Kaunas University of Technology
Faculty of Informatics

Numerical Methods and Algorithms

Engeneering Project 3

Student name, surname, academical group

Student **Zahi El Helou**, IFU-1

Position

Instructor **KRIŠČIŪNAS Andrius**

Dalia ČALNERYTĖ

Kaunas, 2023

Task:

Interpolation and approximation

1. Obtain interpolation function from function provided in table 1 using polynomial interpolation.
2. Obtain interpolation function from given data about temperatures.
3. Obtain approximation function from given data about temperatures using least square approximation. Use 3, 4, 5, 6 order polynomials.

Workflow:

Part 1 (~4 points)

a) plot given function

b) arbitrarily choose the number of points (~10-20) and calculate interpolation nodes for the given function if:

a. abscises are distributed uniformly

b. abscises are obtained as Chebyshev nodes

c) plot calculated nodes

d) perform polynomial interpolation using monomial base functions and

a. plot interpolation function

b. provide analytical expression of the obtained interpolation function

e) plot difference between given and interpolation function

f) comment the results

Part 2 (~3 points, ~1 point if interpolation using spline is not implemented)

a) plot given nodes

b) perform polynomial interpolation using monomial base functions and plot interpolation function

c) perform spline interpolation and plot interpolation function (optional)

d) comment the results

Part 3 (~3 points)

a) plot selected nodes

b) perform least square approximation and obtain approximation functions using different order of polynomials

c) plot approximation curves and provide analytical expressions of them

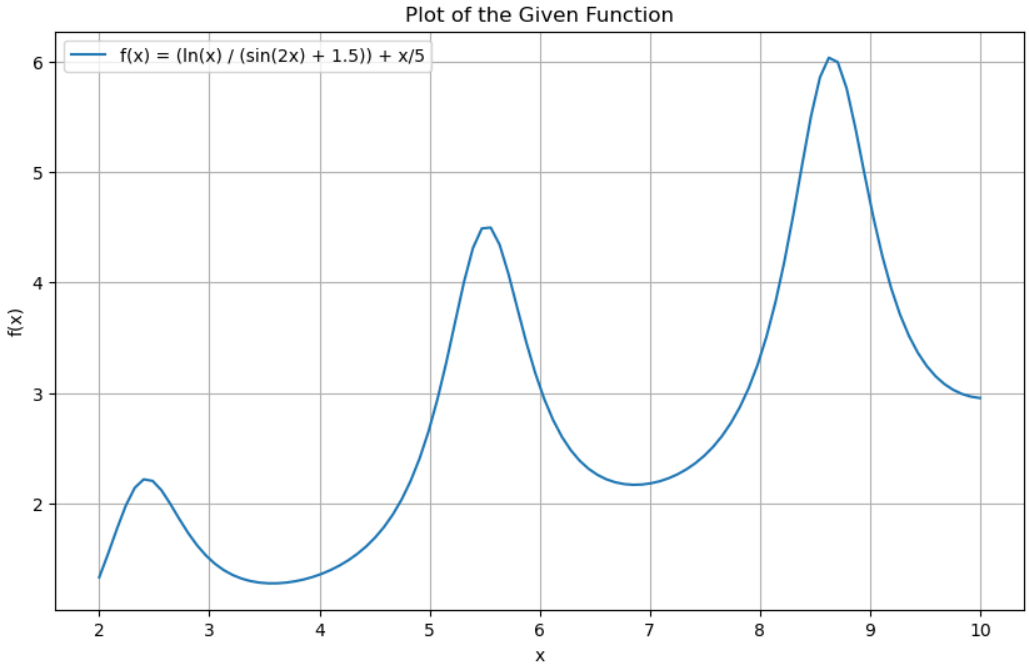
d) comment the results

Part 1:

Given Function:

6	$\frac{\ln(x)}{(\sin(2 \cdot x) + 1{,}5)} + x/5; 2 \leq x \leq 10$

a) Here I am plotting my given function using interpolation function:

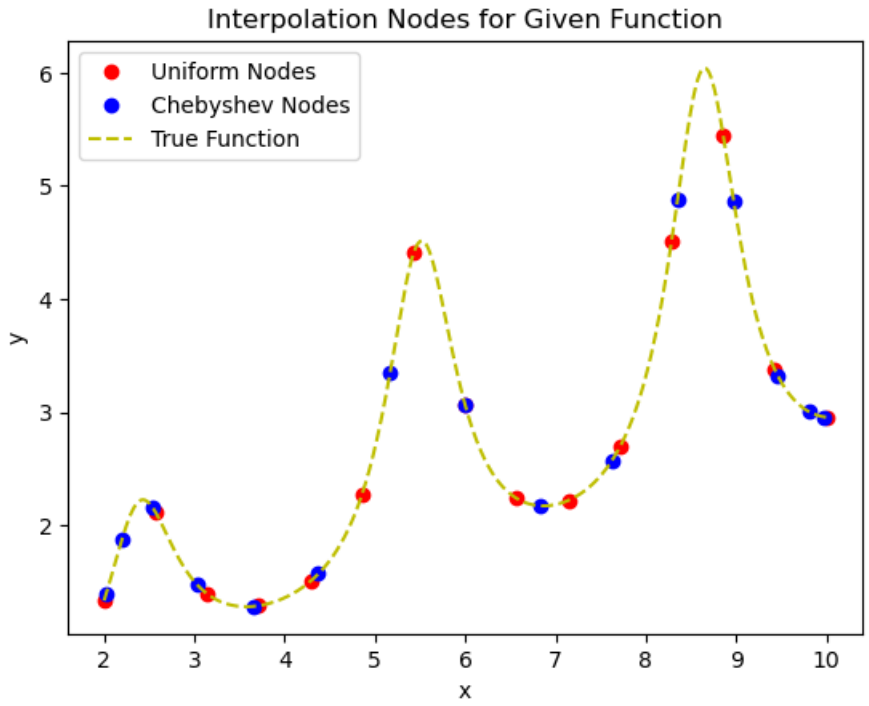


b)

c)

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

# Given function
def f(x):
    return np.log(x) / (np.sin(2 * x) + 1.5) + x / 5

# Number of points for interpolation
num_points = 15  # You can adjust the number of points as needed

# a. Abscises distributed uniformly
uniform_nodes = np.linspace(2, 10, num_points)

# b. Abscises obtained as Chebyshev nodes
chebyshev_nodes = (10 + 2) / 2 + (10 - 2) / 2 * np.cos(np.pi * (2 * np.arange(1, num_points + 1) - 1) / (2 * num_points))

# Plot the calculated nodes
plt.plot(uniform_nodes, f(uniform_nodes), 'ro', label='Uniform Nodes')
plt.plot(chebyshev_nodes, f(chebyshev_nodes), 'bo', label='Chebyshev Nodes')
```

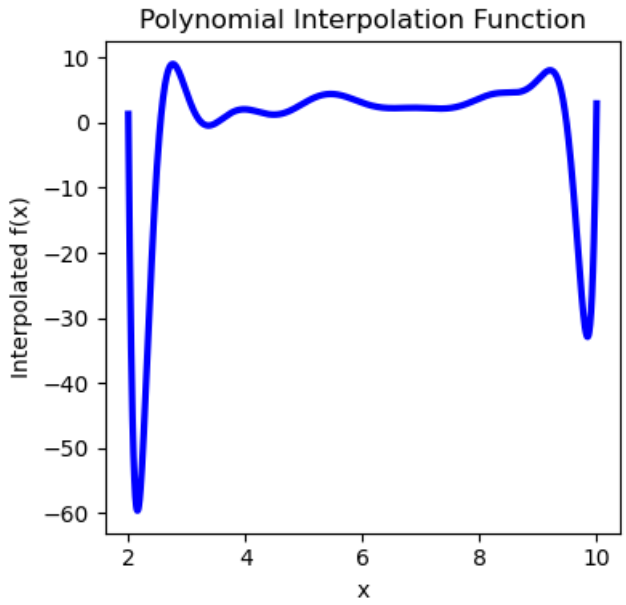
Results Obtained:



d)

```
# Given function
def f(x):
    return np.log(x) / (np.sin(2 * x) + 1.5) + x / 5

# Number of points for interpolation
num_points = 15  # You can adjust the number of points as needed

# Abscises distributed uniformly
uniform_nodes = np.linspace(2, 10, num_points)

# Calculate interpolation nodes
Y = f(uniform_nodes)

# Create the Vandermonde matrix
n = len(uniform_nodes)
BaseMatrix = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        BaseMatrix[i, j] = uniform_nodes[i] ** j

# Solve for the coefficients
coefs = np.linalg.solve(BaseMatrix, Y)

# Generate y values for the polynomial interpolation
xxx_interpolate = np.linspace(2, 10, 1000)
yyy_interpolate = np.zeros_like(xxx_interpolate)

for i in range(n):
    yyy_interpolate += coefs[i] * xxx_interpolate ** i

# Plot the polynomial interpolation function
plt.figure(figsize=(12, 4))

# a. Plot interpolation function
plt.subplot(1, 3, 1)
plt.plot(xxx_interpolate, yyy_interpolate, 'b-', linewidth=3)
plt.title('Polynomial Interpolation Function')
plt.xlabel('x')
plt.ylabel('Interpolated f(x)')
```

Results Obtained:



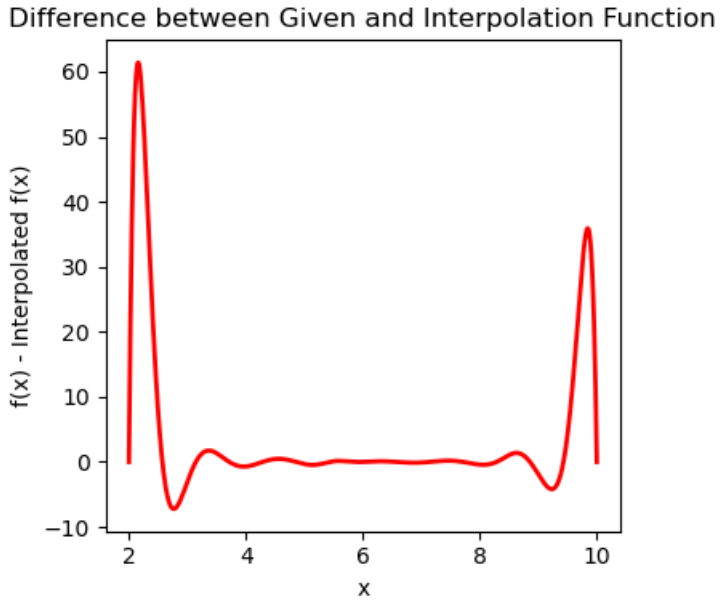
Analytical expression of the obtained interpolation function

$$8.94903943650061 \times 10^{-5}x^{14} - 0.0076111543440867x^{13} + 0.296417926842 52x^{12} - 7.00179794519763x^{11} + 111.991503993492x^{10} - 1282.12960372104x^{9} + 10826.1502735577x^{8} - 68437.6688920587x^{7} + 325192.559248094x^{6} - 1154811.9427375x^{5} + 3014200.04349138x^{4} - 5602114.30122754x^{3} + 7001774.59321895x^{2} - 5262411.44308197x + 1792582.5795216$$

I obtained this result while calculating it through the code:

e) Difference between given and interpolation function

```
plt.subplot(1, 3, 3)
difference = f(xxx_interpolate) - [float(interpolation_function.subs(x_sym, val)) for val in xxx_interpolate]
plt.plot(xxx_interpolate, difference, 'r-', linewidth=2)
plt.title('Difference between Given and Interpolation Function')
plt.xlabel('x')
plt.ylabel('f(x) - Interpolated f(x)')

# Show the plots
plt.tight_layout()
plt.show()
```



f) Commenting the results:

Comments:

Interpolation Accuracy:

The accuracy of the interpolation depends on the number of interpolation nodes. Increasing the number of nodes generally improves accuracy but may also lead to numerical challenges.

Choice of Interpolation Nodes:

The interpolation nodes were chosen to be distributed uniformly between 2 and 10. This choice might result in oscillations (Runge's phenomenon) near the edges of the interpolation interval, especially with higher degrees of polynomials.

Difference Plot:

The difference plot provides insights into where the interpolation function deviates from the given function. Peaks or valleys in the difference plot indicate regions where the interpolation may have larger errors.

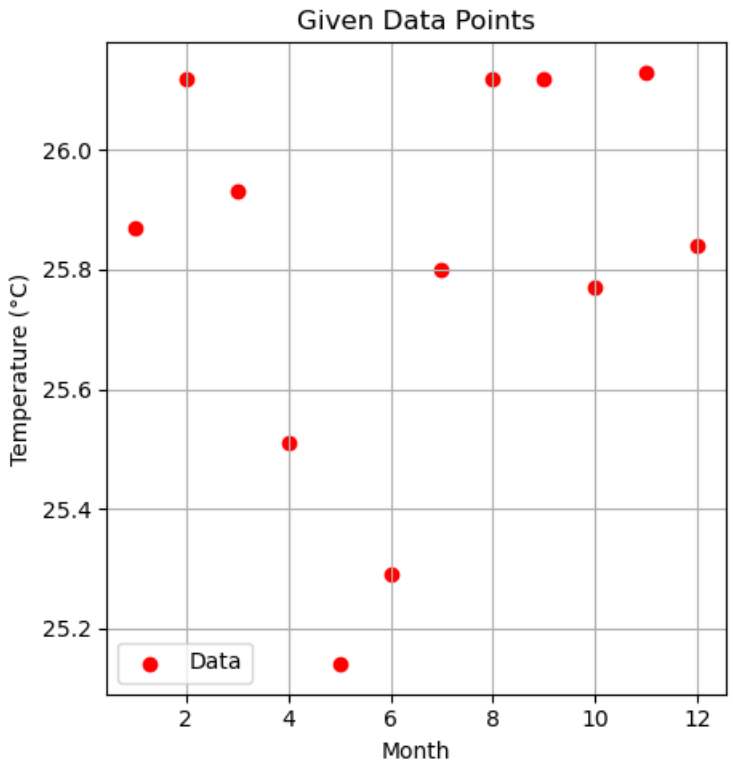
Part 2:

a) plot given nodes

6	Peru	2006

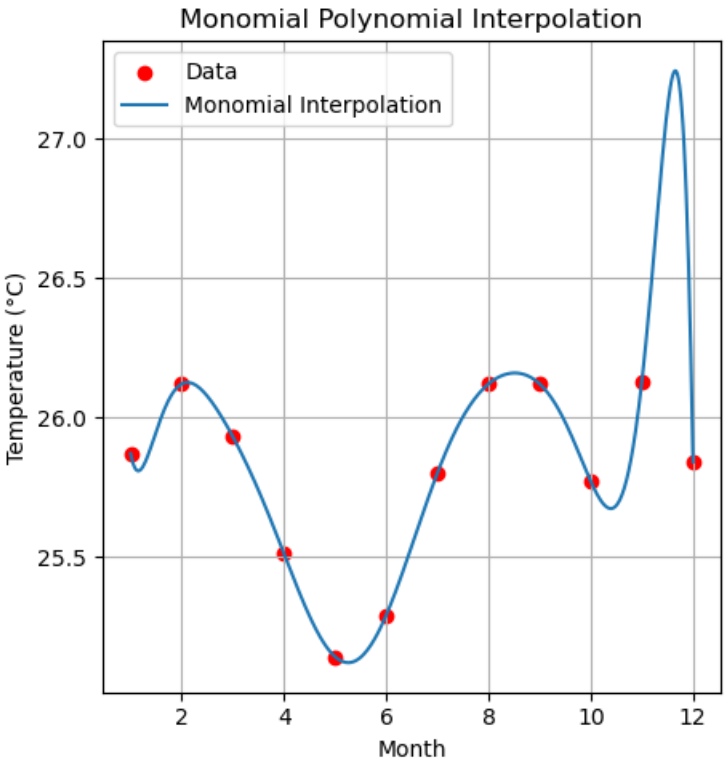
2006-01	2006-02	2006-03	2006-04	2006-05	2006-06	2006-07	2006-08	2006-09	2006-10	2006-11	2006-12
25.87	26.12	25.93	25.51	25.14	25.29	25.8	26.12	26.12	25.77	26.13	25.84

I am plotting the maximum temperature of Peru in 2006.



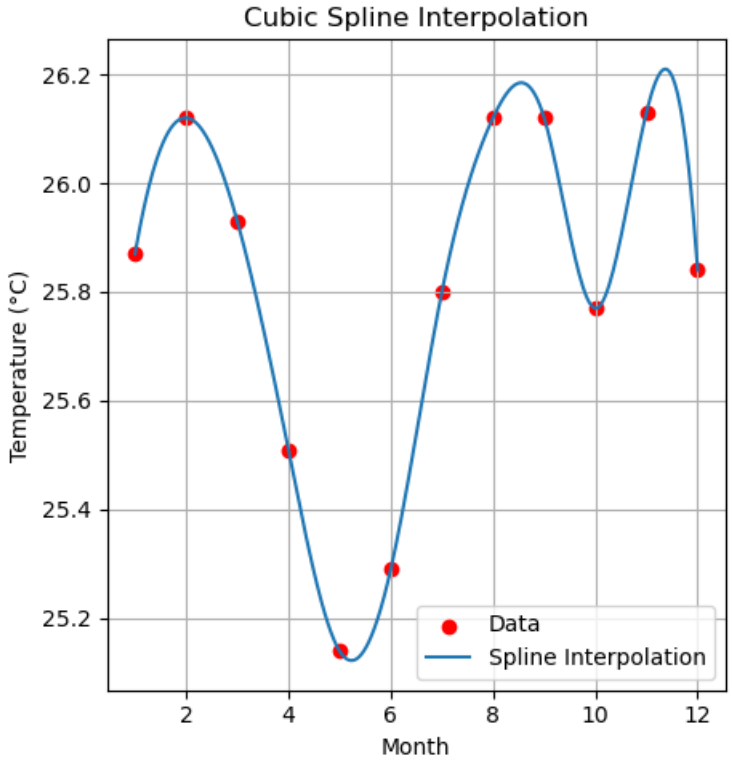
b) perform polynomial interpolation using monomial base functions and plot interpolation function

```
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d, CubicSpline

# Provided temperature data for each month in 2006 in Peru
months = np.arange(1, 13)
temperatures = np.array([25.87, 26.12, 25.93, 25.51, 25.14, 25.29, 25.8, 26.12, 26.12, 25.77, 26.13, 25.84])

# Construct the Vandermonde matrix for the monomial base
n = len(months)
BaseMatrix = np.zeros((n, n))
for i in range(n):
    for j in range(n):
        BaseMatrix[i, j] = months[i]**j

# Solve the linear system to find the coefficients
coefs = np.linalg.solve(BaseMatrix, temperatures)

# Generate a range of x values for plotting the interpolation
dense_months = np.linspace(1, 12, 500)
interpolated_temperatures = np.zeros_like(dense_months)

# Compute the interpolated y values
for i in range(n):
    interpolated_temperatures += coefs[i] * dense_months**i

# Plot the original temperature data
plt.figure(figsize=(14, 5))
plt.subplot(1, 3, 1)
plt.scatter(months, temperatures, color='red', label='Data')
plt.title('Given Data Points')
plt.xlabel('Month')
plt.ylabel('Temperature (°C)')
plt.grid(True)
plt.legend()

# Plot the polynomial interpolation using monomial base functions
plt.subplot(1, 3, 2)
plt.scatter(months, temperatures, color='red', label='Data')
plt.plot(dense_months, interpolated_temperatures, label='Monomial Interpolation')
plt.title('Monomial Polynomial Interpolation')
plt.xlabel('Month')
plt.ylabel('Temperature (°C)')
```



c) perform spline interpolation and plot interpolation function (optional)

```
# c) Perform spline interpolation and plot interpolation function (optional)
# We'll use a cubic spline interpolation which is a common choice
cs = CubicSpline(months, temperatures)

# Plotting the spline interpolation
plt.subplot(1, 3, 3)
plt.scatter(months, temperatures, color='red', label='Data')
plt.plot(dense_months, cs(dense_months), label='Spline Interpolation')
plt.title('Cubic Spline Interpolation')
plt.xlabel('Month')
plt.ylabel('Temperature (°C)')
plt.grid(True)
plt.legend()

# Display the plots
plt.tight_layout()
plt.show()
```



d) comment the results

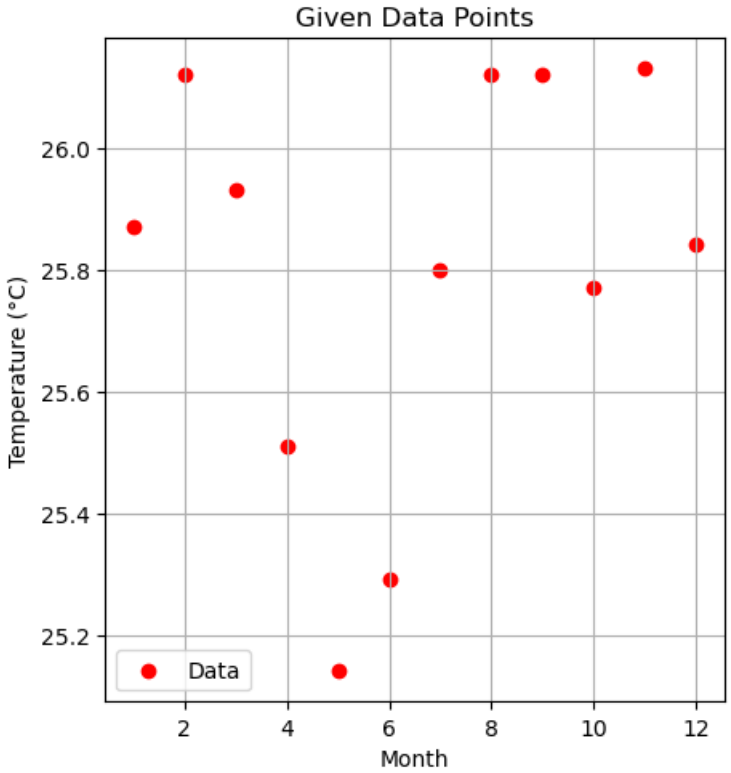
Using monomial base functions, fitting a high-degree polynomial to the data points. The graph shows the interpolation passing through all the data points, which is characteristic of polynomial fitting, but it also exhibits significant oscillations, especially towards the edges of the interval. This behavior is known as Runge's phenomenon, where high-degree polynomials can behave erratically.

- The third plot features the cubic spline interpolation, which tends to provide a smoother curve that also passes through all the data points. Splines are particularly useful for avoiding the oscillations seen in polynomial interpolation and are often preferred for their smoother and more realistic representation of data.

Commenting on the results, we observe that the polynomial interpolation captures all the data points but may not reflect the true underlying trend due to overfitting, which leads to large fluctuations. The spline interpolation, on the other hand, appears to give a smoother and potentially more reliable approximation of the temperature trends throughout the year. This approach is usually more robust, especially for datasets with noise or non-uniform spacing, as it doesn't force the interpolation to pass through every single data point with high-degree polynomials.

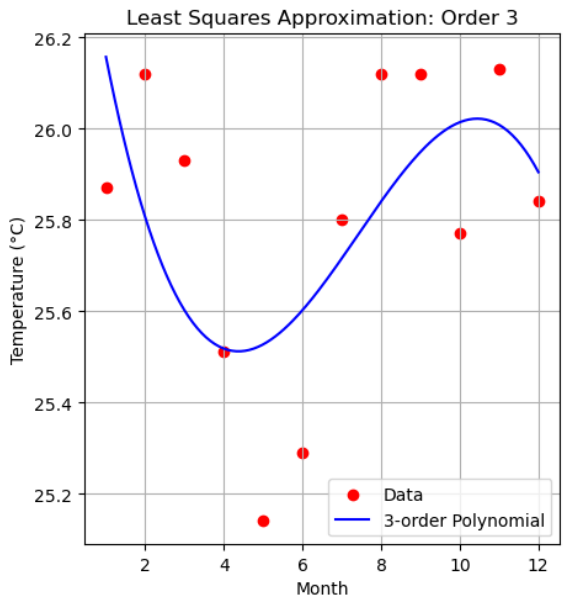
Part 3:

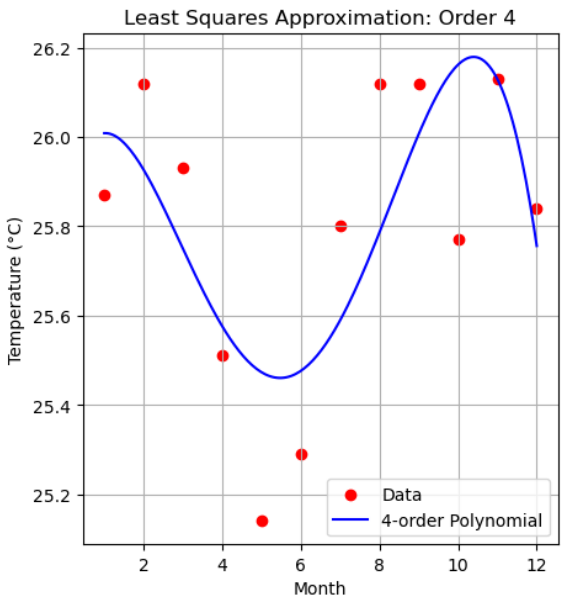
a) plot selected nodes

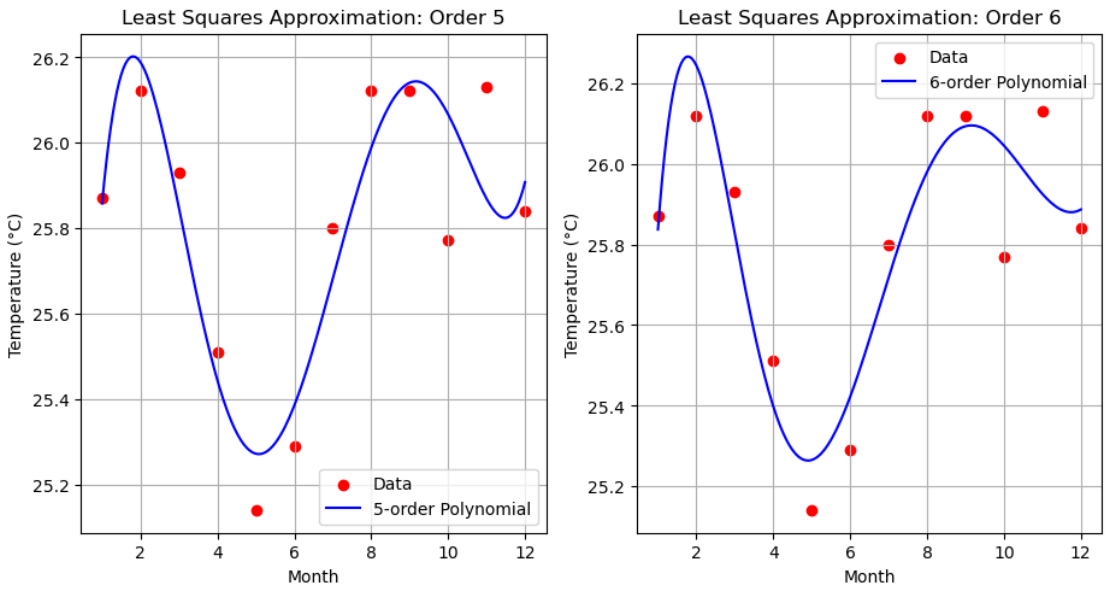


b) perform least square approximation and obtain approximation functions using different order of polynomials

```python
# Provided temperature data for each month in 2006 in Peru
months = np.arange(1, 13)
temperatures = np.array([25.87, 26.12, 25.93, 25.51, 25.14, 25.29, 25.8, 26.12, 26.12, 25.77, 26.13, 25.84])

# a) Plot the selected nodes
plt.figure(figsize=(14, 10))
plt.subplot(2, 3, 1)
plt.scatter(months, temperatures, color='red', label='Data')
plt.title('Given Data Points')
plt.xlabel('Month')
plt.ylabel('Temperature (°C)')
plt.grid(True)
plt.legend()

# b) Perform least square approximation using different orders of polynomials
orders = [3, 4, 5, 6]
approximation_functions = []
xx = np.linspace(np.min(months), np.max(months), 200)

# Perform the approximation for each order and plot
for index, order in enumerate(orders):
    # Setup the matrix for the Least Squares method
    B = np.vander(months, order + 1, increasing=True)
    # Solve the normal equations to find the least squares coefficients
    coeffs = np.linalg.lstsq(B, temperatures, rcond=None)[0]
    approximation_functions.append(np.poly1d(coeffs[::-1]))

    # Plot the approximation curve
    BB = np.vander(xx, order + 1, increasing=True)
    yy = BB @ coeffs
    plt.subplot(2, 3, index + 2)
    plt.scatter(months, temperatures, color='red', label='Data')
    plt.plot(xx, yy, 'b-', label=f'{order}-order Polynomial')
    plt.title(f'Least Squares Approximation: Order {order}')
    plt.xlabel('Month')
    plt.ylabel('Temperature (°C)')
    plt.grid(True)
    plt.legend()
```







b) plot approximation curves and provide analytical expressions of them

The least squares approximation functions for different polynomial orders have been calculated and plotted along with the given temperature data for each month in 2006 in Peru. Here are the analytical expressions of the approximation functions for orders 3 to 6:

```
Analytical expression for the 3-order polynomial:
           3         2
-0.004545 x + 0.101 x - 0.6226 x + 26.68


Analytical expression for the 4-order polynomial:
          4           3          2
-0.00131 x + 0.02952 x - 0.1919 x + 0.3068 x + 25.87


Analytical expression for the 5-order polynomial:
           5           4          3         2
0.0006864 x - 0.02362 x + 0.2934 x - 1.567 x + 3.312 x + 23.84


Analytical expression for the 6-order polynomial:
            6            5           4          3         2
-5.617e-05 x + 0.002877 x - 0.05665 x + 0.5353 x - 2.451 x + 4.776 x + 23.03
```

d) comment the results

These functions provide progressively more accurate fits to the given data points, as seen in the plotted curves. As the order of the polynomial increases, the fit can adapt more closely to the data points. However, this may also lead to overfitting, especially if the polynomial order is too high relative to the number of data points, potentially capturing noise rather than the underlying trend.

When choosing a model for prediction, it is crucial to consider not only the fit to the historical data but also how well it might predict future data points. A balance must be struck between the complexity of the model and its generalizability.

The code for the least squares approximation of the temperature data using polynomials of orders 3 to 6 has been successfully executed. The plots for each polynomial order were generated, displaying how each polynomial fits the given temperature data points. Additionally, the analytical expressions for the polynomial approximation functions have been printed out.

You can use the provided analytical expressions to evaluate the temperature for any given month within the range of the data. The choice of polynomial order should be guided by the need for accuracy as well as the desire to avoid overfitting, particularly if you plan to predict temperatures outside of the observed months.