

## Chapter 7

# Two-Dimensional Trajectories and the ODE Interface

©2002 by Harvey Gould, Jan Tobochnik, and Wolfgang Christian  
11 March 2002

We introduce two Java interfaces for the solution of systems of first-order differential equations and study motion in two dimensions.

### 7.1 Trajectories

You are probably familiar with two-dimensional trajectory problems in the absence of air resistance. For example, if a ball is thrown in the air with an initial velocity  $v_0$  at an angle  $\theta_0$  with respect to the ground, how far will the ball travel in the horizontal direction, and what is its maximum height and time of flight? Suppose that a ball is released at a nonzero height  $h$  above the ground. What is the launch angle for the maximum range? Are your answers still applicable if air resistance is taken into account? We consider these and similar questions in the following.

Consider an object of mass  $m$  whose initial velocity  $\mathbf{v}_0$  is directed at an angle  $\theta_0$  above the horizontal (see Figure 7.1a). The particle is subjected to gravitational and drag forces of magnitude  $mg$  and  $F_d$ ; the direction of the drag force is opposite to  $\mathbf{v}$  (see Figure 7.1b). Newton's equations of motion for the  $x$  and  $y$  components of the motion can be written as

$$m \frac{dv_x}{dt} = -F_d \cos \theta \quad (7.1a)$$

$$m \frac{dv_y}{dt} = -mg - F_d \sin \theta. \quad (7.1b)$$

As an example, let us consider a round steel ball of radius 4 cm. A reasonable assumption for a steel ball of this size and typical speed is that  $F_d = k_2 v^2$ . Because  $v_x = v \cos \theta$  and  $v_y = v \sin \theta$ ,

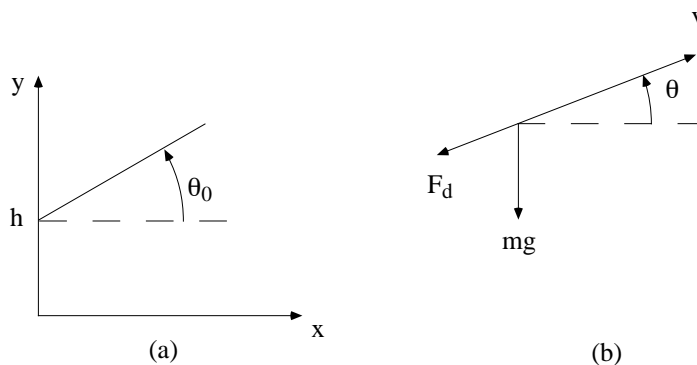


Figure 7.1: (a) A ball is thrown from a height  $h$  at an launch angle  $\theta_0$  measured with respect to the horizontal. The initial velocity is  $\mathbf{v}_0$ . (b) The gravitational and drag forces on a particle.

we can rewrite (7.1) as

$$\frac{dv_x}{dt} = -Cvv_x \quad (7.2a)$$

$$\frac{dv_y}{dt} = -g - Cvv_y, \quad (7.2b)$$

where  $C = k_2/m$ . Note that  $-Cvv_x$  and  $-Cvv_y$  are the  $x$  and  $y$  components of the drag force  $-Cv^2$ . Because (7.2) for the change in  $v_x$  and  $v_y$  involve the square of the velocity,  $v^2 = v_x^2 + v_y^2$ , we cannot calculate the vertical motion of the ball without reference to the horizontal motion, that is, the motion in the  $x$  and  $y$  direction is *coupled*.

The two-dimensional trajectory problem leads to a system of five first-order differential equations for the following variables:  $x$ ,  $v_x$ ,  $y$ ,  $v_y$ , and  $t$ . We could, of course, solve these equations by incorporating a numerical algorithm directly into the class that defines the physics as we did in Chapters 5 and 6. This approach to solving equations works and may, in fact, be desirable for problems where the algorithm is unlikely to change and computational speed is the overriding factor (see Chapter ??). However, this approach is not very object-oriented. Changing the algorithm or the physics requires that we recode the class. But the physics and the algorithm are clearly separate concepts and should be treated as separate classes. To make this separation we introduce the ODE (ordinary differential equation) and `ODESolver` interfaces in Section 7.2.

## 7.2 Solving ODEs with Interfaces

Although interfaces can be used for solving individual differential equations, their strength is in dealing with systems of many equations. The key to using the `OSP` interfaces to solve differential equations is to use arrays for the variables and their derivatives. The `state` array contains the current value for each variable; the `rate` array is calculated on the fly whenever the rate of change of these variables is needed. These arrays are used by two interfaces, `ODE` and `ODESolver`. The `ODE`

interface defines the variables and the derivatives that the numerical method needs to compute the solution, and the `ODESolver` interface defines the numerical method.

The ODE interface contains two methods, `getState` and `getRate`, and is written as follows:

```
package org.opensourcephysics.numerics;
public interface ODE {
    public double[] getState();
    public void getRate(double[] state, double[] rate);
}
```

This interface allows us to encapsulate almost any physics problem that can be written in terms of ordinary differential equations.

The `ODESolver` interface defines methods that initialize and set the step size of a numerical method and is written as follows:

```
package org.opensourcephysics.numerics;
public interface ODESolver {
    public void initialize(double stepSize);
    public double step();
    public void setStepSize(double stepSize);
    public double getStepSize();
}
```

The `initialize` method sets the step size and, if necessary, allocates temporary arrays and initializes variables. It is usually invoked within the solver's constructor but should be invoked again if the number of differential equations change. The other methods are self explanatory.

To solve a system of differential equations, an object that implements the ODE interface is first created and then passed to a differential equation solver in the solver's constructor. For example, if we wish to solve a radioactive decay problem using the Euler algorithm, we set up the problem using the following three statements:

```
ODE decay = new Decay();
ODESolver odeSolver = new Euler(decay);
odeSolver.setStepSize(dt);
```

The differential equations in the ODE object are then solved by repeatedly invoking the solver's `step` method:

```
odeSolver.step();
```

The solver's `step` method advances the state by `dt`. The state array can be read after every step if the variables are to be plotted or analyzed in some fashion.

We first discuss the details using the Euler method. Subsequently, we will use other numerical methods that are predefined in the OSP numerics package (see Table 7.1). Of course, it is up to you to determine if the chosen numerical method is stable and accurate when applied to a given problem. The important point is that we only need to instantiate a different solver to switch numerical methods.

The OSP numerics package implements the Euler algorithm so that it can be used to solve any system of first-order differential equations. The Euler method takes a variable,  $x$  for example, and

Name	algorithm
Euler	Euler algorithm
Verlet	Verlet algorithm
RK4	Runge-Kutta fourth-order algorithm with fixed step size
RK45	Runge-Kutta fifth-order algorithm with variable step size

Table 7.1: Numerical methods included in the OSP package.

advances it to a new value,  $x + \Delta x$ . The change  $\Delta x$  is calculated using the rate of change of  $x$  with respect to some independent parameter,  $\dot{x} = dx/dt$ . We extend this idea to systems of first-order differential equations by labeling the  $n + 1$  variables as  $x_0, x_1, x_2, \dots, x_n$ . The system of equations can be written in the form:

$$\dot{x}_0 = f_0(x_0, x_1, \dots, x_n) \quad (7.3a)$$

$$\dot{x}_1 = f_1(x_0, x_1, \dots, x_n) \quad (7.3b)$$

...

$$\dot{x}_n = f_n(x_0, x_1, \dots, x_n). \quad (7.3c)$$

Each equation specifies the rate of change of a variable in terms of all the other variables. The variable that is used to take the derivative can be any independent parameter, but it is often the time. By convention, we will always list this variable last.

We first apply this notation to the nuclear decay problem in Chapter 5, and write it as two differential equations:

$$\dot{x}_0 = -kx_0 \quad (7.4a)$$

$$\dot{x}_1 = 1. \quad (7.4b)$$

Equation (7.4a) is equivalent to  $dN/dt = -kN$ ; (7.4b) is the rate of change for the time variable,  $dt/dt = 1$ . That is,  $x_0$  is the number of radioactive nuclei, and  $x_1$  is the time. Using this notation, the Euler algorithm for the  $i$ th step is

$$x_{0,i+1} = x_{0,i} - kx_{0,i} \Delta t \quad (7.5a)$$

$$x_{1,i+1} = x_{1,i} + \Delta t. \quad (7.5b)$$

The change in each variable is determined by evaluating the rate at the beginning of the  $i$ th step,  $f_j(x_{0,i}, x_{1,i}, x_{2,i}, \dots, x_{n,i})$ . If the rates are not constant, which they usually are not, there will be a truncation error in the new calculated values.

We next encapsulate the Euler algorithm in Listing 7.1. The `step()` method gets the current state from the ODE object and then sends this same array back to the ODE object to calculate the rates. This array passing is certainly not necessary for the Euler algorithm. However, higher order algorithms often evaluate the rate several times using intermediate values before a final state is calculated. Hence, the ODE interface requires a `getRate` method that evaluates the rate equations for arbitrary states, not just for the current state.

Listing 7.1: Implementation of the Euler algorithm to be used in conjunction with the ODE interface.

```
// Euler algorithm encapsulated as an object
package org.opensourcephysics.numerics;
public class Euler extends Object implements ODESolver {
    private double stepSize = 0.1;
    private int numberOfEquations = 0;
    private double[] rate;
    private ODE ode;

    public Euler(ODE _ode){
        ode = _ode;
        numberOfEquations = ode.getState().length;
        rate = new double[numberOfEquations];
    }

    public double step(){
        double[] state = ode.getState();
        ode.getRate(state, rate);
        for (int i = 0; i < numberOfEquations; i++) {
            state[i] = state[i] + stepSize*rate[i];
        }
        return stepSize;
    }

    public void setStepSize(double _stepSize){
        stepSize = _stepSize;
    }

    public void initialize (double stepSize){setStepSize(stepSize);}

    public double getStepSize() {return stepSize;}
}

```

Note that the state of the system is represented by an array (`state`), which in the decay example represents two variables. Corresponding to these two variables are two rates that are represented by the `rate` array. It might be tempting to choose the name “rates” for the latter array, but it is more consistent to choose a singular name. For example, we speak about the rate of change of the state of a system. The question of the most appropriate name for an array that represents many variables arises often, and it usually is better to choose the singular rather than the plural name.

The `Euler` class works in conjunction with any class that implements the `ODE` interface. Consider, for example, the radioactive decay problem that was discussed in Section 5.4. The `Decay` class encapsulates this problem in Listing 7.2. The most important method is, of course, `getRate`.

Listing 7.2: The `Decay` class.

```
package org.opensourcephysics.sip.ch7;
```

```

import org.opensourcephysics.numerics.ODE;

public class Decay implements ODE {

    double k;           // decay constant
    double[] state = new double[2]; // [particle number, time]

    public Decay() {
        k = 1;           // decay constant
        state [0] = 1.0; // particle number
        state [1] = 0;   // time
    }

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

    public void getRate(double[] state, double[] rate) {
        rate[0] = -k*state [0]; // dN/dt = -kN
        rate [1] = 1;           // dt/dt = 1
    }
}

```

We now test the decay class using in `DecayApp`, which is shown in Listing 7.3.

Listing 7.3: Listing of `DecayApp`.

```

package org.opensourcephysics.sip.ch7;

import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class DecayApp {
    public static void main(String[] args) {
        PlottingPanel plottingPanel = new PlottingPanel("Time", "Number", "Number versus Time");
        DrawingFrame plottingFrame = new DrawingFrame(plottingPanel);
        DatasetCollection datasetCollection = new DatasetCollection();
        plottingPanel.addDrawable(datasetCollection); // add dataset collection to plotting panel
        Decay decay = new Decay(); // object that contains decay equations
        ODESolver odeSolver = new Euler(decay); // use Euler method to solve decay equations
        odeSolver.setStepSize (0.1);
        for (int i = 0; i < 100; i++) {
            // append time and particle number
            datasetCollection.append(0, decay.state [1], decay.state [0]);
            odeSolver.step ();
        }
        plottingPanel.repaint ();
    }
}

```

Note that for simplicity, we have hard coded the initial number of particles. The decay object

and the ODE solver are created, but the decay object is created first so that the solver obtains a reference to this object. Also note that there is only one state array, and this array is created in the decay object. The state array is retrieved by other objects, such as the ODE solver. This array is used over and over as the solution is carried forward using the `step` method. It would be inefficient to allocate a new rate array at every step.

Another version of the exponential decay program (called `DecayAnimationApp`) that uses the Animation interface is available in the Chapter 7 package. You may wish to use this program to do Problem 7.1.

*Exercise 7.1. Test of Decay.*

- Compile `DecayApp` verify that its output is the same as what was obtained in Problem 5.2.
- Write a Euler-Richardson class for the ODE interface and modify `DecayApp` so that the ODE solver uses the Euler-Richardson algorithm. Is this solver more accurate than the Euler solver?
- Modify `DecayApp` so that the ODE solver is RK4 (see Table 7.1). What are the changes that you need to make? Discuss the advantages (and any disadvantages) of placing the algorithm in a separate class. Is the RK4 solver more accurate than the Euler solver? (The nature of the fourth-order Runge-Kutta algorithm (RK4) is discussed in Appendix 7A.)

### 7.3 Two-Dimensional Trajectories

Computing the trajectory of a particle in two dimensions is now straightforward. The equations of motion can be written as rate equations as follows:

$$\dot{x}_0 = x_1 \quad (7.6a)$$

$$\dot{x}_1 = F_x(x_0, x_1, x_2, x_3, x_4)/m \quad (7.6b)$$

$$\dot{x}_2 = x_3 \quad (7.6c)$$

$$\dot{x}_3 = F_y(x_0, x_1, x_2, x_3, x_4)/m \quad (7.6d)$$

$$\dot{x}_4 = 1. \quad (7.6e)$$

The forces  $F_x$  and  $F_y$  are determined by (7.2a) and (7.2b), respectively. In Exercise 7.2 we modify the `DecayApp` program to solve these equations.

*Problem 7.2. Trajectory of a steel ball*

- Create a class that encapsulates (7.6). Include a method to set the initial conditions using the angle  $\theta_0$  above the horizontal and the initial velocity of  $v_0$ .
- Compute the two-dimensional trajectory of a ball moving in air and plot  $y$  as a function of  $x$ . Use the Euler-Richardson algorithm. As a check on your program, first neglect air resistance so that you can compare your computed results with the exact results. Assume that a ball is thrown from ground level at an angle  $\theta_0$  above the horizontal with an initial velocity of  $v_0 = 15$  m/s. Vary  $\theta_0$  and show that the maximum range occurs at  $\theta_0 = \theta_{\max} = 45^\circ$ . What is  $R_{\max}$ , the maximum range, at this angle? Compare your numerical value to the analytical result  $R_{\max} = v_0^2/g$ .

- c. Suppose that a steel ball is thrown from a height  $h$  at an angle  $\theta_0$  above the horizontal with the same initial speed as in part (b). If you neglect air resistance, do you expect  $\theta_{\max}$  to be larger or smaller than  $45^\circ$ ? Although this problem can be solved analytically, you can determine the numerical value of  $\theta_{\max}$  without changing your program. What is  $\theta_{\max}$  for  $h = 2$  m? By what percent is the range  $R$  changed if  $\theta$  is varied by 2% from  $\theta_{\max}$ ?
- d. Consider the effects of air resistance on the range and optimum angle of a steel ball. For a ball of mass 7 kg and cross-sectional area  $0.01 \text{ m}^2$ , the parameter  $C_2$  is approximately 0.1. What are the units of  $C_2$ ? However, it is convenient to exaggerate the effects of air resistance so that you can more easily determine the qualitative nature of the effects. Compute the optimum angle for  $h = 2$  m,  $v_0 = 30$  m/s, and  $C = k_2/m = 0.1$ , and compare your answer to the value found in part (c). Is  $R$  more or less sensitive to changes in  $\theta_0$  from  $\theta_{\max}$  than in part (b)? Determine the optimum launch angle and the corresponding range for the more realistic value of  $C = 0.001$ . A detailed discussion of the maximum range of the ball has been given by Lichtenberg and Wills (see references).

*Problem 7.3. Coupled motion*

Consider the motion of two identical objects that both start from a height  $h$ . One object is dropped vertically from rest and the other is thrown with a horizontal velocity  $v_0$ . Which object reaches the ground first?

- a. Give reasons for your answer assuming that air resistance can be neglected.
- b. Assume that air resistance cannot be neglected and that the drag force is proportional to  $v^2$ . Give reasons for your anticipated answer for this case. Then perform numerical simulations using, for example,  $C = k_2/m = 0.1$ ,  $h = 10$  m, and  $v_0 = 30$  m/s. Are your qualitative results consistent with your anticipated answer? If they are not, the source of the discrepancy might be an error in your program. Or the discrepancy might be due to your failure to anticipate the effects of the coupling between the vertical and horizontal motion.
- c. Suppose that the drag force is proportional to  $v$  rather than to  $v^2$ . Is your anticipated answer similar to that in part (b)? Do a simulation to test your intuition.

*Problem 7.4. ODE and the Pendulum*

Rewrite `PendulumApp` from Chapter 6 so that it uses the ODE interface. What do you have to do to consider a damped pendulum?

## 7.4 Crossed $\mathbf{E}$ and $\mathbf{B}$ Fields

[xx not finished xx] Coupled equations of motion occur in electrodynamics when a charged particle moves through a magnetic field. Consider a particle of mass  $m$  and charge  $q$  traveling with velocity  $\mathbf{v}$  through an electric field  $\mathbf{E}$  and a magnetic field  $\mathbf{B}$ . The equations of motion can be written in vector form as:

$$m\dot{\mathbf{r}} = \vec{v} \tag{7.7}$$

and

$$\dot{\mathbf{v}} = q\vec{\mathbf{E}} + q\vec{v} \times \vec{\mathbf{B}}. \quad (7.8)$$

If we restrict ourselves to the special case of constant perpendicular fields with the magnetic field in the  $\hat{z}$  direction and the electric field in the  $\hat{y}$  direction, then the net force acts only in the  $x$ - $y$  plane. The rate equations for the velocity components can then be written as

$$m \frac{dv_x}{dt} = qv_y B_z \quad (7.9a)$$

$$m \frac{dv_y}{dt} = qE_y - qv_x B_z. \quad (7.9b)$$

To solve these equations and show an animation, we define a drawable charged particle class as follows:

## 7.5 Spin of Balls in Flight

[xx not finished xx] Of particular interest to sports fans is the curve of balls in flight due to their rotation and the effect of air resistance on the range and speed of baseballs and golf balls.

The equations of motion are

$$\dot{\mathbf{r}} = \mathbf{v} \quad (7.10)$$

and

$$\dot{\mathbf{v}} = \mathbf{g} - k_D v \mathbf{v} + k_L \boldsymbol{\omega} \times \mathbf{v}, \quad (7.11)$$

where  $k_D$  and  $k_L$  are the drag and lift coefficients, respectively. Typical values for a baseball are  $k_D = 0.0037$  and  $k_L = 0.0029$

The rate equations for the velocity components can be written as

$$m \frac{dv_x}{dt} = -k_D v v_x + k_L (\omega_y v_z - \omega_z v_y) \quad (7.12a)$$

$$m \frac{dv_y}{dt} = -k_D v v_y + k_L (\omega_z v_x - \omega_x v_z) \quad (7.12b)$$

$$m \frac{dv_z}{dt} = -k_D v v_z + k_L (\omega_x v_y - \omega_y v_x) - g. \quad (7.12c)$$

## 7.6 Projects

Applications of the ideas and methods that we have discussed for trajectories are important in the physics of clouds. For example, to understand the behavior of falling water droplets, it is necessary to take into account drag resistance as well as droplet growth by condensation and other mechanisms. Because of the variety and complexity of the mechanisms, simulation plays an essential role.

Another area of interest is sports. The trajectory of various shapes through the air such as baseballs, footballs, and javelins can be very involved because the complete description of their motion requires three spatial coordinates, three angles, and six derivatives.

*Project 7.5.* Sports

[xx see AJP articles on the physics of sports. xx]

*Project 7.6.* Comparison of algorithms

- a. Appendix 7A summarizes some of the more commonly used algorithms for the numerical solution of Newton's equations of motion. Compare the Euler, Euler-Richardson, Verlet, and fourth-order Runge Kutta algorithms by computing  $x(t)$ ,  $v(t)$ , and  $E_n$ , where  $E_n$  is the total energy after the  $n$ th step for the Morse potential  $V(x) = e^{-2x} - 2e^{-x}$  considered in Project 6.17. Which one of these algorithms is the best trade-off between speed, accuracy, and simplicity?
- b. Consider a particle of unit mass in the Lennard-Jones potential (see Section ??)

$$V(r) = 4\left[\left(\frac{1}{r}\right)^{12} - \left(\frac{1}{r}\right)^6\right]. \quad (7.13)$$

Consider the motion of a particle in two dimensions. Choose initial conditions such that the particle experiences the steeply repulsive part of the potential and test the various algorithms considered in part (a).

## Appendix 7A: Numerical Integration of Newton's Equation of Motion

We summarize several of the common finite difference methods for the solution of Newton's equations of motion with continuous force functions. The variety of algorithms currently in use is evidence that no single method is superior under all conditions.

To simplify the notation, we consider the motion of a particle in one dimension and write Newton's equations of motion in the form

$$\frac{dv}{dt} = a(t), \quad (7.14a)$$

and

$$\frac{dx}{dt} = v(t), \quad (7.14b)$$

where  $a(t) \equiv a(x(t), v(t), t)$ . The goal of finite difference methods is to determine the values of  $x_{n+1}$  and  $v_{n+1}$  at time  $t_{n+1} = t_n + \Delta t$ . We already have seen that  $\Delta t$  must be chosen so that the integration method generates a stable solution. If the system is conservative,  $\Delta t$  must be sufficiently small so that the total energy is conserved to the desired accuracy.

The nature of many of the integration algorithms can be understood by expanding  $v_{n+1} = v(t_n + \Delta t)$  and  $x_{n+1} = x(t_n + \Delta t)$  in a Taylor series. We write

$$v_{n+1} = v_n + a_n \Delta t + O((\Delta t)^2), \quad (7.15a)$$

and

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + O((\Delta t)^3). \quad (7.15b)$$

The familiar Euler method is equivalent to retaining the  $O(\Delta t)$  terms in (7.15a):

$$v_{n+1} = v_n + a_n \Delta t \quad (7.16a)$$

and

$$x_{n+1} = x_n + v_n \Delta t. \quad (\text{Euler}) \quad (7.16b)$$

Because order  $\Delta t$  terms are retained in (7.16), the local truncation error, the error in one time step, is order  $(\Delta t)^2$ . The global error, the total error over the time of interest, due to the accumulation of errors from step to step is order  $\Delta t$ . This estimate of the global error follows from the fact that the number of steps into which the total time is divided is proportional to  $1/\Delta t$ . Hence, the order of the global error is reduced by a factor of  $1/\Delta t$  relative to the local error. We say that a method is  $n$ th order if its global error is order  $(\Delta t)^n$ . The Euler method is an example of a *first-order* method.

The Euler method is asymmetrical because it advances the solution by a time step  $\Delta t$ , but uses information about the derivative only at the beginning of the interval. We already have found that the accuracy of the Euler method is limited and that frequently its solutions are not stable. We also found that a simple modification of (7.16) yields solutions that are stable for oscillatory systems. For completeness, we repeat the Euler-Cromer algorithm or last-point approximation here:

$$v_{n+1} = v_n + a_n \Delta t, \quad (7.17a)$$

and

$$x_{n+1} = x_n + v_{n+1} \Delta t. \quad (\text{Euler-Cromer}) \quad (7.17b)$$

An obvious way to improve the Euler method is to use the mean velocity during the interval to obtain the new position. The corresponding *midpoint* method can be written as

$$v_{n+1} = v_n + a_n \Delta t, \quad (7.18a)$$

and

$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n) \Delta t. \quad (\text{midpoint}) \quad (7.18b)$$

Note that if we substitute (7.18a) for  $v_{n+1}$  into (7.18b), we obtain

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n \Delta t^2. \quad (7.19)$$

Hence, the midpoint method yields second-order accuracy for the position and first-order accuracy for the velocity. Although the midpoint approximation yields exact results for constant acceleration, it usually does not yield much better results than the Euler method. In fact, both methods are equally poor, because the errors increase with each time step.

A higher order method whose error is bounded is the *half-step* method. In this method the average velocity during an interval is taken to be the velocity in the middle of the interval. The

half-step method can be written as

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + a_n \Delta t, \quad (7.20a)$$

and

$$x_{n+1} = x_n + v_{n+\frac{1}{2}} \Delta t. \quad (\text{half-step}) \quad (7.20b)$$

Note that the half-step method is not self-starting, that is, (7.20a) does not allow us to calculate  $v_{\frac{1}{2}}$ . This problem can be overcome by adopting the Euler algorithm for the first half step:

$$v_{\frac{1}{2}} = v_0 + \frac{1}{2} a_0 \Delta t. \quad (7.20c)$$

Because the half-step method is stable, it is a common textbook method.

The *Euler-Richardson method* was introduced in Chapter 5, but we list it here for completeness.

$$v_m = v_n + \frac{1}{2} a_n \Delta t \quad (7.21a)$$

$$y_m = y_n + \frac{1}{2} v_n \Delta t \quad (7.21b)$$

$$t_m = t_n + \frac{1}{2} \Delta t \quad (7.21c)$$

$$a_m = F(y_m, v_m, t_m)/m, \quad (7.21d)$$

and

$$v_{n+1} = v_n + a_m \Delta t \quad (7.22a)$$

$$y_{n+1} = y_n + v_m \Delta t. \quad (\text{Euler-Richardson}) \quad (7.22b)$$

We will see that the Euler-Richardson algorithm is equivalent to the second-order Runge-Kutta algorithm (see (7.31)).

One of the most common drift-free higher order algorithms is commonly attributed to Verlet. We write the Taylor series expansion for  $x_{n-1}$  in a form similar to (7.15b):

$$x_{n-1} = x_n - v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2. \quad (7.23)$$

If we add the forward and reverse forms, (7.15b) and (7.23) respectively, we obtain

$$x_{n+1} + x_{n-1} = 2x_n + a_n (\Delta t)^2 + O((\Delta t)^4) \quad (7.24)$$

or

$$x_{n+1} = 2x_n - x_{n-1} + a_n (\Delta t)^2. \quad (7.25a)$$

Similarly, the subtraction of the Taylor series for  $x_{n+1}$  and  $x_{n-1}$  yields

$$v_n = \frac{x_{n+1} - x_{n-1}}{2\Delta t}. \quad (\text{original Verlet}) \quad (7.25b)$$

Note that the global error associated with the *Verlet* algorithm (7.25) is third-order for the position and second-order for the velocity. However, the velocity plays no part in the integration of the

equations of motion. In the numerical analysis literature, the Verlet algorithm is known as the “explicit central difference method.”

Because this form of the Verlet algorithm is not self-starting, another algorithm must be used to obtain the first few terms. An additional problem is that the new velocity (7.25b) is found by computing the difference between two quantities of the same order of magnitude. As we discussed in Chapter 5, such an operation results in a loss of numerical precision and may give rise to roundoff errors.

A mathematically equivalent version of the original Verlet algorithm is given by

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 \quad (7.26a)$$

and

$$v_{n+1} = v_n + \frac{1}{2} (a_{n+1} + a_n) \Delta t. \quad (\text{velocity Verlet}) \quad (7.26b)$$

We see that (7.26), known as the *velocity* form of the Verlet algorithm, is self-starting and minimizes roundoff errors. Because we will make no use of (7.25) in the text, we will refer to (7.26) as the Verlet algorithm.

We can derive (7.26) from (7.25) by the following considerations. We first solve (7.25b) for  $x_{n-1}$  and write  $x_{n-1} = x_{n+1} - 2v_n \Delta t$ . If we substitute this expression for  $x_{n-1}$  into (7.25a) and solve for  $x_{n+1}$ , we find the form (7.26a). Then we use (7.25b) to write  $v_{n+1}$  as:

$$v_{n+1} = \frac{x_{n+2} - x_n}{2\Delta t}, \quad (7.27)$$

and use (7.25a) to obtain  $x_{n+2} = 2x_{n+1} - x_n + a_{n+1}(\Delta t)^2$ . If we substitute this form for  $x_{n+2}$  into (7.27), we obtain

$$v_{n+1} = \frac{x_{n+1} - x_n}{\Delta t} + \frac{1}{2} a_{n+1} \Delta t. \quad (7.28)$$

Finally, we use (7.26a) for  $x_{n+1}$  to eliminate  $x_{n+1} - x_n$  from (7.28); after some algebra we obtain the desired result (7.26b).

Another useful algorithm that avoids the roundoff errors of the original Verlet algorithm is due to Beeman and Schofield. We write the *Beeman* algorithm in the form:

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{6} (4a_n - a_{n-1}) (\Delta t)^2 \quad (7.29a)$$

and

$$v_{n+1} = v_n + \frac{1}{6} (2a_{n+1} + 5a_n - a_{n-1}) \Delta t. \quad (7.29b)$$

Note that (7.29) does not calculate particle trajectories more accurately than the Verlet algorithm. Its advantage is that in general it does a better job of maintaining energy conservation. However, the Beeman algorithm is not self-starting.

The most common finite difference method for solving ordinary differential equations is the *Runge-Kutta* method. To explain the method, we first consider the solution of the first-order

differential equation

$$\frac{dx}{dt} = f(x, t). \quad (7.30)$$

The *second-order* Runge-Kutta solution of (7.30) can be written using standard notation as:

$$k_1 = f(x_n, t_n)\Delta t \quad (7.31a)$$

$$k_2 = f\left(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t \quad (7.31b)$$

$$x_{n+1} = x_n + k_2 + O((\Delta t)^3). \quad (7.31c)$$

The interpretation of (7.31) is as follows. The Euler method assumes that the slope  $f(x_n, t_n)$  at  $(x_n, t_n)$  can be used to extrapolate to the next step, that is,  $x_{n+1} = x_n + f(x_n, t_n)\Delta t$ . A plausible way of making a better estimate of the slope is to use the Euler method to extrapolate to the midpoint of the interval and then to use the midpoint slope across the full width of the interval. Hence, the Runge-Kutta estimate for the slope is  $f(x^*, t_n + \frac{1}{2}\Delta t)$ , where  $x^* = x_n + \frac{1}{2}f(x_n, t_n)\Delta t$  (see (7.31b)).

The application of the second-order Runge-Kutta method to Newton's equation of motion (7.14) yields

$$k_{1v} = a_n(x_n, v_n, t_n)\Delta t \quad (7.32a)$$

$$k_{1x} = v_n\Delta t \quad (7.32b)$$

$$k_{2v} = a\left(x_n + \frac{k_{1x}}{2}, v_n + \frac{k_{1v}}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t \quad (7.32c)$$

$$k_{2x} = \left(v_n + \frac{k_{1v}}{2}\right)\Delta t, \quad (7.32d)$$

and

$$v_{n+1} = v_n + k_{2v} \quad (7.32e)$$

$$x_{n+1} = x_n + k_{2x} \quad (\text{second-order Runge Kutta}) \quad (7.32f)$$

In the *fourth-order* Runge-Kutta algorithm, the derivative is computed at the beginning of the time interval, in two different ways at the middle of the interval, and again at the end of the interval. The two estimates of the derivative at the middle of the interval are given twice the weight of the other two estimates. The algorithm for the solution of (7.30) can be written in standard notation as

$$k_1 = f(x_n, t_n)\Delta t \quad (7.33a)$$

$$k_2 = f\left(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t \quad (7.33b)$$

$$k_3 = f\left(x_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t \quad (7.33c)$$

$$k_4 = f(x_n + k_3, t_n + \Delta t)\Delta t, \quad (7.33d)$$

and

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \quad (7.34)$$

The application of the fourth-order Runge-Kutta method to Newton's equation of motion (7.14) yields

$$k_{1v} = a(x_n, v_n, t_n)\Delta t \quad (7.35a)$$

$$k_{1x} = v_n\Delta t \quad (7.35b)$$

$$k_{2v} = a\left(x_n + \frac{k_{1x}}{2}, v_n + \frac{k_{1v}}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t \quad (7.35c)$$

$$k_{2x} = \left(v_n + \frac{k_{1v}}{2}\right)\Delta t \quad (7.35d)$$

$$k_{3v} = a\left(x_n + \frac{k_{2x}}{2}, v_n + \frac{k_{2v}}{2}, t_n + \frac{\Delta t}{2}\right)\Delta t \quad (7.35e)$$

$$k_{3x} = \left(v_n + \frac{k_{2v}}{2}\right)\Delta t \quad (7.35f)$$

$$k_{4v} = a(x_n + k_{3x}, v_n + k_{3v}, t + \Delta t) \quad (7.35g)$$

$$k_{4x} = (v_n + k_{3x})\Delta t, \quad (7.35h)$$

and

$$v_{n+1} = v_n + \frac{1}{6}(k_{1v} + 2k_{2v} + 2k_{3v} + k_{4v}) \quad (7.36a)$$

$$x_{n+1} = x_n + \frac{1}{6}(k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x}). \quad (\text{fourth-order Runge-Kutta}) \quad (7.36b)$$

Because Runge-Kutta methods are self-starting, they are frequently used to obtain the first few iterations for an algorithm that is not self-starting.

Our last example is the *predictor-corrector* method. The idea is to first *predict* the value of the new position:

$$x_p = x_{n-1} + 2v_n\Delta t. \quad (\text{predictor}) \quad (7.37)$$

The predicted value of the position allows us to predict the acceleration  $a_p$ . Then using  $a_p$ , we obtain the *corrected* values of  $v_{n+1}$  and  $x_{n+1}$ :

$$v_{n+1} = v_n + \frac{1}{2}(a_p + a_n)\Delta t \quad (7.38a)$$

$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n)\Delta t. \quad (\text{corrected}) \quad (7.38b)$$

The corrected values of  $x_{n+1}$  and  $v_{n+1}$  are used to obtain a new predicted value of  $a_{n+1}$ , and hence a new predicted value of  $v_{n+1}$  and  $x_{n+1}$ . This process is repeated until the predicted and corrected values of  $x_{n+1}$  differ by less than the desired value. Note that the predictor-corrector method is not self-starting.

As we have emphasized, there is no single algorithm for solving Newton's equations of motion that is superior under all conditions. It is usually a good idea to start with a simple algorithm, and then to try a higher order algorithm to see if any real improvement is obtained.

## References and Suggestions for Further Reading

F. S. Acton, *Numerical Methods That Work*, The Mathematical Association of America (1990), Chapter 5.

Paul L. DeVries, *A First Course in Computational Physics*, John Wiley & Sons (1994).

Tao Pang, *Computational Physics*, Cambridge University Press (1997).

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, second edition, Cambridge University Press (1992). Chapter 16 discusses the integration of ordinary differential equations.