Tutorial 1

https://github.com/Blink29/Numerical-Techniques → Assignment 1

Name: Paurush Kumar Roll Number: NA22B002

Problem Statement

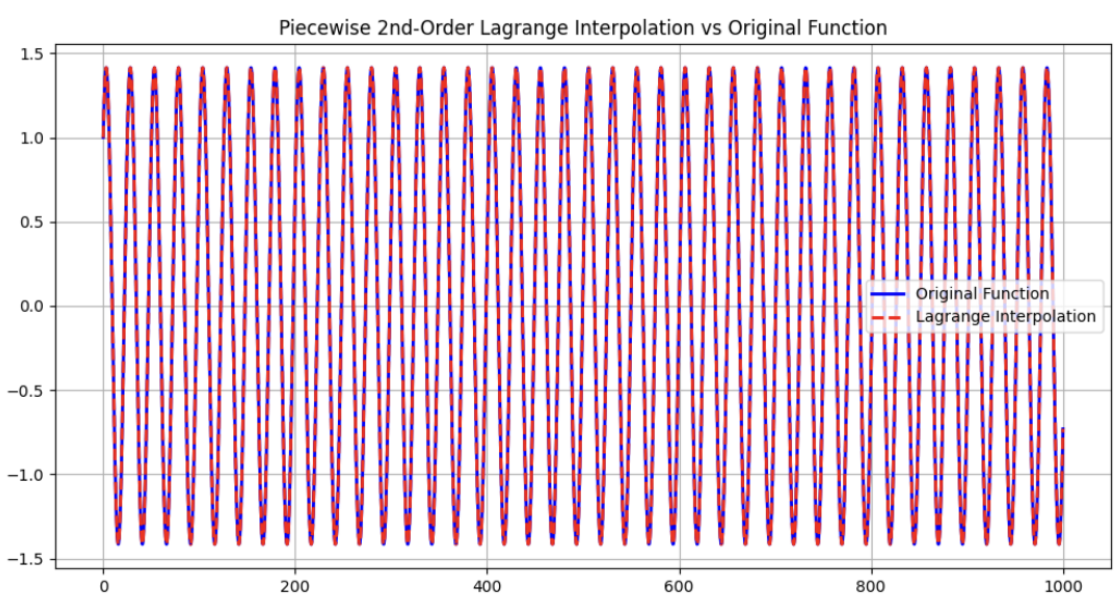
1. Construct cubic spline and 2nd-order Lagrange interpolations for a given function. Compare their interpolated values with the original function and compute the absolute error.

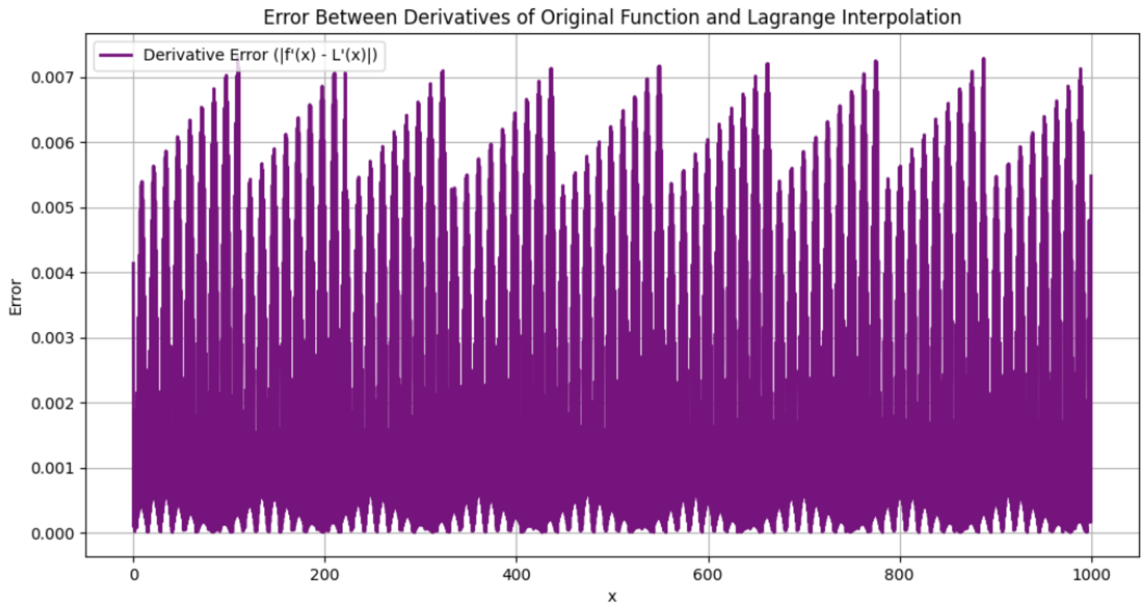
2. Differentiate the original function and both interpolations. Compare their derivatives and calculate the absolute error.

Function: $sin(x/4) + cos(x/4)$

Results

For Lagrangian






```
import numpy as np
import matplotlib.pyplot as plt
from numdifftools import Derivative

def f(x):
    return np.sin(x / 4) + np.cos(x / 4)

class PiecewiseLagrangeInterpolator:
    def __init__(self, func, num_points):
        self.func = func
        self.num_points = num_points
        self.x_vals = None
        self.y_vals = None

    def generate_points(self, x_min=0, x_max=1000):
        self.x_vals = np.linspace(x_min, x_max, self.num_poin
ts)
        self.y_vals = self.func(self.x_vals)

    def lagrange_2nd_order(self, x, x_window, y_window):
        term0 = y_window[0] * ((x - x_window[1]) * (x - x_win
```

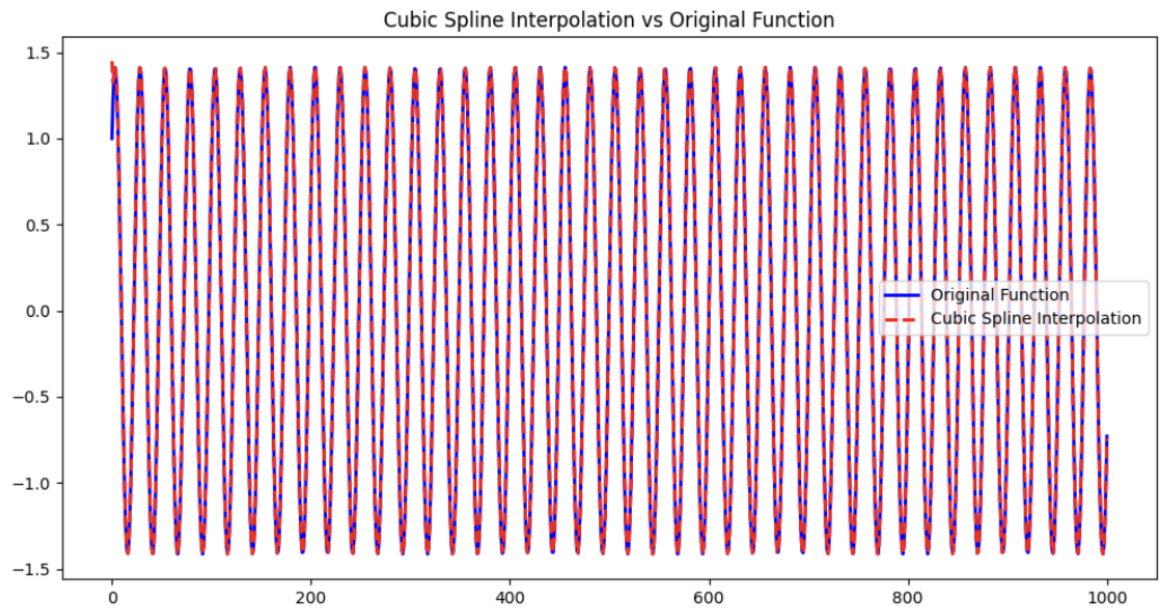
```
dow[2])) / ((x_window[0] - x_window[1]) * (x_window[0] - x_wi
ndow[2]))
        term1 = y_window[1] * ((x - x_window[0]) * (x - x_win
dow[2])) / ((x_window[1] - x_window[0]) * (x_window[1] - x_wi
ndow[2]))
        term2 = y_window[2] * ((x - x_window[0]) * (x - x_win
dow[1])) / ((x_window[2] - x_window[0]) * (x_window[2] - x_wi
ndow[1]))
        return term0 + term1 + term2

    def interpolate(self, x_dense):
        y_interp = np.zeros_like(x_dense)
        n = len(self.x_vals)

        for i in range(n - 2):  # Slide window of 3 points
            x_window = self.x_vals[i:i + 3]
            y_window = self.y_vals[i:i + 3]

            # Find points in the current segment to interpola
te
            mask = (x_dense >= x_window[0]) & (x_dense <= x_w
indow[-1])
            y_interp[mask] = [self.lagrange_2nd_order(x, x_wi
ndow, y_window) for x in x_dense[mask]]

        return y_interp

    def lagrange_derivative(self, x_dense):
        y_derivative = np.zeros_like(x_dense)
        n = len(self.x_vals)

        for i in range(n - 2):  # Slide window of 3 points
            x_window = self.x_vals[i:i + 3]
            y_window = self.y_vals[i:i + 3]

            # Derivative terms for 2nd-order Lagrange polynom
```

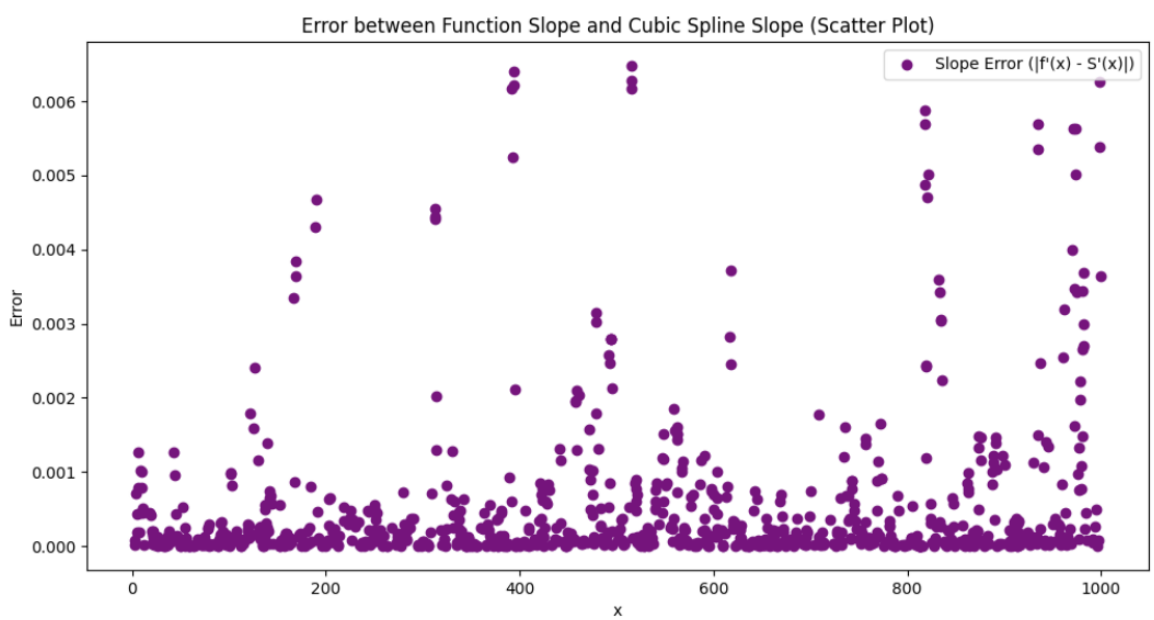
```
ial
            def derivative_lagrange(x):
                term0 = y_window[0] * ((x - x_window[1]) + (x
- x_window[2])) / ((x_window[0] - x_window[1]) * (x_window[0]
- x_window[2]))
                term1 = y_window[1] * ((x - x_window[0]) + (x
- x_window[2])) / ((x_window[1] - x_window[0]) * (x_window[1]
- x_window[2]))
                term2 = y_window[2] * ((x - x_window[0]) + (x
- x_window[1])) / ((x_window[2] - x_window[0]) * (x_window[2]
- x_window[1]))
                return term0 + term1 + term2

            # Find points in the current segment to compute t
he derivative
            mask = (x_dense >= x_window[0]) & (x_dense <= x_w
indow[-1])
            y_derivative[mask] = [derivative_lagrange(x) for
x in x_dense[mask]]

        return y_derivative

    def plot_and_save(self):
        x_dense = np.linspace(self.x_vals[0], self.x_vals[-
1], 10000)
        y_orig = self.func(x_dense)
        y_interp = self.interpolate(x_dense)

        # Plot original and interpolation
        plt.figure(figsize=(12, 6))
        plt.plot(x_dense, y_orig, label="Original Function",
color="blue", linewidth=2)
        plt.plot(x_dense, y_interp, label="Lagrange Interpola
tion", linestyle="--", color="red", linewidth=2)
        plt.title("Piecewise 2nd-Order Lagrange Interpolation
vs Original Function")
```

```
        plt.legend()
        plt.grid(True)
        plt.savefig("lagrange_interpolation_plot.png")
        plt.show()

        # Derivative comparison
        orig_derivative = Derivative(self.func)(x_dense)
        lagrange_derivative = self.lagrange_derivative(x_dens
e)
        derivative_error = np.abs(orig_derivative - lagrange_
derivative)

        # Plot derivative error
        plt.figure(figsize=(12, 6))
        plt.plot(x_dense, derivative_error, label="Derivative
Error (|f'(x) - L'(x)|)", color="purple", linewidth=2)
        plt.title("Error Between Derivatives of Original Func
tion and Lagrange Interpolation")
        plt.xlabel("x")
        plt.ylabel("Error")
        plt.grid(True)
        plt.legend()
        plt.savefig("lagrange_derivative_error_plot.png")
        plt.show()

        print(f"Maximum derivative error: {np.max(derivative_
error):.6f}")

if __name__ == "__main__":
    num_points = 1000
    lagrange_interp = PiecewiseLagrangeInterpolator(f, num_po
ints)
    lagrange_interp.generate_points()
    lagrange_interp.plot_and_save()
```

- Maximum error: 0.001416 Lagrange
- Maximum derivative error: 0.007307

For Cubic





```
import numpy as np
import matplotlib.pyplot as plt
import numdifftools as nd

class CubicSplineInterpolator:
    def __init__(self, func, num_points):
        self.func = func
        self.num_points = num_points
        self.x_vals = None
        self.y_vals = None
        self.a = None
        self.b = None
        self.c = None
        self.d = None

    def generate_points(self):
        self.x_vals = np.sort(np.random.uniform(0, 1000, sel
f.num_points))
        self.y_vals = self.func(self.x_vals)

    def compute_cubic_spline(self):
        n = self.num_points - 1
        h = np.diff(self.x_vals)
        alpha = np.zeros(n)

        for i in range(1, n):
            alpha[i] = (3/h[i] * (self.y_vals[i+1] - self.y_v
als[i]) - 3/h[i-1] * (self.y_vals[i] - self.y_vals[i-1]))

        l = np.ones(n+1)
        mu = np.zeros(n)
        z = np.zeros(n+1)

        for i in range(1, n):
            l[i] = 2 * (self.x_vals[i+1] - self.x_vals[i-1])
```

```
- h[i-1] * mu[i-1]
            mu[i] = h[i] / l[i]
            z[i] = (alpha[i] - h[i-1] * z[i-1]) / l[i]

        self.c = np.zeros(n+1)
        self.b = np.zeros(n)
        self.d = np.zeros(n)
        self.a = self.y_vals[:-1]

        for j in range(n-1, -1, -1):
            self.c[j] = z[j] - mu[j] * self.c[j+1]
            self.b[j] = (self.y_vals[j+1] - self.y_vals[j]) /
h[j] - h[j] * (self.c[j+1] + 2 * self.c[j]) / 3
            self.d[j] = (self.c[j+1] - self.c[j]) / (3 * h
[j])

    def spline(self, x):
        i = np.searchsorted(self.x_vals, x) - 1
        i = max(min(i, self.num_points - 2), 0)
        dx = x - self.x_vals[i]
        return self.a[i] + self.b[i] * dx + self.c[i] * dx**2
+ self.d[i] * dx**3

    def compute_error(self, x_random):
        f_prime = np.array([nd.Derivative(self.func)(x) for x
in x_random])
        spline_prime = np.array([self.spline_derivative(x) fo
r x in x_random])
        return f_prime, spline_prime, np.abs(f_prime - spline
_prime)

    def spline_derivative(self, x):
        i = np.searchsorted(self.x_vals, x) - 1
        i = max(min(i, self.num_points - 2), 0)
        dx = x - self.x_vals[i]
        return self.b[i] + 2 * self.c[i] * dx + 3 * self.d[i]
```

```
* dx**2

    def plot(self):
        x_interp = np.linspace(0, 1000, 1000)
        y_interp = np.array([self.spline(x) for x in x_inter
p])
        y_orig = self.func(x_interp)

        plt.figure(figsize=(12, 6))
        plt.plot(x_interp, y_orig, label="Original Function",
color="blue", linewidth=2)
        plt.plot(x_interp, y_interp, label="Cubic Spline Inte
rpolation", linestyle="--", color="red", linewidth=2)
        plt.title("Cubic Spline Interpolation vs Original Fun
ction")
        plt.legend()
        plt.savefig('spline_vs_function2.png')
        plt.show()

    def plot_error(self, x_random):
        f_prime, spline_prime, error = self.compute_error(x_r
andom)

        plt.figure(figsize=(12, 6))
        plt.scatter(x_random, error, label="Slope Error (|
f'(x) - S'(x)|)", color="purple", marker='o')
        plt.title("Error between Function Slope and Cubic Spl
ine Slope (Scatter Plot)")
        plt.xlabel("x")
        plt.ylabel("Error")
        plt.legend()
        plt.savefig('slope_error_scatter2.png')
        plt.show()

        avg_error = np.mean(error)
        print(f"Average error in slope: {avg_error:.6f}")
```

```
if __name__ == "__main__":
    def f(x):
       return np.sin(x/4) + np.cos(x/4)

    num_points = 1000
    spline_interp = CubicSplineInterpolator(f, num_points)

    spline_interp.generate_points()
    spline_interp.compute_cubic_spline()
    spline_interp.plot()
    random_points = np.random.uniform(0, 1000, 1000)
    spline_interp.plot_error(random_points)
```

- Average error in slope: 0.000547