ten times, and one value is very different from the rest? Can it just be discarded? To decide whether a value can be discarded, one can use the theory above and extend it. To start, you calculate the mean and the standard deviation. Subsequently you calculate the occurrence of the outlier P_{out}, using the error function: $P_{out} = 2Erf(x_{out}, \overline{x}, \sigma)$. You can use this site to use the error function

If $N \cdot P_{out}$ is smaller than 0.5, the measurement may be discarded. Disregarding a measurement should be mentioned in the report! You also have to calculate a new mean value and uncertainty as the data set has changed.

Error Function(s) There are two types of error functions, the $erf(x)$ and the $Erf(x, \overline{x}, \sigma)$. Which are defined like this:

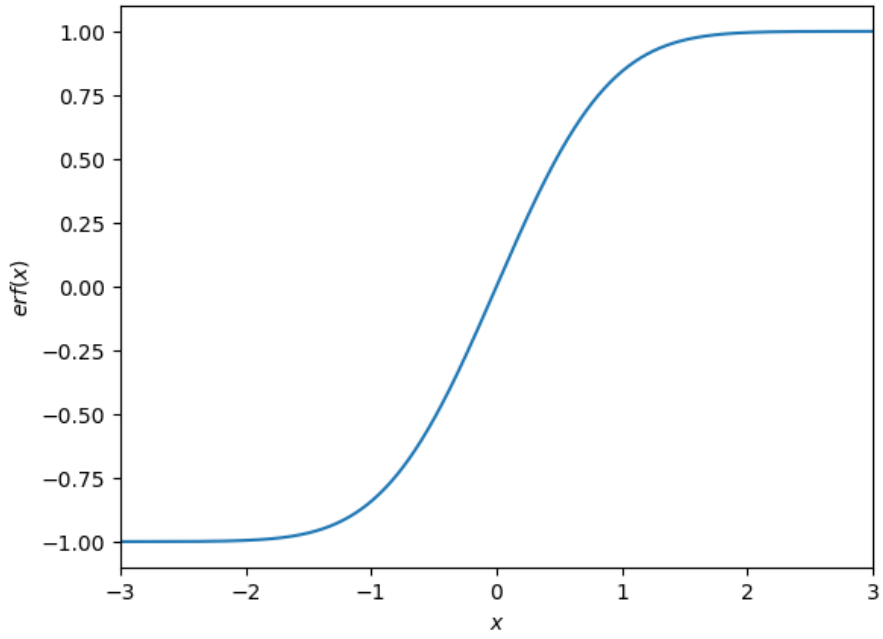
$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{13}$$

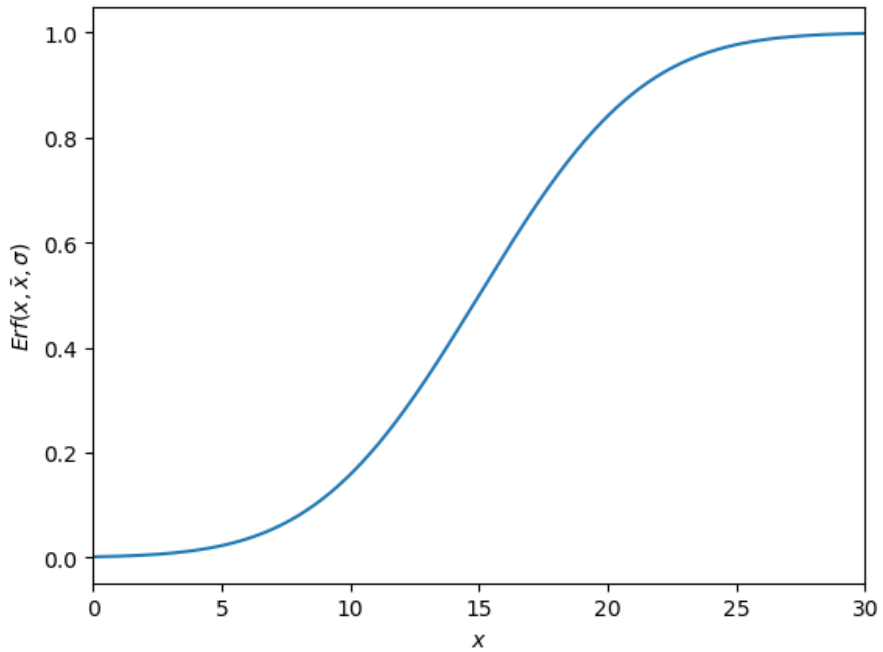
$$Erf(x, \overline{x}, \sigma_x) = \frac{1}{2}\left[1 + erf\left(\frac{x - \overline{x}}{\sqrt{2}\sigma_x}\right)\right] \tag{14}$$

The easiest way is to use the error function is to import it from the scipy package. A plot of both functions can be seen in the graph below.

```
#erf(x)
x = np.linspace( -3, 3, 1000)
plt.plot(x, special.erf(x))
plt.xlabel('$x$')
plt.ylabel('$erf(x)$')
plt.xlim( -3,3)
plt.show()

#Erf(x_out, x_bar, sigma)
# parameters
sigma = 5
x_bar = 15
```

```
def Erf(x, x_bar, sigma):
    return 0.5*(1+ special.erf((x -x_bar)/(np.sqrt(2)*sigma)))

x = np.linspace(0, 30, 10000)
plt.plot(x, Erf(x, x_bar, sigma))
plt.xlabel('$x$')
plt.ylabel(r'$Erf(x,\bar{x},\sigma)$')
plt.xlim(0,30)
plt.show()
```





What can be seen is that the function $Erf(x, \overline{x}, \sigma)$ is basically a shifted version of the the $erf(x)$ function. As by dividing it by two and adding 0.5 shifts it up and makes it smaller. The $x_{out} - \overline{x}$ shifts it so that the mean is at the center of the function and dividing it by σ 'stretches' the function so that it is in the right range. What one also can see if that when you have an outlier which is higher than the mean the Erf will return a value higher than 0.5 which will always result in a value which cannot be discarded. When this happens you have to do $1 - Erf(x_{out}, \overline{x}, \sigma)$, this you multiply by $2N$,

or one can use:

$$1 - erf\left(\frac{|x_{out} - \overline{x}|}{\sigma}\right) \tag{15}$$

This you still have to multiply by N (not by 2).

Significant figures

Significant figures are essential to physics. Most of you are probably already familiar with them. Significant figures are important because they indicate the uncertainty of a value. The number of figures after the comma of $\overline{x}$ and α are always the same!

$$20 \pm 1 \tag{16}$$

$$0.25 \pm 0.02 \tag{17}$$

$$10.25 \pm 0.20 \tag{18}$$

A brief reminder on how to determine the number of significant figures:

- All non-zero digits are significant: $2.998 \cdot 10^8$ m/s has four significant figures.
- All zeroes between non-zero digits are significant: $6.02214179 \cdot 10^{23}$ mol^{-1} has nine significant figures.
- Zeroes to the left of the first non-zero digits are not significant: 0.51 MeV has two significant figures.
- Zeroes at the end of a number to the right of the decimal point are significant: $1.60 \cdot 10^{-19}$C has three significant figures.
- If a number ends in zeroes without a decimal point, the zeroes might be significant: 270 might have two or three significant figures.

Rules For most numbers, it is not hard to round of to the correct number of significant figures:

$$6.626.6 \tag{19}$$

$$5.675.7 \tag{20}$$

However, always rounding up 0.5 to 1 will result in a higher rounded value. So the rule is: even numbers before a 5 are cut, odd numbers before a 5 are rounded:

$$3.453.4 \text{ (since 4 is even)} \tag{21}$$

$$3.553.6 \text{ (since 5 is odd)} \tag{22}$$

With adding and/or subtracting the least number of figures after the comma is decisive. This also means that the total number of significant figures might change:

$$1.23 + 45.6 = 46.8 \tag{23}$$

$$8.2 + 3.5 = 11.7 \tag{24}$$

$$100.5 - 2.5 = 98.0 \tag{25}$$

With multiplication or division the least number of significant figures is decisive:

$$1.2 \cdot 345.6 = 4.1 \cdot 10^2 \tag{26}$$

$$5/2.00 = 2 \tag{27}$$

This last example needs perhaps a further explanation: 5 / 2.00 = 2.5. However, one figure is allowed. 2 is an even number, so the last decimal is cut.

Error propagation

If the uncertainty is known for one value, and that value is used in an equation, the result of that equation will also have some degree of uncertainty. Often, we have multiple variables each having their own degree of uncertainty. There are two ways to calculate this uncertainty: the functional approach, which involves propagating $P \pm \alpha$ throughout the function, and the calculus approach, which is a linearization of the function.

Functional approach When a function Z only depends on a single variable A, the uncertainty α_Z can be calculated using:

$$\alpha_Z = \frac{f(A + \alpha_A) - f(A - \alpha_A)}{2} \tag{28}$$

Note
You just measured the radius of a circle and want to calculate its circumference using $O = 2\pi r$. The measured radius is 2.0±0.2 cm. The circumference will thus be: $O = 2\pi r = 13$ cm. Its uncertainty is given by: $\alpha_O = \frac{2\pi 2.2 - 2\pi 1.8}{2} = 1$ cm

The circumference will thus be: $O = 13 \pm 1$ cm.

When a function Z depends on multiple independent variables, like for instance (P = UI), the uncertainty needs to be calculated separately for each value using (29), (30) and (31). This method can be used for any number of independent variables.

$$\alpha_{P,U} = \frac{P(U + \alpha_U, I) - P(U - \alpha_U, I)}{2} \tag{29}$$

$$\alpha_{P,I} = \frac{P(U, I + \alpha_I) - P(U, I - \alpha_I)}{2} \tag{30}$$

$$\alpha_P = \sqrt{{\alpha_{P,U}}^2 + {\alpha_{P,I}}^2} \tag{31}$$

Note
You measured the voltage over and the current through a light bulb. The voltage is 6.0 ± 0.2 V, the current 0.25 ± 0.01 A. What is the electrical power of the light bulb?

$$\alpha_{P,U} = \frac{P(U + \alpha_U, I) - P(U - \alpha_U, I)}{2} = \frac{6.2 \cdot .25 - 5.8 \cdot .25}{2} = 0.05W \tag{32}$$

$$\alpha_{P,I} = \frac{P(U, I + \alpha_I) - P(U, I - \alpha_I)}{2} = \frac{6.0 \cdot .26 - 6.0 \cdot .24}{2} = 0.06W \tag{33}$$

$$\alpha_P = \sqrt{{\alpha_{P,U}}^2 + {\alpha_{P,I}}^2} = \sqrt{0.05^2 + 0.06^2} = 0.08W \tag{34}$$

$P = 1.50 \pm 0.08$ W

Calculus approach The calculus approach uses a linearization to determine the effect a measured value A has on the value Z. For a single variable function $Z(A)$ the uncertainty in Z is given by:

$$\alpha_Z = \frac{\partial Z}{\partial A}\alpha_A \tag{35}$$

Note
You just measured the radius of a circle and want to calculate its circumference using $O = 2\pi r$. The measured radius is 2.0. The circumference will thus be: $O = 2\pi r = 13$ cm. Its uncertainty is given by: $\frac{dO}{dr}\alpha_r = 2\pi\alpha_r = 1$ cm

The circumference will thus be: $O = 13 \pm 1$ cm.

The linearization will result in an error that becomes noticeable when the error of A is relatively big and there is a lot of curve in the function in that area of A.

The general form for the calculus approach of Z(A, B,...) with uncertainties (α_A), (α_B),... is given in (36).

$$\alpha_Z = \sqrt{\left(\frac{\partial Z}{\partial A}\cdot\alpha_A\right)^2 + \left(\frac{\partial Z}{\partial B}\cdot\alpha_B\right)^2 + ...} \tag{36}$$

An example of the calculus approach for ($P = UI$) is given in (37).

$$\alpha_P^2 = \left(\frac{\partial P}{\partial U}\cdot\alpha_U\right)^2 + \left(\frac{\partial P}{\partial I}\cdot\alpha_I\right)^2 \tag{37}$$

Resulting in (38).

$$\alpha_p^2 = (I\cdot\alpha_U)^2 + (U\cdot\alpha_I)^2 \tag{38}$$

Dividing both sides of the equation by $P(U,I)^2$ yields us a simple and direct equation for α_P:

$$\left(\frac{\alpha_P}{P}\right)^2 = \left(\frac{\alpha_U}{U}\right)^2 + \left(\frac{\alpha_I}{I}\right)^2 \tag{39}$$

More general, for a function $f(y,x) = cy^n x^m$ the uncertainty in f can be calculated by:

$$\left(\frac{\alpha_f}{f}\right)^2 = \left(\frac{\alpha_c}{c}\right)^2 + n^2\left(\frac{\alpha_y}{y}\right)^2 + m^2\left(\frac{\alpha_x}{x}\right)^2 \tag{40}$$

Advantages of both methods The advantage of the functional approach is that it does not use the approximation of a linearization and is therefore more accurate. This is usually only noticeable when the uncertainty of a measured value is large and the function has a lot of bend. The advantage of the calculus approach is that it gives a clearer relation between the uncertainty of a variable and how it propagates into the uncertainty of the determined value

Error in function fit

You have plotted your data (y_i) with their uncertainties and are now looking for a function $y(x_i)$ that best describes the data set (Note: earlier we talked about a measurement M and function F, this is the same principle). You make an educated guess (or use a theoretical framework) to predict the function. An estimate of how well your function predicts the data is given by:

$$\chi^2 = \sum(y_i - y(x_i))^2 \tag{41}$$

If there is a perfect match between data and fit, the sum will be 0. Most curve fitting tools use this principle and are looking for the values for the variables for which this sum is minimal. A good fit goes at least through 2/3 of the error bars.

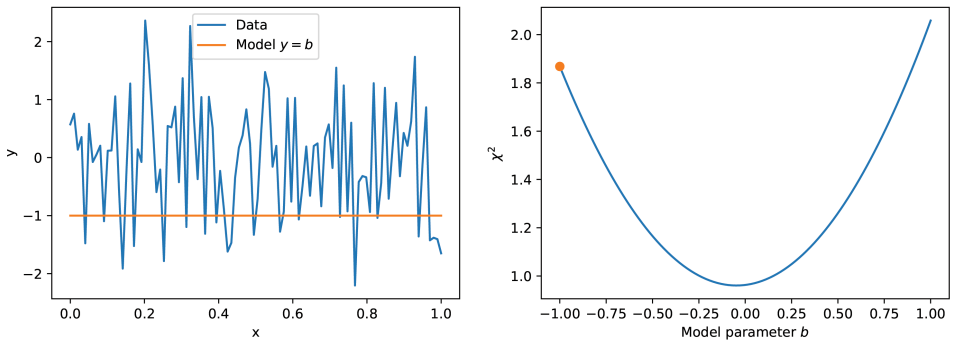


Figure 6: At the left the data and function fit, at the right the sum of the residuals. There is an optimum where the sum of the residuals is minimal.

We can learn a lot about our data by looking at the residuals: $y_i - y(x_i)$. There might still be a detectable pattern, hidden in the noise. Using Python is an excellent way to find out what your data is telling you...

Poisson distribution We covered so far the Gaussian distribution. However, there is also the Poisson distribution which is important when counting, e.g., radioactive decay. The Poisson distribution is a discrete probability distribution.

The chance of k counting events in a certain amount of time, with the expected value λ is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{42}$$

The standard deviation of the Poisson distribution is given by the square root of the expected value: $(\sigma = \sqrt{\lambda})$.

Example

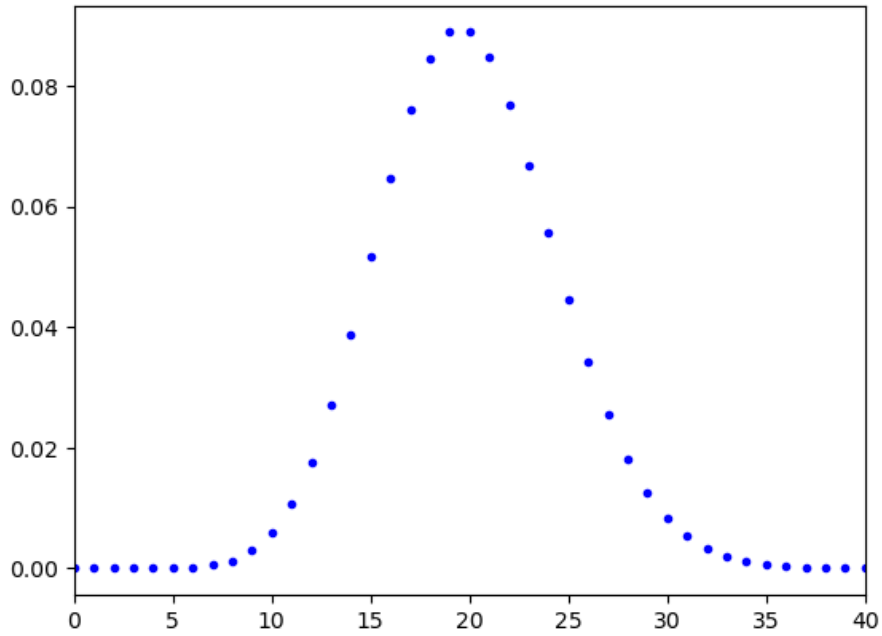
In a restaurant there are on average 20 people per day. How does the probability distribution of the number of people look like?

The probability of the number of visitors is given in Figure 7. The chance is given by:

$$P(X = k) = \frac{20^k e^{-20}}{k!} \tag{43}$$

The shape of the function resembles the Gaussian distribution. This is not really strange as for large numbers of λ the Poisson distribution indeed becomes similar to the Gaussian distribution. However, for small numbers of λ, the Poisson distribution is not symmetrical.

```
#poison plot
lamb = 20 #lambda

k = np.arange(0,2*lamb+1,1)

def Poisson_prob(k, lamb):
    return  np.exp( -lamb)*np.power(lamb, k, dtype = "float")/special.factorial(k)

plt.figure()
plt.plot(k, Poisson_prob(k, lamb), "b.")
plt.xlim(0,2*lamb)
plt.show()
```



Practice

These questions will help you digest the information described above. The answers can be found at the end of this manual. Do not forget to look at the python assignments!

Assignment

The final assignment for Measurement and uncertainty consists of a Python Jupyter Notebook with questions. You are allowed to bring code, this manual, notes. If it concerns a morning session the assignment is available from 8:45. Handing in your work, using Brightspace, can be done until 11:30. If it concerns a afternoon session, the assignment is available from 13:45. Handing in your work, using Brightspace, can be done until 16:00. This information will be provided before the test as well.

0.2.3 Exercises

Some exercises to practice for the exam. The difficulty of the exercises are representative for the exam.

You don't have to make all assignments, however, you should feel comfortable with Python and get well acquainted with measurement & uncertainty.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import erf
from scipy.optimize import curve_fit
```

Mean and standard deviation

You probably know the `np.mean` and the `np.std` commands. However you can also do them yourself using the definitions:

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{44}$$

$$\sigma_x = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2} \tag{45}$$

Use these on the following dataset and see if you get the same results as the `np.mean` and the `np.std` commands.

Exercise 1 The dataset is a measurement from 1 single flat by 11 different persons.

Measurement Number	Total Height (m)
1	141.02
2	148.15
3	157.27
4	115.81
5	149.22
6	207.52
7	180.20
8	161.92
9	164.28
10	206.80
11	42.50

Questions

a Calculate in two different ways the mean value, the standard deviation.

b How would you as a physicist write down the size of the flat?

If you notice a difference between the `np.std` function and the manual function I would encourage you to look at the documentation of the `np.std` function Here.

```
L = np.array([141.02,148.15,157.27,115.81,149.22,207.52,180.2,161.92,164.28,206.8,42.5])
```

Exercise 2 A shop owner wants to know how many people visit his shop. He installs a device that counts the number of people that enter the shop every minute. In total 1000 measurements were done.

Questions

a Import the data_1.dat file.

b Determine the average value as well as the standard deviation

c Plot the data in a histogram. What kind of distribution do you think the data is best described by, and why?

```
Data = np.genfromtxt("data_1.dat")
```

The Error Function and Chauvenaut's theorem

For using Chauvenaut's theorem you need to use the error function. The error function is defined as follows:

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{46}$$

This is a so called non elementary function. It can be approximated numerically, luckily you don't have to do this yourself because `scipy` already has a function for this. The Erf is a function (more formally called the cdf function) which is more often used in statistics and is used in the Chauvenaut's theorem.

$$Erf(x, \overline{x}, \sigma_x) = \frac{1}{2}[1 + erf(\frac{x - \overline{x}}{\sqrt{2}\sigma_x})] \tag{47}$$

Exercise 3 **a** Use `from scipy.special import erf` and make a plot of the error function in the range [-3,3] to see how the function behaves.

b Use this to plot the Erf function, with the σ_x and $\overline{x}$ found in exercise 1(for this you have to change the domain of your plot). This function is the same as the one that can be found on this site.

c Use the dataset from exercise 1 to see if the last value (with length 42.5 m) can be discarded. And if so, what kind of influence does this have on the mean and the std.

d What are the pros and cons of removing a point from a dataset?

```
from scipy.special import erf

#Erf function
def Erf(x,mean_u,sig):
    return 0.5*(1+erf((x -mean_u)/(np.sqrt(2)*sig)))



#plot for error function


#plot for Erf function


#check whether the outlier can be discarded
```

Exercise 4

Height vs. weigth

Attached is a csv file which contains the length (first column) and the weight (second column) of a (male) person.

a Convert the data to metric units as it is now in inches and in lbs.

b Make a scatter plot and see if you if you can fit a linear relation.

c Find the mean and std of both the height and weight.

d Make a histogram for both and overlay a Gaussian to see if the data follows a normal distribution.

```
from scipy.optimize import curve_fit
from scipy.stats import norm

#import data
data = np.genfromtxt('weight -height.csv',delimiter=',',dtype=float,skip_header=1)
height = data[]
```

```
weight = data[]

#convert to metric units


#calculate mean and std


#linear function to fit


#curvefit


#plot
plt.figure(figsize=(12,4))


plt.show()

#Histogram plot #1


#Histogram plot #2
```

Exercise 5 In an experiment 10 measurements were taken of the voltage over and current through a Si-diode. The results are displayed in the following table together with their uncertainties:

I (μA)	α_I (μA)	V (mV)	α_V (mV)
3034	4	670	4
1494	2	636	4
756.1	0.8	604	4
384.9	0.4	572	4
199.5	0.3	541	4
100.6	0.2	505	4
39.93	0.05	465	3
20.11	0.03	432	3
10.23	0.02	400	3
5.00	0.01	365	3
2.556	0.008	331	3
1.269	0.007	296	2
0.601	0.007	257	2
0.295	0.006	221	2
0.137	0.006	181	2
0.067	0.006	145	2

diode is expected to behave according to the following relation: $I = a(e^{bV} - 1)$, where a and b are unknown constants .

a Use the curve_fit function to determine whether this is a valid model to describe the dataset.

Hint: use a logscale on the y-axis

```
I = np.array([3034,1494,756.1,384.9,199.5,100.6,39.93,20.11,10.23,5.00,2.556,1.269,0.601,0.295
a_I = np.array([4,2,0.8,0.4,0.3,0.2,0.05,0.03,0.02,0.01,0.008,0.007,0.007,0.006,0.006,0.006])*
V = np.array([670,636,604,572,541,505,465,432,400,365,331,296,257,221,181,145])*1e -3
```

```
a_V = np.array([4,4,4,4,4,4,3,3,3,3,3,2,2,2,2,2])*1e -3

#Function to fit
def current(V,a,b):
    return a*(np.exp(b*V) -1)


#Make fit
```

Exercise 6 A student measures the position of a simple mass-spring system. Unfortunately, he accidently moves his measuring device during the experiment. He is not sure if the device was put back in the right position and wants to know if there is a systematic error in his data. The dataset consists of 400 position measurements (in cm) over the course of 5 seconds. The data is expected to follow a sine function with an amplitude of 4.5 and a period of 10π.

a Import the data_2.dat file.

b Plot the raw data, calculate and plot the residuals and determine whether there is indeed a systematic error in the data.

c If so, approximate the magnitude of the systematic error and the time at which the occurs.

```
t = np.linspace(0,5,400)

#sine function to fit the data
def sine(x):
    return 4.5*np.sin(x*5*np.pi)

#plot of data and sine function

#calculate residuals

#plot of Residuals
```

Exercise 7

Functional vs. Calculus Approach

You have both seen the functional and the calculus method in calculating the propagation of an error.

Apply both methods on the following functions:

Z(x)=x-1$\frac{}{x+1(48)}$
With $x = 3.2 \pm 0.2$

Z(x)=e$^{x^2}$(49)
With $x = 8.745 \pm 0.005$

Exercise 8 The gravitational force between two bodies can be described with Newton's law of universal gravitation: $F = \frac{Gm_1m_2}{r^2}$, where G is the gravitational constant, m_i the masses of the bodies and r the distance between the bodies.

Suppose that a meteorite of mass $(4.739 \pm 0.154) \cdot 10^8$kg at a distance of $(2.983 \pm 0.037) \cdot 10^6$m is moving towards the earth. Determine the attracting force between the meteorite. Use both the

functional and the calculus approach to calculate the uncertainty in F and compare the results. You can use the following values:

Earth mass: $(5.9722 \pm 0.0006) \cdot 10^{24}$kg

Gravitational constant: $(6.67259 \pm 0.00030) \cdot 10^{-11}$m^3 s^{-2} kg^{-1}

```
#function for gravitational force
def FG(G,m1,m2,r):
    return G * m1 * m2 /(r*r)
#values
G = 6.6759e -11
u_G = 0.00030e -11

m1 = 4.739e8
u_m1 = 0.154e8

m2 = 5.9722e24
u_m2 = 0.0006e24

r = 2.983e6
u_r = 0.037e6

#value of gravitational force


#calculus approach


#Calculate difference between methods
```

Exercise 9 The behaviour of many gases can be well approximated by the ideal gas law: $PV = nRT$, where P is the pressure, V the volume of the gas, n the amount of substance in moles, T the temperature of the gas. R is the ideal gas constant, which is actually just the Boltzmann constant multiplied by the Avogrado constant, with a value of 8.314 J·K^{-1}·mol^{-1}.

A student performs an experiment with a closed containter of 3.23 ± 0.01 litres filled with 0.172 ± 0.001 mol Helium gas. The volume remains constant throughout the experiment and no particles can enter or leave the container. The student slowly heats the gas and measures the change in pressure. The results are shown in the following table.

Temperature (K)	Pressure (kPa)
293.2	128.57
304.4	134.44
313.8	137.93
327.3	145.33
335.0	187.97
348.3	155.16
359.1	158.85
371.6	164.08

uncertainty in T is 0.2 K for all values.

a Calculate the uncertainty in P using the calculus approach

b Fit the data to the model. What do you see? Is the data well described by the ideal gas law?

```
def Press(V,n,R,T):  #ideal gas law
    return n*R*T/V
```

```
#values
R = 8.314

n = 0.172
u_n = 0.001

V = 3.32e -3
u_V = 0.01e -3

#data
T = np.array([293.2,304.4,313.8,327.3,335.0,348.3,359.1,371.6])
u_T = np.ones(len(T))*0.2

P = np.array([128.57,134.44,137.93,145.33,187.97,155.16,158.85,164.08])*1e3

#model


#plot
```

c One data point seems to be an outlier. Use Chauvenet's criterium to determine whether the point can be discarded.

Exercise 10 When measuring there is always a very real possibility of a systematic error. One of these systematic errors can be found in a mass-springsystem. Normally the period of a mass-spring system is given by: $T = 2\pi\sqrt{\frac{m}{C}}$. Here m is the mass and C is the spring constant. However this formula assumes that you have a massless spring, this is not true unfortunately. This means that the mass of the spring is also vibrating, we should thus change the formula to take this into account. This gives the following equation: $T = 2\pi\sqrt{\frac{m+\Delta m}{C}}$, where Δm is the systematic error.

With the measurements that we have we can find both the spring constant and its uncertainties. The array m is an array with the values for the measured m and the array T is an array with all the measured data for the period. You can disregard the invalid use of significant figures.

a Plot the data

b Find the parameters Δm and C with its corresponding uncertainties

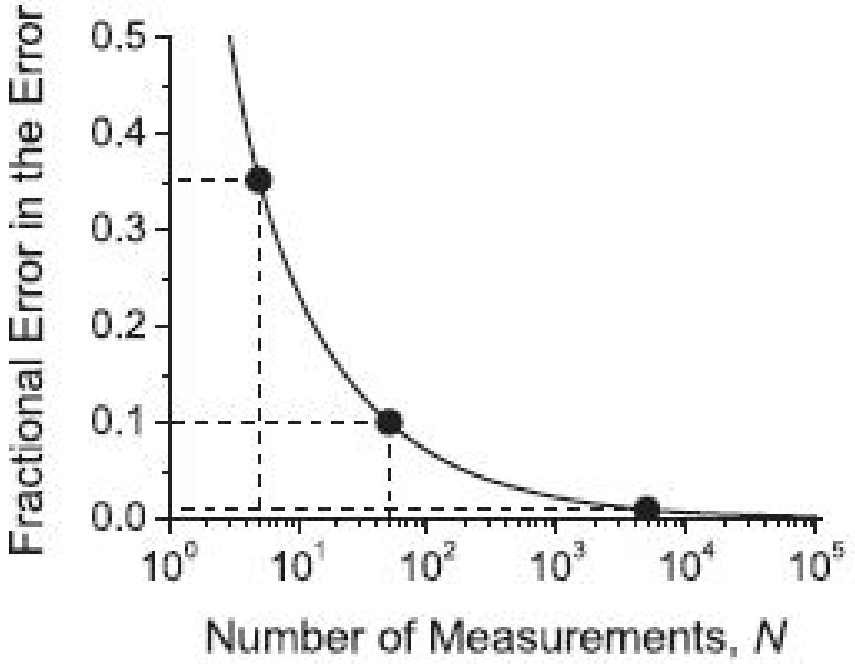
c Plot the fitted function over the data and look at the residuals

```
from scipy.optimize import curve_fit

m = np.array([50,100,150,200,250,300])
T = np.array([2.47,3.43,4.17,4.80,5.35,5.86])
```

Error in the Error

There is always an error in the error. This number tells you whether you use 1 or 2 significant values for your uncertainty.

errror in the error $= 1\frac{}{\sqrt{2N-2}(50)}$



Exercise 11 See if you can reproduce the plot shown above, use a logarithmic scale on the x-axis and make the horizontal and vertical dashed lines (you can only do one point to show that you know how to do it).

Hint: use `plt.hlines` *and* `plt.vlines` *to make it a bit easier (look at the documentation to see how the functions work).*

```
def err_err(N):
    return 1/(np.sqrt(2*N - 2))
```

0.2.4 Answers

Some exercises to practice for the exam. The difficulty of the exercises are representative for the exam.

You don't have to make all assignments, however, you should feel comfortable with Python and get well acquainted with measurement & uncertainty.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import erf
```

Mean and standard deviaton

You probably know the `np.mean` and the `np.std` commands. However you can also do them yourself using the defenitions:

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{51}$$

$$\sigma_x = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu_x)^2} \tag{52}$$

Use these on the following dataset and see if you get the same results as the `np.mean` and the `np.std` commands.

Exercise 1 The dataset is a measurement from 1 single flat by 11 different persons.

Measurement Number	Total Height (m)
1	141.02
2	148.15
3	157.27
4	115.81
5	149.22
6	207.52
7	180.20
8	161.92
9	164.28
10	206.80
11	42.50

Questions

a Calculate in two different ways the mean value, the standard deviation.

b How would you as a phyisicst write the size of the flat down?

If you notice a difference between the ***`np.std`*** *function and the manual function I would encourage you to look at the documentation of the* ***`np.std`*** *function Here.*

```
#Data from the table
L = np.array([141.02,148.15,157.27,115.81,149.22,207.52,180.2,161.92,164.28,206.8,42.5])

#Calculate mean and std 'manually'
Flat_av = np.sum(L)/len(L)
Flat_std = np.sqrt(np.sum((L -Flat_av)**2)/(len(L) -1))

print('Tree height is {:.0f} +/ - {:.0f} m'.format(round(Flat_av, -1),round(Flat_std/np.sqrt(l

#Compare values with numpy functions, if true, nothing happens
assert Flat_av == np.mean(L), 'The results for the mean are different'
assert Flat_std == np.std(L,ddof=1), 'The results for the standard deviation are different'
```

```
Tree height is 150 +/ - 10 m
```

Exercise 2 A shop owner wants to know how many people visit his shop. He installs a device that counts the number of people that enter the shop every minute. In total 1000 measurements were done.

Questions

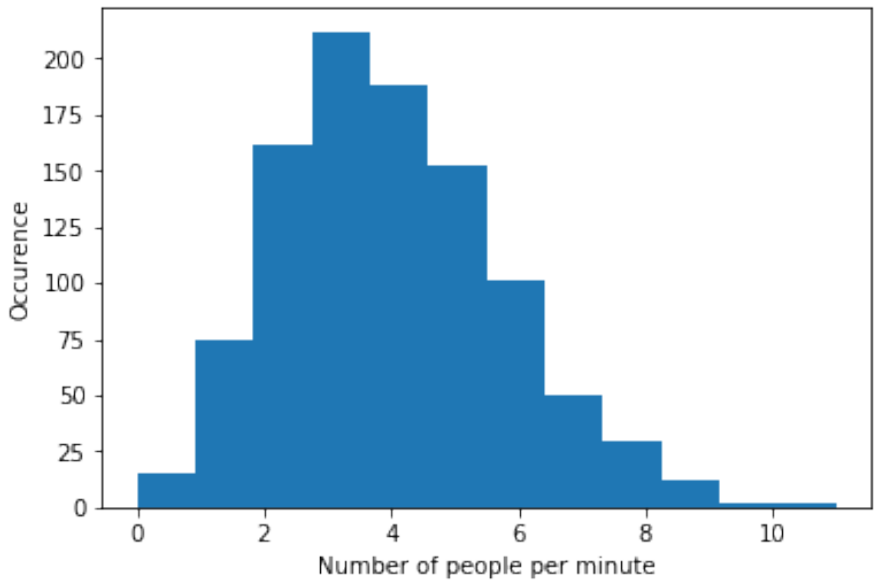
a Import the data_1.dat file.

b Determine the average value as well as the standard deviation

c Plot the data in a histogram. What kind of distribution do you think the data is best described by, and why?

```
#Import data
Data = np.genfromtxt("data_1.dat")

#Calculate mean and std
Pers_mean = np.mean(Data)
Pers_std = np.std(Data,ddof=1)

print('Average value: %.3f \n\
Standard deviation: %.3f ' %(Pers_mean,Pers_std))

#Make the plot
f,g,_ = plt.hist(Data,bins= 12)
plt.xlabel('Number of people per minute')
plt.ylabel('Occurence')
plt.show()

print('It looks like a Poisson distribution. It concerns a counting experiment, so that makes
```

```
Average value: 3.885
Standard deviation: 1.925
```




```
It looks like a Poisson distribution. It concerns a counting experiment, so that makes sense.
```

The Error Function and Chauvenaut's theorem

For using Chauvenaut's theorem you need to use the error function. The error function is defined as follows:

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{53}$$

This is a so called non elementary function. It can be approached numerically, luckily you don't have to do this yourself because `scipy` already has a function for this. The Erf is a function (more formally called the cdf function) which is more often used in statistics and is used in the Chauvenaut's theorem.

$$Erf(x, \overline{x}, \sigma_x) = \frac{1}{2}[1 + erf(\frac{x - \overline{x}}{\sqrt{2}\sigma_x})] \tag{54}$$

Exercise 3 **a** Use `from scipy.special import erf` and make a plot of the error function in the range [-3,3] to see how the function behaves.

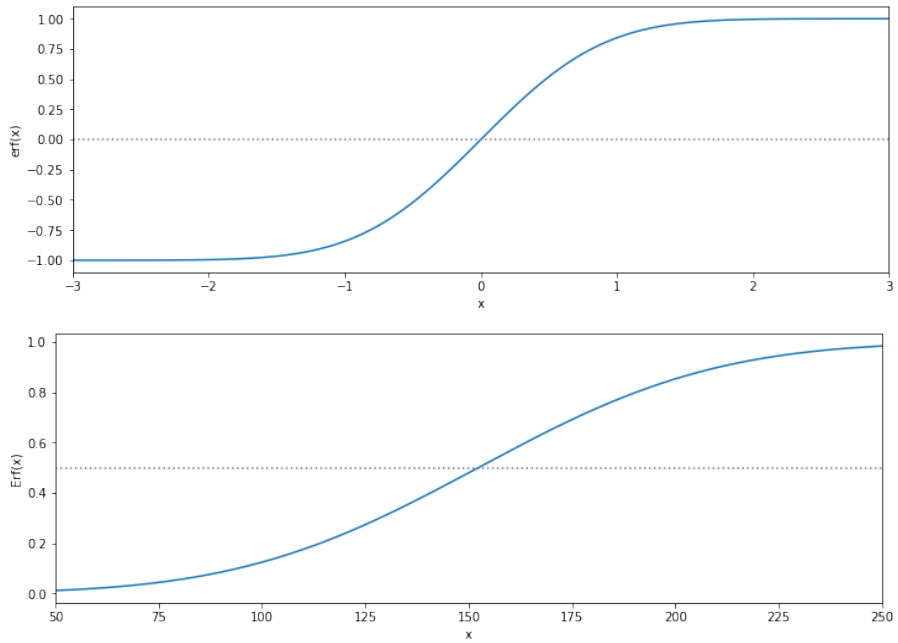
b Use this to plot the Erf function, with the σ and $\overline{x}$ found in exercise 1(for this you have to change the domain of your plot). This function is the same as the one that can be found on this site.

c Use the dataset from exercise 1 to see if the last value (with length 42.5 m) can be discarded. And if so, what kind of influence does this have on the mean and the std.

d What are the pros and cons of removing the point from the dataset.

```
from scipy.special import erf

#Erf function
def Erf(x,mean_u,sig):
    return 0.5*(1 + erf((x -mean_u)/(np.sqrt(2)*sig)))

#Make linspace so it runs from [ -3,3] with 100 steps
x = np.linspace( -3,3,100)

#plot for error function
plt.figure(figsize=(12,4))
plt.plot(x,erf(x))
plt.axhline(c='grey',linestyle=':')
plt.xlabel('x')
plt.ylabel('erf(x)')
plt.xlim(min(x),max(x))
plt.show()

#Make another linspace for range [0,30] with 100 steps
x2 = np.linspace(50,250,200)

#plot for Erf function
plt.figure(figsize=(12,4))
plt.plot(x2,Erf(x2,Flat_av,Flat_std))
plt.axhline(0.5,c='grey',linestyle=':')
plt.xlabel('x')
plt.ylabel('Erf(x)')
plt.xlim(min(x2),max(x2))
plt.show()

#Find outlier
out = np.min(L)

#Check whether the outlier can be discarded
P = 2*Erf(out,Flat_av,Flat_std)

if P*len(L) < 0.5:
    print('The measurement may be discarded.')
```

```
else:
    print('The measurement may not be discarded.')

L_new = np.delete(L,np.where(L == 42.5))

Flat_av_new = np.mean(L_new)
Flat_std_new = np.std(L_new,ddof=1)

print('Tree height is {:.0f} +/ - {:.0f} m'.format(round(Flat_av_new, -1),round(Flat_std_new/n
```




```
The measurement may be discarded.
Tree height is 160 +/ - 10 m
```

The pros of removing specific parts of the dataset is that your mean and your std don't get influenced heavily by specific outliers. But by removing specific parts it is also possible that you are influencing your mean and std for the worse. Especially in small datasets you do not know if something went wrong with measureing or if the std of the measured value is just very large.

Exercise 4

Height vs. weigth

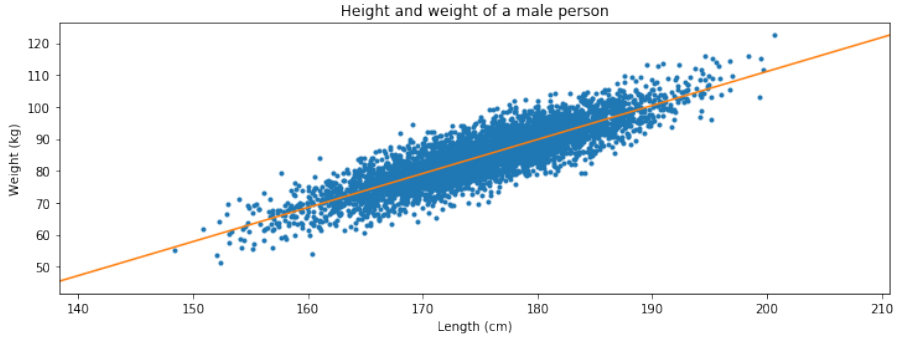
Attached is a csv file which contains the height (first column) and the weight (second column) of a (male) person.

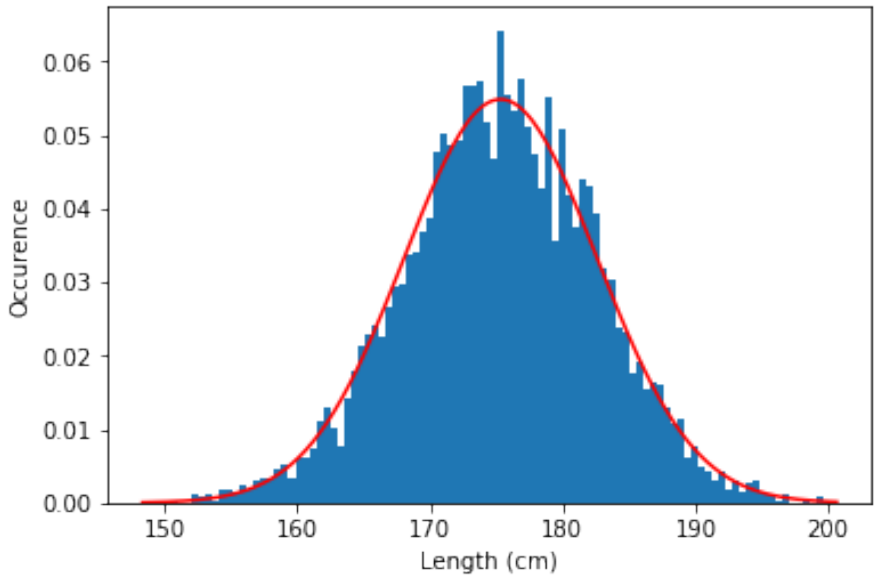
a Convert the data to metric units as it is now in inches and in lbs.
b Make a scatter plot and see if you if you can fit a lineair relation.
c Find the mean and std of both the height and weight.
d Make a histogram for both and overlay a Gaussian to see if the data follows a normal distribution.

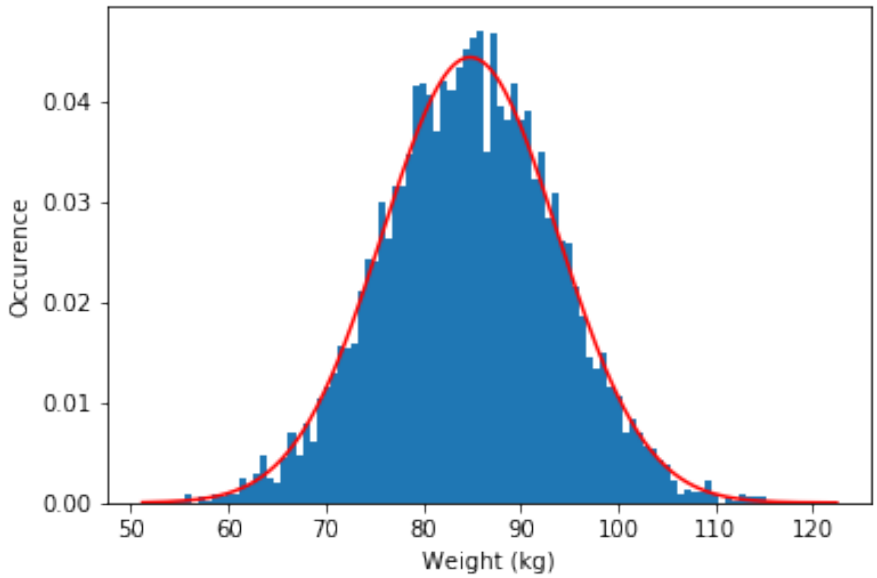
```
from scipy.optimize import curve_fit
from scipy.stats import norm

#Import data
data = np.genfromtxt('weight -height.csv',delimiter=',',dtype=float,skip_header=1)
```

```
#Extract height and weight from dataset
height = data[:,0] #Extract first column
weight = data[:,1] #Extract second column

#Transform to metric
height = height*2.54
weight = weight*0.4536

#Calculate mean and std
height_mean = np.mean(height)
height_std = np.std(height,ddof=1)
weight_mean = np.mean(weight)
weight_std = np.std(weight,ddof=1)

#Linear function to fit
def lin(x,a,b):
    return a*x + b

#Find parameters a and b (in val)
val, cov = curve_fit(lin,height,weight)

#Make x -and y coordinates to plot the fit
x = np.linspace(np.min(height) -10,np.max(height)+10)
y = lin(x,val[0],val[1])

#Plot the data
plt.figure(figsize=(12,4))
plt.title('Height and weight of a male person')
plt.plot(height,weight,'.',label='data')
plt.xlabel('Length (cm)')
plt.ylabel('Weight (kg)')
plt.xlim(min(x),max(x))

#Plot the fit
plt.plot(x,y,label='fit')
plt.show()


### Histograms with gaussians ####

N=200
x2 = np.linspace(np.min(height),np.max(height),N)

#Histogram plot #1
plt.figure()
plt.hist(height,bins=100,density = 1) #Note the density=1 , this rescales the histogram

#Make values for the Gaussian plot
dist = norm.pdf(x2,height_mean,height_std)

#Plot the Gaussian
plt.plot(x2,dist,'r')
plt.xlabel('Length (cm)')
```

```
plt.ylabel('Occurence')
plt.show()

#Make a x array to use in the Gaussian
x2 = np.linspace(np.min(weight),np.max(weight),N)

#Histogram plot #2
plt.figure()
plt.hist(weight,bins=100,density=1) #Note the density=1 , this rescales the histogram

#Make the Gaussian
dist = norm.pdf(x2,weight_mean,weight_std)

#Plot the Gaussian
plt.plot(x2,dist,'r') #Plot the Guassian
plt.xlabel('Weight (kg)')  #xlabel
plt.ylabel('Occurence')   #ylabel
plt.show()  #Show plot
```







Exercise 5 In an experiment 10 measurements were taken of the voltage over and current through a Si-diode. The results are displayed in the following table together with their uncertainties:

I (μA)	α_I (μA)	V (mV)	α_V (mV)
3034	4	670	4
1494	2	636	4
756.1	0.8	604	4
384.9	0.4	572	4
199.5	0.3	541	4
100.6	0.2	505	4
39.93	0.05	465	3
20.11	0.03	432	3
10.23	0.02	400	3
5.00	0.01	365	3
2.556	0.008	331	3
1.269	0.007	296	2
0.601	0.007	257	2
0.295	0.006	221	2
0.137	0.006	181	2
0.067	0.006	145	2

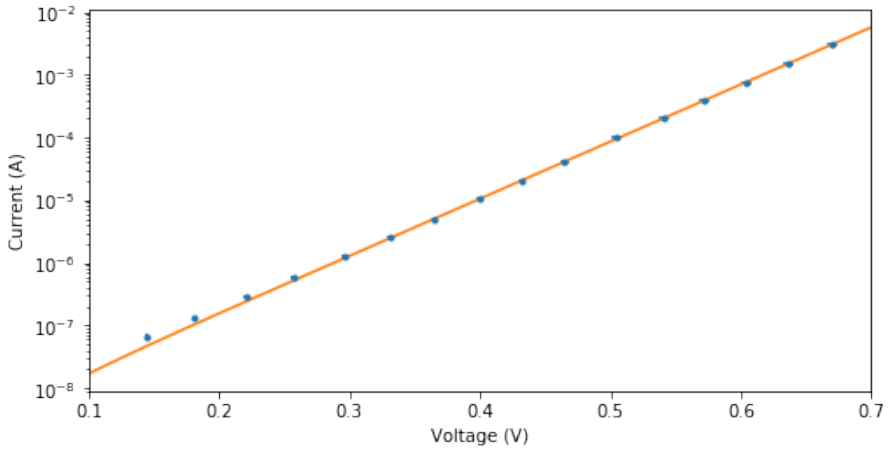
diode is expected to behave according to the following relation: $I = a(e^{bV} - 1)$, where a and b are unknown constants .

a Use the curve_fit function to determine whether this is a valid model to describe the dataset.

Hint: use a logscale on the y-axis

```
from scipy.optimize import curve_fit

#Data
I = np.array([3034,1494,756.1,384.9,199.5,100.6,39.93,20.11,10.23,5.00,2.556,1.269,0.601,0.295
a_I = np.array([4,2,0.8,0.4,0.3,0.2,0.05,0.03,0.02,0.01,0.008,0.007,0.007,0.006,0.006,0.006])*
V = np.array([670,636,604,572,541,505,465,432,400,365,331,296,257,221,181,145])*1e -3
a_V = np.array([4,4,4,4,4,4,3,3,3,3,3,2,2,2,2,2])*1e -3


#Function to fit
```

```
def current(V,a,b):
    return a*(np.exp(b*V) -1)

#Make the fit
popt, pcov = curve_fit(current,V,I)

#Make x -and y coordinates for plotting
x = np.linspace(0.1,0.7)
y = current(x,*popt)

#Make plots
plt.figure(figsize=(8,4))
plt.errorbar(V,I,xerr=a_V,yerr=a_I,linestyle='none',marker= '.',label='data')
plt.plot(x,y,label='fit')
plt.yscale('log')
plt.xlabel('Voltage (V)')
plt.ylabel('Current (A)')
plt.xlim(0.1,0.7)
plt.show()
print('The model is in good agreement with the measurements, it is therefore a valid model.')
```




```
The model is in good agreement with the measurements, it is therefore a valid model.
2.3796115208119935e -09 20.983641479652693
```

Exercise 6 A student measures the position of a simple mass-spring system. Unfortunately, he accidently moves his measuring device during the experiment. He is not sure if the device was put back in the right position and wants to know if there is a systematic error in his data. The dataset consists of 400 position measurements (in cm) over the course of 5 seconds. The data is expected to follow a sine function with an amplitude of 4.5 and a period of 10π.

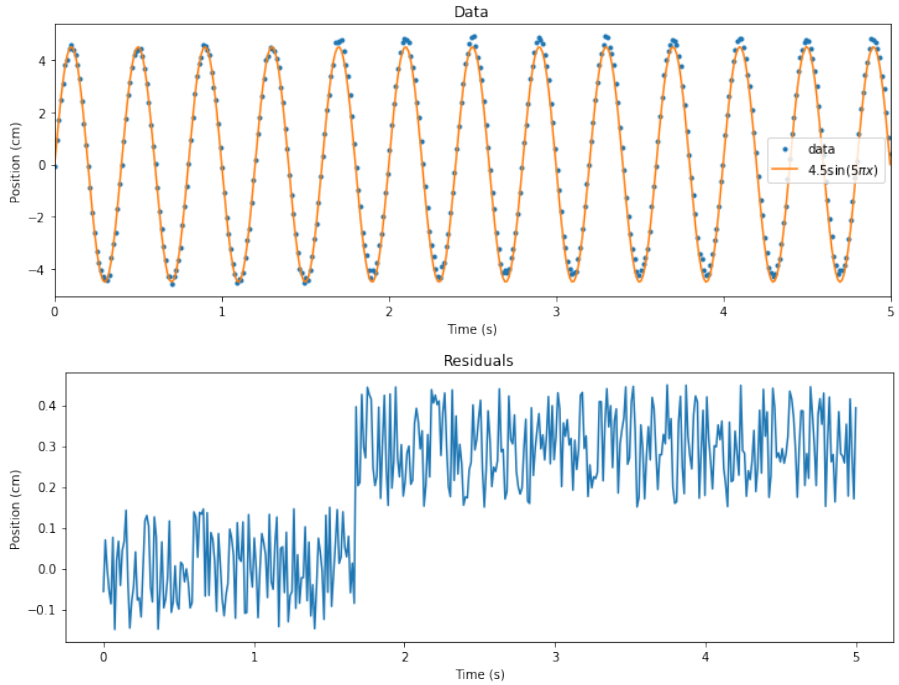
a Import the data_2.dat file.

b Plot the raw data, calculate and plot the residuals and determine whether there is indeed a systematic error in the data.

c If so, approximate the magnitude of the systematic error and the time at which the occurs.

```
#Import data
data = np.genfromtxt('data_2.dat')

#Make linspace based on text given
t = np.linspace(0,5,400)
```

```
#sine function to fit the data
def sine(x):
    return 4.5*np.sin(x*5*np.pi)

#Plot of data and sine function
plt.figure(figsize=(12,4))
plt.plot(t,data, '.', label='data')
plt.plot(t,sine(t), label='$4.5\sin(5\pi x$)')
plt.xlim(min(t),max(t))
plt.ylabel('Position (cm)')
plt.xlabel('Time (s)')
plt.legend()
plt.show()

#Calculate residuals
R = data - sine(t)

#plot of Residuals
plt.figure(figsize=(12,4))
plt.title('Residuals')
plt.plot(t,R)
plt.ylabel('Position (cm)')
plt.xlabel('Time (s)')
plt.show()
```



Explanation: Shift seems to occur around t = 1.7 s and has a magnitude of 0.3 cm

Exercise 7

Functional vs. Calculus Approach

You have both seen the functional and the analytical method in calculating the propagation of an error.

Apply both methods on the following functions:

Z(x)=x-1$\frac{}{x+1(55)}$
With $x = 3.2 \pm 0.2$

Z(x)=$e^{x^2}(56)$
With $x = 8.745 \pm 0.005$

```
#First function
def func1(x):
    return (x -1)/(x+1)

#Second function
def func2(x):
    return np.exp(x**2)

#Function for the functional method
def functional(x,sig,f):
    return (f(x+sig) - f(x -sig))/2

#First function
x_1 = 3.2
sig = 0.2
error_functional = functional(x_1,sig,func1)
error_calculus = sig*((x_1+1) -(x_1 -1))/(x_1+1)**2  #Derivative was calculated by hand

print('First function \n\
Value of Z: %.2f \n\
Error with functional approach: %.2f \n\
Error with calculus approach: %.2f \n\
' %(func1(x_1),error_functional,error_calculus))

#Second function
x_2 = 8.745
sig = 0.005
error_functional = functional(x_2,sig,func2)
error_calculus = sig*2*x_2*func2(x_2)


print('Second function \n\
Value of Z: %.1e \n\
Error with functional approach: %.0e \n\
Error with calculus approach: %.0e \
' %(func2(x_2),error_functional,error_calculus))

First function
Value of Z: 0.52
Error with functional approach: 0.02
Error with calculus approach: 0.02

Second function
Value of Z: 1.6e+33
Error with functional approach: 1e+32
Error with calculus approach: 1e+32
```

Exercise 8 The gravitational force between two bodies can be described with Newton's law of universal gravitation: $F = \frac{Gm_1m_2}{r^2}$, where G is the gravitational constant, m_i the masses of the bodies and r the distance between the bodies.

Suppose that a meteorite of mass $(4.739 \pm 0.154) \cdot 10^8$kg at a distance of $(2.983 \pm 0.037) \cdot 10^6$m is moving towards the earth. Determine the attracting force between the meteorite. Use both the functional and the calculus approach to calculate the uncertainty in F and compare the results. You can use the following values:

Earth mass: $(5.9722 \pm 0.0006) \cdot 10^{24}$kg

Gravitational constant: $(6.67259 \pm 0.00030) \cdot 10^{-11}\text{m}^3\ \text{s}^{-2}\ \text{kg}^{-1}$

```
#function for gravitational force
def FG(G,m1,m2,r):
    return G * m1 * m2 /(r*r)

#values
G = 6.6759e -11
u_G = 0.00030e -11

m1 = 4.739e8
u_m1 = 0.154e8

m2 = 5.9722e24
u_m2 = 0.0006e24

r = 2.983e6
u_r = 0.037e6

#value of gravitatonal force
F_m = FG(G,m1,m2,r)

#Calculus appraoch
r2 = r**2
u_F2 = (m1*m2/r2*u_G)**2 + (G*m2/r2*u_m1)**2 + (G*m1/r2*u_m2)**2 + ( -2*G*m1*m2/(r2*r)*u_r)**2
u_F_calc = np.sqrt(u_F2)

#Funcional approach
U_F2 = ((FG(G+u_G,m1,m2,r) -FG(G -u_G,m1,m2,r))/2)**2 + ((FG(G,m1+u_m1,m2,r) -FG(G,m1 -u_m1,m2
u_F_func = np.sqrt(U_F2)


print("F = %.2f +/ - %.2f 10^10 N \n" %(F_m/1e10,u_F_func/1e10))

diff = np.abs(u_F_calc -u_F_func)

print("Uncertainties \n\
Caluculus approach: %f 10^8 N \n\
Functional approach: %f 10^8 N \n\
Difference: %f 10^8 N" %(u_F_calc/1e8,u_F_func/1e8,diff/1e8))
```

```
F = 2.12 +/ - 0.09 10^10 N

Uncertainties
Caluculus approach: 8.680952 10^8 N
Functional approach: 8.681935 10^8 N
Difference: 0.000984 10^8 N
```

Exercise 9 The behaviour of many gases can be well approximated by the ideal gas law: $PV = nRT$, where P is the pressure, V the volume of the gas, n the amount of substance in moles, T the temperature of the gas. R is the ideal gas constant, which is actually just the Boltzmann constant multiplied by the Avogrado constant, with a value of 8.314 $\mathrm{J{\cdot}K^{-1}{\cdot}mol^{-1}}$.

A student performs an experiment with a closed containter of 3.23 ± 0.01 litres filled with 0.172 ± 0.001 mol Helium gas. The volume remains constant throughout the experiment and no particles can enter or leave the container. The student slowly heats the gas and measures the change in pressure. The results are shown in the following table.

Temperature (K)	Pressure (kPa)
293.2	128.57
304.4	134.44
313.8	137.93
327.3	145.33
335.0	187.97
348.3	155.16
359.1	158.85
371.6	164.08

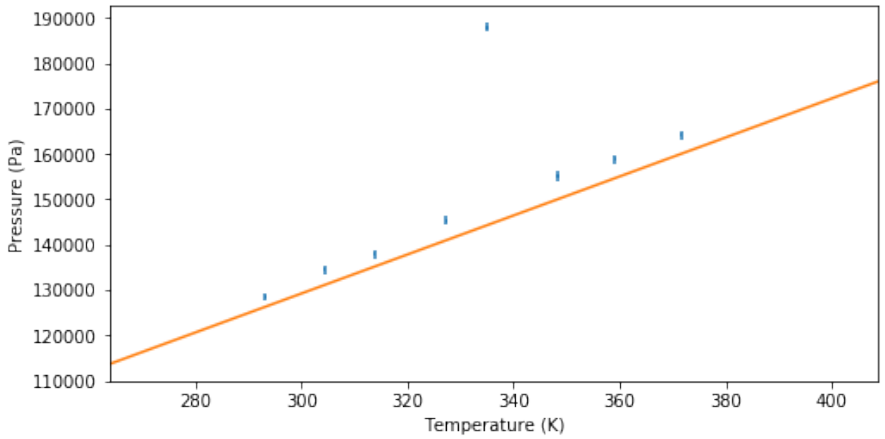
uncertainty in T is 0.2 K for all values.

a Calculate the uncertainty in P using the calculus approach

b Fit the data to the model. What do you see? Is the data well described by the ideal gas law?

```
def Press(V,n,R,T):  #ideal gas law
    return n*R*T/V

#values
R = 8.314

n = 0.172
u_n = 0.001

V = 3.32e -3
u_V = 0.01e -3

#Data
T = np.array([293.2,304.4,313.8,327.3,335.0,348.3,359.1,371.6])
u_T = np.ones(len(T))*0.2

P = np.array([128.57,134.44,137.93,145.33,187.97,155.16,158.85,164.08])*1e3
u_P2 = (R*T/V*u_n)**2 + (n*R/V * u_T)**2 + ( - n*R*T/(V*V)*u_V)**2
u_P = np.sqrt(u_P2)

#model
P_calc = Press(V,n,R,T)

#plot
linT = np.linspace(0.9*np.min(T),1.1*np.max(T),100)
plt.figure(figsize=(8,4))
plt.errorbar(T,P,xerr=u_T,yerr=u_P,label='data',linestyle='none') #data
plt.plot(linT,Press(V,n,R,linT),label='Ideal gas law')
plt.xlabel('Temperature (K)')
plt.ylabel('Pressure (Pa)')
plt.xlim(np.min(linT),np.max(linT))
#plt.legend()
```

```
plt.show()
```



c One data point seems to be an outlier. Use Chauvenet's criterium to determine wether the point can be discarded.

```
#Calculate mean,std and value
Press_mean = np.mean(P)
Press_std = np.std(P,ddof=1)
Press_val = np.max(P)

#Use Erf which was defined before
Q = Erf(Press_val,Press_mean,Press_std)

#It is a higher outlier so 1 -Q
if Q > 0.5:
    Q = (1 -Q)

#Use Chauvenets criterion
N =  2 * len(P) *Q

if N < 0.5:
    print('The value can be discarded.')
else:
    print('The value cannot be discarded.')

The value can be discarded.
```

Exercise 10 When measuring there is always a very real possibility of a systematic error. One of these systematic erros can be found in a mass-springsystem. Normally the period of a mass-spring system is given by: $T = 2\pi\sqrt{\frac{m}{C}}$. Here m is the mass and C is the spring constant. However this formula assumes that you have a massless spring, this is not true onfortunately. This means that the mass of the spring is also vibrating, we should thus change the formula to take this into account. This gives the following equation: $T = 2\pi\sqrt{\frac{m+\Delta m}{C}}$, where Δm is the systematic error.

With the measurements that we have we can find both the spring constant and its uncertainties. The array m is an array with the values for the measured m and the array T is an array with all the measured data for the period. You can disregard the invalid use of significant figures.

a Plot the data

b Find the parameters Δm and C with its corresponding uncertainties

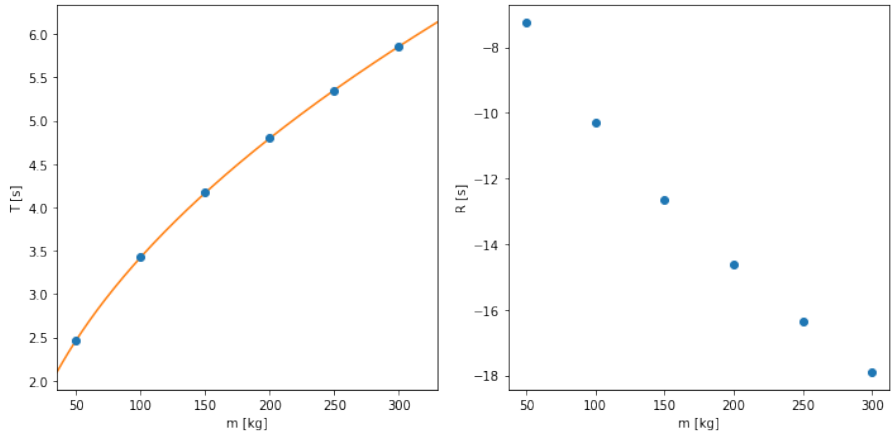
c Plot the fitted function over the data and look at the residuals

```
from scipy.optimize import curve_fit
```

```
m = np.array([50,100,150,200,250,300])
T = np.array([2.47,3.43,4.17,4.80,5.35,5.86])


#Define Function to fit
def Per(m,dm,C):
    return 2*np.pi*np.sqrt((m+dm)/C)

#Make figure
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(121)
ax.errorbar(m,T,linestyle='none',marker='o')
ax.set_xlabel('m [kg]')
ax.set_ylabel('T [s]')

#Make Fit
vals, uncmat = curve_fit(Per,m,T)

#Make linspace
linm = np.linspace(0.7*np.min(m),1.1*np.max(m),200)

#Plot the fitted line
ax.plot(linm,Per(linm,*vals))

#Set the x -limit
ax.set_xlim(np.min(linm),np.max(linm))

#Finde the uncertainties
u_dm, u_C = np.sqrt(np.diag(uncmat))


#Print the values gotten and its uncertainties
print('dm = {:.1f} +/ - {:.1f} m'.format(vals[0],u_dm))
print('C = {:.1f} +/ - {:.1f} kg m/s^2'.format(vals[1],u_C))

#Make a residual analysis
r = T - Per(m,*popt)

#Plot the risiduals
ax2 = fig.add_subplot(122)
ax2.plot(m,r,linestyle='none',marker='o')
ax2.set_xlabel('m [kg]')
ax2.set_ylabel('R [s]')

#Make it a bit pretier
plt.tight_layout()
```

```
dm = 4.1 +/ - 0.2 m
C = 349.9 +/ - 0.4 kg m/s^2
```

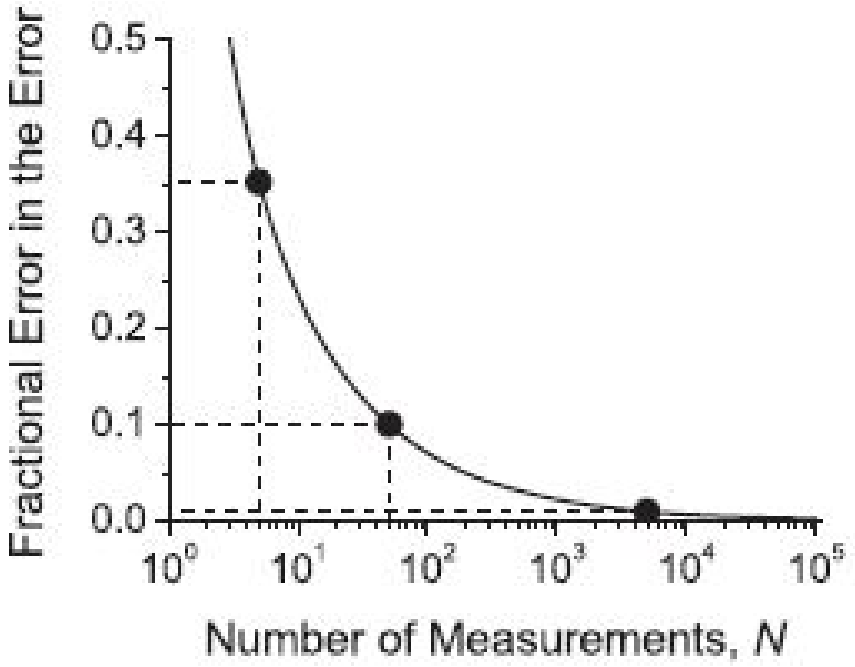
```
C:\Programs\Anaconda3\lib\site -packages\ipykernel_launcher.py:9: RuntimeWarning: invalid valu
  if __name__ == '__main__':
```



Error in the Error

There is always an error in the error. This number tells you whether you use 1 or 2 significant values for your uncertainty.

$$\text{errror in the error} = 1\frac{}{\sqrt{2N-2}(57)}$$

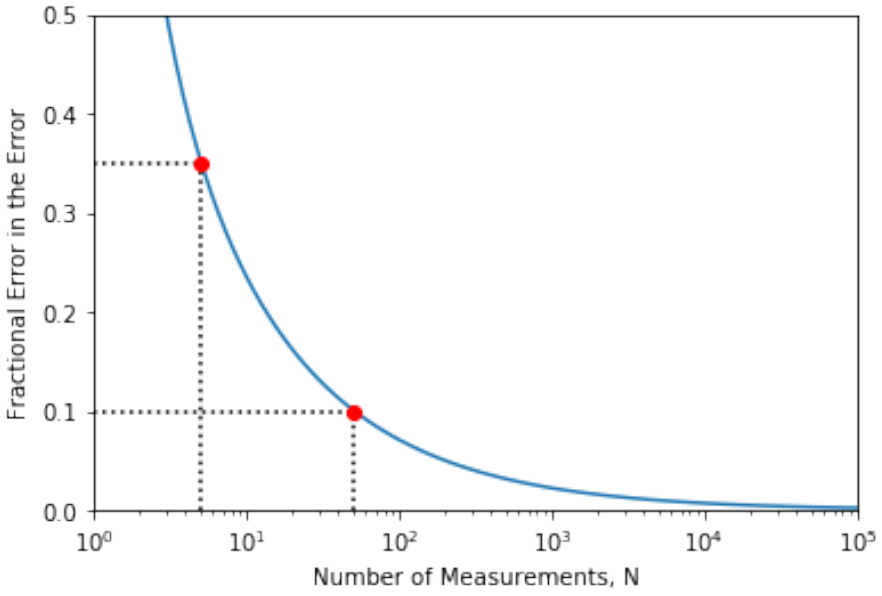


Exercise 11 See if you can reproduce the plot shown above, use a logarithmic scale on the x-axis and make the horizontal and vertical dashed lines (you can only do one point to show that you know how to do it).

Hint: use `plt.hlines` *and* `plt.vlines` *to make it a bit easier (look at the documentation to see how the functions work).*

```
def err_err(N):
    return 1/(np.sqrt(2*N - 2))

#Make logspace and fill in y values
x = np.logspace(0.4,5,100)
y = err_err(x)
```

```
#Make plot
plt.figure(figsize=(6,4))
plt.plot(x,y)
plt.xscale('log')
plt.ylabel('Fractional Error in the Error')
plt.xlabel('Number of Measurements, N')
plt.xlim(1,1e5)
plt.ylim(0,0.5)

#For point (5,0.35)
plt.plot(5,0.35,'o',c='red')
plt.hlines(0.35,1,5,linestyle=':')
plt.vlines(5,0,0.35,linestyle=':')

#For point (50,0.1)
plt.plot(50,0.1,'o',c='red')
plt.hlines(0.1,1,50,linestyle=':')
plt.vlines(50,0,0.1,linestyle=':')

plt.show()
```



0.2.5 Exam Data-analysis in Python

This assignment has to be made individually. You can start at 13:30h, it has to submitted before 17:30h.

You are allowed to use previous materials you made and used in Python.

Important:

```
provide enough information so we can understand what you are doing.
you can create cells (as markdown) to create comments and explanations.
you can use # as well to comment.
always use significant figures and units (if given) for your final answer
```

You are allowed to calculate things by hand as well, but then show or explain your final answer.

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

from scipy.optimize import curve_fit
from scipy.stats import norm
```

Task 1

In this task you will show that you understand the basics of Python.

1. Make an array "x_raw" running from 0 to 100 (including both 0 and 100) with an interval where all elements are integers (so [0,1,2,3...100]). (1pt)
2. Now use x_raw to make a new array "x_even" which consists of all even numbers in the array "x_raw". (1pt)
3. Define a function $f(a, x)$ that returns $a \cdot x^2$. Use it to make an array $y = f(2.0, x\_even)$. (1pt)
4. Now, replace $y = f(2.0, 20)$ by 2500. Note, counting to the index corresponding to $f(2.0, 20)$ by hand is not allowed! (1pt)

We now have a distorted dataset.

5. Use curve fitting to find the best value of a and the uncertainty in a. Present the two values as an experimental physicist would do. (3pt)
6. Make a test array from 0 to 100 with an interval of 0.1 and use this test array to graph the function fit into the data. Show the function fit as a dotted red line. (3pt)

```
### YOUR CODE
```

Task 2

A physical quantity A has been measured in order to calculate a different physical quantity Z. The value of A is 1.075 ± 0.003.

Calculate the physical quantity Z and its uncertainty using the calculus approach for:

1. $Z = 2.5 \cdot \sin(2\pi A)$ (3pt)

and the functional approach for:

2. $Z = \sqrt{3} \cdot A^4$ (3pt)

(Disregard the unit(s))

```
### YOUR CODE
```

Task 3

Eliza is taking repeated independent measurements using a digital multimeter. However, she notes that every time the same value is showing up. Does this mean that the quantity she is measuring has no uncertainty? Explain. (1pt)

```
### YOUR ANSWER
```

Task 4

In your final project you received a dataset stored in 'data.csv'. You are asked to (1) investigate the pattern in the data, (2) define a function that probably fits that pattern, (3) use curve fitting to present the values of the parameters and their uncertainties, (4) plot the fit on the data and study the residuals: investigate whether this these are random noise that can be described using a Gaussian distribution and if so, what the standard deviation is of that noise.

First load the 'data.csv' file. Then carry out this data-analysis. (9pt)

```
### YOUR CODE
```

0.3 Writing your report

0.3.1 Goal of writing a report

Report writing is important – for you!

In the Introductory Laboratory Course, you practise not only conducting experiments but also writing reports about them. When doing this, you have to arrange your data logically, summarise what you have done and develop a feeling for what is really important and what is not. Your written lab report is the natural culmination of the experimental process, enabling anyone who is interested to learn about your findings.

But writing a good lab report is much more than that. In many ways, this written account of a single experiment is very much like other scientific texts: journal articles, conference papers and so on. In terms of structure, they are all quite similar. So practising with lab reports also starts preparing you to compile those other documents. In your future scientific career, there is a good chance that you will spend a fair amount of time producing – and assessing – such papers, so you need to learn to do that well and efficiently.

That may seem a long time off, but remember too that lab reports have a very personal component. They describe what an individual has actually been doing. For the writer, producing a good report should be matter of pride. The reader of a poor report will not think much of the person responsible, whereas a good one demonstrates that the author has understood their subject and is able to convey it clearly and comprehensibly. This makes readers more likely to take that person seriously. So written reports give you the chance to make a good impression. When applying for a job, for instance, a good account of your graduation project makes for an excellent "calling card".

This course is about writing reports of your experiments during the Introductory Laboratory Course. These experiments will be limited in scope, in that you are not making any new discoveries, but the basic principles that you learn here apply equally to simple lab reports and ground-breaking articles revealing important scientific breakthroughs.

0.3.2 Structure and content of a report

We provide a LaTeX template. If you adhere to this template, all necessary content, described below, is covered.

Title + title page:

Complete and concise. Contains no pagination errors. Provides an overall picture of the report. Front, core and back matter are numbered correctly.

Summary/abstract:

Correctly and concisely summarises the report in the order topic, problem, method, results, discussion, conclusion in not more than 250 words. Can be read independently by someone with little time. No references.

Introduction:

Provides background information. Includes a definition of the problem, a key question and the reason for the experiment. Explains the importance of the outcome. Briefly introduces the experimental principles and method used. Briefly describes the structure of the report. Can include a boxed text.

Theoretical framework:

A brief theoretical framework is presented which is necessary to understand the experiment. It should be clear how the research question can be answered using this theoretical framework. All symbols are explained.

Experimental method:

Describes the experimental method and setup used, preferably with a clear sketch. Only a limited number of essential formulas are included, with correct references. A working formula derived from these can clarify the experiment and introduce a brief error analysis; any longer error analyses are provided in an appendix.

Results (can be combined with Discussion; tables and graphs are assessed separately):

Contains the results of the measurements explained under "Experimental method", in the form of tables and graphs. These are introduced in the text. The reader is informed of the outcome. In a short report, a discussion of the results can follow immediately (see "Discussion"). The number of significant figures is correct for the error found. Good distinction between main text and any appendices; the main text can be read without referring to the appendices. Appendices are numbered and the main text contains references to them.

Discussion (can be combined with Results):

The results actually obtained are compared correctly with those expected or obtained in previous experiments. Discusses or suggests the likely causes of any anomalies. Describes the implications of the results in a way that shows that the student understands the topic.

Conclusion (and recommendations, if relevant):

Answers the key questions. Conclusions follow naturally from the contents of the previous sections and are presented concisely but fully. Any recommendations follow on logically from the conclusions. Can be read as a text in its own right. Contains no references.

Literature/References:

Entries comply with the format requirements. Contains only titles referred to in the text, and contains all of them.

Figures (graphs):

Figures are numbered, are referred to in the text and do not have a title. Comprehensible in black and white. Include an informative caption (underneath). Axes are labelled with value symbols, units and possibly some text. Symbols and lines are explained in the caption, and preferable not in a legend. Error flags are either included or their absence is discussed. Scale marks ("ticks") are legible. Only multiples of 1, 2 or 5 are used, with axes preferably including an "0" point. Figures are not "print screen" cut and paste.

Tables:

Tables are numbered, are referred to in the text and have an informative caption (above). Rows and columns are clearly "headed" with symbols and units. None contain constant values. (The number of significant figures is assessed under "Results".) Tables with no more than 5 measurements should be within the text. Tables with more than 5 measurements should be given in the Appendix.

0.3.3 Structure of writing

Writing a report is difficult. Many students struggle with it. We can not teach you how to write a report within a three-week course. However, we can provide some tips.

Try to reverse engineer writing the report. That is to say, start with the conclusion, the claim you want to make and are able to defend. It can be very short: *In this free-fall experiment, the gravitational acceleration is found to be 9.813 ± 0.002 m/s^2, what is in strong agreement with the literature value of 9.812 ± 0.002 m/s^2.* If that is the claim you can and want to defend in the study, you can figure out what graphs and calculations (results section) is to be provided to have peers agree with you. Write, in bullet point format, what graphs will be included, what you will say about the graphs, what additional calculations are provided etc.

Then it is probably time to write, again in bullet point format, the introduction. Use the V-shape structure, i.e., introduce the topic of interest in a general way, making clear how it is important to many. As the introduction proceeds, it should become more and more specific and end to the specific issue addressed in the study.

Since you have analysed your data, you will be able to provide the theoretic content that is required to understand the experiment. This should not be a big problem to write down.

In the method section you write down what you did, but more important, why you made specific choices (range / interval / measurement instruments). In other words, substantiate every choice and elaborate why other choices result in less reliable/valid data or were not an option. This might be a limitation in financial resources, time constraints, limitations to the given equipment, etc.

0.3.4 Resources

The best resource is probably the TA. However, the available time is limited. Provide an discuss an outline as quickly as possible. Do not produce many pages of text if there is no agreement on what should be in the report.

There are other resources as well, we provide some:

Reporting Results:

Reporting results van Van Aken & Hosford provide a concise book with unwritten rules of presenting results. The book also provides a clear overview of measurement and uncertainty.

A guide:

A guide to effective writing offers guidelines for various texts. Although the focus is on Industrial Design, the essence of writing a clear and concise report is the same.

Nature:

Nature Physics offers, as many other journals, a list of requirements addressing text, length, figures, tables, lay-out etc. You can click on *submit article* and see the author's guidelines.

Physics review:

Physics review provides as well a list of requirements and guidelines.

0.3.5 Example

An example of a report is given in the Appendix of this manual.

0.3.6 Checks

There are two scheduled day-parts were you ought to work in the first report. The first day-part is voluntary, you can agree with your partner to work at home. However, the second day-part has a mandatory aspect. You need to show your report or draft version at least once to a TA. (S)He will help you by providing feedback. This prevents you from making often made mistakes or mistakes that could be easily avoided, such as forgetting to write a clear, concise subscript for each figure.

0.4 First experiment

0.4.1 Determining g

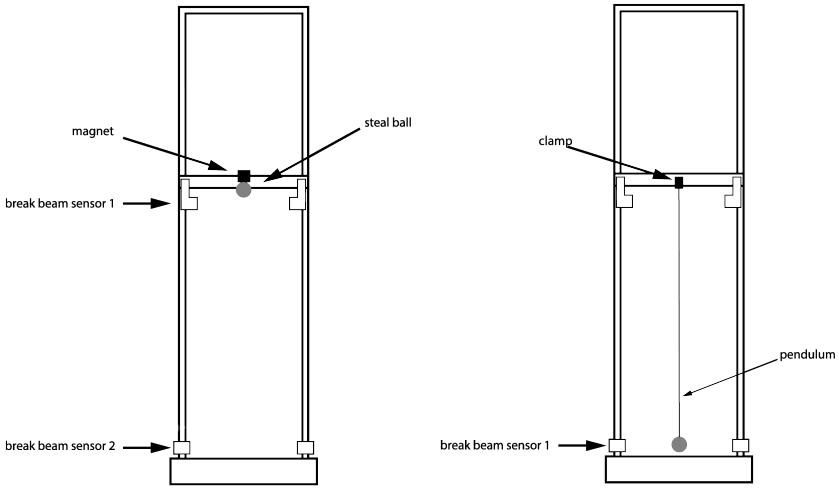


Figure 8: The same setup can be used for both experiments

Experimental goal

In this experiment you will determine the gravitational acceleration on earth, g. Find the fourth significant figure and determine g within 0.1% accuracy of the value established by scientists. The determined values should be compared to the literature value.

Learning goal

The physics involved is pretty easy, but devising a proper method, analyzing the data, calculating the uncertainties, presenting you data in a convincing matter, writing a proper paper, probably is not. With this experiment you will learn how to:

1. Devise a proper method.
2. Process, analyse and present your data.
3. Write a concise, scientific paper on your findings.

Introduction

The gravitational force causes you to fall (accelerate) back to earth when you jump. The value of the gravitational acceleration g is well known. Near the surface in the Netherlands it has a value of 9.812 m/s^2. How can we determine this value with simple methods but high precision ourselves?

Theory

The pendulum According to the theory of a simple physical pendulum which consists of a point mass hanging on a mass-less wire, see figure 1, the period of this pendulum is described by:

$$T = 2\pi\sqrt{l/g} \tag{58}$$

When the period and the length are determined, the gravitational acceleration can be calculated.

Free fall The movement of a falling object in a vacuum can be described by:

$$y(t) = y_0 + v_0 t + \frac{gt^2}{2} \tag{59}$$

If the fall time for different heights is measured, this expression can be used to determine the gravitational acceleration.

Agreement analysis Experimental values can be compared with the known empirical value and with each other. Two values a and b are not in good agreement when:

$$|v| = |a - b| > 2\sqrt{u_a^2 + u_b^2} = 2u_v \tag{60}$$

Method

Preparation phase

1. Choose either the Pendulum or the Freefall experiment
2. Install the free Arduino software on your laptop: www.arduino.cc and download the script for the experiment from Brightspace.
3. Do a trial run for the experiment using the stopwatch on your mobile phone.
4. Carry out the experiment five times for a fixed length or height.
5. Determine the average measured time, the standard deviation and the measurement uncertainty.
6. Is the measurement uncertainty in time small enough to determine g within 0.5% accuracy?
7. Use the break-beam sensor to determine the time for five repeated readings. Compare this result with the values determined before.
8. The provided theory is only valid for specific conditions. For the required accuracy in this experiment, the theory should be extended. Identify what is missing and needs to be added.
9. Develop a plan for the experiment to determine g within the required accuracy. Substantiate each choice you make, identify and validate assumptions. Carry out extra trials if necessary.
10. Have your plan checked by the TA.
11. Prepare a Python script to carry out the data analysis.

Experimental phase Carry out the experiment using the approved plan.

Evaluation phase

1. Process your data according your plan.
2. Carry out the agreement analysis.
3. Present your findings in a 'scientific' paper. You will write most of this on two scheduled afternoons. You are allowed to write in pairs and hand in a single paper.

0.4.2 Labjournal

> **Note**
> Template for labjournaal. See here on using markdown.

General

Name:
Title of the experiment:
Starting date:
Expected enddate:
Partner:
Goal of the experiment:
Research question:
Expectations or Hypothesis:
Desired accuracy:

```
#import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
import math
from scipy.optimize import curve_fit
```

Preparation

Assignments:
Method:
Theory:
Independent variable:
Dependent variable:
Controlled variablen:
Measurement instruments & Settings:
Procedure:
Setup(drawing or picture):
Notes:
About accuracy:

Execution

```
# Measurements: Explain the names of variables provide only raw data in np.arrays!
```

Observations:
Notes:

Processing

Description of processing of raw data into scientific evidence:

```
#Data processing and analysis:
```

```
#Data processing and analysis:
```

```
#Data processing and analysis:
```

Describing the pattern in the processed data:

```
#Calculations of e.a. measurement uncertainties, and providing final answers.
```

Notes:

0.5 Second experiment

0.5.1 Boltzmann

Goal

This experiment consists of two parts. The goal of these experiments are to get acquainted with electronics and electronic instruments. These instruments are often used in more advanced physics labs.

Part 1: Boltzmann constant

The goal of the experiment is to gain experience with Digital Multi Meters (DMMs) with an experiment in which you determine the Boltzmann constant. We use a frequently used setup, the "voltage divider".

This video is a great introduction to this experiment.

Background

Measurements using DMMs DMMs are versatile measuring instruments. These instruments are used to measure voltage, current, resistance, frequencies, periods, capacities, induction, temperature, features of diodes etc. Advanced DMMs have the option to be controlled using a computer.

The difference in price is often determined by its functionality, resolution and sensitivity. The resolution of a digital instrument is the ratio between the smallest counts and the largest counts that can be displayed. This is determined by the number of counts that can be displayed. A simple DMM often has $3\frac{1}{2}$ digits, which means that the DMM can display three whole digits (0-9) and an additional (first) digit which has the value 0 or $\pm$ 1. As such, this DMM can display the number 0-1999, a total of 2000 counts. The resolution of this DMM is thus $\frac{1}{2000}$ or 0.05%.

The sensitivity is the smallest change that still can be noted. A sensitivity of 1μV implies that signals smaller than 1μV can not be detected. The sensitivity does depend on the amplitude of the signal we want to measure. Suppose we want to measure a 15V signal using a $3\frac{1}{2}$ DMM, the best range is 20V. This means a sensitivity of 10mV. Using a $5\frac{1}{2}$ DMM, we can get a sensitivity of 0.01mV. The ultimate sensitivity of a DMM depends on the resolution and the smallest range. Example: the sensitivity of a $6\frac{1}{2}$ digit DMM with the smallest range of 200mV is 0.1μV.

However, measuring 0.2μV does not mean this is the exact value you measure. This does depend on the accuracy, and the accuracy is not determined by the last digit of the DMM alone. The accuracy is often defined as '..% reading + ..%range + .. counts".

Example
Suppose we use a $3\frac{1}{2}$ digit DMM and read a value of 1.234V. For the Dynatek D9100 with a 2V range is stated that the inaccuracy is given by: $\pm$0.5

Ideal instruments A DMM can be used as a voltmeter or ammeter. In the ideal case a voltmeter has an infinite internal resistance and an ammeter no resistance. In many cases these assumptions are valid. However, in cases where the circuit resistance is approximately 0 or ∞, we can not neglect the features of the meters. Using these meters inevitably changes the features of the circuits itself, see Figure 9.

In this experiment we will determine the (U, I)-characteristics of a silicon diode. However, during the experiment the current through the diode easily changes with 6 decades (10^{-2}A $\rightarrow$ 10^{-8}A). Therefore we measure the current indirectly, using a voltage divider.

Voltage divider In this experiment we use a voltage divider circuit to make a (U, I)-characteristic of a diode, see Figure 10. A simple voltage divider is a circuit with two resistors in series "sharing" the voltage of the source. If you do not remember the rules that apply in series circuits, look these up yourselves. The formulas are given below.

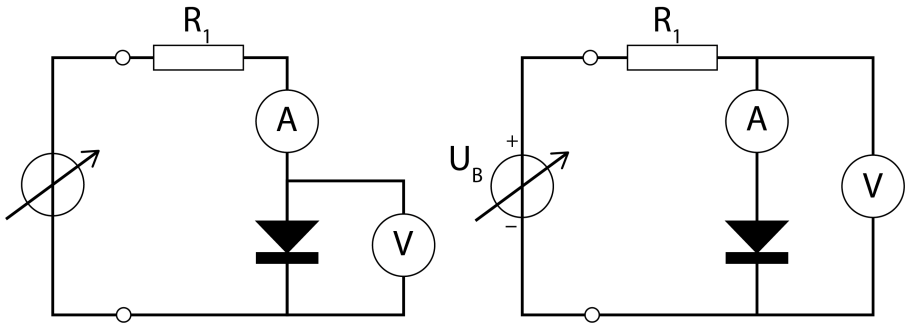


Figure 9: Two ways of measuring current and voltage through a diode

Note

`Series circuit`

$$I_t = I_1 = I_2 = \frac{U_t}{R_t} = \frac{U_1 + U_2}{R_1 + R_2} \tag{61}$$

$$U_1 = \frac{R_1}{R_1 + R_2} U_t \tag{62}$$

`Parallel circuit`

$$I_t = I_1 + I_2 = \frac{U_1}{R_1} + \frac{U_2}{R_2} \tag{63}$$

`Power dissipation`

$$P_t = U_t I_t = U_1 I_1 + U_2 I_2 \tag{64}$$

An advantages of this circuit is that we do not measure current directly. As the current changes a few decades, we approximate the original circuit (without instruments).

Semiconductor diode A diode is a semiconductor. In a semiconductor the energy of the electrons are distributed according the Boltzmann distribution. The average energy of an electron is $k_B T$, where k_B is the Boltzmann constant and T is the absolute temperature. The current flowing through the circuit depends on the height of the energy barrier relative to $k_B T$. On its turn, the height of the barrier depends on the the applied voltage. The higher the applied voltage the lower the energy barrier.

The Boltzmann distribution of the energy of the electrons, and the presence of a barrier is given by a theoretical formula that relates the applied voltage over and the current through a diode. This is an exponential function given by:

$$I_D(U_D) = I_0 \left(e^{-\frac{qU}{nk_B T}} - 1 \right) \tag{65}$$

where q is the charge of an electron, $-1.602 \cdot 10^{-19}$C, n an ideality factor which is 2 for Si, I_0 the reverse current when V_D is strongly negative. For more (background) information, refer to Wolfson and others [2007], chapter 27-28.

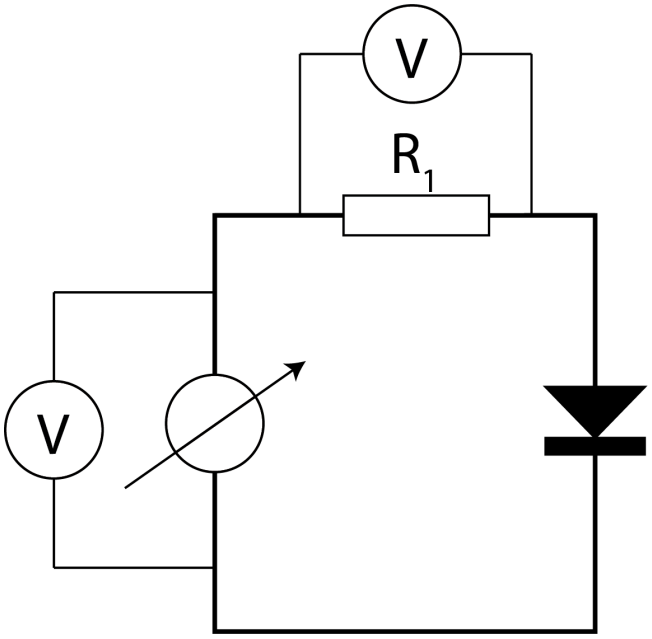


Figure 10: The experimental setup to determine the (U, I)-characteristic of the diode

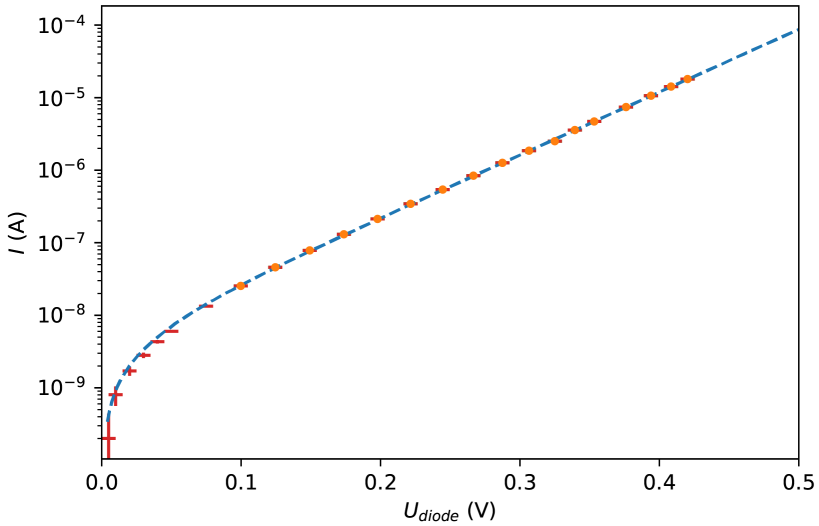


Figure 11: The (U, I)-characteristic of the diode as determined by two first year physics students (2019). The slope is used to determine the Boltzmann constant.

In order to enable us to determine the (U, I)-characteristics of a silicon diode (over a range which is as large as possible), we should consider a number of topics before we actually conduct the experiment. We note that we aim to determine Boltzmann's constant using a logarithmic plot. To do this accurately, it is key to distribute measurements evenly along the range of the variables (diode current I and diode voltage U). In practice, it is convenient to start at (or near) the maximum value for the current and subsequently decrease the current with a constant factor for each measurement. However, it might be simpler to measure, plot and choose the next value based upon your findings.

Method

Preparation phase Prepare you python script in which you collect and process your data. Carry out the following steps:

1. Load the various libraries. (single cell)
2. Make numpy arrays for your raw measurements. (single cell)
3. Make numpy arrays for the processed data and their uncertainties. Use the manual of the DMM to calculate the uncertainties. (single cell)
4. Make a single cell in which you plot the (U, I)-characteristics of the diode.
5. Setup a single cell for curve fitting.
6. Setup a single cell for calculating the Boltzmann constant and its uncertainty.

Experimental phase Determine the exact resistance of the resistors you will use, calculate the uncertainty. Build the circuit shown in Figure 10. Start your measurements with a 6.0V source voltage. Slowly decrease the source voltage using the plot. Make sure you have at least 15 measurements in the range $0.1\text{V} - 0.6\text{V}$ for U_{diode}. Take care of an even spread interval.

Evaluation phase Run your script and determine the Boltzmann constant accordingly. You should be at least within 5% accuracy.

Part 2 As function of temperature
https://contemporary-physicslab.github.io/NP-new-style/main/Deel5/BoltzmannT.html

0.5.2 RC-circuits

The goal of the experiment is to gain experience with oscilloscopes with an experiment in which you do measurements on a RC-circuit. You first determine the value of a capacitor and subsequently build and measure a low pass filter.

Background

Capacitors and RC-circuits A capacitor consists of two plates that are separated a small distance from each other. You can store electrical charge on the plates. Its capacitance can be calculated by:

$$C = \frac{Q}{V} \tag{66}$$

Where C is the capacitance, Q the amount of charge stored on the capacitor and V the voltage over the capacitor.

To charge a capacitor, you simply apply a voltage over the capacitor. To prevent damage while (dis)charging, a resistor should be used, see Figure 12. While the capacitor is charged, more and more electrons are collected on a plate producing a counter voltage. This thus means that further charging the capacitor becomes more difficult and the voltage over the capacitor increases while the voltage of the resistor decreases:

$$V_{source} = V_R + V_C \tag{67}$$

Knowing that the current is defined by:

$$I = \frac{\Delta Q}{\Delta t} \tag{68}$$

and applying Ohm's law results in the following differential equation:

$$\frac{\partial V}{\partial t} = R\frac{\partial I}{\partial t} + \frac{1}{C}I \tag{69}$$

When a constant voltage is suddenly applied ($V_{source} = 0 \rightarrow V$), the current through the RC-circuit is described by:

$$I(t) = \frac{V}{R}e^{-t/RC} \tag{70}$$

The product of $R \cdot C$ is often called the RC-time, τ_0. The RC-time is the time required to charge the capacitor from 0V to 3.2V ($63.2\% = 1 - e^{-1}$) of the voltage applied by the source. Equation (70) becomes then:

$$I(t) = \frac{V}{R}e^{-t/\tau_0} \tag{71}$$

When a alternating voltage is applied, the capacitor will charge and discharge. When the applied voltage changes from sign, the capacitor quickly discharges at the start. However, the amount of current flowing through the circuit quickly reduces (this idea is applied in flash lights of cameras).

If the frequency of the alternating voltage is very high, the capacitor can be regarded as a simple wire without resistance. The current flowing through the circuit is determined by the resistor. The voltage over the resistor is roughly the same as the voltage applied by the source. However, if the frequency is low, the current is dominated by the characteristics of the capacitor.

According to the theory, the voltage gain over the capacitor can be calculated by:

$$\left|\frac{V_C}{V_{in}}\right| = \frac{1}{\sqrt{1 + (\omega RC)^2}} \tag{72}$$

There will be a phase shift between the applied voltage and the current which is dependent on the applied frequency. According to theory, the phase shift ϕ is calculated by:

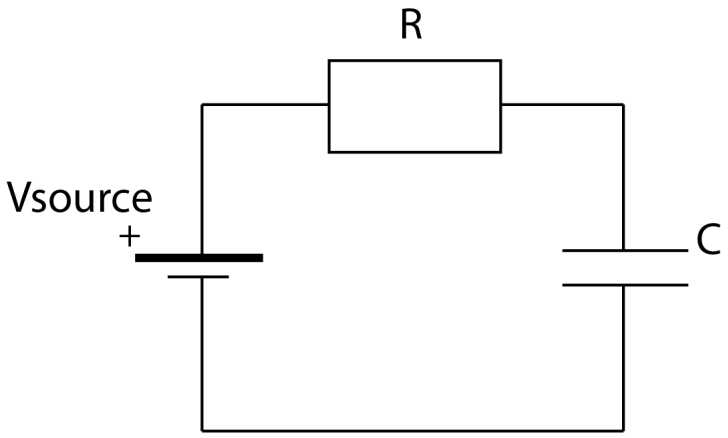


Figure 12: The RC-circuit to charge a capacitor

$$\phi = tan^{-1}(\omega CR) \tag{73}$$

where $\omega = 2\pi f$.

The transit in behaviour becomes notably at the cutoff frequency:

$$f_{\text{cutoff}} = \frac{1}{2\pi RC} \tag{74}$$

In this experiment we determine the characteristics of the RC-circuit as described above using a oscilloscope.

Measurements using Oscilloscopes You can find oscilloscopes in most physics lab rooms. It is an all round measure- and test-instrument. The oscilloscope displays the measured quantity as function of time or a second quantity on a screen. The input signal is always an electrical voltage. For measurements of temperature, light intensity etc. you need a sensor which turns the measured quantity in an electrical output.

As most signals in nature are analogue, these are converted to a digital signal using an Analogue-Digital-Converter (ADC). The resolution of the digital signal depends on the number of bits (N) in the voltage range U_{mm}. This range is divided in 2^N equal intervals.

A usual ADC in a digital oscilloscope has $N = 8$ bits (the SoundBlaster ranges from $N = 12$ to 14 bits usually, the DAC in your MP3-player can have up to $N = 18$ bits). The resolution determines the level up to which details can be distinguished. The resolution is limited by the criteria of fast ADC-conversion: for measuring signals of 10MHz, the conversion needs to last no more than 10ns (in principle). In practice, combining speed and resolution is difficult. The latter means such equipment is quite expensive.

During this experiment, our digital oscilloscopes have $N = 8$ bit resolution, such that the screen is divided in $2^8 = 256$ intervals vertically.

The Agilent DSO3062A (used in this experiment) has a bandwidth of 60MHz, meaning (in theory) signals with a period of 17ns can be measured. When measuring a single channel, the time-dependent signal is shown. In this setting, the vertical axis displays the voltage, and the time is shown on the horizontal axis. A 2-channel oscilloscope allows two signals to be displayed on a shared horizontal axis, with each having its own vertical axis Y1 and Y2. An example is shown in Figure 13. Such feature allows to compare the time-dependent behaviour of the signals, e.g. the relative phase-shift.

To display a signal, it is first written to memory, after which it is shown on a display, which is often a Liquid Crystal Display (LCD). Several signals can be saved and subsequently compared and processed: subtract, multiply, integrate, average, etc. The data can also be saved to a PC for further processing.

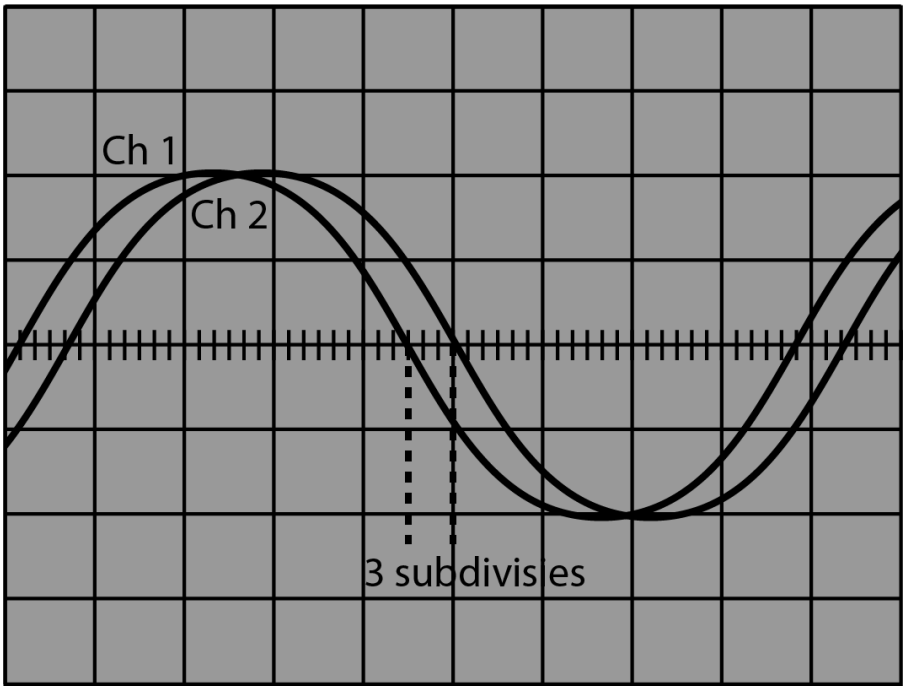


Figure 13: A 2-channel oscilloscope allows to compare two signal, e.g. determine the phase difference.

Basic settings To study a signal effectively, two settings are important: the vertical sensitivity and the temporal axis. The former refers to the set vertical range in Volts per division (V/div), The temporal resolution is measured in s/div, where a fewer seconds per division means that higher frequency signals can be seen.

A scope displays a signal as shown in Figure **??**. Its settings are configured as follows:

- Amplification: 2V/div
- Temporal resolution: 5ms/div
- Trigger slope: +
- Coupling: DC

Determine the period, top-to-top amplitude and the pulse length.

Triggering To read a signal effectively, it helps if the scope is not continuously updating the screen real-time. Triggering is a tool designed for this purpose. When a trigger level is set, the oscilloscope only starts to display a signal when it reaches that setpoint. The triggerlevel can be approached from the top or bottom, and must be configured by the slope+ or slope- setting.

Processing An advantage of a digital scope is that the signal can be processed right away. Various options can be selected with the `MEASURE` button. The most important are `TIME`, which has the option `FREQUENCY` and `VOLTAGE`, which has the option `UTT` which refers to Top-Top Voltage.

Measuring frequencies is easy to do in the time domain. The frequency can be calculated by determining the time difference between two identical (both value and slope) points on the curve. This difference is the period T of the signal, its inverse $f = \frac{1}{T}$ is the frequency.

Determination of the phase difference is similar. We need a two-channel scope. First, determine the period of the signal. Do the same for the relative shift between the signals. The ratio of the two yield the phase shift, in units of period. Multiplying by 360 results in the phase shift in degrees.

Method

Determining the RC-time

1. Pick any combination of R and C so that the cut-off frequency is somewhere between 100 and 1000Hz.

2. Build the RC-circuit, connect the oscilloscope in such a way that the GND of both the oscilloscope and frequency generator are connected.

3. Apply a square wave with a frequency of $\approx 1/(10\tau)$. Connect the main out with **Hi**. Make sure that the offset is 0V.

4. Trigger the signal. Use mode/coupling. Choose a positive slope. What happens if you alter the trigger level?

5. Draw the signal that is displayed.

We can determine the RC-time in various ways. We start by doing this by hand to get more familiar with the oscilloscope.

1. Adjust the time division so you only see one wave.

2. Hit `SINGLE`. This provides a single measurement (instead of a continuous signal).

3. Calculate the RC-time using its definition (63.2% of the maximum applied voltage). It only has to be a rough estimate.

4. A more accurate value is obtained when we first determine the minimum and maximum voltage. To do so, press: `MEASURE` / `VOLTAGE` / V_{pp}; V_{max}; V_{min}. In the lower left corner you will find the V_{pp} value (peak-peak), V_{max} (maximum value), V_{min} (minimum value).

5. Use these values to calculate $U_C = 63,2\%\Delta U$.

6. Press `CURSOR` and subsequently `TRACK`. The tracking point can be moved. You can also use a second Tracker. Set both trackers in the right position so you can determine the RC-time.

7. There is another way to measure RC-time. You can use the rise-time. This is the time required for the signal to go from 10% to 90% of the maximum voltage. The RC-time is roughly $\tau = t_r/2.2$

8. Press `MEASURE` / `TIME` / `RISE TIME`. Use the oscilloscope to determine the rise time.

9. Compare the three values you have found for the Rise time. Which one is most reliable? Do these values deviate from the calculated RC-time?

Characteristics of a low-pass filter Now you know the cut-off frequency, you can determine the characteristics of a low-pass filter. Measure from two decades below cut-off frequency up to two decades above cut-off frequency the Vpp and phase shift. **Be sure to use a SINE wave now!** Per decade, obtain at least three measurements. In the decade of the cut-off frequency ($100 - 1000$Hz) obtain at least five measurements. Note: you thus have at least 17 measurements!

Plot your measurements using a logscale for the frequency. You can use the commando `plt.xscale('log')`. Use this graph to determine, again, the cut-off frequency. Compare it with the calculated, theoretical value.

Low-pass filter applied A Low-Pass Filter can be used to filter noise superposed over a signal. To experience the effectiveness, we will send a signal with noise to the RC circuit used in the previous exercise. The noisy signal can be send with a desktop computer to the signal generator.

- Download and open the file containing the wave with noise: *sineWaveWithNoise.xls*. This file contains macros to connect with the signal generator, therefore you must enable content in Excel if you're asked to in a banner.
- The top of the Excel file has a few marked cells in which values can be altered, such as the amplitude (V_{PP}) and frequency f. If you wish, you can also edit the noise frequency. The graph below shows both the main signal (which we want to measure) and the noise.
- Press the `UTILITY` button, then `I/O` and finally `SHOW USB ID`. Copy this address into cell D3 in your Excel sheet.

- Test if everything works with a voltage of $2V_{PP}$ and a frequency of 1Hz. Press `SEND TO FUNCTION GENERATOR` and check on the oscilloscope if the signal from the graph is generated.

The signal send to the function generator is constructed by adding a base signal with the given frequency to a sine wave with the noise frequency (initially set to 100 times the base frequency), resulting in the blue curve shown in the Excel graph. This curve should match with channel 1 on your oscilloscope.

1. For which frequencies do you expect the filter to work optimal, i.e. the highest signal-to-noise ratio (S/N)? What are your expectations for lower and higher frequencies?

2. Set the frequency in Excel to the one found in the previous question, and study channel 2 of the oscilloscope. Add a picture (either a photo or a drawing) to your lab journal.

3. Take a look what happens for other frequencies up to 3 decades difference. Note for each decade (6 in total) whether the working of the filter improves, deteriorates or stays unchanged. Does this meet your expectations?

4. If the results in the previous question are not clearly showing a difference in the filter quality (S/N), the chosen cut-off frequency was probably poorly chosen. Go again through your calculations to find out if errors were made.

0.5.3 Determination of the half-life of Potassium-40

Goal

In the first part of this experiment you will determine the half-life of the naturally occurring radioactive atomic species ^{40}K, an isotope of the element potassium. You will do this by establishing of the activity and the number of radioactive nuclei of ^{40}K samples. In the second part you will establish a histogram of the number of measured decay processes and you will fit a Poisson probability function to the histogram. The fit will also yield a value of the half-life of ^{40}K, which should be compared with the value resulting from the first part. The question is which method yields the best result and whether there is a trade-off.

Introduction

The natural background of ionizing radiation Although not generally known, everyone is exposed to ionizing radiation of natural origin. This radiation originates from both cosmic and terrestrial sources. Terrestrial radiation arises from several kinds of radioactive substances. Some of these substances are present because of human activity (nuclear tests, nuclear power plants). A substantial part of these substances, however, occurs naturally and comprises radioactive nuclides with a long lifetime. This category includes uranium isotopes and the radioactive decay products of these isotopes, including the noble gas radon. The natural composition of potassium also includes and isotope with a very long lifetime: ^{40}K.

This potassium isotope dominantly decays to ^{40}Ca by emission of an electron, so-called β^--decay. Further processes are β^+-decay (emission of a positron, i.e. a positively charged electron) or capture of an electron from an electron shell close to the nucleus (electron capture, EC). In these cases the decay product is ^{40}Ar. A decay scheme of ^{40}K is given in Figure 1 [1].

The lifetime of ^{40}K must be of the same order of magnitude as that of the lifetime of the Earth. Otherwise, the ^{40}K formed in creation of the Earth would have disappeared already by radioactive decay.

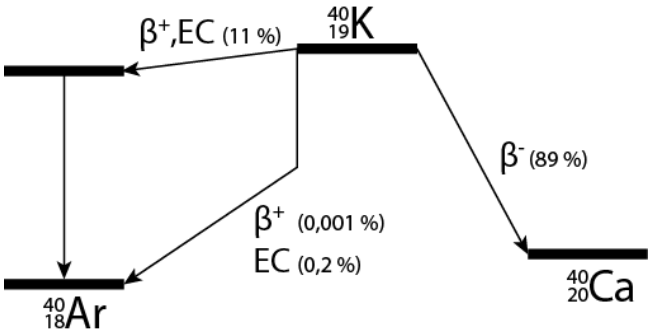


Figure 14: Decay scheme of ^{40}K. Vertical lines correspond to emission of gamma radiation. EC: electron capture.

Equivalent dose from background radiation In expressing the potential risks due to radiation the equivalent dose is often used. This is the radiation load due to ionizing radiation. The equivalent dose is the absorbed dose (absorbed energy per kg material) multiplied by a factor representing the biological effectiveness of the radiation. The SI unit of equivalent dose is the Sievert (Sv).

In the Netherlands, the equivalent dose resulting from natural sources is roughly 2mSv per year of which roughly 10% is due to the decay of ^{40}K. In comparison, having an X-ray in a hospital also gives an equivalent dose of several tenths of mSv. A skiing holiday or a transatlantic flight, since one is higher up in the atmosphere, give equivalent doses of one hundredth to one tenth of a mSv.

Theory of the radioactive decay

Average behaviour Suppose we have a number of $N(t)$ nuclei of a radioactive nuclide at time t. It is reasonable to assume that the number of nuclei that decays (ΔN) on average in a time span Δt is both proportional to $N(t)$ and Δt:

$$\Delta N = -\lambda N(t)\Delta t \tag{75}$$

Here, the proportionality factor λ is called the decay constant. Equation (75) is in fact a differential equation for $N(t)$. Its solution,the law of *radioactive decay*, is an exponentially decaying function in time:

$$N(t) = N(0)e^{-\lambda t} \tag{76}$$

The inverse of λ, i.e. $\tau = \frac{1}{\lambda}$, is called the *average lifetime* of the nuclide. This is the time it takes for the number of parent nuclei to decrease to 1/e of its initial value. Additionally, the half-life $t_{1/2}$ is often used. This is the time it takes for half the original nuclei to decay. By substituting in Eq. (76) the value $\frac{1}{2}N(0)$ for $N(t)$, it easily follows that

$$t_{1/2} = \frac{\ln(2)}{\lambda} = \tau \ln(2) \tag{77}$$

The decay constant (and hence the average lifetime and the half-life) is a characteristic of a radioactive nuclide, for example ^{40}K. The activity of a radioactive source is the number of nuclei that decay per unit of time:

$$A(t) = \left|\frac{dN(t)}{dt}\right| = \lambda N(0)e^{-\lambda t} \tag{78}$$

The activity also decays exponentially in time, again with constant $\tau = \frac{1}{\lambda}$. The activity has the dimension t^{-1} and thus can be expressed in s^{-1}. However, there is a distinct SI-unit for activity of a radioactive source, the becquerel (Bq). In this context, we note that a *count rate should be expressed in the unit in* s^{-1} or a similar unit (*e.g.* minutes^{-1}). A count rate only *becomes an activity with unit* Bq *after correction for a number of effects*, such as background radiation and the efficiency of the setup.

Statistics of the decay Equation (76) describes the average behaviour of radioactive decay. On a short time scale, however, the decay has a random character. The number of decay processes taking place in a time interval Δt varies in time and shows a statistical behaviour around the mean μ. The statistical probability that k nuclei of the N(t) nuclei present in the sample decay in the time interval Δt is given by the Poisson probability distribution $P_\mu(k)$:

$$P_\mu(k) = \frac{\mu^k}{k!}e^{-\mu} \tag{79}$$

The parameter μ is the mean number of decay processes in a time interval Δt. The value of μ can be calculated from the values of k measured for a very large number of in-dependent time intervals Δt (of equal size). The function $k!$ (pronounce: k faculty) is the factorial function given by $k! = 1 \times 2 \times 3 \times ... \times k$ (with 0!=1).

Although Eq. (79) gives a relatively simple function, it is difficult to remember how μ and k are placed in the expression (Advice: keep on paying attention to this!). From Eq. (79) it follows that the Poisson probability distribution is determined by only a single parameter μ, the mean of the distribution. Figure 2 shows the plot of a Poisson distribution for $\mu = 3$. It can be noticed immediately that the Poisson distribution is not symmetric, a property most easily seen for $\mu \leq 5$.

An expression for μ based on the average behaviour of the radioactive decay can be obtained by multiplying the average decrease of the number of radioactive nuclei per unit of time [i.e. the activity, Eq. (78)] with the time interval Δt:

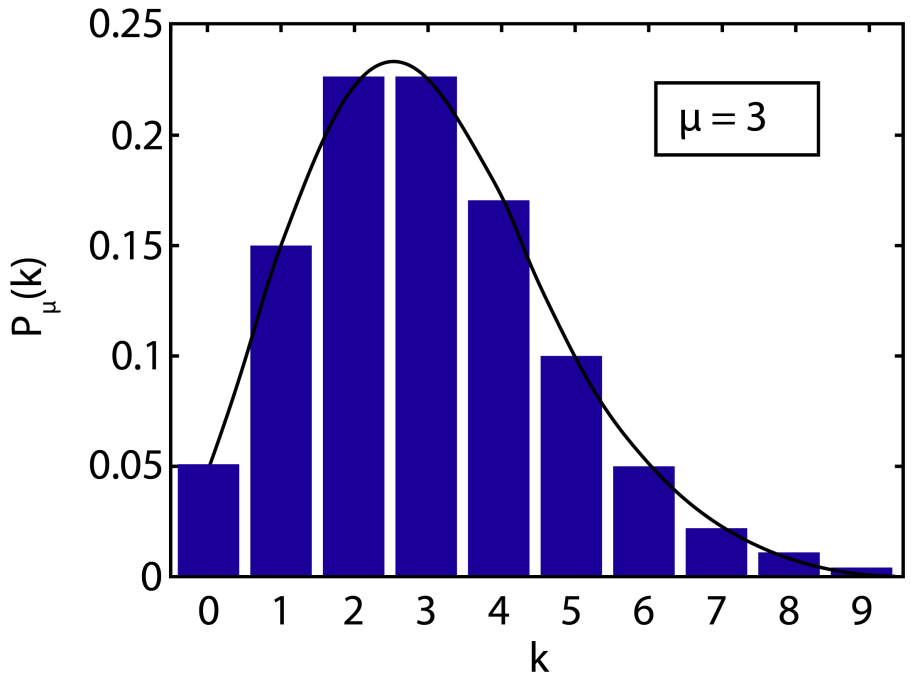


Figure 15: Poisson probability distribution for $\mu = 3$, plotted together with the histogram to which the distribution has been fitted. The probability distribution is asymmetric. The curve has been plotted as a continuous curve, but in reality the distribution only holds for integer values of k.

$$\mu = \left|\frac{dN(t)}{dt}\right| \Delta t = \lambda N(0) e^{-\lambda t} \Delta t = \lambda N(t) \Delta t = \frac{\ln 2}{t_{1/2}} N \Delta t \tag{80}$$

Omitting the time dependence of N in the last part of this equation indicates that the number of radioactive nuclei only decreases negligibly in the time interval Δt. (N.b.: usually, the notation for a time interval is Δt; generally, t denotes a point in time.)

Typically, in measuring radioactive decay, a counting experiment is performed. In such an experiment the number of decay processes in the time interval Δt is counted by detection of the decay products. By repeating the counting many times and making a histogram of the count totals in the time intervals Δt, a distribution is obtained. This distribution will confirm the Poisson distribution. Apart from the mean, the standard deviation σ is an important characteristic of the Poisson distribution. In particular, σ determines the width of the distribution. From the definition of the standard deviation and Eq. (79), one can derive:

$$\sigma = \sqrt{\mu} \tag{81}$$

Thus, the parameter μ is not only the mean of the distribution, but also determines its standard deviation. Equation (81) is often generalised to the square root rule for an individual counting experiment (not limited to radioactive decay; for example the number of babies born per month in a hospital): the uncertainty in a counting experiment with a count total of k is $\sqrt{k}$. The fractional uncertainty then is

$$\frac{u(k)}{k} = \frac{\sqrt{k}}{k} = \frac{1}{\sqrt{k}}, \tag{82}$$

implying it pays off to score a higher count total, for instance by using a source of higher activity or by measuring during a longer time period.

We now address two more topics about the Poisson distribution: the normalisation of the distribution and the calculation of the mean using Eq. (79).

Normalisation: For a probability distribution the sum of all probabilities equals unity. This applies to the Poison distribution as well:

$$\sum_{k=0}^{\infty} P_\mu(k) = 1. \tag{83}$$

The correctness of Eq. (83) immediately follows from the Taylor series expansion of e^μ : $e^\mu = \sum_{k=0}^{\infty} \mu^k/k!$

Calculation of the mean: the property that μ is the mean number of decay processes can be proven by summing over all possible value of k, each multiplied by the probability of its occurrence:

$$\bar{k} = \sum_{k=0}^{\infty} kP_\mu(k) = \sum_{k=0}^{\infty} k\frac{\mu^k}{k!}e^{-\mu} = \mu e^{-\mu}\sum_{k=1}^{\infty}\frac{\mu^{k-1}}{(k-1)!} = \mu \tag{84}$$

The Poisson distribution typically holds for a probability experiment with a very large population and a small probability of success. Here, success means that a nucleus decays; k in Eq. (79) counts the number of successes. For radioactive decay the population is the very large number of radioactive nuclei (here: $N \sim 10^{18}$), while the probability of success is given by the small number $w = \frac{\Delta t}{\tau}$ (here: $w \sim 10^{-16}$). In the terminology of statistics, each individual counting experiment during a time period Δt can be seen as N attempts (since there are N nuclei). In each counting experiment, the count is made how many of N attempts lead to success.

In the limit of very large μ, the asymmetric Poisson distribution turns into the symmetric normal distribution, which in general is determined by two independent parameters (the mean and the standard deviation). For rare and random decay processes, however, only the parameter μ of the original Poisson distribution determines the resulting normal distribution (again with: mean = μ and standard deviation = $\sqrt{\mu}$).

Determination of the Half-Life of ^{40}K

From the preceding chapter it follows that the half-life of a nuclide can be determined by measuring the activity as a function of time. One could measure the count rate and, if needed, correct it for background radiation. The corrected count rate is proportional to the activity. If the corrected count rate is plotted on a logarithmic scale versus time, then within the measurement uncertainty a straight line results with a slope $-\lambda$, from which τ can be calculated using Eq. (77).

However, a sample containing ^{40}K is not expected to show a decrease of activity during any realistic measurement time, given the half-life of billions of years. Therefore, our first method to obtain the half-life is based on Eq. (78), by determining $A(0)$ and $N(0)$. Indeed, according to Eq. (78) we have

$$\lambda = \frac{A(0)}{N(0)} \tag{85}$$

And from Eq. (77) it follows

$$t_{1/2} = \frac{N(0)}{A(0)}\ln 2 \tag{86}$$

The other way of determining τ is by experimentally obtaining the histogram of count totals, each total measured during the time interval Δt. By fitting a Poisson distribution to the histogram the mean μ is obtained, which via Eq. (80) leads to τ.

Determining the number of ^{40}K nuclei in one gram of KCl The quantities $N(0)$ and $A(0)$ in the equation for the half-life relate to the same number of ^{40}K nuclei. This situation can be realized by reducing measurement results from different samples to the mass of one gram of KCl. When we redefine $N(0)$ as the number of ^{40}K nuclei in one gram of KCl, then $N(0)$ can be expressed in p, N_A and M. p is the fraction of the presence (abundance) of the isotope ^{40}K in the natural isotope mixture, N_A is the Avogadro constant, M is the molar mass of KCl. The values of p, N_A and M can be looked up in a data compendium, for example [3].

Determining the activity of one gram of KCl To determine the activity of one gram of KCl, we use the setup depicted in Figure 16. The heart of the setup is a Geiger-Muller (GM) detector, applied in this experiment to detect electrons.

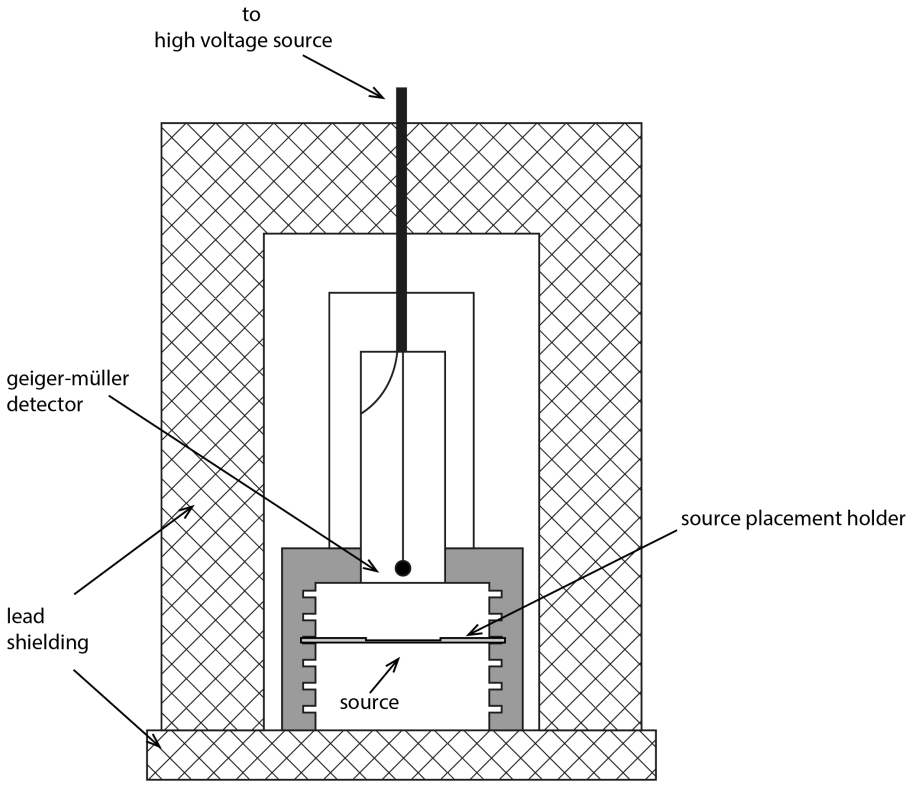


Figure 16: Geiger-Muller counting setup for determining the half-live of ^{40}K. Sample holders are available to accommodate different thicknesses of KCl powder. The holder should be placed in the uppermost slot.

Typically, the number of detections induced by the KCl sample is counted during a certain time period, using a counter/timer. The count rate R is obtained by dividing the count total N_{count} by the measurement time $\Delta t : R = N_{count}/\Delta t$. Since the count rate is not equal to the activity, it is therefore expressed in s^{-1} and not in Bq. The count rate is recalculated into the activity by making several corrections. The discussion below is limited to those corrections that give a significant contribution.

1. Background radiation The count rate R should be corrected for the background radiation. To determine the background, a count is made without the sample in place. The count is taken over a time period as long as reasonably possible, to minimize the relative uncertainty of the correction. The resulting count rate R_{corr} corrected for the background then is

$$R_{\mathrm{corr}} = \frac{N_{\mathrm{count}}}{\Delta t} - \frac{N_{\mathrm{b}}}{\Delta t_{\mathrm{b}}}, \tag{87}$$

where N_b is the count total for the background and Δt_b the duration of background counting. From R_{corr} the count rate per gram is given by

$$r_{\mathrm{corr}} = \frac{R_{corr}}{m} \tag{88}$$

with M the mass of the sample in grams.

2. Efficiency Since not each decay induces a detection, further correction of the count rate is needed. The ratio of the count rate to the activity is called the *efficiency*. The total efficiency is determined by three factors: (a) the *self-absorption* in the source, (b) the *β-efficiency* of the GM detector and (c) the *geometric efficiency* of the setup.

a. *Self-absorption*.

With increasing sample thickness, more decay products will be absorbed in the source itself and will thus not reach the detector. This effect of self-absorption can be corrected for by measuring the rate r_{corr} as a function of sample thickness d and by extrapolating the results to zero thickness. To enable this, sample holders are available to accommodate different thicknesses of KCl powder. The extrapolation including uncertainty analysis is performed in Python, using a least squares fit of a polynomial to the data points and taking into account the error bars of the points. When using a polynomial, the following relation between r_{corr} and d is assumed to hold:

$$r_{\mathrm{corr}}(d) = a_0 + a_1 d + a_2 d^2 + \ldots \tag{89}$$

The least squares fit yields the coefficients $a_0, a_1, a_2, ...$ of the polynomial. Extrapolation to zero thickness implies setting $d = 0$ in the fitted polynomial, thus obtaining $r_{corr,0} = r_{corr}(0) = a_0$. Thus, $r_{corr,0}$ is the self-absorption corrected count rate of 1 gram of KCl, obtained via determining the count rate per gram as a function of sample thickness including correction for the background. In the symbol $r_{corr,0}$ the index "corr" denotes correction for the background, while the index "0" denotes extrapolation to zero sample thickness.

b. *The β-efficiency, ϵ*

This is the ratio of the number of detected β-particles to the number of β-particles reaching the GM-tube. The value of ϵ is given in the data sheet of the setup.

c. *Geometric efficiency G*

This is the ratio of the number of electrons reaching the detector to the number of electrons that are emitted by the source. Since electron emission from the source is omnidirectional and the detector does not completely enclose the source, only a fraction of the electrons will reach the detector. This geometric efficiency relates to the solid angle under which the detector "sees" the source (solid angle is the three-dimensional equivalent of angle in the plane). For a point source the solid angle can be easily calculated. In our case, however, the source is not a point source, resulting in a rather complex expression for G:

$$G = 0.5 \times \left[1 - \frac{1}{(1+\beta)^{1/2}} - \gamma\frac{3}{8}\frac{\beta}{(1+\beta)^{5/2}} - \gamma^2\left(-\frac{5}{16}\frac{\beta}{(1+\beta)^{7/2}} + \frac{35}{64}\frac{\beta^2}{(1+\beta^{9/2})}\right) - \ldots\right] \tag{90}$$

Here $\beta = (R_{\mathrm{d}}/s)^2$ and $\gamma = (R_{\mathrm{s}}/s)^2$, with s the distance between the source and the detector window, R_d the radius of the circular and flat detector window and R_s the radius of the circular and flat source (see Figure 17).

You can verify that:

$$\begin{array}{ll} s = 6.7\mathrm{mm}, & u(s) = 0.1\mathrm{mm(highest\ position)}, R_{\mathrm{d}} = 14.5\mathrm{mm}, \\ u\left(R_{\mathrm{d}}\right) = 0.3\mathrm{mm} R_{\mathrm{s}} = 15.00\mathrm{mm}, & u\left(\mathrm{R_s}\right) = 0.05\mathrm{mm} \end{array} \tag{91}$$

Equation (90) is rather complex, making the complete uncertainty analysis for the efficiency G rather involved. Therefore, the value of G and the corresponding uncertainty are given here: $G = 0.215$, with $u(G) = 0.006$.

3. Fraction of β^--decay

The last correction to be discussed relates to the property of the GM detector of only detecting the β^--decay. The very few positrons emitted in the β^+-decay (see Figure 14) annihilate with electrons of the K^+ and Cl^- ions and thus will not reach the detector. This positron annihilation results in γ-radiation, for which the detector has a very low efficiency. Electron capture (EC, see Figure 14) will mainly result in X-ray photons, of which a very high fraction is absorbed in the source. It follows that of the three decay channels only the fraction of β^--decay is seen by the detector. This fraction is denoted by the symbol f.

Assignments

For the formulation of the problem, see the chapter GOAL. A separate afternoon will be scheduled to work on the problem analysis aad a measurement plan. For data processing and data plotting you will

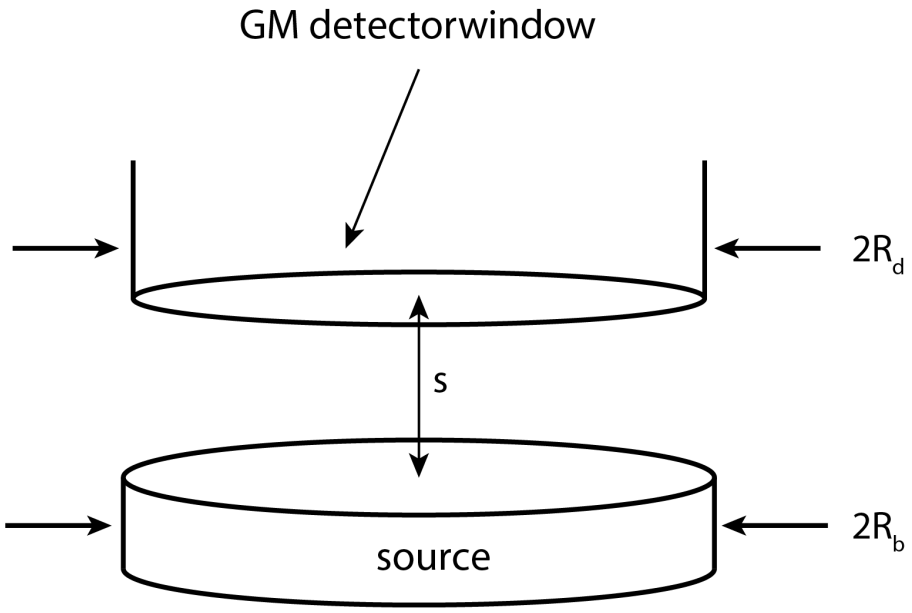


Figure 17: Geometry of the source and detector, with indication of relevant dimensions.

use Python. In particular, you will use Python for making least squares fits of functions (polynomial and Poisson distribution) to the experimental data.

Part 1: Determining the half-life from the activity and the number of nuclei Before equation (86) can be applied to determine the half-life, the number of nuclei $N(0)$ and the measured activity $A(0)$, both per gram, should be expressed. These can be calculated from the measuring data or taken from literature. The related assignments are:

1. Express $N(0)$ in p, N_A and M.
2. Look up the values for p, N_A and M, including their uncertainties, and write these down.
3. Write $A(0)$ as a function of the quantities $r_{corr,0}$, ϵ, G and f. You will calculate the value of $r_{corr,0}$ in assignment 9 below, the three other quantities can be looked up.
4. Look up the values for ϵ, G and f, including their uncertainties, and write these down.
5. Write the half-life $t_{1/2}$ as a function of the quantities p, N_A, M, $r_{corr,0}$, ϵ, G and f using Eq. (86) and the expressions derived for $N(0)$ and $A(0)$
6. Derive the equation for the relative uncertainty of $t_{1/2}$ using the expression for $t_{1/2}$ just obtained. Determine those contributions to the total uncertainty that can be neglected.
7. Study the measurement setup. The sample holders should be placed in the upper-most slot, since the G-value given in paragraph 4.2 holds for this position.
8. The raw count data, measured as a function of sample thickness, must be converted to count rates r_{corr}. Based on Eqs. (87) and (88), write r_{corr} as a function of quantities that either can be measured directly or can be set to a certain value.
9. To determine $r_{corr,0}$, make a plot in Python of r_{corr} as a function of d and extrapolate the polynomial fit to $d = 0$. Include the error bars for the data points in your plot, for which you will need the uncertainty of r_{corr}. Derive the expression for the uncertainty of r_{corr} and determine which contributions can be neglected.
10. What is the literature value of the half-life of ^{40}K in literature?

Part 2: Determining the half-life from the Poisson distribution You will repeatedly count the number k of decayed nuclei during a time interval Δt and determine the Poisson distribution $P_\mu(k)$ corresponding to the count totals. The resulting fit parameter is μ, which is related to the half-life according to Eq. (80). Assignments:

1. To obtain an accurate approximation of the Poisson distribution, you have to perform at least 200 individual counts. Choose the time interval Δt such that the average number of counts in the interval is less than 10. Choose the sample holder with the most shallow cavity (why?) and again use the uppermost slot.

2. Determine the mean from the count totals, using Python. This will be your first estimate of μ. A logical way for this is by putting the count totals in a vector N. Additionally, use:

   ```
   min(N)
   max(N)
   len(N)
   mu = sum(N)/len(N)
   ```

3. With Python, plot a histogram of the count totals. Then add the Poisson distribution for the value of μ (from assignment 12) to the same plot. For the plot you may approximate the Poisson distribution by replacing $k!$ in Eq. (79) with the gamma function for the argument $(k+1)$. By doing so, $P_\mu(k)$ can also be calculated for non-integer values of k. Don't forget to add to the top of the script:

   ```
   from scipy.special import gamma
   ```

   For calculating and plotting a histogram, with 25 bins, with the centers corresponding to the number of counts within Δt (here 0 to 24), the input in Python will be

   ```
   n = plt.hist(N, bins=25, range=[ -0.5, 24.5], rwidth=0.9)
   ```

   We like to add the theoretical distribution corresponding to the value of μ of Assignment 12 to the plot. To this end prepare an array with closely spaced values `kf`. For instance:

   ```
   kf = np.linspace( -0.5,24.5,100)
      Pois = len(N)*(mu**kf)*np.exp( -mu)/gamma(kf+1)
      plt.plot(kf,Pois,color='k', linestyle='dashed')
   ```

If needed, you can adapt the axes: range, labels, font size.

1. Then, use `curve_fit` to fit the frequencies (= the number in the bins) to the Poisson distribution. The fit function is:

   ```
   def func(x,a):
           y = len(N)*(a**x)*exp( -a)/gamma(x+1)
           return y
   ```

   NOTE: It's not possible to directly multiply `func()` in the argument of `curve_fit` with `len(N)`. Therefore, we have to incorporate `len(N)` in the function definition. You could fit this as a parameter as well, but this is not required. This yields the fitted $\mu(= a)$, as well as the extreme values of μ determined by the error in the fit.

2. Plot the Poisson distribution corresponding to the μ just obtained in the same plot that already has the histogram and the previously derived distribution (assignment 13). Evaluate the similarity of the two distributions to the histogram.

   Is the figure complete? Save it.

   (a) Compare the obtained half-life to the value of Part 1, and compare each result to the literature value.

Measurement Plan Before performing the measurements, you have to set up a measurement plan. Discuss the plan with the assistant, whom you can also ask about issues that remained unclear. If needed, perform a test measurement. After approval by the assistant, you will execute the measurement plan on the afternoon scheduled.

[1] *Radiological Health Handbook*. U.S. Department of Health Education and Welfare, 1970.

[2] I.G. Hughes and T.P.A. Hase, *Measurements and their Uncertainties*. Oxford University Press, 2010.

[3] A.M. James and M.P. Lord, *MAcMillan's Chemical and Physical Data,* MacMillan Press, 1992

0.5.4 Spectral lines of Hg and Na

Goal

In this experiment, we will study the light emitted by a mercury lamp (Hg) and sodium (Na). The spectrum for such lamps is comprised of a few clear spectral lines. The goal for this experiment will be to determine the wavelengths of those lines as accurately as possible. We will use a diffraction grating to do so. One of the questions related to this experiment is: how small can the difference in wavelength of two spectral lines be for those lines to still show up separately on the setup?

Preparation

Before you arrive at the practical, you are expected to have properly studied this chapter of the manual, and to have written down the answers to the twelve questions that are noted in your lab journal. If any questions came up during your preparation, write them down in your journal and ask them at the beginning of the experiment.

Theoretical Background

What is a diffraction grating? A diffraction grating is a plate with a very large number of parallel lines on it, which are spaced an equal distance from each other. A schematic representation is shown in Figure **??**.

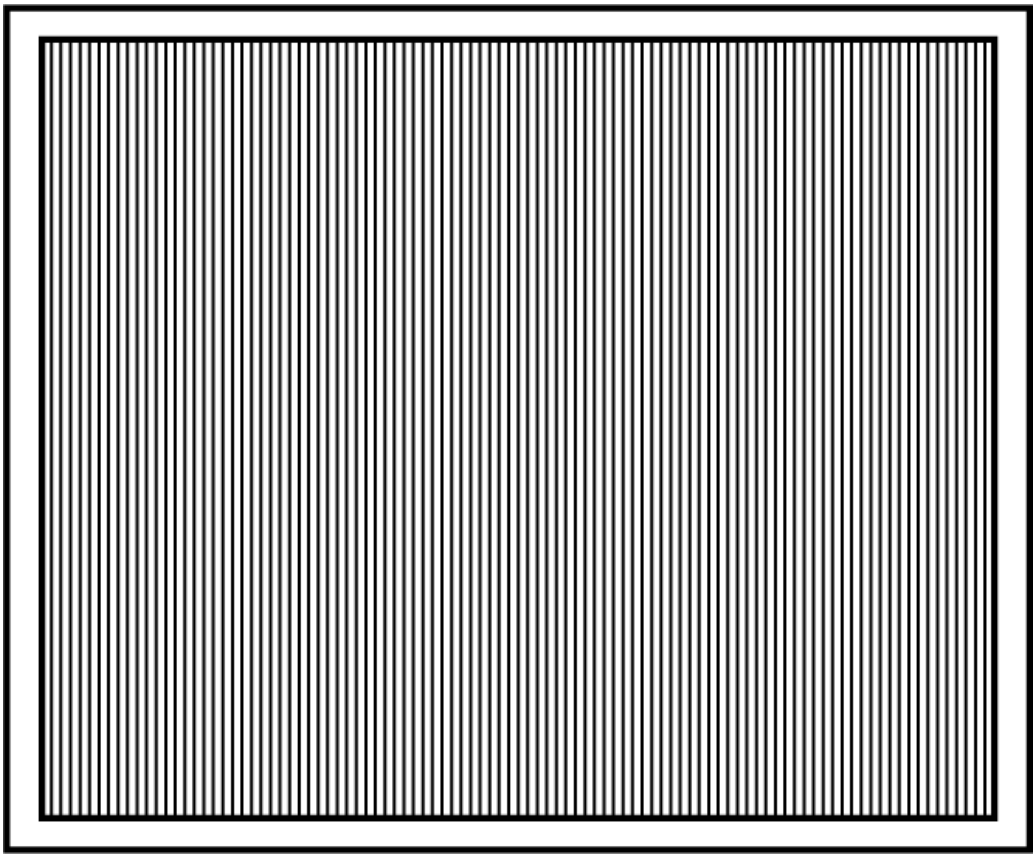


Figure 18: Schematic representation of a diffraction grating. Typical measurements are 5x5 cm. The number of lines per mm is in the range 100-1000.

Typical measurements are 5x5 cm and a typical value for the number of lines per mm is in the range 100-1000. A diffraction grating can be either transmissive or or reflective, known, respectively, as a transmission or a reflection grating.

What does a diffraction grating do? A diffraction grating is an optical instrument which diffracts light. The larger the wavelength, the larger the diffraction. Figure 19 shows what happens when 'white' light with a continuous colour spectrum between blue and red falls on a *transmission grating*. A part of the light goes straight ahead and remains white. This is the 0^{th} order diffraction pattern. Under and above of this order we see spectacular ribbons of which show a colour range from red to blue". These are the 1^{st} order diffraction patterns. These patters can repeat itself multiple times (2^{nd} order, 3^{rd} order, ...).

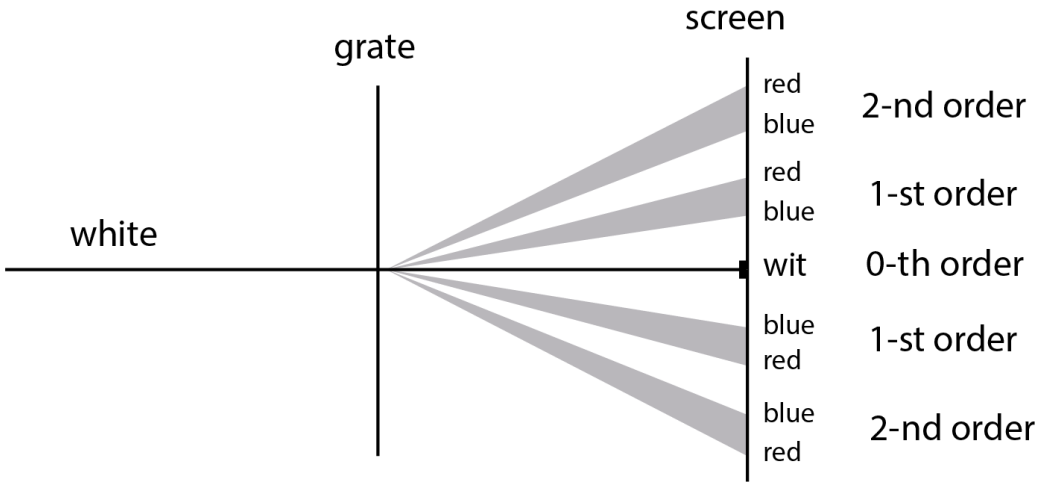


Figure 19: A transmissive grating in operation. White light with a continuous colour spectrum from blue to red is split into multiple colours and orders.

The purpose of a diffraction grating is to study the colour spectrum of a light source. As seen in Figure 19, the vertical distance on the screen is a measurement for the wavelength of a certain colour. A diffraction grating also allows one to select a desired colour from white light.

Light rays and wavefronts This experiment focuses on the wave characteristics of light. The terms interference and diffraction are central to this. The Huygens Principle is often used to explain diffraction phenomena. This principle relates to the extension of wavefronts. A wavefront is a collection of points where the "vibration" is in the same phase. Waves in water are in fact wavefronts. If we throw a stone into the water, then we see wavefronts that extend in a circular fashion. If we look at waves washing up at the beach, we see a series of parallel wavefronts. These characteristic wavefronts are shown schematically in Figure 20.

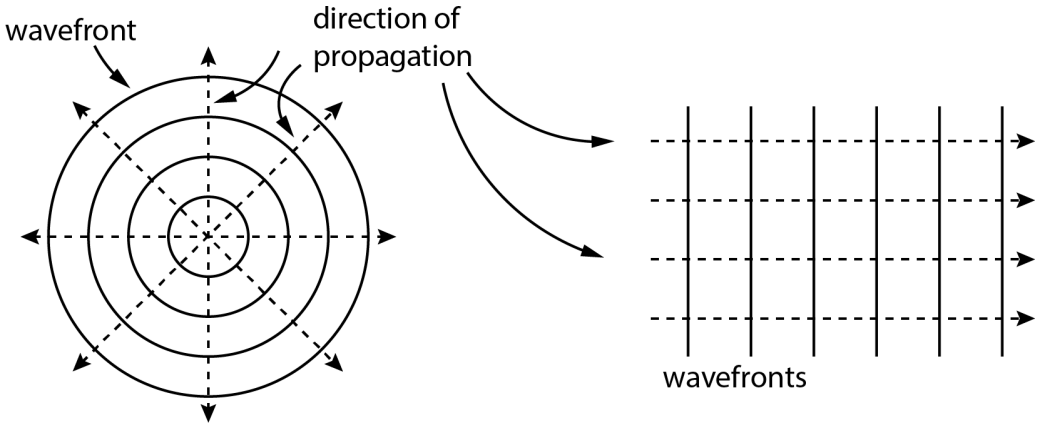


Figure 20: Two characteristic wave patterns. Left: the extension of circular (spherical) shaped wavefronts. Right: plane (flat) waves moving to the right (= parallel light bundle).

The relation with light rays is that the propagation direction of the light in a certain point is displayed by a light ray. If a wave propagates in an isotropic medium, then the direction of it is

perpendicular to the wavefront. See Figure 20.

Huygens principle (Wolfson, ch. 32.5) *All points on a wavefront work like point sources for spherically shaped 'waves' which propagate with lightspeed in the medium. A short time Δt later, a new wavefront is formed by the surface that touches all 'waves' which propagate in the previous direction of propagation.*

Figure 21 shows how this works. On the left we see how a spherically shaped wavefront at t develops itself to a new spherically shaped wavefront at $t + \Delta t$. For four points on the wavefront at $t = t$ is shown how the waves develop themselves in a time interval Δt. The new wavefront at $t + \Delta t$ is the 'tangent' line to these waves.

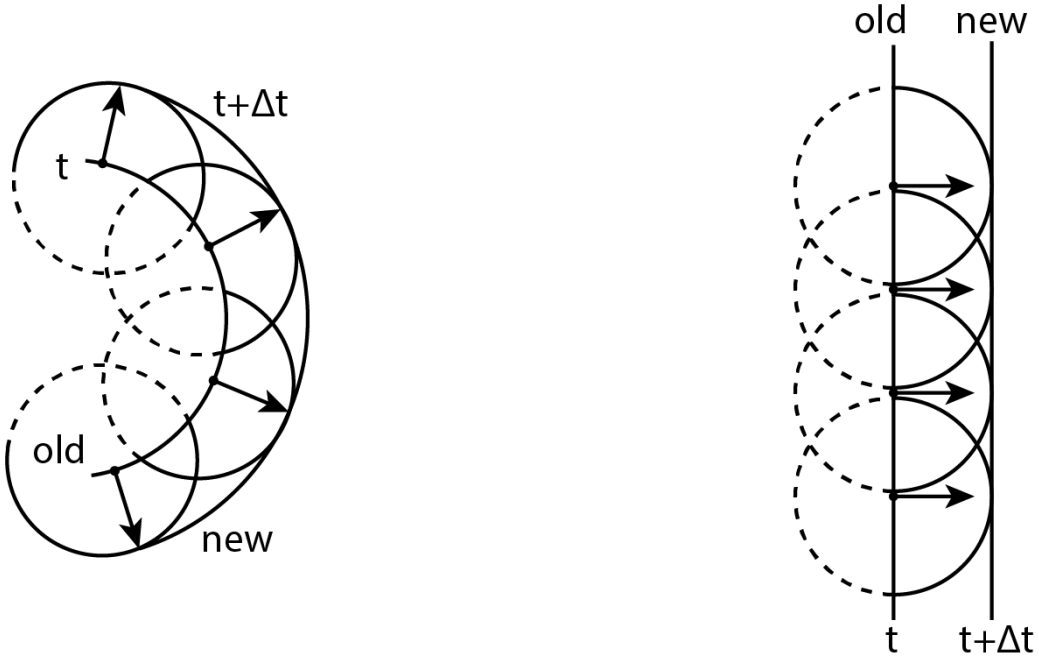


Figure 21: Propagation of a wavefront according to the Huygens principle. Left a spherically shaped wavefront and right a plane wavefront.

The image on the right in Figure 21 displays the same way of propagation for a plane wavefront. As said, the Huygens Principle explains the diffraction of light. For the first application, we look at what happens when a plane wave falls on a screen with a hole in it. The diameter of the hole is comparable with or smaller than the wavelength of the light. See Figure 22.

In Figure 22 a plane wave falls on the screen from the left side. The light that falls in the hole, emits spherically propagating waves according to the Huygens principle. If the size of the hole is of the same order as the wavelength, most of the light is diffracted. To see a demonstration with a tank filled with water, look up 'ripple tank diffraction' on YouTube. In Fig. 32.33 (c) of Wolfson you can also see how the intensity of the light.

Interference The next question is: what happens if there are two holes in the screen? This is shown in Figure 23 where a plane wave falls perpendicularly from the left side on a screen with two holes (diameter $\geq$ wavelength). The distance between the holes, d, is three times the wavelength for instance. The plane wave causes the holes to emit spherically shaped waves in phase. The light in a point P (See Figure 23) is a superposition of the light that is emitted by the two sources. Both sources of light interfere with each other. The distance between the upper hole and P that is covered by the light is not the same as the distance between the bottom hole and P covered by the light. The length of the path covered by the light is different, meaning that the phase of the two waves at P is not equal. However, it is possible that the phase difference is exactly the same as a whole number times 2π, as shown in Figure 23. The phase difference is 2π and the waves interfere constructively in

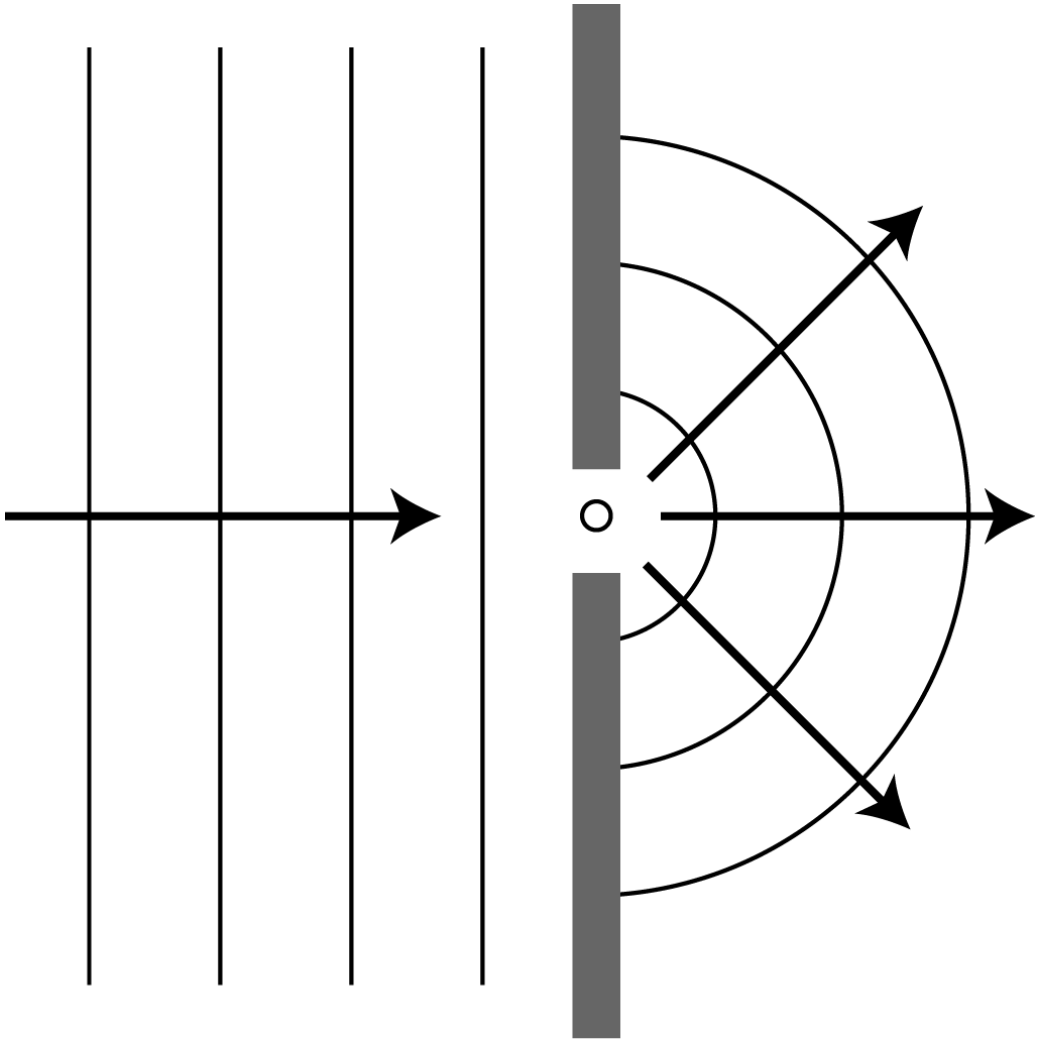


Figure 22: A plane light wave falls on a screen with a hole with a diameter comparable to the wavelength. According to the principle of Huygens, the hole acts like a point source that emits spherically extending waves.

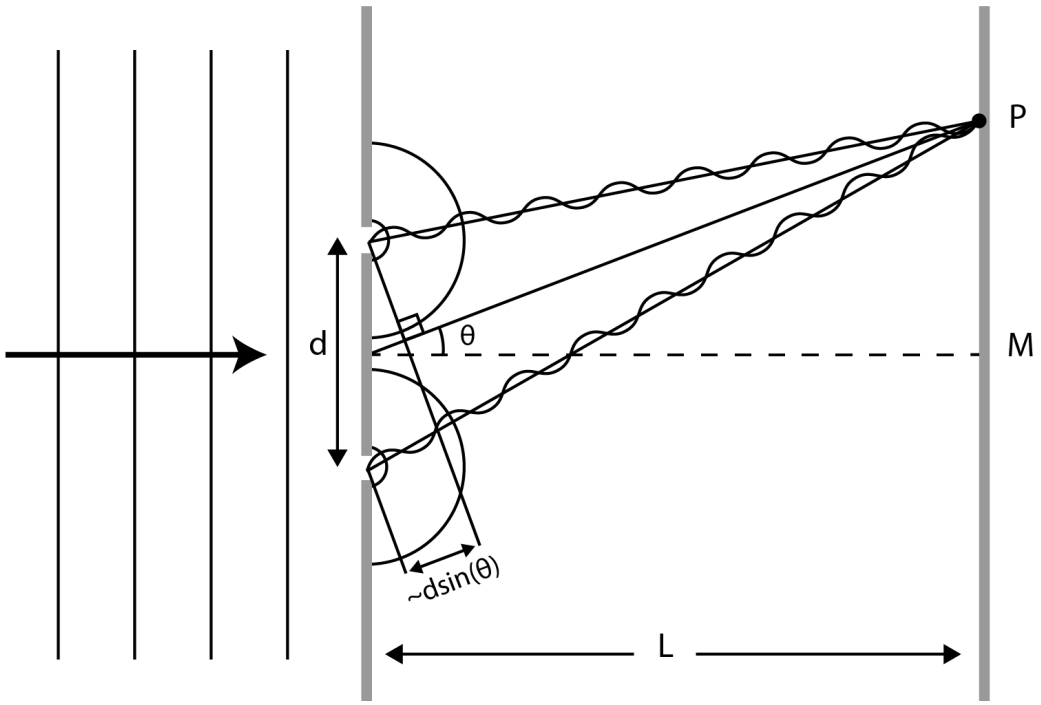


Figure 23: A plane light wave falls on a screen with two holes. These holes work like point sources that emit spherically propagating waves in phase. Both waves interfere constructively in point P in this situation.

P. For points just above and below P, the phase difference is different and there is no longer 100% constructive interference. Thus the light intensity will be lower.

It can be deduced from Figure 23 that the path difference can be approached by $d \sin\theta$. The angle θ determines the place of P (and vice versa). Constructive interference occurs when:

$$d \sin\theta = m\lambda \tag{92}$$

with m an integer that we call the order.

Interference in "a point far away". Multiple sources. In Figure 23 the distance between the screen and a source is of the same order as the distance between two sources. This concerns distances of the order 10 μm. In practice we work with distance in the range of 10 - 100 cm. So, we are interested in what happens in *points* that are *far away* from the sources.

Figure 24 shows that the paths from the sources towards P now run parallel to each other. Because of this, the condition for constructive interference (eqn. (92)) is exact.

It is not hard to imagine what happens if we add an extra hole. We make sure the third hole is located on the line that goes through the two other holes and that the mutual distance is d (see Figure 24 above). The third source also emits, in phase, a wave that contributes to the amplitude in the far point P. The correlated path runs parallel to the other path. The path difference is λ or 2λ. So the contribution of the third source interferes constructively with the other sources.

We can continue this reasoning for $4, 5, .., N$ holes that lie on a line with mutual distance d. In this manner we construct the theoretical description for a diffraction grating with N lines. The amplitude of the electric (magnetic) field of the light in P is N times as large as the amplitude that is produced by a single hole.

If we look at Figure 24, it is tempting to drawn lines that go through the points where the drawn waves run in phase (the dotted lines on the right side in Figure 24). These lines are perpendicular to the direction of propagation and you might be inclined to conceive these lines as wavefronts of a parallel bundle. That thought is fine, but you must be sure that for instance the points on the dotted line between the points a and b (in Figure 24) vibrate in phase with the vibrations in a and b. Close to the sources this is a difficult matter, but *far away* from the sources this is certainly the case. You can

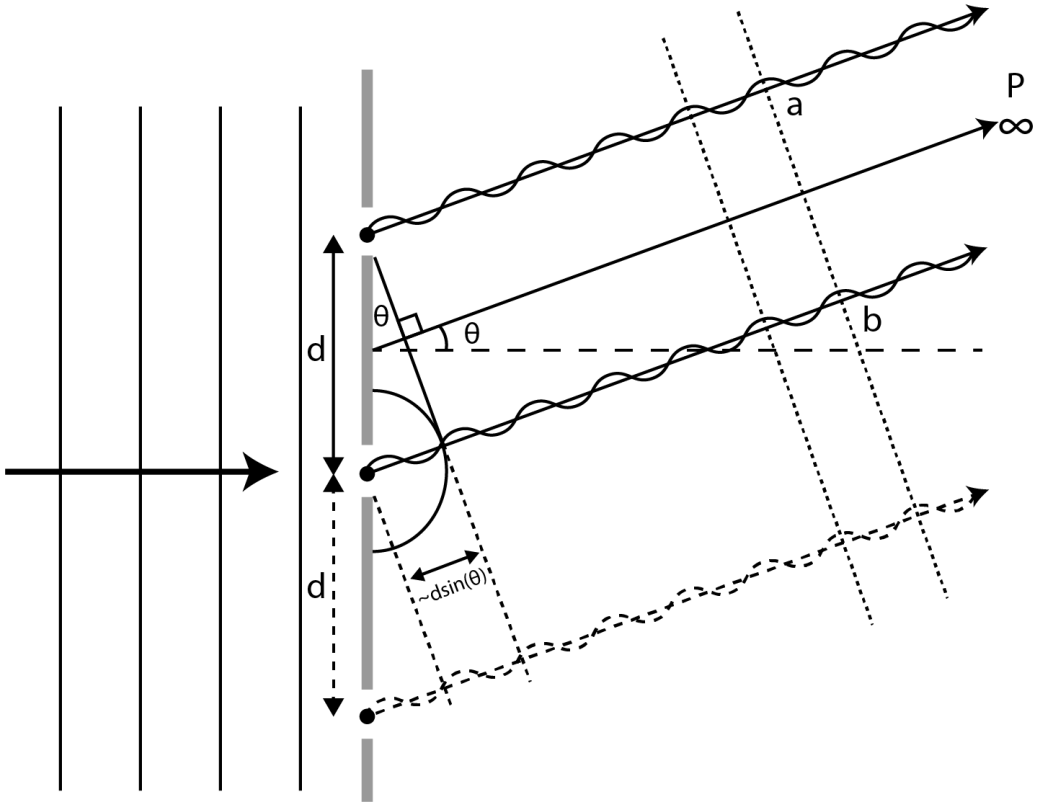


Figure 24: The upper part of this figure is identical to Figure 23, the only difference is that the point P is now infinitely far away from the sources. There is constructive interference in point P. The bottom right part of the figure shows that extra holes with spacing d also contribute to the constructive interference. The slanted striped lines on the right side of the figure indicate the development of plane wavefronts (parallel bundle).

determine this using Huygens' principle. This also follows from the wave equation. For light waves that are possible in a uniform medium, the direction of propagation is always perpendicular to the wavefronts.

From transmission to reflection grating. Grating equation. Up until now, we have discussed the operation of a diffraction grating according to a transmission grating. However, reflection gratings are more commonly used. These have accurate grooves instead of translucent lines. Everything that has been said above about plane waves, point sources, including (92) is also true for reflection gratings.

At the derivation of eqn. (92) it is assumed that the incoming plane wave falls perpendicular into the diffraction grating. That restriction does not have to be made. The plane wave can also fall at an angle with the normal of the diffraction grating. Figure 25 demonstrates how the angles of the exiting bundles can be found for a parallel bundle that falls at angle i with the normal of the diffraction grating.

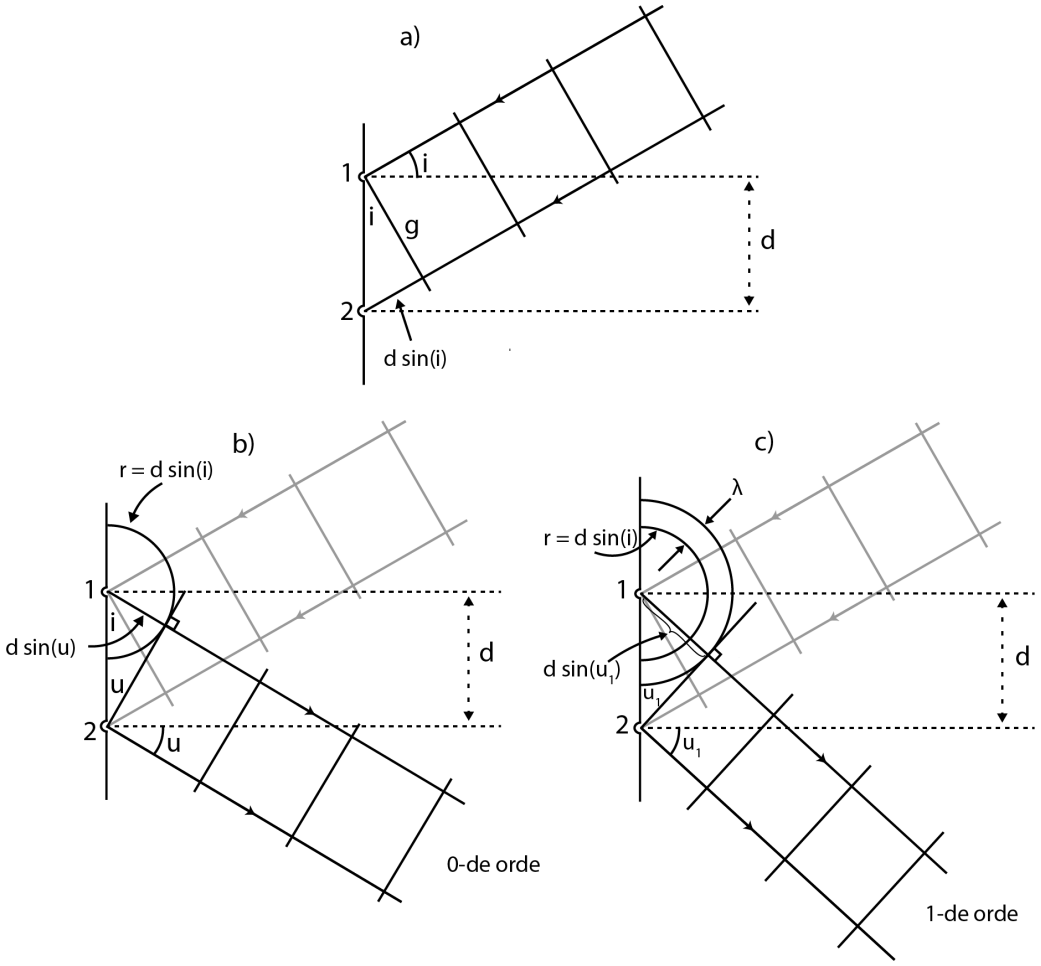


Figure 25: onstruction of the angle of reflection for a reflection grating. a) a plane wave falls at an angle i onto the diffraction grating. The wavefront g reaches grating line 1. This incites the development of a spherical wavefront from 1. A time interval $\Delta t = (d \sin i)/c$ later reaches the plane front g grating line 2. b) at this point the spherical wavefront from 1 has the radius $d \sin i$. The vibrations on this wavefront and the ones in point 2 are in phase. The line that goes through 2 and touches the spherical wavefront determines the direction of propagation of the 0-th order exiting wavefronts (wavefront is perpendicular to direction of propagation) and c) the construction of the direction of the 1-st order exiting bundle.

In Figure 25a, a parallel bundle falls at an angle i with the normal of the diffraction grating. A plane wavefront will first reach grating point 1 and, according to Huygens, that point emits spherically shaped waves. A time Δt later the wavefront reaches grating 2. From Figure 25a it turns out that

$\Delta t = (d \sin i)/c$ (d is the distance between two grating lines and c is the speed of light). In Figure 25b we see how far the spherical wave from grating point 1 has propagated. As said, the phase of points on a wavefront is by definition equal, but in this case the vibration in grating point 2 is also in the same phase. We can now draw a line which goes through grating point 2 and touches the spherical wave. This line is cut by the perpendicular line that goes through grating point 1 and the point where the first line touches the spherical wave. These two lines can be seen as the "precursors" a wavefront / direction of propagation pair before the exiting wave (as discussed at Figure 24). Just like in Figure 24 more grating lines can be added to this consideration.

Figure 25b is special because in the way that it turns out from the figure that $d \sin u = d \sin i$. In this case the exiting bundle is called the 0^{th} order diffraction bundle.

Of course not only the drawn wavefront in Figure 25b has the same phase as the phase in grating point 2. Wavefronts belonging to spherical waves that have left earlier,have the phase as the phase in grating point 2 as well (apart from a difference of $2\pi n$ of course). In Figure 25c you can see a bundle emerge from the sketching of the tangent line on the wavefront with radius $(d \sin i) + \lambda$. It turns out from the figure that in this case $d \sin u_1 = (d \sin i) + \lambda$. Of course we can also do this for wavefronts with radius $(d \sin i) + m\lambda$, with m an integer. For an incident angle i we can determine the angles u_m from possibly exiting bundles with:

$$d \sin u_m = d \sin i + m\lambda \tag{93}$$

This is the grating equation. It follows from this equation that, if a plane wave with a certain wavelength falls on the diffraction grating under an angle i, different plane waves are reflected with the same wavelength under different angles u_m. We can determine the wavelength of a certain component by accurately measuring i and u_m, and using eqn. (93).

Processing and generating light with a flat wavefront A direct measurement of the direction of a parallel bundle will not be very accurate. It is far more effective if we first let the bundle fall on a positive lens. The bundle then converges to a point in the focal plane. See Figure 26.

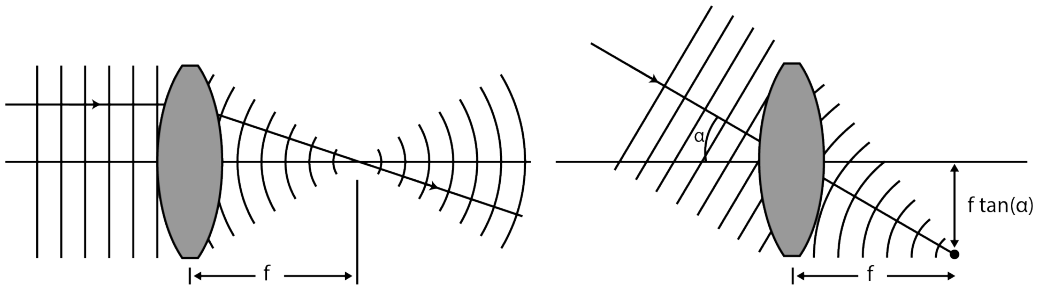


Figure 26: The course of the wavefronts at the focusing of a light bundle with a plane wavefront. On the right, the relation between the direction of the bundle and the position of the focal point is shown.

When the focal point is placed on a screen, the position of the spot can be used to determine the direction of the bundle. The relation between the direction of the bundle and the position on the screen is shown on the right side in Figure 26.

Incident light bundle. Optical system. At the derivation of eqn. (93) we assume that the incident light is a parallel bundle. This assumption is not justified for the lamp used in this experiment. The light of the mercury lamp is divergent by nature. To counter this, we will need to *collimate* the sources.

Collimation is easily done for a point source. You only need to place the source in the focal point of a convergent lens. In fact, the exact opposite of what is happening in Figure 26. Unfortunately, the lamp used in our experiment is not a point source. A small slit placed in the focal point of a lens will be needed to collimate the source, see Figure 27.

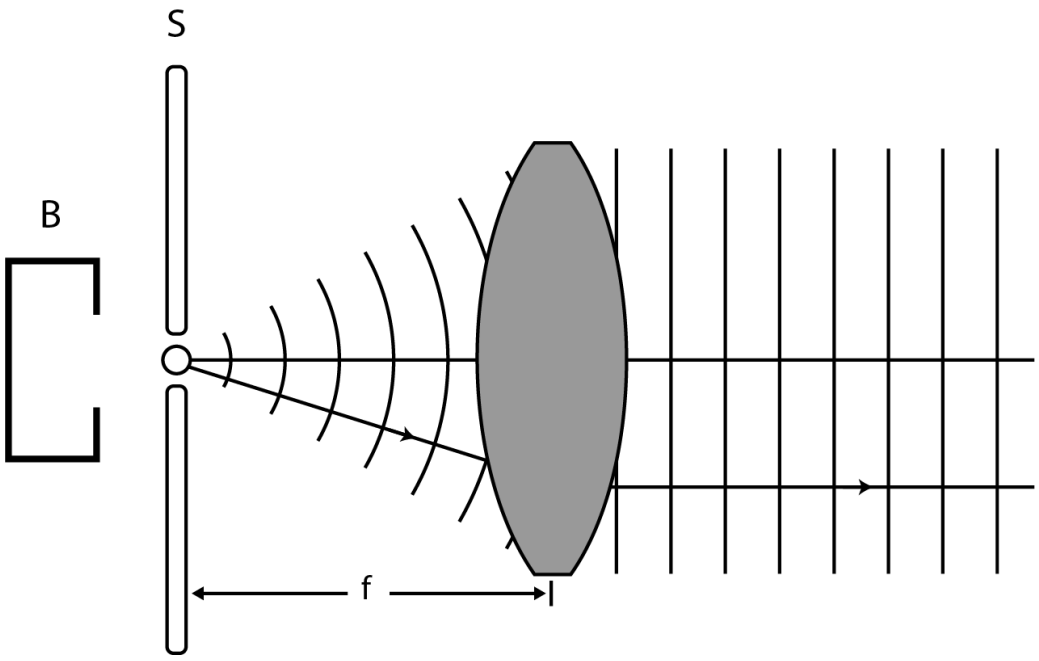


Figure 27: By placing the lens at the right position, the source is collimated.

The narrower the slit, the better the light from the lens can be characterised as one plane wave with one sharp direction. A disadvantage is of course that the intensity of the light that falls on the diffraction grating diminishes as the slit is narrowed.

Our setup for the determination of the wavelength of spectral lines with a diffraction grating now consists of: a source (lamp), a slit, a collimating lens, the diffraction grating, an image lens, and a screen. These parts will be setup in such a way that an image of the slit is formed on the screen.

Method

Setup The setup that we will use for the determination of the wavelength of the visible spectral lines of the Hg lamp is shown in Figure 28. This setup is also called the monochromator setup. The only moving part of the setup is the diffraction grating. The grating will be mounted on a turntable with an angle distribution that makes it possible to measure the angle (See Figure 28).

Using a slit directly behind the lamp, and a lens F_1 produce a collimated (parallel) light bundle. This bundle hits the diffraction grating. We have seen that, because of the diffraction of the light through the grating, depending on the wavelength, parallel bundles are reflected at different angles. By changing the angle of the diffraction grating in respect to the incident light bundle, we can make a certain colour fall on a set point on the screen or the camera with the help of lens F_2. From the according angle, we can calculate the wavelength of the light.

At one specific angle for the diffraction grating, it seems like the light is conveyed. At this angle it seems like the light is portrayed by a normal mirror (via the lens F_2) to the set point on the screen. In fact, the 0^{th} order diffraction pattern is portrayed in the set point. This does not change with the wavelength of the light. This position is shown in Fig. 11 by the dotted line. The angle between the normal on the diffraction grating and the direction of the incident light is α (thus the angle between the incident and reflected wave is 2α). Note that α is determined by the positions of the lamp, the lens F_1, the lens F_2, and the point on the screen.

If we want to use a certain setup in practice for, for example, the measurement of a number of spectral lines, then we measure the angle between the normal of the diffraction grating and the 'mirror position'. In Figure 28 this angle is ϕ.

Elaboration If we call the incident angle i and the reflected angle u, then the following formulas apply (see Figure 28)

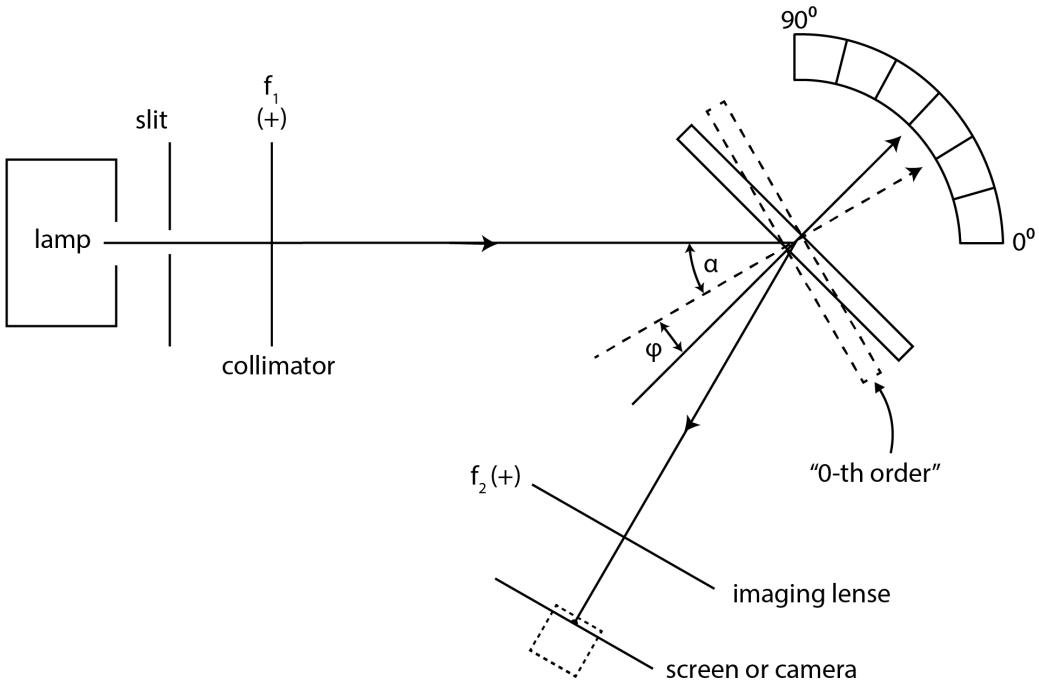


Figure 28: Setup for measuring the wavelength of spectral lines.

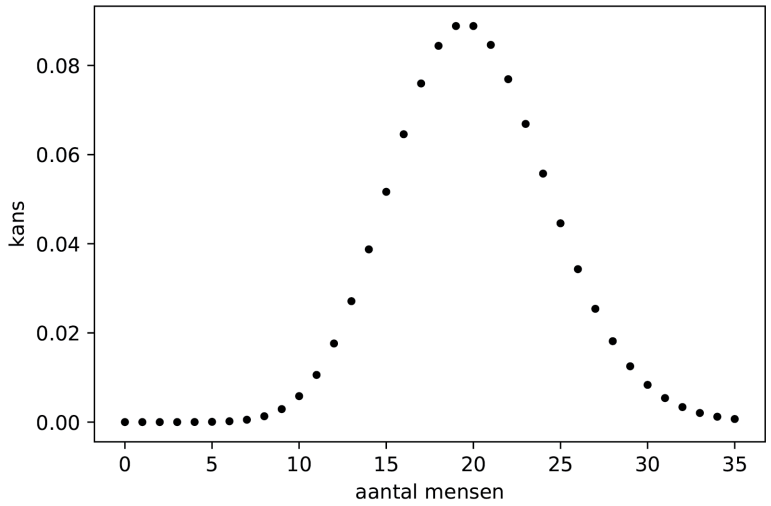


Figure 7: A poisson distribution of the chance that a number of people visit the restaurant

(94)

$$2\cos\alpha\sin\phi = -mN\lambda \tag{95}$$

In which N is the number of lines per length unit, λ the wavelength and the order of diffraction m. Note that m can be either positive or negative.

If the difference between two spectral lines is very small, they cannot be distinguished by the naked eye, and the diffraction grating cannot be rotated accurately enough.

To still be able to measure the little wavelength difference, we replace the screen by a camera. The enlargement of the camera allows us to be able see and accurately measure the separation between the spectral lines.

The procedure for this is as follows. From the grating equation (93) follows that with a set angle of incidence i, a small change in the wavelength λ is paired with a small change in the angle of reflection u. The relation is:

$$\Delta\lambda = \frac{\cos u}{mN}\Delta u \tag{96}$$

We calculate Δu from the distance of the spectral lines on the camera and the focal point of f_2. The unit of Δu is radian.

Resolving (separating) power Spectral lines have a certain width. It is clear that when the distance between two separate lines decreases, they will overlap and become indistinguishable from each other, at least with the naked eye. The intensity of the light as function of the wavelength is shown in Figure 29.

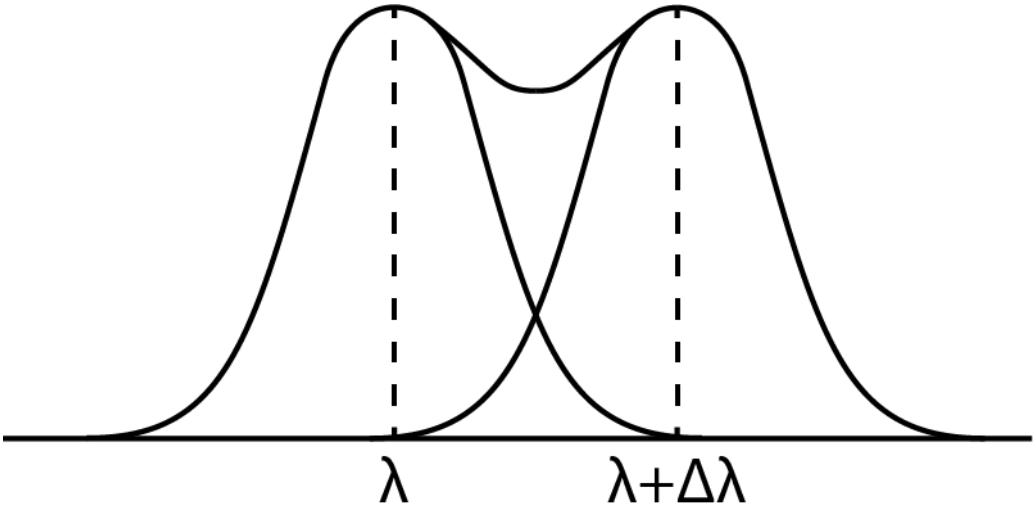


Figure 29: Resolving power

The resolving power of the setup is characterized by the quantity R:

$$R = \frac{\lambda}{\Delta\lambda} \tag{97}$$

in which $\Delta\lambda$ is the smallest observable difference in wavelength. This equation only holds for the first order. For higher orders we expect more spacing between spectral lines. Therefore the resolving power is higher at higher orders.

In practice, we make a prediction for $\Delta\lambda$ based on a picture of two nearby spectral lines. We use this estimated value and the average value of the two spectral lines. We can compare this value with the theoretical upper limit:

$$R_{theor} = \frac{\lambda}{\Delta\lambda} = mN_{tot} \tag{98}$$

in which m is the order of the spectral lines. N_{tot} is the total number of grading lines involved in the interference process. If the entire grating is used, N_{tot} resembles the total number of lines of the grating.

Method

Preparation phase **First and foremost:** visor trusts that you will use the equipment with care at the practicum. This applies especially to the handling with the diffraction gratings and lenses. Do not be clumsy! Read the following instructions and apply them!

- **ONLY HOLD THE DIFFRACTION GRATINGS AT THE RECTANGULAR FRAME OR AT THE STEM OF THE FRAME. A fingerprint or a scratch can not be cleaned or repaired. Diffraction gratings are very expensive.**
- The spectral lamps become hot when they are turned on. **Only hold the stem of the lamp. Otherwise you can get burned badly.**

- Hold the lenses and other optical components at the stem of the frame.
- Make sure that all components are tightly secured with the clamp and screw on the breadboard.
- **Again: Do not touch the lenses and diffraction gratings in the central part.**
- **Do not leave equipment on the edge of the table. Secure them. Things can fall on the ground and break.**
- **Do not leave components on the table. This way you prevent damage.**
- Do not look into the lamp directly. This can damage your eyes.
- Turn the lamp, camera and computer off when you do not use them.

A tip: When setting up the experiments, you have to make sure that the entirety of the optical system is in line. Make sure that the elements that have to be in line, really do line up. Look at your setup from above and use the lines on the breadboard. Also make sure all elements are at the same height. Look from the side and use the height indicator.

Experimental phase

1. Build the monochromator setup. Use the Hg lamp and the 600 lines per mm diffraction grating. Start with the screen. Hold enough distance to be able to replace the screen by the camera. Control the collimation and assure yourself that the screen is in the focal point of f_2.

2. Rotate the diffraction grating such that $i(= \alpha + \phi) = 0$. (use autocollimation, ask the assistant). Read the angle from the angle scale. This the zero point. Pick a point on the screen and rotate the diffraction grating such that the 0-th order falls exactly on that point. How do you recognise the 0-th order here? Read the angle scale and determine α. Do not forget to estimate the uncertainty in the angle. Play around with the width of the slit.

3. Now rotate the diffraction grating systematically and notate the angle for each spectral line that falls on the chosen point on the screen. Do this for as many orders as possible. Make a table with the headings: colour, order, reading, ϕ, λ, u(λ) and the result from preparation question 10. How do you get ϕ? Use Python to help you fill in the last three columns. Also note that a set of colours represents one order.

4. Make a plot in which you plot the calculated wavelength on the vertical axis with its respected uncertainty against the order m on the horizontal axis. Plot the literary values as horizontal dotted lines in the same figure.

5. Calculate the weighted average of the wavelength for each measured spectral line and calculate the uncertainty in this average value. Take as weight $w = 1/(u(\lambda))^2$. Use the given formulas from the Appendix for this. Does the result contradict the literary value?

6. Next, study the splitting of the yellow Hg lines. Replace the screen by the camera. Portray an order of the yellow lines on the camera (note: The colour can be different on the screen. Pay close attention.). (Think about the width of the slit and the use of the filter). Save the result for the report. (Use the save icon at the middle top.) Determine the distance between the lines on the screen in pixels. Calculate the difference in the angle of reflection Δu from this. Check if your value of u is still current. Calculate the difference in the wavelength $\Delta\lambda$ with eqn. (96). What is the uncertainty $u(\Delta\lambda)$?

7. Replace the Hg-lamp with the Na-lamp and use the camera instead of the screen. Measure the angels of the visible spectral lines. What is the angle that corresponds with the 0^{th} order? Specify the orders to the measured angles and calculate for each order the wavelength and its uncertainty. Again try to find splitting for all lines (except the 0^{th} order of course). Do the individual results agree with the literature values? Calculate the weighted average and its uncertainty.

Evaluation phase Determine the difference in wavelength of the two Na lines. Try to do this for all the orders. What is the associated uncertainty? At what order do you obtain the best results? Are the results in agreement with the literature values?

What is the smallest difference in wavelength that you can determine using this set up? What is the resolving power of this set up? What is the expected, theoretical value for the resolving power? What can be done to improve the results?

Finally Suggestions for the improvement of the experiment and the manual or ideas about what you would like to research with the used instruments, please mail to: c.f.j.pols@tudelft.nl

Appendix

For the calculation of the weighted average you need to first calculate the weights, this is done in the following way:

$$w_i = \frac{1}{u(\lambda_i)^2} \tag{99}$$

This means that datapoints with a high uncertainty get a lower weight. And datapoints with a low uncertainty get a high weight (thus are more important). To find the final value for λ we use the following equation:

$$\overline{\lambda} = \frac{\sum_i^N w_i \lambda_i}{\sum_i^N w_i} \tag{100}$$

The final error/uncertainty in λ is equal to:

$$u(\overline{\lambda}) = \sqrt{\frac{1}{\sum_i^N w_i}} \tag{101}$$

0.5.5 Falling waterdroplet

Goal

What is the velocity of falling water droplets, how strong is the drag force exerted on these droplets by the surrounding air and what is their shape? These questions are addressed in this assignment. The primary goal of the experiment is to determine the drag coefficient, which is a measure for the drag force experienced by a falling water droplet in air.

Background

Introduction Newton's second law provides the equation of motion for a falling water droplet in air:

$$m\frac{dv}{dt} = mg - F_D - F_B \tag{102}$$

Here m is the mass of the droplet, v the velocity of the droplet, g the gravitational acceleration, F_D the drag force and F_B the buoyant force. The velocity is defined relative to the stationary air, taking downward as positive. The drag force is the result of friction of the droplet with the air. The buoyant force is described by Archimedes' principle.

In the rest of this manual F_B is neglected. It can be expected that F_D depends on the velocity of the droplet. In this experiment you will determine F_D for a range of occurring velocities. To this end, you will perform two experiments:

1. with a drop test, you will determine the position-time relation of the falling droplet
2. with a visualization experiment you will take photographs of a floating droplet to determine its shape.

Falling droplet and fluid mechanics It is generally assumed that the drag force F_D is proportional to the velocity squared of the moving particle, in this case a water droplet:

$$F_D = \frac{1}{2}C_D A_\perp \rho_{air} v^2 \tag{103}$$

Here C_D is the drag coefficient, $A_\perp$ the cross-sectional area of the droplet perpendicular to the direction of the velocity and ρ_{air} the air density. The value of C_D, in turn, can also depend on the velocity. Furthermore, C_D depends on the shape of the droplet.

In the field of transport phenomena, one often uses characteristic numbers (in Dutch: kentallen). These are dimensionless parameters that describe a system in a general way. The Reynolds number, Re, is the most important characteristic number in fluid mechanics. It is not only used to determine whether a flow is laminar or turbulent, but also to realize similarity between two different flows. The Reynolds number is defined as:

$$\mathrm{Re} = \frac{\rho v d}{\mu}, \tag{104}$$

with d is a characteristic dimension, here the largest diameter of the droplet perpendicular to the velocity, and μ the dynamic viscosity of air. Figure 30 gives the dependence of C_D on the Re-number, for a spherical and a cylindrical particle.

For a low Re-number (Re < 1) Stokes' law applies:

$$F_D = 3\pi\mu d v \tag{105}$$

On the basis of the above, it follows for Re < 1:

$$C_D = \frac{24}{\mathrm{Re}} \tag{106}$$

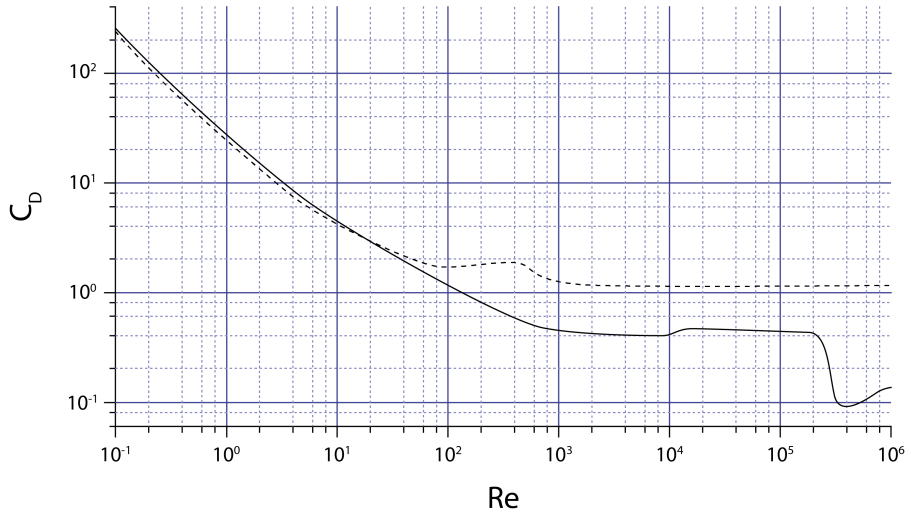


Figure 30: Drag coefficient C_D as a function of the Re-number, for a sphere (—-) and a disk (- - -).

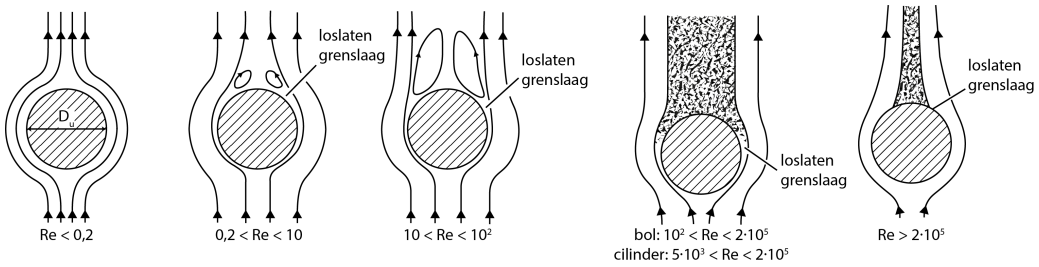


Figure 31: Flow pattern around a sphere as a function of the Re-number.

With increasing Re-number, vortices arise at the rear of the particle (see Figure 31). A wake is created; the boundary layer is "released" from the particle as a result of the inertia of the surrounding medium, that prefers to move straight on. N.b.: This line of reasoning assumes a medium flowing around the particle, but that situation turns out to be equivalent to the situation of the particle moving through the medium, as in the present experiment. For larger Re-numbers, the vortices increase in size and the location where the boundary layer is released moves to the front side of the particle. For even larger Re-numbers (10^3 - $2 \cdot 10^5$, Newton regime) the wake becomes irregular and turbulent. Vortices are then carried along by the flow and new vortices are generated at the rear of the particle. Eventually, for very large Re-number, the boundary layer becomes turbulent and the location where the boundary layer is released moves back to the rear of the particle. In the transition range from the Stokes to the Newton regime, we can use the Schiller-Naumann relation:

$$C_D = \frac{24}{\mathrm{Re}}(1 + 0.15\mathrm{Re}^{0.687}) \tag{107}$$

For objects moving with high velocity through a thin medium ($10^3 < \mathrm{Re} < 2 \times 10^5$), Figure 30 shows that C_D is approximately constant.

Solution of the equation of motion Using (103) and $\beta = C_D A\rho/2$, (102) can be rewritten as follows:

$$m\frac{dv}{dt} = mg - \beta v^2 \tag{108}$$

Here F_B is neglected, as indicated earlier. This way of writing of the equation suggests that the parameter β is constant. In that case the drag force would be proportional to the velocity squared. In general, however, this is not true, since C_D yet depends on the Re-number (see Figure 30). Equation (108) is a non-linear (and therefore difficult to solve) differential equation in the velocity of the droplet. To illustrate the velocity behaviour, we have plotted in Figure 32 the exact solution of Eq. (108) for a

situation representative of this experiment. Herein, it has been taken into account that β depends on the velocity, through C_D. This solution has been obtained using numerical methods. In the Figure it can be seen that the droplet after approximately four seconds almost reaches its saturation velocity of 12.9 m/s. This exactly equals the value $v_{sat} = \sqrt{(mg/\beta)}$ following from Eq. (108) for $C_D = 0.4$. As will become clear, the maximum drop time in the experiment is about 0.6 s. Thus, the saturation velocity is far from reached.

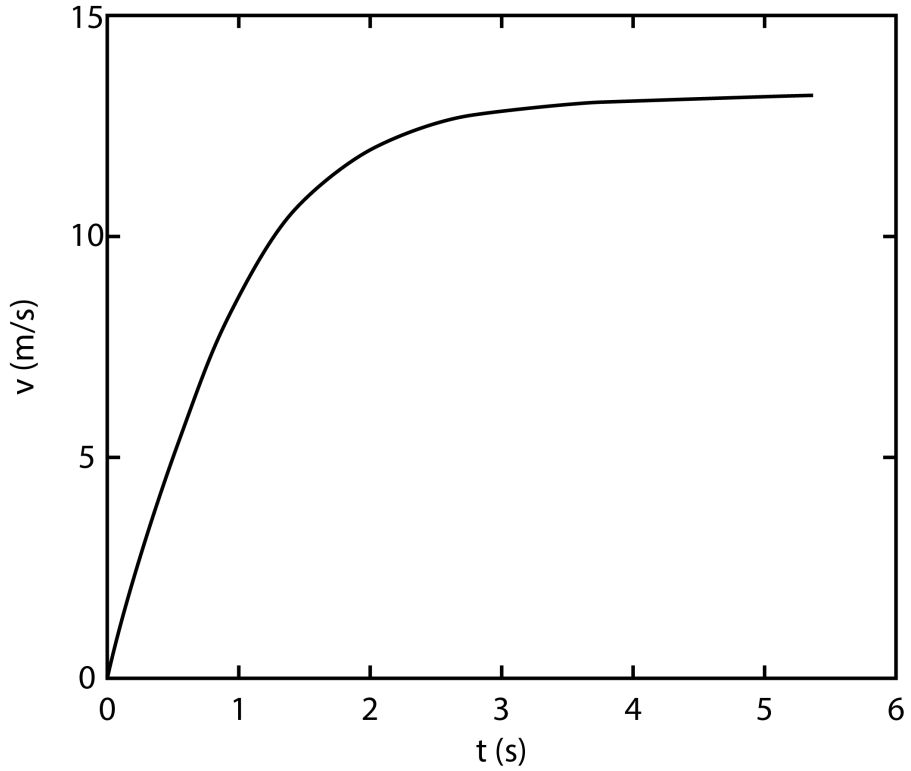


Figure 32: The velocity of a falling water droplet in air as a function of the drop time. The diameter of the droplet, taken here as spherical, is 6 mm. After about 4 s, the saturation velocity is almost reached. The curve is the exact solution, obtained by numeric integration of Eq. (108).

An approximate solution of Eq. (108), satisfying the present goal, can be obtained in the following globally sketched way. Integration of Eq. (108) gives

$$v(t) = gt - \frac{1}{m}\int_0^t \beta(v(t'))v(t')^2 \mathrm{d}t. \tag{109}$$

This did not help much yet, as we have expressed $v(t)$ as an integral of the function $v(t)$, and that is just the quantity we are looking for. However, by repeated substitution of the expression for $v(t)$ "in itself" and by assuming that β is time-independent, the following approximation for the drop velocity $v(t)$ is found:

$$v(t) = v_0 + gt - \frac{1}{3}\frac{\beta}{m}g^2t^3 + \frac{2}{15}\left(\frac{\beta}{m}\right)^2 g^3t^5 + ... \tag{110}$$

The four terms in the expression for $v(t)$ are the initial velocity, the free drop term, the first order correction and the second order correction, respectively. From further theory, it follows that for this experiment the first order correction is sufficient, provided $t < 0.7$ s. In that case, the second order correction is negligibly small compared to the first order correction. It turns out that the this condition for the drop time is met in the experiment.

In Figure 33 curves for the drop velocity and the drop distance have been plotted, indicating the error with respect to the exact solution if only the first order term in Eq. (110) is included. In this, for the first order term $C_D = 0.4$ has been taken, the constant value for a spherical droplet also used for Figure 30. Given the maximum drop time of about 0.6 s, the error is less than 1% for the drop

velocity and less that 0.3% for the drop distance. In this experiment we neglect these errors. Note that the larger values of C_D, occurring for the smaller velocities at the start of the drop trajectory, apparently do not play a significant role.

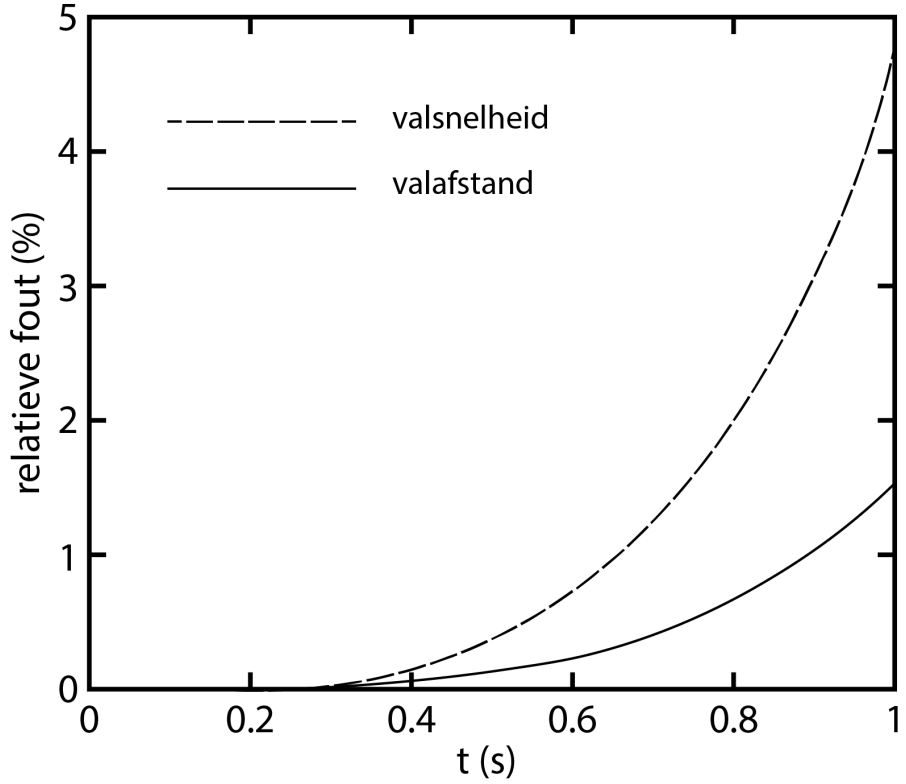


Figure 33: The relative error in the drop velocity and the drop distance of the water droplet with respect to the exact result, when only taking into account the effect of the drag force to first order. The diameter of the spherical droplet is 6 mm for both curves.

Experimental approach Integration of Eq. (110) with neglect of the second order correction leads to

$$s(t) = s_0 + v_0 t + \frac{1}{2}gt^2 - \frac{\beta g^2}{12m}t^4 \tag{111}$$

where s_0 is the travelled distance at $t = 0$. When we design the experiment such that at $t = 0$ both the traveled distance and the velocity are zero, then Eq. (111) reduces to

$$s(t) - \frac{1}{2}gt^2 = -\frac{\beta g^2}{12m}t^4 \tag{112}$$

In words, this says that the drop distance in air at any time is reduced with respect to the drop distance of a free fall by an amount proportional to the drop time to the fourth power. The proportionality constant includes the parameter β, which in turn is proportional to C_D. This result immediately suggests the experimental approach: for various drop distances s_i measure the corresponding drop times t_i ($i = 1, 2, 3, ..., n$) and put the measured data points in a plot with the quantity $\Delta = s(t) - gt^2/2$ on the vertical axis and t^4 on the horizontal axis. A linear fit to the data points then has the slope $-\beta g^2/(12m)$. Since we have $\beta = \rho_{air} A_\perp C_D/2$, C_D can be determined from the slope, provided that m and $A_\perp$ are known. The mass m is determined through weighing, while the perpendicular area $A_\perp$ is determined in a visualization experiment of a floating water droplet (see the next paragraph).

Figure 34 gives a sketch of the setup. Using a pipette, you will make a droplet, which is released once it is big enough. With two optical detectors you will measure the time t it takes for the droplet

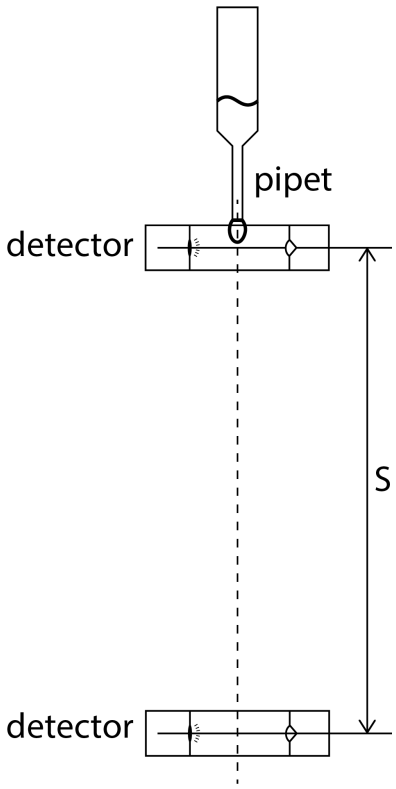


Figure 34: Schematic of the drop test. The droplet falls a distance s.

to fall s meter. The drop distance s equals the distance between the two detectors. The first detector is located at $s = 0$, as close as possible to the point where the droplet is released from the pipette. The position of the second detector can be varied. The detectors consist of a laser and a photodiode, together forming a light gate. When a droplet passes a light gate, the laser beam is briefly interrupted, causing a pulse in the signal of the photodiode. A counter connected to the light gates measures the time difference between the two pulses, i.e. the drop time t.

Visualization experiment When you ask somebody to draw a droplet, it is very likely that the person asked will draw the shape of a tear: thick at the bottom and converging to a tip at the top. A falling water droplet, however, is rather flat than elongated. A falling droplet resembles an "M&M". To establish the shape of a droplet that is subject to air drag and to determine its cross-sectional area $A_\perp$, and from this the drag coefficient C_D, you will take photographs of the droplet. To enable this, you will float the droplet by placing it in an upward air flow in the setup depicted in Figure 35.

Position the pipette a few centimeters above the wire mesh, with the blower set to about 20 Volt, and create a droplet. Try to keep the floating droplet stable in the air flow long enough to take sharp photographs. This requires some trial and error and optimization with (among other things) the blower voltage and the pipette's position relative to the mesh. A transparent cylinder is available to guide the air flow, if necessary. Dry the mesh with a tissue if a droplet has fallen onto it; otherwise the setup is not ready for the next attempt.

While floating a droplet, it is difficult to press the camera button and at the same time keep the camera setup stable. Therefore, use the camera's remote control to take the photographs. Take quite a number of photographs and view them on your laptop or on a computer of the laboratory course.

Determine the cross section and volume of the droplet. Determine the droplet's absolute dimensions using an object of known size in the photograph. The floating droplet is heavier that the falling droplet, but we suppose that its shape is the same. Beware: after the experiment, all data on the memory card of the camera will be deleted. Therefore, copy photos to a memory stick or to your laptop. Otherwise, you will not have a photograph for your report.

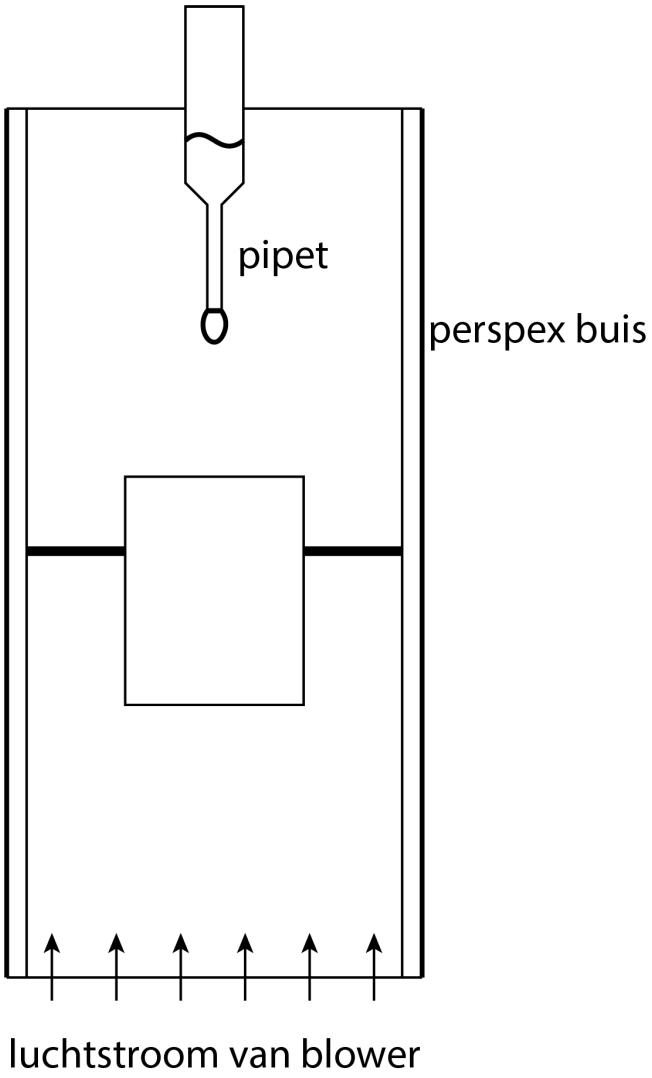


Figure 35: Setup to float a droplet.

Assignments

For the formulation of the problem see chapter 1, GOAL, and chapter 2, section 2.1. A separate afternoon has been scheduled for the problem analysis and setting up the measurement plan.

Do the following assignments.

Set up a measurement plan and discuss this with the teaching assistant (TA). After approval by the TA, execute the plan on the scheduled afternoon.

Determine the drop time for at least 10 positions of the lower detector. The lower detector cannot be positioned above the edge of the table. Handle the setup very carefully and calmly, to prevent that you have to re-align it.

Use Python for the data processing. Ask the TA for help, if necessary.

Write a report on the experiment, according to the guidelines of the Physics Laboratory Course. The deadline for handing in the report and the lab journal is known at the administration of the Physics Laboratory, room A001.

Addendum taking a picture

To take a good picture of the droplet, light is required. It must be really bright as the droplet is still moving and you get only a good picture if the shutter speed is really high. To obtain a useful picture without wasting much time, we should make the decisions rather than the camera. The camera is therefore set to *manual* with a shutter speed of 1/320 s, a diafragma of f9, and an ISO value of 400. The flashlight is turned off.

Setting the camera to manual also requires a manual focus. Choose the largest focal distance possible (55mm) and focus on the tip of the needle of the pipet. Move the camera as close as possible towards the setup. Focussing should still be possible.

By taking the picture from a small angle, you will have a light spot from one of the lamps in the background. This gives an excellent contrast to determine the circumference of the drop. Keeping the shutter release button pressed while letting the pipette drip gently gives the best chance of a usable image. Note that the drop may come towards or float away from the camera. If so, the pipette needle is in focus sharp. However, the water droplet will be out of focus. This will contribute to the measurement uncertainty.

0.5.6 Millikan

0.6 Test page

to sort things out:

$$a * x + b$$
$$z = f(x)$$

Bibliography

R. Wolfson and others. *Essential university physics*, volume 1. Addison-Wesley San Francisco, CA, 2007.