Name: Phan Tran Thanh Huy

ID: ITCSIU22056

Lab 6

Activity 1: Piecewise Linear Interpolation

Code:

```python
# Given data points
x_values = [0, 1, 2, 3, 4]
y_values = [1, 2.2, 3.0, 3.6, 4.5]

# 1. Piecewise linear interpolation function
def linear_interpolation(x_values, y_values, x):
    for i in range(len(x_values) - 1):
        xi, xi1 = x_values[i], x_values[i + 1]
        if xi <= x <= xi1:
            yi, yi1 = y_values[i], y_values[i + 1]
            # Apply linear interpolation formula
            return yi + ((yi1 - yi) / (xi1 - xi)) * (x - xi)
    raise ValueError("x is out of bounds of the given data")

# 2. Estimate f(2.5)
x_to_estimate = 2.5
estimated_value = linear_interpolation(x_values, y_values, x_to_estimate)

# Output the result
print(f"Estimated f({x_to_estimate}) using piecewise linear interpolation is: {estimated_value}")
```

Result:

```
PS E:\Homework\TMC\Lab6> & C:/Python312/python.exe e:/Homework/TMC/Lab6/p1.py
Estimated f(2.5) using piecewise linear interpolation is: 3.3
PS E:\Homework\TMC\Lab6>
```

Activity 2: Lagrange Polynomial Interpolation

Code:

```python
# Given data points
x_values = [0, 1, 2, 3]
y_values = [1, 2, 1, 3]

# 1. Lagrange interpolation
def lagrange_interpolation(x_values, y_values, x):
    total = 0
    n = len(x_values)
    for j in range(n):
        term = y_values[j]
        for i in range(n):
            if i != j:
                term *= (x - x_values[i]) / (x_values[j] - x_values[i])
        total += term
    return total

# 2. Estimate f(1.5)
x_to_estimate = 1.5
estimated_value = lagrange_interpolation(x_values, y_values, x_to_estimate)

# Output the result
print(f"Estimated f({x_to_estimate}) using Lagrange interpolation is: {estimated_value}")
```

Result:

```
PS E:\Homework\TMC\Lab6> & C:/Python312/python.exe e:/Homework/TMC/Lab6/p2.py
Estimated f(1.5) using Lagrange interpolation is: 1.4375
```

Activity 3: Newton interpolating polynomial

Code:

```python
# Given data points
x_values = [1, 2, 4, 7]
y_values = [3, 6, 10, 20]

def newton_divided_diff(x, y):
    n = len(x)
    table = [[0 for _ in range(n)] for _ in range(n)]

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

    for j in range(1, n):
        for i in range(n - j):
            numerator = table[i + 1][j - 1] - table[i][j - 1]
            denominator = x[i + j] - x[i]
            table[i][j] = numerator / denominator

    coeffs = [table[0][j] for j in range(n)]
    return coeffs, table

def newton_polynomial(x_data, coeffs, x):
    n = len(coeffs)
    result = coeffs[0]
    product_term = 1.0
    for i in range(1, n):
        product_term *= (x - x_data[i - 1])
        result += coeffs[i] * product_term
    return result

# Step 1: Get coefficients using divided differences (with dynamic programming)
coefficients, table = newton_divided_diff(x_values, y_values)

# Step 2: Estimate f(3)
x_to_estimate = 3
estimated_value = newton_polynomial(x_values, coefficients, x_to_estimate)

# Output results
print(f"Estimated f({x_to_estimate}) using Newton interpolation: {estimated_value:.6f}")
```

Result:

```
PS E:\Homework\TMC\Lab6> & C:/Python312/python.exe e:/Homework/TMC/Lab6/p3.py
Estimated f(3) using Newton interpolation is: 8.133333333333335
```

Activity 4: Comparison & Error Analysis

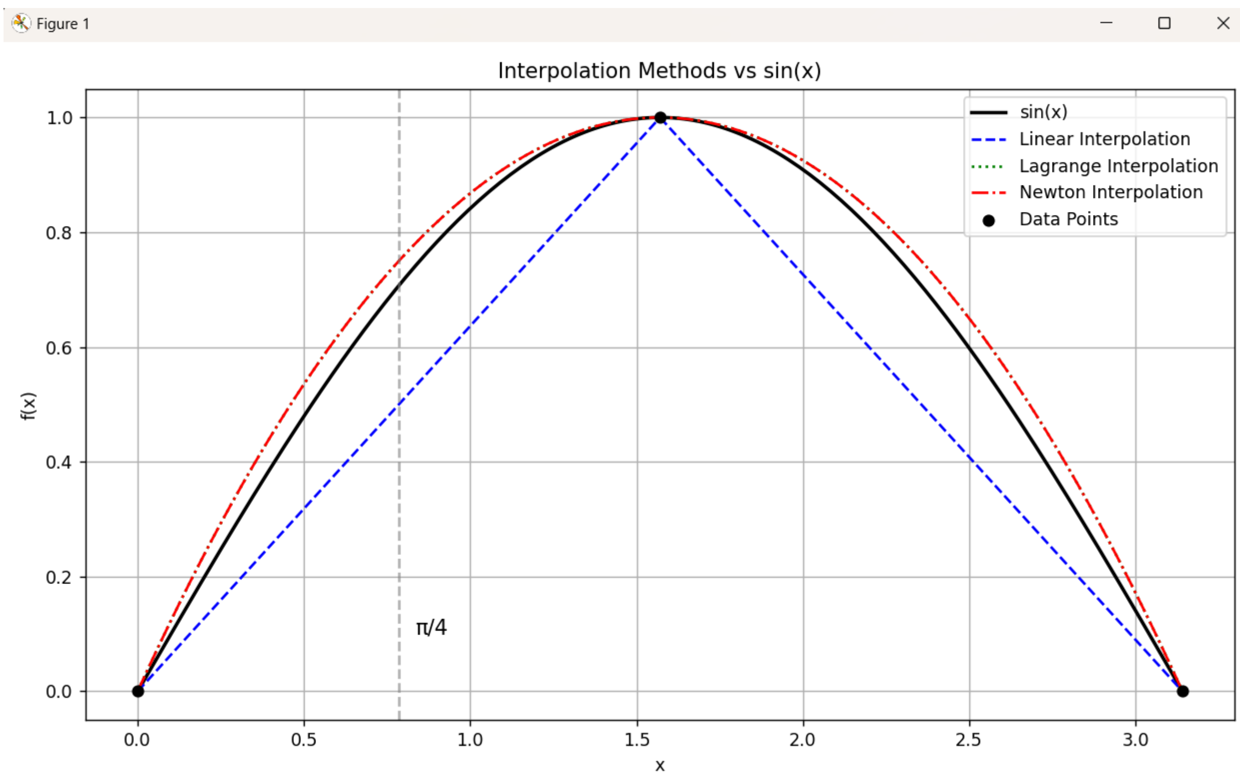
Code:

```python
import math
import numpy as np
import matplotlib.pyplot as plt

# Data points
x_values = [0, math.pi / 2, math.pi]
y_values = [math.sin(x) for x in x_values]
x_target = math.pi / 4
true_value = math.sin(x_target)

# 1. Linear interpolation (piecewise)
def linear_interpolation(x_vals, y_vals, x):
    for i in range(len(x_vals) - 1):
        if x_vals[i] <= x <= x_vals[i + 1]:
            xi, xi1 = x_vals[i], x_vals[i + 1]
            yi, yi1 = y_vals[i], y_vals[i + 1]
            return yi + ((yi1 - yi) / (xi1 - xi)) * (x - xi)
    raise ValueError("x out of bounds")

# 2. Lagrange interpolation
def lagrange_interpolation(x_vals, y_vals, x):
    total = 0
    n = len(x_vals)
    for j in range(n):
        term = y_vals[j]
        for i in range(n):
            if i != j:
                term *= (x - x_vals[i]) / (x_vals[j] - x_vals[i])
        total += term
    return total

# 3. Newton interpolation
def newton_divided_diff(x, y):
    n = len(x)
    table = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        table[i][0] = y[i]
    for j in range(1, n):
        for i in range(n - j):
            table[i][j] = (table[i + 1][j - 1] - table[i][j - 1]) / (x[i + j] - x[i])
    coeffs = [table[0][j] for j in range(n)]
    return coeffs

def newton_polynomial(x_data, coeffs, x):
    result = coeffs[0]
    product = 1
    for i in range(1, len(coeffs)):
        product *= (x - x_data[i - 1])
        result += coeffs[i] * product
    return result

# Apply all three methods
linear_result = linear_interpolation(x_values, y_values, x_target)
lagrange_result = lagrange_interpolation(x_values, y_values, x_target)
newton_coeffs = newton_divided_diff(x_values, y_values)
newton_result = newton_polynomial(x_values, y_values, x_target)

# Compute errors
linear_error = abs(true_value - linear_result)
lagrange_error = abs(true_value - lagrange_result)
newton_error = abs(true_value - newton_result)

# Output results
print(f"True value: sin(π/4) = {true_value}")
print("\nInterpolation Results and Errors at x = π/4:")

print(f"Linear interpolation:   {linear_result:.6f} | Absolute Error: {linear_error:.6f}")
print(f"Lagrange interpolation: {lagrange_result:.6f} | Absolute Error: {lagrange_error:.6f}")
print(f"Newton:   {newton_result:.6f} | Absolute Error: {newton_error:.6f}")

# Function and domain
x_dense = np.linspace(0, math.pi, 200)
true_y = np.sin(x_dense)

# Interpolations over the dense x values
linear_y = [linear_interpolation(x_values, y_values, xi) for xi in x_dense]
lagrange_y = [lagrange_interpolation(x_values, y_values, xi) for xi in x_dense]
newton_y = [newton_polynomial(x_values, newton_coeffs, xi) for xi in x_dense]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(x_dense, true_y, label='sin(x)', color='black', linewidth=2)
plt.plot(x_dense, linear_y, '--', label='Linear Interpolation', color='blue')
plt.plot(x_dense, lagrange_y, ':', label='Lagrange Interpolation', color='green')
plt.plot(x_dense, newton_y, '-.', label='Newton Interpolation', color='red')

# Plot original data points
plt.scatter(x_values, y_values, color='black', zorder=5, label='Data Points')

# Mark π/4
plt.axvline(x=math.pi/4, color='gray', linestyle='--', alpha=0.6)
plt.text(math.pi/4 + 0.05, 0.1, 'π/4', fontsize=12)

plt.title('Interpolation Methods vs sin(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Result:

```
PS E:\Homework\TMC\Lab6> & C:/Python312/python.exe e:/Homework/TMC/Lab6/p4.py
True value: sin(π/4) = 0.7071067811865476

Interpolation Results and Errors at x = π/4:
Linear:   0.500000 | Absolute Error: 0.207107
Lagrange: 0.750000 | Absolute Error: 0.042893
Newton:   0.750000 | Absolute Error: 0.042893
```

Plot:



Which method is computationally most efficient?

Linear is fastest, but Newton is the best polynomial method when using dynamic programming for large datasets.

When is Lagrange better than Newton?

- A one-off interpolation for a small number of points.
- Simpler mathematical formulation (no table or recursion).

When is Newton better than Lagrange?

- Reuse the interpolation for many values of x.
- Add data points incrementally (extendable).
- Optimize performance using divided differences table.