Chapter 2
Approximation of a function

This chapter and the next examine the most commonly used methods in computational science. Here we concentrate on some basic aspects associated with numerical approximation of a function, interpolation, least-squares and spline approximations of a curve, and numerical representations of uniform and other distribution functions. We are only going to give an introductory description of these topics here as a preparation for other chapters and many of the issues will be revisited in a greater depth later. Note that some of the material covered here would require much more space if discussed thoroughly. For example, complete coverage of the issues involved in creating good random-number generators could form a separate book. Therefore, we only focus on the basics of the topics here.

2.1 Interpolation

In numerical analysis, the results obtained from computations are always approximations of the desired quantities and in most cases are within some uncertainties. This is similar to experimental observations in physics. Every single physical quantity measured carries some experimental error. We constantly encounter situations in which we need to interpolate a set of discrete data points or to fit them to an adjustable curve. It is extremely important for a physicist to be able to draw conclusions based on the information available and to generalize the knowledge gained in order to predict new phenomena.

Interpolation is needed when we want to infer some local information from a set of incomplete or discrete data. Overall approximation or fitting is needed when we want to know the general or global behavior of the data. For example, if the speed of a baseball is measured and recorded every $1/100$ of a second, we can then estimate the speed of the baseball at any moment by interpolating the recorded data around that time. If we want to know the overall trajectory, then we need to fit the data to a curve. In this section, we will discuss some very basic interpolation schemes and illustrate how to use them in physics.

Linear interpolation

Consider a discrete data set given from a discrete function $f_i = f(x_i)$ with $i = 0, 1, \ldots, n$. The simplest way to obtain the approximation of $f(x)$ for $x \in [x_i, x_{i+1}]$ is to construct a straight line between x_i and x_{i+1}. Then $f(x)$ is given by

$$f(x) = f_i + \frac{x - x_i}{x_{i+1} - x_i}(f_{i+1} - f_i) + \Delta f(x), \tag{2.1}$$

which of course is not accurate enough in most cases but serves as a good start in understanding other interpolation schemes. In fact, any value of $f(x)$ in the region $[x_i, x_{i+1}]$ is equal to the sum of the linear interpolation in the above equation and a quadratic contribution that has a unique curvature and is equal to zero at x_i and x_{i+1}. This means that the error $\Delta f(x)$ in the linear interpolation is given by

$$\Delta f(x) = \frac{\gamma}{2}(x - x_i)(x - x_{i+1}), \tag{2.2}$$

with γ being a parameter determined by the specific form of $f(x)$. If we draw a quadratic curve passing through $f(x_i)$, $f(a)$, and $f(x_{i+1})$, we can show that the quadrature

$$\gamma = f''(a), \tag{2.3}$$

with $a \in [x_i, x_{i+1}]$, as long as $f(x)$ is a smooth function in the region $[x_i, x_{i+1}]$; namely, the kth-order derivative $f^{(k)}(x)$ exists for any k. This is the result of the Taylor expansion of $f(x)$ around $x = a$ with the derivatives $f^{(k)}(a) = 0$ for $k > 2$. The maximum error in the linear interpolation of Eq. (2.1) is then bounded by

$$|\Delta f(x)| \leq \frac{\gamma_1}{8}(x_{i+1} - x_i)^2, \tag{2.4}$$

where $\gamma_1 = \max[|f''(x)|]$ with $x \in [x_i, x_{i+1}]$. The upper bound of the error in Eq. (2.4) is obtained from Eq. (2.2) with γ replaced by γ_1 and x solved from $d\Delta f(x)/dx = 0$. The accuracy of the linear interpolation can be improved by reducing the interval $h_i = x_{i+1} - x_i$. However, this is not always practical.

Let us take $f(x) = \sin x$ as an illustrative example here. Assuming that $x_i = \pi/4$ and $x_{i+1} = \pi/2$, we have $f_i = 0.707$ and $f_{i+1} = 1.000$. If we use the linear interpolation scheme to find the approximate value of $f(x)$ at $x = 3\pi/8$, we have the interpolated value $f(3\pi/8) \simeq 0.854$ from Eq. (2.1). We know, of course, that $f(3\pi/8) = \sin(3\pi/8) = 0.924$. The actual difference is $|\Delta f(x)| = 0.070$, which is smaller than the maximum error estimated with Eq. (2.4), 0.077.

The above example is a very simple one, showing how most interpolation schemes work. A continuous curve (a straight line in the above example) is constructed from the given discrete set of data and then the interpolated value is read off from the curve. The more points there are, the higher the order of the curve can be. For example, we can construct a quadratic curve from three

data points and a cubic curve from four data points. One way to achieve higher-order interpolation is through the Lagrange interpolation scheme, which is a generalization of the linear interpolation that we have just discussed.

The Lagrange interpolation

Let us first make an observation about the linear interpolation discussed in the preceding subsection. The interpolated function actually passes through the two points used for the interpolation. Now if we use three points for the interpolation, we can always construct a quadratic function that passes through all the three points. The error is now given by a term on the order of h^3, where h is the larger interval between any two nearest points, because an x^3 term could be added to modify the curve to pass through the function point if it were actually known. In order to obtain the generalized interpolation formula passing through $n+1$ data points, we rewrite the linear interpolation of Eq. (2.1) in a symmetric form with

$$\begin{aligned} f(x) &= \frac{x - x_{i+1}}{x_i - x_{i+1}} f_i + \frac{x - x_i}{x_{i+1} - x_i} f_{i+1} + \Delta f(x) \\ &= \sum_{j=i}^{i+1} f_j \, p_{1j}(x) + \Delta f(x), \end{aligned} \tag{2.5}$$

where

$$p_{1j}(x) = \frac{x - x_k}{x_j - x_k}, \tag{2.6}$$

with $k \neq j$. Now we can easily generalize the expression to an nth-order curve that passes through all the $n+1$ data points,

$$f(x) = \sum_{j=0}^{n} f_j \, p_{nj}(x) + \Delta f(x), \tag{2.7}$$

where $p_{nj}(x)$ is given by

$$p_{nj}(x) = \prod_{k \neq j}^{n} \frac{x - x_k}{x_j - x_k}. \tag{2.8}$$

In other words, $\Delta f(x_j) = 0$ at all the data points. Following a similar argument to that for linear interpolation in terms of the Taylor expansion, we can show that the error in the nth-order Lagrange interpolation is given by

$$\Delta f(x) = \frac{\gamma}{(n+1)!}(x - x_0)(x - x_1) \cdots (x - x_n), \tag{2.9}$$

where

$$\gamma = f^{(n+1)}(a), \tag{2.10}$$

with $a \in [x_0, x_n]$. Note that $f(a)$ is a point passed through by the $(n+1)$th-order curve that also passes through all the $f(x_i)$ with $i = 0, 1, \ldots, n$. Therefore, the maximum error is bounded by

$$|\Delta f(x)| \leq \frac{\gamma_n}{4(n+1)} h^{n+1}, \tag{2.11}$$

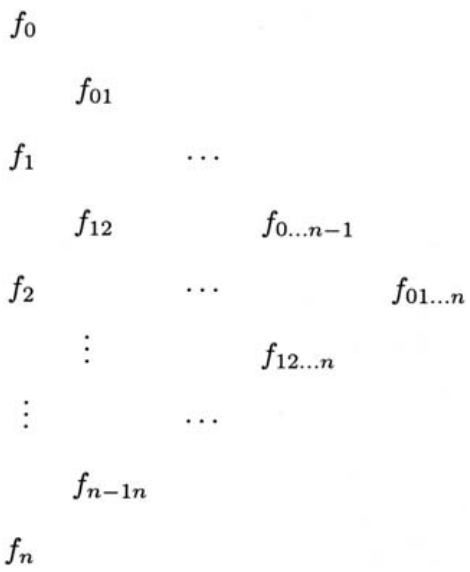

Fig. 2.1 The hierarchy in the Aitken scheme for $n+1$ data points.

where $\gamma_n = \max[|f^{(n+1)}(x)|]$ with $x \in [x_0, x_n]$ and h is the largest $h_i = x_{i+1} - x_i$. The above upper bound can be obtained by replacing γ with γ_n in Eq. (2.9) and then maximizing the pairs $(x - x_0)(x - x_n)$, $(x - x_1)(x - x_{n-1})$, ..., and $(x - x_{(n-1)/2})(x - x_{(n+1)/2})$ individually for an even $n + 1$. For an odd $n + 1$, we can choose the maximum value nh for the $x - x_i$ that is not paired. Equation (2.7) can be rewritten into a power series

$$f(x) = \sum_{k=0}^{n} a_k x^k + \Delta f(x), \tag{2.12}$$

with a_k given by expanding $p_{nj}(x)$ in Eq. (2.7). Note that the generalized form reduces to the linear case if $n = 1$.

The Aitken method

One way to achieve the Lagrange interpolation efficiently is by performing a sequence of linear interpolations. This scheme was first developed by Aitken (1932). We can first work out n linear interpolations with each constructed from a neighboring pair of the $n + 1$ data points. Then we can use these n interpolated data points to achieve another level of $n - 1$ linear interpolations with the next neighboring points of x_i. We repeat this process until we obtain the final result after n levels of consecutive linear interpolations. We can summarize the scheme in the following equation:

$$f_{i\ldots j} = \frac{x - x_j}{x_i - x_j} f_{i\ldots j-1} + \frac{x - x_i}{x_j - x_i} f_{i+1\ldots j}, \tag{2.13}$$

with $f_i = f(x_i)$ to start. If we want to obtain $f(x)$ from a given set f_i for $i = 0, 1, \ldots, n$, we can carry out n levels of consecutive linear interpolations as shown in Fig. 2.1, in which every column is constructed from the previous column by

Table 2.1. *Result of the example with the Aitken method*

x_i	f_i	f_{ij}	f_{ijk}	f_{ijkl}	f_{ijklm}
0.0	1.000 000				
		0.889 246			
0.5	0.938 470		0.808 792		
		0.799 852		0.807 272	
1.0	0.765 198		0.806 260		0.807 473
		0.815 872		0.807 717	
1.5	0.511 828		0.811 725		
		0.857 352			
2.0	0.223 891				

linear interpolations of the adjacent values. For example,

$$f_{012} = \frac{x - x_2}{x_0 - x_2} f_{01} + \frac{x - x_0}{x_2 - x_0} f_{12} \tag{2.14}$$

and

$$f_{01234} = \frac{x - x_4}{x_0 - x_4} f_{0123} + \frac{x - x_0}{x_4 - x_0} f_{1234}. \tag{2.15}$$

It can be shown that the consecutive linear interpolations outlined in Fig. 2.1 recover the standard Lagrange interpolation. The Aitken method also provides a way of estimating the error of the Lagrange interpolation. If we use the five-point case, that is, $n + 1 = 5$, as an illustrative example, the error in the Lagrange interpolation scheme is roughly given by

$$\Delta f(x) \approx \frac{|f_{01234} - f_{0123}| + |f_{01234} - f_{1234}|}{2}, \tag{2.16}$$

where the differences are taken from the last two columns of the hierarchy.

Let us consider the evaluation of $f(0.9)$, from the given set $f(0.0) = 1.000\,000$, $f(0.5) = 0.938\,470$, $f(1.0) = 0.765\,198$, $f(1.5) = 0.511\,828$, and $f(2.0) = 0.223\,891$, as an actual numerical example. These are the values of the zeroth-order Bessel function of the first kind, $J_0(x)$. All the consecutive linear interpolations of the data with the Aitken method are shown in Table 2.1.

The error estimated from the differences of the last two columns of the data in the table is

$$\Delta f(x) \approx \frac{|0.807\,473 - 0.807\,273| + |0.807\,473 - 0.807\,717|}{2} \simeq 2 \times 10^{-4}.$$

The exact result of $f(0.9)$ is 0.807 524. The error in the interpolated value is $|0.807\,473 - 0.807\,524| \simeq 5 \times 10^{-5}$, which is a little smaller than the estimated error from the differences of the last two columns in Table 2.1. The following

program is an implementation of the Aitken method for the Lagrange interpolation, using the given example of the Bessel function as a test.

```
// An example of extracting an approximate function
// value via the Lagrange interpolation scheme.

import java.lang.*;
public class Lagrange {
  public static void main(String argv[]) {
    double xi[] = {0, 0.5, 1, 1.5, 2};
    double fi[] = {1, 0.938470, 0.765198, 0.511828,
      0.223891};
    double x = 0.9;
    double f = aitken(x, xi, fi);
    System.out.println("Interpolated value: " + f);
  }

// Method to carry out the Aitken recursions.

  public static double aitken(double x, double xi[],
    double fi[]) {
    int n = xi.length-1;
    double ft[] = (double[]) fi.clone();
    for (int i=0; i<n; ++i) {
      for (int j=0; j<n-i; ++j) {
        ft[j] = (x-xi[j])/(xi[i+j+1]-xi[j])*ft[j+1]
               +(x-xi[i+j+1])/(xi[j]-xi[i+j+1])*ft[j];
      }
    }
    return ft[0];
  }
}
```

After running the above program, we obtain the expected result, $f(0.9) \simeq 0.807\,473$. Even though we have a quite accurate result here, the interpolation can be influenced significantly by the rounding error in some cases if the Aitken procedure is carried out directly. This is due to the change in the interpolated value being quite small compared to the actual value of the function during each step of the consecutive linear interpolations. When the number of data points involved becomes large, the rounding error starts to accumulate. Is there a better way to achieve the interpolation?

A better way is to construct an indirect scheme that improves the interpolated value at every step by updating the differences of the interpolated values from the adjacent columns, that is, by improving the corrections of the interpolated values over the preceding column rather than the interpolated values themselves. The effect of the rounding error is then minimized. This procedure is accomplished with the *up-and-down method*, which utilizes the upward and downward corrections

$$\Delta_{ij}^{+} = f_{j\ldots j+i} - f_{j+1\ldots j+i}, \tag{2.17}$$

$$\Delta_{ij}^{-} = f_{j\ldots j+i} - f_{j\ldots j+i-1}, \tag{2.18}$$

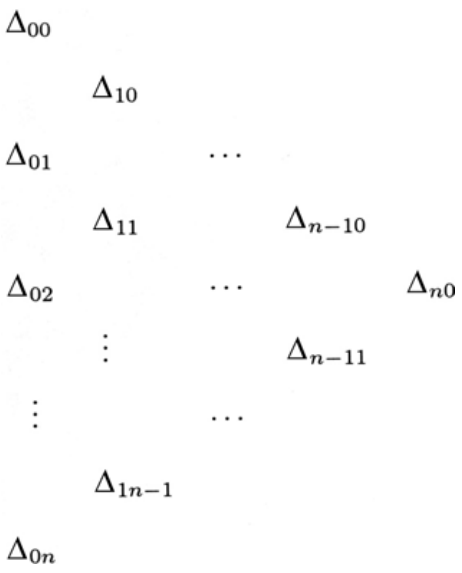

Fig. 2.2 The hierarchy for both Δ_{ij}^{+} and Δ_{ij}^{-} in the upward and downward correction method for $n+1$ data points.

defined at each step from the differences between two adjacent columns. Here Δ_{ij}^{-} is the downward (going down along the triangle in Fig. 2.1) correction and Δ_{ij}^{+} is the upward (going up along the triangle in Fig. 2.1) correction. The index i here is for the level of correction and j is for the element in each level. The hierarchy for both Δ_{ij}^{+} and Δ_{ij}^{-} is show in Fig. 2.2. It can be shown from the definitions of Δ_{ij}^{+} and Δ_{ij}^{-} that they satisfy the following recursion relations:

$$\Delta_{ij}^{+} = \frac{x_{i+j} - x}{x_{i+j} - x_j}(\Delta_{i-1j}^{+} - \Delta_{i-1j+1}^{-}), \tag{2.19}$$

$$\Delta_{ij}^{-} = \frac{x_j - x}{x_{i+j} - x_j}(\Delta_{i-1j}^{+} - \Delta_{i-1j+1}^{-}), \tag{2.20}$$

with the starting column $\Delta_{0j}^{\pm} = f_j$. Here x_j is chosen as the data point closest to x. In general, we use the upward correction as the first correction if $x < x_j$. Otherwise, the downward correction is used. Then we alternate the downward and upward corrections in the steps followed until the final result is reached. If the upper (lower) boundary of the triangle in Fig. 2.2 is reached during the process, only downward (upward) corrections can be used afterward.

We can use the numerical values of the Bessel function in Table 2.1 to illustrate the method. Assume that we are still calculating $f(x)$ with $x = 0.9$. It is easy to see that the starting point should be $x_j = 1.0$, because it is closest to $x = 0.9$. So the zeroth-order approximation of the interpolated data is $f(x) \approx f(1.0)$. The first correction to $f(x)$ is then Δ_{11}^{+}. In the next step, we alternate the direction of the correction and use the downward correction, Δ_{21}^{-} in this example, to improve $f(x)$ further. We can continue the procedure with another upward correction and another downward correction to reach the final result. We can write a simple program to accomplish what we have just described. It is a good practice to write a method, function, or subroutine, depending on the particular language used, with x, x_i, and f_i, for $i = 0, 1, \ldots, n$, being the input and $f(x)$ being the output. The

method can then be used for any interpolation task. Here is an implementation of the up-and-down method in Java.

```
// Method to complete the interpolation via upward and
// downward corrections.
  public static double upwardDownward(double x,
    double xi[], double fi[]) {
    int n = xi.length-1;
    double dp[][] = new double[n+1][];
    double dm[][] = new double[n+1][];
 // Assign the 1st columns of the corrections
    dp[0] = (double[]) fi.clone();
    dm[0] = (double[]) fi.clone();
 // Find the closest point to x
    int j0 = 0, k0 = 0;
    double dx = x-xi[0];
    for (int j=1; j<=n; ++j) {
      double dx1 = x-xi[j];
      if (Math.abs(dx1)<Math.abs(dx)) {
        j0 = j;
        dx = dx1;
      }
    }
    k0 = j0;
 // Evaluate the rest of the corrections recursively
    for (int i=1; i<=n; ++i) {
      dp[i] = new double[n-i+1];
      dm[i] = new double[n-i+1];
      for (int j=0; j<n-i+1; ++j) {
        double d = dp[i-1][j]-dm[i-1][j+1];
        d /= xi[i+j]-xi[j];
        dp[i][j] = d*(xi[i+j]-x);
        dm[i][j] = d*(xi[j]-x);
      }
    }
 // Update the interpolation with the corrections
    double f = fi[j0];
    for (int i=1; i<=n; ++i) {
      if (((dx<0)||(k0==n)) && (j0!=0)) {
        j0--;
        f += dp[i][j0];
        dx = -dx;
      }
      else {
        f += dm[i][j0];
        dx = -dx;
        k0++;
      }
    }
     return f;
  }
```

We can replace the Aitken method in the earlier example program with this method. The numerical result, with the input data from Table 2.1, is exactly the same as the earlier result, $f(0.9) = 0.807\,473$, as expected.

2.2 Least-squares approximation

As we have pointed out, interpolation is mainly used to find the local approximation of a given discrete set of data. In many situations in physics we need to know the global behavior of a set of data in order to understand the trend in a specific measurement or observation. A typical example is a polynomial fit to a set of experimental data with error bars.

The most common approximation scheme is based on the least squares of the differences between the approximation $p_m(x)$ and the data $f(x)$. If $f(x)$ is the data function to be approximated in the region $[a, b]$ and the approximation is an mth-order polynomial

$$p_m(x) = \sum_{k=0}^{m} a_k x^k, \tag{2.21}$$

we can construct a function of a_k for $k = 0, 1, \ldots, m$ as

$$\chi^2[a_k] = \int_a^b [\, p_m(x) - f(x)]^2 \, dx \tag{2.22}$$

for the continuous data function $f(x)$, and

$$\chi^2[a_k] = \sum_{i=0}^{n} [\, p_m(x_i) - f(x_i)]^2 \tag{2.23}$$

for the discrete data function $f(x_i)$ with $i = 0, 1, \ldots, n$. Here we have used a generic variable a_k inside a square bracket for a quantity that is a function of a set of independent variables $a_0, a_1, \ldots, a_m$. This notation will be used throughout this book. Here χ^2 is the conventional notation for the summation of the squares of the deviations.

The least-squares approximation is obtained with $\chi^2[a_k]$ minimized with respect to all the $m + 1$ coefficients through

$$\frac{\partial \chi^2[a_k]}{\partial a_l} = 0, \tag{2.24}$$

for $l = 0, 1, 2, \ldots, m$. The task left is to solve this set of $m + 1$ linear equations to obtain all the a_l. This general problem of solving a linear equation set will be discussed in detail in Chapter 5. Here we consider a special case with $m = 1$, that is, the linear fit. Then we have

$$p_1(x) = a_0 + a_1 x, \tag{2.25}$$

with

$$\chi^2[a_k] = \sum_{i=0}^{n} (a_0 + a_1 x_i - f_i)^2 \,. \tag{2.26}$$

From Eq. (2.24), we obtain

$$(n + 1)a_0 + c_1 a_1 - c_3 = 0, \tag{2.27}$$

$$c_1 a_0 + c_2 a_1 - c_4 = 0, \tag{2.28}$$

where $c_1 = \sum_{i=0}^{n} x_i$, $c_2 = \sum_{i=0}^{n} x_i^2$, $c_3 = \sum_{i=0}^{n} f_i$, and $c_4 = \sum_{i=0}^{n} x_i f_i$. Solving these two equations together, we obtain

$$a_0 = \frac{c_1 c_4 - c_2 c_3}{c_1^2 - (n+1)c_2}, \tag{2.29}$$

$$a_1 = \frac{c_1 c_3 - (n+1)c_4}{c_1^2 - (n+1)c_2}. \tag{2.30}$$

We will see an example of implementing this linear approximation in the analysis of the data from the Millikan experiment in the next section. Note that this approach becomes very involved when m becomes large.

Here we change the strategy and tackle the problem with orthogonal polynomials. In principle, we can express the polynomial $p_m(x)$ in terms of a set of orthogonal polynomials with

$$p_m(x) = \sum_{k=0}^{m} \alpha_k u_k(x), \tag{2.31}$$

where $u_k(x)$ is a set of real orthogonal polynomials that satisfy

$$\int_a^b u_k(x) w(x) u_l(x)\, dx = \langle u_k | u_l \rangle = \delta_{kl} \mathcal{N}_k, \tag{2.32}$$

with $w(x)$ being the weight whose form depends on the specific set of orthogonal polynomials. Here δ_{kl} is the Kronecker δ function, which is 1 for $k = l$ and 0 for $k \neq l$, and $\mathcal{N}_k$ is a normalization constant. The coefficients α_k can be formally related to a_j by a matrix transformation, and are determined with $\chi^2[\alpha_k]$ minimized. Note that $\chi^2[\alpha_k]$ is the same quantity defined in Eq. (2.22) or Eq. (2.23) with $p_m(x)$ from Eq. (2.31). If we want the polynomials to be orthonormal, we can simply divide $u_k(x)$ by $\sqrt{\mathcal{N}_k}$. We will also use the notation in the above equation for the discrete case with

$$\langle u_k | u_l \rangle = \sum_{i=0}^{n} u_k(x_i) w(x_i) u_l(x_i) = \delta_{kl} \mathcal{N}_k. \tag{2.33}$$

The orthogonal polynomials can be generated with the following recursion:

$$u_{k+1}(x) = (x - g_k) u_k(x) - h_k u_{k-1}(x), \tag{2.34}$$

where the coefficients g_k and h_k are given by

$$g_k = \frac{\langle x u_k | u_k \rangle}{\langle u_k | u_k \rangle}, \tag{2.35}$$

$$h_k = \frac{\langle x u_k | u_{k-1} \rangle}{\langle u_{k-1} | u_{k-1} \rangle}, \tag{2.36}$$

with the starting $u_0(x) = 1$ and $h_0 = 0$. We can take the above polynomials and show that they are orthogonal regardless of whether they are continuous or discrete; they always satisfy $\langle u_k | u_l \rangle = \delta_{kl} \mathcal{N}_k$. For simplicity, we will just consider the case with $w(x) = 1$. The formalism developed here can easily be generalized to the cases with $w(x) \neq 1$. We will have more discussions on orthogonal polynomials in Chapter 6 when we introduce special functions and Gaussian quadratures.

The least-squares approximation is obtained if we find all the coefficients α_k that minimize the function $\chi^2[\alpha_k]$. In other words, we would like to have

$$\frac{\partial \chi^2[\alpha_k]}{\partial \alpha_j} = 0 \tag{2.37}$$

and

$$\frac{\partial^2 \chi^2[\alpha_k]}{\partial \alpha_j^2} > 0, \tag{2.38}$$

for $j = 0, 1, \ldots, m$. The first-order derivative of $\chi^2[\alpha_k]$ can easily be obtained. After exchanging the summation and the integration in $\partial \chi^2[\alpha_k]/\partial \alpha_j = 0$, we have

$$\alpha_j = \frac{\langle u_j | f \rangle}{\langle u_j | u_j \rangle}, \tag{2.39}$$

which ensures a minimum value of $\chi^2[\alpha_k]$, because $\partial^2 \chi^2[\alpha_k]/\partial \alpha_j^2 = 2\langle u_j | u_j \rangle$ is always greater than zero. We can always construct a set of discrete orthogonal polynomials numerically in the region $[a, b]$. The following method is a simple example for obtaining a set of orthogonal polynomials $u_k(x_i)$ and the coefficients α_k for a given set of discrete data f_i at data points x_i.

```
// Method to generate the orthogonal polynomials and the
// least-squares fitting coefficients.

  public static double[][] orthogonalPolynomialFit
    (int m, double x[], double f[]) {
    int n = x.length-1;
    double u[][] = new double[m+1][n+2];
    double s[] = new double[n+1];
    double g[] = new double[n+1];
    double h[] = new double[n+1];

 // Check and fix the order of the curve
    if (m>n) {
      m = n;
      System.out.println("The highest power"
        + " is adjusted to: " + n);
    }

 // Set up the zeroth-order polynomial
    for (int i=0; i<=n; ++i) {
      u[0][i] = 1;
      double stmp = u[0][i]*u[0][i];
      s[0] += stmp;
      g[0] += x[i]*stmp;
      u[0][n+1] += u[0][i]*f[i];
    }
    g[0] = g[0]/s[0];
    u[0][n+1] = u[0][n+1]/s[0];

 // Set up the first-order polynomial
    for (int i=0; i<=n; ++i) {
      u[1][i] = x[i]*u[0][i]-g[0]*u[0][i];
```

```
      s[1] += u[1][i]*u[1][i];
      g[1] += x[i]*u[1][i]*u[1][i];
      h[1] += x[i]*u[1][i]*u[0][i];
      u[1][n+1] += u[1][i]*f[i];
    }
    g[1] = g[1]/s[1];
    h[1] = h[1]/s[0];
    u[1][n+1] = u[1][n+1]/s[1];
// Obtain the higher-order polynomials recursively
    if (m >= 2) {
      for (int i=1; i<m; ++i) {
        for (int j=0; j<=n; ++j) {
          u[i+1][j] = x[j]*u[i][j]-g[i]*u[i][j]
            -h[i]*u[i-1][j];
          s[i+1] += u[i+1][j]*u[i+1][j];
          g[i+1] += x[j]*u[i+1][j]*u[i+1][j];
          h[i+1] += x[j]*u[i+1][j]*u[i][j];
          u[i+1][n+1] += u[i+1][j]*f[j];
        }
        g[i+1] = g[i+1]/s[i+1];
        h[i+1] = h[i+1]/s[i];
        u[i+1][n+1] = u[i+1][n+1]/s[i+1];
      }
    }
    return u;
  }
```

Note that this method is only for discrete functions, with both the polynomials and least-squares fitting coefficients returned in a combined matrix. This is the typical case encountered in data analysis in science. Even with a continuous variable, it is a common practice to discretize the data first and then perform the curve fitting. However, if we must deal with continuous functions without discretization, we can generate the orthogonal polynomials with a continuous variable and apply an integration quadrature for the evaluation of the integrals involved. We will discuss the methods for generating continuous polynomials in Chapter 6. Integration quadratures will be introduced in the next section with several practical examples. A more elaborate scheme called Gaussian quadratures will be studied in Section 6.9.

2.3 The Millikan experiment

Here we use a set of data from the famous oil drop experiment of Millikan as an example to illustrate how we can actually employ the least-squares approximation discussed above. Millikan (1910) published his famous work on the oil drop experiment in *Science*. Based on the measurements of the charges carried by all the oil drops, Millikan concluded that the charge carried by any object is a multiple (with a sign) of a fundamental charge, the charge of an electron (for negative charges) or the charge of a proton (for positive charges). In the article, Millikan extracted the fundamental charge by taking the average of the measured

Table 2.2. *Data from the Millikan experiment*

k	4	5	6	7	8	9	10	11
q_k	6.558	8.206	9.880	11.50	13.14	14.82	16.40	18.04
k	12	13	14	15	16	17	18	
q_k	19.68	21.32	22.96	24.60	26.24	27.88	29.52	

charges carried by all the oil drops. Here we are going to take the data of Millikan and make a least-squares fit to a straight line. Based on the fit, we can estimate the fundamental charge and the accuracy of the Millikan measurement. Millikan did not use the method that we are discussing here to reach his conclusion, of course.

Each measured data point from the Millikan experiment is assigned an integer. The measured charges q_k (in units of 10^{-19} C) and the corresponding integers k are listed in Table 2.2.

From the average charges of the oil drops, Millikan concluded that the fundamental charge is $e = 1.65 \times 10^{-19}$ C, which is very close to the currently accepted value of the fundamental charge, $e = 1.602\,177\,33(49) \times 10^{-19}$ C. Let us take the Millikan's data obtained in Table 2.2 and apply the least-squares approximation discussed in the preceding section for a discrete function to find the fundamental charge and the order of the error in the measurements. We can take the linear equation

$$q_k = ke + \Delta q_k \tag{2.40}$$

as the approximation of the measured data. For simplicity, Δq_k is taken as a constant Δq to represent the overall error associated with the measurement. We leave the problem with different Δq_k as an exercise for the reader. Note that if we fit the data to a straight line in the xy plane, Δq is the intercept on the y axis. The following program applies the least-squares approximation and calculates the fundamental charge e and the overall error Δq using the data from the Millikan experiment.

```
// An example of applying the least-squares approximation
// to the Millikan data on a straight line q=k*e+dq.

import java.lang.*;
public class Millikan {
  public static void main(String argv[]) {
    double k[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
      15, 16, 17, 18};
    double q[] = {6.558, 8.206, 9.880, 11.50, 13.14,
      14.81, 16.40, 18.04, 19.68, 21.32, 22.96, 24.60,
      26.24, 27.88, 29.52};
    int n = k.length-1;
    int m = 21;
    double u[][] = orthogonalPolynomialFit(m, k, q);
```

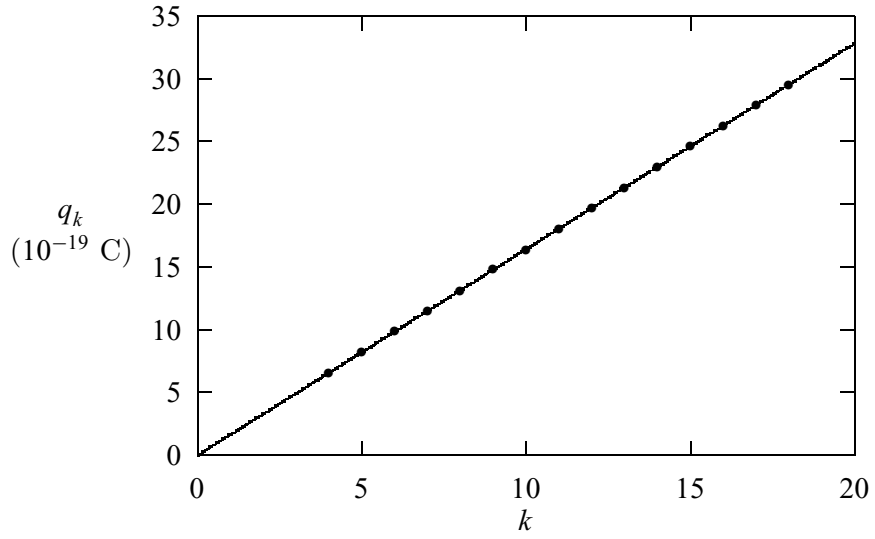

Fig. 2.3 The Millikan measurements of charges carried by the oil drops (dots) and the least-squares approximation of a linear fit (solid line).

```
    double sum = 0;
    for (int i=0; i<=n; ++i) sum += k[i];
    double e = u[1][n+1];
    double dq = u[0][n+1]-u[1][n+1]*sum/(n+1);
    System.out.println("Fundamental charge: " + e);
    System.out.println("Estimated error: " + dq);
  }

  public static double[][] orthogonalPolynomialFit
    (int m, double x[], double f[]) {...}
}
```

Note that we have used . . . to represent the body of a method that we introduced earlier to avoid repeating the material. We will do this throughout the book, but the same programs are listed in their entirety on the website for the book. We have also used the fact that $u_0 = 1$, $u_1(x) = x - g_0$, and $g_0 = \sum_{i=0}^{n} x_i/(n+1)$ from Eqs. (2.34) and (2.35) in the above program. So from the linear function $f(x) = a_0 + a_1 x = \alpha_0 + \alpha_1 u_1(x)$, we know that $a_1 = \alpha_1$ and $a_0 = \alpha_0 - g_0 = \alpha_0 - \sum_{i=0}^{n} x_i/(n+1)$ after taking $x = 0$.

After we compile and run the above program, we obtain $e \simeq 1.64 \times 10^{-19}$ C, and the intercept on the y axis gives us a rough estimate of the error bar $|\Delta e| \approx |\Delta q| = 0.03 \times 10^{-19}$ C. The Millikan data and the least-squares approximation of Eq. (2.40) with e and Δq obtained from the above program are plotted in Fig. 2.3. Note that the measured data are very accurately represented by the straight line of the least-squares approximation.

The approach here is very general and we can easily fit the Millikan data to a higher-order curve by changing $m = 1$ to a higher value, for example, $m = 21$, as we actually did in the above program. After running the program, we found that other coefficients are vanishingly small. However, if the data set is nonlinear, other coefficients will become significant.

For the linear fit of the Millikan data, we can take a much simpler approach, like the one given in Eqs. (2.25)–(2.30). The following program is the result of such an approach.

```
// An example of directly fitting the Millikan data to
// p1(x) = a0+a1*x.

import java.lang.*;
public class Millikan2 {
  public static void main(String argv[]) {
    double x[] = {4, 5, 6, 7, 8, 9, 10, 11,
      12, 13, 14, 15, 16, 17, 18};
    double f[] = {6.558, 8.206, 9.880, 11.50,
      13.14, 14.81, 16.40, 18.04, 19.68, 21.32,
      22.96, 24.60, 26.24, 27.88, 29.52};
    int n = x.length-1;
    double c1 = 0, c2 = 0, c3 = 0, c4 = 0;
    for (int i=0; i<=n; ++i) {
      c1 += x[i];
      c2 += x[i]*x[i];
      c3 += f[i];
      c4 += f[i]*x[i];
    }
    double g = c1*c1-c2*(n+1);
    double a0 = (c1*c4-c2*c3)/g;
    double a1 = (c1*c3-c4*(n+1))/g;
    System.out.println("Fundamental charge: " + a1);
    System.out.println("Estimated error: " + a0);
  }
}
```

Of course, we end up with exactly the same result. The point is that sometimes we may need a general approach because we have more than one problem in mind, but other times we may need a direct approach that is simple and fast. We must learn to be able to deal with problems at different levels of complexity accordingly. In certain problems this can make the difference between finding a solution or not.

2.4 Spline approximation

In many instances, we have a set of data that varies rapidly over the range of interest, for example, a typical spectral measurement that contains many peaks and dips. Then there is no such thing as a global polynomial behavior. In such situations, we want to fit the function locally and to connect each piece of the function smoothly. A *spline* is such a tool; it interpolates the data locally through a polynomial and fits the data overall by connecting each segment of the interpolation polynomial by matching the function and its derivatives at the data points.

Assuming that we are trying to create a spline function that approximates a discrete data set $f_i = f(x_i)$ for $i = 0, 1, \ldots, n$, we can use an mth-order

polynomial

$$p_i(x) = \sum_{k=0}^{m} c_{ik} x^k \tag{2.41}$$

to approximate $f(x)$ for $x \in [x_i, x_{i+1}]$. Then the coefficients c_{ik} are determined from the smoothness conditions at the nonboundary data points with the lth-order derivative there satisfying

$$p_i^{(l)}(x_{i+1}) = p_{i+1}^{(l)}(x_{i+1}), \tag{2.42}$$

for $l = 0, 1, \ldots, m-1$. These conditions and the values $p_i(x_i) = f_i$ provide $(m+1)(n-1)$ equations from the nonboundary points. So we still need $m+1$ equations in order to solve all the $(m+1)n$ coefficients a_{ik}. Two additional equations $p_0(x_0) = f_0$ and $p_{n-1}(x_n) = f_n$ are obvious and the remaining $m-1$ equations are provided by the choice of some of $p_0^{(l)}(x_0)$ and $p_{n-1}^{(l)}(x_n)$ for $l = m-1, m-2, \ldots$.

The most widely adopted spline function is the cubic spline with $m = 3$. In this case, the number of equations needed from the derivatives of the polynomials at the boundary points is $m - 1 = 2$. One of the choices, called the natural spline, is made by setting the highest-order derivatives to zero at both ends up to the number of equations needed. For the cubic spline, the natural spline is given by the choices of $p_0''(x_0) = 0$ and $p_{n-1}''(x_n) = 0$.

To construct the cubic spline, we can start with the linear interpolation of the second-order derivative in $[x_i, x_{i+1}]$,

$$p_i''(x) = \frac{1}{x_{i+1} - x_i}[(x - x_i)p_{i+1}'' - (x - x_{i+1})p_i''], \tag{2.43}$$

where $p_i'' = p_i''(x_i) = p_{i-1}''(x_i)$ and $p_{i+1}'' = p_{i+1}''(x_{i+1}) = p_i''(x_{i+1})$. If we integrate the above equation twice and use $p_i(x_i) = f_i$ and $p_i(x_{i+1}) = f_{i+1}$, we obtain

$$p_i(x) = \alpha_i(x - x_i)^3 + \beta_i(x - x_{i+1})^3 + \gamma_i(x - x_i) + \eta_i(x - x_{i+1}), \tag{2.44}$$

where

$$\alpha_i = \frac{p_{i+1}''}{6h_i}, \tag{2.45}$$

$$\beta_i = -\frac{p_i''}{6h_i}, \tag{2.46}$$

$$\gamma_i = \frac{f_{i+1}}{h_i} - \frac{h_i p_{i+1}''}{6}, \tag{2.47}$$

$$\eta_i = \frac{h_i p_i''}{6} - \frac{f_i}{h_i}, \tag{2.48}$$

with $h_i = x_{i+1} - x_i$. So if we find all p_i'', we find the spline. Applying the condition

$$p_{i-1}'(x_i) = p_i'(x_i) \tag{2.49}$$

to the polynomial given in Eq. (2.44), we have

$$h_{i-1}p''_{i-1} + 2(h_{i-1}+h_i)p''_i + h_i p''_{i+1} = 6\left(\frac{g_i}{h_i} - \frac{g_{i-1}}{h_{i-1}}\right), \tag{2.50}$$

where $g_i = f_{i+1} - f_i$. This is a linear equation set with $n-1$ unknowns p''_i for $i = 1, 2, \ldots, n-1$. Note that the boundary values are fixed by $p''_0 = p''_n = 0$.

We can write the above equations in a matrix form

$$\begin{pmatrix} d_1 & h_1 & 0 & \cdots & \cdots & 0 \\ h_1 & d_2 & h_2 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & h_{n-3} & d_{n-2} & h_{n-2} \\ 0 & \cdots & \cdots & 0 & h_{n-2} & d_{n-1} \end{pmatrix} \begin{pmatrix} p''_1 \\ p''_2 \\ \vdots \\ p''_{n-2} \\ p''_{n-1} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{pmatrix}, \tag{2.51}$$

or equivalently,

$$\mathbf{A}\mathbf{p}'' = \mathbf{b}, \tag{2.52}$$

where $d_i = 2(h_{i-1} + h_i)$ and $b_i = 6(g_i/h_i - g_{i-1}/h_{i-1})$. The coefficient matrix $\mathbf{A}$ in this problem is a real, symmetric, and tridiagonal matrix with elements

$$A_{ij} = \begin{cases} d_i & \text{if } i = j, \\ h_i & \text{if } i = j-1, \\ h_{i-1} & \text{if } i = j+1, \\ 0 & \text{otherwise.} \end{cases} \tag{2.53}$$

Because of the simplicity of the coefficient matrix, the solution of the equation set becomes quite straightforward. Here we will limit ourselves to the problem with the coefficients given in a tridiagonal matrix and leave the solution of a general linear equation set to Chapter 5.

In general, we can decompose an $m \times m$ square matrix $\mathbf{A}$ into a product of a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$,

$$\mathbf{A} = \mathbf{L}\mathbf{U}, \tag{2.54}$$

with $L_{ij} = 0$ for $i < j$ and $U_{ij} = 0$ for $i > j$. We can choose either $U_{ii} = 1$ or $L_{ii} = 1$. This scheme is called the LU decomposition. The choice of $U_{ii} = 1$ is called the Crout factorization and the alternative choice of $L_{ii} = 1$ is called the Doolittle factorization. Here we work out the Crout factorization and leave the Doolittle factorization as an exercise. For a tridiagonal matrix $\mathbf{A}$ with

$$A_{ij} = \begin{cases} d_i & \text{if } i = j, \\ e_i & \text{if } i = j-1, \\ c_{i-1} & \text{if } i = j+1, \\ 0 & \text{otherwise,} \end{cases} \tag{2.55}$$

the matrices **L** and **U** are extremely simple and are given by

$$L_{ij} = \begin{cases} w_i & \text{if } i = j, \\ v_{i-1} & \text{if } i = j+1, \\ 0 & \text{otherwise}, \end{cases} \tag{2.56}$$

and

$$U_{ij} = \begin{cases} 1 & \text{if } i = j, \\ t_i & \text{if } i = j-1, \\ 0 & \text{otherwise}. \end{cases} \tag{2.57}$$

The elements in **L** and **U** can be related easily to d_i, c_i, and e_i if we multiply **L** and **U** and compare the two sides of Eq. (2.54) element by element. Then we have

$$v_i = c_i, \tag{2.58}$$

$$t_i = e_i/w_i, \tag{2.59}$$

$$w_i = d_i - v_{i-1}t_{i-1}, \tag{2.60}$$

with $w_1 = d_1$. The solution of the linear equation $\mathbf{Az} = \mathbf{b}$ can be obtained by forward and backward substitutions because

$$\mathbf{Az} = \mathbf{LUz} = \mathbf{Ly} = \mathbf{b}. \tag{2.61}$$

We can first solve $\mathbf{Ly} = \mathbf{b}$ and then $\mathbf{Uz} = \mathbf{y}$ with

$$y_i = (b_i - v_{i-1}y_{i-1})/w_i, \tag{2.62}$$

$$z_i = y_i - t_i z_{i+1}, \tag{2.63}$$

with $y_1 = b_1/w_1$ and $z_m = y_m$. The following program is an implementation of the cubic spline with the solution of the tridiagonal linear equation set through Crout factorization.

```
// An example of creating cubic-spline approximation of
// a discrete function fi=f(xi).

import java.lang.*;
import java.io.*;
import java.util.*;
public class Spline {
  final static int n = 20;
  final static int m = 100;
  public static void main(String argv[]) throws
    IOException {
    double xi[] = new double[n+1];
    double fi[] = new double[n+1];
    double p2[] = new double[n+1];

 // Read in data points xi and and data fi
    BufferedReader r = new BufferedReader
      (new InputStreamReader
```

```
        (new FileInputStream("xy.data")));
    for (int i=0; i<=n; ++i) {
      StringTokenizer
        s = new StringTokenizer(r.readLine());
      xi[i] = Double.parseDouble(s.nextToken());
      fi[i] = Double.parseDouble(s.nextToken());
    }
    p2 = cubicSpline(xi, fi);

 // Find the approximation of the function
    double h = (xi[n]-xi[0])/m;
    double x = xi[0];
    for (int i=1; i<m; ++i) {
      x += h;

   // Find the interval where x resides
      int k = 0;
      double dx = x-xi[0];
      while (dx > 0) {
        ++k;
        dx = x-xi[k];
      }
      --k;

   // Find the value of function f(x)
      dx = xi[k+1]-xi[k];
      double alpha = p2[k+1]/(6*dx);
      double beta = -p2[k]/(6*dx);
      double gamma = fi[k+1]/dx-dx*p2[k+1]/6;
      double eta = dx*p2[k]/6 - fi[k]/dx;
      double f = alpha*(x-xi[k])*(x-xi[k])*(x-xi[k])
        +beta*(x-xi[k+1])*(x-xi[k+1])*(x-xi[k+1])
        +gamma*(x-xi[k])+eta*(x-xi[k+1]);
      System.out.println(x + " " + f);
    }
  }

// Method to carry out the cubic-spline approximation
// with the second-order derivatives returned.

  public static double[] cubicSpline(double x[],
    double f[]) {
    int n = x.length-1;
    double p[] = new double [n+1];
    double d[] = new double [n-1];
    double b[] = new double [n-1];
    double c[] = new double [n-1];
    double g[] = new double [n];
    double h[] = new double [n];

 // Assign the intervals and function differences
    for (int i=0; i<n; ++i) {
      h[i] = x[i+1]-x[i];
      g[i] = f[i+1]-f[i];
    }

 // Evaluate the coefficient matrix elements
    for (int i=0; i<n-1; ++i) {
      d[i] = 2*(h[i+1]+h[i]);
```

```
      b[i] = 6*(g[i+1]/h[i+1]-g[i]/h[i]);
      c[i] = h[i+1];
    }

 // Obtain the second-order derivatives
    g = tridiagonalLinearEq(d, c, c, b);
    for (int i=1; i<n; ++i) p[i] = g[i-1];
    return p;
  }

// Method to solve the tridiagonal linear equation set.

  public static double[] tridiagonalLinearEq(double d[],
    double e[], double c[], double b[]) {
    int m = b.length;
    double w[] = new double[m];
    double y[] = new double[m];
    double z[] = new double[m];
    double v[] = new double[m-1];
    double t[] = new double[m-1];

 // Evaluate the elements in the LU decomposition
    w[0] = d[0];
    v[0] = c[0];
    t[0] = e[0]/w[0];
    for (int i=1; i<m-1; ++i) {
      w[i]  = d[i]-v[i-1]*t[i-1];
      v[i]  = c[i];
      t[i]  = e[i]/w[i];
    }
    w[m-1]  = d[m-1]-v[m-2]*t[m-2];

 // Forward substitution to obtain y
    y[0] = b[0]/w[0];
    for (int i=1; i<m; ++i)
      y[i] = (b[i]-v[i-1]*y[i-1])/w[i];

 // Backward substitution to obtain z
    z[m-1] = y[m-1];
    for (int i=m-2; i>=0; --i) {
      z[i] = y[i]-t[i]*z[i+1];
    }
    return z;
  }
}
```

We have used a set of uniform random numbers from the generator in next section as the data function at uniformly spaced data points in [0, 1]. The output of the above program is plotted together with the original data in Fig. 2.4.

The above approach can be generalized to a higher-order spline. For example, the *quintic spline* is achieved by including forth-order and fifth-order terms. Then the polynomial in the interval $[x_i, x_{i+1}]$ is given by

$$p_i(x) = \alpha_i(x - x_i)^5 + \beta_i(x - x_{i+1})^5 + \gamma_i(x - x_i)^3 + \eta_i(x - x_{i+1})^3 + \delta_i(x - x_i) + \sigma_i(x - x_{i+1}), \tag{2.64}$$

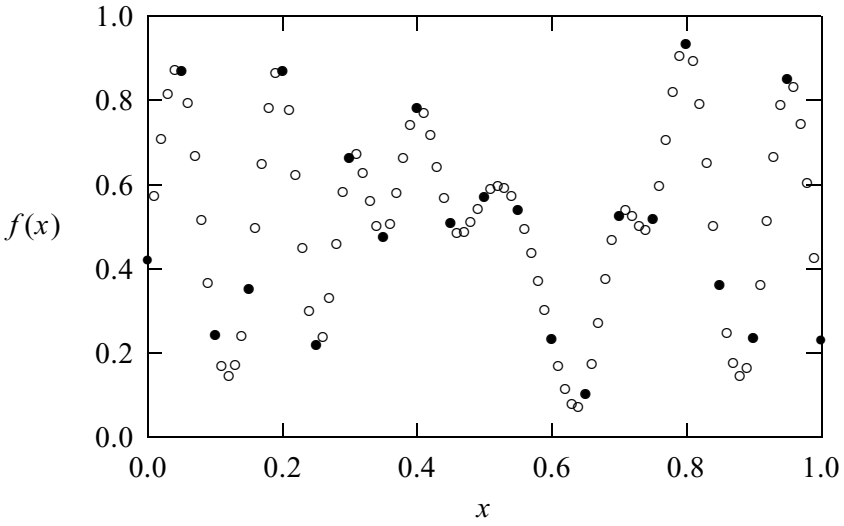

Fig. 2.4 An example of a cubic-spline approximation. The original data are shown as solid circles and the approximations as open circles.

where

$$\alpha_i = \frac{p_{i+1}^{(4)}}{120h_i}, \tag{2.65}$$

$$\beta_i = -\frac{p_i^{(4)}}{120h_i}. \tag{2.66}$$

We must also have $p_i(x_i) = f_i$ and $p_i(x_{i+1}) = f_{i+1}$, which give

$$h_i^5\alpha_i + h_i^3\gamma_i + h_i\delta_i = f_{i+1}, \tag{2.67}$$

$$h_i^5\beta_i + h_i^3\eta_i + h_i\sigma_i = -f_i, \tag{2.68}$$

where $h_i = x_{i+1} - x_i$. In order for the pieces to join smoothly, we also impose that $p_{i-1}^{(l)}(x_i) = p_i^{(l)}(x_i)$ for $l = 1, 2, 3$. From $p'_{i-1}(x_i) = p'_i(x_i)$, we have

$$5h_{i-1}^4\alpha_{i-1} + 3h_{i-1}^2\gamma_{i-1} + \delta_{i-1} + \sigma_{i-1} = 5h_i^4\beta_i + 3h_i^2\eta_i + \delta_i + \sigma_i. \tag{2.69}$$

For $l = 2$, we have

$$10h_{i-1}^3\alpha_{i-1} + 3h_{i-1}\gamma_{i-1} = 10h_i^3\beta_i + 3h_i\eta_i. \tag{2.70}$$

The continuity of the third-order derivative $p_{i-1}^{(3)}(x_i) = p_i^{(3)}(x_i)$ leads to

$$10h_{i-1}^2\alpha_{i-1} + \gamma_{i-1} = 10h_i^2\beta_i + \eta_i. \tag{2.71}$$

Note that we always have $p_i^{(4)} = 120h_{i-1}\alpha_{i-1} = -120h_i\beta_i$. Then from the equations with $l = 2$ and $l = 3$, we obtain

$$\gamma_i = -\frac{h_i^2 + 3h_ih_{i+1} + 2h_{i+1}^2}{36(h_i + h_{i+1})}p_{i+1}^{(4)}, \tag{2.72}$$

$$\eta_i = \frac{2h_{i-1}^2 + 3h_{i-1}h_i + 2h_i^2}{36(h_{i-1} + h_i)}p_i^{(4)}. \tag{2.73}$$

Then we can use Eqs. (2.67) and (2.68) to obtain δ_i and σ_i in terms of $p_i^{(4)}$. Substituting these into Eq. (2.69), we arrive at the linear equation set for $p_i^{(4)}$ with a coefficient matrix in tridiagonal form that can be solved as done for the cubic spline.

2.5 Random-number generators

In practice, many numerical simulations need random-number generators either for setting up initial configurations or for generating new configurations. There is no such thing as *random* in a computer program. A computer will always produce the same result if the input is the same. A random-number generator here really means a pseudo-random-number generator that can generate a long sequence of numbers that can imitate a given distribution. In this section we will discuss some of the basic random-number generators used in computational physics and other computer simulations.

Uniform random-number generators

The most useful random-number generators are those with a uniform distribution in a given region. The three most important criteria for a good uniform random-number generator are the following (Park and Miller, 1988).

First, a good generator should have a long period, which should be close to the range of the integers adopted. For example, if 32-bit integers are used, a good generator should have a period close to $2^{31} - 1 = 2\,147\,483\,647$. The range of the 32-bit integers is $[-2^{31}, 2^{31} - 1]$. Note that one bit is used for the sign of the integers.

Second, a good generator should have the best *randomness*. There should only be a very small correlation among all the numbers generated in sequence. If $\langle A \rangle$ represents the statistical average of the variable A, the k-point correlation function of the numbers generated in the sequence $\langle x_{i_1} x_{i_2} \cdots x_{i_k} \rangle$ for $k = 2, 3, 4, \ldots$ should be very small. One way to illustrate the behavior of the correlation function $\langle x_i x_{i+l} \rangle$ is to plot x_i and x_{i+l} in the xy plane. A good random-number generator will have a very uniform distribution of the points for any $l \neq 0$. A poor generator may show stripes, lattices, or other inhomogeneous distributions.

Finally, a good generator has to be very fast. In practice, we need a lot of random numbers in order to have good statistical results. The speed of the generator can become a very important factor.

The simplest uniform random-number generator is built using the so-called linear congruent scheme. The random numbers are generated in sequence from the linear relation

$$x_{i+1} = (a x_i + b) \bmod c, \tag{2.74}$$

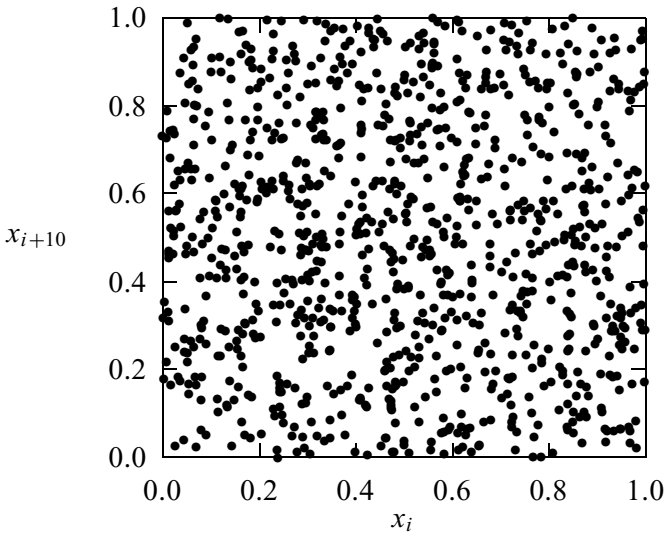

Fig. 2.5 One thousand points of the random-number pairs (x_i, x_{i+10}) normalized to the range of [0, 1].

where a, b, and c are *magic numbers*: their values determine the quality of the generator. One common choice, $a = 7^5 = 16\,807$, $b = 0$, and $c = 2^{31} - 1 = 2\,147\,483\,647$, has been tested and found to be excellent for generating unsigned 32-bit random integers. It has the full period of $2^{31} - 1$ and is very fast. The correlation function $\langle x_{i_1} x_{i_2} \ldots x_{i_k} \rangle$ is very small. In Fig. 2.5, we plot x_i and x_{i+10} (normalized by c) generated using the linear congruent method with the above selection of the magic numbers. Note that the plot is very homogeneous and random. There are no stripes, lattice structures, or any other visible patterns in the plot.

Implementation of this random-number generator on a computer is not always trivial, because of the different numerical range of the integers specified by the computer language or hardware. For example, most 32-bit computers have integers in $[-2^{31}, 2^{31} - 1]$. If a number runs out of this range by accident, the computer will reset it to zero. If the computer could modulate the integers by $2^{31} - 1$ automatically, we could implement a random-number generator with the above magic numbers a, b, and c simply by taking consecutive numbers from $x_{i+1} = 16\,807 x_i$ with any initial choice of $1 < x_0 < 2^{31} - 1$. The range of this generator would be $[0, 2^{31} - 1]$. However, this automatic modulation would cause some serious problems in other situations. For example, when a quantity is out of range due to a bug in the program, the computer would still wrap it back without producing an error message. This is why, in practice, computers do not modulate numbers automatically, so we have to devise a scheme to modulate the numbers generated in sequence. The following method is an implementation of the uniform random-number generator (normalized to the range of [0, 1]) discussed above.

```
// Method to generate a uniform random number in [0,1]
// following x(i+1)=a*x(i) mod c with a=pow(7,5) and
// c=pow(2,31)-1.  Here the seed is a global variable.

  public static double ranf() {
    final int a = 16807, c = 2147483647, q = 127773,
      r = 2836;
    final double cd = c;
    int h = seed/q;
    int l = seed%q;
    int t = a*l-r*h;
    if (t > 0) seed = t;
    else seed = c+t;
    return seed/cd;
  }
```

Note that in the above code, we have used two more magic numbers $q = c/a$ and $r = c \bmod a$. A program in Pascal that implements a method similar to that above is given by Park and Miller (1988). We can easily show that the above method modulates numbers with $c = 2^{31} - 1$ on any computer with integers of 32 bits or more. To use this method, the seed needs to be a global variable that is returned each time that the method is called. To show that the method given here does implement the algorithm correctly, we can set the initial seed to be 1, and then after 10 000 steps, we should have 1 043 618 065 returned as the value of the seed (Park and Miller, 1988).

The above generator has a period of $2^{31} - 1$. If a longer period is desired, we can create similar generators with higher-bit integers. For example, we can create a generator with a period of $2^{63} - 1$ for 64-bit integers with the following method.

```
// Method to generate a uniform random number in [0,1]
// following x(i+1)=a*x(i) mod c with a=pow(7,5) and
// c=pow(2,63)-1.  Here the seed is a global variable.

  public static double ranl() {
    final long a = 16807L, c = 9223372036854775807L,
      q = 548781581296767L, r = 12838L;
    final double cd = c;
    long h = seed/q;
    long l = seed%q;
    long t = a*l-r*h;
    if (t > 0) seed = t;
    else seed = c+t;
    return seed/cd;
  }
```

Note that we have used $a = 7^5$, $c = 2^{63} - 1$, $q = c/a$, and $r = c \bmod a$ in the above method with the seed being a 64-bit (`long`) integer.

In order to start the random-number generator differently every time, we need to have a systematic way of obtaining a different initial seed. Otherwise, we would not be able to obtain fair statistics. Almost every computer language has intrinsic

routines to report the current time in an integer form, and we can use this integer to construct an initial seed (Anderson, 1990). For example, most computers can produce $0 \le t_1 \le 59$ for the second of the minute, $0 \le t_2 \le 59$ for the minute of the hour, $0 \le t_3 \le 23$ for the hour of the day, $1 \le t_4 \le 31$ for the day of the month, $1 \le t_5 \le 12$ for the month of the year, and t_6 for the current year in common era. Then we can choose

$$i_s = t_6 + 70\Big(t_5 + 12\{t_4 + 31[t_3 + 23(t_2 + 59t_1)]\}\Big) \tag{2.75}$$

as the initial seed, which is roughly in the region of $[0, 2^{31} - 1]$. The results should never be the same as long as the jobs are started at least a second apart.

We now demonstrate how to apply the random-number generator and how to initiate the generator with the current time through a simple example. Consider evaluating π by randomly throwing a dart into a unit square defined by $x \in [0, 1]$ and $y \in [0, 1]$. By comparing the areas of the unit square and the quarter of the unit circle we can see that the chance of the dart landing inside a quarter of the unit circle centered at the origin of the coordinates is $\pi/4$. The following program is an implementation of such an evaluation of π in Java.

```
// An example of evaluating pi by throwing a dart into a
// unit square with 0<x<1 and 0<y<1.

import java.lang.*;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class Dart {
  final static int n = 1000000;
  static int seed;
  public static void main(String argv[]) {

 // Initiate the seed from the current time
    GregorianCalendar t = new GregorianCalendar();
    int t1 = t.get(Calendar.SECOND);
    int t2 = t.get(Calendar.MINUTE);
    int t3 = t.get(Calendar.HOUR_OF_DAY);
    int t4 = t.get(Calendar.DAY_OF_MONTH);
    int t5 = t.get(Calendar.MONTH)+1;
    int t6 = t.get(Calendar.YEAR);
    seed = t6+70*(t5+12*(t4+31*(t3+23*(t2+59*t1))));
    if ((seed%2) == 0) seed = seed-1;

 // Throw the dart into the unit square
    int ic = 0;
    for (int i=0; i<n; ++i) {
      double x = ranf();
      double y = ranf();
      if ((x*x+y*y) < 1) ic++;
    }
    System.out.println("Estimated pi: " + (4.0*ic/n));
  }

  public static double ranf() {...}
}
```

An even initial seed is usually avoided in order to have the full period of the generator realized. Note that in Java the months are represented by the numerals 0–11, so we have added 1 in the above program to have it between 1 and 12. To initiate the 64-bit generator, we can use the method `getTime()` from the `Date` class in Java, which returns the current time in milliseconds in a 64-bit (`long`) integer, measured from the beginning of January 1, 1970.

Uniform random-number generators are very important in scientific computing, and good ones are extremely difficult to find. The generator given here is considered to be one of the best uniform random-number generators.

New computer programming languages such as Java typically come with a comprehensive, intrinsic set of random-number generators that can be initiated automatically with the current time or with a chosen initial seed. We will demonstrate the use of such a generator in Java later in this section.

Other distributions

As soon as we obtain good uniform random-number generators, we can use them to create other types of random-number generators. For example, we can use a uniform random-number generator to create an exponential distribution or a Gaussian distribution.

All the exponential distributions can be cast into their simplest form

$$p(x) = e^{-x} \tag{2.76}$$

after a proper choice of units and coordinates. For example, if a system has energy levels of $E_0, E_1, \ldots, E_n$, the probability for the system to be at the energy level E_i at temperature T is given by

$$p(E_i, T) \propto e^{-(E_i - E_0)/k_B T}, \tag{2.77}$$

where k_B is the Boltzmann constant. If we choose $k_B T$ as the energy unit and E_0 as the zero point, the above equation reduces to Eq. (2.76).

One way to generate the exponential distribution is to relate it to a uniform distribution. For example, if we have a uniform distribution $f(y) = 1$ for $y \in [0, 1]$, we can relate it to an exponential distribution by

$$f(y)\,dy = dy = p(x)\,dx = e^{-x}\,dx, \tag{2.78}$$

which gives

$$y(x) - y(0) = 1 - e^{-x} \tag{2.79}$$

after integration. We can set $y(0) = 0$ and invert the above equation to have

$$x = -\ln(1 - y), \tag{2.80}$$

which relates the exponential distribution of $x \in [0, \infty]$ to the uniform distribution of $y \in [0, 1]$. The following method is an implementation of the exponential random-number generator given in the above equation and is constructed from a uniform random-number generator.

```
// Method to generate an exponential random number from a
// uniform random number in [0,1].

  public static double rane() {
    return -Math.log(1-ranf());
  }

  public static double ranf() {...}
```

The uniform random-number generator obtained earlier is used in this method. Note that when this method is used in a program, the seed still has to be a global variable.

As we have pointed out, another useful distribution in physics is the Gaussian distribution

$$g(x) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-x^2/2\sigma^2}, \tag{2.81}$$

where σ is the variance of the distribution, which we can take as 1 for the moment. The distribution with $\sigma \neq 1$ can be obtained via the rescaling of x by σ. We can use a uniform distribution $f(\phi) = 1$ for $\phi \in [0, 2\pi]$ and an exponential distribution $p(t) = e^{-t}$ for $t \in [0, \infty]$ to obtain two Gaussian distributions $g(x)$ and $g(y)$. We can relate the product of a uniform distribution and an exponential distribution to a product of two Gaussian distributions by

$$\frac{1}{2\pi} f(\phi)\, d\phi\; p(t)\, dt = g(x)\, dx\; g(y)\, dy, \tag{2.82}$$

which gives

$$e^{-t} dt\, d\phi = e^{-(x^2+y^2)/2} dx\, dy. \tag{2.83}$$

The above equation can be viewed as the coordinate transform from the polar system (ρ, ϕ) with $\rho = \sqrt{2t}$ into the rectangular system (x, y), that is,

$$x = \sqrt{2t}\cos\phi, \tag{2.84}$$

$$y = \sqrt{2t}\sin\phi, \tag{2.85}$$

which are two Gaussian distributions if t is taken from an exponential distribution and ϕ is taken from a uniform distribution in the region $[0, 2\pi]$. With the availability of the exponential random-number generator and uniform random-number generator, we can construct two Gaussian random numbers immediately from Eqs. (2.74) and (2.75). The exponential random-number generator itself can be obtained from a uniform random-number generator as discussed above. Below we show how to create two Gaussian random numbers from two uniform random numbers.

```
// Method to create two Gaussian random numbers from two
// uniform random numbers in [0,1].

  public static double[] rang() {
    double x[] = new double[2];
    double r1 = - Math.log(1-ranf());
    double r2 = 2*Math.PI*ranf();
    r1 = Math.sqrt(2*r1);
    x[0] = r1*Math.cos(r2);
    x[1] = r1*Math.sin(r2);
    return x;
  }

  public static double ranf() {...}
```

In principle, we can generate any given distribution numerically. For the Gaussian distribution and exponential distribution, we construct the new generators with the integral transformations in order to relate the distributions sought to the distributions known. A general procedure can be devised by dealing with the integral transformation numerically. For example, we can use the Metropolis algorithm, which will be discussed in Chapter 10, to obtain any distribution numerically.

Percolation in two dimensions

Let us use two-dimensional percolation as an example to illustrate how a random-number generator is utilized in computer simulations. When atoms are added to a solid surface, they first occupy the sites with the lowest potential energy to form small two-dimensional clusters. If the probability of occupying each empty site is still high, the clusters grow on the surface and eventually form a single layer, or a percolated two-dimensional network. If the probability of occupying each empty site is low, the clusters grow into island-like three-dimensional clusters. The general problem of the formation of a two-dimensional network can be cast into a simple model with each site carrying a fixed occupancy probability.

Assume that we have a two-dimensional square lattice with $n \times n$ lattice points. Then we can generate $n \times n$ random numbers $x_{ij} \in [0, 1]$ for $i = 0, 1, \ldots, n-1$ and $j = 0, 1, \ldots, n-1$. The random number x_{ij} is further compared with the assigned occupancy probability $p \in [0, 1]$. The site is occupied if $p > x_{ij}$; otherwise the site remains empty. Clusters are formed by the occupied sites. A site in each cluster is, at least, a nearest neighbor of another site in the same cluster. We can gradually change p from 0 to 1. As p increases, the sizes of the clusters increase and some clusters merge into larger clusters. When p reaches a critical probability p_c, there is one cluster of occupied sites, which reaches all the boundaries of the lattice. We call p_c the percolation threshold. The following method is the core of the simulation of two-dimensional percolation, which assigns a false value to an empty site and a true value to an occupied site.

```
// Method to create a 2-dimensional percolation lattice.

import java.util.Random;
  public static boolean[][] lattice(double p, int n) {
    Random r = new Random();
    boolean y[][] = new boolean[n][n];
    for (int i=0; i<n; ++i) {
      for (int j=0; j<n; ++j) {
        if (p > r.nextDouble()) y[i][j] = true;
        else y[i][j] = false;
      }
    }
    return y;
  }
```

Here y_{ij} is a boolean array that contains true values at all the occupied sites and false values at all the empty sites. We can use the above method with a program that has p increased from 0 to 1, and can sort out the sizes of all the clusters formed by the occupied lattice sites. In order to obtain good statistical averages, the procedure should be carried out many times. For more discussions on percolation, see Stauffer and Aharony (1992) and Grimmett (1999). Note that we have used the intrinsic random-number generator from Java: `nextDouble()` is a method in the `Random` class, and it creates a floating-point random number in [0, 1] when it is called. The default initiation of the generator, as used above, is from the current time.

The above example is an extremely simple application of the uniform random-number generator in physics. In Chapters 8, 10, and 11, we will discuss the application of random-number generators in other simulations. Interested readers can find more on different random-number generators in Knuth (1998), Park and Miller (1988), and Anderson (1990).

Exercises

2.1 Show that the error in the nth-order Lagrange interpolation scheme is bounded by

$$|\Delta f(x)| \leq \frac{\gamma_n}{4(n+1)} h^{n+1},$$

where $\gamma_n = \max[|f^{(n+1)}(x)|]$, for $x \in [x_0, x_n]$.

2.2 Write a program that implements the Lagrange interpolation scheme directly. Test it by evaluating $f(0.3)$ and $f(0.5)$ from the data taken from the error function with $f(0.0) = 0$, $f(0.4) = 0.428\,392$, $f(0.8) = 0.742\,101$, $f(1.2) = 0.910\,314$, and $f(1.6) = 0.970\,348$. Examine the accuracy of the interpolation by comparing the results obtained from the interpolation with the exact values $f(0.3) = 0.328\,627$ and $f(0.5) = 0.520\,500$.

2.3 The Newton interpolation is another popular interpolation scheme that adopts the polynomial

$$p_n(x) = \sum_{j=0}^{n} c_j \prod_{i=0}^{j-1}(x - x_i),$$

where $\prod_{i=0}^{-1}(x - x_i) = 1$. Show that this polynomial is equivalent to that of the Lagrange interpolation and the coefficients c_j here are recursively given by

$$c_j = \frac{f_j - \sum_{k=0}^{j-1} c_k \prod_{i=0}^{k-1}(x_j - x_i)}{\prod_{i=0}^{j-1}(x_j - x_i)}.$$

Write a subprogram that creates all c_j with given x_i and f_i.

2.4 Show that the coefficients in the Newton interpolation, defined in Exercise 2.3, can be cast into divided differences recursively as

$$a_{i\ldots j} = \frac{a_{i+1\ldots j} - a_{i\ldots j-1}}{x_j - x_i},$$

where $a_i = f_i$ are the discrete data and $a_{0\ldots i} = c_i$ are the coefficients in the Newton interpolation. Write a subprogram that creates c_i in this way. Apply this subprogram to create another subprogram that evaluates the interpolated value from the nested expression of the polynomial

$$p_n(x) = c_0 + (x - x_0)\{c_1 + (x - x_1)[c_2 + \cdots + (x - x_{n-1})c_n \cdots]\}.$$

Use the values of the Bessel function in Section 2.1 to test the program.

2.5 The Newton interpolation of Exercise 2.3 can be used inversely to find approximate roots of an equation $f(x) = 0$. Show that $x = p_n(0)$ is such an approximate root if the polynomial

$$p_n(z) = \sum_{j=0}^{n} c_j \prod_{i=0}^{j-1}(z - f_i),$$

where $\prod_{i=0}^{-1}(x - f_i) = 1$. Develop a program that estimates the root of $f(x) = e^x \ln x - x^2$ with $x_i = 1, 1.1, \ldots, 2.0$.

2.6 Modify the program in Section 2.3 for the least-squares approximation to fit the data set $f(0.0) = 1.000\,000$, $f(0.2) = 0.912\,005$, $f(0.4) = 0.671\,133$, $f(0.6) = 0.339\,986$, $f(0.8) = 0.002\,508$, $f(0.9) = -0.142\,449$, and $f(1.0) = -0.260\,052$ and the corresponding points of $f(-x) = f(x)$ as the input. The data are taken from the Bessel function table with $f(x) = J_0(3x)$. Compare the results with the well-known

approximate formula for the Bessel function,

$$f(x) = 1 - 2.249\,999\,7\,x^2 + 1.265\,620\,8\,x^4 - 0.316\,386\,6\,x^6 + 0.044\,447\,9\,x^8 - 0.003\,944\,4\,x^{10} + 0.000\,210\,0\,x^{12}.$$

2.7 Use the program in Section 2.3 that fits the Millikan data directly to a linear function to analyze the accuracy of the approximation. Assume that the error bars in the experimental measurements are $|\Delta q_k| = 0.05 q_k$. The accuracy of a curve fitting is determined from

$$\chi^2 = \frac{1}{n+1} \sum_{i=0}^{n} \frac{|f(x_i) - p_m(x_i)|^2}{\sigma_i^2},$$

where σ_i is the error bar of $f(x_i)$ and m is the order of the polynomial. If $\chi \ll 1$, the fitting is considered successful.

2.8 For periodic functions, we can approximate the function in one period with the periodic boundary condition imposed. Modify the cubic-spline program in Section 2.4 to have $p_{n-1}(x_n) = p_0(x_0)$, $p'_{n-1}(x_n) = p'_0(x_0)$, and $p''_{n-1}(x_n) = p''_0(x_0)$. Test the program with $f_i \in [0, 1]$ and $x_i \in [0, 1]$ generated randomly and sorted according to $x_{i+1} \geq x_i$ for $n = 20$.

2.9 Modify the cubic-spline program given in Section 2.4 to have the LU decomposition carried out by the Doolittle factorization.

2.10 Use a fifth-order polynomial

$$p_i(x) = \sum_{k=0}^{5} a_{ik} x^k$$

for $x \in [x_i, x_{i+1}]$ to create a quintic spline approximation for a function $f(x)$ given at discrete data points, $f_i = f(x_i)$ for $i = 0, 1, \ldots, n$, with $p_0^{(4)}(x_0) = p_{n-1}^{(4)}(x_n) = p_0^{(3)}(x_0) = p_{n-1}^{(3)}(x_n) = 0$. Find the linear equation set whose solution provides $p_i^{(4)}$ at every data point. Is the numerical problem here significantly different from that of the cubic spline? Modify the cubic-spline program given in Section 2.4 to one for the quintic spline and test your program by applying it to approximate $f_i \in [0, 1]$, generated randomly, at $x_i \in [0, 1]$, also generated randomly and sorted according to $x_{i+1} \geq x_i$ for $n = 40$.

2.11 We can approximate a function $f(x)$ as a linear combination of a set of basis functions $B_{nk}(x)$ through

$$p(x) = \sum_{j=-\infty}^{\infty} A_j B_{nj-n}(x),$$

for a chosen integer $n \geq 0$, where the functions $B_{nk}(x)$ are called B splines

of degree n, which can be constructed recursively with

$$B_{nk}(x) = \frac{x - x_k}{x_{k+n} - x_k} B_{n-1k}(x) + \frac{x_{l+n+1} - x}{x_{k+n+1} - x_{k+1}} B_{n-1k+1}(x),$$

starting from

$$B_{0k}(x) = \begin{cases} 1 & \text{for } x_k \leq x < x_{k+1}, \\ 0 & \text{elsewhere.} \end{cases}$$

If $x_{k+1} \geq x_k$ and $f_k = p(x_k)$, show that

$$B_{nk}(x) > 0 \quad \text{and} \quad \sum_{j=-\infty}^{\infty} B_{nj}(x) = 1.$$

Develop a computer program to implement B splines of degree 3. Compare the result for a set data against the cubic-spline approximation.

2.12 If we randomly drop a needle of unit length on an infinite plane that is covered by parallel lines one unit apart, the probability of the needle landing in a gap (not crossing any line) is $1 - 2/\pi$. Derive this probability analytically. Write a program that can sample the needle dropping process and calculate the probability without explicitly using the value of π. Compare you numerical result with the analytical result and discuss the possible sources of error.

2.13 Generate 21 pairs of random numbers (x_i, f_i) in $[0, 1]$ and sort them according to $x_{i+1} \geq x_i$. Treat them as a discrete set of function $f(x)$ and fit them to $p_{20}(x)$, where

$$p_m(x) = \sum_{k=0}^{m} \alpha_k u_k(x),$$

with $u_k(x)$ being orthogonal polynomials. Use the method developed in Section 2.2 to obtain the corresponding $u_k(x)$ and α_k. Numerically show that the orthogonal polynomials satisfy $\langle u_k | u_l \rangle = \delta_{kl}$. Compare the least-squares approximation with the cubic-spline approximation of the same data and discuss the cause of difference.

2.14 Develop a scheme that can generate any distribution $w(x) > 0$ in a given region $[a, b]$. Write a program to implement the scheme and test it with $w(x) = 1$, $w(x) = e^{-x^2}$, and $w(x) = x^2 e^{-x^2}$. Vary a and b, and compare the results with those from the uniform random-number generator and the Gaussian random-number generator given in Section 2.5.

2.15 Generate a large set of Gaussian random numbers and sort them into an increasing order. Then count the data points falling into each of the uniformly divided intervals. Use these values to perform a least-squares fit of the generated data to the function $f(x) = ae^{-x^2/2\sigma^2}$, where a and σ are the parameters to be determined. Comment on the quality of the generator based on the fitting result.

2.16 Write a program that can generate clusters of occupied sites in a two-dimensional square lattice with $n \times n$ sites. Determine $p_c(n)$, the threshold probability with at least one cluster reaching all the boundaries. Then determine p_c for an infinite lattice from

$$p_c(n) = p_c + \frac{c_1}{n} + \frac{c_2}{n^2} + \frac{c_3}{n^3} + \cdots ,$$

where c_i and p_c can be solved from the $p_c(n)$ obtained.