CHAPTER 3: INTERPOLATION

PRESENTED BY:
DESI NOFITASARI | 24083010058
DIVA ANGGRAENI | 24083010065
SITI RANIA AZARIA | 24083010072

Table of Contents

Definition

what is interpolation?

Interpolation is the process of finding a function f(x) that passes through all data points (xi, yi) to estimate the values between those points.

Polynomial Interpolation

$$Pn\left(x\right) = \sum_{k=0}^{n} a_k \phi_k \left(x\right)$$

what is polynomial interpolation?

Polynomial interpolation is a method for finding a polynomial Pn(x) that passes through all data points (x0,y0), (x1,y1),...,(xn,yn).

components

general form → overall structure of the polynomial.

basis function → basic pattern to form the polynomial.

Monomial Form

represents the polynomial as a combination of powers of x.

basis form

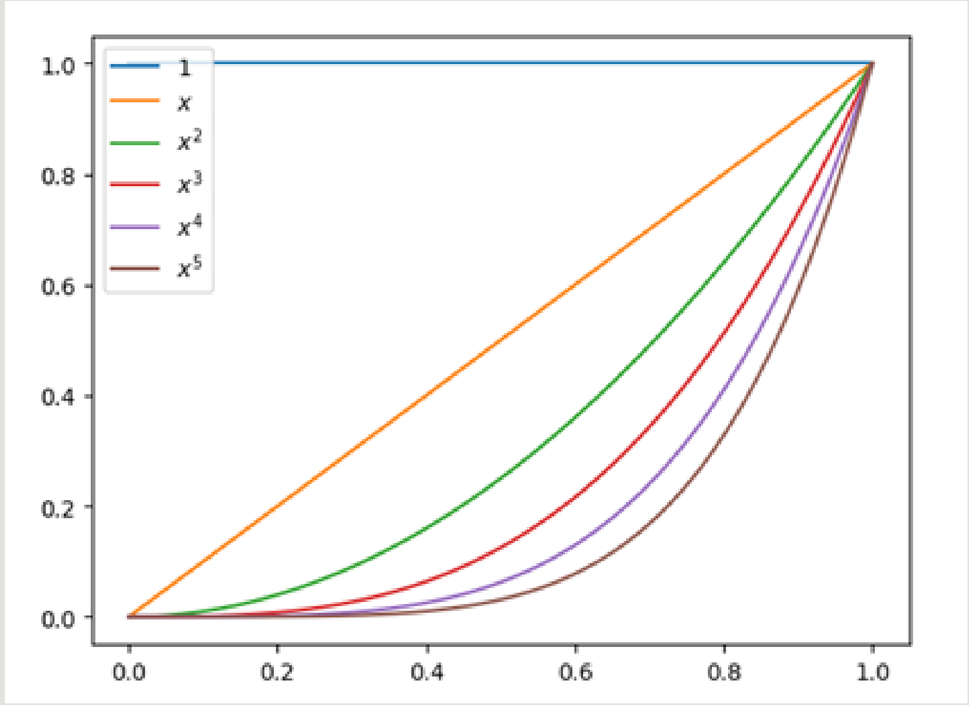
$$\phi_k(x) = x^k$$

represents x raised to the k-th power.

general form

$$P_n(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

Each coefficient → how strongly its power of x affects polynomial.



higher powers bend more sharply.

Monomial Form

monomial interpolation

```
import numpy as np
A =np.array([[1,-1,1], [1,1,1], [1,2,4]])
A
```

define matrix (A)

```
array([[ 1, -1,  1],
       [ 1,  1,  1],
       [ 1,  2,  4]])
```

```
y = np.array([-6, 0, 6])
y
```

define data vector (y)

```
array([-6,  0,  6])
```

```
 np.linalg.solve(A, y)
```

solution

```
array([-4.,  3.,  1.])
```

$$a_0 = -4 \qquad a_1 = 3 \qquad a_2 = 1$$

$$P_2(x) = a_0 + a_1 x + a_2 x^2 \quad \rightarrow \quad P_2(x) = -4 + 3x + x^2$$

Lagrange Form

directly from data points without solving a system of equations.

basis form

$$\phi_k(x) = L_k(x) = \prod_{j=0, j\neq k}^{n} \left(\frac{x - x_j}{x_k - x_j}\right)$$

builds each Lk(x) using all data points except Xk

general form

$$P_n(x) = \sum_{k=0}^{n} y_k L_k(x)$$

combines all Lk(x) weighted by Yk

Lagrange Form

function code:

The input points are (-1, -6), (1, 0), and (2, 6). Since three points are supplied, the computer uses Lagrange's method to find the single best quadratic curve that perfectly hits all of them.

```python
import numpy as np
from scipy.interpolate import lagrange
import matplotlib.pyplot as plt

# data points
x = np.array([-1., 1., 2.])
y = np.array([-6., 0., 6.])

# Construct the Lagrange Polynomial
P_lagrange = lagrange(x, y)

c = P_lagrange.coefficients
print(f"Lagrange Polynomial Coefficients (c2, c1, c0): {c}")
print(f"Verification of P(x) at data points: {P_lagrange(x)}")

print(f"Simplified Lagrange Polynomial Result:")
print(f"P(x) = {c[2]:.0f} + {c[1]:.0f}x + {c[0]:.0f}x^2")

# plot
plt.figure(figsize=(9, 6)) # Set the figure size
x_plot = np.linspace(np.min(x) - 1, np.max(x) + 1, 200)
y_plot = P_lagrange(x_plot)

# Plot the interpolating polynomial
plt.plot(x_plot, y_plot, label='Lagrange Interpolating Polynomial $P(x)$', color='blue', linewidth=2)
```

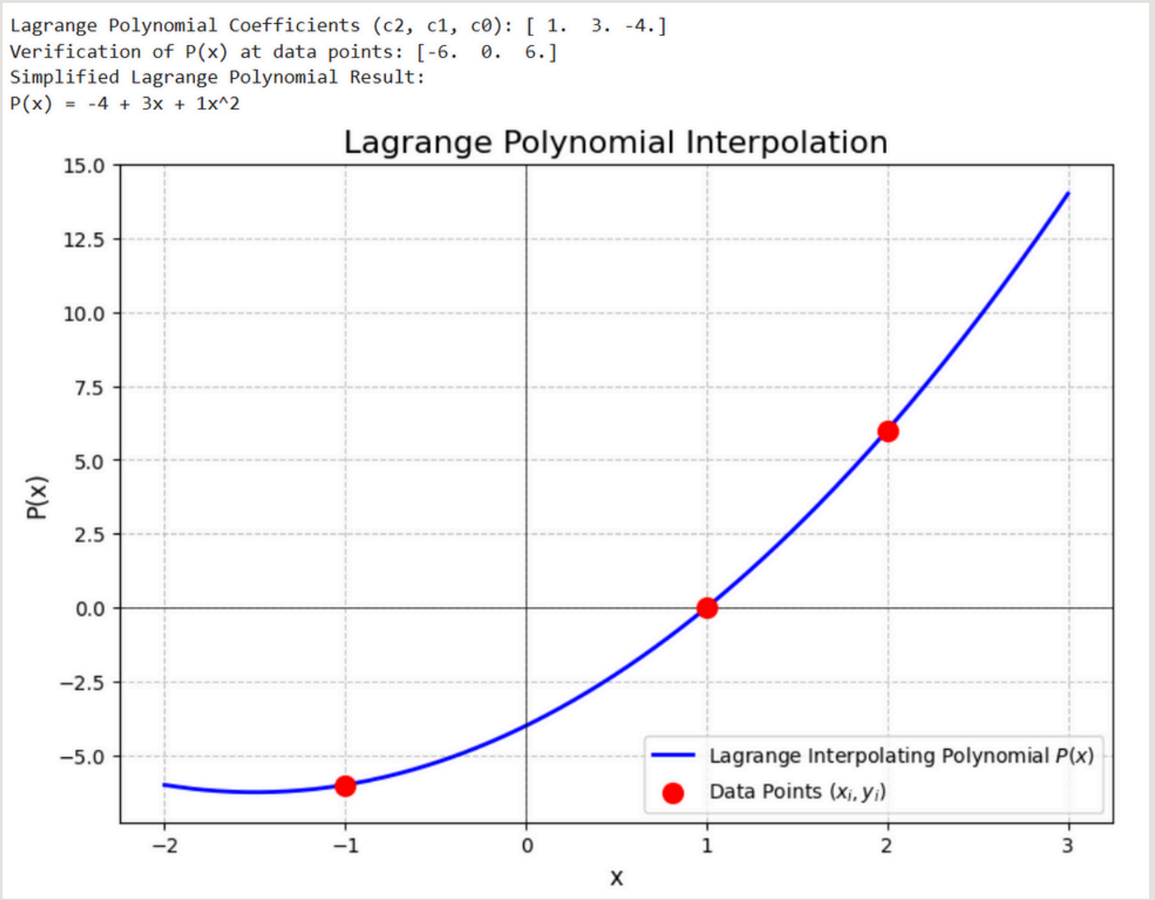
```python
# Plot the original data points
plt.scatter(x, y, color='red', marker='o', s=100, zorder=5, label='Data Points $(x_i, y_i)$')

plt.title('Lagrange Polynomial Interpolation', fontsize=16)
plt.xlabel('x', fontsize=12)
plt.ylabel('P(x)', fontsize=12)
plt.grid(True, linestyle='--', alpha=0.7)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.legend(fontsize=10)
plt.show()
```

Lagrange Form

output:

From the output confirms the final result is P(x) = x^2 + 3x - 4. This is the exact same polynomial previously found using the monomial method. The blue graph visually demonstrates that this curve is smooth and accurately hits all three red points (our input data).

```
Lagrange Polynomial Coefficients (c2, c1, c0): [ 1.  3. -4.]
Verification of P(x) at data points: [-6.  0.  6.]
Simplified Lagrange Polynomial Result:
P(x) = -4 + 3x + 1x^2
```



Newton Form

basis form

$$\phi_k(x) = \prod_{j=0}^{k-1}(x - x_j)$$

Each $\phi_k(x)$ is formed by multiplying $(x - x_j)$ for previous points.

general form

$$P_n(x) = \sum_{k=0}^{n} a_k \pi_k(x)$$

Combines all $\phi_k(x)$ weighted by divided difference coefficients a_k.

(explicit)

$$P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + \ldots + a_n(x - x_0)(x - x_1)\ldots(x - x_{n-1})$$

Newton Form

```
def diff(x, y):
    m = x.size
    a = np.zeros(m)
    for i in range(m):
        a[i] = y[i]
    for j in range(1, m):
        for i in np.flip(np.arange(j,m)):
            a[i] = (a[i]-a[i-1]) / (x[i]-x[i-(j)])
    return a

diff(np.array([-1,1,2]), np.array([-6,0,6]))
```

```
array([-6.,  3.,  1.])
```

```
diff(np.array([1.765,1.760,1.755,1.750]),
 np.array([0.92256,0.92137,0.92021,0.91906]))
```

```
array([ 0.92256   ,  0.238     ,  0.6       , 26.66666667])
```

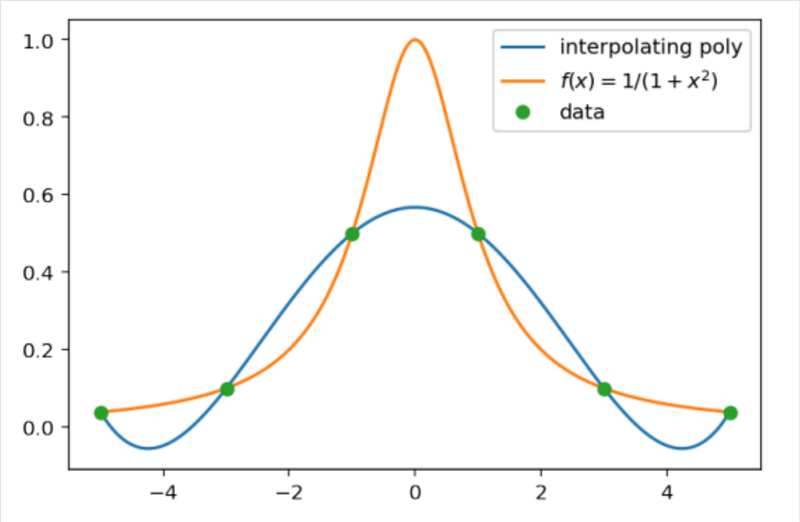
```
def newton(x, y, z):
    m = x.size # here m is the number of data points, not the degree
    # of the polynomial
    a = diff(x, y)
    sum = a[0]
    pr = 1.0
    for j in range(m-1):
        pr *= (z-x[j])
        sum += a[j+1]*pr
    return sum

newton(np.array([1.765,1.760,1.755,1.750]),
 np.array([0.92256,0.92137,0.92021,0.91906]), 1.761)
```

```
np.float64(0.92160496)
```

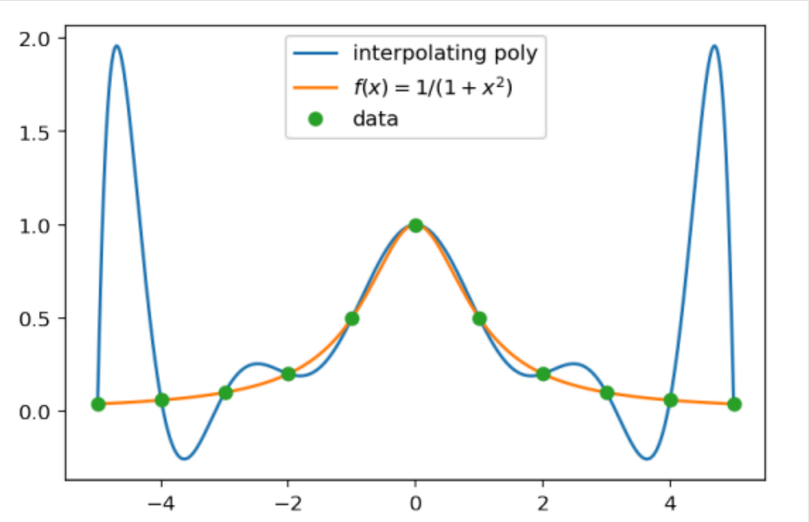
diff computes the divided differences (coefficients a). newton then uses these coefficients to efficiently evaluate the polynomial at a point z.

High Degree Polynomial Interpolation

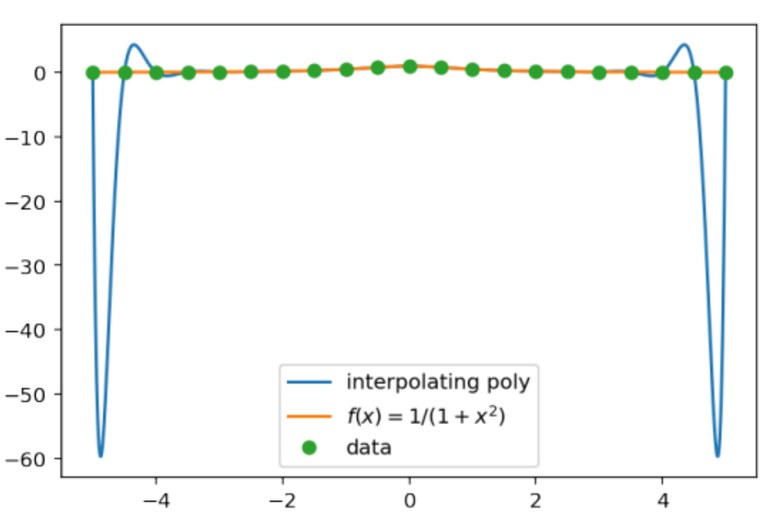
High-degree polynomial interpolation uses large-degree polynomials, creating complex curves but causing endpoint oscillations and instability despite fitting all data points.



Degree 6

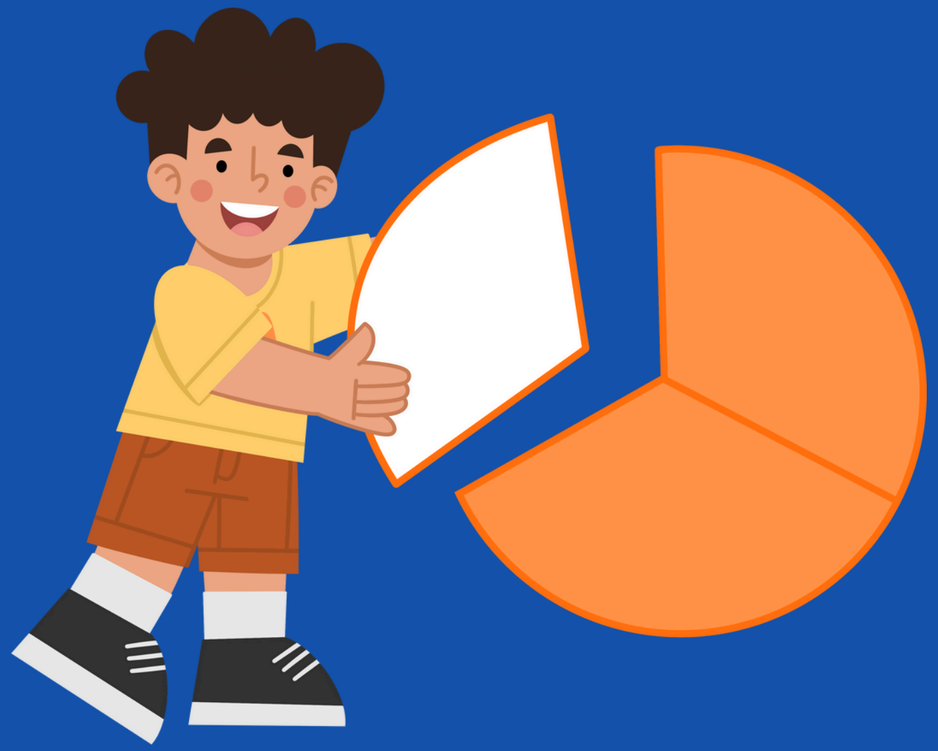


Degree 11



Degree 21

Hermite Interpolation



what is hermite interpolation?

HERMITE INTERPOLATION IS A METHOD THAT CONSTRUCTS A POLYNOMIAL THAT MATCHES BOTH THE FUNCTION VALUES AND THEIR DERIVATIVES AT GIVEN POINTS. BY INCORPORATING SLOPE INFORMATION, HERMITE INTERPOLATION PRODUCES SMOOTHER AND MORE ACCURATE CURVES THAN STANDARD INTERPOLATION METHODS. IT IS ESPECIALLY USEFUL WHEN THE BEHAVIOR OF THE FUNCTION IS INFLUENCED BY BOTH ITS VALUE AND RATE OF CHANGE AT EACH DATA POINT.

Hermite Interpolation

Numerik Code :

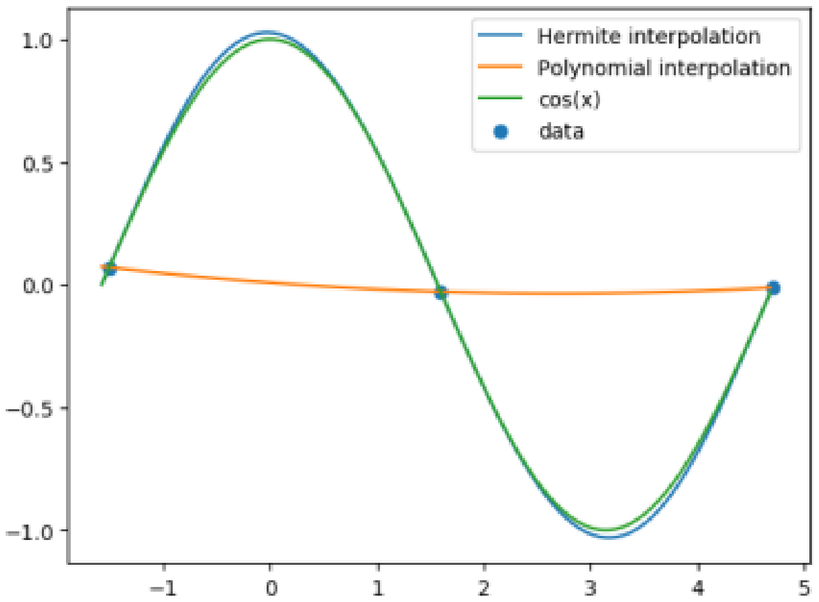


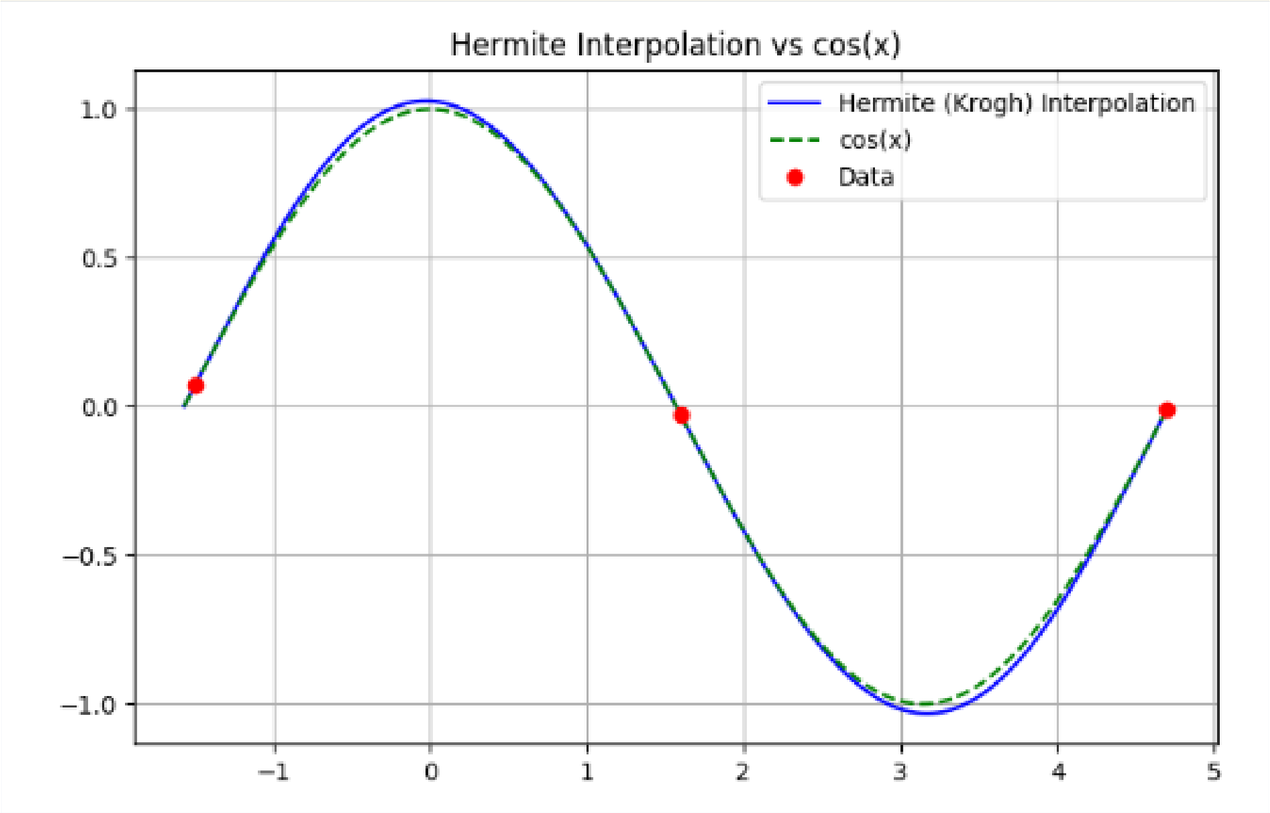
Figure 3.3

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

# Function to compute Hermite divided differences
def hdiff(x, y, yprime):
    m = x.size  # number of data points
    l = 2 * m
    z = np.zeros(l)
    a = np.zeros(l)

    # Duplicate x values for Hermite interpolation
    for i in range(m):
        z[2 * i] = x[i]
        z[2 * i + 1] = x[i]

    # Duplicate y values
    for i in range(m):
        a[2 * i] = y[i]
        a[2 * i + 1] = y[i]

    # Compute the first divided differences (using derivatives)
    for i in np.flip(np.arange(1, m)):
        a[2 * i + 1] = yprime[i]
        a[2 * i] = (a[2 * i] - a[2 * i - 1]) / (z[2 * i] - z[2 * i - 1])

    a[1] = yprime[0]

    # Compute the higher-order divided differences
    for j in range(2, l):
        for i in np.flip(np.arange(j, l)):
            a[i] = (a[i] - a[i - 1]) / (z[i] - z[i - j])

    return a
```

```python
# Main function for Hermite interpolation
def hermite(x, y, yprime, w):
    m = x.size
    a = hdiff(x, y, yprime)
    z = np.zeros(2 * m)

    # Duplicate x values
    for i in range(m):
        z[2 * i] = x[i]
        z[2 * i + 1] = x[i]

    total = a[0]
    pr = 1.0
    for j in range(2 * m - 1):
        pr *= (w - z[j])
        total += a[j + 1] * pr

    return total

# Example usage
xaxis = np.linspace(-np.pi / 2, 3 * np.pi / 2, 120)
x = np.array([-1.5, 1.6, 4.7])
y = np.array([0.071, -0.029, -0.012])
yprime = np.array([1, -1, 1])

funct = np.cos(xaxis)
interp = hermite(x, y, yprime, xaxis)

plt.plot(xaxis, interp, label='Hermite Interpolation')
plt.plot(xaxis, funct, label='cos(x)')
plt.plot(x, y, 'o', label='Data')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```

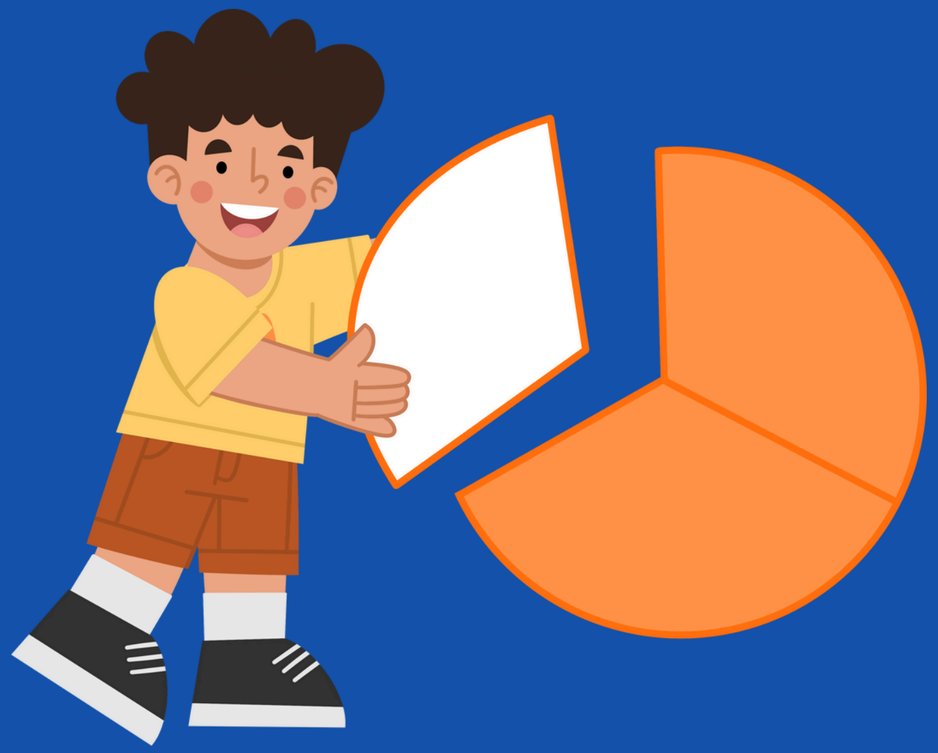
Hermite Interpolation

Instan code :




```python
import numpy as np
from scipy.interpolate import KroghInterpolator
import matplotlib.pyplot as plt

# Data points and function values
x = np.array([-1.5, 1.6, 4.7])
y = np.array([0.071, -0.029, -0.012])
yprime = np.array([1, -1, 1])  # derivative values at each point

# Build combined data of x and y together with derivatives
x_all = np.repeat(x, 2)  # each point repeated twice
y_all = np.ravel(np.column_stack([y, yprime]))

# Create Hermite interpolator (KroghInterpolator)
interp = KroghInterpolator(x_all, y_all)

# Evaluate at new points
x_new = np.linspace(-np.pi/2, 3*np.pi/2, 200)
y_new = interp(x_new)

# Original function for comparison (for example cos(x))
funct = np.cos(x_new)

# Plot interpolation result and the original function
plt.figure(figsize=(8,5))
plt.plot(x_new, y_new, 'b-', label='Hermite (Krogh) Interpolation')
plt.plot(x_new, funct, 'g--', label='cos(x)')
plt.plot(x, y, 'ro', label='Data')
plt.title('Hermite Interpolation vs cos(x)')
plt.legend()
plt.grid(True)
plt.show()
```

Hermite Interpolation



Numerical vs Instant Hermite Interpolation

FROM THE COMPARISON OF THE CODES TESTED, BOTH APPROACHES TO HERMITE INTERPOLATION PRODUCE THEORETICALLY IDENTICAL CURVES, BUT DIFFER IN STABILITY. THE NUMERICAL (MANUAL) CODE BUILDS THE HERMITE POLYNOMIAL STEP-BY-STEP USING DIVIDED DIFFERENCES, WHICH HELPS SHOW THE UNDERLYING MATH BUT IS MORE PRONE TO MISTAKES AND NUMERICAL ISSUES. IN CONTRAST, THE INSTANT SCIPY IMPLEMENTATION COMPUTES THE POLYNOMIAL AUTOMATICALLY, MAKING IT MORE STABLE, FASTER, AND LESS ERROR-PRONE. OVERALL, BOTH METHODS AIM FOR THE SAME RESULT, BUT THE SCIPY VERSION IS MORE RELIABLE IN PRACTICE.

Piecewise Polynomials: Spline Interpolation

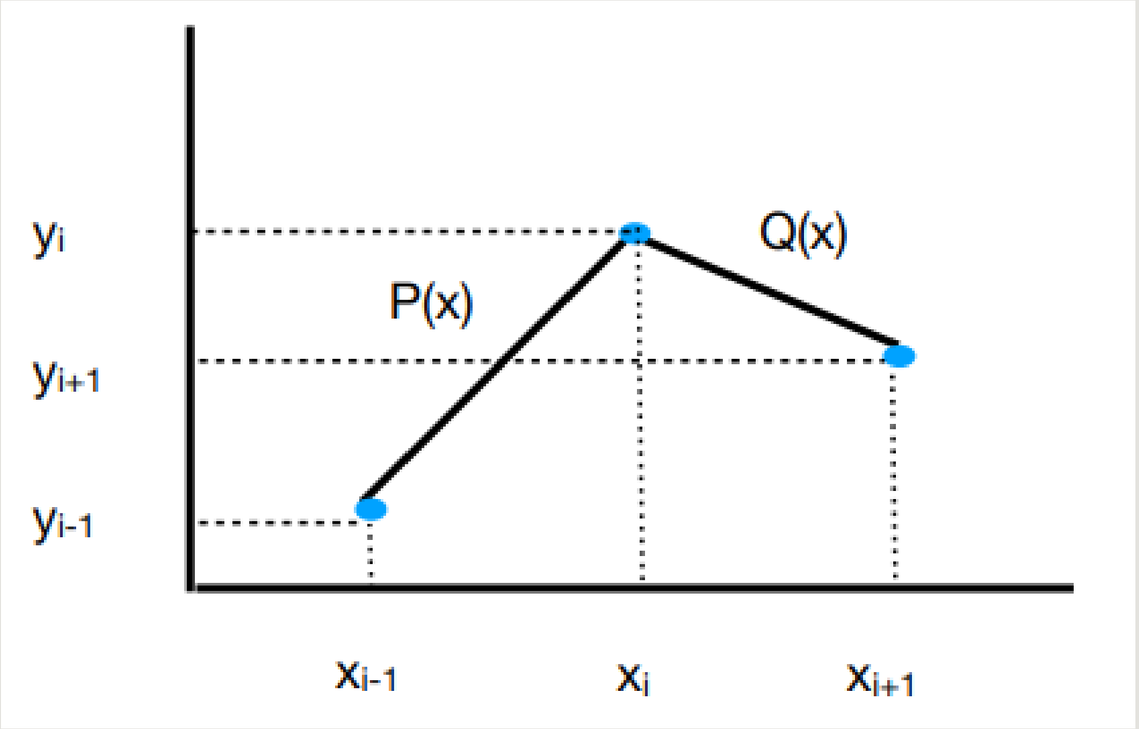
what is spline interpolation?

Spline interpolation is an interpolation method that divides the data domain into several intervals, then in each interval a low-degree polynomial is created.

Linear Spline

Linear splines connect points with straight lines, but are less smooth because their derivatives are not continuous.

$$P_{(x)} = ax + b$$



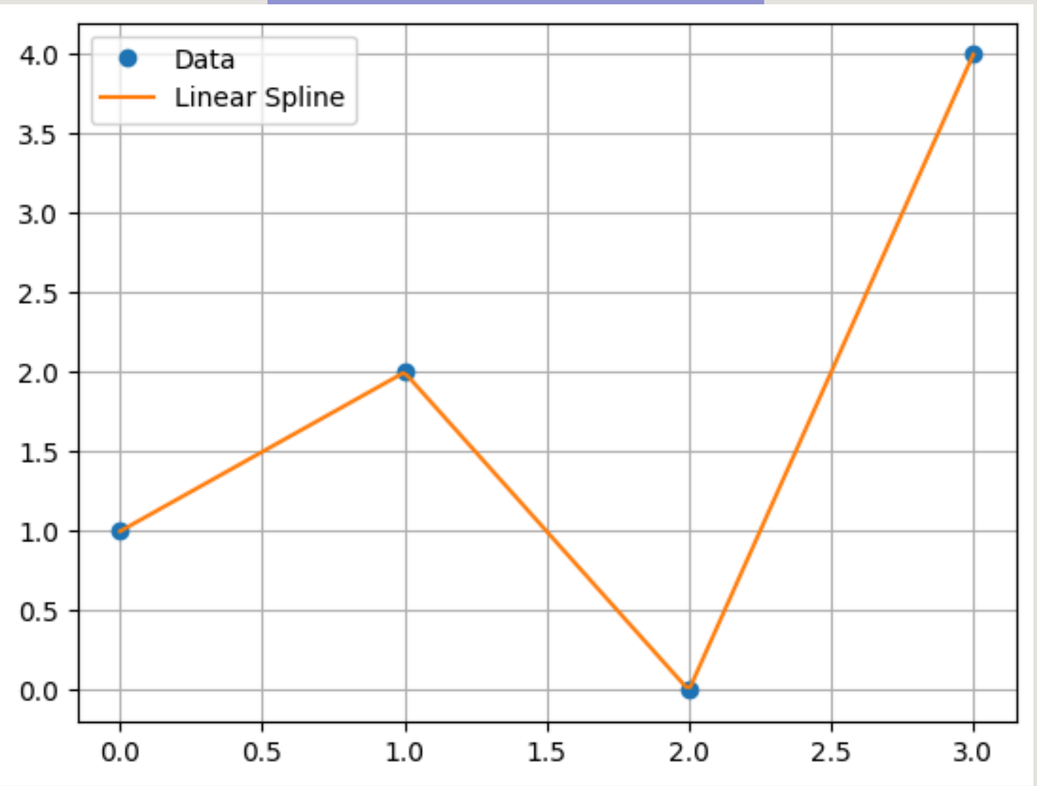
P(x) and Q(x) connect three data points. The derivative is discontinuous at the central point (xi, yi)

Linear Spline

INPUT

```
## LINEAR SPLINE
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# data example
x = np.array([0, 1, 2, 3])
y = np.array([1, 2, 0, 4])

linear_spline = interp1d(x, y, kind='linear')

xx = np.linspace(0, 3, 200)
plt.plot(x, y, 'o', label='Data')
plt.plot(xx, linear_spline(xx), label='Linear Spline')
plt.legend()
plt.grid(True)
plt.show()
```

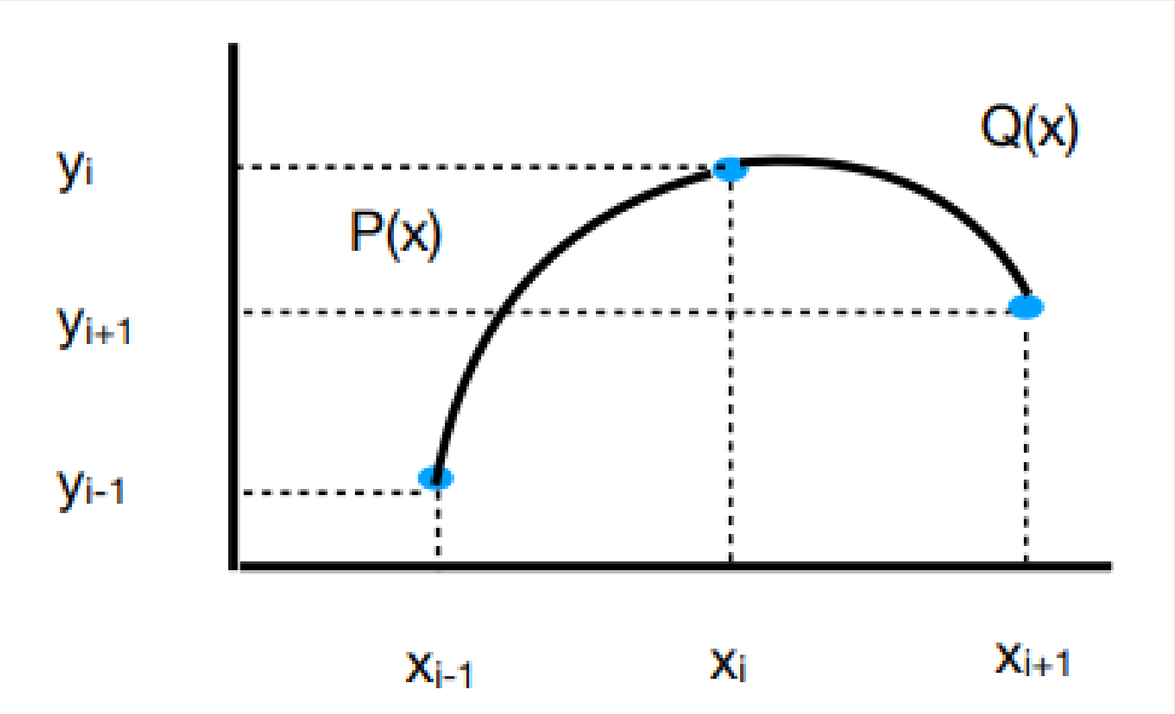
OUTPUT



Quadratic Spline

$$P_{(x)} = a + bx + cx^2$$

Quadratic splines use degree-two polynomials. C^1 smoothness, meaning the function and its first derivative are continuous at the join points.



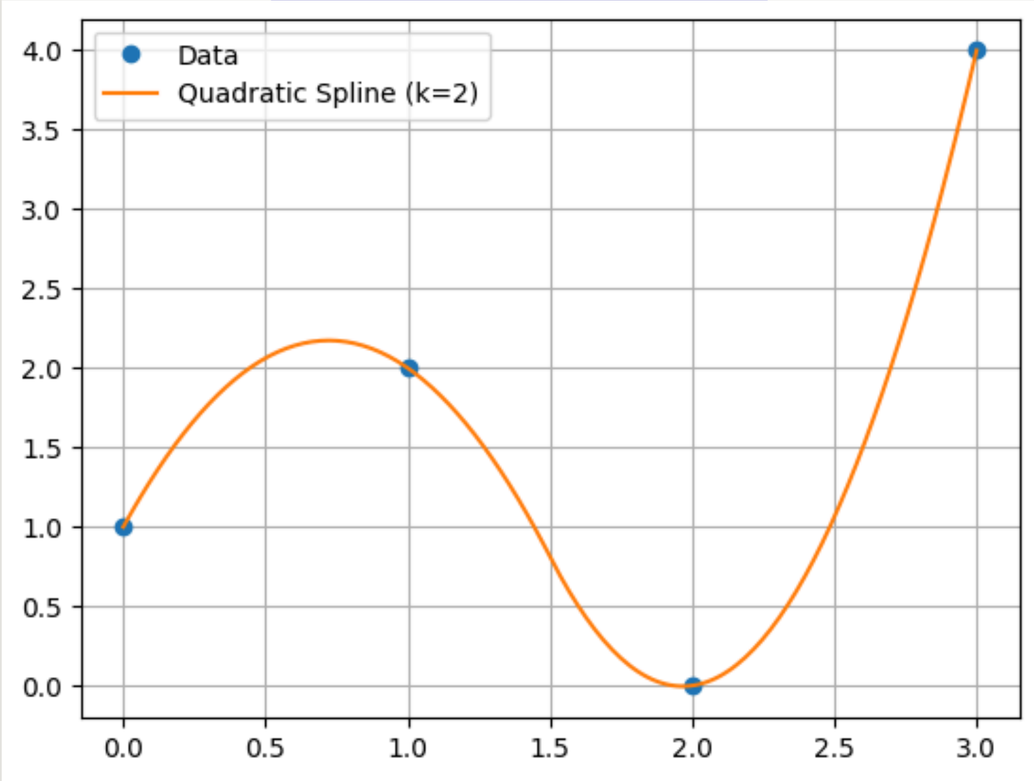
P(x) and Q(x) → two quadratic polynomials
Xi → same value and slope.

Quadratic Spline

INPUT

```
## QUADRATIC SPLINE
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import InterpolatedUnivariateSpline

# data example
x = np.array([0, 1, 2, 3])
y = np.array([1, 2, 0, 4])

quad_spline = InterpolatedUnivariateSpline(x, y, k=2) #k=2 because it is a polynomial of degree 2

xx = np.linspace(0, 3, 200)
plt.plot(x, y, 'o', label='Data')
plt.plot(xx, quad_spline(xx), label='Quadratic Spline (k=2)')
plt.legend()
plt.grid(True)
plt.show()
```

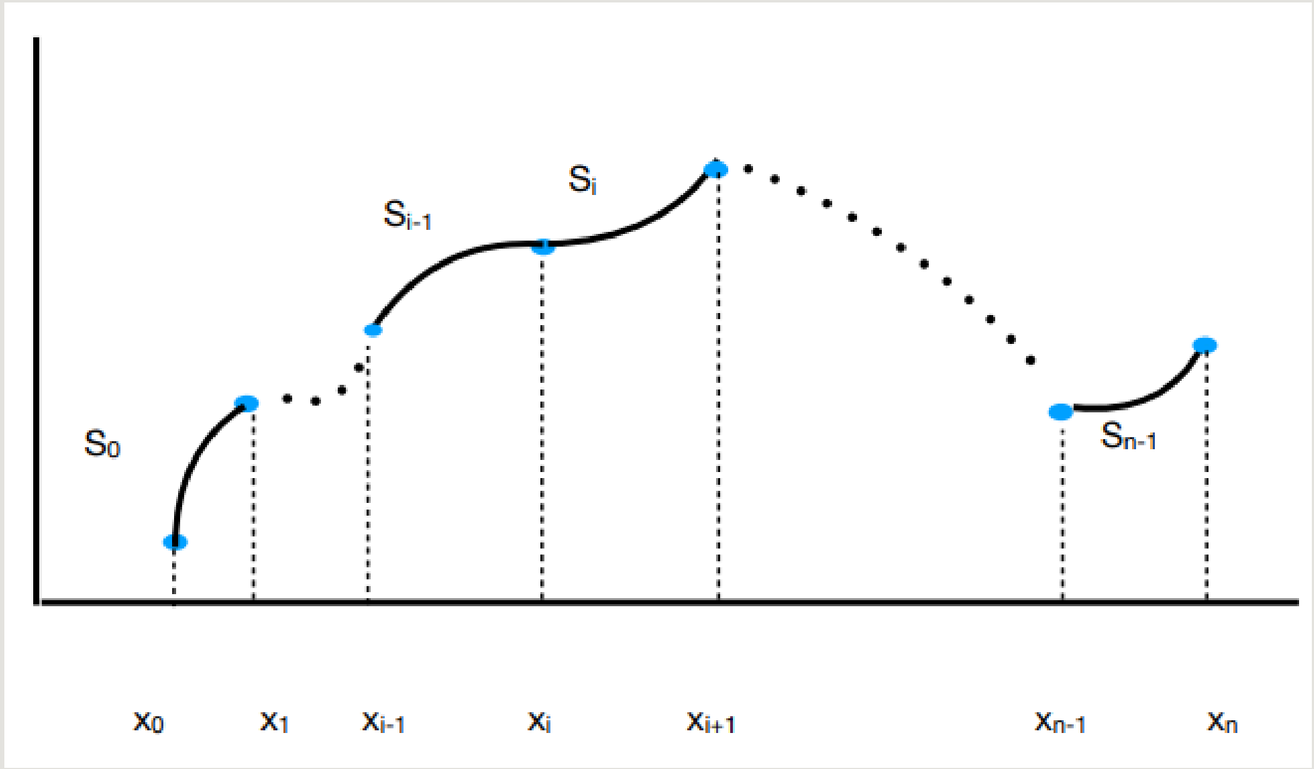
OUTPUT



Cubic Spline

$$P_{(x)} = a + bx + cx^2 + dx^3$$

Cubic splines use degree-three polynomials. C^2 smoothness, ensuring the function, first, and second derivatives are continuous



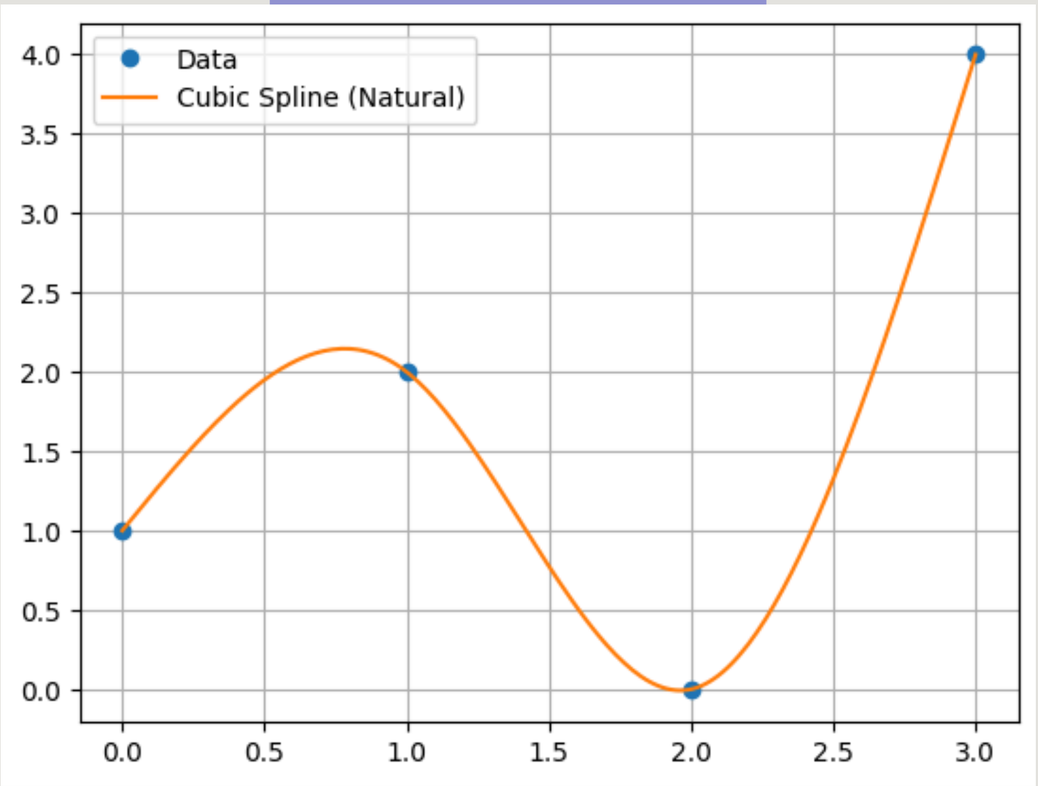
S0,S1,S2 → cubic polynomials
Xi → continuous and smooth curve

Cubic Spline

INPUT

```
## CUBIC SPLINE
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline

# data example
x = np.array([0, 1, 2, 3])
y = np.array([1, 2, 0, 4])

cubic_spline = CubicSpline(x, y, bc_type='natural')

xx = np.linspace(0, 3, 200)
plt.plot(x, y, 'o', label='Data')
plt.plot(xx, cubic_spline(xx), label='Cubic Spline (Natural)')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT



2025

THANK YOU!

Terima Kasih Atas Masukannya