Homework set 6

Before you turn this problem in, make sure everything runs as expected (in the menubar, select Kernel → Restart Kernel and Run All Cells...).

Please **submit this Jupyter notebook through Canvas** no later than **Mon Dec. 11, 9:00**. **Submit the notebook file with your answers (as .ipynb file) and a pdf printout. The pdf version can be used by the teachers to provide feedback. A pdf version can be made using the save and export option in the Jupyter Lab file menu.**

Homework is in **groups of two**, and you are expected to hand in original work. Work that is copied from another group will not be accepted.

Exercise 0

Write down the names + student ID of the people in your group.

Loes Bijman 15211312

Sacha Gijsbers 12798525

About imports

Please import the needed packages by yourself.

Exercise 1

N.B.1 tentative points for each part are: 2+1.5+2+2+1.5 (and one point for free gives 10).

N.B.2 you are to implement the methods yourself.

Given a function f, let $T(f, a, b, m)$ denote the composite trapezoid rule with m subintervals over the interval $[a, b]$.

(a)

Approximate the integral of x^{-3} over $[a, b] = [\frac{1}{10}, 100]$ by the composite trapezoid rule $T(f, a, b, m)$ for $m = 2^k$. Find the smallest k such that the exact error is less than $\epsilon = 10^{-3}$. Explain the slow convergence.

In [1]:
```
import scipy

def function(x):
    return x**(-3)

def trapezoid_rule(function, a, b, m):
    h = (b - a) / m
    result = 0.5 * (function(a) + function(b)) # start with beginning and endpoint
    for i in range(1, m):
        result += function(a + i * h) # add up the values of the function at each interval boundary
    return result * h

def integral(function, epsilon, a, b, k):
    exact_error = 1
    exact_integral = scipy.integrate.quad(function, a, b)[0]
    while exact_error >= epsilon:
        m = 2**k
        approx_integral = trapezoid_rule(function, a, b, m)
        exact_error = abs(exact_integral - approx_integral)
        k += 1
    return k, exact_error

epsilon = 1e-3
a = 1 / 10
b = 100
k = 1

k_integral, error = integral(function, epsilon, a, b, k)

print(f'The smallest k for which the exact error is less than epsilon = {epsilon} is k = {k_integral - 1}, with an error of {error}.')
```

```
The smallest k for which the exact error is less than epsilon = 0.001 is k = 18, with an error of 0.00036306889014525723.
```

The slow convergence could be caused by the behavior of the function x^{-3}, where the values grow rapidly towards infinity as x approaches zero. Therefore, achieving accuracy demands higher precision, requiring smaller subintervals (m) to accurately capture the function's behavior in this region. This leads to slower convergence, especially for values of x near zero.

(b)

To improve the convergence rate of the above problem, we may use an adaptive strategy, as discussed in the book and the lecture. Consider the following formulas for approximate integration

$$I_1(f, a, b) = T(f, a, b, 1)$$
$$I_2(f, a, b) = T(f, a, b, 2).$$

Show, based on the error estimates for the trapezoid rule using the Taylor series (book example 8.2) that the error in I_2 can be estimated by a formula of the form

$$E_2 = C(I_1 - I_2)$$

and determine the constant C (if you can't find C, you may take $C = 0.5$).

We start with the $I_1 - I_2$ term. Using the definition of the trapezoid rule, we find

$$\begin{aligned} I_1 - I_2 &= \frac{b-a}{2}\big(f(a)+f(b)\big) - \frac{b-a}{4}\Big[\big(f(a)+f(m)\big) + \big(f(b)+f(m)\big)\Big] && (1) \\ &= \frac{b-a}{4}\big(f(a)+f(b)\big) - \frac{b-a}{2}f(m) && (2) \end{aligned}$$

We now express $f(a)$ and $f(b)$ in terms of the midpoint ($m = \frac{a+b}{2}$) using the Taylor expansion around m.

We observe that $m + \frac{a-b}{2} = a$ and $m + \frac{b-a}{2} = b$. We find

$$\begin{aligned} f(a) &= f(m) + (\frac{a-b}{2})f'(m) + (\frac{a-b}{2})^2\frac{f''(m)}{2} + (\frac{a-b}{2})^3\frac{f'''(m)}{6} + \mathcal{O}((a-b)^4) && (3) \\ f(b) &= f(m) + (\frac{b-a}{2})f'(m) + (\frac{b-a}{2})^2\frac{f''(m)}{2} + (\frac{b-a}{2})^3\frac{f'''(m)}{6} + \mathcal{O}((b-a)^4) && (4) \end{aligned}$$

As $\left(\frac{a-b}{2}\right)^n + \left(\frac{b-a}{2}\right)^n = 0$ for odd n, we observe

$f(a) + f(b) = 2f(m) + \left(\frac{b-a}{2}\right)^2 f''(m) + \mathcal{O}((b-a)^4)$

Using this expression, we find

$$\begin{aligned} I_1 - I_2 &= \frac{b-a}{4}\Big[2f(m) + (\frac{b-a}{2})^2 f''(m) + \mathcal{O}((b-a)^4)\Big] - \frac{b-a}{2}f(m) && (5) \\ &= \frac{1}{32}\Big[(b-a)^3 f''(m) + \mathcal{O}((b-a)^5)\Big] && (6) \end{aligned}$$

Now we determine E_2, the error of I_2. We have $E_{am} + E_{mb}$, where m is the midpoint. We denote $c = \frac{3a+b}{4}$ as the middle point of am and $d = \frac{a+3b}{4}$ as the middle point of mb. From example 8.2 we know that

$$\begin{aligned} E_{am} &= 2 \cdot \frac{f''(c)}{12}(m-a)^3 + \mathcal{O}((m-a)^5) && (7) \\ E_{mb} &= 2 \cdot \frac{f''(d)}{12}(b-m)^3 + \mathcal{O}((b-m)^5) && (8) \end{aligned}$$

As $m - a = b - m = \frac{b-a}{2}$ we find

$E_2 = E_{am} + E_{mb} = \frac{1}{6}\left(\frac{b-a}{2}\right)^3\big(f''(c) + f''(d)\big) + \mathcal{O}((b-a)^5)$

We again use a Taylor expansion around m to find expressions for $f''(c)$ and $f''(d)$. Since $m + \frac{a-b}{4} = \frac{3a+b}{4} = c$ and $m + \frac{b-a}{4} = \frac{a+3b}{4} = d$, we can write

$$\begin{aligned} f''(c) &= f''(m) + (\frac{a-b}{4})f'''(m) + (\frac{a-b}{4})^2\frac{f^{(4)}(m)}{2} + \mathcal{O}((a-b)^3) && (9) \\ f''(d) &= f''(m) + (\frac{b-a}{4})f'''(m) + (\frac{b-a}{4})^2\frac{f^{(4)}(m)}{2} + \mathcal{O}((b-a)^3) && (10) \end{aligned}$$

We thus obtain

$f''(c) + f''(d) = 2f''(m) + \left(\frac{b-a}{4}\right)^2 f^{(4)}(m) + \mathcal{O}((b-a)^4)$

Substituting this expression in E_2 then gives

$$\begin{aligned} E_2 &= \frac{1}{6}(\frac{b-a}{2})^3\Big[2f''(m) + (\frac{b-a}{4})^2 f^{(4)}(m) + \mathcal{O}((b-a)^4)\Big] + \mathcal{O}((b-a)^5) && (11) \\ &= \frac{1}{24}\Big[(b-a)^3 f''(m) + \mathcal{O}((b-a)^5)\Big] && (12) \end{aligned}$$

Therefore, we have $E_2 = C(I_1 - I_2)$ for $C = \frac{1}{3}$.

(c)

An adaptive strategy for computing the integral on an interval $[a, b]$ now is: Compute I_2 and E_2, and accept I_2 as an approximation when the estimated error E_2 is less or equal than a desired tolerance ϵ. Otherwise, apply the procedure to $\int_a^{\frac{b+a}{2}} f(x)\,dx$ and $\int_{\frac{b+a}{2}}^b f(x)\,dx$ with tolerances $\frac{\epsilon}{2}$.

Write a recursive python routine that implements the adaptive strategy.

Then apply this routine to the function x^{-3} with a, b, ϵ as before. What is the exact error in the obtained approximation?

In [2]:
```
def function(x):
    return x**(-3)

def trapezoid_rule(function, a, b, m):
    h = (b - a) / m
    result = 0.5 * (function(a) + function(b))
    for i in range(1, m):
        result += function(a + i * h)
    return result * h

def adaptive_integral(function, epsilon, a, b):
    I1 = trapezoid_rule(function, a, b, 1)
    I2 = trapezoid_rule(function, a, b, 2)
    E2 = 1/3 * (I1 - I2)

    if E2 > epsilon:
        midpoint = (a + b) / 2
        left_integral = adaptive_integral(function, epsilon / 2, a, midpoint)
        right_integral = adaptive_integral(function, epsilon / 2, midpoint, b)
        return left_integral + right_integral

    return I2

epsilon = 1e-3
a = 1 / 10
b = 100

approx = adaptive_integral(function, epsilon, a, b)
exact_integral = scipy.integrate.quad(function, a, b)[0]
error = abs(exact_integral - approx)
print(f'The obtained approximation using adaptive strategy is {approx}, with an estimated error of {error}.')
```

```
The obtained approximation using adaptive strategy is 50.00014849011892, with an estimated error of 0.00019849011894024216.
```

(d)

Modify the code of (c) so that the number of function evaluations is counted and that no unnecessary function evaluations are performed. Compare the number of function evaluations used in the adaptive strategy of (c) with the result of (a). (*Hint*: To count the number of function evaluations, you may use a global variable that is incremented by the function each time it is called.)

In [3]:
```
import numpy as np

def function(x):
    global function_evals
    function_evals += 1
    return x**(-3)

def adaptive_integral(function, epsilon, a, b):
    global function_evals
    function_evals += 1
    I1 = trapezoid_rule(function, a, b, 1)
    I2 = trapezoid_rule(function, a, b, 2)
    E2 = 0.5 * (I1 - I2)

    if E2 > epsilon:
        midpoint = (a + b) / 2
        left_integral, left_error = adaptive_integral(function, epsilon / 2, a, midpoint)
        right_integral, right_error = adaptive_integral(function, epsilon / 2, midpoint, b)
        return left_integral + right_integral, left_error + right_error

    return I2, E2

epsilon = 1e-3
a = 1 / 10
b = 100

function_evals = 0
k_integral, a_error = integral(function, epsilon, a, b, k)
function_evals_a = function_evals
print(f'Number of function evaluations in unadapted strategy (a): {function_evals_a}')

function_evals = 0
c_approx, c_error = adaptive_integral(function, epsilon, a, b)
function_evals_c = function_evals
print(f'Number of function evaluations in adaptive strategy (c): {function_evals_c}')

print(f'The strategy in (a) uses {np.round(function_evals_a/function_evals_c, 2)} more evaluations then the adaptive strategy from (c)')
```

```
Number of function evaluations in unadapted strategy (a): 524703
Number of function evaluations in adaptive strategy (c): 71394
The strategy in (a) uses 7.35 more evaluations then the adaptive strategy from (c)
```

The adaptive strategy from (c) uses Approximately 7.35 times less function evaluations than (a). Less evaluations means that the strategy uses less computing power.

(e)

In the course of executing the recursive procedure, some subintervals are refined (split in two subintervals) while others aren't as a result of the choices made by the algorithm. It turns out that the choices made by this algorithm are not always optimal. Other algorithms, that decide in a different way which subinterval needs to be refined, may be more efficient in the sense that they require less function evaluations (while using the same formulas for the approximate integral and the approximate error associated with a subinterval).

Can you explain why this is the case? Discuss briefly possible alternative approaches.

A reason for suboptimal choices could be that the adaptive algorithm might not adequately consider the local behavior of the function within each subinterval. This could cause the algorithm to split intervals that do not necessarily need further subdivision or neglect areas that require more attention.

Alternative approaches could aim to optimize the selection of subintervals by considering local properties of the function, refining areas that need more attention, and reducing unnecessary function evaluations to enhance the efficiency of adaptive quadrature methods. Here the integrand will be sampled densely where it is difficult to integrate and sparsely where it is easy.

In [ ]: