

IV

Model Types



17

Ordinary Differential Equations

Francisco Esquembre

Universidad de Murcia

Wolfgang Christian

Davidson College

17.1	Introduction	17-1
	The Simple Pendulum • Dynamics of Chemical Reactions • Predator–Prey Population Dynamics	
	Planetary Motion	
17.2	Numerical Solution	17-3
	Implementation Techniques	
17.3	Taylor Methods	17-5
17.4	Runge–Kutta Methods	17-7
17.5	Implementation	17-8
17.6	Adaptive Step	17-11
17.7	Implementation of Adaptive Step	17-12
17.8	Performance and Other Methods	17-15
17.9	State Events	17-19
17.10	The OSP Library	17-20

17.1 Introduction

Right from the invention of calculus, ordinary differential equations (ODEs) have been used to model continuous systems in all scientific and engineering disciplines. The idea is simple in principle and applies to physical situations in which information can be obtained about the continuous rate of change of the state variables of the system. The hope is that we will be able to ascertain the evolution of the system using this information and the knowledge of its initial state.

We first consider some examples from different disciplines.

17.1.1 The Simple Pendulum

A simple pendulum is a physical abstraction in which a point mass m oscillates in a vertical plane at the end of a rod of length L with negligible mass. The motion of a simple pendulum can be modeled by the ODE:

$$mL^2\ddot{\theta}(t) + b\dot{\theta}(t) + mgL \sin \theta(t) = \tau_e(t) \quad (17.1)$$

where $\theta(t)$ is the angular position of the rod of the pendulum with respect to the vertical, b the friction coefficient, g the acceleration due to gravity, and $\tau_e(t)$ a time-dependent external torque which drives the motion. Typically, the pendulum starts at time $t = 0$ from a given angle θ_0 with zero initial angular velocity.

17.1.2 Dynamics of Chemical Reactions

The Brusselator was proposed in 1968 by R. Lefever and the Nobel Prize winner I. Prigogine, as a model for an autocatalytic, oscillating chemical reaction. The mechanism for the reaction is given by



We are interested in the evolution of the intermediate products X and Y . If we assume that the concentration of the reactants A and B is kept constant, $a = [A]$ and $b = [B]$, and denote the concentrations of the products as $x = [X]$ and $y = [Y]$, the system is described by the differential equations:

$$\dot{x}(t) = k_1 a - k_2 b x(t) + k_3 x^2(t) y(t) - k_4 x(t) \quad (17.3a)$$

$$\dot{y}(t) = k_2 b x(t) - k_3 x^2(t) y(t) \quad (17.3b)$$

The initial values of x and y are zero. For appropriate values of the parameters, this system can exhibit oscillatory behavior.

17.1.3 Predator–Prey Population Dynamics

ODEs are frequently used in biology to model population dynamics. The famous Lotka–Volterra model describes the evolution of two species, one of which preys (feeds) on the other, using the ODEs:

$$\dot{x}_1(t) = ax_1(t) - bx_1(t)x_2(t) \quad (17.4a)$$

$$\dot{x}_2(t) = cx_1(t)x_2(t) - dx_2(t) \quad (17.4b)$$

where x_1 represents the number of prey and x_2 is the number of predators (in appropriate units so that they take continuous values in the interval $[0, 1]$), and a , b , c , and d are the parameters. The terms ax_1 and $-dx_2$ account for the reproduction rate of each species in the absence of interaction with the other and the nonlinear terms represent the effects of predation on the reduction of prey and the reproduction of the predators. This model was used in the mid-1920s to study how the intensity of fishing affected the different fish populations in the Mediterranean Sea.

17.1.4 Planetary Motion

Kepler discovered his three laws of planetary motion after a titanic analysis of years of astronomical observations by Tycho Brahe. Newton's inverse-square law of gravitation allows us to reformulate this motion in modern terms using the equations for a massless test particle about a particle of mass M located at the origin:

$$\ddot{x}(t) = -GM \frac{x(t)}{(x^2(t) + y^2(t))^{3/2}} \quad (17.5a)$$

$$\ddot{y}(t) = -GM \frac{y(t)}{(x^2(t) + y^2(t))^{3/2}} \quad (17.5b)$$

where M is the combined mass of the Sun and the planet, and G is the universal gravitational constant. The xy -coordinates for the massless test particle are the relative position of the planet with respect to the Sun in this two-body approximation of the solar system. The initial values for x and y and its derivatives are taken from direct astronomical observation.

The examples presented here are typical. Formulating the problem with the help of ODEs and stating the existence and uniqueness of a solution under reasonable regularity conditions are relatively easy tasks. But finding an analytic solution is often difficult and might be impossible. Although analytic solutions, such as in the case of linear ODEs, are very important theoretically, the vast majority of problems cannot be solved in this manner. In these cases, we must use numerical techniques to approximate the evolutionary behavior predicted by the model. This chapter introduces some of these techniques.

The numerical algorithms we will discuss can be implemented in many different ways, using almost any programming language. We have chosen Java to illustrate particular algorithmic implementations. Java is an object-oriented programming language that is designed to run on a virtual computer that can be implemented in any modern operating system. This promise of platform independence has become a reality, and we can now write programs that have attractive graphical user interfaces and support common tasks such as printing, disk access, and copy–paste data exchange with other applications.

Although these features can be implemented using the standard Java library, it usually requires much programming and a working knowledge of the Java application programmer interface (API). Learning the Java API is essential for software developers, but experienced programmers usually adopt or develop an add-on code library. To enable scientists and engineers to quickly begin writing their own programs, we have developed the Open Source Physics (OSP) library (Christian, 2007). This library enables users to quickly perform common visualization tasks such as creating a graph.

```
PlotFrame plot= new PlotFrame("f(t)", "t", "Sine Function");
for (int i = 0; i < 100; i++) {
    double t=i*2*Math.PI/100.0;
    plot.append(0,t,Math.sin(t));    // Add data to graph #0
}
```

You do not need to know much about Java to understand that these statements graph a sine function. The advantage of an object-oriented approach is that readers who are unfamiliar with or uninterested in Java may treat the code listings in this chapter as pseudocode while Java programmers can compile and run the examples. Source code for the pendulum, the chemical reaction, the predator–prey, and the planetary motion examples are available on the OSP website (http://www.opensourcephysics.org/CRC_examples). The website also has ready-to-run (executable) versions of these programs available for downloading.

An alternative approach to coding the examples in Java is to use a high-level modeling program. The *Easy Java Simulations* (Ejs) authoring tool uses the OSP library to provide a simple way to create your own simulations for continuous (and noncontinuous) systems. Ejs has a graphical drag-and-drop user interface that helps users build models and connect models to graphical elements to visualize the state of the system and its evolution. These simulations can then be used to explore the behavior of the system under different conditions. Examples of Ejs (including the examples in this chapter) can be found at http://fem.um.es/Ejs/CRC_examples.

17.2 Numerical Solution

The mathematical formulation most frequently used in modeling continuous systems is that of an *initial-value problem* for a first-order ordinary differential equation.¹ An initial-value problem is an expression of the form:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t), \quad \mathbf{x}(a) = \mathbf{x}_0 \quad (17.6)$$

¹ODEs can also appear in the form of *boundary value problems*, where the information about the state of the system is provided partly in one initial instant of time, and partly in one final instant of time (Press et al., 1992; Keller, 1992).

where t (which often represents the time) is called the independent variable and can take values in a given interval $[a, b]$ of the real line, $\mathbf{x}(t)$ represents a k -dimensional vector of real numbers that describes the state of the system at a given instant, and the vector function $\mathbf{f}$ conveys all the information that relates the rate of change of the state vector with the value of the independent variable and the state itself. The information provided at the initial instant of time, $\mathbf{x}(a) = \mathbf{x}_0$, is called the initial condition, and is necessary to completely determine a solution out of the many possible solutions.

ODEs in which the right-hand side has no explicit dependence on time, $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t))$, are called *autonomous*. Autonomous equations are of interest because they describe *dynamical systems* whose evolution depends only on their internal state independent of time. Nonautonomous equations contain an explicit time dependence, $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t)$. The simple pendulum in Section 17.1 is an example of a nonautonomous equation, while all the others describe dynamical systems.

Every nonautonomous ODE can be reformulated as a dynamical system by adding the time to the system of differential equations. That is, we add the independent variable t to create a new state vector $\dot{\mathbf{x}} = (\mathbf{x}(t), t)$ and we add the trivial time rate $dt/dt = 1$ to create a new rate $\dot{\mathbf{f}} = (\mathbf{f}(\mathbf{x}, t), 1)$. Although adding another variable makes the geometrical interpretation more difficult, it is inconsequential (and sometimes more convenient) from a computational point of view. We will use this approach in our implementation code.

We have chosen to represent the dynamical system state $(x_1, x_2, x_3, \dots, x_k, t)$ as a vector because most interesting phenomena involve multiple state variables. Even in cases where the state can be described by a single number, derivatives of order higher than one usually appear. For instance, a typical physics model, such as the pendulum, obeys Newton's second law $\mathbf{F} = m\mathbf{a}$, which is a second-order differential equation. These cases are contained in our definition above, because any higher-order differential equation can be rewritten as a new first-order ODE. As an example, we can construct a system of first-order differential equations from Eq. (17.1) by introducing the angular velocity ω as the rate of change of the angle θ . The driven pendulum can then be written as the following coupled system of first-order differential equations for the variables (θ, ω, t) :

$$\dot{\theta} = \omega \quad (17.7a)$$

$$\dot{\omega} = -\frac{g}{L} \sin(\theta) - \frac{b}{mL^2} \omega + \frac{1}{mL^2} \tau_e(t) \quad (17.7b)$$

$$\dot{t} = 1 \quad (17.7c)$$

The numerical solution of ordinary differential equations is a well-studied problem in numerical analysis. Practically all numerical techniques are based on *difference methods*. In these methods, we attempt to obtain approximate values of the solution at a sequence of mesh points $a = t_0 < t_1 < \dots < t_N = b$. Although obtaining a solution at a finite number of mesh points appears restrictive, the idea is actually very useful. In many cases, scientists and engineers use models to study how systems evolve in time by simulating them with different parameters and initial conditions, plotting or displaying the state of the system at regular time steps. This is precisely what difference methods do.

Solving differential equations numerically is both a science and an art. There exist many authoritative books on the subject, both because the discipline is very mathematically advanced, and because different numerical techniques are developed for particular types of systems. A universal solving method for ordinary differential equations does not exist and practitioners should select a method based on requirements such as speed, accuracy, and the conservation of important physical properties such as energy.

The numerical methods most frequently used fall into the following categories: Taylor methods, Runge–Kutta methods, multistep methods, and extrapolation methods. We concentrate on Taylor and Runge–Kutta methods first because they work well in many situations and are easy to implement (particularly Runge–Kutta methods), and because they will help us illustrate the main concepts and details for the computer implementation of difference methods. We will discuss other methods in Section 17.8.

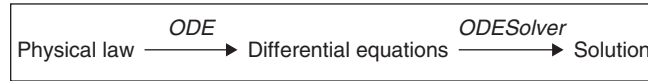


FIGURE 17.1 Differential equations can be solved using the `ODE` interface to define the equations and the `ODESolver` interface to define the numerical algorithm.

17.2.1 Implementation Techniques

The numerical solution of differential equations can be made simpler using object-oriented programming techniques that separate the differential equations from the numerical algorithm. The `ODE` and `ODESolver` interfaces shown in Figure 17.1 are code files that will be described in Section 17.5.

A Java interface is a list of methods that an object can perform. Note that Java allows us to define variables using an interface as a variable type. The `ODE` interface in the OSP library defines methods (similar to subroutines or functions) that enable us to encapsulate an initial value problem such as Eq. (17.6) in a Java class. The `ODESolver` interface defines methods that implement numerical algorithms for solving differential equations. The most important method in `ODESolver` is the `step` method that advances the state of the differential equation. This allows us to write a code such as

```

ODE ode = new Pendulum();           // creates ODE
ODESolver solver = new RK4(ode);    // creates numerical method
solver.initialize(0.01);            // each step advances by 0.01
for(int i=0; i<100; i++) {          // advances by 100 steps
    solver.step();
}
  
```

In the spirit of object-oriented programming, the details of how the differential equations are defined and solved are hidden in the `Pendulum` and `RK4` objects. This hiding is known as *encapsulation* and is a hallmark of good object-oriented design. A user can solve the pendulum problem using a different numerical algorithm by creating (instantiating) a different `ODESolver`. The OSP library contains many differential equation algorithms. The user assumes that these algorithms have been properly programmed and the library assumes that the user has obeyed the OSP API.

17.3 Taylor Methods

The first difference method we will study is based on the Taylor series expansion of the solution $\mathbf{x}(t)$. Namely

$$\mathbf{x}(t+h) = \mathbf{x}(t) + h\dot{\mathbf{x}}(t) + \frac{h^2}{2!}\ddot{\mathbf{x}}(t) + \frac{h^3}{3!}\mathbf{x}^{(3)}(t) + \cdots \quad (17.8)$$

If we know the value of $\mathbf{x}(t)$, we can compute $\dot{\mathbf{x}}(t)$ from the differential equation. The second and subsequent derivatives of $\mathbf{x}$ at t can be obtained by repeatedly differentiating Eq. (17.6). Thus,

$$\ddot{\mathbf{x}}(t) = \mathbf{f}_t(\mathbf{x}(t), t) + \mathbf{f}_x(\mathbf{x}(t), t)\mathbf{f}(\mathbf{x}(t), t) \quad (17.9)$$

and similarly for higher-order derivatives. Note, however, that the expression becomes more complicated as we compute higher-order derivatives.

Because we actually know from Eq. (17.6) the value of $\mathbf{x}$ at t_0 , we can take $h_0 = t_1 - t_0$, and use a Taylor expansion to obtain $\mathbf{u}_1$, an approximation to $\mathbf{x}(t_1)$. We can now use $\mathbf{u}_1$ to repeat the process with $h_1 = t_2 - t_1$ and obtain an approximation of $\mathbf{x}(t_2)$, $\mathbf{u}_2$, and so on until we reach t_N .

Consider again the example of the simple pendulum (with no external torque, for simplicity). By repeatedly differentiating Eq. (17.7) and applying the described numerical scheme, we obtain the coupled

recurrent sequences:

$$\theta_{n+1} = \theta_n + h_n \omega_n + \frac{h_n^2}{2!} \left[-\frac{g}{L} \sin(\theta_n) - \frac{b}{mL^2} \omega_n \right] + \dots \quad (17.10a)$$

$$\begin{aligned} \omega_{n+1} = \omega_n + h_n \left[-\frac{g}{L} \sin(\theta_n) - \frac{b}{mL^2} \omega_n \right] \\ + \frac{h_n^2}{2!} \left[-\frac{g}{L} \cos(\theta_n) \omega_n + \frac{b}{mL^2} \frac{g}{L} \sin(\theta_n) + \left(\frac{b}{mL^2} \right)^2 \omega_n \right] + \dots \end{aligned} \quad (17.10b)$$

$$t_{n+1} = t_n + h_n \quad (17.10c)$$

for $n = 0, 1, \dots, N-1$, where $t_0 = 0$, $\theta_0 = \theta(0)$, and $\omega_0 = 0$.

Implementing such an iterative procedure on a computer is straightforward. We only need to choose the number of terms in the Taylor expansion that we will use, and the appropriate values for the mesh points t_n . The typical choice is to first choose N (the number of steps), and then take all t_n equally spaced in the interval $[a, b]$. This choice leads to $h_n = h = (b - a)/N$, for all n .

We have implemented this algorithm and plotted the approximate solution of the pendulum for the algorithm including first-, second-, and third-order derivative terms of the Taylor expansion using $h = 0.1$. The result, shown in Figure 17.2, displays the typical behavior of Taylor methods. The first-order method (the largest plot in the figure), also known as the Euler method, provides a very poor approximation to the solution. A second-order expansion does a better job, but still not perfect. The solution that includes third-order terms (the darker one in the figure) is almost indistinguishable from the true solution.

The reason for this behavior can be explained using the error term for the Taylor expansion. If we include up to p order terms in this expansion, the remainder (the *local error*) has the form $O(h^{p+1})$. Thus, for the same value of h , we gain an order of magnitude in the approximation with each term. A method with $O(h^{p+1})$ local error is said to be of order p . Although the error in one step of the method is of the order of h^{p+1} , we need to take $(b - a)/h$ steps to go from a to b . Thus, the global error can be shown to be of the form $O(h^p)$.

The approximation described above can also be improved for any order by reducing the value of h . However, reducing h increases both the computational effort and the round-off error. Thus, we will need to balance the order of the method with the right value of h for our problem.

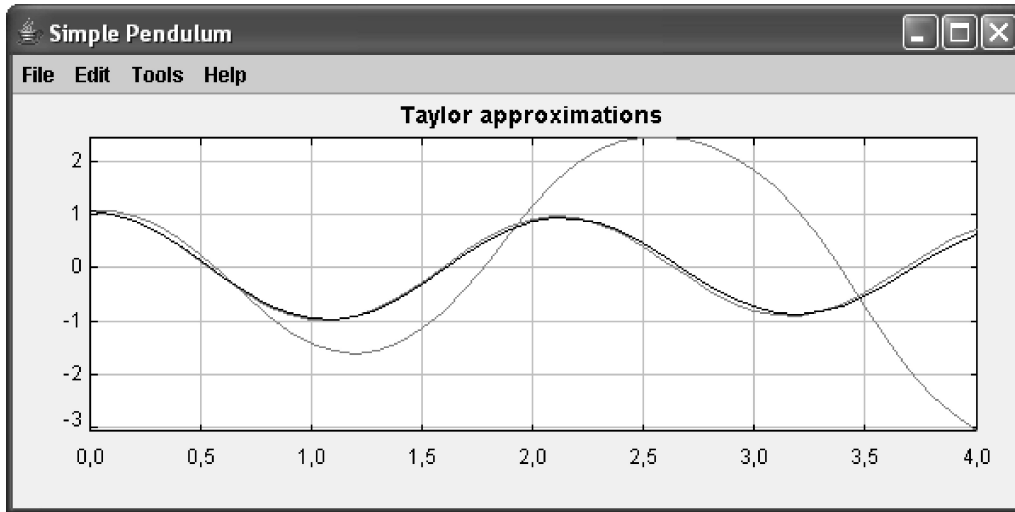


FIGURE 17.2 Taylor methods applied to the pendulum example.

17.4 Runge–Kutta Methods

Although methods based on a Taylor series expansion can be made very accurate by taking sufficiently many terms, computing higher-order derivatives becomes increasingly complicated and the resulting code cannot be reused for a different problem. For this reason, numerical analysts have developed methods with similar accuracy that are easier to implement and reuse. In particular, these methods only require evaluations of the function $\mathbf{f}$ that defines the differential equation. The Runge–Kutta methods described here are among the most popular.

Whereas Taylor methods advance the solution by evaluating $\mathbf{f}$ and its derivatives at a single point, Runge–Kutta methods advance the solution $\mathbf{u}_n$ by evaluating $\mathbf{f}$ at several intermediate points in the interval $[t_n, t_{n+1}]$. These intermediate results are combined in such a way as to match the Taylor expansion of the solution up to a given order. The precise formulation of an s -stage explicit Runge–Kutta method is the following:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{u}_n, t_n) \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{u}_n + h_n a_{21} \mathbf{k}_1, t_n + c_2 h_n) \\ &\vdots \\ \mathbf{k}_s &= \mathbf{f}(\mathbf{u}_n + h_n(a_{s1} \mathbf{k}_1 + \cdots + a_{s,s-1} \mathbf{k}_{s-1}), t_n + c_s h_n) \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + h_n(b_1 \mathbf{k}_1 + b_2 \mathbf{k}_2 + \cdots + b_s \mathbf{k}_s) \end{aligned} \quad (17.11)$$

Note that the method describes a simple recurrent algorithm, which is easy to implement on a computer and requires only the evaluation of the function $\mathbf{f}$. The method is determined by the parameters, which are listed traditionally in the form of a table (Table 17.1). Usually, the c_i satisfies the condition: $c_i = \sum_{j=1}^{i-1} a_{ij}$.

For a given s , the algorithm consists of choosing appropriate values of the a , b , and c parameters that provide a good approximation of the solution. The case $s=2$ illustrates the procedure. Consider the second-order Taylor approximation of the solution at t_n given by

$$\mathbf{x}(t_n + h_n) = \mathbf{x}(t_n) + h_n \mathbf{f} + \frac{h_n^2}{2!} (\mathbf{f}_t + \mathbf{f}_x \mathbf{f}) + O(h_n^3) \quad (17.12)$$

where $\mathbf{f}$ and its derivatives are evaluated at $(\mathbf{x}(t_n), t_n)$. If we expand the value of the approximation produced by the method in Eq. (17.11) for $s=2$ and use the Taylor expansion of the function $\mathbf{f}$, we find that

$$\begin{aligned} \mathbf{u}_{n+1} &= \mathbf{u}_n + h_n[b_1 \mathbf{f}(\mathbf{u}_n, t_n) + b_2 \mathbf{f}(\mathbf{u}_n + h_n a_{21} \mathbf{f}(\mathbf{u}_n, t_n), t_n + c_2 h_n)] \\ &= \mathbf{u}_n + h_n b_1 \mathbf{f} + h_n b_2 (\mathbf{f} + h_n c_2 \mathbf{f}_t + h_n a_{21} \mathbf{f}_x \mathbf{f}) + O(h_n^3) \\ &= \mathbf{u}_n + h_n(b_1 + b_2) \mathbf{f} + h_n^2(b_2 c_2 \mathbf{f}_t + b_2 a_{21} \mathbf{f}_x \mathbf{f}) + O(h_n^3) \end{aligned} \quad (17.13)$$

TABLE 17.1 Generic Runge–Kutta Table of Coefficients

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
c_s	a_{s1}	a_{s2}	$\cdots$	$a_{s,s-1}$	
	b_1	b_2	$\cdots$	b_{s-1}	b_s

TABLE 17.2 Coefficients of the Classical Fourth-Order Runge–Kutta Method

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	2/6	2/6	1/6

Except where explicitly indicated, $\mathbf{f}$ and its derivatives are evaluated at $(\mathbf{u}_n, t_n)$. Hence, if $\mathbf{u}_n$ is a good approximation of $\mathbf{x}(t_n)$, the method will approximate the solution at $t_n + h_n$ with a third-order local error (which leads to a second-order method) if $b_1 + b_2 = 1$, $b_2 c_2 = 1/2$, and $b_2 a_{21} = 1/2$. Because we have four unknowns and only three equations, this expression gives a one-parameter family of methods for Runge–Kutta algorithms of order 2. The most popular are the following:

(Heun’s method)

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \frac{h_n}{2} [\mathbf{f}(\mathbf{u}_n, t_n) + \mathbf{f}(\mathbf{u}_n + h_n \mathbf{f}(\mathbf{u}_n, t_n), t_n + h_n)] \quad (17.14)$$

(Midpoint method)

$$\mathbf{u}_{n+1} = \mathbf{u}_n + h_n \mathbf{f} \left(\mathbf{u}_n + \frac{h_n}{2} \mathbf{f}(\mathbf{u}_n, t_n), t_n + \frac{h_n}{2} \right) \quad (17.15)$$

(Ralston’s method)

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \frac{h_n}{4} \left[\mathbf{f}(\mathbf{u}_n, t_n) + 3\mathbf{f} \left(\mathbf{u}_n + \frac{2}{3} h_n \mathbf{f}(\mathbf{u}_n, t_n), t_n + \frac{2}{3} h_n \right) \right] \quad (17.16)$$

which correspond to $b_1 = 1/2$, $b_1 = 0$, and $b_1 = 1/4$, respectively. Among all second-order Runge–Kutta methods, Ralston’s method² provides a minimum bound of a specified type for the local error (Ralston, 1962).

The same approach can be used to obtain families of higher-order Runge–Kutta methods, although the algebra becomes much more complicated and more sophisticated techniques must be used (Hairer et al., 2000; Butcher, 1987).

The best seller of all Runge–Kutta methods is the fourth-order *classical* method given in Table 17.2, which requires four rate evaluations per step. But this method is certainly not the end of the story, and we will give higher-order methods in Section 17.6. However, for orders 5 and above, all Runge–Kutta methods require a number of stages strictly greater than the order. This limitation is one of the *Butcher’s barriers*.

17.5 Implementation

Runge–Kutta methods provide powerful, high-precision algorithms for solving initial value problems. In this section we show how these methods can be used to create computer algorithms that are effective, easy to program, and easy to reuse for different ODE problems. In particular, we will show how the OSP library implements some of these routines.

Experienced programmers divide a complex problem into several smaller parts that they later combine in a modular way. This division helps implement, test, and, if required, later replace or reuse code. The

²We name Eq. (17.16) as Ralston’s method following Ralston (1962). Other authors define Ralston’s second-order method using different coefficients.

object-oriented features of Java are very appropriate for such an approach. To show how it works in practice, OSP first defines a Java interface that encapsulates the mathematical definition of an ODE:

```
public interface ODE {
    public double[] getState();
    public void getRate(double[] state, double[] rate);
}
```

Encapsulation is an object-oriented concept that combines the data and behavior of an object, hiding the implementation from the user of that object. Encapsulation helps abstract in computer terms the characteristic features of an object. In this case, encapsulation means that to define a particular ODE a user needs to create a class that implements the two methods defined in the interface.³

The `getState` method returns the $(x_0, x_1, \dots, x_k)$ array of state variables, while the `getRate` method evaluates the derivatives using the given state array and stores the result in the given rate array $(\dot{x}_0, \dot{x}_1, \dots, \dot{x}_k)$. Since, as we have seen, Runge–Kutta methods evaluate the rate multiple times as they advance the system by h_n , the state given to `getRate` is usually not the current state of the system.

The OSP library uses the dynamical system approach and considers all ODEs as autonomous. Hence, the independent variable t is always included in the system of differential equations as a final trivial equation $\dot{t}(t) = 1$ (consequently, x_k represents the time). Also, in cases where second-order equations are coded, OSP uses the convention that the velocity coordinate follows the corresponding position coordinate when constructing the state and rate arrays. This ordering makes it possible to efficiently code certain numerical algorithms (such as the Verlet method in Section 17.8).

The second step is to abstract the general behavior of Runge–Kutta solvers. Independent of the number of stages it uses and the values of the parameters, a Runge–Kutta method (and many other methods as well) is encapsulated in the OSP library by the `ODESolver` interface:

```
public interface ODESolver {
    public void initialize(double stepSize);
    public double step();
    public void setStepSize(double stepSize);
    public double getStepSize();
}
```

Any implementing class also needs to provide a public constructor (a method that instantiates an object of the class), which takes an ODE class as a parameter. With this reference to the ODE class, the solver's `initialize` method sets the initial step size and allocates arrays to store temporary values. Typically, the `initialize` method needs to be called only once at the beginning of the program execution.

The `step` method is the heart of the algorithm. The `ODESolver` obtains a reference (pointer) to the state array using the ODE object's `getState` method and advances this state by applying the numeric method defined within the solver. Because adaptive algorithms (see Section 17.6) are free to change the step size, the method returns the value of the step size that was used. The interface also provides the `setStepSize` and `getStepSize` methods to modify and read the step size parameter.

Before showing how a particular Runge–Kutta solver class looks, let us see how the combination of these two interfaces works in practice to create clean, well-structured, and easy to reuse code. We first create a Java class that implements the ODE interface for the (undriven) pendulum example in Section 17.1:

```
public class Pendulum implements ODE {
    double[] state = new double[] {Math.PI/2.0, 0.0, 0.0};
    double m=1.0, L=1.0, g=9.8, b=0.1; // parameters
}
```

³*Method* is the term Java uses for both functions and routines. This usage should not be confused with its use in an expression that refers to an algorithm such as “Runge–Kutta method.”

```

public double[] getState() { return state; }

public void getRate(double[] state, double[] rate ){
    rate[0] = state[1];
    rate[1] = - g/L*Math.sin(state[0]) - b/(m*L*L)*state[1];
    rate[2] = 1; // time derivative
}
}

```

If we assume that we have created an implementation of the ODESolver interface for the fourth-order Runge–Kutta method called RK4, a simple driver program would instantiate both the ODE and ODESolver classes and repeatedly invoke the solver's step method to solve the initial value problem:

```

public class PendulumMain {
    public static void main(String[] args){
        double max = 4; // solution range
        double h = 0.1; // ode step size
        ODE ode = new Pendulum();
        ODESolver ode_solver = new RK4(ode);
        ode_solver.initialize(h);
        while(max>0) {
            double[] state = ode.getState();
            String xStr = "angular position = "+state[0];
            String vStr = ", angular velocity = "+state[1];
            String tStr = ", time = "+state[2];
            System.out.print(xStr+vStr+tStr+"\n");
            max -= ode_solver.step();
        }
    }
}

```

The important point is that the end user only needs to program the class that implements the ODE interface and thereby defines the initial value problem. Numerical algorithms, such as the RK4 class, are implemented in the OSP library. These algorithms can be written (and tested) once, perhaps by a professional numerical analyst, in an independent manner. Switching to a different numerical algorithm only requires instantiating a different ODESolver class (see Table 17.6 for a listing of solvers in the OSP library).

We now list, for completeness, a possible implementation of the step method for the RK4 method:

```

public double step() {
    double state[] = ode.getState();
    ode.getRate(state, k1);
    for(int i = 0;i<numEqn;i++) {
        temp_state[i] = state[i]+stepSize*k1[i]/2;
    }
    ode.getRate(temp_state, k2);
    for(int i = 0;i<numEqn;i++) {
        temp_state[i] = state[i]+stepSize*k2[i]/2;
    }
    ode.getRate(temp_state, k3);
    for(int i = 0;i<numEqn;i++) {
        temp_state[i] = state[i]+stepSize*k3[i];
    }
}

```

```

ode.getRate(temp_state, k4);
for(int i = 0; i < numEqn; i++) {
    state[i] = state[i] + stepSize * (k1[i] + 2*k2[i] + 2*k3[i] + k4[i]) / 6.0;
}
return stepSize;
}

```

17.6 Adaptive Step

An important aspect of solving ODEs numerically is that of choosing the best possible set of mesh points $t_0 < \dots < t_N$, mentioned in Section 17.2. The *fixed step size* approach that we have used so far, that is, $h_n = h = (b - a)/N$, is not always appropriate. The solution of the ODE may vary rapidly in some parts of the $[a, b]$ interval (which requires a small step size), while it may be very smooth in other parts of it (which allows a larger step size). Taking the same step size for the entire interval might result in either loss of precision or in an unnecessary waste of computer resources, or both.

Making an appropriate choice of mesh points in advance can be difficult. A detailed analytical study of the local error at different points within the interval is required. Alternatively, there exist numerical techniques that allow the computer to estimate this error and automatically compute an appropriate value of h_n for each integration step. These techniques are, for Runge–Kutta methods, based on two approaches: interval-halving and embedded formulas.

Interval-Halving

The first approach consists in using, from each point $(\mathbf{u}_n, t_n)$, the same Runge–Kutta method in two parallel computations. The first one applies the method once with a given step size h , while the second uses the method twice, each with half this step size, i.e., $h/2$. We thus obtain two different approximations of the solution at $t_n + h$. Both values are then compared to estimate the error obtained with this step size, to accept the solution or not, and to compute a more appropriate step size (Schilling and Harris, 2000).

Although this extrapolation process can be easily programmed, the resulting code is inefficient. For the case of the fourth-order Runge–Kutta method, we would need 11 different evaluations of the rate function for each individual step and would achieve only a fifth-order approximation.

Embedded Runge–Kutta Formulas

A more efficient scheme was first discovered by Fehlberg, who found a pair of Runge–Kutta formulas that used the same set of coefficients to provide two approximations of different orders. The difference between both approximations can then be used to estimate the error of the lower-order formula, while taking the higher-order approximation as the final output of the method (this is called *local extrapolation*).

Since the original Fehlberg scheme, many other embedded formulas have been found. The most used ones are the 6-stage formulas of Cash and Karp (Table 17.3), and the 7-stage formulas of Dormand and Prince (table not provided), which attempt to minimize the error of the local extrapolation approximation. Table 17.3 contains a pair of Runge–Kutta–Fehlberg formulas of orders 5 and 4. The first row of b -coefficients in Table 17.3 corresponds to the higher-order approximation, while the second row gives the lower-order one. Both Cash–Karp and Dormand–Prince schemes are implemented in the OSP library.

Adapting the Step

Once the estimated error is found, the technique adapts the step size so that this error keeps within reasonable limits, decreasing the step size if the error is too large, and increasing it if the error is too small. A popular procedure to compute the new step size, $\tilde{h}$, consists in using the following formula (Press et al., 1992):

$$\tilde{h} = Sh \left| \frac{\Delta_0}{\Delta_1} \right|^\alpha \quad (17.17)$$

TABLE 17.3 Coefficients for the Embedded Runge–Kutta Formulas Computed by Cash and Karp

0						
1/5	1/5					
3/10	3/40	9/40				
3/5	3/10	−9/10	6/5			
1	−11/54	5/2	−70/27	35/27		
7/8	1631/55296	175/512	575/13824	44275/110592	253/4096	
	37/378	0	250/621	125/594	0	512/1771
	2825/27648	0	18575/48384	13525/55296	277/14336	1/4

where S is a safety constant (such as 0.9) that prevents $\tilde{h}$ from actually reaching the limit for an acceptable step size, h the current step size, Δ_0 denotes the desired accuracy, and Δ_1 measures the current accuracy. Finally, α denotes a constant that (for a p -order method) is $1/(p+1)$ when the step size is increased, and $1/p$ otherwise.

There are different ways to interpret Δ_0 and Δ_1 in Eq. (17.17). Usually, Δ_1 is taken as the maximum of the absolute values of the components of the estimated error (for embedded formulas, of the difference of both approximations). For Δ_0 , the simplest option is to let the user specify a suitable small tolerance, ϵ . For other possibilities such as including a scaling vector, see Press et al. (1992), Hairer et al. (2000), and Enright et al. (1995).

17.7 Implementation of Adaptive Step

Implementing an adaptive algorithm based on a pair of Runge–Kutta embedded formulas is straightforward. The following code shows how the `step` method can be implemented for the Cash–Karp formulas. Note that the code includes additional checks to avoid abrupt changes to the step size. In addition, the algorithm will return if the required precision cannot be attained.

```
public double step() {
    int iterations = 10;
    double currentStep = stepSize, error=0;
    double state[] = ode.getState();
    ode.getRate(state, k[0]); // gets the initial rate
    do {
        iterations--;
        currentStep = stepSize;
        // Compute the k's
        for (int s=1; s<numStages; s++) {
            for (int i=0; i<numEqn; i++) {
                temp_state[i] = state[i];
                for (int j=0; j<s; j++) {
                    temp_state[i] = temp_state[i]+stepSize*a[s-1][j]*k[j][i];
                }
            }
            ode.getRate(temp_state, k[s]);
        }
        // Computes the error
        error = 0;
    } while (iterations > 0);
}
```

```

    for (int i=0; i<numEqn; i++) {
        truncErr = 0;
        for (int s=0; s<numStages; s++) {
            truncErr = truncErr+stepSize*er[s]*k[s][i];
        }
        error = Math.max(error,Math.abs(truncErr));
    }
    if (error<=Float.MIN_VALUE) { // error too small
        error = tol/1.0e5; // actually increase stepSize x10
    }
    // find h step for the next try.
    if(error>tol) { // shrink, but by no more than x10
        double fac = 0.9*Math.pow(error/tol,-0.25);
        stepSize = stepSize*Math.max(fac,0.1);
    }
    else if(error<tol/10.0) { // grow, but no more than x10
        double fac = 0.9*Math.pow(error/tol, -0.2);
        if(fac>1) { // sometimes fac<1 if error/tol close to 1
            stepSize = stepSize*Math.min(fac, 10);
        }
    }
} while(error>tol && iterations>0);
// advance the state
for (int i=0; i<numEqn; i++){
    for (int s=0; s<numStages; s++){
        state[i] = state[i]+currentStep*b5[s]*k[s][i];
    }
}
return currentStep; // step actually taken.
}

```

Independent of how the algorithm is implemented internally, Java's object-oriented features again help us construct easy-to-use and well-structured code for solving ODEs using adaptive methods. The OSP library defines the `ODEAdaptiveSolver` interface that encapsulates the methods for an adaptive solver:

```

public interface ODEAdaptiveSolver extends ODESolver {
    public void setTolerance(double tol);
    public double getTolerance();
    public int getErrorCode();
}

```

Because the interface *extends* the previously defined `ODESolver` interface, it inherits all the methods defined in this interface. The only new methods are those that allow the user to specify or read the desired tolerance (the ϵ mentioned above) and to check for a possible error in the application of the algorithm. This interface allows the user to create a code such as the following:

```

ODEAdaptiveSolver ode_solver = new CashKarp45(ode);
ode_solver.initialize(h);
ode_solver.setTolerance(1.0e-3);

```

Note that the only difference with the previous code is in the instantiation of the `ODEAdaptiveSolver` object and the addition of a line that sets the desired tolerance. All other parts of the driver program and the definition of the ODE remain unchanged.

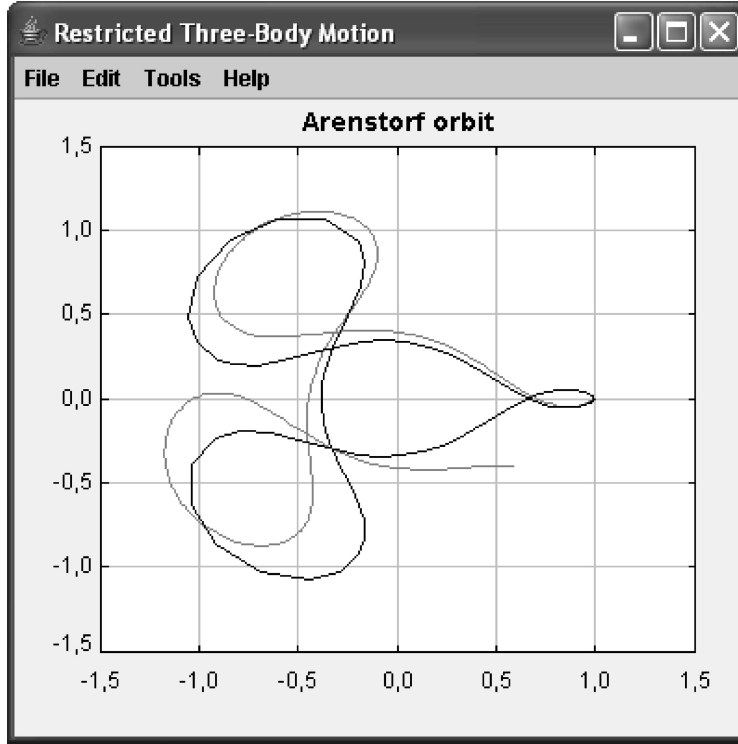


FIGURE 17.3 Arenstorf orbit computed with fixed step (gray) and adaptive (black) Runge–Kutta methods.

Using an adaptive algorithm can dramatically increase the performance of our programs in situations where the solution has regions of different behavior. A typical example is that of the computation of the *Arenstorf orbits*, closed trajectories of the restricted three-body problem (two bodies of masses μ and $1 - \mu$ moving in a circular orbit, and a third body of negligible mass moving in the same plane), such as a satellite–earth–moon system. The equations for the motion of the third body are given by Hairer et al. (2000):

$$\ddot{x}_1(t) = x_1(t) + 2\dot{x}_2(t) - (1 - \mu)\frac{x_1(t) + \mu}{D_1} - \mu\frac{x_1(t) - (1 - \mu)}{D_2} \quad (17.18a)$$

$$\ddot{x}_2(t) = x_2(t) - 2\dot{x}_1(t) - (1 - \mu)\frac{x_2(t)}{D_1} - \mu\frac{x_2(t)}{D_2} \quad (17.18b)$$

where $D_1 = ((x_1(t) + \mu)^2 + x_2^2(t))^{3/2}$ and $D_2 = ((x_1(t) - (1 - \mu))^2 + x_2^2(t))^{3/2}$.

The Arenstorf orbit for the initial values $x_1(0) = 0.994$, $\dot{x}_1(0) = 0$, $x_2(0) = 0$, $\dot{x}_2(0) = -2.0317326295573368357302057924$, and $\mu = 0.012277471$ is displayed in Figure 17.3.⁴ The gray trajectory has been computed using 3000 fixed step iterations of the fourth-order Runge–Kutta algorithm (the trajectory is not even closed), while the black, closed trajectory has been computed by the adaptive Cash–Karp algorithm with a tolerance of 10^{-5} in 123 steps with 98 steps accepted and 25 steps rejected. The fact that the second algorithm is of order 5 is secondary. Comparing the total number of rate evaluations (12,000 versus 713) shows that the adaptive method is far superior.

⁴This high accuracy is required because Arenstorf orbits are very sensitive to small changes in the initial conditions.

Multisteping

Given the relatively small additional effort required to compute two Runge–Kutta solutions and thereby to control the error, is there any reason to use a nonadaptive algorithm? Sometimes. Adaptive algorithms do not work if the rate contains discontinuous functions. A fixed step size is also convenient if the output is to have evenly spaced values.

However, a fixed step size can also be achieved by taking multiple steps. Because the step size is chosen so as to obtain the desired accuracy, the last step will almost always overshoot the desired endpoint. This overshoot can be eliminated by reducing the last step. The `ODEMultistepSolver` class in the OSP library combines multiple steps of any adaptive algorithm in this manner.⁵

Interpolation

Using an adaptive solver to compute the solution at fixed intervals has an obvious inconvenience, it prevents the adaptive algorithm from taking large steps in areas where the estimated error is small. A different technique, interpolation, can help improve performance.

The idea consists in letting the solver work internally at its own pace, taking as large a step as it sees fit and asking it to provide approximated values of the solution at equally spaced points. The algorithm is then assumed to interpolate the internally computed values to produce an approximation at the given points. The interpolation is expected to have a similar order of precision as the internal approximations of the solution.

Interpolation is a bit more complex to implement, since it actually involves the concept of *dense output* (Hairer et al., 2000). This means that, instead of providing an approximate solution to the ODE at a finite set of points of the interval $[a, b]$, the method provides a function that can approximate the solution of the ODE at *any* point of it. The user could then in principle use this function to produce solution points everywhere in the trajectory (a dense output). OSP has implemented this technique using the `ODEInterpolationSolver` interface for two of the most powerful, higher-order methods of the library: the `Dopri5` and `Dopri853` classes. The first one implements a pair of embedded optimized formulas of orders 5 and 4, found by Dormand and Prince, together with dense output interpolation of order 4. The second is based on a pair of embedded formulas of orders 8 and 5, with dense output interpolation of order 7.

These two ODE solvers are available in the OSP library and can be created using factory methods as the code below shows.

```
ODEAdaptiveSolver dopri5=ODEInterpolationSolver.Dopri5(ode);
ODEAdaptiveSolver dopri8=ODEInterpolationSolver.Dopri853(ode);
```

17.8 Performance and Other Methods

Up to this point, we have only used explicit Runge–Kutta methods for solving ordinary differential equations. Explicit Runge–Kutta methods work correctly in almost every situation. But they might not be the most efficient or convenient ones:

- When the ODE to solve is a *stiff* equation for which implicit algorithms have better stability properties and are therefore more efficient.
- When the solution of the ODE is very smooth or the rate function is very expensive to evaluate, a multistep algorithm can be preferred.
- When high accuracy is required (of the order of 10^{-12} or higher), and evaluating the rate function is not cheap in terms of CPU time, extrapolation techniques can be more efficient in this situation.

⁵Multisteping and the `ODEMultistepSolver` class should not be confused with the multistep methods described in Section 17.8.

TABLE 17.4 Coefficients for Radau IIA Fifth-Order Method

$(4 - \sqrt{6})/10$	$(88 - 7\sqrt{6})/360$	$(296 - 169\sqrt{6})/1800$	$(-2 + 3\sqrt{6})/255$
$(4 + \sqrt{6})/10$	$(296 + 169\sqrt{6})/1800$	$(88 + 7\sqrt{6})/360$	$(-2 - 3\sqrt{6})/255$
1	$(16 - \sqrt{6})/36$	$(16 + \sqrt{6})/36$	1/9
	$(16 - \sqrt{6})/36$	$(16 + \sqrt{6})/36$	1/9

- When the long-term behavior or the preservation of certain geometrical properties of the computed solutions of a Hamiltonian system are of interest, a *symplectic* algorithm is preferred.

We now briefly cover these four cases.

Implicit Algorithms and Stiff Equations

An ODE is said to be *stiff* when the solution comprises two or more terms that change at speeds which differ in several orders of magnitude. Consider the simple one-dimensional initial value problem (Chapra and Canale, 2006):

$$\dot{x}(t) = -1000x(t) + 3000 - 2000e^{-t}, \quad x(0) = 0 \quad (17.19)$$

which has the exact solution $x(t) = 3 - \frac{997}{999}e^{-1000t} - \frac{2000}{999}e^{-t}$. The solution contains a slow-changing exponential together with a fast-changing one. Although the fast exponential quickly contributes only very small values, its presence forces a typical explicit algorithm to keep a small step size, even after the transient part of the solution becomes very small.

The stability of an algorithm refers to its capability of not exponentially propagating the (unavoidable) small errors that take place in the solving process, even when taking moderately large step sizes. Implicit algorithms typically show better stability properties and consequently perform better on stiff problems.

A more generic formulation of an s -stage Runge–Kutta algorithm,

$$\begin{aligned} \mathbf{k}_i &= \mathbf{f}(\mathbf{u}_n + h_n \sum_{j=1}^s a_{ij} \mathbf{k}_j, t_n + c_i h_n), \quad i = 1, \dots, s \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + h_n \sum_{i=1}^s b_i \mathbf{k}_i \end{aligned} \quad (17.20)$$

allows, when any of the a_{ij} is not 0, for implicit algorithms, in which the value of the $\mathbf{k}_i$ may appear at both sides of the evaluation of some intermediate rate. If the rate function is nontrivial, finding the solution of the (possibly nonlinear) equations imposes an additional computation burden.

Implicit Runge–Kutta methods often possess better stability properties than their explicit counterparts given reasonably good behavior of the rate function $\mathbf{f}$. Their implementation is, however, more sophisticated, because it must include techniques to solve the nonlinear system of equations.

The OSP library implements an implicit Runge–Kutta method in the `Radau5` class, following the implementation proposed in Hairer and Wanner (2002). This class includes all the necessary routines to implement the Radau IIA fifth-order method given by Table 17.4, with step size control and dense output (interpolation).

Although implementing such an advanced algorithm is reserved to specialists, using it in the OSP framework is very easy. The `Radau5` class is created using the static (factory) method:

```
ODEAdaptiveSolver radau5=ODEInterpolationSolver.Radau5(ode);
```

and can be treated as any other `ODEAdaptiveSolver` object.

To show how the method compares to explicit Runge–Kutta methods, the classical fourth-order Runge–Kutta method requires 400 rate function evaluations (for a step size of 0.001) to achieve comparable results

in the small interval $[0, 0.1]$ when solving Chapra and Canale's initial value problem [Eq. (17.19)]. The adaptive Cash–Karp formulas required 217 evaluations for the same test (the number of steps growing with the length of the interval, not being restricted to the initial transient part of the solution). Finally, Radau5 solves the problem with only 60 evaluations, showing no deviation from the analytical solution.

For nonstiff ODEs, explicit adaptive methods usually perform better than implicit ones. For instance, to obtain similar results for the Arenstorf orbit computation of Section 17.7, Radau5 required 2.5 times the number of rate function evaluations of CashKarp45.

Multistep Methods

These methods take advantage of the information contained in previous steps of the solution. The general algorithm for a $(p + 1)$ -step method is

$$\mathbf{u}_{n+1} = \sum_{j=0}^p a_j \mathbf{u}_{n-j} + h \sum_{j=-1}^p b_j \mathbf{f}_{n-j}, \quad (n \geq p) \quad (17.21)$$

where $\mathbf{f}_j$ represents $\mathbf{f}(\mathbf{u}_j, t_j)$. Note that, if $b_{-1} \neq 0$, the method is implicit and must be solved by iteration. Simple multistep methods for initial value problems belong to two main families: Adams' methods and backwards differentiation formulas.

Adams' methods derive from numerical methods for the equivalent integral equation:

$$\mathbf{x}(t_{n+1}) = \mathbf{x}(t_n) + \int_{t_n}^{t_{n+1}} \mathbf{f}(\mathbf{x}(t), t) dt \quad (17.22)$$

Typical examples of Adams' method are the following fourth-order explicit and implicit methods (Atkinson, 1989):

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \frac{h}{24}(55\mathbf{f}_n - 59\mathbf{f}_{n-1} + 37\mathbf{f}_{n-2} - 9\mathbf{f}_{n-3}) \quad (17.23a)$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \frac{h}{24}(9\mathbf{f}_{n+1} + 19\mathbf{f}_n - 5\mathbf{f}_{n-1} + \mathbf{f}_{n-2}) \quad (17.23b)$$

Sometimes, explicit and implicit Adams' formulas are applied in pairs, in *predictor–corrector* formulas. The latter is a mixed method that tries to obtain some of the stability properties of implicit formulas with the simplicity of implementation of explicit ones. The scheme first uses an explicit method to *predict* the solution of the ODE at t_{n+1} . It then *corrects* the approximation by applying the implicit formula with the value of $\mathbf{f}_{n+1}$ computed from the prediction.

Backwards differentiation formulas are obtained by approximating the solution by a polynomial through a series of past $\mathbf{u}_{n-j}$ points and then taking $\mathbf{u}_{n+1}$ such that the polynomial satisfies the ODE at t_{n+1} . A typical result of this process is (Hairer et al., 2000)

$$\frac{25}{12}\mathbf{u}_{n+1} - 4\mathbf{u}_n + 3\mathbf{u}_{n-1} - \frac{4}{3}\mathbf{u}_{n-2} + \frac{1}{4}\mathbf{u}_{n-3} = h\mathbf{f}_{n+1} \quad (17.24)$$

Both Adams' methods and backwards differentiation formulas need the help of an auxiliary method to start the process (that is, to compute the first $p + 1$ points of the solution) and require that the steps be equally spaced. This requirement makes it complicated (though not impossible) to implement adaptive step versions of the algorithms.

Extrapolation Methods

These algorithms effectively accelerate the convergence by using a power series to extrapolate to zero step size applying successively the algorithm with different step sizes. The Bulirsch–Stoer method chooses a

TABLE 17.5 Partitioned Runge–Kutta Coefficients for the Verlet Method. The Left-Hand Table is Used to Advance the Position and the Right-Hand Table is Used to Advance the Velocity

0	0	0	1/2	1/2	0
1	1/2	1/2	1/2	1/2	0
	1/2	1/2		1/2	1/2

sequence of increasing integer numbers N , and then uses, for each of them, a fixed step H , and $h = H/N$, the formula:

$$\mathbf{v}_1 = \mathbf{u}_n + h\mathbf{f}(\mathbf{u}_n, t_n) \quad (17.25a)$$

$$\mathbf{v}_{i+1} = \mathbf{v}_{i-1} + 2h\mathbf{f}(\mathbf{v}_i, t_n + ih), \quad \text{for } i = 1, 2, \dots, N-1 \quad (17.25b)$$

$$\mathbf{u}_{n+1} = \frac{1}{2}[\mathbf{v}_N + \mathbf{v}_{N-1} + h\mathbf{f}(\mathbf{v}_N, t_n + H)] \quad (17.25c)$$

to advance from $(\mathbf{u}_n, t_n)$ to $(\mathbf{u}_{n+1}, t_n + H)$. The error term for this formula is an expansion with only even powers of h . Standard extrapolation techniques can then be used to obtain an arbitrarily high-order method that accurately approximates the solution in the given interval (Press et al., 1992).

Symplectic Integration Methods

These methods can be the preferred choice when studying Hamiltonian systems:

$$\dot{p}_i = -\frac{\delta H}{\delta q_i}(\mathbf{p}, \mathbf{q}), \quad \dot{q}_i = \frac{\delta H}{\delta p_i}(\mathbf{p}, \mathbf{q}), \quad i = 1, \dots, k \quad (17.26)$$

which define a $2k$ -dimensional ODE with variables q_i and p_i representing the generalized coordinates of the system and the generalized momenta, respectively. Such problems frequently arise when modeling mechanical systems, where the Hamiltonian H is the energy function of the system.

The *symplectic* property of these problems states that the flow of the system preserves the differential 2-form:

$$\omega^2 = \sum_{i=1}^k dp_i \wedge dq_i \quad (17.27)$$

The symbol $\wedge$ represents the exterior or *wedge* product between the two differential 1-forms (do Carmo, 1994). If $k = 1$ the 2-form above can be interpreted as an area in the plane. In this case, the symplectic property has the geometrical interpretation that the area (in phase space) is preserved by the flow. Numerical methods that also preserve this geometric property are called symplectic and are of interest when studying this type of problems.

Although no explicit Runge–Kutta method is symplectic, it is possible to find high-order so-called *partitioned* Runge–Kutta methods that are symplectic. These consist of a pair of Runge–Kutta methods with different sets of coefficients applied “separately” to the variables p_i and q_i . For separable systems, in which the Hamiltonian is of the form $H = T(\mathbf{p}) + V(\mathbf{q})$, such methods do exist (Hairer et al., 2000, 2002; Enright et al., 1995).

A simple lower-order symplectic method that can be applied to solve the equations of motion with acceleration $a(t) = f(x(t), v(t), t)$ for a system of particles is the Verlet algorithm. This is an easy-to-program multistep method that produces stable long-term trajectories. The method has the additional advantage that, although it does not preserve the energy of the system over short times, it does produce accurate averages over long times because the energy oscillates about the mean. It therefore reduces the computational effort for systems where statistical averages are more important than the accuracy of particular trajectories.

The partitioned Runge–Kutta coefficients for the Verlet method are given for (x, v) in Table 17.5. Because these coefficients define a simple implicit algorithm, it is convenient to solve for the values at t_{n+1} . This reformulation, which is included in the OSP library, averages the acceleration at the old and new positions to update the velocity. It is known as the velocity form of the Verlet method and is given by

$$x_{n+1} = x_n + v_n h + \frac{1}{2} a_n h^2 \quad (17.28a)$$

$$v_{n+1} = v_n + \frac{1}{2} (a_{n+1} + a_n) h \quad (17.28b)$$

Important areas in which symplectic methods are preferred to other, perhaps more accurate, methods are molecular dynamics and astrophysics, where large-scale simulations involving thousands of particles are studied over very long time scales (Gray et al., 1994).

17.9 State Events

Sometimes the model of a continuous system needs to include discontinuities to reflect special situations. The typical case is that of a collision between two objects. Although the collision can be modeled at a microscopic scale, it is frequently much more convenient to stick to the macroscopic model and instruct the computer to detect that the collision has taken place, find the precise moment when it happened, compute explicitly the state of the system after the collision, and finally restart the continuous model from it. This is a particular case of what is known as a state event (Cellier and Kofman, 2006).

Because our numerical method solves the ODE at given step sizes (either fixed or adaptive), a particular event will almost always take place in between two successive computed states. A simple way to implement event detection is to provide a real valued function of the state, $h(\mathbf{x})$, which changes sign whenever an event takes place. The algorithm can then keep track of possible changes in the sign of h in each solution step and then apply a standard root-finding algorithm to find the precise instant of time when the event takes place. The user must provide a method (function) that changes the state of the system at that particular time. For example, a collision event would use conservation of energy and momentum to change particle velocities.

The OSP library implements a simple state event mechanism in the `StateEvent` interface. An event takes place whenever the event function h changes from positive to strictly negative (zero does not trigger an event). A sample implementation of this interface for a one-dimensional collision between two planar disks is

```
class OneDimensionalCollisionEvent implements StateEvent {

    public double getTolerance() { return 1.0e-4; } // precision

    public double evaluate(double[] state) { // the event function h
        return Math.abs(state[0]-state[2])-(radius1+radius2);
    }

    public boolean action() { // What to do on collision
        double v_temp = state[1];
        state[1] =
            (2*mass2*state[3]+(mass1-mass2)*state[1])/(mass1+mass2);
        state[3] =
            (2*mass1*v_temp+(mass2-mass1)*state[3])/(mass1+mass2);
        return true; // return state at the collision
    }
}
```

TABLE 17.6 List of ODE Solvers Implemented in the OSP Library (ERK, Explicit Runge–Kutta; IRK, Implicit Runge–Kutta)

Type of Solver	Solver Class Name
ERK fixed step	Euler, Butcher5, Fehlberg8, Heun3, Ralston2, RK4
ERK adaptive	CashKarp45, DormandPrince45
ERK adaptive + multistepping	ODEMultistepSolver
ERK adaptive + interpolation	Dopri5, Dopri853
IRK adaptive + interpolation	Radau5
Predictor–corrector fixed step	Adams4, Adams5, Adams6
Symplectic	Verlet
Event-enabled fixed step	ODEBisectionSolver

Because the model is one-dimensional, the disks have the same y coordinate and the state array is $(x_1, vx_1, x_2, vx_2, t)$.

The `ODEBisectionEventSolver` class in the OSP library can use any fixed step `ODESolver` to advance the system. It checks for possible events and applies the bisection algorithm to find the instant of the event (up to the prescribed tolerance) if an event has occurred. Although more sophisticated techniques can be used, the bisection method is both simple and effective, reduces the number of the computations involved, and provides support for handling situations in which multiple events take place, perhaps simultaneously. Examples of systems displaying quite sophisticated event-handling are available on the web links for this chapter.

17.10 The OSP Library

Table 17.6 summarizes the numerical algorithms implemented in the OSP library. The `Dopri5`, `Dopri853`, and `Radau5` methods were contributed to the library by Andrew Gusev and Yuri B. Senichenkov of Saint Petersburg Polytechnic University, Russia. These high-order differential equation solvers are being distributed in an optional `osp_ode.zip` archive. All other interfaces and algorithms are implemented in the numerics package (`org.opensourcephysics.numerics`) in the core OSP library and are being distributed in the `osp_core.zip` archive. Both code archives are available on the OSP website (<http://www.opensourcephysics.org>).

The OSP project is a synergy of curriculum development, computational physics, and physics education research. One goal of the project is to make a Java library and a large number of Java simulations available for education using the GNU Open Source model. You can redistribute it and/or modify it under the terms of the GNU General Public License (GPL), as published by the Free Software Foundation either version 2 of the License or (at your option) any later version. Code that uses any portion of the code in the `org.opensourcephysics` package or any subpackage (subdirectory) of this package must also be released under the GNU GPL.

Programmers wishing to adopt OSP code for their projects are encouraged to do so, provided that they release their source code under the GNU Open Source GPL. Curricular material and developer resources are being distributed from the OSP server hosted at Davidson College and from other servers.

References

- Atkinson, K. E. (1989). *An Introduction to Numerical Analysis*, 2nd Ed. Wiley, New York.
- Butcher, J. C. (1987). *The Numerical Analysis of Ordinary Differential Equations. Runge–Kutta and General Linear Methods*. Wiley, Chichester.

- Cellier, F. E. and E. Kofman (2006). *Continuous System Simulation*. Springer, New York.
- Chapra, S. C. and R. P. Canale (2006). *Numerical Methods for Engineers*. McGraw-Hill, New York.
- Christian, W. (2007). *Open Source Physics: A Users Guide with Examples*. Addison-Wesley, San Francisco.
- do Carmo, M. P. (1994). *Differential Forms and Applications*. Springer, New York.
- Enright, W. H., D. J. Higham, B. Owren, and P. Sharp (1995). A survey of the explicit Runge–Kutta method. Technical Report 94-291. University of Toronto.
- Gray, S. K., D. W. Noid, and B. G. Sumpter (1994). Symplectic integrators for large scale molecular dynamics simulations: A comparison of several explicit methods. *Journal of Chemical Physics* 101(5) 4062–4072.
- Hairer, E., C. Lubich, and G. Wanner (2002). *Geometric Numerical Integration, Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer, Berlin.
- Hairer, E., S. P. Nørsett, and G. Wanner (2000). *Solving Ordinary Differential Equations I (Nonstiff Problems)*, 2nd Ed. Springer, Berlin.
- Hairer, E. and G. Wanner (2002). *Solving Ordinary Differential Equations II (Stiff and Differential-Algebraic Problems)*, 2nd Ed. Springer, New York.
- Keller, H. B. (1992). *Numerical Methods for Two-Point Boundary-Value Problems*. Dover, New York.
- Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). *Numerical Recipes in C: The Art of Scientific Computing*, 2nd Ed. Cambridge University Press, New York.
- Ralston, A. (1962). Runge–Kutta methods with minimum error bounds. *Mathematics of Computations* 16(80) 431–437.
- Schilling, R. J. and S. L. Harris (2000). *Applied Numerical Methods for Engineers*. Brooks/Cole, Pacific Grove, CA.



QUERY FORM

CRC Press

Handbook of Dynamix System Modeling

JOURNAL TITLE:	C5653
ARTICLE NO:	Ch017

Queries and/or remarks

Query No.	Details Required	Author's Response
AQ1	Please provide the expansion for GNU.	