NOFY080 lecture notes

Emil Varga

February 14, 2025

Contents

Python Language Intermezzos

Code Snippets

1 A quick introduction to basic Python

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 `.py` file extension or in so-called Jupyter notebooks (file extension `.ipynb`).

To run a Python code, open a suitable terminal (on Windows with Anaconda distribution of Python this will be `Anaconda prompt`) and type in `python` a prompt like

```
>>>
```

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`.

Exercise 1. Create a textfile `hello.py` which contains a single line

```
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`, `-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`

1.1.1 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 will be using Spyder, but almost nothing in the course will be exclusive to it.

1.1.2 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. Good for trying things out. [1]

1.2 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

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

Listing 1: Defining variables

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

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. 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

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

1.3 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.,

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

1.4 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.,

```
print("s\tri\ng") #\t and \n are interpreted as tab and new line
print(r"s\tri\ng") # 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.,

```
vari = 5
varf = 3.14
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 `https://docs.python.org/3/library/string.html#formatspec`

1.5 Functions

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

```
    def my_function(my_string):
        print("Someone wants to know how long \"" + my_string + "\" is!")
        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

```
    def calculate(a, b, c=1):
        """
        A function that calculates c*(a + b)

        Parameters:
        -----------
        a, b, c: numbers
            arguments of the calculation

        Returns:
        --------
        result: number
            The result of calculating c*(a + b)
        """
        return c*(a + b)

    >>> calculate(2, 3)
    5
    >>> calculate(2, 3, 4)
    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 span 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*:

```
    def calculate2(a, b, c=1, d=1):
        return c*(a + b)**d

    >>> calculate(2, 3, d=2) # leaves c default, but specifies d
    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.,

```
def add1(x):
    return x + 1

def change_number(x, func):
    return func(x)

>>> change_number(1, add1)
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.

```
>>> changer_number(1, lambda x: 2 + x)
3
```

Listing 5: Lambda functions.

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

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

1.6 Compound types

1.6.1 Lists and tuples

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

```
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`

```
>>> my_list = [1, 2.0, 'three']
>>> my_list.append('one more')
>>> my_list.pop() # removes the last element and returns it
"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 occurence of 'a' or returns an error

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

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

```
    def add_and_subtract(x, y):
        return x+y, x-y #note that the parentheses () are not required in return

    #here we unpack the return value into two variables
    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.

1.6.2 Dictionaries

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

```
    my_dict = {
        'key1': value1
        'key2': value2
    }
```

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

1.7 Control flow

1.7.1 if-elif-else

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

```
    if condition:
        do_if_true()
    elif condition2: #optional
        do_if_2true() #runs only if condition is False and condition2 is True
    else: #optional
        do_if_false() #runs only if both condition and condition2 are False
```

1.7.2 match

Available since Python 3.10 TODO

```
    command = 'yell'
    match command:
        case 'talk':
            print("hello")
        case 'yell':
            print("HELLO!!")
        case _: #default behavior
            print("unknown command")
```

1.8 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.

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

    for value in my_list:
        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`

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

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

```
    while condition:
        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.,

```
    while True: # this will never end
        do_stuff()
        if should_we_stop: # unless we kill it
            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,

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

Listing 6: List comprehension.

Exercise 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 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`.

1.9 Using external modules

Python is famous for having a large number of available libraries to do pretty much everything under the sun.[2] To use these, we first need to import the *module* we need, e.g. to use functions that deal with time we can do

[2] And we will learn how to create our own shortly.

```
import time as t #import the time module and gives it a shorthand name t
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.

```
from module import func1, func2
func1()
func2()
```

Exercise 5. Write a program to caluclate 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: $F_k = F_{k-1} + F_{k-2},\ F_1 = F_2 = 1$

2 NumPy, SciPy, and matplotlib

2.1 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 1: *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, the install the package `lmfit` run the following command in the command line (also works in the ipython REPL, e.g., in Spyder)

```
pip install lmfit

```

To remove the package use `uninstall`, and to upgrade `install -U`. Run `pip help` for a complete list of available commands.

To use numpy we do

```
import numpy as np
```

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

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

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

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

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

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

Exercise 6. Write a program that will (in a loop) add numbers $1+2+3+...+n$ 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 7. 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.

2.1.1 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.,

`arr.mean()`	mean of the array
`arr.std()`	standard deviation, biased estimate of the variance
`arr.sum(), arr.cumsum()`	the sum and cumulative sum
`arr.max(), arr.min()`	maximum and minimum, respectively
`arr.argmax(), arr.argmin()`	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 2: *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.,

```
import numpy as np
items = np.array(["Eggs", "Bread", "Apples"])
```

```
    prices = np.array([50, 20, 40])

    sort_ix = np.argsort(prices)
    print("Items sorted according to prices:")
    print(items[sort_ix])
```

2.1.2 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

```
    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 3: *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

```
    import numpy as np
    def my_load(fn):
        with open(fn, 'r') as file:
            rows = []
            for line in file:
                row = [float(s) for s in line.strip().split()]
                rows.append(row)
        return np.array(rows)
```

Intermezzo 4: *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

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

is roughly equivalent to

```
    file = open(fn, 'r')
    do_stuff()
    file.close()
```

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

Exercise 8. Extend the `my_load(fn)` from the example above to support comments in the files.

2.2 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 [3]

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

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

```
    ax.plot(x, y, '-') # connect the pairs of points given by x and y with full lines
    ax.semilogy(x, y, '--s') # logarithmic y-axis, dashed line with square markers
    ax.loglog(x, y, ':') # logarithmic x and y axis, dotted line
    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]`.[4] We can specify the line width using `lw=` keyword, marker size using `ms=` and color using `color=` [5] 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=` keword 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

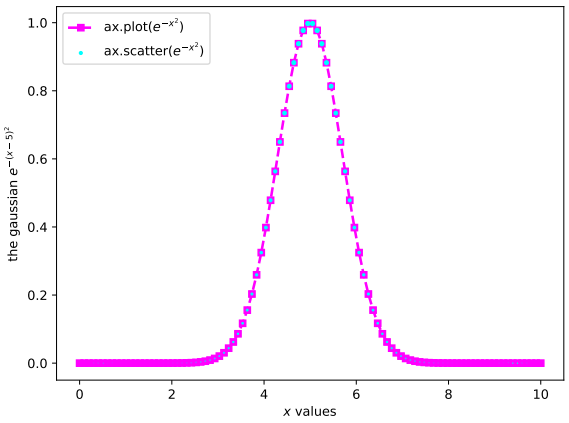
```
import numpy as np
import matplotlib.pyplot as plt

xs = np.linspace(0, 10, 100)
ys = np.exp(-(xs - 5)**2)

plt.close('all') #closes all figures we have opened so far
fig, ax = plt.subplots()

ax.plot(xs, ys, '--s', ms=5, lw=2, color='magenta', label=r'ax.plot($e^{-x^2}$)')
ax.scatter(xs, ys, s=ys*10, marker='o', color='cyan', zorder=3,
           label=r'ax.scatter($e^{-x^2}$)')

ax.set_xlabel('$x$ values')
ax.set_ylabel(r'the gaussian $e^{-(x-5)^2}$')
#where to put the legend, we can also use "best" and let matplotlib guess
ax.legend(loc='upper left')

fig.tight_layout() #reduces some of the whitespace around edges
fig.savefig('gaussian.pdf') #the file format of the image is deduced from the
    extension
```

Listing 9: Complete plotting example.

[3] Note that `plt.subplots()` returns two values. It only returns a single tuple, which is desctructured, or unpacked, into two variables `fig` and `ax`.

[4] See `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html` for complete list of possible formats

[5] See `https://matplotlib.org/stable/users/explain/colors/colors.html`



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

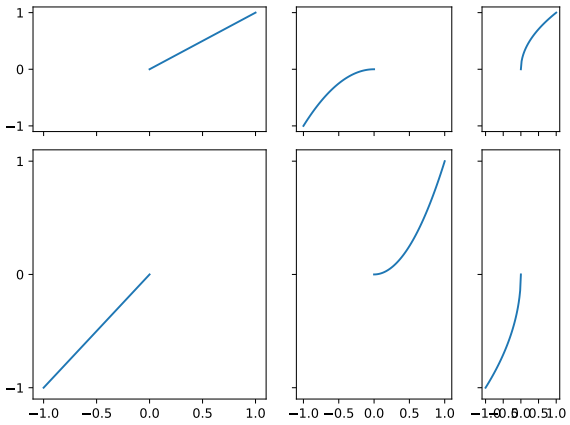
Exercise 9. Write a program, which will, using NumPy and Matplotlib, graph the expressions $y = x$, $y = x^2$ and $y = \sqrt{x}$ from $x = 0$ to $x = 5$ with different line styles (e.g., full line, dashed, dotted) and user-supplied number of points. Try linear, semilogarithmic and logarithmic axes. Label the axes and the curves.

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` [6] 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

```
import numpy as np
import matplotlib.pyplot as plt
nrows = 2
ncols = 3
fig, axes = plt.subplots(nrows, ncols, sharex=True, sharey=True,
                         width_ratios=[3, 2, 1], height_ratios=[1,2])
#axes is now a 2D array of axes which all share the same x and y range

xs = np.linspace(0, 1)

axes[0,0].plot(xs, xs)
axes[1, 1].plot(xs, xs**2)
axes[0, 2].plot(xs, np.sqrt(xs))

axes[1,0].plot(-xs, -xs)
axes[0, 1].plot(-xs, -xs**2)
axes[1, 2].plot(-xs, -np.sqrt(xs))

fig.tight_layout()
fig.savefig('multiax.pdf')
```

Listing 10: Multiple axes in single figure.

[6] `https://matplotlib.org/3.5.0/tutorials/intermediate/gridspec.html`



Exercise 10. 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

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

Intermezzo 5: *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 `import`ed, 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`:

```
def my_module_function(x):
    return 1 + x

print("Hello modules!")

```

`module_user.py`:

```
import module as m
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

```
if __name__ == '__main__':
    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.

Exercise 11. 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/FWHM, where f_0 is the frequency of the maximum response.

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

Exercise 12. Process all files from `lots_of_data` directory similarly to Exc. 11. Plot the FWHM as a function of f_0. Use the solution to Exc. 11 as a module.

Hint: `glob`

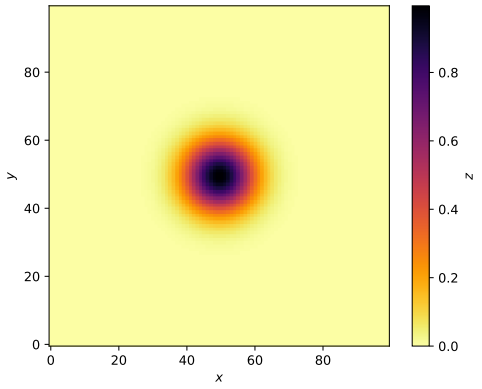
Observables that depend on two control variables can be often plotted using heat maps, for which we can use `ax.imshow(arr2D)`, which takes a 2D array of numbers and maps them to a color of a pixel using a colormap[7] Note however, that `imshow()` is primarily used for images, which conventionally start 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 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,

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

#x and y axis
_xs = np.linspace(0, 10, 100)
_ys = np.linspace(0, 10, 100)
#but we need 100 x 100 points for both x and y that sample the
#the entire interval (0, 10) x (0, 10), this can be done using
#meshgrid
xs, ys = np.meshgrid(_xs, _ys)
zs = np.exp(-(xs - 5)**2 - (ys - 5)**2)

plt.close('all') #closes all figures we have opened so far
fig, ax = plt.subplots()

plot = ax.imshow(zs, cmap='inferno_r', origin='lower')
#similar
#plot = ax.pcolormesh(xs, ys, zs, cmap='inferno_r')
cbar = fig.colorbar(plot) #the color axis scaling
cbar.set_label('$z$')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
```

Listing 11: Two-dimensional plotting example.

[7] See `https://matplotlib.org/stable/users/explain/colors/colormaps.html` for a full list of colormap names.



Exercise 13. 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.

2.2.1 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

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

3 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 $f(x; p_1, p_2, ...p_n)$ minimizes the sum of squared residuals, i.e.,

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

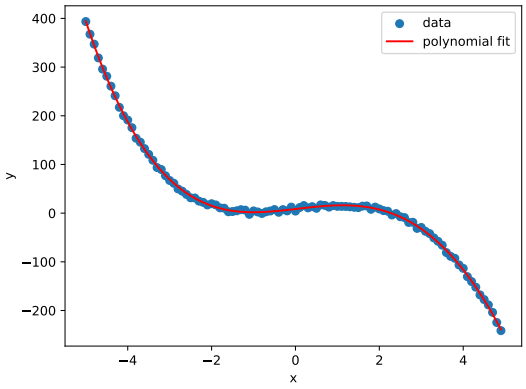
The result of the fit is a set of parameters $p_1, p_2, p_3, ...$ which minimize R^2. Depending on how the function f depends on the parameters p_k we talk about *linear* or *non-linear* fitting. The important difference is the dependence on the parameters p, not on the control variable x.

3.1 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

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

from numpy.polynomial import Polynomial as P

rng = np.random.default_rng() # random number generator

#%% create some noisy data and plot them
noise_amplitude = 10
xs = np.arange(-5, 5, 0.1)
ys = 0.1*xs**4 - 3*xs**3 + 10*xs + 4 + rng.random(len(xs))*noise_amplitude

fig, ax = plt.subplots()
ax.scatter(xs, ys, label='data')

#%% fit the polynomial
poly = P.fit(xs, ys, deg=4)
ax.plot(xs, poly(xs), color='r', label='polynomial fit')
#the coefficients of poly are scaled for a particular domain/window
#to get the ordinary coefficients we need to use .convert()
coef = poly.convert().coef
poly_string = ' + '.join([f'({c:.1f})*x^{k}' for k, c in enumerate(coef)])
print("The estimated polynomial is", poly_string)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend(loc='best')
fig.savefig('polynomial_fit.pdf')
```

Listing 12: Polynomial fitting



Exercise 14. Remove the background from the peaks in exercise 11 using a polynomial fit.
Hint: `np.polynomial.Polynomial.fit()` and boolean arrays

3.2 Non-linear curve fitting

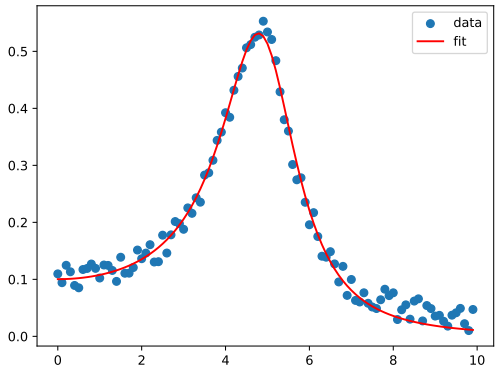
When the fitting function depends nonlinearly on the fit parameters we need to use nonlinear fitting, several libraries exist for this task, and for basic fitting, we can use `scipy.optimize` submodule of `scipy`.

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

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

from scipy.optimize import curve_fit

rng = np.random.default_rng()

#%% function to fit
def lorentzian(xs, height, center, width):
    return height*center**2*width**2/((center**2 - xs**2)**2 + width**2*xs**2)

#%% generate some data and plot
xs = np.arange(0, 10, 0.1)
height = 0.5
center = 5
width = 2
noise = 0.05
ys = lorentzian(xs, height, center, width) + rng.random(len(xs))*noise

fig, ax = plt.subplots()
ax.scatter(xs, ys, label='data')

#%% fit the data
popt, pcov = curve_fit(lorentzian, xs, ys, p0=[1, 1, 1])
# curve_fit returns the optimized parameters (popt) and the covariance matric (pcov)
# the diagonal of the covariance matrix can be used as simple error estimates of the
    parameters
ax.plot(xs, lorentzian(xs, *popt), color='r', label='fit')
#                          ^ this substitutes (positionally) elements of the iterable
    into the function call

print(f"Estimated height: {popt[0]:.3f} +/- {np.sqrt(pcov[0,0]):.3f}")
print(f"Estimated center: {popt[1]:.3f} +/- {np.sqrt(pcov[1,1]):.3f}")
print(f"Estimated width: {popt[2]:.3f} +/- {np.sqrt(pcov[2,2]):.3f}")

```

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

Listing 13: Nonlinear curve fitting.

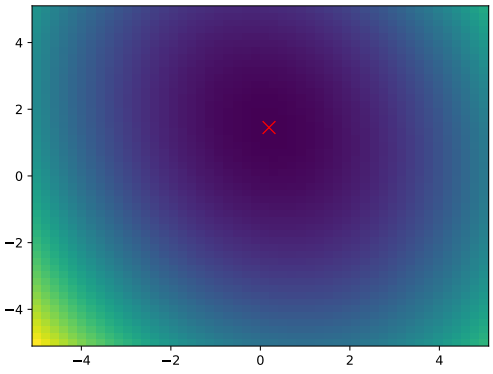


Unlike linear least squares, non-linear curve fitting is iterative and stops once the parameters converge. The fit may never converge for certain problems, therefore, once a configured maximum number of iterations is exceeded the fitting routines *raise an exception* (See 6) which will crash the program 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,

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as optim

def parabola(x, y):
    return 4*x**2 + 2*y**2 + x*y - 6*y -3*x + 5

#the lambda is there to simply turn a function of two parameters into a
#function of a single parameter
xmin = optim.minimize(lambda z: parabola(*z), x0=[0,0])

XX, YY = np.meshgrid(np.linspace(-5, 5), np.linspace(-5, 5))
p = parabola(XX, YY)
fig, ax = plt.subplots()
ax.pcolormesh(XX, YY, p)
ax.plot(*xmin.x, 'x', color='r', ms=10)
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., $\mathrm{cov}(p_i, p_j) = \langle (p_i - \bar{p}_i)(p_j - \bar{p}_j) \rangle$. The diagonal values of the covariance matrix can be used as an estimate of the uncertainties of the fit parameters, i.e.,

```
    #assuming that f is def'ed as f(x, p1, p2)
    p, cov = curve_fit(f, x, y)
    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 $\chi^2 = \sum r_i^2$[8]

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

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

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

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

```
#%% fit the data
#...
bootstrap_N = 100
parameter_N = 3
data_N = len(xs)
```

[8] 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`

```
ps = np.empty((bootstrap_N, parameter_N)) # here we will save all fit parameters
for k in range(bootstrap_N):
    ix = rng.choice(range(data_N), data_N) # some data will be omitted, some will
    occur more than once
    ix.sort()
    popt, _ = curve_fit(lorentzian, xs[ix], ys[ix], p0=[1, 1, 1])
    ps[k,:] = popt

# the fit function does not depend on the sign of the parameters center and width
# and curve_fit will randomly find one or the other. Make sure that we are averaging
# the same signs
ps = abs(ps)
# do the statistics on the set of parameters ps, which is a 2D array.
# work only along one axis, we don't want to average everything together
popt_bootstrap = np.mean(ps, axis=0) #1st axis (0th) averages the rows
perr_bootstrap = np.std(ps, axis=0) #the same for standard deviation
ax.plot(xs, lorentzian(xs, *popt_bootstrap), color='r', label='fit')

print(f"Estimated height: {popt_bootstrap[0]:.3f} +/- {perr_bootstrap[0]:.3f}")
print(f"Estimated center: {popt_bootstrap[1]:.3f} +/- {perr_bootstrap[1]:.3f}")
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.

```

def faulty_function():
    raise ValueError("blergh!")

xs = [-2, -1, 0, 1, 2]
one_over_xs = []
try:
    for x in xs:
        try:
            one_over_xs.append(1/x)
            if x > 1:
                faulty_function()
        except ZeroDivisionError:
            print("Can't divide by zero!")
        except:
            print("something else went wrong")
            raise #propagate the exception further up
finally:
    print("I will always run")

# unless the exception that is raised on line 8 and then sent forward on line 13
    isn't
# handled, this line will not run
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-`raise`d, 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 15. Fit the absolute value $r(f) = \sqrt{x^2 + y^2}$ of the response in exercise 11 to the response of a linear harmonic oscillator plus a polynomial background (polynomial degree 3) and plot the results similarly to exercise 11. Use the simple estimation from exercise 10 as initial estimates of the fit parameters. Use the solution to exercise 10 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)

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

where ω_0 is the (angular) resonance frequency, ω is the frequency of the force F, m is the oscillator mass and γ is damping.

Exercise 16. Estimate the errors of fitting parameters obtained in exercise 15 using bootstrap.

Intermezzo 7: *Object-oriented programming* (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.,

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

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

```
class Difference:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"{self.x} - {self.y}"

    def value(self):
        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.

```
    d1 = Difference(3, 5) #__self__() is called with x=3, y=5 and self refers to
    d1
    d2 = Difference(30, 50)

    print(d1.value()) # -2
    print(d2) # "30 - 50"
```

Several other special method names exist (see `https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types` 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 17. 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 `Fraction`s

4 Digital signal processing

4.1 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 Δt is the **sampling period** and $f_s = 1/\Delta t$ is the **sampling frequency**.

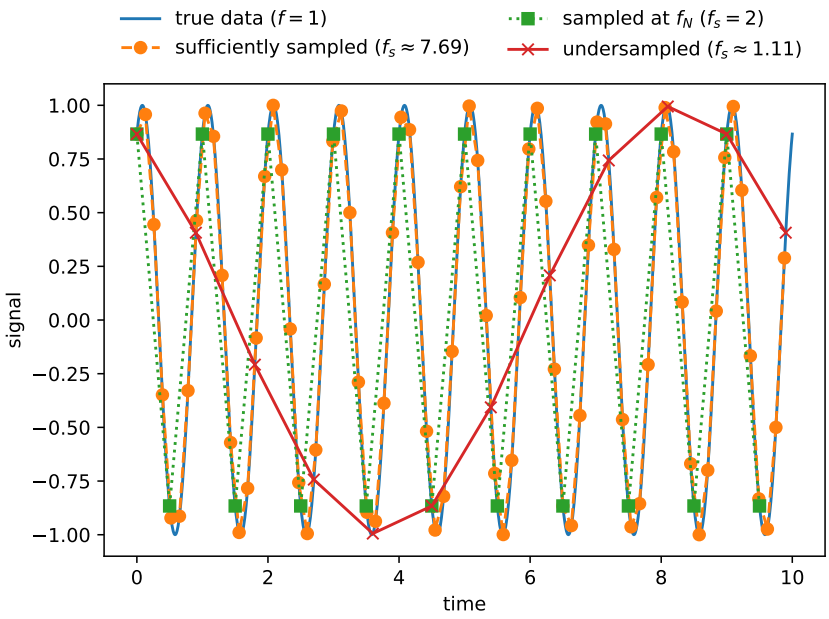
Any information about the original signal $y(t)$ that varies in time faster than Δt is lost. However, if $y(t)$ oscillates at a frequency f it is insufficient to have $f_s \geq f$. Signal frequencies $f > f_s/2$ will appear shifted to $f - f_s/2$, see the output of Lst. 16. The sampled signal contains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

This distortion of 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**

$$f_N = f_s/2. \tag{5}$$

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.interpolate as intp

#create "continuous" data (very high sampling rate)
T = 10 #total length of data in "time" units
xs = np.linspace(0, T, 10000)
ys = np.sin(2*np.pi*xs + 7*np.pi/3)

#interpolate the signal at high sampling rates
signal = intp.interp1d(xs, ys)

def sample(signal, dx, T=T):
    """Samples the quasi-continuous signal with total length T with sample spacing dx
    """
    xs = np.arange(0, T, dx)
    return xs, signal(xs)

fig, ax = plt.subplots()
ax.plot(xs, ys, '-', label='true data ($f = 1$)')

ax.plot(*sample(signal, 0.13), '--o', label=r'sufficiently sampled ($f_s\approx 7.69$)')
ax.plot(*sample(signal, 0.5), ':s', label=r'sampled at $f_N$ ($f_s = 2$)')
#this signal will be aliased
ax.plot(*sample(signal, 0.9), '-x', label=r'undersampled ($f_s \approx 1.11$)')

ax.set_xlabel('time')
ax.set_ylabel('signal')

ax.legend(ncol=2, frameon=False,
          loc='lower left', bbox_to_anchor=(0.0, 1.01))
fig.tight_layout()
fig.savefig('sampling.pdf')
```

Listing 16: Effect of sampling rate



4.2 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,

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

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

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

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

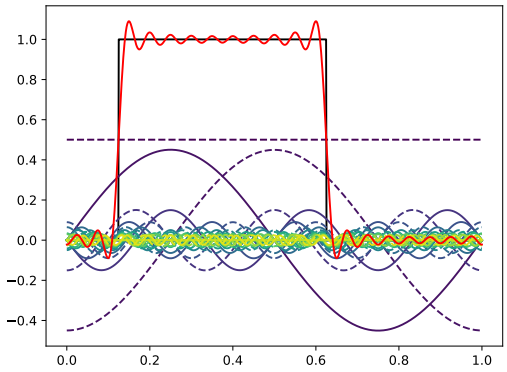
$$A_k = \int_0^T y(t) e^{-2\pi i k t/T} \mathrm{d}t \tag{8}$$

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

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

ts = np.linspace(0, 1, 1000)
dt = ts[1] - ts[0]
ys = np.zeros_like(ts)
start = 0.125
ys[np.logical_and(ts > start, ts < start + 0.5)] = 1

plt.close('all')
fig, ax = plt.subplots()
ax.plot(ts, ys, 'k')
```

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

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 **Fouerier transform**

$$\tilde{y}(\omega) = \int_{-\infty}^{\infty} y(t) e^{-i\omega t} \mathrm{d}t, \tag{9}$$

and the **inverse Fourier transform**

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

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., $\int_{\omega_0}^{\omega_1} |\tilde{y}(\omega)|^2 \mathrm{d}\omega$ is the power of the signal in the frequency band (ω_0, ω_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`)

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

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,

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

The integer frequency indices k represent frequencies $f'_k = k/N$ for $k = 0 \ldots N/2$ and $f'_k = -k/N$ for $k = N/2 \ldots N-1$. 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.

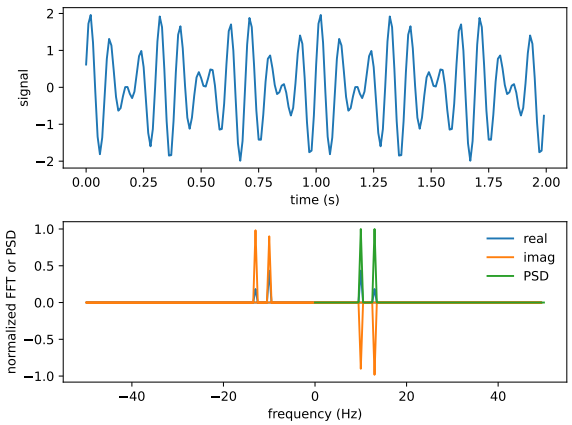
Fourier transforms are implemented in NumPy and SciPy using the **Fast Fourier Transform – FFT** algorithm[9] in the `scipy.fft` submodule. Fourier transform is calculated using `scipy.fft.fft()`

[9]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

```
import numpy as np
import scipy.fft as fft
from scipy.signal import periodogram

import matplotlib.pyplot as plt

#first create some signal
N = 200 #total number of points
dt = 0.01 # sampling period, sampling frequency = 1/dt = 100

ts = np.arange(N)*dt
ys = np.sin(2*np.pi*10*ts + np.pi/7) + np.sin(2*np.pi*13*ts + np.pi/17)

fig, (ax_sig, ax_fft) = plt.subplots(2, 1)
ax_sig.plot(ts, ys)
ax_sig.set_xlabel('time (s)')
ax_sig.set_ylabel('signal')

ax_fft.set_xlabel('frequency (Hz)')
ax_fft.set_ylabel('normalized FFT or PSD')

fft_frequencies = fft.fftfreq(N, dt)
tildey = fft.fft(ys)
ax_fft.plot(fft_frequencies, tildey.real/abs(tildey).max(), label='real')
ax_fft.plot(fft_frequencies, tildey.imag/abs(tildey).max(), label='imag')

#power spectral density
#periodogram() also returns the frequences
psd_freq, psd = periodogram(ys, fs=1/dt) #fs = sampling frequency
ax_fft.plot(psd_freq, psd/psd.max(), label='PSD')
ax_fft.legend(loc='best', frameon=False)
fig.tight_layout()
fig.savefig('fouriers.pdf')
```

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



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

ts = np.linspace(0, 1, 1001)
dt = ts[1] - ts[0]
N = len(ts)
ys = np.zeros_like(ts)
start = 0.125
ys[np.logical_and(ts >= start, ts < start + 0.5)] = 1

plt.close('all')
fig, ax = plt.subplots()
ax.plot(ts, ys, 'k')

# FFT returns all frequencies it can
#ys is real, so we can use rfft
Y = fft.rfft(ys)
frequencies = fft.rfftfreq(N, d=dt)

#how many fourier terms do we want o look at
K = 50

total = 0
cmap = plt.get_cmap('viridis')
norm = plt.Normalize(vmin=0, vmax=K)

for k in range(K):
    #the fourier transform calculates the integral with exp(-i*omega*t)
    #so the signes and real/imag relation to the A and B of the sines and
    #cosines is slightly different
    A = -Y[k].imag/N
    B = Y[k].real/N
    #for oscillating terms we have to double the amplitude for the same
    #reason as when we were calculating A's and B's directly
    if k > 0:
        A *= 2
        B *= 2
    # the 2/N factor comes from the particular normalization used to
    # define fourier transform in SciPy. Various definitions exist.
    sint = A*np.sin(2*np.pi*frequencies[k]*ts)
    cost = B*np.cos(2*np.pi*frequencies[k]*ts)

    ax.plot(ts, sint, color=cmap(norm(k)))
    ax.plot(ts, cost, '--', color=cmap(norm(k)))

    total += sint + cost
ax.plot(ts, total, '-', color='r')

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

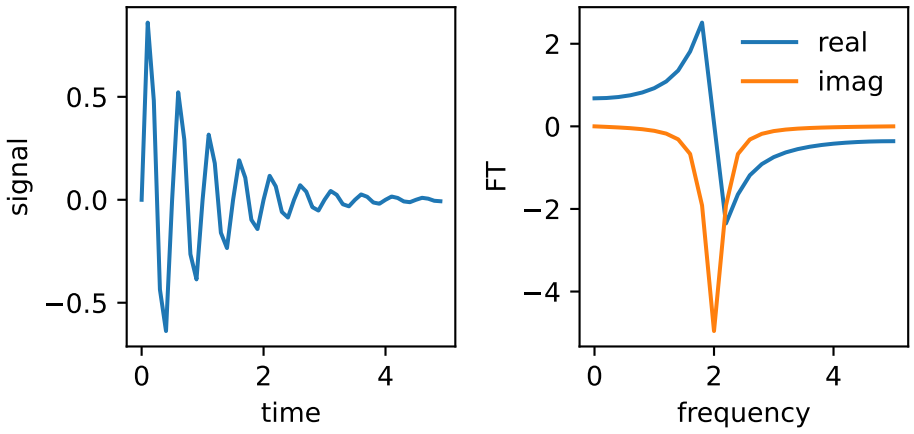
Listing 19: Fourier coefficient of a square wave calculated using FFT

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

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

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:

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

fig, (axt, axf) = plt.subplots(1, 2, figsize=(5, 2.5))

tau = 1
dt = 0.1
t = np.arange(0, 5, dt)
y = np.exp(-t/tau)*np.sin(2*np.pi*2*t)
axt.plot(t, y)

Y = fft.rfft(y)
freq = fft.rfftfreq(len(t), dt)
axf.plot(freq, Y.real, label='real')
axf.plot(freq, Y.imag, label='imag')

axt.set_xlabel('time')
axt.set_ylabel('signal')

axf.set_xlabel('frequency')
axf.set_ylabel('FT')
axf.legend(loc='best', frameon=False)
fig.tight_layout()

fig.savefig('../decaying_exponential.pdf')
```



Intermezzo 8: *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

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

and loading

```
    my_data = np.load("my_data.npy", allow_pickle=True).item()
    print(my_data['date of measurement'])
    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 18. 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 19. 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 8) 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['samples']`. Plot only one file (the pulse is the same for all).

Exercise 20. 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 21. 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).

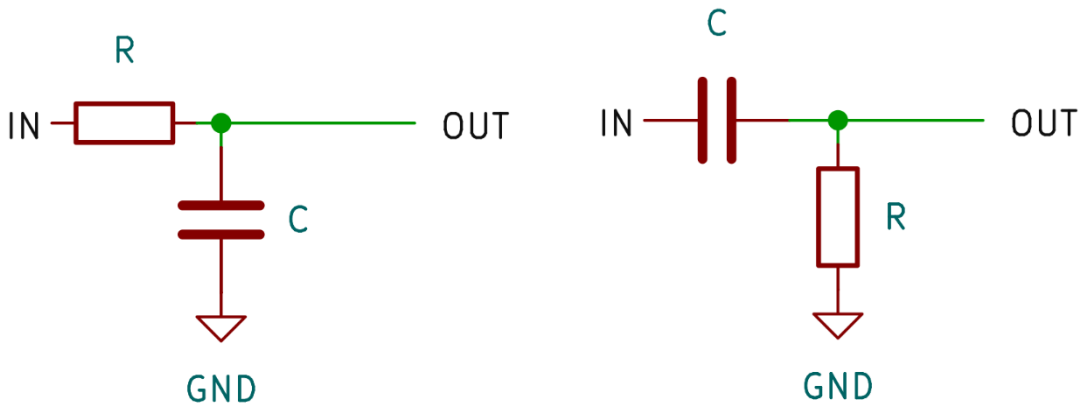


Figure 1: Low-pass (left) and high-pass (right) RC filter.

Exercise 22. As exercise 21, 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

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

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

$$g = 20 \log_{10} \frac{V_{\text{out}}}{V_{\text{in}}} \tag{15}$$

For "absolute" quantities measured in dB (e.g., sound amplitude is common) the measurement is defined with respect to some agreeed upon reference value. For sound the acoustic power (i.e., pressuer squared) is measured relative to 20 µP 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 Ωload about 0.224 V_{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.

4.3 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. 4.3. Depending on the arrangement of the rezistor and capacitor we create a circuit which either attenuates low or high frequences.

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

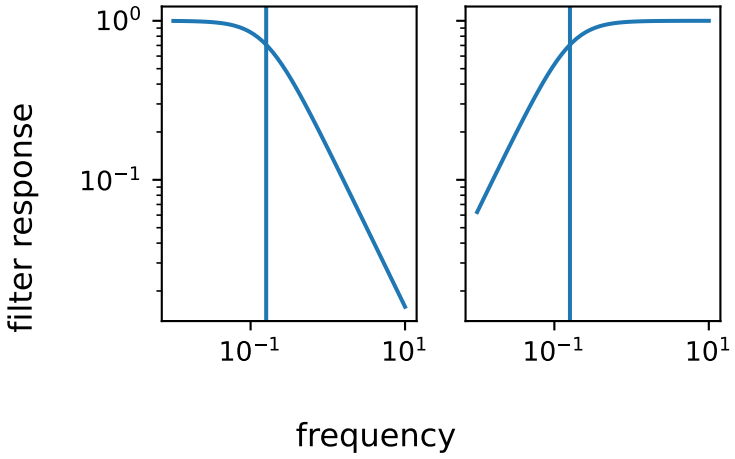
$$\frac{V_{\text{out}}}{V_{\text{in}}} = \frac{1}{1 + i\omega RC}, \tag{16}$$

and for the high pass

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

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

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



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 out signal and $\hat{s}(f)$ is its Fourier transform, the Fourier transform of the signal filtered with a tranfer functino $H(f)$ is simply

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

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.

$$s'(t) = (h * s)(t) \equiv \int_{-\infty}^{t} s(t')h(t-t')\mathrm{d}t, \tag{19}$$

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 ourself 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

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

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 badn 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 filter are 6 dB/oct, since every doubling of frequency remove 6 dB of transmitted power.

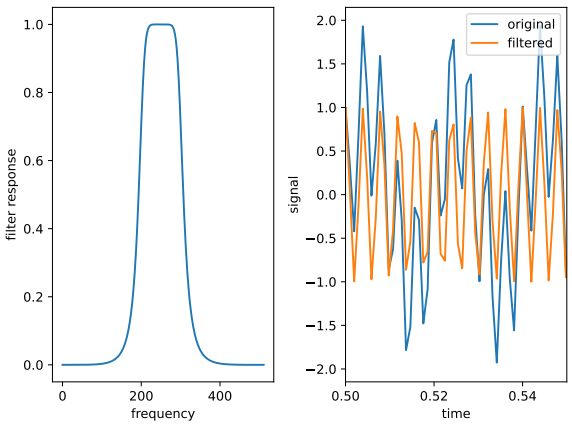
Faster filters can be constructed both electronically and digitally, but as you saw in the examples with the step function, sharp cutoff in the spectrum leads to oscillations, which are typically undesirable. The filters that are maximally flat in the pass-band are the Butterowrth filters, named after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

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

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

```
import scipy.signal as sig
N = 4  # filter order, this would be 24 dB/oct
fs = 1024 # samplig frequency
Wn = [200, 300] # corner frequencies of a band-pass filter

# sos = second order sections
# internal representation of the transfer function
# recommended by scipy
sos = sig.iirfilter(N, Wn, fs=fs, btype='bandpass', output='sos')

#create some signal and filter it
ts = np.arange(0, 1, 1/fs)
ys = np.sin(2*np.pi*50*ts) + np.cos(2*np.pi*250*ts)
ys_filt = sig.sosfilt(sos, ys)

freq, H = sig.sosfreqz(sos, fs=fs)
axH.plot(freq, abs(H))
axsig.plot(ts, ys, label='original')
axsig.plot(ts, ys_filt, label='filtered')
```

:



Exercise 23. 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 24. Demodulate the cleaned-up signal from Exc. 23 (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 25. Calculate the signal envelope from Exc. 24 using the Hilbert transform.

4.4 Interpolation and smoothing.

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

5 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

```
import pyvisa as vi

rm = vi.ResourceManager()

print("Available resources:")
resources = rm.list_resources()
for k, res in enumerate(resources):
    print(f'VISA address {k}: {res}')

#assuming that our pico is on the last address (usually the case)
pico = rm.open_resource(resources[-1],
                        read_termination='\n',
                        write_termination='\n')
print("Identification of the last device:")
pico.write('*IDN?')
resp = pico.read()
print(resp)

print("IDN via query:")
print(pico.query("*IDN?"))

pico.close()
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

```
: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

```
: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

```
import pyvisa as vi

rm = vi.ResourceManager()
dev = rm.open_resource('COM16')
dev.baud_rate = 19200
dev.data_bits = 8
dev.stop_bits = vi.constants.StopBits.one
dev.parity = vi.constants.Parity.none

dev.query('*IDN?')
...
```

Exercise 26. 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 27. 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 $n = 0 \ldots 4$ 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 28. Write a program that will indicate the current temperature (obtained using `:READ:T?` command) using the on-board LEDs. Map the range 20-27°C to 0 to 5 LEDs ON. If the temperature exceeds 28°C, raise a `RuntimeError`, which ends the program. Make sure none of the LEDs stay on when the program ends.

Exercise 29. Write a "digital spirit level", i.e., indicate the current tilt along the y-axis (lengthwise 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 30. 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.,

```
import pyvisa as vi

class Pico:
    def __init__(self, rm):
        ...
    def led(self, led_id, onoff):
        #should raise ValueError if led_id > 4
        ...
    def getT(self):
        ...
    def getP(self):
        ...
    def getACC(self):
        ...
    def getGYR(self):
        ...
    def close(self):
        "Turn off all LEDs and close the session."
        ...

rm = vi.ResourceManager()
pico = Pico(rm)
pico.led(2, 1) #turns the middle LED ON

```

5.1 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,

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

where the `with` statament 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 supressed.

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

```
import numpy as np

class MyNumpyFile:
    def __init__(self, filename, mode='w'):
        print("Opening file.")
        self.file = open(filename, mode)

    def write(self, data):
        np.savetxt(self.file, data)

    def __enter__(self):
        print("Entering with statement")
        return self
    def __exit__(self, *exc):
        print("Closing.")
        self.file.close()

print("I'm about to open the file.")
with MyNumpyFile("my_array.txt", 'w') as file:
    file.write(np.arange(5))
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 31. Extend the Pico class to support the context management protocol.

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

```
    manager = open("hello.txt", 'w')
    file = manager.__enter__()
    try:
        file.write("Hello Context Management.")
```

```
    except:
        if not manager.__exit__(exception_info)
            raise
    else:
        manager.__exit__(None, None, None)
```

5.2 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

```
import pyvisa as vi
rm = vi.ResourceManager()
pico = rm.open_resource(rm.list_resources()[-1])
pico.read_termination = '\n'
pico.write_termination = '\n'
pico.write(':READ:T?')
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

6 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. 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.

6.1 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:

```
import threading as th

def func(thread_name):
    print(f"I'm inside {thread_name}")

threads = []
for k in range(5):
    thread = th.Thread(target=func, args=(f"thread {k}",))
    thread.start()
    threads.append(thread)

for thread in threads:
    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:

```
import threading as th

class MyThread(th.Thread):
    def __init__(self, thread_name):
        super().__init__()
        self.thread_name = thread_name
    def run(self):
        print(f"I'm inside {self.thread_name}")

threads = []
for k in range(5):
    thread = MyThread(f"thread {k}")
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()
```

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 exectures `query(':READ:P?')` and the the other `query(':READ:T?')` there is a chance that the actual order of executed reads and writes will be

```
#thread 1
write(':READ:P?')
# thread 2 running
# thread 2 running
read()
```

```
# thread 2
# thread 1 running
write(':READ:T?')
read()
# 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

```
from threading import Lock
lock = Lock()
```

```
# thread 1
lock.acquire()
# thread 2 running
write(':READ:P?')
read()
lock.release()
# thread 2 running
# thread 2 running
# thread 2 running
```

```
# thread 2
# thread 1 running
lock.acquire() # blocks, until lock is released
# thread 1 running
# thread 1 running
# thread 1 running, lock.acquire() returns
write(':READ:T?')
read()
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 used them in the `with` statament rather than calling the `acquire()` and `release()` directly. A more complete example,

```
import pyvisa as vi
from threading import Thread, Lock

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

def readP(lock):
    with lock: # lock is acquired
        # this will run uninterrupted
        P = pico.query(':READ:P?')
        print(P)
    #lock is released

def readT(lock):
    with lock:
        T = pico.query(':READ:T?')
        print(T)

lock = Lock()
t1 = Thread(target=readP, args=(lock,))
t2 = Thread(target=readT, args=(lock,))

t1.start()
t2.start()
t1.join()
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

```
    # thread 1
    # tries to acquire lock1 and lock2
    # in this order
    lock1.acquire()
    # thread 2 running
    # thread 2 running
    lock2.acquire() # blocks forever
```

```
    # thread 2
    # tries to acquire lock2 and lock1
    # in this order
    # thread 1 running
    lock2.acquire()
    lock1.acquire() # blocks forever
    # 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.,

```
import pyvisa as vi
from threading import Thread, Lock, Event
import time()

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

def keep_reading_P(lock, end_event):
    #keep running until the end_event is set
    while not end_event.is_set():
        with lock: # lock is acquired
            # this will run uninterrupted
            P = pico.query(':READ:P?')
            print(P)
        #lock is released
        time.sleep(1)

lock = Lock()
end = Event()
thr = Thread(target=keep_reading_P, args=(lock,end))
t0 = time.time()
thr.start()
# measure for five second and then signal the thread to end
while True:
    if time.time() - t0 > 5:
        end.set()
        break
    time.sleep(0.1)
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

```
from threading import Thread
from queue import Queue

def thread_func(q):
    while True:
        # waits until a message becomes available
        msg = q.get()
        print(f"Received message {msg}")
        if msg == 'quit':
            break

q = Queue()
thread = Thread(target=thread_func, args=(q,))
thread.start()

```

```
messages = ['hello', 'world', 'quit']
for msg in messages:
    #send the message
    q.put(msg)

thread.join()
```

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

```
import pyvisa as vi
import matplotlib.pyplot as plt
from threading import Thread, Event
from queue import Queue
import time

class Plotter(Thread):
    def __init__(self, queue):
        super().__init__()
        self.fig, self.ax = plt.subplots()
        self.xdata = []
        self.ydata = []
        self.line, = self.ax.plot(self.xdata, self.ydata, '-o')
        self.queue = queue
        self.end = Event()

    def update_plot(self):
        if len(self.xdata) > 0:
            self.line.set_xdata(self.xdata)
            self.line.set_ydata(self.ydata)
            xmin, xmax = min(self.xdata), max(self.xdata)
            ymin, ymax = min(self.ydata), max(self.ydata)
            xmid = 0.5*(xmin + xmax)
            ymid = 0.5*(ymin + ymax)
            dx = xmax - xmin
            dy = ymax - ymin
            self.ax.set_xlim(xmid - 0.55*dx, xmid + 0.55*dx)
            self.ax.set_ylim(ymid - 0.55*dy, ymid + 0.55*dy)
            # plt.draw()

    def pull_data(self):
        # keep reading the data from the queue as long
        # as anything is available
        while not self.queue.empty():
            # if get() is called on an empty queue, it blocks
            # until something becomes available (or a timeout occurs)
            data = self.queue.get()
            print("Received ", data)
            #we can send anything through the queue, for example
            #either a data point or a command to quit
            match data:
                case (x, y):
                    self.xdata.append(x)
                    self.ydata.append(y)
                case 'quit':
                    print("Quitting")
                    self.end.set()

    def run(self):
        while not self.end.is_set():
            self.pull_data()
            self.update_plot()
            time.sleep(0.5)


rm = vi.ResourceManager()
pico = rm.open_resource(rm.list_resources()[-1],
```

```
                        read_termination='\n',
                        write_termination='\n')

data_queue = Queue()
plotter = Plotter(data_queue)
plotter.start()
try:
    t0 = time.time()
    while True:
        t = time.time() - t0
        P = float(pico.query(':READ:P?'))
        print("Sending ", t, P)
        data_queue.put((t, P))
        # the following line updates all open matplotlib plots
        # it MUST be run from the main thread
        plt.pause(0.01)
finally:
    data_queue.put('quit')
    plotter.join()
```

Exercise 32. Write a program that will flash all 5 LEDs with intervals of 0.1, 0.2, 0.5, 1, and 2 s.

Exercise 33. 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.

6.2 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`

```
from multiprocessing import Process, Event, Queue
from queue import Empty
import time

class MyProcess(Process):
    def __init__(self, procname, end, data):
        super().__init__()
        self.procname = procname
        self.end = end
        self.data = data
    def run(self):
        while True:
            try:
                msg = self.data.get(timeout=1)
                print(f"{self.procname} received: {msg}")
            except Empty:
                print(f"{self.procname}: Nothing in queue")
                time.sleep(0.1)

            if self.end.is_set() and self.data.empty():
                break

```

```
if __name__ == '__main__':
    data_queue = Queue()
    end_event = Event()
    process_pool = []
    for k in range(5):
        p = MyProcess(f'proc{k}', end_event, data_queue)
        p.start()
        process_pool.append(p)

    for k in range(10):
        data_queue.put(f"message {k}")
        time.sleep(0.2)

    end_event.set()
    for p in process_pool:
        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

```
from multiprocessing import Pool

def function(x):
    return some_calculation(x)

z = [... data ...]

with Pool(6) as pool:
    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,

```
import numpy as np
import matplotlib.pyplot as plt
from multiprocessing import Pool
import time


def fib(x):
    if x < 2:
        return 1
    return fib(x-1) + fib(x-2)

def calc_fibs(z, nproc):
    with Pool(nproc) as pool:
        fibs = pool.map(fib, z)
    return fibs

def timeit(z, nproc):
    t0 = time.time()
    fibs = calc_fibs(z, nproc)
    t1 = time.time()
    print(f"Calculation with {nproc} processes took {t1 - t0:.3f} s")

z = np.arange(35)
timeit(z, 1)
timeit(z, 2)
timeit(z, 4)
timeit(z, 8)
timeit(z, 16)
```

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

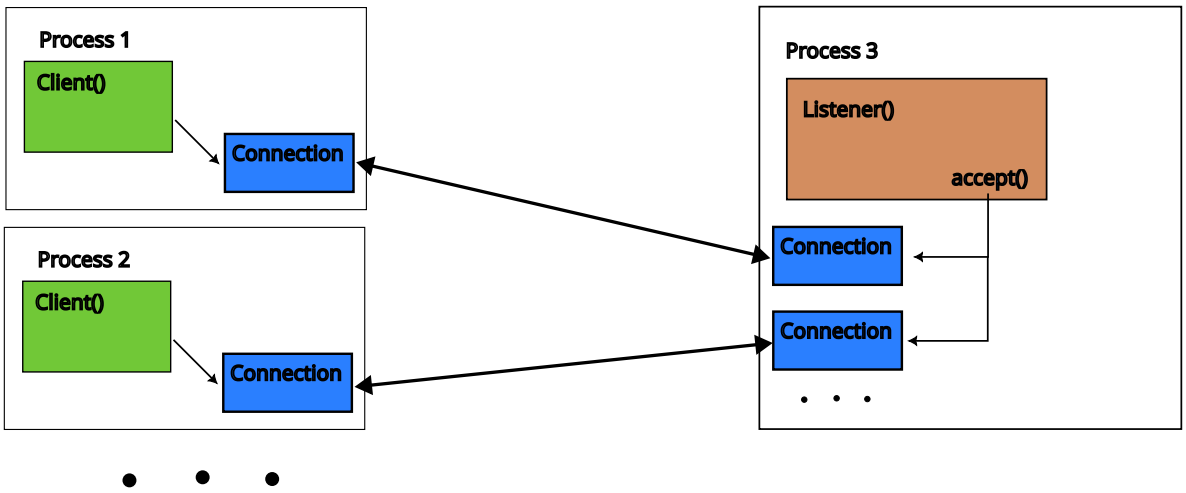


Figure 2: Relationships between Listeners, Clients and Connections

6.3 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 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. 6.3. 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 quite, 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`:

```
from multiprocessing.connection import Listener, Client
from threading import Thread
import time

#we will accept the connections in a separate thread
def run(listener):
    while True:
        # listener.accept() blocks until someone
        # tries to connect
        try:
            with listener.accept() as conn:
                msg = conn.recv()
```

```
                print(f"Received {msg}")
                conn.send(f'Echo {msg}')
        except OSError:
            # OSError is raised when we try to accept
            # with a closed listener
            print("Stopping listening")
            break

# the address and port
# 'localhost' is local computer, use empty string ''
# if you want to access it over the netowrk
# port number 0 means automatic assignment by the OS
address = ('localhost', 6000)
password = b'password'
with Listener(address, authkey=password) as listener:
    print('Address :', listener.address)

    t = Thread(target=run, args=(listener,))
    t.start()
    # now the main thread will just idly sleep, waiting for
    # keyboard interrupt
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print("Quitting...")
    # this will signal the listener to close, however, it
    # will remain open as long as .accept() is blocking
    listener.close()
    # so create a dummy connection to unblock accept()
    # and let the listener close
    try:
        with Client(address, authkey=password) as c:
            c.send('')
            c.recv()
    except ConnectionRefusedError:
        # someone connected before our dummy connection
        # and the listener shut down, don't do anything
        pass
    t.join()
```

client.py

```
import numpy as np
from multiprocessing.connection import Client

address = ('localhost', 6000)
passowrd = b'password'
with Client(address, authkey=passowrd) as conn:
    # send and recv pickle and unpickle, respectively
    # the objects we try to send. So we can send essentially
    # arbitrary data
    conn.send({'key1': 'hello', 'key2': np.arange(5)})
    resp = conn.recv()
    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 34. 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.

`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 threahds 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)

```
from multiprocessing.connection import Listener, Client
from threading import Thread, Lock, Event
import time

import pyvisa as visa

#for wrong passwords
from multiprocessing.context import AuthenticationError

rm = visa.ResourceManager()
pico = rm.open_resource(rm.list_resources()[-1],
                        write_termination='\n',
                        read_termination='\n')
pico_lock = Lock()

handlers_lock = Lock()
client_handlers = []

class ClientHandler(Thread):
    def __init__(self, conn, name=None):
        super().__init__()
        self.name = name
        self.conn = conn
        self._end = Event()

    def stop(self):
        self._end.set()

    def run(self):
        try:
            while not self._end.is_set():
                if self.conn.poll():
                    msg = self.conn.recv()
                    print(f"Received : {msg}")
                    if msg == '':
                        self.conn.send('')
                    else:
                        with pico_lock:
                            resp = pico.query(msg)
                        self.conn.send(resp)
                time.sleep(0.1)
        except EOFError:
            print(f"Client {self.name} quit")
        finally:
            self.conn.close()
            with handlers_lock:
                client_handlers.remove(self)

def run(listener):
    while True:
        try:
            print("Waiting for connection")
            conn = listener.accept()
            if hasattr(listener, 'last_accepted'):
                client_name = listener.last_accepted
            else:
                client_name = None
            handler = ClientHandler(conn, client_name)
```

TODO

Figure 3: Single computer controlling multiple experiments.

```
            handler.start()
            client_handlers.append(handler)
        except OSError:
             print("Stopping listening")
             break
        except AuthenticationError:
             print("Connection attempt with wrong password.")

address = ('', 0)
password = b'NOFY080_2024'
try:
    with Listener(address, authkey=password) as listener:
        actual_address = listener.address
        print('Address :', actual_address)

        t = Thread(target=run, args=(listener,))
        t.start()

        try:
            while True:
                time.sleep(1)
        except KeyboardInterrupt:
            print("Quitting...")

        listener.close()

        try:
            #dummy connection to force the listener to close
            with Client(actual_address, authkey=password) as c:
                c.send('')
                c.recv()
        except ConnectionRefusedError:
            pass
        t.join()
        for handler in client_handlers:
            handler.stop()

        for handller in client_handlers:
            handler.join()
finally:
    pico.close()
    rm.close()
```

6.3.1 Real use case in a lab

In Fig. 6.3.1 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.

7 Solutions of differential equations

7.1 Initial value problems

Initial value problems are differential equations of the form

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(\mathbf{y}, t), \tag{21}$$

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 $\mathbf{y}(\mathrm{d}t) \approx \mathbf{y}(0) + \mathbf{f}(\mathbf{y}(0), 0)\mathrm{d}t$; $\mathbf{y}(2\mathrm{d}t) \approx \mathbf{y}(\mathrm{d}t) + \mathbf{f}(\mathbf{y}(\mathrm{d}t), \mathrm{d}t)\mathrm{d}t$ .... This time stepping is called the *Euler* method and generally requires a small time step to be accurate and numerically stable.[10] 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 $O(\mathrm{d}t)$, or a first-order method, which means that if we halve the time step $\mathrm{d}t$, 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 $O(\mathrm{d}t^4)$) 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 $y(t + \mathrm{d}t)$ from $y(t)$ as

$$y(t + \mathrm{d}t) = y(t) + \frac{\mathrm{d}t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \tag{22}$$

where

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

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, the take-up 4 times as much space (e.g., Adams-Bashforth 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`. Remeber, 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° with initial velocities of 3 m/s along both x and y directions. The ballistic trajectory is the trajectory of a projectile thrown

[10] Numerical *in*stability 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

$$\mathbf{F}_d = -\frac{1}{2}A|\mathbf{v}|\mathbf{v}\rho\mathrm{CD}, \tag{24}$$

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

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

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

g = 9.81 #m/s^2
rho_air = 1 #kg / m^3
ball_r = 0.1 #m
#drag coefficient, https://en.wikipedia.org/wiki/Drag_coefficient
CD = 0.47
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

```
#differentiation of position and velocity
def F(t, z, CD=0):
    x, y, vx, vy = z

    #total velocity
    v_tot = np.sqrt(vx**2 + vy**2)

    air_resistance = 0.5*rho_air*v_tot**2*CD*A_ball

    dxdt = vx
    dydt = vy
    dvxdt = -air_resistance*vx/v_tot
    dvydt = -g - air_resistance*vy/v_tot
    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

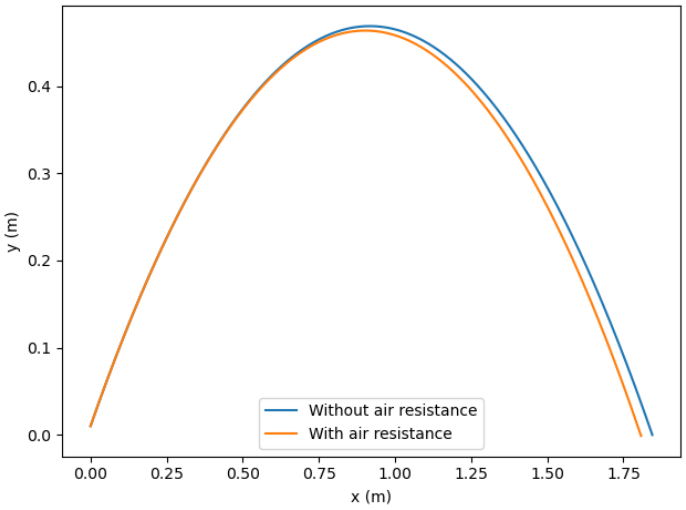
```
def impact(t, z, CD=0):
    return z[1]
# setting the terminal attribute to 0 will cause the calculation
# to end when the event happens
impact.terminal = True
```

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 coeffecient, 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.

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

```
    dt = 1e-3
    print(solution.t_events)
    time = np.arange(0, solution.t_events[0][0]+dt, dt)
    z = solution.sol(time)
    x = z[0,:]
    y = z[1,:]
    return x, y

initial_condition=[0, 0.01, 3, 3]
# without air resistance
x, y = ballistic(initial_condition)
# with air resistance
xvz, yvz = ballistic(initial_condition, CD)
fig, ax = plt.subplots()
ax.plot(x, y, label="Without air resistance")
ax.plot(xvz, yvz, label="With air resistance")
ax.legend(loc='best')
ax.set_xlabel('x (m)')
ax.set_ylabel('y (m)')
fig.tight_layout()
fig.savefig('ballistic.png')
```



Exercise 35. Simulate the population dynamics of predators and prey (foxes F and rabbits R) using the Lotka-Volterra model:

$$\frac{\mathrm{d}R}{\mathrm{d}t} = aR - bFR$$
$$\frac{\mathrm{d}F}{\mathrm{d}t} = dFR - cF$$

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 36. Simulate the Lorenz system (the first properly studied example of deterministic

chaos) given by equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x)$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y$$
$$\frac{\mathrm{d}z}{\mathrm{d}y} = xy - \beta z,$$

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

Intermezzo 9: *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:

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

7.2 Boundary value problems

Boundary value problems are differential equations of the form

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

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 inifinite potential well with additional potential $V(x)$ inside the well.

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

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

where m is the particle mass, ψ 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. 7.2, 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 `dh` 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.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_bvp

a = 1
```

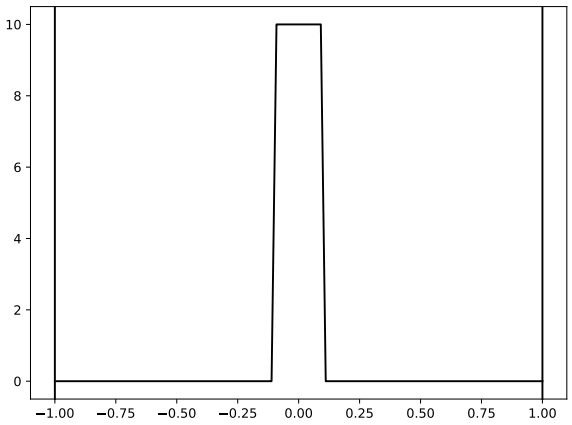


Figure 4: Potential for the Schrödinger equation (26).

```
m = 1
w = 0.2
dh = 10
N = 100

#potential well with a barrier in the middle
def potential(x):
    V = np.zeros_like(x)
    ix = np.logical_and(x > -w/2, x < w/2)
    V[ix] = dh
    return V
```

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

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

and the unknown function $\mathbf{y}$ is then $(\psi, \mathrm{d}\psi/\mathrm{d}x)$. 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.

```
#right-hand side of the Schrodinger equation
def F(x, psi, p):
    E = p[0]
    V = potential(x)
    dpsidx = psi[1]
    dpsidx2 = (V - E)*psi[0]*2*m
    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.

```
def boundaries(ya, yb, p):
    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 × length of `x`.

```
#grid on solve the eqquation
xs = np.linspace(-a, a, N)

#the initial guess of the solution
#the wave function itself
ys0 = np.zeros_like(xs, dtype=complex)
# just so that the initial guess is not identically zero. This choice
# will give us asymmetric solution where the particle is localized in one
# part of the well
ys0[N//4] = 1
#the wave function gradient, we take those simply as zeros
ys1 = np.zeros_like(xs, dtype=complex)
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.

```
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 $\int |\psi(x)|^2 \mathrm{d}x = 1$, 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.

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

7.3 GPU acceleration

Certain types of physical problems can be effeciently 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

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

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 quicktly 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 planests which are independent of each other. Our goal is to use the GPU tu run these in parallel.

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

7.3.1 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

```
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

```
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.

7.3.2 Testing the installation

TODO

7.3.3 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

```
import taichi as ti
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

```
                acc[j,0] += -d[0]/r**3
                acc[j,1] += -d[1]/r**3

@ti.kernel
def step(rs: ti.types.ndarray(dtype=ti.f64),
         vel: ti.types.ndarray(dtype=ti.f64),
         acc: ti.types.ndarray(dtype=ti.f64),
         dt: float):
    N = rs.shape[0]
    for j in range(N):
        for k in range(2):
```

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
def newton(rs: ti.types.ndarray(dtype=ti.f64),
           acc: ti.types.ndarray(dtype=ti.f64)):
    #the types of argument of kernels have to be annotated

    #assume that the array of positions is shape (N, 2)
    N = rs.shape[0]

    for j in range(N):
        # zero out the acceleration of the j'th particle so that we can add to it
        # slices are not supported in taichi
        acc[j,0] = 0
        acc[j,1] = 0

    #this loop runs in parallel, and THERE IS NO SYNCHRONIZATION
    #no mutexes/locks or anything like that. We have to make sure
    #none of the operations we do inside the loop can influence one another
    for j in range(N):
        #only the top-level loop runs in parallel, this one is serialized
        for k in range(N):
            #no self-interaction
            if j != k:
                pk = ti.Vector([rs[k,0], rs[k,1]])
                pj = ti.Vector([rs[j,0], rs[j,1]])

                d = pj - pk
                #this is a cheat, 1e-5 is the minimum possible value
                #this limits the maximum possible velocity for
                #numerical stability
                r = d.norm(1e-5)

                acc[j,0] += -d[0]/r**3
                acc[j,1] += -d[1]/r**3
```

Our problem is relatively simple, so-called *embarrassingly parallel*[11] because the calculations are entirely independet of each other. GPUs are capable of running many task in parallel quickly because 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 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 palents $j = 1$ (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.

7.3.4 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,

```
class Planets:
    def __init__(self, N, D=1, dt=1e-4):
        self.D = D
        self.dt = dt
        #
        self.t = 0
        #randomly distributed initial positions with normal distribution
        #and spread D
        self.rs = D*np.random.randn(N, 2)
        #velocities -- solid body rotation as initial condition
        #so that the system has some nonzero angular momentum
        self.vels = 50*np.column_stack((-self.rs[:,1], self.rs[:,0]))
        #accelerations
        self.acc = np.zeros_like(self.rs)

    def update_accelerations(self):
        #we can simply pass numpy arrays to the kernel
        #this "function", however, does not run inside Python
        #the acc array will be modified
        newton(self.rs, self.acc)

    def euler_step(self, dt=None):
        #Euler step is very inaccurate and requires small dt
        #replace with something better for actually useful simulation
        if dt is None:
            dt = self.guess_dt()
        step(self.rs, self.vels, self.acc, dt)
        self.t += dt
```

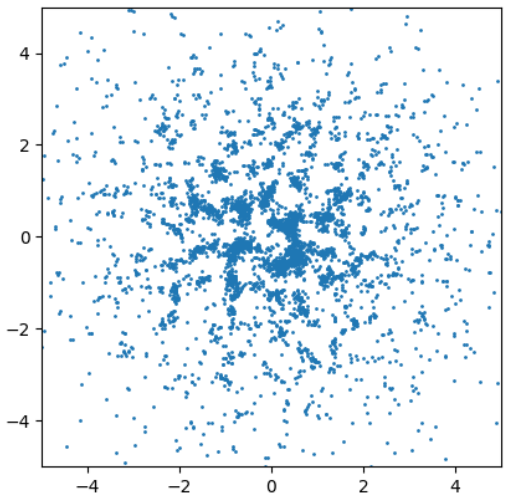
Finally, we create a simulations with 5000 planets, plot a real-time animation using matplotlib and periodically save a picture

```
if __name__ == '__main__':
    import os
```

[11] This is an actual term used in the computer science.

```
    fig, ax = plt.subplots()

    #Create 5000 planets
    planets = Planets(5000, D=1)

    # set up plotting
    plot, = ax.plot(planets.rs[:,0], planets.rs[:,1], 'o', ms=1)
    plot_scale = 5
    ax.set_xlim(-plot_scale, plot_scale)
    ax.set_ylim(-plot_scale, plot_scale)
    ax.set_aspect('equal')

    # where to save the result
    os.makedirs('pictures', exist_ok=True)
    try:
        last_save = 0
        frame = 0
        #the simulation simply proceeds by updating the accelerations
        #and then incrementing the positions and velocities, until
        #the simulations is stopped
        while True:
            planets.update_accelerations()
            planets.euler_step(1e-5)

            # update the interactive plot
            plot.set_xdata(planets.rs[:,0])
            plot.set_ydata(planets.rs[:,1])
            plt.pause(1e-3)

            #Only save the picture. In a real simulation we would save the
            #positions and accelerations so that we can further process them.
            if planets.t - last_save > 1e-3:
                last_save = planets.t
                fig.savefig(f'pictures/frame_{frame:08d}.png', dpi=300)
                frame += 1
            print(planets.t)
    except KeyboardInterrupt:
        print("Qutting...")
```

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



8 What didn't fit

8.1 PID algorithm

TODO

8.2 Symbolic mathematics

TODO: sympy

A Linear Harmonic Oscillator

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

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

where m is the oscillator mass, ω_0 is the resonance frequency, γ 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

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

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

```
    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),

```
    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

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

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

```
import pyvisa as visa
rm = visa.ResourceManager()
print(rm.list_resources())
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.

```
import pyvisa as visa
rm = visa.ResourceManager()
pico = rm.open_resource('ASRL/dev/ttyACM0::INSTR',
                        read_termination='\n',
                        write_termination='\n')
print(pico.query('*IDN?'))
pico.close()
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 $n = 0 \ldots 4$, red LED is 0.

:READ:P?
Reads the pressure in Pa.

:READ:T?
Returns the temperatures as $100T$ where T is the temperature in °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-space separated values in the range -32768 to 32768 which maps to $-500°/s$ to $500°/s$.

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.