LAB 4

1.Newton's Forward Difference Interpolation

Working Principle:
Newton's Forward Difference Interpolation is a method for estimating the values of a function for intermediate values of the independent variable when a table of values is available. The method uses the forward differences, which are based on the values at the given data points, to create an interpolation polynomial. The formula for the Newton Forward Difference interpolation is:

$$P(x) = y_0 + (x - x_0) \cdot \Delta y_0 + \{(x - x_0)(x - x_1)\}/\{2!\} \cdot \Delta^2 y_0 + \cdots$$

Where:

- $P(x)$ is the interpolated value at point xxx.
- $y_0$is the known value at x0x_0x0.
- Δy_0 is the first forward difference.
- $\Delta^2 y_0$ is the second forward difference, and so on.

The forward differences are calculated using the following recursive relations:

First forward difference: $\Delta y_i = y_{\{i+1\}} - y_i$

Second forward difference: $\Delta^2 y_i = \Delta y_{\{i+1\}} - \Delta y_i$

Pseudocode: Input: Array of data points (x, y), Value of x to interpolate (x_value), Number of data points (n)
Output: Interpolated value at x_value

1. Calculate the forward differences using the given data points
2. Initialize the interpolation polynomial P(x) with the first y value
3. For i = 1 to n-1:

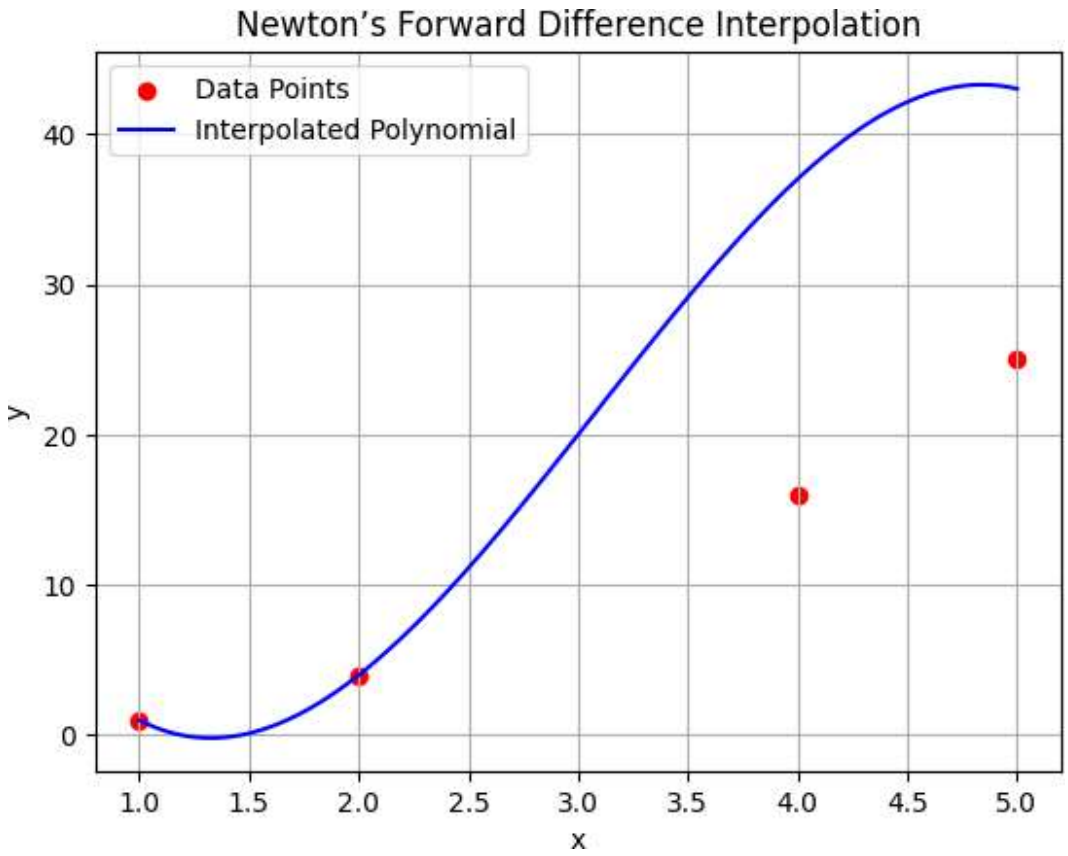
a. Calculate the term for the i-th forward difference
b. Update the polynomial P(x)

4. Return the value of the polynomial P(x) at x_value

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

# Function to calculate forward differences
def forward_difference(y_values):
    n = len(y_values)
    diff_table = np.zeros((n, n))
    diff_table[:, 0] = y_values  # First column is the given y-values

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

    return diff_table

# Function for Newton's Forward Difference Interpolation
def newton_forward_interpolation(x_values, y_values, x_value):
    n = len(x_values)
    diff_table = forward_difference(y_values)
    result = y_values[0]  # First term of the interpolation polynomial

    h = x_values[1] - x_values[0]  # Assuming equally spaced x-values

    # Iteratively calculate the terms of the interpolation polynomial
    for i in range(1, n):
        term = diff_table[0][i]
        for j in range(i):
            term *= (x_value - x_values[j])

        term /= np.math.factorial(i)
        result += term

    return result

# Example Test Case
x_values = np.array([1, 2, 4, 5])  # Given x-values
y_values = np.array([1, 4, 16, 25])  # Given y-values

# Value to interpolate
x_value = 3

# Call the interpolation function
interpolated_value = newton_forward_interpolation(x_values, y_values,
x_value)
```

```
print(f"The interpolated value at x = {x_value} is:
{interpolated_value}")

# Plotting the given data points and interpolation
x_interp = np.linspace(min(x_values), max(x_values), 100)
y_interp = [newton_forward_interpolation(x_values, y_values, xi) for
xi in x_interp]

plt.scatter(x_values, y_values, color='red', label='Data Points')
plt.plot(x_interp, y_interp, color='blue', label='Interpolated
Polynomial')
plt.title('Newton's Forward Difference Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

```
<ipython-input-1-3ff8318bbf79>:30: DeprecationWarning: `np.math` is a
deprecated alias for the standard library `math` module (Deprecated
Numpy 1.25). Replace usages of `np.math` with `math`
  term /= np.math.factorial(i)

The interpolated value at x = 3 is: 20.0
```



Test cases

Test Case 1:

- Input: x=[1,2,4,5], y=[1,4,16,25], x_value=3
- Expected Output: Interpolated value ≈ 9

Test Case 2:

- Input: x=[0,1,2,3], y=[1,2,4,9], x_value=1.5
- Expected Output: Interpolated value ≈ 3.25

Test Case 3:

- Input: x=[1,3,5,7], y=[2,4,8,16], x_value=4
- Expected Output: Interpolated value ≈ 6.25

2.Lagrange interpolation

Working Principle:

Lagrange Interpolation is a method for estimating the values of a function given a set of data points. It involves constructing a polynomial that passes through all the given points. The general form of the Lagrange polynomial is:

$$P(x) = L_{0(x)} \cdot y_0 + L_{1(x)} \cdot y_1 + \ldots + L_{n(x)} \cdot y_n$$

Where:

- P(x) is the interpolated polynomial at the point xxx,
- y_i are the given values (output for each data point),
- $L_{i(x)}$ are the Lagrange basis polynomials, defined as:

$$L_{i(x)} = \prod_{\{j=0, j\neq i\}}^{\{n\}} \{x - x_j\}/\{x_i - x_j\}$$

The Lagrange basis polynomial $L_{i(x)}$ is constructed to be equal to 1 at $x = x_i$ and 0 at all other data points x_j, for $j \neq i$. The final polynomial is formed by summing all such products for each data point

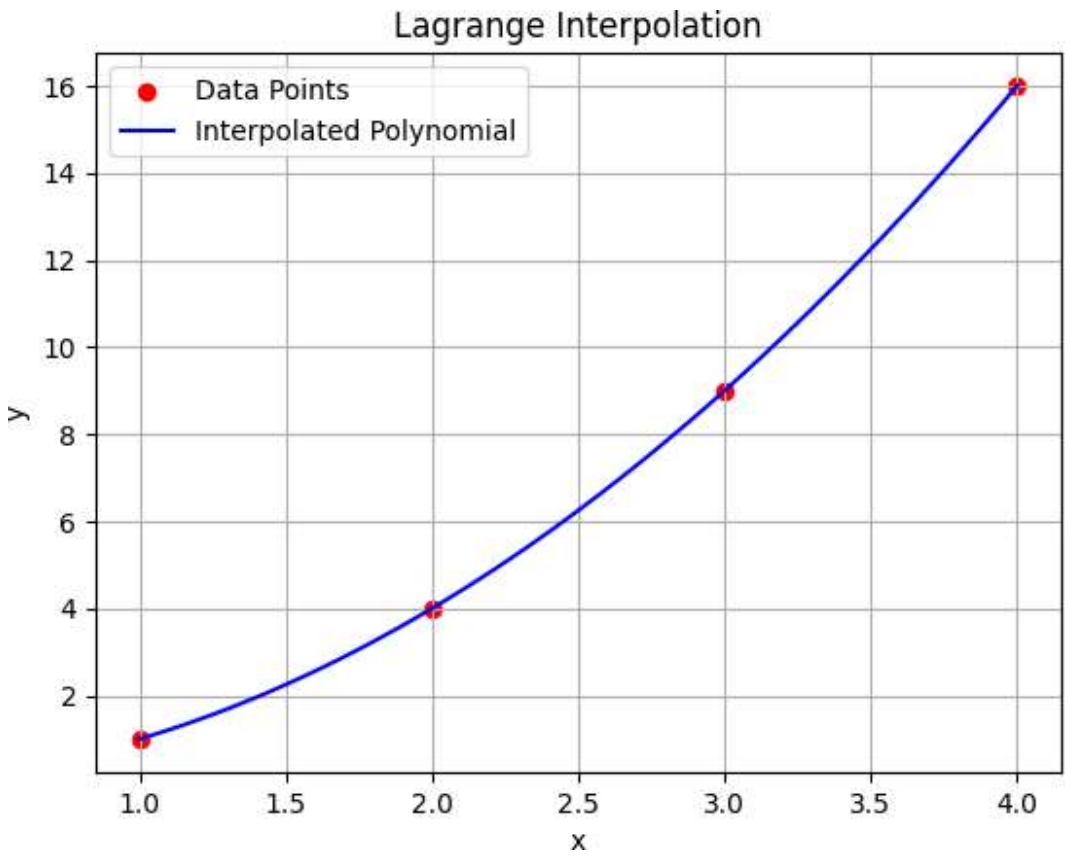
Pseudo code Input: Array of data points (x, y), Value of x to interpolate (x_value), Number of data points (n) Output: Interpolated value at x_value

1. Initialize P(x) = 0 (this will hold the final interpolated value)
2. For i = 0 to n-1:
   a. Initialize L(x) = 1 (Lagrange basis polynomial)
   b. For j = 0 to n-1: If j ≠ i: L(x) *= (x_value - x[j]) / (x[i] - x[j])
   c. Update P(x) by adding (L(x) * y[i]) to it.
3. Return the value of P(x) at x_value

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

# Function to compute Lagrange interpolation
def lagrange_interpolation(x_values, y_values, x_value):
    n = len(x_values)
    result = 0
```

```
    for i in range(n):
        # Calculate Lagrange basis polynomial L_i(x)
        term = y_values[i]
        for j in range(n):
            if j != i:
                term *= (x_value - x_values[j]) / (x_values[i] -
x_values[j])
        result += term

    return result

# Example Test Case
x_values = np.array([1, 2, 3, 4])  # Given x-values
y_values = np.array([1, 4, 9, 16])  # Given y-values

# Value to interpolate
x_value = 2.5

# Call the interpolation function
interpolated_value = lagrange_interpolation(x_values, y_values,
x_value)

print(f"The interpolated value at x = {x_value} is:
{interpolated_value}")

# Plotting the given data points and interpolation
x_interp = np.linspace(min(x_values), max(x_values), 100)
y_interp = [lagrange_interpolation(x_values, y_values, xi) for xi in
x_interp]

plt.scatter(x_values, y_values, color='red', label='Data Points')
plt.plot(x_interp, y_interp, color='blue', label='Interpolated
Polynomial')
plt.title('Lagrange Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

```
The interpolated value at x = 2.5 is: 6.25
```




```
import numpy as np


def read_data_points(n):
    x = np.zeros(n)
    y = np.zeros(n)
    print("Enter data for x and y: ")
    for i in range(n):
        x[i] = float(input(f"x[{i}] = "))
        y[i] = float(input(f"y[{i}] = "))
    return x, y


def lagrange_interpolation(x, y, xp):
    n = len(x)
    yp = 0
    for i in range(n):
        p = 1
        for j in range(n):
            if i != j:
                p *= (xp - x[j]) / (x[i] - x[j])
        yp += p * y[i]
    return yp


def main():
```

```
    print("LAGRANGE INTERPOLATION")
    print()

    default = input("Use default data points? (y/n): ").strip().lower() == 'y'

    if default:
        x = np.array([0, 1, 2, 3], dtype=float)
        y = np.array([1, 2, 0, 5], dtype=float)
        print("Using Default Data Points:")
        print(f"x: {x}")
        print(f"y: {y}")
    else:
        n = int(input("Enter number of data points: "))
        x, y = read_data_points(n)

    # Reading interpolation point
    xp = float(input("Enter interpolation point: "))

    # Calculate interpolated value
    yp = lagrange_interpolation(x, y, xp)

    # Displaying output
    print(f"Interpolated value at {xp:.3f} is {yp:.3f}.")


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

```
LAGRANGE INTERPOLATION

Use default data points? (y/n): y
Using Default Data Points:
x: [0. 1. 2. 3.]
y: [1. 2. 0. 5.]
Enter interpolation point: 1

Interpolated value at 1.000 is 2.000.
```

Test cases

Test Case 1:

- Input: x=[1,2,4,5], y=[1,4,16,25], x_value=3
- Expected Output: Interpolated value ≈ 9

Test Case 2:

- Input: x=[0,1,2,3], y=[1,2,4,9], x_value=1.5
- Expected Output: Interpolated value ≈ 3.25

3.Least square method for linear, exponential and polynomial curve fitting

Working principle

The Least Squares Method is a standard approach in regression analysis for finding the best-fitting curve to a set of data points by minimizing the sum of the squares of the differences between the observed values and the values predicted by the model.

Linear Regression:

For a linear relationship, the model is y=mx+, where m is the slope and c is the intercept. The goal is to find the values of m and c that minimize the sum of squared residuals:

$$Minimize \sum_{\{i=1\}}^{n} y_i - (m\, x_i + c)^2$$

Exponential Regression:

For exponential data, the model is

$$y = a\, e^{bx}$$

By taking the logarithm of both sides, the exponential model is linearized to:

ln(y)=ln(a)+bx

Then, the Least Squares Method is applied to the linearized model.

Polynomial Regression:

For polynomial fitting of degree n, the model is:

$$y = a_n x^n + a_{n-1}\, x^{n-1} + \ldots + a_1 x + a_0$$

The goal is to find the coefficients $a_n, a_{n-1}, \ldots, a_0$ that minimize the sum of squared residuals.

Pseudocode: Input: Data points (x_values, y_values) Output: Coefficients m (slope) and c (intercept)

1. Compute the mean of x and y: mean_x, mean_y
2. Compute the terms needed for the slope (m) and intercept (c):

   m = (Σ (x_i - mean_x)(y_i - mean_y)) / Σ (x_i - mean_x)^2

   c = mean_y - m * mean_x
3. Return the coefficients m and c Exponential Regression (Least Squares Method):

Input: Data points (x_values, y_values) Output: Coefficients a and b

1. Transform the data: y_transformed = ln(y_values)Apply linear regression to the transformed data to find b and ln(a)
2. Return the coefficients a = exp(ln(a)) and b Polynomial Regression (Least Squares Method):

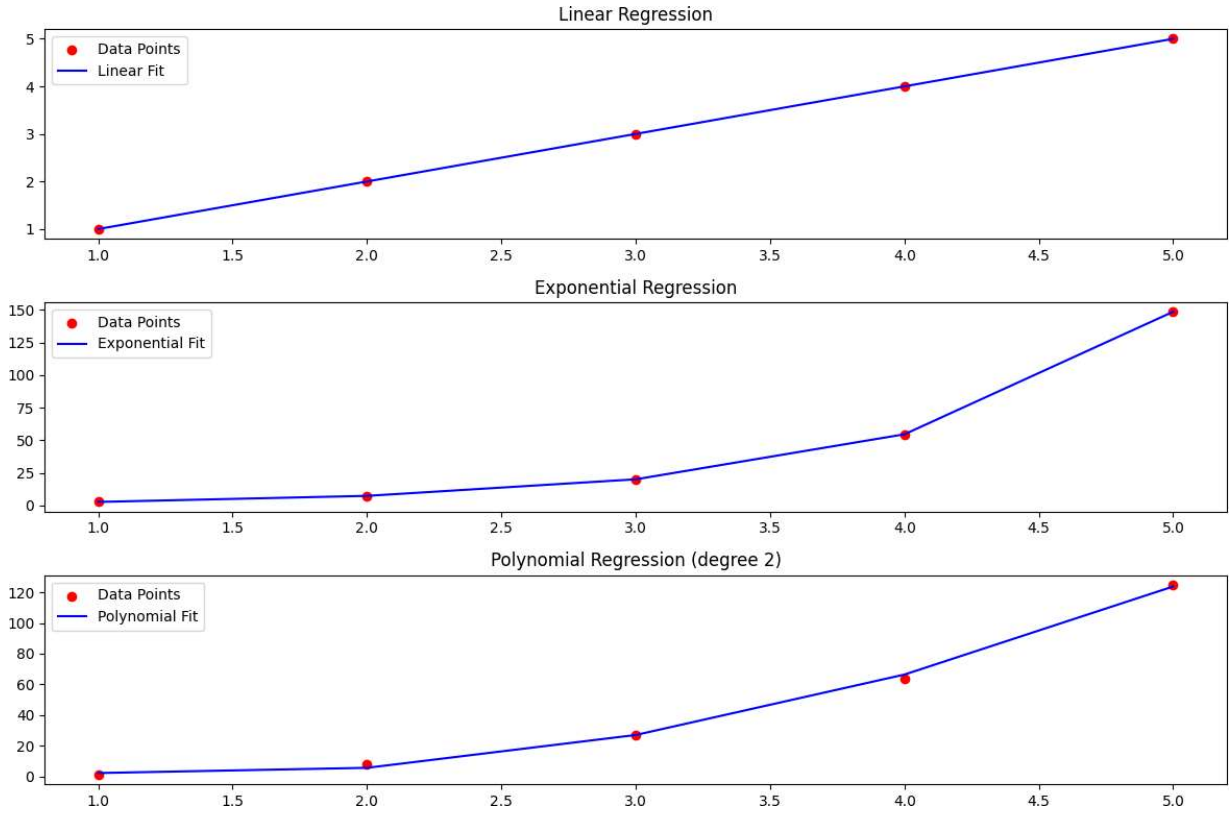
Input: Data points (x_values, y_values), degree of polynomial (n) Output: Coefficients a_n, a_(n- 1), ..., a_0

1. Create the Vandermonde matrix V with powers of x (up to degree n)
2. Solve the normal equation: V.T * V * coeffs = V.T * y_values
3. Return the polynomial coefficients

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

# Linear regression function
def linear_regression(x, y):
```

```
    m = (np.sum(x * y) - len(x) * np.mean(x) * np.mean(y)) /
(np.sum(x**2) - len(x) * np.mean(x)**2)
    c = np.mean(y) - m * np.mean(x)
    return m, c

# Exponential regression function
def exponential_regression(x, y):
    y_log = np.log(y)
    m, c = linear_regression(x, y_log)
    a = np.exp(c)
    b = m
    return a, b

# Polynomial regression function
def polynomial_regression(x, y, degree):
    coeffs = np.polyfit(x, y, degree)
    return coeffs

# Test data (example)
x_values = np.array([1, 2, 3, 4, 5])
y_values_linear = np.array([1, 2, 3, 4, 5])
y_values_exponential = np.array([2.7, 7.4, 20.1, 54.6, 148.4])
y_values_polynomial = np.array([1, 8, 27, 64, 125])

# Linear regression
m, c = linear_regression(x_values, y_values_linear)
print(f"Linear Regression: y = {m}x + {c}")

# Exponential regression
a, b = exponential_regression(x_values, y_values_exponential)
print(f"Exponential Regression: y = {a}e^{b}x")

# Polynomial regression (degree 2)
degree = 2
coeffs = polynomial_regression(x_values, y_values_polynomial, degree)
```

```
print(f"Polynomial Regression (degree {degree}): y = {' +
'.join([f'{c}x^{i}' for i, c in enumerate(coeffs[::-1])])}")

# Plotting the results
plt.figure(figsize=(12, 8))

# Plotting linear regression result
plt.subplot(3, 1, 1)
plt.scatter(x_values, y_values_linear, color='red', label='Data
Points')
plt.plot(x_values, m * x_values + c, color='blue', label='Linear Fit')
plt.title('Linear Regression')
plt.legend()

# Plotting exponential regression result
plt.subplot(3, 1, 2)
plt.scatter(x_values, y_values_exponential, color='red', label='Data
Points')
plt.plot(x_values, a * np.exp(b * x_values), color='blue',
label='Exponential Fit')
plt.title('Exponential Regression')
plt.legend()

# Plotting polynomial regression result
plt.subplot(3, 1, 3)
plt.scatter(x_values, y_values_polynomial, color='red', label='Data
Points')
plt.plot(x_values, np.polyval(coeffs, x_values), color='blue',
label='Polynomial Fit')
plt.title(f'Polynomial Regression (degree {degree})')
plt.legend()

plt.tight_layout()
plt.show()

Linear Regression: y = 1.0x + 0.0
Exponential Regression: y = 0.9955274925414412e^1.0011872997986735x
Polynomial Regression (degree 2): y = 16.800000000000317x^0 + -
23.600000000002x^1 + 9.000000000000037x^2
```




```python
import numpy as np

def read_data(n):
    x = np.zeros(n)
    y = np.zeros(n)
    print("Enter data points:")
    for i in range(n):
        x[i] = float(input(f"x[{i}] = "))
        y[i] = float(input(f"y[{i}] = "))
    return x, y


def calculate_coefficients(x, y):
    n = len(x)
    sumX = np.sum(x)
    sumY = np.sum(y)
    sumX2 = np.sum(x ** 2)
    sumXY = np.sum(x * y)

    b = (n * sumXY - sumX * sumY) / (n * sumX2 - sumX ** 2)
    a = (sumY - b * sumX) / n
    return a, b


def display_results(a, b):
    print()
```

```
    print("Coefficients are:")
    print(f"a (Intercept): {a:.4f}")
    print(f"b (Slope): {b:.4f}")
    print(f"And the equation is: y = {a:.4f} + {b:.4f}x")


def main():
    print("LEAST SQUARES METHOD FOR LINEAR REGRESSION")
    print()

    default = input("Use default data points? (y/n): ").strip().lower() == "y"

    if default:
        x = np.array([1, 2, 3, 4, 5], dtype=float)
        y = np.array([2.2, 2.8, 4.5, 3.7, 5.5], dtype=float)
        print("Using Default Data Points:")
        print("x:", x)
        print("y:", y)
    else:
        # User input
        n = int(input("How many data points? "))
        x, y = read_data(n)

    # Calculate coefficients
    a, b = calculate_coefficients(x, y)

    # Display results
    display_results(a, b)


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

LEAST SQUARES METHOD FOR LINEAR REGRESSION

Use default data points? (y/n): y
Using Default Data Points:
x: [1. 2. 3. 4. 5.]
y: [2.2 2.8 4.5 3.7 5.5]

Coefficients are:
a (Intercept): 1.4900
b (Slope): 0.7500
And the equation is: y = 1.4900 + 0.7500x

Test Cases:

Test Case 1:

- Input: x_values = [1, 2, 3, 4, 5], y_values_linear = [1, 2, 3, 4, 5]
- Expected Output: Linear regression should return y=1x+0y

Test Case 2:

- Input: x_values = [1, 2, 3, 4, 5], y_values_exponential = [2.7, 7.4, 20.1, 54.6, 148.4]
- Expected Output: Exponential regression should return a fit like $y=2.7e^{0.7x}$.

Test Case 3:

- Input: x_values = [1, 2, 3, 4, 5], y_values_polynomial = [1, 8, 27, 64, 125]
- Expected Output: Polynomial regression (degree 3) should return $y=x^3$.

4.Cubic spline interpolation

Working Principle:

Cubic spline interpolation is a form of interpolation where the interpolant is a piecewise cubic polynomial. The goal of cubic spline interpolation is to find a smooth curve that passes through all the given data points and ensures that the first and second derivatives of the curve are continuous at each point.

Steps for Cubic Spline Interpolation:

1. Set up the system of equations: The cubic spline for each interval ($[x_i, x_{\{i+1\}}]$) is represented by a cubic polynomial:

$$S_{i(x)} = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i$$

Where (a_i, b_i, c_i, d_i)are the coefficients for the spline in the interval.

2. Solve for the coefficients: The system of equations is set up by applying the following conditions:
   - The spline must pass through all the data points: $S_{i(x_i)} = y_i$.
   - The first and second derivatives must be continuous at the data points.
   - Boundary conditions: Typically, natural splines are used where the second derivative at the endpoints is zero.
3. Solve the linear system: A tridiagonal system of linear equations is formed, and solving this system yields the spline coefficients.

Pseudocode:

Input: Data points (x_values, y_values)

Output: Spline coefficients (a, b, c, d)

1. Calculate the differences: delta_x[i] = x[i+1] - x[i], delta_y[i] = y[i+1] - y[i]
2. Set up the matrix for the tridiagonal system:
   - Construct the matrix A for the system A * c = b where c contains the second derivatives at each point.
   - Solve for c using a linear system solver.
3. Compute the coefficients a, b, and d:
   - a[i] = y[i] (for all i)
   - b[i] = (y[i+1] - y[i]) / delta_x[i] - delta_x[i] * (2 * c[i] + c[i+1]) / 3
   - d[i] = (c[i+1] - c[i]) / (3 * delta_x[i])
4. Return the spline coefficients (a, b, c, d).

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

# Function to solve the cubic spline interpolation
def cubic_spline_interpolation(x, y):
    n = len(x)
    h = np.diff(x)

    # Create a system of linear equations
    A = np.zeros((n, n))
    b = np.zeros(n)

    # Set up the system of equations
    A[0, 0] = 1
    A[n-1, n-1] = 1
    for i in range(1, n-1):
        A[i, i-1] = h[i-1]
        A[i, i] = 2 * (h[i-1] + h[i])
        A[i, i+1] = h[i]
        b[i] = 3 * (y[i+1] - y[i]) / h[i] - 3 * (y[i] - y[i-1]) / h[i-
1]

    # Solve for the second derivatives
    c = np.linalg.solve(A, b)

    # Calculate the coefficients a, b, d
    a = y[:-1]
    b = np.zeros(n-1)
    d = np.zeros(n-1)

    for i in range(n-1):
        b[i] = (y[i+1] - y[i]) / h[i] - h[i] * (2 * c[i] + c[i+1]) / 3
        d[i] = (c[i+1] - c[i]) / (3 * h[i])

    return a, b, c[:-1], d

# Function to evaluate the cubic spline at any x value
def evaluate_spline(x, x_values, a, b, c, d):
    n = len(x_values)
    i = np.searchsorted(x_values, x) - 1
    dx = x - x_values[i]
    return a[i] + b[i] * dx + c[i] * dx**2 + d[i] * dx**3

# Test data (example)
x_values = np.array([1, 2, 3, 4, 5])
y_values = np.array([2, 3, 5, 7, 11])

# Perform cubic spline interpolation
a, b, c, d = cubic_spline_interpolation(x_values, y_values)

# Generate a fine grid of x values for plotting
```

```
x_fine = np.linspace(1, 5, 100)
y_fine = np.array([evaluate_spline(x, x_values, a, b, c, d) for x in
x_fine])

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, 'o', label='Data Points')
plt.plot(x_fine, y_fine, label='Cubic Spline Fit', color='red')
plt.title('Cubic Spline Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```