Chapter 4
Ordinary differential equations

Most problems in physics and engineering appear in the form of differential equations. For example, the motion of a classical particle is described by Newton's equation, which is a second-order ordinary differential equation involving at least a second-order derivative in time, and the motion of a quantum particle is described by the Schrödinger equation, which is a partial differential equation involving a first-order partial derivative in time and second-order partial derivatives in coordinates. The dynamics and statics of bulk materials such as fluids and solids are all described by differential equations.

In this chapter, we introduce some basic numerical methods for solving ordinary differential equations. We will discuss the corresponding schemes for partial differential equations in Chapter 7 and some more advanced techniques for the many-particle Newton equation and the many-body Schrödinger equation in Chapters 8 and 10. Hydrodynamics and magnetohydrodynamics are treated in Chapter 9.

In general, we can classify ordinary differential equations into three major categories:

(1) initial-value problems, which involve time-dependent equations with given initial conditions;
(2) boundary-value problems, which involve differential equations with specified boundary conditions;
(3) eigenvalue problems, which involve solutions for selected parameters (eigenvalues) in the equations.

In reality, a problem may involve more than just one of the categories listed above. A common situation is that we have to separate several variables by introducing multipliers so that the initial-value problem is isolated from the boundary-value or eigenvalue problem. We can then solve the boundary-value or eigenvalue problem first to determine the multipliers, which in turn are used to solve the related initial-value problem. We will cover separation of variables in Chapter 7. In this chapter, we concentrate on the basic numerical methods for all the three categories listed above and illustrate how to use these techniques to solve problems encountered in physics and other related fields.

4.1 Initial-value problems

Typically, initial-value problems involve dynamical systems, for example, the motion of the Moon, Earth, and the Sun, the dynamics of a rocket, or the propagation of ocean waves. The behavior of a dynamical system can be described by a set of first-order differential equations,

$$\frac{d\mathbf{y}}{dt} = \mathbf{g}(\mathbf{y}, t), \tag{4.1}$$

where

$$\mathbf{y} = (y_1, y_2, \ldots, y_l) \tag{4.2}$$

is the dynamical variable vector, and

$$\mathbf{g}(\mathbf{y}, t) = [g_1(\mathbf{y}, t), g_2(\mathbf{y}, t), \ldots, g_l(\mathbf{y}, t)] \tag{4.3}$$

is the generalized velocity vector, a term borrowed from the definition of the velocity $\mathbf{v}(\mathbf{r}, t) = d\mathbf{r}/dt$ for a particle at position $\mathbf{r}$ and time t. Here l is the total number of dynamical variables. In principle, we can always obtain the solution of the above equation set if the initial condition $\mathbf{y}(t = 0) = \mathbf{y}_0$ is given and a solution exists. For the case of the particle moving in one dimension under an elastic force discussed in Chapter 1, the dynamics is governed by Newton's equation

$$f = ma, \tag{4.4}$$

where a and m are the acceleration and mass of the particle, respectively, and f is the force exerted on the particle. This equation can be viewed as a special case of Eq. (4.1) with $l = 2$: that is, $y_1 = x$ and $y_2 = v = dx/dt$, and $g_1 = v = y_2$ and $g_2 = f/m = -kx/m = -ky_1/m$. Then we can rewrite Newton's equation in the form of Eq. (4.1):

$$\frac{dy_1}{dt} = y_2, \tag{4.5}$$

$$\frac{dy_2}{dt} = -\frac{k}{m}y_1. \tag{4.6}$$

If the initial position $y_1(0)$ and the initial velocity $y_2(0) = v(0)$ are given, we can solve the problem numerically as demonstrated in Chapter 1.

In fact, most higher-order differential equations can be transformed into a set of coupled first-order differential equations in the form of Eq. (4.1). The higher-order derivatives are usually redefined into new dynamical variables during the transformation. The velocity in Newton's equation discussed above is such an example.

4.2 The Euler and Picard methods

For convenience of notation, we will work out our numerical schemes for cases with only one dynamical variable, that is, treating $\mathbf{y}$ and $\mathbf{g}$ in Eq. (4.1) as

one-dimensional vectors or signed scalars. Extending the formalism developed here to multivariable cases is straightforward. We will illustrate such an extension in Sections 4.3 and 4.5.

Intuitively, Eq. (4.1) can be solved numerically as was done in Chapter 1 for the problem of a particle moving in one dimension with the time derivative of the dynamical variable approximated by the average generalized velocity as

$$\frac{dy}{dt} \simeq \frac{y_{i+1} - y_i}{t_{i+1} - t_i} \simeq g(y_i, t_i). \tag{4.7}$$

Note that the indices i and $i + 1$ here are for the time steps. We will also use $g_i = g(y_i, t_i)$ to simplify the notation. The approximation of the first-order derivative in Eq. (4.7) is equivalent to the two-point formula, which has a possible error of $O(|t_{i+1} - t_i|)$. If we take evenly spaced time points with a fixed time step $\tau = t_{i+1} - t_i$ and rearrange the terms in Eq. (4.7), we obtain the simplest algorithm for initial-value problems,

$$y_{i+1} = y_i + \tau g_i + O(\tau^2), \tag{4.8}$$

which is commonly known as the *Euler method* and was used in the example of a particle moving in one dimension in Chapter 1. We can reach the same result by considering it as the Taylor expansion of y_{i+1} at t_i by keeping the terms up to the first order. The accuracy of this algorithm is relatively low. At the end of the calculation after a total of n steps, the error accumulated in the calculation is on the order of $n\,O(\tau^2) \simeq O(\tau)$. To illustrate the relative error in this algorithm, let us use the program for the one-dimensional motion given in Section 1.3. Now if we use a time step of 0.02π instead of $0.000\,02\pi$, the program can accumulate a sizable error. The results for the position and velocity of the particle in the first period are given in Fig. 4.1. The results from a better algorithm with a corrector to be discussed in Section 4.3 and the exact results are also shown for comparison. The accuracy of the Euler algorithm is very low. We have 100 points in one period of the motion, which is a typical number of points chosen in most numerical calculations. If we go on to the second period, the error will increase further. For problems without periodic motion, the results at a later time would be even worse. We can conclude that this algorithm cannot be adopted in actual numerical solutions of most physics problems. How can we improve the algorithm so that it will become practical?

We can formally rewrite Eq. (4.1) as an integral

$$y_{i+j} = y_i + \int_{t_i}^{t_{i+j}} g(y, t)\, dt, \tag{4.9}$$

which is the exact solution of Eq. (4.1) if the integral in the equation can be obtained exactly for any given integers i and j. Because we cannot obtain the integral in Eq. (4.9) exactly in general, we have to approximate it. The accuracy in the approximation of the integral determines the accuracy of the solution. If

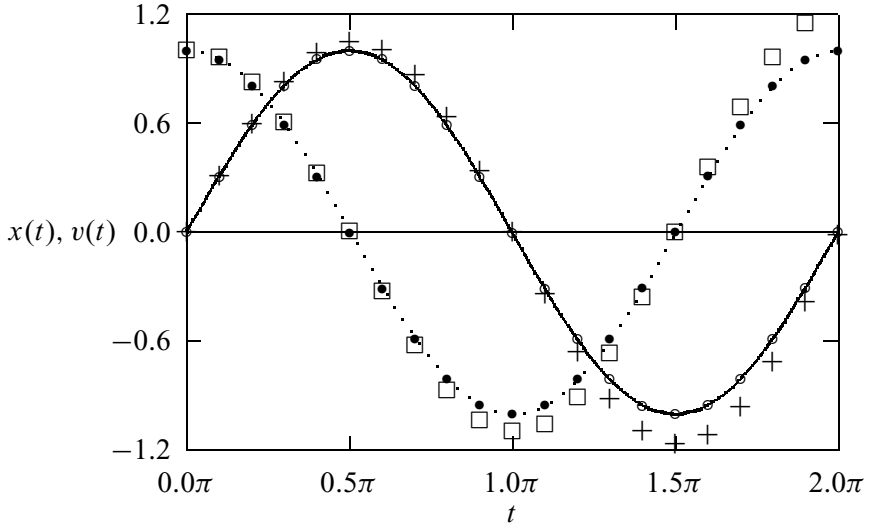

Fig. 4.1 The position (+) and velocity (□) of the particle moving in a one-dimensional space under an elastic force calculated using the Euler method with a time step of 0.02π compared with the position (○) and velocity (•) calculated with the predictor–corrector method and the exact results (solid and dotted lines).

we take the simplest case of $j = 1$ and approximate $g(y, t) \simeq g_i$ in the integral, we recover the Euler algorithm of Eq. (4.8).

The *Picard method* is an adaptive scheme, with each iteration carried out by using the solution from the previous iteration as the input on the right-hand side of Eq. (4.9). For example, we can use the solution from the Euler method as the starting point, and then carry out Picard iterations at each time step. In practice, we need to use a numerical quadrature to carry out the integration on the right-hand side of the equation. For example, if we choose $j = 1$ and use the trapezoid rule for the integral in Eq. (4.9), we obtain the algorithm

$$y_{i+1} = y_i + \frac{\tau}{2}(g_i + g_{i+1}) + O(\tau^3). \tag{4.10}$$

Note that $g_{i+1} = g(y_{i+1}, t_{i+1})$ contains y_{i+1}, which is provided adaptively in the Picard method. For example, we can take the solution at the previous time step as a guess of the solution at the current time, $y_{i+1}^{(0)} = y_i$, and then iterate the solution from right to left in the above equation, namely, $y_{i+1}^{(k+1)} = y_i + \frac{\tau}{2}(g_i + g_{i+1}^{(k)})$. The Picard method can be slow if the initial guess is not very close to the actual solution; it may not even converge if certain conditions are not satisfied. Can we avoid such tedious iterations by an intelligent guess of the solution?

4.3 Predictor–corrector methods

One way to avoid having to perform tedious iterations is to use the so-called predictor–corrector method. We can apply a less accurate algorithm to predict the next value y_{i+1} first, for example, using the Euler algorithm of Eq. (4.8), and then apply a better algorithm to improve the new value, for example, using the Picard algorithm of Eq. (4.10). If we apply this system to the one-dimensional motion studied with the Euler method, we obtain a much better result with the

same choice of time step, for example, $\tau = 0.02\pi$. The following program is the implementation of this simplest predictor–corrector method to such a problem.

```
// A program to study the motion of a particle under an
// elastic force in one dimension through the simplest
// predictor-corrector scheme.

import java.lang.*;
public class Motion2 {
  static final int n = 100, j = 5;
  public static void main(String argv[]) {
    double x[] = new double[n+1];
    double v[] = new double[n+1];

 // Assign time step and initial position and velocity
    double dt = 2*Math.PI/n;
    x[0] = 0;
    v[0] = 1;

 // Calculate other position and velocity recursively
    for (int i=0; i<n; ++i) {

   // Predict the next position and velocity
      x[i+1] = x[i]+v[i]*dt;
      v[i+1] = v[i]-x[i]*dt;

   // Correct the new position and velocity
      x[i+1] = x[i]+(v[i]+v[i+1])*dt/2;
      v[i+1] = v[i]-(x[i]+x[i+1])*dt/2;
    }

 // Output the result in every j time steps
    double t = 0;
    double jdt = j*dt;
    for (int i=0; i<=n; i+=j) {
      System.out.println(t +" " + x[i] +" " + v[i]);
      t += jdt;
    }
  }
}
```

The numerical result from the above program is shown in Fig. 4.1 for comparison. The improvement obtained is significant. With 100 mesh points in one period of motion, the errors are less than the size of the smallest symbol used in the figure. Furthermore, the improvement can also sustain long runs with more periods involved.

Another way to improve an algorithm is by increasing the number of mesh points j in Eq. (4.9). Thus we can apply a better quadrature to the integral. For example, if we take $j = 2$ in Eq. (4.9) and then use the linear interpolation scheme to approximate $g(y, t)$ in the integral from g_i and g_{i+1}, we obtain

$$g(y, t) = \frac{(t - t_i)}{\tau} g_{i+1} - \frac{(t - t_{i+1})}{\tau} g_i + O(\tau^2). \tag{4.11}$$

Now if we carry out the integration with $g(y, t)$ given from this equation, we obtain a new algorithm

$$y_{i+2} = y_i + 2\tau g_{i+1} + O(\tau^3), \tag{4.12}$$

which has an accuracy one order higher than that of the Euler algorithm. However, we need the values of the first two points in order to start this algorithm, because $g_{i+1} = g(y_{i+1}, t_{i+1})$. Usually, the dynamical variable and the generalized velocity at the second point can be obtained by the Taylor expansion around the initial point at $t = 0$. For example, we have

$$y_1 = y_0 + \tau g_0 + \frac{\tau^2}{2}\left(\frac{\partial g_0}{\partial t} + g_0\frac{\partial g_0}{\partial y}\right) + O(\tau^3) \tag{4.13}$$

and

$$g_1 = g(y_1, \tau). \tag{4.14}$$

We have truncated y_1 at $O(\tau^3)$ to preserve the same order of accuracy in the algorithm of Eq. (4.12). We have also used the fact that $dy/dt = g$.

Of course, we can always include more points in the integral of Eq. (4.9) to obtain algorithms with apparently higher accuracy, but we will need the values of more points in order to start the algorithm. This becomes impractical if we need more than two points in order to start the algorithm, because the errors accumulated from the approximations of the first few points will eliminate the apparently high accuracy of the algorithm.

We can make the accuracy even higher by using a better quadrature. For example, we can take $j = 2$ in Eq. (4.9) and apply the Simpson rule, discussed in Section 3.2, to the integral. Then we have

$$y_{i+2} = y_i + \frac{\tau}{3}(g_{i+2} + 4g_{i+1} + g_i) + O(\tau^5). \tag{4.15}$$

This implicit algorithm can be used as the corrector if the algorithm in Eq. (4.12) is used as the predictor.

Let us take the simple model of a motorcycle jump over a gap as an illustrating example. The air resistance on a moving object is roughly given by $\mathbf{f}_r = -\kappa v\mathbf{v} = -cA\rho v\mathbf{v}$, where A is cross section of the moving object, ρ is the density of the air, and c is a coefficient that accounts for all the other factors on the order of 1. So the motion of the system is described by the equation set

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}, \tag{4.16}$$

$$\frac{d\mathbf{v}}{dt} = \mathbf{a} = \frac{\mathbf{f}}{m}, \tag{4.17}$$

where

$$\mathbf{f} = -mg\hat{\mathbf{y}} - \kappa v\mathbf{v} \tag{4.18}$$

is the total force on the system of a total mass m. Here $\hat{\mathbf{y}}$ is the unit vector pointing upward. Assuming that we have the first point given, that is, $\mathbf{r}_0$ and $\mathbf{v}_0$ at $t = 0$, the next point is then obtained from the Taylor expansions and the equation set with

$$\mathbf{r}_1 = \mathbf{r}_0 + \tau \mathbf{v}_0 + \frac{\tau^2}{2}\mathbf{a}_0 + O(\tau^3), \tag{4.19}$$

$$\mathbf{v}_1 = \mathbf{v}_0 + \tau \mathbf{a}_0 + \frac{\tau^2}{2}\frac{d\mathbf{a}_0}{dt} + O(\tau^3), \tag{4.20}$$

where

$$\frac{d\mathbf{a}_0}{dt} = -\frac{\kappa}{m}\left(v_0\mathbf{a}_0 + \frac{\mathbf{v}_0 \cdot \mathbf{a}_0}{v_0}\mathbf{v}_0\right). \tag{4.21}$$

The following program calculates the trajectory of the motorcycle with a given taking-off angle.

```
// An example of modeling a motorcycle jump with the
// two-point predictor-corrector scheme.

import java.lang.*;
public class Jump {
  static final int n = 100, j = 2;
  public static void main(String argv[]) {
    double x[] = new double[n+1];
    double y[] = new double[n+1];
    double vx[] = new double[n+1];
    double vy[] = new double[n+1];
    double ax[] = new double[n+1];
    double ay[] = new double[n+1];

 // Assign all the constants involved
    double g = 9.80;
    double angle = 42.5*Math.PI/180;
    double speed = 67;
    double mass = 250;
    double area = 0.93;
    double density = 1.2;
    double k = area*density/(2*mass);
    double dt = 2*speed*Math.sin(angle)/(g*n);
    double d = dt*dt/2;

 // Assign the quantities for the first two points
    x[0] = y[0] = 0;
    vx[0] = speed*Math.cos(angle);
    vy[0] = speed*Math.sin(angle);
    double v = Math.sqrt(vx[0]*vx[0]+vy[0]*vy[0]);
    ax[0] = -k*v*vx[0];
    ay[0] = -g-k*v*vy[0];
    double p = vx[0]*ax[0]+vy[0]*ay[0];
    x[1] = x[0]+dt*vx[0]+d*ax[0];
    y[1] = y[0]+dt*vy[0]+d*ay[0];
    vx[1] = vx[0]+dt*ax[0]-d*k*(v*ax[0]+p*vx[0]/v);
    vy[1] = vy[0]+dt*ay[0]-d*k*(v*ay[0]+p*vy[0]/v);
```

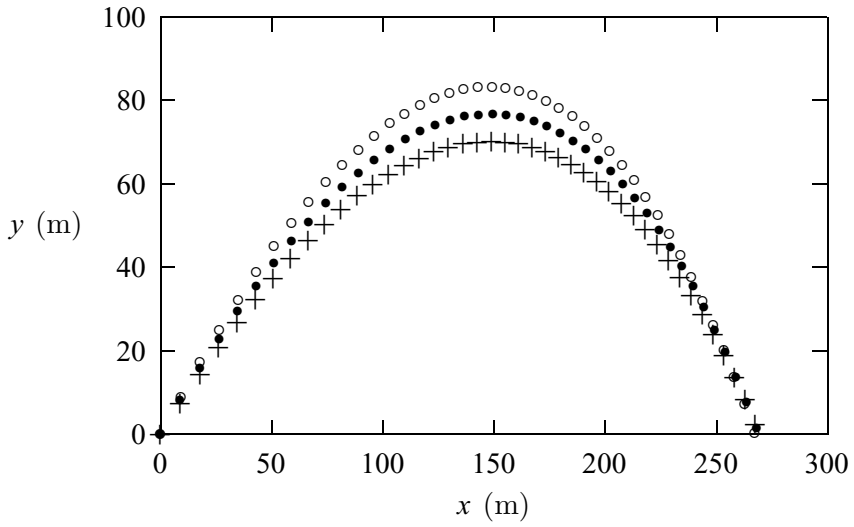

Fig. 4.2 The trejactory of the motorcycle as calculated in the example program with different taking-off angles: 40° (+), 42.5° (•), and 45° (○).

```
    v = Math.sqrt(vx[1]*vx[1]+vy[1]*vy[1]);
    ax[1] = -k*v*vx[1];
    ay[1] = -g-k*v*vy[1];

 // Calculate other position and velocity recursively
    double d2 = 2*dt;
    double d3 = dt/3;
    for (int i=0; i<n-1; ++i) {

   // Predict the next position and velocity
      x[i+2] = x[i]+d2*vx[i+1];
      y[i+2] = y[i]+d2*vy[i+1];
      vx[i+2] = vx[i]+d2*ax[i+1];
      vy[i+2] = vy[i]+d2*ay[i+1];
      v = Math.sqrt(vx[i+2]*vx[i+2]+vy[i+2]*vy[i+2]);
      ax[i+2] = -k*v*vx[i+2];
      ay[i+2] = -g-k*v*vy[i+2];

   // Correct the new position and velocity
      x[i+2] = x[i]+d3*(vx[i+2]+4*vx[i+1]+vx[i]);
      y[i+2] = y[i]+d3*(vy[i+2]+4*vy[i+1]+vy[i]);
      vx[i+2] = vx[i]+d3*(ax[i+2]+4*ax[i+1]+ax[i]);
      vy[i+2] = vy[i]+d3*(ay[i+2]+4*ay[i+1]+ay[i]);
    }

 // Output the result in every j time steps
    for (int i=0; i<=n; i+=j)
      System.out.println(x[i] +" " + y[i]);
  }
}
```

In the above program, we have used the cross section $A = 0.93$ m^2, the taking-off speed $v_0 = 67$ m/s, the air density $\rho = 1.2$ kg/m^3, the combined mass of the motorcycle and the person 250 kg, and the coefficient $c = 1$. The results for three different taking-off angles are plotted in Fig. 4.2. The maximum range is about 269 m at a taking-off angle of about 42.5°.

In principle, we can go further by including more points in the integration quadrature of Eq. (4.9) and interested readers can find many multiple-point formulas in Davis and Polonsky (1965).

4.4 The Runge–Kutta method

The accuracy of the methods that we have discussed so far can be improved only by including more starting points, which is not always practical, because a problem associated with a dynamical system usually has only the first point, namely, the initial condition, given. A more practical method that requires only the first point in order to start or to improve the algorithm is the *Runge–Kutta method*, which is derived from two different Taylor expansions of the dynamical variables and their derivatives defined in Eq. (4.1).

Formally, we can expand $y(t+\tau)$ in terms of the quantities at t with the Taylor expansion

$$\begin{aligned} y(t+\tau) &= y + \tau y' + \frac{\tau^2}{2} y'' + \frac{\tau^3}{3!} y^{(3)} + \cdots \\ &= y + \tau g + \frac{\tau^2}{2}(g_t + g g_y) + \frac{\tau^3}{6}\left(g_{tt} + 2 g g_{ty} + g^2 g_{yy} + g g_y^2 + g_t g_y\right) + \cdots , \end{aligned} \tag{4.22}$$

where the subscript indices denote partial derivatives for example, $g_{yt} = \partial^2 g / \partial y \partial t$. We can also formally write the solution at $t+\tau$ as

$$y(t+\tau) = y(t) + \alpha_1 c_1 + \alpha_2 c_2 + \cdots + \alpha_m c_m, \tag{4.23}$$

with

$$\begin{aligned} c_1 &= \tau g(y, t), \\ c_2 &= \tau g(y + \nu_{21} c_1, t + \nu_{21}\tau), \\ c_3 &= \tau g(y + \nu_{31} c_1 + \nu_{32} c_2, t + \nu_{31}\tau + \nu_{32}\tau), \\ &\vdots \\ c_m &= \tau g\left(y + \sum_{i=1}^{m-1} \nu_{mi} c_i, t + \tau \sum_{i=1}^{m-1} \nu_{mi}\right), \end{aligned} \tag{4.24}$$

where α_i (with $i = 1, 2, \ldots, m$) and ν_{ij} (with $i = 2, 3, \ldots, m$ and $j < i$) are parameters to be determined. We can expand Eq. (4.23) into a power series of τ by carrying out Taylor expansions for all c_i with $i = 1, 2, \ldots, m$. Then we can compare the resulting expression of $y(t+\tau)$ from Eq. (4.23) with the expansion in Eq. (4.22) term by term. A set of equations for α_i and ν_{ij} is obtained by keeping the coefficients for the terms with the same power of τ on both sides equal. By truncating the expansion to the term $O(\tau^m)$, we obtain m equations but with $m + m(m-1)/2$ parameters (α_i and ν_{ij}) to be determined. Thus, there are still options left in choosing these parameters.

Let us illustrate this scheme by working out the case for $m = 2$ in detail. If only the terms up to $O(\tau^2)$ are kept in Eq. (4.22), we have

$$y(t+\tau) = y + \tau g + \frac{\tau^2}{2}(g_t + g g_y). \tag{4.25}$$

We can obtain an expansion up to the same order by truncating Eq. (4.23) at $m = 2$,

$$y(t+\tau) = y(t) + \alpha_1 c_1 + \alpha_2 c_2, \tag{4.26}$$

with

$$c_1 = \tau g(y, t), \tag{4.27}$$
$$c_2 = \tau g(y + \nu_{21} c_1, t + \nu_{21}\tau). \tag{4.28}$$

Now if we perform the Taylor expansion for c_2 up to the term $O(\tau^2)$, we have

$$c_2 = \tau g + \nu_{21}\tau^2(g_t + g g_y). \tag{4.29}$$

Substituting c_1 and the expansion of c_2 above back into Eq. (4.26) yields

$$y(t+\tau) = y(t) + (\alpha_1 + \alpha_2)\tau g + \alpha_2 \tau^2 \nu_{21}(g_t + g g_y). \tag{4.30}$$

If we compare this expression with Eq. (4.25) term by term, we have

$$\alpha_1 + \alpha_2 = 1, \tag{4.31}$$
$$\alpha_2 \nu_{21} = \frac{1}{2}. \tag{4.32}$$

As pointed out earlier, there are only m (2 in this case) equations available but there are $m + m(m-1)/2$ (3 in this case) parameters to be determined. We do not have a unique solution for all the parameters; thus we have flexibility in assigning their values as long as they satisfy the m equations. We could choose $\alpha_1 = \alpha_2 = 1/2$ and $\nu_{21} = 1$, or $\alpha_1 = 1/3$, $\alpha_2 = 2/3$, and $\nu_{21} = 3/4$. The flexibility in choosing the parameters provides one more way to increase the numerical accuracy in practice. We can adjust the parameters according to the specific problems involved.

The most commonly known and widely used Runge–Kutta method is the one with Eqs. (4.22) and (4.23) truncated at the terms of $O(\tau^4)$. We will give the result here and leave the derivation as an exercise to the reader. This well-known fourth-order Runge–Kutta algorithm is given by

$$y(t+\tau) = y(t) + \frac{1}{6}(c_1 + 2c_2 + 2c_3 + c_4), \tag{4.33}$$

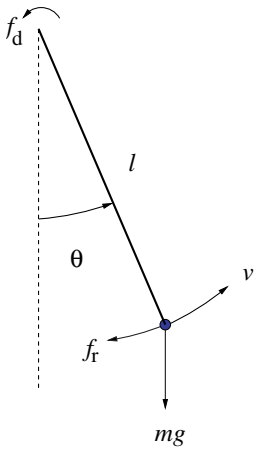

Fig. 4.3 A sketch of a driven pendulum under damping: f_d is the driving force and f_r is the resistive force.

with

$$c_1 = \tau g(y, t), \tag{4.34}$$

$$c_2 = \tau g\left(y + \frac{c_1}{2}, t + \frac{\tau}{2}\right), \tag{4.35}$$

$$c_3 = \tau g\left(y + \frac{c_2}{2}, t + \frac{\tau}{2}\right), \tag{4.36}$$

$$c_4 = \tau g(y + c_3, t + \tau). \tag{4.37}$$

We can easily show that the above selection of parameters does satisfy the required equations. As pointed out earlier, this selection is not unique and can be modified according to the problem under study.

4.5 Chaotic dynamics of a driven pendulum

Before discussing numerical methods for solving boundary-value and eigenvalue problems, let us apply the Runge–Kutta method to the initial-value problem of a dynamical system. Even though we are going to examine only one special system, the approach, as shown below, is quite general and suitable for all other problems.

Consider a pendulum consisting of a light rod of length l and a point mass m attached to the lower end. Assume that the pendulum is confined to a vertical plane, acted upon by a driving force f_d and a resistive force f_r as shown in Fig. 4.3. The motion of the pendulum is described by Newton's equation along the tangential direction of the circular motion of the point mass,

$$ma_t = f_g + f_d + f_r, \tag{4.38}$$

where $f_g = -mg \sin\theta$ is the contribution of gravity along the direction of motion, with θ being the angle made by the rod with respect to the vertical line, and $a_t = ld^2\theta/dt^2$ is the acceleration along the tangential direction. Assume that the

time dependency of the driving force is periodic as

$$f_{\mathrm{d}}(t) = f_0 \cos \omega_0 t, \tag{4.39}$$

and the resistive force $f_{\mathrm{r}} = -\kappa v$, where $v = l d\theta/dt$ is the velocity of the mass and κ is a positive damping parameter. This is a reasonable assumption for a pendulum set in a dense medium under a harmonic driving force. If we rewrite Eq. (4.38) in a dimensionless form with $\sqrt{l/g}$ chosen as the unit of time, we have

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \sin\theta = b\cos\omega_0 t, \tag{4.40}$$

where $q = \kappa/m$ and $b = f_0/ml$ are redefined parameters. As discussed at the beginning of this chapter, we can write the derivatives as variables. We can thus transform higher-order differential equations into a set of first-order differential equations. If we choose $y_1 = \theta$ and $y_2 = \omega = d\theta/dt$, we have

$$\frac{dy_1}{dt} = y_2, \tag{4.41}$$

$$\frac{dy_2}{dt} = -qy_2 - \sin y_1 + b\cos\omega_0 t, \tag{4.42}$$

which are in the form of Eq. (4.1). In principle, we can use any method discussed so far to solve this equation set. However, considering the accuracy required for long-time behavior, we use the fourth-order Runge–Kutta method here.

As we will show later from the numerical solutions of Eqs. (4.41) and (4.42), in different regions of the parameter space (q, b, ω_0) the system has quite different dynamics. Specifically, in some parameter regions the motion of the pendulum is totally chaotic.

If we generalize the fourth-order Runge–Kutta method discussed in the preceding section to multivariable cases, we have

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{1}{6}(\mathbf{c}_1 + 2\mathbf{c}_2 + 2\mathbf{c}_3 + \mathbf{c}_4), \tag{4.43}$$

with

$$\mathbf{c}_1 = \tau\mathbf{g}(\mathbf{y}_i, t_i), \tag{4.44}$$

$$\mathbf{c}_2 = \tau\mathbf{g}\left(\mathbf{y}_i + \frac{\mathbf{c}_1}{2}, t_i + \frac{\tau}{2}\right), \tag{4.45}$$

$$\mathbf{c}_3 = \tau\mathbf{g}\left(\mathbf{y}_i + \frac{\mathbf{c}_2}{2}, t_i + \frac{\tau}{2}\right), \tag{4.46}$$

$$\mathbf{c}_4 = \tau\mathbf{g}(\mathbf{y}_i + \mathbf{c}_3, t_i + \tau), \tag{4.47}$$

where $\mathbf{y}_i$ for any i and $\mathbf{c}_j$ for $j = 1, 2, 3, 4$ are multidimensional vectors. Note that generalizing an algorithm for the initial-value problem from the single-variable case to the multivariable case is straightforward. Other algorithms we have discussed can be generalized in exactly the same fashion.

In principle, the pendulum problem has three dynamical variables: the angle between the rod and the vertical line, θ, its first-order derivative $\omega = d\theta/dt$, and

the phase of the driving force $\phi = \omega_0 t$. This is important because a dynamical system cannot be chaotic unless it has three or more dynamical variables. However in practice, we only need to worry about θ and ω because $\phi = \omega_0 t$ is the solution of ϕ.

Any physical quantities that are functions of θ are periodic: for example, $\omega(\theta) = \omega(\theta \pm 2n\pi)$, where n is an integer. Therefore, we can confine θ in the region $[-\pi, \pi]$. If θ is outside this region, it can be transformed back with $\theta' = \theta \pm 2n\pi$ without losing any generality. The following program is an implementation of the fourth-order Runge–Kutta algorithm as applied to the driven pendulum under damping.

```
// A program to study the driven pendulum under damping
// via the fourth-order Runge-Kutta algorithm.

import java.lang.*;
public class Pendulum {
  static final int n = 100, nt = 10, m = 5;
  public static void main(String argv[]) {
    double y1[] = new double[n+1];
    double y2[] = new double[n+1];
    double y[] = new double[2];

 // Set up time step and initial values
    double dt = 3*Math.PI/nt;
    y1[0] = y[0] = 0;
    y2[0] = y[1] = 2;

 // Perform the 4th-order Runge-Kutta integration
    for (int i=0; i<n; ++i) {
      double t = dt*i;
      y = rungeKutta(y, t, dt);
      y1[i+1] = y[0];
      y2[i+1] = y[1];

   // Bring theta back to the region [-pi, pi]
      int np = (int) (y1[i+1]/(2*Math.PI)+0.5);
      y1[i+1] -= 2*Math.PI*np;
    }

 // Output the result in every m time steps
    for (int i=0; i<=n; i+=m) {
      System.out.println("Angle: " + y1[i]);
      System.out.println("Angular velocity: " + y2[i]);
      System.out.println();
    }
  }

// Method to complete one Runge-Kutta step.

  public static double[] rungeKutta(double y[],
    double t, double dt) {
    int l = y.length;
    double c1[] = new double[l];
    double c2[] = new double[l];
    double c3[] = new double[l];
    double c4[] = new double[l];
```

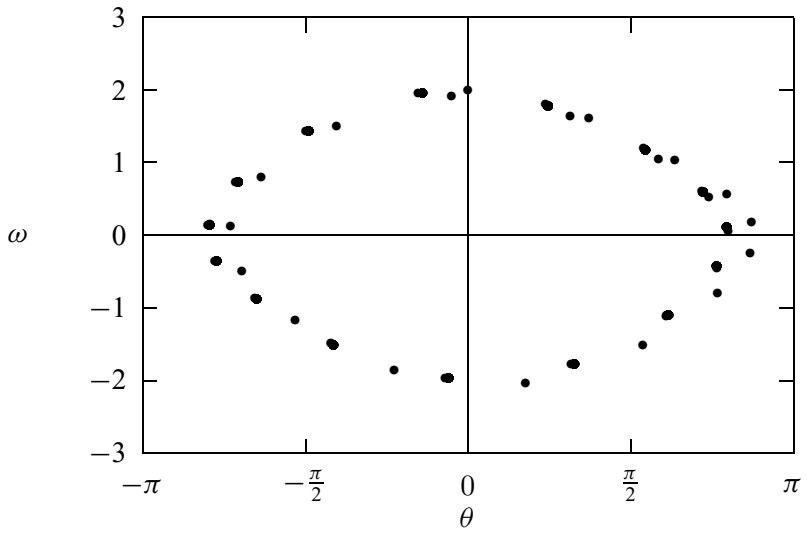

Fig. 4.4 The angular velocity ω versus the angle θ, with parameters $\omega_0 = 2/3$, $q = 0.5$, and $b = 0.9$. Under the given condition the system is apparently periodic. Here 1000 points from 10 000 time steps are shown.

```
    c1 = g(y, t);
    for (int i=0; i<l; ++i) c2[i] = y[i] + dt*c1[i]/2;
    c2 = g(c2, t+dt/2);
    for (int i=0; i<l; ++i) c3[i] = y[i] + dt*c2[i]/2;
    c3 = g(c3, t+dt/2);
    for (int i=0; i<l; ++i) c4[i] = y[i] + dt*c3[i];
    c4 = g(c4, t+dt);
    for (int i=0; i<l; ++i)
      c1[i] = y[i] + dt*(c1[i]+2*(c2[i]+c3[i])+c4[i])/6;
    return c1;
  }
// Method to provide the generalized velocity vector.
  public static double[] g(double y[], double t) {
    int l = y.length;
    double q = 0.5, b = 0.9, omega0 = 2.0/3;
    double v[] = new double[l];
    v[0] = y[1];
    v[1] = -Math.sin(y[0])+b*Math.cos(omega0*t);
    v[1] -= q*y[1];
    return v;
  }
}
```

Depending on the choice of the three parameters, q, b, and ω_0, the system can be periodic or chaotic. In Fig. 4.4 and Fig. 4.5, we show two typical numerical results. The dynamical behavior of the pendulum shown in Fig. 4.4 is periodic in the selected parameter region, and the dynamical behavior shown in Fig. 4.5 is chaotic in another parameter region. We can modify the program developed here to explore the dynamics of the pendulum through the whole parameter space and many important aspects of chaos. Interested readers can find discussions on these aspects in Baker and Gollub (1996).

Several interesting features appear in the results shown in Fig. 4.4 and Fig. 4.5. In Fig. 4.4, the motion of the system is periodic, with a period $T = 2T_0$, where

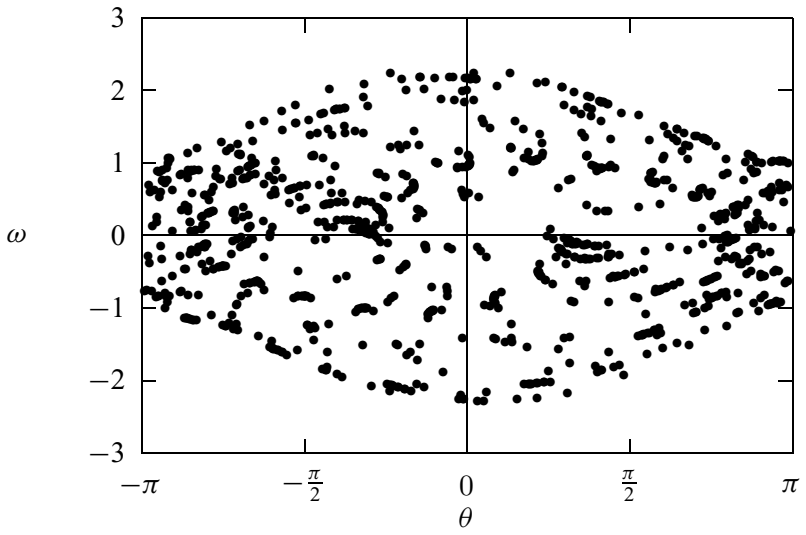

Fig. 4.5 The same plot as in Fig. 4.4, with parameters $\omega_0 = 2/3$, $q = 0.5$, and $b = 1.15$. The system at this point of the parameter space is apparently chaotic. Here 1000 points from 10 000 time steps are shown.

$T_0 = 2\pi/\omega_0$ is the period of the driving force. If we explore other parameter regions, we would find other periodic motions with $T = nT_0$, where n is an even, positive integer. The reason why n is even is that the system is moving away from being periodic to being chaotic; period doubling is one of the routes for a dynaimcal system to develop chaos. The chaotic behavior shown in Fig. 4.5 appears to be totally irregular; however, detailed analysis shows that the phase-space diagram (the ω–θ plot) has self-similarity at all length scales, as indicated by the fractal structure in chaos.

4.6 Boundary-value and eigenvalue problems

Another class of problems in physics requires the solving of differential equations with the values of physical quantities or their derivatives given at the boundaries of a specified region. This applies to the solution of the Poisson equation with a given charge distribution and known boundary values of the electrostatic potential or of the stationary Schrödinger equation with a given potential and boundary conditions.

A typical boundary-value problem in physics is usually given as a second-order differential equation

$$u'' = f(u, u'; x), \tag{4.48}$$

where u is a function of x, u' and u'' are the first-order and second-order derivatives of u with respect to x, and $f(u, u'; x)$ is a function of u, u', and x. Either u or u' is given at each boundary point. Note that we can always choose a coordinate system so that the boundaries of the system are at $x = 0$ and $x = 1$ without losing any generality if the system is finite. For example, if the actual boundaries are at $x = x_1$ and $x = x_2$ for a given problem, we can always bring them back to $x' = 0$

and $x' = 1$ with a transformation

$$x' = (x - x_1)/(x_2 - x_1). \tag{4.49}$$

For problems in one dimension, we can have a total of four possible types of boundary conditions:

(1) $u(0) = u_0$ and $u(1) = u_1$;
(2) $u(0) = u_0$ and $u'(1) = v_1$;
(3) $u'(0) = v_0$ and $u(1) = u_1$;
(4) $u'(0) = v_0$ and $u'(1) = v_1$.

The boundary-value problem is more difficult to solve than the similar initial-value problem with the differential equation. For example, if we want to solve an initial-value problem that is described by the differential equation given in Eq. (4.48) with x replaced by time t and the initial conditions $u(0) = u_0$ and $u'(0) = v_0$, we can first transform the differential equation as a set of two first-order differential equations with a redefinition of the first-order derivative as a new variable. The solution will follow if we adopt one of the algorithms discussed earlier in this chapter. However, for the boundary-value problem, we know only $u(0)$ or $u'(0)$, which is not sufficient to start an algorithm for the initial-value problem without some further work.

Typical eigenvalue problems are even more complicated, because at least one more parameter, that is, the eigenvalue, is involved in the equation: for example,

$$u'' = f(u, u'; x; \lambda), \tag{4.50}$$

with a set of given boundary conditions, defines an eigenvalue problem. Here the eigenvalue λ can have only some selected values in order to yield acceptable solutions of the equation under the given boundary conditions.

Let us take the longitudinal vibrations along an elastic rod as an illustrative example here. The equation describing the stationary solution of elastic waves is

$$u''(x) = -k^2 u(x), \tag{4.51}$$

where $u(x)$ is the displacement from equilibrium at x and the allowed values of k^2 are the eigenvalues of the problem. The wavevector k in the equation is related to the phase speed c of the wave along the rod and the allowed angular frequency ω by the dispersion relation

$$\omega = ck. \tag{4.52}$$

If both ends ($x = 0$ and $x = 1$) of the rod are fixed, the boundary conditions are then $u(0) = u(1) = 0$. If one end ($x = 0$) is fixed and the other end ($x = 1$) is free, the boundary conditions are then $u(0) = 0$ and $u'(1) = 0$. For this problem, we can obtain an analytical solution. For example, if both ends of the rod are

fixed, the eigenfunctions

$$u_l(x) = \sqrt{2}\sin k_l x \tag{4.53}$$

are the possible solutions of the differential equation. Here the eigenvalues are given by

$$k_l^2 = (l\pi)^2, \tag{4.54}$$

with $l = 1, 2, \ldots, \infty$. The complete solution of the longitudinal waves along the elastic rod is given by a linear combination of all the eigenfunctions with their associated initial-value solutions as

$$u(x, t) = \sum_{l=1}^{\infty}(a_l \sin \omega_l t + b_l \cos \omega_l t)u_l(x), \tag{4.55}$$

where $\omega_l = ck_l$, and a_l and b_l are the coefficients to be determined by the initial conditions. We will come back to this problem in Chapter 7 when we discuss the solution of a partial differential equation.

4.7 The shooting method

A simple method for solving the boundary-value problem of Eq. (4.48) and the eigenvalue problem of Eq. (4.50) with a set of given boundary conditions is the so-called *shooting method*. We will first discuss how this works for the boundary-value problem and then generalize it to the eigenvalue problem.

We first convert the second-order differential equation into two first-order differential equations by defining $y_1 = u$ and $y_2 = u'$, namely,

$$\frac{dy_1}{dx} = y_2, \tag{4.56}$$

$$\frac{dy_2}{dx} = f(y_1, y_2; x). \tag{4.57}$$

To illustrate the method, let us assume that the boundary conditions are $u(0) = u_0$ and $u(1) = u_1$. Any other types of boundary conditions can be solved in a similar manner.

The key here is to make the problem look like an initial-value problem by introducing an adjustable parameter; the solution is then obtained by varying the parameter. Because $u(0)$ is given already, we can make a guess for the first-order derivative at $x = 0$, for example, $u'(0) = \alpha$. Here α is the parameter to be adjusted. For a specific α, we can integrate the equation to $x = 1$ with any of the algorithms discussed earlier for initial-value problems. Because the initial choice of α can hardly be the actual derivative at $x = 0$, the value of the function $u_\alpha(1)$, resulting from the integration with $u'(0) = \alpha$ to $x = 1$, would not be the same as u_1. The idea of the shooting method is to use one of the root search algorithms to find the appropriate α that ensures $f(\alpha) = u_\alpha(1) - u_1 = 0$ within a given tolerance δ.

Let us take an actual numerical example to illustrate the scheme. Assume that we want to solve the differential equation

$$u'' = -\frac{\pi^2}{4}(u+1), \tag{4.58}$$

with the given boundary conditions $u(0) = 0$ and $u(1) = 1$. We can define the new variables $y_1 = u$ and $y_2 = u'$; then we have

$$\frac{dy_1}{dx} = y_2, \tag{4.59}$$

$$\frac{dy_2}{dx} = -\frac{\pi^2}{4}(y_1+1). \tag{4.60}$$

Now assume that this equation set has the initial values $y_1(0) = 0$ and $y_2(0) = \alpha$. Here α is a parameter to be adjusted in order to have $f(\alpha) = u_\alpha(1) - 1 = 0$. We can combine the secant method for the root search and the fourth-order Runge–Kutta method for initial-value problems to solve the above equation set. The following program is an implementation of such a combined approach to the boundary-value problem defined in Eq. (4.58) or Eqs. (4.59) and (4.60) with the given boundary conditions.

```
// An example of solving a boundary-value problem via
// the shooting method.  The Runge-Kutta and secant
// methods are used for integration and root search.

import java.lang.*;
public class Shooting {
  static final int n = 100, ni=10, m = 5;
  static final double h = 1.0/n;
  public static void main(String argv[]) {
    double del = 1e-6, alpha0 = 1, dalpha = 0.01;
    double y1[] = new double [n+1];
    double y2[] = new double [n+1];
    double y[] = new double [2];

 // Search for the proper solution of the equation
    y1[0] = y[0] = 0;
    y2[0] = y[1] = secant(ni, del, alpha0, dalpha);
    for (int i=0; i<n; ++i) {
      double x = h*i;
      y = rungeKutta(y, x, h);
      y1[i+1] = y[0];
      y2[i+1] = y[1];
    }

 // Output the result in every m points
    for (int i=0; i<=n; i+=m)
      System.out.println(y1[i]);
  }

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

// Method to provide the function for the root search.

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

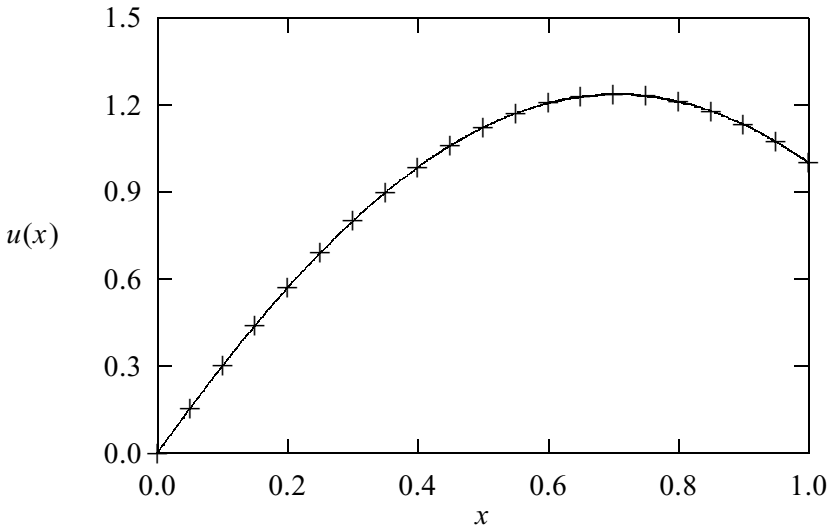

Fig. 4.6 The numerical solution of the boundary-value problem of Eq. (4.58) by the shooting method (+) compared with the analytical solution (solid line) of the same problem.

```
    double y[] = new double[2];
    y[0] = 0;
    y[1] = x;
    for (int i=0; i<n-1; ++i) {
      double xi = h*i;
      y = rungeKutta(y, xi, h);
    }
    return y[0]-1;
  }

  public static double[] rungeKutta(double y[],
    double t, double dt) {...}

// Method to provide the generalized velocity vector.

  public static double[] g(double y[], double t) {
    int k = y.length;
    double v[] = new double[k];
    v[0] = y[1];
    v[1] = -Math.PI*Math.PI*(y[0]+1)/4;
    return v;
  }
}
```

Note how we have combined the secant and Runge–Kutta methods. The boundary-value problem solved in the above program can also be solved exactly with an analytical solution

$$u(x) = \cos\frac{\pi x}{2} + 2\sin\frac{\pi x}{2} - 1. \tag{4.61}$$

We can easily check that the above expression does satisfy the equation and the boundary conditions. We plot both the numerical result obtained from the shooting method and the analytical solution in Fig. 4.6. Note that the shooting method provides a very accurate solution of the boundary-value problem. It is also a very general method for both the boundary-value and eigenvalue problems.

Boundary-value problems with other types of boundary conditions can be solved in a similar manner. For example, if $u'(0) = v_0$ and $u(1) = u_1$ are given,

we can make a guess of $u(0) = \alpha$ and then integrate the equation set of y_1 and y_2 to $x = 1$. The root to be sought is from $f(\alpha) = u_\alpha(1) - u_1 = 0$. Here $u_\alpha(1)$ is the numerical result of the equation with $u(0) = \alpha$. If $u'(1) = v_1$ is given, the equation $f(\alpha) = u'_\alpha(1) - v_1 = 0$ is solved instead.

When we apply the shooting method to an eigenvalue problem, the parameter to be adjusted is no longer a parameter introduced but the eigenvalue of the problem. For example, if $u(0) = u_0$ and $u(1) = u_1$ are given, we can integrate the equation with $u'(0) = \alpha$, a small quantity. Then we search for the root of $f(\lambda) = u_\lambda(1) - u_1 = 0$ by varying λ. When $f(\lambda) = 0$ is satisfied, we obtain an approximate eigenvalue λ and the corresponding eigenstate from the normalized solution of $u_\lambda(x)$. The introduced parameter α is not relevant here, because it will be automatically modified to be the first-order derivative when the solution in normalized. In other words, we can choose the first-order derivative at the boundary arbitrarily and it will be adjusted to an appropriate value when the solutions are made orthonormal. We will demonstrate this in Section 4.9 with examples.

4.8 Linear equations and the Sturm–Liouville problem

Many eigenvalue or boundary-value problems are in the form of linear equations, such as

$$u'' + d(x)u' + q(x)u = s(x), \tag{4.62}$$

where $d(x)$, $q(x)$, and $s(x)$ are functions of x. Assume that the boundary conditions are $u(0) = u_0$ and $u(1) = u_1$. If all $d(x)$, $q(x)$, and $s(x)$ are smooth, we can solve the equation with the shooting method developed in the preceding section. In fact, we can show that an extensive search for the parameter α from $f(\alpha) = u_\alpha(1) - u_1 = 0$ is unnecessary in this case, because of the principle of superposition of linear equations: any linear combination of the solutions is also a solution of the equation. We need only two trial solutions $u_{\alpha_0}(x)$ and $u_{\alpha_1}(x)$, where α_0 and α_1 are two different parameters. The correct solution of the equation is given by

$$u(x) = au_{\alpha_0}(x) + bu_{\alpha_1}(x), \tag{4.63}$$

where a and b are determined from $u(0) = u_0$ and $u(1) = u_1$. Note that $u_{\alpha_0}(0) = u_{\alpha_1}(0) = u(0) = u_0$. So we have

$$a + b = 1, \tag{4.64}$$

$$u_{\alpha_0}(1)a + u_{\alpha_1}(1)b = u_1, \tag{4.65}$$

which lead to

$$a = \frac{u_{\alpha_1}(1) - u_1}{u_{\alpha_1}(1) - u_{\alpha_0}(1)}, \tag{4.66}$$

$$b = \frac{u_1 - u_{\alpha_0}(1)}{u_{\alpha_1}(1) - u_{\alpha_0}(1)}. \tag{4.67}$$

With a and b given by the above equations, we reach the solution of the differential equation from Eq. (4.63).

Here we would like to demonstrate the application of the above scheme to the linear equation problem defined in Eq. (4.58). The following example program is an implementation of the scheme.

```
// An example of solving the boundary-value problem of
// a linear equation via the Runge-Kutta method.

import java.lang.*;
public class LinearDEq {
  static final int n = 100, m = 5;
  public static void main(String argv[]) {
    double y1[] = new double [n+1];
    double y2[] = new double [n+1];
    double y[] = new double [2];
    double h = 1.0/n;

 // Find the 1st solution via the Runge-Kutta method
    y[1]  = 1;
    for (int i=0; i<n; ++i) {
      double x = h*i;
      y = rungeKutta(y, x, h);
      y1[i+1] = y[0];
    }

 // Find the 2nd solution via the Runge-Kutta method
    y[0] = 0;
    y[1] = 2;
    for (int i=0; i<n; ++i) {
      double x = h*i;
      y = rungeKutta(y, x, h);
      y2[i+1] = y[0];
    }

 // Superpose the two solutions found
    double a = (y2[n]-1)/(y2[n]-y1[n]);
    double b = (1-y1[n])/(y2[n]-y1[n]);
    for (int i=0; i<=n; ++i)
      y1[i] = a*y1[i]+b*y2[i];

 // Output the result in every m points
    for (int i=0; i<=n; i+=m)
      System.out.println(y1[i]);
  }

  public static double[] rungeKutta(double y[],
    double t, double dt) {...}

  public static double[] g(double y[], double t) {...}
}
```

Note that we have only integrated the equation twice in the above example program. This is in clear contrast to the general shooting method, in which many more integrations may be needed depending on how fast the solution converges.

An important class of linear equations in physics is known as the Sturm–Liouville problem, defined by

$$[p(x)u'(x)]' + q(x)u(x) = s(x), \tag{4.68}$$

which has the first-order derivative term combined with the second-order derivative term. Here $p(x)$, $q(x)$, and $s(x)$ are the coefficient functions of x. For most actual problems, $s(x) = 0$ and $q(x) = -r(x) + \lambda w(x)$, where λ is the eigenvalue of the equation and $r(x)$ and $w(x)$ are the redefined coefficient functions. The Legendre equation, the Bessel equation, and their related equations in physics are examples of the Sturm–Liouville problem.

Our goal here is to construct an accurate algorithm which can integrate the Sturm–Liouville problem, that is, Eq. (4.68). In Chapter 3, we obtained the three-point formulas for the first-order and second-order derivatives from the combinations

$$\Delta_1 = \frac{u_{i+1} - u_{i-1}}{2h} = u_i' + \frac{h^2 u_i^{(3)}}{6} + O(h^4) \tag{4.69}$$

and

$$\Delta_2 = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = u_i'' + \frac{h^2 u_i^{(4)}}{12} + O(h^4). \tag{4.70}$$

Now if we multiply Eq. (4.69) by p_i' and Eq. (4.70) by p_i and add them together, we have

$$p_i'\Delta_1 + p_i\Delta_2 = (p_i u_i')' + \frac{h^2}{12}\left(p_i u_i^{(4)} + 2p_i' u_i^{(3)}\right) + O(h^4). \tag{4.71}$$

If we replace the first term on the right-hand side with $s_i - q_i u_i$ by applying the original differential equation and drop the second term, we obtain the simplest numerical algorithm for the Sturm–Liouville problem:

$$(2p_i + hp_i')u_{i+1} + (2p_i - hp_i')u_{i-1} = 4p_i u_i + 2h^2(s_i - q_i u_i), \tag{4.72}$$

which is accurate up to $O(h^4)$. Before we discuss how to improve the accuracy of this algorithm, let us work on an illustrating example. The Legendre equation is given by

$$\frac{d}{dx}\left[(1 - x^2)\frac{du}{dx}\right] + l(l+1)u = 0, \tag{4.73}$$

with $l = 0, 1, \ldots, \infty$ and $x \in [-1, 1]$. The solutions of the Legendre equation are the Legendre polynomials $P_l(x)$. Let us assume that we do not know the value of l but know the first two points of $P_1(x) = x$; then we can treat the problem as an eigenvalue problem.

The following program is an implementation of the simplest algorithm for the Sturm–Liouville problem, in combination with the secant method for the root search, to solve for the eigenvalue $l = 1$ of the Legendre equation.

```
// An example of implementing the simplest method to
// solve the Sturm-Liouville problem.

import java.lang.*;
public class Sturm {
  static final int n = 100, ni = 10;
  public static void main(String argv[]) {
    double del = 1e-6, l = 0.5, dl = 0.1;
    l = secant(ni, del, l, dl);

 // Output the eigenvalue obtained
    System.out.println("The eigenvalue is: " + l);
  }

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

// Method to provide the function for the root search.

  public static double f(double l) {
    double u[] = new double[n+1];
    double p[] = new double[n+1];
    double q[] = new double[n+1];
    double s[] = new double[n+1];
    double p1[] = new double[n+1];
    double h = 1.0/n;
    double u0 = 0;
    double u1 = h;

    for (int i=0; i<=n; ++i){
      double x = h*i;
      p[i] = 1-x*x;
      p1[i] = -2*x;
      q[i]  = l*(l+1);
      s[i]  = 0;
    }
    u = sturmLiouville(h, u0, u1, p, p1, q, s);
    return u[n]-1;
  }

// Method to integrate the Sturm-Liouville problem.

  public static double[] sturmLiouville(double h,
    double u0, double u1, double p[], double p1[],
    double q[], double s[]) {
    int n = p.length-1;
    double u[] = new double[n+1];
    double h2 =  h*h;
    u[0] = u0;
    u[1] = u1;
    for (int i=1; i<n; ++i){
      double c2 = 2*p[i]+h*p1[i];
      double c1 = 4*p[i]-2*h2*q[i];
      double c0 = 2*p[i]-h*p1[i];
      double d  = 2*h2*s[i];
      u[i+1] = (c1*u[i]-c0*u[i-1]+d)/c2;
    }
    return u;
  }
}
```

The eigenvalue obtained from the above program is 1.000 000 000 01, which contains an error on the order of 10^{-11}, in comparison with the exact result of $l = 1$. The error is vanishingly small considering the possible rounding error and the inaccuracy in the algorithm. This is accidental of course. The expected error is on the order of 10^{-6}, given by the tolerance set in the program.

Note that the procedure adopted in the above example is quite general and it is the shooting method for the eigenvalue problem. For equations other than the Sturm–Liouville problem, we can follow exactly the same steps.

If we want to have higher accuracy in the algorithm for the Sturm–Liouville problem, we can differentiate Eq. (4.68) twice. Then we have

$$pu^{(4)} + 2p'u^{(3)} = s'' - 3p''u'' - p^{(3)}u' - p'u^{(3)} - q''u - 2q'u' - qu'', \tag{4.74}$$

where $u^{(3)}$ on the right-hand side can be replaced with

$$u^{(3)} = \frac{1}{p}(s' - p''u' - 2p'u'' - q'u - qu'), \tag{4.75}$$

which is the result of taking the first-order derivative of Eq. (4.68). If we combine Eqs. (4.71), (4.74), and (4.75), we obtain a better algorithm:

$$c_{i+1}u_{i+1} + c_{i-1}u_{i-1} = c_i u_i + d_i + O(h^6), \tag{4.76}$$

where c_{i+1}, c_{i-1}, c_i, and d_i are given by

$$\begin{aligned}
c_{i+1} &= 24p_i + 12hp_i' + 2h^2q_i + 6h^2p_i'' - 4h^2(p_i')^2/p_i \\
&\quad + h^3p_i^{(3)} + 2h^3q_i' - h^3p_i'q_i/p_i - h^3p_i'p_i''/p_i, && (4.77)\\
c_{i-1} &= 24p_i - 12hp_i' + 2h^2q_i + 6h^2p_i'' - 4h^2(p_i')^2/p_i \\
&\quad - h^3p_i^{(3)} - 2h^3q_i' + h^3p_i'q_i/p_i + h^3p_i'p_i''/p_i, && (4.78)\\
c_i &= 48p_i - 20h^2q_i - 8h^2(p_i')^2/p_i + 12h^2p'' \\
&\quad + 2h^4p_i'q_i'/p_i - 2h^4q_i'', && (4.79)\\
d_i &= 24h^2s_i 2h^4s_i'' - 2h^4p_i's_i'/p_i, && (4.80)
\end{aligned}$$

which can be evaluated easily if $p(x)$, $q(x)$, and $s(x)$ are explicitly given. When some of the derivatives needed are not easily obtained analytically, we can evaluate them numerically. In order to maintain the high accuracy of the algorithm, we need to use compatible numerical formulas.

For the special case with $p(x) = 1$, the above coefficients reduce to much simpler forms. Without sacrificing the apparently high accuracy of the algorithm, we can apply the three-point formulas to the first- and second-order derivatives

of $q(x)$ and $s(x)$. Then we have

$$c_{i+1} = 1 + \frac{h^2}{24}(q_{i+1} + 2q_i - q_{i-1}), \tag{4.81}$$

$$c_{i-1} = 1 + \frac{h^2}{24}(q_{i-1} + 2q_i - q_{i+1}), \tag{4.82}$$

$$c_i = 2 - \frac{5h^2}{60}(q_{i+1} + 8q_i + q_{i-1}), \tag{4.83}$$

$$d_i = \frac{h^2}{12}(s_{i+1} + 10s_i + s_{i-1}), \tag{4.84}$$

which are slightly different from the commonly known Numerov algorithm, which is an extremely accurate scheme for linear differential equations without the first-order derivative term, that is, Eq. (4.62) with $d(x) = 0$ or Eq. (4.68) with $p(x) = 1$. Many equations in physics have this form, for example, the Poisson equation with spherical symmetry or the one-dimensional Schrödinger equation.

The Numerov algorithm is derived from Eq. (4.70) after applying the three-point formula to the second-order derivative and the fourth-order derivative given in the form of a second-order derivative from Eq. (4.68),

$$u^{(4)}(x) = \frac{d^2}{dx^2}[-q(x)u(x) + s(x)]. \tag{4.85}$$

If we apply the three-point formula to the second-order derivative on the right-hand side of the above equation, we obtain

$$u^{(4)}(x) = \frac{(s_{i+1} - q_{i+1}u_{i+1}) - 2(s_i - q_iu_i) + (s_{i-1} - q_{i-1}u_{i-1})}{h^2}. \tag{4.86}$$

Combining the above equation with $\Delta_2 = (u_{i+1} - 2u_i + u_{i-1})/h^2 = u_i'' + h^2u^{(4)}/12$ and $u_i'' = s_i - q_iu_i$, we obtain the Numerov algorithm in the form of Eq. (4.76) with the corresponding coefficients given as

$$c_{i+1} = 1 + \frac{h^2}{12}q_{i+1}, \tag{4.87}$$

$$c_{i-1} = 1 + \frac{h^2}{12}q_{i-1}, \tag{4.88}$$

$$c_i = 2 - \frac{5h^2}{6}q_i, \tag{4.89}$$

$$d_i = \frac{h^2}{12}(s_{i+1} + 10s_i + s_{i-1}). \tag{4.90}$$

Note that even though the apparent local accuracy of the Numerov algorithm and the algorithm we derived earlier in this section for the Sturm–Liouville problem is $O(h^6)$, the actual global accuracy of the algorithm is only $O(h^4)$ because of the repeated use of the three-point formulas. For more discussion on this issue, see Simos (1993). The algorithms discussed here can be applied to initial-value problems as well as to boundary-value or eigenvalue problems. The Numerov algorithm and the algorithm for the Sturm–Liouville problem usually

have lower accuracy than the fourth-order Runge–Kutta algorithm when applied to the same problem, and this is different from what the apparent local accuracies imply. For more numerical examples of the Sturm–Liouville problem and the Numerov algorithm in the related problems, see Pryce (1993) and Onodera (1994).

4.9 The one-dimensional Schrödinger equation

The solutions associated with the one-dimensional Schrödinger equation are of importance in understanding quantum mechanics and quantum processes. For example, the energy levels and transport properties of electrons in nanostructures such as quantum wells, dots, and wires are crucial in the development of the next generation of electronic devices. In this section, we will apply the numerical methods that we have introduced so far to solve both the eigenvalue and transport problems defined through the one-dimensional Schrödinger equation

$$-\frac{\hbar^2}{2m}\frac{d^2\phi(x)}{dx^2} + V(x)\phi(x) = \varepsilon\phi(x), \tag{4.91}$$

where m is the mass of the particle, $\hbar$ is the Planck constant, ε is the energy level, $\phi(x)$ is the wavefunction, and $V(x)$ is the external potential. We can rewrite the Schrödinger equation as

$$\phi''(x) + \frac{2m}{\hbar^2}\left[\varepsilon - V(x)\right]\phi(x) = 0, \tag{4.92}$$

which is in the same form as the Sturm–Liouville problem with $p(x) = 1, q(x) = 2m\left[\varepsilon - V(x)\right]/\hbar^2$, and $s(x) = 0$.

The eigenvalue problem

For the eigenvalue problem, the particle is confined by the potential well $V(x)$, so that $\phi(x) \to 0$ with $|x| \to \infty$. A sketch of a typical $V(x)$ is shown in Fig. 4.7. In order to solve this eigenvalue problem, we can integrate the equation with the Numerov algorithm from left to right or from right to left of the potential region. Because the wavefunction goes to zero as $|x| \to \infty$, the integration from one side to another requires integrating from an exponentially increasing region to an oscillatory region and then into an exponentially decreasing region.

The error accumulated will become significant if we integrate the solution from the oscillatory region into the exponentially decreasing region. This is because an exponentially increasing solution is also a possible solution of the equation and can easily enter the numerical integration to destroy the accuracy of the algorithm. The rule of thumb is to avoid integrating into the exponential regions, that is, to carry out the solutions from both sides and then match them in the well region. Usually the matching is done at one of the turning points, where the energy is equal to the potential energy, such as x_l and x_r in Fig. 4.7. The

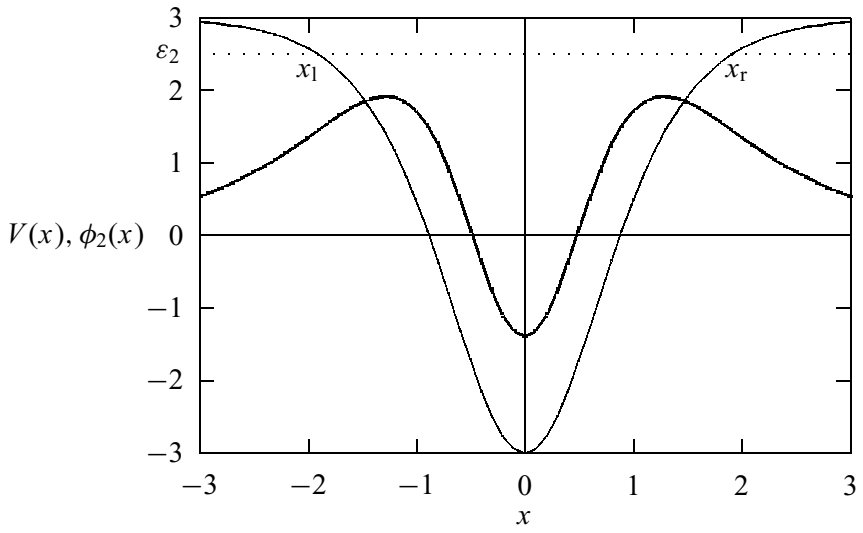

Fig. 4.7 The eigenvalue problem of the one-dimensional Schrödinger equation. Here a potential well $V(x)$ (solid thin line) and the corresponding eigenvalue ε_2 (dotted line) and eigenfunction $\phi_2(x)$ (solid thick line on a relative scale) for the second excited state are illustrated. The turning points x_l and x_r are also indicated.

so-called matching here is to adjust the trial eigenvalue until the solution integrated from the right, $\phi_\mathrm{r}(x)$, and the solution integrated from the left, $\phi_\mathrm{l}(x)$, satisfy the continuity conditions at one of the turning points. If we choose the right turning point as the matching point, the continuity conditions are

$$\phi_\mathrm{l}(x_\mathrm{r}) = \phi_\mathrm{r}(x_\mathrm{r}), \tag{4.93}$$

$$\phi_\mathrm{l}'(x_\mathrm{r}) = \phi_\mathrm{r}'(x_\mathrm{r}). \tag{4.94}$$

If we combine these two conditions, we have

$$\frac{\phi_\mathrm{l}'(x_\mathrm{r})}{\phi_\mathrm{l}(x_\mathrm{r})} = \frac{\phi_\mathrm{r}'(x_\mathrm{r})}{\phi_\mathrm{r}(x_\mathrm{r})}. \tag{4.95}$$

If we use the three-point formula for the first-order derivatives in the above equation, we have

$$\begin{aligned} f(\varepsilon) &= \frac{[\phi_\mathrm{l}(x_\mathrm{r}+h) - \phi_\mathrm{l}(x_\mathrm{r}-h)] - [\phi_\mathrm{r}(x_\mathrm{r}+h) - \phi_\mathrm{r}(x_\mathrm{r}-h)]}{2h\phi(x_\mathrm{r})} \\ &= 0, \end{aligned} \tag{4.96}$$

which can be ensured by a root search scheme. Note that $f(\varepsilon)$ is a function of only ε because $\phi_\mathrm{l}(x_\mathrm{r}) = \phi_\mathrm{r}(x_\mathrm{r}) = \phi(x_\mathrm{r})$ can be used to rescale the wavefunctions.

We now outline the numerical procedure for solving the eigenvalue problem of the one-dimensional Schrödinger equation:

(1) Choose the region of the numerical solution. This region should be large enough compared with the effective region of the potential to have a negligible effect on the solution.

(2) Provide a reasonable guess for the lowest eigenvalue ε_0. This can be found approximately from the analytical result of the case with an infinite well and the same range of well width.

(3) Integrate the equation for $\phi_{\rm l}(x)$ from the left to the point $x_{\rm r}+h$ and the one for $\phi_{\rm r}(x)$ from the right to $x_{\rm r}-h$. We can choose zero to be the value of the first points of $\phi_{\rm l}(x)$ and $\phi_{\rm r}(x)$, and a small quantity to be the value of the second points of $\phi_{\rm l}(x)$ and $\phi_{\rm r}(x)$, to start the integration, for example, with the Numerov algorithm. Before matching the solutions, rescale one of them to ensure that $\phi_{\rm l}(x_{\rm r})=\phi_{\rm r}(x_{\rm r})$. For example, we can multiply $\phi_{\rm l}(x)$ by $\phi_{\rm r}(x_{\rm r})/\phi_{\rm l}(x_{\rm r})$ up to $x=x_{\rm r}+h$. This rescaling also ensures that the solutions have the correct nodal structure, that is, changing the sign of $\phi_{\rm l}(x)$ if it is incorrect.
(4) Evaluate $f(\varepsilon_0)=[\phi_{\rm r}(x_{\rm r}-h)-\phi_{\rm r}(x_{\rm r}+h)-\phi_{\rm l}(x_{\rm r}-h)+\phi_{\rm l}(x_{\rm r}+h)]/2h\phi_{\rm r}(x_{\rm r})$.
(5) Apply a root search method to obtain ε_0 from $f(\varepsilon_0)=0$ within a given tolerance.
(6) Carry out the above steps for the next eigenvalue. We can start the search with a slightly higher value than the last eigenvalue. We need to make sure that no eigenstate is missed. This can easily be done by counting the nodes in the solution; the nth state has a total number of n nonboundary nodes, with $n=0$ for the ground state. A node is where $\phi(x)=0$. This also provides a way of pinpointing a specific eigenstate.

Now let us look at an actual example with a particle bound in a potential well

$$V(x)=\frac{\hbar^2}{2m}\alpha^2\lambda(\lambda-1)\left[\frac{1}{2}-\frac{1}{\cosh^2(\alpha x)}\right], \tag{4.97}$$

where both α and λ are given parameters. The Schrödinger equation with this potential can be solved exactly with the eigenvalues

$$\varepsilon_n=\frac{\hbar^2}{2m}\alpha^2\left[\frac{\lambda(\lambda-1)}{2}-(\lambda-1-n)^2\right], \tag{4.98}$$

for $n=0,1,2,\ldots$. We have solved this problem numerically in the region $[-10,10]$ with 501 points uniformly spaced. The potential well, eigenvalue, and eigenfunction shown in Fig. 4.7 are from this problem with $\alpha=1$, $\lambda=4$, and $n=2$. We have also used $\hbar=m=1$ in the numerical solution for convenience. The program below implements this scheme.

```
// An example of solving the eigenvalue problem of the
// one-dimensional Schroedinger equation via the secant
// and Numerov methods.

import java.lang.*;
import java.io.*;
public class Schroedinger {
  static final int nx = 500, m = 10, ni = 10;
  static final double x1 = -10, x2 = 10, h = (x2-x1)/nx;
  static int nr, nl;
  static double ul[] = new double[nx+1];
  static double ur[] = new double[nx+1];
  static double ql[] = new double[nx+1];
  static double qr[] = new double[nx+1];
```

```
  static double s[] = new double[nx+1];
  static double u[] = new double[nx+1];
  public static void main(String argv[]) throws
    FileNotFoundException {
    double del = 1e-6, e = 2.4, de = 0.1;

 // Find the eigenvalue via the secant search
    e = secant(ni, del, e, de);

 // Output the wavefunction to a file
    PrintWriter w = new PrintWriter
      (new FileOutputStream("wave.data"), true);
    double x = x1;
    double mh = m*h;
    for (int i=0; i<=nx; i+=m) {
      w.println( x + " " + u[i]);
      x += mh;
    }

 // Output the eigenvalue obtained
    System.out.println("The eigenvalue: " + e);
  }

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

// Method to provide the function for the root search.

  public static double f(double x) {
    wave(x);
    double f0 = ur[nr-1]+ul[nl-1]-ur[nr-3]-ul[nl-3];
    return f0/(2*h*ur[nr-2]);
  }

// Method to calculate the wavefunction.

  public static void wave(double energy) {
    double y[] = new double [nx+1];
    double u0 = 0, u1 = 0.01;

 // Set up function q(x) in the equation
    for (int i=0; i<=nx; ++i) {
      double x = x1+i*h;
      ql[i] = 2*(energy-v(x));
      qr[nx-i] = ql[i];
    }

 // Find the matching point at the right turning point
    int im = 0;
    for (int i=0; i<nx; ++i)
      if (((ql[i]*ql[i+1])<0) && (ql[i]>0)) im = i;

 // Carry out the Numerov integrations
    nl = im+2;
    nr = nx-im+2;
    ul = numerov(nl, h, u0, u1, ql, s);
    ur = numerov(nr, h, u0, u1, qr, s);

 // Find the wavefunction on the left
    double ratio = ur[nr-2]/ul[im];
    for (int i=0; i<=im; ++i) {
```

```
      u[i] = ratio*ul[i];
      y[i] = u[i]*u[i];
    }

 // Find the wavefunction on the right
    for (int i=0; i<nr-1; ++i) {
      u[i+im] = ur[nr-i-2];
      y[i+im] = u[i+im]*u[i+im];
    }

 // Normalize the wavefunction
    double sum = simpson(y, h);
    sum = Math.sqrt(sum);
    for (int i=0; i<=nx; ++i) u[i] /= sum;
  }

// Method to perform the Numerov integration.

  public static double[] numerov(int m, double h,
    double u0, double u1, double q[], double s[]) {
    double u[] = new double[m];
    u[0] = u0;
    u[1] = u1;
    double g = h*h/12;
    for (int i=1; i<m-1; ++i) {
      double c0 = 1+g*q[i-1];
      double c1 = 2-10*g*q[i];
      double c2 = 1+g*q[i+1];
      double d  = g*(s[i+1]+s[i-1]+10*s[i]);
      u[i+1] = (c1*u[i]-c0*u[i-1]+d)/c2;
    }
    return u;
  }

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

// Method to provide the given potential in the problem.

  public static double v(double x) {
    double alpha = 1, lambda = 4;
    return alpha*alpha*lambda*(lambda-1)
             *(0.5-1/Math.pow(cosh(alpha*x),2))/2;
  }

// Method to provide the hyperbolic cosine needed.

  public static double cosh(double x) {
    return (Math.exp(x)+Math.exp(-x))/2;
  }
}
```

Running the above program gives $\varepsilon_2 = 2.499\,999$, which is what we expect, comparing with the exact result $\varepsilon_2 = 2.5$, under the given tolerance of 1.0×10^{-6}. Note that we have used the Numerov algorithm with the set of coefficients provided in Eqs. (4.87)–(4.90). We can use the set of coefficients in Eqs. (4.81)–(4.84) and the result will remain the same within the accuracy of the algorithm.

Quantum scattering

Now let us turn to the problem of unbound states, that is, the scattering problem. Assume that the potential is nonzero in the region of $x \in [0, a]$ and that the incident particle comes from the left. We can write the general solution of the Schrödinger equation outside the potential region as

$$\phi(x) = \begin{cases} \phi_1(x) = e^{ikx} + Ae^{-ikx} & \text{for } x < 0, \\ \phi_3(x) = Be^{ik(x-a)} & \text{for } x > a, \end{cases} \tag{4.99}$$

where A and B are the parameters to be determined and k can be found from

$$\varepsilon = \frac{\hbar^2 k^2}{2m}, \tag{4.100}$$

where ε is the energy of the incident particle. The solution $\phi(x) = \phi_2(x)$ in the region of $x \in [0, a]$ can be obtained numerically. During the process of solving $\phi_2(x)$, we will also obtain A and B, which are necessary for calculating the reflectivity $R = |A|^2$ and transmissivity $T = |B|^2$. Note that $T + R = 1$. The boundary conditions at $x = 0$ and $x = a$ are

$$\phi_1(0) = \phi_2(0), \tag{4.101}$$
$$\phi_2(a) = \phi_3(a), \tag{4.102}$$
$$\phi_1'(0) = \phi_2'(0), \tag{4.103}$$
$$\phi_2'(a) = \phi_3'(a), \tag{4.104}$$

which give

$$\phi_2(0) = 1 + A, \tag{4.105}$$
$$\phi_2(a) = B, \tag{4.106}$$
$$\phi_2'(0) = ik(1 - A), \tag{4.107}$$
$$\phi_2'(a) = ikB. \tag{4.108}$$

Note that the wavefunction is now a complex function, as are the parameters A and B. We outline here a combined numerical scheme that utilizes either the Numerov or the Runge–Kutta method to integrate the equation and a minimization scheme to adjust the solution to the desired accuracy. We first outline the scheme through the Numerov algorithm:

(1) For a given particle energy $\varepsilon = \hbar^2 k^2/2m$, guess a complex parameter $A = A_\mathrm{r} + iA_\mathrm{i}$. We can use the analytical results for a square potential that has the same range and average strength of the given potential as the initial guess. Because the convergence is very fast, the initial guess is not very important.
(2) Perform the Numerov integration of the Schrödinger equation

$$\phi_2''(x) + \left[k^2 - \frac{2m}{\hbar^2} V(x)\right]\phi_2(x) = 0 \tag{4.109}$$

from $x = 0$ to $x = a$ with the second point given by the Taylor expansion of $\phi_2(x)$ around $x = 0$ to the second order,

$$\phi_2(h) = \phi_2(0) + h\phi_2'(0) + \frac{h^2}{2}\phi_2''(0) + O(h^3), \tag{4.110}$$

where $\phi_2(0) = \phi_1(0) = 1 + A$, $\phi_2'(0) = \phi_1'(0) = ik(1 - A)$, and $\phi_2''(0) = [2mV(0^+)/\hbar^2 - k^2]\phi_2(0) = [2mV(0^+)/\hbar^2 - k^2](1 + A)$. Note that we have used $V(0^+) = \lim_{\delta\to 0} V(\delta)$ for $\delta > 0$. The second-order derivative here is obtained from the Schrödinger equation. Truncation at the first order would also work, but with a little less accuracy.

(3) We can then obtain the approximation for B after the first integration to the point $x = a$ with

$$B = \phi_2(a). \tag{4.111}$$

(4) Using this B value, we can integrate the equation from $x = a$ back to $x = 0$ with the same Numerov algorithm with the second point given by the Taylor expansion of $\phi_2(x)$ around $x = a$ as

$$\phi_2(a - h) = \phi_2(a) - h\phi_2'(a) + \frac{h^2}{2}\phi_2''(a) + O(h^3), \tag{4.112}$$

where $\phi_2(a) = \phi_3(a) = B$ and $\phi_2'(a) = \phi_3'(a) = ikB$ from the continuity conditions. But the second-order derivative is obtained from the equation, and we have $\phi_2''(a) = [2mV(a^-)/\hbar^2 - k^2]\phi_2(a) = [2mV(a^-)/\hbar^2 - k^2]B$. Note that we have used $V(a^-) = \lim_{\delta\to 0} V(a - \delta)$ for $\delta > 0$.

(5) From the backward integration, we can obtain a new $A^{\text{new}} = A_{\text{r}}^{\text{new}} + iA_{\text{i}}^{\text{new}}$ from $\phi_2^{\text{new}}(0) = 1 + A^{\text{new}}$. We can then construct a real function $g(A_{\text{r}}, A_{\text{i}}) = (A_{\text{r}} - A_{\text{r}}^{\text{new}})^2 + (A_{\text{i}} - A_{\text{i}}^{\text{new}})^2$. Note that $A_{\text{r}}^{\text{new}}$ and $A_{\text{i}}^{\text{new}}$ are both the implicit functions of $(A_{\text{r}}, A_{\text{i}})$.

(6) Now the problem becomes an optimization problem of minimizing $g(A_{\text{r}}, A_{\text{i}})$ as A_{r} and A_{i} vary. We can use the steepest-descent scheme introduced in Chapter 3 or other optimization schemes given in Chapter 5.

Now let us illustrate the scheme outlined above with an actual example. Assume that we are interested in the quantum scattering of a double-barrier potential

$$V(x) = \begin{cases} V_0 & \text{if } 0 \le x \le x_1 \quad \text{or} \quad x_2 \le x \le a, \\ 0 & \text{elsewhere.} \end{cases} \tag{4.113}$$

This is a very interesting problem, because it is one of the basic elements

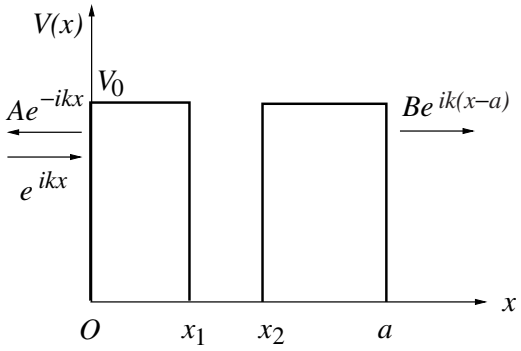

Fig. 4.8 The double-barrier structure of the example. For a system made of GaAs and $Ga_{1-x}Al_xAs$ layers, the barrier regions are formed with $Ga_{1-x}Al_xAs$.

conceived for the next generation of electronic devices. A sketch of the system is given in Fig. 4.8. The problem is solved with the Numerov algorithm and an optimization scheme. We use the steepest-descent method introduced in Chapter 3 in the following implementation.

```
// An example of studying quantum scattering in one
// dimension through the Numerov and steepest-descent
// methods.

import java.lang.*;
import java.io.*;
public class Scattering {
  static final int nx = 100;
  static final double hartree = 0.0116124;
  static final double bohr = 98.964, a = 125/bohr;
  static double e;

  public static void main(String argv[]) throws
    IOException {
    double x[] = new double[2];

// Read in the particle energy
    System.out.println("Enter particle energy: ");
    InputStreamReader c
      = new InputStreamReader(System.in);
    BufferedReader b = new BufferedReader(c);
    String energy = b.readLine();
    e = Double.valueOf(energy).doubleValue();
    e /= hartree;
    x[0] =  0.1;
    x[1] = -0.1;
    double d = 0.1, del = 1e-6;
    steepestDescent(x, d, del);
    double r = x[0]*x[0]+x[1]*x[1];
    double t = 1-r;
    System.out.println("The reflectivity: " + r);
    System.out.println("The transmissivity: " + t);
  }

  public static void steepestDescent(double x[],
    double a, double del) {...}
```

```
// Method to provide function f=gradient g(x).
  public static double[] f(double x[], double h) {...}

// Method to provide function g(x).

  public static double g(double x[]) {
    double h = a/nx;
    double ar = x[0];
    double ai = x[1];
    double ur[] = new double[nx+1];
    double ui[] = new double[nx+1];
    double qf[] = new double[nx+1];
    double qb[] = new double[nx+1];
    double s[] = new double[nx+1];

    for (int i=0; i<=nx; ++i) {
      double xi = h*i;
      s[i] = 0;
      qf[i] = 2*(e-v(xi));
      qb[nx-i] = qf[i];
    }

 // Perform forward integration
    double delta = 1e-6;
    double k = Math.sqrt(2*e);
    double ur0 = 1+ar;
    double ur1 = ur0+k*ai*h
      +h*h*(v(delta)-e)*ur0;
    double ui0 = ai;
    double ui1 = ui0+k*(1-ar)*h
      +h*h*(v(delta)-e)*ui0;
    ur = numerov(nx+1, h, ur0, ur1, qf, s);
    ui = numerov(nx+1, h, ui0, ui1, qf, s);

 // Perform backward integration
    ur0 = ur[nx];
    ur1 = ur0+k*ui[nx]*h
      +h*h*(v(a-delta)-e)*ur0;
    ui0 = ui[nx];
    ui1 = ui0-k*ur[nx]*h
      +h*h*(v(a-delta)-e)*ui0;
    ur = numerov(nx+1, h, ur0, ur1, qb, s);
    ui = numerov(nx+1, h, ui0, ui1, qb, s);

 // Return the function value |a-a_new|*|a-a_new|
    return (1+ar-ur[nx])*(1+ar-ur[nx])
      +(ai-ui[nx])*(ai-ui[nx]);
  }

// Method to provide the potential V(x).

  public static double v(double x) {
    double v0 = 0.3/hartree;
    double x1 = 25/bohr;
    double x2 = 75/bohr;
    if (((x<x1)&&(x>0)) || ((x<a)&&(x>x2)))
      return v0;
    else return 0;
  }
```

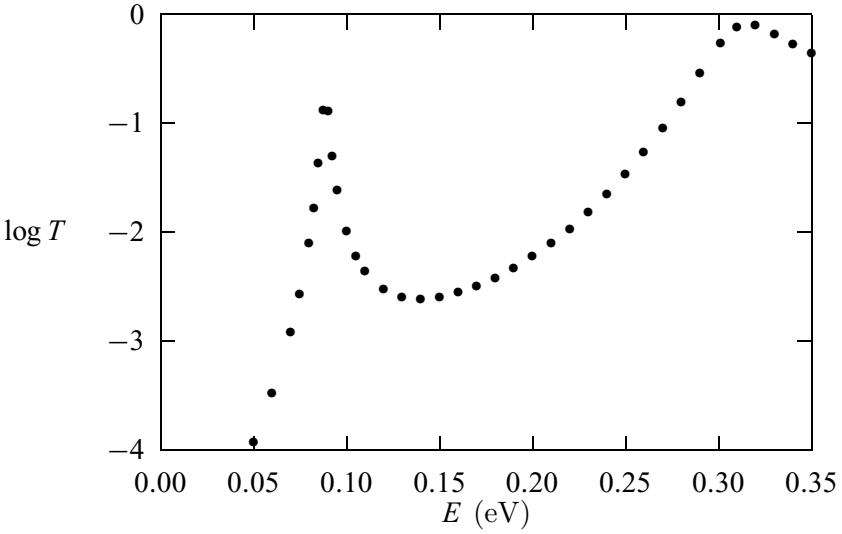

Fig. 4.9 The energy dependence of the transmissivity for a double-barrier potential with a barrier height of 0.3 eV, barrier widths of 25 Å and 50 Å, and the width of the well between the barriers of 50 Å. The transmissivity is plotted on a logarithmic scale.

```
// Method to perform the Numerov integration.
  public static double[] numerov(int m, double h,
    double u0, double u1, double q[], double s[])
    {...}
}
```

We have used the parameters associated with GaAs in the above program with the effective mass of the electron $m = 0.067m_e$ and electric permittivity $\epsilon = 12.53\epsilon_0$, where m_e is the mass of a free electron and ϵ_0 is the electric permittivity of vacuum. Under this choice of effective mass and permittivity, we have given the energy in the unit of the effective Hartree (about 11.6 meV) and length in the unit of the effective Bohr radius (about 99.0 Å).

In Fig. 4.9, we plot the transmissivity of the electron obtained for $V_0 = 0.3$ eV, $x_1 = 25$ Å, $x_2 = 75$ Å, and $a = 125$ Å. Note that there is a significant increase in the transmissivity around a resonance energy $\varepsilon \simeq 0.09$ eV, which is a virtual energy level. The second peak appears at an energy slightly above the barriers. Study of the symmetric barrier case (Pang, 1995) shows that the transmissivity can reach at least 0.999 997 at the resonance energy $\varepsilon \simeq 0.089\,535$ eV, for the case with $x_1 = 50$ Å, $x_2 = 100$ Å, and $a = 150$ Å.

We can also use the Runge–Kutta algorithm to integrate the equation if we choose $y_1(x) = \phi_2(x)$ and $y_2(x) = \phi_2'(x)$. Using the initial conditions for $\phi_2(x)$ and $\phi_2'(x)$ at $x = 0$, we integrate the equation to $x = a$. From $y_1(a)$ and $y_2(a)$, we obtain two different values of B,

$$B_1 = y_1(a), \tag{4.114}$$

$$B_2 = -\frac{i}{k} y_2(a), \tag{4.115}$$

which are both implicit functions of the initial guess of $A = A_r + iA_i$. This means that we can construct an implicit function $g(A_r, A_i) = |B_1 - B_2|^2$ and

then optimize it. Note that both B_1 and B_2 are complex, as are the functions $y_1(x)$ and $y_2(x)$. The Runge–Kutta algorithm in this case is much more accurate, because no approximation for the second point is needed. For more details on the application of the Runge–Kutta algorithm and the optimization method in the scattering problem, see Pang (1995).

The procedure can be simplified in the simple potential case, as suggested by R. Zimmermann (personal communication), if we realize that the Schrödinger equation in question is a linear equation. We can take $B = 1$ and then integrate the equation from $x = a$ back to $x = -h$ with either the Numerov or the Runge–Kutta scheme. The solution at $x = 0$ and $x = -h$ satisfies

$$\phi(x) = A_1 e^{ikx} + A_2 e^{-ikx}, \tag{4.116}$$

and we can solve for A_1 and A_2 with the numerical results of $\phi(0)$ and $\phi(-h)$. The reflectivity and transmissivity are then given by $R = |A_2/A_1|^2$ and $T = 1/|A_1|^2$, respectively. Note that it is not now necessary to have the minimization in the scheme. However, for a more general potential, for example, a nonlinear potential, a combination of an integration scheme and an optimization scheme becomes necessary.

Exercises

4.1 Consider two charged particles of masses m_1 and m_2, and charges q_1 and q_2, respectively. They are moving in the xy plane under the influence of a constant electric field $\mathbf{E} = E_0\hat{\mathbf{x}}$ and a constant magnetic induction $\mathbf{B} = B_0\hat{\mathbf{z}}$, where $\hat{\mathbf{x}}$ and $\hat{\mathbf{z}}$ are unit vectors along the positive x and z axes, respectively. Implement the two-point predictor–corrector method to study the system. Is there a parameter region in which the system is chaotic? What happens if each particle encounters a random force that is in the Gaussian distribution?

4.2 Derive the fourth-order Runge–Kutta algorithm for solving the differential equation

$$\frac{dy}{dt} = g(y, t)$$

with a given initial condition. Discuss the options in the selection of the parameters involved.

4.3 Construct a subprogram that solves the differential equation set

$$\frac{d\mathbf{y}}{dt} = \mathbf{g}(\mathbf{y}, t)$$

with the fourth-order Runge–Kutta method with different parameters from those given in the text. Use the total number of components in $\mathbf{y}$ and the initial condition $\mathbf{y}(0) = \mathbf{y}_0$ as the input to the subprogram. Test the fitness

of the parameters by comparing the numerical result from the subprogram and the known exact result for the motion of Earth.

4.4 Study the driven pendulum under damping numerically and plot the bifurcation diagram (ω–b plot at a fixed θ) with $q = 1/2$, $\omega_0 = 2/3$, and $b \in [1, 1.5]$.

4.5 Modify the example program for the driven pendulum under damping to study the cases with the driving force changed to a square wave with $f_d(t) = f_0$ for $0 < t < T_0/2$ and $f_d(t) = -f_0$ for $T_0/2 < t < T_0$, and to a triangular wave with $f_d(t) = f_0(2t/T_0 - 1)$ for $0 < t < T_0$, where T_0 is the period of the driving force that repeats in other periods. Is there a parameter region in which the system is chaotic?

4.6 The Duffing model is given by

$$\frac{d^2x}{dt^2} + g\frac{dx}{dt} + x^3 = b\cos t.$$

Write a program to solve the Duffing model in a different parameter region of (g, b). Discuss the behavior of the system from the phase diagram of (x, v), where $v = dx/dt$. Is there a parameter region in which the system is chaotic?

4.7 The Henón–Heiles model is used to describe stellar orbits and is given by a two-dimensional potential

$$V(x, y) = \frac{m\omega^2}{2}(x^2 + y^2) + \lambda\left(x^2 y - \frac{y^3}{3}\right),$$

where m is the mass of the star, and ω and λ are two additional model parameters. Derive the differential equation set that describes the motion of the star under the Henón–Heiles model and solve the equation set numerically. In what parameter region does the orbit become chaotic?

4.8 The Lorenz model is used to study climate change and is given by

$$\begin{aligned}\frac{dy_1}{dt} &= a(y_2 - y_1),\\ \frac{dy_2}{dt} &= (b - y_3)y_1 - y_2,\\ \frac{dy_3}{dt} &= y_1 y_2 - cy_3,\end{aligned}$$

where a, b, and c are positive parameters in the model. Solve this model numerically and find the parameter region in which the system is chaotic.

4.9 Consider three objects in the solar system, the Sun, Earth, and Mars. Find the modification of the next period of Earth due to the appearance of Mars starting at the beginning of January 1, 2006.

4.10 Study the dynamics of the two electrons in a classical helium atom. Explore the properties of the system under different initial conditions. Can the system ever be chaotic?

4.11 Apply the Numerov algorithm to solve

$$u''(x) = -4\pi^2 u(x),$$

with $u(0) = 1$ and $u'(0) = 0$. Discuss the accuracy of the result by comparing it with the solution obtained via the fourth-order Runge–Kutta algorithm and with the exact result.

4.12 Apply the shooting method to solve the eigenvalue problem

$$u''(x) = \lambda u(x),$$

with $u(0) = u(1) = 0$. Discuss the accuracy of the result by comparing it with the exact result.

4.13 Develop a program that applies the fourth-order Runge–Kutta and bisection methods to solve the eigenvalue problem of the stationary one-dimensional Schrödinger equation. Find the two lowest eigenvalues and eigenfunctions for an electron in the potential well

$$V(x) = \begin{cases} V_0 \dfrac{x}{x_0} & \text{if } 0 < x < x_0, \\ V_0 & \text{elsewhere.} \end{cases}$$

Atomic units (a.u.), that is, $m_e = e = \hbar = c = 1$, $x_0 = 5$ a.u., and $V_0 = 50$ a.u. can be used.

4.14 Apply the fourth-order Runge–Kutta algorithm to solve the quantum scattering problem in one dimension. The optimization can be achieved with the bisection method through varying A_r and A_i, respectively, within the region $[-1, 1]$. Test the program with the double-barrier potential of Eq. (4.113).

4.15 Find the angle dependence of the angular velocity and the center-of-mass velocity of a meterstick falling with one end on a horizontal plane. Assume that the meterstick is released from an initial angle of $\theta_0 \in [0, \pi/2]$ between the meterstick and the horizontal and that there is kinetic friction at the contact point, with the kinetic friction coefficient $0 \leq \mu_k \leq 0.9$.

4.16 Implement the full-accuracy algorithm for the Sturm–Liouville problem derived in the text in a subprogram. Test it by applying the algorithm to the spherical Bessel equation

$$(x^2 u')' + [x^2 - l(l+1)]u = 0,$$

where $l(l+1)$ are the eigenvalues. Use the known analytic solutions to set up the first two points. Evaluate l numerically and compare it with the exact result. Is the apparent accuracy of $O(h^6)$ in the algorithm delivered?

4.17 Study the dynamics of a compact disk after it is set in rotation on a horizontal table with the rotational axis vertical and along its diameter. Establish a model and set up the relevant equation set that describes the motion of the disk, including the process of the disk falling onto the table. Write a

program that solves the equation set. Does the solution describe the actual experiment well?

4.18 The tippe top is a symmetric top that can spin on both ends. When the top is set in rotation on the end that is closer to the center of mass of the top, it will gradually slow down and flip over to spin on the end that is farther away from the center of mass. Establish a model and set up the relevant equation set that describes the motion of the tippe top, including the process of toppling over. Write a program that solves the equation set. Does the solution describe the actual experiment well?

4.19 The system of two coupled rotors is described by the following equation set:

$$\frac{d^2y_1}{dt^2} + \gamma_1 \frac{dy_1}{dt} + \epsilon \left(\frac{dy_1}{dt} - \frac{dy_2}{dt} \right) = f_1(y_1) + F_1(t),$$
$$\frac{d^2y_2}{dt^2} + \gamma_2 \frac{dy_2}{dt} + \epsilon \left(\frac{dy_2}{dt} - \frac{dy_1}{dt} \right) = f_2(y_2) + F_2(t),$$

where ϵ is the coupling constant, $f_{1,2}$ are periodic functions of period 2π, and $F_{1,2} = \alpha_{1,2} + \beta_{1,2} \sin(\omega_{1,2}t + \phi_{1,2})$. Write a program to study this system. Consider a special case with $\alpha_i = \phi_i = 0$, $\gamma_1 = \gamma_2$, $\omega_1 = \omega_2$, and $f_i(x) = f_0 \sin x$. Is there a parameter region in which the system is chaotic?