Problem 3.1 Comparing Euler algorithms

(a) Write a class that extends `Particle` and models a simple harmonic oscillator for which $F = -kx$. For simplicity, choose units such that $k = 1$ and $m = 1$. Determine the numerical error in the position of the simple harmonic oscillator after the particle has evolved for several cycles. Is the original Euler algorithm stable for this system? What happens if you run for longer times?

(b) Repeat part (a) using the Euler–Cromer algorithm. Does this algorithm work better? If so, in what way?

(c) Modify your program so that it computes the total energy, $E_{\text{sho}} = v^2/2 + x^2/2$. How well is the total energy conserved for the two algorithms? Also consider the quantity $\tilde{E} = E_{\text{sho}} + (\Delta t/2)xp$. What is the behavior of this quantity for the Euler–Comer algorithm? ■

Perhaps it has occurred to you that it would be better to compute the velocity at the middle of the interval rather than at the beginning or at the end. The *Euler–Richardson* algorithm is based on this idea. This algorithm is particularly useful for velocity-dependent forces, but does as well as other simple algorithms for forces that do not depend on the velocity. The algorithm consists of using the Euler algorithm to find the intermediate position y_{mid} and velocity v_{mid} at a time $t_{\text{mid}} = t + \Delta t/2$. We then compute the force, $F(y_{\text{mid}}, v_{\text{mid}}, t_{\text{mid}})$ and the acceleration a_{mid} at $t = t_{\text{mid}}$. The new position y_{n+1} and velocity v_{n+1} at time t_{n+1} are found using v_{mid} and a_{mid} and the Euler algorithm. We summarize the Euler–Richardson algorithm as

$$a_n = F(y_n, v_n, t_n)/m \tag{3.3a}$$

$$v_{\text{mid}} = v_n + \frac{1}{2}a_n\Delta t \tag{3.3b}$$

$$y_{\text{mid}} = y_n + \frac{1}{2}v_n\Delta t \tag{3.3c}$$

$$a_{\text{mid}} = F(y_{\text{mid}}, v_{\text{mid}}, t + \Delta t/2)/m, \tag{3.3d}$$

and

$$v_{n+1} = v_n + a_{\text{mid}}\Delta t \tag{3.4a}$$

$$y_{n+1} = y_n + v_{\text{mid}}\Delta t \quad \text{(Euler–Richardson algorithm).} \tag{3.4b}$$

Although we need to do twice as many computations per time step, the Euler–Richardson algorithm is much faster than the Euler algorithm because we can make the time step larger and still obtain better accuracy than with either the Euler or Euler–Cromer algorithms. A derivation of the Euler–Richardson algorithm is given in Appendix 3A.

Exercise 3.2 The Euler–Richardson algorithm

(a) Extend `FallingParticle` in Listing 2.6 to a new class that implements the Euler–Richardson algorithm. All you need to do is write a new step method.

(b) Using $\Delta t = 0.08, 0.04, 0.02$, and 0.01, determine the error in the computed position when the particle hits the ground. How do your results compare with the Euler algorithm? How does the error in the velocity depend on Δt for each algorithm?

(c) Repeat part (b) for the simple harmonic oscillator and compute the error after several cycles. ■

As we gain more experience simulating various physical systems, we will learn that no single algorithm for solving Newton's equations of motion numerically is superior under all conditions.

The Open Source Physics library includes classes that can be used to solve systems of coupled first-order differential equations using different algorithms. To understand how to use this library, we first discuss *interfaces* and then *arrays*.

3.2 ■ INTERFACES

We have seen how to combine data and methods into a class. A class definition *encapsulates* this information in one place, thereby simplifying the task of the programmer who needs to modify the class and the user who needs to understand or use the class.

Another tool for data abstraction is known as an *interface*. An interface specifies methods that an object performs, but does not implement these methods. In other words, an interface describes the behavior or functionality of any class that implements it. Because an interface is not tied to a given class, any class can *implement* any particular interface as long as it defines all the methods specified by the interface. An important reason for interfaces is that a class can inherit from only one superclass, but it can implement more than one interface.

An example of an interface is the `Function` interface in the numerics package:

```
public interface Function {
   public double evaluate (double x);
}
```

The interface contains one method, `evaluate`, with one argument but no body. Notice that the definition uses the keyword `interface` rather then the keyword `class`.

We can define a class that encapsulates a quadratic polynomial as follows:

```
public class QuadraticPolynomial implements Function {
   double a,b,c;

   public QuadraticPolynomial(double a, double b, double c) {
      this.a = a;
      this.b = b;
      this.c = c;
   }

   public double evaluate (double x) {
      return a*x*x + b*x + c;
   }
}
```

Quadratic polynomials can now be instantiated and used as needed.

```
Function f = new QuadraticPolynomial(1,0,2);
for(int x = 0; x < 10; x++) {
```