TMA4320 - Scientific Computation

Anne Kværno, André Massing

Apr 11, 2025

CONTENTS

This book repository contains the notes for the course TMA4320 Scientific Computation at NTNU. The notes are written in Jupyter notebooks and Myst markdown and are built using Jupyter Book.

The Jupyter notebooks are also meant to be used interactively either during classes and outside of classes. To make sure that the notebooks are displayed correctly and work as expected, you need to have the following Python modules installed in your Python environment:

- ipympl
- ipython
- ipywidgets
- jupyter
- jupyterlab-git
- jupyterlab-myst
- jupyterlab-rise
- matplotlib
- matplotlib-venn
- numpy
- pandas
- plotly
- scipy
- sympy
- tqdm
- webgui-jupyter-widgets

If you also want to generate the Jupyter book itself, you need to have the following Python modules installed in addition to the ones mentioned above:

- jupyter-book
- jupytext
- sphinx
- sphinx_copybutton
- sphinx_exercise
- sphinx_proof
- sphinx_togglebutton
- sphinx-thebe

CHAPTER

ONE

INTRODUCTION TO SCIENTIFIC COMPUTATION

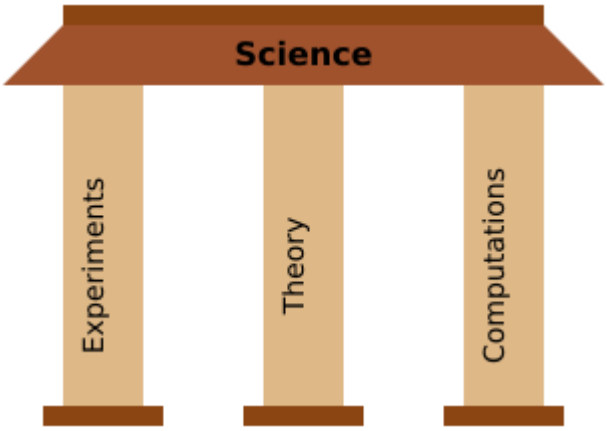
Scientific Computation (Scientific Computing, Computational science)

"*Scientific Computing is the collection of tools, techniques, and theories required to solve on a computer mathematical models of problems in Science and Engineering*" [Golub and Ortega, 2014].

As such Scientific Computing covers a wide range of topics and fields and if you ask 10 different domain experts to define the term *Scientifc Computing*, you will probably get 15 different answers.

This is in a way also reflected in the editorial comments for the Wiki article on Computational Science.

Nevertheless, Scientific Computing is considered the third pillar of science, the others being **Experiments** and **Theory**.



There a lot science disciplines employing the scientific computing for scientific discoveries, e.g.

- Global ocean/climate modeling
- Computational fluid dynamics
- Seismology
- Biophysics
- Population dynamics (e.g.disease spreading)
- Economics
- Medical imaging
- Material science

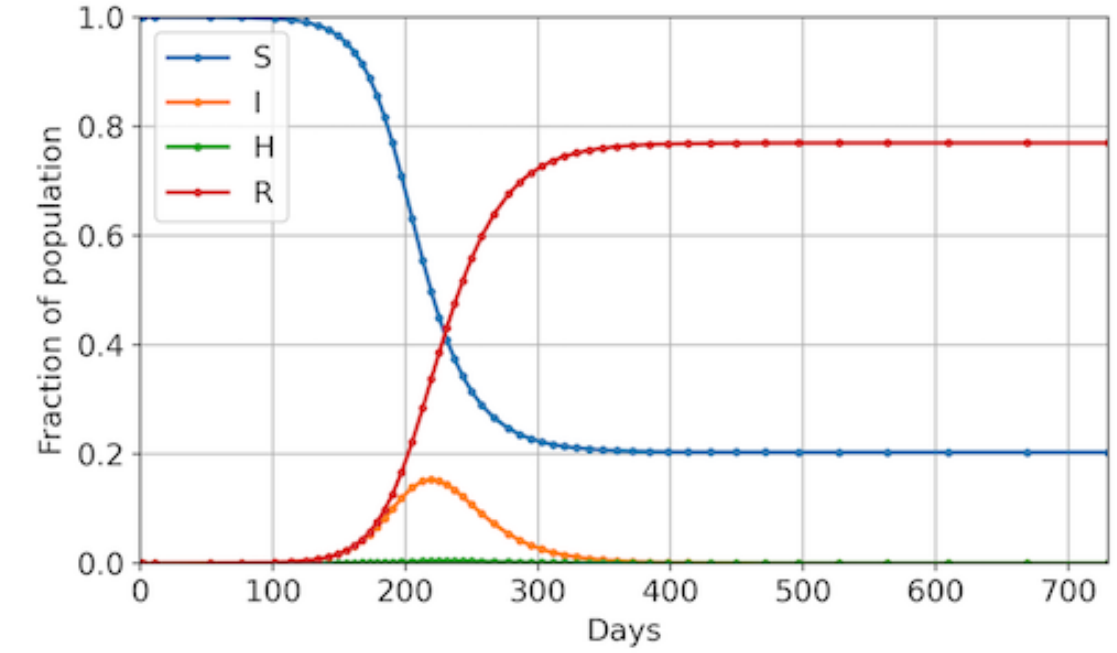
Challenge: Find me a scientific discipline where no computational methods are employed!

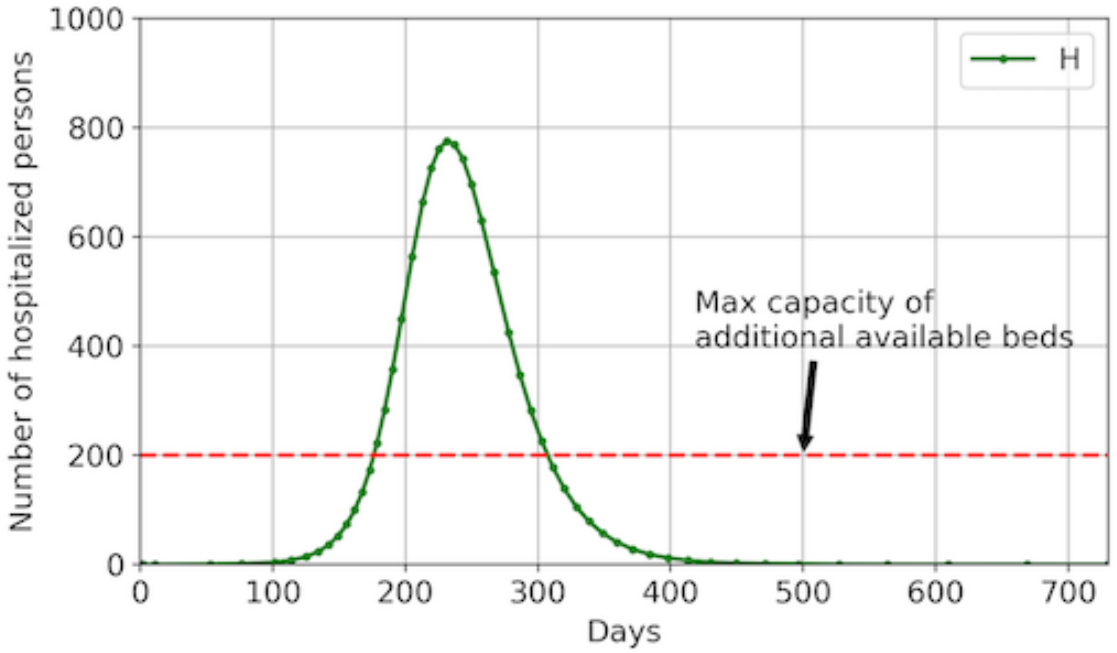
The covid 19 pandemicwas a very good example of a typical science problem involving Scientific Computing, which typically consist of the following steps.

1. Mathematical Modeling
2. Analysis of the mathematical model (Existence, Uniqueness, Continuity)
3. Numerical methods (computational complexity, stability, accuracy)
4. Realization (implemententation)
5. Postprocessing
6. Validation

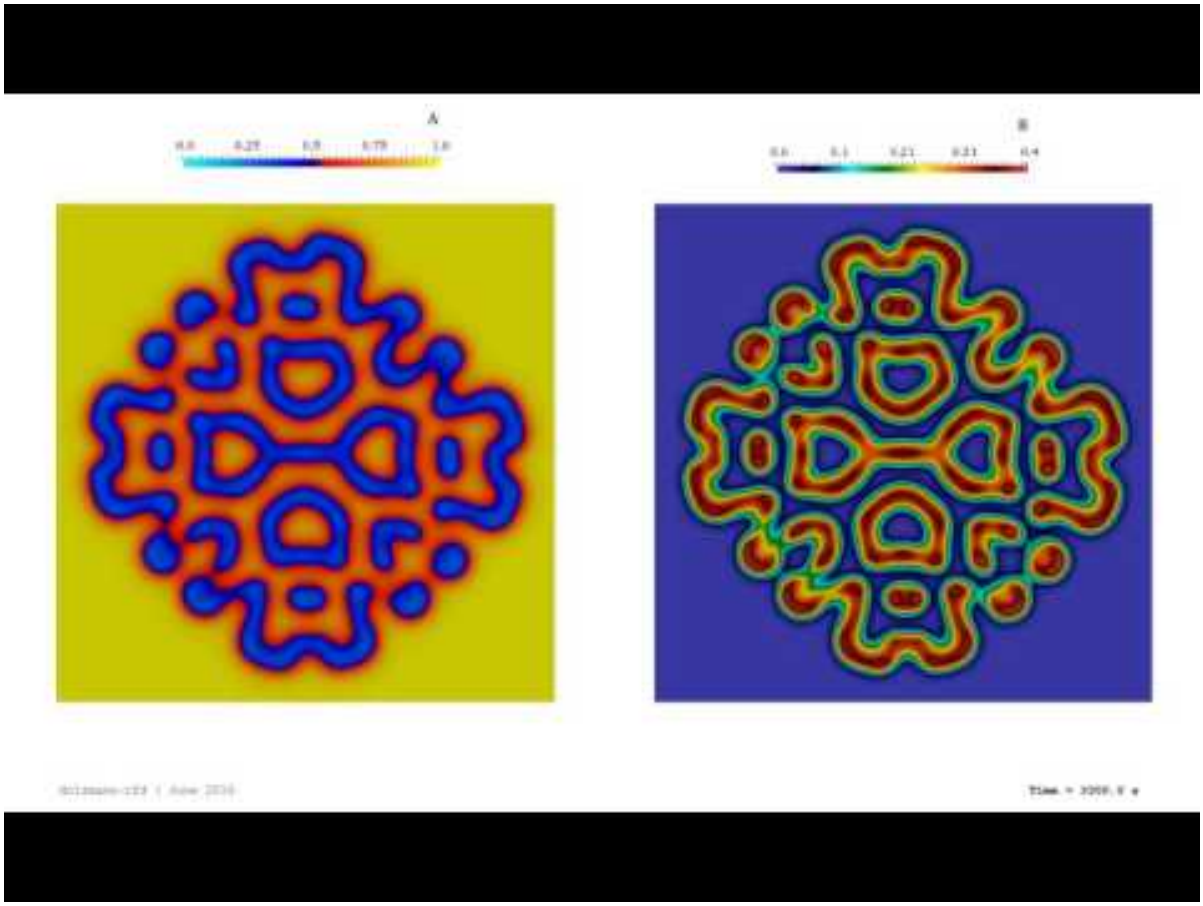
This semester, we will learn about methods which helps you to e.g.

- model and predict the spreading of diseases like Covid 19
- simulate the generation of patterns in biology
- understand how images are compressed

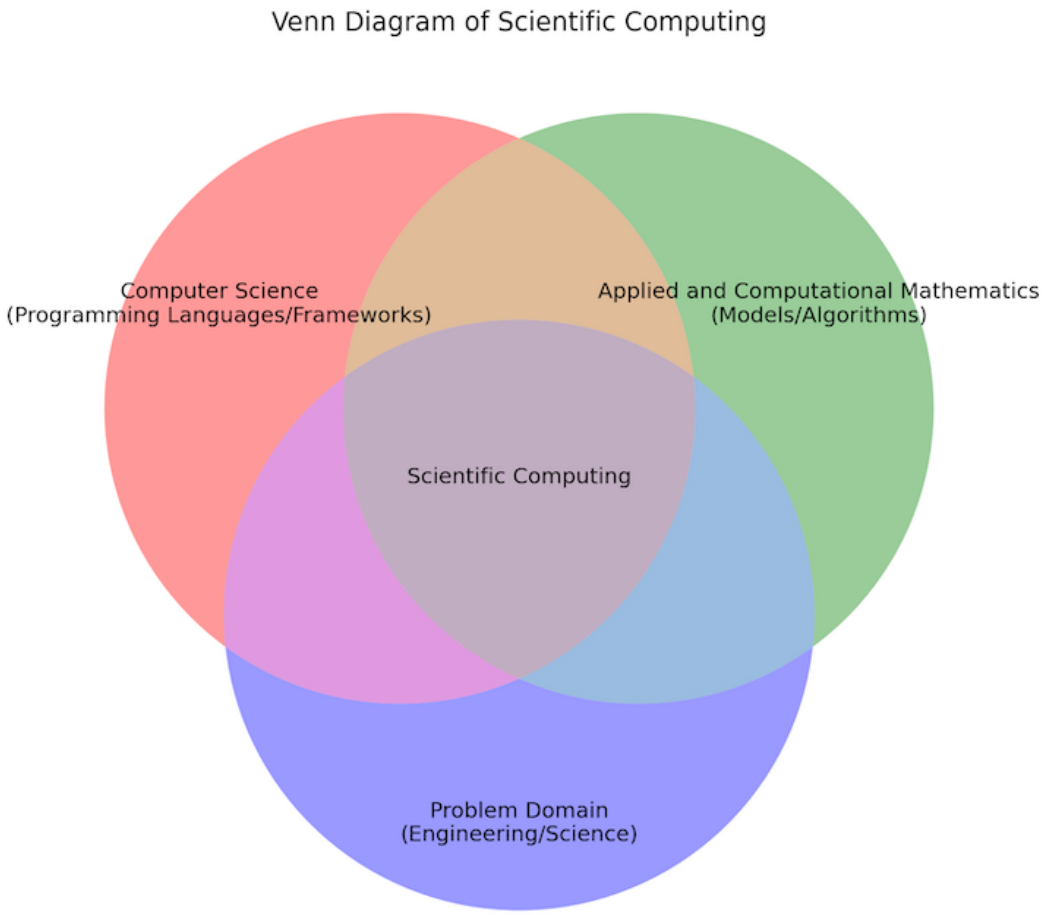





```
from IPython.display import YouTubeVideo, HTML
YouTubeVideo('nw2bPnhtxN8', width=800, height=500)
```



In general, we can think of Scientific as an interdisplinary, computational based approach towards scientific discovery:



Mentimeter time:

Please go to and enter the following code **7381 6459**

1.1 Machine Representation of Numbers

Today we will talk about one important and unavoidable source of errors, namely the way, a computer deals with numbers.

Let's start with two simple tests.

- Define two numbers $a = 0.2$ and $b = 0.2$ and test whether their sum is equal to 0.4.
- Now define two numbers $a = 0.2$ and $b = 0.1$ and test whether their sum is equal to 0.3.

```
# Write your code here
a = 0.2
b = 0.1
sum = 0.3

if (a+b) == sum:
    print("That is what I expected!!")
else:
    print("What the hell is going on??")

diff = a+b
diff = diff - sum
print(f"{diff}")
```

```
What the hell is going on??
5.551115123125783e-17
```

Why is that? The reason is the way numbers are represent on a computer, which will be the topic of the first part of the lecture.

After the lecture I recommed you to take a look which discusses the phenomena we just observed in some detail.

1.1.1 Positional System

On everyday base, we represent numbers using the **positional system**. For instance, when we write 1234.987 to denote the number

$$1234.987 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 + 9 \cdot 10^{-1} + 8 \cdot 10^{-2} + 7 \cdot 10^{-3}$$

using 10 as **base**. This is also known a **decimal system**.

In general for any $\beta \in \mathbb{N}$, $\beta \geqslant 2$, we use the **positional representation**

$$x_\beta = (-1)^s [a_n a_{n-1} \dots a_0 . a_{-1} a_{-2} \dots a_{-m}]_\beta$$

with $a_n \neq 0$ to represent the number

$$x_\beta = \sum_{k=-m}^{n} a_k \beta^k.$$

Here,

- β is called the **base**
- $a_k \in [0, \beta - 1]$ are called the **digits**
- $s \in \{0, 1\}$ defines the **sign**

- $a_n a_{n-1} \dots a_0$ is the **integer** part
- $a_{-1} a_{-2} \dots a_{-m}$ is called the **fractional** part
- The point between a_0 and a_{-1} is generally called the **radix point**

Exercise 1

Write down the position representation of the number $3\frac{2}{3}$ for both the base $\beta = 10$ and $\beta = 3$.

Solution to Exercise 1

- $\beta = 10 : [3.666666666\cdots]_{10}$
- $\beta = 3 : 1 \cdot 3^1 + 0 \cdot 3^0 + 2 \cdot 3^{-1} = [10.2]_3$

To represent numbers on a computer, the most common bases are

- $\beta = 2$ (binary),
- $\beta = 10$ (decimal)
- $\beta = 16$ (hexidecimal).

For the latter one, one uses $1, 2, \dots, 9$, A,B,C,D,E,F to represent the digits. For $\beta = 2, 10, 16$ is also called the binary point, decimal point and hexadecimal point, respectively.

We have already seen that for many (actuall most!) numbers, the fractional part can be infinitely long in order to represent the number exactly. But on a computer, only a finite amount of storage is available, so to represent numbers, only a fixed numbers of digits can be kept in storage for each number we wish to represent.

This will of course automatically introduces errors whenever our number can not represented exactly by the finite number of digits available.

1.1.2 Fix-point system (fasttall system)

Use $N = n + 1 + m$ digits/memory locations to store the number x written as above. Since the binary/decimal point is *fixed* , it is difficult to represent large numbers $\geqslant \beta^{n+1}$ or small numbers $< \beta^{-m}$.

E.g. nowdays we often use 16 (decimal) digits in a computer, if you distributed that evenly to present same number of digits before and after the decimal point, the range or representable numbers is between 10^8 and 10^{-8} **This is very inconvenient**!

Also, small numbers which are located towards the lower end of this range cannot be as accuractely represented as number close to the upper end of this range.

As a remedy, an modified representation system for numbers was introduced, known as **normalized floating point system**.

1.1.3 Normalized floating point system (flyttall system)

Returning to our first example:

$$145397.2346 = 0.1453972346 \cdot 10^6 = 1453972346 \cdot 10^{6-10}$$

In general we write

$$x = (-1)^s 0.a_1 a_2 \dots a_t \beta^e = (-1)^s \cdot m \cdot \beta^{e-t}$$

Here,

- $t \in \mathbb{N}$ is the number of **significant digits** (**gjeldene siffre**)
- e is an integer called the **exponent** (**eksponent**)
- $m = a_1 a_2 \dots a_t \in \mathbb{N}$ is known as the *mantissa*. (*mantisse*)
- Exponent e defines the *scale* of the represented number, typically, $e \in \{e_{\text{min}}, \dots, e_{\text{max}}\}$, with $e_{\text{min}} < 0$ and $e_{\text{max}} > 0$.
- Number of significant digits t defines the **relative accuracy** (**relativ nøyaktighet**).

We define the **finite** set of available **floating point numbers**

$$\mathbb{F}(\beta, t, e_{\text{min}}, e_{\text{max}}) = \{0\} \cup \left\{ x \in \mathbb{R} : x = (-1)^s \beta^e \sum_{i=1}^{t} a_i \beta^{-i}, e_{\text{min}} \leqslant e \leqslant e_{\text{max}}, 0 \leqslant a_i \leqslant \beta - 1 \right\}$$

- Typically to enforce a unique representation and to ensure maximal relative accuracy, one requires that $a_1 \neq 0$ for non-zero numbers.

Exercise 2

What is the smallest (non-zero!) and the largest number you can represent with $\mathbb{F}$?

Solution to Exercise 2

$$\beta^{e_{\text{min}}-1} \leqslant |x| \leqslant \beta^{e_{\text{max}}}(1 - \beta^{-t}) \quad \text{for } x \in \mathbb{F}.$$

Conclusion:

- Every number x satifying $\beta^{e_{\text{min}}-1} \leqslant |x| \leqslant \beta^{e_{\text{max}}}(1 - \beta^{-t})$ but which is **not** in $\mathbb{F}$ can be represented by a floating point number $\text{fl}(x)$ by rounding off to the closest number in $\mathbb{F}$.
- Relative *machine precision* is

$$\frac{|x - \text{fl}(x)|}{|x|} \leqslant \epsilon := \frac{\beta^{1-t}}{2}$$

- $|x| < \beta^{e_{\text{min}}-1}$ leads to **underflow**.
- $|x| > \beta^{e_{\text{max}}}(1 - \beta^{-t})$ leads to **overflow**.

Standard machine presentations nowadays using

- Single precision, allowing for 7-8 sigificant digits
- Double precision, allowing for 16 sigificant digits

1.1.4 Things we don't discuss in this but which are important in numerical mathematics

We see that already by entering data from our model into the computer, we make an unavoidable error. The same also applied for the realization of basics mathematical operations $\{+, -, \cdot, /\}$ etc. on a computer.

Thus it is of importance to understand how errors made in a numerical method are propagated through the numerical algorithms. Keywords for the interested are

- Forward propagation: How does an initial error and the algorithm affect the final solution?
- Backward propagation: If I have certain error in my final solution, how large was the initial error?

CHAPTER

TWO

POLYNOMIAL INTERPOLATION

2.1 Introduction

Polynomials can be used to approximate functions over some bounded interval $x \in [a, b]$. Such polynomials can be used for different purposes. The function itself may be unknown, and only measured data are available. In this case, a polynomial may be used to find approximations to intermediate values of the function. Polynomials are easy to integrate, and can be used to find approximations of integrals of more complicated functions. This will be exploited later in the course. And there are plenty of other applications.

Let's consider the following problem. The estimated mean atmospheric concentration of carbon dioxide in the earth's atmosphere is given in the following table.

year	CO2 (ppm)
1800	280
1850	283
1900	291
2000	370

Is there a simple method to estimate the CO_2 concentration on (a) 1950 and (b) 2050?

This is where **interpolation polynomials** comes into play!

Definition 1 (Interpolation problem)

Given $n+1$ points $(x_i, y_i)_{i=0}^n$, find a polynomial $p(x)$ of lowest possible degree satisfying the **interpolation condition**

$$p(x_i) = y_i, \qquad i = 0, \dots, n. \tag{2.1}$$

The solution $p(x)$ is called the **interpolation polynomial**, the x_i values are called **nodes**, and the points (x_i, y_i) **interpolation points**.

Example 1

Given are the points

x_i	0	2/3	1
y_i	1	1/2	0

.

The corresponding interpolation polynomial is

$$p_2(x) = (-3x^2 - x + 4)/4$$

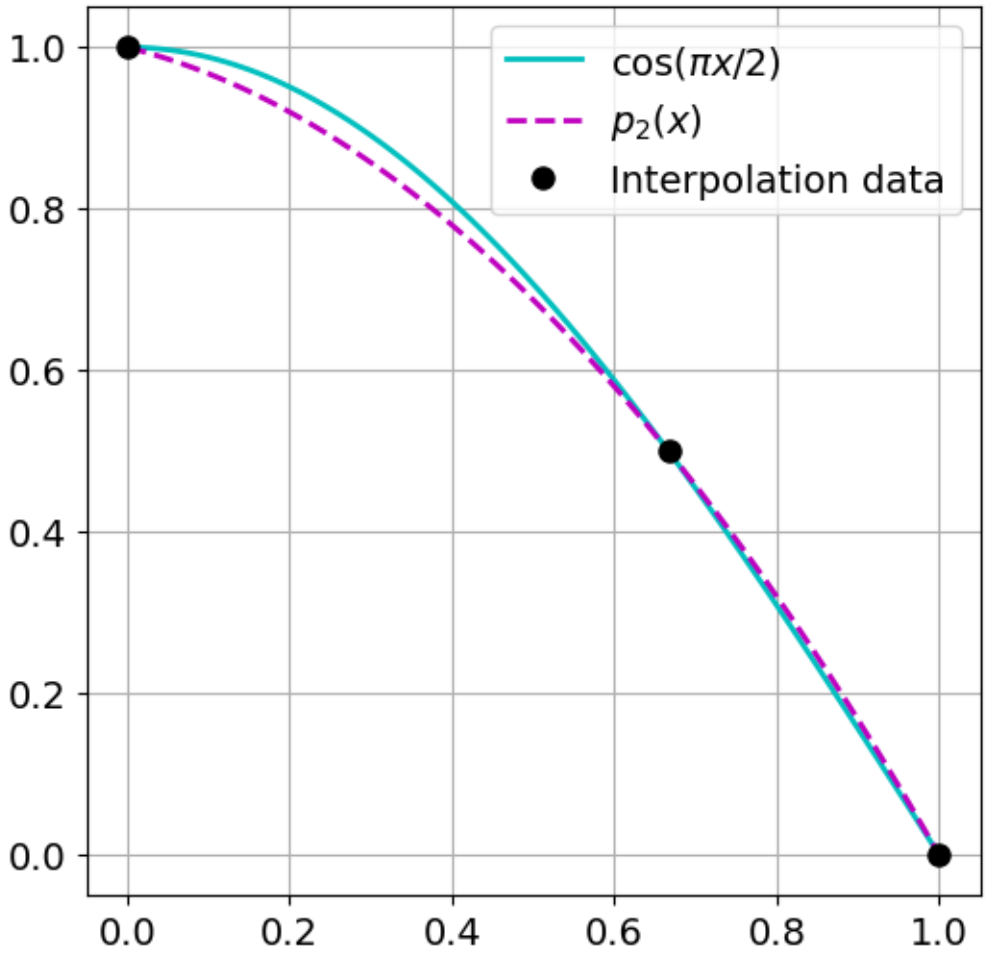
The y-values of this example are chosen such that $y_i = \cos(\pi x_i/2)$. So $p_2(x)$ can be considered as an approximation to $\cos(\pi x/2)$ on the interval $[0, 1]$.

To visualize this, we need to import some modules first, using the following boilerplate code.

```
#%matplotlib widget
import numpy as np
from numpy import pi
from numpy.linalg import solve, norm    # Solve linear systems and compute norms
import matplotlib.pyplot as plt

newparams = {'figure.figsize': (6.0, 6.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 14}
plt.rcParams.update(newparams)
```

```
# Interpolation data
xdata = [0,2/3., 1]
ydata = [1, 1/2., 0]

# Interpolation polynomial
p2 = lambda x : (-3*x**2 -x + 4)/4.

# Grid points for plotting
x = np.linspace(0,1,100)
y = p2(x)

# Original function
f = np.cos(pi*x/2)

plt.figure()
plt.plot(x,f, 'c',x,y,'--m', xdata, ydata, "ok")
plt.legend([r'$\cos(\pi x/2)$', r'$p_2(x)$', 'Interpolation data'])
```

```
<matplotlib.legend.Legend at 0x10e6d6270>
```



Content of this module

In this module, we will discuss the following:

- Method: How to compute the polynomials?
- Existence and uniqueness results.
- Error analysis: If the polynomial is used to approximate a function, how good is the approximation?
- Improvements: If the nodes x_i can be chosen freely, how should we do it in order to reduce the error?

2.2 Preliminaries

Let us start with some useful notation and facts about polynomials.

- A polynomial of degree n is given by

$$p_n(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x_1 + c_0, \qquad c_i \in \mathbb{R}, \quad i = 0, 1, \ldots, n. \tag{2.2}$$

- $\mathbb{P}_n$ is the set of all polynomials of degree n.
- $C^m[a, b]$ is the set of all continuous functions that have continuous first m derivatives.
- The value r is a root or a zero of a polynomial p if $p(r) = 0$.

- A nonzero polynomial of degree n can never have more than n real roots (there may be less).
- A polynomial of degree n with n real roots $r_1, r_2, \dots, r_n$ can be written as

$$p_n(x) = c(x - r_1)(x - r_2)\cdots(x - r_n) = c\prod_{i=1}^{n}(x - r_i).$$

2.3 The direct approach

For a polynomial of degree n the interpolation condition (2.1) is a linear systems of $n + 1$ equations in $n + 1$ unknowns:

$$\sum_{i=0}^{n} x_j^i c_i = y_j, \qquad j = 0, \dots, n.$$

In other words, we try to solve the linear system

$$\underbrace{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix}}_{:=V(x_0,x_1,\dots,x_n)} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}. \tag{2.3}$$

$V(x_0, x_1, \dots, x_n)$ denotes the so-called Vandermonde matrix. It can be shown that

$$\det V(x_0, x_1, \dots, x_n) = \prod_{0 \leqslant i < j \leqslant n} (x_j - x_i)$$

Consequently, $\det V \neq 0$ for n *distinct* nodes $\{x_i\}_{i=0}^n$ and thus (2.3) is uniquely solvable.

If we are basically interested in the polynomials themself, given by the coefficients c_i, $i = 0, 1, \dots, n$, this is a perfectly fine solution. It is for instance the strategy implemented in MATLAB's interpolation routines. However, in this course, polynomial interpolation will be used as a basic tool to construct other algorithms, in particular for integration. In that case, this is not the most convenient option, so we concentrate on a different strategy, which essentially makes it possible to just write up the polynomials.

2.4 Lagrange interpolation

Definition 2

Given $n + 1$ points $(x_i, y_i)_{i=0}^n$ with distinct x_i values. The **cardinal functions** are defined by

$$\ell_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \cdot \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_n}{x_i - x_n}, \qquad i = 0, \dots, n.$$

Observation 1

The cardinal functions have the following properties:

- $\ell_i \in \mathbb{P}_n$, $i = 0, 1, \dots, n$.

- $\ell_i(x_j) = \delta_{ij} = \begin{cases} 1, & \text{when } i = j \\ 0, & \text{when } i \neq j \end{cases}$.
- They are constructed solely from the nodes x_i's.
- They are linearly independent, and thus form a basis for $\mathbb{P}_n$.

Remark 1

The cardinal functions are also often called **Lagrange polynomials**.

The interpolation polynomial is now given by

$$p_n(x) = \sum_{i=0}^{n} y_i \ell_i(x)$$

since

$$p_n(x_j) = \sum_{i=0}^{n} y_i \ell_i(x_j) = y_j, \qquad j = 0, \dots, n.$$

Example 2

Given the points:

x_i	0	1	3
y_i	3	8	6

The corresponding cardinal functions are given by:

$$\ell_0(x) = \frac{(x-1)(x-3)}{(0-1)(0-3)} = \frac{1}{3}x^2 - \frac{4}{3}x + 1$$
$$\ell_1(x) = \frac{(x-0)(x-3)}{(1-0)(1-3)} = -\frac{1}{2}x^2 + \frac{3}{2}x$$
$$\ell_2(x) = \frac{(x-0)(x-1)}{(3-0)(3-1)} = \frac{1}{6}x^2 - \frac{1}{6}x$$

and the interpolation polynomial is given by (check it yourself):

$$p_2(x) = 3\ell_0(x) + 8\ell_1(x) + 6\ell_2(x) = -2x^2 + 7x + 3.$$

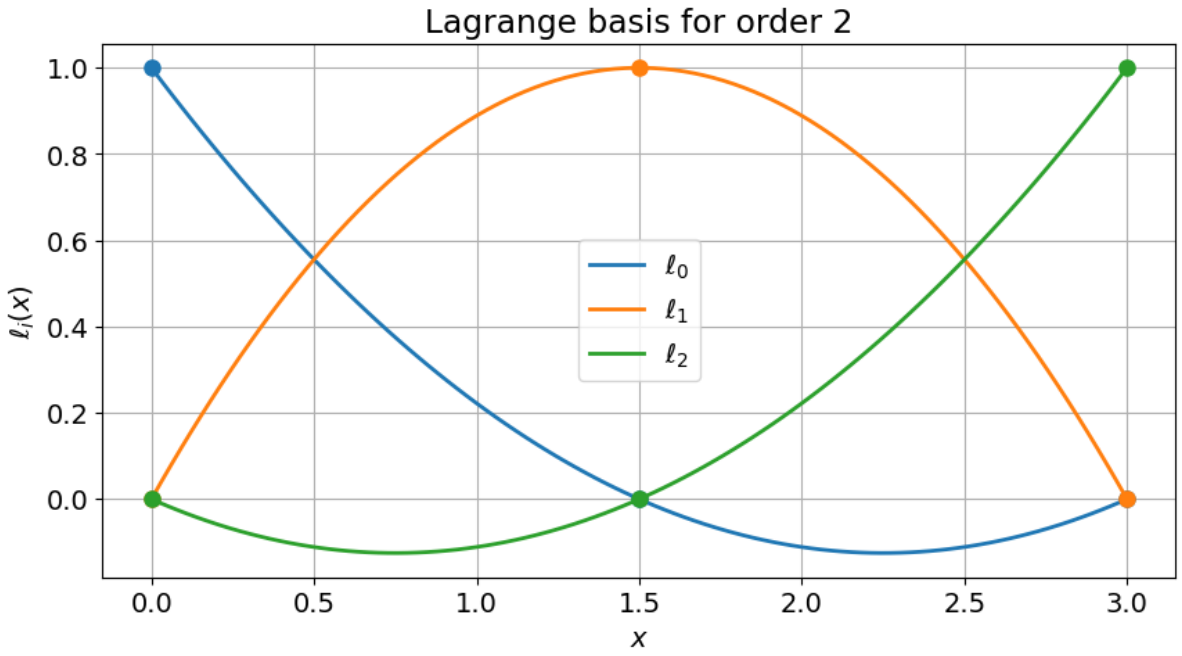
```
import ipywidgets as widgets
from ipywidgets import interact
plt.rcParams['figure.figsize'] = [10, 5]
import scipy.interpolate as ip

def plot_lagrange_basis(a, b, N):
    """ Plot the Lagrange nodal functions for given nodal points."""
    xi = np.linspace(a,b,N)
    N = xi.shape[0]
```

(continues on next page)

(continued from previous page)

```
    nodal_values = np.ma.identity(N)

    # Create finer grid to print resulting functions
    xn = np.linspace(xi[0],xi[-1],100)
    fig = plt.figure()

    for i in range(N):
        L = ip.lagrange(xi, nodal_values[i])
        line, = plt.plot(xn, L(xn), "-", label=(r"$\ell_{%d}$"%i))
        plt.plot(xi, L(xi), "o", color=line.get_color())

    plt.legend()
    plt.title("Lagrange basis for order %d" % (N-1))
    plt.xlabel(r"$x$")
    plt.ylabel(r"$\ell_i(x)$")
    plt.show()
```

```
a, b = 0, 3
N = 3
plot_lagrange_basis(a, b, N)
```




```
# Define a helper function to be connected with the slider
a, b = 0, 3
plp = lambda N : plot_lagrange_basis(a, b, N)
slider = widgets.IntSlider(min = 2,
                           max = 10,
                           step = 1,
                           description="Number of interpolation points N",
                           value = 3)
interact(plp, N=slider)
```

```
interactive(children=(IntSlider(value=3, description='Number of interpolation␣
↪points N', max=10, min=2), Outpu…
```

```
<function __main__.<lambda>(N)>
```

2.5 Implementation

The method above is implemented as two functions:

- `cardinal(xdata, x)`: Create a list of cardinal functions $\ell_i(x)$ evaluated in x.
- `lagrange(ydata, l)`: Create the interpolation polynomial $p_n(x)$.

Here, `xdata` and `ydata` are arrays with the interpolation points, and `x` is an array of values in which the polynomials are evaluated.

You are not required to understand the implementation of these functions, but you should understand how to use them.

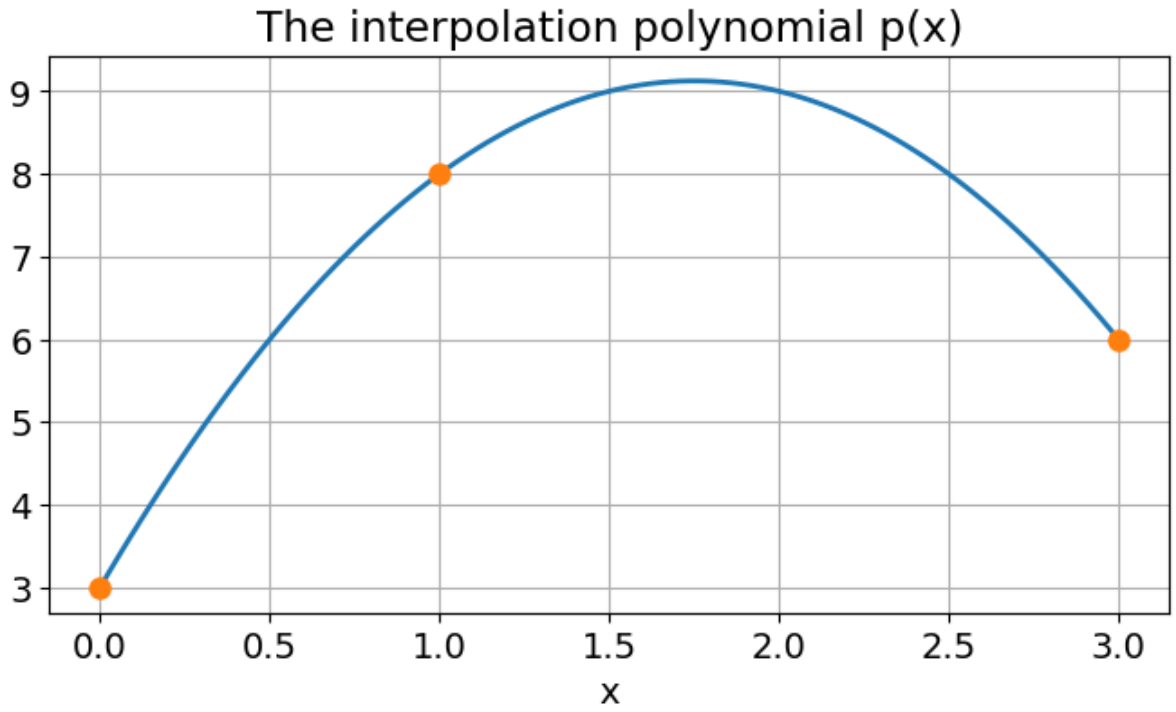
```
from math import factorial
newparams = {'figure.figsize': (8.0, 4.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 14}
plt.rcParams.update(newparams)
```

```
def cardinal(xdata, x):
    """
    cardinal(xdata, x):
    In: xdata, array with the nodes x_i.
        x, array or a scalar of values in which the cardinal functions are evaluated.
    Return: l: a list of arrays of the cardinal functions evaluated in x.
    """
    n = len(xdata)              # Number of evaluation points x
    l = []
    for i in range(n):          # Loop over the cardinal functions
        li = np.ones(len(x))
        for j in range(n):      # Loop to make the product for l_i
            if i is not j:
                li = li*(x-xdata[j])/(xdata[i]-xdata[j])
        l.append(li)            # Append the array to the list
    return l

def lagrange(ydata, l):
    """
    lagrange(ydata, l):
    In: ydata, array of the y-values of the interpolation points.
         l, a list of the cardinal functions, given by cardinal(xdata, x)
    Return: An array with the interpolation polynomial.
    """
    poly = 0
    for i in range(len(ydata)):
        poly = poly + ydata[i]*l[i]
    return poly
```

Exercise 3

1. Let's test the functions on the interpolation points of *Example 2*. and the resulting interpolation polynomial.
2. Redo the exercise for some points of your own choice.

```
# Insert your code here
xdata = [0, 1, 3]           # The interpolation points
ydata = [3, 8, 6]
x = np.linspace(0, 3, 101)     # The x-values in which the polynomial is evaluated
l = cardinal(xdata, x)      # Find the cardinal functions evaluated in x
p = lagrange(ydata, l)      # Compute the polynomial evaluated in x
plt.plot(x, p)                  # Plot the polynomial
plt.plot(xdata, ydata, 'o')     # Plot the interpolation points
plt.title('The interpolation polynomial p(x)')
plt.xlabel('x');
```



2.6 Existence and uniqueness of interpolation polynomials.

We have already proved the existence of such polynomials, simply by constructing them. But are they unique? The answer is yes!

Theorem 1 (Existence and uniqueness of the interpolation polynomial)

Given $n+1$ points $(x_i, y_i)_{i=0}^n$ with $n+1$ distinct x values. Then there is one and only one polynomial $p_n(x) \in \mathbb{P}_n$ satisfying the interpolation condition

$$p_n(x_i) = y_i, \qquad i = 0, \dots, n.$$

Proof. Suppose there exist two different interpolation polynomials p_n and q_n of degree n interpolating the same $n+1$ points. The polynomial $r(x) = p_n(x) - q_n(x)$ is of degree n with zeros in all the nodes x_i, that is a total of $n+1$ zeros. But then $r \equiv 0$, and the two polynomials p_n and q_n are identical.

2.7 Polynomial interpolation: Error theory

We start by executing some boilerplate code. Afterwards we recall the definition of the `python` function `cardinal` and `lagrange` from the previous lecture.

```
# %matplotlib widget

import numpy as np
from numpy import pi
from numpy.linalg import solve, norm    # Solve linear systems and compute norms
import matplotlib.pyplot as plt

newparams = {'figure.figsize': (6.0, 6.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 14}
plt.rcParams.update(newparams)
```

```
def cardinal(xdata, x):
    """
    cardinal(xdata, x):
    In: xdata, array with the nodes x_i.
        x, array or a scalar of values in which the cardinal functions are evaluated.
    Return: l: a list of arrays of the cardinal functions evaluated in x.
    """
    n = len(xdata)              # Number of evaluation points x
    l = []
    for i in range(n):          # Loop over the cardinal functions
        li = np.ones(len(x))
        for j in range(n):      # Loop to make the product for l_i
            if i is not j:
                li = li*(x-xdata[j])/(xdata[i]-xdata[j])
        l.append(li)            # Append the array to the list
    return l

def lagrange(ydata, l):
    """
    lagrange(ydata, l):
    In: ydata, array of the y-values of the interpolation points.
```

(continues on next page)

(continued from previous page)

```
        l, a list of the cardinal functions, given by cardinal(xdata, x)
    Return: An array with the interpolation polynomial.
    """
    poly = 0
    for i in range(len(ydata)):
        poly = poly + ydata[i]*l[i]
    return poly
```

2.7.1 Error Theory

Given some function $f \in C[a,b]$. Choose $n+1$ distinct nodes in $[a,b]$ and let $p_n(x) \in \mathbb{P}_n$ satisfy the interpolation condition

$$p_n(x_i) = f(x_i), \qquad i = 0, \dots, n.$$

What can be said about the error $e(x) = f(x) - p_n(x)$?

The goal of this section is to cover a few theoretical aspects, and to give the answer to the natural question:

- If the polynomial is used to approximate a function, can we find an expression for the error?
- How can the error be made as small as possible?

Let us start with an numerical experiment, to have a certain feeling of what to expect.

Example 3 (Interpolation of $\sin x$)

Let $f(x) = \sin(x)$, $x \in [0, 2\pi]$. Choose $n+1$ equidistributed nodes, that is $x_i = ih$, $i = 0, \dots, n$, and $h = 2\pi/n$.

Calculate the interpolation polynomial using the functions `cardinal` and `lagrange`. Plot the error $e_n(x) = f(x) - p_n(x)$ for different values of n. Choose $n = 4, 8, 16$ and 32. Notice how the error is distributed over the interval, and find the maximum error $\max_{x\in[a,b]} |e_n(x)|$ for each n.

```
# Define the function
def f(x):
    return np.sin(x)


# Set the interval
a, b = -5, 5                    # The interpolation interval
x = np.linspace(a, b, 101)      # The 'x-axis'

# Set the interpolation points
n = 6                           # Interpolation points
xdata = np.linspace(a, b, n+1)  # Equidistributed nodes (can be changed)
ydata = f(xdata)

# Evaluate the interpolation polynomial in the x-values
l = cardinal(xdata, x)
p = lagrange(ydata, l)
```

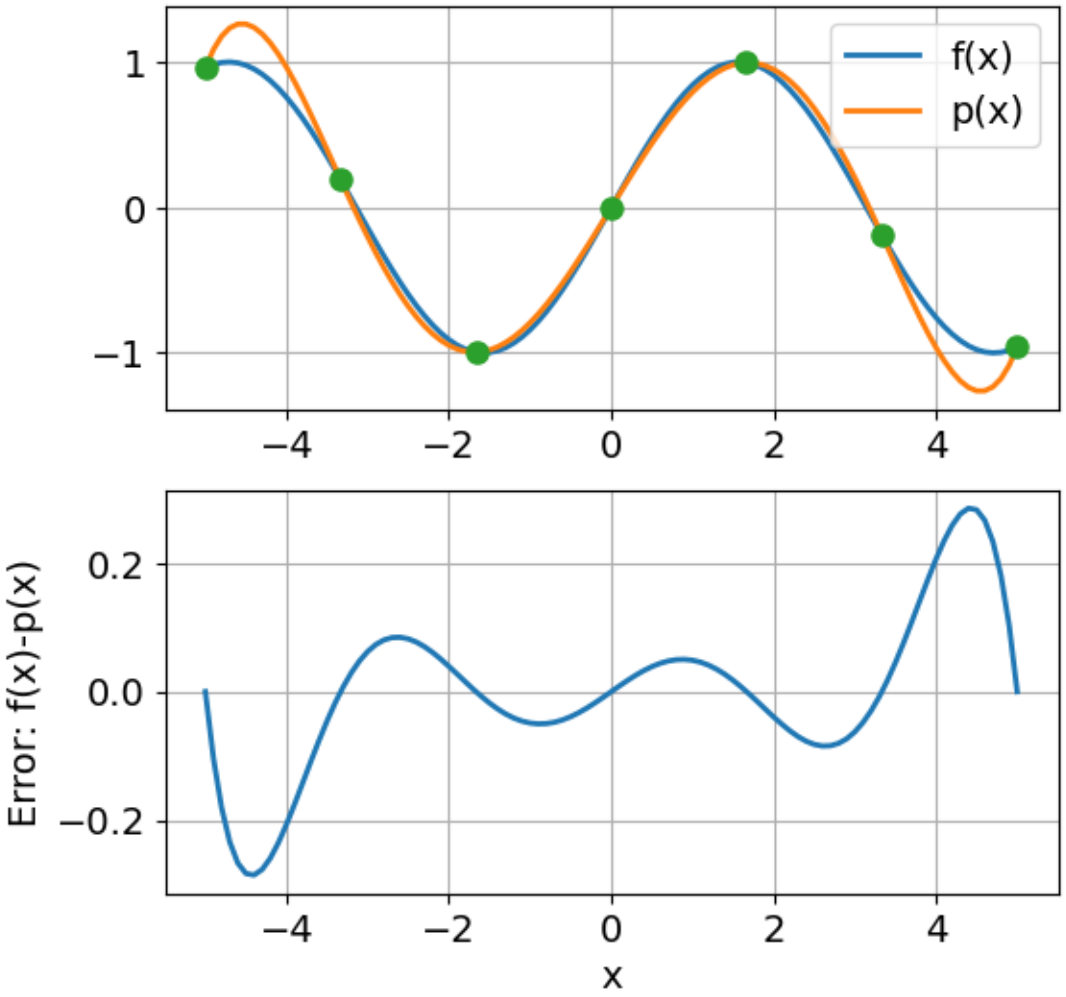
```
# Plot f(x) og p(x) and the interpolation points
plt.figure()
plt.subplot(2,1,1)
```

(continues on next page)

(continued from previous page)

```
plt.plot(x, f(x), x, p, xdata, ydata, 'o')
plt.legend(['f(x)','p(x)'])
plt.grid(True)

# Plot the interpolation error
plt.subplot(2,1,2)
plt.plot(x, (f(x)-p))
plt.xlabel('x')
plt.ylabel('Error: f(x)-p(x)')
plt.grid(True)
print("Max error is {:.2e}".format(max(abs(p-f(x)))))
```

```
Max error is 2.86e-01
```



Exercise 4 (Interpolation of $\frac{1}{1+x^2}$)

Repeat the previous experiment with Runge's function

$$f(x) = \frac{1}{1+x^2}, \qquad x \in [-5, 5].$$

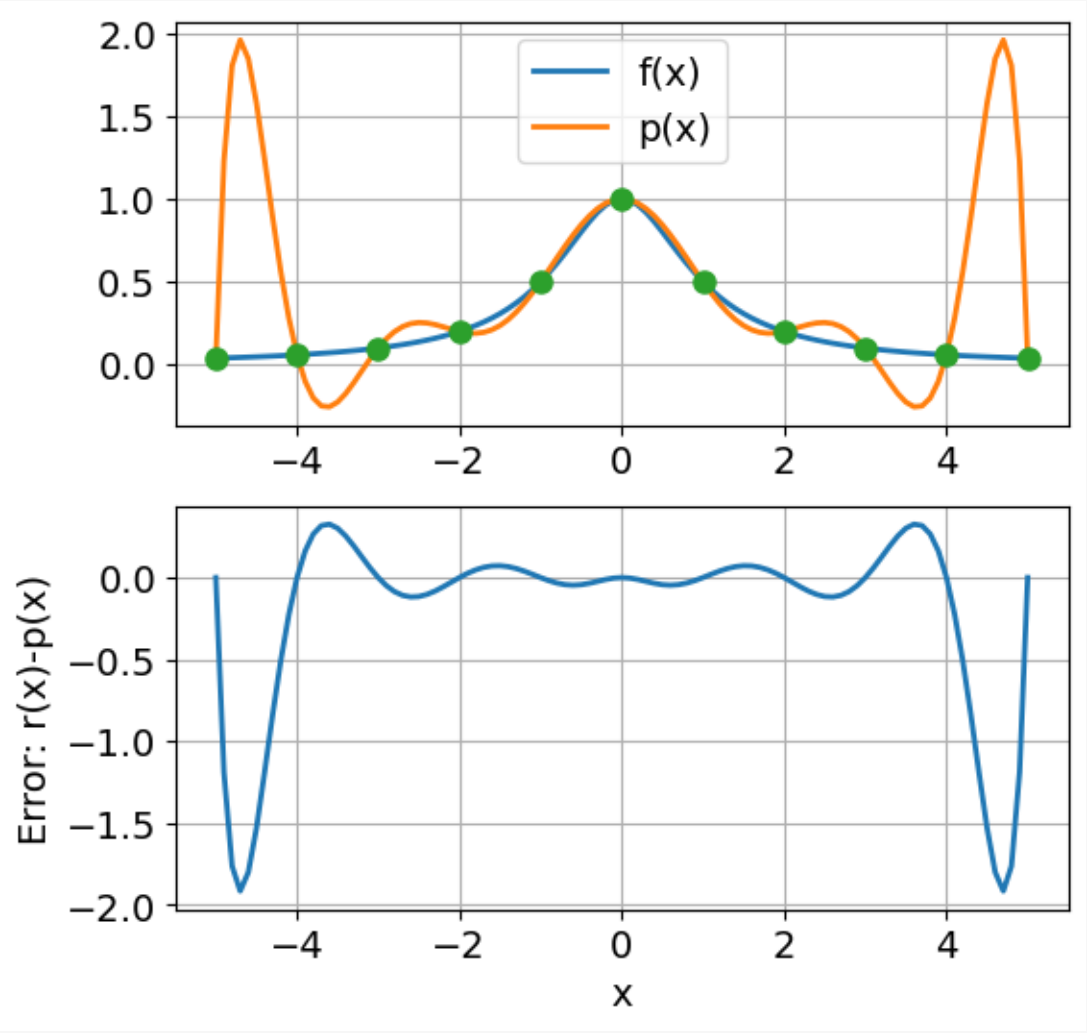
```
# Insert your code here
```

Solution to Exercise 4 (Interpolation of $\frac{1}{1+x^2}$)

```
# Define the function
def r(x):
    return 1/(1+x**2)

# Set the interval
a, b = -5, 5                    # The interpolation interval
x = np.linspace(a, b, 101)      # The 'x-axis'

# Set the interpolation points
n = 10                            # Interpolation points
xdata = np.linspace(a, b, n+1)   # Equidistributed nodes (can be changed)
ydata = r(xdata)

# Evaluate the interpolation polynomial in the x-values
l = cardinal(xdata, x)
p = lagrange(ydata, l)
```

```
# Plot rx) og p(x) and the interpolation points
plt.figure()
plt.subplot(2,1,1)
plt.plot(x, r(x), x, p, xdata, ydata, 'o')
plt.legend(['f(x)','p(x)'])
plt.grid(True)

# Plot the interpolation error
plt.subplot(2,1,2)
plt.plot(x, (r(x)-p))
plt.xlabel('x')
plt.ylabel('Error: r(x)-p(x)')
plt.grid(True)
print("Max error is {:.2e}".format(max(abs(p-r(x)))))
```

```
Max error is 1.92e+00
```



Observation 2

We see that approximation of Runge's functions is much worse then for the $\sin(x)$ function and is not uniformly bounded. In fact, it seems that the maximum error does not decrease with an increasing number of (uniformly distributed!) interpolation nodes, but the large errors are squeezed more and more towards to interval endpoints.

Taylor polynomials once more. Before we turn to the analysis of the interpolation error $e(x) = f(x) - p_n(x)$, we quickly recall (once more) Taylor polynomials and their error representation. For $f \in C^{n+1}[a,b]$ and $x_0 \in (a,b)$, we defined the n-th order Taylor polynomial $T^n_{x_0} f(x)$ of f around x_0 by

$$T^n_{x_0} f(x) := \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

Note that the Taylor polynomial is in fact a polynomial of order n which not only interpolates f in x_0, but also its first, second etc. and n-th derivative $f', f'', \ldots f^{(n)}$ in x_0!

So the Taylor polynomial the unique polynomial of order n which interpolates the *first* n *derivatives* of f in a *single point* x_0. In contrast, the interpolation polynomial p_n is the unique polynomial of order n which *interpolates only the* 0*-order* (that is, f itself), but in n *distinctive points* $x_0, x_1, \ldots x_n$.

For the Taylor polynomial $T_{x_0}^n f(x)$ we have the error representation

$$f(x) - T_{x_0}^n f(x) = R_{n+1}(x_0) \qquad \text{where } R_{n+1}(x_0) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1},$$

with ξ between x and x_0.

Of course, we usually don't know the exact location of ξ and thus not the exact error, but we can at least estimate it and bound it from above:

$$|f(x) - T_{x_0}^n f(x)| \leqslant \frac{M}{(n+1)!} h^{n+1}$$

where

$$M = \max_{x \in [a,b]} |f^{(n+1)}(x)| \qquad \text{and} \qquad h = |x - x_0|.$$

The next theorem gives us an expression for the interpolation error $e(x) = f(x) - p_n(x)$ which is similar to what we have just seen for the error between the Taylor polynomial and the original function f.

Theorem 2 (Interpolation error)

Given $f \in C^{(n+1)}[a, b]$. Let $p_n \in \mathbb{P}_n$ interpolate f in $n + 1$ distinct nodes $x_i \in [a, b]$. For each $x \in [a, b]$ there is at least one $\xi(x) \in (a, b)$ such that

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i).$$

Proof.

We start fromt the Newton polynomial $\omega_{n+1} =: \omega(x)$

$$\omega(x) = \prod_{i=0}^{n} (x - x_i) = x^{n+1} + \cdots .$$

Clearly, the error in the nodes, $e(x_i) = 0$. Choose an *arbitrary* $x \in [a, b]$, $x \in [a, b]$, where $x \neq x_i$, $i = 0, 1, \ldots, n$. For this fixed x, define a function in t as:

$$\varphi(t) = e(t)\omega(x) - e(x)\omega(t).$$

where $e(t) = f(t) - p_n(t)$.

Notice that $\varphi(t)$ is as differentiable with respect to t as $f(t)$. The function $\varphi(t)$ has $n + 2$ distinct zeros (the nodes and the fixed x). As a consequence of Rolle's theorem, the derivative $\varphi'(t)$ has at least $n + 1$ distinct zeros, one between each of the zeros of $\varphi(t)$. So $\varphi''(t)$ has n distinct zeros, etc. By repeating this argument, we can see that $\varphi^{n+1}(t)$ has at least one zero in $[a, b]$, let us call this $\xi(x)$, as it does depend on the fixed x.

Since $\omega^{(n+1)}(t) = (n + 1)!$ and $e^{(n+1)}(t) = f^{(n+1)}(t)$ we obtain

$$\varphi^{(n+1)}(\xi) = 0 = f^{(n+1)}(\xi)\omega(x) - e(x)(n + 1)!$$

which concludes the proof.

Observation 3

The interpolation error consists of three elements: The derivative of the function f, the number of interpolation points $n + 1$ and the distribution of the nodes x_i. We cannot do much with the first of these, but we can choose the two others. Let us first look at the most obvious choice of nodes.

2.7.2 Equidistributed nodes

The nodes are *equidistributed* over the interval $[a, b]$ if $x_i = a + ih$, $h = (b - a)/n$, $i = 0, \ldots, n$ In this case it can be proved that:

$$|\omega(x)| \leq \frac{h^{n+1}}{4} n!$$

such that

$$|e(x)| \leq \frac{h^{n+1}}{4(n+1)} M, \qquad M = \max_{x \in [a,b]} |f^{(n+1)}(x)|.$$

for all $x \in [a, b]$.

Let us now see how good this error bound is by an example.

Exercise 5 (Interpolation error for $\sin(x)$ revisited)

Let again $f(x) = \sin(x)$ and $p_n(x)$ the polynomial interpolating $f(x)$ in $n + 1$ equidistributed points on $[a, b] = [0, 2\pi]$. An upper bound for the error for different values of n can be found easily. Clearly, $\max_{x \in [0,2\pi]} |f^{(n+1)}(x)| = M = 1$ for all n, so

$$|e_n(x)| = |f(x) - p_n(x)| \leq \frac{1}{4(n+1)} \left(\frac{2\pi}{n}\right)^{n+1}, \qquad x \in [a, b].$$

Use the code in the first Example of this lecture to verify the result for $n = 2, 4, 8, 16$. How close is the bound to the real error?

```
# Insert your code here
```

2.7.3 Optimal choice of interpolation points

So how can the error be reduced? For a given n there is only one choice: to distribute the nodes in order to make the maximum of $|\omega(x)| = \prod_{j=0}^{n} |x - x_i|$ as small as possible. We will first do this on a standard interval $[-1, 1]$, and then transfer the results to some arbitrary interval $[a, b]$.

Let us start taking a look at $\omega(x)$ for equidistributed nodes on the interval $[-1, 1]$, for different values of n:

```
newparams = {'figure.figsize': (6,3)}
plt.rcParams.update(newparams)

def omega(xdata, x):
    # compute omega(x) for the nodes in xdata
    n1 = len(xdata)
    omega_value = np.ones(len(x))
    for j in range(n1):
        omega_value = omega_value*(x-xdata[j])   # (x-x_0)(x-x_1)...(x-x_n)
    return omega_value
```

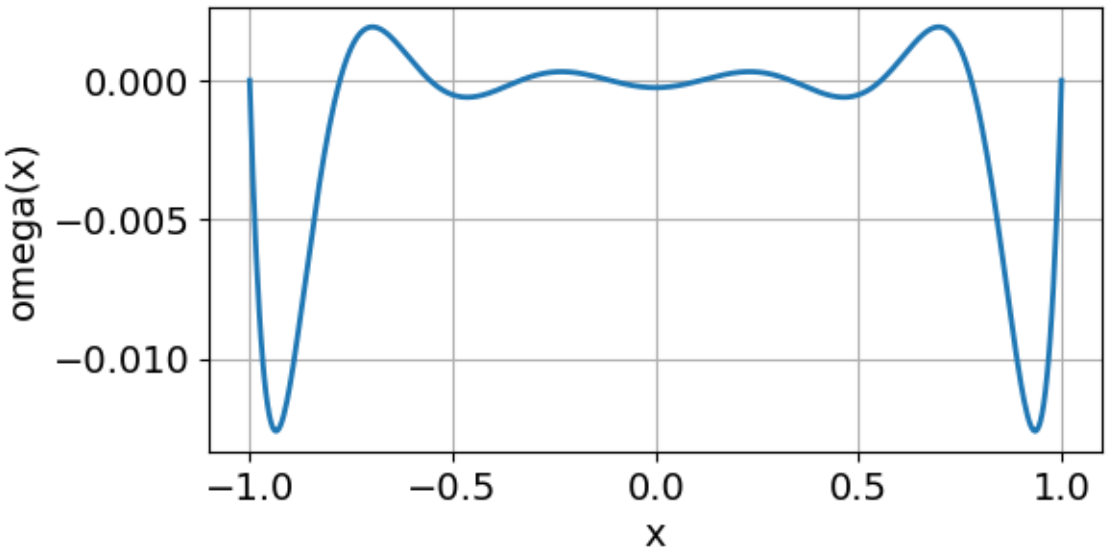
```
# Plot omega(x)
n = 10                              # Number of interpolation points is n+1
a, b = -1, 1                       # The interval
x = np.linspace(a, b, 501)
xdata = np.linspace(a, b, n)
```

(continues on next page)

(continued from previous page)

```
plt.plot(x, omega(xdata, x))
plt.grid(True)
plt.xlabel('x')
plt.ylabel('omega(x)')
print("n = {:2d}, max|omega(x)| = {:.2e}".format(n, max(abs(omega(xdata, x)))))
```

```
n = 10, max|omega(x)| = 1.26e-02
```

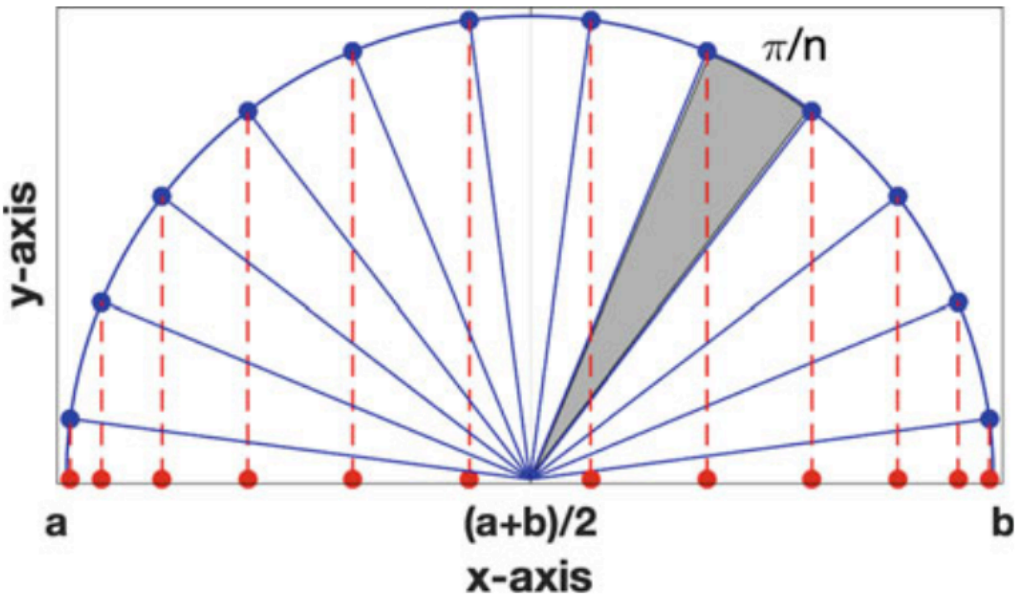


Run the code for different values of n. Notice the following:

- $\max_{x\in[-1,1]} |\omega(x)|$ becomes smaller with increasing n.
- $|\omega(x)|$ has its maximum values near the boundaries of $[-1, 1]$.

A a consequence of the latter, it seems reasonable to move the nodes towards the boundaries. It can be proved that the optimal choice of nodes are the *Chebyshev-nodes*, given by

$$\tilde{x}_i = \cos\left(\frac{(2i+1)\pi}{2(n+1)}\right), \qquad i = 0, \dots, n$$



Chebyshev nodes. Figure taken from [Holmes, 2023], p.233.

Let $\omega_{Cheb}(x) = \prod_{j=1}^{n}(x - \tilde{x}_i)$. It is then possible to prove that

$$\frac{1}{2^n} = \max_{x\in[-1,1]} |\omega_{Cheb}(x)| \leq \max_{x\in[-1,1]} |q(x)|$$

for all polynomials $q \in \mathbb{P}_n$ such that $q(x) = x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$.

The distribution of nodes can be transferred to an interval $[a, b]$ by the linear transformation

$$x = \frac{b-a}{2}\tilde{x} + \frac{b+a}{2}$$

where $x \in [a, b]$ and $\tilde{x} \in [-1, 1]$.

By doing so we get

$$\omega(x) = \prod_{j=0}^{n}(x - x_i) = \left(\frac{b-a}{2}\right)^{n+1} \prod_{j=0}^{n}(\tilde{x} - \tilde{x}_i) = \left(\frac{b-a}{2}\right)^{n+1} \omega_{Cheb}(\tilde{x}).$$

From the theorem on interpolation errors we can conclude:

Theorem 3 (Interpolation error for Chebyshev interpolation)

Given $f \in C^{(n+1)}[a, b]$, and let $M_{n+1} = \max_{x\in[a,b]} |f^{(n+1)}(x)|$. Let $p_n \in \mathbb{P}_n$ interpolate f i $n+1$ Chebyshev-nodes $x_i \in [a, b]$. Then

$$\max_{x\in[a,b]} |f(x) - p_n(x)| \leq \frac{(b-a)^{n+1}}{2^{2n+1}(n+1)!} M_{n+1}.$$

The Chebyshev nodes over an interval $[a, b]$ are evaluated in the following function:

```
def chebyshev_nodes(a, b, n):
    # n Chebyshev nodes in the interval [a, b]
    i = np.array(range(n))                  # i = [0,1,2,3, ....n-1]
    x = np.cos((2*i+1)*pi/(2*(n)))          # nodes over the interval [-1,1]
    return 0.5*(b-a)*x+0.5*(b+a)          # nodes over the interval [a,b]
```

Exercise 6 (Chebyshev interpolation)

a) Plot $\omega_{Cheb}(x)$ for $3, 5, 9, 17$ interpolation points on the interval $[-1, 1]$.

b) Repeat Example 3 using Chebyshev interpolation on the functions below. Compare with the results you got from equidistributed nodes.

$$f(x) = \sin(x), \qquad x \in [0, 2\pi]$$
$$f(x) = \frac{1}{1+x^2}, \qquad x \in [-5, 5].$$

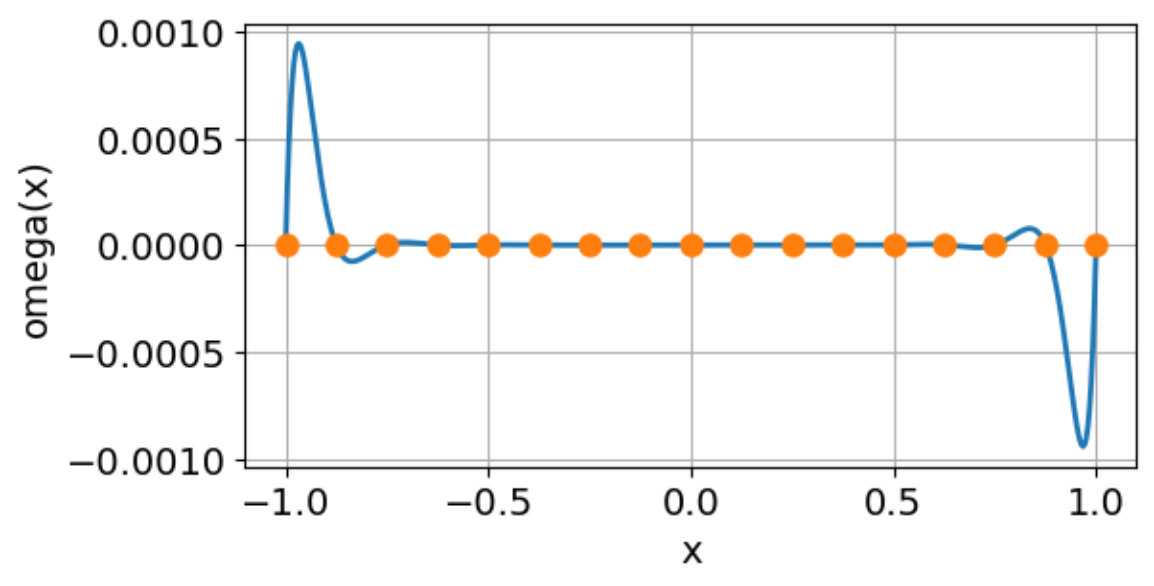
Solution to Exercise 6 (Chebyshev interpolation)

a) Let's plot $\omega(x)$ first for n equidistributed nodes and then $\omega_{Cheb}(x)$ for $5, 9, 17, 25$ interpolation points on the interval $[-1, 1]$.

```
# Insert your code here
# Define number of interpolation points
n = 17
#
a, b = -1, 1                    # The interval
x = np.linspace(a, b, 501)
```

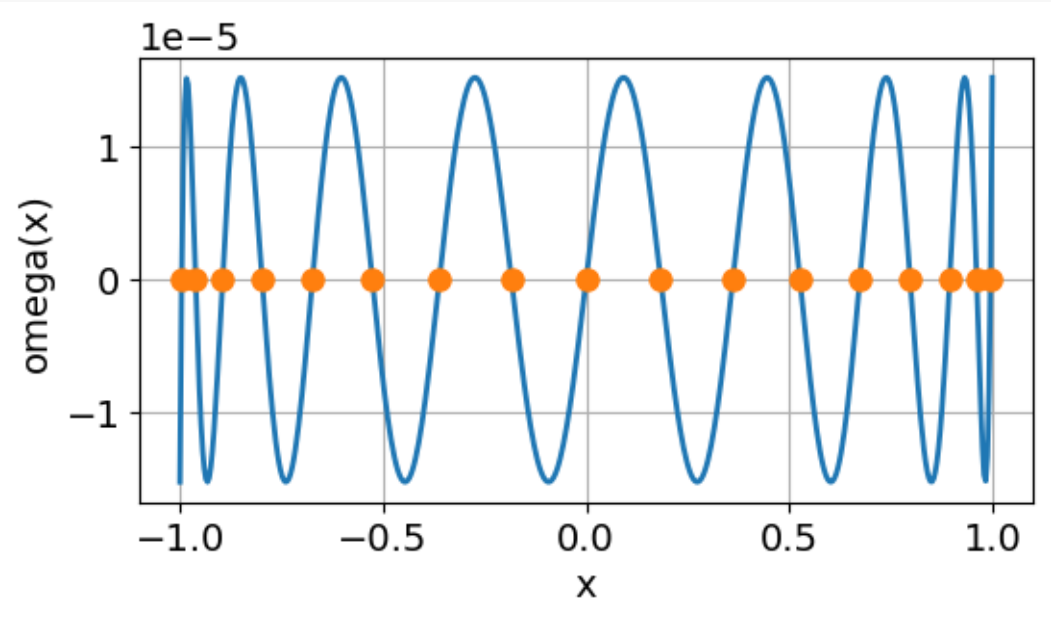
```
# equidistributes nodes
xdata = np.linspace(a, b, n)

plt.plot(x, omega(xdata, x))
plt.plot(xdata,omega(xdata, xdata), "o")
plt.grid(True)
plt.xlabel('x')
plt.ylabel('omega(x)')
print("n = {:2d}, max|omega(x)| = {:.2e}".format(n, max(abs(omega(xdata, x)))))
```

```
n = 17, max|omega(x)| = 9.43e-04
```




```
# Chebyshev nodes
xdata = chebyshev_nodes(a, b, n)

plt.plot(x, omega(xdata, x))
plt.plot(xdata,omega(xdata, xdata), "o")
plt.grid(True)
plt.xlabel('x')
plt.ylabel('omega(x)')
print("n = {:2d}, max|omega(x)| = {:.2e}".format(n, max(abs(omega(xdata, x)))))
```

```
n = 17, max|omega(x)| = 1.53e-05
```



b) Let's interpolate the following functions

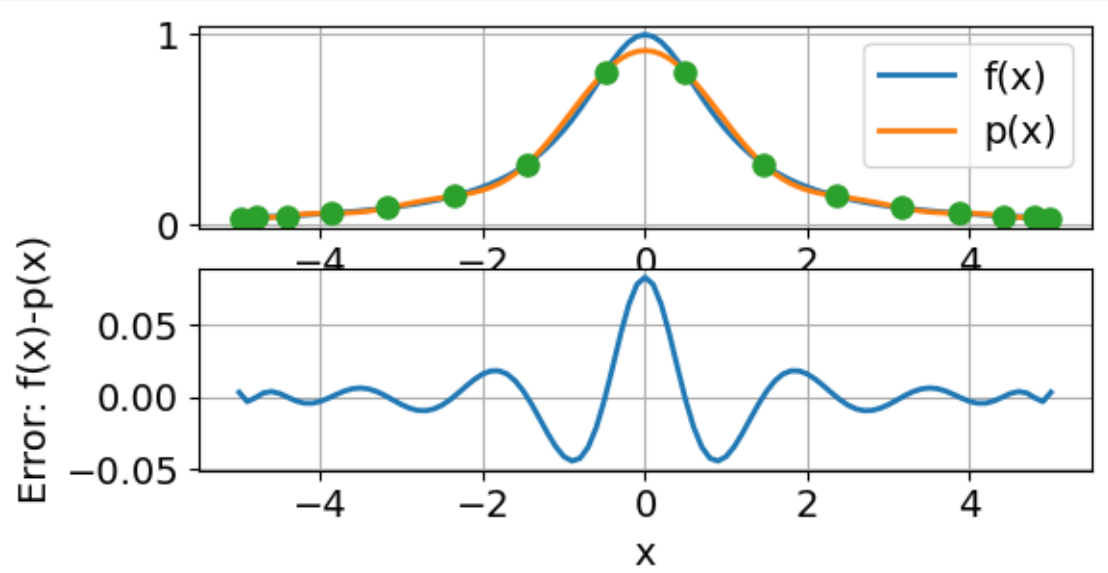
$$f(x) = \sin(x), \qquad x \in [0, 2\pi]$$
$$f(x) = \frac{1}{1+x^2}, \qquad x \in [-5, 5].$$

using Chebyshev interpolation nodes.

```
# Define the function
def f(x):
    return 1/(1+x**2)

# Set the interval
a, b = -5, 5                  # The interpolation interval
#a, b = 0, 2*pi               # The interpolation interval
x = np.linspace(a, b, 101) # The 'x-axis'

# Set the interpolation points
n = 16                         # Interpolation points
#xdata = np.linspace(a, b, n) # Equidistributed nodes (can be changed)
xdata = chebyshev_nodes(a, b, n)
ydata = f(xdata)

# Evaluate the interpolation polynomial in the x-values
l = cardinal(xdata, x)
p = lagrange(ydata, l)
```

```
# Plot f(x) og p(x) and the interpolation points
plt.subplot(2,1,1)
plt.plot(x, f(x), x, p, xdata, ydata, 'o')
plt.legend(['f(x)','p(x)'])
plt.grid(True)

# Plot the interpolation error
plt.subplot(2,1,2)
plt.plot(x, (f(x)-p))
plt.xlabel('x')
plt.ylabel('Error: f(x)-p(x)')
plt.grid(True)
print("Max error is {:.2e}".format(max(abs(p-f(x)))))
```

```
Max error is 8.31e-02
```



For information: Chebfun is software package which makes it possible to manipulate functions and to solve equations with accuracy close to machine accuracy. The algorithms are based on polynomial interpolation in Chebyshev nodes.

TODO

Add ipywidgets slider for better visualization/interactivity.

2.8 Summary

This chapter introduces the theory and practice of polynomial interpolation, a foundational tool in scientific computing with applications in approximation, numerical integration, and differential equations. The goal is to approximate a function $f(x)$ using a polynomial that passes through a given set of data points (x_i, y_i). The chapter progresses from basic definitions to error analysis and optimal node selection.

 Section 2.1 – Introduction and Applications

- Motivation: estimating unknown values from discrete data (e.g., historical CO_2 levels).
- Interpolation problem: find a polynomial $p(x) \in \mathbb{P}_n$ such that $p(x_i) = y_i$.

 Section 2.2 – Preliminaries

- Definitions: polynomial degree, zero/root, polynomial spaces $\mathbb{P}_n$, and their properties.
- Review of the algebraic structure of polynomials.

 Section 2.3 – Direct Approach: Vandermonde System

- Constructing the interpolation polynomial by solving a linear system with a Vandermonde matrix.
- Uniqueness and solvability when nodes x_i are distinct.

 Section 2.4 – Lagrange Interpolation

- Definition and construction of Lagrange basis polynomials $\ell_i(x)$.

- Expression of $p(x) = \sum y_i \ell_i(x)$.
- Advantages: no need to solve a system, polynomials are explicit.

 Section 2.5 – Implementation

- Python code for computing cardinal functions and interpolation polynomials.
- Visualization of Lagrange basis and interpolated curves.

 Section 2.6 – Existence and Uniqueness

- The interpolation polynomial exists and is unique for n+1 distinct nodes.
- Proof via contradiction using the root structure of polynomials.

 Section 2.7 – Error Theory

- Defines error $e(x) = f(x) - p_n(x)$.
- Derivation of the interpolation error formula using the Newton form and Rolle's Theorem.
- Error depends on smoothness of f, number of nodes, and distribution of nodes
- Uniformly spaced nodes can lead to large interpolation errors near the boundaries (Runge's phenomenon).
- Optimal distribution of nodes minimizes the maximum of $|\omega(x)| = \prod(x - x_i)$.
- Chebyshev nodes reduce error and avoid Runge's phenomenon.
- Derivation of error bound for interpolation with Chebyshev nodes.

 Learning Outcomes for Chapter 2

By the end of this chapter, students will be able to:

 Theory and Definitions

- Define the interpolation problem and identify when a unique solution exists.
- Explain the role of the interpolation polynomial and nodes.
- Distinguish between different polynomial bases (monomial, Lagrange).

✍ Construction Methods

- Construct interpolation polynomials using:
  - Direct method via Vandermonde matrices
  - Lagrange interpolation using cardinal functions
- Implement polynomial interpolation numerically and visualize results.

 Error Analysis

- Derive and interpret the interpolation error formula.
- Analyze how the error depends on:
  - The degree of the interpolating polynomial
  - The function's smoothness
  - The distribution of interpolation nodes

 Experiments and Applications

- Perform numerical experiments interpolating function with different node distributions.
- Demonstrate how Runge's phenomenon arises from equidistant nodes.

- Estimate the maximum interpolation error for various choices of n and node distributions.

 Chebyshev Interpolation

- Compute and use Chebyshev nodes to reduce interpolation error.
- Compare error behavior of equidistributed and Chebyshev node interpolation.
- Derive and apply error bounds for Chebyshev interpolation.

CHAPTER

THREE

NUMERICAL INTEGRATION: INTERPOLATORY QUADRATURE RULES

As usual we start by importing the some standard boilerplate code.

```
%matplotlib inline

import numpy as np
from numpy import pi
from math import sqrt
from numpy.linalg import solve, norm    # Solve linear systems and compute norms
import matplotlib.pyplot as plt
import matplotlib.cm as cm

newparams = {'figure.figsize': (10.0, 10.0),
             'axes.grid': True,
             'lines.markersize': 8,
             'lines.linewidth': 2,
             'font.size': 14}
plt.rcParams.update(newparams)
```

3.1 Introduction

Imagine you want to compute the finite integral

$$I[f](a,b) = \int_a^b f(x)\,\mathrm{d}x.$$

The "usual" way is to find a primitive function F (also known as the indefinite integral of f) satisfying $F'(x) = f(x)$ and then to compute

$$\int_a^b f(x)\,\mathrm{d}x = F(b) - F(a).$$

While there are many analytical integration techniques and extensive tables to determine definite integral for many integrands, more often than not it may not feasible or possible to compute the integral. For instance, what about

$$f(x) = \frac{\log(2+\sin(1/2-\sqrt(x))^6)}{\log(\pi+\arctan(\sqrt{1-\exp(-2x-\sin(x))})}?$$

Finding the corresponding primitive is highly likely a hopeless endeavor. And sometimes there even innocent looking functions like e^{-x^2} for which there is not primitive functions which can expressed as a composition of standard functions such as $\sin, \cos$. etc.

A **numerical quadrature** or a **quadrature rule** is a formula for approximating such definite integrals $I[f](a,b)$. Quadrature rules are usually of the form

$$Q[f](a,b) = \sum_{i=0}^{n} w_i f(x_i),$$

where x_i, w_i for $i = 0, 1, \dots, n$ are respectively the *nodes/points* and the *weights* of the quadrature rule.

To emphasize that a quadrature rule is defined by some given quadrature points $\{x_i\}_{i=0}^n$ and weights $\{w_i\}_{i=0}^n$, we sometimes might write

$$Q[f](\{x_i\}_{i=0}^n, \{w_i\}_{i=0}^n) = \sum_{i=0}^{n} w_i f(x_i).$$

If the function f is given from the context, we will for simplicity denote the integral and the quadrature simply as $I(a,b)$ and $Q(a,b)$.

Example 4 (Quadrature rules from previous math courses)

The trapezoidal rule, the midpoint rule and Simpson's rule known from previous courses are all examples of numerical quadratures, and we quickly review them here, in addition to the very simple (and less accurate) left and right endpoint rules.

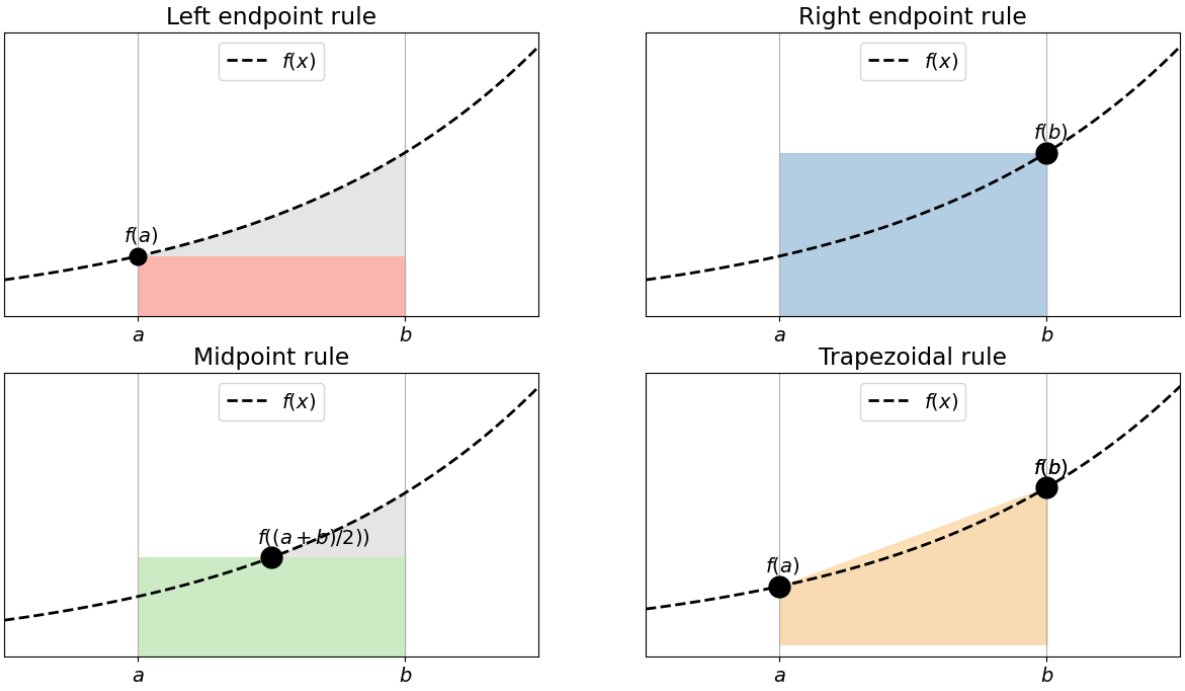
```
colors = plt.get_cmap("Pastel1").colors

def plot_qr_examples():
    f = lambda x : np.exp(x)
    fig, axs = plt.subplots(2,2)
    fig.set_figheight(8)
    # fig.set_figwidth(8)
    fig.set_figwidth(fig.get_size_inches()[0]*1.5)
    #axs[0].add_axes([0.1, 0.2, 0.8, 0.7])
    a, b = -0.5,0.5
    l, r = -1.0, 1.0
    x_a = np.linspace(a, b, 100)

    for raxs in axs:
        for ax in raxs:
            ax.set_xlim(l, r)
            x = np.linspace(l, r, 100)
            ax.plot(x, f(x), "k--", label="$f(x)$")
            ax.fill_between(x_a, f(x_a), alpha=0.1, color='k')
            ax.xaxis.set_ticks_position('bottom')
            ax.set_xticks([a,b])
            ax.set_xticklabels(["$a$", "$b$"])
            ax.set_yticks([])
            ax.legend(loc="upper center")

    # Left endpoint rule
    axs[0,0].bar(a, f(a), b-a, align='edge', color=colors[0])
    axs[0,0].plot(a,f(a), 'ko', markersize="12")
    axs[0,0].set_title("Left endpoint rule")
    axs[0,0].annotate('$f(a)$',
             xy=(a, f(a)), xytext=(-10, 10),
             textcoords="offset points")
```

(continues on next page)

(continued from previous page)

```
    # Right endpoint rule
    axs[0,1].bar(a, f(b), b-a, align='edge',  color=colors[1])
    axs[0,1].plot(b,f(b), 'ko', markersize="15")
    axs[0,1].set_title("Right endpoint rule")
    axs[0,1].annotate('$f(b)$',
             xy=(b, f(b)), xytext=(-10, 10),
             textcoords="offset points")
    # Midpoint rule
    axs[1,0].bar(a, f((a+b)/2), b-a, align='edge', color=colors[2])
    axs[1,0].plot((a+b)/2,f((a+b)/2), 'ko', markersize="15")
    axs[1,0].set_title("Midpoint rule")
    axs[1,0].annotate('$f((a+b)/2))$',
             xy=((a+b)/2, f((a+b)/2)), xytext=(-10, 10),
             textcoords="offset points")

    # Trapezoidal rule
    axs[1,1].set_title("Trapezoidal rule")
    axs[1,1].fill_between([a,b], [f(a), f(b)], alpha=0.8, color=colors[4])
    axs[1,1].plot([a,b],f([a,b]), 'ko', markersize="15")
    axs[1,1].annotate('$f(a)$',
             xy=(a, f(a)), xytext=(-10, 10),
             textcoords="offset points")
    axs[1,1].annotate('$f(b)$',
             xy=(b, f(b)), xytext=(-10, 10),
             textcoords="offset points")
    axs[1,1].annotate('$f(b)$',
             xy=(b, f(b)), xytext=(-10, 10),
             textcoords="offset points")
```

```
plot_qr_examples()
```



- **Left and right endpoint rule** are among the simplest possible quadrature rule defined by

$$QL[f](a,b) := f(a)(b-a) \quad \text{and} \quad QR[f](a,b) := f(b)(b-a)$$

respectively. The (single) quadrature point for $QL[\cdot]$ and $QR[\cdot]$ is given by $x_0 = a$ and $x_0 = b$ respectively, and both use the corresponding weight $w_0 = b - a$.

- **Midpoint rule** is the quadrature rule defined by

$$Q[f](a,b) := (b-a) f\left(\frac{a+b}{2}\right).$$

The node is given by the midpoint, $x_0 = \frac{a+b}{2}$ with the corresponding weight $w_0 = b - a$.

$$Q[f](a,b) = w_0 f(x_0)$$

- **Trapezoidal rule** is given by

$$Q[f](a,b) := (b-a)\left(\frac{f(a)+f(b)}{2}\right)$$

and thus the nodes are defined by $x_0 = a$, $x_1 = b$ with corresponding weights $w_0 = w_1 = \frac{b-a}{2}$.

- Finally, **Simpson's rule** which you know from Matte 1, is defined as follows:

$$Q[f](a,b) = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right),$$

which we identify as quadrature rule with 3 points $x_0 = a, x_1 = \frac{a+b}{2}, x_2 = b$ and corresponding weights $w_0 = w_2 = \frac{b-a}{6}$ and $w_1 = \frac{4(b-a)}{6}$.

In this note we will see how quadrature rules can be constructed from integration of interpolation polynomials. We will demonstrate how to do error analysis and how to find error estimates.

3.2 Quadrature based on polynomial interpolation.

This section relies on the content of the note on polynomial interpolation, in particular the section on Lagrange polynomials.

Choose $n+1$ distinct nodes x_i, $i = 0, \ldots, n$ in the interval $[a,b]$, and let $p_n(x)$ be the interpolation polynomial satisfying the interpolation condition

$$p_n(x_i) = f(x_i), \qquad i = 0, 1, \ldots n.$$

We will then use $\int_a^b p_n(x)\,\mathrm{d}x$ as an approximation to $\int_a^b f(x)\,\mathrm{d}x$. By using the Lagrange form of the polynomial

$$p_n(x) = \sum_{i=0}^{n} f(x_i)\ell_i(x)$$

with the cardinal functions $\ell_i(x)$ given by

$$\ell_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j},$$

the following quadrature formula is obtained

$$\begin{aligned} I[f](a,b) \approx Q[f](a,b) &= \int_a^b p_n(x)\,\mathrm{d}x \\ &= \sum_{i=0}^{n} f(x_i) \int_a^b \ell_i(x)\,\mathrm{d}x = \sum_{i=0}^{n} w_i f(x_i) = Q(a,b), \end{aligned}$$

where the weights in the quadrature are simply the integral of the cardinal functions over the interval

$$w_i = \int_a^b \ell_i(x)\,\mathrm{d}x \quad \text{for } i = 0, \dots, n.$$

Let us derive three schemes for integration over the interval $[0, 1]$, which we will finally apply to the integral

$$I(0,1) = \int_0^1 \cos\left(\frac{\pi}{2}x\right) = \frac{2}{\pi} = 0.636619\dots\,.$$

Example 5 (The trapezoidal rule revisited)

Let $x_0 = 0$ and $x_1 = 1$. The cardinal functions and thus the weights are given by

$$\ell_0(x) = 1 - x, \qquad w_0 = \int_0^1 (1-x)\,\mathrm{d}x = 1/2$$
$$\ell_1(x) = x, \qquad w_1 = \int_0^1 x\,\mathrm{d}x = 1/2$$

and the corresponding quadrature rule is the trapezoidal rule (usually denoted by T) recalled in exa-known-qr-rules with $[a, b] = [0, 1]$:

$$\mathrm{T}[f](0,1) = \frac{1}{2}\left[f(0) + f(1)\right].$$

Example 6 (Gauß-Legendre quadrature for $n = 2$)

Let $x_0 = 1/2 + \sqrt{3}/6$ and $x_1 = 1/2 - \sqrt{3}/6$. Then

$$\ell_0(x) = -\sqrt{3}x + \frac{1+\sqrt{3}}{2}, \qquad w_0 = \int_0^1 \ell_0(x)\,\mathrm{d}x = 1/2,$$
$$\ell_1(x) = \sqrt{3}x + \frac{1-\sqrt{3}}{2}, \qquad w_1 = \int_0^1 \ell_1(x)\,\mathrm{d}x = 1/2.$$

The quadrature rule is

$$\mathrm{GL}[f](0,1) = \frac{1}{2}\left[f\left(\frac{1}{2} - \frac{\sqrt{3}}{6}\right) + f\left(\frac{1}{2} + \frac{\sqrt{3}}{6}\right)\right].$$

Example 7 (Simpson's rule revisited)

We construct Simpson's rule on the interval $[0, 1]$ by choosing the nodes $x_0 = 0$, $x_1 = 0.5$ and $x_2 = 1$. The corresponding cardinal functions are

$$\ell_0 = 2(x - 0.5)(x - 1) \qquad \ell_1(x) = 4x(1 - x) \qquad \ell_2(x) = 2x(x - 0.5)$$

which gives the weights

$$w_0 = \int_0^1 \ell_0(x)\,\mathrm{d}x = \frac{1}{6}, \qquad w_1 = \int_0^1 \ell_1(x)\,\mathrm{d}x = \frac{4}{6}, \qquad w_2 = \int_0^1 \ell_2(x)\,\mathrm{d}x = \frac{1}{6}$$

such that

$$\mathrm{S}[F](0,1) := \sum_{i=0}^{2} w_i f(x_i) = \frac{1}{6}\left[\, f(0) + 4f(0.5) + f(1) \,\right].$$

Exercise 7 (Accuracy of some quadrature rules)

Use the `QR` function below

to compute an approximate value of integral for $f(x) = \cos\left(\frac{\pi}{2}x\right)$

$$I[f](0,1) = \int_0^1 \cos\left(\frac{\pi}{2}x\right) = \frac{2}{\pi} = 0.636619\ldots\,.$$

using the quadrature rules from exa-trap-rule-revist-exa-simpson-rule. Tabulate the corresponding quadrature errors $I[f](0,1) - Q[f](0,1)$.

```
def QR(f, xq, wq):
    """ Computes an approximation of the integral f
    for a given quadrature rule.

    Input:
        f:  integrand
        xq: list of quadrature nodes
        wq: list of quadrature weights
    """
    n = len(xq)
    if (n != len(wq)):
        raise RuntimeError("Error: Need same number of quadrature nodes and weights!")
    return np.array(wq)@f(np.array(xq))
```

Hint. You can start with (fill in values for any ...)

```
# Define function
def f(x):
    return ...

# Exact integral
int_f = 2/pi

# Trapezoidal rule
xq = ...
wq = ...

qr_f = ...
print("Q[f] = {}".format(qr_f))
print("I[f] - Q[f] = {:.10e}".format(int_f - qr_f))
```

```
# Insert your code here.
```

Solution to Exercise 7 (Accuracy of some quadrature rules)

```
# Define function
```

```
def f(x):
    return np.cos(pi/2*x)

# Exact integral
int_f = 2.0/pi
print("I[f] = {}".format(int_f))

# Trapezoidal
xq = [0,1]
wq = [0.5, 0.5]

qr_f = QR(f, xq, wq)
print("Error for the trapezoidal rule")
print("Q[f] = {}".format(qr_f))
print("I[f] - Q[f] = {:.10e}".format(int_f - qr_f))

# Gauss-Legendre
print("Error for Gauss-Legendre")
xq = [0.5-sqrt(3)/6, 0.5+sqrt(3)/6]
wq = [0.5, 0.5]

qr_f = QR(f, xq, wq)
print("Q[f] = {}".format(qr_f))
print("I[f] - Q[f] = {:.10e}".format(int_f - qr_f))

# Simpson
print("Error for Simpson")
xq = [0, 0.5, 1]
wq = [1/6., 4/6., 1/6.]

qr_f = QR(f, xq, wq)
print("Q[f] = {}".format(qr_f))
print("I[f] - Q[f] = {:.10e}".format(int_f - qr_f))
```

```
I[f] = 0.6366197723675814
Error for the trapezoidal rule
Q[f] = 0.5
I[f] - Q[f] = 1.3661977237e-01
Error for Gauss-Legendre
Q[f] = 0.6356474078605917
I[f] - Q[f] = 9.7236450699e-04
Error for Simpson
Q[f] = 0.6380711874576983
I[f] - Q[f] = -1.4514150901e-03
```

Remark 2

We observe that with the same number of quadrature points, the Gauß-Legendre quadrature gives a much more accurate answer then the trapezoidal rule. So the choice of nodes clearly matters. Simpon's rule gives very similar results to Gauß-Legendre quadrature, but it uses 3 instead of 2 quadrature nodes. The quadrature rules which based on polynomial interpolation and *equidistributed quadrature nodes* go under the name **Newton Cotes formulas** (see below).

3.3 Degree of exactness and an estimate of the quadrature error

Motivated by the previous examples, we know take a closer look at how assess the quality of a method. We start with the following definition.

Definition 3 (The degree of exactness (presisjonsgrad))

A numerical quadrature has degree of exactness d if $Q[p](a,b) = I[p](a,b)$ for all $p \in \mathbb{P}_d$ and there is at least one $p \in \mathbb{P}_{d+1}$ such that $Q[p](a,b) \neq I[p](a,b)$.

Since both integrals and quadratures are linear in the integrand f, the degree of exactness is d if

$$\begin{aligned} I[x^j](a,b) &= Q[x^j](a,b), \qquad j = 0, 1, \ldots, d, \\ I[x^{d+1}](a,b) &\neq Q[x^{d+1}](a,b). \end{aligned}$$

Observation 4

All quadratures constructed from Lagrange interpolation polynomials in $n+1$ distinct nodes will automatically have a degree of exactness of **at least** n. This follows immediately from the fact the interpolation polynomial $p_n \in \mathbb{P}_n$ of any polynomial $q \in \mathbb{P}_n$ is just the original polynomial q itself. But sometimes the degree of exactness can be even higher as the next exercise shows!

Exercise 8 (Degree of exactness for some quadrature rules)

- What is the degree of exactness for the left and right endpoint rule from exa-known-qr-rules?
- What is the degree of exactness for the trapezoidal and midpoint rule from exa-known-qr-rules?
- What is the degree of exactness for Gauß-Legendre quadrature for 2 points from exa:gauss-legend-quad?
- What is the degree of exactness for Simpson's rule from exa-simpson-rule?

We test the degree of exactness for each of theses quadratures by

- computing the exact integral $I[x^n](0,1) = \int_0^1 x^n dx$ for let's say $n = 0, 1, 2, 3, 4$
- computing the corresponding numerical integral $Q[x^n](0,1)$ using the given quadrature rule.
- look at the difference $I[x^n](0,1) - Q[x^n](0,1)$ for each of the quadrature rules.

You can start from the code outline below.

Hint. You can do this either using pen and paper (boring!) or numerically (more fun!), using the code from Exercise 7, see the code outline below.

$$EL[f](0,1) = 1 \cdot f(0), \quad ER[f](0,1) = 1 \cdot f(1),$$

$$M[f](0,1) = 1 \cdot f(\tfrac{1}{2}), \quad T[f](0,1) = \frac{1}{2}\left[f(0) + f(1)\right].$$

$$GL(0,1) = \frac{1}{2}\left[f\left(\frac{1}{2} - \frac{\sqrt{3}}{6}\right) + f\left(\frac{1}{2} + \frac{\sqrt{3}}{6}\right)\right], \quad S(0,1) = \frac{1}{6}\left[f(0) + 4f(0.5) + f(1)\right].$$

```
from collections import namedtuple
qr_tuple = namedtuple("qr_tuple", ["name", "xq", "wq"])

quadrature_rules =  [
    qr_tuple(name = "left endpoint rule", xq=[0], wq=[1]),
    qr_tuple(name = "right endpoint rule", xq=[1], wq=[1])
]
print(quadrature_rules)

# n defines maximal monomial powers you want to test
for n in range(5):

    print("=========================================")
    print(f"Testing degree of exactness for n = {n}")

    # Define function
    def f(x):
        return x**n

    # Exact integral
    int_f = 1./(n+1)

    for qr in quadrature_rules:
        print("-----------------------------------------")
        print(f"Testing {qr.name}")

        qr_f = QR(f, qr.xq, qr.wq)
        print(f"Q[f] = {qr_f}")
        print(f"I[f] - Q[f] = {int_f - qr_f:.16e}")
```

```
# Insert your code here
```

It is mentimeter time! Let's go to https://www.menti.com/ for a little quiz.

Solution to Exercise 8 (Degree of exactness for some quadrature rules)

Observation 5

(Will be updated after the mentimeter!)

- left and right end point rule have degree of exactness = 0
- mid point rule has degree of exactness = 1
- trapezoidal rule has degree of exactness = 1
- Gauß-Legendre rule has degree of exactness = 3
- Simpson rule has degree of exactness = 3

3.4 Estimates for the quadrature error

Theorem 4 (Error estimate for quadrature rule with degree of exactness n)

Assume that $f \in C^{n+1}(a,b)$ and let $Q[\cdot](\{x_i\}_{i=0}^n, \{w_i\}_{i=0}^n)$ be a quadrature rule which has degree of exactness n. Then the quadrature error $|I[f] - Q[f]|$ can be estimated by

$$|I[f] - Q[f]| \leqslant \frac{M}{(n+1)!} \int_a^b \prod_{i=0}^n |x - x_i| \, \mathrm{d}x$$

where $M = \max_{\xi \in [a,b]} |f^{n+1}(\xi)|$.

Proof. Let $p_n \in \mathbb{P}_n$ be the interpolation polynomial satisfying $p_n(x_i) = f(x_i)$ for $i = 0, \dots, n$. Thanks to the error estimate for the interpolation error, we know that

$$f(x) - p_n(x) = \frac{f^{n+1}(\xi(x))}{(n+1)!} \prod_{k=0}^n (x - x_k).$$

for some $\xi(x) \in (a,b)$. Since $Q(a,b)$ has degree of exactness n we have $I[p_n] = Q[p_n] = Q[f]$ and thus

$$\begin{aligned} |I[f] - Q[f]| = |I[f] - I[p_n]| &\leqslant \int_a^b |f(x) - p_n(x)| \, \mathrm{d}x \\ &= \int_a^b \left| \frac{f^{n+1}(\xi(x))}{(n+1)!} \prod_{k=0}^n (x - x_k) \right| \mathrm{d}x \\ &\leqslant \frac{M}{(n+1)!} \int_a^b \prod_{k=0}^n |(x - x_k)| \, \mathrm{d}x, \end{aligned}$$

which concludes the proof.

The advantage of the previous theorem is that it is easy to prove. On downside is that the provided estimate can be rather crude, and often sharper estimates can be established. We give two examples here of some sharper estimates (but without proof).

Theorem 5 (Error estimate for the trapezoidal rule)

For the trapezoidal rule, there is a $\xi \in (a,b)$ such that

$$I[f] - Q[f] = \frac{(b-a)^3}{12} f''(\xi).$$

Theorem 6 (Error estimate for Simpson's rule)

For Simpson's rule, there is a $\xi \in (a, b)$ such that

$$I[f] - Q[f] = -\frac{(b-a)^5}{2880} f^4(\xi).$$

3.5 Newton-Cotes formulas

We have already seen that *given* $n+1$ distinct but otherwise arbitrary quadrature nodes $\{x_i\}_{i=0}^n \subset [a, b]$, we can construct a quadrature rule $\mathrm{Q}[\cdot](\{x_i\}_{i=0}^{n+1}, \{w_i\}_{i=0}^{n+1})$ based on polynomial interpolation which has degree of exactness equals to n.

An classical example was the trapezoidal rule, which are based on the two quadrature points $x_0 = a$ and $x_1 = b$ and which has degree of exactness equal to 1.

The trapezoidal is the simplest example of a quadrature formula which belongs to the so-called **Newton Cotes formulas**.

By definition, **Newton-Cotes formulas** are quadrature rules which are based on **equidistributed nodes** $\{x_i\}_{i=0}^n \subset [a, b]$ and have degree of exactness equals to n.

The simplest choices here — the *closed* Newton-Cotes methods — use the nodes $x_i = a + ih$ with $h = (b-a)/n$. Examples of these are the Trapezoidal rule and Simpson's rule. The main appeal of these rules is the simple definition of the nodes.

If n is odd, the Newton-Cotes method with $n+1$ nodes has degree of precision n; if n is even, it has degree of precision $n+1$. The corresponding convergence order for the composite rule is, as for all such rules, one larger than the degree of precision, provided that the function f is sufficiently smooth.

However, for $n \geq 8$ negative weights begin to appear in the definitions. Note that for a positive function $f(x) \geqslant 0$ we have that the integral $I[f](a, b) \geqslant 0$ But for a quadrature rule with negative weights **we have not necessarily that** $Q[f](a, b) \geqslant 0$**!** This has the undesired effect that the numerical integral of a positive function can be negative.

In addition, this can lead to cancellation errors in the numerical evaluation, which may result in a lower practical accuracy. Since the rules with $n = 6$ and $n = 7$ yield the same convergence order, this mean that it is mostly the rules with $n \leq 6$ that are used in practice.

The *open* Newton-Cotes methods, in contrast, use the nodes $x_i = a + (i + 1/2)h$ with $h = (b-a)/(n+1)$. The simplest example here is the midpoint rule. Here negative weights appear already for $n \geq 2$. Thus the midpoint rule is the only such rule that is commonly used in applications.

3.6 Numerical integration: Composite quadrature rules

As usual, we import the necessary modules before we get started.

```
%matplotlib inline

import numpy as np
from numpy import pi
from math import sqrt
from numpy.linalg import solve, norm    # Solve linear systems and compute norms

import matplotlib.pyplot as plt
import matplotlib.cm as cm

#import ipywidgets as widgets
#from ipywidgets import interact, fixed
```

(continues on next page)

(continued from previous page)

```
newparams = {'figure.figsize': (16.0, 8.0),
             'axes.grid': True,
             'lines.markersize': 8,
             'lines.linewidth': 2,
             'font.size': 14}
plt.rcParams.update(newparams)
#plt.xkcd()
```

3.6.1 General construction of quadrature rules

In the following, you will learn the steps on how to construct realistic algorithms for numerical integration, similar to those used in software like Matlab of SciPy. The steps are:

Construction.

1. Choose $n+1$ distinct nodes on a standard interval I, often chosen to be $I = [-1, 1]$.
2. Let $p_n(x)$ be the polynomial interpolating some general function f in the nodes, and let the $Q[f](-1, 1) = I[p_n](-1, 1)$.
3. Transfer the formula Q from $[-1, 1]$ to some interval $[a, b]$.
4. Design a composite formula, by dividing the interval $[a, b]$ into subintervals and applying the quadrature formula on each subinterval.
5. Find an expression for the error $E[f](a, b) = I[f](a, b) - Q[f](a, b)$.
6. Find an expression for an estimate of the error, and use this to create an adaptive algorithm.

3.6.2 Constructing quadrature rules on a single interval

We have already seen in the previous Lecture how quadrature rules on a given interval $[a, b]$ can be constructed using polynomial interpolation.

For $n+1$ quadrature points $\{x_i\}_{i=0}^n \subset [a, b]$, we compute weights by

$$w_i = \int_a^b \ell_i(x)\,\mathrm{d}x \quad \text{for } i = 0, \dots, n.$$

where $\ell_i(x)$ are the cardinal functions associated with $\{x_i\}_{i=0}^n$ satisfying $\ell_i(x_j) = \delta_{ij}$ for $i, j = 0, 1, \dots, n$. The resulting quadrature rule has (at least) degree of exactness equal to n.

But how to you proceed if you know want to compute an integral on a different interval, say $[c, d]$? Do we have to reconstruct all the cardinal functions and recompute the weights?

The answer is NO! One can easily transfer quadrature points and weights from one interval to another. One typically choose the simple **reference interval** $\hat{I} = [-1, 1]$. Then you determine some $n+1$ quadrature points $\{\hat{x}_i\}_{i=0}^n \subset [-1, 1]$ and quadrature weights $\{\widehat{w}_i\}_{i=0}^n$ to define a quadrature rule $Q(\hat{I})$

The quadrature points can then be transferred to an arbitrary interval $[a, b]$ to define a quadrature rule $Q(a, b)$ using the transformation

$$x = \frac{b-a}{2}\hat{x} + \frac{b+a}{2}, \quad \text{so} \quad \mathrm{d}x = \frac{b-a}{2}\,\mathrm{d}\hat{x},$$

and thus we define $\{x_i\}_{i=0}^n$ and $\{w_i\}_{i=0}^n$ by

$$x_i = \frac{b-a}{2}\hat{x}_i + \frac{b+a}{2}, \quad w_i = \frac{b-a}{2}\widehat{w}_i \quad \text{for } i = 0, \dots n.$$

Example: Simpson's rule

- Choose standard interval $[-1, 1]$. For Simpson's rule, choose the nodes $x_0 = -1$, $x_1 = 0$ and $x_2 = 1$. The corresponding cardinal functions are
  - $\ell_0 = \frac{1}{2}(x^2 - x), \qquad \ell_1(x) = 1 - x^2, \qquad \ell_2(x) = \frac{1}{2}(x^2 + x).$

which gives the weights

- $w_0 = \int_{-1}^{1} \ell_0(x)dx = \frac{1}{3}, \qquad w_1 = \int_{-1}^{1} \ell_1(x)dx = \frac{4}{3}, \qquad w_2 = \int_{-1}^{1} \ell_2(x)dx = \frac{1}{3}$

such that

- $\int_{-1}^{1} f(t)dx \approx \int_{-1}^{1} p_2(x)dx = \sum_{i=0}^{2} w_i f(x_i) = \frac{1}{3}\left[\, f(-1) + 4f(0) + f(1) \,\right].$
- After transferring the nodes and weights, Simpson's rule over the interval $[a, b]$ becomes
  - $S(a, b) = \frac{b-a}{6}\left[\, f(a) + 4f(c) + f(b) \,\right], \qquad c = \frac{b+a}{2}.$

3.6.3 Composite quadrature rules

To generate more accurate quadrature rule $Q(a, b)$ we have in principle two possibilities:

- Increase the order of the interpolation polynomial used to construct the quadrature rule.
- Subdivide the interval $[a, b]$ into smaller subintervals and apply a quadrature rule on each of the subintervals, leading to **Composite Quadrature Rules** which we will consider next.

```
colors = plt.get_cmap("Pastel1").colors

def plot_cqr_examples(m):
    f = lambda x : np.exp(x)
    fig, axs = plt.subplots(1,2)
    fig.set_figheight(4)
    fig.set_figwidth(fig.get_size_inches()[0]*1)
    #axs[0].add_axes([0.1, 0.2, 0.8, 0.7])
    a, b = -0.5,0.5
    l, r = -1.0, 1.0
    x_a = np.linspace(a, b, 100)


    for ax in axs:
        ax.set_xlim(l, r)
        x = np.linspace(l, r, 100)
        ax.plot(x, f(x), "k--", label="$f(x)$")
        #ax.fill_between(x_a, f(x_a), alpha=0.1, color='k')
        ax.xaxis.set_ticks_position('bottom')
        ax.set_xticks([a,b])
        ax.set_xticklabels(["$a$", "$b$"])
        ax.set_yticks([])
        ax.legend(loc="upper center")
```

(continues on next page)

(continued from previous page)

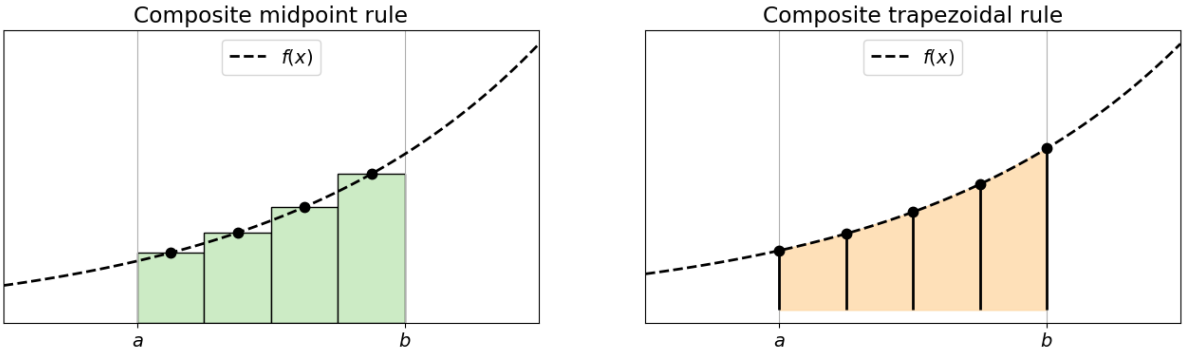
```
    h = (b-a)/m
    # Compute center points for each interval
    xcs = np.linspace(a+h/2, b-h/2, m)
    xis = np.linspace(a,b,m+1)


    # Midpoint rule
    axs[0].bar(xis[:-1], f(xcs), h, align='edge', color=colors[2], edgecolor="black")
    axs[0].plot(xcs,f(xcs), 'ko', markersize=f"{6*(m+1)/m}")
    axs[0].set_title("Composite midpoint rule")


    # Trapezoidal rule
    axs[1].set_title("Composite trapezoidal rule")
    axs[1].fill_between(xis, f(xis), alpha=0.8, color=colors[4])
    axs[1].plot(xis,f(xis), 'ko', markersize=f"{6*(m+1)/m}")
    plt.vlines(xis, 0, f(xis), colors="k")
    plt.show()
```

```
import ipywidgets as widgets
from ipywidgets import interact

slider = widgets.IntSlider(min = 1,
                           max = 20,
                           step = 1,
                           description="Number of subintervals m",
                           value = 1)
interact(plot_cqr_examples, m=slider)
```

```
interactive(children=(IntSlider(value=1, description='Number of subintervals m',␣
↪max=20, min=1), Output()), _d…
```

```
<function __main__.plot_cqr_examples(m)>
```

```
m = 4
plot_cqr_examples(m)
```



Select $m \geqslant 1$ and divide $[a, b]$ into m equally spaced subintervals $[x_{i-1}, x_i]$ defined by $x_i = a + ih$ with $h = (b - a)/m$ for $i = 1, \ldots, m$. Then for a given quadrature rule $\mathrm{Q}[\cdot](x_{i-1}, x_i)$ the corresponding composite quadrature rule $\mathrm{CQ}[\cdot]([x_{i-1}, x_i]_{i=1}^m)$ is given by

$$\int_a^b f\,\mathrm{d}x \approx \mathrm{CQ}[f]([x_{i-1},x_i]_{i=1}^m) = \sum_{i=1}^m \mathrm{Q}[f](x_{i-1},x_i).$$

(eqquad:composite_qr}

3.7 Composite trapezoidal rule

Using the trapezoidal rule

$$\mathrm{T}[f](x_{i-1},x_i) = \tfrac{h}{2}f(x_{i-1}) + \tfrac{h}{2}f(x_i)$$

the resulting composite trapezoidal rule is given by

$$\int_a^b f\,\mathrm{d}x \approx \mathrm{CT}[f]([x_{i-1},x_i]_{i=1}^m) = h\left[\tfrac{1}{2}f(x_0) + f(x_1) + \ldots + f(x_{m-1}) + \tfrac{1}{2}f(x_m)\right]$$

Exercise 9 (Testing the accuracy of the composite trapezoidal rule)

Have a look at the `CT` function which implements the composite trapezoidal rule:

```
def CT(f, a, b, m):
    """ Computes an approximation of the integral f
    using the composite trapezoidal rule.
    Input:
        f:  integrand
        a:  left interval endpoint
        b:  right interval endpoint
        m:  number of subintervals
    """
    x = np.linspace(a,b,m+1)
    h = float(b - a)/m
    fx = f(x[1:-1])
    ct = h*(np.sum(fx) + 0.5*(f(x[0]) + f(x[-1])))
    return ct
```

Use this function to compute an approximate value of integral

$$I(0,1) = \int_0^1 \cos\left(\frac{\pi}{2}x\right) = \frac{2}{\pi} = 0.636619\ldots\,.$$

for $m = 4, 8, 16, 32, 64$ corresponding to $h = 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}$. Tabulate the corresponding quadrature errors $I(0,1) - Q(0,1)$. What do you observe?

```
# Insert your code here

def f(x):
    return np.cos(np.pi/2*x)

a, b = 0, 1
m = 2
int_f = 2/np.pi
```

(continues on next page)

(continued from previous page)

```
qr_f = CT(f, a, b, m)
print(f"Exact value {int_f}")
print(f"Numerical integration for m = {m} gives {qr_f}")
print(f"Difference  = {int_f - qr_f}")

for m in [4, 8, 16, 32, 64]:
    qr_f = CT(f, a, b, m)
    print(f"Exact value {int_f}")
    print(f"Numerical integration for m = {m} gives {qr_f}")
    print(f"Difference  = {int_f - qr_f}")
```

```
Exact value 0.6366197723675814
Numerical integration for m = 2 gives 0.6035533905932737
Difference  = 0.03306638177430765
Exact value 0.6366197723675814
Numerical integration for m = 4 gives 0.6284174365157311
Difference  = 0.008202335851850262
Exact value 0.6366197723675814
Numerical integration for m = 8 gives 0.6345731492255537
Difference  = 0.002046623142027637
Exact value 0.6366197723675814
Numerical integration for m = 16 gives 0.6361083632808496
Difference  = 0.0005114090867317511
Exact value 0.6366197723675814
Numerical integration for m = 32 gives 0.6364919355013015
Difference  = 0.00012783686627992896
Exact value 0.6366197723675814
Numerical integration for m = 64 gives 0.636587814113642
Difference  = 3.195825393942364e-05
```

Solution to Exercise 9 (Testing the accuracy of the composite trapezoidal rule)

```
# Define function
def f(x):
    return np.cos(pi*x/2)

# Exact integral
int_f = 2/pi

# Interval
a, b = 0, 1

# Compute integral numerically
for m in [4, 8, 16, 32, 64]:
    cqr_f = CT(f, a, b, m)
    print(f"I[f] = {int_f}")
    print(f"Q[f, {a}, {b}, {m}] = {qr_f}")
    print(f"I[f] - Q[f] = {int_f - qr_f:.3e}")
```

Remark

We observe that for each *doubling* of the number of subintervals we decrease the error by a *fourth*. That means that if we look at the quadrature error $I[f] - \mathrm{CT}[f]$ as a function of the number of subintervals m (or equivalently as a function of h), then $|I[f] - \mathrm{CT}[f]| \approx \frac{C}{m^2} = Ch^2$.

3.7.1 Error estimate for the composite trapezoidal rule

We will now theoretically explain the experimentally observed convergence rate in the previous Exercise 9.

First we have to recall the error estimate for for the trapezoidal rule on a **single interval** $[a, b]$. If $f \in C^2(a, b)$, then there is a $\xi \in (a, b)$ such that

$$I[f] - \mathrm{T}[f] = \frac{(b-a)^3}{12} f''(\xi).$$

Theorem (Quadrature error estimate for composite trapezoidal rule)

Let $f \in C^2(a, b)$, then the quadrature error $I[f] - \mathrm{CT}[f]$ for the composite trapezoidal rule can be estimated by

$$|I[f] - \mathrm{CT}[f]| \leqslant \frac{M_2}{12} \frac{(b-a)^3}{m^2} = \frac{M_2}{12} h^2 (b-a) \tag{3.1}$$

where $M_2 = \max_{\xi \in [a,b]} |f''(\xi)|$.

Proof.

$$\begin{aligned} |I[f] - \mathrm{CT}[f]| &= \left| \sum_{i=1}^{m} \Big[\int_{x_{i-1}}^{x_i} f(x)\,\mathrm{d}x - \big(\tfrac{h}{2} f(x_{i-1}) + \tfrac{h}{2} f(x_i)\big) \Big] \right| \\ &\leqslant \sum_{i=1}^{m} \frac{h^3}{12} |f''(\xi_i)| \leqslant M_2 \sum_{i=1}^{m} \frac{(h)^3}{12} \leqslant M_2 \sum_{i=1}^{m} \frac{(h)^3}{12} \\ &= M_2 \frac{h^3}{12} \underbrace{m}_{\frac{(b-a)}{h}} = \frac{M_2}{12} h^2 (b-a) = \frac{M_2}{12} \frac{(b-a)^3}{m^2} \end{aligned}$$

3.7.2 Interlude: Convergence of h-dependent approximations

Let X be the exact solution, and $X(h)$ some numerical solution depending on a parameter h, and let $e(h)$ be the norm of the error, so $e(h) = \|X - X(h)\|$. The numerical approximation $X(h)$ converges to X if $e(h) \to 0$ as $h \to 0$. The order of the approximation is p if there exists a positive constant M such that

$$e(h) \leq Mh^p$$

This is often expresed using the Big $\mathcal{O}$-notation,

$$e(h) = \mathcal{O}(h^p) \qquad \text{as } h \to 0.$$

This is often used when we are not directly interested in any expression for the constant M, we only need to know it exists.

Again, we see that a higher approximation order p leads for small values of h to a better approximation of the solution. Thus we are generally interested in approximations of higher order.

Numerical verification

The following is based on the assumption that $e(h) \approx Ch^p$ for some unknown constant C. This assumption is often reasonable for sufficiently small h.

Choose a test problem for which the exact solution is known and compute the error for a decreasing sequence of h_k's, for instance $h_k = H/2^k$, $k = 0, 1, 2, \ldots$. The procedure is then quite similar to what was done for iterative processes.

$$\begin{array}{rcl} e(h_{k+1}) & \approx & Ch_{k+1}^p \\ e(h_k) & \approx & Ch_k^p \end{array} \quad \Rightarrow \quad \frac{e(h_{k+1})}{e(h_k)} \approx \left(\frac{h_{k+1}}{h_k}\right)^p \quad \Rightarrow \quad p \approx \frac{\log\left(e(h_{k+1})/e(h_k)\right)}{\log\left(h_{k+1}/h_k\right)}$$

For one refinement step where one passes from $h_k \to h_{k+1}$, the number

$$EOC(k) \approx \frac{\log\left(e(h_{k+1})/e(h_k)\right)}{\log\left(h_{k+1}/h_k\right)}$$

is often called the "**Experimental order of convergence** at refinement level k"

Since

$$e(h) \approx Ch^p \quad \Rightarrow \quad \underbrace{\log e(h)}_{y} \approx \underbrace{\log C}_{a} + p\,\underbrace{\log h}_{x}$$

a plot of $e(h)$ as a function of h using a logarithmic scale on both axes (a log-log plot) will be a straight line with slope p. Such a plot is referred to as an *error plot* or a *convergence plot*.

Exercise 10 (Convergence order of composite trapezoidal rule)

Examine the convergence order of composite trapezoidal rule.

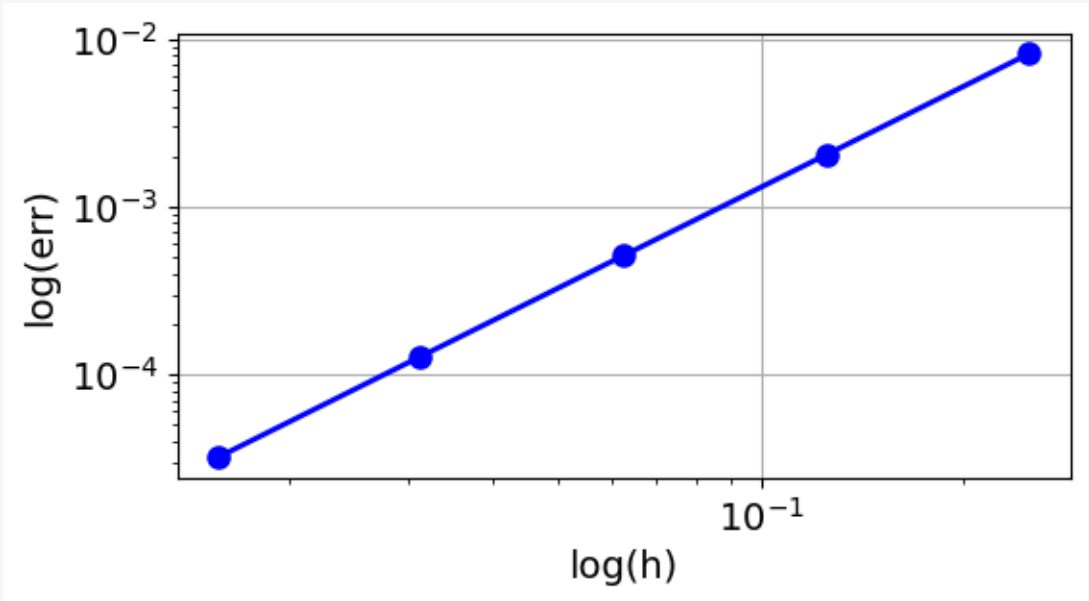
```
# Insert your code here.
```

Solution to Exercise 10 (Convergence order of composite trapezoidal rule)

```
# Define function
def f(x):
    return np.cos(pi*x/2)

# Exact integral
int_f = 2/pi

# Interval
a, b = 0, 1

errs = []
hs = []

# Compute integral numerically
for m in [4, 8, 16, 32, 64]:
    cqr_f = CT(f, a, b, m)
    print("Number of subintervals m = {}".format(m))
    print("Q[f] = {}".format(cqr_f))
    err = int_f - cqr_f
```

```
    errs.append(err)
    hs.append((b-a)/m)
    print("I[f] - Q[f] = {:.10e}".format(err))

hs = np.array(hs)
errs = np.array(errs)

eocs = np.log(errs[1:]/errs[:-1])/np.log(hs[1:]/hs[:-1])
print(eocs)
plt.figure(figsize=(6, 3))
plt.loglog(hs, errs, "bo-")
plt.xlabel("log(h)")
plt.ylabel("log(err)")

# Adding infinity in first row to eoc list
# to make it the same length as errs
eocs = np.insert(eocs, 0, np.inf)
```

```
Number of subintervals m = 4
Q[f] = 0.6284174365157311
I[f] - Q[f] = 8.2023358519e-03
Number of subintervals m = 8
Q[f] = 0.6345731492255537
I[f] - Q[f] = 2.0466231420e-03
Number of subintervals m = 16
Q[f] = 0.6361083632808496
I[f] - Q[f] = 5.1140908673e-04
Number of subintervals m = 32
Q[f] = 0.6364919355013015
I[f] - Q[f] = 1.2783686628e-04
Number of subintervals m = 64
Q[f] = 0.636587814113642
I[f] - Q[f] = 3.1958253939e-05
[2.00278934 2.00069577 2.00017385 2.00004346]
```



```
# Do a pretty print of the tables using panda
import pandas as pd
#from IPython.display import display

table = pd.DataFrame({'Error': errs, 'EOC' : eocs})
display(table)
print(table)
```

```
      Error       EOC
0  0.008202       inf
1  0.002047  2.002789
2  0.000511  2.000696
3  0.000128  2.000174
4  0.000032  2.000043
```

```
      Error       EOC
0  0.008202       inf
1  0.002047  2.002789
2  0.000511  2.000696
3  0.000128  2.000174
4  0.000032  2.000043
```

Exercise 11 (Composite Simpson's rule)

The composite Simpson's rule is considered in detail in homework assignment 2.

Theorem (Quadrature error estimate for composite Simpon's rule)

Let $f \in C^4(a,b)$, then the quadrature error $I[f] - \mathrm{CT}[f]$ for the composite trapezoidal rule can be estimated by

$$|I[f] - \mathrm{CSR}[f]| \leqslant \frac{M_4}{2880}\frac{(b-a)^5}{m^4} = \frac{M_4}{2880}h^4(b-a) \tag{3}$$

where $M_4 = \max_{\xi\in[a,b]} |f^{(4)}(\xi)|$.

Proof.

Will be part of homework assignment 2.

3.8 Summary

This chapter introduces numerical integration (quadrature) as a powerful alternative to analytical integration when closed-form antiderivatives are difficult or impossible to obtain. It builds upon ideas from polynomial interpolation to construct interpolatory quadrature rules, and explores both theoretical and practical aspects.

 Section 3.1 – Introduction and Classical Rules

- Motivation for numerical quadrature using examples of complicated integrands.
- Classical rules reviewed: Left endpoint, Right endpoint, Midpoint, Trapezoidal, and Simpson's rule.
- All rules are cast in the general quadrature form $Q[f] = \sum w_i f(x_i)$.

 Section 3.2 – Quadrature from Polynomial Interpolation

- Derivation of quadrature rules by integrating Lagrange interpolants.
- Demonstrates how weights are computed as integrals of cardinal basis functions.
- Examples include revisiting Simpson's rule and introducing Gauß-Legendre quadrature.
- A first look at the accuracy of quadrature rules

 Section 3.3 – Degree of Exactness

- Defines degree of exactness as the highest degree of polynomial a rule integrates exactly.
- Empirical testing and code examples to compute this degree for various quadrature rules.
- Notable results: Simpson and Gauss-Legendre have higher precision per point than trapezoidal or midpoint.

 Section 3.4 – Error Estimates

- Derives general error bounds for quadrature based on interpolation theory.
- Presents sharper estimates for specific rules:
- Trapezoidal error: $\frac{(b-a)^3}{12} f''(\xi)$
- Simpson error: $-\frac{(b-a)^5}{2880} f^{(4)}(\xi)$

 Section 3.5 – Newton-Cotes Formulas

- Discusses rules with equispaced nodes.
- Highlights issues with negative weights for higher-order Newton-Cotes rules, limiting practical use to $n \leq 6$.
- Differentiates between closed (includes endpoints) and open (excludes endpoints) Newton-Cotes rules.

 Section 3.6 – Composite Quadrature Rules

- General construction of quadrature rules and composite quadrature rules starting from a reference interval.
- Improves accuracy by dividing the domain into subintervals and applying quadrature on each.
- Derives and implements composite trapezoidal and midpoint rules.

 Section 3.7 – Composite Trapezoidal Rule & Convergence

- Experiments show error decreases as $\mathcal{O}(h^2)$.
- Includes full implementation and analysis of error convergence plots.
- Defines experimental order of convergence (EOC) and demonstrates how to estimate it numerically.

 Learning Outcomes for Chapter 3

By the end of this chapter, students will be able to:

 Conceptual Understanding

- Explain the motivation for numerical integration and its relevance when antiderivatives are unknown or intractable.
- Describe classical quadrature rules and identify their strengths and weaknesses.

 Quadrature Design and Implementation

- Derive numerical quadrature rules from polynomial interpolation using Lagrange polynomials.
- Compute weights in quadrature rules from cardinal functions.
- Implement rules such as:
  - Left/right endpoint

  - Midpoint and trapezoidal
  - Simpson's rule
  - Gauß-Legendre quadrature
- Implement composite quadrature rules for improved accuracy.

 Error Analysis

- Define and determine the degree of exactness of a quadrature rule.
- Use interpolation error estimates to bound integration error based on function regularity and number of nodes.
- Apply error estimates for single interval quadrature rules to derive bounds for composite rules.

 Theory and Practice

- Discuss Newton-Cotes formulas, including the issues with negative weights and their practical implications.
- Implement composite quadrature rules for improved accuracy on general intervals.
- Explain and apply the concept of experimental order of convergence (EOC) using log-log plots.
- Use composite quadrature rules (e.g., composite trapezoidal) to efficiently and accurately approximate integrals.

CHAPTER

FOUR

NUMERICAL SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS

The topic of this note is the numerical solution of systems of ordinary differential equations (ODEs). This has been discussed in previous courses, see for instance the web page Differensialligninger from Mathematics 1, as well as in Part 1 of this course, where the Laplace transform was introduced as a tool to solve ODEs analytically.

Before we present the first numerical methods to solve ODEs, we want to look at a number of examples which hopefully will will serve as test examples throughout this topic.

4.1 Whetting your appetite

4.1.1 Scalar first order ODEs

A scalar, first-order ODE is an equation on the form

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0,$$

where $y'(t) = \frac{dy}{dx}$. The *inital condition* $y(t_0) = y_0$ is required for a unique solution.

Notice.

It is common to use the term *initial value problem (IVP)* for an ODE for which the inital value $y(t_0) = y_0$ is given, and we only are interested in the solution for $t > t_0$. In these lecture notes, only initial value problems are considered.

Example 8 (Population growth and decay processes)

One of the simplest possible IVP is given by

$$y'(t) = \lambda y(t), \quad y(t_0) = y_0. \tag{4.1}$$

For $\lambda > 0$ this equation can present a simple model for the growth of some population, e.g., cells, humans, animals, with unlimited resources (food, space etc.). The constant λ then corresponds to the *growth rate* of the population.

Negative $\lambda < 0$ typically appear in decaying processes, e.g., the decay of a radioactive isotopes, where λ is then simply called the *decay rate*.

The analytical solution to ode:eq:exponential is

$$y(t) = y_0 e^{\lambda(t-t_0)} \tag{4.2}$$

and will serve us at several occasions as reference solution to assess the accuracy of the numerical methods to be introduced.

Example 9 (Time-dependent coefficients)

Given the ODE

$$y'(t) = -2ty(t), \quad y(0) = y_0.$$

for some given initial value y_0. The general solution of the ODE is

$$y(t) = Ce^{-t^2},$$

where C is a constant. To determine the constant C, we use the initial condition $y(0) = y_0$ yielding the solution

$$y(t) = y_0 e^{-t^2}.$$

4.1.2 Systems of ODEs

A system of m ODEs are given by

$$\begin{aligned} y_1' &= f_1(t, y_1, y_2, \dots, y_m), & y_1(t_0) &= y_{1,0} \\ y_2' &= f_2(t, y_1, y_2, \dots, y_m), & y_2(t_0) &= y_{2,0} \\ &\vdots & &\vdots \\ y_m' &= f_m(t, y_1, y_2, \dots, y_m), & y_m(t_0) &= y_{m,0} \end{aligned}$$

and can be written more compactly as

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \qquad \mathbf{y}(t_0) = \mathbf{y}_0$$

where we use boldface to denote vectors in $\mathbb{R}^m$,

$$\mathbf{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_m(t) \end{pmatrix}, \qquad \mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} f_1(t, y_1, y_2, \dots, y_m), \\ f_2(t, y_1, y_2, \dots, y_m), \\ \vdots \\ f_m(t, y_1, y_2, \dots, y_m), \end{pmatrix}, \qquad \mathbf{y}_0 = \begin{pmatrix} y_{1,0} \\ y_{2,0} \\ \vdots \\ y_{m,0} \end{pmatrix}.$$

Example 10 (Lotka-Volterra equation)

The Lotka-Volterra equation is a system of two ODEs describing the interaction between preys and predators over time. The system is given by

$$\begin{aligned} y'(t) &= \alpha y(t) - \beta y(t) z(t) \\ z'(t) &= \delta y(t) z(t) - \gamma z(t) \end{aligned}$$

where x denotes time, $y(t)$ describes the population of preys and $z(t)$ the population of predators. The parameters α, β, δ and γ depends on the populations to be modeled.

Example 11 (Spreading of diseases)

Motivated by the recent corona virus pandemic, we consider a (simple!) model for the spreading of an infectious disease, which goes under the name SIR model.

The SIR models divides the population into three population classes, namely

- S(t): number individuals **susceptible** for infection,
- I(t): number **infected** individuals, capable of transmitting the disease,
- R(t): number **removed** individuals who cannot be infected due death or to immunity after recovery

The model is of the spreading of a disease is based on moving individual from S to I and then to R. A short derivation can be found in [Langtangen and Linge, 2016].

The final ODE system is given by

$$S' = -\beta SI \tag{4.3}$$
$$I' = \beta SI - \gamma I \tag{4.4}$$
$$R' = \gamma I, \tag{4.5}$$

where β denotes the infection rate, and γ the removal rate.

4.1.3 Higher order ODEs

An initial value ODE of order m is given by

$$u^{(m)} = f(t, u, u', \dots, u^{(m-1)}), \qquad u(t_0) = u_0, \quad u'(t_0) = u_0', \quad \dots, \quad u^{(m-1)}(t_0) = u_0^{(m-1)}.$$

Here $u^{(1)} = u'$ and $u^{(m+1)} = \frac{du^{(m)}}{dx}$, for $m > 0$.

Example 12 (Van der Pol's equation)

Van der Pol's equation is a second order differential equation, given by

$$u^{(2)} = \mu(1 - u^2)u' - u, \qquad u(0) = u_0, \quad u'(0) = u_0',$$

where $\mu > 0$ is some constant. As initial values $u_0 = 2$ and $u_0' = 0$ are common choices.

Van der Pol's equation describes a non-conservative oscillator with non-linear damping and can be used (possibly with modifications) to model electrical circuits, heartbeats, and other biological systems exhibiting oscillatory behavior.

Later in this module we will see how such equations can be rewritten as a system of first order ODEs. Systems of higher order ODEs can be treated similarly.

4.2 Numerical solution of ordinary differential equations: Euler's and Heun's method

As always we start by running some necessary boilerplate code.

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

newparams = {'figure.figsize': (6.0, 6.0),
             'axes.grid': True,
```

(continues on next page)

(continued from previous page)

```
            'lines.markersize': 8,
            'lines.linewidth': 2,
            'font.size': 14}
plt.rcParams.update(newparams)
```

4.2.1 Euler's method

Now we turn to our first numerical method, namely Euler's method, known from Mathematics 1. We quickly review two alternative derivations, namely one based on *numerical differentiation* and one on *numerical integration*.

Derivation of Euler's method.

Euler's method is the simplest example of a so-called **one step method (OSM)**. Given the IVP

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0,$$

and some final time T, we want to compute an approximation of $y(t)$ on $[t_0, T]$.

We start from t_0 and choose some (usually small) time step size τ_0 and set the new time $t_1 = t_0 + \tau_0$. The goal is to compute a value y_1 serving as approximation of $y(t_1)$.

To do so, we Taylor expand the exact (but unknown) solution $y(t_0 + \tau)$ around x_0:

$$y(t_0 + \tau) = y(t_0) + \tau y'(t_0) + \frac{1}{2}\tau^2 y''(t_0) + \cdots .$$

Assume the step size τ to be small such that the solution is dominated by the first two terms. In that case, these can be used as the numerical approximation in the next step $t_1 := t_0 + \tau$:

$$y(t_0 + \tau) \approx y(t_0) + \tau y'(t_0) = y_0 + \tau f(t_0, y_0)$$

which means we compute

$$y_1 := y_0 + \tau_0 f(t_0, y_0).$$

as an approximation to $y(t_1)$

Now we can repeat this procedure and choose the next (possibly different) time step τ_1 and compute a numerical approximation y_2 for $y(t)$ at $t_2 = t_1 + \tau_1$ by setting

$$y_2 = y_1 + \tau_1 f(t_1, y_1).$$

The idea is to repeat this procedure until we reached the final time T resulting in the following

Algorithm (Euler's method)

Input Given a function $f(t, y)$, initial value (t_0, y_0) and maximal number of time steps N.

Output Array $\{(t_k, y_k)\}_{k=0}^{N}$ collecting approximate function value $y_k \approx y(t_k)$.

- Set $t = t_0$.
- `while` $t < T$:
  - Choose τ
  - $y_{k+1} := y_k + \tau f(t_k, y_k)$
  - $t_{k+1} := t_k + \tau_k$
  - $t := t_{k+1}$

So we can think of the Euler method as a method which approximates the continuous but unknown solution $y(t) : [t_0, T] \to \mathbb{R}$ by a discrete function $y_\Delta : \{t_0, t_1, \dots, t_{N_t}\}$ such that $y_\Delta(t_k) := y_k \approx y(t_k)$.

How to choose τ_i? The simplest possibility is to set a maximum number of steps $N_{\max} = N_t$ and then to chose a *constant time step* $\tau = (T - t_0)/N_{\max}$ resulting in $N_{\max} + 1$ equidistributed points. Later we will also learn, how to choose the *time step adaptively*, depending on the solution's behavior.

Also, in order to compute an approximation at the next point t_{k+1}, Euler's method only needs to know f, τ_k and the solution y_k at the *current* point t_k, but not at earlier points $t_{k-1}, t_{k-2}, \dots$ Thus Euler's method is an prototype of a so-called **One Step Method (OSM)**. We will formalize this concept later.

Numerical solution between the nodes.

At first we have only an approximation of $y(t)$ at the $N_t + 1$ nodes $y_\Delta : \{t_0, t_1, \dots, t_{N_t}\}$. If we want to evaluate the numerical solution between the nodes, a natural idea is to extend the discrete solution linearly between each pair of time nodes t_k, t_{k+1}. This is compatible with the way the numerical solution can be plotted, namely by connected each pair (t_k, y_k) and (t_{k+1}, y_{k+1}) with straight lines.

Interpretation: Euler's method via forward difference operators.

After rearranging terms, we can also interpret the computation of an approximation $y_1 \approx y(t_1)$ as replacing the derivative $y'(t_0) = f(t_0, y_0)$ with a **forward difference operator**

$$f(t_0, y_0) = y'(t_0) \approx \frac{y(t_1) - y(t_0)}{\tau}$$

Thus *Euler's method replace the differential quotient by a difference quotient.*

Alternative derivation via numerical integration. Recall that for a function $f : [a, b] \to \mathbb{R}$, we can approximate its integral $\int_a^b f(t)\,\mathrm{d}t$ using a *very simple* left endpoint quadrature rule from *Example 4*,

$$\int_a^b f(t)\,\mathrm{d}t \approx (b - a) f(a). \tag{4.6}$$

Turning to our IVP, we now formally integrate the ODE $y'(t) = f(t, y(t))$ on the time interval $I_k = [t_k, t_{k+1}]$ and then apply the left endpoint quadrature rule to obtain

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} y'(t)\,\mathrm{d}t = \int_{t_k}^{t_{k+1}} f(t, y(t))\,\mathrm{d}t \approx \underbrace{(t_{k+1} - t_k)}_{\tau_k} f(t_k, y(t_k))$$

Sorting terms gives us back Euler's method

$$y(t_{k+1}) \approx y(t_k) + \tau_k f(t_k, y(t_k)).$$

4.2.2 Implementation of Euler's method

Euler's method can be implemented in only a few lines of code:

```
def explicit_euler(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        ys.append(y + dt*f(t, y))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

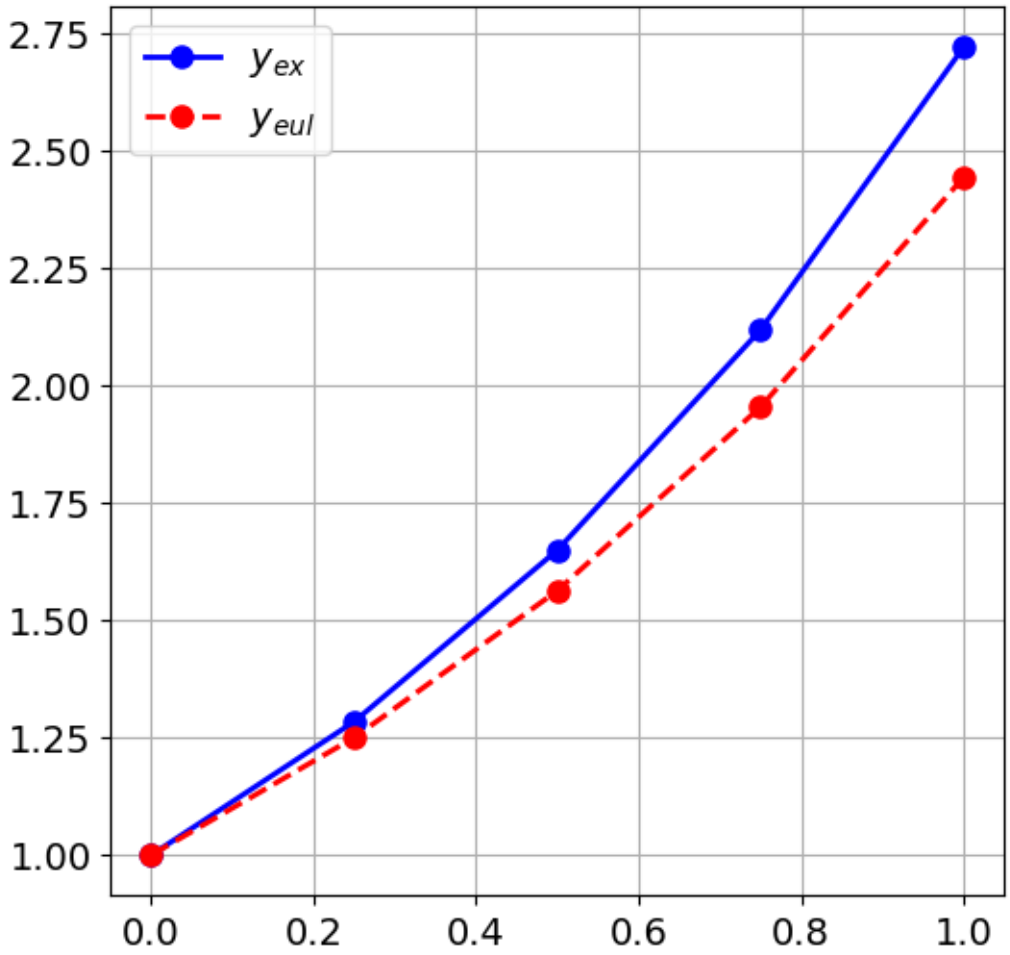
Let's test Euler's method with the simple IVP given in *Example 8*.

```
t0, T = 0, 1
y0 = 1
lam = 1
Nmax = 4

# rhs of IVP
f = lambda t,y: lam*y
print(f(t0, y0))

# Compute numerical solution using Euler
ts, ys_eul = explicit_euler(y0, t0, T, f, Nmax)

# Exact solution to compare against
y_ex = lambda t: y0*np.exp(lam*(t-t0))
ys_ex = y_ex(ts)
```

```
1
```

```
# Plot it
plt.figure()
plt.plot(ts, ys_ex, 'b-o')
plt.plot(ts, ys_eul, 'r--o')
plt.legend(["$y_{ex}$", "$y_{eul}$" ])
```

```
<matplotlib.legend.Legend at 0x109456a50>
```



Plot the solution for various N_t, say $N_t = 4, 8, 16, 32$ against the exact solution given in *Example 8*.

Exercise 12 (Error study for the Euler's method)

We observed that the more we decrease the constant step size τ (or increase $N_{\max}$), the closer the numerical solution gets to the exact solution.

Now we ask you to quantify this. More precisely, write some code to compute the error

$$\max_{i \in \{0,\dots,N_{\max}\}} |y(t_i) - y_i|$$

for $N_{\max} = 4, 8, 16, 32, 64, 128$. How does the error reduces if you double the number of points?

Complete the following code outline by filling in the missing code indicated by `....`

```
def error_study(y0, t0, T, f, Nmax_list, solver, y_ex):
    """
    Performs an error study for a given ODE solver by computing the maximum error
    between the numerical solution and the exact solution for different values of␣
↪Nmax.
    Print the list of error reduction rates computed from two consecutively solves.
```

(continues on next page)

(continued from previous page)

```
    Parameters:
      y0 : Initial condition.
      t0 : Initial time.
      T (float): Final time.
      f (function): Function representing the ODE.
      Nmax_list (list of int): List of maximum number of steps to use in the solver.
      solver (function): Numerical solver function.
      y_ex (function): Exact solution function.

    Returns:
      None
    """
    max_errs = []
    for Nmax in Nmax_list:
        # Compute list of timestep ts and computed solution ys
        ts, ys = ...
        # Evaluate y_ex in ts
        ys_ex = ...
        # Compute max error for given solution and print it
        max_errs.append(...)
        print(f"For Nmax = {Nmax:3}, max ||y(t_i) - y_i||= {max_errs[-1]:.3e}")
    # Turn list into array to allow for vectorized division
    max_errs = np.array(max_errs)
    rates = ...
    print("The computed error reduction rates are")
    print(rates)


# Define list for N_max and run error study
Nmax_list = [4, 8, 16, 32, 64, 128]
error_study(y0, t0, T, f, Nmax_list, explicit_euler, y_ex)
```

```
# Insert code here
```

Solution to Exercise 12 (Error study for the Euler's method)

```
def error_study(y0, t0, T, f, Nmax_list, solver, y_ex):
    max_errs = []
    for Nmax in Nmax_list:
        ts, ys = solver(y0, t0, T, f, Nmax)
        ys_ex = y_ex(ts)
        errors = ys - ys_ex
        max_errs.append(np.abs(errors).max())
        print(f"For Nmax = {Nmax:3}, max ||y(t_i) - y_i||= {max_errs[-1]:.3e}")
    max_errs = np.array(max_errs)
    rates = max_errs[:-1]/max_errs[1:]
    print("The computed error reduction rates are")
    print(rates)

Nmax_list = [4, 8, 16, 32, 64, 128]
error_study(y0, t0, T, f, Nmax_list, explicit_euler, y_ex)
```

```
For Nmax =   4, max ||y(t_i) - y_i||= 2.769e-01
For Nmax =   8, max ||y(t_i) - y_i||= 1.525e-01
For Nmax =  16, max ||y(t_i) - y_i||= 8.035e-02
For Nmax =  32, max ||y(t_i) - y_i||= 4.129e-02
For Nmax =  64, max ||y(t_i) - y_i||= 2.094e-02
For Nmax = 128, max ||y(t_i) - y_i||= 1.054e-02
The computed error reduction rates are
[1.81560954 1.89783438 1.94599236 1.97219964 1.98589165]
```

4.2.3 Heun's method

Before we start looking at more exciting examples, we will derive a one-step method that is more accurate than Euler's method. Note that Euler's method can be interpreted as being based on a quadrature rule with a degree of exactness equal to 0. Let's try to use a better quadrature rule!

Again, we start from the *exact representation*, but this time we use the trapezoidal rule, which has a degree of exactness equal to 1, yielding

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} f(t, y(t))\,\mathrm{d}t \approx \frac{\tau_k}{2}\left(f(t_{k+1}, y(t_{k+1})) + f(t_k, y(t_k))\right)$$

This suggest to consider the scheme

$$y_{k+1} - y_k = \frac{\tau_k}{2}\left(f(t_{k+1}, y_{k+1}) + f(t_k, y_k)\right)$$

But note that starting from y_k, we cannot immediately compute y_{k+1} as it appears also in the expression $f(t_{k+1}, y_{k+1})$! This is an example of an **implicit method**. We will discuss those later in detail.

To turn this scheme into an **explicit** scheme, the idea is now to approximate y_{k+1} appearing in f with an explicit Euler step:

$$y_{k+1} = y_k + \frac{\tau_k}{2}\left(f(t_{k+1}, y_k + \tau_k f(t_k, y_k)) + f(t_k, y_k)\right).$$

Observe that we have now nested evaluations of f. This can be best arranged by computing the nested expression in stages, first the inner one and then the outer one. This leads to the following recipe.

Algorithm (Algorithm Heun's method)

Given a function $f(t, y)$ and an initial value (t_0, y_0).

- Set $t = t_0$.
- `while` $t < T$:
  - Choose τ_k
  - Compute stage $k_1 := f(t_k, y_k)$
  - Compute stage $k_2 := f(t_k + \tau_k, y_k + \tau_k k_1)$
  - $y_{k+1} := y_k + \frac{\tau_k}{2}(k_1 + k_2)$
  - $t_{k+1} := t_k + \tau_k$
  - $t := t_{k+1}$

The function `heun` can be implemented as follows:

```
def heun(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        k1 = f(t,y)
        k2 = f(t+dt, y+dt*k1)
        ys.append(y + 0.5*dt*(k1+k2))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

Exercise 13 (Comparing Heun with Euler)

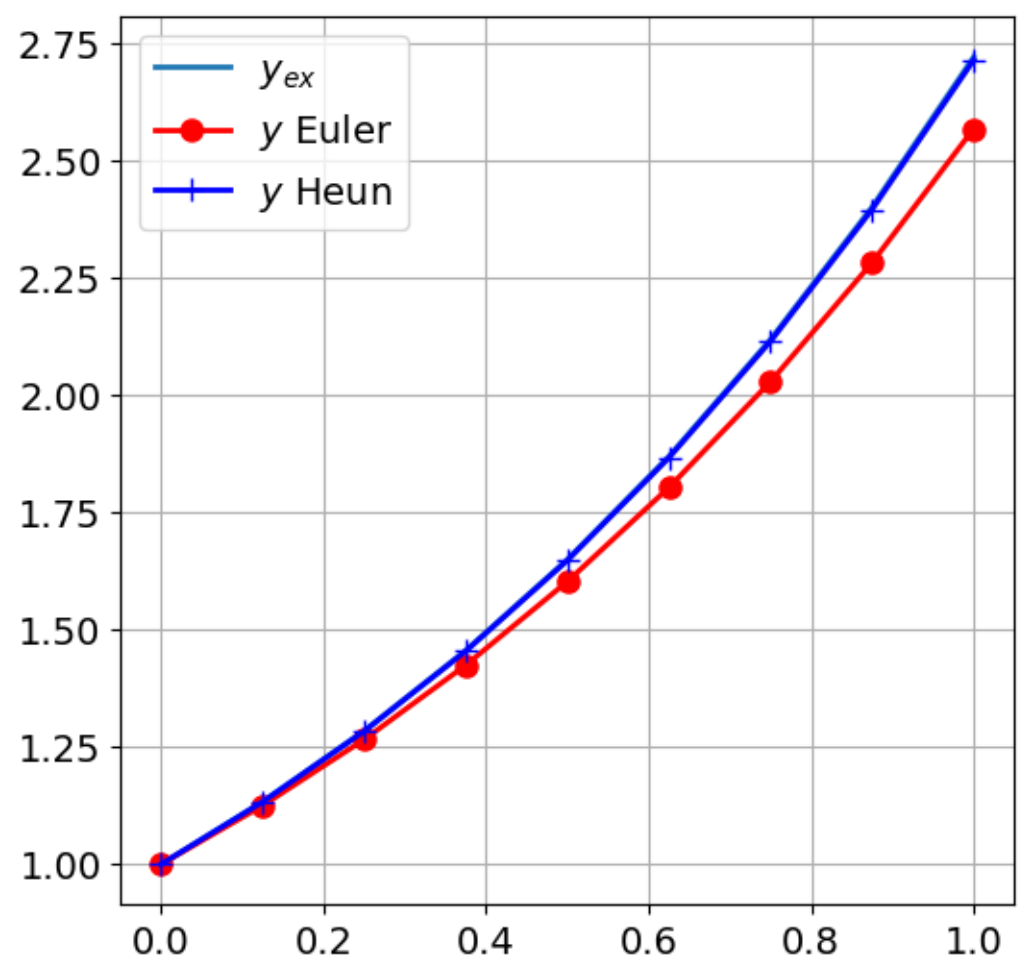
Solve *Example 8* with Heun, and plot both the exact solution, y_{eul} and y_{heun} for $N_t = 4, 8, 16, 32$.

```
# Insert code here.
```

Solution to Exercise 13 (Comparing Heun with Euler)

```
t0, T = 0, 1
y0 = 1
lam = 1
Nmax = 8

# rhs of IVP
f = lambda t,y: lam*y

# Compute numerical solution using Euler and Heun
ts, ys_eul = explicit_euler(y0, t0, T, f, Nmax)
ts, ys_heun = heun(y0, t0, T, f, Nmax)

# Exact solution to compare against
y_ex = lambda t: y0*np.exp(lam*(t-t0))
ys_ex = y_ex(ts)

# Plot it
plt.figure()
plt.plot(ts, ys_ex)
plt.plot(ts, ys_eul, 'ro-')
plt.plot(ts, ys_heun, 'b+-')
plt.legend(["$y_{ex}$", "$y$ Euler", "$y$ Heun" ])
```

```
<matplotlib.legend.Legend at 0x109735a90>
```



Exercise 14 (Error rates for Heun's method)

Redo Exercise 12 with Heun.

```
# Insert code here.
```

Solution to Exercise 14 (Error rates for Heun's method)

```
Nmax_list = [4, 8, 16, 32, 64, 128]
error_study(y0, t0, T, f, Nmax_list, heun, y_ex)
```

```
For Nmax =   4, max ||y(t_i) - y_i||= 2.343e-02
For Nmax =   8, max ||y(t_i) - y_i||= 6.441e-03
For Nmax =  16, max ||y(t_i) - y_i||= 1.688e-03
For Nmax =  32, max ||y(t_i) - y_i||= 4.322e-04
For Nmax =  64, max ||y(t_i) - y_i||= 1.093e-04
For Nmax = 128, max ||y(t_i) - y_i||= 2.749e-05
The computed error reduction rates are
[3.63726596 3.81482383 3.9067187  3.95322679 3.97658594]
```

4.2.4 Applying Heun's and Euler's method

Exercise 15 (The Lotka-Volterra equation revisited)

Solve the Lotka-Volterra equation

$$y'(t) = \alpha y(t) - \beta y(t) z(t)$$
$$z'(t) = \delta y(t) z(t) - \gamma z(t)$$

In this example, use the parameters and initial values

$$\alpha = 2, \quad \beta = 1, \quad \delta = 0.5, \quad \gamma = 1, \qquad y_{1,0} = 2, \quad y_{2,0} = 0.5.$$

Use Euler's method to solve the equation over the interval $[0, 20]$, and use $\tau = 0.02$. Try also other step sizes, e.g. $\tau = 0.1$ and $\tau = 0.002$. What do you observe?

Now use Heun's method with $\tau = 0.1$ Also try smaller step sizes.

Compare Heun's and Euler's method. How small do you have to chose the time step in Euler's method to visually match the solution from Heun's method?

Note

In this case, the exact solution is not known. What is known is that the solutions are periodic and positive. Is this the case here? Check for different values of τ.

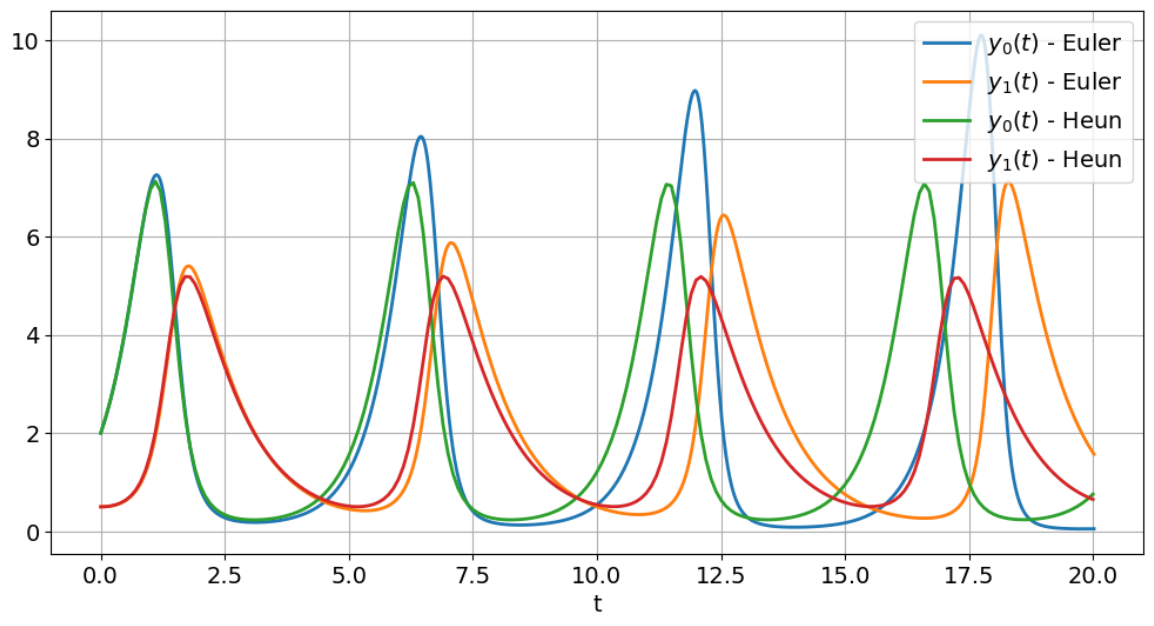
Solution to Exercise 15 (The Lotka-Volterra equation revisited)

```
# Reset plotting parameters
plt.rcParams.update({'figure.figsize': (12,6)})

# Define rhs
def lotka_volterra(t, y):
    # Set parameters
    alpha, beta, delta, gamma = 2, 1, 0.5, 1
    # Define rhs of ODE
```

```
    dy = np.array([alpha*y[0]-beta*y[0]*y[1],
                   delta*y[0]*y[1]-gamma*y[1]])
    return dy

t0, T = 0, 20             # Integration interval
y0 = np.array([2, 0.5])   # Initital values
# Solve the equation
tau = 0.02
Nmax = int((T-t0)/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_eul = explicit_euler(y0, t0, T, lotka_volterra, Nmax)

# Plot results
plt.figure()
plt.plot(ts, ys_eul)

# Solve the equation
tau = 0.1
Nmax = int((T-t0)/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_heun = heun(y0, t0, T, lotka_volterra, Nmax)

plt.plot(ts, ys_heun)
plt.xlabel('t')
plt.legend(['$y_0(t)$ - Euler', '$y_1(t)$ - Euler', '$y_0(t)$ - Heun', '$y_1(t)$ -␣
↪Heun'],
        loc="upper right" )
```

```
Nmax = 1000
Nmax =  200
```

```
<matplotlib.legend.Legend at 0x1095f2710>
```



4.2.5 Higher order ODEs

How can we numerically solve higher order ODEs using, e.g., Euler's or Heun's method?

Given the m-th order ODE

$$u^{(m)}(t) = f(t, u(t), u'(x), \dots, u^{(m-1)}).$$

For a unique solution, we assume that the initial values

$$u(t_0), u'(t_0), u''(t_0), \dots, u^{(m-1)}(t_0)$$

are known.

Such equations can be written as a system of first order ODEs by the following trick: Let

$$y_1(x) = u(x), \quad y_2(x) = u'(x), \quad y_3(x) = u^{(2)}(x), \quad \dots \quad , y_m(x) = u^{(m-1)}(x)$$

such that

$$\begin{aligned} y_1' &= y_2, & y_1(a) &= u(a) \\ y_2' &= y_3, & y_2(a) &= u'(a) \\ &\vdots & &\vdots \\ y_{m-1}' &= y_m, & y_{m-1}(a) &= u^{(m-2)}(a) \\ y_m' &= f(t, y_1, y_2, \dots, y_{m-1}, y_m), & y_m(a) &= u^{(m-1)}(a) \end{aligned}$$

which is nothing but a system of first order ODEs, and can be solved numerically exactly as before.

Exercise 16 (Numerical solution of Van der Pol's equation)

Recalling *Example 12*, the Van der Pol oscillator is described by the second order differential equation

$$u'' = \mu(1 - u^2)u' - u, \qquad u(0) = u_0, \quad u'(0) = u_0'.$$

It can be rewritten as a system of first order ODEs:

$$\begin{aligned} y_1' &= y_2, & y_1(0) &= u_0, \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1, & y_2(0) &= u_0'. \end{aligned}$$

a) Let $\mu = 2$, $u(0) = 2$ and $u'(0) = 0$ and solve the equation over the interval $[0, 20]$, using the explicit Euler and $\tau = 0.05$. Play with different step sizes, and maybe also with different values of μ.

b) Repeat the previous numerical experiment with Heun's method. Try to compare the number of steps you need to perform with Euler vs Heun to obtain visually the "same" solution. (That is, you measure the difference of the two numerical solutions in the "eyeball norm".)

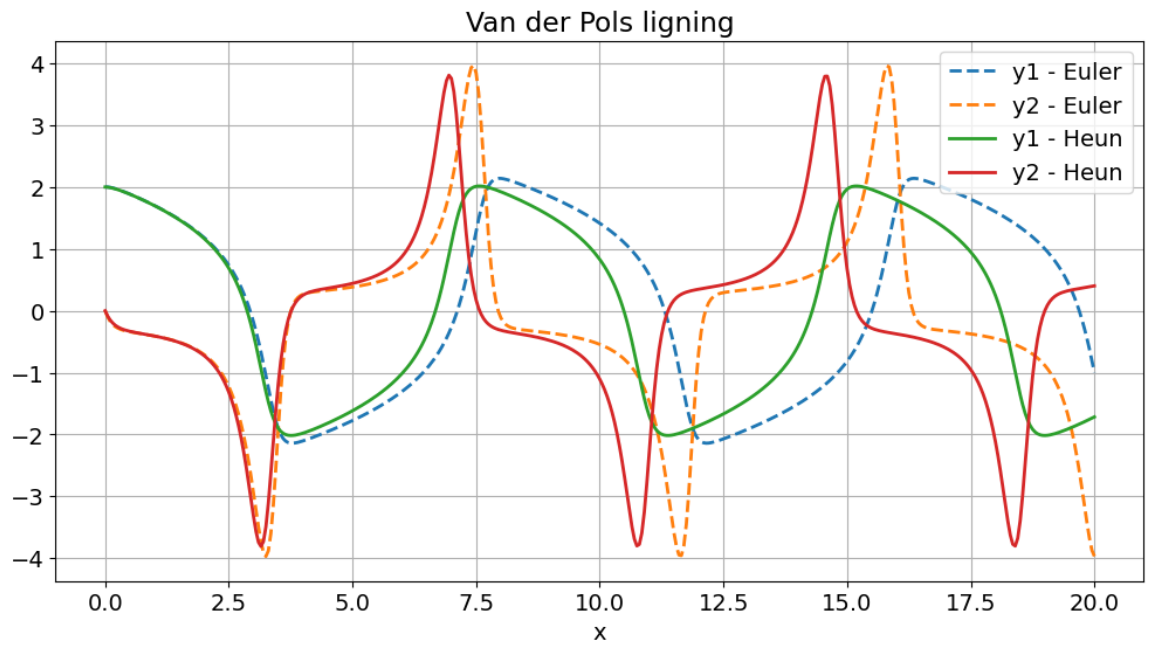
```
# Insert code here.
```

Solution to Exercise 16 (Numerical solution of Van der Pol's equation)

```
# Define the ODE
def f(t, y):
    mu = 2
    dy = np.array([y[1],
```

```
                mu*(1-y[0]**2)*y[1]-y[0] ])
    return dy

# Set initial time, stop time and initial value
t0, T = 0, 20
y0 = np.array([2,0])

# Solve the equation using Euler and plot
tau = 0.05
Nmax = int(20/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_eul  = explicit_euler(y0, t0, T, f, Nmax)

plt.figure()
plt.plot(ts,ys_eul, "--");

# Solve the equation using Heun
tau = 0.05
Nmax = int(20/tau)
print("Nmax = {:4}".format(Nmax))
ts, ys_heun = heun(y0, t0, T, f, Nmax)

plt.plot(ts,ys_heun);
plt.xlabel('x')
plt.title('Van der Pols ligning')
plt.legend(['y1 - Euler','y2 - Euler', 'y1 - Heun','y2 - Heun'],loc='upper right');
```

```
Nmax =  400
Nmax =  400
```



Observation (Euler vs. Heun)

We clearly see that Heun's method requires far fewer time steps compared to Euler's method to obtain the same (visual) solution. For instance, in the case of the Lotka-Volterra example we need with $\tau \approx 10^{-4}$ roughly 1000x more time steps for Euler than for Heuler's method, which produced visually the same solution for $\tau = 0.1$

Looking back at algorithmic realization of *Euler's method* and *Heun's method* we can compare the estimated cost for **a single time step**. Assuming that the evalution of the rhs f dominants the overall runtime cost, we observe that Euler's method requires one function evaluation while Heun's method's requires two function evaluation. That means that a single time step in Heun's method cost roughly twice as much as Euler's method. With the total number of time steps required by each method, we expect that Heun's method will result a speed up factor of roughly 500.

Let's check whether we obtain the estimated speed factor by measuring the executation time of each solution method.

To do so you can use `%timeit` and `%%timeit` magic functions in IPython/Jupyterlab, see corresponding documentation.

In a nutshell, `%%timeit` measures the executation time of an entire cell, while `%timeit` only measures only the executation time of a single line, e.g. as in

```
%timeit my_function()
```

Regarding the usage of `timeit`: To obtain reliable timeings, `timeit` does not perform a single run, but rather a number of runs , and in each run, the given statement is executed times in a loop. This can sometimes lead to large waiting time, so you can change that by time

```
%timeit -r <R> -n <N> my_function()
```

Also if you want to store the value of the best run by passing the option `-o`:

```
timings_data = %timeit -o my_function()
```

which stores the data from the timing experiment. You can access the best time measured in seconds by

```
timings_data.best
```

```
t0, T = 0, 20            # Integration interval
y0 = np.array([2, 0.5])  # Initital values
```

```
%%timeit
tau = 1e-4
Nmax = int(20/tau)
ts, ys_eul = explicit_euler(y0, t0, T, lotka_volterra, Nmax)
```

```
490 ms ± 1.08 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
tau = 0.1
Nmax = int(20/tau)
ts, ys_heun = heun(y0, t0, T, lotka_volterra, Nmax)
```

```
1.01 ms ± 14.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

4.3 Numerical solution of ordinary differential equations: Error analysis of one step methods

As always, we start by importing the necessary modules:

4.3.1 One Step Methods

In the previous lecture, we introduced the explicit Euler method and Heun's method. Both methods require only the function f, the step size τ_k, and the solution y_k at the *current* point t_k, without needing information from earlier points $t_{k-1}, t_{k-2}, \ldots$. This motivates the following definition.

Definition 4 (One step methods)

A one step method (OSM) defines an approximation to the IVP in the form of a discrete function $y_\Delta : \{t_0, \ldots, t_N\} \to \mathbb{R}^n$ given by

$$y_{k+1} := y_k + \tau_k \Phi(t_k, y_k, y_{k+1}, \tau_k) \tag{4.7}$$

for some **increment function**

$$\Phi : [t_0, T] \times \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^+ \to \mathbb{R}^n.$$

The OSM is called **explicit** if the increment function Φ does not depend on y_{k+1}, otherwise it is called **implicit**.

Example 13 (Increment functions for Euler and Heun)

The increment functions for Euler's and Heun's methods are defined as follows:

$$\Phi(t_k, y_k, y_{k+1}, \tau_k) = f(t_k, y_k), \qquad \Phi(t_k, y_k, y_{k+1}, \tau_k) = \tfrac{1}{2}\left(f(t_k, y_k) + f(t_{k+1}, y_k + \tau_k f(t_k, y_k))\right).$$

4.3.2 Local and global truncation error of OSM

Definition 5 (Local truncation error)

The **local truncation error** $\eta(t, \tau)$ is defined by

$$\eta(t, \tau) = y(t) + \tau\Phi(t, y(t), y(t+\tau), \tau) - y(t+\tau). \tag{4.8}$$

$\eta(t, \tau)$ is often also called the **local discretization** or **consistency error**.

A one step method is called **consistent of order** $p \in \mathbb{N}$ if there is a constant $C > 0$ such that

$$|\eta(t, \tau)| \leqslant C\tau^{p+1} \quad \text{for } \tau \to 0.$$

A short-hand notation for this is to write $\eta(t, \tau) = \mathcal{O}(\tau^{p+1})$ for $\tau \to 0$.

Example 14 (Consistency order of Euler's method)

Euler's method has consistency order $p = 1$.

Definition 6 (Global truncation error)

For a numerical solution $y_\Delta : \{t_0, \dots, t_N\} \to \mathbb{R}$ the **global truncation error** is defined by

$$e_k(t_{k-1}, \tau_{k-1}) = y(t_k) - y_k \quad \text{for } k = 1, \dots, N. \tag{4.9}$$

A one step method is called **convergent with order** $p \in \mathbb{N}$ if

$$\max_{k \in \{1,\dots,N\}} |e_k(t_{k-1}, \tau_{k-1})| = \mathcal{O}(\tau^p) \tag{4.10}$$

with $\tau = \max_k \tau_k$.

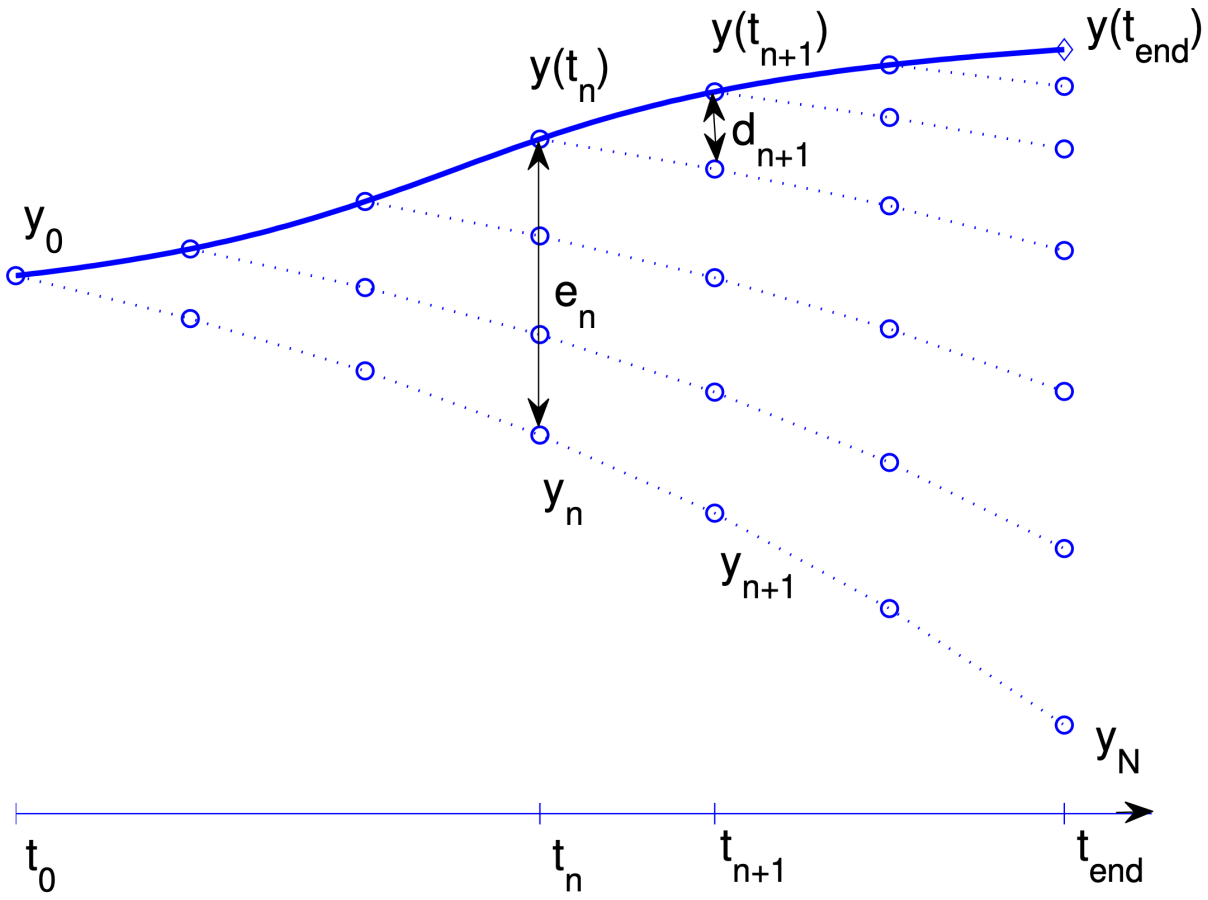


Figure. Lady Windermere's fan, named after a comedy play by Oscar Wilde. The figure describes the transport and the accumulation of the local truncation errors $\eta(t_n, \tau_n) =: d_{n+1}$ into the global error $e_N = y(t_N) - y_N$ at the end point $t_N = t_{\text{end}}$.

Discussion.

If a one step method has convergence order equal to p, the maximum error $e(\tau) = \max_k |e(t_k, \tau)|$ can be thought as a function of the step size τ is of the form

$$e(\tau) = O(\tau^p) \leqslant C\tau^p.$$

This implies that if we change the time step size from τ to e.g. $\frac{\tau}{2}$, we can expect that the error decreases from $C\tau^p$ to $C(\frac{\tau}{2})^p$, that is, the error will be reduced by a factor 2^{-p}.

How can we determine the convergence rate by means of numerical experiments?

Starting from $e(\tau) = O(\tau^p) \leqslant C\tau^p$ and taking the logarithm gives

$$\log(e(\tau)) \leqslant p\log(\tau) + \log(C).$$

Thus $\log(e(\tau))$ is a linear function of $\log(\tau)$ and the slope of this linear function corresponds to the order of convergence p.

So if you have an *exact solution* at your disposal, you can for an increasing sequence `Nmax_list` defining a descreasing sequence of *maximum* time-steps $\{\tau_0, \dots, \tau_N\}$ and solve your problem numerically and then compute the resulting exact error $e(\tau_i)$ and plot it against τ_i in a log $-$ log plot to determine the convergence order.

In addition you can also compute the experimentally observed convergence rate EOC for $i = 1, \dots M$ defined by

$$\text{EOC}(i) = \frac{\log(e(\tau_i)) - \log(e(\tau_{i-1}))}{\log(\tau_i) - \log(\tau_{i-1})} = \frac{\log(e(\tau_i)/e(\tau_{i-1}))}{\log(\tau_i/\tau_{i-1})}$$

Ideally, $\text{EOC}(i)$ is close to p.

This is implemented in the following `compute_eoc` function.

```
def compute_eoc(y0, t0, T, f, Nmax_list, solver, y_ex):
    errs = [ ]
    for Nmax in Nmax_list:
        ts, ys = solver(y0, t0, T, f, Nmax)
        ys_ex = y_ex(ts)
        errs.append(np.abs(ys - ys_ex).max())
        print("For Nmax = {:3}, max ||y(t_i) - y_i||= {:.3e}".format(Nmax,errs[-1]))

    errs = np.array(errs)
    Nmax_list = np.array(Nmax_list)
    dts = (T-t0)/Nmax_list

    eocs = np.log(errs[1:]/errs[:-1])/np.log(dts[1:]/dts[:-1])

    # Insert inf at beginning of eoc such that errs and eoc have same length
    eocs = np.insert(eocs, 0, np.inf)

    return errs, eocs
```

Here, `solver` is any ODE solver wrapped into a Python function which can be called like this

```
ts, ys = solver(y0, t0, T, f, Nmax)
```

Exercise 17

Use the `compute_eoc` function and any of the examples with a known analytical solution from the previous lecture to determine convergence order for Euler's.

Start from importing the Eulers's method from the previous lecture,

```
def explicit_euler(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        ys.append(y + dt*f(t, y))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

and copy and complete the following code snippet to compute the EOC for the explicit Euler method:

```
# Data for the ODE
# Start/stop time
t0, T = ...
# Initial value
y0 = ...
# growth/decay rate
lam = ...

# rhs of IVP
f = lambda t,y: ...

# Exact solution to compare against
# Use numpy version of exo, namely np.exp
y_ex = lambda t: ...

# List of Nmax for which you want to run the study
Nmax_list = [... ]

# Run convergence test for the solver
errs, eocs = compute_eoc(...)
print(eocs)

# Plot rates in a table
table = pd.DataFrame({'Error': errs, 'EOC' : eocs})
display(table)
print(table)
```

```
# Insert code here.
```

Solution.

```
t0, T = 0, 1
y0 = 1
lam = 1

# rhs of IVP
f = lambda t,y: lam*y

# Exact solution to compare against
y_ex = lambda t: y0*np.exp(lam*(t-t0))

Nmax_list = [4, 8, 16, 32, 64, 128, 256, 512]

errs, eocs = compute_eoc(y0, t0, T, f, Nmax_list, explicit_euler, y_ex)
print(eocs)
```

(continues on next page)

(continued from previous page)

```
table = pd.DataFrame({'Error': errs, 'EOC' : eocs})
display(table)
print(table)
```

```
For Nmax =   4, max ||y(t_i) - y_i||= 2.769e-01
For Nmax =   8, max ||y(t_i) - y_i||= 1.525e-01
For Nmax =  16, max ||y(t_i) - y_i||= 8.035e-02
For Nmax =  32, max ||y(t_i) - y_i||= 4.129e-02
For Nmax =  64, max ||y(t_i) - y_i||= 2.094e-02
For Nmax = 128, max ||y(t_i) - y_i||= 1.054e-02
For Nmax = 256, max ||y(t_i) - y_i||= 5.290e-03
For Nmax = 512, max ||y(t_i) - y_i||= 2.650e-03
[       inf 0.86045397 0.9243541  0.96050605 0.9798056  0.98978691
 0.99486396 0.99742454]
```

	Error	EOC
0	0.276876	inf
1	0.152497	0.860454
2	0.080353	0.924354
3	0.041292	0.960506
4	0.020937	0.979806
5	0.010543	0.989787
6	0.005290	0.994864
7	0.002650	0.997425

	Error	EOC
0	0.276876	inf
1	0.152497	0.860454
2	0.080353	0.924354
3	0.041292	0.960506
4	0.020937	0.979806
5	0.010543	0.989787
6	0.005290	0.994864
7	0.002650	0.997425

Exercise 18

Redo the previous exercise with Heun's method.

Start from importing the Heun's method from yesterday's lecture.

```
def heun(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        k1 = f(t,y)
        k2 = f(t+dt, y+dt*k1)
        ys.append(y + 0.5*dt*(k1+k2))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

```
# Insert code here.
```

Solution.

```
solver = heun
errs, eocs = compute_eoc(y0, t0, T, f, Nmax_list, solver, y_ex)
print(eocs)

table = pd.DataFrame({'Error': errs, 'EOC' : eocs})
display(table)
print(table)
```

```
For Nmax =   4, max ||y(t_i) - y_i||= 2.343e-02
For Nmax =   8, max ||y(t_i) - y_i||= 6.441e-03
For Nmax =  16, max ||y(t_i) - y_i||= 1.688e-03
For Nmax =  32, max ||y(t_i) - y_i||= 4.322e-04
For Nmax =  64, max ||y(t_i) - y_i||= 1.093e-04
For Nmax = 128, max ||y(t_i) - y_i||= 2.749e-05
For Nmax = 256, max ||y(t_i) - y_i||= 6.893e-06
For Nmax = 512, max ||y(t_i) - y_i||= 1.726e-06
[       inf 1.86285442 1.93161644 1.96595738 1.98303072 1.99153035
 1.99576918 1.99788562]
```

	Error	EOC
0	0.023426	inf
1	0.006441	1.862854
2	0.001688	1.931616
3	0.000432	1.965957
4	0.000109	1.983031
5	0.000027	1.991530
6	0.000007	1.995769
7	0.000002	1.997886

```
      Error       EOC
0  0.023426       inf
1  0.006441  1.862854
2  0.001688  1.931616
3  0.000432  1.965957
4  0.000109  1.983031
5  0.000027  1.991530
6  0.000007  1.995769
7  0.000002  1.997886
```

4.3.3 A general convergence result for one step methods

Note

In the following discussion, we consider only **explicit** methods where the increment function Φ **does not** depend on y_{k+1}.

Theorem 9 (Convergence of one-step methods)

Assume that there exist positive constants M and D such that the increment function satisfies

$$\|\Phi(t, \mathbf{y}; \tau) - \Phi(t, \mathbf{z}; \tau)\| \leq M\|\mathbf{y} - \mathbf{z}\|$$

and the local trunctation error satisfies

$$\|\eta(t, \tau)\| = \|\mathbf{y}(t + \tau) - (\mathbf{y}(t) + \tau\Phi(t, \mathbf{y}(t), \tau))\| \leqslant D\tau^{p+1}$$

for all t, $\mathbf{y}$ and $\mathbf{z}$ in the neighbourhood of the solution.

In that case, the global error satisfies

$$\max_{k\in\{1,\ldots,N\}} \|e_k(t_{k-1}, \tau_{k-1})\| \leqslant C\tau^p, \qquad C = \frac{e^{M(T-t_0)} - 1}{M} D,$$

where $\tau = \max_{k\in\{0,1,\ldots,N_t\}} \tau_k$.

Proof. We omit the proof here

TODO

Add proof and discuss it in class if time permits.

It can be proved that the first of these conditions are satisfied for all the methods that will be considered here.

Summary.

The convergence theorem for one step methods can be summarized as

"local truncation error behaves like $\mathcal{O}(\tau^{p+1})$" + "Increment function satisfies a Lipschitz condition" $\Rightarrow$ "global truncation error behaves like $\mathcal{O}(\tau^p)$"

or equivalently,

"consistency order p" + "Lipschitz condition for the Increment function" $\Rightarrow$ "convergence order p".

4.3.4 Convergence properties of Heun's method

Thanks to *Theorem 9*, we need to show two things to prove convergence and find the corresponding convergence of a given one step methods:

- determine the local truncation error, expressed as a power series in in the step size τ
- the condition $\|\Phi(t, y, \tau) - \Phi(t, z, \tau)\| \leqslant M\|y - z\|$

Determining the consistency order. The local truncation error is found by making Taylor expansions of the exact and the numerical solutions starting from the same point, and compare. In practice, this is not trivial. For simplicity, we will here do this for a scalar equation $y'(t) = f(t, y(t))$. The result is valid for systems as well

In the following, we will use the notation

$$f_t = \frac{\partial f}{\partial t}, \qquad f_y = \frac{\partial f}{\partial y}, \qquad f_{tt} = \frac{\partial^2 f}{\partial t^2} \qquad f_{ty} = \frac{\partial^2 f}{\partial t \partial y} \qquad \text{etc.}$$

Further, we will surpress the arguments of the function f and its derivatives. So f is to be understood as $f(t, y(t))$ although it is not explicitly written.

The Taylor expansion of the exact solution $y(t+\tau)$ is given by

$$y(t+\tau) = y(t) + \tau y'(t) + \frac{\tau^2}{2} y''(t) + \frac{\tau^3}{6} y'''(t) + \cdots .$$

Higher derivatives of $y(t)$ can be expressed in terms of the function f by using the chain rule and the product rule for differentiation.

$$\begin{aligned}
y'(t) &= f, \\
y''(t) &= f_t + f_y y' = f_t + f_y f, \\
y'''(t) &= f_{tt} + f_{ty} y' + f_{yt} f + f_{yy} y' f + f_y f_t + f_y f_y y' = f_{tt} + 2 f_{ty} f + f_{yy} f^2 + f_y f_t + (f_y)^2 f.
\end{aligned}$$

Find the series of the exact and the numerical solution around x_0, y_0 (any other point will do equally well). From the discussion above, the series for the exact solution becomes

$$y(t_0+\tau) = y_0 + \tau f + \frac{\tau^2}{2}(f_t + f_y f) + \frac{\tau^3}{6}(f_{tt} + 2 f_{ty} f + f_{yy} f^2 + f_y f_t + (f_y)^2 f) + \cdots ,$$

where f and all its derivatives are evaluated in (t_0, y_0).

For the numerical solution we get

$$\begin{aligned}
k_1 &= f(t_0, y_0) = f, \\
k_2 &= f(t_0 + \tau, y_0 + \tau k_1) \\
&= f + \tau f_t + f_y \tau k_1 + \frac{1}{2} f_{tt} \tau^2 + f_{ty} \tau \tau k_1 + \frac{1}{2} f_{yy} \tau^2 k_1^2 + \cdots \\
&= f + \tau(f_t + f_y f) + \frac{\tau^2}{2}(f_{tt} + 2 f_{ty} f + f_{yy} f^2) + \cdots , \\
y_1 &= y_0 + \frac{\tau}{2}(k_1 + k_2) = y_0 + \frac{\tau}{2}(f + f + \tau(f_t + f_y f) + \frac{\tau^2}{2}(f_{tt} + 2 f_{ty} k_1 + f_{yy} f^2)) + \cdots \\
&= y_0 + \tau f + \frac{\tau^2}{2}(f_t + f_y f) + \frac{\tau^3}{4}(f_{tt} + 2 f_{ty} f + f_{yy} f^2) + \cdots
\end{aligned}$$

and the local truncation error will be

$$\eta(t_0, \tau) = y(t_0 + \tau) - y_1 = \frac{\tau^3}{12}(-f_{tt} - 2 f_{ty} f - f_{yy} f^2 + 2 f_y f_t + 2 (f_y)^2 f) + \cdots$$

The first nonzero term in the local truncation error series is called **the principal error term**. For τ sufficiently small this is the term dominating the error, and this fact will be used later.

Although the series has been developed around the initial point, series around $x_n, y(t_n)$ will give similar results, and it is possible to conclude that, given sufficient differentiability of f there is a constant D such that

$$\max_i |\eta(t_i, \tau)| \leq D \tau^3.$$

Consequently, Heun's method is of consistency order 2.

Lipschitz condition for Φ. Further, we have to prove the condition on the increment function $\Phi(t, y)$. For f differentiable, there is for all y, z some ξ between x and y such that $f(t, y) - f(t, z) = f_y(t, \xi)(y - z)$. Let L be a constant such that $|f_y| < L$, and for all x, y, z of interest we get

$$|f(t, y) - f(t, z)| \leq L |y - z|.$$

The increment function for Heun's method is given by

$$\Phi(t,y) = \frac{1}{2}(f(t,y) + f(t+\tau, y+\tau f(t,y))).$$

By repeated use of the condition above and the triangle inequalitiy for absolute values we get

$$\begin{aligned}
|\Phi(t,y) - \Phi(t,z)| &= \frac{1}{2}|f(t,y) + f(t+\tau, y+f(t,y)) - f(t,z) - f(t+\tau, z+f(t,z)| \\
&\leq \frac{1}{2}(|f(t,y) - f(t,z)| + |f(t+\tau, y+\tau f(t,y)) - f(t+\tau, z+\tau f(t,z)|) \\
&\leq \frac{1}{2}(L|y-z| + L|y+\tau f(t,y) - z - \tau f(t,z)|) \\
&\leq \frac{1}{2}(2L|y-z| + \tau L^2|y-z|) \\
&= (L + \frac{\tau}{2}L^2)|y-z|.
\end{aligned}$$

Assuming that the step size τ is bounded upward by some τ_0, we can conclude that

$$|\Phi(t,y) - \Phi(t,z)| \leq M|y-z|, \qquad M = L + \frac{\tau_0}{2}L^2.$$

Thanks to *Theorem 9*, we can conclude that Heun's method is convergent of order 2.

In the next part, when we introduce a large class of one step methods known as Runge-Kutta methods, of which Euler's and Heun's method are particular instances. For Runge-Kutta methods we will learn about some algebraic conditions known as order conditions.

4.4 Numerical solution of ordinary differential equations: Higher order Runge-Kutta methods

As always, we start by importing some important Python modules.

4.4.1 Runge-Kutta Methods

In the previous lectures we introduced *Euler's method* and *Heun's method* as particular instances of the *One Step Methods*, and we presented the general error theory for one step method.

In this note we will consider one step methods which go under the name **Runge-Kutta methods (RKM)**. We will see that Euler's method and Heun's method are instance of RKMs. But before we start, we will derive yet another one-step method, known as *explicit midpoint rule* or *improved explicit Euler method*.

As for Heun's method, we start from the IVP $y' = f(t,y)$, integrate over $[t_k, t_{k+1}]$ and apply the midpoint rule:

$$\begin{aligned}
y(t_{k+1}) - y(t_k) &= \int_{t_k}^{t_{k+1}} f(t, y(t))\,\mathrm{d}t \\
&\approx \tau_k f(t_k + \tfrac{1}{2}\tau_k, y(t_k + \tfrac{1}{2}\tau_k))
\end{aligned}$$

Since we cannot determine the value $y(t_k + \frac{1}{2}\tau_k)$ from this system, we borrow an idea from derivation of Heun's method and approximate it using a half explicit Euler step

$$y(t_k + \tfrac{1}{2}\tau_k) \approx y(t_k + \tfrac{1}{2}\tau_k f(t_k, y(t_k)),$$

leading to the following one-step methods: Given y_k, τ_k and f, compute

$$y_{k+1} := y_k + \tau_k f(t_k + \tfrac{1}{2}\tau_k, y_k + \tfrac{1}{2}\tau_k f(t_k, y_k)). \tag{4.11}$$

The nested function expression can again be rewritten using 2 *stage derivatives*, which leads to the following form of the **explicit midpoint rule** or **improved explicit Euler method**:

$$\begin{aligned} k_1 &:= f(t_k, y_k) \\ k_2 &:= f(t_k + \tfrac{\tau_k}{2}, y_k + \tfrac{\tau_k}{2} k_1) \\ y_{k+1} &:= y_k + \tau_k k_2 \end{aligned}$$

Exercise 19 (Analyzing the improved explicit Euler method)

a) Find the increment function Φ for the improved explicit Euler method.

b) Assuming the right-hand side f of a given IVP satisfies a Lipschitz condition $\|f(t,y) - f(t,z)\| \leqslant M\|y - z\|$ with a constant L_f independent of t, show that the incremement function Φ of the improved Euler method does also satisfies a Lipschitz condition for some constant L_Φ.

Hint. Get some inspiration from the corresponding result for Heun's method derived in `ErrorAnalysisNuMeODE` notes.

c) Show the improved explicit Euler method is consistent of order 2 if the right-hand side f of a given IVP is in C^2.

Hint. Get some inspiration from the corresponding result for Heun's method derived in `ErrorAnalysisNuMeODE` notes.

Solution to Exercise 19 (Analyzing the improved explicit Euler method)

a)

$$\Phi(t_k, y_k, \tau_k) = f\left(t_k + \frac{\tau}{2}, y_k + \frac{\tau}{2} f(t_k, y_k)\right).$$

b) Using the increment function above, the Lipschitz condition for f, the triangle inequality and then the Lipschitz condition for f again we get the following:

$$\begin{aligned} |\Phi(t,y) - \Phi(t,z)| &= |f\left(t + \frac{\tau}{2}, y + \frac{\tau}{2} f(t,y)\right) - f\left(t + \frac{\tau}{2}, z + \frac{\tau}{2} f(t,z)\right)| \\ &\leq M|y + \frac{\tau}{2} f(t,y) - z - \frac{\tau}{2} f(t,z)| \\ &\leq M|y - z| + M\frac{\tau}{2}|f(t,y) - f(t,z)| \\ &\leq M|y - z| + M^2\frac{\tau}{2}|y - z| \\ &= (M + \frac{\tau}{2} M^2)|y - z|. \end{aligned}$$

Assuming that the step size τ is bounded upward by some τ_0, we can conclude that

$$|\Phi(t,y) - \Phi(t,z)| \leq L_\Phi |y - z|, \qquad L_\Phi = M + \frac{\tau_0}{2} M^2.$$

c) As before, we have the following for the exact solution:

$$y(t_0 + \tau) = y_0 + \tau f + \frac{\tau^2}{2}(f_t + f_y f) + \frac{\tau^3}{6}(f_{tt} + 2f_{ty} f + f_{yy} f f + f_y f_x f + f_y f_t + (f_y)^2 f) + \cdots,$$

where f and all its derivatives are evaluated in (t_0, y_0).

For the numerical solution we get

$$\begin{aligned} k_1 &= f(t_0, y_0) = f, \\ k_2 &= f(t_0 + \frac{\tau}{2}, y_0 + \frac{\tau}{2}k_1) \\ &= f + \frac{\tau}{2}f_t + f_y\frac{\tau}{2}k_1 + \frac{1}{8}f_{tt}\tau^2 + f_{ty}\frac{\tau^2}{4}k_1 + \frac{1}{8}f_{yy}\tau^2k_1^2 + \cdots \\ &= f + \frac{\tau}{2}(f_t + f_y f) + \frac{\tau^2}{8}(f_{tt} + 2f_{ty}f + f_{yy}f^2) + \cdots \\ y_1 &= \tau k_2 \end{aligned}$$

and the local truncation error will be

$$\eta(t_0, \tau) = y(t_0 + \tau) - y_1 = \frac{\tau^3}{24}(-f_{tt} - 2f_{ty}f - f_{yy}f^2 + 4f_y f_t + 4(f_y)^2 f) + \cdots$$

Recall that the **explicit Euler method** is defined by

$$k_1 := f(t_k, y_k) \tag{4.12}$$
$$y_{k+1} := y_k + \tau_k k_1 \tag{4.13}$$

And **Heun's method** or **explicit trapezoidal rule** is similar to the improved explicit Euler method given by

$$k_1 := f(t_k, y_k) \tag{4.14}$$
$$k_2 := f(t_k + \tau_k, y_k + \tau_k k_1) \tag{4.15}$$
$$y_{k+1} := y_k + \tau_k(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2) \tag{4.16}$$

Note that for all schemes so far, we are able to successively compute the stage derivatives, starting from $k_1 = f(t_k, y_k)$.

This is not the case for the last one-step method we encountered so far, namely the **implicit trapezoidal rule** or **Crank-Nicolson method**:

$$y_{k+1} := y_k + \tau_k(\tfrac{1}{2}\underbrace{f(t_k, y_k)}_{:=k_1} + \tfrac{1}{2}\underbrace{f(t_k + \tau_k, y_{k+1}))}_{:=k_2}$$

Using stage derivatives, we obtain this time

$$k_1 := f(t_k, y_k) \tag{4.17}$$
$$k_2 := f(t_k + \tau_k, y_k + \tau(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2)) \tag{4.18}$$
$$y_{k+1} := y_k + \tau_k(\frac{1}{2}k_1 + \frac{1}{2}k_2) \tag{4.19}$$

The previous examples and the wish for constructing higher (> 2) one-step methods leads to following definition

Definition 7

Given b_j, c_j, and a_{jl} for $j, l = 1, \ldots s$, a Runge-Kutta method is defined by the recipe

$$k_j := f(t_k + c_j\tau, y_i + \tau_k \sum_{l=1}^{s} a_{jl}k_l) \quad j = 1, \ldots s, \tag{4.20}$$

$$y_{k+1} := y_k + \tau_k \sum_{j=1}^{s} b_j k_j \tag{4.21}$$

Runge-Kutta schemes are often specified in the form of a **Butcher table**:

$$\begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$$

If $a_{ij} = 0$ for $j \geqslant i$ the Runge-Kutta method is called **explicit** as the stages k_i are defined explicitly and can be computed successively:

$$k_1 := f(t_k + c_1\tau_k, y_k)$$

$$k_2 := f(t_k + c_2\tau_k, y_k + \tau_k a_{21} k_1)$$

$$k_3 := f(t_k + c_3\tau_k, y_k + \tau_k a_{31} k_1 + \tau a_{32} k_2)$$

$$\vdots$$

$$k_j := f(t_k + c_j\tau_k, y_k + \tau_k \sum_{l=1}^{j-1} a_{jl} k_l)$$

$$\vdots$$

$$k_s := f(t_k + c_s\tau_k, y_k + \tau_k \sum_{l=1}^{s-1} a_{sl} k_l)$$

$$y_{k+1} := y_k + \tau \sum_{j=1}^{s} b_j k_j$$

Exercise 20 (Butcher tables for some well-known Runge-Kutta methods)

Write down the Butcher table for the

1. explicit Euler
2. Heun's method (explicit trapezoidal rule)
3. Crank-Nicolson (implicit trapezoidal rule)
4. improved explicit Euler method (explicit midpoint rule)

and go to "www.menti.com" and take the quiz.

A) $\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$ **B)** $\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$ **C)** $\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$ **D)** $\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$

Solution to Exercise 20 (Butcher tables for some well-known Runge-Kutta methods)

The correct pairing is

1. explicit Euler: **B)**
2. Heun's method (explicit trapezoidal rule): **C)**
3. Crank-Nicolson (implicit trapezoidal rule): **D)**
4. improved explicit Euler method (explicit midpoint rule): **A)**

We show a verbose solution for explicit Euler, improved explicit Euler and Crank-Nicolson.

Explicit Euler method: Since we have only one stage derivative, this is an example of a 1-stage Runge-Kutta method (s=1). Looking at the definition of the stage and the final step, we see that

$$k_1 := f(t_k, y_k) = f(t_k + \underbrace{0}_{c_1} \cdot \tau_k, y_k + \tau_k \underbrace{0}_{a_{11}} \cdot k_1) \Rightarrow c_1 = a_{11} = 0$$

$$y_{k+1} := y_k + \tau_k k_1 = y_k + \tau_k \underbrace{1}_{b_1} \cdot k_1 \Rightarrow b_1 = 1$$

Thus, the Butcher table is

$$\mathbf{B)} \quad \begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$$

Improved explicit Euler method: Since we have to stage derivatives, this is an example of a 2-stage Runge-Kutta method (s=2). Looking at the definition of the stages and the final step, we see that

$$k_1 := f(t_k, y_k) = f(t_k + \underbrace{0}_{c_1} \cdot \tau_k, y_k + \tau_k \underbrace{0}_{a_{11}} \cdot k_1 + \tau_k \underbrace{0}_{a_{21}} \cdot k_2) \Rightarrow c_1 = a_{11} = a_{21} = 0$$

$$k_2 := f(t_k + \tfrac{\tau_k}{2}, y_k + \tfrac{\tau_k}{2} k_1)$$

$$= f(t_k + \underbrace{\tfrac{1}{2}}_{c_2} \tau_k, y_k + \tau_k \underbrace{\tfrac{1}{2}}_{a_{21}} \cdot k_1 + \tau_k \underbrace{0}_{a_{22}} \cdot k_2) \Rightarrow c_2 = \tfrac{1}{2}, a_{21} = \tfrac{1}{2}, a_{22} = 0$$

$$y_{k+1} := y_k + \tau_k k_2 = y_k + \tau_k \underbrace{0}_{b_1} \cdot k_1 \tau_k \underbrace{1}_{b_2} \cdot k_2 \Rightarrow b_1 = 0, b_2 = 1$$

Thus, the Butcher table is

$$\mathbf{A)} \quad \begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$

Crank-Nicolson method: Since we have to stage derivatives, this is an example of a 2-stage Runge-Kutta method (s=2). Looking at the definition of the stages and the final step, we see that

$$k_1 := f(t_k, y_k) = f(t_k + \underbrace{0}_{c_1} \cdot \tau_k, y_k + \tau_k \underbrace{0}_{a_{11}} \cdot k_1 + \tau_k \underbrace{0}_{a_{21}} \cdot k_2) \Rightarrow c_1 = a_{11} = a_{21} = 0$$

$$k_2 := f(t_k + \tau_k, y_k + \tau_k \tfrac{1}{2} k_1 + \tau_k \tfrac{1}{2} k_2)$$

$$= f(t_k + \underbrace{1}_{c_1} \cdot \tau_k, y_k + \tau_k \underbrace{\tfrac{1}{2}}_{a_{21}} k_1 + \tau_k \underbrace{\tfrac{1}{2}}_{a_{22}} k_2) \Rightarrow c_1 = 1, a_{21} = a_{22} = \tfrac{1}{2}$$

$$y_{k+1} := y_k + \tau_k(\frac{1}{2} k_1 + \frac{1}{2} k_2) = y_k + \tau_k \underbrace{\frac{1}{2}}_{b_1} k_1 + \tau_k \underbrace{\frac{1}{2}}_{b_2} k_2$$

Thus, the Butcher table is

$$\mathbf{D)} \quad \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

4.4.2 Implementation of explicit Runge-Kutta methods

Below you will find the implementation a general solver class `ExplicitRungeKutta` which at its initialization takes in a Butcher table and has `__call__` function

```
        def __call__(self, y0, f, t0, T, n):
```

and can be used like this

```
        # Define Butcher table
        a = np.array([[0, 0, 0],
                      [1.0/3.0, 0, 0],
                      [0, 2.0/3.0, 0]])

        b = np.array([1.0/4.0, 0, 3.0/4.0])

        c = np.array([0,
                      1.0/3.0,
                      2.0/3.0])

        # Define number of time steps
        n = 10

        # Create solver using the Butcher table
        rk3 = ExplicitRungeKutta(a, b, c)

        # Solve problem (applies __call__ function)
        ts, ys = rk3(y0, t0, T, f, Nmax)
```

The complete implementation is given here:

```
class ExplicitRungeKutta:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __call__(self, y0, t0, T, f, Nmax):
        # Extract Butcher table
        a, b, c = self.a, self.b, self.c

        # Stages
        s = len(b)
        ks = [np.zeros_like(y0, dtype=np.double) for s in range(s)]

        # Start time-stepping
        ys = [y0]
        ts = [t0]
        dt = (T - t0)/Nmax

        while(ts[-1] < T):
            t, y = ts[-1], ys[-1]

            # Compute stages derivatives k_j
            for j in range(s):
                t_j = t + c[j]*dt
                dY_j = np.zeros_like(y, dtype=np.double)
                for l in range(j):
```

(continues on next page)

(continued from previous page)

```
                    dY_j += dt*a[j,l]*ks[l]

                ks[j] = f(t_j, y + dY_j)

            # Compute next time-step
            dy = np.zeros_like(y, dtype=np.double)
            for j in range(s):
                dy += dt*b[j]*ks[j]

            ys.append(y + dy)
            ts.append(t + dt)

        return (np.array(ts), np.array(ys))
```

Example 15 (Implementation and testing of the improved Euler method)

We implement the **improved explicit Euler** from above and plot the analytical and the numerical solution. To determine the experimental order of convergence, we use again the `compute_eoc` function.

```
def compute_eoc(y0, t0, T, f, Nmax_list, solver, y_ex):
    errs = [ ]
    for Nmax in Nmax_list:
        ts, ys = solver(y0, t0, T, f, Nmax)
        ys_ex = y_ex(ts)
        errs.append(np.abs(ys - ys_ex).max())
        print("For Nmax = {:3}, max ||y(t_i) - y_i||= {:.3e}".format(Nmax,errs[-1]))

    errs = np.array(errs)
    Nmax_list = np.array(Nmax_list)
    dts = (T-t0)/Nmax_list

    eocs = np.log(errs[1:]/errs[:-1])/np.log(dts[1:]/dts[:-1])

    # Insert inf at beginning of eoc such that errs and eoc have same length
    eocs = np.insert(eocs, 0, np.inf)

    return errs, eocs
```

Here is the implementation of the full example.

```
# Define Butcher table for improved Euler
a = np.array([[0,   0],
              [0.5, 0]])
b = np.array([0, 1])
c = np.array([0, 0.5])

# Create a new Runge Kutta solver
rk2 = ExplicitRungeKutta(a, b, c)

t0, T = 0, 1
y0 = 1
lam = 1
Nmax = 10

# rhs of IVP
```

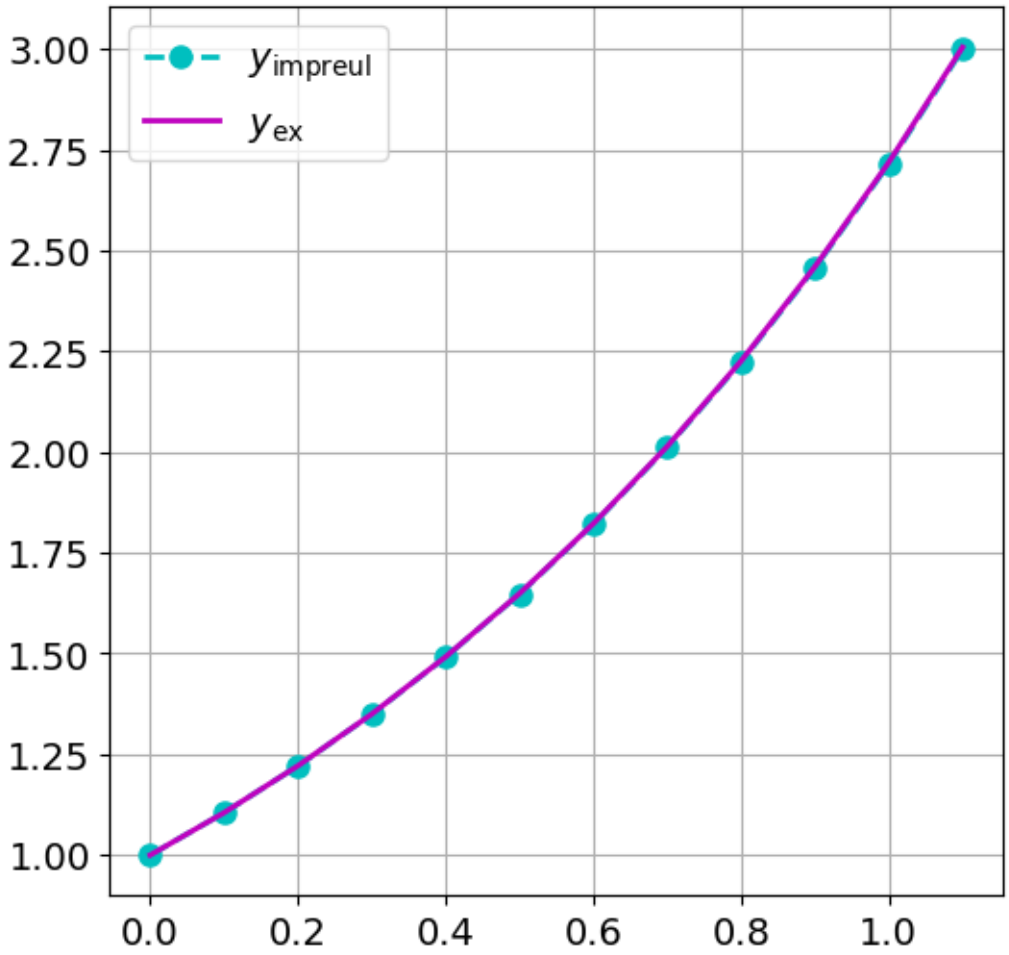
(continues on next page)

(continued from previous page)

```
f = lambda t,y: lam*y

# the solver can be simply called as before, namely as function:
ts, ys = rk2(y0, t0, T, f, Nmax)

plt.figure()
plt.plot(ts, ys, "c--o", label=r"$y_{\mathrm{impreul}}$")

# Exact solution to compare against
y_ex = lambda t: y0*np.exp(lam*(t-t0))

# Plot the exact solution (will appear in the plot above)
plt.plot(ts, y_ex(ts), "m-", label=r"$y_{\mathrm{ex}}$")
plt.legend()

# Run an EOC test
Nmax_list = [4, 8, 16, 32, 64, 128]

errs, eocs = compute_eoc(y0, t0, T, f, Nmax_list, rk2, y_ex)
print(errs)
print(eocs)

# Do a pretty print of the tables using panda

import pandas as pd
from IPython.display import display

table = pd.DataFrame({'Error': errs, 'EOC' : eocs})
display(table)
```

```
For Nmax =   4, max ||y(t_i) - y_i||= 2.343e-02
For Nmax =   8, max ||y(t_i) - y_i||= 6.441e-03
For Nmax =  16, max ||y(t_i) - y_i||= 1.688e-03
For Nmax =  32, max ||y(t_i) - y_i||= 4.322e-04
For Nmax =  64, max ||y(t_i) - y_i||= 1.093e-04
For Nmax = 128, max ||y(t_i) - y_i||= 2.749e-05
[2.34261385e-02 6.44058991e-03 1.68830598e-03 4.32154479e-04
 1.09316895e-04 2.74901378e-05]
[       inf 1.86285442 1.93161644 1.96595738 1.98303072 1.99153035]
```

	Error	EOC
0	0.023426	inf
1	0.006441	1.862854
2	0.001688	1.931616
3	0.000432	1.965957
4	0.000109	1.983031
5	0.000027	1.991530



While the term Runge-Kutta methods nowadays refer to the general scheme defined in Definition *Definition 1: Runge-Kutta methods*, particular schemes in the "early days" were named by their inventors, and there exists also the the classical 4-stage Runge-Kutta method which is defined by

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

a) Starting from this Butcher table, write down the explicit formulas for computing $k_1, \ldots, k_4$ and y_{k+1}.

b) Build a solver based on the classical Runge-Kutta method using the `ExplicitRungeKutta` class and determine the convergence order experimentally.

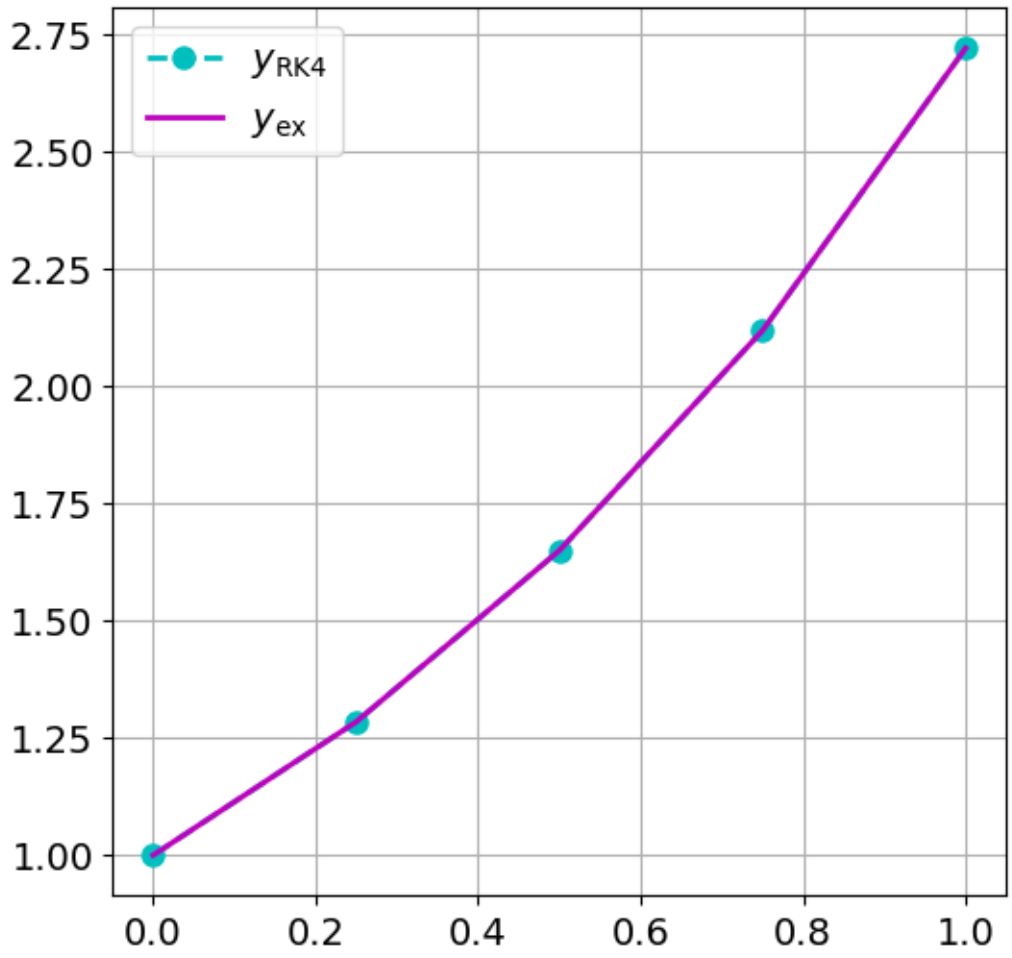
```
# Insert your code here

# Define Butcher table for improved Euler
a = np.array([[0,   0,   0,  0],
              [1/2, 0,   0,  0],
              [0,   1/2, 0,  0],
              [0,   0,   1,  0]])
b = np.array([1/6, 1/3, 1/3, 1/6])
c = np.array([0, 1/2, 1/2, 1])
```

(continues on next page)

(continued from previous page)

```
# Create a new Runge Kutta solver
rk4 = ExplicitRungeKutta(a, b, c)

t0, T = 0, 1
y0 = 1
lam = 1
Nmax = 4

# the solver can be simply called as before, namely as function:
ts, ys = rk4(y0, t0, T, f, Nmax)

plt.figure()
plt.plot(ts, ys, "c--o", label=r"$y_{\mathrm{RK4}}$")
plt.plot(ts, y_ex(ts), "m-", label=r"$y_{\mathrm{ex}}$")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1154f8910>
```




```
# Run an EOC test
Nmax_list = [4, 8, 16, 32, 64, 128]
errs, eocs = compute_eoc(y0, t0, T, f, Nmax_list, rk4, y_ex)
```

(continues on next page)

(continued from previous page)

```
table = pd.DataFrame({'Error': errs, 'EOC' : eocs})
display(table)
```

```
For Nmax =   4, max ||y(t_i) - y_i||= 7.189e-05
For Nmax =   8, max ||y(t_i) - y_i||= 4.984e-06
For Nmax =  16, max ||y(t_i) - y_i||= 3.281e-07
For Nmax =  32, max ||y(t_i) - y_i||= 2.105e-08
For Nmax =  64, max ||y(t_i) - y_i||= 1.333e-09
For Nmax = 128, max ||y(t_i) - y_i||= 8.384e-11
```

	Error	EOC
0	7.188926e-05	inf
1	4.984042e-06	3.850388
2	3.281185e-07	3.925028
3	2.104785e-08	3.962472
4	1.332722e-09	3.981225
5	8.384093e-11	3.990577

Note

For the **explicit** Runge-Kutta methods, the $s \times s$ matrix is in fact just a lower left triangle matrix, and often, the 0s in the diagonal and upper right triangle are simply omitted. So, the Butcher table for the classical Runge-Kutta method reduces to

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

Note

If f depends only on t but not on y, the ODE $y' = f(t, y) = f(t)$ reduces to a simpe integration problem, and in this case the classical Runge-Kutta methods reduces to the classical Simpson's rule for numerical integration.

More generally, when applied to simple integration problems, Runge-Kutta methods reduces to various quadrature rules over the interval $[t_i, t_{i+1}]$ with s quadrature points, where the integration points $\{\xi_j\}_{i=1}^s$ and correspond weights $\{w_i\}_{i=1}^s$ of the quadrature rules are given by respectively $\xi_i = t_i + c_j\tau$, $w_i = b_i$ for $j = 1, \dots, s$

See this wiki page for a list of various Runge-Kutta methods.

4.4.3 Runge-Kutta Methods via Numerical Integration

This section provides a supplemental and more in-depth motivation of how to arrive at the general concept of Runge-Kutta methods via numerical integration, similar to the ideas we already presented when we derived Crank-Nicolson, Heun's method and the explicit trapezoidal rule.

For a given time interval $I_i = [t_i, t_{i+1}]$ we want to compute y_{i+1} assuming that y_i is given. Starting from the exact expression

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} y(t)' \, \mathrm{d}t = \int_{t_i}^{t_{i+1}} f(t, y(t)) \, \mathrm{d}t,$$

the idea is now to approximate the integral by some quadrature rule $\mathrm{Q}[\cdot](\{\xi_j\}_{j=1}^s, \{b_j\}_{j=1}^s)$ defined on I_i. Then we get

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t, y(t)) \, \mathrm{d}t$$

$$\approx \tau \sum_{j=0}^{s} b_j f(\xi_j, y(\xi_j))$$

Now we can define $\{c_j\}_{j=1}^s$ such that $\xi_j = t_i + c_j \tau$ for $j = 1, \dots, s$

Exercise 21 (A first condition on b_j)

Question: What value do you expect for $\sum_{j=1}^s b_j$?

Choice A: $\sum_{j=1}^s b_j = \tau$

Choice B: $\sum_{j=1}^s b_j = 0$

Choice C: $\sum_{j=1}^s b_j = 1$

Solution to Exercise 21 (A first condition on b_j)

The correct answer is **C**

In contrast to pure numerical integration, we don't know the values of $y(\xi_j)$. Again, we could use the same idea to approximate

$$y(\xi_j) - y(t_i) = \int_{t_i}^{t_i + c_j \tau} y'(t) \, \mathrm{d}t = \int_{t_i}^{t_i + c_j \tau} f(t, y(t)) \, \mathrm{d}t$$

but then again we get a closure problem if we choose new quadrature points. The idea is now to **not introduce even more new quadrature points** but to use same $y(\xi_j)$ to avoid the closure problem. Note that this leads to an approximation of the integrals $\int_{t_i}^{t_i + c_j \tau}$ with possible nodes **outside** of $[t_i, t_i + c_j \tau]$.

This leads us to

$$y(\xi_j) - y(t_i) = \int_{t_i}^{t_i + c_j \tau} f(t, y(t)) \, \mathrm{d}t$$

$$\approx c_j \tau \sum_{l=1}^{s} \tilde{a}_{jl} f(\xi_l, y(\xi_l))$$

$$= \tau \sum_{l=1}^{s} a_{jl} f(\xi_l, y(\xi_l))$$

where we set $c_j \tilde{a}_{jl} = a_{jl}$.

Exercise 22 (A first condition on a_{jl})

Question: What value do you expect for $\sum_{l=1}^{s} a_{jl}$?

Choice A: $\sum_{l=1}^{s} a_{jl} = \frac{1}{c_j}$

Choice B: $\sum_{l=1}^{s} a_{jl} = c_j$

Choice C: $\sum_{l=1}^{s} a_{jl} = 1$

Choice D: $\sum_{l=1}^{s} a_{jl} = \tau$

Solution to Exercise 22 (A first condition on a_{jl})

The correct answer is **B**

The previous discussion leads to the following alternative but equivalent definition of Runge-Kutta derivatives via *stages* Y_j (and not stage derivatives k_j):

Definition 8 (Runge-Kutta methods using stages $\{Y_l\}_{l=1}^{s}$)

Given b_j, c_j, and a_{jl} for $j, l = 1, \dots s$, the Runge-Kutta method is defined by the recipe

$$Y_l := y_i + \tau \sum_{j=1}^{s} a_{lj} f(t_i + c_j \tau, Y_j) \quad \text{for } l = 1, \dots s,$$

$$y_{i+1} := y_i + \tau \sum_{j=1}^{s} b_j f(t_i + c_j \tau, Y_j)$$

Note that in the final step, all the function evaluation we need to perform have already been performed when computing Y_j.

Therefore one often rewrite the scheme by introducing **stage derivatives** k_l

$$k_j := f(t_i + c_j \tau, Y_j) \tag{4.22}$$

$$= f(t_i + c_j \tau, y_i + \tau \sum_{l=1}^{s} a_{jl} k_l) \quad j = 1, \dots s, \tag{4.23}$$

so the resulting scheme will be

$$k_j := f(t_i + c_j \tau, y_i + \tau \sum_{l=1}^{s} a_{jl} k_l) \quad j = 1, \dots s, \tag{4.24}$$

$$y_{i+1} := y_i + \tau \sum_{j=1}^{s} b_j k_j \tag{4.25}$$

which is exactly what we used as definition for general Runge-Kutta methods in the previous section.

4.4.4 Convergence of Runge-Kutta Methods

The convergence theorem for one-step methods gave us some necessary conditions to guarantee that a method is convergent order of p: "consistency order p" + "Increment function satisfies a Lipschitz condition" $\Rightarrow$ "convergence order p.

``local truncation error behaves like $\mathcal{O}(\tau^{p+1})$"

- "Increment function satisfies a Lipschitz condition" $\Rightarrow$ "global truncation error behaves like $\mathcal{O}(\tau^p)$"

It turns out that for f is at least C^1 with respect to all its arguments then the increment function Φ associated with any Runge-Kutta methods satisfies a Lipschitz condition. The next theorem provides us a simple way to check whether a given Runge-Kutta (up to 4 stages) attains a certain consistency order.

Theorem 10 (Order conditions for Runge-Kutta methods)

Let the right-hand side f of an IVP be of C^p. Then a Runge - Kutta method has consistency order p if and only if all the conditions up to and including p in the table below are satisfied.

p	conditions
1	$\sum_{i=1}^{s} b_i = 1$
2	$\sum_{i=1}^{s} b_i c_i = 1/2$
3	$\sum_{i=1}^{s} b_i c_i^2 = 1/3$ $\sum_{i,j=1}^{s} b_i a_{ij} c_j = 1/6$
4	$\sum_{i=1}^{s} b_i c_i^3 = 1/4$ $\sum_{i,j=1}^{s} b_i c_i a_{ij} c_j = 1/8$ $\sum_{i,j=1}^{s} b_i a_{ij} c_j^2 = 1/12$ $\sum_{i,j,k=1}^{s} b_i a_{ij} a_{jk} c_k = 1/24$

where sums are taken over all the indices from 1 to s.

Proof. We don't present a proof, but the most straight-forward approach is similar to the way we show that Heun's method and the improved Euler's method are consistent of order 2: You perform a Taylor-expansion of the real solution and express all derivatives of $y(t)$ in terms of derivatives of f by invoking the chain rule. Then you perform Taylor expansion of the various stages in the Runge-Kutta methods and gather all terms with the the same τ order. To achieve a certain consistency order p, the terms paired with τ^k, $k = 0, \dots, p$ in the Taylor-expansion of the discrete solution must match the corresponding terms of the exact solution, which in turn will lead to certain condition for b_j, c_j and a_{ij}.

Of course, this get quite cumbersome for higher order methods, and luckily there is a beautiful theory, will tells you how to do Taylor-expansion of the discrete and exact solution in term of derivatives of f in very structured manner. See [Hairer and Wanner, 1993](Chapter II.2).

Exercise 23 (Applying order conditions to Heun's method)

Apply the conditions to Heun's method, for which $s = 2$ and the Butcher tableau is

$$\begin{array}{c|cc} c_1 & a_{11} & a_{12} \\ c_2 & a_{21} & a_{22} \\ \hline & b_1 & b_2 \end{array} = \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}.$$

Solution to Exercise 23 (Applying order conditions to Heun's method)

The order conditions are:

$p = 1$ $\quad b_1 + b_2 = \frac{1}{2} + \frac{1}{2} = 1$ OK

$p = 2$ $\quad b_1 c_1 + b_2 c_2 = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2}$ OK

$p = 3$ $\quad b_1 c_1^2 + b_2 c_2^2 = \frac{1}{2} \cdot 0^2 + \frac{1}{2} \cdot 1^2 = \frac{1}{2} \neq \frac{1}{3}$ Not satisfied

$$b_1(a_{11}c_1 + a_{12}c_2) + b_2(a_{21}c_1 + a_{22}c_2) = \frac{1}{2}(0 \cdot 0 + 0 \cdot 1) + \frac{1}{2}(1 \cdot 0 + 0 \cdot 1)$$

$$= 0 \neq \frac{1}{6}$$ Not satisfied

The method is of order 2.

Exercise 24 (Applying order conditions to the classical Runge-Kutta method)

In our numerical experiment earlier, we observed that classical 4-stage Runge-Kutta method defined by

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

had convergence order 4. Now use the order conditions to show, that this method indeed has **consistency order** 4.

Apply the conditions to Heun's method, for which $s = 2$ and the Butcher tableau is

Theorem 11 (Convergence theorem for Runge-Kutta methods)

Given the IVP $y' = f(t, y), y(0) = y_0$. Assume $f \in C^p$ and that a given Runge-Kutta method satisfies the order conditions from *Theorem 10* up to order p. Then the Runge-Kutta method is convergent of order p.

Proof. We only sketch the proof. First, the method has consistency order p thanks to the fullfilment of the order condition, *Theorem 10*. Thus, we only need to show that the increment function Φ satisfies a Lipschitz condition. This can be achieved by employing a similar "bootstrapping" argument we used when proved that the increment function associated with the Heun's method satisfies a Lipschitz condition.

4.5 Numerical solution of ordinary differential equations: Error estimation and step size control

As always, we start by import some important Python modules.

```
import numpy as np
from numpy import pi
from numpy.linalg import solve, norm
import matplotlib.pyplot as plt

# Do a pretty print of the tables using panda
import pandas as pd
from IPython.display import display

# Use a funny plotting style
# plt.xkcd()
newparams = {'figure.figsize': (6.0, 6.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 14}
plt.rcParams.update(newparams)
```

This goal of this section is to develop Runge Kutta methods with automatic adaptive time-step selection.

Adaptive time-step selection aims to dynamically adjust the step size during the numerical integration process to balance accuracy and computational efficiency. By increasing the step size when the solution varies slowly and decreasing it when the solution changes rapidly, adaptive methods ensure that the local error remains within a specified tolerance. This approach not only enhances the precision of the solution but also optimizes the computational resources, making it particularly valuable for solving complex and stiff ODEs where fixed step sizes may either fail to capture important dynamics or result in unnecessary computations.

In this notebook, we will again focus **explicit** Runge-Kutta methods

TODO

Add solution of three-body problem as an motivational example. Will be done after the submission of project 2 as project 2 requires to implement an adaptive RK method from scratch.

4.5.1 Error estimation

Given two methods, one of order p and the other of order $p+1$ or higher. Assume we have reached a point $(t_n, \mathbf{y}_n)$. One step forward with each of these methods can be written as

$$\begin{aligned} \mathbf{y}_{n+1} &= \mathbf{y}_n + \tau\Phi(t_n, \mathbf{y}_n; \tau), && \text{order } p, \\ \hat{\mathbf{y}}_{n+1} &= \mathbf{y}_n + \tau\widehat{\Phi}(t_n, \mathbf{y}_n; \tau), && \text{order } \hat{p} = p+1 \text{ or more.} \end{aligned}$$

Let $\mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n)$ be the exact solution of the ODE through $(t_n, \mathbf{y}_n)$. We would like to find an estimate for **consistency error** or **the local error** $\mathbf{l}_{n+1}$, that is, the error in one step starting from $(t_n, \mathbf{y}_n)$,

$$\mathbf{l}_{n+1} = \mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n) - \mathbf{y}_{n+1}.$$

As we have already seen, the local error is determined by finding the power series in τ for both the exact and numerical solutions. The local error is of order p if the lowest order terms in the series, where the exact and numerical solutions differ, are of order $p+1$. Therefore, the local errors of the two methods are:

$$\begin{aligned} \mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n) - \mathbf{y}_{n+1} &= \Psi(t_n, y_n)\tau^{p+1} + \dots, \\ \mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n) - \hat{\mathbf{y}}_{n+1} &= \widehat{\Psi}(t_n, y_n)\tau^{p+2} + \dots, \end{aligned}$$

where $\Psi(t_n, y_n)$ is a term consisting of method parameters and differentials of $\mathbf{f}$ and ... contains all the terms of the series of order $p+2$ or higher. Taking the difference gives

$$\hat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1} = \Psi(t_n, \mathbf{y}_n)\tau^{p+1} + \dots.$$

Assume now that τ is small, such that the *principal error term* $\Psi(t_n, y_n)\tau^{p+1}$ dominates the error series. Then a reasonable approximation to the unknown local error $\mathbf{l}_{n+1}$ is the *local error estimate* $\mathbf{le}_{n+1}$:

$$\mathbf{le}_{n+1} = \hat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1} \approx \mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n) - \mathbf{y}_{n+1}.$$

4.5.2 Stepsize control

The next step is to control the local error, that is, choose the step size so that $\|\mathbf{le}_{n+1}\| \leq \text{Tol}$ for some given tolerance Tol, and for some chosen norm $\|\cdot\|$.

Essentially: Given $t_n, \mathbf{y}_n$ and a step size τ_n.

- Do one step with the method of choice, and find an error estimate $\mathbf{le}_{n+1}$.
- if $\|\mathbf{le}\|_{n+1} < \text{Tol}$
  - Accept the solution $t_{n+1}, \mathbf{y}_{n+1}$.
  - If possible, increase the step size for the next step.
- else
  - Repeat the step from $(t_n, \mathbf{y}_n)$ with a reduced step size τ_n.

In both cases, the step size will change. But how? From the discussion above, we have that

$$\|\mathbf{le}_{n+1}\| \approx D\tau_n^{p+1}.$$

where $\mathbf{le}_{n+1}$ is the error estimate we can compute, D is some unknown quantity, which we assume almost constant from one step to the next.

What we want is a step size τ_{new} such that

$$\text{Tol} \approx D\tau_{new}^{p+1}.$$

From these two approximations we get:

$$\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|} \approx \left(\frac{\tau_{new}}{\tau_n}\right)^{p+1} \qquad \Rightarrow \qquad \tau_{new} \approx \left(\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|}\right)^{\frac{1}{p+1}} \tau_n.$$

That is, if the current step τ_n was rejected, we try a new step τ_{new} with this approximation. However, it is still possible that this new step will be rejected as well. To avoid too many rejected steps, it is therefore common to be a bit restrictive when choosing the new step size, so the following is used in practice:

$$\tau_{new} = P \cdot \left(\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|}\right)^{\frac{1}{p+1}} \tau_n.$$

where the *pessimist factor* $P < 1$ is some constant, normally chosen between 0.5 and 0.95.

4.5.3 Implementation

We have all the bits and pieces for constructing an adaptive ODE solver based on Euler's and Heuns's methods. There are still some practical aspects to consider:

- The combination of the two methods, can be written as

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(t_n, \mathbf{y}_n), \\
\mathbf{k}_2 &= \mathbf{f}(t_n + \tau, \mathbf{y}_n + \tau\mathbf{k}_1), \\
\mathbf{y}_{n+1} &= \mathbf{y}_n + \tau\mathbf{k}_1, && \text{Euler} \\
\hat{\mathbf{y}}_{n+1} &= \mathbf{y}_n + \frac{\tau}{2}(\mathbf{k}_1 + \mathbf{k}_2), && \text{Heun} \\
\mathbf{le}_{n+1} &= \|\hat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1}\| = \frac{\tau}{2}\|\mathbf{k}_2 - \mathbf{k}_1\|.
\end{aligned}$$

- Even if the error estimate is derived for the lower order method, in this case Euler's method, it is common to advance the solution with the higher order method, since the additional accuracy is for free.
- Adjust the last step to be able to terminate the solutions exactly in T.
- To avoid infinite loops, add some stopping criteria. In the code below, there is a maximum number of allowed steps (rejected or accepted).

A popular class of Runge - Kutta methods with an error estimate consists of so-called **embedded Runge - Kutta methods** or **Runge - Kutta pairs**, and the coefficients can be written in a Butcher tableau as follows

c_1	a_{11}	a_{12}	$\cdots$	a_{1s}	
c_2	a_{21}	a_{22}	$\cdots$	a_{2s}	
$\vdots$	$\vdots$			$\vdots$	
c_s	a_{s1}	a_{s2}	$\cdots$	a_{ss}	
	b_1	b_2	$\cdots$	b_s	Order p
	$\widehat{b}_1$	$\widehat{b}_2$	$\cdots$	$\widehat{b}_s$	Order $\widehat{p} = p + 1$

A major advantage of such embedded RKMs is that we need to compute the the s stage derivatives k_i **only once** and can use them for **both RKM**! Remember that stage derivatives can be expensive to compute.

The order difference between the two different methods is soley determine by the use of different weights $\{b_i\}_{i=1}^s$ and $\{\widehat{b}_i\}_{i=1}^s$.

Since

- $\mathbf{y}_{n+1} = \mathbf{y}_n + \tau_n \sum_{i=1}^{s} b_i \mathbf{k}_i$
- $\hat{\mathbf{y}}_{n+1} = \mathbf{y}_n + \tau_n \sum_{i=1}^{s} \hat{b}_i \mathbf{k}_i$

the error estimate is simply given by

$$\mathbf{le}_{n+1} = \tau_n \sum_{i=1}^{s} (\hat{b}_i - b_i)\mathbf{k}_i.$$

Recalling Euler and Heun,

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Euler Heun

and the Heun-Euler pair can be written as

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

A particular mention deserves also the classical *4-stage Runge-Kutta method* from a previous notebook, which can be written as

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

See this list of Runge - Kutta methods for more. For the last one there exist also a embedded Runge-Kutta 4(3) variant due to **Fehlberg**:

$$\begin{array}{c|ccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{6} \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & 0 \end{array}$$

Outlook. In your homework/project assignment, you will be asked to implement automatic adaptive time step selection based on embedded Runge-Kutta methods. You can either develop those from scratch or start from the `ExplicitRungeKutta` class we presented earlier and incorporate code for error estimation and time step selection.

4.6 Numerical solution of ordinary differential equations: Stiff problems

And of course we want to import the required modules.

```
# import matplotlib.font_manager
# matplotlib.font_manager.findfont("Humor Sans")
```

4.6.1 Explicit Euler method and a stiff problem

We start by taking a second look at the IVP

$$y'(t) = \lambda y(t), \quad y(t_0) = y_0. \tag{4.26}$$

with the analytical solution

$$y(t) = y_0 e^{\lambda(t-t_0)}. \tag{4.27}$$

Recall that for $\lambda > 0$ this equation can present a simple model for the growth of some population, while a negative $\lambda < 0$ typically appears in decaying processes (read "negative growth").

So far we have only solved *((21))* numerically for $\lambda > 0$. Let's start with a little experiment. First, we set $y_0 = 1$ and $t_0 = 0$. Next, we chose different λ to model processes with various decay rates, let's say

$$\lambda \in \{-10, -50, -250\}.$$

For each of those λ, we set a reference step length

$$\tau_\lambda = \frac{2}{|\lambda|}$$

(we will soon see why!) and compute a numerical solution using the explict Euler method for three different time steps, namely for $\tau \in \{0.1\tau_\lambda, \tau_\lambda, 1.1\tau_\lambda\}$ and plot the numerical solution together with the exact solution.

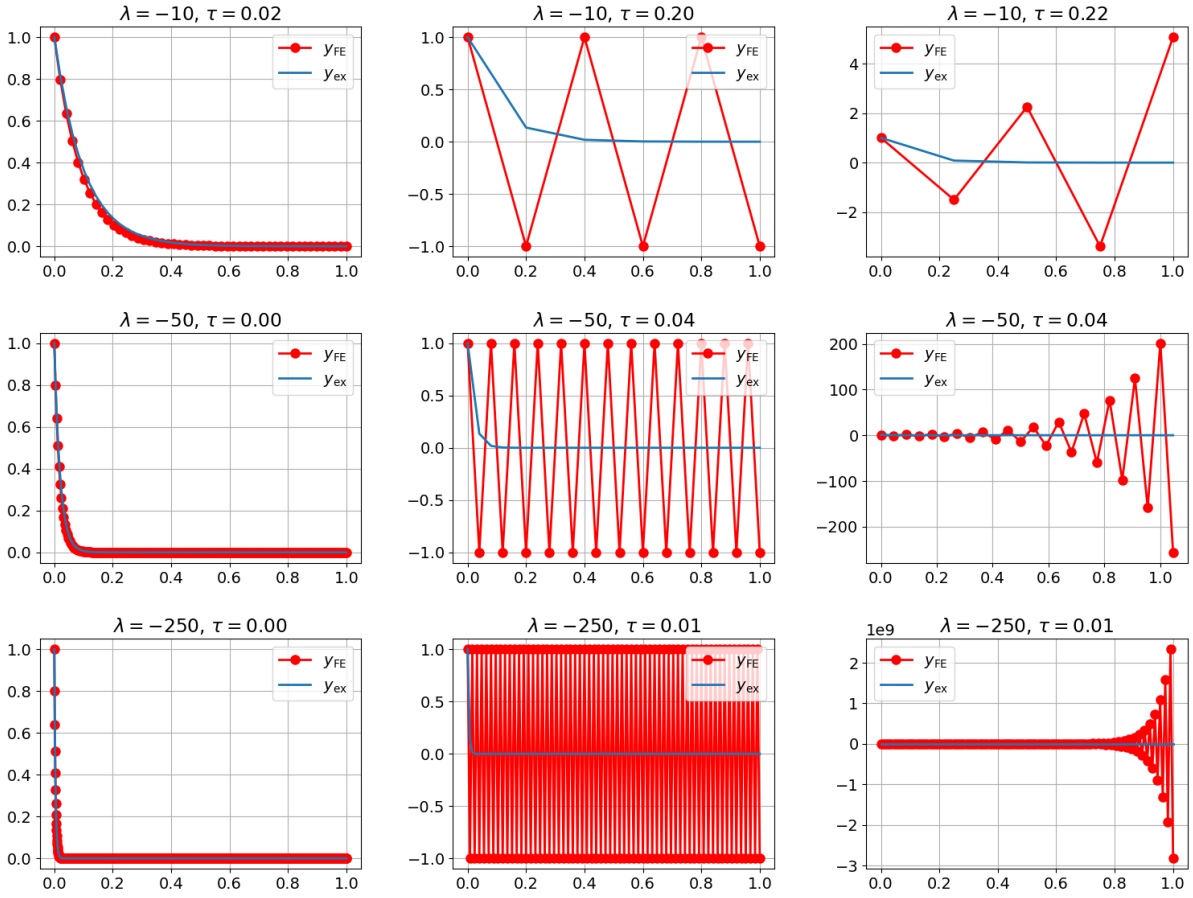
```
def explicit_euler(y0, t0, T, f, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        ys.append(y + dt*f(t, y))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

```
plt.rcParams['figure.figsize'] = (16.0, 12.0)
t0, T = 0, 1
y0 = 1
lams = [-10, -50, -250]

fig, axes = plt.subplots(3,3)
fig.tight_layout(pad=3.0)

for i in range(len(lams)):
    lam = lams[i]
    tau_l = 2/abs(lam)
    taus = [0.1*tau_l, tau_l, 1.1*tau_l]

    # rhs of IVP
    f = lambda t,y: lam*y

    # Exact solution to compare against
    y_ex = lambda t: y0*np.exp(lam*(t-t0))

    # Compute solution for different time step size
    for j in range(len(taus)):
        tau = taus[j]
        Nmax = int(1/tau)
```

(continues on next page)

(continued from previous page)

```
        ts, ys = explicit_euler(y0, t0, T, f, Nmax)
        ys_ex = y_ex(ts)
        axes[i,j].set_title(f"$\\lambda = {lam}$, $\\tau = {tau:0.2f}$")
        axes[i,j].plot(ts, ys, "ro-")
        axes[i,j].plot(ts, ys_ex)
        axes[i,j].legend([r"$y_{\mathrm{FE}}$", "$y_{\\mathrm{ex}}$"])
```



Observation 7

Looking at the first column of the plot, we observe a couple of things. First, the numerical solutions computed with a time step $\tau = 0.1\tau_\lambda$ closely resembles the exact solution. Second, the exact solution approaches for larger t a stationary solution (namely 0), which does not change significantly over time. Third, as expected, the exact solution decays the faster the larger the absolute value of λ is. In particular for $\lambda = -250$, the exact solution y_{ex} drops from $y_{\text{ex}}(0) = 1$ to $y_{\text{ex}}(0.05) \approx 3.7 \cdot 10^{-6}$ at $t = 0.05$, and at $t = 0.13$, the exact solution is practically indistinguishable from 0 as $y_{\text{ex}}(0.13) \approx 7.7 \cdot 10^{-16}$.

Looking at the second column, we observe that a time-step size $\tau = \tau_\lambda$, the numerical solution oscillates between -1 and 1, and thus the numerical solution does not resemble at all the monotonic and rapid decrease of the exact solution. The situation gets even worse for a time-step size $\tau > \tau_\lambda$ (third column) where the the numerical solution growths exponentially (in absolute values) instead of decaying exponentially as the y_{ex} does.

So what is happening here? Why is the explicit Euler method behaving so strangely? Having a closer look at the compu-

tation of a single step in Euler's method for this particular test problem, we see that

$$y_{i+1} = y_i + \tau f(t_i, y_i) = y_i + \tau\lambda y_i = (1+\tau\lambda)y_i = (1+\tau\lambda)^2 y_{i-1} = ... = (1+\tau\lambda)^{i+1} y_0$$

Thus, for this particular IVP, the next step y_{i+1} is simply computed by by multiplying the current value y_i with the the function $(1+\tau\lambda)$.

$$y_{i+1} = R(z)^{i+1} y_0, \quad z = \tau\lambda$$

where $R(z) = (1+z)$ is called the **stability function** of the explicit Euler method.

Now we can understand what is happening. Since $\lambda < 0$ and $\tau > 0$, we see that as long as $\tau\lambda > -2 \Leftrightarrow \tau < \dfrac{2}{|\lambda|}$, we have that $|1+\tau\lambda| < 1$ and therefore, $|y_i| = |1+\tau\lambda|^{i+1} y_0$ is decreasing and converging to 0 For $\tau = \dfrac{2}{|\lambda|} = \tau_\lambda$, we obtain

$$y_{i+1} = (1+\tau\lambda)^{i+1} y_0 = (-1)^{i+1} y_0$$

so the numerical solution will be jump between -1 and 1, exactly as observed in the numerical experiment. Finally, for $\tau > \dfrac{2}{|\lambda|} = \tau_\lambda$, $|1+\tau\lambda| > 1$, and $|y_{i+1}| = |1+\tau\lambda|^{i+1} y_0$ is growing exponentially.

Note, that is line of thoughts hold independent of the initial value y_0. So even if we just want to solve our test problem *((21)) away* from the transition zone where y_{ex} drops from 1 to almost 0, we need to apply a time-step $\tau < \tau_\lambda$ to avoid that Euler's method produces a completely wrong solution which exhibits exponential growth instead of exponential decay.

Summary

For the IVP problem stiff:ode:eq:exponential, Euler's method has to obey a time step restriction $\tau < \dfrac{2}{|\lambda|}$ to avoid numerical instabilities in the form of exponential growth.

This time restriction becomes more severe the larger the absolute value of $\lambda < 0$ is. On the other hand, the larger the absolute value of $\lambda < 0$ is, the faster the actual solution approaches the stationary solution 0. Thus it would be reseaonable to use large time-steps when the solution is close to the stationary solution. Nevertheless, because of the time-step restriction and stability issues, we are forced to use very small time-steps, despite the fact that the exact solution is not changing very much. This is a typical characteristic of a **stiff problem**. So the IVP problem stiff:ode:eq:exponential gets "stiffer" the larger the absolute value of $\lambda < 0$ is, resulting in a severe time step restriction $\tau < \dfrac{2}{|\lambda|}$ to avoid numerical instabilities.

Outlook. Next, we will consider other one-step methods and investigate how they behave when applied to the test problem stiff:ode:eq:exponential. All these one step methods will have a common, that the advancement from y_k to y_{k+1} can be written as

$$y_{k+1} = R(z) y_k \quad \text{with } z = \tau\lambda$$

for some **stability function** $R(z)$.

With our previous analysis in mind we will introduce the following

Definition 9 (Stability domain)

Let $R(z)$ be the stability function for some one-step function. Then the domain

$$\mathcal{S} = \{z \in \mathbb{R} : |R(z)| \leqslant 1\} \tag{4.28}$$

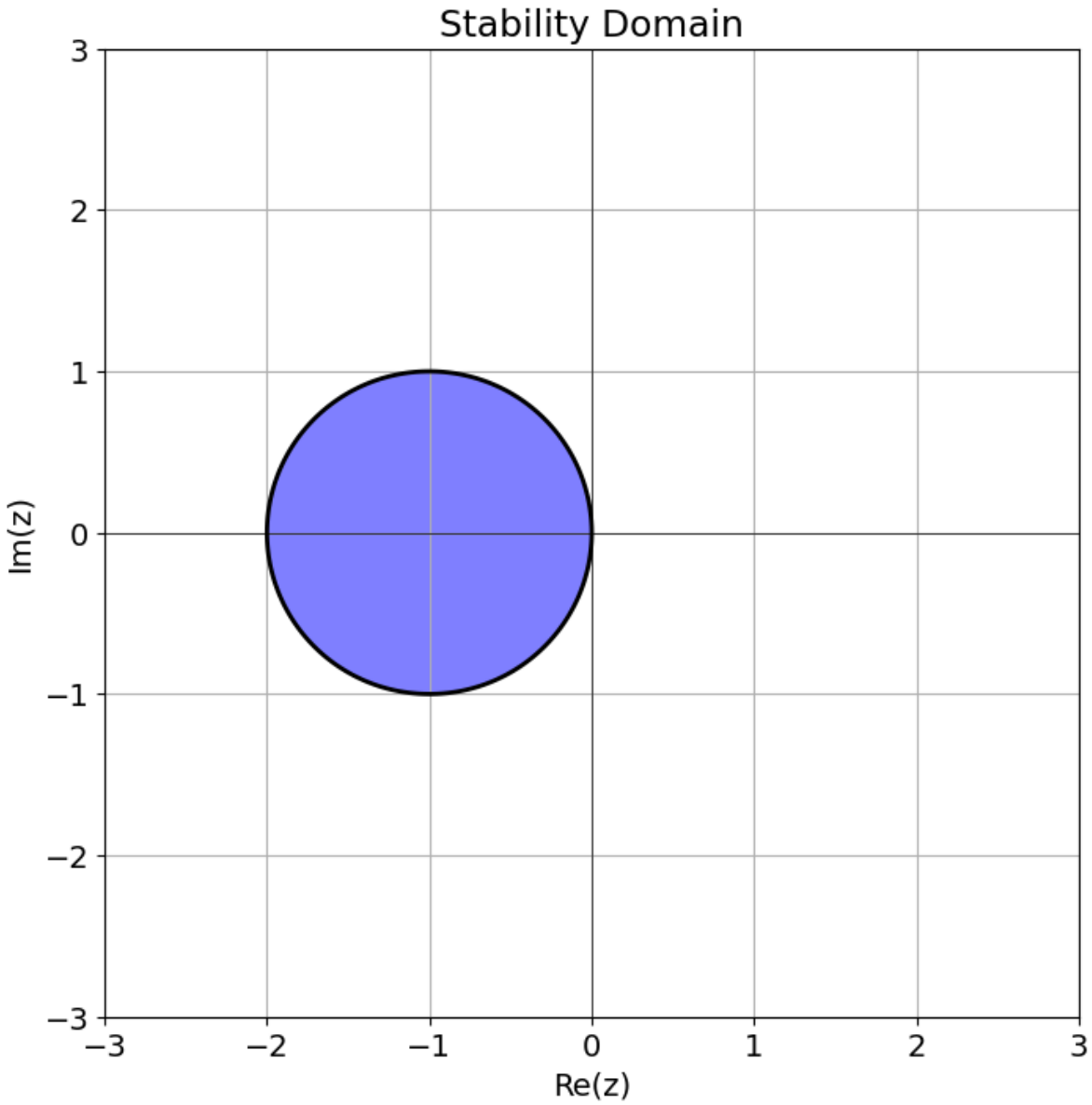
is called the **domain of stability**.

Remark 4

Usually, one consider the entire complex plane in the definition of the domain of stability, that is, $\mathcal{S} = \{z \in \mathbb{C} : |R(z)| \leqslant 1\}$ but in this course we can restrict ourselves to only insert real arguments in the stability function.

Let's plot the domain of stability for the explicit Euler method.

```
def r_fe(z):
    return 1 + z

plot_stability_domain(r_fe)
```



Important

Time-step restrictions for explicit RKM

Unfortunately, all **explicit Runge-Kutta methods** when applied to the simple test problem (4.26) will suffer from similar problems as the explicit Euler method, for the following reason:

It can be shown that for any **explicit** RKM, its corresponding stability function $r(z)$ must be a polynomial in z. Since complex polynomials satisfy $|r(z)| \to \infty$ for $|z| \to \infty$, its **domain of stability** as defined above must be bounded. Consequently, there will a constant C such that any time step $\tau > \dfrac{C}{|\lambda|}$ will lead to numerical instabilities.

4.6.2 The implicit Euler method

Previously, we considered Euler's method, for the first-order IVP

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0$$

where the new approximation y_{k+1} at t_{k+1} is defined by

$$y_{k+1} := y_k + \tau f(t_k, y_k)$$

We saw that this could be interpreted as replacing the differential quotient y' by a **forward difference quotient**

$$f(t_k, y_k) = y'(t_k) \approx \frac{y(t_{k+1}) - y(t_k)}{\tau}$$

Here the term "forward" refers to the fact that we use a forward value $y(t_{k+1})$ at t_{k+1} to approximate the differential quotient at t_k.

Now we consider a variant of Euler's method, known as the **implicit** or **backward** Euler method. This time, we simply replace the differential quotient y' by a **backward difference quotient**

$$f(t_k, y_k) = y'(t_k) \approx \frac{y(t_k) - y(t_{k-1})}{\tau}$$

resulting in the following

Algorithm 3 (Implicit/backward Euler method)

Given a function $f(t, y)$ and an initial value (t_0, y_0).

- Set $t = t_0$, choose τ.
- `while` $t < T$:
  - $y_{k+1} := y_k + \tau f(t_{k+1}, y_{k+1})$
  - $t_{k+1} := t_k + \tau$
  - $t := t_{k+1}$

Note that in contrast to the explicit/forward Euler, the new value of y_{k+1} is only *implicitly* defined as it appears both on the left-hand side and right-hand side. Generally, if f is nonlinear in its y argument, this amounts to solve a non-linear equation, e.g., by using fix-point iterations or Newton's method. But if f is linear in y, that we only need to solve a *linear system*.

Let's see what we get if we apply the backward Euler method to our model problem.

Exercise 25 (Implicit/backward Euler method)

a) Show that the backward difference operator (and therefore the backward Euler method) has consistency order 1, that is,

$$y(t) + \tau f(t+\tau, y(t+\tau)) - y(t+\tau) = \mathcal{O}(\tau^2)$$

b) Implement the implicit/backward Euler method

```
def implicit_euler(y0, t0, T, lam, Nmax):
  ...
```

for the IVP *((21))*. Note that we now take λ as a parameter, and not a general function f as we want to keep as simple as possible Otherwise we need to implement a nonlinear solver if we allow for arbitrary right-hand sides f. You use the code for `explicit_euler` as a start point.

c) Write down the Butcher table for the implicit Euler method.

d) Rerun the numerical experiment from the previous section with the implicit Euler method. Do you observe any instabilities?

e) Find the stability function $R(z)$ for the implicit Euler satisfying

$$y_{k+1} = R(\tau\lambda) y_k \tag{4.29}$$

and use it to explain the much better behavior of the implicit Euler when solving the initial value problem (4.26).

Solution to Exercise 25 (Implicit/backward Euler method)

a) As before, we simply do a Taylor expansion of y, but this time around $t+\tau$. Then

$$y(t) = y(t+\tau) - \tau y'(t+\tau) + \mathcal{O}(\tau^2) = y(t+\tau) - \tau f(t+\tau, y(t+\tau)) + \mathcal{O}(\tau^2)$$

which after rearranging terms is exactly *(4)*.

b)

```
# Warning, implicit Euler is only implement for the test equation
# not a general f!
def implicit_euler(y0, t0, T, lam, Nmax):
    ys = [y0]
    ts = [t0]
    dt = (T - t0)/Nmax
    while(ts[-1] < T):
        t, y = ts[-1], ys[-1]
        ys.append(y/(1-dt*lam))
        ts.append(t + dt)
    return (np.array(ts), np.array(ys))
```

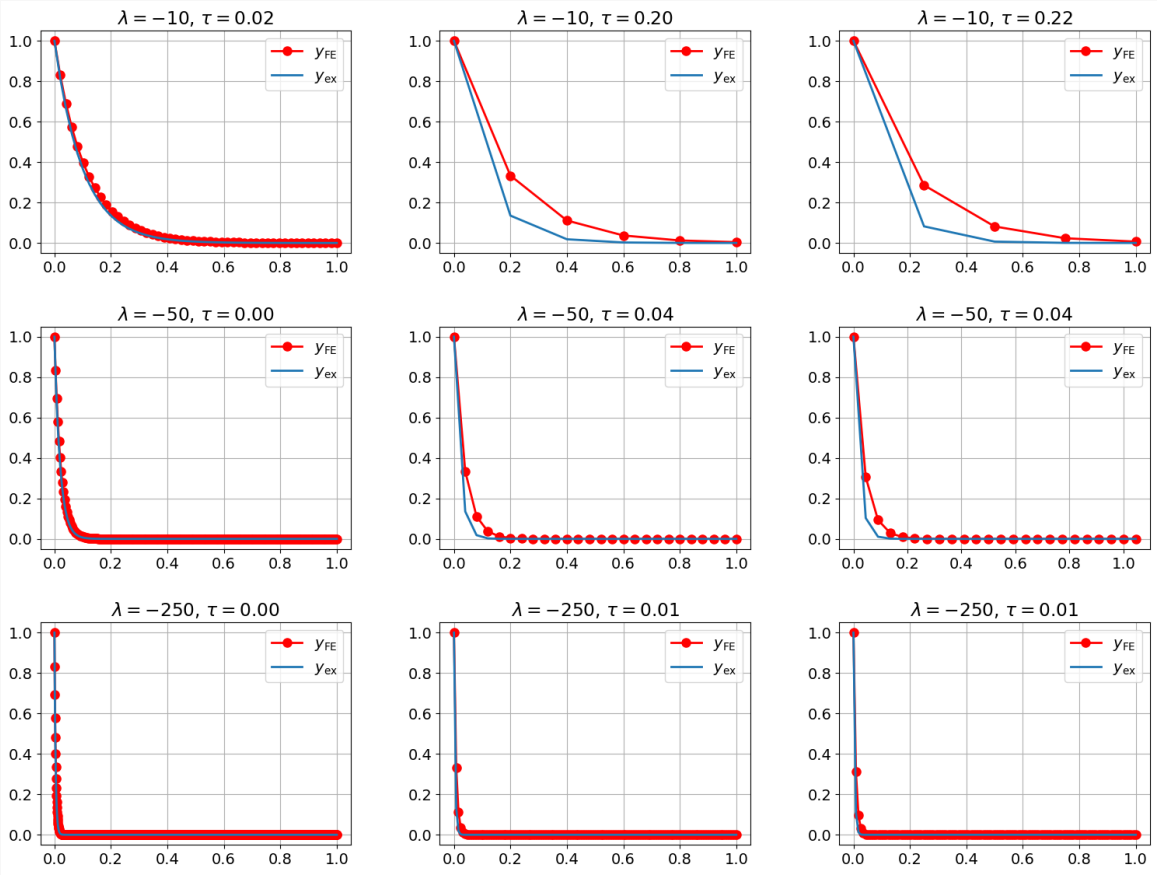
c)

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

d)

```
plt.rcParams['figure.figsize'] = (16.0, 12.0)
t0, T = 0, 1
y0 = 1
```

```
lams = [-10, -50, -250]

fig, axes = plt.subplots(3,3)
fig.tight_layout(pad=3.0)

for i in range(len(lams)):
    lam = lams[i]
    tau_l = 2/abs(lam)
    taus = [0.1*tau_l, tau_l, 1.1*tau_l]

    # rhs of IVP
    f = lambda t,y: lam*y

    # Exact solution to compare against
    y_ex = lambda t: y0*np.exp(lam*(t-t0))

    # Compute solution for different time step size
    for j in range(len(taus)):
        tau = taus[j]
        Nmax = int(1/tau)
        ts, ys = implicit_euler(y0, t0, T, lam, Nmax)
        ys_ex = y_ex(ts)
        axes[i,j].set_title(f"$\\lambda = {lam}$, $\\tau = {tau:0.2f}$")
        axes[i,j].plot(ts, ys, "ro-")
        axes[i,j].plot(ts, ys_ex)
        axes[i,j].legend([r"$y_{\mathrm{FE}}$", "$y_{\\mathrm{ex}}$"])
```



e) For $y' = \lambda y =: f(t, y)$, the implicit Euler gives

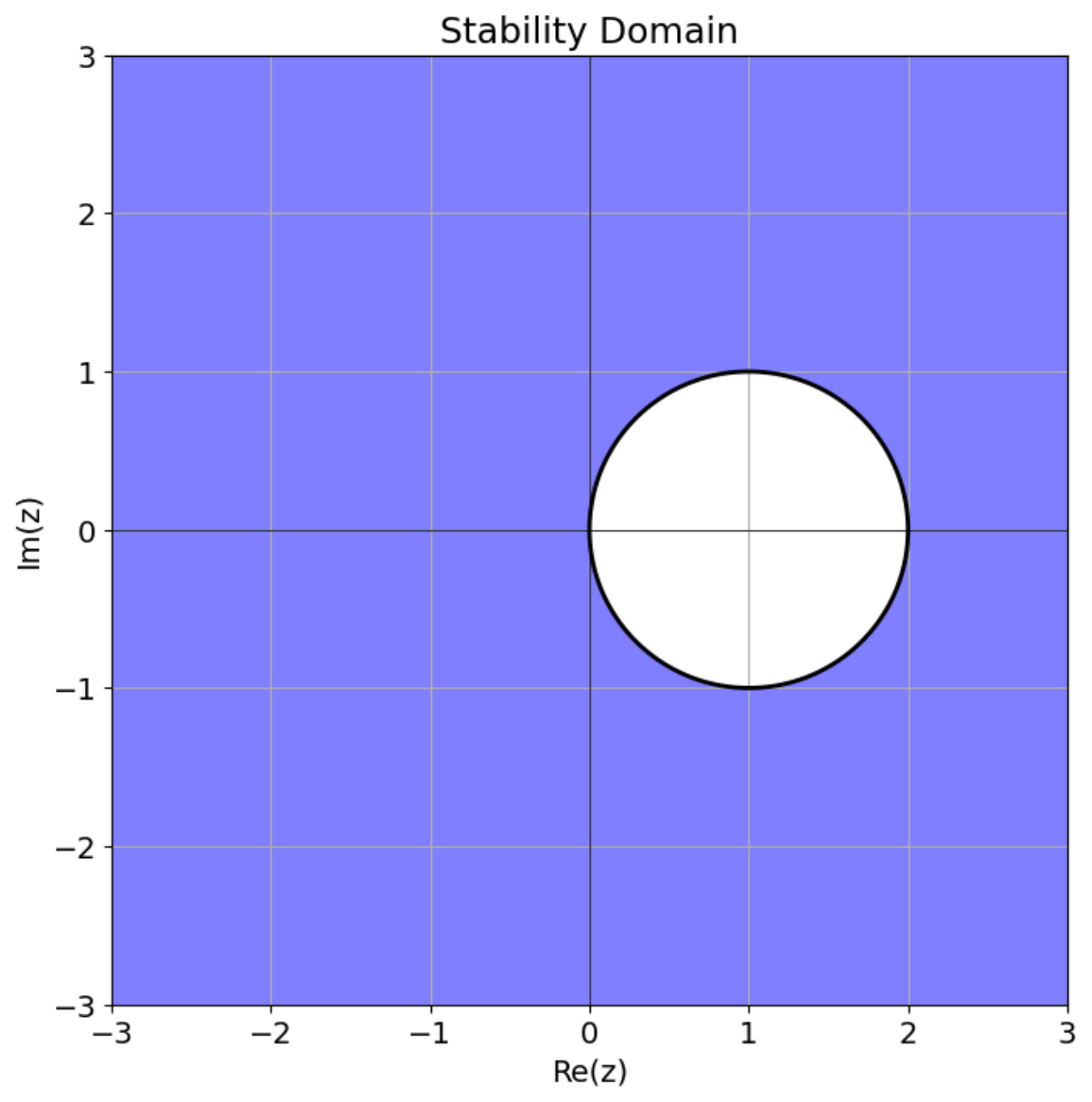
$$y_{k+1} = y_k + \tau\lambda y_{k+1} \tag{4.30}$$

$$\Leftrightarrow \; y_{k+1} = \frac{1}{1-\tau\lambda} y_k = \left(\frac{1}{1-\tau\lambda}\right)^{k+1} y_0.$$

Thus $R(z) = \frac{1}{1-z}$. The domain of stability is $\mathcal{S} = (-\infty, 0] \cup [2, \infty)$, in particular, no matter how we chose τ, $|R(\lambda z)| < 1$ for $\lambda < 0$. So the implicit Euler method is stable for the test problem *((21))*, independent of the choice of the time step.

We can even plot it:

```
def r_fe(z):
    return 1/(1 - z)

plot_stability_domain(r_fe)
```



4.6.3 The Crank-Nicolson

Both the explicit/forward and the implicit/backward Euler method have consistency order 1. Next we derive 2nd order method. We start exactly as in the derivation of Heun's method presented in the `IntroductionNuMeODE.ipynb` notebook.

Again, we start from the *exact integral representation*, and apply the trapezoidal rule

$$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} f(t, y(t))\,\mathrm{d}t \approx \frac{\tau_k}{2}\left(f(t_{k+1}, y(t_{k+1})) + f(t_k, y(t_k))\right)$$

This suggest to consider the *implicit* scheme

$$y_{k+1} = y_k + \frac{\tau_k}{2}\left(f(t_{k+1}, y_{k+1}) + f(t_k, y_k)\right)$$

which is known as the **Crank-Nicolson method**.

Exercise 26 (Investigating the Crank-Nicolson method)

a) Determine the Butcher table for the Crank-Nicolson method.

Solution. We can rewrite Crank-Nicolson using two stage-derivatives k_1 and k_2 as follows.

$$\begin{aligned} k_1 &= f(t_k, y_k) = f(t_k + 0\cdot\tau, y_k + \tau(0\cdot k_1 + 0\cdot k_2)) \\ k_2 &= f(t_{k+1}, y_{k+1}) = f(t_k + 1\cdot\tau, y_k + \tau(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2)) \\ y_{k+1} &= y_k + \tau(\tfrac{1}{2}k_1 + \tfrac{1}{2}k_2) \end{aligned}$$

and thus the Butcher table is given by

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}.$$

b) Use the order conditions discussed in the `RungeKuttaNuMeODE.ipynb` to show that Crank-Nicolson is of consistency/convergence order 2.

c) Determine the stability function $R(z)$ associated with the Crank-Nicolson method and discuss the implications on the stability of the method for the test problem *((21))*.

Solution. With $f(t, y) = \lambda y$,

$$y_{k+1} = y_k + \frac{\tau}{2}\lambda y_{k+1} + \frac{\tau}{2}\lambda y_k$$

and thus

$$y_{k+1} = \frac{1 + \frac{\tau\lambda}{2}}{1 - \frac{\tau\lambda}{2}} y_k$$

and therefore

$$R(z) = \frac{1 + \frac{z}{2}}{1 - \frac{z}{2}}.$$

As result, the stability domain $(-\infty, 0] \subset \mathcal{S}$, in particular, Crank-Nicolson is stable for our test problem, independent of the choice of the time-step.

d) Implement the Crank-Nicolson method to solve the test problem (stiff:ode:eq:exponential) numerically.

Hint. You can start from `implicit_euler` function implemented earlier, you only need to change a single line.

e) Check the convergence rate for your implementation by solving (stiff:ode:eq:exponential) with $\lambda = 2, t_0 = 1, T = 2$ and $y_0 = 1$ for various time step sizes and compute the corresponding experimental order of convergence (EOC)

f) Finally, rerun the stability experiment from the section *Explicit Euler method and a stiff problem* with Crank-Nicolson.

4.7 A modelling/simulation mini-project: The SIR model and some extensions

In this somewhat longer exercise/mini project we take a closer look at the SIR model briefly introduced in *Example 11*, how to modify the model to account for e.g. hospitalized patients of time-limited immunity, and how to solve the resulting models numerically using Runge-Kutta methods.

4.7.1 SIR model

Recall that the SIR model is given by

$$S' = -\beta SI \tag{4.31}$$
$$I' = \beta SI - \gamma I \tag{4.32}$$
$$R' = \gamma I, \tag{4.33}$$

where

- S(t): porpotion of individuals **susceptible** for infection,
- I(t): porpotion of **infected** individuals, capable of transmitting the disease,
- R(t): porpotion of **removed** individuals who cannot be infected due death or to immunity
- β: the **infection rate**, and
- γ: the **removal rate**.

a) Show that the total number of induviduals $N(t)$ is constant with respect to time

Solution

We see that:

$$N'(t) = S'(t) + I'(t) + R'(t) = 0 \tag{4.34}$$

b) Looking at

$$I' = \beta SI - \gamma I$$

we see that an outbreak of a disease (increase of number of infections) can only occur if $I'(0) > 0$, equivalent to

$$0 < \beta S(0)I(0) - \gamma I(0) \Leftrightarrow \underbrace{\frac{\beta}{\gamma}}_{:=R_0} S(0) > 1$$

Use Heuns Method to solve the SIR model . Assume we are modelling Trondheim with 200000 habitants, and start with one infected induvidual, and set $\gamma = 1/18$. Test with different values for R_0 and see what the result is.

Solution

```
%matplotlib widget
# %matplotlib inline

import ipywidgets as widgets
from ipywidgets import interact, fixed
import numpy as np
from numpy import pi
from numpy.linalg import solve, norm
import matplotlib.pyplot as plt
```

```
#same as lecture notes
class ExplicitRungeKutta:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __call__(self, y0, t0, T, f, Nmax):
        # Extract Butcher table
        a, b, c = self.a, self.b, self.c

        # Stages
        s = len(b)
        ks = [np.zeros_like(y0, dtype=np.double) for s in range(s)]

        # Start time-stepping
        ys = [y0]
        ts = [t0]
        dt = (T - t0)/Nmax

        while(ts[-1] < T):
            t, y = ts[-1], ys[-1]

            # Compute stages derivatives k_j
            for j in range(s):
                t_j = t + c[j]*dt
                dY_j = np.zeros_like(y, dtype=np.double)
                for l in range(j):
                    dY_j += dt*a[j,l]*ks[l]

                ks[j] = f(t_j, y + dY_j)

            # Compute next time-step
            dy = np.zeros_like(y, dtype=np.double)
            for j in range(s):
                dy += dt*b[j]*ks[j]

            ys.append(y + dy)
            ts.append(t + dt)

        return (np.array(ts), np.array(ys))

a    = np.array([[0,   0],
                 [1., 0]])
b = np.array([1/2, 1/2.])
c = np.array([0., 1])
```

(continues on next page)

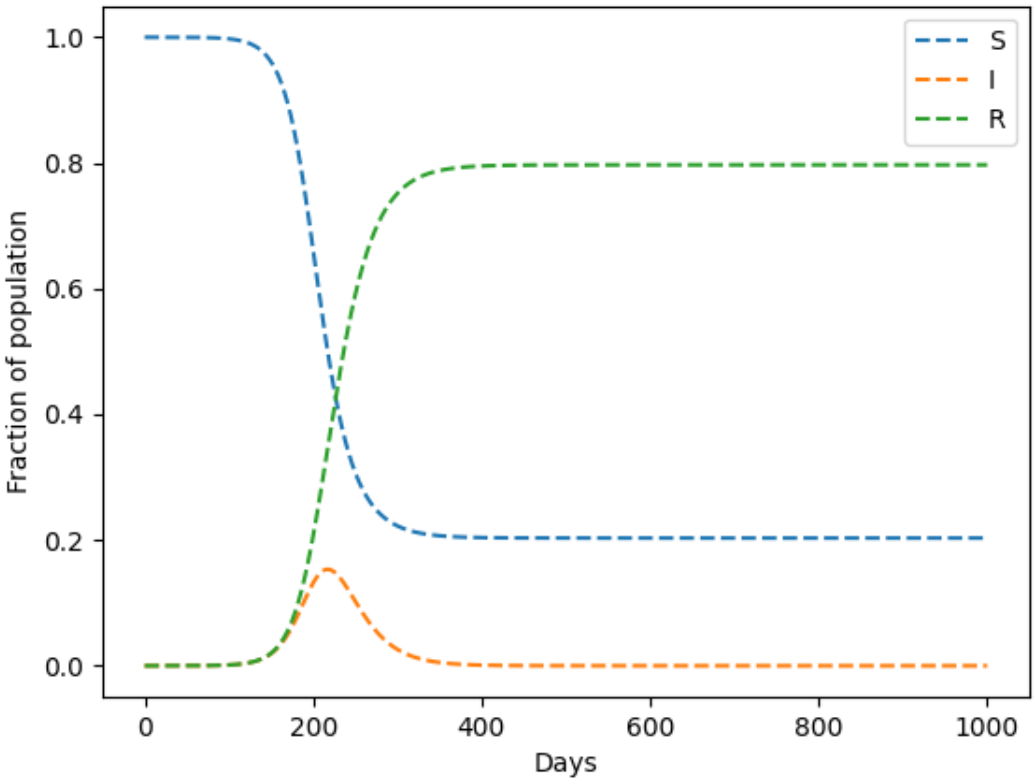
(continued from previous page)

```
heun = ExplicitRungeKutta(a, b, c)
```

```
class SIR:
    def __init__(self, beta, gamma):
        self.beta = beta # infectional rate
        self.gamma = gamma # removal rate

    def __call__(self, t, y):
        return np.array([-self.beta*y[0]*y[1],
                         self.beta*y[0]*y[1] - self.gamma*y[1],
                         self.gamma*y[1]])
```

```
# Data for the SIR model
# denote basic reproduction number by r0
r0 = 2
gamma = 1/18.
beta = r0*gamma

# Define a model
sir = SIR(beta=beta, gamma=gamma)

# Trondheim has 200.000 inhabitants we start with 1 infected person
I_0 = 1/200000
S_0 = 1 - I_0
R_0 = 0

y0 = np.array([S_0,
               I_0,
               R_0])

t0, T = 0, 1000 # days, we consider a whole year



# Run method
Nmax = 10000
ts, ys = heun(y0, t0, T, sir, Nmax)
```

```
plt.figure()
plt.plot(ts, ys, "--")
plt.legend(["S", "I", "R"])
plt.xlabel("Days")
plt.ylabel("Fraction of population")
plt.show()
```



4.7.2 SIRH model

We now look at the SIHR model, which also includes the number of hopsitalized induviduals

The SIHR model is given by

$$S' = -\beta SI \tag{4.35}$$
$$I' = \beta SI - \gamma_r I - \gamma_h I \tag{4.36}$$
$$H' = \gamma_h I - \delta H \tag{4.37}$$
$$R' = \gamma_r I + \delta H, \tag{4.38}$$

we assume the same total γ as for the simpler SIR model; that is,

$$\gamma_r + \gamma_h =: \gamma = 1/18$$

Assume that

- we have the same total γ as for the simpler SIR model; that is, $\gamma_r + \gamma_h =: \gamma = 1/18$
- that 3.5% of all infected individuals will be hospitalized which means that $\gamma_h = 0.035\gamma$
- hospitalized individuals stay 14 days in the hospital on average, that is $\delta = 1/14$
- St. Olav's hospital has roughly 1000 beds with roughly 80% of them being occupied

Again startin with one infected induvidual, find what is (approximately) the largest basic reproduction number R_0 for which we will not exceed the maximal number of available beds?

Solution

Note that the solution code provided below uses some more sophisticated Jupyter widgets features, but it is just used as extra interface sugar. We will use a simple slider interface which sets the basic reproduction number R_0 and then automatically updates the solution plots.

```
# define SIHR class similar to SIR before
class SIHR:
    def __init__(self, beta, gamma_r, gamma_h, delta):
        self.beta = beta # infectional rate
        self.gamma_r = gamma_r # removal rate
        self.gamma_h = gamma_h # removal rate
        self.delta = delta

    def __call__(self, t, y):
        return np.array([-self.beta*y[0]*y[1],
                         self.beta*y[0]*y[1] - self.gamma_r*y[1]-self.gamma_h*y[1],
                         self.gamma_h*y[1] - self.delta*y[2],
                         self.gamma_r*y[1]+self.gamma_h*y[2]])

# initial data
# Trondheim has 200.000 inhabitants we start with 1 infected person
N = 2.0e5
I_0 = 1/N
S_0 = 1 - I_0
H_0 = 0
R_0 = 0

y0 = np.array([S_0,
               I_0,
               H_0,
               R_0])
Nyears = 2
t0, T = 0, Nyears*365 # days, we consider 2 years year
```

```
# Prepare plot
Nbeds = 1000
Nfree_beds = 200

# def plot_dynamics_sihr(r0, solver, ax):
def plot_dynamics_sihr(r0, solver, ax):
    # Orginal removal rate from SIR model
    gamma = 1/18.
    beta = r0*gamma
    # We split it into gamma = gamma_r + gamma_h
    # assuming that 3.5 % are hospitalized
    gamma_h =0.035*gamma
    gamma_r = gamma - gamma_h
    # Assume 14 days of hospitilization
    delta = 1./14

    # Define a model for given r0
    sihr = SIHR(beta=beta, gamma_h=gamma_h, gamma_r=gamma_r, delta=delta)

    # Solve
```

(continues on next page)

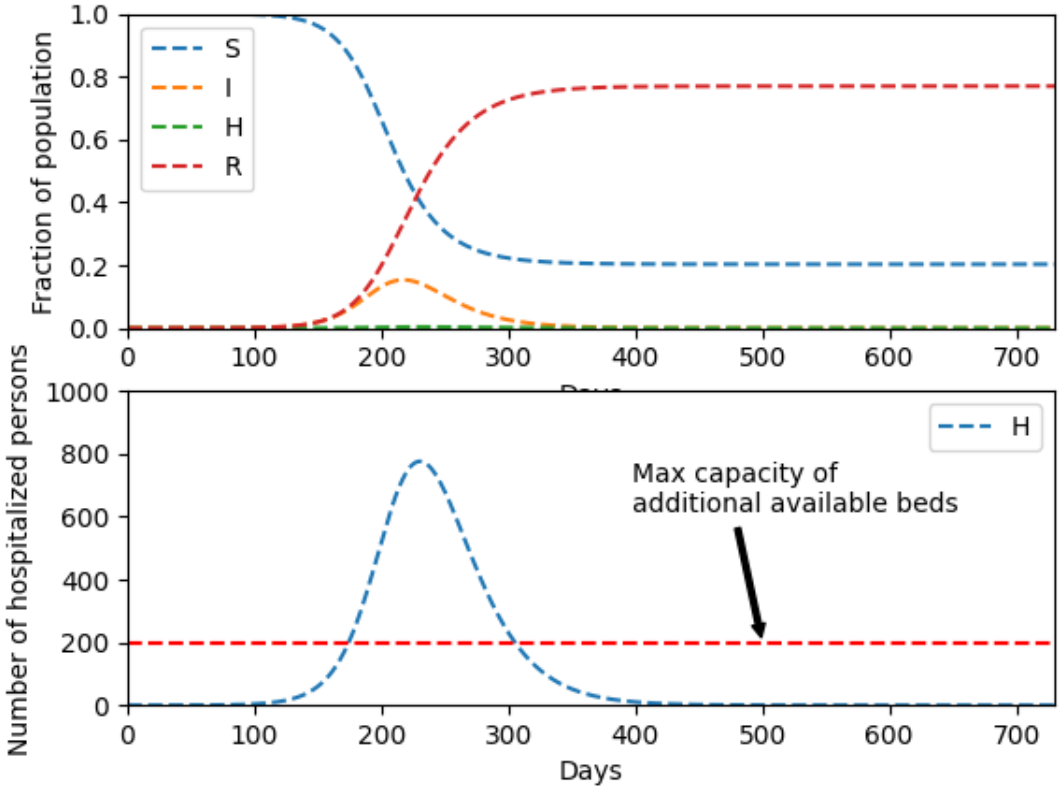
(continued from previous page)

```
    ts, ys = solver(y0, t0, T, sihr, Nmax)
    ax[0].clear()
    ax[0].plot(ts, ys, "--", markersize=3)
    ax[0].legend(["S", "I", "H", "R"])
    ax[0].set_xlim(0,Nyears*365)
    ax[0].set_ylim(0, 1.0)
    ax[0].set_xlabel("Days")
    ax[0].set_ylabel("Fraction of population")

    ax[1].clear()
    ax[1].plot(ts, 2e5*ys[:,2], "--", markersize=3, label="H")
    ax[1].legend()
    ax[1].set_xlabel("Days")
    ax[1].set_ylabel("Number of hospitalized persons")
    ax[1].set_xlim(0,Nyears*365)
    ax[1].set_ylim(0, Nbeds)
    ax[1].hlines([Nfree_beds], t0,T, colors="r", linestyles="dashed")
    ax[1].annotate('Max capacity of \nadditional available beds',
             xy=(500, Nfree_beds), xytext=(-50, 50),
             textcoords="offset points",
             arrowprops=dict(arrowstyle="simple",facecolor='black',
                             relpos=(0.315,0)))
    print(f"Maximum of additional Covid 19 caused hospitalization: {np.abs(ys[:,2]).
↪max()*200000}")
```

```
plt.close()
fig, ax = plt.subplots(2,1)
widgets.interact(plot_dynamics_sihr, r0=(1.0, 3.0, 0.01), solver=fixed(heun),␣
↪ax=fixed(ax))
# widgets.interact(plot_dynamics_sihr, r0=(1.0, 3.0, 0.01), solver=fixed(heun),␣
↪ax=fixed(ax))
# pds = lambda r0 : plot_dynamics_sihr(r0, heun, ax)
# widgets.interact(pds, r0=(1.0, 3.0, 0.01))
```

```
interactive(children=(FloatSlider(value=2.0, description='r0', max=3.0, min=1.0,␣
↪step=0.01), Output()), _dom_c…
```

```
<function __main__.plot_dynamics_sihr(r0, solver, ax)>
```



4.7.3 SIHRt model

Redo part 2, but this time develop and use an extension the SIHR model to account for time-limited immunity, assuming 1 year of immunity for each recovered person. Consider a time-period of 5 years and find out how many "infection waves" will occur where the maximum capacity of beds are exceeded.

Solution.

- For $R_0 = 2$ there are 2 waves, one around Day 250 and one around Day 805

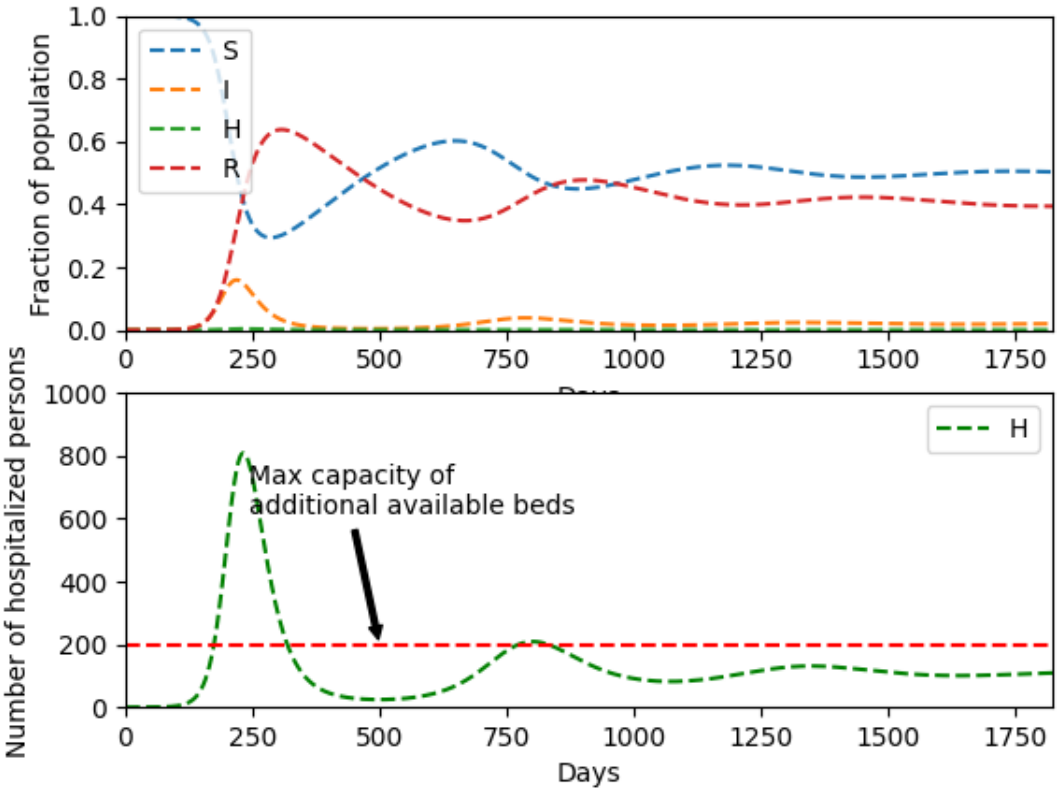
```
class SIHRt:
    def __init__(self, beta, gamma_r, gamma_h, delta, sigma):
        self.beta = beta # infectional rate
        self.gamma_r = gamma_r # removal rate
        self.gamma_h = gamma_h # removal rate
        self.delta = delta
        self.sigma = sigma

    def __call__(self, t, y):
        return np.array([-self.beta*y[0]*y[1]+self.sigma*y[3],
                         self.beta*y[0]*y[1] - self.gamma_r*y[1]-self.gamma_h*y[1],
                         self.gamma_h*y[1] - self.delta*y[2],
                         self.gamma_r*y[1]+self.gamma_h*y[2]-self.sigma*y[3]])
```

```
# Prepare plot
Nbeds = 1000
Nfree_beds = 200

Nyears = 5
t0, T = 0, Nyears*365 # days, we consider 5 years year

def plot_dynamics_sihrt(r0, solver, ax):
    ##### Data for the SIHRt model
    # Orginal removal rate from SIR model
    # We split it into gamma = gamma_r + gamma_h
    # assuming that 3.5 % are hospitalized
    gamma = 1/18.
    beta = r0*gamma
    gamma_h =0.035*gamma
    gamma_r = gamma - gamma_h
    # Assume 14 days of hospitilization
    delta = 1./14
    # One year of immunization
    sigma = 1/365.

    # Define a model for given r0
    sihrt = SIHRt(beta=beta, gamma_h=gamma_h, gamma_r=gamma_r, delta=delta,␣
 ↪sigma=sigma)

    # Solve
    ts, ys = solver(y0, t0, T, sihrt, Nmax)
    ax[0].clear()
    ax[0].plot(ts, ys, "--", markersize=3)
    ax[0].legend(["S", "I", "H", "R"])
    ax[0].set_xlim(0,Nyears*365)
    ax[0].set_ylim(0, 1.0)
    ax[0].set_xlabel("Days")
    ax[0].set_ylabel("Fraction of population")

    ax[1].clear()
    ax[1].plot(ts, 2e5*ys[:,2], "--g", markersize=3, label="H")
    ax[1].legend()
    ax[1].set_xlabel("Days")
    ax[1].set_ylabel("Number of hospitalized persons")
    ax[1].set_xlim(0,Nyears*365)
    ax[1].set_ylim(0, Nbeds)
    ax[1].hlines([Nfree_beds], t0,T, colors="r", linestyles="dashed")
    ax[1].annotate('Max capacity of \nadditional available beds',
             xy=(500, Nfree_beds), xytext=(-50, 50),
             textcoords="offset points",
             arrowprops=dict(arrowstyle="simple",facecolor='black',
                             relpos=(0.315,0)))
    print(f"Maximum of additional Covid 19 caused hospitalization: {np.abs(ys[:,2]).
 ↪max()*200000}")
```

```
plt.close()
fig, ax = plt.subplots(2,1)
widgets.interact(plot_dynamics_sihrt, r0=(1.0, 3.0, 0.01), solver=fixed(heun),␣
 ↪ax=fixed(ax))
```

```
interactive(children=(FloatSlider(value=2.0, description='r0', max=3.0, min=1.0,␣
↪step=0.01), Output()), _dom_c…
```

```
<function __main__.plot_dynamics_sihrt(r0, solver, ax)>
```



4.8 Summary

This chapter addresses the numerical solution of initial value problems (IVPs) for ordinary differential equations (ODEs), a cornerstone of scientific computing. It begins with motivating examples from population dynamics, epidemic modeling (SIR), predator-prey systems (Lotka–Volterra), and nonlinear oscillators (Van der Pol equation). The chapter then systematically introduces and analyzes various numerical methods for solving ODEs, with a progression from basic to more advanced techniques:

- Section 4.1 – Motivation and Modeling
  - Scalar first-order ODEs and systems of ODEs.
  - Real-world examples such as exponential growth/decay, time-dependent coefficients, and the SIR and Lotka-Volterra models.
  - Rewriting higher-order ODEs as first-order systems.
- Section 4.2 – Euler's and Heun's Methods
  - Derivation of Euler's method from Taylor series and integral approximations.

  - Introduction of Heun's method as a predictor-corrector scheme for improved accuracy.
  - Implementation details and example applications.
- Section 4.3 – Error Analysis of One-Step Methods
  - Concepts of local truncation error and global error.
  - Orders of accuracy and consistency.
  - Lipschitz continuity and convergence proofs.
- Section 4.4 – Higher Order Runge-Kutta Methods
  - General form of explicit Runge-Kutta methods.
  - Classical fourth-order RK4 method.
  - Trade-off between computational cost and accuracy.
- Section 4.5 – Adaptive Time Stepping and Error Estimation
  - Step size control using local error estimates.
  - Embedded Runge-Kutta pairs (e.g., Fehlberg).
  - Balancing stability, efficiency, and accuracy.
- Section 4.6 – Stiff ODEs
  - Definition and examples of stiffness.
  - Stability regions and limitations of explicit methods.
  - Introduction to implicit methods for stiff problems (briefly, as a motivation for later courses).
- Section 4.7 – Mini Project: SIR Model Simulation
  - Full implementation of the SIR model using numerical ODE solvers.
  - Visual exploration of disease dynamics and parameter sensitivity.
  - Demonstration of how simple models can produce rich dynamic behavior.

? Learning Outcomes for Chapter 4

By the end of this chapter, students will be able to:

- Modeling and Understanding
  - Have a basic understanding of how initial value problems (IVPs) for scalar and systems of ODEs can be used to model phenomena in biology, epidemiology, and physics.
  - Convert higher-order differential equations into first-order systems.
- Numerical Methods
  - Derive and implement Euler's method and Heun's method, and explain their relation to Taylor expansion and quadrature rules.
  - Derive and implement higher-order, explicit Runge-Kutta methods, including the classical fourth-order RK4 method.
  - Understand formal descriptions of Runge-Kutta methods using stages, stage derivatives and Butcher tableaux
- Error Analysis
  - Know how to write one-step methods using increment functions
  - Understand the concept of local truncation error, global error, consistency order and convergence order.

  - Derive Lipschitz conditions for Runge-Kutta methods and understand their implications for translating local consistency order into global convergence order.
  - Derive the local truncation error and Lipschitz conditions for Euler's method and Heun's method
  - Assess the accuracy, efficiency and convergence order of different one-step methods numerically through the method of manufactured solutions (EOC studies)
- Adaptive Methods
  - Apply adaptive step size control using embedded Runge-Kutta pairs and local error estimators.
  - Evaluate the trade-offs between step size, accuracy, and computational cost.
- Stiffness and Stability
  - Identify stiff ODE problems and explain why explicit methods may fail.
  - Derive stability function and stability regions for explicit methods. motivate the use of implicit methods in stiff contexts (conceptual level).

CHAPTER

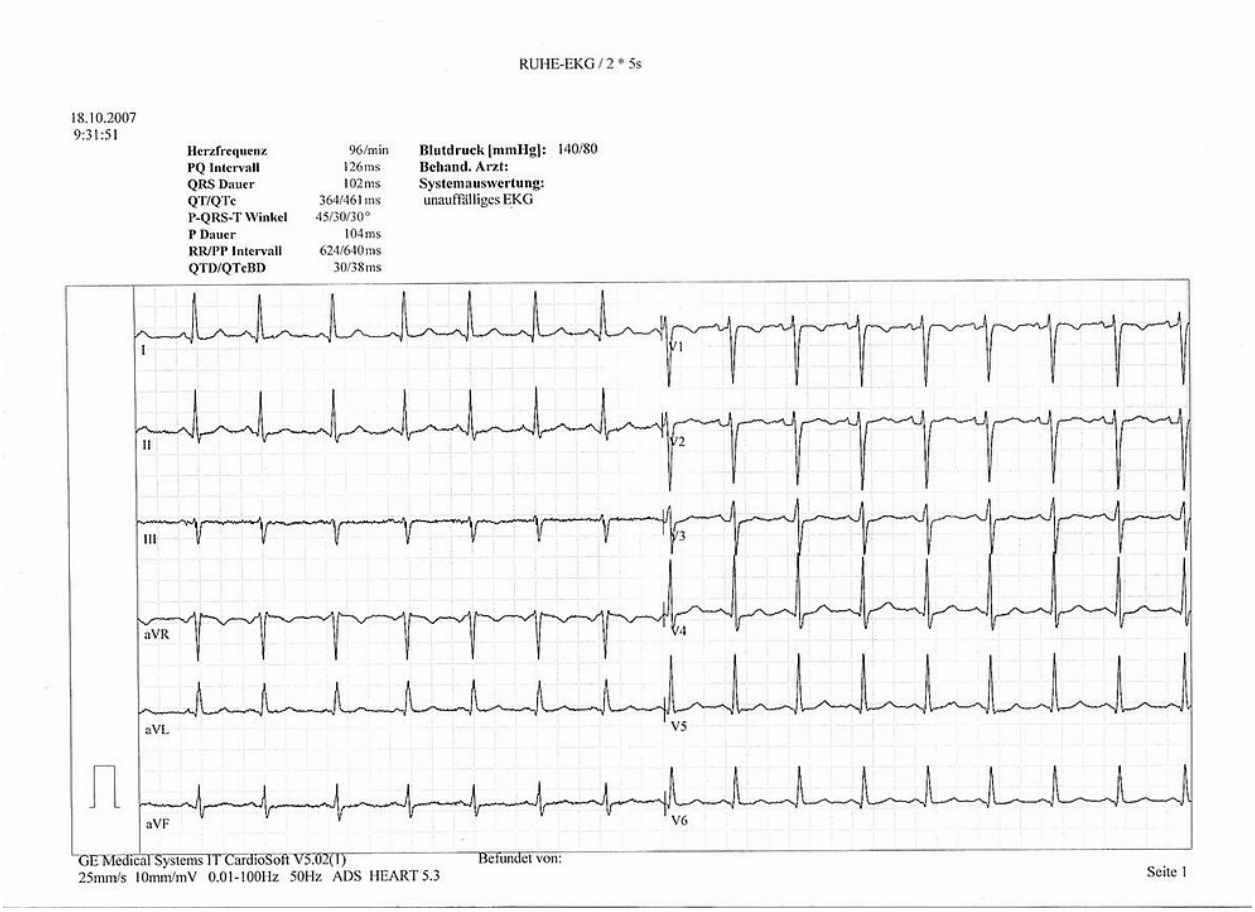
FIVE

THE DISCRETE FOURIER TRANSFORM AND ITS APPLICATIONS

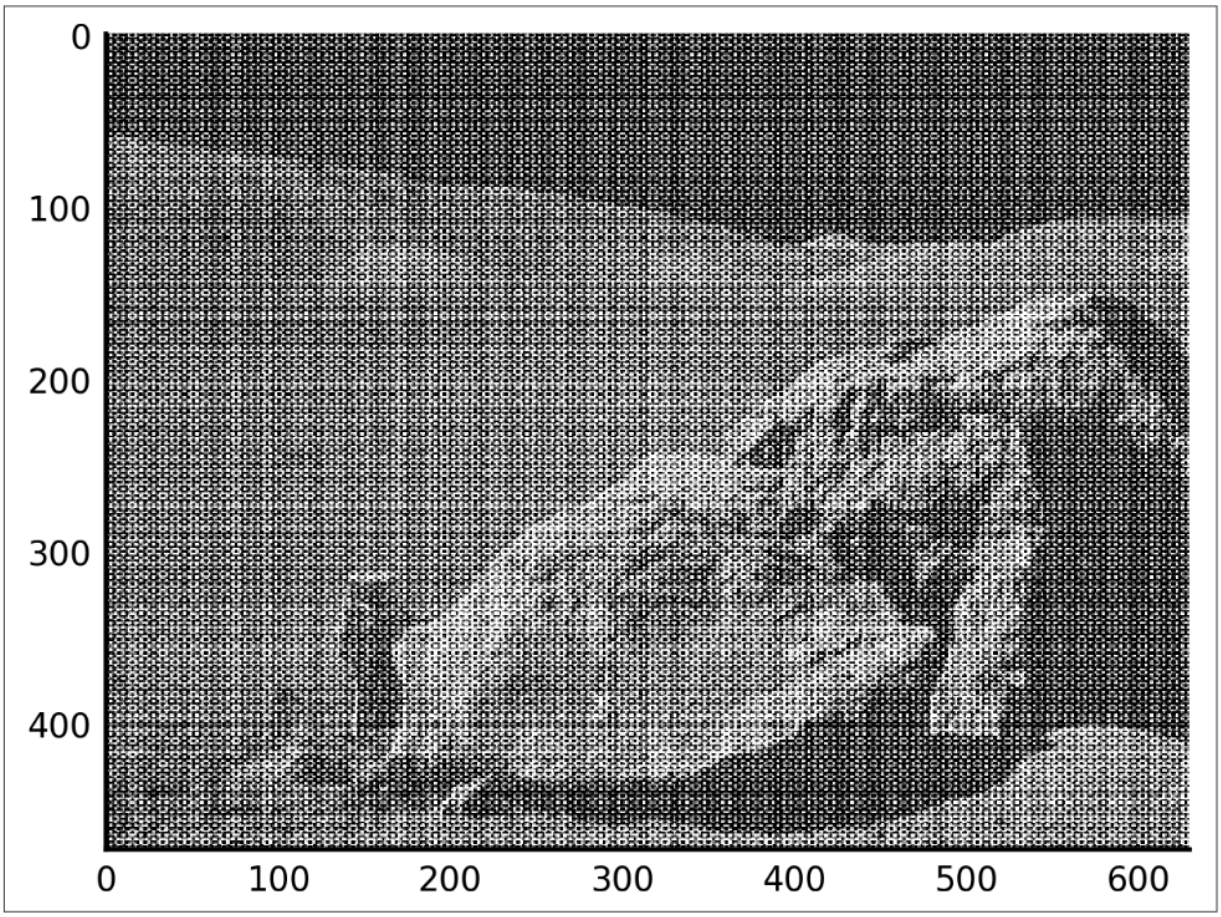
5.1 Motivation

The discrete Fourier transform and its efficient implementation in form of the so-called Fast Fourier Transform is considered to be among the *top 10 most important algorithms* in applied mathematics.

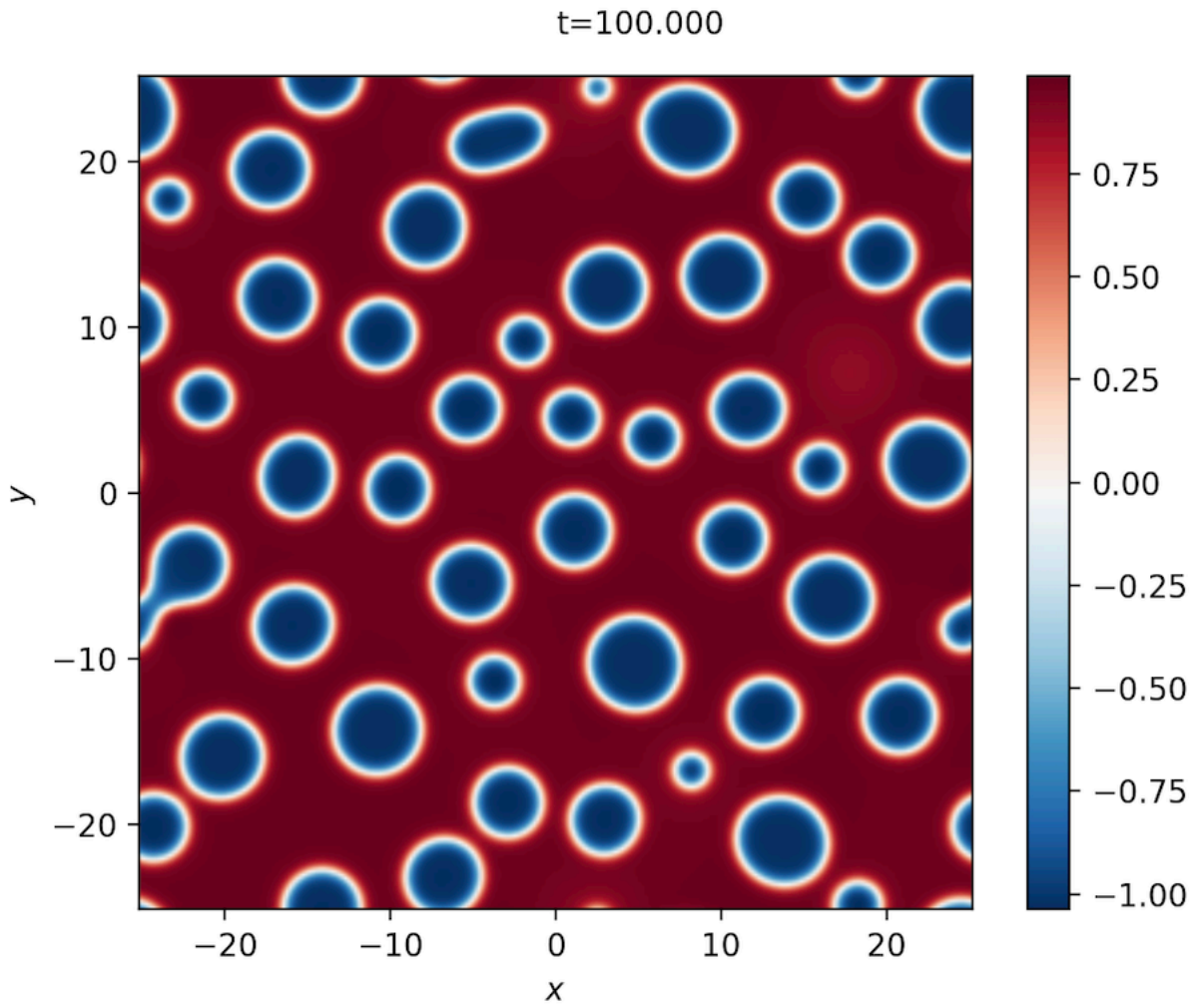
In this module we will have its foundation and briefly discuss applications to topics such a signal analysis, image processing/denoising, and the solution of partial differential equations.



Example of an (healthy) electrocardiography.



Famous noisy image of the moon landing.



Snapshot of the Cahn-Hilliard equation modeling phase separation.

5.2 Preliminaries

First we recall some fundamental concepts, ideas and identities from Matte 4K, see in particular week 35 - week 38.

5.2.1 Complex numbers

[Complex numbers]
$z = a + bi$ where a and b are real numbers and $i = \sqrt{-1}$ is the imaginary unit. We write $\Re(z) = a$ and $\Im(z) = b$ for the real and imaginary part of z.

[Complex conjugate]
of z is $\bar{z} = a - bi$.

[Euler's formula]
$e^{i\theta} = \cos\theta + i\sin\theta$.

[Polar form]
$z = x + iy = re^{i\theta}$ where $r = |z| = \sqrt{x^2 + y^2}$ is the **magnitude** and $\theta = \arg z = \arctan(y/x)$ is the **argument**

or **phase**.

[n-th roots of unity]
For given n, the N-th roots of unity are the solutions to the equation $z^n = 1$. There are n distinct roots which are given by

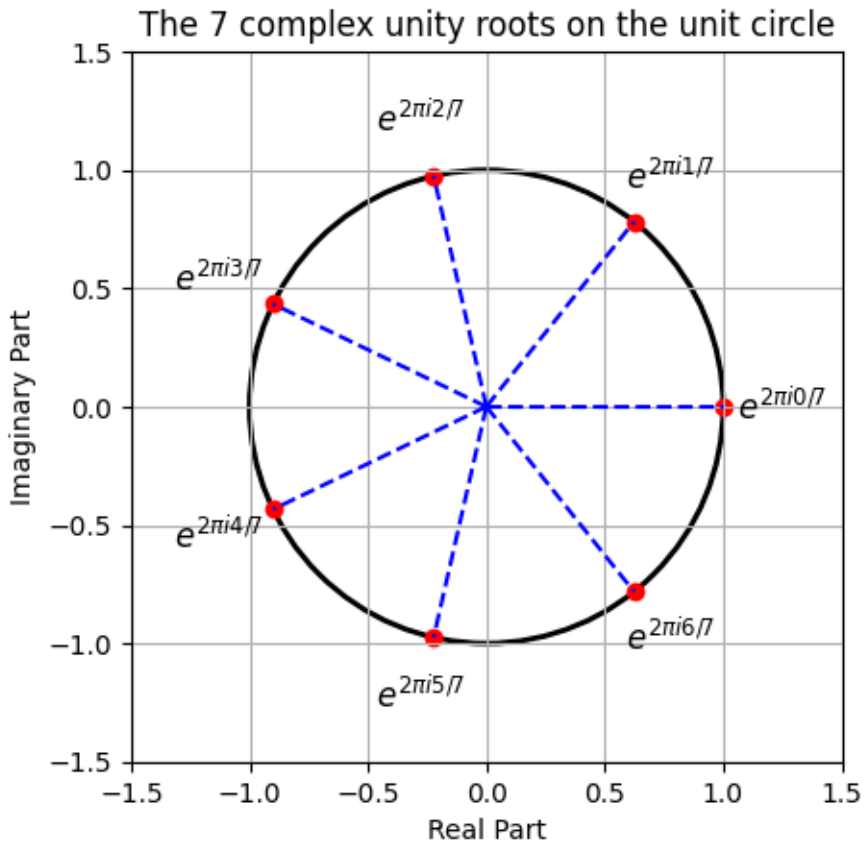
$$\omega_N^k = e^{2\pi i k/N} \quad \text{for } k = 0, 1, \dots, N-1.$$

(four:eq:unityroots)

Observation 8

We have the following easily verifiable properties of the roots of unity for $k, l \in \mathbb{Z}$:

$$(\omega_N^k)^l = \omega_N^{lk} = (\omega_N^l)^k, \qquad \omega_N^{k+l} = \omega_N^k \omega_N^l, \qquad \overline{\omega_N^k} = \omega_N^{-k}, \qquad \omega_N^{k+N} = \omega_N^k.$$




```
# TODO: Present operation of complex numbers in Python
z1 = complex(1,2)
print(z1)
z2 = 1 + 2j
print(z2)
```

```
(1+2j)
(1+2j)
```

5.2.2 Complex inner product spaces and orthogonal systems

Definition 10 (Complex inner product space)

Let V be a complex vector space. An inner product on V is a function $\langle \cdot, \cdot \rangle : V \times V \to \mathbb{C}$ that satisfies the following properties for all $f, g, h \in V$ and all $\alpha, \beta \in \mathbb{C}$:

1. **Linearity in the first argument:**

$$\langle \alpha f + \beta g, h \rangle = \alpha \langle f, h \rangle + \beta \langle g, h \rangle.$$

2. **Conjugate symmetry:**

$$\langle f, g \rangle = \overline{\langle g, f \rangle}.$$

3. **Positive definiteness:**

$$\langle f, f \rangle \geq 0,$$

with equality if and only if $f = 0$.

As with all inner product spaces, a norm by

$$\|f\| = \sqrt{\langle f, f \rangle}. \tag{5.1}$$

and we have the Cauchy-Schwarz inequality

$$|\langle f, g \rangle| \leqslant \|f\| \|g\|. \tag{5.2}$$

For an inner product space, the Cauchy-Schwarz inequality holds

$$|\langle f, g \rangle| \leqslant \|f\| \|g\|$$

and equality holds if and only if f and g are linearly dependent.

Definition 11 (Orthogonal system)

A sequences/family $\{\phi_n\}_{n \in \mathbb{N}}$ of non-zero vectors ϕ_n in a complex inner product space V is said to be **orthogonal** if

$$\langle \phi_n, \phi_m \rangle = 0, \quad n \neq m.$$

If in addition $\|\phi_n\| = 1$ for all n, then the system is said to be **orthonormal**.

For a given interval $[a, b]$, we define the set of square-integrable, possibly complex-valued function $L^2(I)$ by

$$L^2(I) = \left\{ f : I \to \mathbb{C} \mid \int_I |f(x)|^2 dx < \infty \right\}. \tag{5.3}$$

Here, the interval I can be either finite, semi-infinite or infinite, i.e., the end point choices $a = -\infty$ and/or $b = \infty$ are allowed.

For $f, g \in L^2(I)$, an inner product is defined by

$$\langle f, g \rangle = \int_I f(x)\bar{g}(x)dx. \tag{5.4}$$

From hereon, we think of $L^2(I)$ as a inner product space equipped with the inner product defined by (5.4).

Set $[a, b] = [-\pi, \pi]$. We have the following orthogonal systems in $L^2([-\pi, \pi])$.

Example 16

The set of functions $\{e^{inx}\}_{n\in\mathbb{Z}}$ is an orthogonal system in $L^2([-\pi, \pi])$. Correspondingly, the set $\{e^{inx}/\sqrt{2\pi}\}_{n\in\mathbb{Z}}$ is an orthonormal system in $L^2([-\pi, \pi])$.

The set of functions

$$\{e^{inx}\}_{n\in\mathbb{Z}} \quad \text{and} \quad \{e^{inx}/\sqrt{2\pi}\}_{n\in\mathbb{Z}}$$

are orthogonal and orthonormal system in $L^2([-\pi, \pi])$, respectively.

Example 17

The set of functions

$$\{1\} \cup \{\cos(nx)\}_{n=1}^{\infty} \cup \{\sin(nx)\}_{n=1}^{\infty} \quad \text{and} \quad \{1/\sqrt{2\pi}\} \cup \{\cos(nx)/\sqrt{\pi}\}_{n=1}^{\infty} \cup \{\sin(nx)/\sqrt{\pi}\}_{n=1}^{\infty}$$

are orthogonal and orthonormal systems in $L^2([-\pi, \pi])$, respectively.

5.3 Fouries series

Let's consider a periodic function $f(x)$ with period 2π, i.e., $f(x + 2\pi) = f(x)$ for all x. Then the formal **complex** Fourier series of $f(x)$ is given by

$$f(x) = \sum_{k=-\infty}^{\infty} c_k e^{ikx} \tag{5.5}$$

where $\{c_k\}_{k\in\mathbb{Z}}$ are the Fourier coefficients given by

$$c_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x)e^{-ikx}\, dx \tag{5.6}$$

We denote by $S_N(f, x)$ the N-th partial sum of the Fourier series of $f(x)$, i.e.,

$$S_N(f, x) = \sum_{k=-N}^{N} c_k e^{ikx}. \tag{5.7}$$

We recall that $S_N(f, x)$ can we rewritten in terms of sin and cos functions:

$$\sum_{k=-n}^{n} c_k e^{ikx} = \frac{a_0}{2} + \sum_{k=1}^{N} a_k \cos(kx) + b_k \sin(kx), \tag{5.8}$$

where

$$a_0 = 2c_0, \quad a_k = c_k + c_{-k}, \quad b_k = i(c_k - c_{-k}).$$

We set $L_p^2([-\pi, \pi])$ to be the set of periodic functions with period 2π which are square-integrable over some (and thus any) interval $[a, a + 2\pi]$ of length 2π.

5.4 The discrete Fourier transform

5.4.1 Motivation

Let's assume we have an L-periodic function $f(x)$, which is defined on the interval $[0, L)$. We define the N equidistant points on this interval as $x_k = k\frac{L}{N}$, where $k = 0, 1, \dots, N-1$ with corresponding function values $f_k = f(x_k)$.

Based on the points x_k and sampled function values f_k we now want to compute/approximate the Fourier series for the function $f(x)$. As we have N sampling points, it seems natural to compute N Fourier coefficients $c_k(f)$ for the Fourier series expansion of $f(x)$. To compute the integrals for the Fourier coefficients, we recall the definition of the **composite trapezoidal rule** to approximate integrals. For a given L-periodic $g(x)$, the integral of $g(x)$ over the interval $[0, L)$ can be approximated by

$$\int_0^L g(x)\,dx \approx \frac{L}{N}\left(\frac{g_0}{2} + g_1 + g_2 + \dots + g_{N-1} + \frac{g_N}{2}\right) \tag{5.9}$$

$$= \frac{L}{N}\left(g_0 + g_1 + g_2 + \dots + g_{N-1}\right) \tag{5.10}$$

where we set $g_k = g(x_k)$ and used the fact that $g_0 = g_N$ for L-periodic functions.

We apply this formula to approximate the Fourier coefficients $c_k(f)$ of the function $f(x)$:

$$c_k(f) = \hat{f}(n) = \frac{1}{L}\int_0^L f(x)e^{-i2\pi kx/L}\,dx \tag{5.11}$$

$$\approx \frac{1}{N}\sum_{l=0}^{N-1} f_l e^{-i2\pi kx_l/L} \tag{5.12}$$

$$= \frac{1}{N}\sum_{l=0}^{N-1} f_l e^{-i2\pi kl/N} \tag{5.13}$$

$$= \frac{1}{N}\sum_{l=0}^{N-1} f_k \omega_N^{-kl} \tag{5.14}$$

Definition 12 (Discrete Fourier transform)

The discrete Fourier transform (DFT) of a sequence $f = \{f_0, f_1, \dots, f_{N-1}\} \in \mathbb{C}^N$ is itself a sequence $\hat{f} = \{\hat{f}_0, \hat{f}_1, \dots, \hat{f}_{N-1}\} \in \mathbb{C}^N$ defined by

$$\hat{f}_k = \frac{1}{N}\sum_{l=0}^{N-1} f_l \omega_N^{-lk}, \tag{5.15}$$

where $\omega_N = e^{-i2\pi/N}$.

In matrix notation, the DFT can be written as

$$\hat{f} = \mathcal{F}_N f$$

where $\mathcal{F}_N$ is the (symmetric!) Fourier matrix with elements $F_{k,l} = \omega_N^{-kl}$, i.e.

$$\mathcal{F}_N = \frac{1}{N}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^{-1} & \omega_N^{-2} & \cdots & \omega_N^{-(N-1)} \\ 1 & \omega_N^{-2} & \omega_N^{-4} & \cdots & \omega_N^{-2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{-(N-1)} & \omega_N^{-2(N-1)} & \cdots & \omega_N^{-(N-1)(N-1)} \end{pmatrix}$$

5.4.2 Discrete inner products and orthogonality systems

The approximation of the Fourier coefficients $c_k(f)$ by the DFT can be interpreted as a discrete inner product of the function values $f(x_k) = f_k$ with the complex exponentials $e^{-i2\pi kl/N} = \omega_N^{-kl}$. To facilitate the analysis of the DFT, we introduce the following discrete inner product.

Definition 13 (Discrete inner product)

For two complex sequences $f = \{f_0, f_1, \dots, f_{N-1}\} \in \mathbb{C}$ and $g = \{g_0, g_1, \dots, g_{N-1}\} \in \mathbb{C}$, the discrete inner product is defined as

$$\langle f, g\rangle_N = \frac{1}{N}\sum_{l=0}^{N-1} f_l \overline{g_l} \tag{5.16}$$

where $\overline{g_l}$ denotes the complex conjugate of g_l.

As before, we assume that we have a sequence of N equidistant points $x_k = k\frac{L}{N}$ on the interval $I = [0, L)$.

For the given interval I, let us now again consider the complex exponential functions $\omega^l(x) := e^{i2\pi lx/L}$, $l \in \mathbb{Z}$. Previously, we have seen that these functions form an orthogonal system with respect to the continuous inner product $\langle f, g\rangle = \int_0^L f(x)\overline{g(x)}\,dx$. Funnily enough, these functions satisfy a very similar orthogonality property with respect to the discrete inner product:

Theorem 12 (Orthogonality of complex exponentials)

$$\langle \omega^l, \omega^m\rangle_N = \begin{cases} 1 & \text{if } (l-m)/N \in \mathbb{Z}, \\ 0 & \text{else.} \end{cases}$$

Before we turn to the proof of this theorem, we make the very important observation that the evaluation of $\omega^l(x)$ at the points x_k is given by the k-th power of the l-th N-th root of unity $\omega_N^l = e^{i2\pi l/N}$, i.e. $\omega^l(x_k) = e^{i2\pi lk/N} = \omega_N^{lk}$, or in other words

$$(\omega^l(x_k))_{k=0}^{N-1} = (\omega_N^{lk})_{k=0}^{N-1} = (1, \omega_N^l, \omega_N^{2l}, \dots, \omega_N^{(N-1)l}).$$

(four:eq:nth_root_vectors)

Proof. First we recall a fundamental identity for geometric sums, namely that for any given $q \neq 1$, we have

$$\sum_{k=0}^{N-1} q^{N-1} = \frac{1-q^N}{1-q}.$$

For two complex exponentials ω^l and ω^m, we compute the discrete inner product

$$N\langle \omega^l, \omega^m \rangle_N = \sum_{k=0}^{N-1} \omega^l(x_k)\overline{\omega^m(x_k)} \tag{5.17}$$

$$= \sum_{k=0}^{N-1} \omega_N^{lk}\omega_N^{-mk} \tag{5.18}$$

$$= \sum_{k=0}^{N-1} (\omega_N^{l-m})^k \tag{5.19}$$

If $l-m = pN$ for some $p \in \mathbb{Z}$, then $\omega_N^{l-m} = e^{i2\pi(l-m)/N} = e^{i2\pi p} = 1$ and the sum evaluates to N. Otherwise we use the geometric sum identity to obtain

$$\sum_{k=0}^{N-1} (\omega_N^{l-m})^k = \frac{1-\omega_N^{(l-m)N}}{1-\omega_N^{l-m}} = \frac{1-e^{i2\pi(l-m)}}{1-\omega_N^{l-m}} = 0.$$

Let us record a number of important consequences of this orthogonality property.

Corollary 1

1. For $0 \leqslant l, m \leqslant N-1$, the complex exponentials ω^l and ω^m are orthonormal with respect to the discrete inner product

   $$\langle \omega^l, \omega^m \rangle_N = \delta_{lm}.$$

   Equivalently, the N vectors

   $$(\omega^l(x_k))_{k=0}^{N-1} = (\omega_N^{lk})_{k=0}^{N-1} = (1, \omega_N^l, \omega_N^{2l}, \dots, \omega_N^{(N-1)l}) \quad l = 0, 1, \dots, N-1$$

   are orthonormal with respect to the discrete inner product. In particular, they forma a **orthornormal basis** of the vector space $\mathbb{C}^N$.

2. The numerical quadrature rule

   $$\int_0^L f(x)\,dx \approx \frac{L}{N}\sum_{k=0}^{N-1} f(x_k)$$

   is in fact exact for the integration of the complex exponential functions $\omega^l(x)$ for $0 \leqslant l \leqslant N-1$. Moreover, the quadrature rule is exact for the integration for $\sin(\pm 2\pi N/Lx)$ but not for $\cos(\pm 2\pi N/Lx)$.

Exercise 27

Prove the last statement using the the discrete orthogonality of the complex exponentials.

Important

The definition of the discrete Fourier transform in Equation (5.15) can be neatly rewritten in terms of the discrete inner product as

$$\hat{f}_k = \langle f(x), \omega^k(x) \rangle_N \tag{5.20}$$
$$= \langle (f_l)_{l=0}^{N-1}, (\omega^k(x_l))_{l=0}^{N-1} \rangle_N \tag{5.21}$$
$$= \frac{1}{N}(f_0, f_1, \dots, f_{N-1}) \cdot (1, \omega_N^l, \omega_N^{2l}, \dots, \omega_N^{(N-1)l}). \tag{5.22}$$

where $\cdot$ denotes the usual (complex) scalar product of two vectors.

This strongly resembles the definition of the usual Fourier coefficients $c_k(f) = \hat{f}(k)$:

$$\hat{f}(k) = \frac{1}{L}\int_0^L f(x)\overline{\omega^k(x)}\,dx \tag{5.23}$$
$$= \frac{1}{L}\langle f, \omega^k \rangle \tag{5.24}$$

Or in other words: The discrete Fourier coefficient $\hat{f}_k$ resulting from the DFT is a discrete inner product of the function values f_k with the complex exponentials $\omega^l(x_k)$, while the usual Fourier coefficient $\hat{f}(k) = c_k(f)$ is the continuous inner product of the function $f(x)$ with the complex exponentials $\omega^l(x)$.

We can use the previous observations to show the the matrix associated with the discrete Fourier transform in invertible and to compute its inverse.

Theorem 13

The matrix of the DFT is invertible and its inverse $\mathcal{F}_N^{-1}$ is given by

$$\mathcal{F}_N^{-1} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^1 & \omega_N^2 & \cdots & \omega_N^{(N-1)} \\ 1 & \omega_N^2 & \omega_N^4 & \cdots & \omega_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{(N-1)} & \omega_N^{2(N-1)} & \cdots & \omega_N^{(N-1)(N-1)} \end{pmatrix}$$

Proof. We simply compute the product of the DFT matrix $\mathcal{F}_N$ and its inverse $\mathcal{F}_N^{-1}$:

$$(\mathcal{F}_N^{-1}\mathcal{F}_N)_{l,m} = \sum_{k=0}^{N-1} F_{l,k}^{-1} F_{k,m} = \frac{1}{N}\sum_{k=0}^{N-1} \omega_N^{lk}\omega_N^{-km} = \langle \omega^l, \omega^m \rangle_N = \delta_{lm} \tag{5.25}$$

This gives rise to the following definition of the inverse DFT.

Definition 14 (Inverse discrete Fourier transform)

The inverse discrete Fourier transform (IDFT) of a sequence $c = \{c_0, c_1, \dots, c_{N-1}\} \in \mathbb{C}^N$ is itself a sequence $f = \{f_0, f_1, \dots, f_{N-1}\} \in \mathbb{C}^N$ defined by

$$f_l = \sum_{k=0}^{N-1} c_k \omega_N^{lk} \tag{5.26}$$

Important note!

There a lot of different conventions for the normalization of the DFT and the IDFT. Another common one is to use the normalization factor $1/\sqrt{N}$ for both the DFT and the IDFT, since the columns (respectively rows) of the resulting matrices are then orthonormal with respect to the inner product `(fou:eq:disc_inner_product)`. and hence they **unitary**, i.e. $\mathcal{F}_N^* \mathcal{F}_N = \overline{\mathcal{F}_N}^\top \mathcal{F}_N = \mathcal{I}$.

Another possible convention is to use the normalization factor $1/N$ for the IDFT and 1 for the DFT. And sometimes, even the sign in complex exponential is changed!

As a consequence, you should always check the normalization conventions both in the used in the in the literature or in software packages! In `scipy.fft` module which we will use later, the function which computes the DFT has an optional flag to switch between the different conventions.

5.5 Trigonometric interpolation and friends

5.5.1 The trigonometric interpolation problem

As usual, we start from a given interval $I = [0, L)$ and N equidistant points $x_k = k\frac{L}{N}$, $k = 0, 1, \dots, N-1$. we now want to consider the following interpolation problem

Definition 15 (Trigonometric interpolation)

For a given list of sampled function values $f_l = f(x_l)$, $0 \leqslant l \leqslant N-1$, find a (the?) trigonometric polynomial $q(x)$ of the form

$$q_N(x) = \sum_{k=0}^{N-1} c_k e^{i2\pi kx/L} = \sum_{k=0}^{N-1} c_k \omega^k(x) \tag{5.27}$$

which interpolates the function values f_l at the points x_l, i.e.

$$q_N(x_l) = f_l, \quad 0 \leqslant l \leqslant N-1. \tag{5.28}$$

Similar to the Vandermonde matrix approach (sec:poly-interpol-direct) for the polynomial interpolation problem we discussed in Chapter *Polynomial interpolation*, we can set up a linear system for the coefficients c_k,

$$f_l = q_N(x_l) = \sum_{k=0}^{N-1} c_k \omega^k(x_l) = \sum_{k=0}^{N-1} c_k \omega_N^{kl}, \quad 0 \leqslant l \leqslant N-1.$$

Or in matrix-vector notation

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^1 & \omega_N^2 & \cdots & \omega_N^{(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{(N-1)} & \omega_N^{2(N-1)} & \cdots & \omega_N^{(N-1)^2} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{N-1} \end{pmatrix}.$$

Aha! This matrix is exactly the one we have seen in the definition on the inverse DFT, only that we now want to solve the inverse problem, i.e. we want to find the coefficients c_k for a given set of function values f_l. But since the inverse DFT is invertible, and its inverse is given by the DFT matrix, we see that there is a unique solution for the coefficients c_k. We record this observation in the following theorem.

Theorem 14 (Trigonometric interpolation problem)

There exists a unique trigonometric polynomial $q(x)$ of the form (5.27) which solves the interpolation problem (5.28), and the coefficients c_k are given by the DFT of the sample vector $\{f_l\}_{l=0}^{N-1}$, i.e.

$$c_k = \hat{f}_k = \frac{1}{N} \sum_{l=0}^{N-1} f_l \omega_N^{-lk}.$$

The trigonometric polynomial $q(x)$ is called the **trigonometric interpolant** of the function values f_l at the points x_l, and sometimes we write

$$q_N(x) =: \pi^N f(x).$$

to emphasize the relation between the function $f(x)$ and its trigonometric interpolant.

Often it desirable to compute a trigonmetric interpolating polynomial which frequencies are centered around 0.

More precisely, assuming that $N = 2n + 1$ is odd, we wish to find a polynomial $\tilde{q}_N(x)$ of the form

$$\tilde{q}_N(x) = \sum_{k=-n}^{n} \tilde{c}_k \omega^k(x)$$

which satisfies the interpolation conditions $\tilde{q}_N(x_l) = f_l$ for $0 \leqslant l \leqslant N-1$.

This one can in fact easily constructed from $q_N(x)$ respectively the coefficients c_k as we will see now. We recall that the original polynomial $q_N(x)$ satisfies

$$f_l = q(x_l) = \sum_{k=0}^{N-1} c_k \omega_N^{kl} \tag{5.29}$$

$$= \sum_{k=0}^{n} c_k \omega_N^{kl} + \sum_{k=n+1}^{N-1} c_k \omega_N^{kl} \tag{5.30}$$

for $0 \leqslant l \leqslant N-1$.

Now we simply shift the index in the second sum by $-N$ and use the periodicity of the N-th roots of unity $\omega_N^{k+N} = \omega_N^k$ to see that

$$q_N(x_l) = \sum_{k=0}^{n} c_k \omega_N^{kl} + \sum_{k=-n}^{-1} c_{k+N} \omega_N^{(k+N)l} \tag{5.31}$$

$$= \sum_{k=0}^{n} c_k \omega_N^{kl} + \sum_{k=-n}^{-1} c_{k+N} \omega_N^{k} = \sum_{k=-n}^{n} \tilde{c}_k \omega_N^{kl} \tag{5.32}$$

holds for $0 \leqslant l \leqslant N-1$, where we set

$$\tilde{c}_k = \begin{cases} c_k & \text{for } 0 \leqslant k \leqslant n, \\ c_{k+N} & \text{for } -n \leqslant k \leqslant -1. \end{cases}$$

Since $\omega_N^{kl} = \omega^k(x_l)$ we thus conclude that the trigonometric polynomial

$$\tilde{q}_N(x) = \sum_{k=-n}^{n} \tilde{c}_k \omega^k(x) = \sum_{k=-(N-1)/2}^{(N-1)/2} \tilde{c}_k \omega^k(x)$$

also satisfies the interpolation conditions $\tilde{q}_N(x_l) = f_l$ for $0 \leqslant l \leqslant N-1$.

Note

This resembles more closely the form of the truncated complex Fourier series for an L periodic function f,

$$f \sim \sum_{k=-n}^{n} \hat{f}(k) e^{i2\pi kx/L} = \sum_{k=-n}^{n} \hat{f}(k) \omega^k(x)$$

For even $N = 2n$, we can follow the same line of thoughts to see that with

$$\tilde{c}_k = \begin{cases} c_k & \text{for } 0 \leqslant k \leqslant n-1, \\ c_{k+N} & \text{for } -n \leqslant k \leqslant -1. \end{cases}$$

the polynomial

$$\tilde{q}(x) = \sum_{k=-n}^{n-1} \tilde{c}_k \omega^k(x) \tilde{q}(x) = \sum_{k=-N/2}^{N/2-1} \tilde{c}_k \omega^k(x)$$

satisfies the interpolation conditions $\tilde{q}(x_l) = f_l$ for $0 \leqslant l \leqslant N-1$.

Because of the simple index shift, we can use the discrete fast Fourier transform of $\{f_l\}_{l=0}^N$ to compute both c_k and $\tilde{c}_k$. As a matter of convention the following indexing is commonly used for the Fourier coefficients (But always check!):

For $N = 2n+1$ we have

$$[\hat{f}_0, \hat{f}_1, \dots, \hat{f}_{N-1}] = [c_0, c_1, \dots, c_n, c_{n+1}, \dots, c_{N-1}] \tag{5.33}$$
$$= [\tilde{c}_0, \dots, \tilde{c}_n, \tilde{c}_{-n}, \dots, \tilde{c}_{-1}] \tag{5.34}$$
$$= [\hat{f}_0, \hat{f}_1, \dots, \hat{f}_n, \hat{f}_{-n}, \dots, \hat{f}_{-1}] \tag{5.35}$$

For $N = 2n$ we have instead

$$[\hat{f}_0, \hat{f}_1, \dots, \hat{f}_{N-1}] = [c_0, c_1, \dots, c_{n-1}, c_n, \dots, c_{N-1}] \tag{5.36}$$
$$= [\tilde{c}_0, \dots, \tilde{c}_{n-1}, \tilde{c}_{-n}, \dots, \tilde{c}_{-1}] \tag{5.37}$$
$$= [\hat{f}_0, \hat{f}_1, \dots, \hat{f}_{n-1}, \hat{f}_{-n}, \dots, \hat{f}_{-1}] \tag{5.38}$$

5.5.2 Real trigonometric interpolation

Similar to the different representations of the Fourier series, cf. fou:eq:fourier-partial-real, we sometimes want to represent the trigonometric interpolant in terms of cos and sin functions. This is particularly useful to arrive at real-valued trigonometric interpolant in the case that the sampled function real-valued.

To arrive at such a rewrite representation, we proceed exactly in the same way as in the case of the Fourier series: Simply rewrite the complex exponential functions in terms of cos and sin functions via Euler's formula and collect the terms with the same trigonometric functions.

Let's start with the case of an odd $N = 2n + 1$. Then

$$\tilde{q}_N(x) = \sum_{k=-n}^{n} c_k e^{i2\pi kx/L} = c_0 + \sum_{k=1}^{n} c_k e^{i2\pi kx/L} + c_{-k} e^{-i2\pi kx/L} \tag{5.39}$$

$$= c_0 + \sum_{k=1}^{n} c_k \left(\cos(2\pi kx/L) + i\sin(2\pi kx/L)\right) \tag{5.40}$$

$$+ \sum_{k=1}^{n} c_{-k} \left(\cos(2\pi kx/L) - i\sin(2\pi kx/L)\right) \tag{5.41}$$

$$= c_0 + \sum_{k=1}^{n} \left((c_k + c_{-k})\cos(2\pi kx/L) + i(c_k - c_{-k})\sin(2\pi kx/L)\right) \tag{5.42}$$

$$= \frac{a_0}{2} + \sum_{k=1}^{n} \left(a_k \cos(2\pi kx/L) + b_k \sin(2\pi kx/L)\right) \tag{5.43}$$

where we set $a_k = c_k + c_{-k}$ and $b_k = i(c_k - c_{-k})$.

For real-valued samples $\{f_l\}_{l=0}^{N-1}$, we can now make the following observation:

Since $\overline{f_l} = f_l$ for real-valued f_l, we infer that $\overline{c_k} = c_{-k}$ since

$$\overline{c_k} = \overline{\hat{f}_k} = \overline{\frac{1}{N}\sum_{l=0}^{N-1} f_l \omega_N^{-lk}} = \frac{1}{N}\sum_{l=0}^{N-1} f_l \omega_N^{lk} = \hat{f}_{-k} = c_{-k}$$

Consequently,

$$a_k = c_k + c_{-k} = c_k + \overline{c_k} = 2\Re(c_k) = 2\Re(\hat{f}_k) = \frac{2}{N}\Re\left(\sum_{l=0}^{N-1} f_l \omega_N^{-lk}\right) \tag{5.44}$$

$$= \frac{2}{N}\sum_{l=0}^{N-1} f_l \cos(2\pi kl/N)\frac{2}{N} = \sum_{l=0}^{N-1} f_l \cos(2\pi kx_l/L) \tag{5.45}$$

Similarly, we have that

$$b_k = i(c_k - c_{-k}) = i(c_k - \overline{c_k}) = 2\Im(c_k) = 2\Im(\hat{f}_k) = \frac{2}{N}\Im\left(\sum_{l=0}^{N-1} f_l \omega_N^{-lk}\right) \tag{5.46}$$

$$= \frac{2}{N}\sum_{l=0}^{N-1} f_l \sin(2\pi kl/N) = \frac{2}{N}\sum_{l=0}^{N-1} f_l \sin(2\pi kx_l/L) \tag{5.47}$$

Consequently, a_k and b_k are real-valued, and the trigonometric interpolant $\tilde{q}_N(x)$ is in fact a real-valued trigonometric polynomial for all $x \in [0, L)$ (and not only for the sampled points x_l)!

For even $N = 2n$, we follow the convention above and we consider again

$$\tilde{q}_N(x) = \sum_{k=-n}^{n-1} c_k e^{i2\pi kx/L} = c_{-n} e^{-i2\pi nx/L} + \sum_{k=-(n-1)}^{n-1} c_k e^{i2\pi kx/L} \tag{5.48}$$

We can rewrite the sum in the second term as above to obtain

$$\tilde{q}_N(x) = c_{-n}e^{-i2\pi nx/L} + \frac{a_0}{2} + \sum_{k=1}^{n-1}(a_k\cos(2\pi kx/L) + b_k\sin(2\pi kx/L))$$

As before, we conclude that a_k and b_k are real-valued if $\{f_l\}_{l=0}^{N-1}$ are real-valued, and so is the resulting superposition of cos and sin functions.

But what about the term $c_{-n}e^{-i2\pi nx/L}$?

First let us observe that the term c_{-n} is in fact real-valued, since (recalling that $N = 2n$)

$$c_{-n} = \frac{1}{N}\sum_{l=0}^{N-1} f_l e^{i2\pi nl/(2n)} = \frac{1}{N}\sum_{l=0}^{N-1} f_l(-1)^l = \frac{1}{N}\sum_{l=0}^{N-1} f_l\cos(2\pi nl/N) = \frac{1}{N}\sum_{l=0}^{N-1} f_l\cos(2\pi nx_l/L)$$

In particular this means, that $\tilde{q}_N(x)$ is **not** a purely real-valued trigonometric polynomial, it only happens to be real-valued at the sample points x_l!

To remedy this situation and obtain a real-valued trigonometric interpolant, we recall that

$$\Re(\tilde{q}_N(x_l)) = \tilde{q}_N(x_l)$$

must hold for all $0 \leqslant l \leqslant N-1$ since $\tilde{q}_N(x_l) = f_l$ is real-valued. Consequently, we can simply $\Re(\tilde{q}_N(x))$ to find a real-valued trigonometric interpolating polynomial.

But since c_{-n} is real-valued, and thus $\Re(c_{-n}e^{-i2\pi nx/L}) = c_{-n}\Re(e^{-i2\pi nx/L})$, we see that (after setting $2a_n := c_{-n}$)

$$\Re(\tilde{q}_N(x)) = \Re(\tfrac{a_n}{2}e^{-i2\pi nx/L}) + \frac{a_0}{2} + \sum_{k=1}^{n-1}(a_k\cos(2\pi kx/L) + b_k\sin(2\pi kx/L)) \tag{5.49}$$

$$= \frac{a_n}{2}\cos(2\pi nx/L) + \frac{a_0}{2} + \sum_{k=1}^{n-1}(a_k\cos(2\pi kx/L) + b_k\sin(2\pi kx/L)) \tag{5.50}$$

is a real-valued trigonometric interpolant of the real-valued function values f_l at the points x_l.

Let us summarize this in the following theorem.

Theorem 15 (Real trigonometric interpolation)

Let $\{f_l\}_{l=0}^{N-1}$ be a list of real-valued function values at the points $x_l = l\frac{L}{N}$, $0 \leqslant l \leqslant N-1$. Define the coefficients a_k and b_k by

$$a_k = \sum_{l=0}^{N-1} f_l\cos(2\pi kx_l/L), \quad b_k = \sum_{l=0}^{N-1} f_l\sin(2\pi kx_l/L).$$

If $N = 2n+1$ is odd, then the real-valued trigonometric polynomial $p_n(x)$ of order n given by

$$p_n(x) = \frac{a_0}{2} + \sum_{k=1}^{n}(a_k\cos(2\pi kx/L) + b_k\sin(2\pi kx/L)) \tag{5.51}$$

satisfies $f_l = p_n(x_l)$ for $0 \leqslant l \leqslant N-1$.

If $N = 2n$ is even on the other hand, then the real-valued interpolating trigonometric polynomial $p_n(x)$ of order n given by

$$p_n(x) = \frac{a_0}{2} + \sum_{k=1}^{n-1}(a_k\cos(2\pi kx/L) + b_k\sin(2\pi kx/L)) + \frac{a_n}{2}\cos(2\pi nx/L). \tag{5.52}$$

5.5.3 Best approximation properties

We now take another at the properties of the discrete Fourier transform, the resulting coefficients $\hat{f}_k$ and the properties of trigonometric polynomials constructed from them.

As usual we assume we have a function $f(x)$ which is L-periodic and we sample it at N equidistant points $x_l = l\frac{L}{N}$, $0 \leqslant l \leqslant N-1$.

We have seen that the trigonometric interpolant

$$\pi^N f(x) := q_N(x) := \sum_{k=0}^{N-1} \hat{f}_k \omega^k(x)$$

interpolates the function values $f_l = f(x_l)$ at the points x_l.

But what happens if we truncate this sum and consider a lower order trigonometric polynomial of the form

$$\pi_m^N f(x) := q_m(x) := \sum_{k=0}^{m} \hat{f}_k \omega^k(x)$$

for $0 \leqslant m < N-1$?

To gain some insight into this question, we change slightly our perspective and focus on the vector presentation of the sampled function values $\{f_l\}_{l=0}^{N-1}$. We write

$$\mathbf{F} = [f_0, f_1, \dots, f_{N-1}]$$

As we noted before, the N vectors

$$\mathbf{W^k} = (\omega^k(x_l))_{l=0}^{N-1} = (\omega_N^{lk})_{l=0}^{N-1} = [1, \omega_N^k, \omega_N^{2k}, \dots, \omega_N^{(N-1)k}] \quad k = 0, 1, \dots, N-1$$

define a **orthonormal basis** of $\mathbb{C}^N$ with respect to the discrete inner product $\langle \cdot, \cdot \rangle_N$.

Now the interpolation property $\pi^N f(x_l) = q_N(x_l) = f_l$ for $0 \leqslant l \leqslant N-1$ is nothing else than the statement that the vector $\mathbf{F}$ can be written as a linear combination of the vectors $\mathbf{W^l}$, and the coefficients of this linear combination are given by the DFT of $\mathbf{F}$.

$$\begin{aligned}
\mathbf{F} &= [f_0, f_1, \dots, f_{N-1}] && (5.53)\\
&= [q_N(x_0), q_N(x_1) \dots, q_N(x_{N-1})] && (5.54)\\
&= \left[\sum_{k=0}^{N-1} \hat{f}_k \omega^k(x_0), \dots, \sum_{k=0}^{N-1} \hat{f}_k \omega^k(x_{N-1}) \right] && (5.55)\\
&= \sum_{k=0}^{N-1} \hat{f}_k \left[\omega^k(x_0), \dots, \omega^k(x_{N-1}) \right] && (5.56)\\
&= \sum_{k=0}^{N-1} \hat{f}_k \left[1, \omega_N^k, \dots, \omega_N^{k(N-1)} \right] && (5.57)\\
&= \sum_{k=0}^{N-1} \hat{f}_k \mathbf{W^k} && (5.58)
\end{aligned}$$

But do you remember this definition of the $\hat{f}_k$? Recall that we could write them with the discrete inner product as

$$\begin{aligned}
\hat{f}_k &= \langle f(x), \omega^k(x) \rangle_N && (5.59)\\
&= \langle (f_l)_{l=0}^{N-1}, (\omega^l(x_l))_{l=0}^{N-1} \rangle_N && (5.60)\\
&= \langle \mathbf{F}, \mathbf{W^k} \rangle_N && (5.61)
\end{aligned}$$

In other words

$$\mathbf{F} = \sum_{k=0}^{N-1} \langle \mathbf{F}, \mathbf{W}^k \rangle_N \mathbf{W}^{\mathbf{k}}$$

Note that this exactly the usual way for computing the presentation of a vector in terms of an orthonormal basis.

It we repeat this argument for the truncated trigonometric polynomial $\pi_m^N f(x) = q_m(x) \sum_{k=0}^m \hat{f}_k \omega^k(x)$, we obtain

$$\begin{aligned}
\mathbf{Q}_m &:= [q_m(x_0), q_m(x_1) \dots, q_m(x_{N-1})] && (5.62)\\
&= \left[\sum_{k=0}^m \hat{f}_k \omega^k(x_0), \dots, \sum_{k=0}^m \hat{f}_k \omega^k(x_{N-1}) \right] && (5.63)\\
&= \sum_{k=0}^m \hat{f}_k \mathbf{W}^{\mathbf{k}} && (5.64)\\
&= \sum_{k=0}^m \langle \mathbf{F}, \mathbf{W}^k \rangle_N \mathbf{W}^{\mathbf{k}} && (5.65)
\end{aligned}$$

Aha! We see that $\mathbf{Q}_m$ is nothing else than the **orthogonal projection** $\Pi_m \mathbf{F}$ of $\mathbf{F}$ onto the m dimensional subspace of $\mathbb{C}^N$,

$$V_m = \text{span}(\{\mathbf{W}^{\mathbf{0}}, \dots, \mathbf{W}^m\})$$

with respect to the discrete inner product $\langle \cdot, \cdot \rangle_N$!

Equivalently, we can say that the truncated trigonometric polynomial

$$\begin{aligned}
\pi_m^N f(x) = q_m(x) &= \sum_{k=0}^m \hat{f}_k \omega^k(x) && (5.66)\\
&= \sum_{k=0}^m \langle f, \omega^k \rangle_N \omega^k(x) && (5.67)
\end{aligned}$$

is the orthogonal projection of the function $f(x)$ onto the subspace of trigonometric polynomials of order m

$$T_m^{\mathbb{C}} = \text{span}(\{\omega^0, \dots, \omega^m\}) = \{r_m(x) = \sum_{k=0}^m d_k \omega^k(x) \mid d_k \in \mathbb{C}\}$$

with respect to the discrete inner product $\langle \cdot, \cdot \rangle_N$.

Now remember that in general, for orthogonal projection $\Pi_m \mathbf{F}$ of a vector $\mathbf{F}$ onto a subspace V_m, the projection error $\mathbf{F} - \Pi_m \mathbf{F}$ is orthogonal to the subspace V_m,

$$\langle \mathbf{F} - \Pi_m \mathbf{F}, \mathbf{R}_m \rangle_N = 0 \quad \text{for all } \mathbf{R}_m \in V_m, \tag{5.68}$$

or equivalently

$$\langle f - \pi_m^N f, r_m \rangle_N = 0 \quad \text{for } r_m \in T_m^{\mathbb{C}}. \tag{5.69}$$

We can use this property to derive a best approximation property of the truncated trigonometric polynomial $\pi_m^N f(x)$.

Theorem 16 (Best approximation property of the truncated trigonometric polynomial)

The truncated trigonometric polynomial $\pi_m^N f(x)$ satisfies a best approximation property in the sense that

$$\|f - \pi_m^N f\|_N = \min_{r_m \in T_{\mathbb{C}}^N} \|f - r_m\|_N.$$

In other words, $\pi_m^N f(x)$ minimizes the squared error

$$\sum_{l=0}^{N-1} |f_l - r_m(x_l)|^2$$

among all trigonometric polynomials r_m of order m. We say it say $\pi_m^N f(x)$ has the **least square error**. among all trigonometric polynomials of order m.

Proof.

For any given trigonometric polynomial $r_m(x) = \sum_{k=0}^m d_k \omega^k(x)$ of order m, we use the orthogonality property (5.69) to see that

$$\begin{aligned}
\|f - \pi_m^N f\|_N^2 &= \langle f - \pi_m^N f, f - \pi_m^N f\rangle_N && (5.70)\\
&= \langle f - \pi_m^N f, f - r_m + r_m - \pi_m^N f\rangle_N && (5.71)\\
&= \langle f - \pi_m^N f, f - r_m\rangle_N + \underbrace{\langle f - \pi_m^N f, r_m - \pi_m^N f\rangle_N}_{=0 \text{ by orthogonality}} && (5.72)\\
&= \langle f - \pi_m^N f, f - r_m\rangle_N && (5.73)\\
&\leqslant \|f - \pi_m^N f\|_N \|f - r_m\|_N && (5.74)
\end{aligned}$$

Assuming If $\|f - \pi_m^N f\|_N = 0$

Otherwise if $\|f - \pi_m^N f\|_N > 0$, we can divide by this term to obtain

$$\|f - \pi_m^N f\|_N \leqslant \|f - r_m\|_N$$

Since r_m was an arbitrary trigonometric polynomial of order m, we see that the truncated trigonometric polynomial $\pi_m^N f(x)$ satisfies the minimization property

$$\|f - \pi_m^N f\|_N = \min_{r_m \in T_{\mathbb{C}}^N} \|f - r_m\|_N.$$

5.6 Using the discrete Fourier transform

5.6.1 The fast Fourier transform (FFT)

Recall that for a sequence $f = \{f_0, f_1, \dots, f_{N-1}\} \in \mathbb{C}^N$ of sampling points the DFT can be written as

$$\hat{f} = \mathbf{F}_N f$$

with $\hat{f} = \{\hat{f}_0, \hat{f}_1, \dots, \hat{f}_{N-1}\} \in \mathbb{C}^N$ where $\mathcal{F}_N$ is the (symmetric!) Fourier matrix with elements $F_{k,l} = \omega_N^{-kl}$, i.e.

$$\mathbf{F}_N = \frac{1}{N}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^{-1} & \omega_N^{-2} & \cdots & \omega_N^{-(N-1)} \\ 1 & \omega_N^{-2} & \omega_N^{-4} & \cdots & \omega_N^{-2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_N^{-(N-1)} & \omega_N^{-2(N-1)} & \cdots & \omega_N^{-(N-1)(N-1)} \end{pmatrix}$$

(fou:eq:fourier_matrix)

where $\omega_N = e^{-2\pi i/N}$ is the N-th root of unity.

Since a naive matrix-vector multiplication has complexity $\mathcal{O}(N^2)$ (N row* column multiplications with N additions/multiplication each), the direct computation of the DFT via its Fourier matrix becomes dramatically slow for large N.

Luckily, there are a lot of symmetries in the Fourier matrix that can be exploited to reduce the complexity to $\mathcal{O}(N \log N)$. The resulting algorithm is called the Fast Fourier Transform and is considered to be among the *top 10 most important algorithms* in applied mathematics.

While we do not have time to go into the details of the FFT algorithm, we will here reproduce (slightly modified) the nice explanation given in [Brunton and Kutz, 2022], Chapter 2.2.

in particular if the number of data points N is a power of 2.

The basic idea behind the FFT is that the DFT may be implemented much more efficiently if the number of data points N is a power of 2. To get an idea of how symmetries in the Fourier matrix can be exploited, consider the case $N = 2^{10} = 1024$. In this case, the DFT matrix $\mathbf{F}_{1024}$ may be written as

$$\hat{\mathbf{f}} = \mathbf{F}_{1024}\mathbf{f} = \begin{bmatrix} \mathbf{I}_{512} & \mathbf{D}_{512} \\ \mathbf{I}_{512} & -\mathbf{D}_{512} \end{bmatrix} \begin{bmatrix} \mathbf{F}_{512} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_{512} \end{bmatrix} \begin{bmatrix} \mathbf{f}_{\text{even}} \\ \mathbf{f}_{\text{odd}} \end{bmatrix}$$

where $\mathbf{f}_{\text{even}}$ are the even index elements of $\mathbf{f}$, $\mathbf{f}_{\text{odd}}$ are the odd index elements of $\mathbf{f}$, $\mathbf{I}_{512}$ is the 512×512 identity matrix, and $\mathbf{D}_{512}$ is given by

$$\mathbf{D}_{512} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & \omega_{512} & 0 & \cdots & 0 \\ 0 & 0 & \omega_{512}^{-2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \omega_{512}^{-511} \end{bmatrix}$$

This expression can be derived from a careful accounting and reorganization of the terms in the DFT matrix `fou:eq:fourier_matrix`.

If $N = 2^p$, this process can be repeated, and $\mathbf{F}_{512}$ can be represented by $\mathbf{F}_{256}$, which can then be represented by $\mathbf{F}_{128} \to \mathbf{F}_{64} \to \mathbf{F}_{32} \to \cdots$. If $N \neq 2^p$, the vector can be padded/filled with zeros until it is a power of 2 . The FFT then involves an efficient interleaving of even and odd indices of sub-vectors of $\mathbf{f}$, and the computation of several smaller 2×2 DFT computations, with a total complexity of $\mathcal{O}(N \log N)$.

5.6.2 Using the FFT in Python

Now it is time to use the FFT in Python. We will use the `fft` modules from the `scipy` library, which provides a fast implementation of the FFT algorithm. The example material below is taken and adapted from [Brunton and Kutz, 2022], Chapter 2.2. which the authors of the book kindly made available on GitHub, specifically from the example CH02_SEC02_2_Denoise.ipynb

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, ifft, fftfreq, fftshift
import pandas as pd

# plt.rcParams['figure.figsize'] = [15, 5]
```

Create a signal with a single

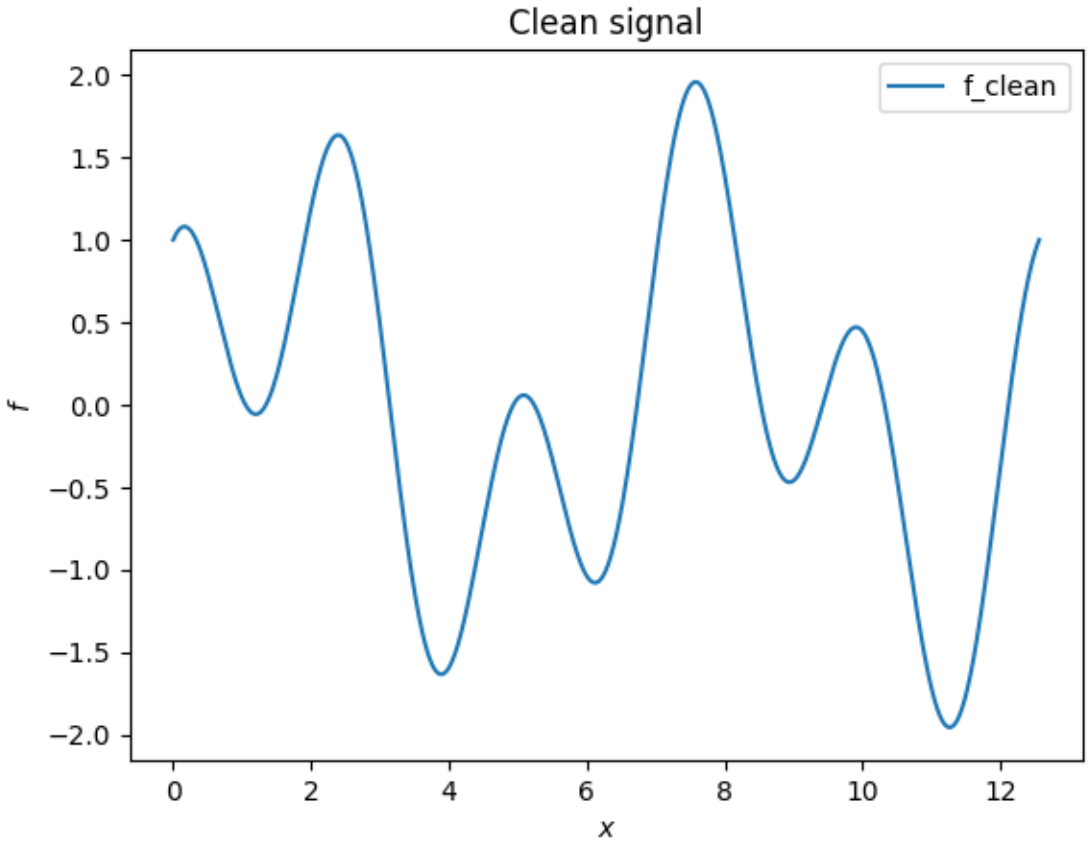
```
# Create a simple signal with two frequencies

# Interval length
```

(continues on next page)

(continued from previous page)

```
L = 4*np.pi
f_clean = lambda x : np.sin(2*np.pi/L*2*x) + np.cos(2*np.pi/L*5*x) # Sum of 2␣
↪frequencies
```

```
x = np.linspace(0, L, 1000)
plt.clf()
plt.plot(x, f_clean(x), label='f_clean')
plt.xlabel(r'$x$')
plt.ylabel(r'$f$')
plt.title('Clean signal')
plt.legend()
plt.show()
```



Next, we sample this signal using 100 sample points.

Warning

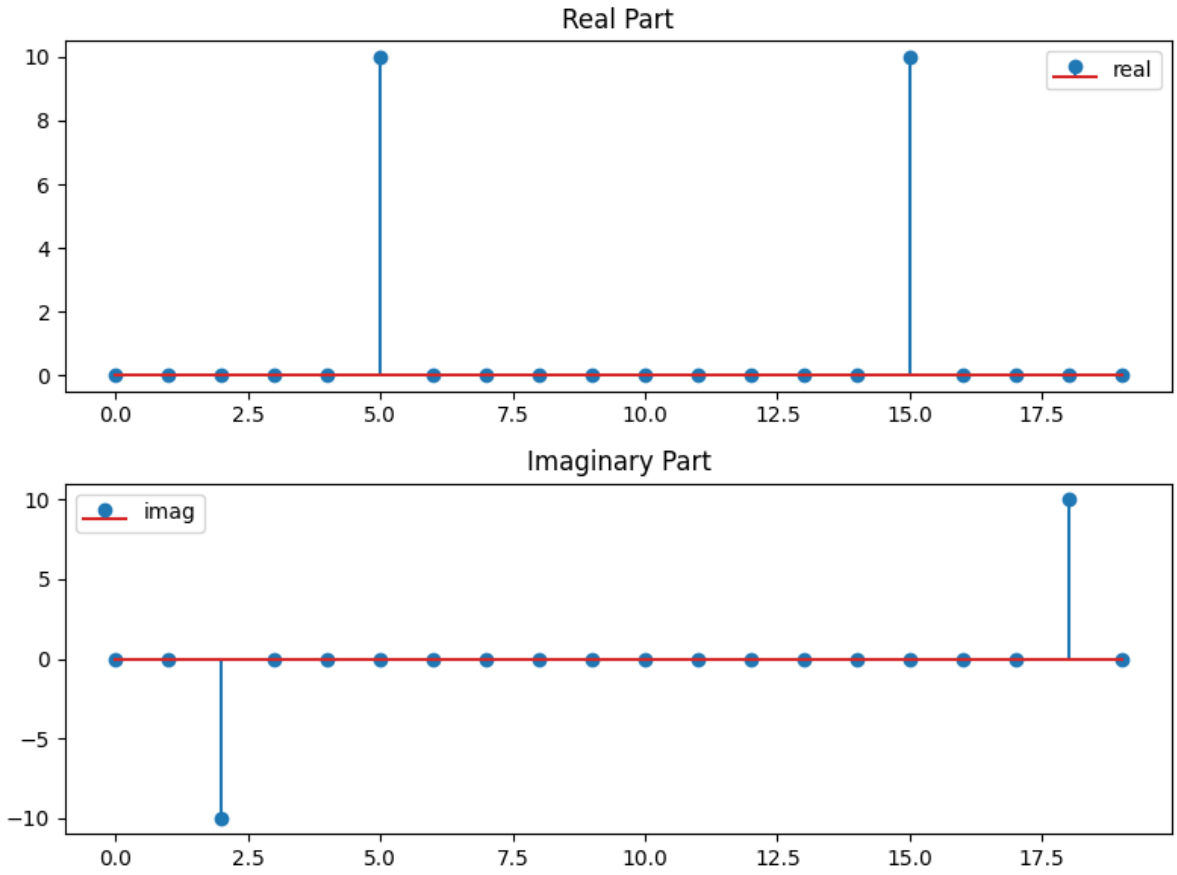
When sampling an supposedly periodic signal over a given domain, you have to make sure to **exclude** the endpoint of the domain from the sampling points. This can be achieved by using the `endpoint=False` argument in the `np.linspace` function. Otherwise you will get rare artifacts in the Fourier transform as you basically sample the signal at the same point twice.

```
# Sample the signal
N = 20
```

(continues on next page)

(continued from previous page)

```
xs = np.linspace(0, L, N, endpoint=False)
fcs = f_clean(xs)

fcs_hat = fft(fcs)
fig = plt.figure(figsize=(8, 6))
axs = fig.subplots(2, 1)
axs[0].stem(fcs_hat.real, label='real')
axs[0].set_title('Real Part')
axs[0].legend()

axs[1].stem(fcs_hat.imag, label='imag')
axs[1].set_title('Imaginary Part')
axs[1].legend()
plt.tight_layout()
```

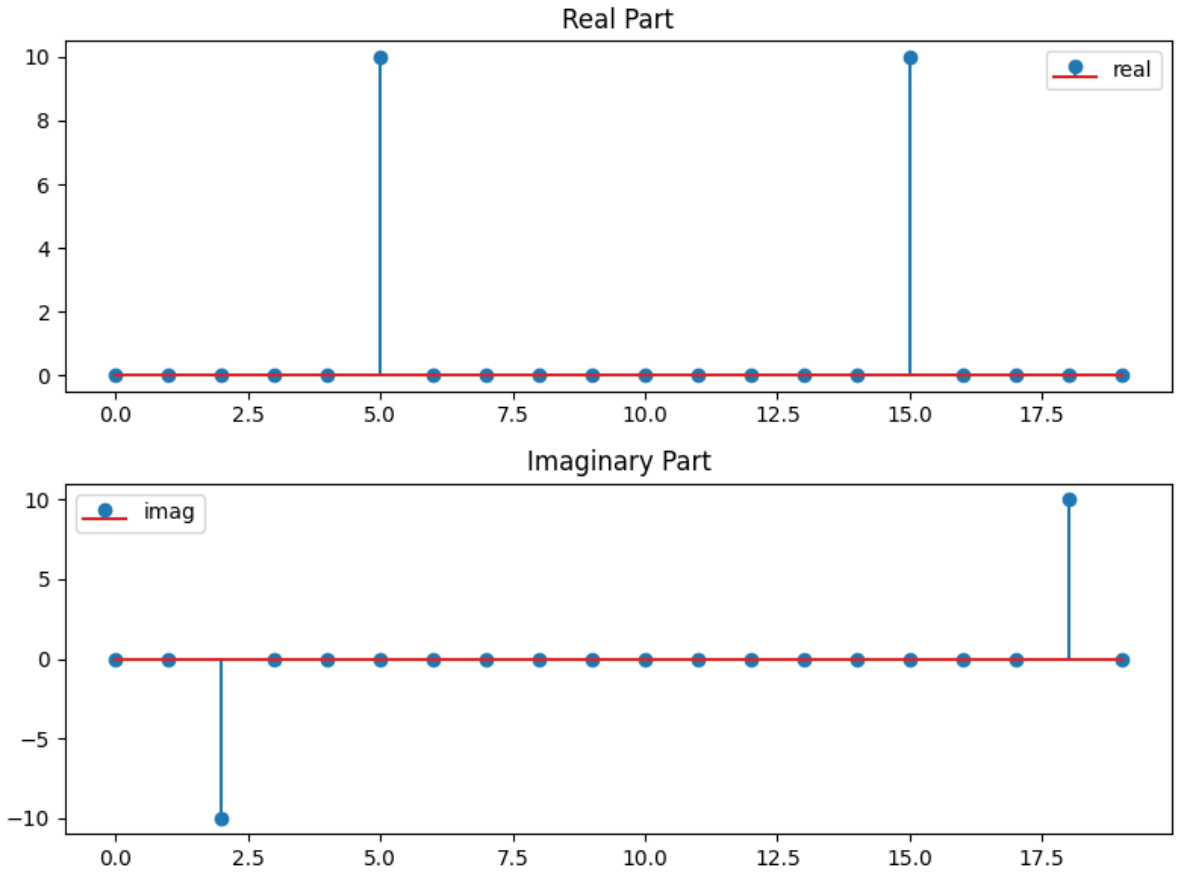



```
# Sample the signal
N = 20
xs = np.linspace(0, L, N, endpoint=False)
fcs = f_clean(xs)

fcs_hat = fft(fcs)
fig = plt.figure(figsize=(8, 6))
axs = fig.subplots(2, 1)
axs[0].stem(fcs_hat.real, label='real')
axs[0].set_title('Real Part')
```

(continues on next page)

(continued from previous page)

```
axs[0].legend()

axs[1].stem(fcs_hat.imag, label='imag')
axs[1].set_title('Imaginary Part')
axs[1].legend()
plt.tight_layout()
```



Ok, let's try to understand this plot a bit better. First, we need to recall the ordering of the FFT output as discussed in *Trigonometric interpolation and friends*:

For $N = 2n + 1$ we have

$$[\hat{f}_0, \hat{f}_1, \dots, \hat{f}_n, \hat{f}_{-n}, \dots, \hat{f}_{-1}] \tag{5.75}$$

while for $N = 2n$ we have instead the order

$$[\hat{f}_0, \hat{f}_1, \dots, \hat{f}_{n-1}, \hat{f}_{-n}, \dots, \hat{f}_{-1}] \tag{5.76}$$

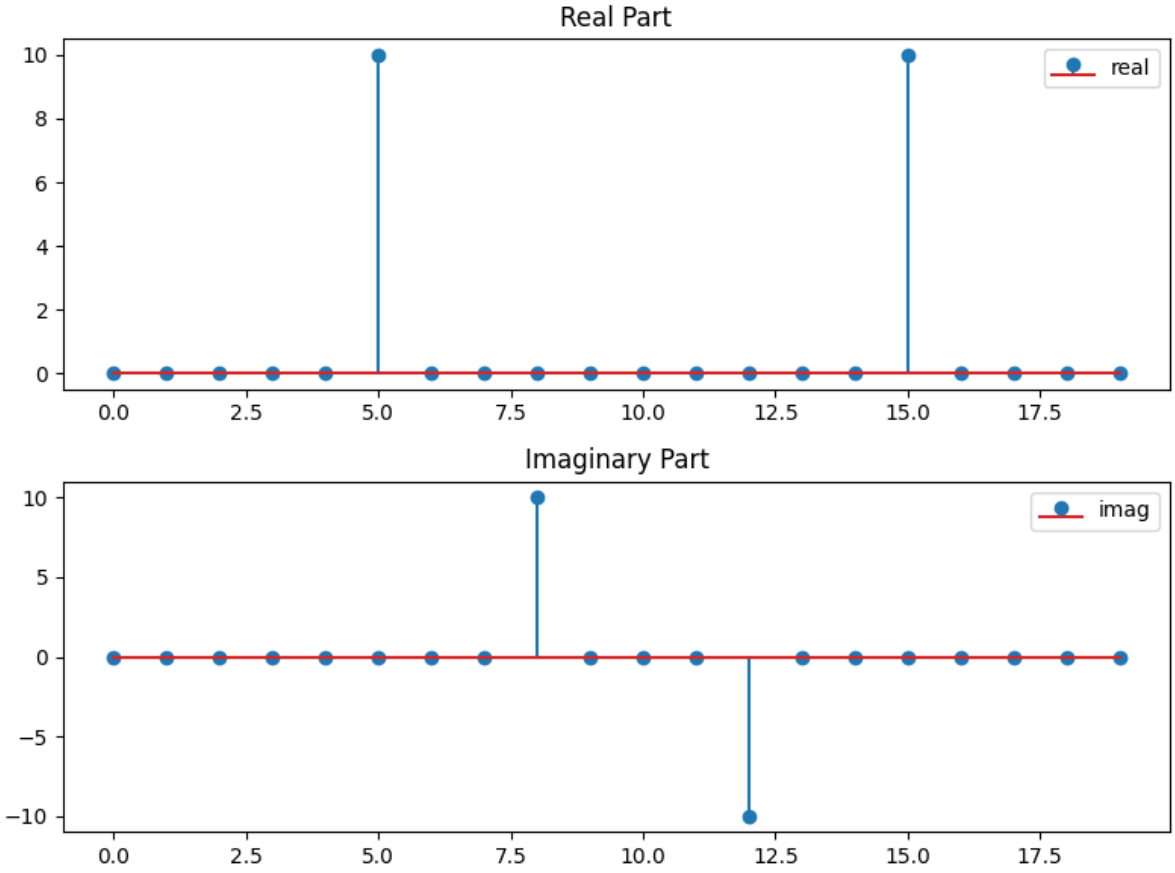
Thus we need to shift the output of the FFT accordingly to get the correct ordering. This can be done using the `fftshift` function from the `scipy.fft` module.

```
# Sample the signal
N = 20
xs = np.linspace(0, L, N, endpoint=False)
fcs = f_clean(xs)
```

(continues on next page)

(continued from previous page)

```
fcs_hat_shift = fftshift(fcs_hat)
fig = plt.figure(figsize=(8, 6))
axs = fig.subplots(2, 1)
axs[0].stem(fcs_hat_shift.real, label='real')
axs[0].set_title('Real Part')
axs[0].legend()

axs[1].stem(fcs_hat_shift.imag, label='imag')
axs[1].set_title('Imaginary Part')
axs[1].legend()
plt.tight_layout()
```



This makes now almost sense :) but right now the x-axis is just the index of the array. It would be nice to adapt the x-axis to the frequency. We can easily obtain the relevant frequencies by using the `fftfreq` function from the `scipy.fft` module.

The `fftfreq` function takes as input the number of sample points and the spacing between the sample points and returns the Discrete Fourier Transform sample frequencies. More precisely, it returns

```
f = [0, 1, ...,   N/2-1,     -N/2, ..., -1] / (d*N)   if N is even
f = [0, 1, ..., (N-1)/2, -(N-1)/2, ..., -1] / (d*N)   if N is odd
```

So in order obtain the **sampling frequencies**, we need to multiply the output of `fftfreq` by the length `d*N` the the interval over which we sampled the signal:

```
freqs = fftfreq(N) # Assumes a sampling width of d = 1
print(freqs)
```

```
[ 0.    0.05  0.1   0.15  0.2   0.25  0.3   0.35  0.4   0.45 -0.5  -0.45
 -0.4  -0.35 -0.3  -0.25 -0.2  -0.15 -0.1  -0.05]
```

```
freqs = fftfreq(N, d=1/N)
print(freqs)
# Equivalent to
freqs = np.fft.fftfreq(N, d=L/N)*L
print(freqs)
```

```
[  0.   1.   2.   3.   4.   5.   6.   7.   8.   9. -10.  -9.  -8.  -7.
  -6.  -5.  -4.  -3.  -2.  -1.]
[  0.   1.   2.   3.   4.   5.   6.   7.   8.   9. -10.  -9.  -8.  -7.
  -6.  -5.  -4.  -3.  -2.  -1.]
```

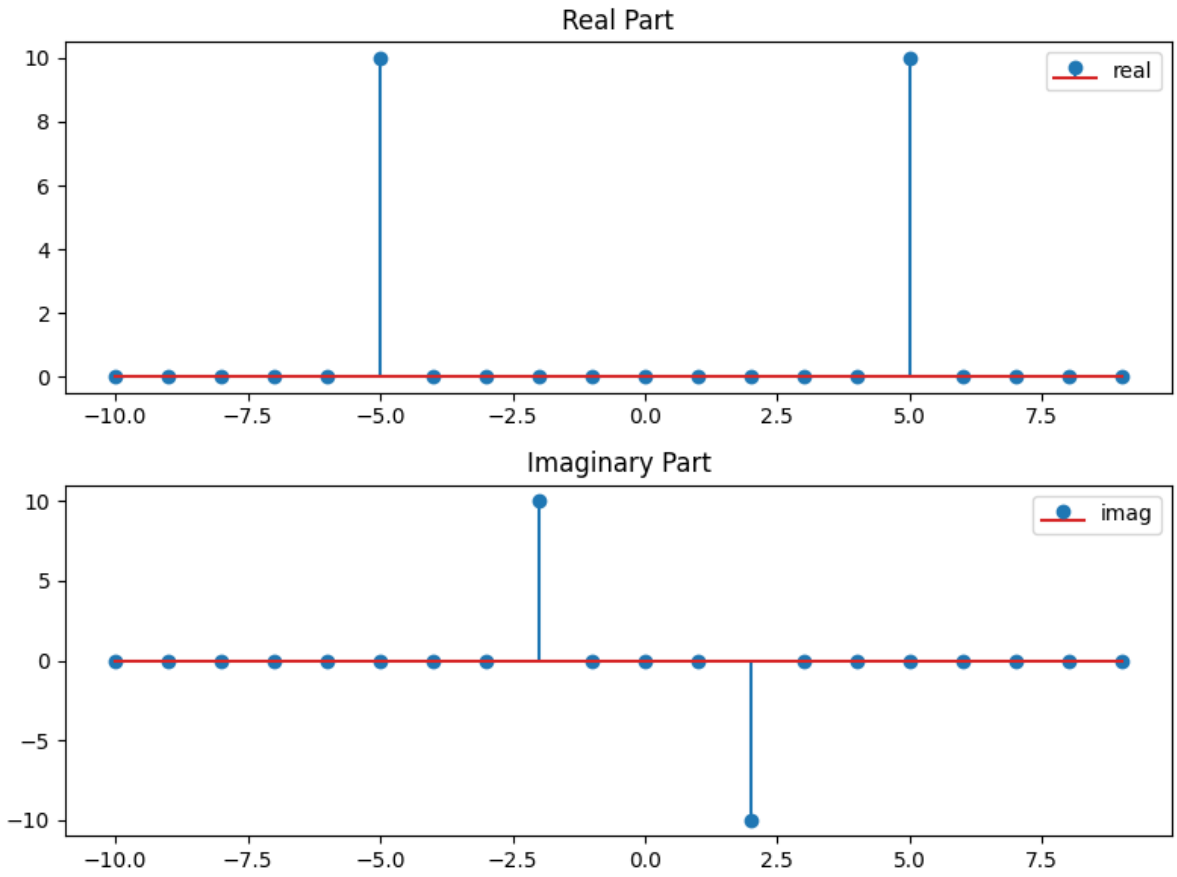
This might not look exactly as the frequencies we expected for N samples, does it? Note for the fundamental frequency f_0 is given by $f_0 = 2\pi/L$ where T is the total time spanned by the signal.

So know we can plot the signal and its Fourier transform in a single plot. But before we again have to shift the frequencies to the correct order.

```
freqs_shift = fftshift(fftfreq(N, d=1/N))
print(freqs_shift)
```

```
[-10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.  -2.  -1.   0.   1.   2.   3.
   4.   5.   6.   7.   8.   9.]
```

```
fig = plt.figure(figsize=(8, 6))
axs = fig.subplots(2, 1)
axs[0].stem(freqs_shift, fcs_hat_shift.real, label='real')
axs[0].set_title('Real Part')
axs[0].legend()

axs[1].stem(freqs_shift, fcs_hat_shift.imag, label='imag')
axs[1].set_title('Imaginary Part')
axs[1].legend()
plt.tight_layout()
```

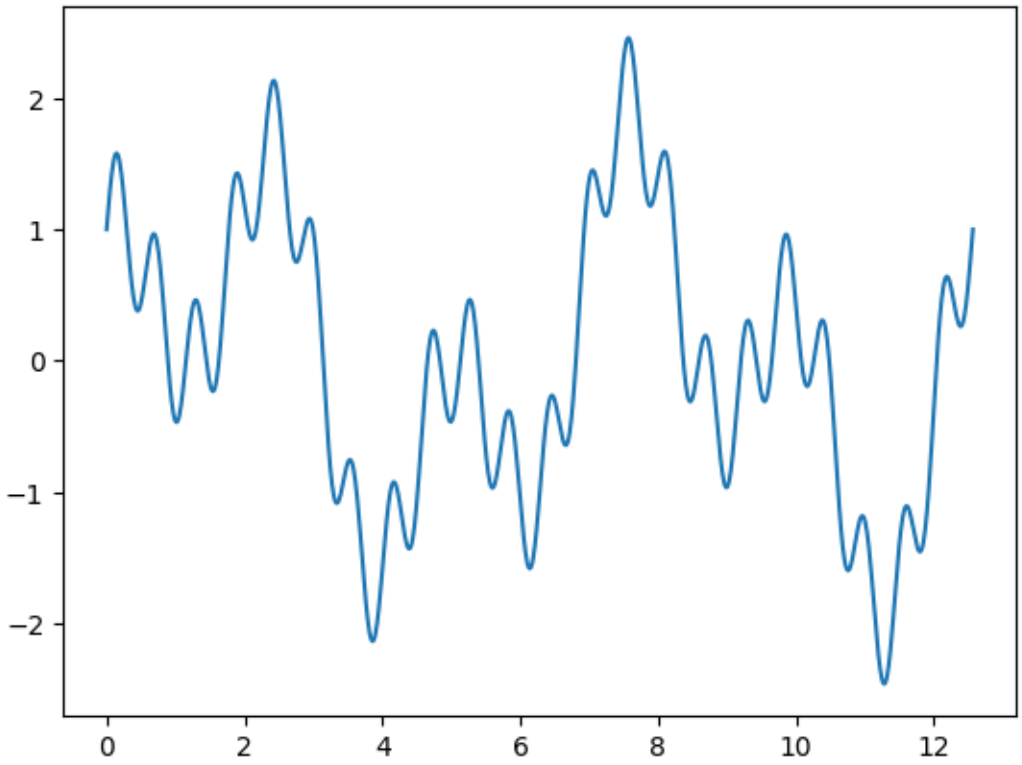


Note the hight of the peaks in the Fourier transform plot and recall that the the `fft` functions return the Fourier coefficients which are not normalized by the number of sample points. If we divide by N we get the correct amplitudes of the Fourier coefficients, which should $1/2$ for a $\cos(2\pi k/Lx)$ or $\sin(2\pi k/Lx)$ type of signal.

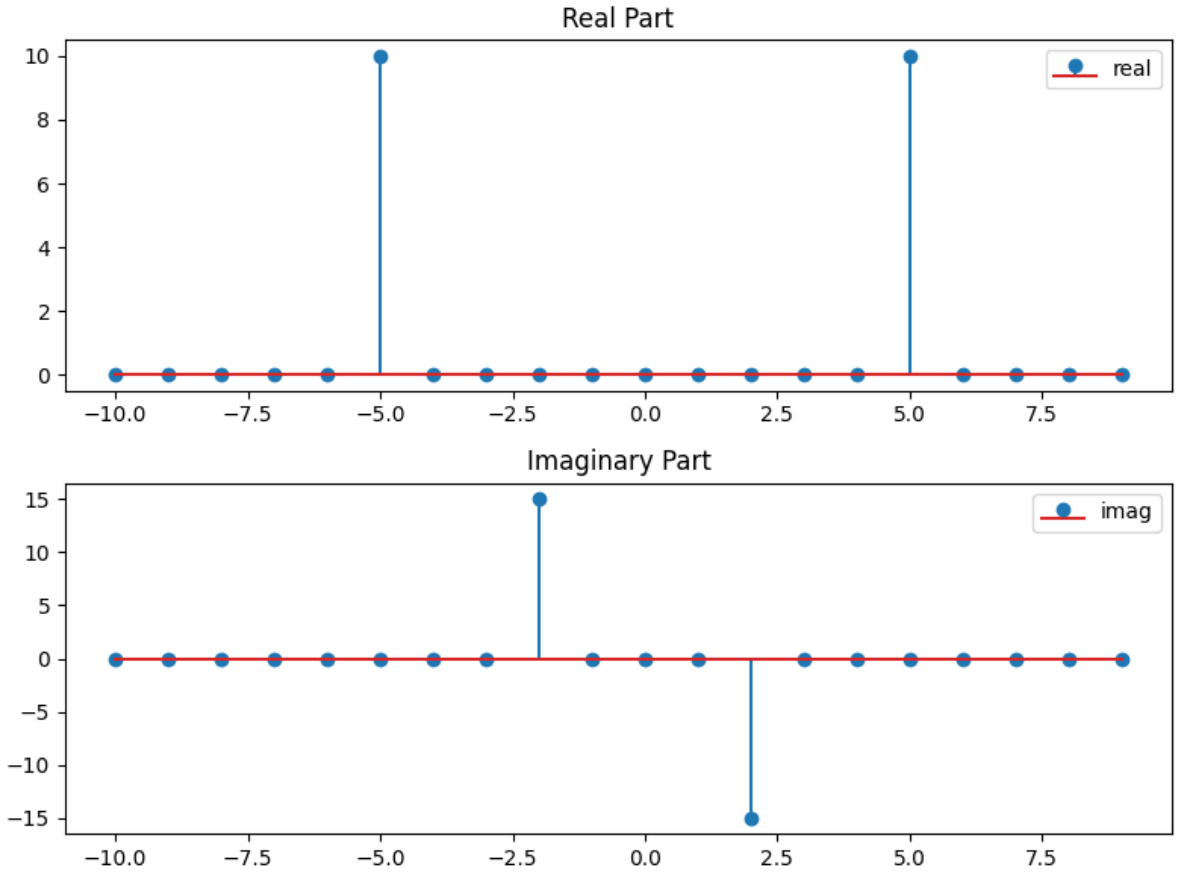
5.6.3 Aliasing and Nyquist frequency

To the previous signal we add a high frequency component to the signal and sample it again with the same sampling rate as before.

```
# For N = 20
N=20
f_alias = lambda x :  0.5*np.sin((2+20)*np.pi/L*2*x) + f_clean(x)

plt.figure()
plt.plot(x, f_alias(x), label='f_alias')
plt.show()
```




```
f_alias_shift = fftshift(fft(f_alias(xs)))

fig = plt.figure(figsize=(8, 6))
axs = fig.subplots(2, 1)
axs[0].stem(freqs_shift, f_alias_shift.real, label='real')
axs[0].set_title('Real Part')
axs[0].legend()

axs[1].stem(freqs_shift, f_alias_shift.imag, label='imag')
axs[1].set_title('Imaginary Part')
axs[1].legend()
plt.tight_layout()
```



We note that we get the same frequencies as for the original clean signal, but the amplitude for the imaginary part is different, namely 1.5 of the original signal. The reason for that boils down to the previously stated orthogonality properties of the trigonometric functions with respect to the discrete inner product:

$$\langle \omega^l, \omega^m \rangle_N = \begin{cases} 1 & \text{if } (l-m)/N \in \mathbb{Z}, \\ 0 & \text{else.} \end{cases}$$

In particular that means, that ω^l and ω^{l+N} cannot be distinguished for a sampling number of N. In other words, ω^{l+N} is an **alias** of ω^l. This is known as **aliasing** and is a common problem in signal processing.

In our concrete example, the function $\sin(2\pi/L 2x)$ and and $0.5\sin((2+20)\pi/L 2x)$ were indistinguishable for the sampling rate of $N = 20$ and thus they appear as the same frequency in the Fourier transform with the amplitudes added.

It can be shown that for signals with a maximal frequency of p, aliasing can be avoided if the sampling rate $> 2p$. This is called the **Nyquist frequency**.

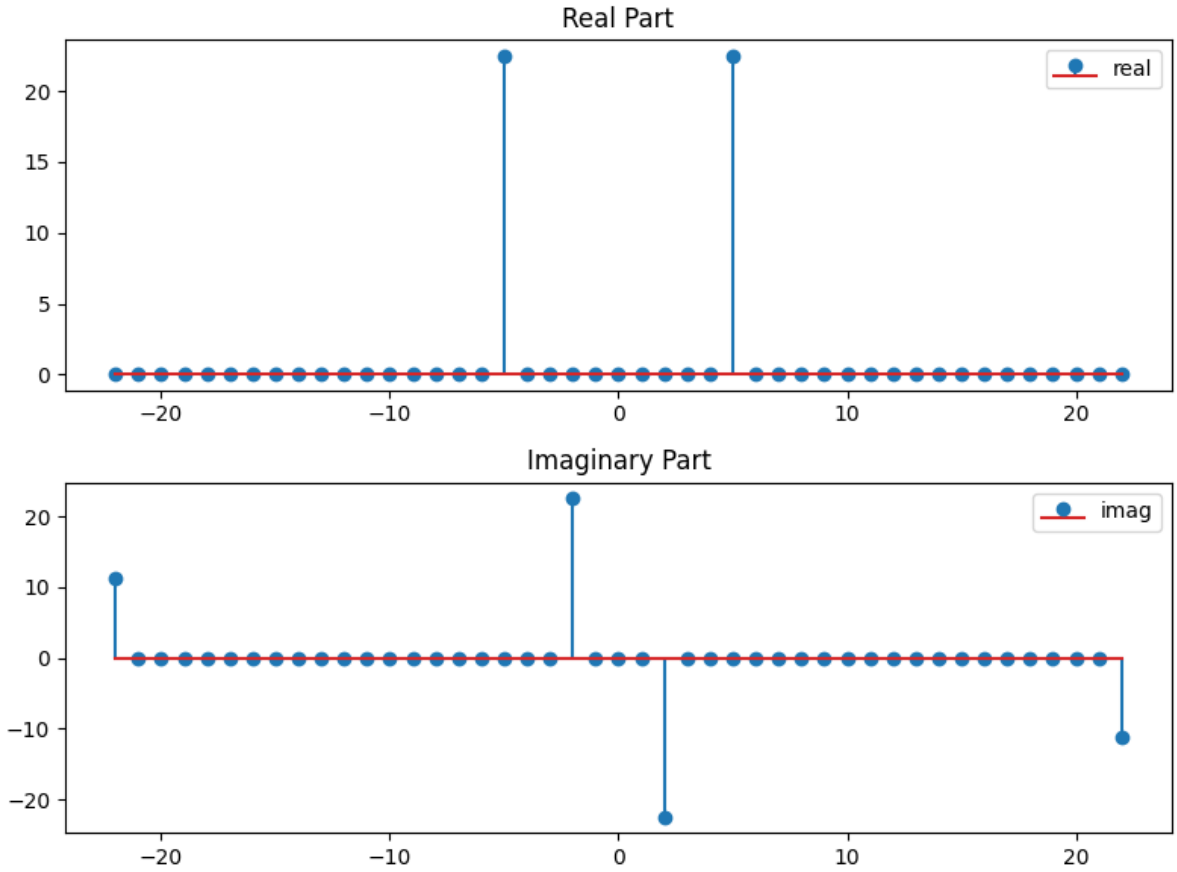
So let's increase the sampling rate to $N = 2 \cdot 22 + 1 = 45$ and sample the signal again.

```
# Sample the signal
N = 45
xs = np.linspace(0, L, N, endpoint=False)
f_alias_shift = fftshift(fft(f_alias(xs)))
freqs_shift = fftshift(fftfreq(N, d=1/N))

fig = plt.figure(figsize=(8, 6))
axs = fig.subplots(2, 1)
axs[0].stem(freqs_shift, f_alias_shift.real, label='real')
```

(continues on next page)

(continued from previous page)

```
axs[0].set_title('Real Part')
axs[0].legend()

axs[1].stem(freqs_shift, f_alias_shift.imag, label='imag')
axs[1].set_title('Imaginary Part')
axs[1].legend()
plt.tight_layout()
```



5.7 Numerical differentiation and spectral derivatives

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fft2, ifft, ifft2, fftfreq, fftshift
import pandas as pd
```

Assume we have a function $f \in C_p^k(0, L)$ (the space of k times continuously differentiable, L-period functions).

Given a set of N equi-distributed sampling points $x_l = lN/L, l = 0, \dots, N-1$ with distance $h = N/L$, we denote by $f = \{f(x_l)\}_{l=0}^{N-1} \in \mathbb{R}^N$ the corresponding vector of the samples of f and write $f(l) = f(x_l)$.

Let's consider the following 4 ways to approximate the derivative of f at x_l:

$$\text{Forward difference } \partial^+ f(l) = \frac{f_{l+1} - f_l}{h} \tag{5.77}$$

$$\text{Backward difference } \partial^- f(l) = \frac{f_l - f_{l-1}}{h} \tag{5.78}$$

$$\text{Central difference } \partial^\circ f(l) = \frac{f_{l+1} - f_{l-1}}{2h} \tag{5.79}$$

$$\text{Spectral derivative } \partial^{\mathcal{F}} f(l) = \mathcal{F}_N^{-1}(i\mathbf{k}\mathcal{F}_N(f))(l) \tag{5.80}$$

Here $\mathbf{k}$ is the wave number vector $2\pi/L(0, 1, \dots, N/2-1, -N/2, -N/2+1, \dots, -1)$,

5.7.1 Aproximation properties of the finite difference operators

Proposition 1

Assuming that the function f is sufficiently differentiable, the following estimates hold:

$$\partial^+ f(x_k) - f'(x_k) = \mathcal{O}(h) = \mathcal{O}(N^{-1}) \tag{5.81}$$

$$\partial^- f(x_k) - f'(x_k) = \mathcal{O}(h) = \mathcal{O}(N^{-1}) \tag{5.82}$$

$$\partial^\circ f(x_k) - f'(x_k) = \mathcal{O}(h^2) = \mathcal{O}(N^{-2}) \tag{5.83}$$

for $h \to 0$ (respectively $N \to \infty$).

Proof. The proof is based on the Taylor expansion of the function f around the point x_k.

1. **Forward difference**: Using Taylor expansion around x_l:

$$f(x_{l+1}) = f(x_l) + hf'(x_l) + \frac{h^2}{2} f''(x_l) + \mathcal{O}(h^3)$$

Therefore,

$$\partial^+ f(x_l) = \frac{f(x_{l+1}) - f(x_l)}{h} = f'(x_l) + \frac{h}{2} f''(x_l) + \mathcal{O}(h^2)$$

Hence,

$$\partial^+ f(x_l) - f'(x_l) = \frac{h}{2} f''(x_l) + \mathcal{O}(h^2) = \mathcal{O}(h)$$

2. **Backward difference**: Using Taylor expansion around x_l:

$$f(x_{l-1}) = f(x_l) - hf'(x_l) + \frac{h^2}{2} f''(x_l) - \mathcal{O}(h^3)$$

Therefore,

$$\partial^- f(x_l) = \frac{f(x_l) - f(x_{l-1})}{h} = f'(x_l) - \frac{h}{2} f''(x_l) + \mathcal{O}(h^2)$$

Hence,

$$\partial^- f(x_l) - f'(x_l) = \frac{h}{2} f''(x_l) + \mathcal{O}(h^2) = \mathcal{O}(h)$$

3. **Central difference**: Using Taylor expansion around x_l:

$$f(x_{l+1}) = f(x_l) + hf'(x_l) + \frac{h^2}{2}f''(x_l) + \frac{h^3}{6}f'''(x_l) + \mathcal{O}(h^4)$$

$$f(x_{l-1}) = f(x_l) - hf'(x_l) + \frac{h^2}{2}f''(x_l) - \frac{h^3}{6}f'''(x_l) + \mathcal{O}(h^4)$$

Therefore,

$$\partial^\circ f(x_l) = \frac{f(x_{l+1}) - f(x_{l-1})}{2h} = f'(x_l) + \mathcal{O}(h^2)$$

Hence,

$$\partial^\circ f(x_l) - f'(x_l) = \mathcal{O}(h^2)$$

5.7.2 Numerical experiments

Next, we implement the 4 numerical differentiation methods and compare their performance on the functions

$$f(x) = 0.1e^{1+\sin(x)} + 0.1\sin(4x)$$

given on the interval $[0, 2\pi)$.

First, for four different sampleing sizes $N = 4, 8, 12, 16$, plot the exact derivate $f'(x)$ and the corresponding 4 different approximations. Afterward, perform a convergence study of the error of the various numerical differentiation methods, for $N = 4, 8, 12, 16, 20, 24, 28, 32, 36$ and tabulate the error and the convergence rate of the numerical differentiation methods. Discuss the results and estimate **theoretically** based on your tabulated values, how large N should be chosen so that the forward/backward/central finite differences obtain a given accuracy as the spectral derivative for $N = 24$.

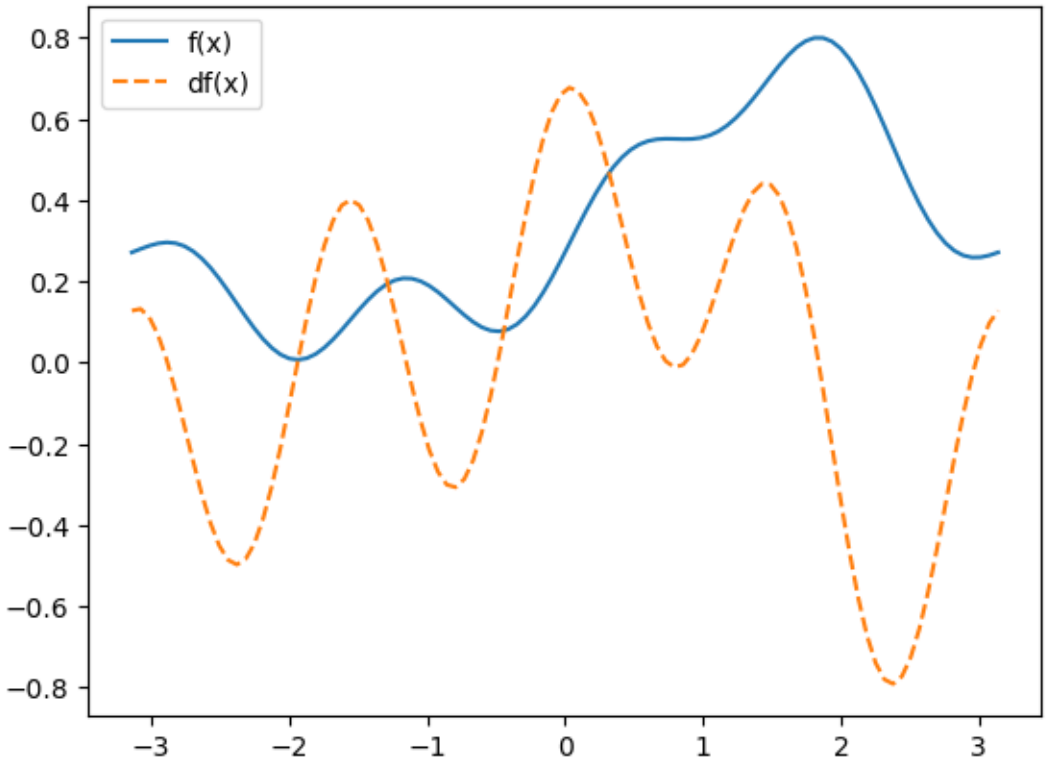
```
# Forward Euler
def df_forward(f, x):
    dx = x[1] - x[0]
    df = np.zeros_like(x)
    df[:-1] = (f(x[1:]) - f(x[:-1]))/dx
    df[-1] = (f(x[0]) - f(x[-1]))/dx
    return df

# Backward Euler
def df_backward(f, x):
    dx = x[1] - x[0]
    df = np.zeros_like(x)
    df[1:] = (f(x[1:]) - f(x[:-1]))/dx
    df[0] = (f(x[0]) - f(x[-1]))/dx
    return df

# Central difference
def df_central(f, x):
    dx = x[1] - x[0]
    df = np.zeros_like(x)
    df[1:-1] = (f(x[2:]) - f(x[:-2]))/(2*dx)
    df[0] = (f(x[1]) - f(x[-1]))/(2*dx)
    df[-1] = (f(x[0]) - f(x[-2]))/(2*dx)
    return df
```

(continues on next page)

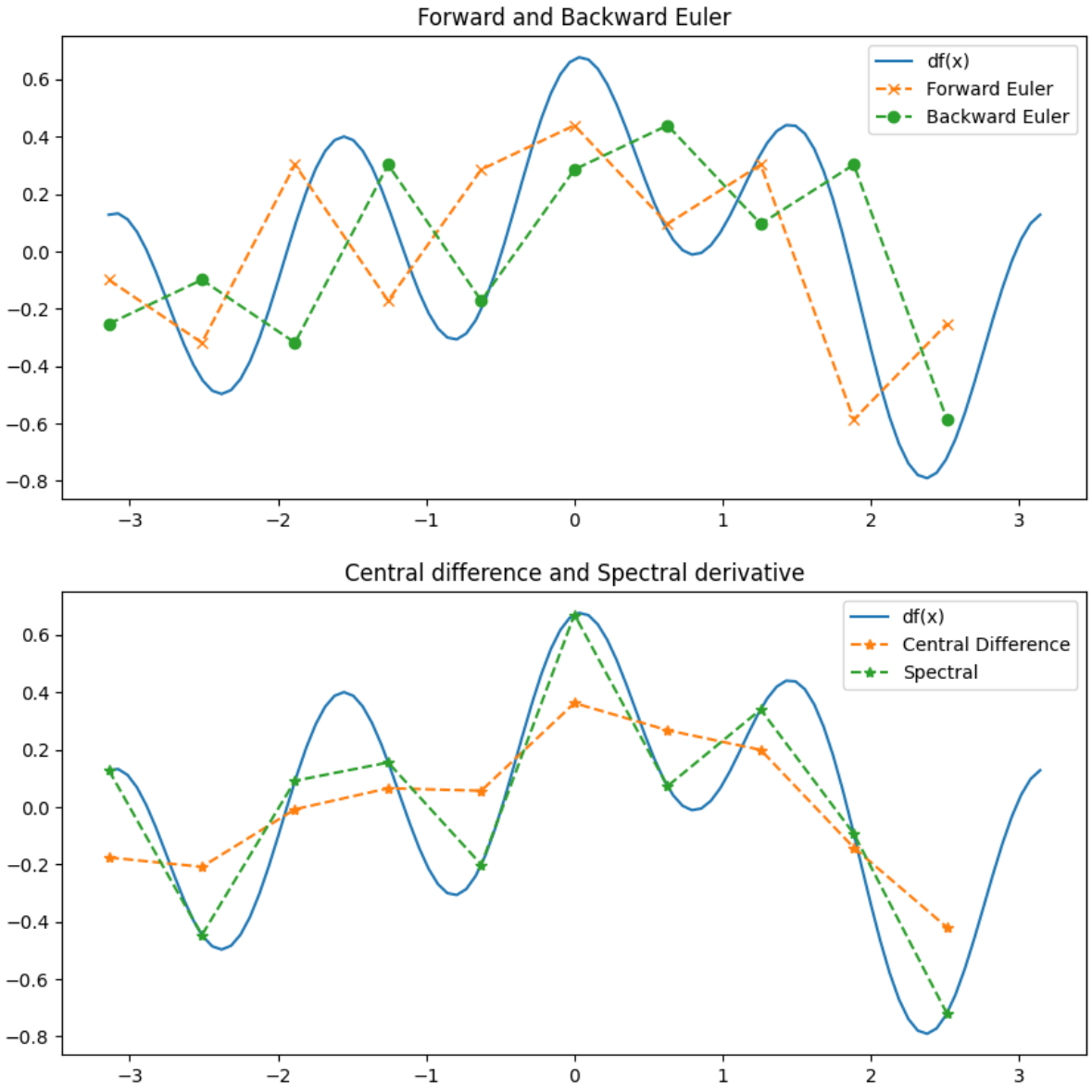
(continued from previous page)

```
# Very important:
# Use arange instead of linspace to obtain half-open intervals
# This is important since we have periodic boundary conditions!
def df_spectral(f, x):
    # You can either do
    # f_hat = fft(f(x))
    # N, dx = len(x), (x[1] - x[0])
    # k = fftfreq(N, d=dx)*2*np.pi
    # df_hat = 1j * k * f_hat
    # df = ifft(df_hat).real
    # return df
    # ... or write a one-liner
    return ifft(1j*fftfreq(len(x), (x[1] - x[0])/(2*np.pi))*fft(f(x))).real
```

```
L = 2*np.pi
# L = np.pi
N = 10
dx = L/N
# Note that we do not need to include the last point
# due to the periodic boundary conditions
x = np.arange(-L/2, L/2, dx)

# Define a function and its derivative
# f = lambda x: np.cos(x) + 0.5*np.sin(4*x)
# df = lambda x: -np.sin(x) + 0.4*np.cos(4*x)

f = lambda x:  0.1*np.exp(1+np.sin(x)) + 0.1*np.sin(4*x)
df = lambda x: 0.1*np.cos(x)*np.exp(1+np.sin(x)) + 0.4*np.cos(4*x)


xfine = np.linspace(-L/2, L/2, 10*N)
# plt.figure(figsize=(10, 6))
plt.plot(xfine, f(xfine), label='f(x)')
plt.plot(xfine, df(xfine), "--", label='df(x)')
plt.legend()
plt.show()
```




```
fig, axes = plt.subplots(2, 1, figsize=(10, 10))

# First subplot: df, Forward Euler, Backward Euler
axes[0].plot(xfine, df(xfine), label='df(x)')
axes[0].plot(x, df_forward(f, x),  "--x", label='Forward Euler')
axes[0].plot(x, df_backward(f, x), "--o", label='Backward Euler')
axes[0].legend()
axes[0].set_title('Forward and Backward Euler')

# Second subplot: df, Central Euler, Spectral
axes[1].plot(xfine, df(xfine), label='df(x)')
axes[1].plot(x, df_central(f, x),  "--*", label='Central Difference')
axes[1].plot(x, df_spectral(f, x),  "--*", label='Spectral')
axes[1].legend()
axes[1].set_title('Central difference and Spectral derivative')

plt.show()
```




```python
f = lambda x: np.exp(1+np.sin(x))
df = lambda x: np.cos(x)*np.exp(1+np.sin(x))

# Try this one afterwards
# f = lambda x: np.sin(x)
# df = lambda x: np.cos(x)

def compute_eoc(f, df, L, N_list, df_num):
    errs = []
    for N in N_list:
        dx = L/N
        x = np.arange(-L/2, L/2, dx)
        errs.append(np.abs( df(x) - df_num(f, x), np.inf).max())
        # print(f'N = {N}, error = {errs[-1]}')
    errs = np.array(errs)
    N_list = np.array(N_list)
```

(continues on next page)

(continued from previous page)

```
    eocs = np.log(errs[1:]/errs[:-1])/np.log(N_list[:-1]/N_list[1:])
    eocs = np.insert(eocs, 0, np.inf)
    return errs, eocs

N_list = [4 + 4*k for k in range(0,9)]
print(N_list)
```

```
[4, 8, 12, 16, 20, 24, 28, 32, 36]
```

```
table = pd.DataFrame(index=N_list)
for method in [df_forward, df_backward, df_central, df_spectral]:
    errs, eocs = compute_eoc(f, df, L, N_list, method)
    table[method.__name__ + " err"] = errs
    table[method.__name__ + " eoc"] = eocs

display(table)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 3
      1 table = pd.DataFrame(index=N_list)
      2 for method in [df_forward, df_backward, df_central, df_spectral]:
----> 3     errs, eocs = compute_eoc(f, df, L, N_list, method)
      4     table[method.__name__ + " err"] = errs
      5     table[method.__name__ + " eoc"] = eocs

Cell In[5], line 13, in compute_eoc(f, df, L, N_list, df_num)
     11     dx = L/N
     12     x = np.arange(-L/2, L/2, dx)
---> 13     errs.append(np.abs( df(x) - df_num(f, x), np.inf).max())
     14     # print(f'N = {N}, error = {errs[-1]}')
     15 errs = np.array(errs)

TypeError: return arrays must be of ArrayType
```

Discussion.

- The forward and backward difference operators have a first order convergence rate, while the central difference operator has a second order convergence rate. This can be clearly seen in the convergence study, where double the number of sampling points reduces the error by a factor of 4 for the central difference operator, but only by a factor of 2 for the forward and backward difference operators.
- The spectral derivative has a much higher convergence rate than the finite difference operators. For $N = 28$, the error of the spectral derivative is already roughly at machine precisison. Consequently, the error of the spectral derivative cannot get smaller for N roughly larger than 24, which is why we don't observe any positive convergence rate for $N > 28$.

5.8 Solving PDEs with the Fourier spectral method in 2D

We will discuss the Fourier spectral method for solving PDEs and focus on the 2D Poisson equation and the heat equation.

5.8.1 Fourier techniques in 2D

As in the one-dimensional, we can perform a Fourier expansion of the solution $u(x, y)$, where the Fourier coefficients in 2D are simply given by computing 1D Fourier coefficients in each direction.

$$\hat{u}(k_x, k_y) = \frac{1}{L_x L_y} \int_0^{L_x} \int_0^{L_y} u(x, y) e^{-i2\pi k_x/L_x x} e^{-i2\pi k_y/L_y y} \, \mathrm{d}x \, \mathrm{d}y \tag{5.84}$$

$$= \frac{1}{L_x L_y} \int_0^{L_x} \int_0^{L_y} u(x, y) e^{-i2\pi (k_x/L_x, k_y/L_y)\cdot(x,y)} \, \mathrm{d}x \, \mathrm{d}y \tag{5.85}$$

$$= \frac{1}{|\Omega|} \int_\Omega u(\mathbf{x}) e^{-i2\pi \mathbf{k}\cdot\mathbf{x}} \, \mathrm{d}\mathbf{x} \tag{5.86}$$

with $\mathbf{x} = (x, y)$ and $\mathbf{k} = (k_x/L_x, k_y/L_y)$.

Being a bit in rush :), we simply summarize here that one can basically develop many of the Fourier techniques we discussed for the 1D case in the 2D case, just simply applying all the relevant concepts to each spatial direction separately.

For example, the formal 2D Fourier series of a function $f(x, y)$ is given by

$$u(x, y) \sim \sum_{k_x=-\infty}^{\infty} \sum_{k_y=-\infty}^{\infty} \hat{u}(k_x, k_y) e^{i2\pi(k_x x/L_x + k_y y/L_y)} \tag{5.87}$$

$$= \sum_{\mathbf{k}\in\mathbb{Z}^2} \hat{u}(\mathbf{k}) e^{i2\pi \mathbf{k}\cdot\mathbf{x}} \tag{5.88}$$

where the 2D Fourier coefficients are defined as above.

As in the 1D case, we can then derive the following identities for derivatives of periodic functions and their corresponding Fourier coefficients:

$$(\partial_x u(x, y))^\wedge(k_x, k_y) = -i2\pi k_x/L_x \hat{f}(k_x, k_y) \tag{5.89}$$

$$(\partial_y u(x, y))^\wedge(k_x, k_y) = -i2\pi k_y/L_y \hat{f}(k_x, k_y) \tag{5.90}$$

$$(\partial_{xx} u(x, y))^\wedge(k_x, k_y) = -(2\pi k_x/L_x)^2 \hat{f}(k_x, k_y) \tag{5.91}$$

$$(\partial_{yy} u(x, y))^\wedge(k_x, k_y) = -(2\pi k_y/L_y)^2 \hat{f}(k_x, k_y) \tag{5.92}$$

$$(\Delta u(x, y))^\wedge(k_x, k_y) = -(2\pi k_x/L_x)^2 \hat{f}(k_x, k_y) - (2\pi k_y/L_y)^2 \hat{f}(k_x, k_y) \tag{5.93}$$

Due to the appearance of the factors $2\pi/L_x$ and $2\pi/L_x$ it also very common to ease the notation by defining the wavenumber vector $\tilde{\mathbf{k}}$ as

$$\tilde{\mathbf{k}} = (\tilde{k}_x, \tilde{k}_y) = 2\pi(k_x/L_x, k_y/L_y). \tag{5.94}$$

This way, the Laplacian in Fourier space becomes simply

$$(\Delta u)^\wedge(k_x, k_y) = -|\tilde{\mathbf{k}}|^2 \hat{u}(\mathbf{k}).$$

Let's use this now to solve the Poisson equation in 2D.

5.8.2 2D Poisson equation

Let's consider the 2D Poisson equation is given by

$$-\Delta u(x,y) = -(\partial_{xx} + \partial_{yy})u(x,y) = f(x,y)$$

on a domain $\Omega = [0, L_x) \times [0, L_y)$ supplemented with periodic boundary conditions.

To solve this equation on a continuous level, we can do a Fourier expansion of the solution $u(x,y)$ and the right-hand side $f(x,y)$, and then solve for the Fourier coefficients, exactly as you did in the 1D case back in the Matte 4K course. Then the Fourier coefficients of the solution are given by

$$\hat{u}(\mathbf{k}) = \frac{\hat{f}(\mathbf{k})}{|\tilde{\mathbf{k}}|^2}$$

Note that this can only be done if the wavenumber vector $\tilde{\mathbf{k}}$ is not zero, i.e., $\tilde{\mathbf{k}} \neq 0$. But observe that the Fourier coefficients for $(k_x, k_y) = (0,0)$

$$\hat{u}(\mathbf{0}) = \frac{1}{|\Omega|}\int_\Omega u(x,y)\,\mathrm{d}x\,\mathrm{d}y$$

is simply the **mean value** of the solution $u(x,y)$ over the domain Ω. The division by 0 "problem" is directly related to the fact that there is a ambiguity in the solution of the Poisson equation, since the Laplacian of a constant (and thus periodic!) function is zero, and therefore for any solution u the function $u + c$ is also a solution, and thus the solution is only determined up to a constant. To eliminate this ambiguity, the convention is to prescribe the mean value of the solution, for instance to zero, i.e., by requiring $\int_\Omega u(x,y)\,\mathrm{d}x\,\mathrm{d}y = 0$. This will then uniquely determine the zero mode $\hat{u}(\mathbf{0})$.

Note that this is not a problem if you e.g. want to solve the Poisson problem with a lower order term, i.e., if you have a Poisson equation of the form

$$-\Delta u + cu = f.$$

for some constant $c > 0$, since then the solution on the Fourier side is given by

$$\hat{u}(\mathbf{k}) = \frac{\hat{f}(\mathbf{k})}{(c + |\tilde{\mathbf{k}}|^2)}$$

This make sense, because in this case, adding a non-zero constant function to u will change the right-hand side of the Poisson equation and thus the solution is unique.

We will now exploit these formulas and ideas numerically, and solve the Poisson equation numerically by using the (2 dimensional) fast fourier transform (FFT) to approximate the Fourier coefficients of the right-hand side $f(x,y)$, divide then by the norm of the wavenumber vector, and then use the inverse FFT to compute the solution $u(x,y)$. This is the so-called **Fourier spectral method**. Le't see how this works in the following code.

```
# %matplotlib widget
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fft2, ifft, ifft2, fftfreq, fftshift
import pandas as pd
```

First, we consider a periodic function $u(x,y)$ on the domain $\Omega = [0, 2\pi) \times [0, 2\pi)$ for which we can easily compute the corresponding right-hand side $f(x,y)$.

Let's use the following function

$$u(x,y) = \sin(x)\cos(2y)$$

We can easily compute the right-hand side $f(x, y)$ by inserting this function into the Poisson equation and obtaining

$$-\Delta u(x, y) = -(\partial_{xx} + \partial_{yy})u(x, y) = 5\sin(x)\cos(2y)$$

We could also say the the given u is an eigenfunction of the Laplacian with eigenvalue -5. Let's start by plotting the function $u(x, y)$ and the right-hand side $f(x, y)$.

Here, we need some constructs from the `numpy` and `matplotlib` libraries.

```
# Define the domain
Lx, Ly = 2*np.pi, 2*np.pi

# Define 1d samplings for x and y directions
# Nx, Ny = 32, 32
Nx, Ny = 3, 4
x = np.linspace(-Lx/2, Lx/2, Nx, endpoint=False)
y = np.linspace(-Ly/2, Ly/2, Ny, endpoint=False)

# Generate a 2d sampling grid to be evaluate functions of x and y
X, Y = np.meshgrid(x, y)
# X, Y = np.meshgrid(x, y, sparse=True)
print(f"X = {X}")
print(f"Y = {Y}")
```

```
X = [[-3.14159265 -1.04719755  1.04719755]
 [-3.14159265 -1.04719755  1.04719755]
 [-3.14159265 -1.04719755  1.04719755]
 [-3.14159265 -1.04719755  1.04719755]]
Y = [[-3.14159265 -3.14159265 -3.14159265]
 [-1.57079633 -1.57079633 -1.57079633]
 [ 0.          0.          0.        ]
 [ 1.57079633  1.57079633  1.57079633]]
```

```
# We can also define a sparse 2d sampling grid
X, Y = np.meshgrid(x, y, sparse=True)
print(f"X = {X}")
print(f"Y = {Y}")
```

```
X = [[-3.14159265 -1.04719755  1.04719755]]
Y = [[-3.14159265]
 [-1.57079633]
 [ 0.        ]
 [ 1.57079633]]
```

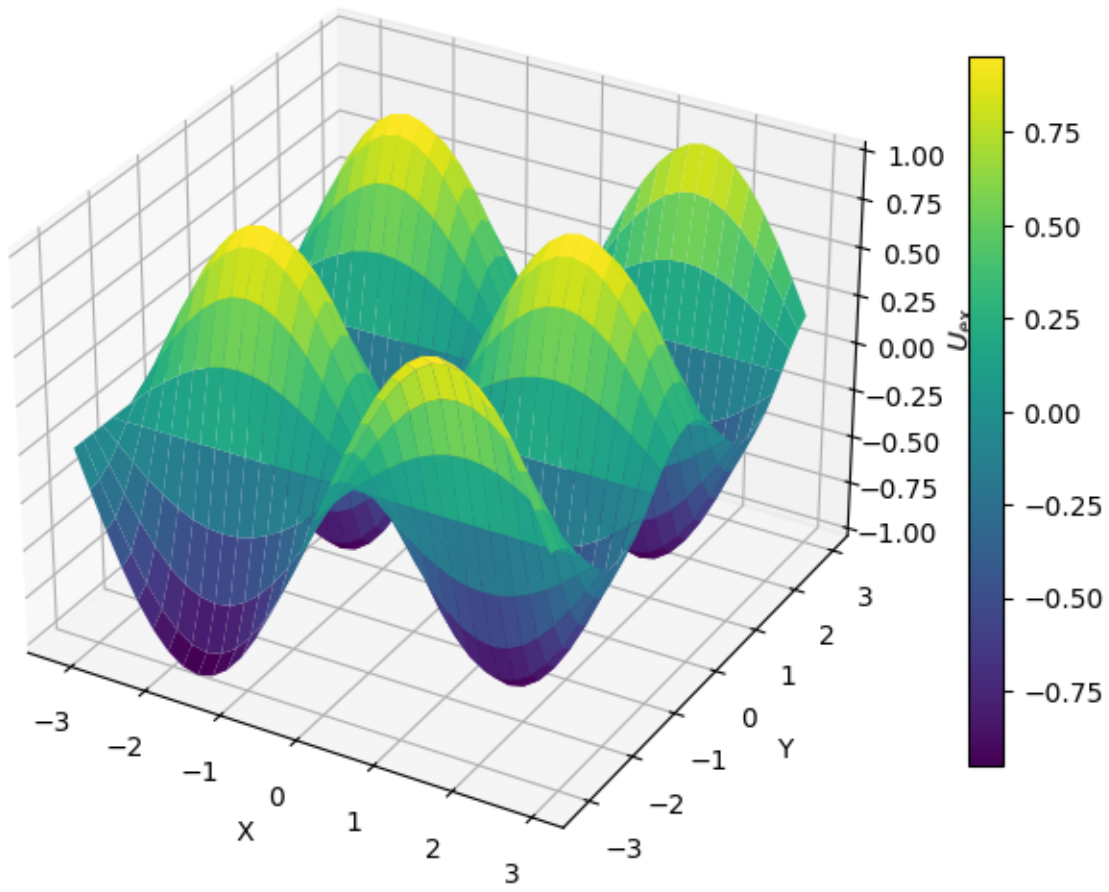
Now that we have understood how the `meshgrid` arrays look like, let's use a finer mesh.

```
Nx, Ny = 32, 32
x = np.linspace(-Lx/2, Lx/2, Nx, endpoint=False)
y = np.linspace(-Ly/2, Ly/2, Ny, endpoint=False)

# Generate a 2d sampling grid to be evaluate functions of x and y
X, Y = np.meshgrid(x, y, sparse=True)

# Define the solution
def u_ex(x, y):
    return np.sin(x)*np.cos(2*y)
```

(continues on next page)

(continued from previous page)

```
# Evaluate the exact solution on the grid
U_ex = u_ex(X, Y)

# Plot the exact solution as surface plot
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, U_ex, cmap='viridis', antialiased=True)
fig.colorbar(surf, shrink=0.6)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel(r'$U_\mathrm{ex}$')
plt.show()
```



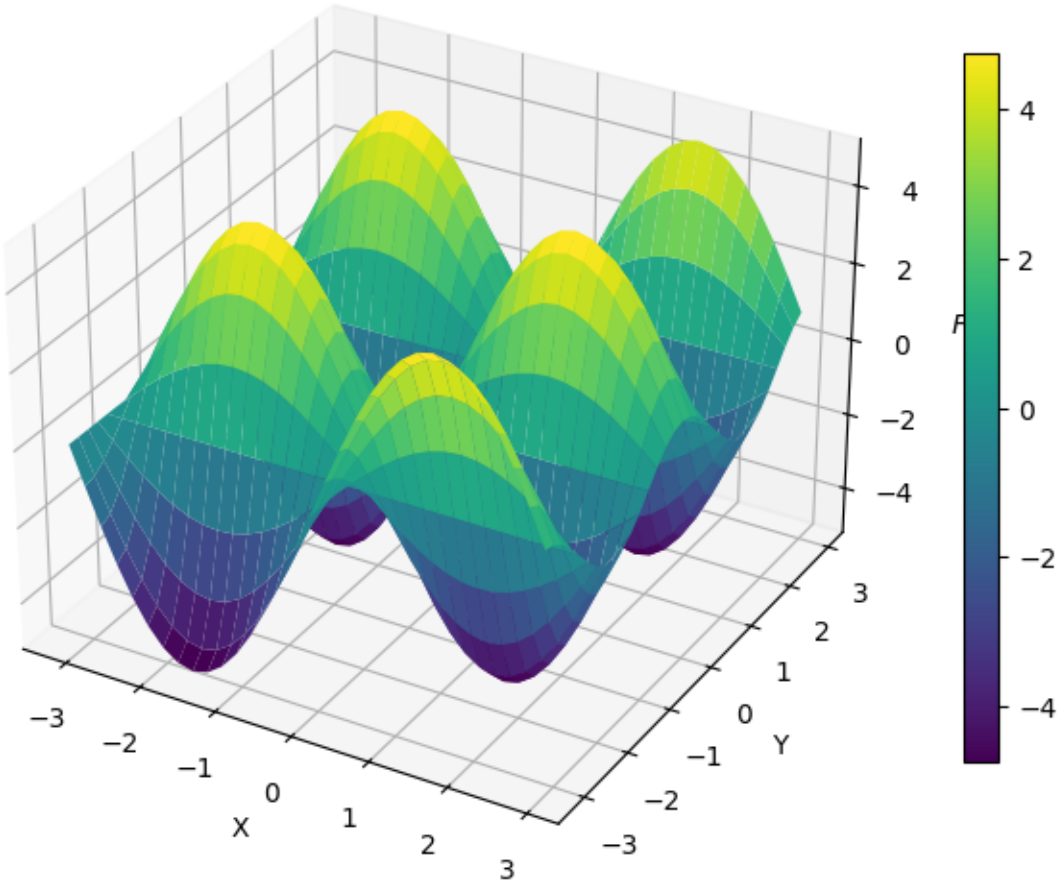
Let'a also plot the right-hand side $f(x, y)$.

```
# Define the solution
def f(x, y):
    return 5*np.sin(x)*np.cos(2*y)

# Evaluate the exact solution on the grid
F = f(X, Y)
```

(continues on next page)

(continued from previous page)

```
# Plot the exact solution as surface plot
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, F, cmap='viridis', antialiased=True)
fig.colorbar(surf, shrink=0.6)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel(r'$F$')
plt.show()
```



Now we write a tiny function consisting of basically 10 lines of which computes the solution $u(x, y)$ **on the Fourier side** by using the formula

$$\hat{u}(\mathbf{k}) = \frac{\hat{f}(\mathbf{k})}{|\tilde{\mathbf{k}}|^2}.$$

```
def solve_poisson(F, Lx, Ly, Nx, Ny):

    # Compute the FFT of the right-hand side using the 2D FFT
    F_hat = fft2(F)

    # Compute wave number grid
```

(continues on next page)

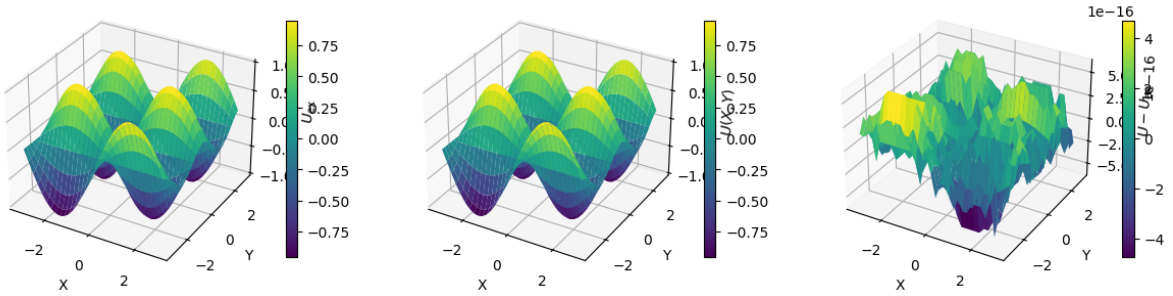
(continued from previous page)

```
    kx = fftfreq(Nx, d=Lx/Nx)*2*np.pi
    ky = fftfreq(Ny, d=Ly/Ny)*2*np.pi
    KX, KY = np.meshgrid(kx, ky, sparse=True)

    # Compute the Poisson operator in Fourier space
    K2 = KX**2 + KY**2

    # Just modified to avoid division by zero
    # We set the zero frequency component to 0 explicitly below
    K2[0, 0] = 1

    # Solve the Poisson equation in Fourier space
    U_hat = F_hat / K2

    # Set the zero frequency component to zero
    # This corresponds to setting the average value of the solution to zero
    U_hat[0, 0] = 0

    # Compute the inverse 2D FFT to get the solution
    return U_hat
```

Let's apply this function and plot the exact solution $U_{\text{ex}}(x, y)$, the numerical solution and the error $U - U_{\text{ex}}$.

```
U_hat = solve_poisson(F, Lx, Ly, Nx, Ny)
U = ifft2(U_hat).real

# Plot the solution
fig = plt.figure(figsize=(15, 5))
ax = fig.add_subplot(131, projection='3d')
surf = ax.plot_surface(X, Y, U_ex, cmap='viridis', antialiased=True)
fig.colorbar(surf, shrink=0.6)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel(r'$U_\mathrm{ex}$')

ax = fig.add_subplot(132, projection='3d')
surf = ax.plot_surface(X, Y,  U, cmap='viridis')
fig.colorbar(surf, shrink=0.6)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel(r'$U(X, Y)$')

U_err = U - U_ex
print(f"Error norm: {np.abs(U_err).max()}")
ax = fig.add_subplot(133, projection='3d')
surf = ax.plot_surface(X, Y,  U_err, cmap='viridis')
fig.colorbar(surf, shrink=0.6)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel(r'$U-U_{\mathrm{ex}}$')
plt.show()
```

```
Error norm: 6.106226635438361e-16
```



Let's try a more complicated manufactured solution. Since computing the rhs is tedious and error-prone, we will use the sympy library to compute the rhs. We will pass the exact solution as string, the following function will then compute the rhs for us, and return the exact solution and the rhs as `numpy` compatible functions.

```
# Exact solution and its -Laplacian
def manufacture_solution_poisson(u_str):
    """
    Generate the exact solution and its corresponding right-hand side for the Poisson␣
 ↪equation.

    This function takes a string representation of the exact solution `u(x, y)` and␣
 ↪computes
    its Laplacian to generate the corresponding right-hand side `f(x, y)` for the␣
 ↪Poisson equation.
    The function returns `u(x, y)` and `f(x, y)` as `numpy`-compatible callable␣
 ↪functions.

    Parameters:
        u_str (str): A string representation of the exact solution `u(x, y)`.

    Returns:
        tuple: A tuple containing two functions:
            - u (function): The exact solution `u(x, y)` as a `numpy`-compatible␣
 ↪function.
            - f (function): The right-hand side `f(x, y)` as a `numpy`-compatible␣
 ↪function.
    """
    import sympy as sy
    from sympy import sin, cos, exp
    x, y = sy.symbols('x y')
    u_sy = eval(u_str)
    laplace = lambda u: sy.diff(u, x, x) + sy.diff(u, y, y)
    f_sy = -sy.simplify(laplace(u_sy))
    print(f'u = {u_sy}')
    print(f'f = {f_sy}')
    u = sy.lambdify((x, y), u_str, modules='numpy')
    f = sy.lambdify((x, y), f_sy, modules='numpy')
    return u, f

# u_ex_str = 'sin(x)*cos(2*y)'
u_ex_str = 'exp(sin(x)) + cos(2*y)'
u_ex, f = manufacture_solution_poisson(u_ex_str)
```

```
u = exp(sin(x)) + cos(2*y)
f = -(-sin(x) + cos(x)**2)*exp(sin(x)) + 4*cos(2*y)
```

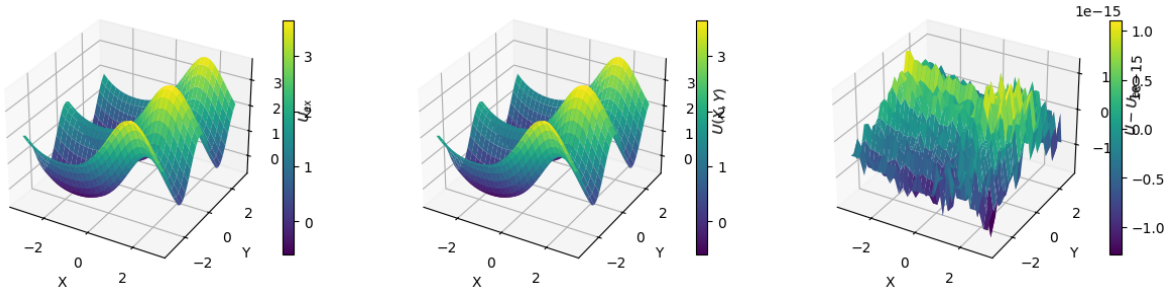
Let's solve the Poisson equation with the new manufactured solution.

```
# Example usage
Lx, Ly = 2*np.pi, 2*np.pi
# Nx, Ny = 10,10,
Nx, Ny = 32, 32
x = np.linspace(-Lx/2, Lx/2, Nx, endpoint=False)
y = np.linspace(-Ly/2, Ly/2, Ny, endpoint=False)

# Generate a 2d sampling grid to be evaluate functions of x and y
X, Y = np.meshgrid(x, y, sparse=True)

F = f(X, Y)
U_hat = solve_poisson(F, Lx, Ly, Nx, Ny)
U = ifft2(U_hat).real
# Re-adjust solution to same mean value as exact solution
# Only necessary for comparison with manufactured solution
# which does not have zero mean
U_ex = u_ex(X, Y)
U += np.mean(U_ex)

U_err = U - U_ex
err = np.abs(U_err).max()
print(f'Error: {err}')
```

```
Error: 1.7763568394002505e-15
```

And let's plot the exact solution, the numerical solution and the error.

```
# Plot the solution
fig = plt.figure(figsize=(15, 5))
ax = fig.add_subplot(131, projection='3d')
surf = ax.plot_surface(X, Y, U_ex, cmap='viridis', antialiased=True)
fig.colorbar(surf, shrink=0.6)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel(r'$U_\mathrm{ex}$')

ax = fig.add_subplot(132, projection='3d')
surf = ax.plot_surface(X, Y,  U, cmap='viridis')
fig.colorbar(surf, shrink=0.6)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel(r'$U(X, Y)$')

ax = fig.add_subplot(133, projection='3d')
surf = ax.plot_surface(X, Y,  U_err, cmap='viridis')
fig.colorbar(surf, shrink=0.6)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel(r'$U-U_{\mathrm{ex}}$')
plt.show()
```

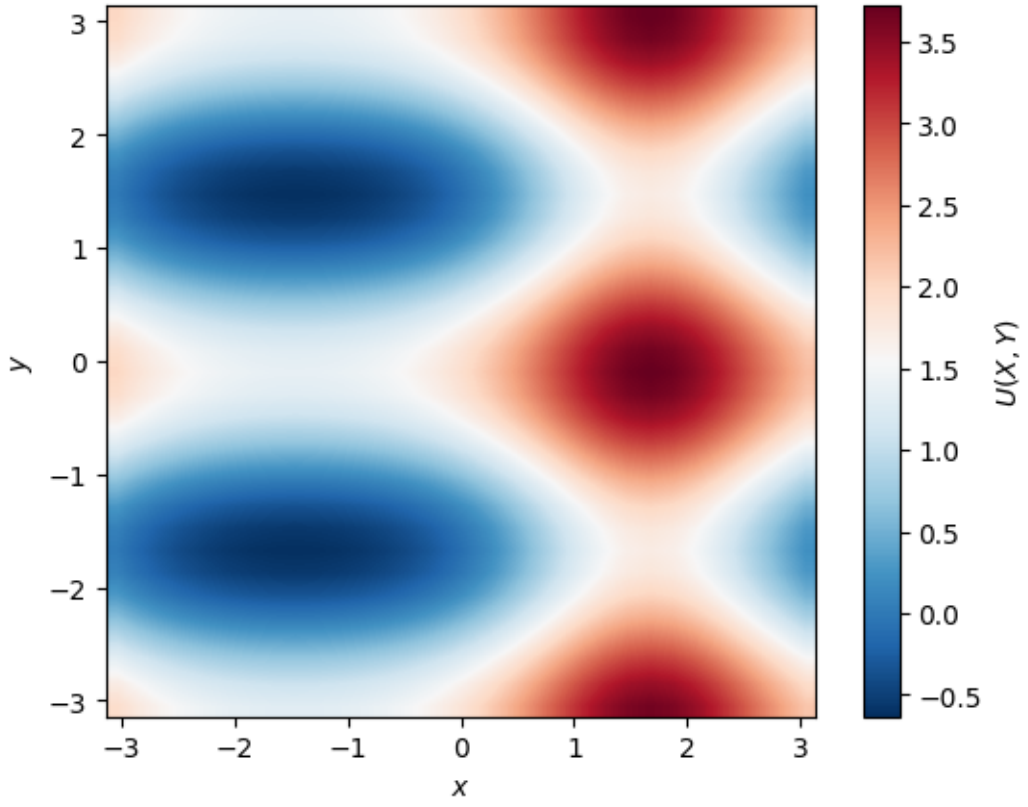


A last thing: Since the 3D surface plots are rather slow, there are not really suitable when visualizing solutions on fine grids or a lot of snapshots. You can use the `imshow_plot_u` function from our homecooked little wrapper module `project_tools` to plot the solution as a 2D image:

```
import os
import os.path
import sys
# Add path to project_tools.py to Python's search path
project_tools_path = os.path.join(os.getcwd(), '../project_3_2025')
if project_tools_path not in sys.path:
    sys.path.append(project_tools_path)
import project_tools as pot
```

```
pot.imshow_plot_u(U, Lx, Ly, cblabel=r'$U(X, Y)$')
```

```
(<Figure size 640x480 with 2 Axes>, <Axes: xlabel='$x$', ylabel='$y$'>)
```



5.9 A Fourier spectral solver for the heat equation

5.9.1 The heat equation

Next, we discuss how we can combine the Fourier spectral method with the time-stepping methods we have discussed in the previous notebooks. We will consider the heat equation

$$\partial_t u(x,y,t) - \kappa \Delta u(x,y,t) = g(x,y,t), \quad (x,y) \in \Omega, \quad t \in (0,T), \tag{5.95}$$

As before, the problem is to be solved on a given rectangular domain $\Omega = [0, L_x) \times [0, L_y) \subset \mathbb{R}^2$ with periodic boundary conditions, i.e. the same considerations as for the Poisson equation apply. Moreover, the problem is supplemented with **initial conditions** $u(0,x,y) = u_0(x,y)$, which we also assume to satisfy the periodic boundary conditions.

Here, the unknown function u represents the temperature distribution, and κ is the thermal diffusivity which assume to be constant.
The right-hand side g is a given heat source.

Physically, the heat equation can be derived from energy conservation principles. One start from the assumption that change of the total (heat-related) energy in any given domain Ω is equal to the total heat flux $\mathbf{q}$ into the domain and energy produced by a volumetric heat source. This leads to the equation

$$\frac{d}{dt}\int_\Omega u \,\mathrm{d}V = -\int_{\partial\Omega} \mathbf{q}\cdot n \,\mathrm{d}S + \int_\Omega g \,\mathrm{d}V,$$

Here the sign is chosen such that the right-hand side is positive when heat is entering into the domain through the surface.

Since the domain is constant, we can move the time derivative inside the integral. Morover using the divergence theorem, we can rewrite the surface integral as a volume integral to arrive at

$$\int_\Omega \partial_t u \,\mathrm{d}V = \int_{\partial\Omega} (g - \nabla\cdot\mathbf{q}) \,\mathrm{d}V$$

Since the domain is arbitrary, we can conclude that the integrands must be equal, which leads us to

$$\partial_t u + \nabla\cdot\mathbf{q} = g$$

Finally, a constitutive relation between the heat flux $\mathbf{q}$ and the temperature gradient ∇u is needed. This is given by Fourier's law of heat conduction, which states that the heat flux can be expressed as $\mathbf{q} = -\kappa\nabla u$, i .e. the flux is proportional to the temperature gradient, and the constant of proportionality is the thermal diffusivity κ. The sign is chosen such that heat flows from hot to cold regions. This leads to the heat equation

$$\partial_t u - \nabla\cdot(\kappa\nabla u) = \partial_t u - \kappa\Delta u = g$$

if the diffusivity κ is constant.

5.9.2 A first attempt to solve the heat equation

Assuming a rectangular domain $\Omega = [0, L_x) \times [0, L_y)$ and periodic boundary conditions, we can use the Fourier spectral method to solve the heat equation. Developing both the solution the right-hand side in a Fourier series, and using the facts that

- the Laplacian of a Fourier mode is proportional to the square of the wavenumber, and
- the operations $\partial_t(\cdot)$ and $\widehat{(\cdot)}$ commute for sufficiently well-behaving functions,

we find that

$$\widehat{\partial_t u}(k_x, k_y, t) = \partial_t \hat{u}(k_x, k_y, t) = -\kappa|\tilde{\mathbf{k}}|^2 \hat{u}(k_x, k_y, t) + \hat{g}(k_x, k_y, t) \tag{5.96}$$

which is now just an ordinary differential equation for the Fourier coefficients $\hat{u}$. Here, $\tilde{\mathbf{k}} = 2\pi(k_x, k_y)$ is the wavenumber vector, and $|\tilde{\mathbf{k}}|^2 = 4\pi^2(k_x^2 + k_y^2)$ its norm squared.

Of course, when we discretize this equation in space, we use the discrete Fourier transform instead. What kind of time-stepping method should we use? Let's try the simplest one, the **forward** or **explicit** Euler method, assuming a time step size of $\tau = T/N_t$. That leads to the following scheme: Given the solution $\hat{u}^n$ at time $t_n = n\tau$, we compute the right-hand side $g^n = g(x, y, t_n)$, and then we compute the Fourier coefficients $\hat{u}^{n+1}$ at time $t_{n+1} = (n+1)\tau$ according to

$$\hat{u}^{n+1}(k_x, k_y) = \hat{u}^n(k_x, k_y) + \tau\left(-\kappa|\tilde{\mathbf{k}}|^2 \hat{u}^n(k_x, k_y) + \hat{g}^n(k_x, k_y)\right)$$

Here, we use the short-hand notation $\hat{g}^n(k_x, k_y) = \hat{g}(k_x, k_y, t_n)$ for the Fourier coefficients of the right-hand side at time t_n. Let's implement this scheme in the following code snippet.

But wait a minute! The ODE (5.96) looks very much like the test equation stiff:ode:eq:exponential we discussed in the previous notebook *Numerical solution of ordinary differential equations: Stiff problems*!! Indeed, if we set $\lambda = -\kappa|\mathbf{k}|^2$, we have exactly the same ODE! What does that mean for our first attempt to solve the heat equation?

To guarantee that the forward Euler method remains stable during the time-stepping, we need to ensure that the time step size τ is chosen such that the stability condition

$$\tau \leqslant 2/\lambda = 2/(\kappa|\tilde{\mathbf{k}}|^2)$$

.

In other words, whenever we double the number of grid points in each direction, we need to reduce the time step size to one fourth to keep the time-stepping stable! This is a very severe restriction, and we need to find a better time-stepping method to solve the heat equation. In the context of discretizing time-dependent PDEs, such conditions where the time step size is (severely) restricted by the space discretization parameter are known as the **CFL condition** (from the names of Courant, Friedrichs, and Lewy).

To avoid such a time step size restriction, we need to find a time-stepping method that is unconditionally stable, i.e. a method that does not depend on the time step size. One such method, namely the **backward** or **implicit** Euler method, we already discussed in the previous notebook *Numerical solution of ordinary differential equations: Stiff problems*. Applying this method to the heat equation, we arrive at the following scheme: Given the solution $\hat{u}^n$ at time $t_n = n\tau$, we compute the right-hand side at $g^{n+1} = g(x, y, t_{n+1})$, and then we compute the Fourier coefficients $\hat{u}^{n+1}$ at time $t_{n+1} = (n+1)\tau$ according to

$$\hat{u}^{n+1}(k_x, k_y) = \frac{\hat{u}^n(k_x, k_y) + \tau\hat{g}^{n+1}(k_x, k_y)}{1 + \tau\kappa|\tilde{\mathbf{k}}|^2} \tag{5.97}$$

Let's try this out! Conceptually, the implementation is very small step from our implementation of the Poisson solver, now we divide by $1 + \tau\kappa|\tilde{\mathbf{k}}|^2$ instead of dividing it by $|\tilde{\mathbf{k}}|^2$, and we need to add a loop over the time steps.

```
%matplotlib widget
import numpy as np
from scipy.fft import fft2, ifft2, fft, fftfreq, fftshift, ifft, ifftshift

import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib.animation as animation

import pandas as pd
```

```
def heat_backward_euler(*, kappa,
                        X, Y, U0,
                        t0, T, Nt,
                        g=None):
    # Extract relevant grid data
    x, y = X[0,:], Y[:,0]
    Nx, Ny = len(x), len(y)
    if Nx < 2 or Ny < 2:
        raise ValueError("Grids must have at least two points in each space direction!
↪")
    # Find space grid sampling size
    dx, dy = x[1] - x[0], y[1] - y[0]

    # Compute wave number grid
    kx = fftfreq(Nx, d=dx/(2*np.pi))
    ky = fftfreq(Ny, d=dy/(2*np.pi))
    KX, KY = np.meshgrid(kx, ky, sparse=True)

    # Compute multiplier for Poisson operator in Fourier space
    K2 = KX**2 + KY**2

    # Time stepping
    t = t0
    dt = (T-t0)/Nt

    # Store FFT of  solution together with t at each time step in a list
    # BAD PRACTICE: We should use a generator instead of a list, see below!
    U_list = []
    # Compute the FFT of the initial data
    U_hat = fft2(U0)

    # For convenience and easier plotting and animation
    U_list.append((ifft2(U_hat).real, t))

    while t < T-dt/2:
        # Compute the right-hand side at t + dt in Fourier space
        if g is not None:
            G_hat = fft2(g(X,Y,t+dt))
        else:
            G_hat = 0

        # Compute solution in Fourier space and transform back to physical space
        U_hat = (U_hat + dt*G_hat)/(1+dt*kappa*K2)

        # Store current solution and time step
        t = t + dt
        U_list.append((ifft2(U_hat).real, t))

    return U_list
```

The following function is useful to manufacture solutions for the heat equation and it works similar to the `manufacture_solution_poisson` function we used in the previous notebook *2D Poisson equation*.

```
def manufacture_solution_heat(u_str, kappa):
    """
    Manufacture a solution for the heat equation.
```

(continues on next page)

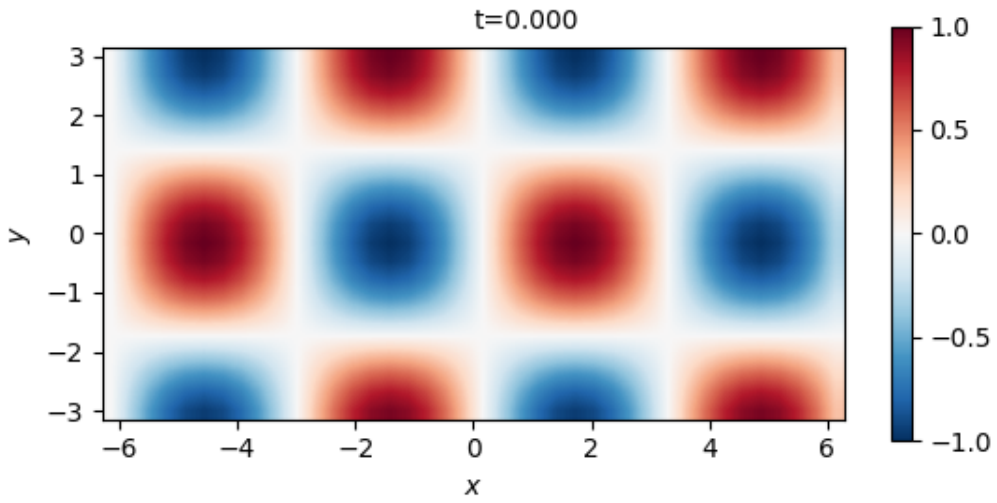
(continued from previous page)

```
    This function takes a symbolic expression for the solution `u` of the heat␣
↪equation
    and computes the corresponding source term `g` such that the heat equation is␣
↪satisfied:

        ∂u/∂t - kappa * Δu = g

    Parameters:
    -----------
    u_str : str
        A string representing the symbolic expression for the solution `u` as a␣
↪function of `x`, `y`, and `t`.
        Example: 'sin(x)*cos(y)*exp(-2*kappa*t)'
    kappa : float
        The thermal diffusivity constant.

    Returns:
    --------
    u : function
        A function of (x, y, t) representing the manufactured solution `u`.
    g : function
        A function of (x, y, t) representing the source term `g`.

    Notes:
    ------
    - The symbolic computation is performed using the `sympy` library.
    - The returned functions `u` and `g` are compatible with NumPy arrays for␣
↪efficient numerical evaluation.

    Example:
    --------
    >>> u_ex, g = manufacture_solution_heat('sin(x)*cos(y)*exp(-2*kappa*t)', kappa=1.
↪0)
    >>> print(u_ex(0, 0, 0))  # Evaluate u at (x, y, t) = (0, 0, 0)
    >>> print(g(0, 0, 0))     # Evaluate g at (x, y, t) = (0, 0, 0)
    """
    import sympy as sy
    from sympy import sin, cos, exp
    x, y, t = sy.symbols('x y t')
    u_sy = eval(u_str)
    laplace = lambda u: sy.diff(u, x, x) + sy.diff(u, y, y)
    g_sy =  sy.diff(u_sy, t) - kappa*sy.simplify(laplace(u_sy))
    u = sy.lambdify((x, y, t), u_sy, modules='numpy')
    g = sy.lambdify((x, y, t), g_sy, modules='numpy')
    print(f'u = {u_sy}')
    print(f'u0 = {u_sy.subs(t, 0)}')
    print(f'g = {g_sy}')
    return u, g
```

```
# Example usage
kappa = 1
u_ex_str = 'sin(x)*cos(y)*exp(-2*kappa*t)'
u_ex, g = manufacture_solution_heat(u_ex_str, kappa)
# Set g to None if you rhs is 0
g = None
```

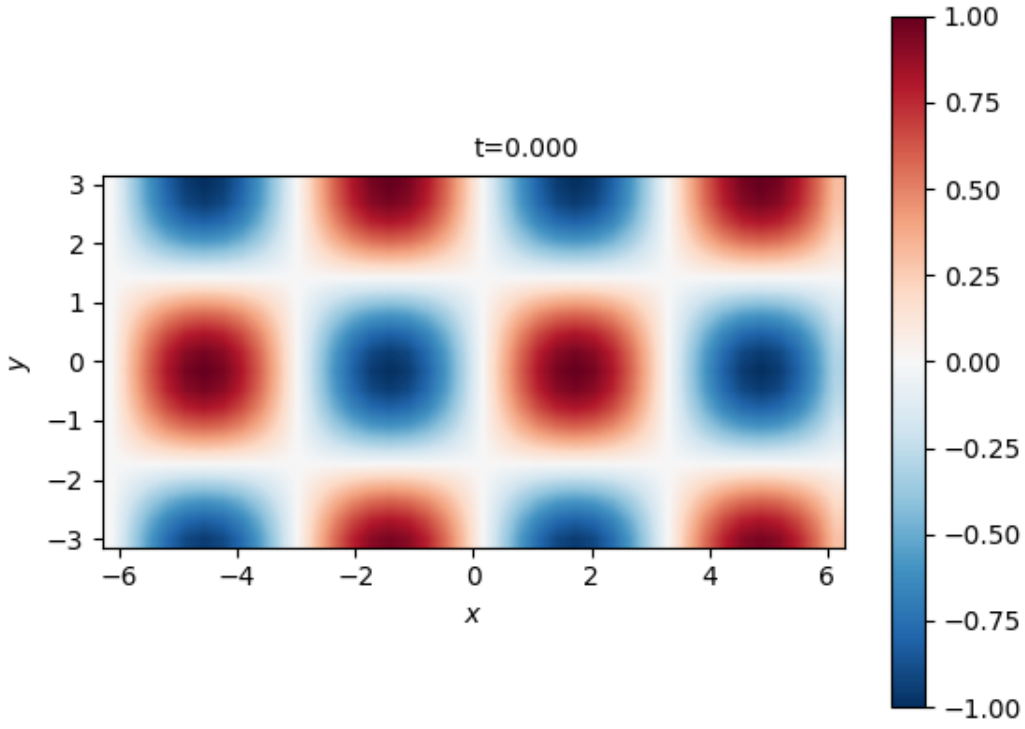
```
u = exp(-2*t)*sin(x)*cos(y)
u0 = sin(x)*cos(y)
g = 0
```

```
# Prepare grid and initial data
Lx, Ly = 4*np.pi, 2*np.pi
Nx, Ny = 40, 20
x = np.linspace(-Lx/2, Lx/2, Nx, endpoint=False)
y = np.linspace(-Ly/2, Ly/2, Ny, endpoint=False)
X, Y = np.meshgrid(x, y, sparse=True)
U0 = u_ex(X,Y,0)

t0, T = 0, 1
Nt = 10

plt.close('all')
fig = plt.figure()
ax = fig.add_subplot(111)
img = ax.imshow(U0, cmap='RdBu_r', interpolation='bilinear', extent=[-Lx/2, Lx/2, -Ly/
 ↪2, Ly/2])
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
tx = ax.text(0,Ly/2*1.1,f"t={0.0:.3f}",
            bbox=dict(boxstyle="round",ec='white',fc='white'))
cbar = plt.colorbar(img, ax=ax, shrink=0.6)  # Adjust the shrink parameter to make␣
 ↪the colorbar smaller
cbar.set_ticks(np.linspace(U0.min(), U0.max(), num=5))  # Set more ticks in the␣
 ↪colorbar
```




```
import os
import os.path
import sys
# Add path to project_tools.py to Python's search path
project_tools_path = os.path.join(os.getcwd(), '../project_3_2025')
if project_tools_path not in sys.path:
    sys.path.append(project_tools_path)
import project_tools as pot
```

```
U_list = heat_backward_euler(kappa=kappa,
                             X=X, Y=Y, U0=U0,
                             t0=t0, T=T, Nt=Nt, g=None)


ani = pot.create_animation(U_list, Lx, Ly)
```




```
ani.to_jshtml()
from IPython.display import HTML
html_animation = ani.to_jshtml()
display(HTML(html_animation))
```

```
<IPython.core.display.HTML object>
```

```
def compute_eoc_transient(*,
                          kappa, u_ex, U0, g,
                          X, Y, t0, T, Nt_list):
    errs_Nt = []
    for Nt in Nt_list:
        U_list = heat_backward_euler(kappa=kappa,
                                     X=X, Y=Y, U0=U0,
                                     t0=t0, T=T, Nt=Nt,
                                     g=g)
        errs_t = []
        for U, t in U_list:
            U_ex = u_ex(X,Y,t)
            U_err = U - U_ex
            # Record maximum error at current time step
            errs_t.append(np.abs(U_err).max())
        # Record maximum error over all time steps
        errs_Nt.append(np.array(np.abs(errs_t).max()))

    Nt_list = np.array(Nt_list)
```

(continues on next page)

(continued from previous page)

```
    errs_Nt = np.array(errs_Nt)
    eocs = np.log(errs_Nt[1:]/errs_Nt[:-1])/np.log(Nt_list[:-1]/Nt_list[1:])
    eocs = np.insert(eocs, 0, np.inf)
    return errs_Nt, eocs
```

```
kappa = 1.0
u_ex_str = 'sin(x)*cos(y)*exp(-2*kappa*t)'
# u_ex_str = '(exp(1+sin(x)*sin(x))+exp(1+cos(y)*cos(y)))*exp(-4*kappa*t)'
u_ex, g = manufacture_solution_heat(u_ex_str, kappa)
# Set g to None if the manufactured solution does not have a source term
g = None


Lx, Ly = 2*np.pi, 2*np.pi
Nx, Ny = 20, 20
x = np.linspace(-Lx/2, Lx/2, Nx, endpoint=False)
y = np.linspace(-Ly/2, Ly/2, Ny, endpoint=False)
X, Y = np.meshgrid(x, y, sparse=True)
t0, T = 0, 1
U0 = u_ex(X,Y,0)

Nt_start = 10
Nt_refs = 6
# Alternative set of  time step numbers where
# Nt = 20000 satisfies the CFL for theta = 0
# Nt_start = 10000
# Nt_refs = 3

Nt_list = [Nt_start*2**i for i in range(Nt_refs+1)]
print(Nt_list)
```

```
u = exp(-2.0*t)*sin(x)*cos(y)
u0 = sin(x)*cos(y)
g = 0
[10, 20, 40, 80, 160, 320, 640]
```

```
errs, eocs = compute_eoc_transient(kappa=kappa,
                                   u_ex=u_ex, U0=U0, g=g,
                                   X=X, Y=Y, t0=t0, T=T, Nt_list=Nt_list)
table = pd.DataFrame({'Nt': Nt_list, 'error': errs, 'EOC': eocs})
display(table)
```

	Nt	error	EOC
0	10	0.033998	inf
1	20	0.017664	0.944656
2	40	0.009010	0.971194
3	80	0.004551	0.985292
4	160	0.002287	0.992567
5	320	0.001147	0.996263
6	640	0.000574	0.998127

5.9.3 Implementation of time-stepping schemes using generator functions in Python

What are generator functions in Python? Any Python function that has the `yield` keyword in its body is a generator function: a function which, when called, returns a generator object. (In other words, a generator function is a generator factory.) A generator function builds a generator object that wraps the body of the function.

Generators are iterators, but you can only iterate over them once. It's because they do not store all the values in memory, they generate the values on the fly. You use them by iterating over them, either with a 'for' loop or explicitly applying the 'next()' function.

For us, the main advantage of using generator functions is that they allow us to write code that is more readable and more efficient as it avoids the need to store all the time steps in memory. In project 3, you will be asked to perform a time-stepping method that requires a large number of time steps (4000) on a large grid (256x256). In this case, storing all the time steps in memory would be very inefficient and will highly likely lead to a memory error.

Why not immediately invert the IFFT at each time step? In the implementation below, we do not invert the IFFT at each time step. This is because the IFFT is despite its $\mathcal{O}(N)\log(N)$ complexity an expensive operation, and we want to avoid it as much as possible. Instead, we store the Fourier coefficients at each time step and invert the IFFT only if relevant, e.g. if we want to plot the solution at certain time steps or compute a quantity of interest.

Let's implement the time-stepping schemes using generator functions in the following code snippet. We will also use the `tqdm` package to display a progress bar during the time-stepping.

```
from tqdm import tqdm

def heat_backward_euler_gen(*, kappa,
                            X, Y, U0,
                            t0, T, Nt,
                            g=None):

    # Extract relevant grid data
    x, y = X[0,:], Y[:,0]
    Nx, Ny = len(x), len(y)
    if Nx < 2 or Ny < 2:
        raise ValueError("Grids must have at least two points in each space direction!
↪")
    # Find space grid sampling size
    dx, dy = x[1] - x[0], y[1] - y[0]

    # Compute wave number grid
    kx = fftfreq(Nx, d=dx/(2*np.pi))
    ky = fftfreq(Ny, d=dy/(2*np.pi))
    KX, KY = np.meshgrid(kx, ky, sparse=True)

    # Compute multiplier for Biharmonic operator in Fourier space
    K2 = KX**2 + KY**2

    # Time stepping
    t = t0
    dt = (T-t0)/Nt

    # Compute the FFT of the initial data
    U_hat = fft2(U0)

    # For convenience and easier plotting and animation,
    # we will yield the initial solution  before starting the time stepping
    yield (U_hat, t)
```

(continues on next page)

(continued from previous page)

```
    with tqdm(total=T) as pbar:
        while t < T-dt/2:
            # Compute the right-hand side at t + theta dt in Fourier space
            if g is not None:
                # Store the old value of G_hat
                G_hat = fft2(g(X,Y,t+dt))
            else:
                G_hat = 0

            # Compute solution in Fourier space and transform back to physical space
            U_hat = (U_hat + dt*G_hat)/(1+dt*kappa*K2)

            t = t + dt
            pbar.update(dt)
            yield U_hat, t
```

```
# Prepare grid and initial data
Lx, Ly = 2*np.pi, 2*np.pi
Nx, Ny = 20, 20
x = np.linspace(-Lx/2, Lx/2, Nx, endpoint=False)
y = np.linspace(-Ly/2, Ly/2, Ny, endpoint=False)
X, Y = np.meshgrid(x, y, sparse=True)
U0 = u_ex(X,Y,0)

t0, T = 0, 1
Nt = 10

# The function call now does not return a list, but a generator!
solver  = heat_backward_euler_gen(kappa=kappa,
                                  X=X, Y=Y, U0=U0,
                                  t0=t0, T=T, Nt=Nt, g=None)
print(f"Solver is of type: {type(solver)}")
```

```
Solver is of type: <class 'generator'>
```

```
# We can now iterate using the generator to get the solution at each time step
for U_hat, t in solver:
    print(f"Time: {t:.3f}")
    U = ifft2(U_hat).real
    print(f"Max error at time step: {np.abs(U - u_ex(X,Y,t)).max()}")
```

```
Time: 0.000
Max error at time step: 3.3306690738754696e-16
```

```
  0%|
↪         | 0/1 [00:00<?, ?it/s]
```

```
100%|
↪9999999999999999/1 [00:00<00:00, 1062.39it/s]
```

```
Time: 0.100
Max error at time step: 0.01460258025535166
Time: 0.200
```

(continues on next page)

(continued from previous page)

```
Max error at time step: 0.024124398408805314
Time: 0.300
Max error at time step: 0.029892067609677553
Time: 0.400
Max error at time step: 0.03292412230253178
Time: 0.500
Max error at time step: 0.03399813084501874
Time: 0.600
Max error at time step: 0.033703764768182154
Time: 0.700
Max error at time step: 0.032484683292047095
Time: 0.800
Max error at time step: 0.03067152136672255
Time: 0.900
Max error at time step: 0.028507811246228476
Time: 1.000
Max error at time step: 0.02617029965323317
```

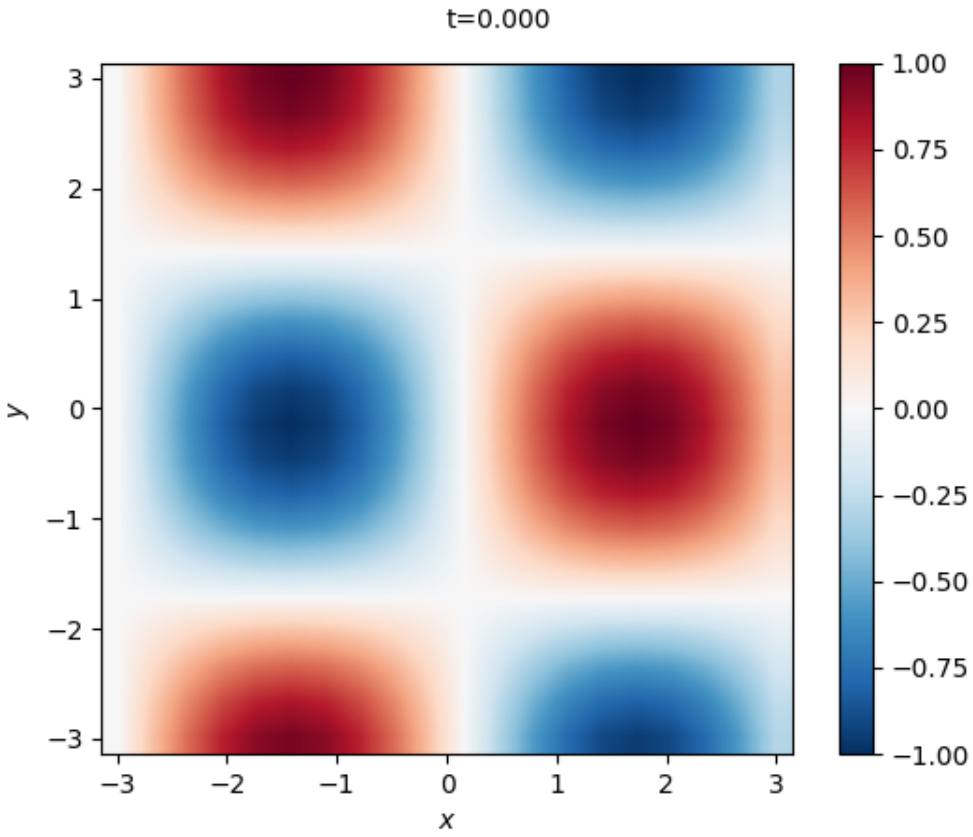
```
Nt = 1000
# Take a snapshot every 10th solution for animation purposes
Nt_snap = Nt/100
# Try also this with snapshots every 1000th solution!
# Nt = 1000000
solver  = heat_backward_euler_gen(kappa=kappa,
                                  X=X, Y=Y, U0=U0,
                                  t0=t0, T=T, Nt=Nt, g=None)

U_snapshots = []
snap_counter = 0
for U_hat, t in solver:
    if snap_counter % Nt_snap == 0:
        U_snapshots.append((ifft2(U_hat).real, t))
    snap_counter += 1
print(f"Number of snapshots: {len(U_snapshots)}")
```

```
  0%|
↪         | 0/1 [00:00<?, ?it/s]
```

```
/Users/andrmas/.local/share/virtualenvs/tma4320_scientific_computation-vlMPykwy/
↪lib/python3.13/site-packages/tqdm/std.py:636: TqdmWarning: clamping frac to␣
↪range [0, 1]
  full_bar = Bar(frac,

100%|██████████████████████████████████████████████████████████████████████| 1.
↪0000000000000007/1 [00:00<00:00, 130.04it/s]
```

```
Number of snapshots: 101
```

```
ani = pot.create_animation(U_snapshots, Lx, Ly)
```




```
ani.to_jshtml()
from IPython.display import HTML
html_animation = ani.to_jshtml()
display(HTML(html_animation))
```

```
<IPython.core.display.HTML object>
```

5.10 Image processing using the Fast Fourier Transform

In this section, we have a glance at how the Fast Fourier Transform (FFT) can be used to process images. The FFT is a powerful tool for analyzing the frequency content of signals, including images. By transforming an image into the frequency domain, we can manipulate its frequency components to achieve various effects, such as filtering, compression, and enhancement.

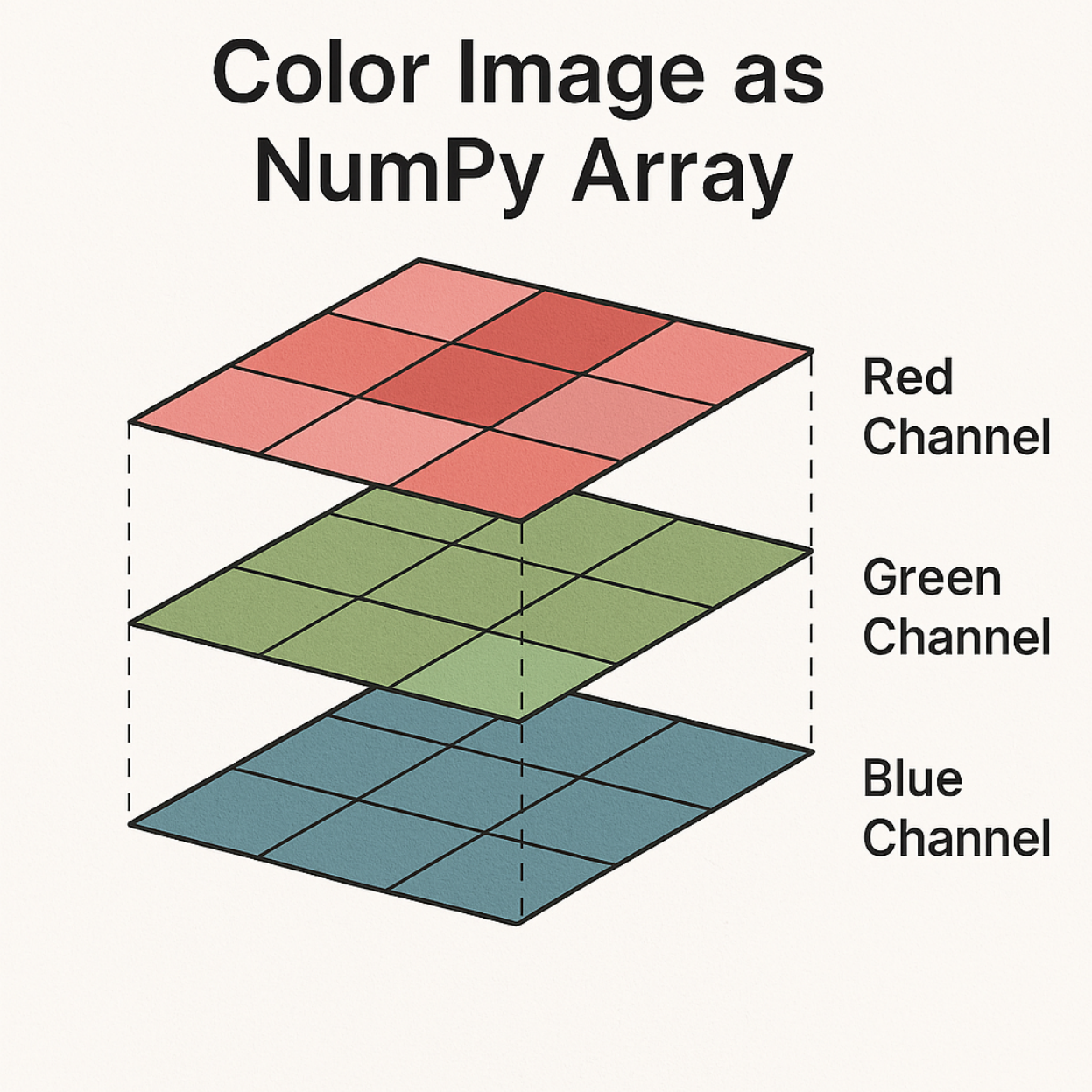
The examples below are taken and adapted from [Brunton and Kutz, 2022], Chapter 2.2. which the authors of the book kindly made available on GitHub, see in particular the examples CH02_SEC06_2_Compress.ipynb and CH02_SEC06_3_Denoise.ipynb.

We can load images stored in jpeg or png format directly from the filesystem using the `imread` function from `matplotlib.image`. This function takes the path to the image file as an argument and returns a NumPy array representing the image.

When a color image is loaded, the resulting array has three dimensions:

- **height**: the number of pixel rows in the image

- **width**: the number of pixel columns in the image
- **color**: channels: the number of color channels (3 for RGB images)



For digital images using the Red-Green-Blue (RGB) color model, the color of a single pixel is represented by a vector like [R, G, B], where R, G, and B are typically integers in the range [0, 255]:

- R = Red intensity
- G = Green intensity
- B = Blue intensity

Each component is typically an integer between 0 and 255 (8-bit unsigned integer), where:

- 0 means no intensity of that color
- 255 means maximum intensity

The final color seen at that pixel is a mix of the red, green, and blue light in the specified intensities.

Examples:

- [255, 0, 0] → Pure Red
- [0, 255, 0] → Pure Green
- [0, 0, 255] → Pure Blue
- [255, 255, 255] → White (full intensity of all colors)
- [0, 0, 0] → Black (no color/light)
- [128, 128, 128] → Gray (equal parts of R, G, B)

For grayscale images, the array has two dimensions (height, width), with each pixel represented by a single intensity value ranging from 0 (black) to 255 (white).

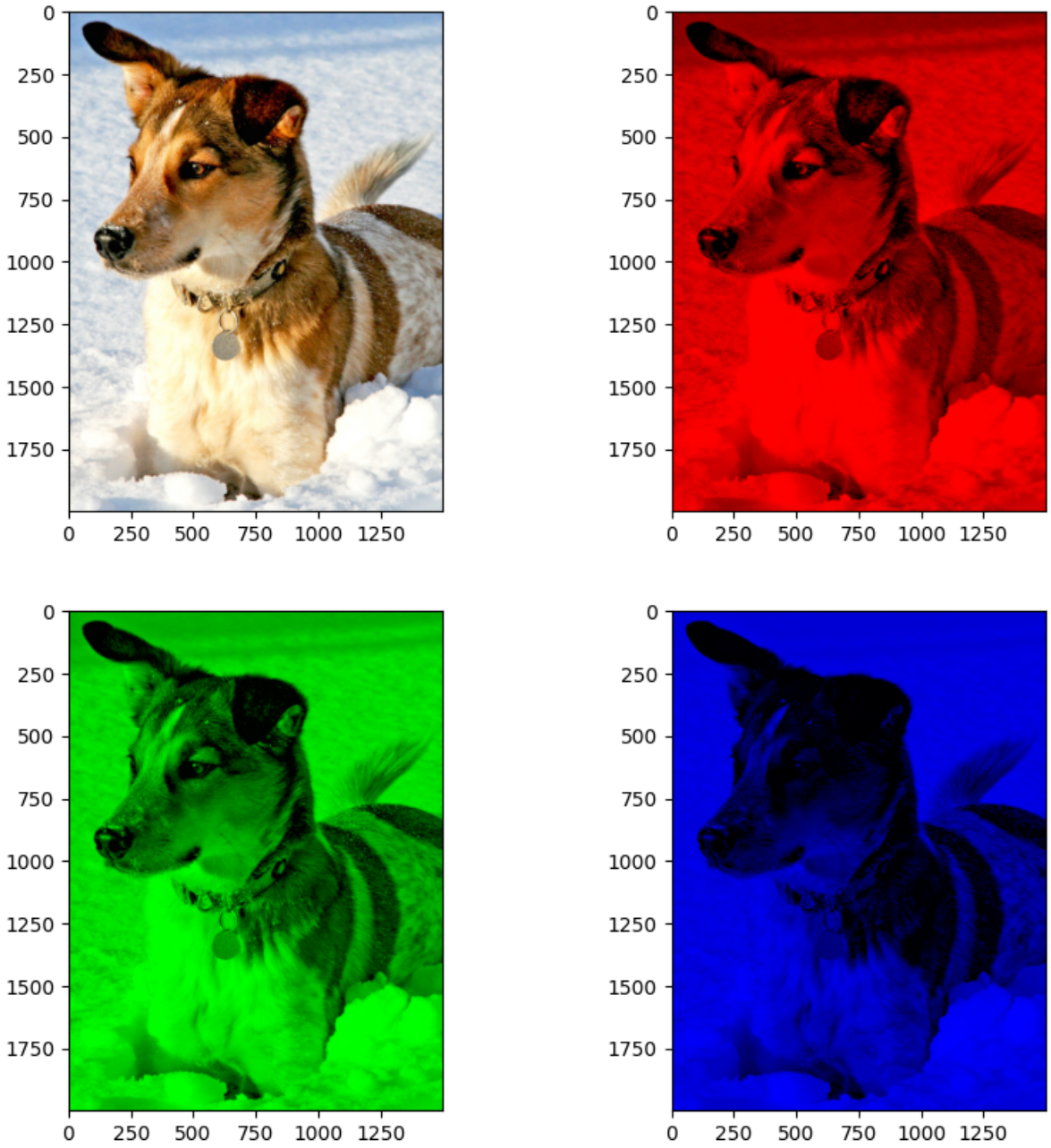
Let's load an image using the `imread` function from `matplotlib.image` submodule and display it using `matplotlib.pyplot.imshow` function, once as original image and then with only one channel activated at a time.

```
dogimage = imread('dog.jpg').copy()
print(dogimage.shape)
print(dogimage.min())
print(dogimage.max())
```

```
(2000, 1500, 3)
0
255
```

```
fig, ax = plt.subplots(2, 2, figsize=(10, 10))
ax[0,0].imshow(dogimage, cmap='gray')

dogimage_red = dogimage.copy()
dogimage_red[:, :, [1,2]] = 0
ax[0,1].imshow(dogimage_red)

dogimage_green = dogimage.copy()
dogimage_green[:, :, [0,2]] = 0
ax[1,0].imshow(dogimage_green)

dogimage_blue = dogimage.copy()
dogimage_blue[:, :, [0,1]] = 0
ax[1,1].imshow(dogimage_blue)
```

```
<matplotlib.image.AxesImage at 0x1289fd090>
```

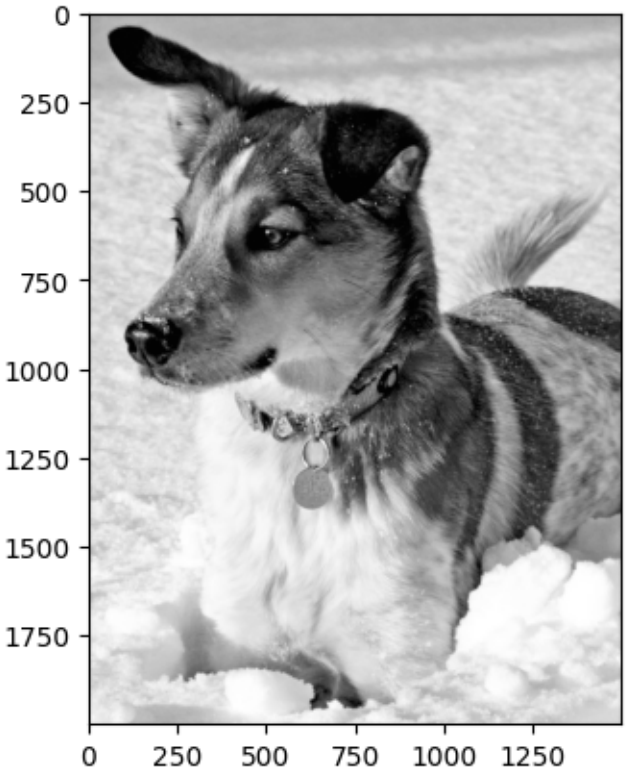


Now we turn the image into a gray scale image by simply taking the average of the three channels. This results in a 2D array of shape (height, width) instead of a 3D array of shape (height, width, 3). The resulting image is a gray scale image where each pixel is represented by a single value between 0 and 255. The value 0 represents black and the value 255 represents white. The values in between represent different shades of gray.

When you plotting the image, you must tell `imshow` that the image is gray scale by passing the `cmap` argument with the value `gray`. The `cmap` argument is short for color map. The default value is `viridis`, which is a color map that is perceptually uniform and works well for most applications. However, when displaying gray scale images, we want to use a different color map.

```
doggray = np.mean(dogimage, axis=2)
plt.imshow(doggray, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x128b5ad50>
```



5.10.1 Compressing grayscale images using the FFT

Let us illustrate how to use the FFT to compress a grayscale image.

```
doggray_hat = fft2(doggray)
# Sort by magnitude to determine the most significant frequencies
doggray_hat_sort = np.sort(np.abs(doggray_hat.flatten()))
```

```
# Zero out all small coefficients and inverse transform
import math
keep_ratios  = (1.0, 0.1, 0.05, 0.01, 0.002)
for i in range(len(keep_ratios)):
    keep = keep_ratios[i]
    thresh = doggray_hat_sort[int(np.floor((1-keep)*len(doggray_hat_sort)))]
    ind = np.abs(doggray_hat)>thresh           # Find small indices
    Atlow = doggray_hat * ind                  # Threshold small indices
    Alow = np.fft.ifft2(Atlow).real  # Compressed image
    plt.figure()
    plt.imshow(Alow,cmap='gray')
    plt.axis('off')
    plt.title('Compressed image: keep = ' + str(keep))
```


Compressed image: keep = 1.0


Compressed image: keep = 0.1

Compressed image: keep = 0.05

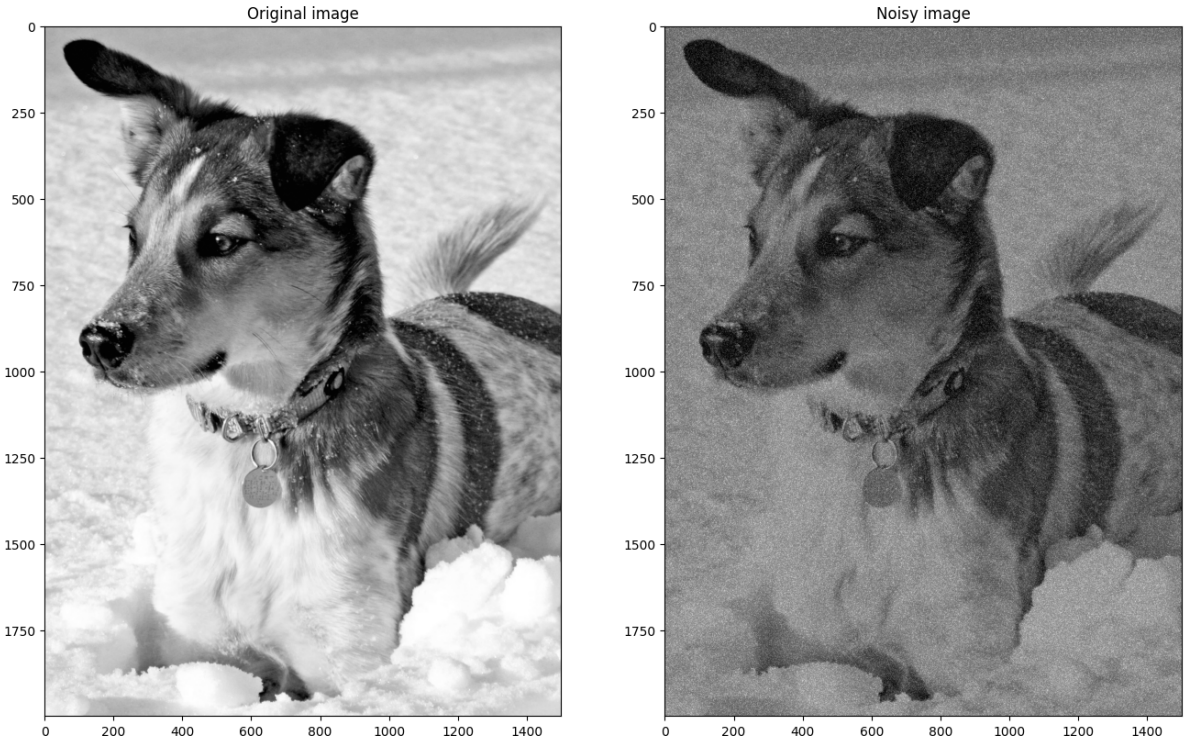
Compressed image: keep = 0.01

5.10.2 A simple denoising example using the FFT

We illustrate how to use the FFT to denoise a grayscale image. First, we add some noise to the image. The noise is generated using a standard normal distribution with a mean of 0 and variance of 1. The noise is then added to the original image, resulting in a noisy image.

```
B = np.mean(dogimage, -1); # Convert RGB to grayscale

## Add some noise
Bnoise = B + 200*np.random.randn(*B.shape).astype('uint8') # Add some noise
fig,axs = plt.subplots(1,2,figsize=(16,16))
axs[0].imshow(B,cmap='gray')
axs[0].set_title('Original image')
axs[1].imshow(Bnoise,cmap='gray')
axs[1].set_title('Noisy image')
```

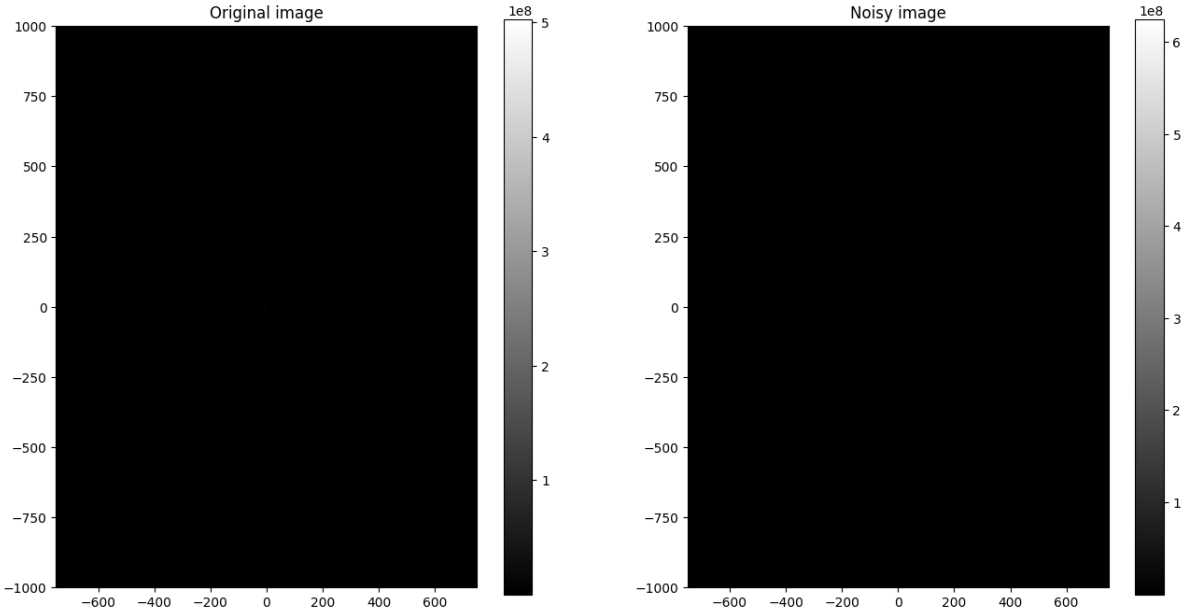
```
Text(0.5, 1.0, 'Noisy image')
```



Let's try to see how their respective Fourier transforms look like.

```
Bhat_shift = fftshift(fft2(B)) # FFT of noisy image
Bnoisehat_shift = fftshift(fft2(Bnoise)) # FFT of noisy image
ny,nx = B.shape

fig,axs = plt.subplots(1,2,figsize=(16,16))
img0 = axs[0].imshow(np.abs(Bhat_shift),cmap='gray',
                     extent=(-nx/2,nx/2,-ny/2,ny/2))
axs[0].set_title('Original image')
fig.colorbar(img0, orientation='vertical', shrink=0.5)

img1 = axs[1].imshow(np.abs(Bnoisehat_shift),cmap='gray',
                         extent=(-nx/2,nx/2,-ny/2,ny/2))
axs[1].set_title('Noisy image')
fig.colorbar(img1, orientation='vertical', shrink=0.5)
```

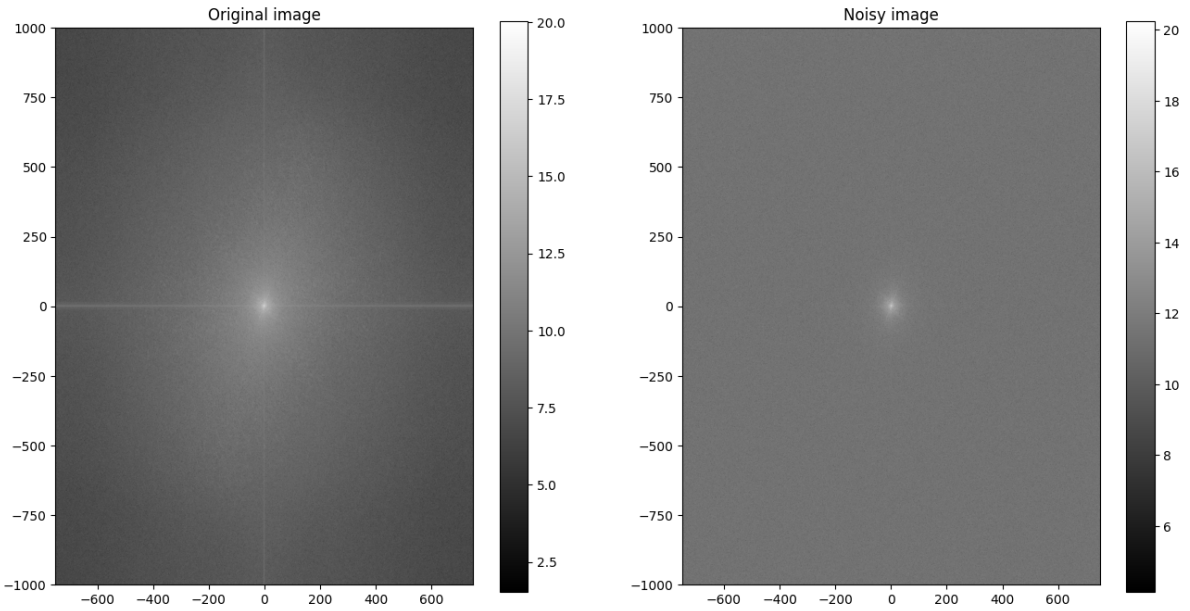
```
<matplotlib.colorbar.Colorbar at 0x128c82710>
```



This was not very helpful! Indeed, large parts of the image's information are concentrated in low frequencies, which are hard to see on a linear scale. Using a logarithmic scale makes low frequencies more visible.

```
log_Bhat_shift = np.log(1 + np.abs(Bhat_shift))
log_Bnoisehat_shift = np.log(1 + np.abs(Bnoisehat_shift))
ny,nx = B.shape

fig,axs = plt.subplots(1,2,figsize=(16,16))
img0 = axs[0].imshow(log_Bhat_shift,cmap='gray',
                     extent=(-nx/2,nx/2,-ny/2,ny/2))
axs[0].set_title('Original image')
# axs[0].axis('off')
fig.colorbar(img0, orientation='vertical', shrink=0.5)

img1 = axs[1].imshow(log_Bnoisehat_shift,cmap='gray',
                     extent=(-nx/2,nx/2,-ny/2,ny/2))
axs[1].set_title('Noisy image')
# axs[1].axis('off')
fig.colorbar(img1, orientation='vertical', shrink=0.5)
```

```
<matplotlib.colorbar.Colorbar at 0x12c6811d0>
```



We see that most of the information is concentrated in the low frequencies, as their amplitudes are significantly larger than those of the high frequencies. We also see that adding noise to the image has added high frequency components to the Fourier transform. A simple filtering technique is to remove the high frequency components from the Fourier transform of the noisy image by using a threshold frequency, thus zeroing out the high frequencies.

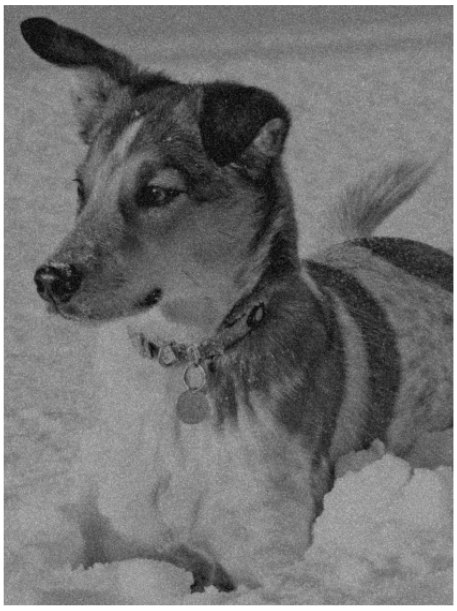
In the code below, we simply neglect all gradients above a certain threshold frequency $f_{\text{tres}} = 150$.

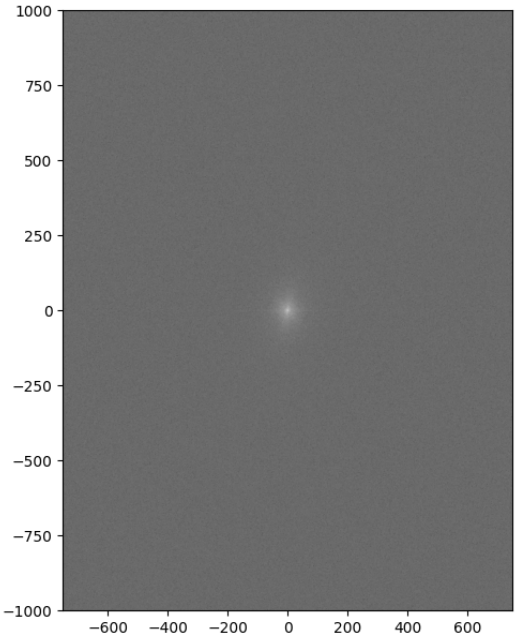
```
B = np.mean(dogimage, -1); # Convert RGB to grayscale

## Denoise
Bnoise = B + 200*np.random.randn(*B.shape).astype('uint8') # Add some noise
Bt = np.fft.fft2(Bnoise)
Btshift = np.fft.fftshift(Bt)

F = np.log(np.abs(Btshift)+1) # Put FFT on log scale

fig,axs = plt.subplots(2,2,figsize=(16,16))

axs[0,0].imshow(Bnoise,cmap='gray')
axs[0,0].axis('off')

axs[0,1].imshow(F,cmap='gray',
                      extent=(-nx/2,nx/2,-ny/2,ny/2))

ny,nx = B.shape
X,Y = np.meshgrid(np.arange(-nx/2+1,nx/2+1),np.arange(-ny/2+1,ny/2+1))
# xgrid = np.fft.ifftshift(np.arange(-nx/2+1,nx/2+1))
# ygrid = np.fft.ifftshift(np.arange(-ny/2+1,ny/2+1))
# X,Y = np.meshgrid(ygrid,xgrid)
R2 = np.power(X,2) + np.power(Y,2)
ind = R2 < 150**2
Btshiftfilt = Btshift * ind
Ffilt = np.log(np.abs(Btshiftfilt)+1) # Put FFT on log scale

axs[1,1].imshow(Ffilt,cmap='gray',
                      extent=(-nx/2,nx/2,-ny/2,ny/2))
```

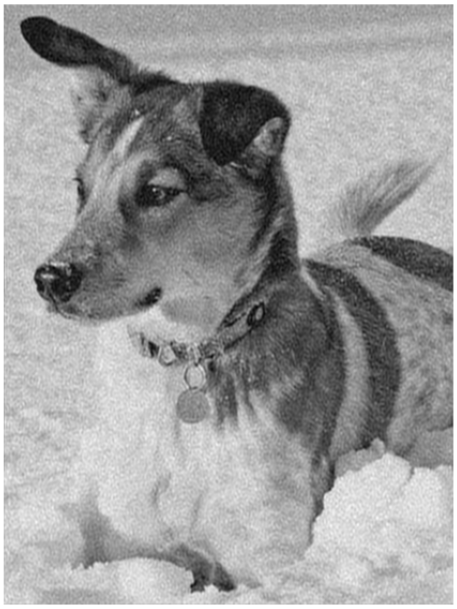
(continues on next page)

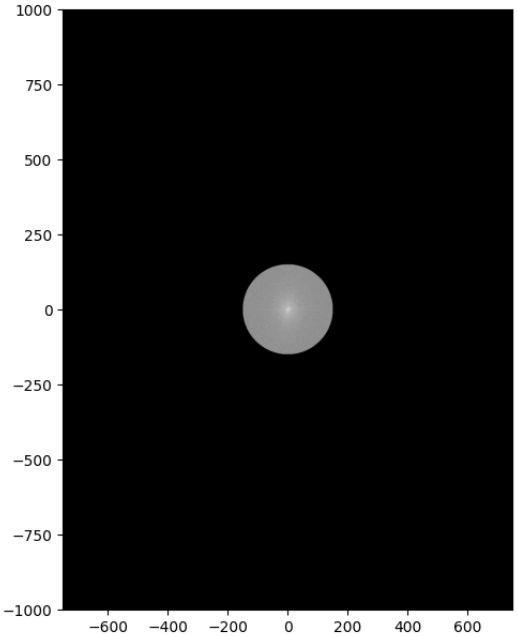
(continued from previous page)

```
Btfilt = np.fft.ifftshift(Btshiftfilt)
Bfilt = np.fft.ifft2(Btfilt).real
axs[1,0].imshow(Bfilt,cmap='gray')
axs[1,0].axis('off')

plt.show()
```









Remark 5

We want to stress that the shown examples/techniques are quite simplistic and mainly presented for educational purposes, that is, to give you an idea where/how the FFT can be used in image processing. Of course, there are many, many more sophisticated techniques for image compression and denoising, such as windowed Fourier transform, wavelet transforms, principal component analysis (PCA), and deep learning-based methods which we cannot cover here, see ee.g. [Brunton and Kutz, 2022], and references therein.

5.11 Exercises on the discrete Fourier transform

Exercise 28

Consider two periodic functions $f(x)$ and $g(x)$ with period $L = 2$ and their truncated Fourier series $f_N(x)$ and $g_N(x)$. The errors E_N between each function and its trigonometric approximation can be computed using Parseval's identity,

$$E_N = \int_{-1}^{1} f^2(x)\,\mathrm{d}x - 2\sum_{k=-N}^{N} |c_k|^2\,.$$

a) Let $f(x) = \mathrm{e}^{-x}$ for $x \in [-1, 1)$ and consider its periodic extension with period $L = 2$. Find on the Fourier coefficients of $f(x)$, calculate the error E_N for $N = 2$, 4 and 8.

b) Now, do the same for $g(x) = \mathrm{e}^{-|x|}$.

c) Why is the error so much larger when approximating $f(x)$ than in the case of $g(x)$?

Solution to Exercise 28

a) We first find the Fourier coefficients,

$$c_k = \frac{1}{2}\int_{-1}^{1} e^{-(1+ik\pi)x}\,dx = \frac{(-1)^k \sinh(1)}{1+ik\pi}, \tag{5.98}$$

and then integrate

$$\int_{-1}^{1} f^2(x)\mathrm{d}x = \int_{-1}^{1} e^{-2x}\mathrm{d}x = \frac{e^2 - e^{-2}}{2} = \sinh(2),$$

Then we get an expression for the error,

$$E_N = \sinh(2) - 2\sum_{k=-N}^{N} \frac{\sinh^2(1)}{(1+ik\pi)^2} = \sinh(2) - \sum_{k=-N}^{N} \frac{2\sinh^2(1)}{1+k^2\pi^2}.$$

Inserting values for N: $E_2 = 0.22$, $E_4 = 0.123$, $E_8 = 0.066$

b) Now, we find the Fourier coefficients for $g(x)$

$$c_k = \frac{1}{2}\int_{-1}^{1} \mathrm{e}^{-|x|}\mathrm{e}^{-ik\pi x}\,\mathrm{d}x = \frac{1}{1+k^2\pi^2}\left(1 - \mathrm{e}^{-1}(-1)^k\right).$$

As in a), we need to compute the integral of g(x) squared

$$\int_{-1}^{1} g^2(x)\mathrm{d}x = \int_{-1}^{1} e^{-2|x|}\mathrm{d}x = 1 - e^{-2},$$

$$E_N = 1 - e^{-2} - 2\sum_{k=-N}^{N}\left(\frac{1}{1+k^2\pi^2}\left(1-\mathrm{e}^{-1}(-1)^k\right)\right)^2.$$

Inserting values for N gives: $E_2 = 1.19e-3, E_4 = 2e-4, E_8 = 2.8e-5$

c) $g(x)$ is smoother than $f(x)$ and the Fourier series for $g(x)$ are therefor a better approximation.

Exercise 29

a) Use DFT (by hand) to find the coefficients c_k for the trigonometric polynomial which interpolates the following datapoints

t	0	0.25	0.5	0.75
Datapoint	1+i	i	-1-i	-i

b) Plot (using Python) the real and imaginary parts of corresponding interpolation polynomial $Q_k(t) = \frac{1}{n}\sum_{k=0}^{n-1} c_k e^{2\pi i k t}$, together with the datapoints.

Solution to Exercise 29

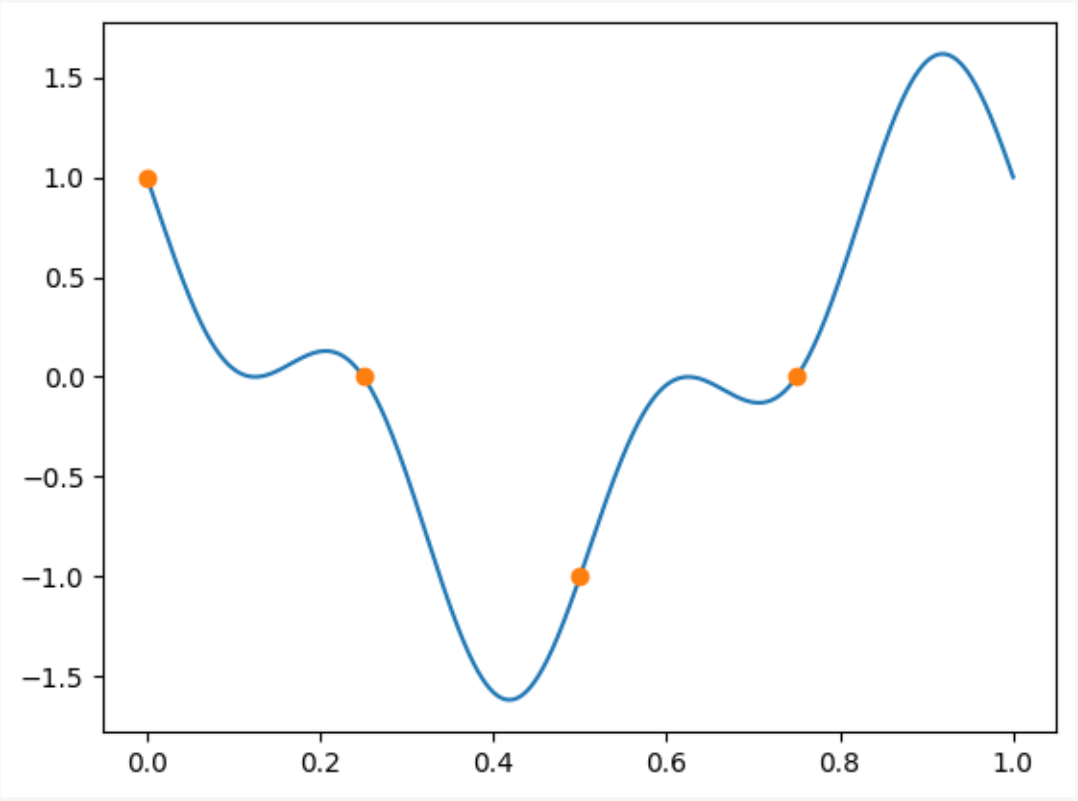
We have four datapoints, and recall the definition of $\omega_N^{-1} = e^{-2\pi i/n} = e^{-i\pi/2} = -i$, use this in the Fourier matrix,

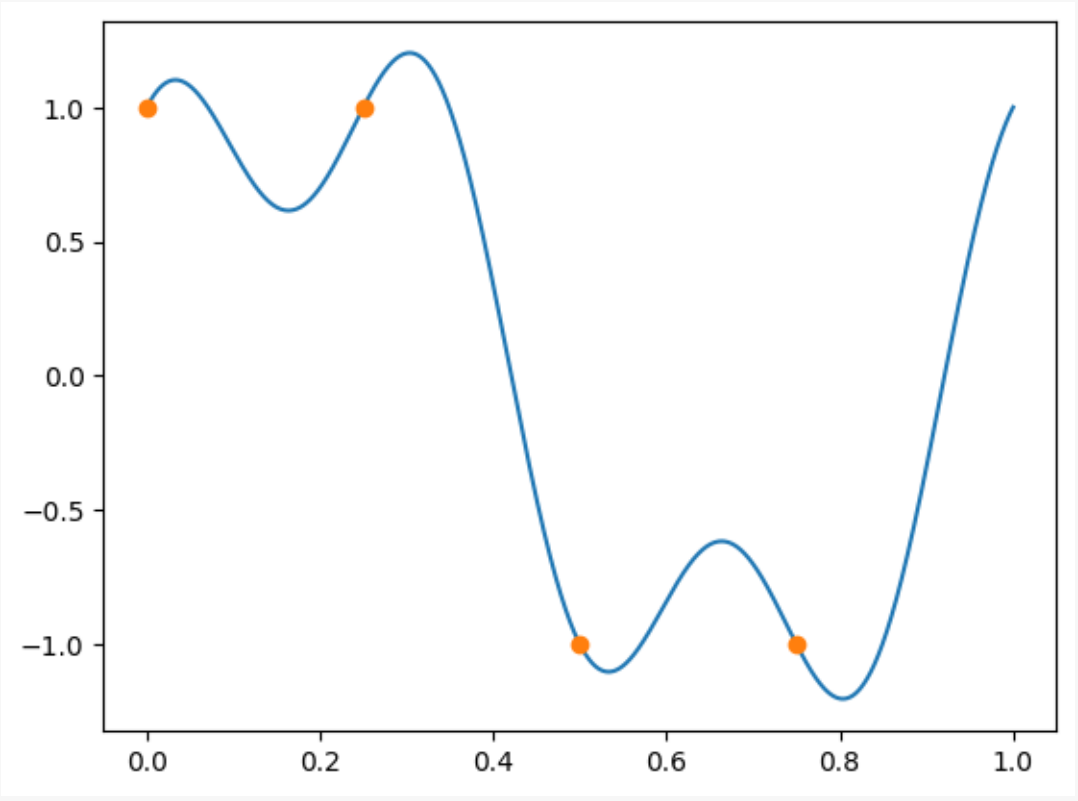
$$F_4 = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_N^{-1} & \omega_N^{-2} & \omega_N^{-3} \\ 1 & \omega_N^{-2} & \omega_N^{-4} & \omega_N^{-6} \\ 1 & \omega_N^{-3} & \omega_N^{-6} & \omega_N^{-9} \end{pmatrix} = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix} \tag{5.99}$$

We can then multiply the matrix with the vector of datapoints and get the coefficients,

$$c_k = \frac{1}{4}F_4 D_4 = \frac{1}{4}[0, 4+2i, 0, 2i]. \tag{5.100}$$

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

N = 4
datapoints = np.array([1+1j,1j,-1-1j,-1j])
t = np.array([0,0.25,0.5,0.75])
c_k = 1/N*np.array([0,4 + 2j,0,2j]) #found by matrix multiplication as above

x = np.linspace(0,1,10000)

#build the interpolation polynomial
S_N = c_k[0]
for i in range(1,N):
    S_N += c_k[i]*np.exp(2j*np.pi*x*i)


plt.plot(x, S_N.real)
plt.plot(t, datapoints.real,marker='o', linestyle='')
plt.show()

plt.plot(x, S_N.imag)
plt.plot(t, datapoints.imag,marker='o', linestyle='')
```


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

Exercise 30

In this task we use the datapoints given in the codeblock below

a) Interpolate using DFT, this time using numpy/scipy to find the coefficients. Plot the corresponding trigonometric interpolation polynomial Q_{10}, and check that it interpolates the datapoints.

b) Using the Euler formula, and fact that all datapoints are real-valued, we have the following formula to rewrite Q_{10},

$$P_n(t) = \frac{1}{n}\sum_{k=0}^{n-1} a_k \cos(2\pi ikt) - b_k \sin(2\pi ikt),$$

where $c_k = a_k + ib_k$. Plot P_{10} and check that it is the same as Q_{10}.

c) Try plotting

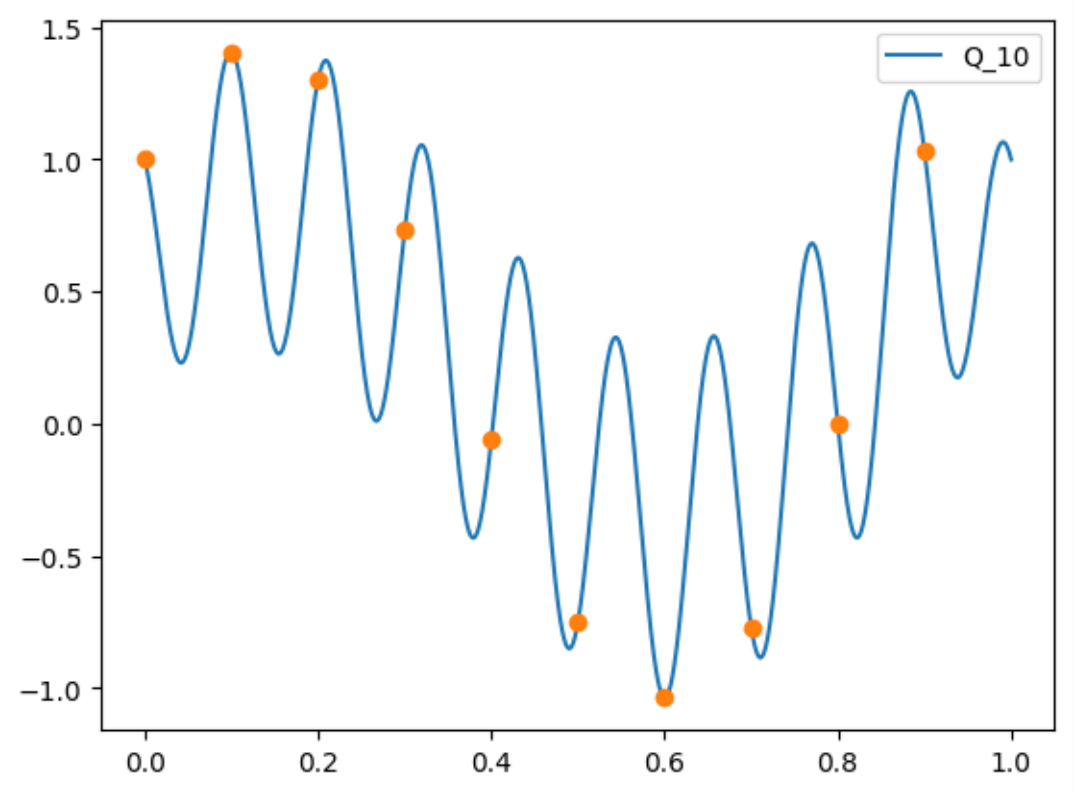
$$\tilde{P}_{10} = \frac{a_0}{n} + \frac{2}{n}\sum_{k=0}^{n/2-1} (a_k \cos(2\pi ikt) - b_k \sin(2\pi ikt)) + \frac{a_{n/2}}{n}\cos(\pi it)$$

Explain why $\tilde{P}_{10}$ also interpolates the same datapoints, with only half of the Fourier terms

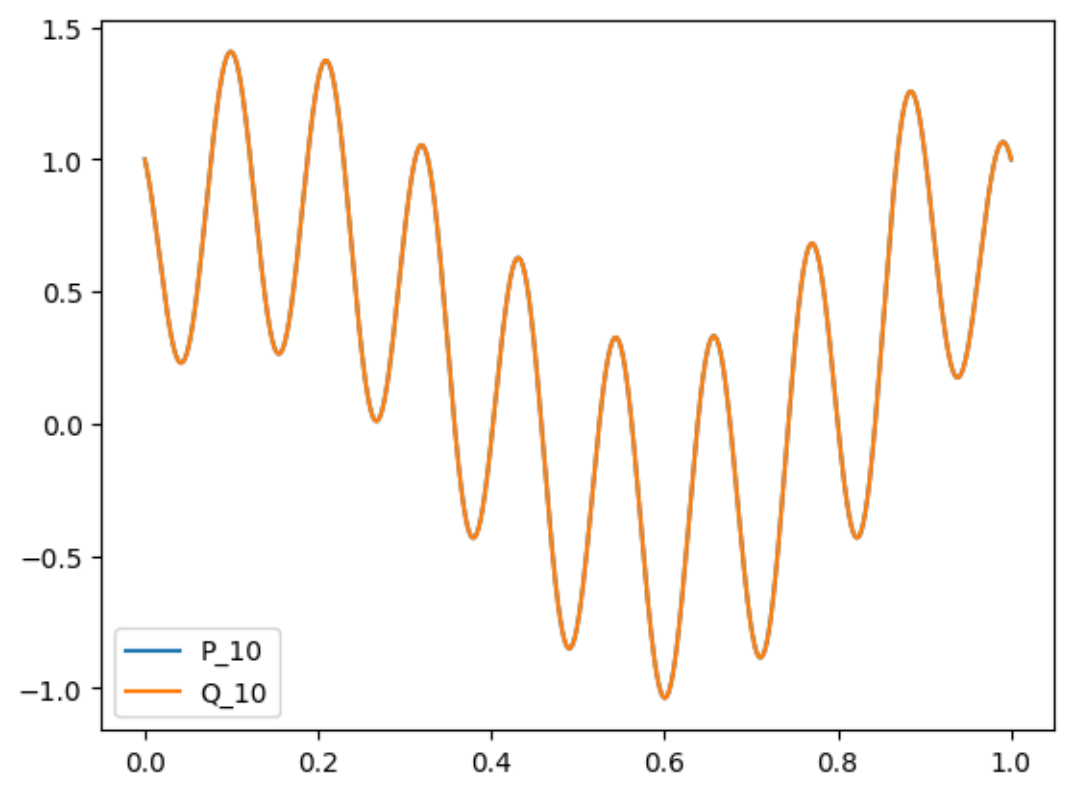
Solution to Exercise 30

```
datapoints=np.array([ 1.0, 1.40680225, 1.30007351, 0.73203952, -0.06123174, -0.75,␣
↪-1.03680225, -0.77007351, -0.00203952, 1.03123174])
t = np.array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

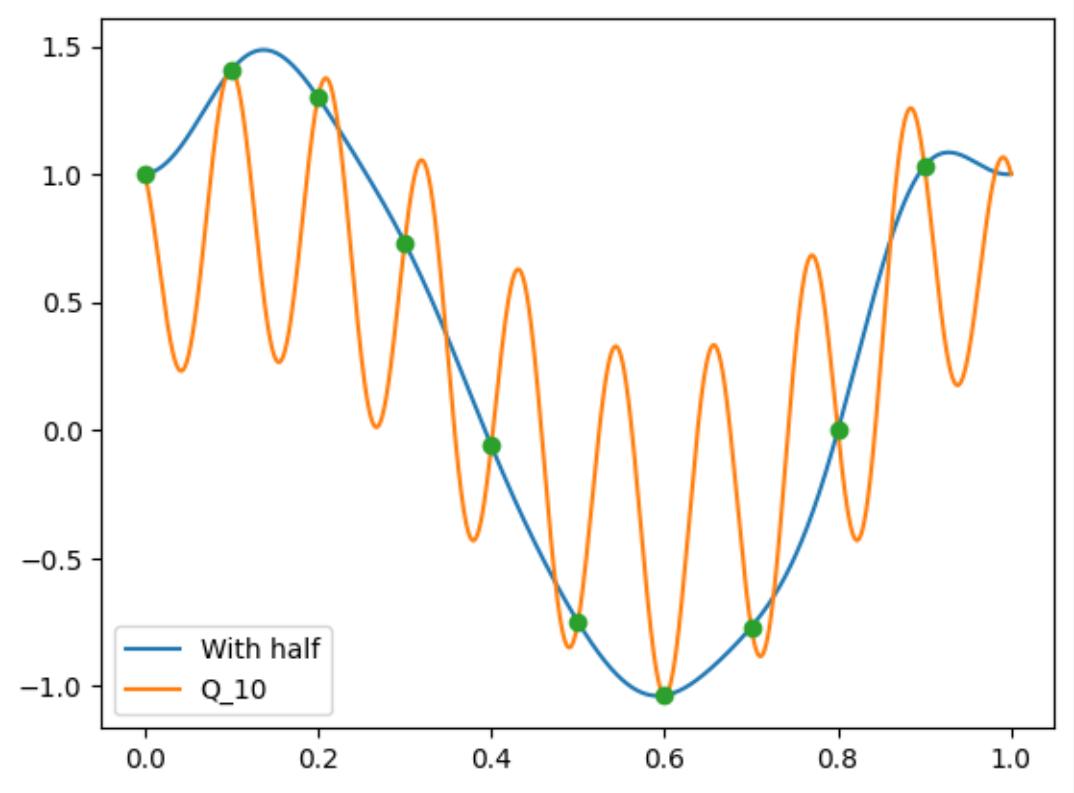
```
N = 10
x = np.linspace(0,1,10000)
dft = np.fft.fft(datapoints)

Q_N = dft[0]*1/N
for i in range(1,N):
    Q_N += 1/(N)*dft[i]*np.exp(2j*np.pi*x*i)


plt.plot(x, Q_N.real,label="Q_10")
plt.plot(t, datapoints,marker='o', linestyle='')
plt.legend()
plt.show()
```




```
P_N = dft[0].real*1/N#*np.exp(x*1j*0)
for i in range(1,N):
    P_N += 1/(N)*(dft[i].real*np.cos(i*2*np.pi*x) - dft[i].imag*np.sin(i*2*np.
 ↪pi*x))


plt.plot(x, P_N.real, label='P_10')
plt.plot(x, Q_N.real,label="Q_10")
plt.legend()
plt.show()
```




```
P_N2 = dft[0].real*1/N#*np.exp(x*1j*0)
for i in range(1,int(N/2)):
    P_N2+= 2/(N)*(dft[i].real*np.cos(i*2*np.pi*x) - dft[i].imag*np.sin(i*2*np.
 ↪pi*x))
P_N2 += 1/(N)*dft[int(N/2)].real*np.cos(N*np.pi*x)

plt.plot(x, P_N2.real, label='With half')
plt.plot(x, Q_N.real,label="Q_10")
plt.plot(t, datapoints,marker='o', linestyle='')
#plt.plot(x, func(x).real)
plt.legend()
plt.show()
```



If we have real-valued datapoints x_k, and do a DFT to get c_k, we have that c_0 is real valued, and $c_{n-k} = \bar{c}_k$ (you can print out the dft above and check). Using also the trigonometric identities $\cos(2(n-k)\pi t) = \cos(2k\pi t)$, $\sin(2(n-k)\pi t) = -\sin(2k\pi t)$, we will get the formula above (for an even number datapoints, a similar exists of odd number of datapoints).

5.12 Summary for Chapter 5

This chapter introduces the Discrete Fourier Transform (DFT) and its efficient implementation via the Fast Fourier Transform (FFT), one of the most important algorithms in computational mathematics. It presents both the theoretical foundations and diverse applications, ranging from signal and image processing to numerical differentiation and spectral methods for partial differential equations (PDEs).

 Section 5.1 – Motivation

- Importance of DFT/FFT in applied mathematics, data analysis, and PDEs.
- Example applications: ECG signal analysis, image denoising, and pattern formation in materials (e.g. phase separation).

 Section 5.2 – Preliminaries

- Review of complex numbers, Euler's formula, roots of unity, and inner products in complex vector spaces.
- Definitions and properties of orthogonal and orthonormal systems in L^2 spaces.

 Section 5.3 – Brief review of Fourier Series

- Representation of periodic functions using complex exponentials.
- Derivation and interpretation of Fourier coefficients.

- Relationship between trigonometric and exponential form of Fourier series.

 Section 5.4 – The Discrete Fourier Transform (DFT)

- Derivation of DFT from Fourier series and numerical quadrature.
- Matrix formulation of the DFT and its interpretation as a change of basis in $\mathbb{C}^N$.
- Discrete orthogonality and inner product spaces.
- Inverse DFT

 Section 5.5 – Trigonometric Interpolation

- Interpolation of periodic data using trigonometric polynomials.
- Connections between DFT and trigonometric interpolation
- Best approximation properties of truncated trigonometric polynomials.

 Section 5.6 – Using DFT

- Efficient computation of DFT using the Fast Fourier Transform (FFT).
- Use of FFT in Python using NumPy and SciPy libraries.
- Aliasing and Nyquist frequencsy

 Section 5.7 – Numerical Differentiation and Spectral Derivatives

- Use of DFT for high-accuracy differentiation and comparison with finite difference methods.

 Section 5.8 – Solving PDEs with Fourier Spectral Methods

- 2D Fourier series representation of functions on a rectangular domain.
- Fourier-space representation of differential operators.
- Using FFTs to solve elliptic PDEs e.g. Poisson with periodic boundary conditions.

 Section 5.9 – A Fourier Solver for the Heat Equation

- Combining FFTs with time-stepping methods for ODEs to solve parabolic PDEs (e.g. heat equations) with periodic boundary conditions.
- Full implementation and simulation of the 2D heat equation using FFT.
- Visual analysis of solution dynamics over time.

 Section 5.10 – Image Processing with FFT

- Application of 2D FFTs to filter noise in grayscale images.
- Frequency domain thresholding and reconstruction.

 Learning Outcomes for Chapter 5

By the end of this chapter, students will be able to:

 Mathematical Foundations

- Recall and apply key properties of complex numbers, Euler's identity, and roots of unity.
- Define and work with orthogonal systems in L^2 and understand their role in Fourier analysis.
- Derive and interpret the Fourier series of periodic functions.

 Discrete Fourier Transform Theory

- Derive the Discrete Fourier Transform (DFT) and explain its connection to Fourier series and quadrature.

- Fast Fourier Transforms (FFT) efficiently using Python.
- Interpret the DFT as a projection onto a basis of discrete complex exponentials.

 Signal and Image Analysis

- Analyze discrete data (e.g. signals) in the frequency domain using the DFT.
- Visualize and interpret both time-domain and frequency-domain representations of functions and signals.
- Interpret magnitude and phase of Fourier coefficients in practical contexts (e.g. signal strength, periodicity).
- Apply Fourier filtering to smooth or denoise data and images.
- Understand how Fourier methods can be used for signal and image compression.

⚙ Numerical Methods and PDEs

- Use DFT to perform spectral differentiation
- Implement Fourier spectral methods for stationary PDEs (e.g. Poisson equation) with periodic boundary conditions.
- Combine Fourier spectral methods and one-step time-stepping methods to numerically solve time-dependent PDEs (e.g. heat equation, Poisson equation) with periodic boundary conditions.
- Understand how stiff ODE system arises from Fourier spectral methods and leads to time-step restrictions (CFL conditions) and how to address them using implicit methods.
- Assess the accuracy and efficiency of Fourier spectral methods using manufactured solutions including EOC studies for time-dependent PDEs.

CHAPTER

SIX

PROJECT 3: SIMULATION OF PHASE SEPARATION IN A BINARY MIXTURE

After the introductory presentation of the background for this project, we will now dive into the mathematical and computational modeling of phase separation phenomena.

As a particular model for the phase separation of a binary mixture, we consider here the so-called Cahn-Hilliard equation. The Cahn-Hilliard equation models the evolution of a mass-conservative, two-component system described by a (rescaled) concentrations $u(x,t) \in [-1,1]$ of component 1 and $-u(x,t)$ of component 2. So the entire system is rescaled so that the total concentration is zero. In the literature, you will also often find a different rescaling, where the total concentration is one and two concentration of the two components are described by u and $1-u$. But for symmetry reasons, we prefer the above rescaling.

Due to mass conservation, it is sufficient to consider the evolution of the concentration of first component u. The resulting Cahn-Hilliard equation then reads

$$\partial_t u = \nabla \cdot (M \nabla \mu), \quad \mu = -\kappa \Delta u + f(u) \quad \text{on } \Omega \times (0,T) \tag{6.1}$$

where M is the **mobility**, κ is the **gradient energy coefficient**, and $f(u) = F'(u) = \frac{\mathrm{d}}{\mathrm{d}u} F(u)$ is the derivative of the **Helmholtz free energy density** $F(u)$. Here, μ is the so-called **chemical potential**, and it can be shown that it is the variational derivative

$$\mu = \frac{\delta \mathcal{E}}{\delta u}$$

of the so-called **Ginzburg-Landau** free energy function

$$\mathcal{E}(u) = \int_\Omega \Big(\frac{\kappa}{2} |\nabla u|^2 + F(u) \Big) \, \mathrm{d}x.$$

with respect to u. For later purposes, we refer to

$$\mathcal{E}_{\text{int}}(u) = \int_\Omega \frac{\kappa}{2} |\nabla u|^2 \, \mathrm{d}x \tag{6.2}$$

$$\mathcal{E}_{\text{mix}}(u) = \int_\Omega F(u) \, \mathrm{d}x. \tag{6.3}$$

as the **interface energy** and the **mixing energy**, respectively.

We want to point out the formal similarities with heat equation

$$\partial_t T = -k \nabla \mathbf{q}$$

discussed in the lecture, where the temperature evolution T is driven by gradient of the heat flux $\nabla \cdot \mathbf{q}$, with the heat flux given by $\mathbf{q} = -k \nabla T$ and k being the heat conductivity. From that perspective the Cahn-Hilliard equation can be considered as a nonlinear generalization of the heat equation, where a concentration flux $\mathbf{j} = -M \nabla \mu$ is driven by

the gradient of the chemical potential μ. However, the complicated relation between the chemical potential μ and the concentration u makes the Cahn-Hilliard equation a much more challenging problem and results in a vastly different behavior, generating solutions with complex patterns and structures.

Combining the two equations in eq{eq:cahn-hilliard-system} into one, we obtain the Cahn-Hilliard equation

$$\partial_t u - \nabla \cdot (M \nabla (f(u) - \kappa \Delta u)) = 0 \quad \text{on } \Omega \times (0, T) \tag{6.4}$$
$$\partial_n \mu = 0 \quad \text{on } \Gamma \times (0, T) \tag{6.5}$$
$$\partial_n u = 0 \quad \text{on } \Gamma \times (0, T) \tag{6.6}$$

To complete the model, we need to specify the energy density $F(u)$. Typically, F is described by a double-well potential, and from thermodynamical considerations, it is often chosen to be a logarithmic function of the form

$$F(u) = \frac{\theta_c}{2}(1 - u^2) + \frac{\theta}{2}\left[(1 - u)\ln\Big(\frac{1 - u}{2}\Big) + (1 + u)\ln\Big(\frac{1 + u}{2}\Big)\right] \tag{6.7}$$

for $u \in (-1, 1)$. The logarithmic terms correspond to the mixing entropy of the two components. The critical temperature θ_c is the temperature above which the homogeneous mixture is stable, and below which phase separation can occur. At temperatures below θ_c, the system tends to separate into two distinct phases to minimize its free energy.

Deriving (6.7) yields

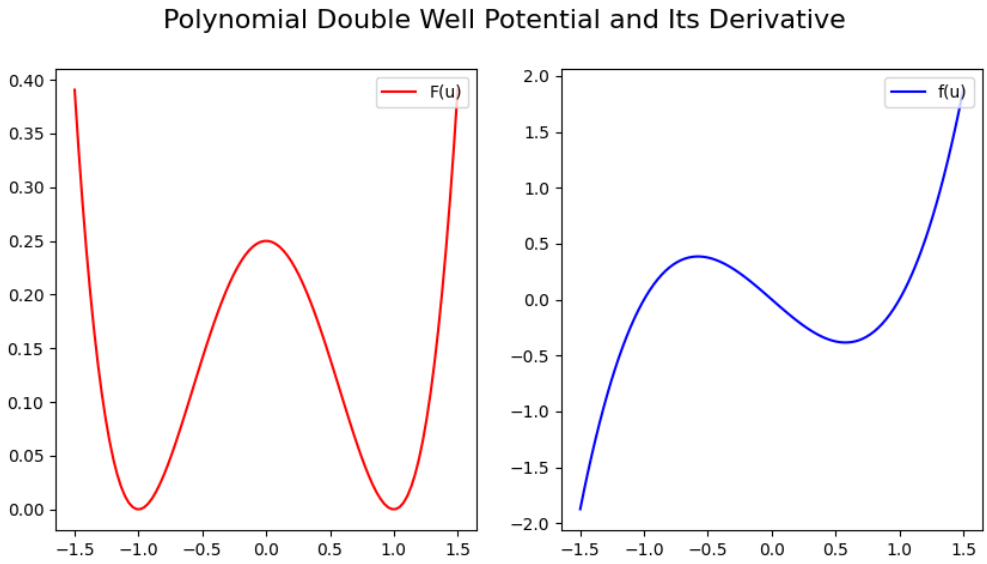
$$f(u) = F'(u) = -\theta_c u + \frac{\theta}{2}\ln\Big(\frac{1 + u}{1 - u}\Big).$$

The logarithmic terms appearing in $f(u)$ can be challenging for both the theoretical analysis and the numerical solution of the Cahn-Hilliard equations. Consequently, F is often regularized and approximated (with proper rescaling) by a polynomial double well potential of the form

$$F(u) = \frac{1}{4}(u^2 - 1)^2, \quad \text{leading to} \quad f(u) = u^3 - u.$$

(proj:eq:double-well-potential-polynomial) which we will use throughout this project and is plotted below.

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

def F(u):
    return 0.25*(u**2-1)**2
def f(u):
    return u**3-u

x = np.linspace(-1.5, 1.5, 100)
plt.close('all')
fig = plt.figure(figsize=(10, 5))
fig.suptitle("Polynomial Double Well Potential and Its Derivative", fontsize=16)
ax1 = fig.add_subplot(121)
ax1.plot(x, F(x), "r", label='F(u)')
ax1.legend(loc='upper right')

ax2 = fig.add_subplot(122)
ax2.plot(x, f(x), "b", label='f(u)')
ax2.legend(loc='upper right')

plt.show()
```



It can be shown that solutions of the Cahn-Hilliard equation drive the system towards a state of minimal energy $\mathcal{E}(u)$.

Assuming a constant mobility $M = 1$, the Cahn-Hilliard equation simplifies to

$$\partial_t u + \kappa \Delta^2 u - \Delta(u^3 - u) = 0.$$

Here, Δ^2 denotes the **biharmonic** operator, which is the square of the Laplacian operator Δ, i.e. $\Delta^2 u = \Delta(\Delta u)$.

This is a hefty PDE, as it involves fourth-order spatial derivatives, a nonlinear term, and a time derivative. We do not discuss the well-posedness of the Cahn-Hilliard equation here which requires some sophisticated mathematical tools and techniques (but for the mathematical inclined reader, we refer to [Miranville, 2019] and [Bänsch *et al.*, 2023]). Instead, we will focus on the numerical solution of the Cahn-Hilliard equation.

The overall goal of this project is to gradually develop, implement and thoroughly test numerical methods for the Cahn-Hilliard equation in Python and to finally study the dynamics of phase separation of a binary mixture by means of numerical simulations.

To guide you through this project and demonstrate how to develop a numerical solver for a complex PDE in a structured way, we have divided this project into 6 smaller subtasks. In each of the tasks, you will be asked to consider simplified versions of the Cahn-Hilliard equation, develop a corresponding numerical method while also discussing some of their theoretical properties, and validate your implementation using carefully craft tests. Through the series of test problems, we will gradually add complexity to the model and the numerical methods. Afterwards you will have a (actually two!) fully functional and thorougly test solver(s) for the Cahn-Hilliard equation at hand, which you then can use to study the phase separation of a binary mixture.

The project will consists of the following tasks:

1. Before we start implementing numerical methods, we have a closer look at some of the fundamental theoretical properties of the Cahn-Hilliard, which we then later will try to observe in the numerical experiments.
2. Turning to the implementation tasks, we will start by implementing a spectral solver for the biharmonic-type problem of the form

$$\Delta^2 u + cu = g$$

Variants of this equation will then later solved for each time-step in a time-stepping scheme for the Cahnn-Hilliard equation.

3. Next, we will combine our spectral solver with a simple time-stepping schemes for the Cahn-Hilliard equation where we neglect the nonlinear terms for the moment. The resulting time-dependent PDE is the biharmonic cousin of the heat equation:

$$\partial_t u + \kappa \Delta^2 u = g.$$

4. In the next step, we will add the missing nonlinear term and consider a first-order time-stepping scheme for the full Cahn-Hilliard problem, where we discuss the challenges with and approaches for handling the non-linear term.
5. Afterwards, we will consider a more sophisticated, second-order in time method based on a 2 stage Runge-Kutta method. derive.
6. Finally, we will use the numerical method to study the phase separation of a binary mixture in a simple domain more closely. We will first briefly review the known typical phenomena appearing in phase separation dynamics before you will be asked to conduct your own numerical experiments to study these phenomena in more detail.

Important note: Throughout the project, we will only consider periodic boundary conditions for a simple domain $\Omega = [0, L_x) \times [0, L_y)$, which allows us to use Fourier spectral methods for the spatial discretization.

6.1 Guidelines and tips for the project

- **Proper mathematical descriptions.** Before you start implementing a numerical method, **always provide a short but complete mathematical description of the method you are about to implement**. E.g. for a time-stepping method, describe in mathematical terms how you compute the solution at the next time step based on the solution at the current time step. This will help you to avoid mistakes and to understand the behavior of the method.
- **Well-documented code.** Make sure that your code is well documented, in particular that you write a short description of the purpose of each function in the form of docstrings and that you provide inline comments (1-2 lines) briefly explaining the relevant code blocks.
- **Presentation of results.** When presenting results, do not just e.g. show a picture of a solution or display a convergence table, but make sure that you comment on your results sufficiently, in particular when we asked guiding questions or if the results are unexpected, or if you observe something interesting.
- **Visualizations.** Include visualizations of your results to illustrate your findings. You can use the plotting/animation functions provided in the `project_tools.py` module which you can find in this folder. Alternatively, you can of course use your own plotting functions. **But when you submit your final report, make sure that you remove all cell outputs produced from the `create_animation` function** or similar animation related output. Those take a lot of storage and make the notebook unnecessarily large. Instead we will ask you to either provide pictures of solution snapshots or to generate a more lightweight gif animation consisting of a few frames.

  For interactive visualizations where you can zoom in/out, make animation etc. you should execute the magic command `%matplotlib widget` once somewhere in the beginning of the notebook. But when you generate the final report, switch back to `%matplotlib inline` to avoid large notebook files/storage of animations. Finally, if you use `%matplotlib widget` in conjunction with Vscode, you sometimes might run into problems with the plots not showing up. This happens when too many figures have been instantiated. Best course of action is then to restart Vscode, simply resetting the output cells and kernel of the notebook won't help.
- **Use of AI.** Please document your use of AI tools in your project properly by filling out the AI declaration form you can find on NTNU's internal wiki, either in English, Norwegian (Bokmål) or Norwegian (Nynorsk). The form should be submitted in addition the final report. Obviously, we can not and will not put an restrictions on how you use AI tools in your project, but we would like to know if/how you used them and what you used them for, and how you made sure that AI generated results are reliable. The form should be submitted with the final report. Download the AI declaration form for technical and science subjects
- **Submission.** As before, the final report should be submitted as a single Jupyter notebook which has been executed from start to finish. The notebook should contain all relevant code, results, and discussions. Make sure that the code has been completely executed without errors and that the results are displayed in the notebook.

6.2 Task 1: A first closer look at the Cahn-Hilliard equation

Before we start developing a numerical method for the Cahn-Hilliard equation, let us discuss some basic properties of the so physical meaning of the different terms in the equation.

1. Plot the logarithmic potential $F(u)$ for $u \in (-1, 1)$ for some θ values in $[0.7, 1.6]$ with with $\theta < \theta_c$, $\theta = \theta_c$ and $\theta > \theta_c$ for a critical temperature $\theta_c = 1.5$ What do you observe?
2. Show that the Cahn-Hilliard equation is invariant under the transformation $u \mapsto -u$, i.e. if $u(x, t)$ is a solution, then $-u(x, t)$ is also a solution.
3. Show that solutions of the Cahn-Hilliard equation which are periodic on a rectangular domain $\Omega = [0, L_x) \times [0, L_y)$ are mass conservative in the sense

$$\frac{\mathrm{d}}{\mathrm{d}t} \int_\Omega u(x, t) \, \mathrm{d}x = 0.$$

   In other words, the total mass $\int_\Omega u(x, t) \, \mathrm{d}x$ does not change over time.

Hint: Multiply the Cahn-Hilliard equation by constant 1 and integrate over a rectangular domain and use the divergence theorem to perform integration by parts. What can you say about the boundary integral?

6.3 Task 2: A spectral solver for the biharmonic equation

Next, following the example of the Poisson solver from the lecture, we ask you to implement a spectral solver for the biharmonic equation supplemented with a 0-order term in $2d$.

The resulting equation reads

$$\Delta^2 u + cu = f \quad \text{in } \Omega,$$

on a rectangular domain $\Omega = [0, L_x) \times [0, L_y)$, where Δ^2 is the biharmonic operator and $c \geqslant 0$ is a non-negative constant.

The problem is to be solved on a given rectangular domain $\Omega = [0, L_x) \times [0, L_y) \subset \mathbb{R}^2$ with periodic boundary conditions. More precisely, we assume that all derivatives up to order 3 are periodic. Different side lengths L_x and L_y are allowed and should be incorporated in the implementation.

Now implement a spectral solver for the biharmonic equation using the fast Fourier transform. The solver should be implemented in a function of the form

```
def biharmonic_solver(X, Y, F, c, mean=0.0):
    """
    Solve the biharmonic equation in 2D using the spectral method.

    Parameters:
        X (ndarray): 2D array of x-coordinates.
        Y (ndarray): 2D array of y-coordinates.
        F (ndarray): 2D array representing the right-hand side of the biharmonic␣
↪equation.
        c (float): Constant coefficient in the biharmonic equation.
        mean (float, optional): Desired mean value of the solution in case c = 0.␣
↪Default is 0.0.

    Returns:
        U (ndarray): 2D array representing the solution to the biharmonic equation.
    """
    pass # Add your code here
```

Your solver should also be able to gracefully handle the case $c = 0$ by prescribing a (user-defined) mean value for the solution. This will come in handy when you are asked to test your solver using manufactured solutions which do not necessarily have a 0 mean value.

To verify the correctness of your implementation, run a number convergence studies using the **manufactured solutions** from the following 2 test cases, each of them posed on the rectangular domain $\Omega = [0, 2\pi) \times [0, 4\pi)$:

- $u(x, y) = \sin(8(x - 1)) \cos(4y)$, $c = 1$, and $N_x = 4, 8, 15, 16, 20, 32$ and $N_y = 2N_x$.
- $u(x, y) = \exp(\sin^2(x) + \cos(2y))$, $c = 0$, $N_x = 4 + 4k$, $k = 0, 1 \ldots, 9$ and $N_y = 2N_x$.

Remember to compute the corresponding right-hand side f for the manufactured solutions. For each series, compute the experimental order of convergence (EOC) with respect to the maximum norm over the grid points report them in a table.

Provide also a surface plot of both the exact solution, the numerical solution and the error function for $N_x = 15, 16$ in the first series and $N_x = 32$ in the second series.

Discuss the results.

Hints:

- To compute the right-hand side f for the manufactured solutions, it might be helpful to start using the `sympy` module, see lecture notes and the tutorial. This will definitely pay-off later when you are asked to manufacture solution for the Cahn-Hilliard equation!
- Recall that numerically the check $c = 0$ is not trivial. As a rule of thumb, you can check e.g. whether c is smaller the smallest non-zero **quartic** frequency in the Fourier space, i.e., $c < \text{tol} \cdot k_{\text{min}}^4$, where k_{min} is the smallest non-zero frequency in the Fourier space.

6.4 Task 3: A spectral solver for the transient biharmonic equation

As the third step towards a solver for the Cahn-Hilliard equation, you are now ask to implement, test and verify a solver for the **time-dependent** biharmonic equation

$$\partial_t u + \kappa \Delta^2 u = g. \tag{6.8}$$

As before, the problem is to be solved on a given rectangular domain $\Omega = [0, L_x) \times [0, L_y) \subset \mathbb{R}^2$ with periodic boundary conditions, i.e. the same considerations as for the biharmonic equation apply. Moreover, the problem is supplemented with **initial conditions** $u(0, x, y) = u_0(x, y)$, which we also assume to satisfy the periodic boundary conditions.

This problems resembles very closely the heat equation, the only difference being that the spatial differential operator is now given by the biharmonic operator, and not $-\Delta$ as in the heat equation. Consequently, we take a similar approach as we took for the heat equation in the lecture, and combine a Fourier spectral method in space with a one-step time-stepping methods in time. The one-step method we ask you to use here is called the θ-method.

6.4.1 The θ-method

Let's forget for the moment that we want to solve the biharmonic equation, and consider a general linear ODE of the form

$$\partial_t U = F(t, U), \quad U(t_0) = U_0.$$

The θ-method is defined as follows. As usual, we first discretize the time interval $[0, T]$ into N equidistant time steps of size $\tau = T/N$. The algorithm is started from the initial condition U^0. at time t_0. Then, given the solution U^n at time t_n, the new solution U^{n+1} at time step $t_{n+1} = t_n + \tau$ is computed as

$$U^{n+1} = U^n + \tau \left(\theta F(t_{n+1}, U^{n+1}) + (1 - \theta) F(t_n, U^n)\right) \quad \text{for } n = 0, 1, 2, \ldots, N - 1.$$

6.4.2 Theoretical tasks

1) Rewrite the θ-method as a 2-stage Runge-Kutta method, derive the corresponding Butcher table, and discuss the consistency order of the θ-method for various values of θ. For which values does the θ-method reduces to other well-known time-stepping methods?

Hint: Have a look at the order conditions for Runge-Kutta methods in the lecture notes.

2) Next, determine the stability function $r_\theta(z)$ of the θ-method and plot the stability region of the θ-method in the complex plane for $\theta = 0, 0.25, 0.498, 0.5, 0.502, 0.75, 1$. For which values of θ does the θ-method seem to be A-stable? What do you conjecture for a general θ? Do you have an idea of how the stability region of the θ-method looks in general/depends on θ?

3) Finally, we ask you to put your conjecture on solid mathematical grounds. More precisely, determine mathematically the stability region of the θ-method and how it depends on the value of θ.

Hint: The border between the stable and unstable region is given by $\partial S_\theta = \{z \in \mathbb{C} : |r_\theta(z)| = 1\}$ and this can be transformed into a simple equation for a circle in the complex plane. How does the center and radius of this circle depend on θ?

6.4.3 Computational tasks

After this theoretical warm-up, we now turn to the implementation of the spectral solver for the transient biharmonic equation. Please implement a solver for the transient biharmonic equation combining the Fourier spectral method in space with the θ-method in time.

1) Before you start implementing, please provide a brief **mathematical description** of the resulting numerical scheme, describing how a new solution is computed from the previous solution for each time step.

For the implementation, the following specifications for the solver interface be met: The solvers should be implemented as a **generator function** using the `yield` statement to return the **discrete Fourier transform** of the solution at each time step, and the current time. The generator function should have the following signature:

```
def transient_biharmonic_solver(*, kappa,
                                   X, Y, U0,
                                   t0, T, Nt,
                                   theta=0.5,
                                   g=None):
    """
    Solve the transient biharmonic equation using the theta method.

    Parameters:
    -----------
    kappa (float): Diffusion coefficient.
    X (ndarray): 2D array of x-coordinates.
    Y (ndarray): 2D array of y-coordinates.
    U0 (ndarray): Initial condition array.
    t0 (float): Initial time.
    T (float): Final time.
    Nt (int): Number of time steps.
    g (callable, optional): Source term function g(X, Y, t). Defaults to None.

    Yields:
    -------
    tuple: A tuple containing the discrete Fourier transform of U at t, and the␣
↪current time t.
    """
```

(continues on next page)

(continued from previous page)

```
    # Prepare relevant data for Fourier transform
    ...
    ...

    # Compute DFT of the initial value
    ...

    # Add your time-stepping loop here
    # Time stepping
    t = t0
    dt = (T-t0)/Nt

    # For convenience when plotting, computing errors, etc.,
    # return the initial solution and initial time.
    yield U_hat, t

    while t < T-dt/2:
        # Solve for next time step and update time
        ...
        ...
        yield Uhat, t
```

which then can be used to solve the transient biharmonic as follows:

```
# Set up the problem
...
...

solver = transient_biharmonic_solver(kappa=kappa,
                                     X=X, Y=Y, U0=U0,
                                     t0=t0, T=T, Nt=Nt,
                                     theta=theta,
                                     g=None)

for Uhat, t in solver:
    # Do something with the solution
    ...
    ...
```

To verify your implementation and assess the stability and accuracy of the solver, we ask you perform the following tasks:

2) First, study the convergence order of the time-stepping scheme for $\theta \in \{0, 0.5, 1\}$. Use the function $u_{\text{ex}}(x, y, t) = \sin(x)\cos(y)\exp(-\lambda\kappa t)$ to manufacture a solution for the transient biharmonic equation. Choose λ such that the manufactured solution leads to a homogeneous source term $g(x, y, t) = 0$.

Set $\Omega = [-\pi, \pi)^2$ and $\kappa = 1$ and choose $N = N_x = N_y = 20$ sampling points/subintervals in each space direction. Furthermore, set $t_0 = 0$, $T = 1$. Now solve the problem successively for $N_t = 10, 20, 40, 80, 160, 320, 640$ time steps with equidistant time steps $\tau = T/N_t$. For each run calculate the error in the so-called $L^\infty L^\infty$ norm defined by

$$\|E\|_{L^\infty L^\infty} = \max_{k\in\{0,N_t\}} \max_{i,j\in\{1,\dots,N\}} |u_{\text{ex}}(x_i, y_j, t_k) - U^k(x_j, y_j)|,$$

and compute the EOC with respect to the time step size τ (or equivalently the number of time steps N_t).

Display your results in a table showing the number of steps, resulting error and experimentally observed convergence order for each refinement in time. Do this for $\theta = 0, 0.5, 1$, starting with $\theta = 1$, then $\theta = 0.5$ and finally $\theta = 0$.

Discuss the results and relate them to the theoretical results you derived in the first part of the task. Comment on the

suitability of the $\theta = 0$-method for the transient biharmonic equation. In particular, explain why for $\theta = 0$ the scheme fails by deriving the resulting CLF condition for this case.

Finally, based on the derived CFL condition find the minimal number N_{CLF} of time steps N_t for which the scheme should be stable and repeat the convergence study for $\theta = 0$ with $N_t = 0.5N_{CLF}, N_{CFL}, 2N_{CFL}, 4N_{CFL}$.

3) Finally, to prepare yourself for computing manufactured solutions for the Cahn-Hilliard equation in the next task, we ask you to rerun the convergence study for the transient biharmonic equation for $\theta = 0.5, 1$ with the manufactured solution

$$(\exp(1 + \sin(x)\sin(x)) + \exp(1 + \cos(y)\cos(y)))\exp(-4\kappa t)$$

resulting in a highly non-trivial source term $g(x, y, t)$. Use $N_t = 10, 20, 40, 80, 160, 320, 640$ as before for the EOC study.

Hint: Make your live easy and use the sympy to manufacture solutions by computing the resulting source terms. See the example from the lecture notes.

6.5 Task 4: A first IMEX solver for the Cahn-Hilliard equation

Now we have finally arrived at the point where we can start implementing a first solver for the Cahn-Hilliard equation. You are tasked with extending (parts of) your transient biharmonic solver from the previous task to include the additional non-linearity $\Delta f(u)$ arising from the Cahn-Hilliard equation. The non-linearity poses several challenge for the time-stepping scheme:

- As an explicit time-stepping scheme needs to satisfy severe time-step restrictions when combined with the transient biharmonic operator (see previous task), an implicit time-stepping scheme seems to be more appropriate. On the other hand, the non-linearity $\Delta f(u)$ is computationally expensive to solve implicitly, as it would require the solution of a non-linear system of equations at each time step.
- Moreover, as the product of functions translates into convolution in the Fourier space, the non-linearity $f(u)$ translates into a convolution of the form $\widehat{u^3}(k) - \hat{u}(k) = \hat{u} * \hat{u} * \hat{u}(k) - \hat{u}(k)$, which is extremely expensive to compute!

To address these challenge, we need to briefly discuss some import concepts when solving the Cahn-Hilliard equation.

6.5.1 IMEX time-stepping scheme based on the Implicit/Explicit Euler method

The first challenge can be addressed by using an Implicit-Explicit (IMEX) time-stepping scheme. The IMEX approach is best explained by looking at a general ODE system of the form

$$U_t + \mathbf{L}U = \mathbf{N}(U), \quad U(0) = U_0,$$

Again, we start from a equidistant time grid $t_n = n\tau$ with $\tau = (T - t_0)/N_t$ and denote the numerical solution at time t_n by U^n. Starting from the initial condition U^0 at time t_0, and given the solution U^n at time t_n, the new solution U^{n+1} at time step $t_{n+1} = t_n + \tau$ is then computed as

$$U^{n+1} + \tau\mathbf{L}U^{n+1} = U^n + \tau\mathbf{N}(U^n).$$

This explains the name "IMEX" (Implicit-Explicit) time-stepping scheme, as the linear parts of the equation are treated implicitly, while the non-linear part is treated explicitly. In particular, for $\mathbf{N} = 0$, the scheme reduces to the Implicit Euler method, while for $\mathbf{L} = 0$, the scheme reduces to the Explicit Euler method.

Note that this idea will be used to solve the Cahn-Hilliard equation in the Fourier space, so below you have to think of U^n being $\hat{u}^n$ and $\mathbf{L}$ being the discrete Fourier transform of the Biharmonic operator (potentially plus some lower terms).

6.5.2 Pseudo-spectral methods for nonlinear PDEs

To address the second challenge, we will implement a so-called **pseudo-spectral** method. The idea is to compute the non-linear term in the physical space, while the linear terms are computed in the Fourier space. This allows for an efficient computation of the non-linear term, while still benefiting from the high accuracy of the Fourier space method for the linear terms. So to compute $\widehat{N(u^n)}$ at t_{n+1}, we start from the (DFT of the) solution of the previous time step $\hat{u}^n$ and proceed as follows:

$$\hat{u}^n \overset{\mathcal{F}^{-1}}{\mapsto} u^n \mapsto N(u^n) \overset{\mathcal{F}}{\mapsto} \widehat{N(u^n)}.$$

So in contrast to the previous task, where we only transformed the solution back to the physical space for "post-processing" tasks such as visualization or error computations, we now need to transform the solution back and forth for the actual solution computation. Hence, the name **pseudo-spectral** method is frequently used to refer to such kind of methods, as we cannot stay in the Fourier space for the entire computation.

6.5.3 Convex splitting of the Cahn-Hilliard equation

Finally, we need to specify how exactly we want to split the linear parts from the non-linear parts of the Cahn-Hilliard equation. Note that the the term

$$\Delta f(u) = \Delta(u^3 - u)$$

in the Cahn-Hilliard equation actually contains a linear part $-\Delta u$ and a non-linear part Δu^3.

If we decide to treat all linear terms implicitly and all non-linear terms explicitly, we can split the Cahn-Hilliard equation as follows:

$$\frac{u^{n+1} - u^n}{\tau} + \kappa\Delta^2 u^{n+1} + \Delta u^{n+1} = \Delta(u^n)^3 \tag{6.9}$$

Unfortunately, the $+\Delta u^{n+1}$ has an unfortunate sign, which we see when transforming this equation to the Fourier space:

$$\hat{u}^{n+1} + \tau(\kappa\tilde{\mathbf{k}}^4\hat{u}^{n+1} - \tilde{\mathbf{k}}^2\hat{u}^{n+1}) = \hat{u}^n + \tau\tilde{\mathbf{k}}^2\widehat{(u^n)^3} \tag{6.10}$$

Consequently, this solution method can get unstable if $1 + \kappa\tilde{\mathbf{k}}^4 - \tilde{\mathbf{k}}^2 < -1$ which can easily happen for reasonable values of κ and $\mathbf{k}$.

As a remedy, it is common to split the Δ slightly differently, starting from a splitting parameter a to obtain

$$\Delta f(u) = \Delta(u^3 - u) = \underbrace{a\Delta u}_{f_1(u)} + \underbrace{\Delta u^3 - (1+a)\Delta u}_{f_2(u)}.$$

where $f_1(u)$ is treated implicitly and $f_2(u)$ is treated explicitly. Note that for $a = -1$, we recover the original complete splitting. But typically, we choose at least $a \geqslant 0$ to avoid the mentioned sign issue in the Fourier space, but for reasons we don't have time to discuss here, one chooses $a \sim 1.5$. For $a \geqslant 2$, this splitting results from a "convex" splitting of the free energy functional.

The result is then the following IMEX scheme for the Cahn-Hilliard equation:

$$\frac{u^{n+1} - u^n}{\tau} + \kappa\Delta^2 u^{n+1} - a\Delta u^{n+1} = \Delta\big((u^n)^3 - (1+a)u^n\big) \tag{6.11}$$

which of course needs to be translated to the Fourier space.

6.5.4 Computational tasks

1) Before you start implementing, please provide a brief **mathematical description** of the resulting numerical scheme, in particular, describe how a new solution is computed in the Fourier space from the previous solution for each time step. To ensure that you later can verify the correctness of your implementation using the method of manufactured solutions, make sure that your solver can solve the Cahn-Hilliard equation with a source term $g(x, y, t)$, i.e., the equation to be solved should be of the form

$$\partial_t u - \nabla \cdot (M \nabla (f(u) - \kappa \Delta u)) = g \quad \text{on } \Omega \times (0, T) \tag{6.12}$$

Treat the source term g implicitly in the time-stepping scheme (similar to the linear terms).

2) Implement the resulting IMEX scheme for the Cahn-Hilliard equation. For the implementation, the solver interface should meet similar specifications as in the previous task, i.e., the solvers should be implemented as a **generator function** using the `yield` statement to return the **discrete fourier transform** of the solution at each time step, and the current time. The generator function should have the following signature:

```
def cahn_hilliard_backward_euler(*,
                                 kappa,
                                 X, Y, U0,
                                 t0, T, Nt,
                                 g,
                                 alpha=1.5):
    """
    Implements the Cahn-Hilliard equation solver using the backward Euler method
    with a convex-concave splitting approach.

    Parameters:
    -----------
    kappa : float
        Diffusion coefficient for the biharmonic operator.
    X : ndarray
        2D array representing the x-coordinates of the grid.
    Y : ndarray
        2D array representing the y-coordinates of the grid.
    U0 : ndarray
        Initial condition for the solution.
    t0 : float
        Initial time.
    T : float
        Final time.
    Nt : int
        Number of time steps.
    g : callable or None
        Source term as a function of (X, Y, t). If None, no source term is applied.
    alpha : float, optional
        Convex-concave splitting parameter. Default is 1.5.

    Yields:
    -------
    tuple: A tuple containing the discrete Fourier transform of U at t, and the␣
↪current time t.

    """
```

3) As before, we want you to verify the correctness of your implementation by comparing the numerical solution to the exact solution for a simple test case. To this end, we ask you to run a convergence study similar to the one in the previous task.

To manufacture a solution, start from the exact solution

$$u_{\text{ex}} = \sin(x)\cos(y)\exp(-4\kappa t)$$

Clearly, this solution *will not satisfy* the Cahn-Hilliard equation with a non-zero source term due to the additional non-linearity. Instead, compute the corresponding right-hand side g for the Cahn-Hilliard equation. This might be a good time to automate the computation of the right-hand side using symbolic computation tools such as `sympy` (see lecture notes)!

Now for $\Omega = [0, 16\pi)^2$, $N_x = N_y = 64$, $t_0 = 0$, $T = 1$ and $\kappa \in \{1.0, 0.01\}$, run convergence studies for the IMEX solver using $N_t \in \{100, 200, 400, 800, 1600, 3200\}$, where you tabulate the $L^\infty L^\infty$ error against the number of number of time steps N_t. Discuss briefly your results.

6.6 Task 5: A more sophisticated IMEX solver for the Cahn-Hilliard equation

Before we finally run some more interesting simulation of the actual Cahn-Hilliard equation with realistic initial conditions (and no source term!), we want to improve the previous IMEX solver for the Cahn-Hilliard equation and implement a more sophisticated solver for the Cahn-Hilliard equation which can be shown to be second-order accurate in time.

Again, the scheme is best explained by looking at a general ODE system of the form

$$U_t = \mathbf{L}U + \mathbf{N}(U), \quad U(0) = U_0,$$

Now the Song scheme from [Song, 2016] is a 3-stage IMEX Runge-Kuttta scheme given by

$$U^{(1)} = U^n + \tau(\mathbf{L}U^{(1)} + \mathbf{N}(U^n)), \tag{6.13}$$
$$U^{(2)} = \alpha_{10}U^n + \alpha_{11}U^{(1)} + \beta_1\tau(\mathbf{L}U^{(2)} + \mathbf{N}(U^{(1)})), \tag{6.14}$$
$$U^{n+1} = \alpha_{20}U^n + \alpha_{21}U^{(1)} + \alpha_{22}U^{(2)} + \beta_2\tau(\mathbf{L}U^{n+1} + \mathbf{N}(U^{(2)})), \tag{6.15}$$

6.6.1 Theoretical tasks

1) ~~Show that this~~ This scheme has consistency order $p = 2$ if the coefficients β_i, α_{ij} satisfies the following order conditions:

$$\begin{cases} \alpha_{10} + \alpha_{11} = 1, \\ \alpha_{20} + \alpha_{21} + \alpha_{22} = 1, \\ \alpha_{21} + \alpha_{22}\alpha_{11} + \alpha_{22}\beta_1 + \beta_2 = 1, \\ \alpha_{21} + \alpha_{22}\alpha_{11} + \alpha_{22}\alpha_{11}\beta_1 + \alpha_{22}\beta_1^2 + \alpha_{21}\beta_2 + \alpha_{22}\alpha_{11}\beta_2 + \alpha_{22}\beta_1\beta_2 + \beta_2^2 = \frac{1}{2}, \\ \alpha_{22}\beta_1 + \alpha_{11}\beta_2 + \beta_1\beta_2 = \frac{1}{2}. \end{cases}$$

~~To arrive at these equations, employ a similar Taylor-expansion technique as shown in one of the homework exercises and in the derivation of the consistency order of Heun's method and the improved Euler methods.~~

This set of equations cannot be solved uniquely. Here, we consider the following coefficients which solve the above

equations:

$$\alpha_{10} = \frac{3}{2}, \quad \alpha_{11} = -\frac{1}{2}, \quad \alpha_{20} = 0, \quad \alpha_{21} = 0, \quad \alpha_{22} = 1, \quad \beta_1 = \frac{1}{2}, \quad \beta_2 = 1. \tag{6.16}$$

$$\alpha_{10} = 2, \quad \alpha_{11} = -1, \quad \alpha_{20} = \frac{1}{2}, \quad \alpha_{21} = 0, \quad \alpha_{22} = \frac{1}{2}, \quad \beta_1 = 1, \quad \beta_2 = 1. \tag{6.17}$$

$$\alpha_{10} = 2, \quad \alpha_{11} = -1, \quad \alpha_{20} = 0, \quad \alpha_{21} = \frac{1}{2}, \quad \alpha_{22} = \frac{1}{2}, \quad \beta_1 = 1, \quad \beta_2 = \frac{1}{2}. \tag{6.18}$$

$$\alpha_{10} = \frac{5}{2}, \quad \alpha_{11} = -\frac{3}{2}, \quad \alpha_{20} = \frac{2}{3}, \quad \alpha_{21} = 0, \quad \alpha_{22} = \frac{1}{3}, \quad \beta_1 = \frac{3}{2}, \quad \beta_2 = 1. \tag{6.19}$$

2) Apply this scheme now to the the Cahn-Hilliard equation formulated in the Fourier space. Here, in each of the above stages, you should apply the **same convex splitting** as in the previous task. As always, before you start implementing the scheme, provide a brief **mathematical description** of the resulting numerical scheme, in particular, describe how a new solution is computed in the Fourier space from the previous solution for each time step.

6.6.2 Computational tasks

3) Rerun the EOC study from the previous task for the new IMEX scheme for all 4 sets of coefficients, compare the results against each other and with the previous IMEX scheme. Which of the 4 Song coefficients sets will you favor for the remaining project? Here, you can focus on the case $\kappa = 0.01$.

Important note: To take into account an inhomogeneous right-hand side $\mathbf{G}$, you can implement the following slightly modified scheme:

$$U^{(1)} = U^n + \tau(\mathbf{L}U^{(1)} + \mathbf{N}(U^n) + \mathbf{G}^{n+1/2}), \tag{6.20}$$

$$U^{(2)} = \alpha_{10}U^n + \alpha_{11}U^{(1)} + \beta_1\tau(\mathbf{L}U^{(2)} + \mathbf{N}(U^{(1)}) + \mathbf{G}^{n+1/2}), \tag{6.21}$$

$$U^{n+1} = \alpha_{20}U^n + \alpha_{21}U^{(1)} + \alpha_{22}U^{(2)} + \beta_2\tau(\mathbf{L}U^{n+1} + \mathbf{N}(U^{(2)}) + \mathbf{G}^{n+1/2}), \tag{6.22}$$

where $\mathbf{G}^{n+1/2} = \mathbf{G}(t_n + \tau/2)$.

6.7 Task 6: Simulation of phase separation phenomena

In this final task, we will use the Cahn-Hilliard solvers from Task 4 and Task 5 to study the dynamics of phase separation phenomena more closely, using realistic initial conditions and zero external forces.

The dynamics of phase separation phenomena are driven by the antagonist effects of mixing and interface energy. Due to its double-well potential, the mixing energy tends to drive the system away from the homogeneous state $u = 0$ towards the phase-separated states $u = \pm 1$. On the other hand, the interface energy tends to minimize the total interfacial area between the two phases, as phase boundaries are regions of high energy where the field u exhibits large gradients to do its rapid change from $+1$ to -1. If the total energy of the system had been solely determined by the interface energy, the a completely homogeneous state would have been preferred. It is this competition between the two effects which leads to the formation of complex patterns and structures during the phase separation process. In the absence of external forces, the system evolves towards a state of minimal free energy, leading to the formation of distinct phases with characteristic length scales. This process is characterized by several distinct stages, including spinodal decomposition, nucleation and growth, and coarsening and Oswald ripening.

Spinodal decomposition and Ostwald ripening are two distinct mechanisms observed during phase separation phenomena. Spinodal decomposition occurs when a homogeneous mixture becomes unstable and spontaneously separates into distinct phases due to small fluctuations in composition. This process is characterized by the rapid formation of interconnected structures with a characteristic wavelength, driven by the reduction of free energy. In contrast, Ostwald ripening describes the later-stage coarsening process, where larger domains grow at the expense of smaller ones due to differences in chemical potential. This leads to a reduction in the total interfacial energy of the system, resulting in the growth of larger, more

stable structures over time. Together, these processes illustrate the interplay between mixing energy and interface energy in driving phase separation dynamics.

The time scales for the spinodal decomposition and Ostwald ripening are determined by the characteristic length scale of the system, the mobility of the components, and the temperature of the system. In general, spinodal decomposition occurs on (much) shorter time scales, while Ostwald ripening occurs on longer time scales.

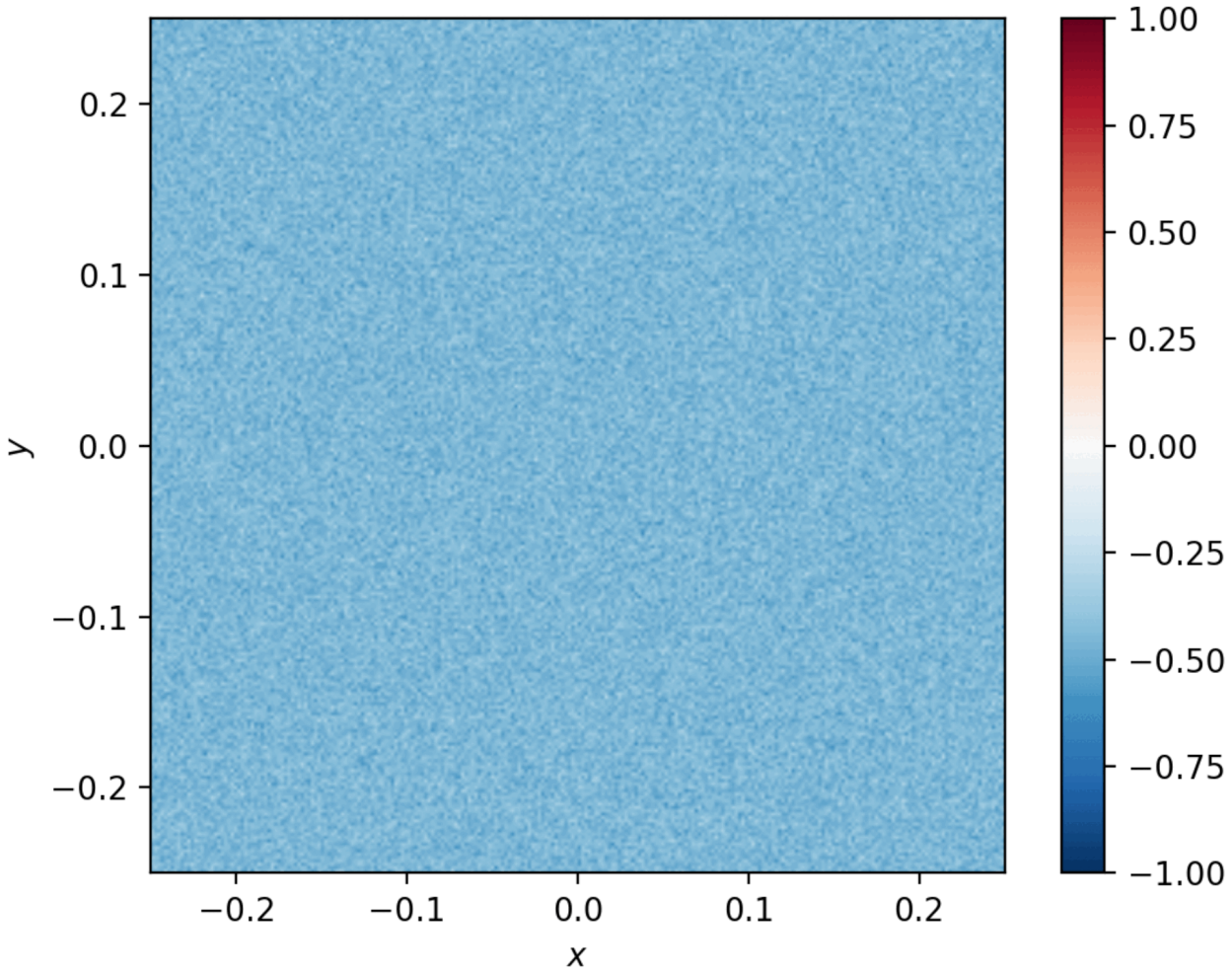


Figure. Combined snapshops illustrating initial spinodal decomposition and subsequent Ostwald ripening in a phase separation process. Note the animation is not linear in time, and the time between the shown snapshots is not constant (which you will find out when you run the simulation!).

The word "spinodal" is term from material science that describe the transformation of a system of two or more components in a metastable phase into two stable phases.

The word "Ostwald" refers to the German chemist Wilhelm Ostwald, who first described the ripening process in 1896.

To this end we want you to run a number of simulations of the Cahn-Hilliard equation based on following setup

- Domain: $\Omega = [0, 0.5]^2$
- Time interval: $I = [0, 4.0]$, and $I = [0, 0.01]$
- Spatial resolution: $Nx = Ny = 256$
- $\kappa = 0.0025^2$

- two different initial conditions:

$$u_0(x, y) = 0.05\text{rand}(x, y) \tag{6.23}$$
$$u_0(x, y) = -0.45 + 0.05\text{rand}(x, y) \tag{6.24}$$

- Two different time step sizes $\tau \in \{10^{-3}, 10^{-4}\}$

The first initial condition is a random perturbation of the base state $u_0(x, y) = 0.0$, and can produced by the following code snippet:

```
rng = np.random.default_rng(12345)
noise = 0.05
u0_base = 0.0
U0 = np.ones_like((Ny, Nx))
U0 = np.full((Ny, Nx), u0_base) +  noise*rng.standard_normal((Ny, Nx))
```

Please use the random seed 12345 used above to make your code reproducible and the results comparable.

For each simulation the following tasks should be performed:

- Plot the total mass $\int_\Omega u \, \mathrm{d}x$ of the concentration field u at each time step.
- Plot both the mixing energy, the interface energy and the total energy of the system at each time step.
- Generate a number of snapshots of the concentration field u at different time steps.

Don't forget to explain how you calculate both the total mass and the energies!

Hint. It might be useful to create an animation/movie of the snapshots to inspect the dynamics of the phase separation more closely. When you generate animations, do not more than 200-400 frames, otherwise the creation of the animation will just take too long.

Discuss the results with respect the two different initial conditions, the two different time step sizes, and the two different solvers. In particular, you should think about the following questions:

- How does the total mass of the concentration field evolve in time?
- How does the mixing energy, the interface energy and the total energy of the system evolve in time?
- Can you spot any differences in the evolution of the energies for the two different initial conditions? If so, can you relate those to the evolution of the snapshots?
- How does the time step size influence the evolution of the energies and the snapshots?
- How does solver choice influence the evolution of the energies and the snapshots?
- Identify roughly when the spinodal decomposition and the Ostwald ripening occur in the simulations? At which time does the evolution slows down siginificantly? And when does the system reach am equilibrium state?

When you have identified roughly the time scale $[0, T_{sd}]$ for the spinodal decomposition, rerun the simulation with your favorite/best working solver for a time interval $I = [0, T_{sd}]$ using 40000 time steps.

Finally, we want you to generate and include an animated gif image combining 3-5 snapshots of the concentration field u during the spinodal decomposition and 3-5 snapshots from Ostwald ripening process including on of the final equilibrium state.

Note. If your Song based solver is not working properly, fall back to the simpler IMEX solver from Task 4 to run the simulations.

HAVE FUN!

CHAPTER

SEVEN

BIBLIOGRAPHY

BIBLIOGRAPHY

[BK22] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2022. URL: https://books.google.com/books?hl=en&lr=&id=rxNkEAAAQBAJ&oi=fnd&pg=PR9&dq=brunton+kutz+data-driven+science+and+engineering&ots=knCXY4PC4n&sig=UTsnBqxyw2_rA8qigc8-oRrmqAM (visited on 2025-03-19).

[BanschDGP23] Eberhard Bänsch, Klaus Deckelnick, Harald Garcke, and Paola Pozzi. *Interfaces: Modeling, Analysis, Numerics*. Volume 51 of Oberwolfach Seminars. Springer Nature Switzerland, Cham, 2023. ISBN 978-3-031-35549-3 978-3-031-35550-9. URL: https://link.springer.com/10.1007/978-3-031-35550-9 (visited on 2024-01-04), doi:10.1007/978-3-031-35550-9.

[GO14] Gene H. Golub and James M. Ortega. *Scientific Computing and Differential Equations: An Introduction to Numerical Methods*. Elsevier, 2014.

[HW93] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Volume 375. Springer, 1993.

[Hol23] Mark H. Holmes. *Introduction to Scientific Computing and Data Analysis*. Volume 13 of Texts in Computational Science and Engineering. Springer International Publishing, Cham, 2023. ISBN 978-3-031-22429-4 978-3-031-22430-0. URL: https://link.springer.com/10.1007/978-3-031-22430-0 (visited on 2025-01-12), doi:10.1007/978-3-031-22430-0.

[LL16] H.P. Langtangen and S. Linge. *Programming for Computations — Python*. Volume 15 of Texts in Computational Science and Engineering. Springer, 2016.

[Mir19] Alain Miranville. *The Cahn-Hilliard Equation: Recent Advances and Applications*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, January 2019. ISBN 978-1-61197-591-8. URL: https://epubs.siam.org/doi/book/10.1137/1.9781611975925 (visited on 2023-01-29), doi:10.1137/1.9781611975925.

[Son16] Huailing Song. Energy SSP-IMEX Runge–Kutta methods for the Cahn–Hilliard equation. *Journal of Computational and Applied Mathematics*, 292:576–590, January 2016. URL: https://www.sciencedirect.com/science/article/pii/S0377042715003921 (visited on 2024-12-18), doi:10/g5c5vf.

PROOF INDEX

fou:def:trig_interp

fou:prop:inverse_dft

fou:thm:best_approx_trig_poly

fou:thm:orthogonality

fou:thm:real_trig_interp

fou:thm:trig_interp

four:exa:ortho-complex

four:exa:ortho-real

my-observation

observation-0

observation-11

observation-12

observation-3

observation-5

observation-8

ode:alg:euler-meth

ode:alg:heun-meth

ode:def:consist_err

ode:def:global_err

ode:def:one-step-meth

ode:def:runge-kutta-meth

ode:exa:increment_function_euler_heun

ode:exa:lotka-volterra

ode:exa:spreading-disease

ode:exa:time-dep-coef

ode:exa:van-der-pol

ode:thm:osm-convergence-theory

proposition-0

remark-0

remark-2

remark-4

remark-6

theorem-13

theorem-14

theorem-7

thm-csr-err-estim

thm-ctr-error-est

thm-err-est-qr

thm-interpol-error

thm:rk-convergence

thm:rk-order-conditions