

# NOFY080 lecture notes

Emil Varga, Marek Talíř

November 13, 2025

## Contents

<b>1 Setting up the environment</b>	<b>4</b>
1.1 Running Python code . . . . .	5
1.2 A quick note on text editors . . . . .	6
1.2.1 Jupyter Notebooks . . . . .	6
<b>2 Python basics</b>	<b>8</b>
2.1 Variables and types . . . . .	8
2.2 User input . . . . .	8
2.3 Printing and strings . . . . .	8
2.4 Functions . . . . .	9
2.5 Compound types . . . . .	10
2.5.1 Lists and tuples . . . . .	10
2.5.2 Dictionaries . . . . .	11
2.6 Control flow . . . . .	11
2.6.1 if-elif-else . . . . .	11
2.6.2 match . . . . .	11
2.7 Loops . . . . .	12
2.8 Using external modules . . . . .	13
2.9 Object-oriented programming (OOP) . . . . .	14
<b>3 NumPy, SciPy, and matplotlib</b>	<b>16</b>
3.1 numpy . . . . .	16
3.1.1 Basic measures and operations with the arrays . . . . .	17
3.1.2 File I/O . . . . .	18
3.2 matplotlib . . . . .	19
3.2.1 Review of basic plotting commands . . . . .	23
<b>4 Least squares fitting and interpolation</b>	<b>24</b>
4.1 Polynomial fitting. . . . .	24
4.2 Non-linear curve fitting . . . . .	25
<b>5 Digital signal processing</b>	<b>30</b>
5.1 Digital representation of a continuous signal . . . . .	30
5.2 Spectral analysis . . . . .	31
5.3 Filters . . . . .	39
5.4 Interpolation and smoothing. . . . .	42
<b>6 Communication with instruments</b>	<b>43</b>
6.1 Context Management Protocol . . . . .	45
6.2 Troubleshooting common issues . . . . .	47

<b>7 Parallel Execution</b>	<b>48</b>
7.1 Multithreading . . . . .	48
7.2 Multiprocessing . . . . .	52
7.3 Inter-process Communication . . . . .	54
7.3.1 Real use case in a lab . . . . .	58
<b>8 Solutions of differential equations</b>	<b>59</b>
8.1 Initial value problems . . . . .	59
8.2 Boundary value problems . . . . .	62
8.3 GPU acceleration . . . . .	64
8.3.1 Setting up the environment . . . . .	65
8.3.2 Testing the installation . . . . .	65
8.3.3 GPU Kernels . . . . .	65
8.3.4 Using the kernels . . . . .	67
<b>9 What didn't fit</b>	<b>69</b>
9.1 PID algorithm . . . . .	69
9.2 Symbolic mathematics . . . . .	69
<b>A Linear Harmonic Oscillator</b>	<b>70</b>
<b>B Setting up communication with instruments</b>	<b>70</b>
B.1 Supported commands . . . . .	71
B.2 Hacking the firmware . . . . .	71
<b>C Basics of git version control system</b>	<b>71</b>
<b>D Basics of Linux</b>	<b>71</b>
D.1 Filesystem structure . . . . .	71
D.2 Moving files between lab and personal computer . . . . .	72
D.3 Command line interface . . . . .	72
D.3.1 Users and permissions . . . . .	75
D.4 Accessing remote servers using CLI . . . . .	76

## Python Language Intermezzos

Writing modules. . . . .	15
Package installation – pip . . . . .	16
NANs and INFs. . . . .	17
Paths . . . . .	18
with statement (basic) . . . . .	18
Exceptions and error handling . . . . .	28
Storing data and metadata in numpy binary files. . . . .	37
3D plotting . . . . .	62

## Code Snippets

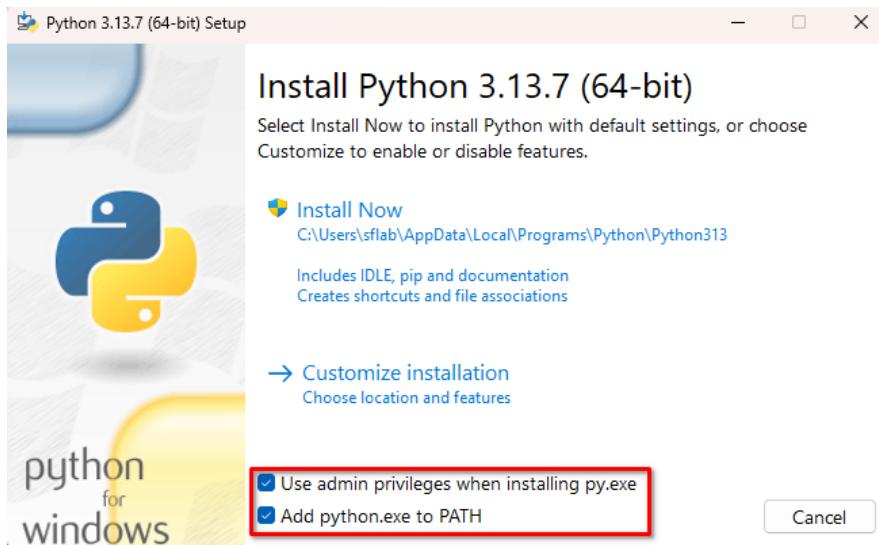
1 Defining variables . . . . .	8
2 Function definition . . . . .	9
3 Optional and keyword arguments. . . . .	10
4 Higher order functions. . . . .	10
5 Lambda functions. . . . .	10

6	List comprehension. . . . .	13
7	Array creation. . . . .	16
8	Basic plotting. . . . .	19
9	Complete plotting example. . . . .	19
10	Multiple axes in single figure. . . . .	21
11	Two-dimensional plotting example. . . . .	22
12	Polynomial fitting . . . . .	24
13	Nonlinear curve fitting. . . . .	25
14	Minimization of a scalar function of multiple parameters. . . . .	26
15	Estimating fit parameter errors using bootstrap. The data $(x_i y_i)$ are created in the same way as in Lst. 13. . . . .	27
16	Effect of sampling rate . . . . .	30
17	Fourier series of a square pulse. . . . .	31
18	Calculation of Fourier transform and power spectral density. . . . .	34
19	Fourier coefficient of a square wave calculated using FFT . . . . .	35
20	Fourier transform of a decaying sine wave. . . . .	36
21	RC filter response . . . . .	39
22	Using IIR filter. . . . .	41
	./example_code/visa_intro.py . . . . .	43
	./example_code/context_management_file.py . . . . .	46
	./example_code/threads_baisc.py . . . . .	48
	./example_code/threads_oop.py . . . . .	48
	./example_code/threads_locks.py . . . . .	49
	./example_code/threads_events.py . . . . .	50
	./example_code/threads_queues_minimal.py . . . . .	50
	./example_code/threads_queues.py . . . . .	51
	./example_code/multiprocessing_process.py . . . . .	52
	./example_code/process_pools_minimal.py . . . . .	53
	./example_code/process_pools.py . . . . .	53
	./example_code/ipc_server.py . . . . .	55
	./example_code/ipc_client.py . . . . .	56
	./example_code/instrument_server.py . . . . .	56
	./example_code/ivp_ballistic.py . . . . .	60
	./example_code/ivp_ballistic.py . . . . .	60
	./example_code/ivp_ballistic.py . . . . .	60
	./example_code/ivp_ballistic.py . . . . .	60
	./example_code/bvp_schrodinger.py . . . . .	62
	./example_code/bvp_schrodinger.py . . . . .	63
	./example_code/bvp_schrodinger.py . . . . .	63
	./example_code/bvp_schrodinger.py . . . . .	64
	./example_code/bvp_schrodinger.py . . . . .	64
	./example_code/bvp_schrodinger.py . . . . .	64
	./example_code/taichi/gravity.py . . . . .	66
	./example_code/taichi/gravity.py . . . . .	66
	./example_code/taichi/gravity.py . . . . .	67
	./example_code/taichi/gravity.py . . . . .	67
	./example_code/list_resources.py . . . . .	70
	./example_code/idn.py . . . . .	70

# 1 Setting up the environment

If you do not have Python installed, you can download it from <https://www.python.org/downloads/> (specifically the newest version for Windows here).

After running the installer, enable the "Add python.exe to PATH" option (and I also recommend enabling administrator privileges), see screenshot



Click through the installation, open Windows PowerShell and verify the functionality with the following two commands.

```
python --version
```

and

```
pip --version
```

Both commands should print the program version and not any error message. If you are using Linux, you probably already have Python installed. If not, use your distribution's package manager, e.g., on Fedora:

```
sudo dnf install python3
```

Typical programming in Python heavily relies on external packages, which can have different versions and dependencies. To avoid conflicts between different projects, it is recommended to use a virtual environment for each project. We will be using such a virtual environment in this course to ensure a consistent development environment for everyone. Virtual environments are the recommended way to manage dependencies in Python projects.

To manage these virtual environments we will be using the open-source tool uv, available from <https://github.com/astral-sh/uv>. If you have any version of Python already installed, you can install uv by simply running

```
1 pip install uv
```

Using pip is recommended, but you can also use the stand-alone installers on <https://github.com/astral-sh/uv>.

To test the installation, run in your command line

```
1 uv --version
```

and something similar to uv 0.5.30 should appear.

Now we will create a project called NOFY080\_2025 that will hold all the files related to this course, run

```
1 uv init NOFY080_2025
```

which will create a new directory with the same name, with a simple "Hello World" Python file, some additional files that uv uses to track dependencies of your project, and set up a git repository in it (you don't have to use git in this course, but those interested can learn some basics in Appendix C).

Now change to the project directory

```
1 cd NOFY080_2025
```

and add packages that we will need throughout this course

```
1 uv add numpy scipy matplotlib
```

which will create the virtual environment and install the specified packages (a directory .venv should appear in the project directory).

If you wanted to create a virtual environment without installing any packages, you could use the command

```
1 uv venv <name of the virtual environment>
```

To activate the virtual environment, run the following command on Linux or Mac OS

```
1 source .venv/bin/activate
```

and on Windows

```
1 .venv\Scripts\activate
```

Now, when you run `python`, the virtual environment version will be run, rather than your system version. To deactivate the virtual environment, simply run `deactivate`.

## 1.1 Running Python code

Python is an interpreted language that does not need to be compiled before it is run. We save Python code in text files with the `.py` file extension or in so-called Jupyter notebooks (file extension `.ipynb`).

To test our installation, create a new Python file (a text file with the suffix `.py`) `test.py` in our project directory with the following content

```
1 import matplotlib.pyplot as plt
2
3 plt.plot([1, 2, 3], [4, 5, 6], '-o')
4 plt.show()
```

To run it, you can either manually activate the virtual environment and run it using `python test.py` or use `uv run test.py`. A simple graph should appear. In the rest of these lecture notes, whenever we invoke the `python` command from the command line, we mean the one in the virtual environment.

To run Python code directly without saving it to a file, open a suitable terminal and type in `python` and a prompt like

```
1      >>>
```

will appear. This is the Read-Evaluate-Print-Loop (REPL); any code entered will be run and the result printed on the screen. To exit, enter `quit()`. To execute a file `myfile.py`, first navigate to its directory and then run it simply with `python myfile.py` (note that if you have your virtual environment activated, the actual Python script does not have to be located in your project directory).

**Exercise 1.1.** Create a text file `hello.py` which contains a single line

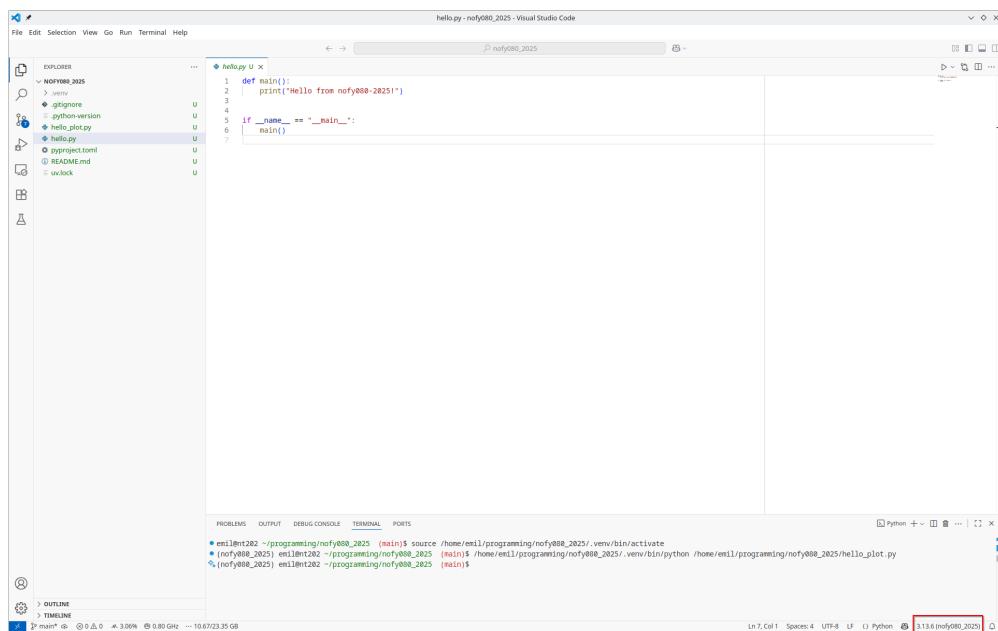
```
1     print("Hello world!")
```

and then run this file.

Sometimes we do not want the Python interpreter to quit immediately after finishing running our program (e.g., we want to inspect variables created during the program run); this can be done using `python -i file.py`, where `-i` stands for *interactive*. Alternatively, a more user-friendly (color-coded syntax, auto-completion, etc.) version of the interactive Python REPL can be invoked using `ipython` (or `ipython3`, depending on your installation), which can also be used to interactively run files using `ipython3 -i myfile.py`.

## 1.2 A quick note on text editors

Any text editor can be used to write Python code, including the default Windows Notepad. However, it is beneficial to your mental health to use at least something with syntax highlighting, such as Notepad++, or a more feature-rich editor such as VS Code or Spyder where you can run your file without switching to a terminal. In this course, I recommend using VS Code with the Python extension installed, because it makes working with virtual environments easier. Opening a directory in VS Code will automatically detect the virtual environment and use it when the file is run. The name of the virtual environment currently used is indicated in the bottom right corner (highlighted in red):



Working with environments in Spyder is somewhat more involved; see the guide here.

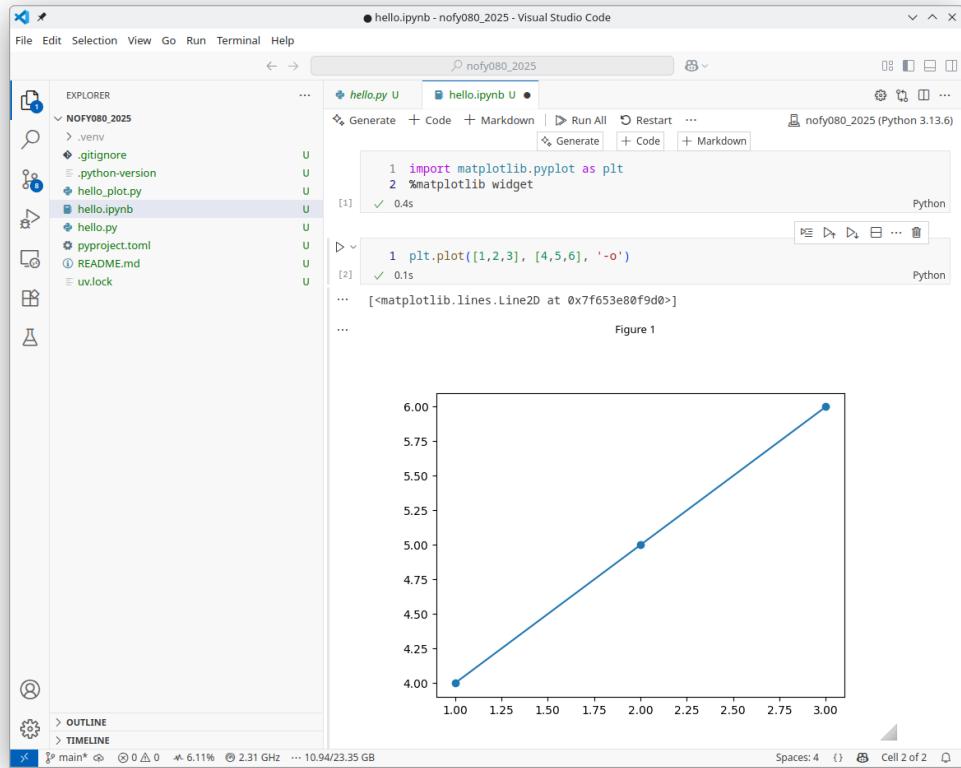
### 1.2.1 Jupyter Notebooks

Jupyter notebooks (run using `jupyter-notebook` in the command line) provide an easy-to-use interactive environment with an interface running in a web browser. They are good for trying things out<sup>1</sup>. Jupyter notebooks also work well with virtual environments and VS Code. Navigate to your project directory and run

```
1 uv pip install jupyter ipympl
```

<sup>1</sup>In this course we will be (mostly) using scripts due to easier splitting into modules and fewer issues with parallel programming, which we will encounter later on.

The `ipymp1` package enables the use of interactive Matplotlib plots in Jupyter notebooks. Next, also install the `jupyter` extension in VS Code. Now simply create a file with the extension `.ipynb` and you can use it within VS Code as you would in the browser. On the first run, you will be asked to pick the virtual environment in which the notebook should run; the one we created should be one of the options if the notebook is saved in the project directory (same directory as the `.venv` directory).



## 2 Python basics

### 2.1 Variables and types

Variables are created by assigning a value to an unused variable name, declarations, such as in C/C++ or Pascal are not necessary

```
1 my_integer = 5
2 my_float = 3.14
3 my_string = "double quote string"
4 my_string2 = 'single quote string'
5 my_string3 = "you can use 'single' quote in double quote string"
6 my_bool = False
```

Listing 1: Defining variables

Note that variables generally do not have a fixed type (i.e., an integer or a string), and doing something like `my_integer = "a string"` will not result in type error. Python itself does not care about what type the variable is as long as the operations we do with it are supported. However, if we need to know the type of a variable `v` we can find it using `type(v)`.

Basic arithmetic operations `+, -, *, /` work as expected on numbers. Exponentiation is `**`. Note, however, that `/` automatically *promotes* integers to float. For integer division, we can use `//` and for the remainder `%`. Arithmetic operations are also defined for some non-number types where they "make sense", i.e., strings can be concatenated using `+`. Comparison operators are `==` for equality (**not to be confused with `=`, which is the assignment of a value to a variable**), `!=` for inequality and `<, <=, >, >=` for ordering. Boolean operators are `and`, `or`, `not`.

There are also **bitwise** operators `&, |, ^, ~` (bitwise AND, OR, NOT and XOR, respectively). These operate on individual bits of a number. Careful about the confusion between exponentiation `**` and bitwise exclusive or `^`.

Complex numbers are supported in Python. The syntax to specify a complex number is, e.g., `3 + 5j`, which translates to mathematical  $3 + 5i$ . Specifically, the imaginary unit is `1j`. Notice that the letter `j` follows the number immediately.

**Exercise 2.1.** Try out the basic arithmetic in REPL. Try multiplying a string by an integer. What happens when you try dividing?

A summary of basic built-in operators

<code>+, -, *, /, **</code>	basic arithmetic, exponentiation
<code>and, or, not</code>	boolean operations
<code>&lt;, &lt;=, ==, &gt;, &gt;=</code>	comparison
<code>&amp;,  , ^, ~</code>	bitwise operations, bitwise AND, OR, NOT and XOR (exclusive or)

### 2.2 User input

We will be mostly writing non-interactive scripts, however, to get basic text-based input from the user we can the `input()` function, **which always returns a string**. We have to convert it to a number (using `int()` or `float()`) ourself if that is what we expect the user to enter, e.g.,

```
1 age = int(input("How old are you? "))
2 print("You will be 100 in", 100-age, "years.")
```

### 2.3 Printing and strings

We have already met function `print()`, which prints a text representation of whatever objects pasted to it to standard output. It accepts one or more arguments.

To turn any value to a string we can use the function `str()`. Special characters, such as new line or tab, need to be *escaped*, e.g., `\n` and `\t`, respectively. Backslashes also need to be escaped `\\\`. To stop python from trying to interpret backslashes as special characters (useful, e.g., for specifying file paths on Windows) we can use raw strings, e.g.,

```
1 print("s\tri\nng") #\t and \n are interpreted as tab and new line
2 print(r"s\tri\nng") # raw string, note the r immediately before "
```

For crafting more complicated way of printing our variables we can use *format* strings (or f-strings) by enclosing the variable we wanted inserted into the string in {}, e.g.,

```
1 vari = 5
2 varf = 3.14
3 fstr = f>An interger value {vari:05d}, a float value {varf:010.6f}, exponential
   form {varf:g}"
```

The variable substitution is specified as `{expression:format_specifier}`, the format specifiers are optional. Here the format specifier `05d` means a decimal integer with total width of at least 5 characters and `010.6f` means floating point number with 6 digits after the decimal period and total width of at least 10 characters.

For a full specification of possible format specifiers see the official documentation.

## 2.4 Functions

Functions are defined using the `def` keyword. Functions may take argument and return a value.

```
1 def my_function(my_string):
2     print("Someone wants to know how long \"{}\" + my_string + \"{}\" is!")
3     return len(my_string) #returns the length of a my_string
```

Note the indentation, which is how blocks of code are identified in Python, i.e. there is no `begin`, `end` as in Pascal or {} as in C/C++.

Multiple arguments are separated with a comma and arguments can have a default value, e.g. the argument `c` will be 1 unless specified

```
1 def calculate(a, b, c=1):
2     """
3         A function that calculates c*(a + b)
4
5         Parameters:
6         -----
7         a, b, c: numbers
8             arguments of the calculation
9
10        Returns:
11        -----
12        result: number
13            The result of calculating c*(a + b)
14        """
15        return c*(a + b)
16
17 >>> calculate(2, 3)
18 5
19 >>> calculate(2, 3, 4)
20 20
```

Listing 2: Function definition

Here the long string delimited by triple quotes `"""` is the so-called documentation string of the function (or **docstring** for short). Triple quotes can be used anywhere and indicate only that the string spans multiple lines. The docstring then can be accessed using the `help()` function or by text editors. The formatting used in the above example is commonly understood by IDEs, e.g., Spyder will display it nicely formatted in the Help pane (upper right corner of the window by default).

Arguments with default values must come after the required arguments. With multiple optional arguments we can call the function using *keyword arguments*:

```

1 def calculate2(a, b, c=1, d=1):
2     return c*(a + b)**d
3
4 >>> calculate(2, 3, d=2) # leaves c default, but specifies d
5 25

```

Listing 3: Optional and keyword arguments.

Functions can be passed around in variables and arguments, specifically, functions can accept other functions as arguments, e.g.,

```

1 def add1(x):
2     return x + 1
3
4 def change_number(x, func):
5     return func(x)
6
7 >>> change_number(1, add1)
8 2

```

Listing 4: Higher order functions.

Short, simple functions (that fit on one line) can be defined in-line as so-called anonymous, or *lambda*, functions, e.g.

```

1 >>> change_number(1, lambda x: 2 + x)
2 3

```

Listing 5: Lambda functions.

Lambdas are especially useful for partially filling out argument lists of existing functions, e.g.

```

1 >>> change_number(1, lambda x: calculate2(x, 4, c=4, d=0.5))
2 8.94427190999916

```

## 2.5 Compound types

### 2.5.1 Lists and tuples

A **list** is a collection of any python objects defined using square brackets [],

```

1 my_list = [1, 2.17, "a string", ['another', 'list']]

```

The values in the list can be accessed by indexing the list **starting from 0**, e.g., `my_list[1] == 2.17`. Negative indices count from the back, i.e., `my_list[-1]` is the last element, `my_list[-2]` second-to-last etc.

Lists are an example of an *object*, objects have associated functions called methods, which are called with a dot syntax. Useful methods for lists are

- `append` and `pop`

```

1 >>> my_list = [1, 2.0, 'three']
2 >>> my_list.append('one more')
3 >>> my_list.pop() # removes the last element and returns it
4 "one more"

```

- `my_list.sort()`, sorts the list in-place if all elements of the list can be compared; to create a new list with sorted elements we can use `sorted(my_list)`
- `my_list.index('a')` finds the index of first occurrence of 'a' or returns an error

Apart from simple indexing, subsets of lists can be accessed using *slicing*, e.g.,

```
1  l = [1,2,3,4,5,6,7,8,9]
2  l[2:5] # [3,4,5]
3  l[-5:-2] # [5,7,9]
```

Generally the syntax is `my_list[start_idx:stop_idx:step]`, `start_idx` is inclusive, `stop_idx` is exclusive. All three are optional, start defaults to 0, stop defaults to -1 and step defaults to 1.

**Tuples** behave in many ways similarly to lists, with the exception that they are **immutable**, i.e., the tuple itself or the values in it cannot be changed. Indexing and slicing, however, work the same. Tuples are defined using ordinary brackets `()`. If the tuple contains only one value a comma needs to be included to distinguish it from a bracketed expression, i.e., `single_tuple = (1,)`.

A common use for tuples is to return multiple values from a function, i.e.,

```
1  def add_and_subtract(x, y):
2      return x+y, x-y #note that the parentheses () are not required in return
3
4  #here we unpack the return value into two variables
5  a, s = add_and_subtract(3, 2) # a==5, s==1
```

Note that we did not store the return value as a tuple but immediately split it into two separate variables. This is called **unpacking** and can be used for any compound type.

## 2.5.2 Dictionaries

Dictionaries are a collection of assignments from a key to a value of any type, often the keys are strings e.g.,

```
1  my_dict = {
2      'key1': value1,
3      'key2': value2,
4 }
```

which can then be accessed as `my_dict['key2']` etc.

## 2.6 Control flow

### 2.6.1 if-elif-else

The `if` statement takes a boolean expression and runs the code in its code block if the condition evaluates to `True`.

```
1  if condition:
2      do_if_true()
3  elif condition2: #optional
4      do_if_2true() #runs only if condition is False and condition2 is True
5  else: #optional
6      do_if_false() #runs only if both condition and condition2 are False
```

### 2.6.2 match

A more general matching of value to action is available via `match`<sup>2</sup>.

```
1  command = 'yell'
2  match command:
3      case 'talk':
4          print("hello")
5      case 'yell':
6          print("HELLO!!!")
7      case _: #default behavior
8          print("unknown command")
```

<sup>2</sup>Available since Python 3.10

`match` can be especially powerful with unpacking, where `match` tries to match the pattern of its argument as best as it can. This allows you to, for example, process commands and their arguments, e.g.,

```

1 def process_command(command):
2     match command:
3         case ('speak', value):
4             print(f"Hello {value}")
5         case 'yell':
6             print('HELLO!!')
7         case ('yell', value):
8             print(f"HELLO!! {value}")
9         case (command, value):
10            print(f"I'm supposed to \"{command}\" with \"{value}\"")

```

which behaves as

```

1 >>> process_command(('speak', 1))
2 Hello 1
3 >>> process_command('yell')
4 HELLO!!
5 >>> process_command(('yell', 2))
6 HELLO!! 2
7 >>> process_command('shout', 20)
8 I'm supposed to "shout" with "20"

```

Note, however, that if the argument to `match` does not match any provided pattern nothing happens, i.e., it acts similarly to `if` with a missing `else` clause:

```

1 >>> process_command('shout')
2 # no output

```

**Exercise 2.2.** Write a function that takes the coefficients  $a$ ,  $b$ , and  $c$  of a quadratic equation  $ax^2 + bx + c = 0$  as input and prints its solutions  $x_1$  and  $x_2$ . If no real solution exists, the function should print an error to the command line.

## 2.7 Loops

To loop for a given number of iteration we can use the `for` loops with a `range()` function. We can also loop over anything iterative, e.g., a list or dict.

```

1 for k in range(10):
2     print(k)
3
4 for value in my_list:
5     print(value)

```

The general form of `range()` is `range(start, stop, step)`, start is inclusive, stop is exclusive and step is by default 1. `range()` does not return a list but something called an *iterator*. We can turn it into a list using `list(range(start, stop, step))`.

If we want to iterate over a list and we need to work with both the index and the value we can use `enumerate`

```

1 for index, value in enumerate(my_list):
2     print(f"my_list[{index}] = {value}")

```

To loop while a condition is true we can use the `while` loop,

```

1 while condition:
2     do_stuff()

```

Execution of loops can be modified using `continue`, which skips the rest of the code in the iteration and goes to the next one or `break` which immediately ends the loop. A common usage is with infinite loops, e.g.,

```

1  while True: # this will never end
2      do_stuff()
3      if should_we_stop: # unless we kill it
4          break

```

Loops are often used to construct lists, which can be simplified using **list comprehension** such as

```
[expression(k) for k in iterable if condition(k)] # if condition is optional
```

For example,

```

1 [k**2 for k in range(5)] # list of squares for k < 5
2 [k**2 for k in range(5) if k % 2 == 0] # list of squares of even numbers

```

Listing 6: List comprehension.

**Exercise 2.3.** Write a program, which asks a user their name and replies "Hello <name>!".

1. Change the program such that if the user's name is "Andrej", the greeting changes to "Ciao Andrej!".
2. Separate the logic of constructing the greeting to a separate function.

**Exercise 2.4.** Write a program, which will accept arbitrary number of names from the command line and at the end prints them back in alphabetical order.

1. The user will enter beforehand the number of names they want to enter. *Hint: range()* and a **for** loop
2. The program will stop asking for new names once the user enters an empty string. *Hint: break*
3. If the user entered the name "Emil", it will be skipped in the final output. *Hint: continue*.

## 2.8 Using external modules

Python is famous for having a large number of available libraries to do pretty much everything under the sun.<sup>3</sup> To use these, we first need to import the *module* we need, e.g. to use functions that deal with time we can do

```

1 import time as t #import the time module and gives it a shorthand name t
2 print(t.time()) #we prepend the module name to the function to call it

```

If there is only a small number of functions we want to use from a given module we can import them specifically and don't have to use the module name to call them, e.g.

```

1 from module import func1, func2
2 func1()
3 func2()

```

**Exercise 2.5.** Write a program to calculate the Fibonacci number  $F_n$  using both loops and recursion. Measure the time to calculate the first 20  $F_n$  using both methods using `time.time()`. *Hint:  $F_k = F_{k-1} + F_{k-2}$ ,  $F_1 = F_2 = 1$*

---

<sup>3</sup>And we will learn how to create our own shortly.

**Exercise 2.6.** File `exercises/2.6.py` contains data about particles in an electric field  $E = 1 \text{ kV/m}$  which is oriented along the  $z$ -axis.

The data have the form of a list of dictionaries with the following structure:

```

1 [ 
2     {'charge': -1,
3      'id': 1,
4      'mass': 0.4113513116324291,
5      'position': [-1.5188809980735316e-06,
6                     6.639994580333568e-06,
7                     8.468849433621426e-06],
8      'velocity': [-0.016976782951560687,
9                     -0.43614541046795063,
10                    0.24858497024060777]},
11     {...}
12 ]

```

Positions and velocities are given in units of meters and meters per second as vectors in  $[x, y, z]$  space, mass  $m$  is in MeV and charge  $Q$  is in the units of elementary charge. The id field serves only for identification.

Find the id of the particle with the highest energy given by

$$E = -\frac{1}{2}EQz + \frac{1}{2}mv^2,$$

where  $z$  is the position and  $v$  is the total velocity.

Hint:  $m[\text{kg}] = m[\text{eV}] \frac{e}{c^2}$ , where  $e$  and  $c$  are elementary charge and speed of light in SI units. These are already defined in `scipy.constants`

```
1 from scipy.constants import elementary_charge, c
```

## 2.9 Object-oriented programming (OOP)

OOP is a programming technique for associating data with functions and separating implementation details of partial problems from the rest of the code. If used correctly, it can help with writing easy-to-read, extensible and reusable code.

In OOP language, objects are instances of classes. Everything in Python is an object. For example, the number 5 is the instance of a class `int`. In modern Python (i.e., version 3 and higher) the concepts type and class have the same meaning. The built-in function `type()` returns the object's class (or type), e.g.,

```

1 >>> type(2)
2 <class 'int'>

```

To create new classes we use the `class` keyword, e.g., to create a class that represents differences

```

1 class Difference:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __str__(self):
7         return f"{self.x} - {self.y}"
8
9     def value(self):
10        return self.x - self.y

```

Functions defined inside classes are called **methods**. The first argument, conventionally called `self`, refers to the object that is calling the method. `__init__` is a special method that creates the object,

`__str__` is a special function that should create a readable textual representation of the object (used with `print`).

Using classes and objects is straightforward, e.g.

```
1 d1 = Difference(3, 5) # __self__( ) is called with x=3, y=5 and self refers to d1
2 d2 = Difference(30, 50)
3
4 print(d1.value()) # -2
5 print(d2) # "30 - 50"
```

Several other special method names exist (see documentation for more), e.g. `__add__(self, other)` which is called for `x + y` with `x` being `self` and `y` being `other`. Similarly there are `__sub__`, `__mul__` and `__truediv__` for `-`, `*`, `/`, respectively.

**Exercise 2.7.** Implement a `Fraction` class that represents a fraction that is initialized by two numbers – numerator and denominator. The class should support basic arithmetic (`+, -, *, /`) with numbers and other `Fractions`

**Intermezzo 1:** *Writing modules.* Any python file can be imported by any other python file as a module using the `import` keyword, the name of the module is the name of the file (without the `.py` ending). When `imported`, the file is first executed, i.e., any code that sits outside of function definitions will run, for example, take two files in the same directory

`module.py`:

```
1 def my_module_function(x):
2     return 1 + x
3
4 print("Hello modules!")
```

`module_user.py`:

```
1 import module as m
2 print(m.my_module_function(1))
```

Upon running `module_user.py`, "Hello modules!" will be printed. This is usually not desirable. To prevent any code from running when imported and only allow it to run when the file is run directly we can do

```
1 if __name__ == '__main__':
2     print("Hello modules.")
```

Here, `__name__` is a special variable defined by Python itself that contains the name of the module associated with the current file. Its value is '`__main__`', if and only if it was directly.

To import a file as a module Python must be able to find it. By default, Python looks in the current working directory `os.getcwd()` and in the directories listed in the list `sys.path` in the package `sys`. If we want to load a package from somewhere else we can simply `sys.path.append()` its directory to the path variable.

## 3 NumPy, SciPy, and matplotlib

### 3.1 numpy

These three libraries form the basis of many (probably most) of scientific Python scripts. NumPy provides a numpy array, a fast way of storing numerical data in memory, and functions to work with them. SciPy provides a large number of algorithms including least squares fitting, signal processing, or spatial clustering and matplotlib allows for the creation of high-quality and customizable plots. Both are designed to work with numpy arrays.

**Intermezzo 2:** *Package installation – pip* Packages that are not installed by default can be installed using a command line utility `pip` which downloads packages registered in the Python Package Index (PyPI, <https://pypi.org/>). For example, to install the package `lmfit` run the following command in the command line (also works in the ipython REPL, e.g., in Spyder)

```
1    pip install lmfit  
2
```

To remove the package use `uninstall`, and to upgrade `install -U`. Run `pip help` for a complete list of available commands. If your virtual environment is activated, pip installs packages in this virtual environment.

If you are using uv, the command `uv pip` is compatible with standard `pip`, e.g., you can use `uv pip install lmfit`. Or, if you want to package your project to be used by other people, you can use `uv add lmfit` which also adds `lmfit` as a requirement of your package.

To use numpy we do

```
1    import numpy as np
```

there are several ways to create an array, e.g.

```
1    #directly from lists  
2    arr = np.array([1,2,3,4])  
3    #2D array  
4    arr_2D = np.array([[1,2,3],  
5                      [4,5,6],  
6                      [7,8,9]])  
7    # similar to list(range()), but none of the start, stop, or step have to be  
8    # integers  
9    arr = np.arange(start, stop, step)  
10   #linearly spaced values  
11   arr = np.linspace(start, stop, count)  
12   #filled with zeros  
13   np.zeros(length)  
14   #filled with ones  
15   np.ones(length)
```

Listing 7: Array creation.

By default np arrays store type float. For ones and zeros it is often useful to use boolean values (True and False) as `np.ones(10, dtype=bool)` will create an array of length 10 filled with True values. Arithmetic works on an element by element basis, therefore when we add/multiply/divide/compare the arrays must be the same length, e.g.,

```
1    >>> np.array([1,2,3]) + np.array([4,5,6])  
2    array([5,6,7])
```

However, if one of the operands is a scalar number, the operation *broadcasts*, e.g.,

```
1    >>> np.array([1, 2, 3]) + 10  
2    array([11, 12, 13])
```

Array slicing works similarly to list slicing, e.g., `arr[start:stop:step]`. Additionally, we can also index into an array using a list of indices, e.g., `arr[[0, 2, 4]]` will return an array containing 1st, 3rd and 5th element of array `arr`. Finally, using boolean arrays we can create an array subset of elements where the indexing array is true, e.g.,

```
1 arr1 = np.linspace(0, 10, 50)
2 b = np.sqrt(arr1) > 3 # array of bools True where np.sqrt(arr1) > 3
3 arr1[b] # only the elements of arr1 whose square root is larger than 3
```

**Exercise 3.1.** Write a program that will (in a loop) add numbers  $1 + 2 + 3 + \dots + n$  for any  $n$  using numpy arrays and compare it with the Gauss' expression  $n(n + 1)/2$  for user-supplied  $n$ .  
*Hint:* `np.arange`

**Exercise 3.2.** Write a function, which will return all prime numbers smaller than  $N$  using the Sieve of Eratosthenes method. *Hint:* `enumerate()`, `np.ones(N, dtype=bool)`, boolean array masking.

### 3.1.1 Basic measures and operations with the arrays

Since numpy arrays are intended primarily for numerical data, there are several methods built into them that calculate some basic measures, e.g.,

<code>arr.mean()</code>	mean of the array
<code>arr.std()</code>	standard deviation, biased estimate of the variance
<code>arr.sum()</code> , <code>arr.cumsum()</code>	the sum and cumulative sum
<code>arr.max()</code> , <code>arr.min()</code>	maximum and minimum, respectively
<code>arr.argmax()</code> , <code>arr.argmin()</code>	the index of maximum and minimum, respectively

As for any container type `len(arr)`, returns the length of the array. These also exist in their function forms (e.g., we can call `np.mean(arr)`). Additionally, there are useful functions that deal with arrays in the module itself: `np.diff(arr)` returns an array of differences between nearest neighbors one shorter than `arr`; for more general weighted averaging one can use `np.average(arr, weights)`. Most of these functions have their corresponding nan- versions, which ignore NaN values in the arrays (e.g., `np.nansum`, `np.nanmax` etc.). `np.isnan(arr)` returns an array of the same length as `arr` with True everywhere where `arr` was NaN. For boolean-valued arrays the extension of `and` and `or` are `np.logical_and(arr1, arr2)` and `np.logical_or(arr1, arr2)` and similarly `np.logical_not(arr)`, which create a new array with the logical operation applied to every element of the input arrays. Logical operators by themselves do not work with arrays.

**Intermezzo 3: NaNs and Infs.** Mathematical operations can result in either a Not-a-Number (NaN) or infinities, which are represented in numpy by `np.nan` and `np.inf`. These are special values indicating, for NaN, that an undefined operation was performed, e.g., `np.log(-1)` (Python and Numpy support complex numbers, but operations with real numbers are assumed to result in reals as well – if we wanted to use the complex logarithm we would have to give it a complex -1, i.e., `np.log(-1 + 0j)`).

Simple division by zero, e.g., `1/0`, will result in a `ZeroDivisionError` error. However, when part of an `np.array` it will result in an `±np.inf` and a Warning, e.g., try running `np.array(1)/np.array(0)`.

The idea of not crashing on undefined operations or division by zero is in the idea that often only part of the array is damaged (e.g., a -1 can be a placeholder for a value that's not available) and it might be easier and cleaner to proceed assuming that all operations are valid and then simply throw out the NaNs at the end.

An array can be sorted in-place using `arr.sort()` or a new sorted array can be created using `sorted_arr = np.sort(arr)`. To get an array of indices that would sort the array we can use `np.argsort()`, which is useful when we want to sort one array according to another, e.g.,

```

1 import numpy as np
2 items = np.array(["Eggs", "Bread", "Apples"])
3 prices = np.array([50, 20, 40])
4
5 sort_ix = np.argsort(prices)
6 print("Items sorted according to prices:")
7 print(items[sort_ix])

```

### 3.1.2 File I/O

We will be using numpy and matplotlib to analyze some experimental data. First, we need to get the data, typically from some file on the disk. Assuming that we have data stored as a text file in `data.txt` as two columns of numbers separated by tabs we can load it into a 2D numpy array using

```
1 arr = np.loadtxt("data.txt", comments='%', skiplines=3)
```

which loads `data.txt` into 2D array `arr`, ignores all lines beginning with % and skips first three datalines. If, rather than tabs our data is separated by commas (i.e., a `.csv` file) we can add a keyword argument `delimiter=','` or use `np.loadcsv()`.

**Intermezzo 4:** *Paths* Paths to files or directories can be absolute or relative. Absolute path specifies the absolute position of a file in the filesystem, on Windows it will typically begin with something like `C:\` and on Linux or Mac with the root directory `/`.

A relative path is relative to the current working directory, which can be obtained using `getcwd()` from the `os` module.

Different operating systems use different directory separators, i.e., Windows uses `\`, Linux and Mac `/` and Japanese Windows computers use `¥`. To join directory names in a way that will work everywhere we can use `os.path.join('dir1', 'dir2', 'dir3', ...)`.

To find multiple files whose name fit a pattern we can use the function `glob()` from the module `glob()`. For example, `glob(os.path.join("photos", "photo*.png"))` will return a list of all files that fit the pattern `"photos/photo<any number of any characters>.png"`

Using the file I/O functions from Python standard library is also possible. The function `file = open(filename, mode)` opens the file in a given mode, most common are `'r'` for reading, `'w'` for (over)writing, and `'a'` for appending. All contents of the file can be read in one go using `file.readlines()` or we can iterate over the lines of the file using a for loop (see example below). Inversely, there is `file.write(some_string)` (if the file is opened with mode `'r'` this will fail). Especially for writing, it is important to call `file.close()` after we are done with the file otherwise the changes might not be written to the disk due to caching.

The `np.loadtxt()` with default arguments is roughly equivalent to

```

1 import numpy as np
2 def my_load(fn):
3     with open(fn, 'r') as file:
4         rows = []
5         for line in file:
6             row = [float(s) for s in line.strip().split()]
7             rows.append(row)
8     return np.array(rows)

```

**Intermezzo 5:** *with statement (basic)* `with` is an example of so-called context management which ensures that resources (in this case a file) are properly cleaned up after we are done using them. We will learn how to create our own context managers later, for now, the code

```

1     with open(fn, 'r') as file:
2         do_stuff()

```

is roughly equivalent to

```

1     file = open(fn, 'r')
2     do_stuff()
3     file.close()

```

with the caveat that `with` ensures that `file.close()` runs even if `do_stuff()` crashes the program.

**Exercise 3.3.** Extend the `my_load(fn)` from the example above to support comments in the files.

## 3.2 matplotlib

matplotlib allows plotting data. We plot data inside axes, which are contained in figures. A figure can contain multiple axes. The simplest way to create a new figure with new axes is <sup>4</sup>

```

1     import matplotlib.pyplot as plt
2     fig, ax = plt.subplots()

```

If we have data in arrays `x` and `y` of the same length we can plot them using, e.g.,

```

1     ax.plot(x, y, '-') # connect the pairs of points given by x and y with full lines
2     ax.semilogy(x, y, '--s') # logarithmic y-axis, dashed line with square markers
3     ax.loglog(x, y, ':') # logarithmic x and y axis, dotted line
4     ax.scatter(x, y) #scatter plot, does not connect the points by lines

```

Listing 8: Basic plotting.

The plotting format is specified using a format string immediately following the data which has the format `fmt = [marker][line][color]`.<sup>5</sup> We can specify the line width using `lw=` keyword, marker size using `ms=` and color using `color=`.<sup>6</sup> Note that matplotlib does not care whether the pair of arrays `x` and `y` represents a mathematical function, it will simply connect the points sequentially with lines.

To label the axes we can use `ax.set_xlabel()` and `ax.set_ylabel()`. Data can be labeled by passing `label=` keyword argument with a string to any of the plotting commands and then calling `ax.legend()`. We can save the figure using `fig.savefig(filename)`.

For example, to plot a Gaussian with magenta lines and cyan points we can do

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 xs = np.linspace(0, 10, 100)
5 ys = np.exp(-(xs - 5)**2)
6
7 plt.close('all') #closes all figures we have opened so far
8 fig, ax = plt.subplots()
9
10 ax.plot(xs, ys, '--s', ms=5, lw=2, color='magenta', label=r'ax.plot($e^{-x^2}$)')
11 ax.scatter(xs, ys, s=ys*10, marker='o', color='cyan', zorder=3,
12            label=r'ax.scatter($e^{-x^2}$)')
13
14 ax.set_xlabel('$x$ values')
15 ax.set_ylabel(r'the gaussian $e^{-(x-5)^2}$')

```

<sup>4</sup>Note that `plt.subplots()` returns two values. It only returns a single tuple, which is desctructured, or unpacked, into two variables `fig` and `ax`.

<sup>5</sup>See [https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.plot.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html) for complete list of possible formats

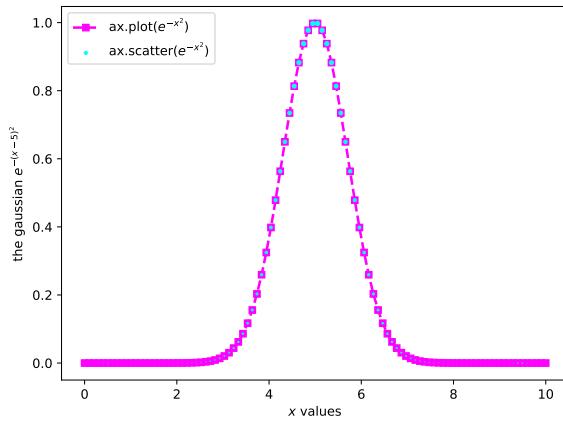
<sup>6</sup>See <https://matplotlib.org/stable/users/explain/colors/colors.html>

```

16 #where to put the legend, we can also use "best" and let matplotlib guess
17 ax.legend(loc='upper left')
18
19 fig.tight_layout() #reduces some of the whitespace around edges
20 fig.savefig('gaussian.pdf') #the file format of the image is deduced from the
    extension

```

Listing 9: Complete plotting example.



Notice that we can control which objects are drawn on top of which using the `zorder=` keyword argument and that basic L<sup>A</sup>T<sub>E</sub>Xmath typesetting is supported, but latex commands beginning with backslash need to be either escaped (i.e., "`\"\\frac{}{}`") or in raw strings (i.e., `r"\\frac{}{}`).

**Exercise 3.4.** Write a program, which will, using NumPy and Matplotlib, graph the expressions  $y = x$ ,  $y = x^2$  and  $y = \sqrt{x}$  from  $x = 0$  to  $x = 5$  with different line styles (e.g., full line, dashed, dotted) and user-supplied number of points. Try linear, semilogarithmic and logarithmic axes. Label the axes and the curves.

### Exercise 3.5.

- Write a function that will integrate an arbitrary dependency given by points  $x_n, y_n$ , i.e., a discrete version of  $\int y(x)dx$ , and use it to calculate the integral of  $y = x^2$  from 0 to 5. (*Hint: the approximate value of the integral is given by the trapezoidal rule:  $\sum_{n=0}^{N-2} 0.5dx_n(y_{n+1} + y_n); dx_n = x_{n+1} - x_n$ ,  $n = 0 \dots N - 1$* )
- Using the function from the previous point, write a function that will integrate an arbitrary single-argument Python function that returns a number, between points  $a$  and  $b$ . The number of points  $N$  should be optional, with a default value of 100.
- Graphically show the convergence of the integral from the previous point to the true value with an increasing number of discretization points  $N$  and compare it with the function `scipy.integrate.quad`. Display the absolute error on a logarithmic scale as a function of  $N$ . For the convergence test, you can use the integral

$$\int_0^1 e^{-x} dx$$

Put the functions from the previous points into a separate .py file and use it as a module.

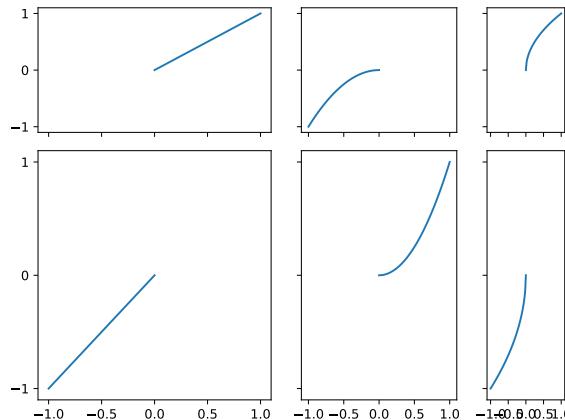
One figure can contain multiple axes created by passing the optional rows and columns arguments to `plt.subplots()`. The multiple axes are created in a regular grid with a specified number of rows and columns. For axes of unequal sizes, we can specify ratios of their widths and heights using `width_ratios` and `height_ratios` (see `gridspec`<sup>7</sup> for more complicated figure layouts). The axes can share the x and y ranges by specifying the boolean keyword arguments `sharex` and `sharey`, respectively, for example

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 nrows = 2
4 ncols = 3
5 fig, axes = plt.subplots(nrows, ncols, sharex=True, sharey=True,
6                         width_ratios=[3, 2, 1], height_ratios=[1, 2])
7 #axes is now a 2D array of axes which all share the same x and y range
8
9 xs = np.linspace(0, 1)
10
11 axes[0,0].plot(xs, xs)
12 axes[1, 1].plot(xs, xs**2)
13 axes[0, 2].plot(xs, np.sqrt(xs))
14
15 axes[1,0].plot(-xs, -xs)
16 axes[0, 1].plot(-xs, -xs**2)
17 axes[1, 2].plot(-xs, -np.sqrt(xs))
18
19 fig.tight_layout()
20 fig.savefig('multiax.pdf')

```

Listing 10: Multiple axes in single figure.



**Exercise 3.6.** Write a program that will read the data from "data.txt" which contains three columns of numbers separated by tabs. Let's call these columns frequency,  $X$  and  $Y$ . Plot the frequency dependence of  $X$  and  $Y$

1. in the same axes ( $X$  full line,  $Y$  dashed line)
2. in separate axes in the same figure sharing x and y range

<sup>7</sup><https://matplotlib.org/3.5.0/tutorials/intermediate/gridspec.html>

Plot also  $X$  vs.  $Y$  in a separate plot. Label the axes appropriately.

**Exercise 3.7.** Find the maximum and the minimum of the absolute value  $R = \sqrt{X^2 + Y^2}$ , and indicate their positions using vertical lines in the graph of frequency dependence of  $X$  and  $Y$  from the previous exercise. Estimate the full-width-at-half-maximum (FWHM) of the peak and the quality factor  $f_0/\text{FWHM}$ , where  $f_0$  is the frequency of the maximum response.

*Hint:* `np.argmin`, `np.argmax`, `ax.axvline`, `ax.axhline`

**Exercise 3.8.** Process all files from `lots_of_data` directory similarly to Exc. 7. Plot the FWHM as a function of  $f_0$ . Use the solution to Exc. 7 as a module.

*Hint:* `glob`

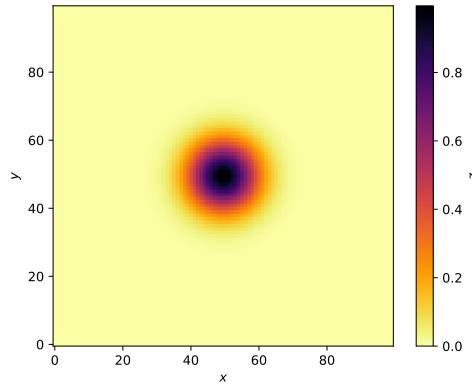
Observables that depend on two control variables can be often plotted using heat maps, for which we can use `ax.imshow(arr2D)`, which takes a 2D array of numbers and maps them to a color of a pixel using a colormap<sup>8</sup> Note however, that `imshow()` is primarily used for images, which conventionally start in the upper left corner with left-handed axes. Data typically start in the lower-left corner with right-handed axes. This can be changed by specifying `origin='lower'` keyword to `imshow()`. Alternatively, for plotting data which are not on a regular grid we can use `ax.pcolormesh(x, y, z)` where  $x$ ,  $y$ ,  $z$  are 2D arrays.

For example, to plot a 2D Gaussian with the '`inferno_r`' colormap,

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #x and y axis
5 _xs = np.linspace(0, 10, 100)
6 _ys = np.linspace(0, 10, 100)
7 #but we need 100 x 100 points for both x and y that sample the
8 #the entire interval (0, 10) x (0, 10), this can be done using
9 #meshgrid
10 xs, ys = np.meshgrid(_xs, _ys)
11 zs = np.exp(-(xs - 5)**2 - (ys - 5)**2)
12
13 plt.close('all') #closes all figures we have opened so far
14 fig, ax = plt.subplots()
15
16 plot = ax.imshow(zs, cmap='inferno_r', origin='lower')
17 #similar
18 #plot = ax.pcolormesh(xs, ys, zs, cmap='inferno_r')
19 cbar = fig.colorbar(plot) #the color axis scaling
20 cbar.set_label('$z$')
21 ax.set_xlabel('$x$')
22 ax.set_ylabel('$y$')
```

Listing 11: Two-dimensional plotting example.

<sup>8</sup>See <https://matplotlib.org/stable/users/explain/colors/colormaps.html> for a full list of colormap names.



**Exercise 3.9.** Plot the Mandelbrot set with configurable range and resolution.

*Hint:* The Mandelbrot set is a set of complex numbers  $c$  for which the series  $z_{n+1} = z_n^2 + c$ ,  $z_0 = 0$  does not diverge. If  $z_n \geq 2$  for any  $n$  the series will diverge. As a convergence criterium we can use that  $z_n < 2 \forall n < N_{\max}$  ( $N_{\max} = 100$ , for example). Plot the set using `plt.imshow(c)` where `c[i, j]` is the number of interactions needed to exceed  $z_n = 2$  (or  $N_{\max}$ ). The entire set is contained in the rectangle with lower-left corner  $-2 - i$  and upper-right corner  $1 + i$  in the complex plane.

### 3.2.1 Review of basic plotting commands

Assuming that the figure and axes were created as `fig, ax = plt.subplots()`, the basic commands for manipulating the plot are

<code>ax.xlabel("the x-label")</code> ,	sets the axis labels
<code>ax.ylabel("the y-label")</code>	
<code>ax.set_xlim(xmin, xmax)</code> ,	sets the axes limits, xmin, xmax etc. can also be used as keywords
<code>ax.set_ylim(ymin, ymax)</code>	
<code>ax.set_aspect('equal')</code>	sets the aspect ratio of the axes to be equal, useful when both axes contain qualitatively similar data
<code>ax.legend(loc=location)</code>	shows the legend at locatoin location $\in$ "upper lower left right" or "best"
<code>fig.tight_layout()</code>	adjust the axes size to fit all labels and reduces whitespace
<code>fig.supxlabel(' xlabel')</code> ,	sets the common figure-wide axis labels for multi-axis figures
<code>fig.supylabel(' ylabel')</code>	
<code>fig.colorbar()</code>	creates a colorbar in the figure
<code>plt.close('all')</code>	closes all open figures

## 4 Least squares fitting and interpolation

Given data represented by a set of points  $[(x_i, y_i)]$ , where  $x_i$  is the control variable and  $y_i$  is the observable (e.g., I control the current through a resistor and observe the voltage), the least-squares fit to a function  $f(x; p_1, p_2, \dots, p_n)$  minimizes the sum of squared residuals, i.e.,

$$R^2 = \sum_i |f(x_i, \{p\}) - y_i|^2. \quad (1)$$

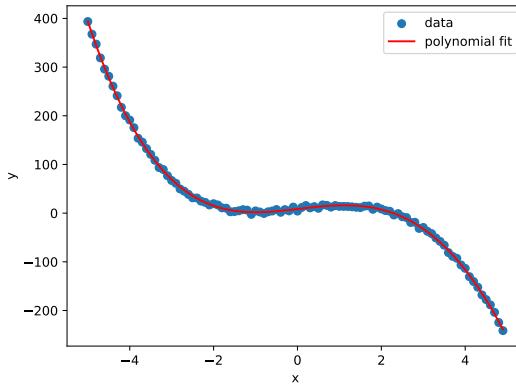
The result of the fit is a set of parameters  $p_1, p_2, p_3, \dots$  which minimize  $R^2$ . Depending on how the function  $f$  depends on the parameters  $p_k$  we talk about *linear* or *non-linear* fitting. The important difference is the dependence on the parameters  $p$ , not on the control variable  $x$ .

### 4.1 Polynomial fitting.

Fitting a polynomial in  $x$  where the fit parameters are the coefficients of the polynomial is a very common example of linear fitting. Interface for using polynomials is contained in `np.polynomial` submodule in the `Polynomial` class, which provides a

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 from numpy.polynomial import Polynomial as P
5
6 rng = np.random.default_rng() # random number generator
7
8 %%# create some noisy data and plot them
9 noise_amplitude = 10
10 xs = np.arange(-5, 5, 0.1)
11 ys = 0.1*xs**4 - 3*xs**3 + 10*xs + 4 + rng.random(len(xs))*noise_amplitude
12
13 fig, ax = plt.subplots()
14 ax.scatter(xs, ys, label='data')
15
16 %%# fit the polynomial
17 poly = P.fit(xs, ys, deg=4)
18 ax.plot(xs, poly(xs), color='r', label='polynomial fit')
19 #the coefficients of poly are scaled for a particular domain/window
20 #to get the ordinary coefficients we need to use .convert()
21 coef = poly.convert().coef
22 poly_string = ' + '.join([f'({c:.1f})*x^{k}' for k, c in enumerate(coef)])
23 print("The estimated polynomial is", poly_string)
24
25 ax.set_xlabel('x')
26 ax.set_ylabel('y')
27 ax.legend(loc='best')
28 fig.savefig('polynomial_fit.pdf')
```

Listing 12: Polynomial fitting



**Exercise 4.1.** Remove the background from the peaks in exercise 7 using a polynomial fit.

*Hint:* `np.polynomial.Polynomial.fit()` and boolean arrays

## 4.2 Non-linear curve fitting

When the fitting function depends nonlinearly on the fit parameters we need to use nonlinear fitting, several libraries exist for this task, and for basic fitting, we can use `scipy.optimize` submodule of `scipy`.

To fit a given function to data we can use `scipy.optimize.curve_fit`, e.g.,

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 from scipy.optimize import curve_fit
5
6 rng = np.random.default_rng()
7
8 #%% function to fit
9 def lorentzian(xs, height, center, width):
10     return height*center**2*width**2/((center**2 - xs**2)**2 + width**2*xs**2)
11
12 #%% generate some data and plot
13 xs = np.arange(0, 10, 0.1)
14 height = 0.5
15 center = 5
16 width = 2
17 noise = 0.05
18 ys = lorentzian(xs, height, center, width) + rng.random(len(xs))*noise
19
20 fig, ax = plt.subplots()
21 ax.scatter(xs, ys, label='data')
22
23 #%% fit the data
24 popt, pcov = curve_fit(lorentzian, xs, ys, p0=[1, 1, 1])
25 # curve_fit returns the optimized parameters (popt) and the covariance matrix (pcov)
26 # the diagonal of the covariance matrix can be used as simple error estimates of the
27 # parameters
28 ax.plot(xs, lorentzian(xs, *popt), color='r', label='fit')
29 #
30 # this substitutes (positionally) elements of the iterable into the function call
31 print(f"Estimated height: {popt[0]:.3f} +/- {np.sqrt(pcov[0,0]):.3f}")
32 print(f"Estimated center: {popt[1]:.3f} +/- {np.sqrt(pcov[1,1]):.3f}")
33 print(f"Estimated width: {popt[2]:.3f} +/- {np.sqrt(pcov[2,2]):.3f}")
34

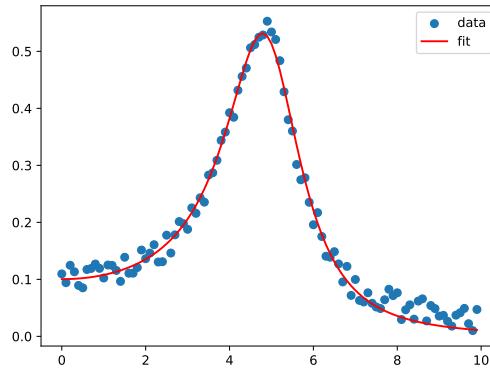
```

```

35 ax.legend(loc='best')
36 fig.savefig('curve_fit.pdf')

```

Listing 13: Nonlinear curve fitting.



Unlike linear least squares, non-linear curve fitting is iterative and stops once the parameters converge. The fit may never converge for certain problems, therefore, once a configured maximum number of iterations is exceeded the fitting routines *raise an exception* (See 6) which will crash the program if not handled.

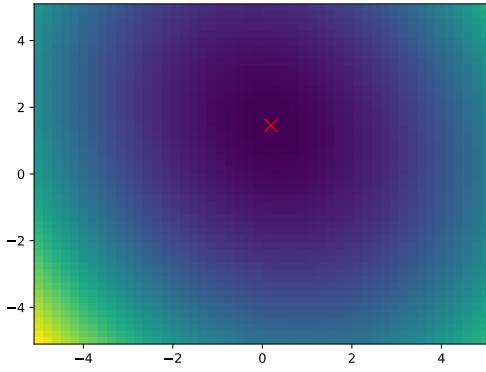
We can use `scipy.optimize.minimize()` for more general optimisation problems. `minimize()` takes a single **scalar** function and an initial guess for the optimal parameters. For example, to find a minimum of a parabola,

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.optimize as optim
4
5 def parabola(x, y):
6     return 4*x**2 + 2*y**2 + x*y - 6*y - 3*x + 5
7
8 #the lambda is there to simply turn a function of two parameters into a
9 #function of a single parameter
10 xmin = optim.minimize(lambda z: parabola(*z), x0=[0,0])
11
12 XX, YY = np.meshgrid(np.linspace(-5, 5), np.linspace(-5, 5))
13 p = parabola(XX, YY)
14 fig, ax = plt.subplots()
15 ax.pcolormesh(XX, YY, p)
16 ax.plot(*xmin.x, 'x', color='r', ms=10)
17 fig.savefig('parabola-minimize.pdf')

```

Listing 14: Minimization of a scalar function of multiple parameters.



**Parameter error estimation** The fitting function `curve_fit` returns two values – the array of the parameters we are interested in and the covariance matrix of the parameters, i.e.,  $\text{cov}(p_i, p_j) = \langle (p_i - \bar{p}_i)(p_j - \bar{p}_j) \rangle$ . The diagonal values of the covariance matrix can be used as an estimate of the uncertainties of the fit parameters, i.e.,

```

1 #assuming that f is def'ed as f(x, p1, p2)
2 p, cov = curve_fit(f, x, y)
3 print(f"p1 = {p[0]} +/- {np.sqrt(cov[0,0])}")

```

The covariance matrix is calculated from the *residuals*  $r_i = y_i - f(x_i, \{p_j\})$ , where  $p_j$  are the optimized parameters as ( $\mathbf{r}$  is a vector with elements  $r_i$ ;  $i = 1 \dots N$ ,  $j = 1 \dots M$ ) as a scaled inverse of the Hessian matrix  $H$  of the objective function  $\chi^2 = \sum r_i^{29}$

$$\text{cov}(p_i, p_j) = \frac{1}{N - M} \begin{pmatrix} \frac{\partial \chi^2}{\partial p_1 \partial p_1} & \frac{\partial \chi^2}{\partial p_1 \partial p_2} & \cdots \\ \frac{\partial \chi^2}{\partial p_2 \partial p_1} & \frac{\partial \chi^2}{\partial p_2 \partial p_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}^{-1}, \quad (2)$$

where  $N$  is the number of data points and  $M$  is the number of parameters. Note that usually, we are interested only in the diagonal for the direct error estimate of the fit parameters, however, if, for example, the studied quantity is a sum of two fit parameters  $z = A + B$ , then the variance of  $z$

$$\text{var}z = \langle (A+B-\bar{A}-\bar{B})^2 \rangle = \langle (A-\bar{A})^2 \rangle + \langle (B-\bar{B})^2 \rangle + 2\langle (A-\bar{A})(B-\bar{B}) \rangle = \text{var}A + \text{var}B + 2\text{cov}(A, B), \quad (3)$$

and the off-diagonal terms of the covariance matrix must be used.

However, the error estimates calculated using the covariance matrix can often be underestimated since the above method is valid only when the model function  $f$  is correct and the data truly have the form  $y_i = f(x_i, p) + e_i$  where data errors  $e_i$  have zero mean and a normal distribution. A more robust, but also more computationally demanding, method of estimating parameter errors is **bootstrap**, where, for several repetitions, we create a random selection of data (of the same length), perform the fit on each created set and calculate mean and standard deviation (and in principle covariance) on the resulting set of fit parameters, an example implementation is shown in Lst. 15

```

1 #%% fit the data
2 #...
3 bootstrap_N = 100
4 parameter_N = 3
5 data_N = len(xs)

```

---

<sup>9</sup>The calculation is slightly more complicated, but similar, when the data points have different weights, see documentation [https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve\\_fit.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html)

```

6 ps = np.empty((bootstrap_N, parameter_N)) # here we will save all fit parameters
7 for k in range(bootstrap_N):
8     # some data will be omitted, some will occur more than once
9     ix = rng.choice(range(data_N), data_N)
10    ix.sort()
11    popt, _ = curve_fit(lorentzian, xs[ix], ys[ix], p0=[1, 1, 1])
12    ps[k,:] = popt
13
14 # the fit function does not depend on the sign of the parameters center and width
15 # and curve_fit will randomly find one or the other. Make sure that we are averaging
16 # the same signs
17 ps = abs(ps)
18 # do the statistics on the set of parameters ps, which is a 2D array.
19 # work only along one axis, we don't want to average everything together
20 popt_bootstrap = np.mean(ps, axis=0) #1st axis (0th) averages over the rows
21 perr_bootstrap = np.std(ps, axis=0) #the same for standard deviation
22 ax.plot(xs, lorentzian(xs, *popt_bootstrap), color='r', label='fit')
23
24 print(f"Estimated height: {popt_bootstrap[0]:.3f} +/- {perr_bootstrap[0]:.3f}")
25 print(f"Estimated center: {popt_bootstrap[1]:.3f} +/- {perr_bootstrap[1]:.3f}")
26 print(f"Estimated width: {popt_bootstrap[2]:.3f} +/- {perr_bootstrap[2]:.3f}")

```

Listing 15: Estimating fit parameter errors using bootstrap. The data  $(x_i y_i)$  are created in the same way as in Lst. 13.

**Intermezzo 6: Exceptions and error handling** Exceptions are a mechanism used by Python to signal errors or other events which need to be handled by your code. If a *raised* exception is not *caught* the program crashes. To handle exceptions we use the `try`, `except`, and `finally` blocks. We can raise an exception using `raise`. Note that exceptions do not always have to indicate an error, for example, a `for` loop is terminated using a `StopIteration` exception.

Example of catching and raising exceptions and using the `finally` block.

```

1
2 def faulty_function():
3     raise ValueError("blergh!")
4
5 xs = [-2, -1, 0, 1, 2]
6 one_over_xs = []
7 try:
8     for x in xs:
9         try:
10             one_over_xs.append(1/x)
11             if x > 1:
12                 faulty_function()
13         except ZeroDivisionError:
14             print("Can't divide by zero!")
15         except:
16             print("something else went wrong")
17             raise #propagate the exception further up
18 finally:
19     print("I will always run")
20
21 # unless the exception that is raised on line 8 and then sent forward on line 13
22 # isn't
23 # handled, this line will not run
24 print(one_over_xs)

```

Notice a few things:

`except` Can catch either a specific type of exception or any type (if the exception type is not specified).

`finally` block always runs, regardless of whether an exception occurred inside the `try` block and is generally meant for proper cleanup (e.g., open files)

**try blocks can be nested:** When an exception is raised, the inner-most `try-except` block tries to handle it. If a suitable `except` is not found or the exception is re-raised, the next enclosing `try-except` block tries to handle it and so on. If the exception gets out of all nested enclosing `try-except` blocks without being handled, the program crashes.

**raise can be used anywhere**, e.g., in functions which do not contain `try`. It is up to the calling code to decide what to do with exceptions

**Exercise 4.2.** Fit the absolute value  $r(f) = \sqrt{x^2 + y^2}$  of the response in exercise 7 to the response of a linear harmonic oscillator plus a polynomial background (polynomial degree 3) and plot the results similarly to exercise 7. Use the simple estimation from exercise 6 as initial estimates of the fit parameters. Use the solution to exercise 6 as a module.

Bonus exercise: make the background polynomial degree adjustable

The complex amplitude of the response of a linear harmonic oscillator to force  $F$  is (see Appendix. A)

$$x(\omega) = \frac{F/m}{\omega_0^2 - \omega^2 + i\omega\gamma}, \quad (4)$$

where  $\omega_0$  is the (angular) resonance frequency,  $\omega$  is the frequency of the force  $F$ ,  $m$  is the oscillator mass and  $\gamma$  is damping.

**Exercise 4.3.** Estimate the errors of fitting parameters obtained in exercise 2 using bootstrap.

## 5 Digital signal processing

### 5.1 Digital representation of a continuous signal

Assume a signal  $y(t)$  (e.g., a voltage) which varies continuously in time. To store and process this signal on a computer we measure it at a set of time instants  $t_i$  yielding values of the signal  $y_i$ , resulting in the discrete representation of the signal as a series of pairs  $(t_i, y_i)$ . The signal is uniformly sampled if  $t_i = i\Delta t$ , where  $\Delta t$  is the **sampling period** and  $f_s = 1/\Delta t$  is the **sampling frequency**.

Any information about the original signal  $y(t)$  that varies in time faster than  $\Delta t$  is lost. However, if  $y(t)$  oscillates at a frequency  $f$  it is insufficient to have  $f_s \geq f$ . Signal frequencies  $f > f_s/2$  will appear shifted to  $f - f_s/2$ , see the output of Lst. 16. The sampled signal contains fictitious frequencies which are not present in the original signal.

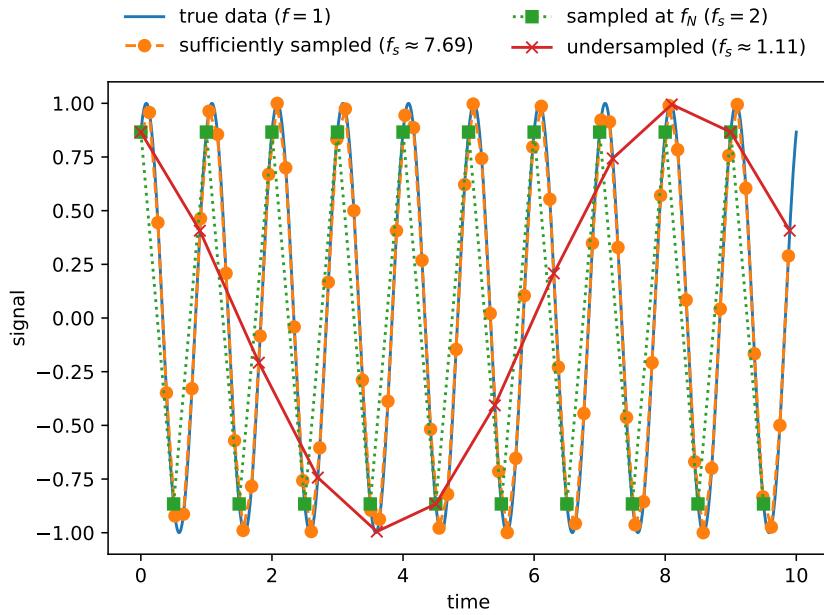
This distortion is called **aliasing** and the *Nyquist theorem* states that in order to avoid aliasing, the highest frequency contained in the signal  $f$  must not be higher than one half of the sampling frequency  $f_s/2$ , which is called the **Nyquist frequency**

$$f_N = f_s/2. \quad (5)$$

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.interpolate as intp
4
5 #create "continuous" data (very high sampling rate)
6 T = 10 #total length of data in "time" units
7 xs = np.linspace(0, T, 10000)
8 ys = np.sin(2*np.pi*xs + 7*np.pi/3)
9
10 #interpolate the signal at high sampling rates
11 signal = intp.interp1d(xs, ys)
12
13 def sample(signal, dx, T=T):
14     """
15         Samples the quasi-continuous signal with
16         total length T with sample spacing dx
17     """
18     xs = np.arange(0, T, dx)
19     return xs, signal(xs)
20
21 fig, ax = plt.subplots()
22 ax.plot(xs, ys, '-.', label='true data ($f = 1$)')
23
24 ax.plot(*sample(signal, 0.13), '--o',
25         label=r'sufficiently sampled ($f_s \approx 7.69$)')
26 ax.plot(*sample(signal, 0.5), ':s',
27         label=r'sampled at $f_N$ ($f_s = 2$)')
28 #this signal will be aliased
29 ax.plot(*sample(signal, 0.9), '-x',
30         label=r'undersampled ($f_s \approx 1.11$)')
31
32 ax.set_xlabel('time')
33 ax.set_ylabel('signal')
34
35 ax.legend(ncol=2, frameon=False,
36           loc='lower left', bbox_to_anchor=(0.0, 1.01))
37 fig.tight_layout()
38 fig.savefig('sampling.pdf')
```

Listing 16: Effect of sampling rate



## 5.2 Spectral analysis

Any periodic signal  $y(t)$  (which can be complex) with a period  $T$ , i.e.,  $y(t+T) = y(t)$ , can be represented as a sum of sine and cosine terms oscillating at angular frequencies  $2\pi n/T$  where  $n$  is an integer,

$$y(t) = \frac{1}{T} \sum_{k=0}^{\infty} A'_k \sin\left(\frac{2\pi k}{T}t\right) + B'_k \cos\left(\frac{2\pi k}{T}t\right), \quad (6)$$

which is called the **Fourier series**. Equivalently, the Fourier series can be expressed using complex exponentials

$$y(t) = \frac{1}{T} \sum_{k=-\infty}^{\infty} A_k e^{2\pi i k t / T}, \quad (7)$$

where for a real signal  $y$  the positive and negative terms are complex conjugate,  $A_k = A_{-k}^*$  (the  $1/T$  factor and sign inside the exponential function are conventional).

Fourier coefficients  $A_k$  can be calculated as

$$A_k = \int_0^T y(t) e^{-2\pi i k t / T} dt \quad (8)$$

For example, consider the directly-calculated Fourier series in Lst. 17

```

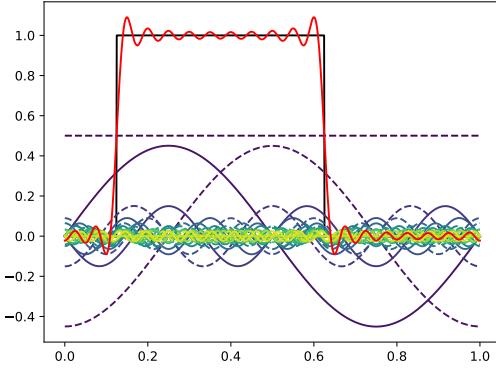
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # create a square pulse from t=0.125 to t=0.625
5 ts = np.linspace(0, 1, 1000)
6 dt = ts[1] - ts[0]
7 ys = np.zeros_like(ts)
8 start = 0.125
9 ys[np.logical_and(ts > start, ts < start + 0.5)] = 1
10
11 plt.close('all')
12 fig, ax = plt.subplots()
```

```

13 ax.plot(ts, ys, 'k')
14
15 #we will represent our signal y(t) with a Fourier series
16 #  $y(t) = \sum_k A_k \sin(2\pi k t) + B_k \cos(2\pi k t)$ 
17
18 #to calculate the individual  $A_k$  and  $B_k$ , we simply have to integrate
19 #the signal with the appropriate sine or cosine term
20 def Ak(ts, ys, k):
21     sint = np.sin(2*np.pi*k*ts)
22     # we multiply by 2 because  $\sin(\dots)^2$  averaged over a period is 1/2
23     return 2*np.sum(ys*sint)*dt
24
25 def Bk(ts, ys, k):
26     if abs(k) > 0:
27         cost = np.cos(2*np.pi*k*ts)
28         return 2*np.sum(ys*cost)*dt
29     else:
30         #if k==0 (cos(...) == 1) the factor 2 is not needed
31         return np.sum(ys)*dt
32
33 #calculate the first K fourier coefficients A and B
34 K = 20
35 As = [Ak(ts, ys, k) for k in range(K)]
36 Bs = [Bk(ts, ys, k) for k in range(K)]
37
38 #total will be the total sum of the Fourier series
39 total = 0
40
41 #prepare a colormap for plotting
42 cmap = plt.get_cmap('viridis')
43 norm = plt.Normalize(vmin=0, vmax=K)
44
45 #iterate over all coefficients and frequencies
46 for k, (A, B) in enumerate(zip(As, Bs)):
47     sint = A*np.sin(2*np.pi*k*ts)
48     cost = B*np.cos(2*np.pi*k*ts)
49
50     # plot the oscillating terms with the color given
51     # by the frequency
52     ax.plot(ts, sint, color=cmap(norm(k)))
53     ax.plot(ts, cost, '--', color=cmap(norm(k)))
54
55     total += sint + cost
56
57 ax.plot(ts, total, '--', color='r')
58 fig.savefig('../fourier_series_square_pulse.pdf')
59
60 #finally, plot the Fourier spectrum itself. Try changing the phase of the
61 #signal by changing the start variable on line 7. Individual As and Bs will
62 #change, but the modulus of the complex number  $|A + iB|$  stays the same.
63 fig_spec, ax_spec = plt.subplots()
64 ax_spec.plot(range(K), As, '-o', label='sine terms, $A$')
65 ax_spec.plot(range(K), Bs, '-o', label='cosine terms, $B$')
66 ax_spec.plot(range(K), np.abs(As + 1j*np.array(Bs)), '-s', label='$|A + iB|$')
67 ax_spec.legend(loc='best')
68 plt.show()

```

Listing 17: Fourier series of a square pulse.



These definitions can be extended to aperiodic signals (which can be imagined as signals with infinitely long periods) which yields the **Fourier transform**

$$\tilde{y}(\omega) = \int_{-\infty}^{\infty} y(t)e^{-i\omega t} dt, \quad (9)$$

and the **inverse Fourier transform**

$$y(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \tilde{y}(\omega)e^{i\omega t} d\omega \quad (10)$$

The absolute value of the Fourier coefficients  $|A|$  is the amplitude of the signal's oscillation at a given frequency and their complex phase is the phase rotation of the sine and cosine terms. For the Fourier transform, since the frequency now varies continuously, the  $|\tilde{y}|$  has a meaning of *density* (called spectral density). In analogy with electrical power  $P = V^2/R$  needed to generate voltage  $V$  across a resistor  $R$ ,  $|\tilde{y}|^2$  is called the *power spectral density*, i.e.,  $\int_{\omega_0}^{\omega_1} |\tilde{y}(\omega)|^2 d\omega$  is the power of the signal in the frequency band  $(\omega_0, \omega_1)$ .

For digital signals, we talk about *discrete Fourier transforms*. For uniformly sampled signals,  $t_n = n\Delta t$  these are in SciPy defined as (in the submodule `scipy.fft`)

$$\tilde{y}[k] = \sum_{n=0}^{N-1} y[n]e^{-2\pi i kn/N}, \quad (11)$$

where  $y[n]$  is the value of the signal  $y$  measured at time  $t_n = n\Delta t$  and  $N$  is the total number of samples. The dimensionless integer frequencies  $k$  run from 0 to  $N - 1$  – that is, the fourier transform has the same length as the original signal.

The inverse discrete Fourier transform is defined similarly,

$$y[n] = \frac{1}{N} \sum_{k=0}^{N-1} \tilde{y}[k]e^{+2\pi i kn/N}. \quad (12)$$

The integer frequency indices  $k$  represent frequencies  $f'_k = k/N$  for  $k = 0 \dots N/2$  and  $f'_k = -k/N$  for  $k = N/2 \dots N - 1$ . To get the actual frequencies we only need to scale  $f'_k$  with the actual sampling frequency  $1/\Delta t$ . Notice that the definitions of the discrete Fourier transform and its inverse do not depend on the sampling rate, only on the fact that the signal is sampled uniformly.

Fourier transforms are implemented in NumPy and SciPy using the **Fast Fourier Transform – FFT** algorithm<sup>10</sup> in the `scipy.fft` submodule. Fourier transform is calculated using `scipy.fft.fft()`

---

<sup>10</sup>FFT is one of the most important algorithms that the entire digital world depends on – e.g., sound and video encoding and wireless communication rely on it heavily.

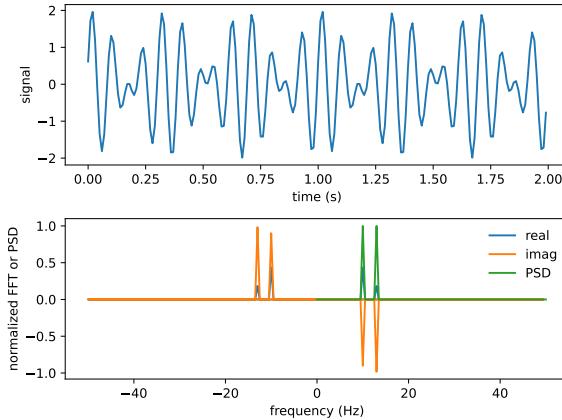
and the actual frequencies can be calculated using a helper function `scipy.fft.fftfreq()`. Both `fft` and `fftfreq` return positive and negative frequencies. For real signals, the negative frequencies do not provide any extra information therefore we can use `scipy.fft.rfft()` and `rfftfreq()` which return only positive frequencies (i.e., the result is half the length of the original signal). If we are only interested in the power spectral density, there is `scipy.signal.periodogram()` which also calculates the frequencies. By default, for real signals, `periodogram` returns only the positive frequencies and both positive and negative frequencies for complex signals. See Lst. 18 for example usage and Lst. 19 for a version of the program in Lst. 17 but using FFT rather than manual calculation of the coefficients

```

1 import numpy as np
2 import scipy.fft as fft
3 from scipy.signal import periodogram
4
5 import matplotlib.pyplot as plt
6
7 #first create some signal
8 N = 200 #total number of points
9 dt = 0.01 # sampling period, sampling frequency = 1/dt = 100
10
11 #time
12 ts = np.arange(N)*dt
13 #signal: sum of two sine waves
14 ys = np.sin(2*np.pi*10*ts + np.pi/7) + np.sin(2*np.pi*13*ts + np.pi/17)
15
16 #plot the signal and label the axes
17 fig, (ax_sig, ax_fft) = plt.subplots(2, 1)
18 ax_sig.plot(ts, ys)
19 ax_sig.set_xlabel('time (s)')
20 ax_sig.set_ylabel('signal')
21
22 ax_fft.set_xlabel('frequency (Hz)')
23 ax_fft.set_ylabel('normalized FFT or PSD')
24
25 #compute the discrete Fourier transform of the signal
26 fft_frequencies = fft.fftfreq(N, dt)
27 Y_fft = fft.fft(ys)
28 ax_fft.plot(fft_frequencies, Y_fft.real/abs(Y_fft).max(), label='real')
29 ax_fft.plot(fft_frequencies, Y_fft.imag/abs(Y_fft).max(), label='imag')
30
31 #power spectral density
32 #periodogram() also returns the frequencies
33 psd_freq, psd = periodogram(ys, fs=1/dt) #fs = sampling frequency
34 ax_fft.plot(psd_freq, psd/psd.max(), label='PSD')
35 ax_fft.legend(loc='best', frameon=False)
36 fig.tight_layout()
37 fig.savefig('fouriers.pdf')
38 plt.show()

```

Listing 18: Calculation of Fourier transform and power spectral density.



```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.fft as fft
4
5 ts = np.linspace(0, 1, 1001)
6 dt = ts[1] - ts[0]
7 N = len(ts)
8 ys = np.zeros_like(ts)
9 start = 0.125
10 ys[np.logical_and(ts >= start, ts < start + 0.5)] = 1
11
12 plt.close('all')
13 fig, ax = plt.subplots()
14 ax.plot(ts, ys, 'k')
15
16 # FFT returns all frequencies it can
17 #ys is real, so we can use rfft
18 Y = fft.rfft(ys)
19 frequencies = fft.rfftfreq(N, d=dt)
20
21 #how many fourier terms do we want to look at
22 K = 50
23
24 total = 0
25 cmap = plt.get_cmap('viridis')
26 norm = plt.Normalize(vmin=0, vmax=K)
27
28 for k in range(K):
29     #the fourier transform calculates the integral with exp(-i*omega*t)
30     #so the signs and real/imag relation to the A and B of the sines and
31     #cosines is slightly different
32     A = -Y[k].imag/N
33     B = Y[k].real/N
34     #for oscillating terms we have to double the amplitude for the same
35     #reason as when we were calculating A's and B's directly
36     if k > 0:
37         A *= 2
38         B *= 2
39     # the 2/N factor comes from the particular normalization used to
40     # define fourier transform in SciPy. Various definitions exist.
41     sint = A*np.sin(2*np.pi*frequencies[k]*ts)
42     cost = B*np.cos(2*np.pi*frequencies[k]*ts)
43
44     ax.plot(ts, sint, color=cmap(norm(k)))
45     ax.plot(ts, cost, '--', color=cmap(norm(k)))
46
47     total += sint + cost

```

```

48 ax.plot(ts, total, '--', color='r')
49
50 # However, to get the total sum of the first K fourier terms, we don't
51 # have to use use the loop explicitly. We can simply calculate the inverse
52 # fourier transform with frequencies larger than K to zero
53 #IRfft = inverse real fft
54 Y_K = np.copy(Y)
55 Y_K[K:] = 0
56 # this calculates the inverse fourier transform, i.e., the same sum were
57 # were building up in the total variable in the for loop
58 total_irfft = fft.irfft(Y_K, len(ts))
59 #
60 # for INVERSE REAL fft, there is some confusion about what should be the
61 # correct length of the inverse transform, so it's best to specify the
62 # expected length of the output explicitly.
63 ax.plot(ts, total_irfft, ':', color='g', lw=3)

```

Listing 19: Fourier coefficient of a square wave calculated using FFT

One particularly important Fourier transform is that of the exponentially decaying oscillation, i.e.

$$s(t) = e^{-t/\tau} \sin(2\pi ft), \quad (13)$$

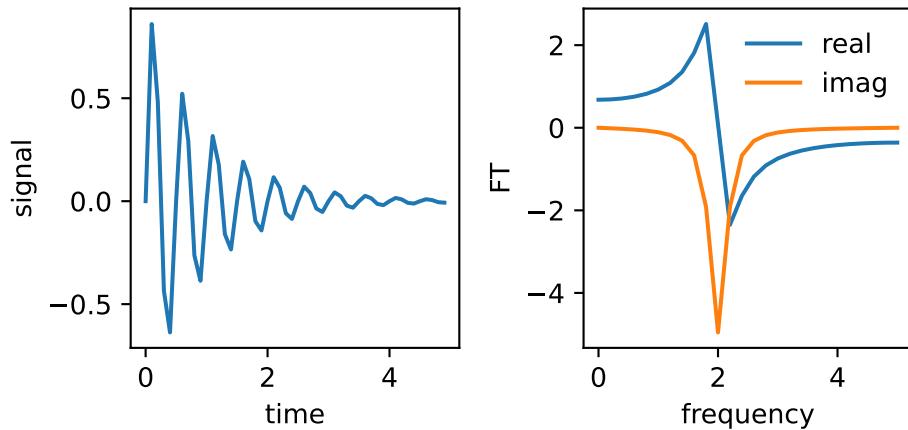
which yields the complex lorentzian which is the response of the linear harmonic oscillator, see Sec. A. The decaying oscillation is, of course, the motion of the damped unforced linear harmonic oscillator. Example:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.fft as fft
4
5 fig, (axt, axf) = plt.subplots(1, 2, figsize=(5, 2.5))
6
7 tau = 1
8 dt = 0.1
9 t = np.arange(0, 5, dt)
10 y = np.exp(-t/tau)*np.sin(2*np.pi*2*t)
11 axt.plot(t, y)
12
13 Y = fft.rfft(y)
14 freq = fft.rfftfreq(len(t), dt)
15 axf.plot(freq, Y.real, label='real')
16 axf.plot(freq, Y.imag, label='imag')
17
18 axt.set_xlabel('time')
19 axt.set_ylabel('signal')
20
21 axf.set_xlabel('frequency')
22 axf.set_ylabel('FT')
23 axf.legend(loc='best', frameon=False)
24 fig.tight_layout()
25
26 fig.savefig('../decaying_exponential.pdf')

```

Listing 20: Fourier transform of a decaying sine wave.



**Intermezzo 7: Storing data and metadata in numpy binary files.** So far, when loading data from a disk, we encountered only simple human-readable text files. This is a very limiting way to store data: the data have to have the form of a rectangular table, the files take up more space than necessary (e.g., text 1.234567890 requires 11 bytes of memory but a `float` representing the same number only needs 8), and it is cumbersome to store metadata.

Many file formats address these issues (e.g., HDF5 is popular). Here we will use a solution that does not require any further external libraries – storing dictionaries in compressed binary `.npy` files, e.g. saving

```

1 raw_data = np.linspace(0, 1, 50)
2 my_data = {
3     'date of measurement': '20241224',
4     'was Mercury in retrograde': False,
5     'data': raw_data
6 }
7 np.save("my_data.npy", my_data)

```

and loading

```

1 my_data = np.load("my_data.npy", allow_pickle=True).item()
2 print(my_data['date of measurement'])
3 print(my_data['data'])

```

**Pickling** is a Python term for storing almost any Python object on disk as binary data. Loading a pickled object can in, some circumstances, be a security risk (i.e., avoid reading pickled data you downloaded from weird places on the internet), therefore we have to allow it explicitly. Functions `np.load` and `np.save` are working with arrays, therefore `np.load()` returns an array of length 1 where our dictionary is the only element. The method `array.item()` returns `array[0]` if `array` is length 1 or raises a `ValueError` exception otherwise. Its purpose is to semantically indicate that we expect the array to only have one element and if it does not, something is wrong.

**Exercise 5.1.** Read data from the directory `timeseries_data` and plot the time-dependence of the approximate mean temperature during the measurement. Read the files using `d = np.load(filename, allow_pickle=True).item()`. The temperature at the beginning and end of the measurement period is `d['Ti']` and `d['Tf']`, respectively.

*Hint:* To get the timestamp (number of seconds since 00:00 1.1.1970) you can use `time.mktime(time.strptime(fn, 'DM_%Y%m%d-%H%M%S.npy'))`, with `fn` the filename.

**Exercise 5.2.** Plot the time series and spectral density of the pulse used to drive the resonance in the `data/timeseries_data` binary data files (see Storing and loading binary files 7) using `scipy.signal.periodogram` and using the (r)FFT functions from `scipy.fft`. The pulse can be loaded as `d['pulse']`, the number of points can be obtained as `d['samples']` (same as `len(d['pulse'])`) and the sampling rate as `d['rate']`. Plot only one file (the pulse is the same for all).

**Exercise 5.3.** Plot the time series (as a function of actual time) of the resonator response (`d['timeseries']`) and its frequency dependence (both real and imaginary components) for the file corresponding lowest temperature in the `timeseries_data` directory. Calculate the frequency response as a ratio of the Fourier transforms of the resonator response and the driving pulse. Plot only frequencies  $|f| < 3000$ , the resonance is in the range of approximately 2000 - 2400 Hz.

**Exercise 5.4.** Plot the magnitude (i.e., absolute value) of the response of the resonator  $r$  as a 2D heat map plot as a function of both frequency and temperature (i.e., each line in the 2D plot should be a single spectrum corresponding to a single temperature).

**Exercise 5.5.** As exercise 4, but average together all spectra closer than 50 mK in temperature (this smoothing technique is called moving average or adjacent averaging).

**Amplitude, power, and decibel** In signal processing we often talk about attenuation or gain (or amplification) of a system, e.g., wiring, amplifiers or attenuators. Gain is defined as the ratio of output to input signal and is often measured in decibels defined as

$$g = 10 \log_{10} \frac{s_{\text{out}}}{s_{\text{in}}} [\text{dB}] \quad (14)$$

or for power, i.e.,  $s = V^2$ ,

$$g = 20 \log_{10} \frac{V_{\text{out}}}{V_{\text{in}}} \quad (15)$$

For "absolute" quantities measured in dB (e.g., sound amplitude is common) the measurement is defined with respect to some agreed upon reference value. For sound the acoustic power (i.e., pressure squared) is measured relative to  $20 \mu\text{Pa}$  in air. In electronics, particularly radiofrequency (RF) engineering, a common unit is dBm, where 0 dBm is equivalent to 1 mW of power, i.e., for a  $50 \Omega$  load about  $0.224 \text{ V}_{\text{RMS}}$ .

For amplitudes, doubling the signal corresponds to +3 dB and halving the signal to -3 dB and for power it is +6 dB and -6 dB.

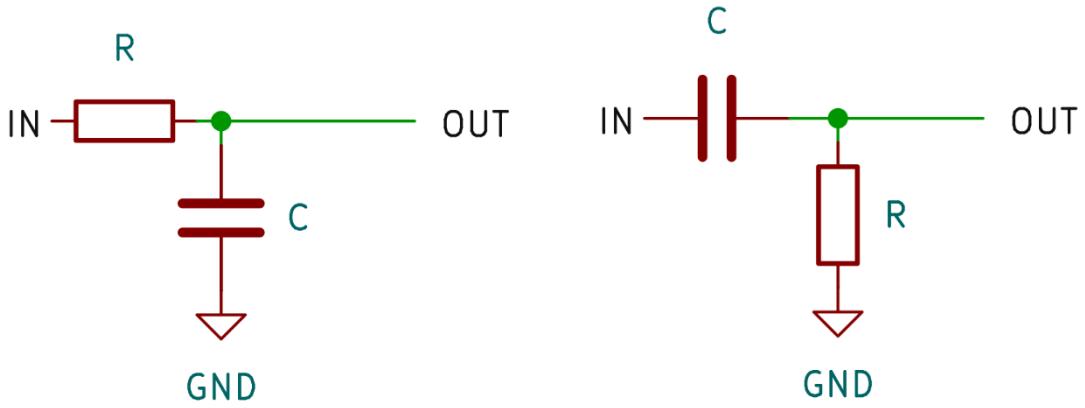


Figure 1: Low-pass (left) and high-pass (right) RC filter.

### 5.3 Filters

Filtering is a procedure by which we remove certain frequency ranges from the input signal. These procedures are general, but the simplest ones are based on the analogy with simple electronic RC filters, see Fig. 5.3. Depending on the arrangement of the resistor and capacitor we create a circuit which either attenuates low or high frequencies.

Using the impedance of a capacitor  $Z_C = (i\omega C)^{-1}$  for voltage oscillating at angular frequency  $\omega$ , we get for the low-pass filter

$$\frac{V_{\text{out}}}{V_{\text{in}}} = \frac{1}{1 + i\omega RC}, \quad (16)$$

and for the high pass

$$\frac{V_{\text{out}}}{V_{\text{in}}} = \frac{i\omega RC}{1 + i\omega RC}. \quad (17)$$

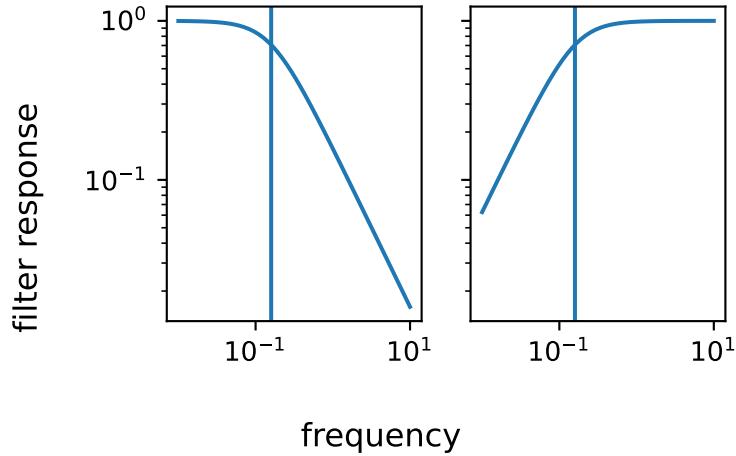
The above expressions are called transfer functions of the filter. Quantity  $RC = \tau$  is called the time constant. The corner frequency  $f_c = 1/(2\pi\tau)$  is the frequency when the filter starts acting. We can plot the response of the filters with the following code

```

1 tau=1
2 freqs = np.logspace(-2, 1)
3 low_pass = 1/(1 + 1j*2*np.pi*freqs*tau)
4 high_pass = 1j*2*np.pi*freqs*tau/(1 + 1j*2*np.pi*freqs*tau)
5 fig, (axL, axH) = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(4,2.5))
6 axL.loglog(freqs, abs(low_pass))
7 axH.loglog(freqs, abs(high_pass))
8 axL.axvline(1/(2*np.pi*tau))
9 axH.axvline(1/(2*np.pi*tau))

```

Listing 21: RC filter response



The figure shows the absolute value of the transfer function for the low-pass and high-pass RC filters, respectively. The vertical line is the corner frequency.

Notice that the transfer functions are functions of frequency and the circuits they represent are linear, i.e. they act on each frequency independently. Therefore, if  $s(t)$  is our signal and  $\hat{s}(f)$  is its Fourier transform, the Fourier transform of the signal filtered with a transfer function  $H(f)$  is simply

$$\hat{s}'(f) = H(f)\hat{s}(f), \quad (18)$$

i.e., filters in frequency domain simply multiply the spectrum. In time domain  $s'(t) = \mathcal{F}^{-1}[H\hat{s}]$ ; Fourier transform of a multiplication is a convolution, i.e.

$$s'(t) = (h * s)(t) \equiv \int_{-\infty}^t s(t')h(t-t')dt, \quad (19)$$

where  $h$  is the inverse Fourier transform of  $H$  and is called the convolution kernel. Note that FFT is often the fastest way to calculate a convolution, although we do not have to implement it ourselves since there is `scipy.signal.convolve(a, b)`, which calculates the convolution of signals `a` and `b`.

For the low-pass RC filter it can be proven that

$$h(t) = \begin{cases} e^{-t/\tau} & \text{for } t > 0 \\ 0 & \text{for } t < 0 \end{cases} \quad (20)$$

which can also be easily implemented for streaming data, for which it is often called *exponential smoothing*.

Apart from low-pass and high-pass filters there are also band-pass and band-stop filters. Band-pass only lets through a certain frequency band (i.e., an interval) and band-stop lets through everything except for a certain frequency band. You can imagine band-pass as a high-pass followed by a low-pass in series and band-stop as a low-pass and high-pass in parallel with corner frequencies given by the pass or stop band.

The simple RC filters above are so-called first order. The order of the filter indicates how fast it cuts the signal outside of the *pass band* (i.e., the interval of frequencies which the filter lets through). This is often measured in decibels per octave (dB/oct) which indicates by how many decibel is the signal attenuated if its frequency doubles, for a low-pass filter, or halves for high-pass filter, sufficiently far from the corner frequency. Both of the RC filters are 6 dB/oct, since every doubling of frequency removes 6 dB of transmitted power.

Faster filters can be constructed both electronically and digitally, but as you saw in the examples with the step function, sharp cutoff in the spectrum leads to oscillations, which are typically undesirable. The filters that are maximally flat in the pass-band are the Butterworth filters, named after

Stephen Butterowrth, which have amplitude gain

$$G_n(\omega) = \frac{1}{\sqrt{1 + \frac{\omega^2 n}{\omega_c^{2n}}}},$$

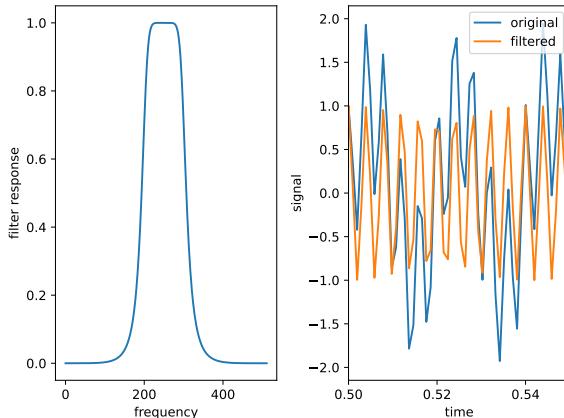
where  $\omega_c$  is the corner (angular) frequency. We can construct these filters using `scipy.signal.iirfilter` and apply it to our signal using `scipy.signal.sosfilt`

```

1 import scipy.signal as sig
2 N = 4 # filter order, this would be 24 dB/oct
3 fs = 1024 # sampling frequency
4 Wn = [200, 300] # corner frequencies of a band-pass filter
5
6 # sos = second order sections
7 # internal representation of the transfer function
8 # recommended by scipy
9 sos = sig.iirfilter(N, Wn, fs=fs, btype='bandpass', output='sos')
10
11 #create some signal and filter it
12 ts = np.arange(0, 1, 1/fs)
13 ys = np.sin(2*np.pi*50*ts) + np.cos(2*np.pi*250*ts)
14 ys_filt = sig.sosfilt(sos, ys)
15
16 freq, H = sig.sosfreqz(sos, fs=fs)
17 axH.plot(freq, abs(H))
18 axsig.plot(ts, ys, label='original')
19 axsig.plot(ts, ys_filt, label='filtered')
```

Listing 22: Using IIR filter.

:



**Exercise 5.6.** Write a function, which will filter the input signal using

1. low-pass RC filter with adjustable cutoff frequency
2. high-pass RC filter similarly to 1.
3. Butterworth band-pass filter constructed using `scipy.signal.iirfilter()` and applied using `scipy.signal.lfilter()`

For 1. and 2. plot the the filter transfer function. Test the filtering on the attached `noisy_data.npy` and plot the signal in both the time and frequency domain before and after filtering. For the

low-pass filter, filter out everything above 100 Hz. For high-pass, everything below 1000 Hz and for band-pass leave only the range (4310, 4330) Hz.

**Exercise 5.7.** Demodulate the cleaned-up signal from Exc. 6 (using Butterworth band pass filter) on the carrier frequency 4321 Hz (multiply by  $e^{i\omega t}$  and low-pass filter) and plot the envelope modulating the carrier wave.

**Exercise 5.8.** Calculate the signal envelope from Exc. 7 using the Hilbert transform.

## 5.4 Interpolation and smoothing.

TODO: `scipy.ndimage.gaussian_filter()`, `scipy.signal.savgol_filter()`, `scipy.interpolate.interp1d()`, `scipy.interpolate.UnivariateSpline()`

## 6 Communication with instruments

Instrumentation can be connected to a computer through a wide variety of interfaces, including USB, GPIB, RS-232 or Ethernet or emulated serial port over USB. Each of these interfaces require specific OS support, drivers or libraries to use. This was, fortunately, to a large degree solved by the Virtual Instrument Software Architecture (VISA) which abstracts and unifies most of the interfaces into VISA "sessions".

Most of the communication over VISA sessions is text-based. A text command is sent to an instrument, the instrument performs some action and, optionally, replies a text response back. To use VISA we will use the `pyvisa` python module. This, however, is only a python interface to the system VISA library. There are several implementations of the VISA library (sometimes called the VISA backend) from multiple vendors, e.g., from National Instruments (NI-VISA, most common) or Keysight companies. We will be using an open-source fully-python VISA backend `pyvisa-py`. For our purposes will also need `pyserial`. If you want to use USB, Ethernet or GPIB instruments, one of the free-to-download proprietary implementation is probably the best way forward.

In this course we will be using the Raspberry Pi Pico in place of an actual instruments. The Pico is programmed to respond to several commands that roughly follow the SCPI syntax (see Appendix B). A simple VISA program that communicates with an instrument might look like this

```
1 import pyvisa as vi
2
3 rm = vi.ResourceManager()
4
5 print("Available resources:")
6 resources = rm.list_resources()
7 for k, res in enumerate(resources):
8     print(f'VISA address {k}: {res}')
9
10 #assuming that our pico is on the last address (usually the case)
11 pico = rm.open_resource(resources[-1],
12                         read_termination='\n',
13                         write_termination='\n')
14 print("Identification of the last device:")
15 pico.write('*IDN?')
16 resp = pico.read()
17 print(resp)
18
19 # performs the same in one step
20 # query = write + read
21 print("IDN via query:")
22 print(pico.query("*IDN?"))
23
24 pico.close()
25 rm.close()
```

On the first line we import the `pyvisa` module. Creating new sessions is done via the Resource Manager `rm`. In the VISA naming convention instruments are called 'resources' and they are identified by a resource address, which will typically indicate the type of interface used, e.g., addressed that begin with `ASRL` are serial instruments, `GPIB` and `USB` etc.

The resource manager provides the `list_resources()` method which returns a list of all available resources. For USB and GPIB these are usually the instruments actually connected to the computer. For serial interface, all available ports might be listed regardless of whether something is connected to them or not.

To start a VISA session, we use the `open_resource()` method of the resource manager. In addition, here we specify the read and write termination, which are characters that indicate that the transmission is finished (think of army movies when they say "over" to a walkie-talkie). The new line '`\n`', carriage return '`\r`' and their combination '`\r\n`' are common. Our pico expects the commands to be terminated by new line '`\n`' and terminates its responses with a new line as well. For other instruments, this is

something you have to find in the manual, but '`\n`' is the most common.

The open VISA session is represented by the object `pico` returned by `open_resource()`, whose most important methods are `write()`, `read()` and `query()`, which is simply a write immediately followed by a read.

Most scientific instruments will communicate via text strings which usually adhere to the syntax specified by Standard Commands for Programmable Instruments (SCPI), which tries to define a common syntax for instruments of similar type. SCPI commands typically have the format

```
1 :COMMAND:SUBCommand:SUBCommand ARG1 ARG2 ...
```

where the capitalized substrings can be used as a shorthand. For example, on most digital multimeters (e.g., the venerable Keithley 2000 series), command

```
1 :MEAS:VOLT:DC?
```

will measure the DC voltage. The question mark at the end indicates that the command returns a value. Instruments that support SCPI in addition support several basic commands such as `*IDN?`, which returns an identification string, `*STB?`, which returns a single byte with various status bits of the instrument or `*RST`, which resets the instrument. At the end, both the VISA session and the resource manager should be properly closed using the `close()` method.

In most cases, nothing other than the resource address is necessary for the `rm.open_resource()` method to successfully open the session. However, especially for some older instruments connected via RS-232, some additional information is often required. Assuming that a device was opened as `dev = rm.open_resource(address)`, before we can start communicating we might need to set the following:

**baud rate** Number of changes to the communication signal per second (i.e., changes from low to high voltage). Can be set via `dev.baud_rate` to an integer.

**data bits** Number of actual data bits in a single "packet" of data exchanged between the instrument and computer (usually 8). Can be set via `dev.data_bits` to an integer.

**stop bits** The number of bits indicating beginning and end of a character transmitted on serial cable (usually 1). Can be set via `dev.stop_bits` to one of `pyvisa.constants.StopBits.SB` where `SB` is `one`, `one_and_a_half` or `two`.

**parity** A simple check for corrupted transfer. Some instruments send a bit indicating whether the sum of bits in the last character (i.e., its ASCII code) was odd or even. Can be set via `dev.parity` to one of `pyvisa.constants.Parity.P` where `P` is `none` (most common, no parity check), `even` or `odd`.

For example, if we had an instrument on address `COM16` and its manual says that it expects baud rate of 19200, 8 data bits, one stop bit and no parity we could open the communication as

```
1 import pyvisa as vi
2
3 rm = vi.ResourceManager()
4 dev = rm.open_resource('COM16')
5 dev.baud_rate = 19200
6 dev.data_bits = 8
7 dev.stop_bits = vi.constants.StopBits.one
8 dev.parity = vi.constants.Parity.none
9
10 dev.query('*IDN?')
11 ...
```

**Exercise 6.1.** Create a program that automatically finds the pico if it is connected to the computer. Send the `*IDN?` query to every available VISA resource and find the one that responds `'PICO'`. Resources (serial ports) which are not connected to any instrument will raise a timeout

exception, which must be handled without crashing the program.

**Exercise 6.2.** Open the communication to the pico and make the white and blue LEDs flash in a "police car"-like pattern (i.e., two quick flashes of white followed by two quick flashes of blue) on repeat. Use the `:LED n x`, where  $n = 0 \dots 4$  is the LED number and  $x$  is either 0 or 1, which turns the LED off or on. Make sure that when your program is killed (either by Ctrl-C or the interrupt button in Spyder) all LEDs are turned off.

**Exercise 6.3.** Write a program that will indicate the current temperature (obtained using `:READ:T?` command) using the on-board LEDs. Map the range  $20\text{-}27^\circ\text{C}$  to 0 to 5 LEDs ON. If the temperature exceeds  $28^\circ\text{C}$ , raise a `RuntimeError`, which ends the program. Make sure none of the LEDs stay on when the program ends.

**Exercise 6.4.** Write a "digital spirit level", i.e., indicate the current tilt along the y-axis (length-wise along the PICO) using the LEDs (you can decide on whatever you think is the best way to use the LEDs). Note that the accelerometer is not soldered to the PCB perfectly level. Assume that when the program is run, the pico is level and use the acceleration vector measured at the beginning as the reference value.

**Exercise 6.5.** Write an object interface to the pico, which should be initialized using only the resource manager instance and should find the correct address on its own, i.e.,

```
1  import pyvisa as vi
2
3  class Pico:
4      def __init__(self, rm):
5          ...
6      def led(self, led_id, onoff):
7          #should raise ValueError if led_id > 4
8          ...
9      def getT(self):
10         ...
11     def getP(self):
12         ...
13     def getACC(self):
14         ...
15     def getGYR(self):
16         ...
17     def close(self):
18         "Turn off all LEDs and close the session."
19         ...
20
21     rm = vi.ResourceManager()
22     pico = Pico(rm)
23     pico.led(2, 1) #turns the middle LED ON
24
```

## 6.1 Context Management Protocol

Particularly with instruments that control actual laboratory hardware, proper shutdown and cleanup, even in the event of a software error, is very important. Imagine an oven, which keeps heating up, or

a motor which keeps spinning uncontrollably because of a typo in your program. We saw in Exercises 27 – 29 that a controlled shutdown can be handled using the `finally` clause after a `try` block. This is suboptimal, because we have to remember to write the `finally` block, which is often does not change much, in every program in which we use a given instrument.

Python has a solution for this, which we already saw, called *context management protocol*, which is what the `with` statement uses. Recall using files,

```
1 with open("hello.txt", 'w') as file:
2     file.write('Hello Context Management.')
```

where the `with` statement makes sure that after we are done using the file it is properly `close()`d

To use the context management with our objects we only need to define two special methods, `__enter__()` and `__exit__()`, which are run at the beginning and end of the `with` statement. The enter method should return the object we want to use (i.e., file in the above example), often it simply returns `self`. The exit method should do all the necessary cleanup. It takes three additional arguments besides `self` which contain information about whether an exception occurred and what type. Most of the time we can ignore the exceptions and simply let it propagate. If we return `False` from the exit method, the exception is suppressed.

Take a simple example. Say we want to have file-like object that can be used with `with` as an ordinary file but the method `write()` also supports NumPy arrays. We could do the following

```
1 import numpy as np
2
3 class MyNumpyFile:
4     def __init__(self, filename, mode='w'):
5         print("Opening file.")
6         self.file = open(filename, mode)
7
8     def write(self, data):
9         np.savetxt(self.file, data)
10
11    def __enter__(self):
12        print("Entering with statement")
13        return self
14    def __exit__(self, *exc):
15        print("Closing.")
16        self.file.close()
17
18 print("I'm about to open the file.")
19 with MyNumpyFile("my_array.txt", 'w') as file:
20     file.write(np.arange(5))
21 print("The end.")
```

which prints

```
I'm about to open the file.
Opening file.
Entering with statement
Closing.
The end
```

Notice that the method `__enter__()` simply returns `self`. This is because the context manager and the object representing the file are the same. In the `__exit__()` method we simply lump together all information about any exception that might have occurred in the `*exc` argument list and ignore them, only doing the necessary cleanup. Not returning anything is equivalent to returning `None`, which is not `False`, so if any exception occurred it will simply continue on its merry way to the nearest `except` clause.

**Exercise 6.6.** Extend the Pico class to support the context management protocol.

For completeness, the example with `open()` is essentially equivalent to the following code

```

1 manager = open("hello.txt", 'w')
2 file = manager.__enter__()
3 try:
4     file.write("Hello Context Management.")
5 except:
6     if not manager.__exit__(exception_info)
7         raise
8 else:
9     manager.__exit__(None, None, None)

```

## 6.2 Troubleshooting common issues

Communication with external hardware can fail for several reasons. Assuming that the hardware itself and the connecting cables are functioning some common causes of problems are:

**exception in opening VISA session** Depending on the operating system and VISA backend used, `rm.open_resource()` can raise an exception claiming that access to the resource is denied or that the device is busy. This is most commonly caused by previously opened VISA session, which was not closed. Only one VISA session with a given instrument is allowed at any one time.

This is a common problem in IDEs such as Spyder, which run files in an interactive console, where variables remain accessible even if the program exits unexpectedly. Either ensure that sessions are closed properly even if the program crashes (i.e., `finally` or context management) or close the console in which the program was run.

**resource opens but communication times out** An attempt to read will time out if the device does not respond in an expected way, most often because of a mistake in the command. If the command sent is definitely correct, timeout exception is usually the symptom of incorrectly set up VISA session configuration, e.g., the read/write termination characters, or some of the serial port configuration options mentioned above.

**resource opens, and no timeout occurs, but the response is wrong** Consider the following code

```

1 import pyvisa as vi
2 rm = vi.ResourceManager()
3 pico = rm.open_resource(rm.list_resources()[-1])
4 pico.read_termination = '\n'
5 pico.write_termination = '\n'
6 pico.write(':READ:T?')
7 print(pico.query('*IDN?'))

```

which prints the temperature reading rather than the expected identification string. This is because the response to `:READ:T?` was never read, so it remained in the input buffer until the earliest call to `read` which came in the `query()`. If the backend and resource type supports it, you can call `pico.clear()` to discard the buffers, or you can call `pico.read(timeout=0)` and discard the timeout exception if the buffer happens to be empty

## 7 Parallel Execution

Python distinguishes between two types of parallelism: multithreading and multiprocessing. Threads provide only an illusion of true parallelism: for a single Python process (i.e., when you run `python your_file.py`) only one thread runs at the same time, but execution switches between multiple threads to create an illusion of parallelism. Even if your script runs on a multi-core (or multi-CPU) machine, only one core will be used<sup>11</sup>. Multiprocessing, on the other hand, can run multiple python processes truly in parallel, simultaneously on more CPU cores if available.

While threads might seem pointless, they are often easier to use and especially for applications which wait on either hardware I/O, network or user interaction they are often the better choice. Multiprocess applications can be faster for sufficiently big problems, however, for short-running programs they are often *slower* because handling of the multiple process introduces *overhead* which can be comparable to the solution of the problem itself.

### 7.1 Multithreading

Threads are represented by `Thread` objects from the `threading` module. Threads execute a given *target* function when they are *started* and after they are finished they have to be *joined* back to the parent thread. Example usage:

```
1 import threading as th
2
3 def func(thread_name):
4     print(f"I'm inside {thread_name}")
5
6 threads = []
7 # create and start 5 threads and save them to a list
8 for k in range(5):
9     thread = th.Thread(target=func, args=(f"thread {k}",))
10    thread.start()
11    threads.append(thread)
12
13 # wait for all threads to finish
14 for thread in threads:
15     thread.join()
```

However, often an object-oriented approach is more convenient than target functions. We can define our own thread objects by simply inheriting from the `Thread` and defining the `run()` method. A functionally equivalent example to the above:

```
1 import threading as th
2
3 class MyThread(th.Thread):
4     def __init__(self, thread_name):
5         # It is necessary to initialize the parent class as well
6         super().__init__()
7         self.thread_name = thread_name
8     def run(self):
9         print(f"I'm inside {self.thread_name}")
10
11 threads = []
12 for k in range(5):
13     thread = MyThread(f"thread {k}")
14     thread.start()
15     threads.append(thread)
16
17 for thread in threads:
18     thread.join()
```

---

<sup>11</sup>Since Python version 3.13, the *global interpreter lock* (GIL) which causes threads to run without true parallelism can be optionally disabled, although the GIL will remain the default behaviour for the foreseeable future. See here for more details

Threads need to be able to communicate with each other and share resources predictably. Consider the VISA `query()` function, which is simply `write()` immediately followed by a `read()`. If there are two threads, communicating with the same pico device, one executes `query(':READ:P?')` and the other `query(':READ:T?')` there is a chance that the actual order of executed reads and writes will be

```

1 #thread 1          1   # thread 2
2 write(':READ:P?') 2   # thread 1 running
3 # thread 2 running 3   write(':READ:T?')
4 # thread 2 running 4   read()
5 read()             5   # thread 1 running

```

and the thread that asked for pressure will get the temperature and vice versa. This class of bugs is called *race conditions* – i.e., two threads are “racing” each other to compete for the resource, who wins is random. To avoid this, we need to tell Python that we should not be interrupted for a while until we are done with writing and reading. This is achieved using *locks* (or mutexes, from MUTual EXclusion) which are available in the `threading` module as a `Lock` class as

```

1 from threading import Lock
2 lock = Lock()

1 # thread 1          1   # thread 2
2 lock.acquire()      2   # thread 1 running
3 # thread 2 running 3   lock.acquire() # blocks, until lock is released
4 write(':READ:P?')   4   # thread 1 running
5 read()              5   # thread 1 running
6 lock.release()     6   # thread 1 running, lock.acquire() returns
7 # thread 2 running 7   write(':READ:T?')
8 # thread 2 running 8   read()
9 # thread 2 running 9   lock.release()

```

Note that we are passing the object created by the `Lock` class rather than the class itself. This is the case for all synchronisation and communication mechanisms – `Locks`, `Events` and `Queues` and others. Locks support the context management protocol, so typically we use them in the `with` statement rather than calling the `acquire()` and `release()` directly. A more complete example,

```

1 import pyvisa as vi
2 from threading import Thread, Lock
3
4 rm = vi.ResourceManager()
5 # Assuming that our pico is on the last address (usually the case)
6 pico = rm.open_resource(rm.list_resources()[-1],
7                         read_termination='\n',
8                         write_termination='\n')

9
10 def readP(lock):
11     with lock: # lock is acquired
12         # this will run uninterrupted
13         P = pico.query(':READ:P?')
14         print(P)
15     #lock is released
16
17 def readT(lock):
18     with lock:
19         T = pico.query(':READ:T?')
20         print(T)
21
22 lock = Lock()
23 t1 = Thread(target=readP, args=(lock,))
24 t2 = Thread(target=readT, args=(lock,))

25
26 t1.start()
27 t2.start()
28 t1.join()
29 t2.join()

```

Note that if we were to use two locks to lock two separate resources we could get into a situation where two locks wait on each other forever such as this

```

1 # thread 1          1 # thread 2
2 # tries to acquire lock1 and lock2 2 # tries to acquire lock2 and lock1
3 # in this order      3 # in this order
4 lock1.acquire()      4 # thread 1 running
5 # thread 2 running 5 lock2.acquire()
6 # thread 2 running 6 lock1.acquire() # blocks forever
7 lock2.acquire() # blocks forever 7 # thread 1 running

```

which is called a *deadlock*. Be extra careful when you use more than one lock.

The simplest method of communicating between threads are global variables. However, this can quickly become messy and confusing therefore it is typically better to use primitives intended for inter-thread communication. The simplest is `threading.Event` which is a boolean flag which can be set by one thread and reacted on by another, e.g.,

```

1 import pyvisa as vi
2 from threading import Thread, Lock, Event
3 import time()

4
5 rm = vi.ResourceManager()
6 pico = rm.open_resource(rm.list_resources()[-1],
7                         read_termination='\n',
8                         write_termination='\n')

9
10 def keep_reading_P(lock, end_event):
11     #keep running until the end_event is set
12     while not end_event.is_set():
13         with lock: # lock is acquired
14             # this will run uninterrupted
15             P = pico.query(':READ:P?')
16             print(P)
17             #lock is released
18             time.sleep(1)

19
20 lock = Lock()
21 end = Event()
22 thr = Thread(target=keep_reading_P, args=(lock,end))
23 t0 = time.time()
24 thr.start()
25 # measure for five second and then signal the thread to end
26 while True:
27     if time.time() - t0 > 5:
28         end.set()
29         break
30     time.sleep(0.1)
31 thr.join()

```

To send data between threads `Queues` from the module `queue` are useful. We can put a value inside the queue in one thread using the `put()` method and take it out elsewhere using the `get()` method which blocks if the queue is empty. We can also check whether a queue is empty using the `empty()` method. A minimal example

```

1 from threading import Thread
2 from queue import Queue
3
4 # function that the thread will run
5 def thread_func(q):
6     while True:
7         # waits until a message becomes available
8         msg = q.get()
9         print(f"Received message {msg}")
10        if msg == 'quit':
11            break
12
13 # create the queue and start the thread
14 q = Queue()
15 thread = Thread(target=thread_func, args=(q,))

```

```

16 thread.start()
17
18 messages = ['hello', 'world', 'quit']
19 for msg in messages:
20     # send the message
21     q.put(msg)
22
23 thread.join()

```

A more complete example that reads data from the pico in one thread and plots it another

```

1 import pyvisa as vi
2 import matplotlib.pyplot as plt
3 from threading import Thread, Event
4 from queue import Queue
5 import time
6
7 # A thread that plots data received through a queue
8 class Plotter(Thread):
9     def __init__(self, queue):
10         # Initialize the parent class
11         super().__init__()
12
13         # Create a figure and axis for plotting and
14         # data storage
15         self.fig, self.ax = plt.subplots()
16         self.xdata = []
17         self.ydata = []
18         self.line, = self.ax.plot(self.xdata, self.ydata, '-o')
19
20         # queue for receiving data and an event to signal the end
21         self.queue = queue
22         self.end = Event()
23
24         # replot the data and update axes limits
25     def update_plot(self):
26         if len(self.xdata) > 0:
27             self.line.set_xdata(self.xdata)
28             self.line.set_ydata(self.ydata)
29             xmin, xmax = min(self.xdata), max(self.xdata)
30             ymin, ymax = min(self.ydata), max(self.ydata)
31             xmid = 0.5*(xmin + xmax)
32             ymid = 0.5*(ymin + ymax)
33             dx = xmax - xmin
34             dy = ymax - ymin
35             self.ax.set_xlim(xmid - 0.55*dx, xmid + 0.55*dx)
36             self.ax.set_ylim(ymid - 0.55*dy, ymid + 0.55*dy)
37             # plt.draw()
38
39     def pull_data(self):
40         # keep reading the data from the queue as long
41         # as anything is available
42         while not self.queue.empty():
43             # if get() is called on an empty queue, it blocks
44             # until something becomes available (or a timeout occurs)
45             data = self.queue.get()
46             print("Received ", data)
47             # we can send anything through the queue, for example
48             # either a data point or a command to quit
49             match data:
50                 case (x, y):
51                     self.xdata.append(x)
52                     self.ydata.append(y)
53                 case 'quit':
54                     print("Quitting")
55                     self.end.set()

```

```

56
57     def run(self):
58         while not self.end.is_set():
59             self.pull_data()
60             self.update_plot()
61             time.sleep(0.5)
62
63
64 rm = vi.ResourceManager()
65 # change the address to match your system
66 pico = rm.open_resource('ASRL/dev/ttyACM1::INSTR',
67                         read_termination='\n',
68                         write_termination='\n')
69
70 data_queue = Queue()
71 plotter = Plotter(data_queue)
72 plotter.start()
73 try:
74     t0 = time.time()
75     while True:
76         t = time.time() - t0
77         P = float(pico.query(':READ:P?'))
78         print("Sending ", t, P)
79         data_queue.put((t, P))
80         # the following line updates all open matplotlib plots
81         # it MUST be run from the main thread
82         plt.pause(0.01)
83 finally:
84     data_queue.put('quit')
85     plotter.join()

```

**Exercise 7.1.** Write a program that will flash all 5 LEDs with intervals of 0.1, 0.2, 0.5, 1, and 2 s.

**Exercise 7.2.** Write a program that will show an updating plot of pressure. While the program runs, it should be able to accept text commands, and it should support: clear, which clears the current plot and quit, which quits the program cleanly.

## 7.2 Multiprocessing

Multiprocessing can use multiple CPU cores, however, there are restrictions on what kind of variables can be shared between processes. We can create new processes using the `Process` class from the `multiprocessing` module in a very similar manner to threads. The `multiprocessing` module also provides synchronisation and communication facilities similar to `threading`, i.e., `Event`, `Queue` etc. However, you must use the classes in the `multiprocessing` module to communicate between processes. A minimal example, where the main process creates a set of processes and sends messages to them all through a shared `Queue`

```

1 from multiprocessing import Process, Event, Queue
2 from queue import Empty
3 import time
4
5 # The same as for threads:
6 # Inherit from Process, initialize the parent class in __init__
7 # and write your own run() method
8 class MyProcess(Process):
9     def __init__(self, procname, end, data):
10         super().__init__()
11         self.procname = procname

```

```

12     self.end = end
13     self.data = data
14 def run(self):
15     while True:
16         try:
17             msg = self.data.get(timeout=1)
18             print(f"{self.procname} received: {msg}")
19         except Empty:
20             print(f"{self.procname}: Nothing in queue")
21             time.sleep(0.1)
22
23         if self.end.is_set() and self.data.empty():
24             break
25
26 # Using processes is very similar to using threads
27 # However, we must use multiprocessing.Event and multiprocessing.Queue
28 # instead of the threading versions
29 # Also, note that, generally "if __name__ == '__main__':"
30 # guard is required to avoid recursive spawning of subprocesses
31 if __name__ == '__main__':
32     data_queue = Queue()
33     end_event = Event()
34     process_pool = []
35     for k in range(5):
36         p = MyProcess(f'proc{k}', end_event, data_queue)
37         p.start()
38         process_pool.append(p)
39
40     for k in range(10):
41         data_queue.put(f"message {k}")
42         time.sleep(0.2)
43
44     end_event.set()
45     for p in process_pool:
46         p.join()

```

The above code is an example of a *process pool*, which is often the simplest way to speed up problems which involve multiple independent calculations. Multiprocessing already provides a general process pool precisely for this task

```

1 from multiprocessing import Pool
2
3 def function(x):
4     return some_calculation(x)
5
6 z = [... data ...]
7
8 with Pool(6) as pool:
9     result = pool.map(function, z)

```

which applies the function `function(x)` to every element of the sequence (e.g., list, array, ...) in 6 parallel processes collecting the results.

A more complete example that calculates the Fibonacci sequence,

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from multiprocessing import Pool
4 import time
5
6 # A slow recursive function to compute Fibonacci numbers
7 def fib(x):
8     if x < 2:
9         return 1
10    return fib(x-1) + fib(x-2)
11

```

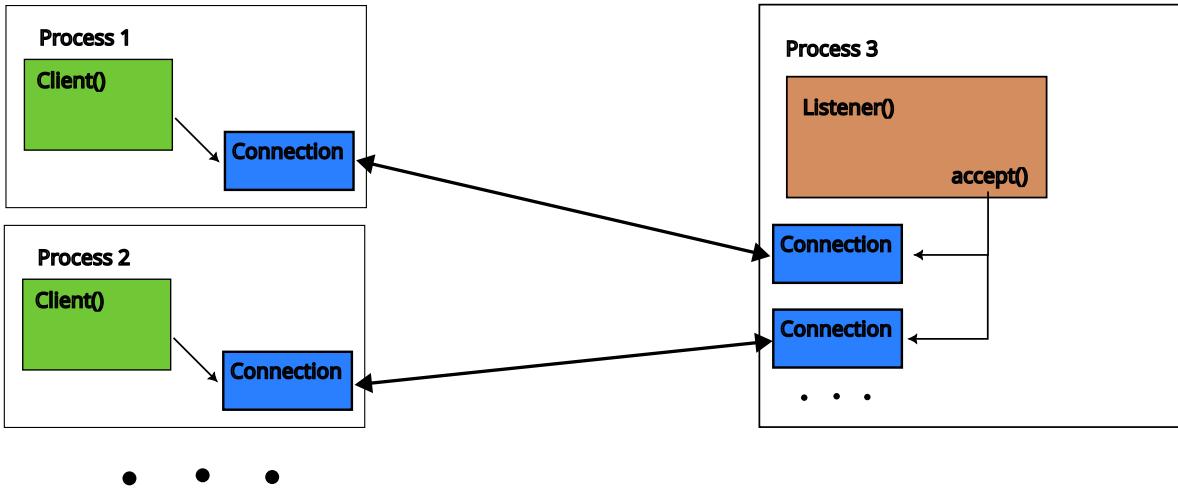


Figure 2: Relationships between Listeners, Clients and Connections

```

12 # Function to compute Fibonacci numbers in parallel using a pool of processes
13 def calc_fibs(z, nproc):
14     with Pool(nproc) as pool:
15         fibs = pool.map(fib, z)
16     return fibs
17
18 # Time the computation with different numbers of processes
19 def timeit(z, nproc):
20     t0 = time.time()
21     fibs = calc_fibs(z, nproc)
22     t1 = time.time()
23     print(f"Calculation with {nproc} processes took {t1 - t0:.3f} s")
24
25 z = np.arange(35)
26 timeit(z, 1)
27 timeit(z, 2)
28 timeit(z, 4)
29 timeit(z, 8)
30 timeit(z, 16)

```

notice that for large number of processes the calculation actually takes *longer*.

### 7.3 Inter-process Communication

So far we created new processes from a python script which we ran. However, any two python processes, i.e., separate runs of any python programs, can communicate with each other. There are several ways to achieve this, such as directly sharing memory (using `multiprocessing.shared_memory` module, see the documentation for example use with numpy arrays) or `sockets`. Socket mechanism is in some way provided by all operating systems as a general way of communication between processes, locally or over network, see Fig. 7.3. Generally, we establish a connection with an address (can be `localhost` if not communicating over network) and a port number. One process acts as the *listener* (i.e., server) which *accepts* connections from *clients* and the two can then send messages to and receive messages from each other.

The example below shows a simple echo server – a server that simply sends back to the client what it receives. It will only accept connections from the local computer on the port 6000. First, the listener object is created using the `with` statement, which is then used to start a separate thread which accepts the connections and responds to clients. The main thread will simply wait for the signal to quit. If we are supposed to quit, close the listener (we also need to unblock the `accept()` which would

wait forever) and quit. Note that the passwords are optional, if you do not specify the `authkey` for the `Listener`, you also do not have to specify it for `client`.

The client is in comparison much simpler. Simply connect to the listener with known address, port and password, and you can start sending and receiving data. Note that `send()` and `recv()` pickle and unpickle (recall pickling from `.npy` files), so almost all Python-objects can be sent.

To test the example below, run `python server.py` in one console and `python client.py` in another.

```
server.py:  
1  from multiprocessing.connection import Listener, Client  
2  from threading import Thread  
3  import time  
4  
5  # we will accept the connections in a separate thread  
6  def handle_connections(listener):  
7      while True:  
8          # listener.accept() blocks until someone  
9          # tries to connect  
10         try:  
11             # accept the connection, echo back what we receive  
12             # and then close the connection (automatically, using 'with')  
13             with listener.accept() as conn:  
14                 msg = conn.recv()  
15                 print(f"Received {msg}")  
16                 conn.send(f'Echo {msg}')  
17         except OSError:  
18             # OSError is raised when we try to accept  
19             # with a closed listener  
20             print("Stopping listening")  
21             break  
22  
23 # the address and port  
24 # 'localhost' is local computer, use empty string ''  
25 # if you want to access it over the network  
26 # port number 0 means automatic assignment by the OS  
27 address = ('localhost', 6000)  
28 password = b'password'  
29 with Listener(address, authkey=password) as listener:  
30     print('Address :', listener.address)  
31  
32     t = Thread(target=handle_connections, args=(listener,))  
33     t.start()  
34     # now the main thread will just idly sleep, waiting for  
35     # keyboard interrupt  
36     try:  
37         while True:  
38             time.sleep(1)  
39     except KeyboardInterrupt:  
40         print("Quitting...")  
41     # this will signal the listener to close, however, it  
42     # will remain open as long as .accept() is blocking  
43     listener.close()  
44     # so create a dummy connection to unblock accept()  
45     # and let the listener close  
46     try:  
47         with Client(address, authkey=password) as c:  
48             c.send(b'')  
49             c.recv()  
50     except ConnectionRefusedError:  
51         # someone connected before our dummy connection  
52         # and the listener shut down, don't do anything  
53         pass  
54 t.join()
```

```

client.py

1 import numpy as np
2 from multiprocessing.connection import Client
3
4 address = ('localhost', 6000)
5 passwd = b'password'
6 with Client(address, authkey=passwd) as conn:
7     # send and recv pickle and unpickle, respectively
8     # the objects we try to send. So we can send essentially
9     # arbitrary data
10    conn.send({'key1': 'hello', 'key2': np.arange(5)})
11    resp = conn.recv()
12    print("Response : ", resp)

```

To allow connections from the network, simply specify the Listener address as `''` (empty string) and the connect with the client to the IP address of the computer running the listener process. Note that pickling and unpickling in `send()` and `recv()` can lead to potential security issues, so it is best to use a password if accepting connections over the network.

**Exercise 7.3.** A server is running on computer with IP address <IP>, on port <PORT> with password <PASSWORD>. This server is controlling a Pico by directly sending whatever string it receives to the pico and sending the response from the pico back. Connect to this server with a client, send the command \*IDN? to query the instrument's identification and blink an LED if the response is correct.

`instrument_server.py`: The code accepts the connections in a separate thread. For each connection it accepts, it starts a new client handler thread. All client handler threads share the same pico, so we have to lock it appropriately before attempting communication, because another client can interrupt us at any moment. The server quits when it receives KeyboardInterrupt (Ctrl-C)

```

1 from multiprocessing.connection import Listener, Client
2 from threading import Thread, Lock, Event
3 import time
4
5 import pyvisa as visa
6
7 #for wrong passwords
8 from multiprocessing.context import AuthenticationError
9
10 rm = visa.ResourceManager()
11 pico = rm.open_resource(rm.list_resources()[-1],
12                         write_termination='\n',
13                         read_termination='\n')
14 pico_lock = Lock()
15
16 handlers_lock = Lock()
17 client_handlers = []
18
19 class ClientHandler(Thread):
20     def __init__(self, conn, name=None):
21         super().__init__()
22         self.name = name
23         self.conn = conn
24         self._end = Event()
25
26     def stop(self):
27         self._end.set()
28
29     def run(self):
30         try:
31             while not self._end.is_set():
32                 if self.conn.poll():

```

```

33         msg = self.conn.recv()
34         print(f"Received : {msg}")
35         if msg == '':
36             self.conn.send('')
37         else:
38             with pico_lock:
39                 resp = pico.query(msg)
40                 self.conn.send(resp)
41             time.sleep(0.1)
42     except EOFError:
43         print(f"Client {self.name} quit")
44     finally:
45         self.conn.close()
46         with handlers_lock:
47             client_handlers.remove(self)
48
49 def run(listener):
50     while True:
51         try:
52             print("Waiting for connection")
53             conn = listener.accept()
54             if hasattr(listener, 'last_accepted'):
55                 client_name = listener.last_accepted
56             else:
57                 client_name = None
58             handler = ClientHandler(conn, client_name)
59             handler.start()
60             client_handlers.append(handler)
61     except OSError:
62         print("Stopping listening")
63         break
64     except AuthenticationError:
65         print("Connection attempt with wrong password.")
66
67 address = ('', 0)
68 password = b'NOFY080_2024'
69 try:
70     with Listener(address, authkey=password) as listener:
71         actual_address = listener.address
72         print('Address :', actual_address)
73
74         t = Thread(target=run, args=(listener,))
75         t.start()
76
77         try:
78             while True:
79                 time.sleep(1)
80         except KeyboardInterrupt:
81             print("Quitting...")
82
83     listener.close()
84
85     try:
86         #dummy connection to force the listener to close
87         with Client(actual_address, authkey=password) as c:
88             c.send('')
89             c.recv()
90     except ConnectionRefusedError:
91         pass
92     t.join()
93     for handler in client_handlers:
94         handler.stop()
95
96     for handler in client_handlers:
97         handler.join()

```

TODO

Figure 3: Single computer controlling multiple experiments.

```
98 finally:  
99     pico.close()  
100    rm.close()
```

### 7.3.1 Real use case in a lab

In Fig. 7.3.1 is a photo of a single computer which controls three experiments connected to two different experimental setups. Each experiment belongs to a different student, who might need to run and modify their measurement Python scripts at the same time, so simple remote desktop sharing is not available.

We ended up using an instrument server similar to the simplified example above, which allowed the students to run the measurement scripts on any computer on the same network, even on their own laptops.

## 8 Solutions of differential equations

### 8.1 Initial value problems

Initial value problems are differential equations of the form

$$\frac{dy}{dt} = \mathbf{f}(\mathbf{y}, t), \quad (21)$$

where  $\mathbf{y}$  is a vector,  $t$  is the time and  $\mathbf{f}$  is an arbitrary function together with the initial condition  $\mathbf{y}(t = 0) = \mathbf{y}_0$ . Note that the equation is always written in the form of a first-order ordinary differential equation. However, differential equation of any order can be re-written as a first order equation by setting the vector  $\mathbf{y} = (y(t), y'(t), y''(t), \dots)$ . For example, a Newton's law of motion,  $\ddot{x} = F/m$  would correspond to  $\mathbf{y} = (x, \dot{x})$  and  $\mathbf{f} = (\dot{x}, F/m)$ .

These types of equations are usually solved by *time stepping*: given that we know  $\mathbf{y}(t = 0) = \mathbf{y}_0$ , we calculate  $\mathbf{y}(dt) \approx \mathbf{y}(0) + \mathbf{f}(\mathbf{y}(0), 0)dt$ ;  $\mathbf{y}(2dt) \approx \mathbf{y}(dt) + \mathbf{f}(\mathbf{y}(dt), dt)dt$  .... This time stepping is called the *Euler* method and generally requires a small time step to be accurate and numerically stable.<sup>12</sup> There are many time stepping schemes that are more accurate and stable than the basic Euler method. The accuracy of the method is often quantified using the big-O notation, i.e., the Euler method is  $O(dt)$ , or a first-order method, which means that if we halve the time step  $dt$ , we also halve the error at every step. Very common are the explicit Runge-Kutta methods, of which the 4-th order version (with accuracy  $O(dt^4)$ ) is probably the most common, i.e., if we halve the time step, the error decreases by factor 16.

The fourth-order Runge Kutta (RK4) calculate  $y(t + dt)$  from  $y(t)$  as

$$y(t + dt) = y(t) + \frac{dt}{6} (k_1 + 2k_2 + 2k_3 + k_4), \quad (22)$$

where

$$\begin{aligned} k_1 &= f(y_t, t) \\ k_2 &= f\left(y_t + \frac{1}{2}dt k_1, t + \frac{1}{2}dt\right) \\ k_3 &= f\left(y_t + \frac{1}{2}dt k_2, t + \frac{1}{2}dt\right) \\ k_4 &= f(y_t + k_3 dt, t + dt). \end{aligned} \quad (23)$$

Notice that for each time step, the function  $f$  has to be evaluated 4 times, and the calculation is about 4 times slower than the Euler method. There are also time stepping methods which instead of taking 4 times as long, they take up 4 times as much space (e.g., Adams-Basforth family of methods) which use the last four steps in the history of  $y(t)$  to estimate the next step.

The RK4 method for a problem of type (21) is implemented in SciPy in `scipy.integrate.solve_ivp()`. The `solve_ivp()` expects the function  $f(t, y)$ , which takes the time and state vector and returns the time derivative of the state vector, the initial condition, and the time range where the evolution should be calculated.

The calculation can also watch for *events* – typically a signal, that a calculation should end. These are functions of time and the state vector which *change sign* when the event occurs. An event can be made terminal (i.e., when even occurs calculation should stop) by simply setting the `terminal` attribute of the function to `True`. Remember, functions are objects, and we can set their attributes as we please (similarly to `self.attribute = value` when working with classes).

**Example:** Calculate the ballistic trajectory of a ball kicked under angle of  $45^\circ$  with initial velocities of 3 m/s along both  $x$  and  $y$  directions. The ballistic trajectory is the trajectory of a projectile thrown

---

<sup>12</sup>Numerical *instability* is the tendency of an error (i.e., the difference between the true solution and its numerical approximation) to oscillate wildly or grow to infinity.

in a medium (e.g., air) which exerts nonlinear drag force on the motion of the form

$$\mathbf{F}_d = -\frac{1}{2}A|\mathbf{v}|\mathbf{v}\rho CD, \quad (24)$$

where  $\mathbf{v}$  is the velocity of the projectile,  $A$  is the area of the projectile projected facing the direction of motion,  $\rho$  is the density of the medium and  $CD$  is the drag coefficient,  $CD \approx 0.47$  for a ball.

**Solution:** Full code is available in `ivp_ballistic.py`. First, we import necessary modules and define the needed constants

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import solve_ivp
4
5 g = 9.81 #m/s^2
6 rho_air = 1 #kg / m^3
7 ball_r = 0.1 #m
8 #drag coefficient, https://en.wikipedia.org/wiki/Drag_coefficient
9 CD = 0.47
10 A_ball = 2*np.pi*ball_r**2

```

Next we define the function  $f$ . We take the state vector  $\mathbf{x} = (x, y, v_x, v_y)$  to represent both the position and velocity vectors. The function must return the time derivative of this 4-component state vector

```

1 #differentiation of position and velocity
2 def F(t, z, CD=0):
3     x, y, vx, vy = z
4
5     #total velocity
6     v_tot = np.sqrt(vx**2 + vy**2)
7
8     air_resistance = 0.5*rho_air*v_tot**2*CD*A_ball
9
10    dxdt = vx
11    dydt = vy
12    dvxdt = -air_resistance*vx/v_tot
13    dvydt = -g - air_resistance*vy/v_tot
14    return np.array([dxdt, dydt, dvxdt, dvydt])

```

Next we define the event that will indicate that the ball hit the ground and make it terminal, because we do not want the calculation to continue past this point

```

1 def impact(t, z, CD=0):
2     return z[1]
3 # setting the terminal attribute to 0 will cause the calculation
4 # to end when the event happens
5 impact.terminal = True

```

Now we have everything we need to calculate the ballistic trajectory. We set the calculation time range to  $(0, 10)$ , 10 being just arbitrary large time, the calculation will be stopped by the impact event. We also pass additional arguments, the drag coefficient, through the usual `args=` keyword argument. Finally, we also ask for a `dense_output`, which will return an interpolation object (`solution.sol` below) to represent the solution. This is useful, because `solve_ivp` uses adaptive size of the time step  $dt$  (i.e., when the derivatives are small and slowly-changing, the time step is bigger), so if we want an evenly sampled solution we have to interpolate it. With the returned solution, we use the time of the event to create an evenly sampled time array and return the interpolated trajectory.

Finally, we calculate the trajectory with and without air resistance and plot. We use slightly non-zero initial condition in the  $y$  position to avoid triggering the event in the beginning.

```

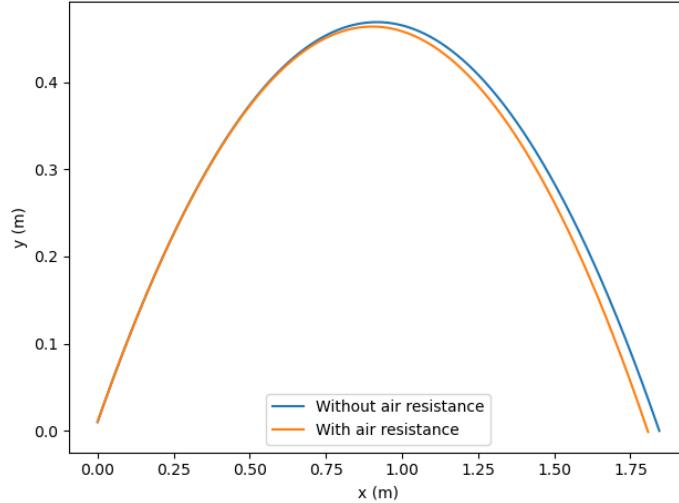
1 def ballistic(initial_condition, CD=0):
2     solution = solve_ivp(F, (0, 10), y0=initial_condition,
3                           events=impact, dense_output=True, args=(CD,))

```

```

4     dt = 1e-3
5     print(solution.t_events)
6     time = np.arange(0, solution.t_events[0][0]+dt, dt)
7     z = solution.sol(time)
8     x = z[0,:]
9     y = z[1,:]
10    return x, y
11
12 initial_condition=[0, 0.01, 3, 3]
13 # without air resistance
14 x, y = ballistic(initial_condition)
15 # with air resistance
16 xvz, yvz = ballistic(initial_condition, CD)
17 fig, ax = plt.subplots()
18 ax.plot(x, y, label="Without air resistance")
19 ax.plot(xvz, yvz, label="With air resistance")
20 ax.legend(loc='best')
21 ax.set_xlabel('x (m)')
22 ax.set_ylabel('y (m)')
23 fig.tight_layout()
24 fig.savefig('ballistic.png')

```



**Exercise 8.1.** Simulate the population dynamics of predators and prey (foxes  $F$  and rabbits  $R$ ) using the Lotka-Volterra model:

$$\begin{aligned}\frac{dR}{dt} &= aR - bFR \\ \frac{dF}{dt} &= dFR - cF\end{aligned}$$

with  $a = 2/3$ ,  $b = 4/3$  and  $c = d = 1$ . Plot  $R(t)$  and  $F(t)$  and the trajectory in phase space  $R - F$ .

**Exercise 8.2.** Simulate the Lorenz system (the first properly studied example of deterministic

chaos) given by equations

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \beta z,\end{aligned}$$

with  $\sigma = 10$ ,  $\rho = 28$ , and  $\beta = 8/3$ . Plot the  $x$ ,  $y$ ,  $z$  trajectory in 3D space.

**Intermezzo 8:** *3D plotting* Matplotlib supports 3D plotting. On recent versions of matplotlib, a simple 3D plot of a line given by arrays  $x$ ,  $y$ , and  $z$  is very simple:

```
1 fig3d = plt.figure()
2 ax3d = fig3d.add_subplot(projection='3d')
3 ax3d.plot(x, y, z, lw=0.5)
```

## 8.2 Boundary value problems

Boundary value problems are differential equations of the form

$$F(\mathbf{y}, x) = 0 \quad (25)$$

where  $F$  is some function that defines our problem that depends on the unknown function itself and its derivatives,  $\mathbf{y}$  is the unknown function of  $x \in [a, b]$  subject to boundary conditions  $B(y_a, y_b) = 0$ . Here the unknown function is a function of space, rather than time, and we need to know the solution "everywhere" at once. The numerical algorithms typically start from some initial guess and then iteratively optimize the solution  $\mathbf{y}$  to fulfill the equation (25) while maintaining the boundary conditions.

Solution of 1D (dimension of  $x$ ) boundary value problems is implemented in SciPy in `scipy.integrate.solve_bvp`, which also allows for finding unknown parameters for which the solution can exist (see example below). For multi-dimensional problems, you typically have to use a dedicated package or write your own.

We will solve a simple example of a 1D Schrödinger equation:

**Example** Calculate the wave functions and energy levels of a particle in infinite potential well with additional potential  $V(x)$  inside the well.

The states of a quantum particle are described with a stationary Schrödinger equation

$$E\psi = -\frac{\hbar^2}{2m} \frac{d^2\psi}{dx^2} + V(x)\psi, \quad (26)$$

where  $m$  is the particle mass,  $\psi$  is the particle wave function and  $E$  is the energy. Calculate in "atomic units",  $\hbar = 1$ ,  $m = 1$  and potential of the shape shown in Fig. 8.2, i.e.,  $V(|x| > 1) = \infty$  and  $V(|x| < 0.1) = 10$  and  $V = 0$  elsewhere.

**Solution:** The full code is available in `bvp_schrodinger.py`.

Begin by importing everything we need, define the constants `w`, the width of the barrier and `ah` the height of the barrier and the potential function itself. The infinite walls at  $x = \pm 1$  will be taken care of by the boundary conditions.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.integrate import solve_bvp
4
5 a = 1
```

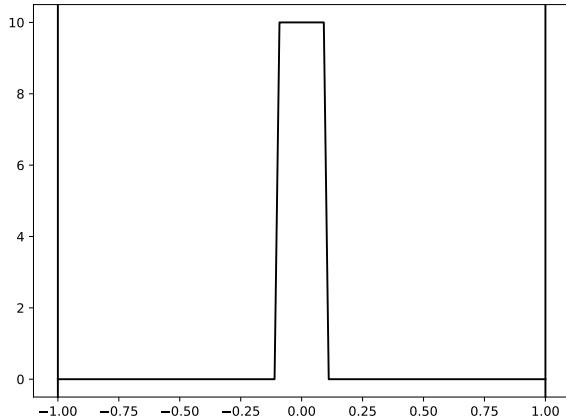


Figure 4: Potential for the Schrödinger equation (26).

```

6 m = 1
7 w = 0.2
8 dh = 10
9 N = 100
10
11 #potential well with a barrier in the middle
12 def potential(x):
13     V = np.zeros_like(x)
14     ix = np.logical_and(x > -w/2, x < w/2)
15     V[ix] = dh
16     return V

```

Next, define the function  $F$  from (25). We rewrite the Schrödinger equation (26) to

$$\frac{d^2\psi}{dx^2} = 2m(V(x) - E)\psi, \quad (27)$$

and the unknown function  $\mathbf{y}$  is then  $(\psi, d\psi/dx)$ . Finally, we also need to find the unknown parameter  $E$ . The function  $F$  can depend on a vector of unknown parameters which `solve_bvp()` finds together with the unknown function. In this case, this is just the energy.

```

1 #right-hand side of the Schrodinger equation
2 def F(x, psi, p):
3     E = p[0]
4     V = potential(x)
5     dpsidx = psi[1]
6     dpsidx2 = (V - E)*psi[0]*2*m
7     return np.row_stack((dpsidx, dpsidx2))

```

Next is the function representing the boundary conditions. This is an arbitrary function of the value of the unknown function at the boundaries and unknown parameters which the `solve_bvp` maintain at zero. We have to return a vector of size 2 (i.e., the number of boundaries) + number of unknown parameters. In our case we simply want the wave function to be zero at the infinite wall. The third boundary condition is arbitrary, and its purpose is to simply ensure that the found solution is not identically 0.

```

1 def boundaries(ya, yb, p):
2     return np.array([ya[0], yb[0], ya[1]-0.0001])

```

The number in the third condition is arbitrary, because the solution provided by `solve_bvp` is not normalized.

Next we need to create the initial condition, both for the wave function and the energy, for the solver to optimize. Note that the original Schrödinger equation (26) has infinitely many solutions. To which solution `solve_bvp()` actually converges is given by this initial condition. First we create the grid, i.e., points in space  $x$  which are the argument of the wave function and create a simple estimate of wave function which is zero everywhere and non-zero in one off center point, in order to converge to a solution localized in one half of the potential well. Remember, the full initial condition is the wave function and its gradient, so `y_0` is a 2D array of size  $2 \times \text{length of } \mathbf{x}$ .

```

1 #grid on solve the eqquation
2 xs = np.linspace(-a, a, N)
3
4 #the initial guess of the solution
5 #the wave function itself
6 ys0 = np.zeros_like(xs, dtype=complex)
7 # just so that the initial guess is not identically zero. This choice
8 # will give us asymmetric solution where the particle is localized in one
9 # part of the well
10 ys0[N//4] = 1
11 #the wave function gradient, we take those simply as zeros
12 ys1 = np.zeros_like(xs, dtype=complex)
13 y0 = np.row_stack((ys0, ys1))

```

For the energy, we simply use the ground state energy of a particle in a simple infinite potential well.

```
1 En = (np.pi*1)**2/(8*a**2)
```

Finally, we have everything we need to call `solve_bvp()`. Physically, the most interesting part is the probability of finding the particle at a position  $x$ , which is given by  $|\psi(x)|^2$ , so we plot that. To be exact, we should also normalize the wave function such that  $\int |\psi(x)|^2 dx = 1$ , however, since we are not calculating any quantum mechanical expectation values and are only interested in the shape of the distribution we plot it as it is, together with the potential and found energy.

```

1 sol_schr = solve_bvp(F, boundaries, xs, y0, p=(En,))
2 #plot the solution
3 fig, ax = plt.subplots(1, 1)
4 ax.set_title("Schrodinger")
5 # probability density of particle position (non-normalized)
6 ax.plot(sol_schr.x, abs(sol_schr.y[0])**2)
7 ax2 = ax.twinx() # y-axis on the right hand side
8 ax2.plot(xs, potential(xs), '--', color='k') # potential
9 ax2.axhline(sol_schr.p[0], ls='--', color='r') # energy of the solution
10 ax2.axvline(-a, color='k') # edges of the potential well
11 ax2.axvline(a, color='k')

```

### 8.3 GPU acceleration

Certain types of physical problems can be efficiently calculated on graphics cards (GPUs). These are problems which involve many independent calculations which can be run in parallel. A prototypical example is the gravitational  $N$ -body problem: calculate the motion of  $N$  planets with equal masses  $m$  that interact with each other via gravity only.

We will consider the 2D case for simplicity of plotting. The total force acting on planet  $j = 1 \dots N$  is

$$\mathbf{F}_j = Gm^2 \sum_{k \neq j} \frac{\mathbf{r}_k - \mathbf{r}_j}{|\mathbf{r}_k - \mathbf{r}_j|^3}, \quad (28)$$

where  $G$  is the gravitational constant and  $\mathbf{r}_k$  are the positions of the planets. The motion of the planet is then given simply by  $m\ddot{\mathbf{r}}_j = \mathbf{F}_j$  and we could use `solve_ivp()` as before. However, this problem becomes more scientifically interesting for large number of planets where `solve_ivp()` would quickly grind to a halt. Therefore we want to make use of fast GPUs.

The calculation will proceed again in steps: we will keep stored all positions and velocities of the planets. In every step, we will calculate the total force acting on all planets and then update positions and velocities via Euler stepping for simplicity. The calculation involves  $N^2$  calculations of forces between planets which are independent of each other. Our goal is to use the GPU to run these in parallel.

The full code is `example_code/taichi/gravity.py`

### 8.3.1 Setting up the environment

We will be using the `taichi` library to accelerate the computation from within Python, which has the advantage that it is rather simple, looks very much like standard Python code and allows using either GPU or CPU with the same code, so you can still try the code below even if you do not have a dedicated GPU in your computer. Taichi unfortunately requires Python version 3.10, which is most likely not the version you have installed. We do not want to clutter the system with multiple python versions, therefore we will use a *virtual environment* to have an isolated installation of Python with its own version and packages.

The easiest way to manage virtual environments is with the `uv` tool <https://docs.astral.sh/uv/getting-started/installation/>. On Windows, run the installation commands in the Windows Powershell.

Once `uv` is installed, create an empty directory, let's call it "gravity", navigate to it in the command line and run

```
1 uv venv -p 3.10
```

which will create the virtual environment with Python version 3.10. You shoud see output that looks similar to

```
Using CPython 3.10.16 interpreter at: /usr/bin/python3.10
Creating virtual environment at: .venv
Activate with: source .venv/bin/activate
```

Activate the virual environment with the command from the last line. Your command line prompt shoud now show the virtual environment, e.g., on my linux machine

```
(gravity) emil@nt202 ~/gravity $
```

This is a blank slate where we need to install all Python libraries we need. We will only need `taichi` and `matplotlib` (and their dependencies), which we can install simply using

```
1 uv pip install taichi matplotlib
```

and now we have everything we need to start writing and running taichi code. However, we have to have the environment activated, otherwise we are using system-wide python which is not aware of the `taichi` library.

### 8.3.2 Testing the installation

TODO

### 8.3.3 GPU Kernels

Taichi allows us to offload specific functions to run on the GPU. These functions are handled differently from the rest of the Python code and are called *kernels*. We will write two kernels, one for calculating the forces and one for the Euler stepping. We will store the data in numpy arrays – this is not the fastest way (taichi can use memory more effeciently, see the taichi documentation for a more optimized version), but it is the simplest.

We import `taichi` and as it to use a GPU is possible using

```

1 import taichi as ti
2 ti.init(ti.gpu)

```

This will use, in order of precedence, NVIDIA CUDA, Vulkan on AMD GPUs or the CPU.

The Euler stepping kernel looks like this

```

1 def step(rs: ti.types.ndarray(dtype=ti.f64),
2         vel: ti.types.ndarray(dtype=ti.f64),
3         acc: ti.types.ndarray(dtype=ti.f64),
4         dt: float):
5     N = rs.shape[0]
6     for j in range(N):
7         for k in range(2):
8             rs[j,k] += dt*vel[j,k]
9             vel[j,k] += dt*acc[j,k]

```

We indicate that the function is a kernel using the `ti.kernel` decorator. The function takes four arguments – the positions of the planets `rs`, the velocities `vel` and accelerations `acc`, which are numpy arrays of 64 bit floats of shape `(N,2)` ( $N$  planets and two coordinates) and the time step `dt` which is a simple float.

Inside, the function look rather ordinary, however it does run inside the GPU and the outer-most for loop is in fact not a loop but iterations are executed in parallel as much as possible. There are some restrictions – for example, you are not allowed to `break` out of the outermost loop. The inner loop (`for k in range(2)`) runs as an ordinary sequential for loop. We do not return anything, but rather modify the input arrays.

A more interesting kernel is the calculation of accelerations (or forces), which looks like this

```

@ti.kernel
1 def newton(rs: ti.types.ndarray(dtype=ti.f64),
2            acc: ti.types.ndarray(dtype=ti.f64)):
3     #the types of argument of kernels have to be annotated
4
5     #assume that the array of positions is shape (N, 2)
6     N = rs.shape[0]
7
8     for j in range(N):
9         # zero out the acceleration of the j'th particle so that we can add to it
10        # slices are not supported in taichi
11        acc[j,0] = 0
12        acc[j,1] = 0
13
14    #this loop runs in parallel, and THERE IS NO SYNCHRONIZATION
15    #no mutexes/locks or anything like that. We have to make sure
16    #none of the operations we do inside the loop can influence one another
17    for j in range(N):
18        #only the top-level loop runs in parallel, this one is serialized
19        for k in range(N):
20            #no self-interaction
21            if j != k:
22                pk = ti.Vector([rs[k,0], rs[k,1]])
23                pj = ti.Vector([rs[j,0], rs[j,1]])
24
25                d = pj - pk
26                #this is a cheat, 1e-5 is the minimum possible value
27                #this limits the maximum possible velocity for
28                #numerical stability
29                r = d.norm(1e-5)
30
31                acc[j,0] += -d[0]/r**3
32                acc[j,1] += -d[1]/r**3
33

```

Our problem is relatively simple, so-called *embarrassingly parallel*<sup>13</sup> because the calculations are entirely independent of each other. GPUs are capable of running many task in parallel quickly because

---

<sup>13</sup>This is an actual term used in the computer science.

they do not allow for synchronisation (or only in a very limited form). This means, that despite running many things in parallel, we do not have any mechanism similar to `Lock()` from multithreading. Therefore, if the iterations of the loop depend on each other, avoiding race conditions becomes somewhat more complicated. For example, imagine some planets are made of matter and some of anti-matter and if they get too close to each other they annihilate. We could implement this for example as

```

1 for j in range(N):
2     for k in range(N):
3         if planet_is_alive(j) and planet_is_alive(k):
4             if distance(j,k) < annihilation_distance:
5                 annihilate(j,k)

```

This would be OK if the loops are sequential. However, when the outermost loop runs in parallel, we could get to a situation where we have planets  $j = 1$  (matter) and  $k = 2, 3$  (anti-matter) which are all close to each other. Since the check on line 3 might be using old data, we could end up running both `annihilate(1,2)` and `annihilate(1,3)` and thus increasing the ratio of matter to anti-matter in the universe.

### 8.3.4 Using the kernels

To represent the state of the simulation – the positions and velocities of the planets and the time we will use a simple class. As an initial condition we will random positions in 2D with normal distribution around origin and solid body rotation for velocities with some initial angular velocity. The class will have only two methods, which call the two kernels we wrote above,

```

1 class Planets:
2     def __init__(self, N, D=1, dt=1e-4):
3         self.D = D
4         self.dt = dt
5         #
6         self.t = 0
7         #randomly distributed initial positions with normal distribution
8         #and spread D
9         self.rs = D*np.random.randn(N, 2)
10        #velocities -- solid body rotation as initial condition
11        #so that the system has some nonzero angular momentum
12        self.vels = 50*np.column_stack((-self.rs[:,1], self.rs[:,0]))
13        #accelerations
14        self.acc = np.zeros_like(self.rs)
15
16    def update_accelerations(self):
17        #we can simply pass numpy arrays to the kernel
18        #this "function", however, does not run inside Python
19        #the acc array will be modified
20        newton(self.rs, self.acc)
21
22    def euler_step(self, dt=None):
23        #Euler step is very inaccurate and requires small dt
24        #replace with something better for actually useful simulation
25        if dt is None:
26            dt = self.guess_dt()
27        step(self.rs, self.vels, self.acc, dt)
28        self.t += dt

```

Finally, we create a simulations with 5000 planets, plot a real-time animation using matplotlib and periodically save a picture

```

1 if __name__ == '__main__':
2     import os
3     fig, ax = plt.subplots()
4
5     #Create 5000 planets
6     planets = Planets(5000, D=1)

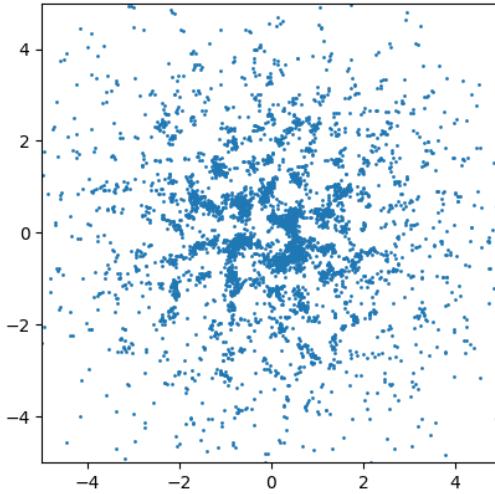
```

```

7
8     # set up plotting
9     plot, = ax.plot(planets.rs[:,0], planets.rs[:,1], 'o', ms=1)
10    plot_scale = 5
11    ax.set_xlim(-plot_scale, plot_scale)
12    ax.set_ylim(-plot_scale, plot_scale)
13    ax.set_aspect('equal')
14
15    # where to save the result
16    os.makedirs('pictures', exist_ok=True)
17    try:
18        last_save = 0
19        frame = 0
20        #the simulation simply proceeds by updating the accelerations
21        #and then incrementing the positions and velocities, until
22        #the simulations is stopped
23        while True:
24            planets.update_accelerations()
25            planets.euler_step(1e-5)
26
27            # update the interactive plot
28            plot.set_xdata(planets.rs[:,0])
29            plot.set_ydata(planets.rs[:,1])
30            plt.pause(1e-3)
31
32            #Only save the picture. In a real simulation we would save the
33            #positions and accelerations so that we can further process them.
34            if planets.t - last_save > 1e-3:
35                last_save = planets.t
36                fig.savefig(f'pictures/frame_{frame:08d}.png', dpi=300)
37                frame += 1
38            print(planets.t)
39    except KeyboardInterrupt:
40        print("Quitting...")

```

After about a minute of runtime on Intel Core i7-7700 CPU (a fairly old CPU) the initial gaussian distribution of the planets develops some interesting structure



## 9 What didn't fit

### 9.1 PID algorithm

TODO

### 9.2 Symbolic mathematics

TODO: sympy

## A Linear Harmonic Oscillator

A linear harmonic oscillator with friction is described by a dynamical equation

$$\ddot{x} + \omega_0^2 x + \gamma \dot{x} = F(t)/m, \quad (29)$$

where  $m$  is the oscillator mass,  $\omega_0$  is the resonance frequency,  $\gamma$  the friction coefficient and  $F(t)$  the external force. Assuming that the force has the form  $F(t) = \Re(\tilde{F}e^{i\omega t})$  and that the solution has the form  $x(t) = \Re(\tilde{x}e^{i\omega t})$  (or applying the Fourier transform to both sides of the equation) we get by simple rearranging

$$\tilde{x} = \frac{1}{m} \frac{\tilde{F}}{\omega_0^2 - \omega^2 + i\gamma\omega} = \chi(\omega)\tilde{F}, \quad (30)$$

where  $\chi(\omega)$  is called susceptibility.

Since Eq. 29 is a linear equation, the response to a sum of forces will be the sum of responses to each force.

## B Setting up communication with instruments

We will use VISA library to talk to instruments. To talk to the Raspberry Pi Pico we will use a Python implementation of the library, for which we need to install at least `pyvisa`, `pyvisa-py` and `pyusb` using

```
1 pip install pyvisa pyvisa-py pyusb
```

which you should run in a command line which is aware of the python installation (e.g., Anaconda Propmt if on Windows with Anaconda Python distribution). On Linux you should also add yourself to the `dialout` group (do not forget the `-a` switch),

```
1 sudo usermod -a <your username> -G dialout
```

and log out and log in.

To test the installation run the `pyvisa-info` from the command line. A bunch of information should be printed out, look for line that looks like

```
1 USB INSTR: Available via PyUSB (1.2.1). Backend: libusb1
```

Next, create a Python script `list_resources.py` with the following code

```
1 import pyvisa as visa
2 rm = visa.ResourceManager()
3 print(rm.list_resources())
4 rm.close()
```

and run it. It should list either nothing or a few COM (or tty on Linux) ports. Next connect the Pico and run the program again, a new address should appear, that is the address we will use to communicate with the Pico.

Next, run the following code, replacing "`ASRL/dev/ttyACM0::INSTR`" with the address you found in the previous step.

```
1 import pyvisa as visa
2 rm = visa.ResourceManager()
3 pico = rm.open_resource('ASRL/dev/ttyACM0::INSTR',
4                         read_termination='\n',
5                         write_termination='\n')
6 print(pico.query('*IDN?'))
7 pico.close()
8 rm.close()
```

The program should print "PICO" and quit without error.

For more full-featured implementation of visa you may consider the implementation from National Instruments (NI). The NI-VISA library is free, but not open source and registration is required. Linux support is also limited to out-of-date kernels.

## B.1 Supported commands

---

### \*IDN?

Query identification string. Should reply PICO.

---

### :LED n m

Turns the LED  $n$  on ( $m=1$ ) or off ( $m=0$ ). The LED number  $n = 0 \dots 4$ , red LED is 0.

---

### :READ:P?

Reads the pressure in Pa.

---

### :READ:T?

Returns the temperatures as  $100T$  where  $T$  is the temperature in  $^{\circ}\text{C}$ .

---

### :READ:PT?

Reads both temperature and pressure

---

### :READ:ACC?

Reads the accelerometer. Returns three space-separated values in the range -32768 to 32768 which maps to  $-2g$  to  $2g$ .

---

### :READ:GYR?

Reads the gyroscope. Returns three space-separated values in the range -32768 to 32768 which maps to  $-500^{\circ}/\text{s}$  to  $500^{\circ}/\text{s}$ .

## B.2 Hacking the firmware

The source code of the program running on the Pico is available [here](#). To compile it, you need to set up Pico SDK, follow the instructions [here](#).

Alternatively you can use MicroPython to run Python code on the Pico directly, follow the instructions [here](#) for set up.

LEDs are wired to GP0 – GP4 pins and the sensors are connected to the I2C0 controller on pins 16 and 17.

## C Basics of git version control system

TODO

## D Basics of Linux

Linux is a kernel of an operating system – the bit that directly handles access to hardware and cannot be directly used by the user. When combined with a commonly used set of programs developed, e.g., by the GNU Project, an actual operating system results sometimes called “GNU/Linux” (although almost everyone just calls it Linux).

Adding extra features on top of that (typically an installer, a package manager and a graphical user interface) results in a *linux distribution* (or *distro*), e.g., Ubuntu, Fedora, OpenSUSE, Mint, Red Hat Enterprise Linux, ... or Gentoo, in the case of the lab computers.

### D.1 Filesystem structure

Linux, and generally Unix-like systems, do not use the filesystem structure which might be familiar from Microsoft Windows. Particularly, the filesystem starts at the **root directory** denoted by a single forward slash /, from which all other directories branch. Multiple hard drives, or partitions on a single drive (which would be called something like c:\, d:\, etc. on Windows) look and behave as ordinary directories.

Important directories are /**home**, where users can store their data in their **home directories** or /**bin** where executable programs are stored.

## D.2 Moving files between lab and personal computer

All students have accounts for storing their files on the faculty servers. These can be accessed through the web at <https://su.mff.cuni.cz/>

You can also access them with file manager on linux by going to `sftp://su.mff.cuni.cz/home/university_username` or `\\\su.mff.cuni.cz\home\university_username` on Windows and logging in with your university credentials. `university_username` is the word based on your name, not the ID number.

## D.3 Command line interface

The operating system gives us access to its services (e.g., writing stuff to disk, or printing stuff on the screen) through a set of programs called a **shell**. Shells can either be graphical (graphical user interface, GUI) or a command line interface (CLI). On Linux, many graphical shells exist, the most popular being Gnome, KDE (used on lab computers) and (I guess) Xfce. Most commonly used text-based shell is called **bash**. In order to use the CLI we can use a “terminal emulator” from within the GUI. Each GUI provides their own version (e.g., `konsole` for KDE, `gnome-terminal` for Gnome). Almost all linux distributions bind the launching of a terminal emulator to a keyboard shortcut Ctrl-Alt-T.

### Basic commands

- **Navigation** For basic navigation, we can use
  - `ls` – lists the contents of the current directory
  - `cd` – changes the directory
  - `pwd` – prints the current working directory
  - `tree` – prints a tree representation of the current directory and its contents

The behavior of commands can be modified with *options*, usually leading with one or two dashes, e.g., `ls -l` lists the current directory in a one-column format with some extra information; `ls -la` also lists hidden files (those whose name begins with a full stop .) and `ls -lh` lists file sizes in a human-readable format (kilobytes, megabytes...) rather than just bytes. Note that it is quite common that filenames on Linux do not have filename extensions (e.g., .exe, .txt etc). The command `tree` by default prints the full contents of all subdirectories, recursively of the current directory, and is thus not particularly useful for at the root of larger directories (e.g., your ~). The optional argument `-L n` limits the depth to which subdirectories are printed, e.g. `tree -L 2` prints the contents of the current directory, its subdirectories, and their subdirectories.

While it is often more convenient to work with a GUI file browser, sometimes it is necessary to switch to CLI, for example, to run a program. Most file browsers support this in some way (usually right-click > open in the terminal, or something similar).

Special directory names are . (dot), which means the current directory and .. (double dot), which refers to the parent directory, and ~ (tilde, above Tab key), which refers to your home directory (which, for me, would be `/home/vargaem` on my computer).

E.g., `cd ..` changes the directory to one level up.

Most terminal emulators will help you to save some typing by autocompletion of names of files, folders or commands. Hitting Tab key will complete the current word as much as it can do so uniquely. Hitting the Tab twice will give you a list of possible completions.

**Exercise D.1.** Create a file with some text in it in a subdirectory in your home dir using the GUI. Then find this file using the CLI.

- **Paths** When moving around the file system we have two options for specifying paths to the exact file or directory we want – *absolute* or *relative* paths. This is not unique to Linux, all operating systems provide these two options in some way
  - absolute paths – a path that specifies the absolute position of a file in the file system. On Linux, these paths will always start with `/`, i.e. `/etc/fstab` (a file that on Linux contains info about the partitioning of the file system). On Windows, these paths start with the drive letter, i.e., `C:\Windows\system32`.
  - relative paths – a path relative to the current position as given by the command `pwd`. Files and folders are searched for in the current working directory. If we need to refer to a file in the parent directory, we have to use `...`

- **Wildcards** Sometimes we want to work only with names of files or directories that fit a certain pattern. This pattern can be most easily specified using *wildcard* characters - `*` (star) substitutes any number (including zero) of any characters; `?` substitutes exactly one character. These wildcards can be used with `ls`, e.g.

```
1 ls *.txt
```

lists all `.txt` files in the current directory

```
1 ls ab?.pdf
```

lists all three-letter PDF files which begin with `ab`. More complex patterns can be formed by specifying which characters can substituted into the pattern using square brackets `[]`, e.g., `data_[1-9].txt` will match `data_1.txt`, `data_2.txt`, ..., `data_9.txt`.

**Exercise D.2.** List all files from the directory `/usr/include` which begin with ‘`a`’ and have a file extension ‘`.h`’.

- **help and man pages** Commands can range from quite simple, like `cd` to quite complicated, like `find` (see below), where one often needs to look at the documentation. Most commands accept an option `-h` or `--help` which will print a short help. For more thorough documentation we can use the `man` command which displays a so-called *man page* of the given command (abbreviated from **manual**), e.g.

```
1 man ls
```

will show the documentation for the `ls` command.

- **Reading files** Text files can be read and displayed directly in the CLI (for editing see next section) using the `cat` command (abbreviated from **concatenate**) which prints one or more files, as text, directly on the command line.

If we want to read a long text file we might need to be able to scroll the view. For that, we can use `less`, which takes one filename and allows to scroll the view using arrows and quit by pressing `q`

- **Copying, moving and deleting**

We **copy** files using `cp`. To copy a file called `source` from subdirectory `path/to` to current directory and call it `destination` we use

```
1 cp path/to/source destination
```

if the destination is a directory, the file is copied into it with the original name. If we want to copy a directory and everything in it, we can use `cp -R` (R for Recursive).

Similar pattern is used by the `move` command `mv` (except `-R` is not used for moving directories). Both `mv` and `cp` support wildcard characters, e.g., `mv *.txt texts/` will move all `*.txt` files to the `texts` directory.

New directories can be created using `mkdir`.

Files are deleted using `rm`, which also supports wildcards. **Only empty directories** can be removed using `rmdir`. Directories with stuff in them can be removed using `rm -rf`. **CAREFUL** `rm` removes files permanently, it does not move them to the trash/recycling bin or something similar. `rm -rf` in combination with wildcards can lead to disasters if not careful, e.g.,

```
1 rm -rf tmp*
```

would remove all files and directories beginning with “tmp” (perhaps some temporary files that are not needed anymore), but

```
1 rm -rf tmp *
```

would remove the file called “tmp” and **everything in the current directory without asking for confirmation**.

**Exercise D.3.** Make a directory e.g., “exercise1”, and copy all `.h` files from `/usr/include` that begin with a or b. Then delete all files beginning with b in the exercise1; after that, delete the entire exercise1 directory (with some files still in it).

- **editing text files**

- `nano`

`Nano` is often installed by default. The bottom row shows keyboard shortcuts for basic operations. Hat ^ means Ctrl, M means alt, i.e., to exit `nano` tells us to do `^X` which means pressing Ctrl-X at the same time. To undo or redo we should do `M-U` or `M-E` respectively, which means pressing Alt-U or Alt-E. `nano` refers to the displayed contents as *buffers* which are then saved to files on disks.

- `emacs`

A lot of features, which can be difficult to set up. Better to avoid.

- `vim` – Vim distinguishes between *edit mode* and *command mode*. By default, it starts in the command mode where the editor expects some commands. To enter the edit mode press `i` and `–INSERT–` appears at the bottom, which indicates the edit mode.

To save a file enter the command mode by pressing Esc (the `–INSERT–` at the bottom disappears) and type `:w+Enter`. To quit, use `:q`. To quit without saving, `:q!`. To quit and save `:wq`.

Files can be searched using Esc- / *search pattern*

- **searching** the command `find` can be used to accomplish fairly complicated tasks, but for simple searches, one can use

```
1 find directory_name -name 'name_pattern'
```

e.g., `find . -name '*.pdf'` will list all PDF files in the current directory including subdirectories. The search is case-sensitive. For case-insensitive search replace `-name` with `-iname`.

**Exercise D.4.** Find all files that end with ‘.conf’ in /etc and its subdirectories.

**Composing and extending** Bash allows redirecting the output of one command somewhere else rather than just the standard print-on-screen. If we want to store the output of a command in a file we can use the **redirection** > e.g.,

```
1 ls -l > directory_contents.txt
```

will save the output of `ls -l` in a file called “`directory_contents.txt`”. Redirection can be used with command `echo`, which simply prints back its input, for quickly creating simple files, e.g.,

```
1 echo A single line in a simple file > file.txt
```

will create a `file.txt` which will contain “A single line in a simple file”. *Note:* an empty file can be quickly created using `touch empty_file`. If we want to use the output of one command as the input of another, we can use the **pipe** |, e.g.,

```
1 ls -l | less
```

will allow us to scroll through the output of `ls -l` with arrows thanks to the `less` command.

Concatenation of multiple files can be done simply using `cat`

```
1 cat file1.txt file2.txt file3.txt \> concatenated_file.txt
```

For sorting (useful with `find`) we can use `sort`, e.g.,

```
1 find . -name '*.pdf' | sort
```

will print all PDF files in current directory, including subdirectories in alphabetical order.

Commonly used commands with pipes are `head` and `tail`, which display first or last 10 lines of their input, respectively. Both accept the option `-n` which changes the default 10 lines, e.g.,

```
1 cat long_text_file.txt | head -n 50
```

displays the first 50 lines from the file `long_text_file.txt`.

**Exercise D.5.** As in Problem 4, but sort the conf files alphabetically and store them in your home directory to a text file.

### D.3.1 Users and permissions

**Users** Ordinary users generally do not have permission to make arbitrary changes to the operating system. The user who can do anything to the computer is called `root` (not to be confused with the `root directory` `/`). To find out under what username we are logged in we can use the command `whoami`

**Permissions and groups** The operating system controls who can do what to which files using permissions. The permissions can be any and all of *read*, *write* and *execute* (or *list* for directories). The permissions are shown using `ls -l` in the first column, e.g., for a random file on my computer

```
1 ls -l /bin/bash
2 -rwxr-xr-x. 1 root root 1444200 Feb 6 2023 /bin/bash
```

Notice that the permissions appear to be listed three times. The first dash “-” indicates a regular file; following that we have “`rwx`”, which means that the owner of the file (in this case the user “`root`”) has read, write, and execute permissions; next “`r-x`” means that the users who belong to the group (in this case also called “`root`”) can read and execute the file, but not write into it. Finally, the last “`r-x`” means that everyone else can read and execute the file, but not write into it.

**sudo** Even if using your personal computer, for security reasons, it is generally a bad idea to work with the root account for ordinary tasks. When we need to execute a command that only a root is allowed to do (e.g., install a program) we use **sudo**, e.g.,

```
1 sudo apt-get install vim
```

will ask you for your password and then install the text editor **vim** on Debian-based Linux distributions (e.g., Ubuntu). Most personal installations of Linux use only one password for the user and **sudo**. Only the administrator has the root password on the lab computers.

## D.4 Accessing remote servers using CLI

If we have a regular account on the remote machine we can log into it with

```
1 ssh username@remote.server.cz
```

If we can only move files (the case for university student storage), in the CLI, to copy a file from the local machine to a remote server where we have an account we can use **scp**, e.g.

```
1 scp file username@su.mff.cuni.cz:
```

will ask you for your password and then copy a file to your student storage. To download a file

```
1 scp username@su.mff.cuni.cz:path/to/a/file .
```

will download a file from the server to the current directory (the dot at the end).

Another possibility is to use the **sftp** with which you can log-in to a server which gives you a limited set of commands (**ls**, **mkdir**, **cp**, **mv** etc.) with which you can organize your files on the remote and a possibility to upload files, using the command **put**, or download files, using the command **get**.

**Exercise D.6.** Try uploading a file and verify that it is present on remote using GUI file explorer. Conversely, create a file on the remote using GUI and download it using CLI.

Alternatively, to student storage, there is also the university OneDrive cloud with web interface (part of Office 365). Windows client works fine, there is also a Linux client, which is a bit more tricky to set up but also works.