ARC COMPUTATION AND INTELLIGENCE TASKS

ROUND2

TASK 1

Coding task 1

To solve this problem, we'll implement a numerical integration method known as the Trapezoidal Rule. This method is straightforward and gives a good approximation for definite integrals. I'll implement this in Python.

Trapezoidal Rule:

The idea is to approximate the area under the curve by dividing the interval [a,b][a, b][a,b] into small subintervals, calculate the area of trapezoids formed by the function over these subintervals, and sum these areas.

PROGRAM

```
import math

def trapezoidal_rule(f, a, b, n=10000):

    Numerically integrates the function f over the interval [a, b] using the Trapezoidal Rule.

    Parameters:

    f (function): The function to integrate.

    a (float): The lower limit of integration.

    b (float): The upper limit of integration.

    n (int): The number of subintervals to use (higher means better approximation).


    Returns:

    float: The approximate value of the integral.

    # Handle case where a or b is ±∞ using a substitution (for the bonus points)

    if math.isinf(a):

    a = -1e6  # substitute with a large negative number
```

```
        if math.isinf(b):
        b = 1e6  # substitute with a large positive number


        h = (b - a) / n  # width of each small interval
        result = 0.5 * (f(a) + f(b))  # first and last terms

        # Summing up areas of the trapezoids
        for i in range(1, n):
        result += f(a + i * h)

        result *= h  # multiply by the width of each trapezoid
        return result

# Example function and integral calculation
def f(x):
        if x == 0:
        return 1  # sin(x)/x tends to 1 as x approaches 0
        return math.sin(x) / x
       a = 0
       b = 1

integral_value = trapezoidal_rule(f, a, b)
print(f"The approximate value of the integral is: {integral_value:.4f}")
```

Explanation:

1. **Function Definition (trapezoidal_rule)**:
    - Takes the function f, the interval [a, b], and an optional parameter n for the number of subintervals.
    - It substitutes a and b with large negative and positive numbers if they are $-\infty$-$\infty$$-\infty$ or $+\infty$+∞+∞, respectively, to handle the bonus case.
2. **Implementation**:
    - The interval [a,b][a, b][a,b] is divided into n subintervals, and the width h of each subinterval is calculated.
    - The integral is approximated by summing the areas of the trapezoids formed over each subinterval.
    - The result is multiplied by h to get the final approximation of the integral.
3. **Edge Case Handling**:
    - When x = 0, sin(x)/x is mathematically defined as 1, so the function handles this by returning 1 when x = 0.

Output:

For the given function f(x)= sin(X)/x and interval [0.1], this implementation should return a value close to 0.9461, which is the known result of this integral.

This approach is simple but effective, and it can be refined further by increasing the number of subintervals n for a more accurate approximation.

TASK 3

Program

```
import cv2

import numpy as np

import matplotlib.pyplot as plt


# Step 1: Load the image as a grayscale image

image_path = ' image_path.jpeg'

# Replace with the path to your image

image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
```

```
# Step 2: Apply Gaussian Blur to the image

blurred_image = cv2.GaussianBlur(image, (5, 5), 0)


# Step 3: Calculate the Sobel kernel for x and y directions

sobel_x = cv2.Sobel(blurred_image, cv2.CV_64F, 1, 0, ksize=3)  # Sobel in x direction

sobel_y = cv2.Sobel(blurred_image, cv2.CV_64F, 0, 1, ksize=3)  # Sobel in y direction


# Step 4: Calculate the gradient magnitude

gradient_magnitude = cv2.magnitude(sobel_x, sobel_y)


# Normalise the result to range [0, 255] for display purposes

gradient_magnitude = cv2.normalize(gradient_magnitude, None, 0, 255,
cv2.NORM_MINMAX)


# Convert to uint8 type

gradient_magnitude = np.uint8(gradient_magnitude)


# Step 5: Display the gradient image

cv2.imshow('Gradient Magnitude', gradient_magnitude)

cv2.waitKey(0)

cv2.destroyAllWindows()


# Optionally save the result

cv2.imwrite('gradient_magnitude.jpg', gradient_magnitude)
```

Explanation:

1. **Loading the Image**:
    - The image is loaded in grayscale mode.

2. **Gaussian Blur**:
    - The image is blurred using a Gaussian filter of size 5×55 \times 55×5 to reduce noise and avoid false edge detection. Gaussian Blur works by averaging pixels with a Gaussian-weighted sum, which results in a smoothing effect on the image.
3. **Sobel Kernels**:
    - Sobel operators are used to detect gradients in the x and y directions. These gradients represent the rate of change in intensity at each pixel, which is crucial for edge detection.
4. **Gradient Calculation**:
    - The gradients from the x and y directions are combined using cv2.magnitude, which computes the Euclidean distance to get the overall gradient magnitude.
5. **Display/Save**:
    - The result is displayed using OpenCV's imshow function, and the gradient image can be saved as a file.

Gaussian Blur

Gaussian Blur is an image filtering technique used to smooth an image by reducing noise and detail. It works by averaging the pixel values in a region according to a Gaussian distribution, which gives more weight to the central pixels and less to the peripheral ones. This makes it particularly useful in preprocessing steps before edge detection.

APTITUDE TASKS

TASK 1

Bots must be assigned an algorithm on the basis of their linear distances from each point. We should minimise the distance travelled by all bots in order to assign a unique destination to each bot.

Indistinguishability : algorithm doesn’t rely on any specific labelling or order of designation of bots.

Decentralisation: no central control, so the decision must emerge from interaction amongst bots.

1) Distance matrix:

Each bot can be assigned their particular distance to each destination using the matrix.

$$\text{Distance Matrix} = \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{pmatrix}$$

2) Permutations :

Total ways of assigning destinations is 3!=6

3) Consensus on minimal distance:

Each of the 6 cases is evaluated, the case with least sum of distance is finalised.

Example:

Assume the distance matrix is:

$$\begin{pmatrix} 2 & 8 & 4 \\ 6 & 3 & 7 \\ 5 & 9 & 1 \end{pmatrix}$$

The possible assignments and their corresponding total distances are:

1. (1 → A, 2 → B, 3 → C): $2 + 3 + 1 = 6$
2. (1 → A, 2 → C, 3 → B): $2 + 7 + 9 = 18$
3. (1 → B, 2 → A, 3 → C): $8 + 6 + 1 = 15$
4. (1 → B, 2 → C, 3 → A): $8 + 7 + 5 = 20$
5. (1 → C, 2 → A, 3 → B): $4 + 6 + 9 = 19$
6. (1 → C, 2 → B, 3 → A): $4 + 3 + 5 = 12$

Minimum distance is 6,

Bot1=A destination

Bot2=B destination

Bot3=C destination

This approach ensures that bots have no control on the central controller , and assign destinations themselves.