Student Name: Nguyen Dinh Khanh Ngan

Student ID: ITCSIU22236

Lab 6

Activity 1: Piecewise Linear Interpolation

Code:

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

x = 2.5

def linear_interpolation(x, x_values, y_values):
    for i in range(len(x_values) - 1):
        if x_values[i] <= x < x_values[i + 1]:
            xi = x_values[i]
            xi_plus_1 = x_values[i + 1]
            yi = y_values[i]
            yi_plus_1 = y_values[i + 1]
            break
    else:
        raise ValueError("x is out of the range of x_values")

    f_x = yi + ((yi_plus_1 - yi) / (xi_plus_1 - xi)) * (x - xi)
    return f_x

result = linear_interpolation(x, x_values, y_values)
print(f"The interpolated value at x = {x} is: {result}")
```

Result:

```
PS E:\Homework\TMC\Lab6> & C:/Python312/python.exe e:/Homework/TMC/Lab6/p1.py
The interpolated value at x = 2.5 is: 3.3
```

Activity 2: Lagrange Polynomial Interpolation

Code:

```python
x_values = [0, 1, 2, 3]
y_values = [1, 2, 1, 3]

x = 1.5

def lagrange_interpolation(x, x_values, y_values):
    n = len(x_values)
    result = 0

    for i in range(n):
        term = y_values[i]
        for j in range(n):
            if j != i:
                term *= (x - x_values[j]) / (x_values[i] - x_values[j])
        result += term

    return result

estimated_value = lagrange_interpolation(x, x_values, y_values)

print(f"The estimated value of f(1.5) using Lagrange interpolation is: {estimated_value}")
```

Result:

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

Activity 3: Newton interpolating polynomial

Code:

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

x_input = 3

def newton_interpolation(x, x_values, y_values):
    n = len(x_values)
    divided_diff = [[0] * n for _ in range(n)]

    for i in range(n):
        divided_diff[i][0] = y_values[i]

    for j in range(1, n):
        for i in range(n - j):
            divided_diff[i][j] = (divided_diff[i + 1][j - 1] - divided_diff[i][j - 1]) / (x_values[i + j] - x_values[i])

    result = divided_diff[0][0]
    product = 1
    for i in range(1, n):
        product *= (x - x_values[i - 1])
        result += divided_diff[0][i] * product

    return result

estimated_value = newton_interpolation(x_input, x_values, y_values)

print(f"The estimated value of f(3) using Newton interpolation is: {estimated_value}")
```

Result:

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

Activity 4: Comparison & Error Analysis

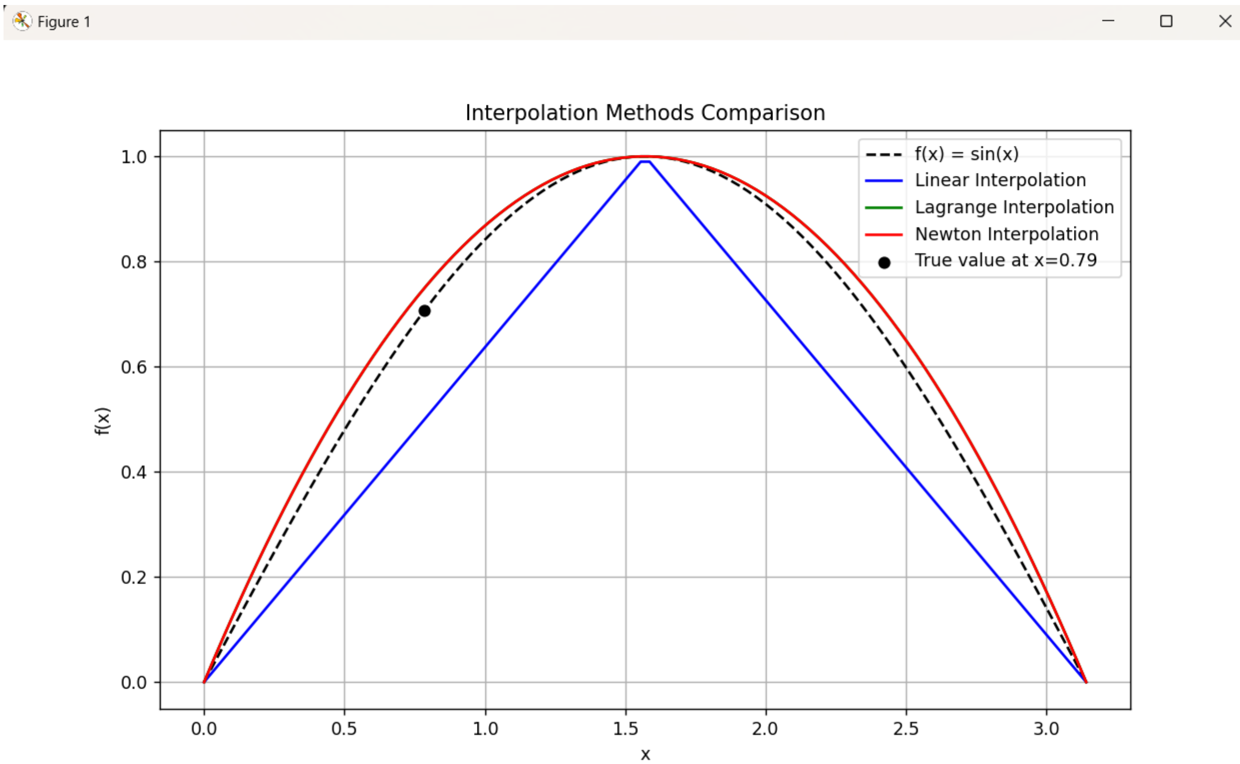
Code:

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

f = np.sin
x_values = np.array([0, np.pi / 2, np.pi])
y_values = f(x_values)

def linear_interpolation(x, x_values, y_values):
    if x == x_values[0]:
        return y_values[0]
    if x == x_values[-1]:
        return y_values[-1]

    for i in range(len(x_values) - 1):
        if x_values[i] <= x < x_values[i + 1]:
            xi = x_values[i]
            xi_plus_1 = x_values[i + 1]
            yi = y_values[i]
            yi_plus_1 = y_values[i + 1]
            break
    else:
        raise ValueError("x is out of the range of x_values")

    f_x = yi + ((yi_plus_1 - yi) / (xi_plus_1 - xi)) * (x - xi)
    return f_x

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

def newton_interpolation(x, x_values, y_values):
    n = len(x_values)
    divided_diff = [[0] * n for _ in range(n)]

    for i in range(n):
        divided_diff[i][0] = y_values[i]

    for j in range(1, n):
        for i in range(n - j):
            divided_diff[i][j] = (divided_diff[i + 1][j - 1] - divided_diff[i][j - 1]) / (x_values[i + j] - x_values[i])

    result = divided_diff[0][0]
    product = 1
    for i in range(1, n):
        product *= (x - x_values[i - 1])
        result += divided_diff[0][i] * product

    return result

x_test = np.linspace(0, np.pi, 100)
y_true = f(x_test)

y_linear = np.array([linear_interpolation(x, x_values, y_values) for x in x_test])
y_lagrange = np.array([lagrange_interpolation(x, x_values, y_values) for x in x_test])
y_newton = np.array([newton_interpolation(x, x_values, y_values) for x in x_test])

x_error = np.pi / 4
y_true_error = f(x_error)

y_linear_error = linear_interpolation(x_error, x_values, y_values)
y_lagrange_error = lagrange_interpolation(x_error, x_values, y_values)
y_newton_error = newton_interpolation(x_error, x_values, y_values)

linear_error = abs(y_true_error - y_linear_error)
lagrange_error = abs(y_true_error - y_lagrange_error)
newton_error = abs(y_true_error - y_newton_error)

print(f"Linear Interpolation Error at x = pi/4: {linear_error}")
print(f"Lagrange Interpolation Error at x = pi/4: {lagrange_error}")
print(f"Newton Interpolation Error at x = pi/4: {newton_error}")

plt.figure(figsize=(10, 6))
plt.plot(x_test, y_true, label="f(x) = sin(x)", color="black", linestyle="--")
plt.plot(x_test, y_linear, label="Linear Interpolation", color="blue")
plt.plot(x_test, y_lagrange, label="Lagrange Interpolation", color="green")
plt.plot(x_test, y_newton, label="Newton Interpolation", color="red")
plt.scatter([x_error], [y_true_error], color="black", zorder=5, label=f"True value at x={x_error:.2f}")
plt.title("Interpolation Methods Comparison")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.grid(True)
plt.show()
```

Result:

```
PS E:\Homework\TMC\Lab6> & C:/Python312/python.exe e:/Homework/TMC/Lab6/p4.py
Linear Interpolation Error at x = pi/4: 0.20710678118654757
Lagrange Interpolation Error at x = pi/4: 0.04289321881345243
Newton Interpolation Error at x = pi/4: 0.04289321881345243
```

Plot:



Question:

Computational Efficiency

1. **Linear Interpolation**: Fastest method. It just uses two points, so it takes constant time, O(1).
2. **Lagrange Interpolation**: Slower than linear. For n points, it takes $O(n^2)$ time because it calculates polynomials for each point.
3. **Newton Interpolation**: More efficient than Lagrange for large datasets. It takes $O(n^2)$ to set up, but O(n) to evaluate once the setup is done, making it faster than Lagrange for many points.

When to use:

- Lagrange
  - Small datasets (few points).
  - One-time interpolation, no need to add more points.
- Newton
  - Large datasets or many points.
  - Add new data points without redoing all the work.