MIDPOINT RULE CALCULATOR BY ABHIMANYU SHARMA AND SMAIRA PANDITA

Introduction

The Midpoint Rule Calculator is a Python program designed to approximate the definite integral of a given function using the midpoint rule numerical integration method. This calculator allows users to input a mathematical function in terms of 'x', specify the lower and upper limits of integration, and choose the number of subintervals to approximate the integral.

Functionality

The program consists of the following key functionalities:

- **Function Input**: Users can input a mathematical function in terms of 'x'. The program supports a wide range of mathematical expressions, including trigonometric functions, exponential functions, logarithmic functions, and more.
- **Plotting Function**: The program plots the function over the specified range to visualize the behavior of the function.
- **Midpoint Rule Calculation**: It computes the definite integral of the function using the midpoint rule method. This method divides the integration interval into subintervals and approximates the integral by summing the areas of rectangles with heights determined by the function evaluated at the midpoints of each subinterval.
- **Result Display**: The program displays the approximate value of the integral and the area under the curve.

Conclusion

The Midpoint Rule Calculator provides a versatile tool for approximating definite integrals numerically. By enabling users to input a wide range of mathematical functions, visualize the function's behavior, and compute the integral, this calculator facilitates problem-solving across various disciplines. Its applications extend to fields where precise integration is essential for analysis, modeling, and decision-making. Overall, the Midpoint Rule Calculator serves as a valuable resource for both students and professionals in diverse fields of study and application.

In [ ]:
```
pip install sympy
```

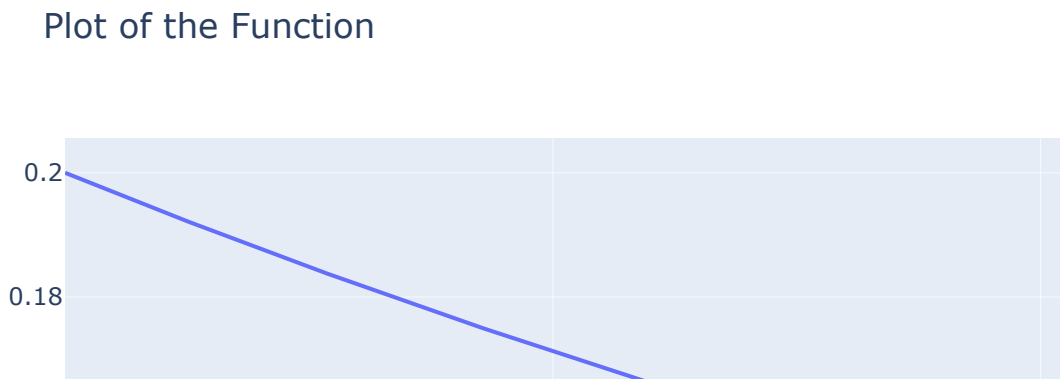
In [1]:
```
import sympy as sp
import numpy as np
import plotly.graph_objs as go


def midpoint_rule(f, a, b, n):
    """
    Compute the definite integral of a function using the midpoint rule.

    Parameters:
        f (function): The function to integrate (assumed to be a SymPy lambdify fur
        a (float): The lower limit of integration.
```

```
        b (float): The upper limit of integration.
        n (int): The number of subintervals.

    Returns:
        float: Approximation of the definite integral.
    """

    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)) or not is
        raise ValueError("Invalid input types. a and b must be numbers, and n must
    if n <= 0:
        raise ValueError("The number of subintervals (n) must be positive.")

    h = (b - a) / n
    integral_sum = 0

    for i in range(n):
        midpoint = a + (i + 0.5) * h
        integral_sum += f(midpoint) * h

    return integral_sum


def get_function_input():
    """
    Prompts the user for a mathematical expression, validates it, and returns
    a SymPy lambdify function.

    Returns:
        function: The user-defined function represented as a SymPy lambdify functic
        str: The original user-entered expression as a string.
    """

    x = sp.symbols('x')
    while True:
        try:
            expression = input("Enter the function you want to integrate in terms o
            # Handle potential division by zero
            if "1/0" in expression:
                print("Division by zero is not allowed. Please enter a valid expres
                continue
            f = sp.lambdify(x, expression, modules=[np, sp])
            return f, expression
        except (ValueError, TypeError, sp.SympifyError):
            print("Invalid input. Please enter a valid mathematical expression.")


def plot_function(f, a, b, expression):
    """
    Plots the function and displays the plot using Plotly.

    Parameters:
        f (function): The function to plot (assumed to be a SymPy lambdify functior
        a (float): The lower limit of the plot.
        b (float): The upper limit of the plot.
        str: The original user-entered expression as a string (for plot title).
    """

    x_vals = np.linspace(a, b, 1000)
    y_vals = f(x_vals)  # Fixed this line

    trace = go.Scatter(x=x_vals, y=y_vals, mode='lines', name=expression)
    layout = go.Layout(title='Plot of the Function', xaxis=dict(title='x'), yaxis=d
    fig = go.Figure(data=[trace], layout=layout)
    fig.show()
```

```
def get_float_input(prompt):
    """
    Prompts the user for a float value, validates it, and returns it.

    Parameters:
        str: The prompt to display to the user.

    Returns:
        float: The user-entered float value.
    """

    while True:
        try:
            value = float(input(prompt))
            return value
        except ValueError:
            print("Invalid input. Please enter a number.")


def get_int_input(prompt):
    """
    Prompts the user for an integer value, validates it, and returns it.

    Parameters:
        str: The prompt to display to the user.

    Returns:
        int: The user-entered integer value.
    """

    while True:
        try:
            value = int(input(prompt))
            return value
        except ValueError:
            print("Invalid input. Please enter an integer.")
def main():
    print("Welcome to the midpoint rule calculator!")
    print("This program calculates the definite integral of a function using the mi
    print("Please enter the following details:")

    f, expression = get_function_input()
    a = get_float_input("Lower limit of integration (a): ")

    while True:
        try:
            b = get_float_input("Upper limit of integration (b): ")
            if b <= a:
                print("Upper limit (b) must be greater than lower limit (a). Please
            else:
                break
        except ValueError:
            print("Invalid input. Please enter a number.")

    n = get_int_input("Number of subintervals (n): ")

    plot_function(f, a, b, expression)

    approx_integral = midpoint_rule(f, a, b, n)
    print("Approximated value of the integral:", approx_integral)

    area_under_curve = abs(approx_integral)
```

```
    print("Area under the curve:", area_under_curve)

if __name__ == "__main__":
    main()
```

```
Welcome to the midpoint rule calculator!
This program calculates the definite integral of a function using the midpoint rul
e.
Please enter the following details:
Enter the function you want to integrate in terms of 'x': 1 / (x**2 + 1)
Lower limit of integration (a): 2
Upper limit of integration (b): 3
Number of subintervals (n): 4
```




```
Approximated value of the integral: 0.1416374564301394
Area under the curve: 0.1416374564301394
```

FUNCTION EXAMPLES WE CAN INTEGRATE.

- `x**3 + 2*x**2 - 5*x + 7`
- `np.sin(x)`
- `np.cos(x)`
- `np.exp(x)`
- `np.log(x + 1)`
- `np.sqrt(x)`
- `x**4 - 3*x**3 + 2*x**2 + 5`
- `1 / (x**2 + 1)`

- `np.tan(x)`
- `np.arctan(x)`
- `np.sinh(x)`
- `np.cosh(x)`
- `np.tanh(x)`
- `np.arcsin(x)`
- `np.arccos(x)`
- `np.arcsinh(x)`
- `np.arccosh(x)`
- `np.arctanh(x)`
- `np.exp(x) - np.sin(x)`
- `np.exp(x) + np.cos(x)`
- `x**3 * np.exp(-x)`
- `x**2 * np.sin(x)`
- `x**3 * np.cos(x)`
- `x**2 * np.exp(-x)`
- `x * np.sin(x) - np.cos(x)`
- `x**2 * np.exp(x) - 2 * x * np.sin(x)`
- `np.sin(x) * np.cos(x)`
- `np.cos(x)**2`
- `np.sin(x)**2`
- `np.exp(x)**2`
- `np.sin(x)**3`
- `np.cos(x)**3`
- `x**3 * np.exp(x**2)`
- `x**2 * np.exp(x) * np.sin(x)`
- `x**2 * np.exp(x) * np.cos(x)`
- `np.sin(x) * np.cos(x) * np.exp(-x)`
- `x * np.sin(x)**2`
- `x * np.cos(x)**2`
- `np.sin(x)**2 * np.cos(x)**2`
- `np.exp(x) * np.sin(x)**2`
- `np.exp(x) * np.cos(x)**2`
- `x**2 * np.sin(x)**2`
- `x**2 * np.cos(x)**2`
- `x * np.sin(x)**3`
- `x * np.cos(x)**3`
- `np.exp(x) * np.sin(x)**3`
- `np.exp(x) * np.cos(x)**3`
- `x**2 * np.sin(x)**3`
- `x**2 * np.cos(x)**3`
- `x**3 * np.sin(x)**3`
- `x**3 * np.cos(x)**3`
- `x**2 * np.sin(x)**4`
- `x**2 * np.cos(x)**4`
- `x**3 * np.sin(x)**4`

- `x**3 * np.cos(x)**4`
- `np.exp(x) * np.sin(x)**4`
- `np.exp(x) * np.cos(x)**4`
- `x**4 * np.sin(x)**4`
- `x**4 * np.cos(x)**4`
- `x**4 * np.sin(x)**3`
- `x**4 * np.cos(x)**3`
- `x**4 * np.sin(x)**2`
- `x**4 * np.cos(x)**2`
- `np.sin(x)**5`
- `np.cos(x)**5`
- `np.sin(x)**6`
- `np.cos(x)**6`
- `x**2 * np.sin(x)**5`
- `x**2 * np.cos(x)**5`
- `x**3 * np.sin(x)**5`
- `x**3 * np.cos(x)**5`
- `x**4 * np.sin(x)**5`
- `x**4 * np.cos(x)**5`
- `x**5 * np.sin(x)**5`
- `x**5 * np.cos(x)**5`
- `x**6 * np.sin(x)**5`
- `x**6 * np.cos(x)**5`
- `np.exp(-x**2)`
- `np.exp(-x**2) * np.sin(x)`
- `np.exp(-x**2) * np.cos(x)`
- `x * np.exp(-x**2)`
- `x**2 * np.exp(-x**2)`
- `x**3 * np.exp(-x**2)`
- `x**4 * np.exp(-x**2)`
- `x**5 * np.exp(-x**2)`
- `x**6 * np.exp(-x**2)`
- `x**7 * np.exp(-x**2)`
- `x**8 * np.exp(-x**2)`
- `x**9 * np.exp(-x**2)`
- `x**10 * np.exp(-x**2)`
- `np.exp(-x) * np.sin(x)`
- `np.exp(-x) * np.cos(x)`
- `x * np.exp(-x)`
- `x**2 * np.exp(-x)`
- `x**3 * np.exp(-x)`
- `x**4 * np.exp(-x)`
- `x**5 * np.exp(-x)`
- `x**6 * np.exp(-x)`
- `x**7 * np.exp(-x)`
- `x**8 * np.exp(-x)`

COMPLEX FUNCTION EXAMPLES WE CAN INTEGRATE.

1. `x**3 + 2*x**2 - 5*x + 7`
2. `np.sin(x)**2 + np.cos(x)**2`
3. `np.tan(x)**2 + 1`
4. `np.sin(x)**3 + np.cos(x)**3`
5. `np.exp(np.sin(x)) * np.cos(x)`
6. `np.log(np.abs(x) + 1)`
7. `np.arcsin(np.sin(x))`
8. `np.arccos(np.cos(x))`
9. `np.arctan(np.tan(x))`
10. `np.sinh(x)**2 + np.cosh(x)**2`
11. `np.exp(x) * np.sin(x)**2 * np.cos(x)**2`
12. `np.exp(x) * np.sin(x)**3 * np.cos(x)**3`
13. `np.sin(x) / np.cos(x)`
14. `np.exp(np.tan(x))`
15. `np.exp(x) * np.arcsin(x)`
16. `np.exp(np.sqrt(x))`
17. `np.sqrt(np.log(x**2 + 1))`
18. `np.sin(x) * np.cos(x) * np.exp(-x**2)`
19. `np.exp(-x) * np.sinh(x)`
20. `np.exp(-x) * np.cosh(x)`
21. `np.exp(-x) * np.tanh(x)`
22. `np.sin(x) * np.exp(-np.cos(x))`
23. `np.exp(-np.sin(x)**2)`
24. `np.sin(x)**2 * np.cos(x)**2`
25. `np.sin(x) * np.exp(-x**2) * np.cos(x)`
26. `np.exp(x) * np.arctan(x)`
27. `np.exp(x) * np.sin(x) * np.cos(x)`
28. `np.sin(x)**3 * np.cos(x)**3`
29. `np.sin(x)**4 * np.cos(x)**4`
30. `np.exp(-x**2) * np.arcsin(x)`
31. `np.exp(-x**2) * np.sin(x)`
32. `np.exp(-x**2) * np.arccos(x)`
33. `np.exp(-x**2) * np.cos(x)`
34. `np.exp(-x**2) * np.arctan(x)`
35. `np.exp(-x**2) * np.tan(x)`
36. `np.exp(-x**2) * np.sinh(x)`
37. `np.exp(-x**2) * np.cosh(x)`
38. `np.exp(-x**2) * np.tanh(x)`
39. `np.sin(x) * np.arccosh(x)`
40. `np.sin(x) * np.arcsinh(x)`
41. `np.sin(x) * np.arctanh(x)`
42. `np.exp(x) * np.arccosh(x)`

```
43. np.exp(x) * np.arcsinh(x)
44. np.exp(x) * np.arctanh(x)
45. np.exp(x) * np.arcsin(x)
46. np.exp(x) * np.arccos(x)
47. np.exp(x) * np.arctan(x)
48. np.exp(x) * np.sinh(x)
49. np.exp(x) * np.cosh(x)
50. np.exp(x) * np.tanh(x)
```

In [ ]: