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.

Zijian Zhang, 14851598

Lina Xiang, 14764369

About imports

Please import the needed packages by yourself.

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

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 [ ]:
```python
def T(f, a, b, m):
    h = (b - a) / m  # length of subintervals
    xs = [a + i * h for i in range(1, m)]  # inner points
    Tf = sum(f(x) for x in xs) + f(a) / 2 + f(b) / 2
    return Tf * h

# Set function, interval, and error tolerance
f = lambda x: x**-3
a, b = 1 / 10, 100.
epsilon = 1e-3

# Compute exact value
int_f = lambda x: -1 / (2 * x**2)
exact_integral = int_f(b) - int_f(a)

# Find smallest k such that |error| < epsilon
for k in range(100):
    m = 2**k
    exact_error = np.abs(exact_integral - T(f, a, b, m))
    if exact_error < epsilon:
        print(f'The smallest k such that the exact error is less than 0.0
        print(f'When k = {k}, exact error = {exact_error}')
        break
```

```
The smallest k such that the exact error is less than 0.001 is: 18
When k = 18, exact error = 0.00036306889090553796
```

Answer:

The slow convergence of the composite trapezoid rule in approximating the integral of $f(x) = x^{-3}$ over the interval $[\frac{1}{10}, 100]$ is due to the function's nature and the trapezoid rule's characteristics. The function $f(x) = x^{-3}$ has a singularity at $x = 0$. Although this point is outside the integration interval, it is close to the lower limit $a = \frac{1}{10}$. Near this singularity, the function changes rapidly, challenging any numerical integration method relying on discrete sampling points. The trapezoid rule, which approximates the area under a curve by dividing the interval into subintervals with linear segments, struggles with functions exhibiting significant curvature or rapid changes over small intervals. Moreover, the trapezoid rule's error is proportional to the function's second derivative, which for $f(x) = x^{-3}$ becomes large as x approaches zero. This leads to a substantial error near the lower limit, contributing significantly to the overall error. Increasing the number of subintervals (m) uniformly doesn't efficiently reduce the error, as only additional subintervals near the lower limit significantly impact it, resulting in slow convergence.

(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

$$\begin{aligned} I_1(f, a, b) &= T(f, a, b, 1) \\ I_2(f, a, b) &= T(f, a, b, 2). \end{aligned}$$

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 $C = 0.5$).

Answer:

Based on the derivation in the textbook, we have known that:

$$\begin{aligned} I(f) = \int_a^b f(x)\mathrm{d}x &= f(c)(b-a) + \frac{f''(c)}{24}(b-a)^3 + \cdots \\ &= M(f) + E(f) + \cdots, \end{aligned} \tag{1}$$

and

$$I_1 = M(f) + 3E(f) + \cdots, \tag{2}$$

where $c = \frac{a+b}{2}$ represents the midpoint of a and b, $M(f)$ represents the midpoint rule, and $E(f)$ represents the first term in the error expansion for the midpoint rule.

Now we perform a Taylor expansion of the function $f(x)$ at the point c and substitute $x = a$ and $x = b$ into the Taylor series:

$$f(a) = f(c) + f'(c)(a-c) + \frac{f''(c)}{2}(a-c)^2 + \cdots \tag{3}$$

$$f(b) = f(c) + f'(c)(b-c) + \frac{f''(c)}{2}(b-c)^2 + \cdots. \tag{4}$$

Substitute equations (3) and (4) into the expression for I_2:

$$I_2 = \frac{f(a) + 2f(c) + f(b)}{4}(b-a), \tag{5}$$

and get:

$$\begin{aligned} I_2 &= f(c)(b-a) + \frac{f''(c)}{16}(b-a)^3 + \cdots \\ &= M(f) + \frac{3}{2}E(f) + \cdots. \end{aligned} \tag{6}$$

Joining equation (1) with (2) and (6), we get:

$$E_2 = I(f) - I_2 = -\frac{1}{2}E(f) + \cdots, \tag{7}$$

and

$$I_1 - I_2 = \frac{3}{2}E(f) + \cdots. \tag{8}$$

So we have:

$$C = \frac{E_2}{I_1 - I_2} \approx -\frac{1}{3}. \tag{9}$$

(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 [ ]:
```
def adaptive_trapezoid(f, a, b, epsilon):
    I1 = T(f, a, b, 1)
    I2 = T(f, a, b, 2)
    E2 = (I2 - I1) / 3  # C = -1/3

    if np.abs(E2) <= epsilon:
        return I2
    else:
        mid = (a + b) / 2
        left_integral = adaptive_trapezoid(f, a, mid, epsilon / 2)
        right_integral = adaptive_trapezoid(f, mid, b, epsilon / 2)
        return left_integral + right_integral


# Compute the integral using the adaptive strategy
approx_integral = adaptive_trapezoid(f, a, b, epsilon)

# Calculate the exact error
exact_error = abs(exact_integral - approx_integral)
print(f'Approximate integral: {approx_integral}')
print(f'Exact integral: {exact_integral}')
print(f'Exact error: {exact_error}')
```

```
Approximate integral: 50.00014849011892
Exact integral: 49.99994999999999
Exact error: 0.00019849011189260313
```

(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 [ ]: # Global variable to count function evaluations
        function_evaluations = 0

        def new_f(x):
            global function_evaluations
            function_evaluations += 1
            return x**-3

        def new_adaptive_trapezoid(f, a, b, fa, fb, epsilon):
            # Midpoint and function evaluation at midpoint
            mid = (a + b) / 2
            fmid = new_f(mid)

            # Trapezoid rule for one and two intervals
            I1 = (b - a) * (fa + fb) / 2
            I2 = (b - a) * (fa + 2 * fmid + fb) / 4
            E2 = (I2 - I1) / 3  # C = -1/3

            if np.abs(E2) <= epsilon:
                return I2
            else:
                left_integral = new_adaptive_trapezoid(new_f, a, mid, fa, fmid, e
                right_integral = new_adaptive_trapezoid(new_f, mid, b, fmid, fb,
                return left_integral + right_integral

        # Initial function evaluations at the endpoints
        fa = new_f(a)
        fb = new_f(b)

        # Compute the integral using the adaptive strategy
        approx_integral = new_adaptive_trapezoid(new_f, a, b, fa, fb, epsilon)
        function_evaluations_c = function_evaluations

        # Count the number of evaluations in part (a)
        function_evaluations = 0
        theoretical_function_evaluations = 0
        for i in range(k+1):
            m = 2**i
            theoretical_function_evaluations += m + 1
            T(new_f, a, b, m)

        # Compare the number of evaluations in part (a) and (c)
        print(f'In part (a), the theoretical number of function evaluations is: {
        print(f'In part (a), the actual number of function evaluations is: {theor
        print(f'In part (c), the number of function evaluations is: {function_eva
```

```
In part (a), the theoretical number of function evaluations is: 524306
In part (a), the actual number of function evaluations is: 524306
In part (c), the number of function evaluations is: 9671
```

Answer:

In part (a), the theoretical number of function evaluations is $\sum_{k=0}^{18}(2^k + 1) = 524306$. Using the modified function, we observe that the actual count of function evaluations in part (a) is 524306, matching the theoretical prediction.

After modifying the code in part (c) to avoid unnecessary function evaluations, the number of evaluations in the adaptive strategy is 9671. This is significantly lower than the 524306 evaluations used in part (a), clearly demonstrating the efficiency of the adaptive strategy in accelerating convergence.

(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.

Answer:

Explanation:

Many functions display varying behaviors across their domain, including regions of rapid change and others with relative flatness. Different algorithms use different methods to decide which subinterval needs to be refined. The method currently in use determines whether further subdivision is required based on the local error estimates for each subinterval. If the estimated error of an interval exceeds a predetermined threshold, the interval is divided into smaller subintervals. More accurate error estimation algorithms can guide the subdivision of intervals more efficiently and reduce unnecessary calculations.

Possible alternative approaches:

- Heuristics: This method is based on knowledge of the behaviour of a particular type of function. For example, if it is known that the function exhibits extreme behaviour around certain specific points or regions (e.g. singularities, discontinuities), the algorithm can preferentially subdivide the subintervals closest to these regions.
- Derivative-based strategy: In this approach, the rate of change of a function is determined by calculating the derivative or gradient of the function over an interval. A large change in the derivative value within an interval means that the function is changing drastically within that interval, so a finer segmentation may be needed to capture this change.
- Error propagation analysis: By analysing how the overall integration error varies with subinterval division, a more optimal interval subdivision strategy can be developed. For example, if the subdivision of certain intervals contributes less to the overall error reduction, one may choose not to subdivide these intervals further.