

**Discovering Knowledge**

**COURSE: GSL 321**

**NUMERICAL ANALYSIS**

**PROJECT REPORT**

**CLASS: BSE – 7A (FALL - 2023)**

**Interactive Visualization of Interpolation and Differentiation Methods**

**Group Members**

|  |  |
| --- | --- |
| **Student Name** | **Enrollment#** |
| Muhammad Abdullah | 02-131202-054 |
| Tahir Zaman | 02-131202-023 |
| Jawad Ahmed | 02-131202-036 |

**Submitted to:**

Course Instructor: Engr. Rahemeen

Lab Instructor: Engr. Noor us Sabah

**Department of Software Engineering**

**BAHRIA UNIVERSITY KARACHI CAMPUS**

**Abstract**

The project "Interactive Visualization of Interpolation and Differentiation Methods" addresses the educational challenge of comprehending mathematical concepts related to interpolation and differentiation. The purpose is to develop an interactive and user-friendly tool that allows users, including students and practitioners in mathematics and engineering, to gain a deeper understanding of these complex mathematical techniques.

The project encompasses a range of interpolation methods, including Newton’s Forward and Backward, Lagrange, and Newton’s Divide Difference, offering a diverse set of tools for users to explore and compare. Additionally, the project incorporates differentiation calculations using Newton’s Forward and Backward Differentiation formulas, providing a comprehensive suite of mathematical functionalities.

Implemented in Python, the project leverages the Matplotlib library for visualization, offering graphical representations that enhance the learning experience. Through a menu-based command-line interface, users can seamlessly navigate and select their desired method, input their data points, and witness the visualization of results, providing a dynamic and interactive learning environment.

The modular architecture of the project ensures scalability, allowing for the potential inclusion of additional interpolation and differentiation techniques in the future. This design choice facilitates an adaptable and extensible platform, accommodating the evolving needs of users and expanding the scope of mathematical exploration.

In conclusion, the project stands as an invaluable educational resource, catering to both beginners and enthusiasts seeking a hands-on approach to mastering interpolation and differentiation. Its user-friendly interface, diverse functionality, and visualizations contribute to a holistic and immersive learning experience in the realm of mathematical analysis.

TABLE OF Contents

[1. Introduction 4](#_Toc151811177)

[2. Problem Statement 4](#_Toc151811178)

[3. Proposed Solution 4](#_Toc151811179)

[**1.1 Features of the project 4**](#_Toc151811180)

[**1.2 Methodology 4**](#_Toc151811181)

[**1.3 Technologies 4**](#_Toc151811182)

[4. Project Scope 5](#_Toc151811183)

[5. Module Distribution 5](#_Toc151811184)

[6. Code 6](#_Toc151811185)

[7. Interfaces 16](#_Toc151811186)

[8. Conclusion 16](#_Toc151811187)

[9. References 16](#_Toc151811188)

# Introduction

The project titled "Visualization of Interpolation and Differentiation Methods" provides an interactive Python program for visualizing and understanding different interpolation and differentiation techniques. The implemented methods include Newton’s Forward Interpolation, Newton’s Backward Interpolation, Newton’s Divide Difference Formula, Lagrange Interpolation, Newton’s Forward Differentiation, and Newton’s Backward Differentiation.

# Problem Statement

Understanding interpolation and differentiation methods can be challenging for students and practitioners in mathematics and engineering. The project addresses the need for a practical and visual tool to aid in comprehending these mathematical concepts. The problem involves the lack of interactive tools that allow users to input their data and visualize the results of different interpolation and differentiation methods.

# Proposed Solution

## Features of the project

1. **Multiple Interpolation Methods:** The project supports various interpolation methods, including Newton’s Forward and Backward, Lagrange, and Newton’s Divide Difference.
2. **Differentiation Calculation:** Users can calculate derivatives using Newton’s Forward and Backward Differentiation formulas.
3. **Visualization:** Matplotlib is utilized for creating visualizations, providing users with graphical representations of the interpolated curves and tangent lines.
4. **User-Friendly Interface:** The project incorporates a user-friendly interface, guiding users through method selection and data input.

## Methodology

The project employs Python programming language and utilizes popular libraries such as NumPy for numerical computations and Matplotlib for interactive data visualization. The algorithms for interpolation and differentiation are implemented in a modular and extensible manner to accommodate future enhancements.

## Technologies

1. Programming Language: Python
2. Platform: Google Colab
3. Numerical Calculation Library: Numpy & Math
4. Visualization Library: Matplotlib

# Project Scope

The project scope encompasses educational, scientific, and practical applications. It serves as a learning tool for students studying numerical analysis while providing valuable insights for professionals and researchers working in diverse fields such as engineering, finance, and medical imaging.

# Module Distribution

The project consists of following modules:

**1. User Input Module:**

Facilitates user input for data points and method selection.

**2. Interpolation Modules:**

Implement various interpolation methods (Newton's Forward and Backward, Divided Differences, Lagrange).

**3. Differentiation Modules:**

Implement differentiation methods (Newton's Forward and Backward).

**4. Visualization Module:**

Utilizes Matplotlib for interactive visualization of interpolation and differentiation results.

*Each Project member is equally responsible for implementing the above stated modules.*

# Code

import matplotlib.pyplot as plt

import numpy as np

def main():

    print("Choose a method:")

    print("-------------------------------------")

    print("Interpolation for equal segments :")

    print("-------------------------------------")

    print("1. Newton’s Forward Interpolation")

    print("2. Newton’s Backward Interpolation")

    print("-------------------------------------")

    print("Interpolation for unequal segments :")

    print("-------------------------------------")

    print("3. Newton’s Divide Difference Formula")

    print("4. Lagrange Interpolation")

    print("-------------------------------------")

    print("Differentiation :")

    print("-------------------------------------")

    print("5. Newton’s Forward Differentiation")

    print("6. Newton’s Backward Differentiation")

    print("-------------------------------------")

    choice = int(input("Enter your choice (1-6): "))

    if choice not in range(1, 7):

        print("Invalid choice. Exiting.")

        return

    if choice == 1:

        def calculate\_u(u, n):

            temp = u

            for i in range(1, n):

                temp = temp \* (u - i)

            return temp

        def factorial(n):

            # Calculate the factorial of a given number n

            f = 1

            for i in range(2, n + 1):

                f \*= i

            return f

        def display\_forward\_difference\_table(x, y, n):

            # Display forward difference table for better visualization

            for i in range(n):

                print(x[i], end="\t")

                for j in range(n - i):

                    print(y[i][j], end="\t")

                print("")

        def interpolate(x, y, n, value):

            # Perform interpolation to estimate the value at a given point

            sum = y[0][0]

            u = (value - x[0]) / (x[1] - x[0])

            for i in range(1, n):

                sum = sum + (calculate\_u(u, i) \* y[0][i]) / factorial(i)

            return round(sum, 6)

        n = int(input("Enter the number of data points: "))

        x = []

        y = []

        # Input the data points

        for i in range(n):

            x\_val = float(input(f"Enter x[{i}]: "))

            y\_val = float(input(f"Enter y[{i}]: "))

            x.append(x\_val)

            y.append([0] \* n)

            y[i][0] = y\_val

        for i in range(1, n):

            for j in range(n - i):

                y[j][i] = y[j + 1][i - 1] - y[j][i - 1]

        print("\nForward Difference Table:")

        display\_forward\_difference\_table(x, y, n)

        value = float(input("\nEnter the value to interpolate at: "))

        result = interpolate(x, y, n, value)

        print(f"\nInterpolated value at {value} is approximately {result}")

        # Visualization

        plt.scatter(x, [y[i][0] for i in range(n)], label='Data Points')

        x\_values = np.linspace(min(x), max(x), 100)

        y\_values = [interpolate(x, y, n, val) for val in x\_values]

        plt.plot(x\_values, y\_values, label='Interpolation Curve')

        plt.scatter(value, result, color='red', marker='o', label=f'Target Value ({value}, {result:.6f})')

        plt.title("Newton’s Forward Interpolation")

        plt.xlabel("X")

        plt.ylabel("Y")

        plt.legend()

        plt.show()

    elif choice == 2:

        # Newton’s Backward Interpolation

        # Function to calculate 'u' as per the formula

        def calculate\_u(u, n):

            temp = u

            for i in range(1, n):

                temp = temp \* (u - i)

            return temp

        # Function to compute the factorial of 'n'

        def factorial(n):

        # Calculate the factorial of a given number n

            f = 1

            for i in range(2, n + 1):

                f \*= i

            return f

        # User input: Number of data points

        n = int(input("Enter the number of data points: "))

        # Initialize lists to store data

        x = []

        y = []

        # Input data points

        for i in range(n):

            x\_val = float(input(f"Enter x[{i}]: "))

            y\_val = float(input(f"Enter y[{i}]: "))

            x.append(x\_val)

            y.append([0] \* n)  # Initialize difference table

            y[i][0] = y\_val

        # Calculate the backward difference table

        for i in range(1, n):

            for j in range(n - 1, i - 1, -1):

                y[j][i] = y[j][i - 1] - y[j - 1][i - 1]

        # Display the backward difference table

        for i in range(n):

            for j in range(i + 1):

                print(y[i][j], end="\t")

            print()

    # User input: Value to interpolate at

        value = float(input("Enter the value to interpolate: "))

        # Initialize 'u' and 'result'

        result = y[n - 1][0]

        u = (value - x[n - 1]) / (x[1] - x[0])

        # Calculate the interpolated value using Newton's backward interpolation

        for i in range(1, n):

            term = calculate\_u(u, i) \* y[n - 1][i] / factorial(i)

            result += term

        # Display the interpolated value

        print(f"\nThe interpolated value at {value} is approximately {result}")

        # Visualize Newton's Backward Interpolation

        x\_points = np.array(x)

        y\_points = np.array([y\_val[0] for y\_val in y])

        x\_range = np.linspace(min(x\_points), max(x\_points), 1000)

        y\_range = np.zeros\_like(x\_range)

        for i in range(n):

            term = y[i][i]

            for j in range(i):

                term \*= (x\_range - x[j])

                term /= (x[i] - x[j])

            y\_range += term

        plt.plot(x\_points, y\_points, 'ro', label='Known Data Points')

        plt.plot(x\_range, y\_range, label="Newton's Backward Interpolation")

        plt.scatter(value, result, color='blue', marker='\*', label='Interpolated Point')

        plt.xlabel('X')

        plt.ylabel('Y')

        plt.title("Newton's Backward Interpolation")

        plt.legend()

        plt.grid(True)

        plt.show()

    elif choice == 3:

        def calculate\_product\_term(i, value, x):

            product = 1

            for j in range(i):

                product = product \* (float(value) - x[j])

            return product

        # Function for calculating the divided difference table

        def calculate\_divided\_difference\_table(x, y, n):

            for i in range(1, n):

                for j in range(n - i):

                    y[j][i] = ((y[j][i - 1] - y[j + 1][i - 1]) / (x[j] - x[i + j]))

            return y

        # Function for applying Newton's divided difference formula

        def apply\_newton\_formula(value, x, y, n):

            result = y[0][0]

            for i in range(1, n):

                result = result + (calculate\_product\_term(i, value, x) \* y[0][i])

            return result

        # Function for displaying the divided difference table

        def display\_divided\_difference\_table(y, n):

            for i in range(n):

                for j in range(n - i):

                    print(round(y[i][j], 4), "\t", end=" ")

                print("")

        n = int(input("Enter the number of inputs: "))

        x = []

        y = [[0 for i in range(n)] for j in range(n)]

        for i in range(n):

            x\_val = float(input(f"Enter x[{i}]: "))

            y\_val = float(input(f"Enter y[{i}]: "))

            x.append(x\_val)

            y[i][0] = y\_val

        # Calculate the divided difference table

        y = calculate\_divided\_difference\_table(x, y, n)

        # Display the divided difference table

        print("\nDivided Difference Table:")

        display\_divided\_difference\_table(y, n)

        # User input: Value to interpolate

        value = input("Enter Value to interpolate: ")

        # Calculate and display the interpolated value

        interpolated\_value = apply\_newton\_formula(value, x, y, n)

        print("\nInterpolated value at", value, "is approximately", round(interpolated\_value, 4))

        # Visualize the Newton's divided difference interpolation

        x\_range = np.linspace(min(x), max(x), 1000)

        y\_range = [apply\_newton\_formula(x\_i, x, y, n) for x\_i in x\_range]

        # Plot the interpolation curve

        plt.plot(x\_range, y\_range, label='Divided Difference Interpolation')

        # Plot the original data points

        plt.scatter(x, [y\_i[0] for y\_i in y], color='red', marker='o', label='Original Data Points')

        # Plot the interpolated point

        interpolated\_value = apply\_newton\_formula(float(value), x, y, n)

        plt.scatter(float(value), interpolated\_value, color='blue', marker='\*', label='Interpolated Point')

        plt.xlabel('X')

        plt.ylabel('Y')

        plt.title('Newton\'s Divided Difference Interpolation')

        plt.legend()

        plt.grid(True)

        plt.show()

    elif choice == 4:

        class DataPoint:

            def \_\_init\_\_(self, x, y):

                self.x = x

                self.y = y

        # Function for Lagrange Interpolation

        def lagrange\_interpolation(data\_points, x\_value):

            result = 0.0

            # Iterate through each data point

            for i in range(len(data\_points)):

                term = data\_points[i].y

                # Calculate the Lagrange term for the current data point

                for j in range(len(data\_points)):

                    if j != i:

                        term \*= (x\_value - data\_points[j].x) / (data\_points[i].x - data\_points[j].x)

                # Add the term to the result

                result += term

            return result

        if \_\_name\_\_ == "\_\_main\_\_":

            data\_points = []

            # Input the number of known data points

            n = int(input("Enter the number of known data points: "))

            # Input x and y values for each data point

            for i in range(n):

                x = float(input(f"Enter x{i + 1}: "))

                y = float(input(f"Enter y{i + 1}: "))

                data\_points.append(DataPoint(x, y))

            # Input the x value for interpolation

            x\_value = float(input("Enter the x value for interpolation: "))

            # Calculate and print the interpolated value

            interpolated\_value = lagrange\_interpolation(data\_points, x\_value)

            print(f"Interpolated value at {x\_value} is: {interpolated\_value}")

            # Visualize the Lagrange interpolation

            x\_points = [point.x for point in data\_points]

            y\_points = [point.y for point in data\_points]

            x\_range = np.linspace(min(x\_points), max(x\_points), 1000)

            y\_range = [lagrange\_interpolation(data\_points, x) for x in x\_range]

            plt.plot(x\_points, y\_points, 'ro', label='Known Data Points')

            plt.plot(x\_range, y\_range, label='Lagrange Interpolation')

            plt.scatter(x\_value, interpolated\_value, color='blue', marker='\*', label='Interpolated Point')

            plt.xlabel('X')

            plt.ylabel('Y')

            plt.title('Lagrange Interpolation')

            plt.legend()

            plt.grid(True)

            plt.show()

    elif choice == 5:

        import math

        def main():

            x = [0.0] \* 20

            y = [[0.0] \* 20 for \_ in range(20)]

            sum\_value = 0.0

            index = 0

            flag = 0

            sign = 1

            # Read the number of data points

            n = int(input("Enter the number of data points: "))

            # Read the actual data for x and y

            print("Enter data:")

            for i in range(n):

                x[i] = float(input(f"x[{i}] = "))

                y[i][0] = float(input(f"y[{i}] = "))

            # Read the calculation point

            xp = float(input("Enter the value of x where you want to calculate the derivative: "))

            # Check if the given point (xp) is a valid point in the x data

            for i in range(n):

                if abs(xp - x[i]) < 0.0001:

                    index = i

                    flag = 1

                    break

            # If the flag is still 0, (xp) is not in the list of x data

            if flag == 0:

                print("Invalid calculation point. Exiting program...")

                exit(0)

            # Generate the Forward Difference Table

            for i in range(1, n):

                for j in range(n - i):

                    y[j][i] = y[j + 1][i - 1] - y[j][i - 1]

            # Calculate the finite difference (step size)

            h = x[1] - x[0]

            # Apply the formula to calculate the sum of terms for derivatives

            for i in range(1, n - index):

                term = (y[index][i] \*\* i) / i

                sum\_value += sign \* term

                sign = -sign

            # Divide by h to get the first derivative

            first\_derivative = sum\_value / h

            # Display the final result

            print(f"The first derivative at x = {xp:.2f} is {first\_derivative:.2f}")

            # Visualize the data points and the tangent line at the calculation point

            x\_range = np.linspace(min(x), max(x), 1000)

            y\_range = np.zeros\_like(x\_range)

            for i in range(1, n - index):

                term = 0.0

                if i != 0:  # Skip division by zero when i is zero

                    term = (y[index][i] \*\* i) / i

                sum\_value += sign \* term

                sign = -sign

            tangent\_line = first\_derivative \* (x\_range - xp) + y[index][0]

            plt.plot(x\_range, y\_range, label='Original Curve')

            plt.plot(x\_range, tangent\_line, label='Tangent Line at x = {:.2f}'.format(xp), linestyle='dashed')

            plt.scatter(x, [y\_val[0] for y\_val in y], color='red', label='Data Points')

            plt.scatter(xp, y[index][0], color='blue', marker='\*', label='Calculation Point')

            plt.xlabel('X')

            plt.ylabel('Y')

            plt.title('First Derivative Calculation')

            plt.legend()

            plt.grid(True)

            plt.show()

        if \_\_name\_\_ == "\_\_main\_\_":

            main()

    elif choice == 6:

        def main():

            x = [0.0] \* 20

            y = [[0.0] \* 20 for \_ in range(20)]

            sum\_value = 0.0

            index = 0

            flag = 0

            # Read the number of data points

            n = int(input("Enter the number of data points: "))

            # Read the actual data for x and y

            print("Enter data:")

            for i in range(n):

                x[i] = float(input(f"x[{i}] = "))

                y[i][0] = float(input(f"y[{i}] = "))

            # Read the calculation point

            xp = float(input("Enter the value of x where you want to calculate the derivative: "))

            # Check if the given point (xp) is a valid point in the x data

            for i in range(n):

              if abs(xp - x[i]) < 0.0001:

                index = i

                flag = 1

                break

            # If the flag is still 0, the given point (xp) is not in the list of x data

            if flag == 0:

                print("Invalid calculation point. Exiting the program...")

                exit(0)

            # Generate the Backward Difference Table

            for i in range(1, n):

                for j in range(n - 1, i - 1, -1):

                    y[j][i] = y[j][i - 1] - y[j - 1][i - 1]

            # Calculate the finite difference (step size)

            h = x[1] - x[0]

            # Apply the formula to calculate the sum of terms for finding derivatives using the backward difference formula

            for i in range(1, index + 1):

                term = (y[index][i] \*\* i) / i

                sum\_value += term

            # Divide by h to get the first derivative

            first\_derivative = sum\_value / h

            # Display the final result

            print(f"The first derivative at x = {xp:.2f} is {first\_derivative:.2f}")

            # Visualize the data points and the tangent line at the calculation point

            x\_range = np.linspace(min(x), max(x), 1000)

            y\_range = np.zeros\_like(x\_range)

            sign = 1  # Initialize 'sign' before the loop

            for i in range(1, n - index):

                term = 0.0

                if i != 0:  # Skip division by zero when i is zero

                    term = (y[index][i] \*\* i) / i

                sum\_value += sign \* term

                sign = -sign

            tangent\_line = first\_derivative \* (x\_range - xp) + y[index][0]

            plt.plot(x\_range, y\_range, label='Original Curve')

            plt.plot(x\_range, tangent\_line, label='Tangent Line at x = {:.2f}'.format(xp), linestyle='dashed')

            plt.scatter(x, [y\_val[0] for y\_val in y], color='red', label='Data Points')

            plt.scatter(xp, y[index][0], color='blue', marker='\*', label='Calculation Point')

            plt.xlabel('X')

            plt.ylabel('Y')

            plt.title('First Derivative Calculation')

            plt.legend()

            plt.grid(True)

            plt.show()

        if \_\_name\_\_ == "\_\_main\_\_":

            main()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

# Interfaces

The project interfaces with users through a command-line interface, providing a menu for method selection and guiding users through the input process. The visualizations are presented using Matplotlib's plotting capabilities.

# Conclusion

The project successfully addresses the need for an interactive tool to visualize interpolation and differentiation methods. The modular approach ensures scalability, and the user-friendly interface enhances accessibility. The visualizations aid in the understanding of complex mathematical concepts.

# References

1. Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical Recipes: The Art of Scientific Computing. Cambridge University Press.
2. Burden, R. L., & Faires, J. D. (2010). Numerical Analysis. Cengage Learning.
3. Kincaid, D., & Cheney, W. (2009). Numerical Analysis: Mathematics of Scientific Computing. American Mathematical Society.