III. Interpolation and Curve Fitting

Interpolation

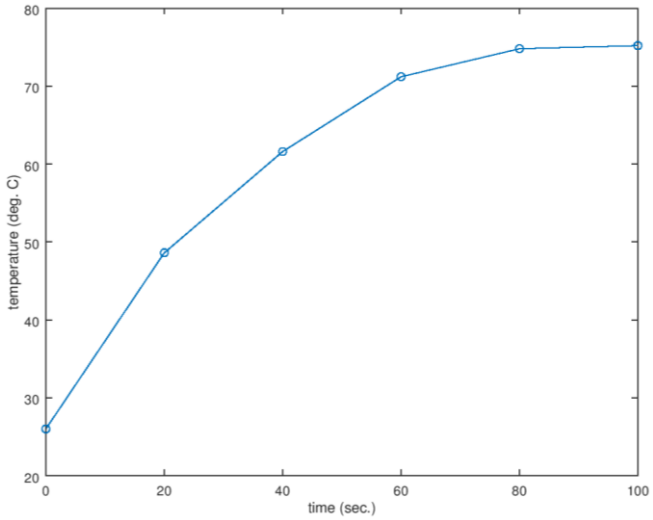
Linear Interpolation

The linear interpolation assumes a straight line equation to interpolate the points between every two given points in the data set.

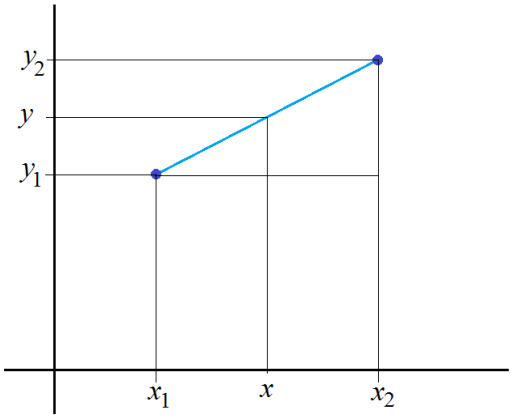
Example 1: Suppose you have a table of readings of temperature of a chemical reaction at even time intervals as following

Time (sec.)	0	20	40	60	80	100
Temperature (°C)	26.0	48.6	61.6	71.2	74.8	75.2

The linear plot to the table is as following



From this set of data, if the temperature of reaction at time 50 sec is required, how will it be found? The simplest method that is used frequently in the school to find mid values from tables is the *linear interpolation*. In this method, the section of the curve connecting the two known points (x_1, y_1) and (x_2, y_2) is assumed to be a straight line. In this case, a straight line is imagined between (40, 61.6) and (60, 71.2).



Since the slope of the line is equal, or by similarity of the two triangles shown, the following equation can be written

$$\frac{y-y_1}{x-x_1}=\frac{y_2-y_1}{x_2-x_1}$$

So, y at given x will be

$$y=y_1+\frac{y_2-y_1}{x_2-x_1}(x-x_1)$$

This simple formula can be saved in a programmable calculator and used to find interpolated values from tables in various field of science and engineering.

Thus, the value of temperature corresponding to time 50 seconds will be

$$temp=61.6+\frac{71.2-61.6}{60-40}(50-40)=66.4\ ^\circ\text{C}$$

The disadvantage of the linear interpolation method is that the other points except the two adjacent to the required one are totally ignored. So, the trend of the curve coinciding the whole set of data is not taken into account and, consequently, the interpolated point is not accurately lying on that curve.

In this section, two well-known methods of interpolation will be introduced and coded, Lagrange's and Newton's interpolation methods.

Lagrange's Method

This method is based on creating a polynomial of degree n. The degree depends on the number of points considered in the data set so they should be $n+1$ points. For example, for a third degree polynomial (cubic), $n=3$, four data points are required and it will written as following

$$y(x)=y_1\ell_1(x)+y_2\ell_2(x)+y_3\ell_3(x)+y_4\ell_4(x)$$

or

$$y(x)=\sum_{i=1}^{n+1}y_i\ell_i(x)$$

where

$$\ell_1(x)=\frac{(x-x_2)}{(x_1-x_2)}\frac{(x-x_3)}{(x_1-x_3)}\frac{(x-x_4)}{(x_1-x_4)}$$

$$\ell_2(x)=\frac{(x-x_1)}{(x_2-x_1)}\frac{(x-x_3)}{(x_2-x_3)}\frac{(x-x_4)}{(x_2-x_4)}$$

$$\ell_3(x)=\frac{(x-x_1)}{(x_3-x_1)}\frac{(x-x_2)}{(x_3-x_2)}\frac{(x-x_4)}{(x_3-x_4)}$$

$$\ell_4(x)=\frac{(x-x_1)}{(x_4-x_1)}\frac{(x-x_2)}{(x_4-x_2)}\frac{(x-x_3)}{(x_4-x_3)}$$

or

$$\ell_i(x) = \prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{(x - x_j)}{(x_i - x_j)}$$

Finally, combining the general forms of the polynomial terms gives:

$$y(x) = \sum_{i=1}^{n+1} y_i \left(\prod_{\substack{j=1 \\ j \neq i}}^{n+1} \frac{(x - x_j)}{(x_i - x_j)} \right)$$

This form can be used as the algorithm for Lagrange's interpolation program. To simplify coding, let's construct the program step by step by using the data in the table of time-temperature in page 1.

Step 1: Define the data set as two lists and number of points `m` where `m` = n+1.

```
x = [0, 20, 40, 60, 80, 100]
y = [26.0, 48.6, 61.6, 71.2, 74.8, 75.2]
m = len(x)  #function returns number of list elements
n = m - 1
```

Step 2: Construct the summation for-loop to n+1. Note that the summation variable (`yp`) be the interpolated *y* value for the given x value (`xp`). The summation variable should initialized as zero before the loop.

```
yp = 0                    #initialization of summation variable
for i in range(n+1):
  #------------------------------
  # product loop will be put here
  #------------------------------
  yp += y[i]*L
```

Step 3: Construct the product loop inside the summation loop for `j = 1` to m but `j` should never be equal to `i`. The product variable should be initialized by one before the loop

```
  L = 1;                      #initialization of product variable
  for j in range(n+1):
      if j != i:              #condition to perform product
            L *= (xp - x[j])/(x[i] - x[j]) #product of x terms
```

Step 4: Put the code parts altogether and add input statement to enter the *x* value (xp) at the beginning and display the result at the end of the program.

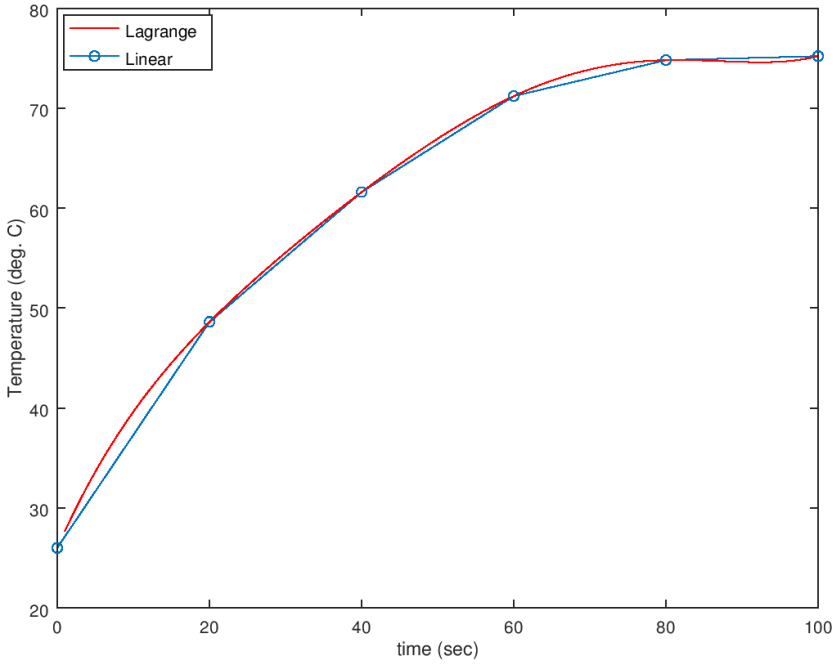
```
x = [0, 20, 40, 60, 80, 100]
y = [26.0, 48.6, 61.6, 71.2, 74.8, 75.2]
m = len(x)
n = m-1
xp = float(input('Enter x: '))
```

```
yp = 0
for i in range(n+1):
    L = 1;
    for j in range(n+1):
        if j != i:
            L *= (xp - x[j])/(x[i] - x[j])
    yp += y[i]*L
print('For x = %.1f, y = %.1f' % (xp, yp))
```

The run or the program gives:

```
Enter x: 50
For x = 50.0, y = 66.9
```

The following graph shows the difference between the 5th degree polynomial curve resulted from Lagrange's method and the linear connection between data points. It is obvious that the results of interpolation of the two methods will be closer between 20 and 60 seconds. (Check for the difference between the result of each method at time = 10 sec.)



Finally, in the example, the points are equally spaced in time. Lagrange's method can be applied for the non-equally spaced data points as well.

Exercise: By using Lagrange interpolation method, find the value of expansion ratio corresponding to weight of 5.5 lb. within a tensile test readings tabulated as follows

Weight (lb.)	2.4	5.1	7.0	8.5	9.7
Expansion ration	0.101	0.128	0.241	0.403	0.677

Newton's Method

Newton's method is applied to given data points in order to obtain a polynomial in the form

$$y(x) = a_0 + (x - x_1)a_1 + (x - x_1)(x - x_2)a_2 + \cdots + (x - x_1)(x - x_2)\dots(x - x_n)a_n$$

This can be achieved in two steps. The First step is the *divided differences* procedure and used to calculated the coefficients of the polynomial, i.e. *a*'s. The second is simple substitution of values of a given *x* into the polynomial to get the interpolated *y*.

The divided differences are applied to create a table of given data plus n columns of differences, where *n* is the degree of the polynomial for *n*+1 data points. For example, the table shown below represents the divided differences levels for 4 data points.

(0)	(1)	(2)	(3)	(4)
x_1	$y_1^{(1)} = y_1$			
x_2	$y_2^{(1)} = y_2$	$y_2^{(2)}$		
x_3	$y_3^{(1)} = y_3$	$y_3^{(2)}$	$y_3^{(3)}$	
x_4	$y_4^{(1)} = y_4$	$y_4^{(2)}$	$y_4^{(3)}$	$y_4^{(4)}$

In this table, the column (0) is the *x* values of the data set, the column (1) is the corresponding *y* values. Thus, the superscripts of over the *y*'s refer to their column numbers.

The column (2) is the differences of the second column with respect to corresponding *x* values and its values are calculated as

$$y_i^{(2)} = \frac{y_i^{(1)} - y_1^{(1)}}{x_i - x_1}, i = 2,3,4$$

Similarly, the column (3) is the differences of the third. So, its values will be

$$y_i^{(3)} = \frac{y_i^{(2)} - y_2^{(2)}}{x_i - x_2}, i = 3,4$$

Finally, the last column contains a single value

$$y_4^{(4)} = \frac{y_4^{(3)} - y_3^{(3)}}{x_4 - x_3}$$

So, the general formula for the divided differences is

$$y_i^{(j+1)} = \frac{y_i^{(j)} - y_j^{(j)}}{x_i - x_j}, j = 1, \dots, n \text{ and } i = j + 1, \dots, n + 1$$

Where $y_1^{(1)} = y_1$ and $y_2^{(1)} = y_2$ and so on.

Example 2: Construct the divided differences table for the following data points:

x	0.0	1.5	2.8	4.4	6.1	8.0
y	0.0	0.9	2.5	6.6	7.7	8.0

Solution: The manual calculation of the given values results in the following table

(0)	(1)	(2)	(3)	(4)	(5)	(6)
0.0	0.0					
1.5	0.9	0.6				
2.8	2.5	0.89286	0.22528			
4.4	6.6	1.5	0.31034	0.053162		
6.1	7.7	1.2623	0.14398	-0.024636	-0.045764	
8.0	8.0	1.0	0.061538	-0.031489	-0.023514	0.011711

Now, let's code the divided differences procedure to construct the table by computer in the following steps:

Step 1: Define the data set as two lists and number of points m, where m = n+1.

```
x = [0.0, 1.5, 2.8, 4.4, 6.1, 8.0]
y = [0.0, 0.9, 2.5, 6.6, 7.7, 8.0]
n = len(x) - 1
```

Step 2: Construct an array with dimensions m x m to save values of $y_i^{(j)}$. The first column will be the given y values. Note that the function zeros() should be imported for the module *NumPy* by the command:

```
import numpy as np
```

So,

```
Dy = np.zeros((n+1, n+1))     # constructs y-differences table
Dy[:,0] = y;                  # assigns values of y to first column of Dy
```

Step 3: Make two nested loops: the j-loop is the outer and controls table columns and i-loop is the inner and controls the differences according to the general formula of the divide differences.

```
for j in range(n):
    for i in range(j+1, n+1):
        Dy[i,j+1] = (Dy[i,j]-Dy[j,j])/(x[i]-x[j])
print(Dy)
```

Notice that the positions and values of i's and j's in the program are similar to theirs in the general formula. The output of this portion of the code is

```
[[ 0.          0.          0.          0.          0.          0.        ]
 [ 0.9         0.6         0.          0.          0.          0.        ]
 [ 2.5         0.89285714  0.22527473  0.          0.          0.        ]
 [ 6.6         1.5         0.31034483  0.05316881  0.          0.        ]
 [ 7.7         1.26229508  0.14397719 -0.02463562 -0.04576731  0.        ]
 [ 8.          1.          0.06153846 -0.03148774 -0.02351571  0.01171137]]
```

Back to the Newton's method, the coefficients of the polynomial will be the values of the main diagonal of the divided differences table. So,

$$a_0 = y_1^{(1)}, a_1 = y_2^{(2)}, a_2 = y_3^{(3)}, \ldots, a_n = y_{n+1}^{(n+1)}$$

This can be coded by two simple ways: (1) by construction of a one dimensional array, *a*, and transferring the main diagonal elements or (2) by using the main diagonal elements directly in computing polynomial terms. The second approach is better since it does not require additional memory.

The second part of Newton's method is calculation of the polynomial for a given *x* value. The polynomial can be rewritten in the following general form

$$y(x) = a_0 + \sum_{i=1}^{n}\left[\prod_{j=1}^{i}(x - x_j)\right]a_i$$

or

$$y(x) = y_1^{(1)} + \sum_{i=1}^{n}\left[\prod_{j=1}^{i}(x - x_j)\right]y_{i+1}^{(i+1)}$$

Thus, the latter form can be coded in the following lines:

```
yp = Dy[0,0]                  # the term of a0
for i in range(n)             # loop form the term of a1 to an
  xprod = 1;                  # xprod initialized for current term
  for j in range(i+1)         # one is added since i starts from 0
    xprod *= xp – x[j]        # product of x differences
  yp += xprod*Dy[i+1,i+1]     # summation of terms
```

Finally, the program can be put all together after adding input and display statements:

```
import numpy as np

x = [0.0, 1.5, 2.8, 4.4, 6.1, 8.0]
y = [0.0, 0.9, 2.5, 6.6, 7.7, 8.0]
n = len(x) - 1
xp = float(input('Enter x: '))
Dy = np.zeros((n+1, n+1))
Dy[:,0] = y
for j in range(n):
    for i in range(j+1, n+1):
        Dy[i,j+1] = (Dy[i,j]-Dy[j,j])/(x[i]-x[j])

yp = Dy[0,0]
for i in range(n):
```

```
    xprod = 1
    for j in range(i+1):
        xprod *= xp - x[j]
    yp += xprod*Dy[i+1,i+1]
print('For x = %.1f, y = %.1f' % (xp, yp))
```

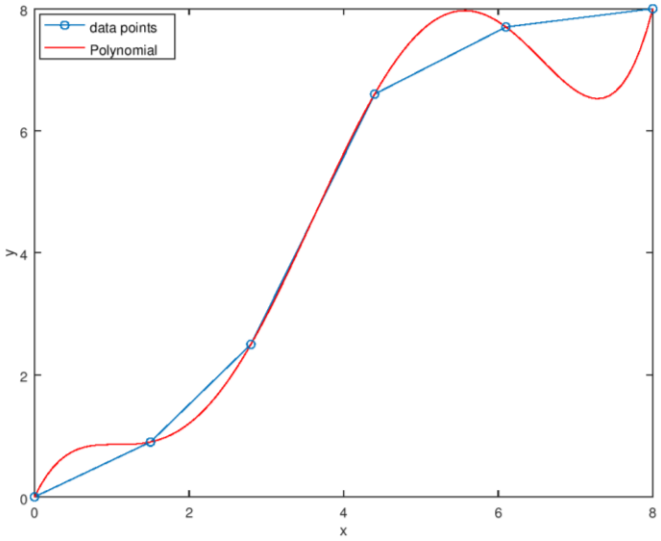
As a first test of the program, let's input values of x from the given table.

```
Enter x: 4.4
For x = 4.4, y = 6.6

Enter x: 8
For x = 8.0, y = 8.0

Enter x: 2.8
For x = 2.8, y = 2.5
```

Thus, the polynomial passes through the given points as shown in the following graph.



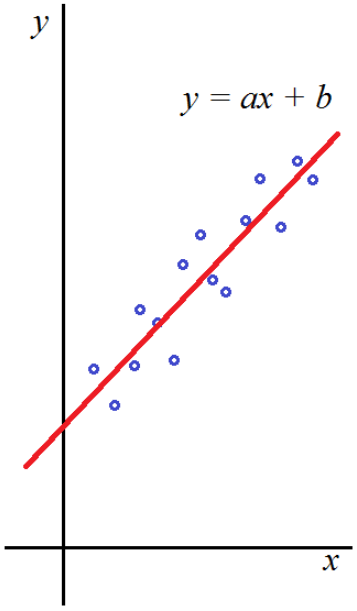
Curve Fitting

Curve fitting is to find the equation of the curve that passes through the given data points with least deviation from the points. Thus, the main difference between interpolation and curve fitting is that the latter does not have to coincide all given data points. The technique used in finding the curve equation is known as the least-squares method where squares of the differences between given points and fitting curve function values are minimized.

Fitting with a Straight Line (Linear Regression)

If the given data points represent a linear behavior of a physical or statistical experiment, for example, they are fitted by a straight line equation

$$f(x) = a + bx$$



The coefficients a and b can be found by the equations

$$a = \frac{\bar{y}\sum x_i^2 - \bar{x}\sum x_i y_i}{\sum x_i^2 - n\bar{x}^2}$$

$$b = \frac{\sum x_i y_i - \bar{x}\sum y_i}{\sum x_i^2 - n\bar{x}^2}$$

Where $\bar{x}$ and $\bar{y}$ are the mean values of given n data points:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

Example 3: Find the equation of the straight line that fits the data:

x	3	4	5	6	7	8
y	0	7	17	26	35	45

Solution: This problem can be solved in two simple ways:

1. Using a for-loop for all required summations and then calculation of a and b.

```
x = [3, 4, 5, 6, 7, 8]
y = [0, 7, 17, 26, 35, 45]
n = len(x)      # number of data points
sumx = sumx2 = sumxy = sumy = 0
for i in range(n):
    sumx += x[i]
    sumx2 += x[i]**2
    sumxy += x[i]*y[i]
    sumy +=  y[i]
xm = sumx / n  # mean of x values
ym = sumy / n  # mean of y values
a = (ym*sumx2 - xm*sumxy)/(sumx2 - n*xm**2)
b = (sumxy - xm*sumy)/(sumx2 - n*xm**2)
print('The straight line equation:')
print('f(x) = (%.3f) + (%.3f)x'%(a,b))
```

The program displays

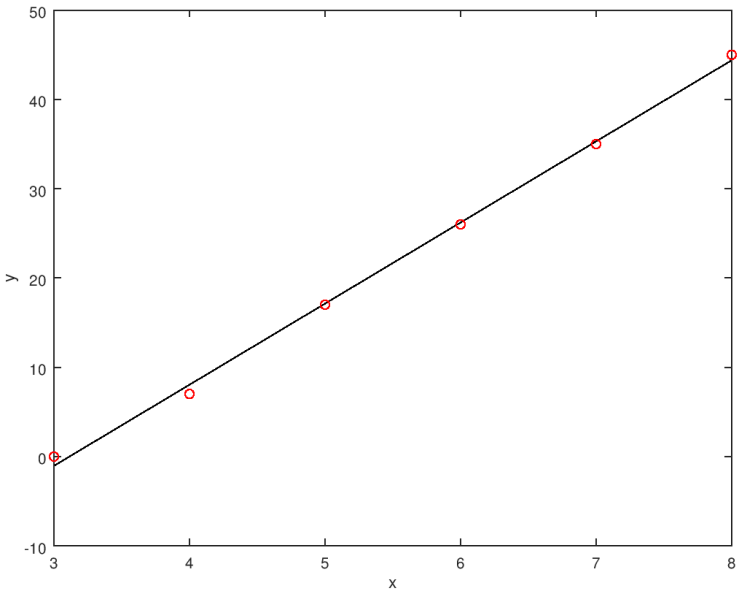
```
The straight line equation:
f(x) = (-28.305) + (9.086)x
```

2. Using NumPy functions: array(), sum()and mean()

```
from numpy import array, sum, mean
x = array([3, 4, 5, 6, 7, 8])
y = array([0, 7, 17, 26, 35, 45])
n = len(x)
a = (mean(y)*sum(x**2)-mean(x)*sum(x*y))/(sum(x**2)-n*mean(x)**2)
b = (sum(x*y)-mean(x)*sum(y))/(sum(x**2)-n*mean(x)**2)
print('The straight line equation:')
print('f(x) = (%.3f) + (%.3f)x'%(a,b))
```

And the output is

```
The straight line equation:
f(x) = (-28.305) + (9.086)x
```



Fitting with a Polynomial Curve

Considering a polynomial has the form

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

If a set of data containing *m* points is to be fitted by the polynomial curve of degree *n*, a system of linear equations is formulated to compute values of the coefficients

$$[A]\{a\} = \{B\}$$

where

$$A=\begin{bmatrix} m & \sum x_i & \sum x_i^2 & \cdots & \sum x_i^n \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \cdots & \sum x_i^{n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_i^n & \sum x_i^{n+1} & \sum x_i^{n+2} & \cdots & \sum x_i^{2n} \end{bmatrix}, \quad B=\begin{Bmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x_i^n y_i \end{Bmatrix}$$

It should be noticed that

$$\sum \quad = \sum_{i=1}^{m}$$

In other words, the all summation signs shown above are from i=1 to m.

Once the all coefficients are calculated, the system can be solved by using a linear system solving technique like Gauss-Elimination. In this section, the function `solve()` from the module `numpy.linalg` will be used. This module contains linear algebra functions.

For example, a 3rd degree polynomial system will be:

$$A=\begin{bmatrix} m & \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^5 \\ \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \sum x_i^6 \end{bmatrix}, \quad B=\begin{Bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \\ \sum x_i^3 y_i \end{Bmatrix}$$

Example 4: Find the quadratic polynomial that fits the following set of data

x	0	1	2	3	4	5
y	2	8	14	28	39	62

Solution: The general form can be programmed in the following steps:

Step 1: Define the given data in x and y arrays, and construct the square matrix and vectors of the system in the dimension n+1.

Step 2: By using two nested i and j loops, define the elements of the matrix *A*. With paying some attention to the relation between the position of each element and the power of its x_i, it can be noticed that the power = row number + column number when considering that the row and column indices in Python start from 0. Accordingly, the power of x_i at each row of *B* is equal to the row number.

Step 3: At the end of the loops, the system can be solved for the vector of the polynomial coefficients by using a numerical method.

So, the code will be as following

```
import numpy as np
```

```
x = np.arange(6)
y = np.array([2, 8, 14, 28, 39, 62])
m = len(x)                    # number of data points
n = 2                         # degree of the polynomial
A = np.zeros((n+1, n+1))
B = np.zeros(n+1)
a = np.zeros(n+1)
# Loops of system formation
for row in range(n+1):
    for col in range(n+1):
        if row == 0 and col == 0:
            A[row,col] = m
            continue
        A[row,col] = np.sum(x**(row+col))
    B[row] = np.sum(x**(row) * y)

a = np.linalg.solve(A,B)   # solution of the linear system [A]{a}={B}

# Display of the results in polynomial form
print('The polynomial: \n')
print('f(x) =\t %f \t'% a[0])
for i in range(1, n+1):
    print('\t %+f x^%d'%(a[i],i))
```

The output:

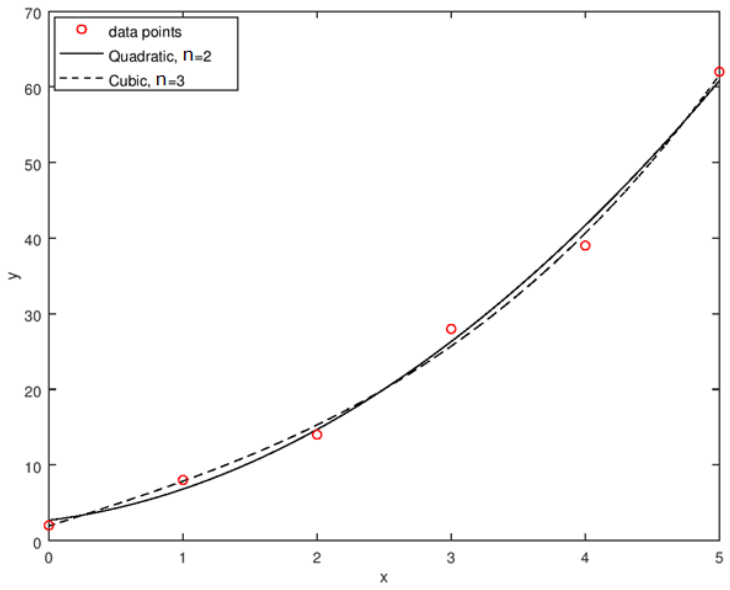
```
The polynomial:
f(x) =   2.678571
         +2.253571 x^1
         +1.875000 x^2
```

To try fitting the given data in the example with a cubic polynomial, all that have to be done is to set n = 3, and the output will be

```
The polynomial:

f(x) =   1.928571
         +5.678571 x^1
         -0.000000 x^2
         +0.250000 x^3
```

The graph shown below compares between the quadratic and cubic polynomial curves in how they fitted the given data points. Practically, the selection of the best fitting curve can be done not only according to the curve behavior, but also to the required polynomial degree and other conditions related to the physical problem.



Interpolation in SciPy

There are many one and multidimensional interpolation functions in the module: `scipy.interpolate`. In this section, `interp1d()` and `lagrange()` interpolation functions are applied to the data given in Example 1.

```
>>> from scipy.interpolate import interp1d, lagrange
>>>  x = [0, 20, 40, 60, 80, 100]
>>> y = [26.0, 48.6, 61.6, 71.2, 74.8, 75.2]
>>> f = interp1d(x,y)    # creates the interpolation function
>>> print(f(20))
48.6
>>> print(f(80))
74.8
>>> print(f(50))
66.4
```

The value of y corresponding to x = 50 is equal to that obtained by using the linear interpolation because the default interpolation kind is 'linear'. Other kinds can be used for better evaluation.

```
>>> f = interp1d(x,y,'quadratic')
>>> print(f(50))
66.95208333333332
```

```
>>> f = interp1d(x,y,'cubic')
>>> print(f(50))
66.945
```

The cubic interpolation has resulted in a value of y equal to that obtained by Lagrange 5th degree polynomial.

```
>>> L = lagrange(x, y)        # creates the interpolation polynomial
>>> print(L)
           5             4             3           2
3.698e-08 x - 9.688e-06 x + 0.0009219 x - 0.04463 x + 1.725 x + 26
>>> L(50)
66.947656249999568
```

For more information about the module `scipy.interpolate` :

https://docs.scipy.org/doc/scipy/reference/interpolate.html

Curve fitting in SciPy

The linear regression can be made by using `linregress()` from the module `scipy.stats` of the statistical functions. It is applied here to solve Example 3.

```
>>> from scipy.stats import linregress
>>> x = [3, 4, 5, 6, 7, 8]
>>> y = [0, 7, 17, 26, 35, 45]
>>> print(linregress(x,y))
LinregressResult(slope=9.0857142857142854, intercept=-28.3047619047619,
rvalue=0.99906516809828694, pvalue=1.3104575468971522e-06,
stderr=0.19656921371950828)
```

If the slope and intercept are to be printed individually, the following method can be used

```
>>> L = linregress(x,y)
>>> L.slope
9.0857142857142854
>>> L.intercept
-28.3047619047619
```

For more information about `linregress()`:

https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.linregress.html

The polynomial curve fit is performed by the function `curve_fit()` from the module `scipy.optimize.` It uses non-linear least squares to fit a function, f, to data. So, in addition to data points, a model function should be given which accordingly the function that fits the data will be created. Example 4 will be solved by `curve_fit()` to compare the results.

```
>>> def f(x, a0, a1, a2):
...     return a0 + a1*x + a2*x**2   # Quadratic polynomial model
...
>>> x = [0, 1, 2, 3, 4, 5]
>>> y = [2, 8, 14, 28, 39, 62]
>>> a,b = curve_fit(f, x, y)
>>> print(a)
[ 2.67857143  2.25357143  1.875   ]  # polynomial coefficients
```

The variable b is a two-dimensional array of the estimated covariance of a. Let's try a the cubic function model,

```
>>> def f(x, a0, a1, a2, a3):
...     return a0 + a1*x + a2*x**2 + a3*x**3
...
>>> a,_ = curve_fit(f, x, y)
>>> print(a)
[  1.92857143e+00   5.67857143e+00  -2.17847962e-12   2.50000000e-01]
```

The underscore is put as a place holder instead of b to indicate to that its values are not required. The third coefficient is approximately equal to zero.

For more information about `curve_fit()`:

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html