Chapter-06 Interpolation and Approximation

Date	@October 10, 2024
tag	done

approximating polynomial function

x	y
1	2.7183
2	7.3891
3	20.0855

$y = f(x) = e^x$

but if that functions is not given, we have to approximate the polynomial function.

degree of polynomial = no of points - 1

if points = 2 , degree will be 1

deg 2 polynomial → matrix size 3×3

$$\begin{bmatrix} 1 & x_o & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_1^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}$$

first column contains 1s

Matrix Method:

numerical instability

computational complexity

Lagrange Method

reduced computational complexity

$P_N(x) = \sum_{i=0}^{N} y_i \ell_i(x)$

$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N} \frac{x - x_j}{x_i - x_j}$

Example:

Date 20
M T W T F S

	x	y	
x_0	3	5	y_0
x_1	5	7	y_1
x_2	7	3	y_2

find $P_2(3.5)$

$$P_2(x) = \sum_{i=0}^{2} l_i(x) \cdot y_i$$

$$P_2(x) = l_0(x)y_0 + l_1(x)y_1 + l_2(x)y_2$$

$l_0(x) = ?$ $\quad l_1(x) = ?$ $\quad l_2(x) = ?$

degree of polynomial

$i=0$

$$l_0(x) = \prod_{\substack{j=0 \\ j \neq i}}^{2} \left(\frac{x - x_j}{x_i - x_j}\right) = \left(\frac{x-x_1}{x_0-x_1}\right)\left(\frac{x-x_2}{x_0-x_2}\right) = \frac{x-5}{-2} \times \frac{x-7}{-4}$$

$j \neq i$ so can't put that.

$i=1$

$$l_1(x) = \prod_{\substack{j=0 \\ j \neq i}}^{2} \left(\frac{x - x_j}{x_i - x_j}\right) = \left(\frac{x-x_0}{x_1-x_0}\right)\left(\frac{x-x_2}{x_1-x_2}\right) = \frac{x-3}{5-3} \cdot \frac{x-7}{5-7} = \frac{x-3}{2} \cdot \frac{x-}{-}$$

can't put $j=1$

$i=2$

$$l_2(x) = \frac{x-x_0}{x_2-x_0} \cdot \frac{x-x_1}{x_2-x_1} = \frac{x-3}{7-3} \cdot \frac{x-5}{7-5} = \frac{x-3}{4} \cdot \frac{x-5}{2}$$

can't put $j=2$

$$P_2(x) = 5\left(\frac{x-5}{-2} \times \frac{x-7}{-4}\right) + 7\left(\frac{x-3}{2}\right)\left(\frac{x-7}{-2}\right) + 3\left(\frac{x-3}{4}\right)\left(\frac{x-5}{2}\right)$$

Define the Vandermonde Matrix as

$$\mathbf{V} = \begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}$$

General Vandermonde matrix

$$\mathbf{V} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^N \\ 1 & x_1 & x_1^2 & \cdots & x_1^N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^N \end{pmatrix}$$

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

# interpolation data (x,y)
x = np.array([0.0, 0.5, 1.0, 1.5])
y = np.array([1.0, -1.0, 2.0, 1.5])
# assembly of Vandermonde matrix V
V = np.vander(x, increasing=True)
# solution of linear system Va=y
a = npl.solve(V,y)
# define more points for plotting
xx = np.linspace(0.0,1.5,100)
# evaluate the interpolating polynomial at xx
p = a[0]+a[1]*xx+a[2]*xx**2+a[3]*xx**3
plt.plot(xx,p)
plt.plot(x,y,'o')
plt.xlabel('x'); plt.ylabel('y'); plt.show()
```

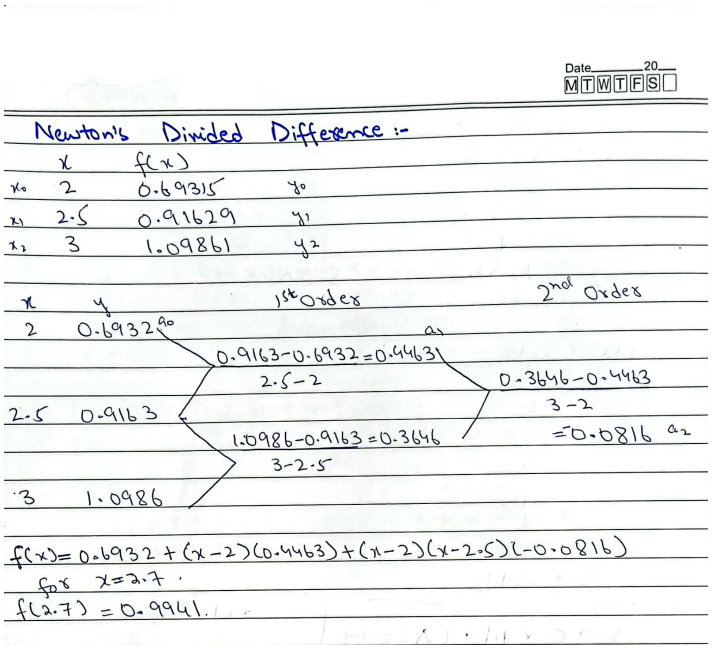
To avoid computation with the Vandermonde matrix we will use a different basis for the set of polynomials, namely the Lagrangian basis functions.

```
def lagrange_basis(z, x):
# Compute the Lagrange basis l_i(z)
# given the nodes x_i stored in vector x
    n = len(x)
    m = len(z)
    basis = np.ones((n, m))
    for i in range(n):
        for j in range(n):
            if i != j:
                basis[i,:] *= (z-x[j])/(x[i]-x[j])
    return basis



def lagrange_interpolant(z, x, y):
#Compute the interpolant using Lagrange polynomials
    n = len(x)
    m = len(z)
    P = np.zeros(m)
    basis = lagrange_basis(z, x)
    for i in range(n):
        P += basis[i,:] * y[i]
    return P



 def f(x):
    return 1.0/x
# define points for creating plot
z = np.linspace(1, 4, 100)
# define data points and store them in x and y
x = np.array([1, 1.5, 4])
y = f(x)
# compute Lagrange interpolant and store it to yz
```

```
yz = lagrange_interpolant(z, x, y)
# plot the results
plt.plot(z,yz,'-')
plt.plot(x,y,'o')
plt.plot(z,f(z),'-.')
plt.legend(["$P_2(x)$","data","$f(x)$"])
plt.xlabel('x'); plt.ylabel('y'); plt.show()
```

```
import scipy.interpolate as spi

def f(x):
    return 1.0 / (1.0 + 25.0 * x**2)
# We evaluate the interpolant at the points z
z = np.linspace(-1, 1, 100)
# We use the data x,y
x = np.linspace(-1, 1, 10)
y = f(x)
# We compute the Lagrange polynomial p
p = spi.lagrange(x, y)
# We compare the graphs of the data and the interpolant
plt.plot(z,p(z),'-')
plt.plot(x,y,'o')
plt.plot(z,f(z),'-.')
plt.legend(["$P_9(z)$","data","$f(z)$"])
plt.xlabel('x'); plt.ylabel('y'); plt.show()
```

Newton’s Interpolation




```
def poly_evaluation(a,x,z):
    #Evaluation of the polynomial with coefficients a at the poi
    N = len(x) - 1 #Degree of polynomial
    p = a[N]
    for k in range(1,N+1):
        p = a[N-k] + (z - x[N-k])*p
    return p

def poly_coeffs(x, y):
    n = len(x)
    a = y.copy()
    for k in range(1,n):
        a[k:n] = (a[k:n] - a[k-1])/(x[k:n] - x[k-1])
    return a

def f(x):
    return 1.0 / (1.0 + 25.0 * x**2)
# z are the points we use to plot the interpolating polynomial
z = np.linspace(-1, 1, 100)
x = np.linspace(-1, 1, 10)
y = f(x)
```

```
a = poly_coeffs(x, y)
# yz are the values of the interpolating polynomial at z
yz = poly_evaluation(a,x,z)
plt.plot(z,yz,'-')
plt.plot(x,y,'o')
plt.plot(z,f(z),'-.')
plt.legend(["$P_9(x)$","data","$f(x)$"])
plt.xlabel('x'); plt.ylabel('y'); plt.show()
```

Hermite Interpolation

Date_____20___
M T W T F S

Hermite :- y y'

	x	$f(x)$	$f'(x)$ → given
x_0	2	0.69315	0.5
x_1	2.5	0.91629	0.4
x_2	3	1.09861	0.3333

Lagrange's $l_i(x)$

$$l_0(x) = \frac{(x-2.5)}{2-2.5}\cdot\frac{(x-3)}{2-3} = 2x^2-11x+15$$

$$l_1(x) = \frac{(x-2)}{2.5-2}\cdot\frac{(x-3)}{2.5-3} = -4x^2+20x-24$$

$$l_2(x) = \frac{(x-2)}{3-2}\cdot\frac{(x-2.5)}{3-2.5} = 2x^2-9x+10$$

$l_0'(x) = 4x-11 \Rightarrow l_0'(x_0) = l_0'(2) = -3$

$l_1'(x) = -8x+20 \Rightarrow l_1'(x_1) = l_1'(2.5) = 0$

$l_2'(x) = 4x-9 \Rightarrow l_2'(x_2) = l_2'(3) = 3$

$H(x) = \sum u_i(x)\cdot y_0 + \sum v_i(x)\cdot y_i'$

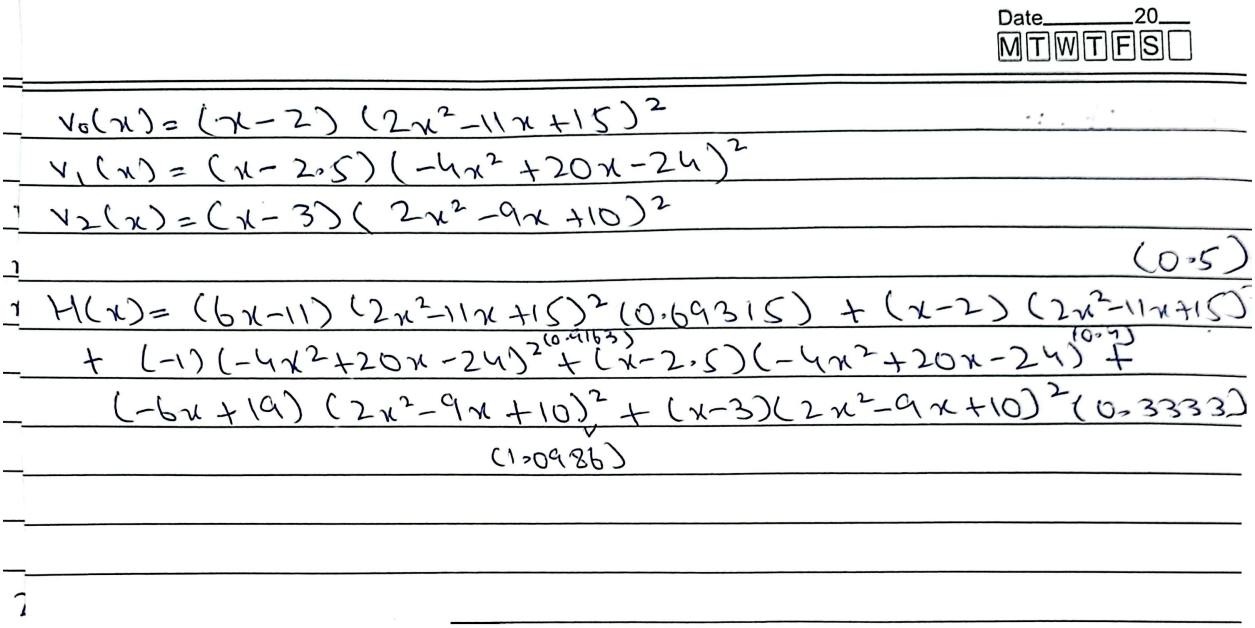
$u_i(x) = [1-2(x-x_i)l_i'(x_i)][l_i(x)]^2$

$v_i(x) = (x-x_i)[l_i(x)]^2$

$u_0(x) = [1-2(x-2)(-3)](2x^2-11x+15) = (6x-11)(2x^2-11x+15)^2$

$u_1(x) = [1-2(x-2.5)(0)](-4x^2+20x-24) = (-4x^2+20x-24)^2$

$u_2(x) = [1-2(x-3)(3)](2x^2-9x+10) = (-6x+19)(2x^2-9x+10)^2$




```
def Hermite(x,y,z):
# x, y: are the interpolating data
# z: values of the derivative at points x
    N = len(x)
    p = np.poly1d([0]) # initialize p. poly1d is a numpy obje
ct for polynomials,  allowing arithmetic and differentiation
    for i in range(N):
        #derive L_k
        L = np.poly1d([1])
        for j in range(N):
            if j!=i:
                L = L*np.poly1d([1.0/(x[i]-x[j]), - x[j]/(x
[i]-x[j])])
        #Derivative of the Lagrange polynomials
        dL = np.polyder(L)
        #Computation of the functions alpha
        alpha = (np.poly1d([1])-2*dL(x[i])*np.poly1d([1,-x
```

```
[i]]))*L**2
        #Computation of the functions beta
        beta = np.poly1d([1,-x[i]])*L**2
        p = p + alpha*y[i]+beta*z[i]
    return p


def f(x):
    return 1.0/(1+25*x**2)
def df(x):
    return -2*25*x/(1+25*x**2)**2
xi = np.linspace(-1,1,10)
yi = f(xi)
zi = df(xi)
x = np.linspace(-1,1,300)
y = f(x)
p = Hermite(xi,yi,zi)
y1 = p(x)
plt.plot(x, y1, label="$H_{19}(x)$")
plt.plot(xi, yi, 'o', label='data')
plt.plot(x, y, '-.',label='$f(x)$')
plt.xlabel('$x$')
plt.ylabel('$y$',rotation=0)
plt.xlim(-1, 1)
plt.legend(loc='upper left', ncol=1)
plt.show()
```

Spline Interpolation:

Continuous Piecewise Linear Interpolation

```
# define the interpolation points
xp = np.linspace(0, 2*np.pi, 10)
```

```
yp = np.sin(xp)
# define the points for ploting
x = np.linspace(0, 2*np.pi, 50)
# generate the linear spline
y = np.interp(x, xp, yp)
# plot spline vs data
plt.plot(x, y, '-x', label='$S_l(x)$')
plt.plot(xp, yp, 'o', label='data')
plt.xlabel('$x$')
plt.ylabel('$y$',rotation=0)
plt.legend(loc='upper right', ncol=1)
```

Disadvatanges:

not smooth

Linear

	x	y	
x_0	0	0	y_0
x_1	10	227.04	y_1
x_2	15	362.78	y_2
x_3	20	517.35	y_3
x_4	22.5	602.97	y_4

· At x = 16, y = ?

$$y_2 = 362.78 + \frac{517.35 - 362.78}{20 - 15}(x - 15)$$

$x = 16$

$$y_{16} = 362.78 + \frac{517.35 - 362.78}{5}(16 - 15)$$

$$= 393.7.$$

$$y_n = y_{upper} + \frac{y_{lower} - y_{upper}}{x_{lower} - x_{upper}}(x - x_{upper})$$

Cubic Spline:

Date 20
M T W T F S

Cubic Splines

⇒ piecewise
⇒ continuity and smoothness
⇒ reduced oscillations in certain conditions
Each $P_n(x)$ will be a 3rd order polynomial.
unknown coefficients → a_i, b_i, c_i, d_i
n → 1 less than no. of data points.

1) Find unknown coefficients (Solve linear equations)

	(4n)	x	y	
<	x_0	3	10	y_0
	x_1	4	15	y_1
<	x_2	6	35	y_2

2 splines so $j=2$

1) Value Matching.

$$S_j(x) = a_j + b_j(x-x_j) + c_j(x-x_j)^2 + d_j(x-x_j)^3$$

⇒ $j=0$ $\quad 3 \le x \le 4$

$$S_0(x) = a_0 + b_0(x-x_0) + c_0(x-x_0)^2 + d_0(x-x_0)^3$$

$$S_0(3) = a_0 + b_0(x-3) + c_0(x-3)^2 + d_0(x-3)^3.$$

Put $x=3$

$S_0(3) = a_0$

$\boxed{a_0 = 10}$

Put $x=4$.

$S_0(4) = a_0 + b_0 + c_0 + d_0 = 15$

$b_0 + c_0 + d_0 = 5$

⇒ $j=1$ $\quad 4 \le x \le 6$

$S_1(x) = a_1 + b_1(x-x_1) + c_1(x-x_1)^2 + d_1(x-x_1)^3.$

P. $S_1(x) = a_1 + b_1(x-4) + c_1(x-4)^2 + d_1(x-4)^3$

Put $x=4$.

$S_1(4) = \boxed{a_1 = 15}$

Put $x=6$

$S_1(6) = a_1 + b_1(2) + c_1(2)^2 + d_1(2)^3$

$35 = 15 + 2b_1 + 4c_1 + 8d_1$

$2b_1 + 4c_1 + 8d_1 = 20$

$b_1 + 2c_1 + 4d_1 = 10$

2) Continuity Conditions:-

$S_0'(x) = b_0 + 2c_0(x-3) + 3d_0(x-3)^2$.

$S_1'(x) = b_1 + 2c_1(x-4) + 3d_1(x-4)^2$.

$S_0'(4) = S_1'(4)$

$b_0 + 2c_0 + 3d_0 = b_1$

$S_0''(x) = 2c_0 + 6d_0(x-3)$

$S_1''(x) = 2c_1 + 6d_1(x-4)$

$S_0''(4) = S_1''(4)$

$2c_0 + 6d_0 = 2c_1$

3) Boundary Condition:

$S_0''(3) = 0$

$2c_0 = 0$

$\boxed{c_0 = 0}$

$S_1''(6) = 0$

$2c_1 + 6d_1(2) = 0$

$2c_1 + 12d_1 = 0$

$c_1 + 6d_1 = 0$

$c_1 = -6d_1$

Solve!

```
import numpy as np
import numpy.linalg as npl
import scipy.sparse as sps

def MyCubicSpline(x, y, dy0, dyN):
    # Returns the moments of f given the data x, y
    # dy0 is the derivative at the left boundary
    # dyN is the derivative at the right boundary
    n = len(x)
    c = np.zeros(n); v = np.zeros(n); u = np.zeros(n-1)
    l = np.zeros(n-1); b = np.zeros(n-1); h = np.zeros(n-1)
    for i in range(n-1):
        h[i] = x[i+1] - x[i]
        b[i] = (y[i+1]-y[i])/h[i]
    u[0] = 1.0
    v[0] = 6.0*(b[0]-dy0)/h[0]
    c[0] = 2.0
    for i in range(1,n-1):
        c[i] = 2.0
        u[i] = h[i]/(h[i-1] + h[i])
        l[i-1] = h[i-1]/(h[i-1] + h[i])
        v[i] = 6.0*(b[i]-b[i-1])/(h[i-1] + h[i])
    l[n-2] = 1.0
    c[n-1] = 2.0
    v[n-1] = 6.0*(dyN-b[n-2])/h[n-2]
    diagonals = [c, l, u]
    A = sps.diags(diagonals, [0, -1, 1]).todense()
    z = npl.solve(A,v)
    return z


def EvalCubicSpline(x, y, z, xx):
    # Returns the cubic spline evaluated at xx
    # z = the moments of the cubic spline
```

```
    # xx = the vector with values of x on which we want the cubi
    n = len(x)
    m = len(xx)
    yy = np.zeros(m)
    for j in range(m):
        xvalue = xx[j]
        # First detect the index i
        for i in range(n-2, -1, -1):
            if (xvalue - x[i] >= 0.0):
                break
    # Implement formula (\ref{eq:clmbcbspl})
        h = x[i+1] - x[i]
        B = -h*z[i+1]/6.0-h*z[i]/3.0+(y[i+1]-y[i])/h
        tmp =z[i]/2.0+(xvalue-x[i])*(z[i+1]-z[i])/6.0/h
        tmp = B+(xvalue-x[i])*tmp
        yy[j] = y[i] + (xvalue - x[i])*tmp
    return yy


def f(x):
    return 1.0/(1+25*x**2)
def df(x):
    return -2*25*x/(1+25*x**2)**2
x = np.linspace(-1,1,9)
y = f(x)
z = df(x)
xx = np.linspace(-1,1,300)
yy = f(xx)
ss = MyCubicSpline(x, y, z[0], z[-1])
zz=EvalCubicSpline(x, y, ss, xx)
plt.plot(xx,yy,xx,zz,x,y,'o')
```

Method of Least Squares:

To solve the polynomial fitting equations using the least squares method, follow these general steps. These steps apply to fitting a line (degree 1), quadratic (degree 2), and cubic (degree 3) polynomials, based on the instructions and the equations you shared:

```
def linear_least_squares(x, y, xx):
# returns the vector yy with the linear least squares
# approximation of the input data (x,y)
    N = len(x)
    p = np.sum(x)
    q = np.sum(y)
    r = np.sum(np.multiply(x,y))
    s = np.sum(np.square(x))
    D = (N+1)*s-p**2
    a = [(N+1)*r - p*q]/D
    b = [s*q - p*r]/D
    yy = a*xx + b
    return yy

x = np.linspace(-1,2,10)
y = 2.0*x+1.0+np.random.rand(10)
xx = np.linspace(-1,2,20)
yy = linear_least_squares(x, y, xx)
plt.plot(xx,yy,'-',x,y,'o')
plt.xlabel('$x$')
plt.ylabel('$y$',rotation=0)
plt.show()
```

1. Linear Fit (Degree 1 Polynomial)

For a line of the form $P(x) = a_0 + a_1 x$, you're given data points:

(−1,2),(−2,3),(−3,4),(−4,5),(−5,6)

- **Set up the least squares error function**:

$E = \sum_{i=1}^{n} (y_i - P(x_i))^2$

where yiy_iyi are the actual values and $P(x_i)$ are the predicted values using the polynomial.

- **Construct the system of equations**:
  You'll have a system of linear equations based on the following formula:
  $$\begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix}$$
  .

2. Quadratic Fit (Degree 2 Polynomial)

For a quadratic polynomial of the form $P(x) = a_0 + a_1 x + a_2 x^2$, use the second set of data points:

(1,1),(2,4),(3,9),(4,16),(5,25)

- **Set up the least squares error function** similarly:

  $E = \sum_{i=1}^{n} (y_i - P(x_i))^2$

- **Construct the system of equations**:
  Again, calculate the necessary sums and solve this 3×3 system for a0, a1, and a2.
  $$\begin{bmatrix} n & \sum x_i & \sum x_i^2 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \end{bmatrix}$$

3. Cubic Fit (Degree 3 Polynomial)

For a cubic polynomial of the form $P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$, use the third set of data points:

(1,1),(2,8),(3,27),(4,64),(5,125)

- **Set up the least squares error function**:

  $E = \sum_{i=1}^{n} (y_i - P(x_i))^2$

- **Construct the system of equations**:
  Once again, calculate the required sums from the data and solve this 4×4 system for a0, a1, a2, and a3.

$$\begin{bmatrix} n & \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^5 \\ \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \sum x_i^6 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \\ \sum x_i^3 y_i \end{bmatrix}$$

Solving the Equations

You can solve these systems of equations by using:

- **Matrix methods** (such as Gaussian elimination, LU decomposition, or using matrix inversion)
- **Numerical tools** (such as Python's `numpy.linalg.solve()`)

Steps to Solve Using Cramer's Rule:

For a system like:

$$\mathbf{Ax} = \mathbf{b}$$

1. **Calculate the determinant of the matrix** A, denoted as $\det(A)$.
2. For each unknown a_i, create a new matrix A_i by replacing the i-th column of A with the vector $\mathbf{b}$.
3. Calculate the determinant of each matrix A_i, denoted as $\det(A_i)$.
4. Solve for each a_i using the formula:

$$a_i = \frac{\det(A_i)}{\det(A)}$$

Example: Linear Fit (Degree 1 Polynomial)

For a degree 1 polynomial $P(x) = a_0 + a_1 x$, we have the system of equations:

$$\begin{bmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix}$$

1. **Calculate the determinant of A:**

$$\det(A) = n \cdot \sum x_i^2 - (\sum x_i)^2$$

2. **Construct A_0 by replacing the first column of A with $\mathbf{b}$:**

$$A_0 = \begin{bmatrix} \sum y_i & \sum x_i \\ \sum x_i y_i & \sum x_i^2 \end{bmatrix}$$

   - Calculate $\det(A_0)$.

3. **Construct A_1 by replacing the second column of A with $\mathbf{b}$:**

$$A_1 = \begin{bmatrix} n & \sum y_i \\ \sum x_i & \sum x_i y_i \end{bmatrix}$$

   - Calculate $\det(A_1)$.

4. **Solve for a_0 and a_1:**

$$a_0 = \frac{\det(A_0)}{\det(A)}, \quad a_1 = \frac{\det(A_1)}{\det(A)}$$

Higher Degree Polynomials

For degree 2 (quadratic) and degree 3 (cubic) polynomials, the same procedure applies, but you will need to:

- Calculate the determinants of 3x3 (for quadratic) or 4x4 (for cubic) matrices.
- Replace each column corresponding to the unknown coefficients with the constants from the right-hand side (the vector $\mathbf{b}$).

Practical Considerations

- For a **degree 1 polynomial**, solving using Cramer's Rule is fairly straightforward since the matrix size is small (2x2).
- For **degree 2** (3x3 matrix) and **degree 3** (4x4 matrix) polynomials, calculating determinants manually can be tedious. Using a numerical solver or a computational tool (like Python's `numpy`) is generally more efficient.

skip the codes

```
def poly_least_squares(x, y, xx, m):
# Computes the least squares approximation of degree m
# for the data points (x,y) evaluated at xx
    N = len(x)
    A = np.zeros((m+1,m+1))
    b = np.zeros(m+1)
    s = np.zeros(2*m+1)
    # Assembly of A and b
    for i in range(N):
        temp = y[i]
        for j in range(m+1):
            b[j] = b[j] + temp
            temp = temp*x[i]
        temp = 1.0
        for j in range(2*m+1):
            s[j] = s[j] + temp
            temp = temp*x[i]
    for i in range(m+1):
```

```
        for j in range(m+1):
            A[i,j] = s[i+j]
    z = np.linalg.solve(A,b)
    # Evaluation of the polynomial at the nodes xx
    p = z[m]
    for j in range(m):
        p = p*xx+z[m-j-1]
    return p


x = np.linspace(-1,2,10)
y = 2.0*np.square(x)+1.0+np.random.rand(10)
xx = np.linspace(-1,2,20)
yy = poly_least_squares(x, y, xx, 2)
plt.plot(xx,yy,'-',x,y,'o')
plt.xlabel('$x$'); plt.ylabel('$y$',rotation=0); plt.show()
```

```
x = np.linspace(0,2,10)
y = 3.0*np.exp(2.0*x)+0.1*np.random.rand(10)
z = np.log(y)
xx = np.linspace(0,2,20)
zz = linear_least_squares(x, z, xx)
yy = np.exp(zz)
plt.plot(xx,yy,'-',x,y,'o')
plt.xlabel('$x$')
plt.ylabel('$y$',rotation=0)
plt.show()
```

```
x = np.linspace(-1,2,10)
y = 2.0*np.square(x)+1.0+np.random.rand(10)
xx = np.linspace(-1,2,20)
yy = np.polyfit(x, y, 2)
```

```
zz = np.poly1d(yy)
plt.plot(xx,zz(xx),'-',x,y,'o')
plt.xlabel('$x$'); plt.ylabel('$y$',rotation=0); plt.show()
```

```
import scipy.interpolate as spi
import matplotlib.pyplot as plt
import numpy as np

def f(x):
    return 1.0 / (1.0 + 25.0 * x**2)
num_points = 10
x = np.linspace(-1, 1, num_points)
y = f(x)
cs = spi.CubicSpline(x, y)
xx = np.linspace(-1, 1, 100)
plt.plot(x, y, 'o', label='data')
plt.plot(xx, f(xx), label='true')
plt.plot(xx, cs(xx), label='Spline')
plt.xlim(-1, 1)
plt.legend(loc='upper left', ncol=1)
plt.xlabel('$x$'); plt.ylabel('$y$',rotation=0); plt.show()
```

Interpolation in two Dimensions

not doinnn

```
def f(x,y):
    z = np.exp(-(x**2+y**2))
```

```
    return z
# Specify that the graph is three-dimensional
fig = plt.figure()
ax = plt.axes(projection='3d')
# Generate the data
x = np.arange(-2, 2, 0.1)
y = np.arange(-2, 2, 0.1)
X, Y = np.meshgrid(x, y)
Z = f(X,Y)
# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap='viridis')
plt.show()
```

```
#define the function f
def f(x,y):
    z = np.exp(-(x**2+y**2))
    return z
# generate one-dimensional grids
x = np.linspace(-2, 2, 10)
y = x.copy()
z = f(x,y)
# generate two-dimensional grid arrays
X, Y = np.meshgrid(x, y)
Z = f(X,Y)
# interpolate the coarse data
z2 = spi.RegularGridInterpolator((x, y), Z, method='cubic')
# evaluated the interpolant on dense grid
xx = np.linspace(-2, 2, 100)
yy = np.linspace(-2, 2, 100)
XX, YY = np.meshgrid(xx, yy)
ZZ = z2((XX,YY))
# plot the original and interpolated data
fig, axes = plt.subplots(nrows=1, ncols=2)
axes[0].pcolormesh(X, Y, Z)
```

```
axes[1].pcolormesh(XX, YY, ZZ)
plt.show()
```