

Chapter 3

Numerical calculus

Calculus is at the heart of describing physical phenomena. As soon as we talk about motion, we must invoke differentiation and integration. For example, the velocity and the acceleration of a particle are the first-order and second-order time derivatives of the corresponding position vector, and the distance traveled by a particle is the integral of the corresponding speed over the elapsed time.

In this chapter, we introduce some basic computational methods for dealing with numerical differentiation and integration, and numerical schemes for searching for the roots of an equation and the extremes of a function. We stay at a basic level, using the simple but practical schemes frequently employed in computational physics and scientific computing. Some of the subjects will be reexamined later in other chapters to a greater depth. Some problems introduced here, such as searching for the global minimum or maximum of a multivariable function, are considered unsolved and are still under intensive research. Several interesting examples are given to illustrate how to apply these methods in studying important problems in physics and related fields.

3.1 Numerical differentiation

One basic tool that we will often use in this book is the Taylor expansion of a function $f(x)$ around a point x_0 :

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0) + \dots + \frac{(x - x_0)^n}{n!}f^{(n)}(x_0) + \dots \quad (3.1)$$

The above expansion can be generalized to describe a multivariable function $f(x, y, \dots)$ around the point (x_0, y_0, \dots) :

$$\begin{aligned} f(x, y, \dots) &= f(x_0, y_0, \dots) + (x - x_0)f_x(x_0, y_0, \dots) \\ &\quad + (y - y_0)f_y(x_0, y_0, \dots) + \frac{(x - x_0)^2}{2!}f_{xx}(x_0, y_0, \dots) \\ &\quad + \frac{(y - y_0)^2}{2!}f_{yy}(x_0, y_0, \dots) + \frac{2(x - x_0)(y - y_0)}{2!}f_{xy}(x_0, y_0, \dots) + \dots, \end{aligned} \quad (3.2)$$

where the subscript indices denote partial derivatives, for example, $f_{xy} = \partial^2 f / \partial x \partial y$.

The first-order derivative of a single-variable function $f(x)$ around a point x_i is defined from the limit

$$f'(x_i) = \lim_{\Delta x \rightarrow 0} \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} \quad (3.3)$$

if it exists. Now if we divide the space into discrete points x_i with evenly spaced intervals $x_{i+1} - x_i = h$ and label the function at the lattice points as $f_i = f(x_i)$, we obtain the simplest expression for the first-order derivative

$$f'_i = \frac{f_{i+1} - f_i}{h} + O(h). \quad (3.4)$$

We have used the notation $O(h)$ for a term on the order of h . Similar notation will be used throughout this book. The above formula is referred to as the *two-point formula* for the first-order derivative and can easily be derived by taking the Taylor expansion of f_{i+1} around x_i . The accuracy can be improved if we expand f_{i+1} and f_{i-1} around x_i and take the difference

$$f_{i+1} - f_{i-1} = 2h f'_i + O(h^3). \quad (3.5)$$

After a simple rearrangement, we have

$$f'_i = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2), \quad (3.6)$$

which is commonly known as the *three-point formula* for the first-order derivative. The accuracy of the expression increases to a higher order in h if more points are used. For example, a *five-point formula* can be derived by including the expansions of f_{i+2} and f_{i-2} around x_i . If we use the combinations

$$f_{i+1} - f_{i-1} = 2h f'_i + \frac{h^3}{3} f_i^{(3)} + O(h^5) \quad (3.7)$$

and

$$f_{i+2} - f_{i-2} = 4h f'_i + \frac{8h^3}{3} f_i^{(3)} + O(h^5) \quad (3.8)$$

to cancel the $f_i^{(3)}$ terms, we have

$$f'_i = \frac{1}{12h} (f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}) + O(h^4). \quad (3.9)$$

We can, of course, make the accuracy even higher by including more points, but in many cases this is not good practice. For real problems, the derivatives at points close to the boundaries are important and need to be calculated accurately. The errors in the derivatives of the boundary points will accumulate in other points when the scheme is used to integrate an equation. The more points involved in the expressions of the derivatives, the more difficulties we encounter in obtaining accurate derivatives at the boundaries. Another way to increase the accuracy is by decreasing the interval h . This is very practical on vector computers. The algorithms for first-order or second-order derivatives are usually fully vectorized, so a vector processor can calculate many points in just one computer clock cycle.

Approximate expressions for the second-order derivative can be obtained with different combinations of f_j . The *three-point formula* for the second-order derivative is given by the combination

$$f_{i+1} - 2f_i + f_{i-1} = h^2 f''_i + O(h^4), \quad (3.10)$$

with the Taylor expansions of $f_{i\pm 1}$ around x_i . Note that the third-order term with $f_i^{(3)}$ vanishes because of the cancellation in the combination. The above equation gives the second-order derivative as

$$f''_i = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2). \quad (3.11)$$

Similarly, we can combine the expansions of $f_{i\pm 2}$ and $f_{i\pm 1}$ around x_i and f_i to cancel the f'_i , $f_i^{(3)}$, $f_i^{(4)}$, and $f_i^{(5)}$ terms; then we have

$$f''_i = \frac{1}{12h^2}(-f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+2}) + O(h^4) \quad (3.12)$$

as the *five-point formula* for the second-order derivative. The difficulty in dealing with the points around the boundaries still remains. We can use the interpolation formulas that we developed in the Chapter 2 to extrapolate the derivatives to the boundary points. The following program shows an example of calculating the first-order and second-order derivatives with the three-point formulas.

```
// An example of evaluating the derivatives with the
// 3-point formulas for f(x)=sin(x).

import java.lang.*;
public class Deriv {
    static final int n = 100, m = 5;
    public static void main(String argv[]) {
        double[] x = new double[n+1];
        double[] f = new double[n+1];
        double[] f1 = new double[n+1];
        double[] f2 = new double[n+1];

        // Assign constants, data points, and function
        int k = 2;
        double h = Math.PI/(2*n);
        for (int i=0; i<=n; ++i) {
            x[i] = h*i;
            f[i] = Math.sin(x[i]);
        }

        // Calculate 1st-order and 2nd-order derivatives
        f1 = firstOrderDerivative(h, f, k);
        f2 = secondOrderDerivative(h, f, k);

        // Output the result in every m data points
        for (int i=0; i<=n; i+=m) {
            double df1 = f1[i]-Math.cos(x[i]);
            double df2 = f2[i]+Math.sin(x[i]);
            System.out.println("x = " + x[i]);
            System.out.println("f'(x) = " + f1[i]);
            System.out.println("Error in f'(x): " + df1);
            System.out.println("f''(x) = " + f2[i]);
            System.out.println("Error in f''(x): " + df2);
        }
    }
}
```

```

        System.out.println();
    }
}

// Method for the 1st-order derivative with the 3-point
// formula. Extrapolations are made at the boundaries.

public static double[] firstOrderDerivative(double h,
    double f[], int k) {
    int n = f.length-1;
    double[] y = new double[n+1];
    double[] xl = new double[k+1];
    double[] fl = new double[k+1];
    double[] fr = new double[k+1];

    // Evaluate the derivative at nonboundary points
    for (int i=1; i<n; ++i)
        y[i] = (f[i+1]-f[i-1])/(2*h);

    // Lagrange-extrapolate the boundary points
    for (int i=1; i<=(k+1); ++i) {
        xl[i-1] = h*i;
        fl[i-1] = y[i];
        fr[i-1] = y[n-i];
    }
    y[0] = aitken(0, xl, fl);
    y[n] = aitken(0, xl, fr);
    return y;
}

// Method for the 2nd-order derivative with the 3-point
// formula. Extrapolations are made at the boundaries.

public static double[] secondOrderDerivative(double h,
    double[] f, int k) {
    int n = f.length-1;
    double[] y = new double[n+1];
    double[] xl = new double[k+1];
    double[] fl = new double[k+1];
    double[] fr = new double[k+1];

    // Evaluate the derivative at nonboundary points
    for (int i=1; i<n; ++i)
        y[i] = (f[i+1]-2*f[i]+f[i-1])/(h*h);

    // Lagrange-extrapolate the boundary points
    for (int i=1; i<=(k+1); ++i) {
        xl[i-1] = h*i;
        fl[i-1] = y[i];
        fr[i-1] = y[n-i];
    }
    y[0] = aitken(0, xl, fl);
    y[n] = aitken(0, xl, fr);
    return y;
}

public static double aitken(double x, double xi[],
    double fi[]) {...}
}

```

Table 3.1. *Derivatives obtained in the example*

x	f'	$\Delta f'$	f''	$\Delta f''$
0	0.999 959	-0.000 041	0.000 004	0.000 004
$\pi/10$	0.951 017	-0.000 039	-0.309 087	-0.000 070
$\pi/5$	0.808 985	-0.000 032	-0.587 736	0.000 049
$3\pi/10$	0.587 762	-0.000 023	-0.809 013	0.000 004
$2\pi/5$	0.309 003	-0.000 014	-0.951 055	0.000 001
$\pi/2$	-0.000 004	-0.000 004	-0.999 980	0.000 020

We have taken a simple function $f(x) = \sin x$, given at 101 discrete points with evenly spaced intervals in the region $[0, \pi/2]$. The Lagrange interpolation is applied to extrapolate the derivatives at the boundary points. The numerical results are summarized in Table 3.1, together with their errors. Note that the extrapolated data are of the same order of accuracy as other calculated values for f' and f'' at both $x = 0$ and $x = \pi/2$ because the three-point Lagrange interpolation scheme is accurate to a quadratic behavior. The functions $\sin x$ and $\cos x$ are well approximated by a linear or a quadratic curve at those two points.

In practice, we may encounter two problems with the formulas used above. The first problem is that we may not have the data given at uniform data points. One solution to such a problem is to perform an interpolation of the data first and then apply the above formulas to the function at the uniform data points generated from the interpolation. This approach can be tedious and has errors from two sources, the interpolation and the formulas above. The easiest solution to the problem is to adopt formulas that are suitable for nonuniform data points. If we use the Taylor expansion

$$f(x_{i\pm 1}) = f(x_i) + (x_{i\pm 1} - x_i)f'(x_i) + \frac{1}{2!}(x_{i\pm 1} - x_i)^2 f''(x_i) + O(h^3) \quad (3.13)$$

and a combination of f_{i-1} , f_i , and f_{i+1} to cancel the second-order terms, we obtain

$$f'_i = \frac{h_{i-1}^2 f_{i+1} + (h_i^2 - h_{i-1}^2) f_i - h_i^2 f_{i-1}}{h_i h_{i-1}(h_i + h_{i-1})} + O(h^2), \quad (3.14)$$

where $h_i = x_{i+1} - x_i$ and h is the larger of $|h_{i-1}|$ and $|h_i|$. This is the three-point formula for the first-order derivative in the case of nonuniform data points. Note that the accuracy here is the same as for the uniform data points. This is a better choice than interpolating the data first because the formula can be implemented in almost the same manner as in the case of the uniform data points. The following method returns the first-order derivative for such a situation.

```

// Method to calculate the 1st-order derivative with the
// nonuniform 3-point formula. Extrapolations are made
// at the boundaries.

public static double[] firstOrderDerivative2
    (double x[], double f[], int k) {
    int n = x.length-1;
    double[] y = new double[n+1];
    double[] xl = new double[k+1];
    double[] fl = new double[k+1];
    double[] xr = new double[k+1];
    double[] fr = new double[k+1];

    // Calculate the derivative at the field points
    double h0 = x[1]-x[0];
    double a0 = h0*h0;
    for (int i=1; i<n; ++i) {
        double h = x[i+1]-x[i];
        double a = h*h;
        double b = a-a0;
        double c = h*h0*(h+h0);
        y[i] = (a0*f[i+1]+b*f[i]-a*f[i-1])/c;
        h0 = h;
        a0 = a;
    }

    // Lagrange-extrapolate the boundary points
    for (int i=1; i<=(k+1); ++i) {
        xl[i-1] = x[i]-x[0];
        fl[i-1] = y[i];
        xr[i-1] = x[n]-x[n-i];
        fr[i-1] = y[n-i];
    }
    y[0] = aitken(0, xl, fl);
    y[n] = aitken(0, xr, fr);
    return y;
}

public static double aitken(double x, double xi[],
    double fi[]) {...}

```

The corresponding three-point formula for the second-order derivative can be derived with a combination of f_{i-1} , f_i , and f_{i+1} to have the first-order terms in the Taylor expansion removed, and we have

$$f''_i = \frac{2[h_{i-1}f_{i+1} - (h_i + h_{i-1})f_i + h_if_{i-1}]}{h_i h_{i+1}(h_i + h_{i-1})} + O(h). \quad (3.15)$$

Note that the accuracy here is an order lower than in the case of uniform data points because the third-order terms in the Taylor expansion are not canceled completely at the same time. However, the reality is that the third-order terms are partially canceled depending on how close the data points are to being uniform. We need to be careful in applying the three-point formula above. If higher accuracy is needed, we can resort to a corresponding five-point formula instead.

The second problem is that the accuracy cannot be controlled during evaluation with the three-point or five-point formulas. If the function $f(x)$ is available

continuously, that is, for any given x in the region of interest, the accuracy in the numerical evaluation of the derivatives can be systematically improved to any desired level through an adaptive scheme. The basic strategy here is to apply the combination of

$$\Delta_1(h) = \frac{f(x+h) - f(x-h)}{2h} \quad (3.16)$$

and $\Delta_1(h/2)$ to remove the next nonzero term in the Taylor expansion and to repeat this process over and over. For example,

$$\Delta_1(h) - 4\Delta_1(h/2) = -3f'(x) + O(h^4). \quad (3.17)$$

Then we can use the difference in the evaluations of $f'(x)$ from $\Delta_1(h)$ and $\Delta_1(h/2)$ to set up a rough criterion for the desired accuracy. The following program is such an implementation.

```
// An example of evaluating 1st-order derivative through
// the adaptive scheme.

import java.lang.*;
public class Deriv3 {
    public static void main(String argv[]) {
        double del = 1e-6;
        int n = 10, m = 10;
        double h = Math.PI/(2*n);

        // Evaluate the derivative and output the result
        int k = 0;
        for (int i=0; i<=n; ++i) {
            double x = h*i;
            double d = (f(x+h)-f(x-h))/(2*h);
            double f1 = firstOrderDerivative3(x, h, d, del, k, m);
            double df1 = f1-Math.cos(x);
            System.out.println("x = " + x);
            System.out.println("f'(x) = " + f1);
            System.out.println("Error in f'(x): " + df1);
        }
    }

    // Method to carry out 1st-order derivative through the
    // adaptive scheme.

    public static double firstOrderDerivative3(double x,
        double h, double d, double del, int step,
        int maxstep) {
        step++;
        h = h/2;
        double d1 = (f(x+h)-f(x-h))/(2*h);
        if (step >= maxstep) {
            System.out.println ("Not converged after "
                + step + " recursions");
            return d1;
        }
        else {
            if ((h*h*Math.abs(d-d1)) < del) return (4*d1-d)/3;
        }
    }
}
```

```

        else return firstOrderDerivative3(x, h, d1, del,
                                         step, maxstep);
    }
}

public static double f(double x) {
    return Math.sin(x);
}
}
}

```

Running the above program we obtain the first-order derivative of the function accurate to the order of the tolerance set in the program.

In fact, we can analytically work out a recursion by combining more $\Delta_1(h/2^k)$ for $k = 0, 1, \dots$. For example, using the combination

$$\Delta_1(h) - 20\Delta_1(h/2) + 64\Delta_1(h/4) = 45f'(x) + O(h^6), \quad (3.18)$$

we cancel the fourth-order terms in the Taylor expansion. Note that the fifth-order term is also canceled automatically because of the symmetry in $\Delta_1(h/2^k)$. This process can be repeated analytically; this scheme is called the Richardson extrapolation. The derivation and implementation of the Richardson extrapolation are left as an exercise for the reader.

A similar adaptive scheme can be devised for the second-order derivative. For example, if we define the function

$$\Delta_2(h) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}, \quad (3.19)$$

and then use $\Delta_2(h)$ and $\Delta_2(h/2)$ to remove the next nonzero terms in the Taylor expansion, we have

$$\Delta_2(h) - 4\Delta_2(h/2) = -3f''(x) + O(h^4). \quad (3.20)$$

Based on the above equation and further addition of the terms involving $\Delta_2(h/2^k)$, we can come up with exactly the same adaptive scheme and the Richardson extrapolation for the second-order derivative.

3.2 Numerical integration

Let us turn to numerical integration. In general, we want to obtain the numerical value of an integral, defined in the region $[a, b]$,

$$S = \int_a^b f(x) dx. \quad (3.21)$$

We can divide the region $[a, b]$ into n slices, evenly spaced with an interval h . If we label the data points as x_i with $i = 0, 1, \dots, n$, we can write the entire integral as a summation of integrals, with each over an individual slice,

$$\int_a^b f(x) dx = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x) dx. \quad (3.22)$$

If we can develop a numerical scheme that evaluates the summation over several slices accurately, we will have solved the problem. Let us first consider each slice separately. The simplest quadrature is obtained if we approximate $f(x)$ in the region $[x_i, x_{i+1}]$ linearly, that is, $f(x) \simeq f_i + (x - x_i)(f_{i+1} - f_i)/h$. After integrating over every slice with this linear function, we have

$$S = \frac{h}{2} \sum_{i=0}^{n-1} (f_i + f_{i+1}) + O(h^2), \quad (3.23)$$

where $O(h^2)$ comes from the error in the linear interpolation of the function. The above quadrature is commonly referred to as the *trapezoid rule*, which has an overall accuracy up to $O(h^2)$.

We can obtain a quadrature with a higher accuracy by working on two slices together. If we apply the Lagrange interpolation to the function $f(x)$ in the region $[x_{i-1}, x_{i+1}]$, we have

$$\begin{aligned} f(x) &= \frac{(x - x_i)(x - x_{i+1})}{(x_{i-1} - x_i)(x_{i-1} - x_{i+1})} f_{i-1} + \frac{(x - x_{i-1})(x - x_{i+1})}{(x_i - x_{i-1})(x_i - x_{i+1})} f_i \\ &\quad + \frac{(x - x_{i-1})(x - x_i)}{(x_{i+1} - x_{i-1})(x_{i+1} - x_i)} f_{i+1} + O(h^3). \end{aligned} \quad (3.24)$$

If we carry out the integration for every pair of slices together with the integrand given from the above equation, we have

$$S = \frac{h}{3} \sum_{j=0}^{n/2-1} (f_{2j} + 4f_{2j+1} + f_{2j+2}) + O(h^4), \quad (3.25)$$

which is known as the *Simpson rule*. The third-order term vanishes because of cancelation. In order to pair up all the slices, we have to have an even number of slices. What happens if we have an odd number of slices, or an even number of points in $[a, b]$? One solution is to isolate the last slice and we then have

$$\int_{b-h}^b f(x) dx = \frac{h}{12} (-f_{n-2} + 8f_{n-1} + 5f_n). \quad (3.26)$$

The expression for $f(x)$ in Eq. (3.24), constructed from the last three points of the function, has been used in order to obtain the above result. The following program is an implementation of the Simpson rule for calculating an integral.

```
// An example of evaluating an integral with the Simpson
// rule over f(x)=sin(x).

import java.lang.*;
public class Integral {
    static final int n = 8;
    public static void main(String argv[]) {
        double f[] = new double[n+1];
        double h = Math.PI/(2.0*n);
        for (int i=0; i<=n; ++i) {
            double x = h*i;
            f[i] = Math.sin(x);
        }
        double sum = f[0] + f[n];
        for (int i=1; i<n; i+=2) {
            sum += 4*f[i];
        }
        for (int i=2; i<n; i+=2) {
            sum += 2*f[i];
        }
        System.out.println("Integral = " + sum*h);
    }
}
```

```

        f[i] = Math.sin(x);
    }
    double s = simpson(f, h);
    System.out.println("The integral is: " + s);
}

// Method to achieve the evenly spaced Simpson rule.

public static double simpson(double y[], double h) {
    int n = y.length-1;
    double s0 = 0, s1 = 0, s2 = 0;
    for (int i=1; i<n; i+=2) {
        s0 += y[i];
        s1 += y[i-1];
        s2 += y[i+1];
    }
    double s = (s1+4*s0+s2)/3;

    // Add the last slice separately for an even n+1
    if ((n+1)%2 == 0)
        return h*(s+(5*y[n]+8*y[n-1]-y[n-2])/12);
    else
        return h*s;
}
}

```

We have used $f(x) = \sin x$ as the integrand in the above example program and $[0, \pi/2]$ as the integration region. The output of the above program is 1.000 008, which has six digits of accuracy compared with the exact result 1. Note that we have used only nine mesh points to reach such a high accuracy.

In some cases, we may not have the integrand given at uniform data points. The Simpson rule can easily be generalized to accommodate cases with nonuniform data points. We can rewrite the interpolation in Eq. (3.24) as

$$f(x) = ax^2 + bx + c, \quad (3.27)$$

where

$$a = \frac{h_{i-1}f_{i+1} - (h_{i-1} + h_i)f_i + h_if_{i-1}}{h_{i-1}h_i(h_{i-1} + h_i)}, \quad (3.28)$$

$$b = \frac{h_{i-1}^2f_{i+1} + (h_i^2 - h_{i-1}^2)f_i - h_i^2f_{i-1}}{h_{i-1}h_i(h_{i-1} + h_i)}, \quad (3.29)$$

$$c = f_i, \quad (3.30)$$

with $h_i = x_{i+1} - x_i$. We have taken $x_i = 0$ because the integral

$$S_i = \int_{x_{i-1}}^{x_{i+1}} f(x) dx \quad (3.31)$$

is independent of the choice of the origin of the coordinates. Then we have

$$S_i = \int_{-h_{i-1}}^{h_i} f(x) dx = \alpha f_{i+1} + \beta f_i + \gamma f_{i-1}, \quad (3.32)$$

where

$$\alpha = \frac{2h_i^2 + h_i h_{i-1} - h_{i-1}^2}{6h_i}, \quad (3.33)$$

$$\beta = \frac{(h_i + h_{i-1})^3}{6h_i h_{i-1}}, \quad (3.34)$$

$$\gamma = \frac{-h_i^2 + h_i h_{i-1} + 2h_{i-1}^2}{6h_i}. \quad (3.35)$$

The last slice needs to be treated separately if $n + 1$ is even, as with the case of uniform data points. Then we have

$$S_n = \int_0^{h_{n-1}} f(x) dx = \alpha f_n + \beta f_{n-1} + \gamma f_{n-2}, \quad (3.36)$$

where

$$\alpha = \frac{h_{n-1}}{6} \left(3 - \frac{h_{n-1}}{h_{n-1} + h_{n-2}} \right), \quad (3.37)$$

$$\beta = \frac{h_{n-1}}{6} \left(3 + \frac{h_{n-1}}{h_{n-2}} \right), \quad (3.38)$$

$$\gamma = -\frac{h_{n-1}}{6} \frac{h_{n-1}^2}{h_{n-2}(h_{n-1} + h_{n-2})}. \quad (3.39)$$

The equations appear quite tedious but implementing them in a program is quite straightforward following the program for the case of uniform data points.

Even though we can make an order-of-magnitude estimate of the error occurring in either the trapezoid rule or the Simpson rule, it is not possible to control it because of the uncertainty involved in the associated coefficient. We can, however, develop an adaptive scheme based on either the trapezoid rule or the Simpson rule to make the error in the evaluation of an integral controllable. Here we demonstrate such a scheme with the Simpson rule and leave the derivation of a corresponding scheme with the trapezoid rule to Exercise 3.9.

If we expand the integrand $f(x)$ in a Taylor series around $x = a$, we have

$$\begin{aligned} S &= \int_a^b f(x) dx \\ &= \int_a^b \left[\sum_{k=0}^{\infty} \frac{(x-a)^k}{k!} f^{(k)}(a) \right] dx \\ &= \sum_{k=0}^{\infty} \frac{h^{k+1}}{(k+1)!} f^{(k)}(a), \end{aligned} \quad (3.40)$$

where $h = b - a$. If we apply the Simpson rule with $x_{i-1} = a$, $x_i = c = (a + b)/2$, and $x_{i+1} = b$, we have the zeroth-level approximation

$$\begin{aligned} S_0 &= \frac{h}{6} [f(a) + 4f(c) + f(b)] \\ &= \frac{h}{6} \left[f(a) + \sum_{k=0}^{\infty} \frac{h^k}{k!} \left(\frac{1}{2^{k-2}} - 1 \right) f^{(k)}(a) \right]. \end{aligned} \quad (3.41)$$

We have expanded both $f(c)$ and $f(b)$ in a Taylor series around $x = a$ in the above equation. Now if we take the difference between S and S_0 and keep only the leading term, we have

$$\Delta S_0 = S - S_0 \approx -\frac{h^5}{2880} f^{(4)}(a). \quad (3.42)$$

We can continue to apply the Simpson rule in the regions $[a, c]$ and $[c, b]$. Then we obtain the first-level approximation

$$S_1 = \frac{h}{12} [f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)], \quad (3.43)$$

where $d = (a + c)/2$ and $e = (c + b)/2$, and

$$\begin{aligned} \Delta S_1 &= S - S_1 \\ &\approx -\frac{(h/2)^5}{2880} f^{(4)}(a) - \frac{(h/2)^5}{2880} f^{(4)}(c) \\ &\approx -\frac{1}{2^4} \frac{h^5}{2880} f^{(4)}(a). \end{aligned} \quad (3.44)$$

We have used that $f^{(4)}(a) \approx f^{(4)}(c)$. The difference between the first-level and zeroth-level approximations is

$$S_1 - S_0 \approx -\frac{15}{16} \frac{h^5}{2880} f^{(4)}(a) \approx \frac{15}{16} \Delta S_0 \approx 15 \Delta S_1. \quad (3.45)$$

The above result can be used to set up the criterion for the error control in an adaptive algorithm. Consider, for example, that we want the error $|\Delta S| \leq \delta$. First we can carry out S_0 and S_1 . Then we check whether $|S_1 - S_0| \leq 15\delta$, that is, whether $|\Delta S_1| \leq \delta$. If it is true, we return S_1 as the approximation for the integral. Otherwise, we continue the adaptive process to divide the region into two halves, four quarters, and so on, until we reach the desired accuracy with $|S - S_n| \leq \delta$, where S_n is the n th-level approximation of the integral. Let us here take the evaluation of the integral

$$S = \int_0^\pi \frac{[1 + a_0(1 - \cos x)^2] dx}{(1 + a_0 \sin^2 x) \sqrt{1 + 2a_0(1 - \cos x)}}, \quad (3.46)$$

for any $a_0 \geq 0$, as an example. The following program is an implementation of the adaptive Simpson rule for the integral above.

```
// An example of evaluating an integral with the adaptive
// Simpson rule.

import java.lang.*;
public class Integral2 {
    static final int n = 100;
    public static void main(String argv[]) {
        double del = 1e-6;
        int k = 0;
        double a = 0;
        double b = Math.PI;
```

```

        double s = simpson2(a, b, del, k, n);
        System.out.println ("S = " + s);
    }

// Method to integrate over f(x) with the adaptive
// Simpson rule.

public static double simpson2(double a, double b,
    double del, int step, int maxstep) {
    double h = b-a;
    double c = (b+a)/2;
    double fa = f(a);
    double fc = f(c);
    double fb = f(b);
    double s0 = h*(fa+4*fc+fb)/6;
    double s1 = h*(fa+4*f(a+h/4)+2*fc
        + 4*f(a+3*h/4)+fb)/12;
    step++;
    if (step >= maxstep) {
        System.out.println ("Not converged after "
            + step + " recursions");
        return s1;
    }
    else {
        if (Math.abs(s1-s0) < 15*del) return s1;
        else return simpson2(a, c, del/2, step, maxstep)
            + simpson2(c, b, del/2, step, maxstep);
    }
}

// Method to provide the integrand f(x).

public static double f(double x) {
    double a0 = 5;
    double s = Math.sin(x);
    double c = Math.cos(x);
    double f = (1+a0*(1-c)*(1-c))
        /((1+a0*s*s)*Math.sqrt(1+2*a0*(1-c)));
    return f;
}
}

```

After running the program, we obtain $S \simeq 3.141\,592\,78$. Note that $a_0 = 5$ is assigned in the above program. In fact, we can assign a random value to a_0 and the result will remain the same within the errorbar. This is because the above integral $S \equiv \pi$, independent of the specific value of a_0 . The integral showed up while solving a problem in Jackson (1999). It is quite intriguing because the integrand has such a complicated dependence on a_0 and x and yet the result is so simple and cannot be obtained from any table or symbolic system. Try it and see whether an analytical solution can be found easily. One way to do it is to show first that the integral is independent of a_0 and then set $a_0 = 0$.

The adaptive scheme presented above provides a way of evaluating an integral accurately. It is, however, limited to cases with an integrand that is given continuously in the integration region. For problems that involve an integrand

given at discrete points, the strength of the adaptive scheme is reduced because the data must be interpolated and the interpolation may destroy the accuracy in the adaptive scheme. Adaptive schemes can also be slow if a significant number of levels are needed. In determining whether to use an adaptive scheme or not, we should consider how critical and practical the accuracy sought in the evaluation is against the speed of computation and possible errors due to other factors involved. Sometimes the adaptive method introduced above is the only viable numerical approach if a definite answer is sought, especially when an analytic result does not exist or is difficult to find.

3.3 Roots of an equation

In physics, we often encounter situations in which we need to find the possible value of x that ensures the equation $f(x) = 0$, where $f(x)$ can either be an explicit or an implicit function of x . If such a value exists, we call it a *root* or *zero* of the equation. In this section, we will discuss only single-variable problems and leave the discussion of multivariable cases to Chapter 5, after we have gained some basic knowledge of matrix operations.

Bisection method

If we know that there is a root $x = x_r$ in the region $[a, b]$ for $f(x) = 0$, we can use the *bisection method* to find it within a required accuracy. The bisection method is the most intuitive method, and the idea is very simple. Because there is a root in the region, $f(a)f(b) < 0$. We can divide the region into two equal parts with $x_0 = (a + b)/2$. Then we have either $f(a)f(x_0) < 0$ or $f(x_0)f(b) < 0$. If $f(a)f(x_0) < 0$, the next trial value is $x_1 = (a + x_0)/2$; otherwise, $x_1 = (x_0 + b)/2$. This procedure is repeated until the improvement on x_k or $|f(x_k)|$ is less than the given tolerance δ .

Let us take $f(x) = e^x \ln x - x^2$ as an example to illustrate how the bisection method works. We know that when $x = 1$, $f(x) = -1$ and when $x = 2$, $f(x) = e^2 \ln 2 - 4 \approx 1$. So there is at least one value $x_r \in [1, 2]$ that would make $f(x_r) = 0$. In the neighborhood of x_r , we have $f(x_r + \delta) > 0$ and $f(x_r - \delta) < 0$.

```
// An example of searching for a root via the bisection
// method for f(x)=exp(x)*ln(x)-x*x=0.

import java.lang.*;
public class Bisect {
    public static void main(String argv[]) {
        double x = 0, del = 1e-6, a = 1, b = 2;
        double dx = b-a;
        int k = 0;
        while (Math.abs(dx) > del) {
            x = (a+b)/2;
```

```

if ((f(a)*f(x)) < 0) {
    b = x;
    dx = b-a;
}
else {
    a = x;
    dx = b-a;
}
k++;
}
System.out.println("Iteration number: " + k);
System.out.println("Root obtained: " + x);
System.out.println("Estimated error: " + dx);
}

// Method to provide function f(x)=exp(x)*log(x)-x*x.

public static double f(double x) {
    return Math.exp(x)*Math.log(x)-x*x;
}
}

```

The above program gives the root $x_r = 1.694\,601 \pm 0.000\,001$ after only 20 iterations. The error comes from the final improvement in the search. This error can be reduced to a lower value by making the tolerance δ smaller.

The Newton method

This method is based on linear approximation of a smooth function around its root. We can formally expand the function $f(x_r) = 0$ in the neighborhood of the root x_r through the Taylor expansion

$$f(x_r) \simeq f(x) + (x_r - x)f'(x) + \dots = 0, \quad (3.47)$$

where x can be viewed as a trial value for the root of x_r at the k th step and the approximate value of the next step x_{k+1} can be derived from

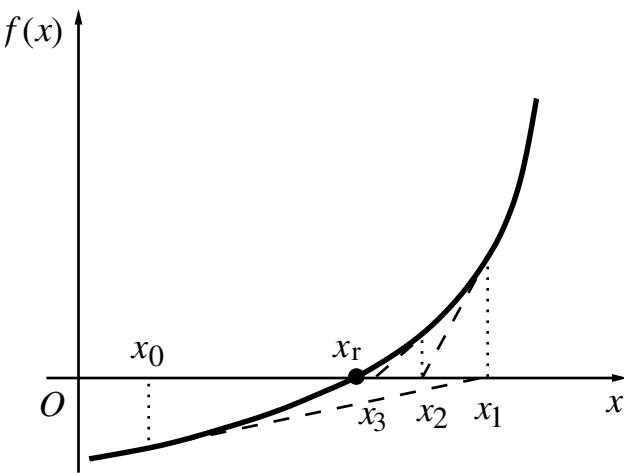
$$f(x_{k+1}) = f(x_k) + (x_{k+1} - x_k)f'(x_k) \simeq 0, \quad (3.48)$$

that is,

$$x_{k+1} = x_k + \Delta x_k = x_k - f_k/f'_k, \quad (3.49)$$

with $k = 0, 1, \dots$. Here we have used the notation $f_k = f(x_k)$. The above iterative scheme is known as the *Newton method*. It is also referred to as the *Newton–Raphson method* in the literature. The above equation is equivalent to approximating the root by drawing a tangent to the curve at the point x_k and taking x_{k+1} as the tangent's intercept on the x axis. This step is repeated toward the root, as illustrated in Fig. 3.1. To see how this method works in a program, we again take the function $f(x) = e^x \ln x - x^2$ as an example. The following program is an implementation of the Newton method.

Fig. 3.1 A schematic illustration of the steps in the Newton method for searching for the root of $f(x) = 0$.



```
// An example of searching for a root via the Newton method
// for f(x)=exp(x)*ln(x)-x*x=0.

import java.lang.*;
public class Newton {
    public static void main(String argv[]) {
        double del = 1e-6, a = 1, b = 2;
        double dx = b-a, x=(a+b)/2;
        int k = 0;
        while (Math.abs(dx) > del) {
            dx = f(x)/d(x);
            x -= dx;
            k++;
        }
        System.out.println("Iteration number: " + k);
        System.out.println("Root obtained: " + x);
        System.out.println("Estimated error: " + dx);
    }

    public static double f(double x) {...}

    // Method to provide the derivative f'(x).

    public static double d(double x) {
        return Math.exp(x)*(Math.log(x)+1/x)-2*x;
    }
}
```

The above program gives the root $x_r = 1.694\,600\,92$ after only five iterations. It is clear that the Newton method is more efficient, because the error is now much smaller even though we have started the search at exactly the same point, and have gone through a much smaller number of steps. The reason is very simple. Each new step size $|\Delta x_k| = |x_{k+1} - x_k|$ in the Newton method is determined according to Eq. (3.49) by the ratio of the value and the slope of the function $f(x)$ at x_k . For the same function value, a large step is created for the small-slope case, whereas a small step is made for the large-slope case. This ensures an efficient update and

also avoids possible overshooting. There is no such mechanism in the bisection method.

Secant method

In many cases, especially when $f(x)$ has an implicit dependence on x , an analytic expression for the first-order derivative needed in the Newton method may not exist or may be very difficult to obtain. We have to find an alternative scheme to achieve a similar algorithm. One way to do this is to replace f'_k in Eq. (3.49) with the two-point formula for the first-order derivative, which gives

$$x_{k+1} = x_k - (x_k - x_{k-1})f_k / (f_k - f_{k-1}). \quad (3.50)$$

This iterative scheme is commonly known as the *secant method*, or the *discrete Newton method*. The disadvantage of the method is that we need two points in order to start the search process. The advantage of the method is that $f(x)$ can now be implicitly given without the need for the first-order derivative. We can still use the function $f(x) = e^x \ln x - x^2$ as an example, in order to make a comparison.

```
// An example of searching for a root via the secant method
// for f(x)=exp(x)*ln(x)-x*x=0.

import java.lang.*;
public class Root {
    public static void main(String argv[]) {
        double del = 1e-6, a = 1, b = 2;
        double dx = (b-a)/10, x = (a+b)/2;
        int n = 6;
        x = secant(n, del, x, dx);
        System.out.println("Root obtained: " + x);
    }

    // Method to carry out the secant search.

    public static double secant(int n, double del,
        double x, double dx) {
        int k = 0;
        double x1 = x+dx;
        while ((Math.abs(dx)>del) && (k<n)) {
            double d = f(x1)-f(x);
            double x2 = x1-f(x1)*(x1-x)/d;
            x = x1;
            x1 = x2;
            dx = x1-x;
            k++;
        }
        if (k==n) System.out.println("Convergence not" +
            " found after " + n + " iterations");
        return x1;
    }

    public static double f(double x) {...}
}
```

The above program gives the root $x_r = 1.694\,601\,0$ after five iterations. As expected, the secant method is more efficient than the bisection method but less efficient than the Newton method, because of the two-point approximation of the first-order derivative. However, if the expression for the first-order derivative cannot easily be obtained, the secant method becomes very useful and powerful.

3.4 Extremes of a function

An associated problem to finding the root of an equation is finding the maxima and/or minima of a function. Examples of such situations in physics occur when considering the equilibrium position of an object, the potential surface of a field, and the optimized structures of molecules and small clusters. Here we consider mainly a function of a single variable, $g = g(x)$, and just touch on the multi-variable case of $g = g(x_1, x_2, \dots, x_l)$ with the steepest-descent method. Other schemes for the multivariable cases are left to later chapters.

Knowing the solution of a nonlinear equation $f(x) = 0$, we can develop numerical schemes to obtain minima or maxima of a function $g(x)$. We know that an extreme of $g(x)$ occurs at the point with

$$f(x) = \frac{dg(x)}{dx} = 0, \quad (3.51)$$

which is a minimum (maximum) if $f'(x) = g''(x)$ is greater (less) than zero. So all the root-search schemes discussed so far can be generalized here to search for the extremes of a single-variable function.

However, at each step of updating the value of x , we need to make a judgment as to whether $g(x_{k+1})$ is increasing (decreasing) if we are searching for a maximum (minimum) of the function. If it is, we accept the update. If it is not, we reverse the update; that is, instead of using $x_{k+1} = x_k + \Delta x_k$, we use $x_{k+1} = x_k - \Delta x_k$. With the Newton method, the increment is $\Delta x_k = -f_k/f'_k$, and with the secant method, the increment is $\Delta x_k = -(x_k - x_{k-1})f_k/(f_k - f_{k-1})$.

Let us illustrate these ideas with a simple example of finding the bond length of the diatomic molecule NaCl from the interaction potential between the two ions (Na^+ and Cl^- in this case). Assuming that the interaction potential is $V(r)$ when the two ions are separated by a distance r , the bond length r_{eq} is the equilibrium distance when $V(r)$ is at its minimum. We can model the interaction potential between Na^+ and Cl^- as

$$V(r) = -\frac{e^2}{4\pi\epsilon_0 r} + V_0 e^{-r/r_0}, \quad (3.52)$$

where e is the charge of a proton, ϵ_0 is the electric permittivity of vacuum, and V_0 and r_0 are parameters of this effective interaction. The first term in Eq. (3.52) comes from the Coulomb interaction between the two ions, but the second term

is the result of the electron distribution in the system. We will use $V_0 = 1.09 \times 10^3$ eV, which is taken from the experimental value for solid NaCl (Kittel, 1995), and $r_0 = 0.330 \text{ \AA}$, which is a little larger than the corresponding parameter for solid NaCl ($r_0 = 0.321 \text{ \AA}$), because there is less charge screening in an isolated molecule. At equilibrium, the force between the two ions,

$$f(r) = -\frac{dV(r)}{dr} = -\frac{e^2}{4\pi\epsilon_0 r^2} + \frac{V_0}{r_0} e^{-r/r_0}, \quad (3.53)$$

is zero. Therefore, we search for the root of $f(x) = dg(x)/dx = 0$, with $g(x) = -V(x)$. We will force the algorithm to move toward the minimum of $V(r)$. The following program is an implementation of the algorithm with the secant method to find the bond length of NaCl.

```
// An example of calculating the bond length of NaCl.

import java.lang.*;
public class Bond {
    static final double e2 = 14.4, v0 = 1090, r0 = 0.33;
    public static void main(String argv[]) {
        double del = 1e-6, r = 2, dr = 0.1;
        int n = 20;
        r = secant2(n, del, r, dr);
        System.out.println("The bond length is " + r +
            " angstroms");
    }

    // Method to carry out the secant search for the
    // maximum of g(x) via the root of f(x)=dg(x)/dx=0.

    public static double secant2(int n, double del,
        double x, double dx) {
        int k = 0;
        double x1 = x+dx, x2 = 0;
        double g0 = g(x);
        double g1 = g(x1);
        if (g1 > g0) x1 = x-dx;
        while ((Math.abs(dx)>del) && (k<n)) {
            double d = f(x1)-f(x);
            dx = -(x1-x)*f(x1)/d;
            x2 = x1+dx;
            double g2 = g(x2);
            if (g2 > g1) x2 = x1-dx;
            x = x1;
            x1 = x2;
            g1 = g2;
            k++;
        }
        if (k==n) System.out.println("Convergence not" +
            " found after " + n + " iterations");
        return x1;
    }
}
```

```

// Method to provide function g(x)=-e2/x+v0*exp(-x/r0).

public static double g(double x) {
    return -e2/x+v0*Math.exp(-x/r0);
}

// Method to provide function f(x)=-dg(x)/dx.

public static double f(double x) {
    return -e2/(x*x)+v0*Math.exp(-x/r0)/r0;
}
}
}

```

The bond length obtained from the above program is $r_{\text{eq}} = 2.36 \text{ \AA}$. We have used $e^2/4\pi\epsilon_0 = 14.4 \text{ \AA eV}$ for convenience. The method for searching for the minimum is modified slightly from the secant method used in the preceding section in order to force the search to move toward the minimum of $g(x)$. We will still obtain the same result as with the secant method used in the earlier example for this simple problem, because there is only one minimum of $V(x)$ around the point where we start the search. The other minimum of $V(x)$ is at $x = 0$ which is also a singularity. For a more general $g(x)$, modifications introduced in the above program are necessary.

Another relevant issue is that in many cases we do not have the explicit function $f(x) = g'(x)$ if $g(x)$ is not an explicit function of x . However, we can construct the first-order and second-order derivatives numerically from the two-point or three-point formulas, for example.

In the example program above, the search process is forced to move along the direction of descending the function $g(x)$ when looking for a minimum. In other words, for $x_{k+1} = x_k + \Delta x_k$, the increment Δx_k has the sign opposite to $g'(x_k)$. Based on this observation, an update scheme can be formulated as

$$x_{k+1} = x_k + \Delta x_k = x_k - ag'(x_k), \quad (3.54)$$

with a being a positive, small, and adjustable parameter. This scheme can be generalized to the multivariable case as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k = \mathbf{x}_k - a \nabla g(\mathbf{x}_k) / |\nabla g(\mathbf{x}_k)|, \quad (3.55)$$

where $\mathbf{x} = (x_1, x_2, \dots, x_l)$ and $\nabla g(\mathbf{x}) = (\partial g / \partial x_1, \partial g / \partial x_2, \dots, \partial g / \partial x_l)$.

Note that step $\Delta \mathbf{x}_k$ here is scaled by $|\nabla g(\mathbf{x}_k)|$ and is forced to move toward the direction of the steepest descent. This is why this method is known as the *steepest-descent method*. The following program is an implementation of such a scheme to search for the minimum of the function $g(x, y) = (x - 1)^2 e^{-y^2} + y(y + 2) e^{-2x^2}$ around $x = 0.1$ and $y = -1$.

```

// An example of searching for a minimum of a multivariable
// function through the steepest-descent method.

import java.lang.*;
public class Minimum {

```

```

public static void main(String argv[]) {
    double del = 1e-6, a = 0.1;
    double x[] = new double[2];
    x[0] = 0.1;
    x[1] = -1;
    steepestDescent(x, a, del);
    System.out.println("The minimum is at"
        + " x= " + x[0] + ", y= " +x[1]);
}

// Method to carry out the steepest-descent search.

public static void steepestDescent(double x[],
    double a, double del) {
    int n = x.length;
    double h = 1e-6;
    double g0 = g(x);
    double fi[] = new double[n];
    fi = f(x, h);
    double dg = 0;
    for (int i=0; i<n; ++i) dg += fi[i]*fi[i];
    dg = Math.sqrt(dg);
    double b = a/dg;
    while (dg > del) {
        for (int i=0; i<n; ++i) x[i] -= b*fi[i];
        h /= 2;
        fi = f(x, h);
        dg = 0;
        for (int i=0; i<n; ++i) dg += fi[i]*fi[i];
        dg = Math.sqrt(dg);
        b = a/dg;
        double g1 = g(x);
        if (g1 > g0) a /= 2;
        else g0 = g1;
    }
}

// Method to provide the gradient of g(x).

public static double[] f(double x[], double h) {
    int n = x.length;
    double z[] = new double[n];
    double y[] = (double[]) x.clone();
    double g0 = g(x);
    for (int i=0; i<n; ++i) {
        y[i] += h;
        z[i] = (g(y)-g0)/h;
    }
    return z;
}

// Method to provide function g(x).

public static double g(double x[]) {
    return (x[0]-1)*(x[0]-1)*Math.exp(-x[1]*x[1])
        +x[1]*(x[1]+2)*Math.exp(-2*x[0]*x[0]);
}
}

```

Note that the spacing in the two-point formula for the derivative is reduced by a factor of 2 after each search, so is the step size in case of overshooting. After running the program, we find the minimum is at $x \simeq 0.107\,355$ and $y \simeq -1.223\,376$. This is a very simple but not very efficient scheme. Like many other optimization schemes, it converges to a local minimum near the starting point. The search for a global minimum or maximum of a multivariable function is a nontrivial task, especially when the function contains a significant number of local minima or maxima. Active research is still looking for better and more reliable schemes. In the last few decades, several advanced methods have been introduced for dealing with function optimization, most noticeably, the simulated annealing scheme, the Monte Carlo method, and the genetic algorithm and programming. We will discuss some of these more advanced topics later in the book.

3.5 Classical scattering

Scattering is a very important process in physics. From systems at the microscopic scale, such as protons and neutrons in nuclei, to those at the astronomical scale, such as galaxies and stars, scattering processes play a crucial role in determining their structures and dynamics. In general, a many-body process can be viewed as a sum of many simultaneous two-body scattering events if coherent scattering does not happen.

In this section, we will apply the computational methods that we have developed in this and the preceding chapter to study the classical scattering of two particles, interacting with each other through a pairwise potential. Most scattering processes with realistic interaction potentials cannot be solved analytically. Therefore, numerical solutions of a scattering problem become extremely valuable if we want to understand the physical process of particle–particle interaction. We will assume that the interaction potential between the two particles is spherically symmetric. Thus the total angular momentum and energy of the system are conserved during the scattering.

The two-particle system

The Lagrangian for a general two-body system can be written as

$$\mathcal{L} = \frac{m_1}{2} v_1^2 + \frac{m_2}{2} v_2^2 - V(\mathbf{r}_1, \mathbf{r}_2), \quad (3.56)$$

where m_i , \mathbf{r}_i , and $v_i = |d\mathbf{r}_i/dt|$ with $i = 1, 2$ are, respectively, the mass, position vector, and speed of the i th particle, and V is the interaction potential between the two particles, which we take to be spherically symmetric, that is, $V(\mathbf{r}_1, \mathbf{r}_2) = V(r_{21})$, with $r_{21} = |\mathbf{r}_2 - \mathbf{r}_1|$ being the distance between the two particles.

We can always perform a coordinate transformation from \mathbf{r}_1 and \mathbf{r}_2 to the relative coordinate \mathbf{r} and the center-of-mass coordinate \mathbf{r}_c with

$$\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1, \quad (3.57)$$

$$\mathbf{r}_c = \frac{m_1 \mathbf{r}_1 + m_2 \mathbf{r}_2}{m_1 + m_2}. \quad (3.58)$$

Then we can express the Lagrangian of the system in terms of the new coordinates and their corresponding speeds as

$$\mathcal{L} = \frac{M}{2} v_c^2 + \frac{m}{2} v^2 - V(r), \quad (3.59)$$

where $r = r_{21}$ and $v = |\mathbf{dr}/dt|$ are the distance and relative speed between the two particles, $M = m_1 + m_2$ is the total mass of the system, $m = m_1 m_2 / (m_1 + m_2)$ is the reduced mass of the two particles, and $v_c = |\mathbf{dr}_c/dt|$ is the speed of the center of mass. If we study the scattering in the center-of-mass coordinate system with $\mathbf{v}_c = d\mathbf{r}_c/dt = \mathbf{0}$, the process is then represented by the motion of a single particle of mass m in a central potential $V(r)$. In general, a two-particle system with a spherically symmetric interaction can be viewed as a single particle with a reduced mass moving in a central potential that is identical to the interaction potential.

We can reach the same conclusion from Newton's equations

$$m_1 \ddot{\mathbf{r}}_1 = \mathbf{f}_1, \quad (3.60)$$

$$m_2 \ddot{\mathbf{r}}_2 = \mathbf{f}_2, \quad (3.61)$$

where the accelerations and forces are given by $\ddot{\mathbf{r}}_i = d^2 \mathbf{r}_i / dt^2$ and $\mathbf{f}_i = -\nabla_i V(r_{21}) = -dV(r_{21})/d\mathbf{r}_i$. Note that, following the convention in physics, we have used two dots over a variable to denote the second-order time derivative of the variable, and we will also use a dot over a variable to denote the first-order time derivative of the variable. Adding the above two equations and using Newton's third law, $\mathbf{f}_1 = -\mathbf{f}_2$, or dividing the corresponding equation by m_i and then taking the difference, we obtain

$$m \ddot{\mathbf{r}} = \mathbf{f}(\mathbf{r}), \quad (3.62)$$

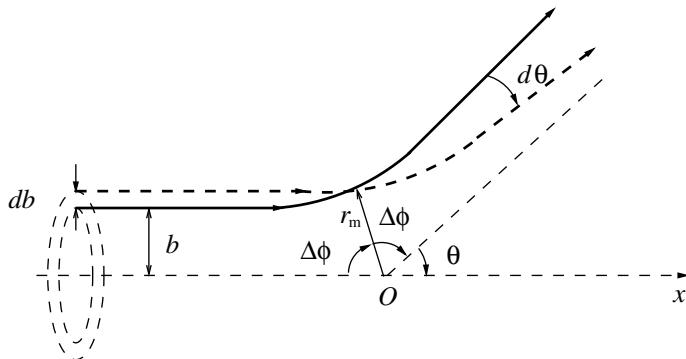
$$M \ddot{\mathbf{r}}_c = \mathbf{0}, \quad (3.63)$$

where $\mathbf{f}(\mathbf{r}) = -\nabla V(r) = -dV(r)/d\mathbf{r}$. So the motion of a two-particle system with an isotropic interaction is equivalent to the constant-velocity motion of the center of mass plus the relative motion of two particles that is described by an effective particle of mass m in a central potential $V(r)$.

Cross section of scattering

Now we only need to study the scattering process of a particle with a mass m in a central potential $V(r)$. Assume that the particle is coming in from the left with an impact parameter b , the shortest distance between the particle and the potential center if $V(r) \rightarrow 0$. A sketch of the process is given in Fig. 3.2.

Fig. 3.2 A sketch of the scattering process of a particle in a central potential.



The total cross section of such a scattering process is given by

$$\sigma = \int \sigma(\theta) d\Omega, \quad (3.64)$$

where $\sigma(\theta)$ is the differential cross section, or the probability of a particle's being found in the solid angle element $d\Omega = 2\pi \sin \theta d\theta$ at the deflection angle θ .

If the particles are coming in with a flux density I (number of particles per unit cross-sectional area per unit time), then the number of particles per unit time within the range db of the impact parameter b is $2\pi I b db$. Because all the incoming particles in this area will go out in the solid angle element $d\Omega$ with the probability $\sigma(\theta)$, we have

$$2\pi I b db = I \sigma(\theta) d\Omega, \quad (3.65)$$

which gives the differential cross section as

$$\sigma(\theta) = \frac{b}{\sin \theta} \left| \frac{db}{d\theta} \right|. \quad (3.66)$$

The reason for taking the absolute value of $db/d\theta$ in the above equation is that $db/d\theta$ can be positive or negative depending on the form of the potential and the impact parameter. However, $\sigma(\theta)$ has to be positive because it is a probability. We can relate this center-of-mass cross section to the cross section measured in the laboratory through an inverse coordinate transformation of Eq. (3.57) and Eq. (3.58), which relates \mathbf{r} and \mathbf{r}_c back to \mathbf{r}_1 and \mathbf{r}_2 . We will not discuss this transformation here; interested readers can find it in any standard advanced mechanics textbook.

Numerical evaluation of the cross section

Because the interaction between two particles is described by a spherically symmetric potential, the angular momentum and the total energy of the system are

conserved during the scattering. Formally, we have

$$l = mbv_0 = mr^2\dot{\phi} \quad (3.67)$$

and

$$E = \frac{m}{2}v_0^2 = \frac{m}{2}(\dot{r}^2 + r^2\dot{\phi}^2) + V(r) \quad (3.68)$$

which are respectively the total momentum and total energy and which are constant. Here r is the radial coordinate, ϕ is the polar angle, and v_0 is the initial impact velocity. Combining Eq. (3.67) and Eq. (3.68) with

$$\frac{d\phi}{dr} = \frac{d\phi}{dt} \frac{dt}{dr}, \quad (3.69)$$

we obtain

$$\frac{d\phi}{dr} = \pm \frac{b}{r^2\sqrt{1 - b^2/r^2 - V(r)/E}}, \quad (3.70)$$

which provides a relation between ϕ and r for the given E , b , and $V(r)$. Here the $+$ and $-$ signs correspond to two different but symmetric parts of the trajectory. The equation above can be used to calculate the deflection angle θ through

$$\theta = \pi - 2\Delta\phi, \quad (3.71)$$

where $\Delta\phi$ is the change in the polar angle when r changes from infinity to its minimum value r_m . From Eq. (3.70), we have

$$\begin{aligned} \Delta\phi &= b \int_{r_m}^{\infty} \frac{dr}{r^2\sqrt{1 - b^2/r^2 - V(r)/E}} \\ &= -b \int_{\infty}^{r_m} \frac{dr}{r^2\sqrt{1 - b^2/r^2 - V(r)/E}}. \end{aligned} \quad (3.72)$$

If we use the energy conservation and angular momentum conservation discussed earlier in Eq. (3.67) and Eq. (3.68), we can show that r_m is given by

$$1 - \frac{b^2}{r_m^2} - \frac{V(r_m)}{E} = 0, \quad (3.73)$$

which is the result of zero r -component velocity, that is, $\dot{r} = 0$. Because of the change in the polar angle $\Delta\phi = \pi/2$ for $V(r) = 0$, we can rewrite Eq. (3.71) as

$$\theta = 2b \left[\int_b^{\infty} \frac{dr}{r^2\sqrt{1 - b^2/r^2}} - \int_{r_m}^{\infty} \frac{dr}{r^2\sqrt{1 - b^2/r^2 - V(r)/E}} \right]. \quad (3.74)$$

The real reason for rewriting the constant π as an integral in the above expression for θ is a numerical strategy to reduce possible errors coming from the truncation of the integration region at both ends of the second term. The integrand in the first integral diverges as $r \rightarrow b$ in much the same way as the integrand in the second integral does as $r \rightarrow r_m$. The errors from the first and second terms cancel each other, at least partially, because they are of opposite signs.

Now we demonstrate how to calculate the differential cross section for a given potential. Let us take the Yukawa potential

$$V(r) = \frac{\kappa}{r} e^{-r/a} \quad (3.75)$$

as an illustrative example. Here κ and a are positive parameters that reflect, respectively, the range and the strength of the potential and can be adjusted. We use the secant method to solve Eq. (3.73) to obtain r_m for the given b and E . Then we use the Simpson rule to calculate the integrals in Eq. (3.74). After that, we apply the three-point formula for the first-order derivative to obtain $d\theta/db$. Finally, we put all these together to obtain the differential cross section of Eq. (3.66).

For simplicity, we choose $E = m = \kappa = 1$. The following program is an implementation of the scheme outlined above.

```
// An example of calculating the differential cross section
// of classical scattering on the Yukawa potential.

import java.lang.*;
public class Collide {
    static final int n = 10000, m = 20;
    static final double a = 100, e = 1;
    static double b;
    public static void main(String argv[]) {
        int nc = 20, ne = 2;
        double del = 1e-6, db = 0.5, b0 = 0.01, h = 0.01;
        double g1, g2;
        double theta[] = new double[n+1];
        double fi[] = new double[n+1];
        double sig[] = new double[m+1];
        for (int i=0; i<=m; ++i) {
            b = b0+i*db;

            // Calculate the first term of theta
            for (int j=0; j<=n; ++j) {
                double r = b+h*(j+1);
                fi[j] = 1/(r*r*Math.sqrt(fb(r)));
            }
            g1 = simpson(fi, h);

            // Find r_m from 1-b*b/(r*r)-V/E = 0
            double rm = secant(nc, del, b, h);

            // Calculate the second term of theta
            for (int j=0; j<=n; ++j) {
                double r = rm+h*(j+1);
                fi[j] = 1/(r*r*Math.sqrt(f(r)));
            }
            g2 = simpson(fi, h);
            theta[i] = 2*b*(g1-g2);
        }

        // Calculate d theta/d b
        sig = firstOrderDerivative(db, theta, ne);

        // Put the cross section in log form with the exact
    }
}
```

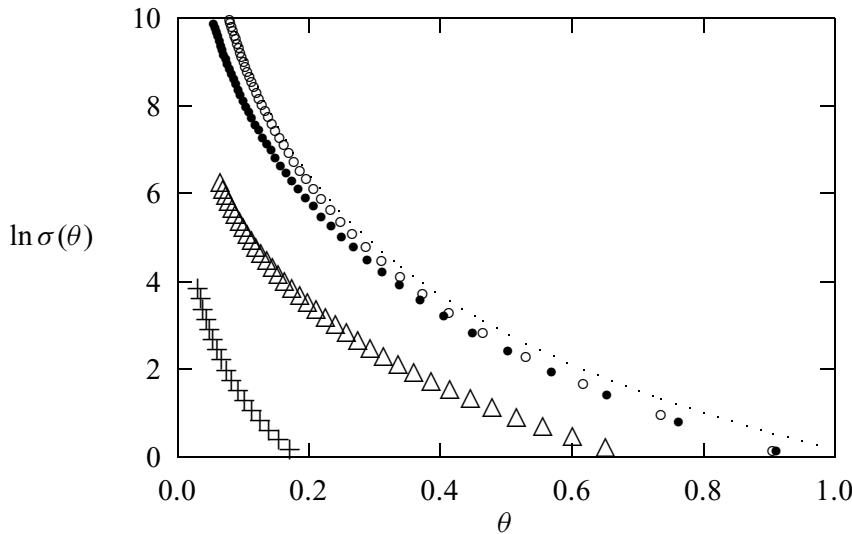


Fig. 3.3 This figure shows the differential cross section for scattering from the Yukawa potential with $a = 0.1$ (crosses), $a = 1$ (triangles), $a = 10$ (solid circles), and $a = 100$ (open circles), together with the analytical result for Coulomb scattering (dots). Other parameters used are $E = m = \kappa = 1$.

```

// result of the Coulomb scattering (ruth)
for (int i=m; i>=0; --i) {
    b = b0+i*db;
    sig[i] = b/(Math.abs(sig[i])*Math.sin(theta[i]));
    double ruth = 1/Math.pow(Math.sin(theta[i]/2),4);
    ruth /= 16;
    double si = Math.log(sig[i]);
    double ru = Math.log(ruth);
    System.out.println("theta = " + theta[i]);
    System.out.println("ln sigma(theta) = " + si);
    System.out.println("ln sigma_r(theta) = " + ru);
    System.out.println();
}
}

public static double simpson(double y[], double h)
{...}

public static double secant(int n, double del,
    double x, double dx) {...}

public static double[] firstOrderDerivative(double h,
    double f[], int m) {...}

public static double aitken(double x, double xi[],
    double fi[]) {...}

// Method to provide function f(x) for the root search.

public static double f(double x) {
    return 1-b*b/(x*x)-Math.exp(-x/a)/(x*e);
}

// Method to provide function 1-b*b/(x*x).

public static double fb(double x) {
    return 1-b*b/(x*x);
}
}

```

The methods called in the program are exactly those given earlier in this the preceding chapter. We have also included the analytical result for Coulomb scattering, which is a special case of the Yukawa potential with $a \rightarrow \infty$. The differential cross section for Coulomb scattering is

$$\sigma(\theta) = \left(\frac{\kappa}{4E}\right)^2 \frac{1}{\sin^4(\theta/2)}; \quad (3.76)$$

this expression is commonly referred to as the Rutherford formula. The results obtained with the above program for different values of a are shown in Fig. 3.3. It is clear that when a is increased, the differential cross section becomes closer to that of the Coulomb scattering, as expected.

Exercises

- 3.1 Write a program that obtains the first-order and second-order derivatives from the five-point formulas. Check its accuracy with the function $f(x) = \cos x \sinh x$ with 101 uniformly spaced points for $0 \leq x \leq \pi/2$. Discuss the procedure used for dealing with the boundary points.
- 3.2 The derivatives of a function can be obtained by first performing the interpolation of $f(x_i)$ and then obtaining the derivatives of the interpolation polynomial. Show that: if $p_1(x)$ is used to interpolate the function and $x_0 = x - h$ and $x_1 = x + h$ are used as the data points, the three-point formula for the first-order derivative is recovered; if $p_2(x)$ is used to interpolate the function and $x_0 = x - h$, $x_1 = x$, and $x_2 = x + h$ are used as the data points, the three-point formula for the second-order derivative is recovered; if $p_3(x)$ is used to interpolate the function and $x_0 = x - 2h$, $x_1 = x - h$, $x_2 = x + h$, and $x_3 = x + 2h$ are used as the data points, the five-point formula for the first-order derivative is recovered; and if $p_4(x)$ is used to interpolate the function and $x_0 = x - 2h$, $x_1 = x - h$, $x_2 = x$, $x_3 = x + h$, and $x_4 = x + 2h$ are used as the data points, the five-point formula for the second-order derivative is recovered.
- 3.3 The Richardson extrapolation is a recursive scheme that can be used to improve the accuracy of the evaluation of derivatives. From the Taylor expansions of $f(x - h)$ and $f(x + h)$, we know that

$$\Delta_1(h) = \frac{f(x + h) - f(x - h)}{2h} = f'(x) + \sum_{k=1}^{\infty} c_{2k} h^{2k}.$$

Starting from $\Gamma_{k0} = \Delta_1(h/2^k)$, show that the Richardson extrapolation

$$\Gamma_{kl} = \frac{4^l}{4^l - 1} \Gamma_{kl-1} - \frac{1}{4^l - 1} \Gamma_{k-l-1},$$

for $0 \leq l \leq k$, leads to

$$f'(x) = \Gamma_{kl} - \sum_{m=l+1}^{\infty} B_{ml} \left(\frac{h}{2^k}\right)^{2m},$$

where

$$B_{kl} = \left(\frac{4^l - 4^k}{4^l - 1} \right) B_{kl-1},$$

with $B_{kk} = 0$. Write a subprogram that generates the Richardson extrapolation and evaluates the first-order derivative $f'(x) \simeq \Gamma_{nn}$ for a given n . Test the subprogram with $f(x) = \sin x$.

- 3.4 Repeat Exercise 3.3 with $\Delta_1(h)$ replaced by

$$\Delta_2(h) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

and $f'(x)$ replaced by $f''(x)$. Is there any significant difference?

- 3.5 Using the fact that

$$\Delta_2(h) - 4\Delta_2(h/2) = -3f''(x) + O(h^4),$$

construct a subprogram that calculates the second-order derivative $f''(x)$ adaptively. (Δ_2 is defined in Exercise 3.4.) Then apply the subprogram to $f(x) = e^{-x} \ln x$. Is there any significant difference between the scheme here and the adaptive scheme for the first-order derivative introduced in Section 3.1?

- 3.6 Derive the Simpson rule with a pair of slices with an equal interval by using the Taylor expansion of $f(x)$ around x_i in the region $[x_{i-1}, x_{i+1}]$ and the three-point formulas for the first-order and second-order derivatives. Show that the contribution of the last slice is also correctly given by the formula in Section 3.2 under such an approach.
- 3.7 Derive the Simpson rule with $f(x) = ax^2 + bx + c$ going through points x_{i-1} , x_i , and x_{i+1} . Determine a , b , and c from the three equations given at the three data points first and then carry out the integration over the two slices with the quadratic curve being the integrand.
- 3.8 Develop a program that can calculate the integral with a given integrand $f(x)$ in the region $[a, b]$ by the Simpson rule with nonuniform data points. Check its accuracy with the integral $\int_0^\infty e^{-x} dx$ with $x_j = jhe^{j\alpha}$, where h and α are small constants.
- 3.9 Expand the integrand of $S = \int_a^b f(x) dx$ in a Taylor series around $x = a$ and show that

$$\Delta S_0 = S - S_0 \approx -\frac{h^3}{12} f''(a),$$

where $h = b - a$ and S_0 is the trapezoid evaluation of S with $x_i = a$ and $x_{i+1} = b$. If the region $[a, b]$ is divided into two, $[a, c]$ and $[c, b]$, with $c = (a + b)/2$, show that

$$\Delta S_1 = S - S_1 \approx -\frac{h^3}{48} f''(a),$$

where S_1 is the sum of the trapezoid evaluations of the integral in the regions $[a, c]$ and $[c, b]$. Using the fact that $S_1 - S_0 \approx 3\Delta S_1$, write a subprogram that performs the adaptive trapezoid evaluation of an integral to a specified accuracy.

- 3.10 The Romberg algorithm is a recursive procedure for evaluating an integral based on the adaptive trapezoid rule with

$$S_{kl} = \frac{4^l}{4^l - 1} S_{kl-1} - \frac{1}{4^l - 1} S_{k-l-1},$$

where S_{k0} is the evaluation of the integral with the adaptive trapezoid rule to the k th level. Show that

$$\lim_{k \rightarrow \infty} S_{kl} = S = \int_a^b f(x) dx$$

for any l . Find the formula for the error estimate $\Delta S_{kl} = S - S_{kl}$.

- 3.11 Apply the secant method developed in Section 3.3 to solve $f(x) = e^{x^2} \ln x^2 - x = 0$. Discuss the procedure for dealing with more than one root in a given region.
- 3.12 Develop a subprogram that implements the Newton method to solve $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, where both \mathbf{f} and \mathbf{x} are l -dimensional vectors. Test the subprogram with $f_1(x_1, x_2) = e^{x_1^2} \ln x_2 - x_1^2$ and $f_2(x_1, x_2) = e^{x_2} \ln x_1 - x_2^2$.
- 3.13 Write a routine that returns the minimum of a single-variable function $g(x)$ in a given region $[a, b]$. Assume that the first-order and second-order derivatives of $g(x)$ are not explicitly given. Test the routine with some well-known functions, for example, $g(x) = x^2$.
- 3.14 Consider clusters of ions $(\text{Na}^+)_n(\text{Cl}^-)_m$, where m and n are small, positive integers. Use the steepest-descent method to obtain the stable geometric structures of the clusters. Use the molecular potential given in Eq. (3.52) for opposite charges but the Coulomb potential for like charges.
- 3.15 Modify the program in Section 3.5 to evaluate the differential cross section of the Lennard–Jones potential

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right].$$

Choose ε as the energy unit and σ as the length unit.

- 3.16 Show that the period of a pendulum confined in a vertical plane is

$$T = 4\sqrt{\frac{\ell}{2g}} \int_0^{\theta_0} \frac{d\theta}{\sqrt{\cos \theta - \cos \theta_0}},$$

where $\theta_0 < \pi$ is the maximum of the angle between the pendulum and the downward vertical, ℓ is the length of the pendulum, and g is the gravitational acceleration. Evaluate this integral numerically for $\theta_0 = \pi/128, \pi/64, \pi/32, \pi/16, \pi/8, \pi/4, \pi/2$, and compare the numerical results with the small-angle approximation $T \simeq 2\pi\sqrt{\ell/g}$.

- 3.17 Show that the time taken for a meterstick to fall on a frictionless, horizontal surface from an initial angle θ_0 to a final angle θ with the horizontal is

$$t = \frac{1}{2} \sqrt{\frac{\ell}{3g}} \int_{\theta}^{\theta_0} \sqrt{\frac{1 + 3 \cos^2 \phi}{\sin \theta_0 - \sin \phi}} d\phi,$$

where ℓ is the length of the meterstick (1 meter) and g is the gravitational acceleration. Evaluate this time numerically with $\theta_0 = \pi/8, \pi/4, \pi/2$, and $\theta = 0$.

- 3.18 For a classical particle of mass m moving in a one-dimensional, symmetric potential well $U(x) = U(-x)$, show that the inverse function is given by

$$x(U) = \frac{1}{2\pi\sqrt{2m}} \int_0^U \frac{T(E)dE}{\sqrt{U-E}},$$

where $T(E)$ is the period of the motion for a given total energy E . Find $U(x)$ numerically if $T/T_0 = 1 + \alpha e^{-E/E_0}$, for $\alpha = 0, 0.1, 1, 10$. Use T_0 , E_0 , and $T_0\sqrt{E_0}/(2\pi\sqrt{2m})$ as the units of the period, energy, and position, respectively. Discuss the accuracy of the numerical results by comparing them with an available analytical result.