Chapter 3

Simulating Particle Motion

We discuss several numerical methods needed to simulate the motion of particles using Newton's laws and introduce *interfaces*, an important Java construct that makes it possible for unrelated objects to declare that they perform the same methods.

3.1 Modified Euler algorithms

To motivate the need for a general differential equation solver, we discuss why the simple Euler algorithm is insufficient for many problems. The Euler algorithm assumes that the velocity and acceleration do not change significantly during the time step Δt. Thus, to achieve an acceptable numerical solution, the time step Δt must be chosen to be sufficiently small. However, if we make Δt too small, we run into several problems. As we do more and more iterations, the round-off error due to the finite precision of any floating point number will accumulate and eventually the numerical results will become inaccurate. Also, the greater the number of iterations, the greater the computer time required for the program to finish. In addition to these problems, the Euler algorithm is unstable for many systems, which means that the errors accumulate exponentially, and thus the numerical solution becomes inaccurate very quickly. For these reasons more accurate and stable numerical algorithms are necessary.

To illustrate why we need algorithms other than the simple Euler algorithm, we make a very simple change in the Euler algorithm and write

$$v(t + \Delta t) = v(t) + a(t)\Delta t \tag{3.1a}$$

$$y(t + \Delta t) = y(t) + v(t + \Delta t)\Delta t, \tag{3.1b}$$

where a is the acceleration. The only difference between this algorithm and the simple Euler algorithm,

$$v(t+\Delta t) = v(t) + a(t)\Delta t \tag{3.2a}$$
$$y(t+\Delta t) = y(t) + v(t)\Delta t, \tag{3.2b}$$

is that the computed velocity at the end of the interval, $v(t+\Delta t)$, is used to compute the new position, $y(t+\Delta t)$ in (3.1b). As we found in Problem 2.12 and will see in more detail in Problem 3.1, this modified Euler algorithm is significantly better for oscillating systems. We refer to this algorithm as the Euler-Cromer algorithm.

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:

$$a_n = F(y_n, v_n, t_n)/m \tag{3.3a}$$
$$v_{\text{mid}} = v_n + \frac{1}{2}a_n\Delta t \tag{3.3b}$$
$$y_{\text{mid}} = y_n + \frac{1}{2}v_n\Delta t \tag{3.3c}$$
$$a_{\text{mid}} = F(y_{\text{mid}}, v_{\text{mid}}, t + \frac{1}{2}\Delta t)/m, \tag{3.3d}$$

and

$$v_{n+1} = v_n + a_{\text{mid}}\Delta t \tag{3.4a}$$
$$y_{n+1} = y_n + v_{\text{mid}}\Delta t. \qquad \text{(Euler-Richardson algorithm)} \tag{3.4b}$$

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 3.

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. Determine the error in the computed position when the particle hits the ground using $\Delta t = 0.08$, 0.04, 0.02, and 0.01. 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++) {
   System.out.println("x = " + x + "  f(x)" + f.evaluate(x));
}
```

By using the `Function` interface, we can write methods that use this mathematical abstraction. For example, we can program a simple plot as follows:

```
public void plotFunction(Function f, double xmin, double xmax) {
   PlotFrame frame = new PlotFrame("x","y", "Function");
   double n = 100;          // number of points in plot
   double x = xmin, dx = (xmax - xmin)/(n-1);
   for (int i = 0; i < 100; i++) {
      frame.append(0, x, f.evaluate());
      x += dx;
   }
   frame.show();    // display frame on screen
}
```

We also can compute a numerical derivative based on the definition of the derivative found in calculus textbooks.

```
public double derivative(Function f, double x, double dx) {
   return (f.evaluate(x+dx) - f.evaluate(x))/dx;
}
```

This way of approximating a derivative is not optimum, but that is not the point here. (A better approximation is given in Problem 3.8.) The important point is that the interface enables us to define the abstract concept $y = f(x)$ and to write code that uses this abstraction.

Exercise 3.3. Function interface

a. Define a class that encapsulates the function $f(u) = ae^{-bu^2}$.

b. Write a test program that plots $f(u)$ with $b = 1$ and $b = 4$. Choose $a = 1$ for simplicity.

c. Write a test program that plots the derivatives of the functions used in part (b) without using the analytic expression for the derivative.

Although interfaces are very useful for developing large scale software projects, you will not need to define interfaces to do the problems in this book. However, you will use several interfaces, including the `Function` interface, that are defined in the Open Source Physics library. We describe two of the more important interfaces in the following sections.

3.3 Drawing

An interface that we will use often is the `Drawable` interface:

```java
package org.opensourcephysics.display;
import java.awt.*;

public interface Drawable {
   public void draw (DrawingPanel panel, Graphics g);
}
```

Notice that this interface contains only one method, `draw`. Objects that implement this interface are rendered in a `DrawingPanel` after they have been added to a `DisplayFrame`. As we saw in Chapter 2, a `DisplayFrame` is made of components including a title bar, menu, and buttons for minimizing and closing the frame. The `DisplayFrame` contains a `DrawingPanel` on which graphical output will be displayed. The `Graphics` class contains methods for drawing simple geometrical objects such as lines, rectangles, and ovals on the panel. In Listing 3.1 we define a class that draws a rectangle using pixel-based coordinates.

Listing 3.1: `PixelRectangle`.

```java
package org.opensourcephysics.sip.ch03;
import java.awt.*; // uses Abstract Window Toolkit (awt)
import org.opensourcephysics.display.*;

public class PixelRectangle implements Drawable {
   int left, top;      // position of rectangle in pixels
   int width, height; // size of rectangle in pixels

   PixelRectangle(int left, int top, int width, int height) {
      this.left = left; // location of left edge
      this.top = top;   // location of top edge
      this.width = width;
      this.height = height;
   }

   public void draw(DrawingPanel panel, Graphics g) {
      // this method implements the Drawable interface
      g.setColor(Color.RED);                  // set drawing color to red
      g.fillRect(left, top, width, height); // draws rectangle
   }
}
```

In method `draw` we used `fillRect`, a primitive method in the `Graphics` class. This method draws a filled rectangle using pixel coordinates with the origin at the top left corner of the panel.

To use `PixelRectangle`, we instantiate an object and add it to a `DisplayFrame` as shown in Listing 3.2.

Listing 3.2: Listing of `DrawingApp`.

```java
package org.opensourcephysics.sip.ch03;
```

```
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;

public class DrawingApp extends AbstractCalculation {
   DisplayFrame frame = new DisplayFrame("x", "y", "Graphics");

   public DrawingApp() {
      frame.setPreferredMinMax(0, 10, 0, 10);
   }

   public void calculate() {
      // gets rectangle location
      int left = control.getInt("xleft");
      int top = control.getInt("ytop");
      // gets rectangle dimensions
      int width = control.getInt("width");
      int height = control.getInt("height");
      Drawable rectangle = new PixelRectangle(left, top, width, height);
      frame.addDrawable(rectangle);
      // frame is automatically rendered after Calculate button is pressed
   }

   public void reset() {
      frame.clearDrawables();               // removes drawables added by the user
      control.setValue("xleft", 60);        // sets default input values
      control.setValue("ytop", 70);
      control.setValue("width", 100);
      control.setValue("height", 150);
   }

   public static void main(String[] args) { // creates a calculation control structure using
      CalculationControl.createApp(new DrawingApp());
   }
}
```

Note that multiple rectangles are drawn in the order that they are added to the drawing panel. Rectangles or portion of rectangles may be hidden because they are outside the drawing panel.

Although it is possible to use pixel-based drawing methods to produce visualizations, creating even a simple graph in such an environment would require much tedious programming. The `DrawingPanel` object passed to the `draw` method simplifies this task by defining a system of *world coordinates* that enable us to specify the location and size of various objects in physical units rather than pixels. In the `WorldRectangle` class in Listing 3.3, methods from the `DrawingPanel` class are used to convert pixel coordinates to world coordinates. The range of the world coordinates in the horizontal and vertical directions is defined in the `frame.setPreferredMinMax` method in `DrawingApp`. (This method is not needed if pixel coordinates are used.)

Listing 3.3: `WorldRectangle` illustrates the use of world coordinates.

```
package org.opensourcephysics.sip.ch03;
```

```
import java.awt.*;
import org.opensourcephysics.display.*;

public class WorldRectangle implements Drawable {
   double left, top;      // position of rectangle in world coordinates
   double width, height; // size of rectangle in world units

   public WorldRectangle(double left, double top, double width, double height) {
      this.left = left; // location of left edge
      this.top = top;   // location of top edge
      this.width = width;
      this.height = height;
   }

   public void draw(DrawingPanel panel, Graphics g) {
      // This method implements the Drawable interface
      g.setColor(Color.RED); // set drawing color to red
      // convert from world to pixel coordinates
      int leftPixels = panel.xToPix(left);
      int topPixels = panel.yToPix(top);
      int widthPixels = (int) (panel.getXPixPerUnit()*width);
      int heightPixels = (int) (panel.getYPixPerUnit()*height);
      g.fillRect(leftPixels, topPixels, widthPixels, heightPixels); // draws rectangle
   }
}
```

Exercise 3.4. Simple graphics

a. Run `DrawingApp` and test how the different inputs change the size and location of the rectangle. Note that the pixel coordinates that are obtained from the control window are not the same as the world coordinates that are displayed.

b. Read the documentation at `<java.sun.com/reference/api/>` for the `Graphics` class, and modify the `WorldRectangle` class to draw lines, filled ovals, and strings of characters. Also play with different colors.

c. Modify `DrawingApp` to use the `WorldRectangle` class and repeat part (a). Note that the coordinates that are displayed and the inputs are now consistent.

d. Define and test a `TextMessage` class to display text messages in a drawing panel using world coordinates to position the text. In the `draw` method use the syntax `g.drawString("string to draw",x,y)`, where `(x,y)` are the pixel coordinates.

Although simple geometric shapes such as circles and rectangles often are all that is needed to visualize many physical models, Java provides a drawing environment based on the Java 2D Application Programming Interface (API) which can render arbitrary geometric shapes, images, and text using composition and matrix-based transformations. We will use a subset of these features to define the `DrawableShape` and `InteractiveShape` classes in the display package of

Open Source Physics, which we will introduce in Chapter 10. (See also the Open Source Physics User's Guide.)

So far we have created rectangles using two different classes. Each implementation of a `Drawable` rectangle defined a different `draw` method. Notice that in the display frame's definition of `addDrawable` in `DrawingApp`, the argument is specified to be the interface `Drawable` rather than a specific class. Any class that implements `Drawable` can be an argument of `addDrawable`. Without the interface construct we would need to write an `addDrawable` method for each type of class.

3.4 Specifying the state of a system using arrays

Imagine writing the code for the numerical solution of the motion of three particles in three dimensions using the Euler-Richardson algorithm. The resulting code would be tedious to write. In addition, for each problem we would need to write and debug new code to implement the numerical algorithm. The complications become worse for better algorithms, most of which are algebraically more complex. Moreover, the numerical solution of simple first-order differential equations is a well developed part of numerical analysis, and thus there is little reason to worry about the details of these algorithms, now that we know how they work. In Section 3.5 we will introduce an interface for solving the differential equations associated with Newton's equations of motion. Before we do so we discuss a few features of arrays that we will need.

As we discussed on page 39, ordered lists of data are most easily stored in arrays. For example, if we have an array variable named `x`, then we can access its first element as `x[0]`, its second element as `x[1]`, etc. All elements must be of the same data type, but they can be just about anything: primitive data types such as doubles or integers, objects, or even other arrays. The following statements show how arrays of primitive data types are defined and instantiated:

```
double[] x;                          // x defined to be an array of doubles
double x[];                          // same meaning as double [] x
x = new double[32];                  // x array created with 32 elements
double[] y = new double[32];         // y array defined and created in one step
int[] num = new int[100];            // array of 100 integers
double [] x,y                        // preferred notation
double x[], y[]                      // same meaning as double [] x,y
double[][] sigma = new double[3][3]; // array of doubles specified by two indices
double[] row = sigma[0];             // reference to first row of sigma array
```

We will adopt the syntax `double[] x` instead of `double x[]`. The array index starts at zero and the largest index is one less than the number of elements. Note that Java supports multiple array indices by creating arrays of arrays. Although `sigma[0][0]` refers to a single value of type `double` in the `sigma` object, we can refer to an entire row of values in the `sigma` object using the syntax `sigma[i]`.

As shown in Chapter 2, arrays can contain objects such as bouncing balls.

```
BouncingBall [] ball = new BouncingBall[2];   // array of two BouncingBall objects
ball[0] = new BouncingBall(0,10.0,0,5.0);     // creates first ball
ball[1] = new BouncingBall(0,-13.0,0,7.0);    // creates second ball
```

The first statement allocates an array of `BouncingBall` objects, each of which is initialized to null. We need to create each object in the array using the `new` operator.

The numerical solution of an ordinary differential equation (frequently called an ODE) begins by expressing the equation as several first-order differential equations. If the highest derivative in the ODE is order n (for example, $d^n x/dt^n$), then it can be shown that the ODE can be written equivalently as n first-order differential equations. For example, Newton's equation of motion is a second-order differential equation and can be written as two first-order differential equations for the position and velocity in each spatial dimension. For example, in one dimension we can write

$$\frac{dy}{dt} = v(t) \tag{3.5a}$$

$$\frac{dv}{dt} = a(t) = F(t)/m. \tag{3.5b}$$

If we have more than one particle, there are additional first-order differential equations for each particle. It is convenient to have a standard way of handling all these cases.

Let us assume that each differential equation is of the form:

$$\frac{dx_i}{dt} = r_i(x_0, x_i, x_2, \cdots x_{n-1}, t), \tag{3.6}$$

where x_i is a dynamical variable such as a position or a velocity. The rate function r_i can depend on any of the dynamical variables including the time t. We will store the values of the dynamical variables in the `state` array and the values of the corresponding rates in the `rate` array. In the following we show some examples:

```
// one particle in one dimension:
state[0]  // stores x
state[1]  // stores v
state[2]  // stores t (time)
// one particle in two dimensions:
state[0]  // stores x
state[1]  // stores vx
state[2]  // stores y
state[3]  // stores vy
state[4]  //stores t
// two particles in one dimension:
state[0]  // stores x1
state[1]  // stores v1
state[2]  // stores x2
state[3]  // stores v2
state[4]  // stores t
```

Although the Euler algorithm does not assume any special ordering of the state variables, we adopt the convention that a velocity rate follows every position rate in the state array so that we can efficiently code the more sophisticated numerical algorithms that we discuss in Appendix 3 and in later chapters. To solve problems for which the rate contains an explicit time dependence, such as a driven harmonic oscillator (see Section 5.4), we store the time variable in the last element of the state array. Thus, for one particle in one dimension, the time is stored in `state[2]`. In this way we can treat all dynamical variables on an equal footing.

Because arrays can be arguments of methods, we need to understand how Java passes variables from the class that calls a method to the method being called. Consider the following method:

```java
public void example(int r, int s[]) {
   r = 20;
   s[0] = 20;
}
```

What do you expect the output of the following statements to be?

```java
int x = 10;
int[] y = {10};   // array of one element initialized to y[0] = 10
example(x, y);
System.out.println("x = " + x + " y[0] = " + y[0]);
```

The answer is that the output will be `x = 10`, `y[0] = 20`. Java parameters are "passed-by-value," which means that the values are copied. The method cannot modify the value of the `x` variable because the method received only a copy of its value. In contrast, when an object or an array is in a method's parameter list, Java passes a copy of the reference to the object or the array. The method can use the reference to read or modify the data in the array or object. For this reason the `step` method of the ODE solvers, discussed in Section 3.6, does not need to explicitly return an updated `state` array, but implicity changes the contents of the state array.

Exercise 3.5. Pass by value

As another example of how Java handles primitive variables differently from arrays and objects, consider the statements

```java
int x = 10;
int y = x;
x = 20;
```

What is `y`? Next consider

```java
int[] x = {10};   // declare an array of one element initialized to the value 10
int[] y = x;
x[0] = 20;
```

What is `y[0]`?

We are now ready to discuss the classes and interfaces from the Open Source Physics library for solving ordinary differential equations.

3.5 The ODE interface

To introduce the ODE interface, we again consider the equations of motion for a falling particle. We use a state array ordered as $s = (y, v, t)$, so that the dynamical equations can be written as:

$$\dot{s_0} = s_1 \tag{3.7a}$$

$$\dot{s_1} = -g \tag{3.7b}$$

$$\dot{s_2} = 1. \tag{3.7c}$$

The `ODE` interface enables us to encapsulate (3.7) in a class. The interface contains two methods, `getState` and `getRate`, as shown in Listing 3.4.

Listing 3.4: The ODE interface.

```java
package org.opensourcephysics.numerics;
public interface ODE {
   public double[] getState();

   public void getRate(double[] state, double[] rate);
}
```

The `getState` method returns the state array $(s_0, s_1, \ldots s_n)$. The `getRate` method evaluates the derivatives using the given state array and stores the result in the rate array, $(\dot{s}_0, \dot{s}_1, \ldots \dot{s}_n)$.

An example of a Java class that implements the `ODE` interface for a falling particle is shown in Listing 3.5.

Listing 3.5: Example of the implementation of the `ODE` interface for a falling particle.

```java
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.numerics.*;

public class FallingParticleODE implements ODE {
   final static double g = 9.8;
   double[] state = new double[3];

   public FallingParticleODE(double y, double v) {
      state[0] = y;
      state[1] = v;
      state[2] = 0;              // initial time
   }

   public double[] getState() { // required to implement ODE interface
      return state;
   }

   public void getRate(double[] state, double[] rate) {
      rate[0] = state[1]; // rate of change of y is v
      rate[1] = -g;
      rate[2] = 1;        // rate of change of time is 1
   }
}
```

3.6 The ODESolver interface

There are many possible numerical algorithms for advancing a system of first-order ODEs from an initial state to a final state. The Open Source Physics library defines ODE solvers such as `Euler` and `EulerRichardson`, as well as `RK4`, a fourth-order algorithm that is discussed in Appendix 3.

You can write additional classes for other algorithms if they are needed. Each of these classes implements the `ODESolver` interface, which is defined in Listing 3.6.

Listing 3.6: The ODE solver interface. Note the four methods that must be defined.

```java
package org.opensourcephysics.numerics;
public interface ODESolver {
   public void initialize(double stepSize);

   public double step();

   public void setStepSize(double stepSize);

   public double getStepSize();
}
```

A system of first-order differential equations is now solved by creating an object that implements a particular algorithm and repeatedly invoking the `step` method for that solver class. The argument for the solver class constructor must be a class that implements the `ODE` interface. As an example of the use of `ODESolver`, we again consider the dynamics of a falling particle.

Listing 3.7: A falling particle program that uses an `ODESolver`.

```java
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.numerics.*;

public class FallingParticleODEApp extends AbstractCalculation {
   public void calculate() {
      // gets initial conditions
      double y0 = control.getDouble("Initial y");
      double v0 = control.getDouble("Initial v");
      // creates ball with initial conditions
      FallingParticleODE ball = new FallingParticleODE(y0, v0);
      // creates ODE solver
      ODESolver solver = new Euler(ball); // note how particular algorithm is chosen
      // sets time step dt in the solver
      solver.setStepSize(control.getDouble("dt"));
      while(ball.state[0]>0) {
         solver.step();
      }
      control.println("final time = "+ball.state[2]);
      control.println("y = "+ball.state[0]+" v = "+ball.state[1]);
   }

   public void reset() {
      control.setValue("Initial y", 10);     // sets default input values
      control.setValue("Initial v", 0);
      control.setValue("dt", 0.01);
   }
```

```
  public static void main(String[] args) { // creates a calculation control structure for t
    CalculationControl.createApp(new FallingParticleODEApp());
  }
}
```

The ODE classes are located in the numerics package, and thus we need to import this package as done in the third statement of `FallingParticleODEApp`. We declare and instantiate the variables `ball` and `solver` in the `calculate` method. Note that `ball`, an instance of `FallingParticleODE`, is the argument of the `Euler` constructor. The object `ball` can be an argument because `FallingParticleODE` implements the `ODE` interface.

It would be a good idea to look at the source code of the ODE `Euler` class in the numerics package. The `Euler` class gets the state of the system using `getState` and then sends this state to `getRate` which stores the rates in the `rate` array. The `state` array is then modified using the `rate` array in the Euler algorithm. You don't need to know the details, but you can read the `step` method of the various classes that implement `ODESolver` if you are interested in how the different algorithms are programmed.

Because `FallingParticleODE` appears to be more complicated than `FallingParticle`, you might ask what we have gained. One answer is that it is now much easier to use a different numerical algorithm. The only modification we need to make is to change the statement

```
ODESolver solver = new Euler(ball);
```

to, for example,

```
ODESolver solver = new EulerRichardson(ball);
```

We have separated the physics (in this case a freely falling particle) from the implementation of the numerical method.

Exercise 3.6. ODE solvers

Run `FallingParticleODEApp` and compare your results with our previous implementation of the Euler algorithm in `FallingParticleApp`. How easy is it to use a different algorithm?

3.7 Effects of Drag Resistance

We have introduced most of the programming concepts that we will use in the remainder of this text. If you are new to programming, you will likely feel a bit confused at this point by all the new concepts and syntax. However, it is not necessary to understand all the details to continue and begin to write your own programs. A prototypical simulation program is given in Listings 3.8 and 3.9. These classes simulate a projectile on the surface of the Earth with no air friction, including a plot of position versus time and an animation of a projectile moving through the air. In the following we discuss more realistic models that can be simulated by modifying the projectile classes.

Listing 3.8: A simple projectile simulation that is useful as a template for other simulations.

```
package org.opensourcephysics.sip.ch03;
import java.awt.*;
import org.opensourcephysics.display.*;
import org.opensourcephysics.numerics.*;

public class Projectile implements Drawable, ODE {
   static final double g = 9.8;
   double[] state = new double[5]; // {x,vx,y,vy,t}
   int pixRadius = 6;              // pixel radius for drawing of projectile
   EulerRichardson odeSolver = new EulerRichardson(this);

   public void setStepSize(double dt) {
      odeSolver.setStepSize(dt);
   }

   public void step() {
      odeSolver.step(); // do one time step using selected algorithm
   }

   public void setState(double x, double vx, double y, double vy) {
      state[0] = x;
      state[1] = vx;
      state[2] = y;
      state[3] = vy;
      state[4] = 0;
   }

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

   public void getRate(double[] state, double[] rate) {
      rate[0] = state[1]; // rate of change of x
      rate[1] = 0;        // rate of change of vx
      rate[2] = state[3]; // rate of change of y
      rate[3] = -g;       // rate of change of vy
      rate[4] = 1;        // dt/dt = 1
   }

   public void draw(DrawingPanel drawingPanel, Graphics g) {
      int xpix = drawingPanel.xToPix(state[0]);
      int ypix = drawingPanel.yToPix(state[2]);
      g.setColor(Color.red);
      g.fillOval(xpix-pixRadius, ypix-pixRadius, 2*pixRadius, 2*pixRadius);
      g.setColor(Color.green);
      int xmin = drawingPanel.xToPix(-100);
      int xmax = drawingPanel.xToPix(100);
      int y0 = drawingPanel.yToPix(0);
      g.drawLine(xmin, y0, xmax, y0); // draw a line to represent the ground
   }
}
```

Listing 3.9: A target class for projectile motion simulation.

```
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class ProjectileApp extends AbstractSimulation {
   PlotFrame plotFrame = new PlotFrame("Time", "x,y", "Position versus time");
   Projectile projectile = new Projectile();
   PlotFrame animationFrame = new PlotFrame("x", "y", "Trajectory");

   public ProjectileApp() {
      animationFrame.addDrawable(projectile);
      plotFrame.setXYColumnNames(0, "t", "x");
      plotFrame.setXYColumnNames(1, "t", "y");
   }

   public void initialize() {
      double dt = control.getDouble("dt");
      double x = control.getDouble("initial x");
      double vx = control.getDouble("initial vx");
      double y = control.getDouble("initial y");
      double vy = control.getDouble("initial vy");
      projectile.setState(x, vx, y, vy);
      projectile.setStepSize(dt);
      double size = (vx*vx+vy*vy)/10; // estimate of size needed for display
      animationFrame.setPreferredMinMax(-1, size, -1, size);
   }

   public void doStep() {
      plotFrame.append(0, projectile.state[4], projectile.state[0]);      // x vs time data
      plotFrame.append(1, projectile.state[4], projectile.state[2]);      // y vs time data
      animationFrame.append(0, projectile.state[0], projectile.state[2]); // trajectory data
      projectile.step();                                                  // advance the sta
   }

   public void reset() {
      control.setValue("initial x", 0);
      control.setValue("initial vx", 10);
      control.setValue("initial y", 0);
      control.setValue("initial vy", 10);
      control.setValue("dt", 0.01);
      enableStepsPerDisplay(true);
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new ProjectileApp());
   }
}
```

The analytical solution for free fall near the Earth's surface, (2.4), is well known and thus finding a numerical solution is useful only as an introduction to numerical methods. It is not difficult to think of more realistic models of motion near the Earth's surface for which the equations of motion do not have simple analytical solutions. For example, if we take into account the variation of the Earth's gravitational field with the distance from the center of the Earth, then the force on a particle is not constant. According to Newton's law of gravitation, the force due to the Earth on a particle of mass m is given by

$$F = \frac{GMm}{(R+y)^2} = \frac{GMm}{R^2(1+y/R)^2} = mg\big(1 - 2\frac{y}{R} + \cdots\big), \tag{3.8}$$

where y is measured from the Earth's surface, R is the radius of the Earth, M is the mass of the Earth, G is the gravitational constant, and $g = GM/R^2$.

Problem 3.7. Position-dependent force

Extend `FallingParticleODE` to simulate the fall of a particle with the position-dependent force law (3.8). Assume that a particle is dropped from a height h with zero initial velocity and compute its impact velocity (speed) when it hits the ground at $y = 0$. Determine the value of h for which the impact velocity differs by one percent from its value with a constant acceleration $g = 9.8\,\text{m/s}^2$. Take $R = 6.37 \times 10^6$ m. Make sure that the one percent difference is due to the physics of the force law and not the accuracy of your algorithm.

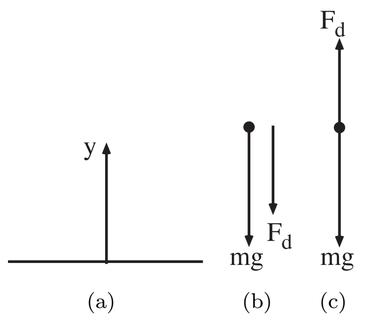

Figure 3.1: (a) Coordinate system with y measured positive upward from the ground. (b) The force diagram for downward motion. (c) The force diagram for upward motion.

For particles near the Earth's surface, a more important modification is to include the drag force due to air resistance. The direction of the drag force $F_d(v)$ is opposite to the velocity of the particle (see Figure 3.1). For a falling body $F_d(v)$ is upward as shown in Figure 3.1(b). Hence, the total force F on the falling body can be expressed as

$$F = -mg + F_d. \tag{3.9}$$

The velocity dependence of $F_d(v)$ is known theoretically in the limit of very low speeds for small objects. In general, it is necessary to determine the velocity dependence of $F_d(v)$ empirically over a limited range of velocities. One way to obtain the form of $F_d(v)$ is to measure y as a function of t and then compute $v(t)$ by calculating the numerical derivative of $y(t)$. Similarly we can use $v(t)$ to compute $a(t)$ numerically. From this information it is possible in principle to find the acceleration as a function of v and to extract $F_d(v)$ from (3.9). However, this procedure introduces errors (see Problem 3.8b) because the accuracy of the derivatives will be less than the accuracy of the measured position. An alternative is to reverse the procedure, that is, assume an explicit form for the v dependence of $F_d(v)$, and use it to solve for $y(t)$. If the calculated values of $y(t)$ are consistent with the experimental values of $y(t)$, then the assumed v dependence of $F_d(v)$ is justified empirically.

The two common assumed forms of the velocity dependence of $F_d(v)$ are

$$F_{1,d}(v) = C_1 v, \tag{3.10a}$$

and

$$F_{2,d}(v) = C_2 v^2, \tag{3.10b}$$

where the parameters C_1 and C_2 depend on the properties of the medium and the shape of the object. In general, (3.10a) and (3.10b) are useful *phenomenological* expressions that yield approximate results for $F_d(v)$ over a limited range of v.

Because $F_d(v)$ increases as v increases, there is a limiting or *terminal velocity* (speed) at which the net force on a falling object is zero. This terminal speed can be found from (3.9) and (3.10) by setting $F_d = mg$ and is given by

$$v_{1,t} = \frac{mg}{C_1}, \qquad \text{(linear drag)} \tag{3.11a}$$

$$v_{2,t} = \Big(\frac{mg}{C_2}\Big)^{1/2}, \qquad \text{(quadratic drag)} \tag{3.11b}$$

for the linear and quadratic cases, respectively. It often is convenient to express velocities in terms of the terminal velocity. We can use (3.10) and (3.11) to write F_d in the linear and quadratic cases as

$$F_{1,d} = C_1 v_{1,t}\Big(\frac{v}{v_{1,t}}\Big) = mg\frac{v}{v_{1,t}}, \tag{3.12a}$$

$$F_{2,d} = C_2 {v_{2,t}}^2\Big(\frac{v}{v_{2,t}}\Big)^2 = mg\Big(\frac{v}{v_{2,t}}\Big)^2. \tag{3.12b}$$

Hence, we can write the net force (per unit mass) on a falling object in the convenient forms

$$F_1(v)/m = -g\Big(1 - \frac{v}{v_{1,t}}\Big), \tag{3.13a}$$

$$F_2(v)/m = -g\Big(1 - \frac{v^2}{{v_{2,t}}^2}\Big). \tag{3.13b}$$

To determine if the effects of air resistance are important during the fall of ordinary objects, consider the fall of a pebble of mass $m = 10^{-2}\,$kg. To a good approximation, the drag force is proportional to v^2. For a spherical pebble of radius 0.01 m, C_2 is found empirically to be approximately $10^{-2}\,$kg/m. From (3.11b) we find the terminal velocity to be about 30 m/s. Because this speed would be achieved by a freely falling body in a vertical fall of approximately 50 m in a

t (s)	position (m)	t (s)	position (m)	t (s)	position (m)
0.2055	0.4188	0.4280	0.3609	0.6498	0.2497
0.2302	0.4164	0.4526	0.3505	0.6744	0.2337
0.2550	0.4128	0.4773	0.3400	0.6990	0.2175
0.2797	0.4082	0.5020	0.3297	0.7236	0.2008
0.3045	0.4026	0.5266	0.3181	0.7482	0.1846
0.3292	0.3958	0.5513	0.3051	0.7728	0.1696
0.3539	0.3878	0.5759	0.2913	0.7974	0.1566
0.3786	0.3802	0.6005	0.2788	0.8220	0.1393
0.4033	0.3708	0.6252	0.2667	0.8466	0.1263

Table 3.1: Results for the vertical fall of a coffee filter. Note that the initial time is not zero. The time difference is ≈ 0.0247. This data also is available in the `falling.txt` file in the ch03 package.

time of about 3 s, we expect that the effects of air resistance would be appreciable for comparable times and distances.

Data often is stored in text files, and it is convenient to be able to read this data into a program for analysis. The `ResourceLoader` class in the Open Source Physics `tools` package makes reading these files easy. This class can read many different data types including images and sound. An example of how to use the `ResourceLoader` class to read string data is given in `DataLoaderApp`.

Listing 3.10: Example of the use of the `ResourceLoader` class to read data into a program.

```java
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.tools.*;

public class DataLoaderApp {
   public static void main(String[] args) {
      String fileName = "falling.txt"; //  reads from directory where DataLoaderApp is locate
      // gets the data file
      Resource res = ResourceLoader.getResource(fileName, DataLoaderApp.class);
      String data = res.getString();
      String[] lines = data.split("\n"); // split string on newline character
      // extract x-y data from every line
      for(int i = 0, n = lines.length;i<n;i++) {
         if(lines[i].trim().startsWith("//")) {
            continue;                                    // skip comment lines
         }
         String[] numbers = lines[i].trim().split("\\s"); // split on any white space
         System.out.print("t = "+numbers[0]);
         System.out.println("  y = "+numbers[1]);
      }
   }
}
```

Problem 3.8. The fall of a coffee filter

a. Use the empirical data for the height $y(t)$ of a coffee filter in the `falling.txt` data file to

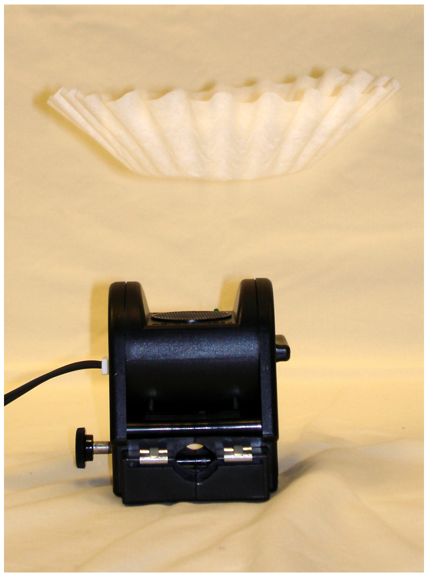

Figure 3.2: A falling coffee filter does not fall with constant acceleration due to the effects of air resistance. The motion sensor below the filter is connected to a computer, which records position data and stores it in a text file.

determine the velocity $v(t)$ using the central difference approximation given by

$$v(t) \approx \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t}. \qquad \text{(central difference approximation)} \tag{3.14}$$

Show that if we write the acceleration as $a(t) \approx [v(t+\Delta t) - v(t)]/\Delta t$ and use the backward difference approximation for the velocity,

$$v(t) \approx \frac{y(t) - y(t-\Delta t)}{\Delta t}, \qquad \text{(backward difference approximation)} \tag{3.15}$$

we can express the acceleration as

$$a(t) \approx \frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{(\Delta t)^2}. \tag{3.16}$$

Use (3.16) to determine the acceleration.

b. Determine the terminal velocity from the data given in the `falling.txt` file. This determination is difficult, in part because the terminal velocity has not been reached during the time that the fall of the coffee filter was observed. Use your approximate results for $v(t)$ and $a(t)$ to plot a as a function of v and, if possible, determine the nature of the velocity dependence of a. Discuss the accuracy of your results for the acceleration.

c. Choose one of the numerical algorithms that we have discussed and write a class that encapsulates this algorithm for the motion of a particle with quadratic drag resistance.

d. Choose the terminal velocity as an input parameter, and take as your first guess for the terminal velocity the value you found in part (b). Make sure that your computed results for the height of the particle, do not depend on Δt to the necessary accuracy. Compare your plot of the computed values of $y(t)$ for different choices of the terminal velocity with the empirical values of $y(t)$ in `falling.txt`.

e. Repeat parts (c) and (d) assuming linear drag resistance. What are the qualitative differences between the two computed forms of $y(t)$ for the same terminal velocity?

f. Visually determine which form of the drag force yields the best overall fit to the data. If the fit is not perfect, what is your criteria for which fit is better? Is it better to match your results to the experimental data at early times or at later times? Or did you adopt another criterion? What can you conclude about the velocity-dependence of the drag resistance on a coffee filter?

Problem 3.9. Effect of air resistance on the ascent and descent of a pebble

a. Verify the claim made in Section 3.7 that the effects of air resistance on a falling pebble can be appreciable. Compute the speed at which a pebble reaches the ground if it is dropped from rest at a height of 50 m. Compare this speed to that of a freely falling object under the same conditions. Assume that the drag force is proportional to v^2 and that the terminal velocity is 30 m/s.

b. Suppose a pebble is thrown vertically upward with an initial velocity v_0. In the absence of air resistance, we know that the maximum height reached by the pebble is $v_0^2/2g$, its velocity upon return to the Earth equals v_0, the time of ascent equals the time of descent, and the total time in the air is $2v_0/g$. Before doing a simulation, give a simple qualitative explanation of how you think these quantities will be affected by air resistance. In particular, how will the time of ascent compare with the time of descent?

c. Do a simulation to determine if your qualitative answers in part (b) are correct. Assume that the drag force is proportional to v^2. Choose the coordinate system shown in Figure 3.1 with y positive upward. What is the net force for $v > 0$ and $v < 0$? We can characterize the magnitude of the drag force by a terminal velocity even if the motion of the pebble is upward and even if the pebble never attains this velocity. Choose the terminal velocity $v_t = 30$ m/s, corresponding to a drag coefficient of $C_2 \approx 0.01089$. It is a good idea to choose an initial velocity that allows the pebble to remain in the air for a time sufficiently long so that the effect of the drag force is appreciable. A reasonable choice is $v(t = 0) = 50$ m/s. You might find it convenient to express the drag force in the form $F_d \propto -$`v*Math.abs(v)`. One way to determine the maximum height of the pebble is to use the statement

```
if (v*vold < 0) {
   control.println("maximum height = " + y);
}
```

where `v` $= v_{n+1}$ and `vold` $= v_n$. Why is this criterion preferable to other criteria that you might imagine using?

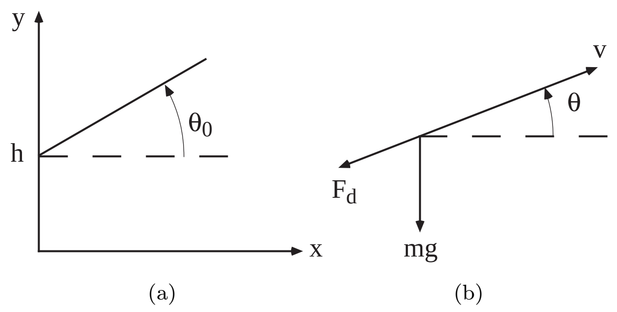

Figure 3.3: (a) A ball is thrown from a height h at an launch angle θ_0 measured with respect to the horizontal. The initial velocity is $\mathbf{v}_0$. (b) The gravitational and drag forces on a particle.

3.8 Two-Dimensional 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 θ_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 θ_0 above the horizontal (see Figure 3.33.3(a)). 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 3.33.3(b)). Newton's equations of motion for the x and y components of the motion can be written as

$$m\frac{dv_x}{dt} = -F_d\cos\theta \tag{3.17a}$$

$$m\frac{dv_y}{dt} = -mg - F_d\sin\theta. \tag{3.17b}$$

For example, let us maximize the range of a round steel ball of radius 4 cm. A reasonable assumption for a steel ball of this size and typical speed is that $F_d = C_2v^2$. Because $v_x = v\cos\theta$ and $v_y = v\sin\theta$, we can rewrite (3.17) as

$$m\frac{dv_x}{dt} = -C_2vv_x \tag{3.18a}$$

$$m\frac{dv_y}{dt} = -mg - C - 2vv_y. \tag{3.18b}$$

Note that $-C_2vv_x$ and $-C_2vv_y$ are the x and y components of the drag force $-C_2v^2$. Because (3.18a) and (3.18b) 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 a falling body without reference to the horizontal component, that is, the motion in the x and y direction is *coupled.*

Problem 3.10. Trajectory of a steel ball

a. Use `Projectile` and `ProjectileApp` to compute the two-dimensional trajectory of a ball moving in air without air friction, and plot y as a function of x. Compare your computed results with the exact results. For example, assume that a ball is thrown from ground level at an angle θ_0 above the horizontal with an initial velocity of $v_0 = 15\,\text{m/s}$. Vary θ_0 and show that the maximum range occurs at $\theta_0 = \theta_{\max} = 45°$. What is $R_{\max}$, the maximum range, at this angle? Compare your numerical result to the analytical result $R_{\max} = v_0^2/g$.

b. Suppose that a steel ball is thrown from a height h at an angle θ_0 above the horizontal with the same initial speed as in part (a). If you neglect air resistance, do you expect $\theta_{\max}$ to be larger or smaller than 45°? What is $\theta_{\max}$ for $h = 2\,\text{m}$? By what percent is the range R changed if θ is varied by 2% from $\theta_{\max}$?

c. 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 $C_2 \approx 0.1$. What are the units of C_2? It is convenient to exaggerate the effects of air resistance so that you can more easily determine the qualitative nature of the effects. Hence, compute the optimum angle for $h = 2\,\text{m}$, $v_0 = 30\,\text{m/s}$, and $C_2/m = 0.1$, and compare your answer to the value found in part (b). Is R more or less sensitive to changes in θ_0 from $\theta_{\max}$ than in part (b)? Determine the optimum launch angle and the corresponding range for the more realistic value of $C_2 = 0.1$. A detailed discussion of the maximum range of the ball has been given by Lichtenberg and Wills.

Problem 3.11. Comparing the motion of two objects

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, $C_2/m = 0.1$, $h = 10\,\text{m}$, and $v_0 = 30\,\text{m/s}$. 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 numerical simulation to test your intuition.

3.9 Decay processes

The power of mathematics when applied to physics comes in part from the fact that seemingly unrelated problems frequently have the same mathematical formulation. Hence, if we can solve one problem, we can solve other problems that might appear to be unrelated. For example, the growth of bacteria, the cooling of a cup of hot water, the charging of a capacitor in a RC circuit, and nuclear decay all can be formulated in terms of equivalent differential equations.

Consider a large number of radioactive nuclei. Although the number of nuclei is discrete, we often may treat this number as a continuous variable because the number of nuclei is very large. In this case the law of radioactive decay is that the rate of decay is proportional to the number of nuclei. Thus we can write

$$\frac{dN}{dt} = -\lambda N, \tag{3.19}$$

where N is the number of nuclei and λ is the decay constant. Of course, we do not need to use a computer to solve this decay equation, and the analytical solution is

$$N(t) = N_0 e^{-\lambda t}, \tag{3.20}$$

where N_0 is the initial number of particles. The quantity λ in (3.19) or (3.20) has dimensions of inverse time.

Problem 3.12. Single nuclear species decay

a. Write a class that solves and plots the nuclear decay problem. Input the decay constant, λ, from the control window. For $\lambda = 1$ and $\Delta t = 0.01$, compute the difference between the analytical result and the result of the Euler algorithm for $N(t)/N(0)$ at $t = 1$ and $t = 2$. Assume that time is measured in seconds.

b. A common time unit for radioactive decay is the half-life, $T_{1/2}$, the time it takes for one-half of the original nuclei to decay. Another natural time scale is the time, τ, it takes for $1/e$ of the original nuclei to decay. Use your modified program to verify that $T_{1/2} = \ln 2/\lambda$. How long does it take for $1/e$ of the original nuclei to decay? How is $T_{1/2}$ related to τ?

c. Because it is awkward to treat very large or very small numbers on a computer, it is convenient to choose units so that the computed values of the variables are not too far from unity. Determine the decay constant λ in units of s^{-1} for $^{238}\mathrm{U} \to {}^{234}\mathrm{Th}$ if the half-life is 4.5×10^9 years. What units and time step would be appropriate for the numerical solution of (3.19)? How would these values change if the particle being modeled were a muon with a half-life of $2.2 \times 10^{-6}\,\mathrm{s}$?

d. Modify your program so that the time t is expressed in terms of the half-life. That is, at $t = 1$ one half of the particles would have decayed and at $t = 2$, one quarter of the particles would have decayed. Use your program to determine the time for 1000 atoms of $^{238}\mathrm{U}$ to decay to 20% of their original number. What would be the corresponding time for muons?

Multiple nuclear decays produce systems of first-order differential equations. Problem 3.13 asks you to model such a system using the techniques similar to those that we have already used.

Problem 3.13. Multiple nuclear decays

a. $^{76}\mathrm{Kr}$ decays to $^{76}\mathrm{Br}$ via electron capture with a half-life of 14.8 h, and $^{76}\mathrm{Br}$ decays to $^{76}\mathrm{Se}$ via electron capture and positron emission with a half-life of 16.1 h. In this case there are two half-lives, and it is convenient to measure time in units of the smallest half-life. Write a program to compute the time dependence of the amount of $^{76}\mathrm{Kr}$ and $^{76}\mathrm{Se}$ over an interval of one week. Assume that the sample initially contains 1 gm of pure $^{76}\mathrm{Kr}$.

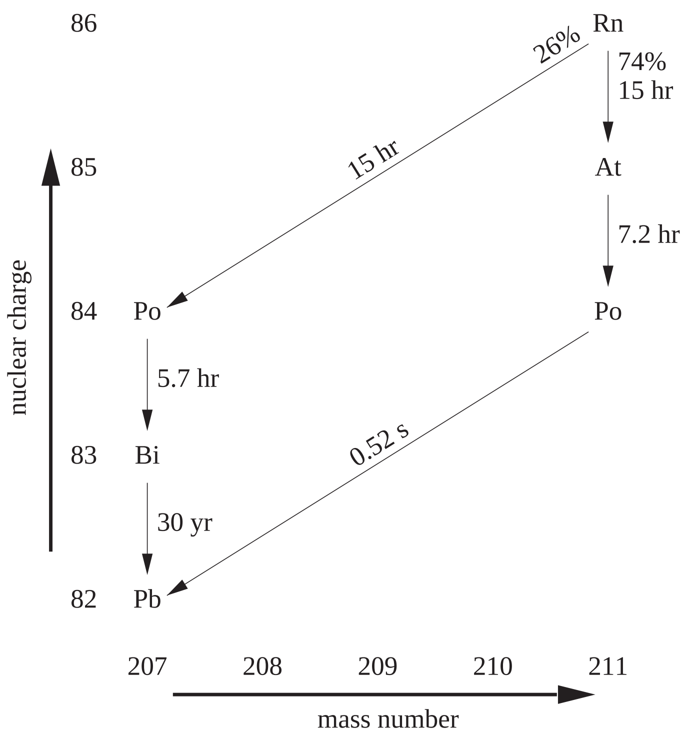

Figure 3.4: The decay scheme of ^{211}Rn. Note that ^{211}Rn decays via two branches, and the final product is the stable isotope ^{207}Pb. All vertical transitions are by electron capture, and all diagonal transitions are by alpha decay. The times represent half-lives.

b. ^{28}Mn decays via beta emission to ^{28}Al with a half-life of 21 h, and ^{28}Al decays by positron emission to ^{28}Si with a half-life of 2.31 min. If we were to use minutes as the unit of time, our program would have to do many iterations before we would see a significant decay of the ^{28}Mn. What simplifying assumption can you make to speed up the computation?

c. ^{211}Rn decays via two branches as shown in Figure 3.4. Make any necessary approximations and compute the amount of each isotope as a function of time, assuming that the sample initially consists of 1 μg of ^{211}Rn.

Problem 3.14. Cooling of a cup of coffee

The nature of the energy transfer from the hot water in a cup of coffee to the surrounding air is complicated and in general involves the mechanisms of convection, radiation, evaporation, and conduction. However, if the temperature difference between the water and its surroundings is not too large, the rate of change of the temperature of the water may be assumed to be proportional to the temperature difference. We can formulate this statement more precisely in terms of a differential equation:

$$\frac{dT}{dt} = -r\,(T - T_s), \tag{3.21}$$

where T is the temperature of the water, T_s is the temperature of its surroundings, and r is the cooling constant. The minus sign in (3.21) implies that if $T > T_s$, the temperature of the water will decrease with time. The value of the cooling constant r depends on the heat transfer mechanism, the contact area with the surroundings, and the thermal properties of the water. The relation (3.21) is sometimes known as Newton's law of cooling, even though the relation is only approximate, and Newton did not express the rate of cooling in this form.

a. Write a program that computes the numerical solution of (3.21). Test your program by choosing the initial temperature $T_0 = 100°\text{C}$, $T_s = 0°\text{C}$, $r = 1$, and $\Delta t = 0.1$.

b. Model the cooling of a cup of coffee by choosing $r = 0.03$. What are the units of r? Plot the temperature T as a function of the time using $T_0 = 87\,°\text{C}$ and $T_s = 17\,°\text{C}$. Make sure that your value of Δt is sufficiently small so that it does not affect your results. What is the appropriate unit of time in this case?

c. Suppose that the initial temperature of a cup of coffee is 87°C, but the coffee can be sipped comfortably only when its temperature is $\leq 75°\text{C}$. Assume that the addition of cream cools the coffee by 5°C. If you are in a hurry and want to wait the shortest possible time, should the cream be added first and the coffee be allowed to cool, or should you wait until the coffee has cooled to 80°C before adding the cream? Use your program to "simulate" these two cases. Choose $r = 0.03$ and $T_s = 17°\text{C}$. What is the appropriate unit of time in this case? Assume that the value of r does not change when the cream is added.

3.10 *Visualizing Three-Dimensional Motion

The world in which we live is three-dimensional (3D), and it sometimes is necessary to visualize phenomena in three dimensions. There are several 3D visualization packages available, including *Java3D* developed by Sun Microsystems and *Java OpenGL* (JOGL), which has been optimized for a variety of graphics hardware. Because we want a three-dimensional visualization framework designed for physics simulations, we have developed our own API.[1]

The Open Source Physics 3D drawing framework is defined in subpackages in the `display3d` package and provides a high level of abstraction for rendering three-dimensional objects. These 3D drawable objects implement the `Element` interface in the `core` package, which enables their position, size, and appearance to be controlled. Elements can be grouped with other elements, can change their visibility, and respond to mouse actions. Listing 3.11 shows that it is not much more difficult to define and manipulate a three-dimensional model than a two-dimensional model. The most significant change is that the program instantiates a `Display3DFrame` and adds `Element` objects such as spheres and boxes to this frame.

Listing 3.11: A three-dimensional bouncing ball created using the Open Source Physics `display3D.simple3d` package.

```java
package org.opensourcephysics.sip.ch03;
import java.awt.*;
```

[1] A framework consists of several classes and an API that does a particular task. In general, these classes are in different packages.

```
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.Display3DFrame;
import org.opensourcephysics.display3d.simple3d.*;
import org.opensourcephysics.display3d.core.Resolution;

public class Ball3DApp extends AbstractSimulation {
   Display3DFrame frame = new Display3DFrame("3D Ball");
   Element ball = new ElementEllipsoid();
   double time = 0, dt = 0.1;
   double vz = 0;

   public Ball3DApp() {
      frame.setPreferredMinMax(-5.0, 5.0, -5.0, 5.0, 0.0, 10.0);
      ball.setXYZ(0, 0, 9);
      ball.setSizeXYZ(1, 1, 1); // ball displayed in 3D as a planar ellipse of size (dx,dy,d
      frame.addElement(ball);
      Element box = new ElementBox();
      box.setXYZ(0, 0, 0);
      box.setSizeXYZ(4, 4, 1);
      box.getStyle().setFillColor(Color.RED);
      // divide sides of box into smaller rectangles
      box.getStyle().setResolution(new Resolution(5, 5, 2));
      frame.addElement(box);
      frame.setMessage("time = "+ControlUtils.f2(time));
   }

   protected void doStep() {
      time += 0.1;
      double z = ball.getZ()+vz*dt-4.9*dt*dt;
      vz -= 9.8*dt;
      if((vz<0)&&(z<1)) {
         vz = -vz;
      }
      ball.setZ(z);
      frame.setMessage("time = "+ControlUtils.f2(time));
   }

   public static void main(String[] args) {
      SimulationControl.createApp(new Ball3DApp());
   }
}
```

Note that the 3D drawing API is similar to the 2D drawing API described in Section 3.3. The `setPreferredMinMax` method, for example, has a variant that accepts up to six double parameters. You can set the size and location of objects in three dimensions before or after they are added to the frame.

Although the `Display3DFrame` is designed for three-dimensional visualizations, it also can show two-dimensional projections. For example, we can project onto the yz-plane by invoking

```
frame.setDisplayMode(VisualizationHints.DISPLAY_PLANAR_YZ);
```

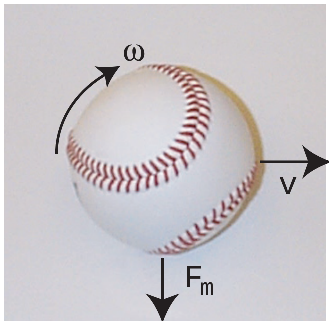

Figure 3.5: The Magnus force on a spinning ball pushes a ball with topspin down.

Projections onto various planes are available at runtime using the frame's menu. The full capabilities of Open Source Physics 3D are discussed in the Open Source Physics User's Guide.

We will require only a small subset of the methods of the Open Source Physics 3D framework to create the three-dimensional visualizations in this book and will introduce the necessary objects as needed. Readers may wish to run the demonstration programs in the ch03 directory to obtain an overview of its drawing capabilities.

Of particular interest to baseball fans is the curve of balls in flight due to their rotation. This force was first investigated in 1850 by G. Magnus and the curvature of the trajectories of spinning objects is now known as the *Magnus effect*. It can be explained qualitatively by observing that the speed of the ball's surface relative to the air is different on opposite edges of the ball. If the drag force has the form $F_{\rm drag} \sim v^2$, then the unbalanced force due to the difference in the velocity on opposite sides of the ball due to its rotation is given by

$$F_{\rm magnus} \sim v\Delta v. \tag{3.22}$$

We can express the velocity difference in terms of the ball's angular velocity and radius and write

$$F_{\rm magnus} \sim vr\omega. \tag{3.23}$$

The direction of the Magnus force is perpendicular to both the velocity and the rotation axis. For example, if we observe a ball moving to the right and rotating clockwise (that is, with topspin), then the velocity of the ball's surface relative to the air at the top, $v+\omega r$, is higher than the velocity at the bottom, $v-\omega r$. Because the larger velocity will produce a larger force, the Magnus effect will contribute a force in the downward direction. These considerations suggest that the Magnus force can be expressed as a vector product:

$$F_{\rm magnus}/m = C_M(\boldsymbol{\omega} \times \mathbf{v}), \tag{3.24}$$

where m is the mass of the ball. The constant, C_M, depends on the radius of the ball, the viscosity of air, and other factors such as the orientation of the stitching. We will assume that the ball is rotating fast enough so that it can be modeled using an average value. (If the ball does not rotate, the pitcher has thrown a knuckleball.) The total force on the baseball is given by

$$\mathbf{F}/m = \mathbf{g} - C_D|\mathbf{v}|\mathbf{v} + C_M(\boldsymbol{\omega} \times \mathbf{v}). \tag{3.25}$$

Equation (3.25) leads to the following rates for the velocity components:

$$\frac{dv_x}{dt} = -C_D v v_x + C_M(\omega_y v_z - \omega_z v_y) \tag{3.26a}$$

$$\frac{dv_y}{dt} = -C_D v v_y + C_M(\omega_z v_x - \omega_x v_z) \tag{3.26b}$$

$$\frac{dv_z}{dt} = -C_D v v_z + C_M(\omega_x v_y - \omega_y v_x) - g, \tag{3.26c}$$

where we will assume that $\boldsymbol{\omega}$ is a constant. The rate for each of the three position variables is the corresponding velocity. Typical parameter values for a 149 gram baseball are $C_D = 6 \times 10^{-3}$ and $C_M = 4 \times 10^{-4}$. See the book by Adair for a more complete discussion.

Problem 3.15. Curveballs

a. Create a class that implements (3.26). Assume that the initial ball is released at $z = 1.8\,\mathrm{m}$ above and $x = 18\,\mathrm{m}$ from home plate. Set the initial angle above the horizontal and the initial speed using the constructor.

b. Write a program that plots the vertical and horizontal deflection of the baseball as it travels toward home plate. First set the drag and Magnus forces to zero and test your program using analytical results for a 40 m/s fast ball. What initial angle is required for the pitch to pass over home plate at a height of 1.5 m?

c. Add the drag force with $C_D = 6 \times 10^{-3}$. What initial angle is required for this pitch to be a strike assuming that the other initial conditions are unchanged? Plot the vertical deflection with and without drag for comparison.

d. Add topspin to the pitch using a typical spin of $\omega_y = 200\,\mathrm{rad/s}$ and $C_M = 4 \times 10^{-4}$. How much does topspin change the height of the ball as it passes over the plate? What about backspin?

e. How much does a 35 m/s curve ball deflect if it is pitched with an initial spin of 200 rad/s?

Problem 3.16. Visualizing baseball trajectories in three dimensions

Add a 3D visualization of the baseball's trajectory to Problem 3.15 using `ElementTrail` to display the path of the ball. The following code fragment shows how a trail is created and used.

```java
ElementTrail trail = new ElementTrail();
trail.setMaximumPoints(500);
trail.getStyle().setLineColor(java.awt.Color.RED);
// frame3D is an OSP3DFrame
frame3D.addElement(trail);
// points are added to a trail to show a trajectory
trail.addPoint(x,y,z);  // adds a point to the trace
```

Coupled three-dimensional equations of motion occur in electrodynamics when a charged particle travels through electric and magnetic fields. The equation of motion can be written in vector form as:

$$m\dot{\mathbf{v}} = q\mathbf{E} + q(\mathbf{v} \times \mathbf{B}), \tag{3.27}$$

where m is the mass of the particle, q is the charge, and $\mathbf{E}$ and $\mathbf{B}$ represent the electric and magnetic fields, respectively. For the special case of a constant magnetic field, the trajectory of a charged particle is a spiral along the field lines with a cyclotron orbit whose period of revolution is $2\pi m/qB$. The addition of an electric field changes this motion dramatically.

The rates for the velocity components of a charged particle using units such that $m = q = 1$ are

$$\frac{dv_x}{dt} = E_x + v_y B_z - v_z B_y \tag{3.28a}$$

$$\frac{dv_y}{dt} = E_y + v_z B_x - v_x B_z \tag{3.28b}$$

$$\frac{dv_z}{dt} = E_z + v_x B_y - v_y B_x. \tag{3.28c}$$

The rate for each of the three position variables is again the corresponding velocity.

Problem 3.17. Motion in electric and magnetic fields

a. Write a program to simulate the two-dimensional motion of a charged particle in a constant electric and magnetic field with the magnetic field in the $\hat{z}$ direction and the electric field in the $\hat{y}$ direction. Assume that the initial velocity is in the x-y plane.

b. Why does the trajectory in part (a) remain in the x-y plane?

c. In what direction does the charge particle drift if there is an electric field in the x direction and a magnetic field in the z direction if it starts at rest from the origin? What type of curve does the charged particle follow?

d. Create a three-dimensional simulation of the trajectory of a particle in constant electric and magnetic fields. Verify that a charged particle undergoes spiral motion in a constant magnetic field and zero electric field. Predict the trajectory if an electric field is added and compare the results of the simulation to your prediction. Consider electric fields that are parallel to and perpendicular to the magnetic field.

Although the trajectory of a charged particle in constant electric and magnetic fields can be solved analytically, the trajectories in the presence of dipole fields cannot. A magnetic dipole with dipole moment $\mathbf{p} = |p|\hat{p}$ produces the following magnetic field:

$$\mathbf{B} = \frac{\mu_0 m}{4\pi\epsilon_0 r^3}[3\hat{p}\cdot\hat{r})\hat{r} - \hat{p}]. \tag{3.29}$$

(The distinction between the symbol p for the dipole moment and p for momentum should be clear from the context.)

***Problem 3.18.** Motion in a magnetic dipole field

Model the Earth's Van Allen radiation belt using the following formula for the dipole field:

$$\mathbf{B} = B_0(\frac{R_E}{R})^3[3\hat{p}\cdot\hat{r})\hat{r} - \hat{p}], \tag{3.30}$$

where R_E is the radius of the Earth and the magnetic field at the equator is $B_0 = 3.5 \times 10^{-5}$ tesla. Note that a 1 MeV electron at 2 Earth radii travels in very tight spirals with a cyclotron period that is much smaller than the travel time between the north and south poles. Better visual results can be obtained by raising the electron energies by a factor of ~ 1000. Use classical dynamics, but include the relativistic dependence of the mass on the particle speed.

3.11 Levels of Simulation

So far we have considered models in which the microscopic complexity of the system of interest has been simplified considerably. Consider for example, the motion of a pebble falling through the air. First we reduced the complexity by representing the pebble as a particle with no internal structure. Then we reduced the number of degrees of freedom even more by representing the collisions of the pebble with the many molecules in the air by a velocity-dependent friction term. The resultant phenomenological model is a fairly accurate representation of realistic physical systems. However, what we gain in simplicity, we lose in range of applicability.

In a more detailed model, the individual physical processes would be represented microscopically. For example, we could imagine doing a simulation in which the effects of the air are represented by a fluid of particles that collide with one another and with the falling body. How accurately do we need to represent the potential energy of interaction between the fluid particles? Clearly the level of detail that is needed depends on the accuracy of the corresponding experimental data and the type of information in which we are interested. For example, we do not need to take into account the influence of the moon on a pebble falling near the Earth's surface. And the level of detail that we can simulate depends in part on the available computer resources.

The terms *simulation* and *modeling* are frequently used interchangeably, and their precise meaning is not important. Many practitioners might say that so far we have solved several mathematical models numerically and have not yet done a simulation. Beginning with the next chapter, we will be able to say that we are doing simulations. The difference is that our models will represent physical systems in more detail, and we will give more attention to what physical quantities we should measure. In other words, our simulations will become more analogous to laboratory experiments.

Appendix 3A: 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 number and 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

$$\frac{dv}{dt} = a(t), \tag{3.31a}$$

$$\frac{dx}{dt} = v(t), \tag{3.31b}$$

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 Δt must be chosen so that the integration method generates a stable solution. If the system is conservative, Δt 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

$$v_{n+1} = v_n + a_n \Delta t + O\big((\Delta t)^2\big), \tag{3.32a}$$

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 + O\big((\Delta t)^3\big). \tag{3.32b}$$

The familiar Euler algorithm is equivalent to retaining the $O(\Delta t)$ terms in (3.32):

$$v_{n+1} = v_n + a_n \Delta t \tag{3.33a}$$

$$x_{n+1} = x_n + v_n \Delta t. \qquad \text{(Euler algorithm)} \tag{3.33b}$$

Because order Δt terms are retained in (3.33), 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 Δt. 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 an algorithm is nth order if its global error is order $(\Delta t)^n$. The Euler algorithm is an example of a *first-order* algorithm.

The Euler algorithm is asymmetrical because it advances the solution by a time step Δt, but uses information about the derivative only at the beginning of the interval. We already have found that the accuracy of the Euler algorithm is limited and that frequently its solutions are not stable. We also found that a simple modification of (3.33) yields solutions that are stable for oscillatory systems. For completeness, we repeat the Euler-Cromer algorithm here:

$$v_{n+1} = v_n + a_n \Delta t, \tag{3.34a}$$

$$x_{n+1} = x_n + v_{n+1} \Delta t. \qquad \text{(Euler-Cromer algorithm)} \tag{3.34b}$$

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

$$v_{n+1} = v_n + a_n \Delta t, \tag{3.35a}$$

and

$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n) \Delta t. \qquad \text{(midpoint algorithm)} \tag{3.35b}$$

Note that if we substitute (3.35a) for v_{n+1} into (3.35b), we obtain

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n \, \Delta t^2. \tag{3.36}$$

Hence, the midpoint algorithm 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 algorithm. In fact, both algorithms are equally poor, because the errors increase with each time step.

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

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + a_n \Delta t, \tag{3.37a}$$

$$x_{n+1} = x_n + v_{n+\frac{1}{2}} \Delta t. \qquad \text{(half-step algorithm)} \tag{3.37b}$$

Note that the half-step algorithm is not self-starting, that is, (3.37a) 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:

$$v_{\frac{1}{2}} = v_0 + \frac{1}{2} a_0 \, \Delta t. \tag{3.37c}$$

Because the half-step algorithm is stable, it is a common textbook algorithm. The Euler-Richardson algorithm can be motivated as follows. We first write $x(t+\Delta t)$ as

$$x_1 \approx x(t+\Delta t) = x(t) + v(t)\Delta t + \frac{1}{2} a(t)(\Delta t)^2. \tag{3.38}$$

The notation x_1 implies that $x(t+\Delta t)$ is related to $x(t)$ by one time step. We also may divide the step Δt into half steps and write the first half step, $x(t+\frac{1}{2}\Delta t)$, as

$$x(t+\frac{1}{2}\Delta t) \approx x(t) + v(t)\frac{\Delta t}{2} + \frac{1}{2} a(t)\big(\frac{\Delta t}{2}\big)^2. \tag{3.39}$$

The second half step, $x_2(t+\Delta t)$, may be written as

$$x_2(t+\Delta t) \approx x(t+\frac{1}{2}\Delta t) + v(t+\frac{1}{2}\Delta t)\frac{\Delta t}{2} + \frac{1}{2} a(t+\frac{1}{2}\Delta t)\big(\frac{\Delta t}{2}\big)^2. \tag{3.40}$$

We substitute (3.39) into (3.40) and obtain

$$x_2(t+\Delta t) \approx x(t) + \frac{1}{2}\big[v(t) + v(t+\frac{1}{2}\Delta t)\big]\Delta t + \frac{1}{2}\big[a(t) + a(t+\frac{1}{2}\Delta t)\big]\big(\frac{1}{2}\Delta t\big)^2. \tag{3.41}$$

Now $a(t+\frac{1}{2}\Delta t) \approx a(t) + \frac{1}{2}a'(t)\Delta t$. Hence to order $(\Delta t)^2$, (3.41) becomes

$$x_2(t+\Delta t) = x(t) + \frac{1}{2}\big[v(t) + v(t+\frac{1}{2}\Delta t)\big]\Delta t + \frac{1}{2}\big[2a(t)\big]\big(\frac{1}{2}\Delta t\big)^2. \tag{3.42}$$

We can find an approximation that is accurate to order $(\Delta t)^3$ by combining (3.38) and (3.42) so that the terms to order $(\Delta t)^2$ cancel. The combination that works is $2x_2 - x_1$, which gives the Euler-Richardson result:

$$x_{\text{er}}(t+\Delta t) \equiv 2x_2(t+\Delta t) - x_1(t+\Delta t) = x(t) + v(t+\frac{1}{2}\Delta t)\Delta t + O(\Delta t)^3. \tag{3.43}$$

The same reasoning leads to an approximation for the velocity accurate to $(\Delta t)^3$ giving

$$v_{\text{er}} \equiv 2v_2(t+\Delta t) - v_1(t+\Delta t) = v(t) + a(t+\frac{1}{2}\Delta t)\Delta t + O(\Delta t)^3. \tag{3.44}$$

A bonus of the Euler-Richardson algorithm is that the quantities $|x_2 - x_1|$ and $|v_2 - v_1|$ give an estimate for the error. We can use these estimates to change the time step so that the error is always within some desired level of precision. We will see that the Euler-Richardson algorithm is equivalent to the second-order Runge-Kutta algorithm (see (3.54)).

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 (3.32b):

$$x_{n-1} = x_n - v_n\Delta t + \frac{1}{2}a_n(\Delta t)^2. \tag{3.45}$$

If we add the forward and reverse forms, (3.32b) and (3.45) respectively, we obtain

$$x_{n+1} + x_{n-1} = 2x_n + a_n(\Delta t)^2 + O\big((\Delta t)^4\big) \tag{3.46}$$

or

$$x_{n+1} = 2x_n - x_{n-1} + a_n(\Delta t)^2. \qquad \text{(leapfrog algorithm)} \tag{3.47a}$$

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

$$v_n = \frac{x_{n+1} - x_{n-1}}{2\Delta t}. \qquad \text{(leapfrog algorithm)} \tag{3.47b}$$

Note that the global error associated with the leapfrog algorithm (3.47) 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. The leapfrog algorithm is also known as the explicit central difference algorithm. Because this form of the leapfrog algorithm is not self-starting, another algorithm must be used to obtain the first few terms. An additional problem is that the new velocity (3.47b) is found by computing the difference between two quantities of the same order of magnitude. Such an operation results in a loss of numerical precision and may give rise to roundoff errors.

A mathematically equivalent version of the leapfrog algorithm is given by

$$x_{n+1} = x_n + v_n\Delta t + \frac{1}{2}a_n(\Delta t)^2 \tag{3.48a}$$

$$v_{n+1} = v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t. \qquad \text{(velocity Verlet algorithm)} \tag{3.48b}$$

We see that (3.48), known as the *velocity* form of the Verlet algorithm, is self-starting and minimizes roundoff errors. Because we will not use (3.47) in the text, we will refer to (3.48) as the Verlet algorithm.

We can derive (3.48) from (3.47) by the following considerations. We first solve (3.47b) 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 (3.47a) and solve for x_{n+1}, we find the form (3.48a). Then we use (3.47b) to write v_{n+1} as:

$$v_{n+1} = \frac{x_{n+2} - x_n}{2\Delta t}, \tag{3.49}$$

and use (3.47a) 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 (3.49), we obtain

$$v_{n+1} = \frac{x_{n+1} - x_n}{\Delta t} + \frac{1}{2} a_{n+1} \Delta t. \tag{3.50}$$

Finally, we use (3.48a) for x_{n+1} to eliminate $x_{n+1} - x_n$ from (3.50); after some algebra we obtain the desired result (3.48b).

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

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{6}(4a_n - a_{n-1})(\Delta t)^2 \tag{3.51a}$$

$$v_{n+1} = v_n + \frac{1}{6}(2a_{n+1} + 5a_n - a_{n-1})\Delta t. \qquad \text{(Beeman algorithm)} \tag{3.51b}$$

Note that (3.51) does not calculate particle trajectories more accurately than the Verlet algorithm. Its advantage is that it generally 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 many algorithms based on this method, we consider the solution of the first-order differential equation

$$\frac{dx}{dt} = f(x, t). \tag{3.52}$$

Runge-Kutta algorithms evaluate the rate, $f(x, t)$, multiple times in the interval $[t, t + \Delta t]$. For example, the classic fourth-order Runge-Kutta algorithm, which we will discuss in the following, evaluates $f(x, t)$ at four times t_n, $t_n + a_1\Delta t$, $t_n + a_2\Delta t$, and $t_n + a_3\Delta t$. Each evaluation of $f(x, t)$ produces a slightly different rate r_1, r_2, r_3, and r_4. The idea is to advance the solution using a weighted average of the intermediate rates:

$$y_{n+1} = y_n + (c_1 r_1 + c_2 r_2 + c_3 r_3 + c_4 r_4)\Delta t. \tag{3.53}$$

The various Runge-Kutta algorithms correspond to different choices for the constants a_i and c_i. These algorithms are classified by the number of intermediate rates $\{r_i,\, i = 1, \cdots, N\}$. The determination of the Runge-Kutta coefficients is difficult for all but the lowest order methods, because the coefficients must be chosen to cancel as many terms in the Taylor series expansion of $f(x, t)$ as possible. The first non-zero expansion coefficient determines the order of the Runge-Kutta algorithm. Fortunately, these coefficients are tabulated in most numerical analysis books.

To illustrate how the various sets of Runge-Kutta constants arise, consider the case $N = 2$. The second-order Runge-Kutta solution of (3.52) can be written using standard notation as:

$$x_{n+1} = x_n + k_2 + O\big((\Delta t)^3\big), \tag{3.54a}$$

where

$$k_2 = f(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2})\Delta t. \tag{3.54b}$$

$$k_1 = f(x_n, t_n)\Delta t \tag{3.54c}$$

Note that the weighted average uses $c_1 = 0$ and $c_2 = 1$. The interpretation of (3.54) is as follows. The Euler algorithm 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 a more accurate determination of the slope is to use the Euler algorithm 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 rate 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 (3.54c)).

The application of the second-order Runge-Kutta algorithm to Newton's equation of motion (3.31) yields

$$
\begin{aligned}
k_{1v} &= a_n(x_n, v_n, t_n)\Delta t && (3.55a)\\
k_{1x} &= v_n \Delta t && (3.55b)\\
k_{2v} &= a(x_n + \frac{k_{1x}}{2}, v_n + \frac{k_{1v}}{2}, t + \frac{\Delta t}{2})\Delta t && (3.55c)\\
k_{2x} &= (v_n + \frac{k_{1v}}{2})\Delta t, && (3.55d)
\end{aligned}
$$

and

$$
\begin{aligned}
v_{n+1} &= v_n + k_{2v} && (3.56a)\\
x_{n+1} &= x_n + k_{2x}. \qquad \text{(second-order Runge Kutta)} && (3.56b)
\end{aligned}
$$

Note that the second-order Runge-Kutta algorithm in (3.55) and (3.56) is identical to the Euler-Richardson algorithm.

Other second-order Runge-Kutta type algorithms exist. For example, if we set $c_1 = c_2 = \frac{1}{2}$ we obtain the endpoint method:

$$y_{n+1} = y_n + \frac{1}{2}k_1 + \frac{1}{2}k_2 \tag{3.57a}$$

where

$$
\begin{aligned}
k_1 &= f(x_n, t_n)\Delta t && (3.57b)\\
k_2 &= f(x_n + k_1, t_n + \Delta t)\Delta t. && (3.57c)
\end{aligned}
$$

And if we set $c_1 = \frac{1}{3}$ and $c_2 = \frac{2}{3}$, we obtain Ralston's method:

$$y_{n+1} = y_n + \frac{1}{3}k_1 + \frac{2}{3}k_2 \tag{3.58a}$$

where

$$
\begin{aligned}
k_1 &= f(x_n, t_n)\Delta t && (3.58b)\\
k_2 &= f(x_n + \frac{3}{4}k_1, t_n + \frac{3}{4}\Delta t)\Delta t. && (3.58c)
\end{aligned}
$$

Note that Ralston's method does not calculate the rate at uniformly spaced subintervals of Δt. In general, a Runge-Kutta method adjusts the partition of Δt as well as the constants a_i and c_i so as to minimize the error.

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 (3.52) can be written in standard notation as

$$\begin{aligned} k_1 &= f(x_n, t_n)\Delta t && (3.59a) \\ k_2 &= f(x_n + \frac{k_1}{2}, t_n + \frac{\Delta t}{2})\Delta t && (3.59b) \\ k_3 &= f(x_n + \frac{k_2}{2}, t_n + \frac{\Delta t}{2})\Delta t && (3.59c) \\ k_4 &= f(x_n + k_3, t_n + \Delta t)\Delta t, && (3.59d) \end{aligned}$$

and

$$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{3.60}$$

The application of the fourth-order Runge-Kutta algorithm to Newton's equation of motion (3.31) yields

$$\begin{aligned} k_{1v} &= a(x_n, v_n, t_n)\Delta t && (3.61a) \\ k_{1x} &= v_n\Delta t && (3.61b) \\ k_{2v} &= a(x_n + \frac{k_{1x}}{2}, v_n + \frac{k_{1v}}{2}, t_n + \frac{\Delta t}{2})\Delta t && (3.61c) \\ k_{2x} &= (v_n + \frac{k_{1v}}{2})\Delta t && (3.61d) \\ k_{3v} &= a(x_n + \frac{k_{2x}}{2}, v_n + \frac{k_{2v}}{2}, t_n + \frac{\Delta t}{2})\Delta t && (3.61e) \\ k_{3x} &= (v_n + \frac{k_{2v}}{2})\Delta t && (3.61f) \\ k_{4v} &= a(x_n + k_{3x}, v_n + k_{3v}, t + \Delta t) && (3.61g) \\ k_{4x} &= (v_n + k_{3x})\Delta t, && (3.61h) \end{aligned}$$

and

$$\begin{aligned} v_{n+1} &= v_n + \frac{1}{6}(k_{1v} + 2k_{2v} + 2k_{3v} + k_{4v}) && (3.62a) \\ x_{n+1} &= x_n + \frac{1}{6}(k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x}). \quad \text{(fourth-order Runge-Kutta)} && (3.62b) \end{aligned}$$

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

As we have discussed, one way to determine the accuracy of a solution is to calculate it twice with two different values of the time step. One way to make this comparison is to choose time steps Δt and $\Delta t/2$ and compare the solution at the desired time. If the difference is small, the error is assumed to be small. This estimate of the error can be used to adjust the value of the time step. If the error is too large, than the time step can be halved. And if the error is much less than the desired value, the time step can be increased so that the program runs faster.

A better way of controlling the step size was developed by Fehlberg who showed that it is possible to evaluate the rate in such as way as to simultaneously obtain two Runge-Kutta approximations with different orders. For example, it is, possible to run a fourth-order and fifth-order algorithm in tandem by evaluating five rates. We thus obtain different estimates of the true solution using different weighed averages of these rates:

$$y_{n+1} = y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 \tag{3.63a}$$
$$y^*_{n+1} = y_n + c^*_1 k_1 + c^*_2 k_2 + c^*_3 k_3 + c^*_4 k_4. \tag{3.63b}$$

Because we can assume that the fifth-order solution is closer to the true solution than the fourth order algorithm, the difference $|y - y^*|$ provides a good estimate of the error of the fourth-order method. If this estimated error is larger than the desired tolerance, then the step size is decreased. If the error is smaller than the desired tolerance, the step size is increased. The `RK45` ODE solver in the numerics package implements this technique for choosing the optimal step size.

In applications where the accuracy of the numerical solution is important, adaptive time step algorithms should always be used. As stated in *Numerical Recipes*: "Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; they can sometimes be factors of ten, a hundred, or more."

Adaptive step size algorithms are not well suited for tabulating functions or for simulation because the intervals between data points are not constant. An easy way to circumvent this problem is to use a method that takes multiple adaptive steps while checking to insure that the last step does not overshoot the desired fixed step size. The `ODEMultistepSolver` implements this technique. The solver acts like a fixed step size solver, even though the solver monitors its internal step size so as to achieve the desired accuracy.

It also is possible to combine the results from a calculation using two different values of the time step to yield a more accurate expression. Consider the Taylor series expansion of $f(t + \Delta t)$ about t:

$$f(t + \Delta t) = f(t) + f'(t)\Delta t + \frac{1}{2!} f''(t)(\Delta t)^2 + \dots \tag{3.64}$$

Similarly, we have

$$f(t - \Delta t) = f(t) - f'(t)\Delta t + \frac{1}{2!} f''(t)(\Delta t)^2 + \dots \tag{3.65}$$

We subtract (3.65) from (3.64) to find the usual central difference approximation for the derivative

$$f'(t) \approx D_1(\Delta t) = \frac{f(t + \Delta t) - f(t - \Delta t)}{2\Delta t} - \frac{(\Delta t)^2}{6} f'''(t). \tag{3.66}$$

The truncation error is order $(\Delta t)^2$. Next consider the same relation, but for a time step that is twice as large.

$$f'(t) \approx D_1(2\Delta t) = \frac{f(t + 2\Delta t) - f(t - 2\Delta t)}{4\Delta t} - \frac{4(\Delta t)^2}{6} f'''(t). \tag{3.67}$$

Note that the truncation error is again order $(\Delta t)^2$, but is four times bigger. We can eliminate this error to leading order by dividing (3.67) by 4 and subtracting it from (3.66):

$$f'(t) - \frac{1}{4} f'(t) = \frac{3}{4} f'(t) \approx D_1(\Delta t) - \frac{1}{4} D_1(2\Delta t),$$

or

$$f'(t) \approx \frac{4D_1(\Delta t) - D_1(2\Delta t)}{3}. \tag{3.68}$$

It is easy to show that the error for $f'(t)$ is order $(\Delta t)^4$. Recursive difference formulas for derivatives can be obtained by canceling the truncation error at each order. This method is called *Richardson extrapolation.*

Another class of algorithms are *predictor-corrector* algorithms. The idea is to first *predict* the value of the new position:

$$x_p = x_{n-1} + 2v_n\Delta t. \qquad \text{(predictor)} \tag{3.69}$$

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}:

$$v_{n+1} = v_n + \frac{1}{2}(a_p + a_n)\Delta t \tag{3.70a}$$

$$x_{n+1} = x_n + \frac{1}{2}(v_{n+1} + v_n)\Delta t. \qquad \text{(corrected)} \tag{3.70b}$$

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 algorithm is not self-starting. The predictor-corrector algorithm gives more accurate positions and velocities than the leapfrog algorithm and is suitable for very accurate calculations. However, it is computationally expensive, needs significant storage (the forces at the last two stages, and the coordinates and velocities at the last step), and becomes unstable for large time steps.

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.

We now discuss an important class of algorithms, known as *symplectic* algorithms, which are particularly suitable for doing long time simulations of Newton's equations of motion when the force is only a function of position. The basic idea of these algorithms derives from the Hamiltonian theory of classical mechanics. We first give some basic results needed from this theory to understand the importance of symplectic algorithms.

In Hamiltonian theory the generalized coordinates, q_i, and momenta, p_i, take the place of the usual positions and velocities familiar from Newtonian theory. The index i labels both a particle and a component of the motion. For example, in a two particle system in two dimensions, i would run from 1 to 4. The Hamiltonian (which for our purposes can be thought as the total energy) is written as

$$H(q_i, p_i) = T + V, \tag{3.71}$$

where T is the kinetic energy and V is the potential energy. Hamilton's theory is most relevant for non-dissipative systems, which we consider here. For example, for a two particle system in two dimensions connected by a spring, H would take the form:

$$H = \frac{p_1^2}{2m} + \frac{p_2^2}{2m} + \frac{p_3^2}{2m} + \frac{p_4^2}{2m} + \frac{1}{2}k(q_1 - q_3)^2 + \frac{1}{2}k(q_2 - q_4)^2, \tag{3.72}$$

where if the particles are labeled as A and B, we have $p_1 = p_{x,A}$, $p_2 = p_{y,A}$, $p_3 = p_{x,B}$, $p_4 = p_{y,B}$, and similarly for the q_i. The equations of motion are written as first-order differential equations known as Hamilton's equations:

$$\dot{p}_i = -\frac{\partial H}{\partial q_i} \tag{3.73a}$$

$$\dot{q}_i = \frac{\partial H}{\partial p_i}, \tag{3.73b}$$

which are equivalent to Newton's second law and an equation relating the velocity to the momentum. The beauty of Hamiltonian theory is that these equations are correct for other coordinate systems such as polar coordinates, and they also describe rotating systems where the momenta become angular momenta, and the position coordinates become angles.

Because the coordinates and momenta are treated on an equal footing, we can consider the properties of flow in phase space where the dimension of phase space includes both the coordinates and momenta. Thus, one particle moving in one dimension corresponds to a two-dimensional phase space. If we imagine a collection of initial conditions in phase space forming a volume in phase space, then one of the results of Hamiltonian theory is that this volume does not change as the system evolves. A slightly different result, called the *symplectic* property, is that the sum of the areas formed by the projection of the phase space volume onto the planes, q_i, p_i, for each pair of coordinates and momenta also does not change with time. Numerical algorithms that have this property are called symplectic algorithms. These algorithms are built from the following two statements which are repeated M times for each time step.

$$p_i^{(k+1)} = p_i^{(k)} + a_k F_i^{(k)} \delta t \tag{3.74a}$$

$$q_i^{(k+1)} = q_i^{(k)} + b_k p_i^{(k+1)} \delta t, \tag{3.74b}$$

where $F_i^{(k)} \equiv -\partial V(q_i^{(k)})/\partial q_i^{(k)}$. The label k runs from 0 to $M-1$ and one time step is given by $\Delta t = M\delta t$. (We will see that δt is the time step of an intermediate calculation that is made during the time step Δt.) Note that in the update for q_i, the already updated p_i is used. For simplicity, we assume that the mass is unity.

Different algorithms correspond to different values of M, a_k, and b_k. For example, $a_0 = b_0 = M = 1$ corresponds to the Euler-Cromer algorithm, and $M = 2$, $a_0 = a_1 = 1$, $b_0 = 2$, and $b_1 = 0$ is equivalent to the Verlet algorithm as we will now show. If we substitute in the appropriate values for a_k and b_k into (3.74), we have

$$p_i^{(1)} = p_i^{(0)} + F_i^{(0)} \delta t \tag{3.75a}$$

$$q_i^{(1)} = q_i^{(0)} + 2p_i^{(1)} \delta t \tag{3.75b}$$

$$p_i^{(2)} = p_i^{(1)} + F_i^{(1)} \delta t \tag{3.75c}$$

$$q_i^{(2)} = q_i^{(1)} \tag{3.75d}$$

We next combine (3.75a) and (3.75c) for the momentum coordinate and (3.75b) and (3.75d) for the position, and obtain

$$p_i^{(2)} = p_i^{(0)} + (F_i^{(0)} + F_i^{(1)}) \delta t \tag{3.76a}$$

$$q_i^{(2)} = q_i^{(0)} + 2p_i^{(1)} \delta t. \tag{3.76b}$$

We take $\delta t = \Delta t/2$ and combine (3.76b) with (3.75a) and find

$$p_i^{(2)} = p_i^{(0)} + \frac{1}{2}(F_i^{(0)} + F_i^{(1)})\Delta t \tag{3.77a}$$

$$q_i^{(2)} = q_i^{(0)} + p_i^{(0)}\Delta t + \frac{1}{2}F_i^{(0)}(\Delta t)^2, \tag{3.77b}$$

which is identical to the Verlet algorithm, (3.48), because for unit mass the force and acceleration are equal.

Reversing the order of the updates for the coordinates and the momenta also leads to symplectic algorithms:

$$q_i^{(k+1)} = q_i^{(k)} + b_k \delta t p_i^{(k)}, \tag{3.78a}$$

$$p_i^{(k+1)} = p_i^{(k)} + a_k \delta t F_i^{(k+1)} \tag{3.78b}$$

A third variation uses (3.74) and (3.78) for different values of k in one algorithm. Thus, if $M = 2$, which corresponds to two intermediate calculations per time step, we could use (3.74) for the first intermediate calculation and (3.78) for the second.

Why are these algorithms important? Because of the symplectic property, these algorithms will simulate an exact Hamiltonian, although not the one we started with in general (see Problem 3.1c, for example). However, this Hamiltonian will be close to the one we wish to simulate if the a_k and b_k are properly chosen. Second, these algorithms frequently are more accurate and stable than nonsymplectic algorithms. Finally, for even values of M the algorithms are time-reversible invariant, which is a property of the actual systems we are trying to simulate. Examples and comparisons for various algorithms are given in the paper by Gray et al.

References and Suggestions for Further Reading

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

Robert. K. Adair, *The Physics of Baseball*, third edition, Harper Collins (2002).

Byron L. Coulter and Carl G. Adler, "Can a body pass a body falling through the air?," Am. J. Phys. **47**, 841–846 (1979). The authors discuss the limiting conditions for which the drag force is linear or quadratic in the velocity.

Alan Cromer, "Stable solutions using the Euler approximation," Am. J. Phys. **49**, 455–459 (1981). The author shows that a minor modification of the usual Euler approximation yields stable solutions for oscillatory systems including planetary motion and the harmonic oscillator (see Chapter 5).

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

Denis Donnelly and Edwin Rogers, "Symplectic integrators: An introduction," Am. J. Phys., to be published.

A. P. French, *Newtonian Mechanics,* W. W. Norton & Company (1971). Chapter 7 has an excellent discussion of air resistance and a detailed analysis of motion in the presence of drag resistance.

Ian R. Gatland, "Numerical integration of Newton's equations including velocity-dependent forces," Am J. Phys. **62**, 259–265 (1994). The author discusses the Euler-Richardson algorithm.

Stephen K. Gray, Donald W. Noid, and Bobby G. Sumpter, "Symplectic integrators for large scale molecular dynamics simulations: A comparison of several explicit methods," J. Chem. Phys. **101** (5), 4062–4072 (1994).

Margaret Greenwood, Charles Hanna, and John Milton, "Air resistance acting on a sphere: Numerical analysis, strobe photographs, and videotapes," Phys. Teacher **24**, 153–159 (1986). More experimental data and theoretical analysis are given for the fall of ping-pong and styrofoam balls. Also see Mark Peastrel, Rosemary Lynch, and Angelo Armenti, "Terminal velocity of a shuttlecock in vertical fall," Am. J. Phys. **48**, 511–513 (1980).

Michael J. Kallaher, editor *Revolutions in Differential Equations: Exploring ODEs with Modern Technology*, The Mathematical Association of America (1999).

K. S. Krane, "The falling raindrop: variations on a theme of Newton," Am. J. Phys. **49**, 113–117 (1981). The author discusses the problem of mass accretion by a drop falling through a cloud of droplets.

George C. McGuire, "Using computer algebra to investigate the motion of an electric charge in magnetic and electric dipole fields," Am. J. Phys. **71** (8), 809–812 (2003).

Rabindra Mehta, "Aerodynamics of sports balls," in Ann. Rev. Fluid Mech. **17**, 151–189 (1985).

Neville de Mestre, *The Mathematics of Projectiles in Sport*, Cambridge University Press (1990). The emphasis of this text is on solving many problems in projectile motion, for example, baseball, basketball, and golf, in the context of mathematical modeling. Many references to the relevant literature are given.

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.

Emilio Segré, *Nuclei and Particles*, second edition, W. A. Benjamin (1977). Chapter 5 discusses decay cascades. The decay schemes described briefly in Problem 3.13 are taken from C. M. Lederer, J. M. Hollander, and I. Perlman, *Table of Isotopes*, sixth edition, John Wiley & Sons (1967).

Lawrence F. Shampine, *Numerical Solution of Ordinary Differential Equations*, Chapman and Hall (1994).