Computational Physics

Exercise Sheet 01

Editha Jasniewicz, Rene-Marcel Lehner

SuSe 2023

ASSIGNMENT N°

01

Folder structure

```
1A_01_Jasniewicz-Lehner.zip
├── E1
│   ├── main.cpp
│   ├── makefile
│   └── plots.ipynb
├── E2
│   ├── E2.py
│   ├── __init__.py
│   ├── __pycache__
│   │   ├── E2.cpython-310.pyc
│   │   └── __init__.cpython-310.pyc
│   └── plotting2.ipynb
├── E3
│   ├── E3.py
│   └── plotting3.ipynb
└── outline.cpp
```

This time, the python scripts for plotting are provided, as well the `outline.cpp`.

E0: Comprehension Questions

Task:

1) What integration routine in 2) and 3) has the best accuracy? Which one is the best to use?

2) To what polynomial degree does Simpson's rule give exact results when approximating integrals of polynomials?

1) We are asked to compare the accuracy of the integration routines from part 2) and 3) of the exercise, which are the Riemann sum and Simpson's rule, respectively.

   The accuracy of an integration routine is often determined by the order of the error term. The error for the Riemann sum scales with the step size, which means it has an error of order $O(h)$, where h is the step size. On the other hand, Simpson's rule has an error of order $O(h^4)$.

   Since the error term in Simpson's rule is smaller, it has better accuracy than the Riemann sum for the same number of intervals or step size. Therefore, Simpson's rule is the best method to use among the two.

2) Simpson's rule gives exact results when approximating integrals of polynomials up to degree 3, i.e., cubic polynomials. This is because Simpson's rule is based on approximating the function using quadratic polynomials (degree 2) over subintervals, and when integrating quadratic polynomials exactly, it can also capture the exact integrals of lower-degree polynomials, such as linear (degree 1) and constant (degree 0) functions, as well as cubic polynomials (degree 3).

E1: Numerical differentiation

Task:

Numerical differentiation is a fundamental tool of any physicists repertoire in modern times. The goal of this Exercise is to practice this method and to create an understanding of the occurring errors.

Calculate the derivatives of the functions

$$f_1(x) = \sin(x)$$
$$f_2(x) = \begin{cases} 2 \cdot \lfloor \frac{x}{\pi} \rfloor - \cos(x \bmod \pi)) + 1 & \text{for } x \geq 0 \\ 2 \cdot \lfloor \frac{x}{\pi} \rfloor + \cos(x \bmod \pi)) + 1 & \text{for } x < 0 \end{cases}$$

analytically and numerically. The expression $\lfloor \rfloor$ is the floor function.
Implement the two-point method

$$f'_{\text{two-point}}(x,h) = \frac{f(x+h) - f(x-h)}{2h}$$

and the four-point method

$$f'_{\text{four-point}}(x,h) = \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}.$$

Use single precision (32-bit) floating point variables in your implementation so that numerical errors are more pronounced.

a) Calculate the first derivative of $f_1(x)$ using the two-point method at a suitable point. Plot the value of the derivative as a function of the step size h and make a suitable choice for h. Plot the error $\Delta f'_{1,\text{two-point}}(x)$ of the analytical result on the interval $[-\pi, +\pi]$.

b) Calculate the second derivative $f''_{1,\text{two-point}}(x)$ and, as before, plot the error and compare with the results of the first derivative. Determine a suitable h in an analogous way to (a).

c) Now calculate the derivative $f'_{1,\text{four-point}}(x)$ using the four-point method and compare the error with $\Delta f'_{1,\text{two-point}}(x)$.

d) Continue comparing the two-point rule and the four-point rule in the same way using the first derivative of $f_2(x)$.

Analytical Derivatives

The derivative of $f_1(x)$ is trivial.
For the derivative of $f_2(x)$ we need to understand the floor function and the operation `mod`. We can easily visualize the behavior of these functions.

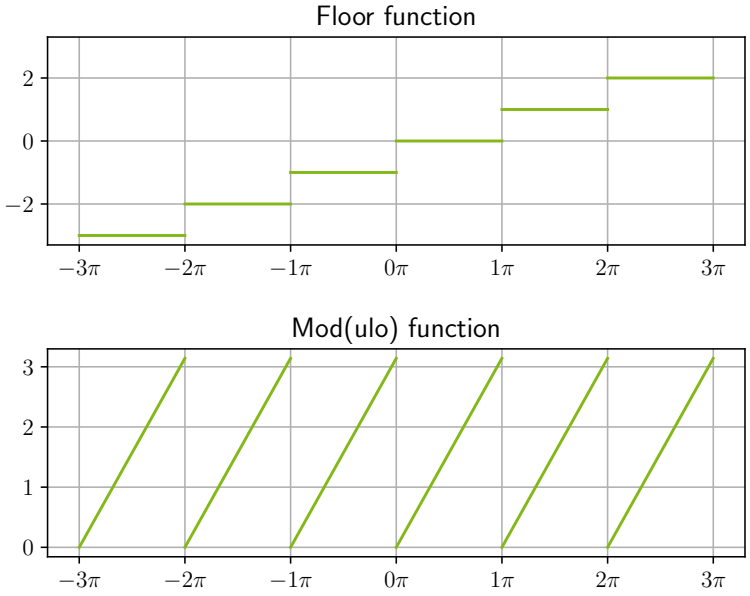

Figure 1.1: Visualisation of $\lfloor \frac{x}{\pi} \rfloor$ and $x \bmod \pi$.

In 1.1, we see that the slopes/derivatives of the two functions are constant and separated at the discontinuities at integer multiples of π.
The derivative of $\lfloor \frac{x}{\pi} \rfloor$ is 0, while the derivative of $x \bmod \pi$ is 1, for $x \neq n\pi$, $n \in \mathbb{Z}$.

Thus we have

$$f_1'(x) = \cos(x)$$
$$f_2'(x) = \begin{cases} \sin(x \bmod \pi)) & \text{for } x \geq 0 \text{ and } x \neq n\pi, n \in \mathbb{N}_0 \\ -\sin(x \bmod \pi)) & \text{for } x < 0 \text{ and } x \neq -n\pi, n \in \mathbb{N} \end{cases}. \tag{1.1}$$

Suitable point: x = $\pi/4$. This point is fixed for all sub-tasks which require a "suitable point choice".

a) For the first derivative using the two-point method, we observe a converging behavior when the step size h is smaller than 1. However, for extremely small values of h, the numerical calculation becomes inaccurate due to the limited precision of the floating-point representation. This phenomenon can be seen in Figure 1.2.

Recalling the previous discussion on Sheet 00, we know that subtraction of similar numbers can be unstable, even for values that are relatively large compared to the overall possible resolution of floating-point numbers. Consequently, it is reasonable to assume that the two-point method, which relies on the subtraction of very similar numbers, is susceptible to instability.

A suitable step size for the interval was chosen to be $h = 0.01$, which is sufficiently small to accurately detect changes in the function while also being large enough to avoid inaccuracies stemming from cancellation.

Note the comparison of relative errors to absolute errors at the end.

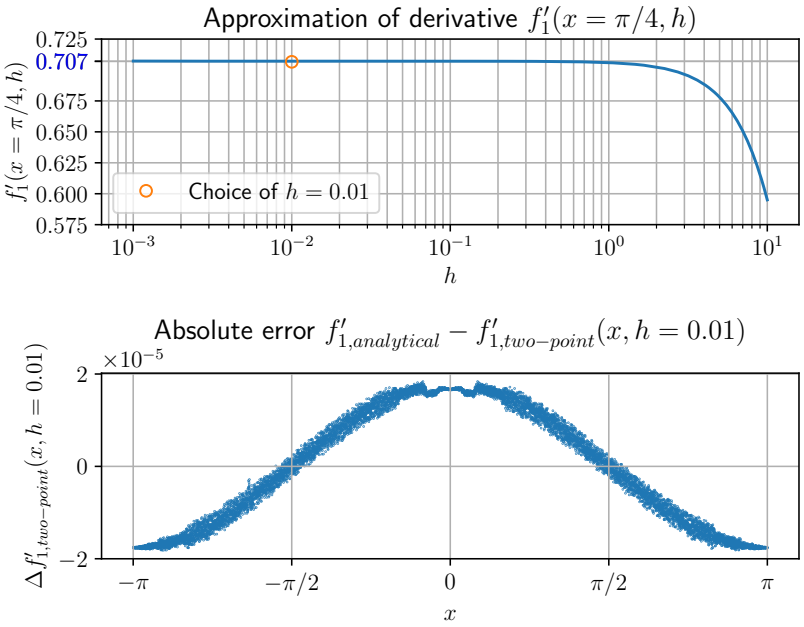

Figure 1.2: a) Approximation of the derivative of f_1 by finding a suitable interval h. The second plot shows the absolute error regarding the analytical solution of the derivative.

Absolute error

Overall, the absolute error is remarkably small, with an order of magnitude of approximately 10^{-5}. Moreover, the error exhibits an oscillatory pattern with a 2π-periodicity, resembling a $\cos x$ function. This insight could potentially be employed to regularize the error, thereby further reducing the absolute error.

It is important to note that the error does not perfectly fit a $\cos x$ function, but rather appears to be somewhat smeared out. This discrepancy is likely attributable to the limited precision of the floating-point representation.

b) The numerical calculation of the second derivative of f_1 using a two-fold application of the two-point method follows a process analogous to that in a).

First, we identify a suitable step size h, which can be chosen identically to $h = 0.01$ from a). We then compare the numerical results with the analytical solution by subtraction.

The visualization in Figure 1.3 reveals a small, patternless absolute error. Notably, the error significantly diminishes in the vicinity of $\pm\pi$ and 0.

c) The implementation of the absolute error for the derivative of f_1' using the four-point method is, once again, analogous to a). In Figure 1.4, the absolute error of the (one-fold) four-point method is compared to the absolute error from a).

The error of the two-point method appears negligible compared to the four-point method, which exhibits an oscillatory pattern akin to a sine-wave.

d) The same methods are applied to calculate and compare the results for the function f_2.

We observe a remarkably high precision for the four-point method and a diverging absolute error for the two-point method. Similar to c), the error of the diverging error appears to oscillate, which likely stems from the nature of the function being approximated.

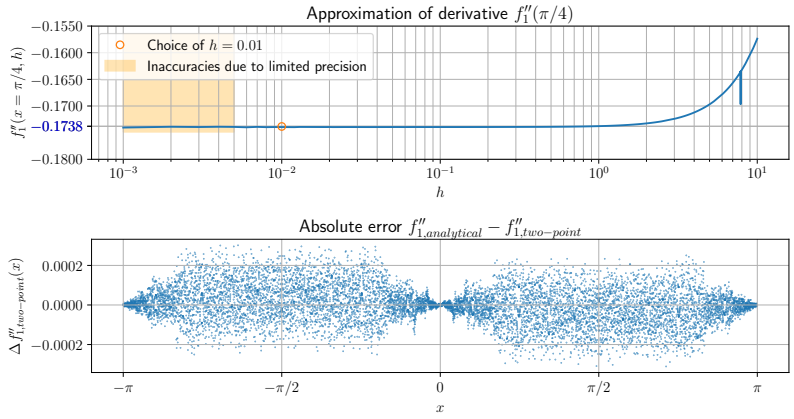

Figure 1.3: b) Approximation of the derivative of f_1' by finding a suitable interval h. The second plot shows the absolute error regarding the analytical solution of the derivative, analogous to a).

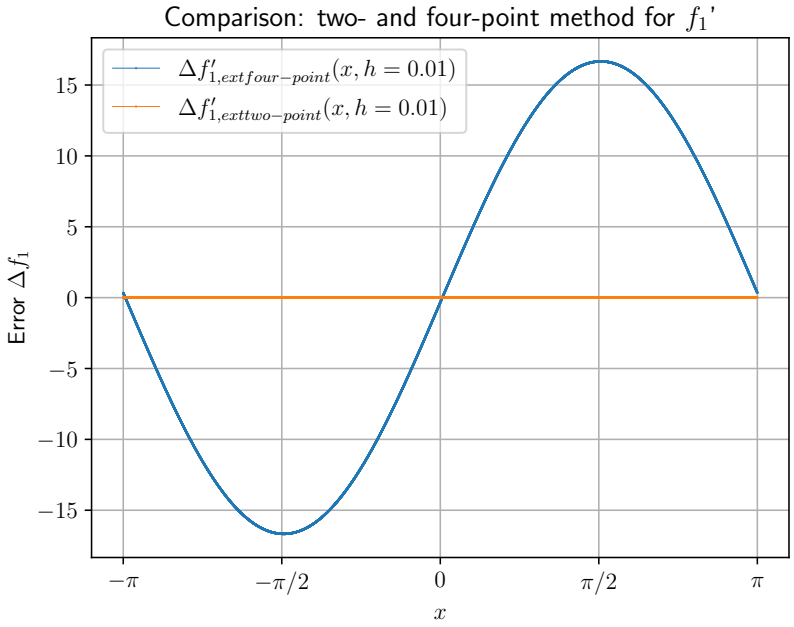

Figure 1.4: c) Comparison of the absolute errors of the derivative of f_1' using the four-point method and two-point method respectively.

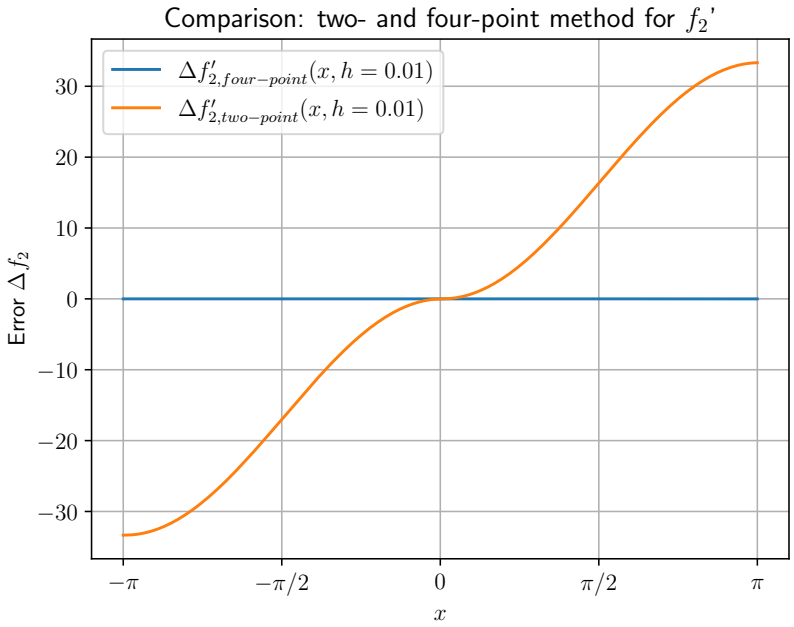

Figure 1.5: d) Comparison of the absolute errors of the derivative of f_2 using the four-point method and two-point method respectively.

Relative vs. absolute errors

Absolute errors are prone to inaccuracies due to cancellation, as similar values are likely to be subtracted from each other. To mitigate this issue, the introduction of a relative error might be beneficial.
We recalculated all four plots with relative errors, as seen in Figures 1.6, 1.7, 1.8, and 1.9. As anticipated, the errors exhibit significantly different behavior. The absolute error in a) was most pronounced at integer multiples of π (see Figure 1.2). With a relative error, this emphasis shifts to $\pm\pi/2$, while other values for x remain very stable. Figure 1.6 displays values predominantly close to 1. The anomalies are most evident in Figures 1.8 and 1.9.
The approximation for the second derivative in b) also transitions to a uniformly distributed relative error with no discernible patterns.
The relative error comparison of c) exhibits the same overall behavior, specifically the anomalous patterns at $\pm\pi/2$.
Finally, the relative errors in d) align with the behavior observed in c), albeit with different values of x for the anomalous phenomena. The four-point method displays spikes of up to 800% in relative error at integer multiples of π, while the two-point method exhibits even larger spikes (5000000%) at $\pm\pi$.
One possible explanation for these discrepancies is the loss of precision close to 0. This situation arises for both $f_1'(x) = \cos(x)$ (at $\pm\pi/2$) and $f_2(x)$ (at $n\pi$).

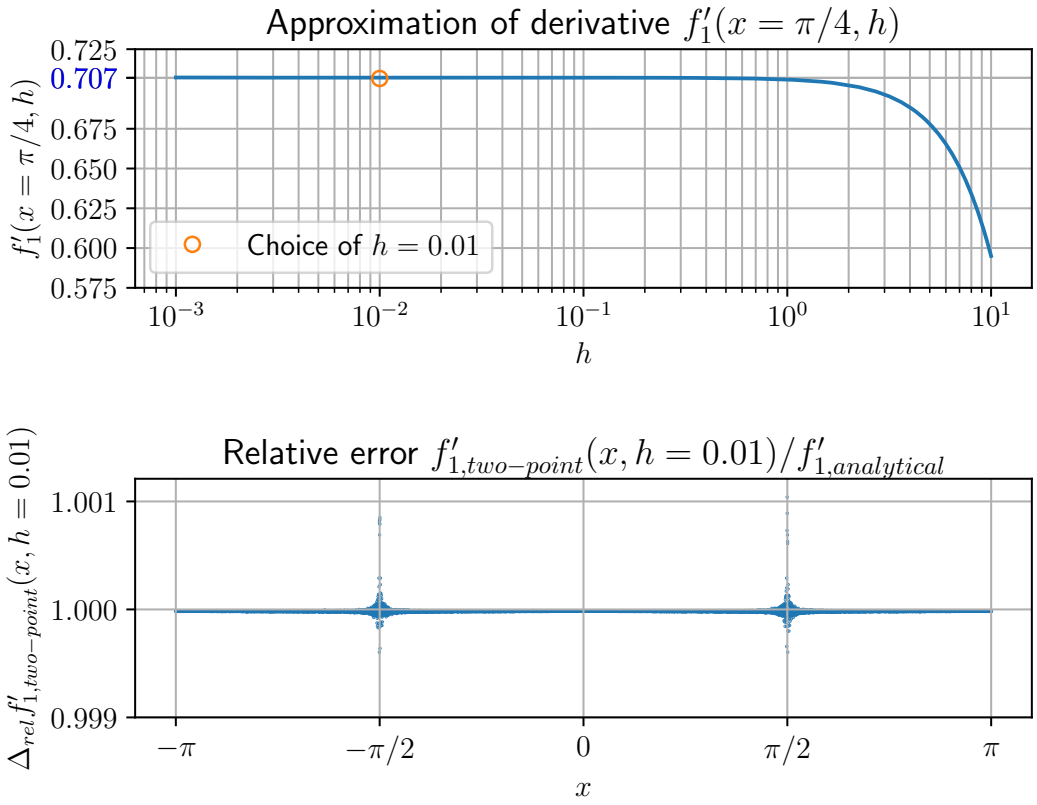

Figure 1.6: a) Relative error introduction for a).

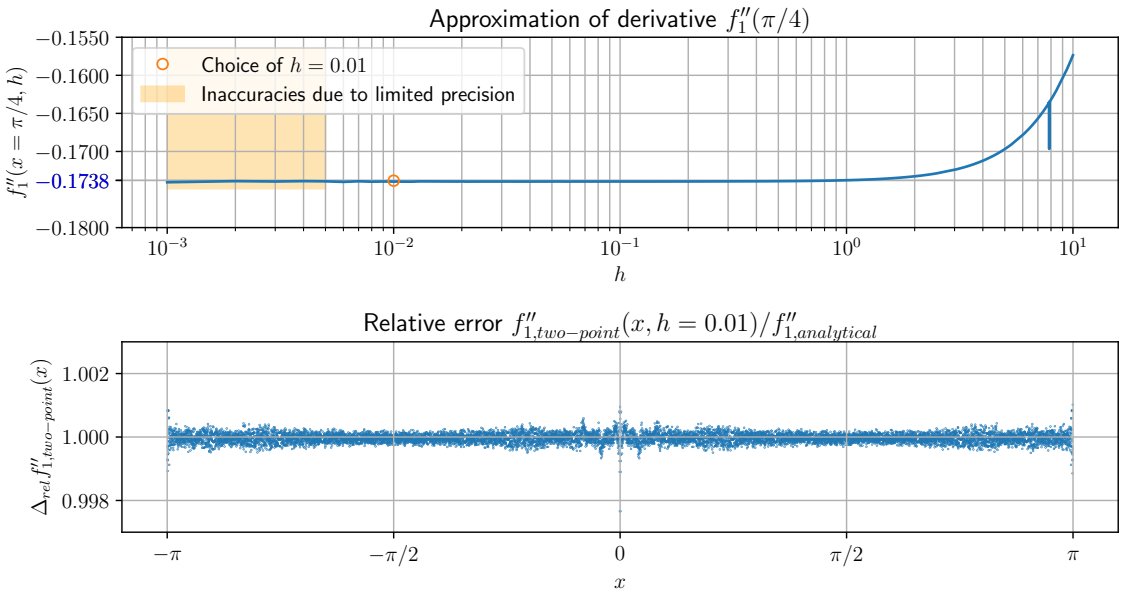

Figure 1.7: b) Relative error introduction for b).

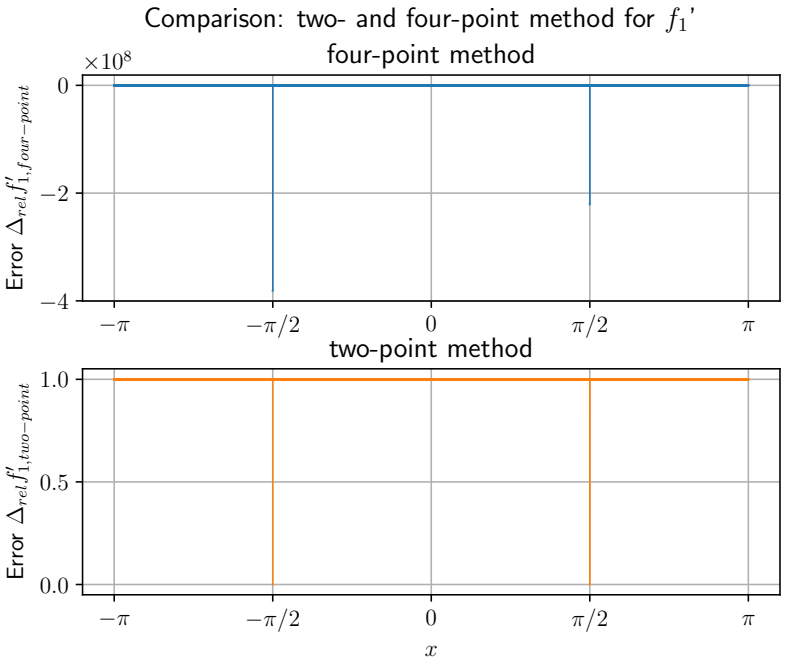

Figure 1.8: c) Relative error introduction for c).

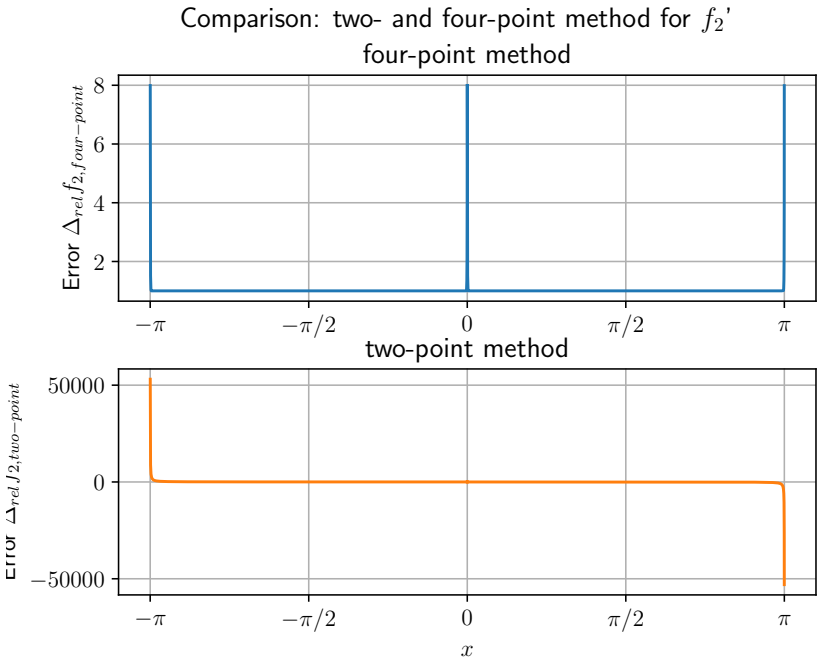

Figure 1.9: d) Relative error introduction for d).

Programming this Exercise is relatively straight-forward. For an easy assessment, an outline is provided.
outline.cpp

```cpp
#include <iostream>
#include <cmath>
#include <fstream>
#include <string>

using namespace std;

// Analytical function declarations for f1 and f2
float f_1(float x);
float df_1(float x);
float ddf_1(float x);
float f_2(float x);
float df_2(float x);

// Function declarations for numerical differentiation methods
float two_point(float (*func)(float), float x, float h);
float four_point(float (*func)(float), float x, float h);

// Function declarations for writing data to files
void write_to_file(float* values, int steps, string filename);
void write_to_file_err(float* values, int steps, string filename);

// Function wrappers for numerical differentiation methods
// The wrapper is necessary, because two_point cannot call itself
float two_point_wrapper_x(float x);
float two_point_wrapper_h(float h);
float four_point_wrapper_x(float x);

int main(int argc, char* argv[]) {

    // Flag to write files into txts
    bool writeFiles = false;
    if (argc > 1 && (string(argv[1]) == "--write" || string(argv[1]) == "-w"))
    writeFiles = true;

    // Defining constants and parameters
    float x_a = M_PIf/4;
    float h = 0.8f;
    int steps_a = 100000;
    int steps_b = 100000;
    int steps_c = 100000;
    int steps_d = 100000;

    // Allocate memory for arrays to store data and errors
    float *a = new float[steps_a];
    float *a_error = new float[steps_a];
    float *b = new float[steps_b];
    float *b_error = new float[steps_b];
    float *c_error = new float[steps_c];
    float *d_error_f1 = new float[steps_d];
    float *d_error_f2 = new float[steps_d];

    // a)
    // Two-point method for f_1, determining best interval h
    // Comparison to analytical solution
    for (int i = 0; i < steps_a; i++) a[i] = two_point(f_1, x_a, (float)i/steps_a);
```

```
    for (int i = 0; i < steps_a; i++) a_error[i] = df_1(((float)i*2*M_PIf/steps_a - M_PIf)) - two_point(f_1, ((float)i*2*M_PIf/steps_a - M_PIf), h);

    // b)
    // Two-point method for the first derivative of f_1, determining best interval h
    // Comparison to analytical solution
    for (int i = 0; i < steps_b; i++) b[i] = two_point(two_point_wrapper_h, x_a, (float)i/steps_b);
    for (int i = 0; i < steps_b; i++) b_error[i] = ddf_1(((float)i*2*M_PIf/steps_b - M_PIf)) - two_point(two_point_wrapper_x, ((float)i*2*M_PIf/steps_b - M_PIf), h);

    // c)
    // Four-point method for f_1
    // Comparison to analytical solution
    for (int i = 0; i < steps_c; i++) c_error[i] = df_1(((float)i*2*M_PIf/steps_c - M_PIf)) - four_point(f_1, ((float)i*2*M_PIf/steps_c - M_PIf), h);

    // d) Four-point method for f_2 and comparison with two-point method
    // Comparison to analytical solution
    for (int i = 0; i < steps_d; i++) d_error_f1[i] = df_2(((float)i*2*M_PIf/steps_d - M_PIf)) - four_point(f_2, ((float)i*2*M_PIf/steps_d - M_PIf), h);
    for (int i = 0; i < steps_d; i++) d_error_f2[i] = df_2(((float)i*2*M_PIf/steps_d - M_PIf)) - two_point(f_2, ((float)i*2*M_PIf/steps_d - M_PIf), h);

    // Write files if flag is set
    // Plots are created in python, matplotlib
    if (writeFiles) {
        write_to_file(a, steps_a, "a.txt");
        write_to_file_err(a_error, steps_a, "a_error.txt");
        write_to_file(b, steps_b, "b.txt");
        write_to_file_err(b_error, steps_b, "b_error.txt");
        write_to_file_err(c_error, steps_c, "c_error.txt");
        write_to_file_err(d_error_f1, steps_d, "d_error2_f2.txt");
        write_to_file_err(d_error_f2, steps_d, "d_error4_f2.txt");
    }

    // Free memory
    delete[] a;
    delete[] a_error;
    delete[] b;
    delete[] b_error;
    delete[] c_error;
    delete[] d_error_f1;
    delete[] d_error_f2;

    return 0;
}
```

The `makefile` is able to accept a flag to write the data to `.txt`-files for the use in the python file.

`make` - Compiles the code.
`make run` - Runs the code. (doesn't save the values)
`make run args="-w"` - Runs the code and saves the values to `.txt`-files.
`make run args="--write"` - Identical to `make run args="-w"`.

E2: Integration routines

Task:

Numerical integration is just as important as numerical differentiation in computational physics. Therefore this exercise's goal is to implement three basic integration routines as generally applicable functions.

Implement integration routines for

a) trapezoid rule,

b) Riemann sum,

c) Simpson's rule

with the following four arguments

1. integrand $f(x)$,
2. lower integration limit a,
3. upper integration limit b,
4. interval width h or number of integration intervals N (for Simpson's rule N should be even).

The implementation of this exercise is, once again, relatively straight-forward.
Considering the potential for computationally intensive calculations, a separate implementation in `C++` could offer performance advantages. However, it should be noted that the current Python implementation employs the efficient `NumPy` library for its calculations.
In Exercise 3, the `E2.py` file is effectively reused by importing it as a separate module, showcasing a modular design that promotes code reusability and maintainability.

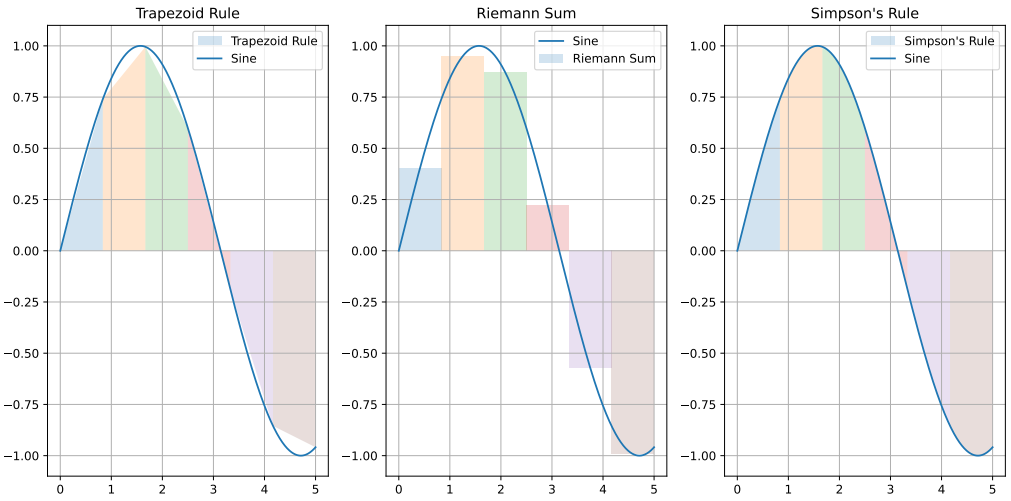

Figure 1.10: Trapezoid, Simpsons's rule and Riemann sum applied to a sine wave.

E2.py

```python
import numpy as np

# Define the trapezoid rule for numerical integration
def trapezoid_rule(f, a, b, h):
    n = int((b - a) / h)             # Calculate the number of intervals
    x = np.linspace(a, b, n + 1)     # Create an array of equally spaced points between a and b
    y = f(x)                         # Evaluate the integrand at the x values
    # Calculate the trapezoid rule sum, including the contributions from the first and last points
    return h * (0.5 * y[0] + 0.5 * y[-1] + np.sum(y[1:-1]))

# Define the Riemann sum for numerical integration
def riemann_sum(f, a, b, h):
    n = int((b - a) / h)             # Calculate the number of intervals
    x = np.linspace(a, b, n + 1)     # Create an array of equally spaced points between a and b
    # Evaluate the integrand at the midpoints of each interval (x[:-1] + h / 2)
    y = f(x[:-1] + h / 2)
    # Calculate the Riemann sum by summing the function values and multiplying by the interval width h
    return h * np.sum(y)

# Define Simpson's rule for numerical integration
def simpsons_rule(f, a, b, N):
    if N % 2 != 0:
        raise ValueError("N must be even for Simpson's rule.")
    N = int
    h = (b - a) / N                  # Calculate the interval width
    x = np.linspace(a, b, N + 1)     # Create an array of equally spaced points between a and b
    y = f(x)                         # Evaluate the integrand at the x values
    # Calculate the Simpson's rule sum, including the contributions from the first and last points,
    # the odd-indexed points (multiplied by 4), and the even-indexed points (multiplied by 2)
    return h / 3 * (y[0] + y[-1] + 2 * np.sum(y[2:-1:2]) + 4 * np.sum(y[1:-1:2]))

# Example usage:
def example_function(x):
    return np.sin(x)

a = 0
b = np.pi
h = 0.01

print("Trapezoid rule:", trapezoid_rule(example_function, a, b, h))
print("Riemann sum:", riemann_sum(example_function, a, b, h))
print("Simpson's rule:", simpsons_rule(example_function, a, b, int((b - a) / h)))
```

This script uses of the powerful index notation of Python arrays. They provide high flexibility while retaining the readability of the code.
General syntax: `array[start:end(:step)]`.
Note that the element `end` is **not** included in the sliced array, while `start` **is**.

A functionality test is provided by approximating a sine function and printing the results.

E3: One-dimensional integrals

Task:

Now that we have implemented three different integrations routines, we can apply them to numerically more complex one-dimensional integrals. This exercise, therefore, focuses on the numerical effort needed to attain high accuracy.

Calculate the integrals

a)

$$I_1 = \int_1^{100} \frac{\exp(-x)}{x} \, \mathrm{d}x$$

(Result: $I_1 \simeq 0.219384$)

b)

$$I_2 = \int_0^1 x \sin\left(\frac{1}{x}\right) \mathrm{d}x$$

(Result: $I_2 \simeq 0.378530$)

numerically with

1. trapezoid rule,
2. Riemann sum,
3. Simpson's rule.

Halve the interval width h until the relative change of the result becomes smaller than 10^{-4}.

In this implementation, we utilized the functions from Exercise 2 to solve the given problem. Two key challenges were encountered and addressed during the development of this code:

1. The computation of the second integrand resulted in a `division by zero` error. To circumvent this issue, a `boolean mask` was employed to compute the values only for non-zero elements. Although this technique is both common and effective, it may be more difficult for students who are unfamiliar with this approach to comprehend the code (ourselves included).
2. The `calculate_integral` function was adapted to accommodate all three integration methods. This proved to be particularly challenging for Simpson's rule, which requires an even number of intervals. A concise inline `if-statement` was implemented to differentiate cases where interval width is used as opposed to cases where interval count is employed.

E3.py

```python
import numpy as np
##################################################
# Importing E2.py as a module
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from E2.E2 import trapezoid_rule, riemann_sum, simpsons_rule
##################################################

# Define the integrands for I1 and I2
def integrand_I1(x):
    return np.exp(-x) / x

def integrand_I2(x):
    # return x * np.sin(1 / x)   This line would return a division by zero error
    # To mitigate this, we can use a boolean mask
    result = np.zeros_like(x)
    mask = x != 0
    result[mask] = x[mask] * np.sin(1 / x[mask])
    return result

# Initialize the parameters for the integrals
a1, b1 = 1, 100
a2, b2 = 0, 1
h_initial_1 = b1 - a1
h_initial_2 = b2 - a2

# Function to calculate the integral using different methods and halving the interval
    width
# Note that Simpson's rule has to be treated differently
def calculate_integral(integrand, a, b, initial_h, integration_method,
    target_relative_change=1e-4):
    h = initial_h if integration_method is not simpsons_rule else 2
    previous_result = integration_method(integrand, a, b, h)
    while True:
        h = h/2 if integration_method is not simpsons_rule else h*2
        current_result = integration_method(integrand, a, b, h)
        relative_change = np.abs((current_result - previous_result) / previous_result
    )
        if relative_change < target_relative_change:
            break
        previous_result = current_result
    return current_result

# Calculate the integrals I1 and I2 with different methods
for method_name, method in [("Trapezoid rule", trapezoid_rule), ("Riemann sum",
    riemann_sum), ("Simpson's rule", simpsons_rule)]:
    result_I1 = calculate_integral(integrand_I1, a1, b1, h_initial_1, method)
    result_I2 = calculate_integral(integrand_I2, a2, b2, h_initial_2, method)
    print(f"{method_name}:")
    print(f"I1     {result_I1:.6f}")
    print(f"I2     {result_I2:.6f}\n")
```

The `__init.py__` file located in the `E2` folder serves to identify and handle `E2.py` as a module.

E3.py approximates both integrals and generates following output:

```
Example function: sin(x) from 0 to pi:
Trapezoid rule: 1.998969410086143
Riemann sum: 1.9989944228122405
Simpson's rule: 2.000000000111338

Trapezoid rule:
I1      0.219386, at h = 0.006042
I2      0.378546, at h = 0.0004883

Riemann sum:
I1      0.219379, at h = 0.01208
I2      0.378532, at h = 0.0004883

Simpson's rule:
I1      0.219384, at N = 2048
I2      0.378537, at N = 4096
```

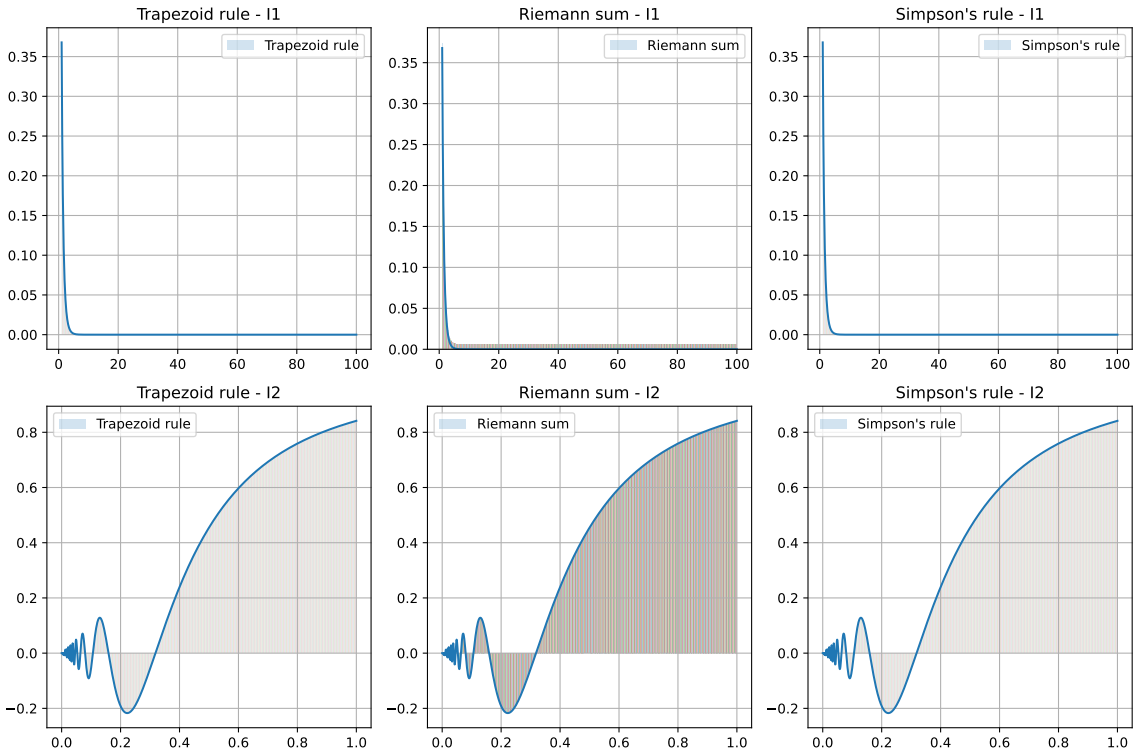

Figure 1.11: Comparison of the Trapezoid, Simpson's rule, and Riemann sum methods applied to both integrals from Exercise 3. Although the differences between the three methods are not overly apparent in this visualization, further examination of the numerical results reveals their distinctions. For a more detailed visualization illustrating the unique approaches of each method, refer to Figure 1.10 from Exercise 2.