① Given Data points
(i)

x: 1 2 3 5 7 8

$f(x)$: 3 6 19 99 291 444

① compute divided Differences

$$f[x_i, x_{i+1}] = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

$$f[1,2] = \frac{6-3}{2-1} = 3$$

$$f[2,3] = \frac{19-6}{3-2} = 13$$

$$f[3,5] = \frac{99-19}{5-3} = 40$$

$$f(5,7) = \frac{291-99}{7-5} = 96$$

$$f(7,8) = \frac{444-291}{8-7} = 153$$

Second divided differences

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$$

$$f(1,2,3] = \frac{13-3}{3-1} = 5$$

$$f[2,3,5] = \frac{40-13}{5-2} = 9$$

$$f[3,5,7] = \frac{96-40}{7-3} = 14$$

$$f[5,7,8] = \frac{153-96}{8-5} = 19$$

Third divided differces:

$$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}] = \frac{f[x_{i+1}, x_{i+2}, x_{i+3}] - f[x_i, x_{i+1}, x_{i+2}]}{x_{i+3} - x_i}$$

$$f[1,2,3,5] = \frac{9-5}{5-1} = 1$$

$$f[2,3,5,7] = \frac{14-9}{7-2} = 1$$

$$f[3,5,7,8] = \frac{19-14}{8-3} = 1$$

fourth divided difference

$$f(x_i, x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}] = \frac{f[x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}] - f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]}{x_{i+4} - x_i}$$

$$f[1,2,3,5,7] = \frac{1-1}{7-1} = 0$$

$$f[2,3,5,7,8] = \frac{1-1}{8-2} = 0$$

Newton Interpolating polynomial

1. order 1 polynomial

$P_1(x) = f[x_0] + (x-x_0) f[x_0, x_1]$

$P_1(x) = 3 + (x-1) = 3$

$P_1(x) = 3x$

2. order 2 polynomial

$P_2(x) = P_1(x) + (x-x_0)(x-x_1) f[x_0, x_1, x_2]$

$P_2(x) = 3 + 3(x-1) + (x-1)(x-2) \cdot 5$

$= 5x^2 - 12x + 10$

3. order 3 polynomial

$P_3(x) = P_2(x) + (x-x_0)(x-x_1)(x-x_2) f[x_0, x_1, x_2, x_3]$

$P_3(x) = 5x^2 - 12x + 10 + (x-1)(x-2)(x-3) \cdot 1$

$P_3(x) = x^3 - x^2 - x + 4$

4. order 4 polynomial

$P_4(x) = P_3(x) + (x-x_0)(x-x_1)(x-x_2)(x-x_3) f(x_0, x_1, x_2, x_3, x_4)$

$P_4 = x^3 - x^2 - x + 4$

Since $f[x_0, x_1, x_2, x_3, x_4] = 0$

Evaluate at $x = 4$

$P_1(4) = 12$

$P_2(4) = 42$

$P_3(4) = 48$

$P_4(4) = 48$

The cubic polynomial (order 3) is likely the best fit for the data since it fits the observed pattern of the data points with out adding unnecessary complexity. The values from the cubic and higher order interpolations match indicating that a cubic polynomial might be a good choice for modeling the data.

(ii) Lagrange interpolating polynomial is given by

$$f_n(x) = \sum_{i=0}^{n} L_i(x) f(x_i)$$

where $L_i(x) = \prod_{j=0}^{n} j \neq i \left(\frac{x - x_j}{x_i - x_j} \right)$

Given data points

x: 1 2 3 5 7 8

$f(x)$: 3 6 19 99 291 444

add data point $x = 9$ $f(x) = 510$

1. order 1 polynomial

$(x_0, f(x_0))$ and $(x_1, f(x_1))$

$$f_1(x) = L_0(x) f(x_0) + L_1(x) f(x_1)$$

$$L_0(x) = \frac{x - x_1}{x_0 - x_1}$$

$$L_1(x) = \frac{x - x_0}{x_1 - x_0}$$

(1,3) (2,6)

$$L_0(x) = \frac{x-2}{1-2} = 2-x$$

$$L_1(x) = \frac{x-1}{2-1} = x-1$$

$$f_1(x) = (2-x)3 + 6(x-1) = 3x$$

order 2 polynomial

Now three points

$(x_0, f(x_0))\ (x_1, f(x_1)), (x_2, f(x_2))$

$f_2(x) = L_0(x) f(x_0) + L_1(x) f(x_1) + L_2(x) f(x_2)$

for points $(1,3)\ (2,6)\ (3,19)$

$$L_0(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)} = \frac{(x-2)(x-3)}{2}$$

$$L_1(x) = \frac{(x-1)(x-3)}{(2-1)(2-3)} = -(x-1)(x-3)$$

$$L_2(x) = \frac{(x-1)(x-2)}{(3-1)(3-2)} = \frac{(x-1)(x-2)}{2}$$

$$f_2(x) = \frac{(x-2)(x-3)}{2} \cdot 3 - (x-1)(x-3) \cdot 6 + \frac{(x-1)(x-2)}{2} \cdot 19$$

order 3 polynomial.

for points $(1,3)\ (2,6)\ (3,19)\ (5,99)$

$$L_0(x) = \frac{(x-2)(x-3)(x-5)}{(1-2)(1-3)(1-5)} = \frac{(x-2)(x-3)(x-5)}{-8}$$

$$L_1(x) = \frac{(x-1)(x-3)(x-5)}{(2-1)(2-3)(2-5)} = \frac{(x-1)(x-3)(x-5)}{6}$$

$$L_2(x) = \frac{(x-1)(x-2)(x-5)}{(3-1)(3-2)(3-5)} = -\frac{(x-1)(x-2)(x-5)}{4}$$

$$L_3(x) = \frac{(x-1)(x-2)(x-3)}{(5-1)(5-2)(5-3)} = \frac{(x-1)(x-2)(x-3)}{24}$$

$$f_3(x) = L_0(x)f(1) + L_1(x)f(2) + L_2(x)f(3) + L_3(x)f(5)$$

order 4 polynomial.

$(1,3)\ (2,6)\ (3,19)\ (5,99)\ (7,291)$

similarly As you increase the polynomial order, it fits the data point exactly, However higher degree polynomials can become sensitive to small change. (lead to overfitting)

Adding a new data point to this increases the polynomial degree which leads to oscillations.
here in this about New data point there is sudden fall.

(iii) Estimate $f(4)$ using linear quadratic and natural cubic spline interpolation.

$x: \quad 1 \quad 2 \quad 3 \quad 5 \quad 7 \quad 8$

$f(x): \quad 3 \quad 6 \quad 19 \quad 99 \quad 291 \quad 444$

$x=4$ falls in $(3,5)$

for linear interpolation between $(3,19)$ & $(5,99)$

$$f(x) = f(x_0) + \frac{x-x_0}{x_1-x_0}\left(f(x_1)-f(x_0)\right)$$

$x_0 = 3 \qquad f(x_0) = 19$

$x_1 = 5 \qquad f(x_1) = 99$

$x = 4$

$$f(4) = 19 + \frac{1}{2}(99-19) = 59$$

Quadratic Interpolation.

using pts $(2,6)$ $(3,19)$ $(5,99)$

$f(x) = ax^2 + bx + c$

$$a(2)^2 + b(2) + c = 6$$
$$a(3)^2 + b(3) + c = 19$$
$$a(5)^2 + b(5) + c = 19$$

$a = 9$ $b = -32$ $c = 34$

$f(x) = 9x^2 - 32x + 34$

$f(4) = 9(4)^2 - 32(4) + 34 = 50$

Natural cubic Spline Interpolation.

Q.2 Given Data.

x	5	10	15	20	25
y	10	18	25	30	33

① Straight fit line.

$$y = a_0 + a_1 x$$

$$n a_0 + a_1 \Sigma x_i = \Sigma y_i$$

$$a_0 \Sigma x_i + a_1 \Sigma x_i^2 = \Sigma x_i y_i$$

$$\Sigma x_i = 75$$

$$\Sigma y_i = 116$$

$$\Sigma x_i^2 = 1375 \qquad \Sigma x_i y_i = 2030$$

$$5a_0 + 75 a_1 = 116$$

$$75 a_0 + 1375 a_1 = 2030$$

$$a_0 = 5.8 \qquad a_1 = 1.16$$

Equation of line.

$$5.8 + 1.16x = y$$

2. parabolic

$$y = a_0 + a_1 x + a_2 x^3$$

$$n a_0 + a_1 \sum x_i + a_2 \sum x_i^2 = \sum y_i$$

$$a_0 \sum x_i + a_1 \sum x_i^2 + a_2 \sum x_i^3 = \sum x_i y_i$$

$$a_0 \sum x_i^2 + a_1 \sum x_i^3 + a_2 \sum x_i^4 = \sum x_i^2 y_i$$

$\sum x_i^3 = 28125$

$\sum x_i^4 = 611875$

$\sum x_i^2 y_i = 40300.$

$a = -0.2 \qquad b = \frac{383}{175} \qquad c = \frac{-6}{175}$

$$y = -0.2 + \frac{383}{175} x + \frac{-6}{175} x^2$$

$$\boxed{y = -0.2 + 2.18x - 0.034x^2}$$

③ Power Exponentiation

$$y = ax^b$$

$$\log y = \log a + \log b$$

$$\log y = A + B \log x$$

$$A = \log(a) \quad , \quad B = b$$

x	5	10	15	20	25
y	10	18	25	30	33
$\log x$	0.699	1	1.176	1.301	1.398
$\log y$	1	1.255	1.398	1.477	1.518

$$\Sigma \log x = 5.574$$

$$\Sigma \log y = 6.648$$

$$\Sigma \log(x)^2 = 7.968$$

$$\Sigma \log(x) \log(y) = 7.726$$

$$nA + B \Sigma \log x = \Sigma \log y$$

$$A \Sigma \log x + B \Sigma \log(x)^2 + \Sigma \log(x) \log(y)$$

Solve. $A = 0.476 \quad B = 0.75$

$$y = 3x^{0.75}$$

Q.3 Solution Muller's Method

$f(x) = x^3 + x^2 - 4x - 4$

Initial guesses $x_0 = 1$, $x_1 = 2$, $x_2 = 3$

for $x_0 = 1$ $f(x_0) = 1^3 + 1^2 - 4 \cdot 4 = -6$

for $x_1 = 2$ $f(x_1) = 2^3 + 2^4 - 4 \cdot 2 - 4 = 0$

for $x_2 = 3$ $f(x_2) = 3^3 + 3^2 - 4 \cdot 3 - 4 = 20$

step sizes $h_0 = x_1 - x_0 = 2 - 1 = 1$

$h_1 = x_2 - x_1 = 3 - 2 = 1$

Differences in fun values

$\delta_0 = f(x_1) - f(x_0) = 0 - (-6) = 6$

$\delta_1 = f(x_2) - f(x_1) = 20 - 0 = 20$

Coeff. $a = a = \frac{\delta_1 - \delta_0}{h_1 + h_0} = \frac{20 - 6}{1 + 1} = 7$

$b = a \cdot h_1 + \delta_1 = 7 \cdot 1 + 20 = 27$

Coeff. c $c = f(x_2) = 20$

Apply quadratic formula.

$$x_3 = x_2 + \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}$$

$$x_3 = 3 + \frac{-2 \times 20}{27 \pm 13}$$

for b

$x_3 = 2, 0.143$

update guess

New guess $x_0 = 2$

$x_1 = 3$

$x_2 = 2$

$f(x_0) = 0$

$f(x_1) = 20$

$f(x_2) = 0$

$x_2 = 2$ is a root because $f(x_2) = 0$

Q·3(ii) $f(x) = x^3 - 0.5x^2 + 4x - 2$

initial guess $x_0 = 0,\ x_1 = 1\ \ x_2 = 2$

$f(x_0) = -2$

$f(x_1) = 2.5$

$f(x_2) = 19$

$h_0 = x_1 - x_0 = 1$

$h_1 = x_2 - x_1 = 2 - 1 = 1$

$$d_0 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{2.5 - (-2)}{1 - 0} = 4.5$$

$$d_1 = \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{19 - 2.5}{2 - 1} = 9.5$$

Calculate a, b, c

$$a = \frac{d - d_0}{h_1 + h_0} = \frac{9.5 - 4.5}{1 + 1} = 2.5$$

$b = a \cdot h_1 + d_1 = 2.5 + 9.5 = 12$

$c = f(x_2) = 19$

$$x_3 = x_2 + \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}$$

$$x_3 = 2 + \frac{(-2) \times 12}{12 \pm \sqrt{(12)^2 - 4 \times 5.2 \times 12}}$$

$$x_3 = 2 + \frac{-24}{12 \pm 4.899}$$

choose the sign that matches with b

$$x_3 = 2 + \frac{-24}{12 + 4.899} = 0.58$$

update the points

$x_0 = 1, \ x_1 = 2 \quad x_2 = 0.58$

$f(x_0 = 1)$ and $f(x_1 = 2)$ and $f(0.58)$

$f(0.58) = 0.34691$

compute new differences and coefficient

$h_0 = x_1 - x_0 = 2 - 1 = 1$

$h_1 = x_2 - x_1 = 0.58 - 2 = -1.42$

$d_0 = \dfrac{f(x_1) - f(x_0)}{x_1 - x_0} = \dfrac{12 - 2.5}{1} = 9.5$

$d_1 = \dfrac{f(x_2) - f(x_1)}{x_2 - x_1} = \dfrac{0.34691 - 12}{-1.42} = \dfrac{-11.65309}{-1.42}$

$= 8.2064$

Now a, b, c

$a = \dfrac{d_1 - d_0}{h_1 + h_0} = \dfrac{8.2064 - 9.5}{-1.42 + 1} = 3.08$

$b = ah_1 + d_1 = 3.08(-1.42) + 8.2064$

$= 3.836$

$c = f(x_2) = 0.34691$

$$x_3 = 0.58 + \frac{-2(0.34691)}{3.836 \pm \sqrt{3.836^2 - 4 \cdot 3.08 \times (0.34691)}}$$

$$x_3 = 0.58 + \frac{-0.69382}{3.836 \pm \sqrt{14.709 - 4.272}}$$

$$x_3 = 0.58 \pm \frac{-0.69382}{3.836 \pm \sqrt{10.437}}$$

choose positive as $b > 0$

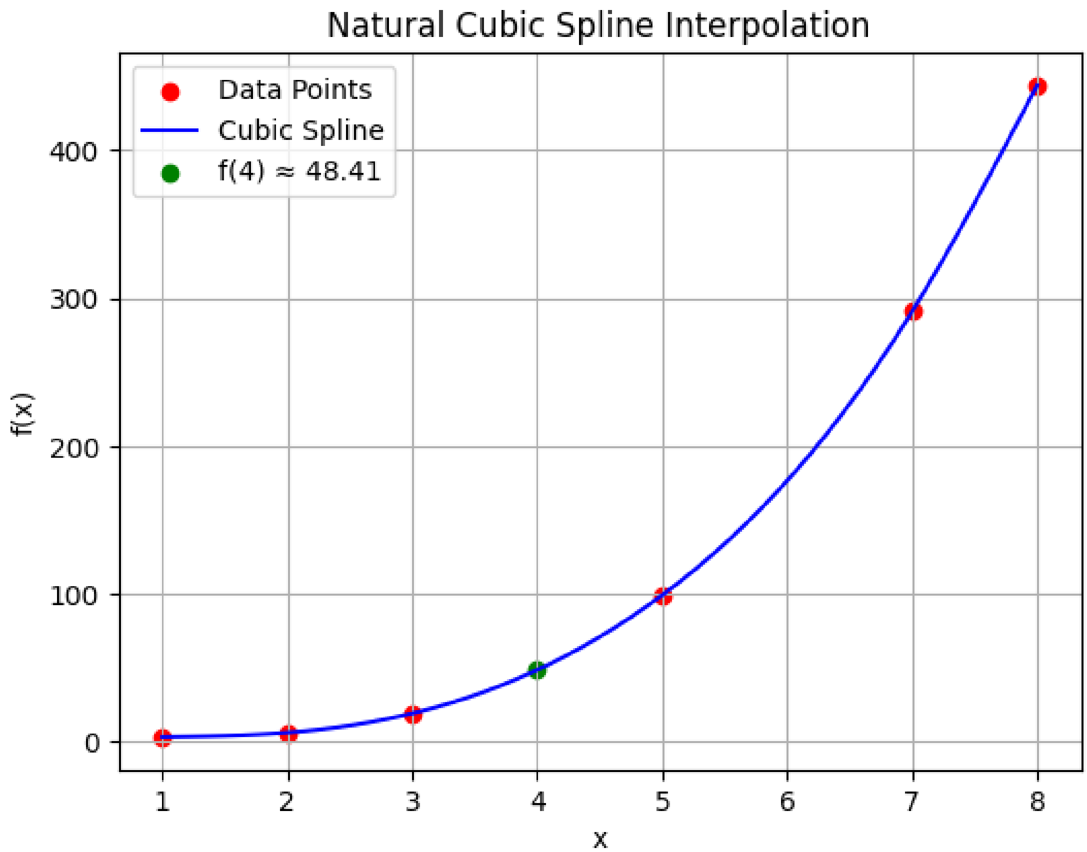
$$x_3 = 0.58 + \frac{-0.69382}{3.836 + 3.23}$$

$$x_3 \simeq 0.49$$

$\boxed{\text{So root is around } \underline{0.5}.}$

```python
#Q1 cubic spinals code
import numpy as np
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt

# Given data points
x_data = np.array([1, 2, 3, 5, 7, 8])
y_data = np.array([3, 6, 19, 99, 291, 444])

# Create a natural cubic spline interpolator
cubic_spline = CubicSpline(x_data, y_data, bc_type='natural')

# Estimate f(4) using cubic spline
f4_cubic_spline = cubic_spline(4)
print(f"Estimated f(4) using cubic spline: {f4_cubic_spline}")

# Generate values for plotting the spline curve
x_plot = np.linspace(min(x_data), max(x_data), 100)
y_plot = cubic_spline(x_plot)

# Plot the spline curve and original data points
plt.scatter(x_data, y_data, color='red', label='Data Points')
plt.plot(x_plot, y_plot, color='blue', label='Cubic Spline')
plt.scatter(4, f4_cubic_spline, color='green', label=f'f(4) ≈
{f4_cubic_spline:.2f}')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.legend()
plt.title('Natural Cubic Spline Interpolation')
plt.grid(True)
plt.show()
```

```
Estimated f(4) using cubic spline: 48.41157205240175
```




```
# q5

import numpy as np

def quadroot(r, s):
    """
    Solves a quadratic equation of the form x^2 + rx + s = 0.
    :param r: Coefficient of x
    :param s: Constant term
    :return: Real and imaginary parts of the roots
    """
    # Calculate the discriminant: r^2 - 4 * s
    discriminant = r**2 - 4*s

    # Case 1: Discriminant is positive -> two real roots
    if discriminant > 0:
        sqrt_disc = np.sqrt(discriminant)  # Square root of the
discriminant
        r1 = (r + sqrt_disc) / 2         # First real root
        r2 = (r - sqrt_disc) / 2         # Second real root
        i1 = i2 = 0                      # No imaginary part (i1 and
i2 are zero)
```

```
    # Case 2: Discriminant is zero -> one real root (repeated)
    elif discriminant == 0:
        r1 = r2 = -r / 2                    # Single real root (double 
root)
        i1 = i2 = 0                         # No imaginary part

    # Case 3: Discriminant is negative -> two complex roots
    else:
        real_part = -r / 2                  # Real part of the complex 
roots
        imag_part = np.sqrt(-discriminant) / 2  # Imaginary part (sqrt 
of negative)
        r1 = r2 = real_part                 # Real part is the same for 
both roots
        i1 = imag_part                      # Positive imaginary part
        i2 = -imag_part                     # Negative imaginary part

    # Return the roots (real and imaginary parts)
    return r1, i1, r2, i2


def bairstow(a, nn, es, rr, ss, maxit):
    """
    Bairstow's method to find real and complex roots of a polynomial.
    :param a: List of polynomial coefficients (highest degree first)
    :param nn: Degree of the polynomial
    :param es: Desired relative error for convergence
    :param rr: Initial guess for 'r'
    :param ss: Initial guess for 's'
    :param maxit: Maximum number of iterations allowed
    :return: Real and imaginary parts of the roots and error flag
    """

    # Initialize arrays to store intermediate values
    b = np.zeros(nn)  # Array 'b' for Bairstow's method calculations
    c = np.zeros(nn)  # Array 'c' for Bairstow's method calculations

    # Initial values of r and s (quadratic approximation)
    r = rr
    s = ss
    n = nn  # Polynomial degree (starts with the given degree)
    ier = 0  # Error flag (0 means no error)

    # Initialize relative error variables
    ea1 = ea2 = 1.0  # These will track the relative error for r and s

    iter = 0  # Initialize iteration counter

    # Loop until the solution converges or the maximum number of 
iterations is reached
```

```python
    while iter < maxit and (ea1 > es or ea2 > es):
        iter += 1  # Increment iteration counter

        # Step 1: Initialize the b and c arrays
        if n >= 2:
            b[n-1] = a[n-1]  # Last term of b is the same as a
            b[n-2] = a[n-2] + r * b[n-1]  # Second last term
        if n >= 3:
            c[n-1] = b[n-1]  # Last term of c is the same as b
            c[n-2] = b[n-2] + r * c[n-1]  # Second last term

        # Step 2: Fill the rest of b and c arrays by moving backward through the polynomial
        for i in range(n-3, -1, -1):
            b[i] = a[i] + r * b[i+1] + s * b[i+2]  # Formula for b values
            c[i] = b[i] + r * c[i+1] + s * c[i+2]  # Formula for c values

        # Step 3: Calculate determinants to adjust r and s
        if n >= 3:
            det = c[1]**2 - c[2] * c[0]  # Calculate determinant
            if abs(det) > 1e-12:
                # Update r and s using the calculated determinant
                dr = (2*b[0] * c[1] - b[1] * c[2]) / det
                ds = (2*b[1] * c[1] - b[0] * c[0]) / det
            else:
                dr = ds = 1.0  # Arbitrary value if determinant is too small
        else:
            dr = ds = 1.0  # Arbitrary update for low-degree polynomials

        # Step 4: Update r and s values
        r += dr
        s += ds

        # Step 5: Compute relative errors for r and s
        if abs(r) > 1e-12:
            ea1 = abs(dr / r) * 100  # Error for r
        if abs(s) > 1e-12:
            ea2 = abs(ds / s) * 100  # Error for s

    # Step 6: Use quadratic solver to find the roots from r and s
    re = []  # List to store real roots
    im = []  # List to store imaginary parts of roots

    # If the degree of the polynomial is 2, we can use the quadratic solver directly
    while n > 2:
```

```
        r1, i1, r2, i2 = quadroot(r, s)  # Solve the quadratic
equation
        re.extend([r1, r2])  # Store real parts of roots
        im.extend([i1, i2])  # Store imaginary parts of roots

        # Step 7: Divide the polynomial to find more roots
        coeffs = np.polydiv(a, [1, -r, s])[0]  # Polynomial division
        a = coeffs  # Update the polynomial coefficients
        n = len(a)  # Update the degree of the polynomial

        # If the degree becomes less than 2, break the loop
        if n <= 2:
            break

    # If the degree is 2, solve the quadratic equation
    if n == 2:
        r1, i1, r2, i2 = quadroot(a[1], a[0])
        re.extend([r1, r2])
        im.extend([i1, i2])

    # If the degree is 1 (linear), just solve for the single root
    elif n == 1:
        if len(a) > 1:  # Ensure there are enough coefficients
            re.extend([-a[0] / a[1]])  # Single real root
            im.extend([0])  # No imaginary part

    # If max iterations were reached, set the error flag
    if iter >= maxit:
        ier = 1  # Set error flag

    # Return the real roots, imaginary parts, and error flag
    return re, im, ier


# Define polynomials to test
polynomials = [
    ([0.7, -4, 6.2, -2], 3, 1e-6, -1, 0, 100),  # First polynomial
    ([-3.704, 16.3, -21.97, 9.34], 3, 1e-6, -1, 0, 100),  # Second
polynomial
    ([1, -2, 6, -2, 5], 4, 1e-6, -1, 0, 100)  # Third polynomial
(degree 4)
]

# Solve each polynomial
for i, (coeffs, nn, es, rr, ss, maxit) in enumerate(polynomials):
    print(f"Polynomial {i+1}:")
    roots, imaginary, ier = bairstow(coeffs, nn, es, rr, ss, maxit)
    print("Real roots:", roots)
    print("Imaginary parts:", imaginary)
```

```
    print("Error flag:", ier)
    print()
```

```
Polynomial 1:
Real roots: [3.0543237960216e+55, -2.671517012619503e+56, 0.0, -
1.65625924311214e+56]
Imaginary parts: [0, 0, 0, 0]
Error flag: 1

Polynomial 2:
Real roots: [3.0475166890291565e+55, -2.665563059027311e+56,
8.74444538902076e+56, 0.0]
Imaginary parts: [0, 0, 0, 0]
Error flag: 1

Polynomial 3:
Real roots: [nan, nan, nan, nan]
Imaginary parts: [nan, nan, nan, nan]
Error flag: 1


<ipython-input-40-6e3d7442d6b5>:87: RuntimeWarning: overflow
encountered in scalar multiply
  ds = (2*b[1] * c[1] - b[0] * c[0]) / det
<ipython-input-40-6e3d7442d6b5>:101: RuntimeWarning: invalid value
encountered in scalar divide
  ea2 = abs(ds / s) * 100  # Error for s
<ipython-input-40-6e3d7442d6b5>:86: RuntimeWarning: invalid value
encountered in scalar divide
  dr = (2*b[0] * c[1] - b[1] * c[2]) / det
<ipython-input-40-6e3d7442d6b5>:87: RuntimeWarning: invalid value
encountered in scalar subtract
  ds = (2*b[1] * c[1] - b[0] * c[0]) / det
```

```
#q6
import pandas as pd

def lagrange_interpolation(x, y, n, xx):
    """
    Calculate the Lagrange interpolation polynomial at a given xx.
    :param x: Array of known x points
    :param y: Array of known y points
    :param n: Number of known points
    :param xx: The x value where interpolation is needed
    :return: Interpolated value at xx
    """
    sum_ = 0.0  # Initialize sum
    for i in range(n):  # Loop over each known point
        product = y[i]  # Initialize product for current term
        for j in range(n):  # Calculate product for current term
```

```
            if i != j:
                product *= (xx - x[j]) / (x[i] - x[j])
        sum_ += product  # Add current term to sum
    return sum_  # Return the interpolated value

# Load the data from CSV file
data = pd.read_csv('/content/soil consolidation data.xlsx -
Table.csv')

# Extract time and settlement values
time_points = data['Time'].values
settlement_points = data['Settlement'].values

# Number of data points
n = len(time_points)

# Points to estimate
x1 = 6
x2 = 23

# Estimate settlements using Lagrange interpolation
settlement_at_x1 = lagrange_interpolation(time_points,
settlement_points, n, x1)
settlement_at_x2 = lagrange_interpolation(time_points,
settlement_points, n, x2)

print(f"Estimated settlement at {x1} days: {settlement_at_x1:.2f}")
print(f"Estimated settlement at {x2} days: {settlement_at_x2:.2f}")
```

```
Estimated settlement at 6 days: 2.97
Estimated settlement at 23 days: 11.49
```

```
#q6(ii)
import pandas as pd
from decimal import Decimal, getcontext

# Set precision for Decimal calculations
getcontext().prec = 50

# Function to find the product term for Newton's Divided Difference
def proterm(i, value, x):
    pro = Decimal(1)
    for j in range(i):
        pro *= (value - x[j])
    return pro

# Function to calculate the divided difference table
def dividedDiffTable(x, y, n):
    for i in range(1, n):
        for j in range(n - i):
            y[j][i] = (y[j][i - 1] - y[j + 1][i - 1]) / (x[j] - x[i +
```

```
j])
    return y

# Function to apply Newton's divided difference formula
def applyFormula(value, x, y, n):
    sum_ = y[0][0]
    for i in range(1, n):
        sum_ += proterm(i, value, x) * y[0][i]
    return sum_

# Function for Lagrange Interpolation
def lagrange_interpolation(x, y, n, xx):
    sum_ = Decimal(0)
    for i in range(n):
        product = Decimal(y[i])
        for j in range(n):
            if i != j:
                product *= (xx - x[j]) / (x[i] - x[j])
        sum_ += product
    return sum_

# Load the data from CSV file
data = pd.read_csv('/content/soil consolidation data.xlsx -
Table.csv')

# Extract time and settlement values and convert them
time_points = [Decimal(int(x)) for x in data['Time'].values]
settlement_points = [Decimal(float(y)) for y in
data['Settlement'].values]

# Number of data points
n = len(time_points)

# Initialize y dynamically based on the number of data points
y = [[Decimal(0) for _ in range(n)] for _ in range(n)]

# Populate the first column of y with settlement points
for i in range(n):
    y[i][0] = settlement_points[i]

# Calculate the divided difference table
y = dividedDiffTable(time_points, y, n)

# Values to be interpolated
value1 = Decimal(6)
value2 = Decimal(23)

# Apply Newton's Divided Difference
settlement_newton_value1 = applyFormula(value1, time_points, y, n)
settlement_newton_value2 = applyFormula(value2, time_points, y, n)
```

```
# Estimate settlements using Lagrange interpolation
settlement_lagrange_value1 = lagrange_interpolation(time_points,
settlement_points, n, value1)
settlement_lagrange_value2 = lagrange_interpolation(time_points,
settlement_points, n, value2)

# Print the results
print("Newton's Divided Difference:")
print(f"Estimated settlement at {value1} days:
{settlement_newton_value1:.20f}")
print(f"Estimated settlement at {value2} days:
{settlement_newton_value2:.20f}")

print("\nLagrange Interpolation:")
print(f"Estimated settlement at {value1} days:
{settlement_lagrange_value1:.20f}")
print(f"Estimated settlement at {value2} days:
{settlement_lagrange_value2:.20f}")
```

```
Newton's Divided Difference:
Estimated settlement at 6 days: 2.96617889404296875000
Estimated settlement at 23 days: 11.49437705427408218384

Lagrange Interpolation:
Estimated settlement at 6 days: 2.96617889404296875000
Estimated settlement at 23 days: 11.49437705427408218384
```

Newton’s Divided Difference Method

Advantages:

- Efficient for adding new data points.
- More numerically stable for small datasets.
- Incremental updates are straightforward.

Limitations:

- Computationally expensive for large datasets.
- Can overfit data.
- Sensitive to the order of data points.

Lagrange Interpolation

Advantages:

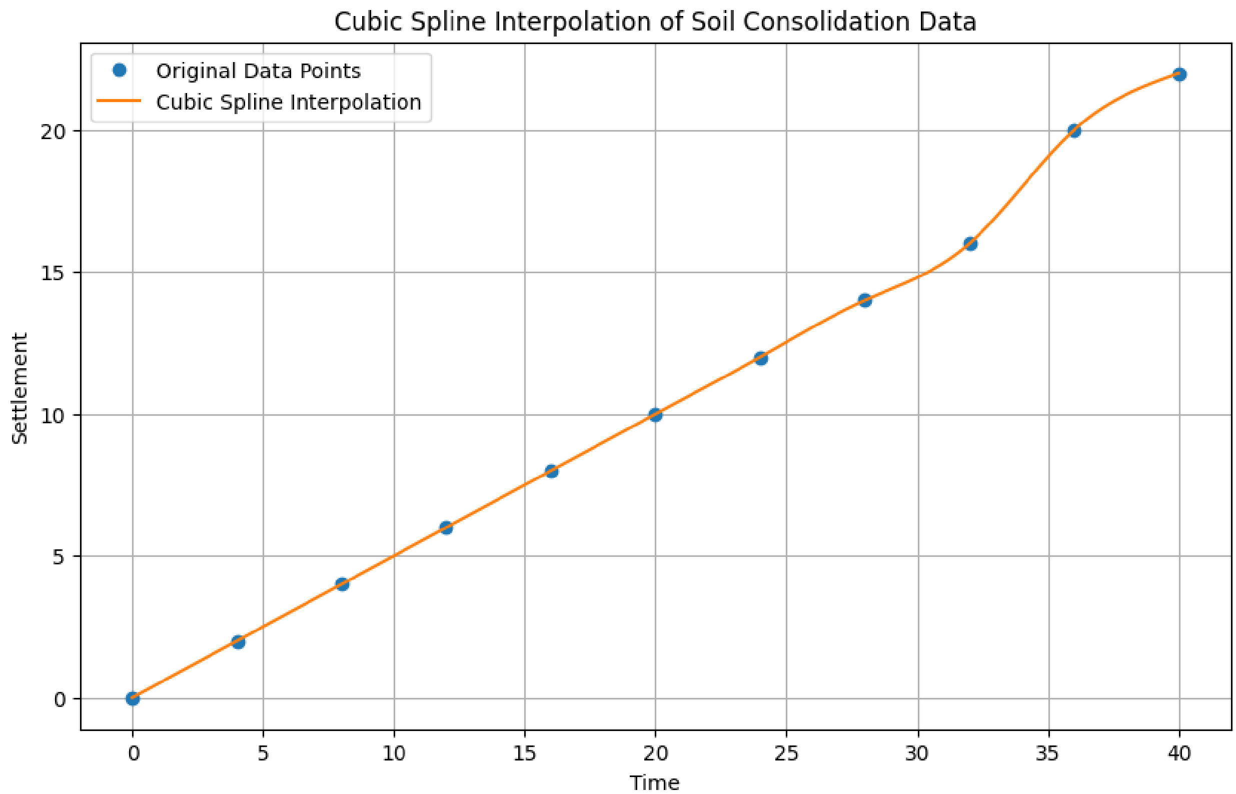
- Simple to implement and understand.
- Passes exactly through all provided points.

Limitations:

- Computationally intensive for large datasets.
- Can suffer from Runge's phenomenon (oscillations).
- High-degree polynomials may be numerically unstable.

Context for Soil Consolidation Data:

- **Newton's Method**: Best for incremental updates and smaller datasets.
- **Lagrange Interpolation**: Effective for small datasets but less efficient for larger ones.

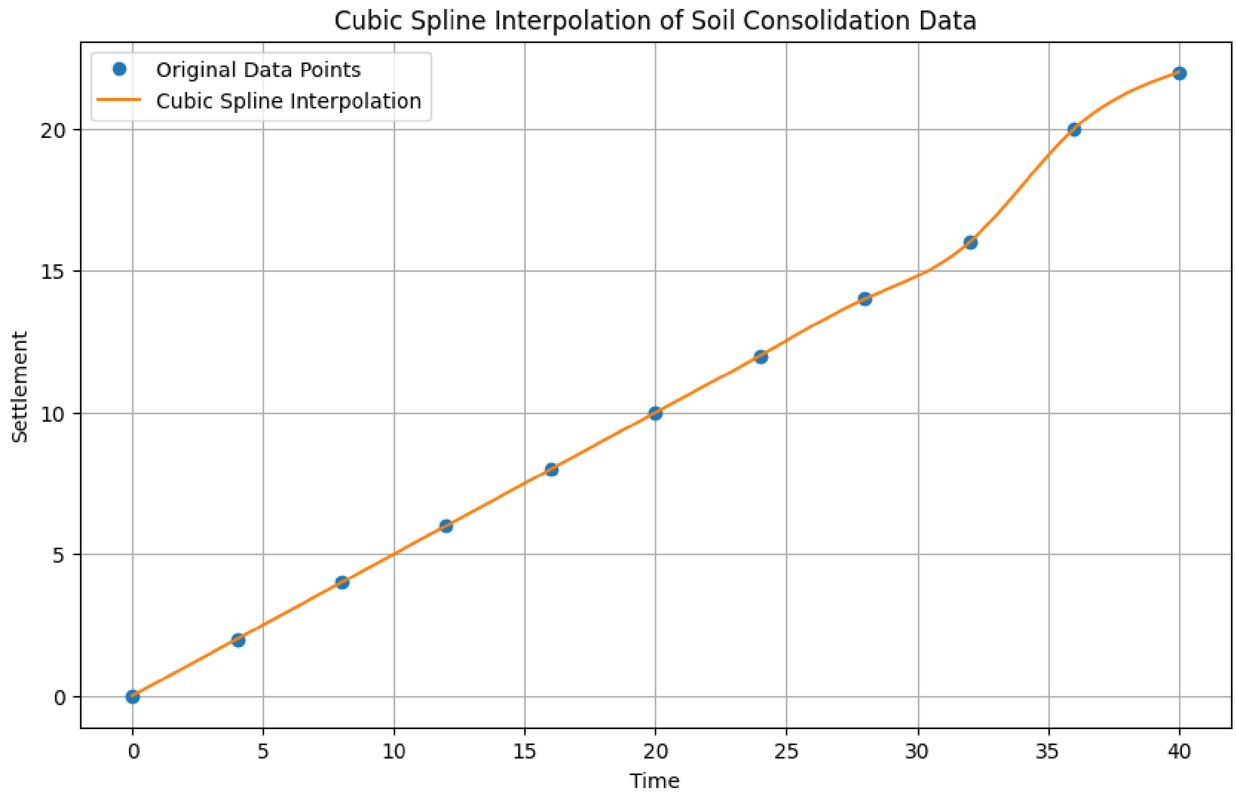



```
#q6(iii)&(iv)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline

# Read the CSV file
data = pd.read_csv('/content/soil consolidation data.xlsx - Table.csv')

# Extract time and settlement values
```

```python
time_points = data['Time'].values
settlement_points = data['Settlement'].values

# Define the tridiagonal matrix solver
def tridiag(x, y):
    n = len(x)
    e = np.zeros(n)
    f = np.zeros(n)
    g = np.zeros(n)
    r = np.zeros(n)

    e[0] = 0
    f[0] = 2 * (x[1] - x[0])
    g[0] = x[1] - x[0]
    r[0] = 6 * (y[1] - y[0]) / (x[1] - x[0])

    for i in range(1, n - 1):
        e[i] = x[i + 1] - x[i]
        f[i] = 2 * (x[i + 1] - x[i - 1])
        g[i] = x[i + 1] - x[i]
        r[i] = 6 * ((y[i + 1] - y[i]) / (x[i + 1] - x[i]) - (y[i] -
y[i - 1]) / (x[i] - x[i - 1]))

    e[n - 1] = x[n - 1] - x[n - 2]
    f[n - 1] = 2 * (x[n - 1] - x[n - 2])
    r[n - 1] = 6 * (y[n - 1] - y[n - 2]) / (x[n - 1] - x[n - 2])

    return e, f, g, r

def decomp(e, f, g, r):
    n = len(f)
    c = np.zeros(n)
    d = np.zeros(n)
    c[0] = g[0] / f[0]
    d[0] = r[0] / f[0]

    for i in range(1, n):
        denom = f[i] - e[i] * c[i - 1]
        c[i] = g[i] / denom
        d[i] = (r[i] - e[i] * d[i - 1]) / denom

    return c, d

def subst(c, d):
    n = len(d)
    d2x = np.zeros(n)
    d2x[-1] = d[-1]
    for i in range(n - 2, -1, -1):
        d2x[i] = d[i] - c[i] * d2x[i + 1]
```

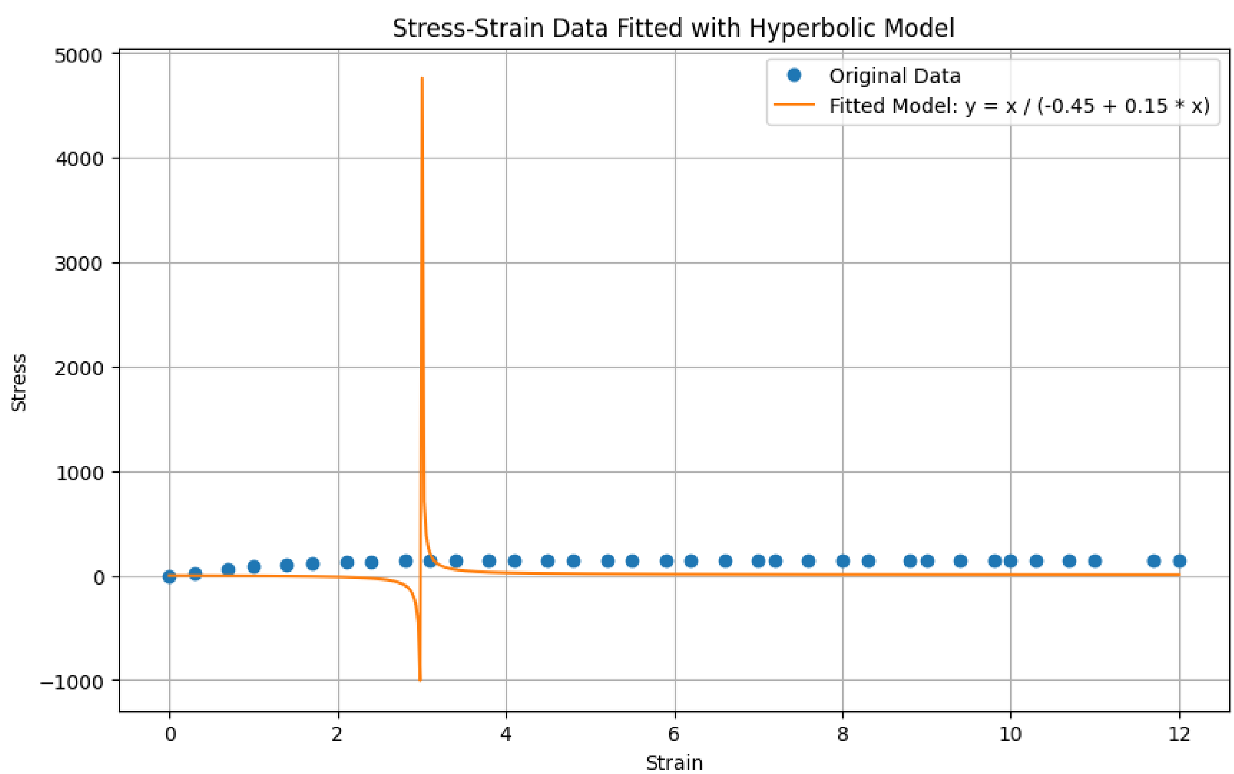
```
    return d2x

def interpol(x, y, d2x, xu):
    n = len(x)
    yu = np.zeros_like(xu)
    dy = np.zeros_like(xu)
    d2y = np.zeros_like(xu)

    for k in range(len(xu)):
        xu_val = xu[k]
        i = np.searchsorted(x, xu_val) - 1
        if i < 0: i = 0
        if i >= n - 1: i = n - 2

        h = x[i + 1] - x[i]
        a = (x[i + 1] - xu_val) / h
        b = (xu_val - x[i]) / h

        c1 = d2x[i] / 6
        c2 = d2x[i + 1] / 6
        c3 = y[i] / h
        c4 = y[i + 1] / h

        t1 = c1 * (a**3 - a**2) * h + c2 * (b**3 - b**2) * h
        t2 = c3 * a * h - c4 * b * h
        t3 = c4 * a - c3 * b
        t4 = c1 * (a**2) * h / 2 - c2 * (b**2) * h / 2

        yu[k] = t1 + t2 + t3 + t4
        dy[k] = 3 * (c1 * a * (a - 1) * h / 6 + c2 * b * (b - 1) * h /
6 + c3 * (b - a) + c4 * (a - b))
        d2y[k] = 6 * (c1 * a / h + c2 * b / h)

    return yu, dy, d2y

# Perform cubic spline interpolation using scipy
cs = CubicSpline(time_points, settlement_points, bc_type='natural')

# Define the range of values to interpolate
x_new = np.linspace(min(time_points), max(time_points), 500)
y_new = cs(x_new)

# Plot the data and cubic spline interpolation
plt.figure(figsize=(10, 6))
plt.plot(time_points, settlement_points, 'o', label='Original Data
Points')
plt.plot(x_new, y_new, '-', label='Cubic Spline Interpolation')
plt.xlabel('Time')
plt.ylabel('Settlement')
plt.title('Cubic Spline Interpolation of Soil Consolidation Data')
```

```
plt.legend()
plt.grid(True)
plt.show()
```




```
#q7

import pandas as pd
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Define the hyperbolic model function
def hyperbolic_model(x, a, b):
    return x / (a + b * x)

# Read the CSV file
data = pd.read_csv('/content/Stress-strain data.xlsx - Sheet1.csv')

# Extract strain and stress values
strain = data['Strain (%)'].values
stress = data['Stress(kPa)'].values

# Fit the model to the data
params, covariance = curve_fit(hyperbolic_model, strain, stress,
p0=[1, 1])
```

```python
# Extract the parameters
a, b = params

# Generate fitted values
strain_fit = np.linspace(min(strain), max(strain), 500)
stress_fit = hyperbolic_model(strain_fit, a, b)

# Plot the original data and the fitted curve
plt.figure(figsize=(10, 6))
plt.plot(strain, stress, 'o', label='Original Data')
plt.plot(strain_fit, stress_fit, '-', label=f'Fitted Model: y = x /
({a:.2f} + {b:.2f} * x)')
plt.xlabel('Strain')
plt.ylabel('Stress')
plt.title('Stress-Strain Data Fitted with Hyperbolic Model')
plt.legend()
plt.grid(True)
plt.show()

# Print the estimated parameters
print(f"Estimated parameters:\n a = {a:.2f}\n b = {b:.2f}")
```



```
Estimated parameters:
 a = -0.45
 b = 0.15

import numpy as np
import pandas as pd

def polynomial_regression(data_file, degree):
    # Step 1: Input Order of Polynomial to Fit, m
    m = degree

    # Step 2: Read the CSV file
    data = pd.read_csv(data_file)
    x = data['Strain (%)'].values
    y = data['Stress(kPa)'].values
    n = len(x)  # Number of data points

    # Step 3: Check Feasibility
    if n <= m:
        print("Error: Not enough data points for the specified
polynomial degree.")
        return

    # Initialize matrices for normal equations
    A = np.zeros((m + 1, m + 1))
    B = np.zeros(m + 1)

    # Step 4: Compute Elements of the Normal Equation
    # Calculate elements of matrix A
    for i in range(m + 1):
        for j in range(i + 1):
            k = i + j
            sum_A = np.sum(x**k)
            A[i, j] = sum_A
            A[j, i] = sum_A  # Matrix A is symmetric

    # Calculate elements of vector B
    for i in range(m + 1):
        sum_B = np.sum(y * x**i)
        B[i] = sum_B

    # Step 5: Solve for Coefficients
    # Solve the normal equation A * coefficients = B
    coefficients = np.linalg.solve(A, B)

    # Step 6: Print Coefficients
    print(f"Polynomial coefficients (degree {m}): {coefficients}")

# Example usage
```

```
polynomial_regression('/content/Stress-strain data.xlsx - Sheet1.csv',
3)
```

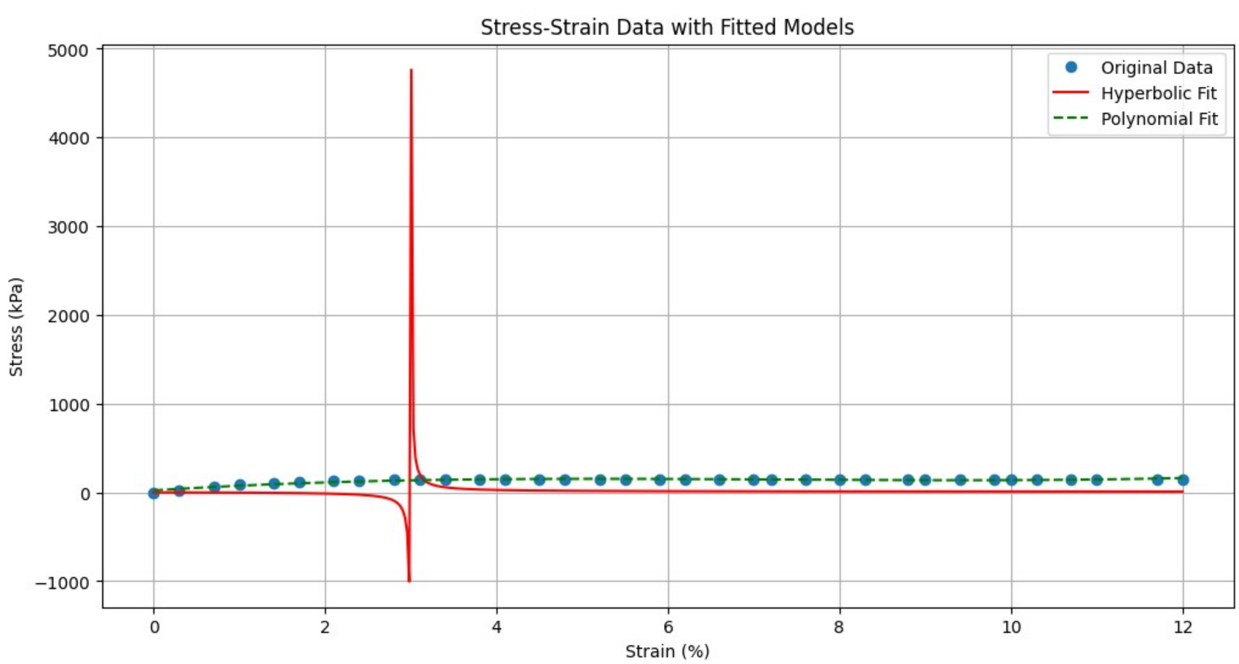
```
Polynomial coefficients (degree 3): [22.2529772  61.75478487 -
9.23744597  0.42183993]
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Define the hyperbolic model function
def hyperbolic_model(x, a, b):
    return x / (a + b * x)

# Polynomial regression function
def polynomial_regression(data_file, degree):
    # Read the CSV file
    data = pd.read_csv(data_file)
    x = data['Strain (%)'].values
    y = data['Stress(kPa)'].values
    n = len(x)  # Number of data points

    # Check feasibility
    if n <= degree:
        print("Error: Not enough data points for the specified
polynomial degree.")
        return None

    # Initialize matrices for normal equations
    A = np.zeros((degree + 1, degree + 1))
    B = np.zeros(degree + 1)

    # Compute elements of matrix A
    for i in range(degree + 1):
        for j in range(i + 1):
            k = i + j
            sum_A = np.sum(x**k)
            A[i, j] = sum_A
            A[j, i] = sum_A  # Matrix A is symmetric

    # Compute elements of vector B
    for i in range(degree + 1):
        sum_B = np.sum(y * x**i)
        B[i] = sum_B

    # Solve for coefficients
    coefficients = np.linalg.solve(A, B)
    return coefficients
```

```
# Function to evaluate polynomial at given points
def evaluate_polynomial(coefficients, x):
    return sum(c * x**i for i, c in enumerate(coefficients))

# Compute standard error of the estimate
def standard_error(y, y_fit):
    residuals = y - y_fit
    return np.sqrt(np.sum(residuals**2) / (len(y) - len(y_fit)))

# Compute correlation coefficient
def correlation_coefficient(y, y_fit):
    return np.corrcoef(y, y_fit)[0, 1]

# Main function to plot the data, fitted models, and compute errors
def plot_fitted_models(data_file, poly_degree):
    # Read the CSV file
    data = pd.read_csv('/content/Stress-strain data.xlsx -
Sheet1.csv')
    x = data['Strain (%)'].values
    y = data['Stress(kPa)'].values

    # Fit the hyperbolic model
    popt, _ = curve_fit(hyperbolic_model, x, y, p0=[1, 1])
    a, b = popt

    # Fit the polynomial model
    poly_coefficients = polynomial_regression(data_file, poly_degree)
    if poly_coefficients is None:
        return

    # Generate x values for plotting
    x_fit = np.linspace(min(x), max(x), 500)
    y_hyperbolic = hyperbolic_model(x_fit, *popt)
    y_polynomial = evaluate_polynomial(poly_coefficients, x_fit)

    # Compute fitted values for the original x data points
    y_hyperbolic_fit = hyperbolic_model(x, *popt)
    y_polynomial_fit = evaluate_polynomial(poly_coefficients, x)

    # Calculate errors and correlation coefficients
    se_hyperbolic = standard_error(y, y_hyperbolic_fit)
    se_polynomial = standard_error(y, y_polynomial_fit)
    r_hyperbolic = correlation_coefficient(y, y_hyperbolic_fit)
    r_polynomial = correlation_coefficient(y, y_polynomial_fit)

    # Print results
    print(f"Hyperbolic Model - Standard Error: {se_hyperbolic:.2f},
Correlation Coefficient: {r_hyperbolic:.2f}")
    print(f"Polynomial Model - Standard Error: {se_polynomial:.2f},
Correlation Coefficient: {r_polynomial:.2f}")
```

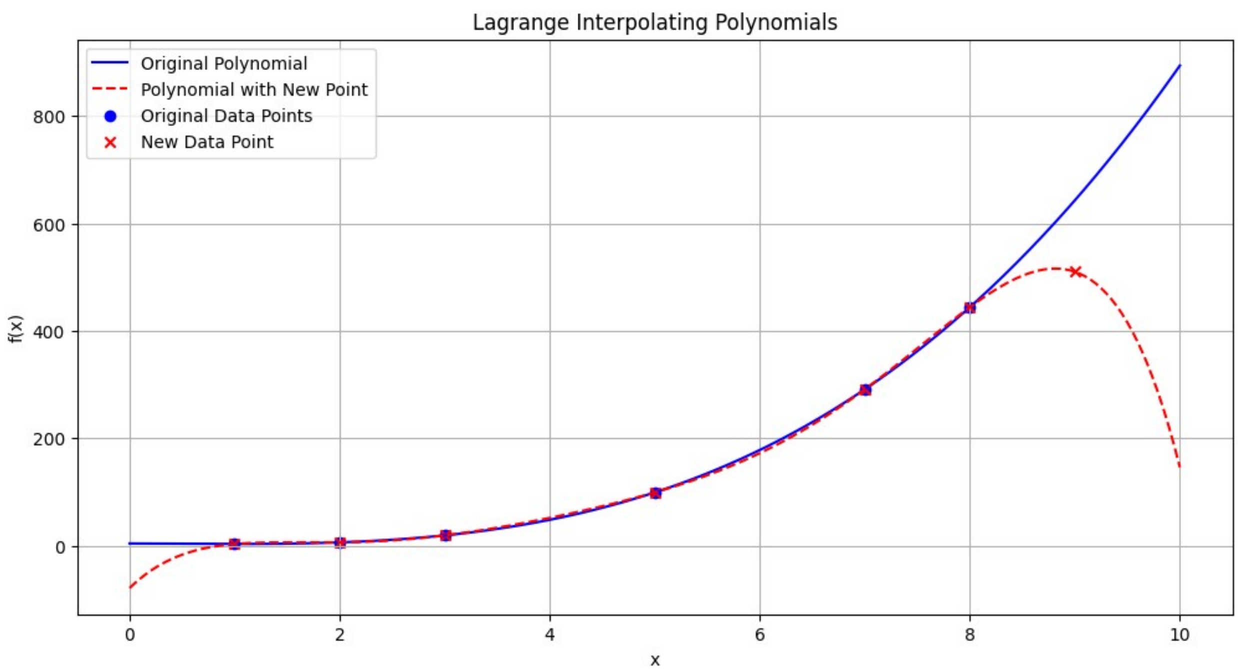
```python
    # Plot the original data and the fitted models
    plt.figure(figsize=(12, 6))
    plt.plot(x, y, 'o', label='Original Data')
    plt.plot(x_fit, y_hyperbolic, '-', label='Hyperbolic Fit',
color='red')
    plt.plot(x_fit, y_polynomial, '--', label='Polynomial Fit',
color='green')
    plt.xlabel('Strain (%)')
    plt.ylabel('Stress (kPa)')
    plt.title('Stress-Strain Data with Fitted Models')
    plt.legend()
    plt.grid(True)
    plt.show()

# Example usage with your CSV file
if __name__ == "__main__":
    plot_fitted_models('/content/Stress-strain data.xlsx -
Sheet1.csv', 3)
```

```
<ipython-input-36-33648f985a5d>:51: RuntimeWarning: divide by zero
encountered in scalar divide
  return np.sqrt(np.sum(residuals**2) / (len(y) - len(y_fit)))

Hyperbolic Model - Standard Error: inf, Correlation Coefficient: 0.14
Polynomial Model - Standard Error: inf, Correlation Coefficient: 0.96
```




```python
# Q1 explainer
```

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

def lagrange_basis(x, x_points, i):
    basis = 1
    for j in range(len(x_points)):
        if j != i:
            basis *= (x - x_points[j]) / (x_points[i] - x_points[j])
    return basis

def lagrange_polynomial(x_points, y_points):
    def polynomial(x):
        total = 0
        for i in range(len(x_points)):
            total += y_points[i] * lagrange_basis(x, x_points, i)
        return total
    return polynomial

# Original data
x_points = np.array([1, 2, 3, 5, 7, 8])
y_points = np.array([3, 6, 19, 99, 291, 444])

# New data point
x_points_new = np.append(x_points, 9)
y_points_new = np.append(y_points, 510)

# Define and evaluate polynomials
poly_original = lagrange_polynomial(x_points, y_points)
poly_new = lagrange_polynomial(x_points_new, y_points_new)

x = np.linspace(0, 10, 400)
y_original = poly_original(x)
y_new = poly_new(x)

# Plot
plt.figure(figsize=(12, 6))
plt.plot(x, y_original, label='Original Polynomial', color='blue')
plt.plot(x, y_new, label='Polynomial with New Point', color='red',
linestyle='--')
plt.scatter(x_points, y_points, color='blue', marker='o',
label='Original Data Points')
plt.scatter(x_points_new, y_points_new, color='red', marker='x',
label='New Data Point')
plt.legend()
plt.title('Lagrange Interpolating Polynomials')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)
plt.show()
```




```
Natural Cubic Spline Interpolation at x = 4: 48.41157205240175
```