Chapter 5

Motion in One Dimension

©2002 by Harvey Gould, Jan Tobochnik, and Wolfgang Christian
14 February 2002

We introduce finite difference methods for obtaining numerical solutions to Newton's equations of motion and discuss the qualitative and quantitative behavior of bodies falling near the earth's surface.

5.1 Background

A common example of a physics problem that requires the solution of a differential equation is the motion of a particle acted on by a force. For simplicity, we first discuss one-dimensional motion so that only a single vector component of position, velocity, and acceleration are needed. These variables are related using the language of differential calculus:

$$v(t) = \frac{dy(t)}{dt} \tag{5.1}$$

and

$$a(t) = \frac{dv(t)}{dt}. \tag{5.2}$$

These quantities are known as *kinematical* quantities, because they describe the motion without regard to its cause.

Why do we need the concept of acceleration in kinematics? The answer can be found only *a posteriori*. Thanks to Newton, we know that the net force acting on a particle determines its acceleration. Newton's second law of motion tells us that

$$a(t) = \frac{F(y, v, t)}{m}, \tag{5.3}$$

where F is the net *force* and m is the *inertial mass.* In general, the force depends on position, velocity, and time. Note that Newton's law implies that the motion of a particle does not depend on d^2v/dt^2 or on any higher derivative of the velocity. It is a property of nature, not of mathematics, that we can find simple explanations for motion.

The two first-order equations (5.1) and (5.3) can be combined to obtain the familiar *second-order* differential equation for the position:

$$\frac{d^2y(t)}{dt^2} = \frac{F}{m}. \tag{5.4}$$

However, we will find it easier to describe the motion of a particle by two coupled first-order differential equations (5.1) and (5.3).

Because many types of systems can be modeled by differential equations such as (5.1) and (5.3), it is important to know how to solve such equations. In general, *analytical* solutions of differential equations, that is, solutions in terms of well-known functions, do not exist. We are therefore motivated to find numerical solutions of differential equations. However, analytical solutions are very important and often exist in special or limiting cases. We often use them to test our numerical solutions.

To introduce some simple ways of analyzing numerical solutions, we begin by studying a single differential equation before we tackle Newton's second law. Our understanding will be aided by a visual display of the dependence of the solution on the relevant parameters.

5.2 The Euler Algorithm

The standard technique for numerically solving a differential equation is to convert the differential equation to a *finite difference* equation. Let us consider a first-order differential equation of the form

$$\frac{dy}{dx} = f(x, y) \tag{5.5}$$

and analyze its meaning. Suppose that at $x = x_0$, y has the value y_0. Because (5.5) tells us how y changes at (x_0, y_0), we can find the *approximate* value of y at the neighboring point $x_1 = x_0 + \Delta x$, if Δx is small. The simplest approximation is to assume that $f(x, y)$, the rate of change of y with respect to x, is constant over the interval x_0 to x_1. Then the approximate value of y at $x_1 = x_0 + \Delta x$ is given by

$$y_1 = y(x_0) + \Delta y \approx y(x_0) + f(x_0, y_0)\Delta x. \tag{5.6}$$

We can repeat this procedure to find the value of y at the point $x_2 = x_1 + \Delta x$:

$$y_2 = y(x_1 + \Delta x) \approx y(x_1) + f(x_1, y_1)\Delta x. \tag{5.7}$$

This procedure can be generalized to calculate the approximate value of y at any point $x_{n+1} = x_n + \Delta x$ by the iterative formula

$$y_{n+1} = y_n + f(x_n, y_n)\Delta x. \qquad \text{(Euler algorithm)} \tag{5.8}$$

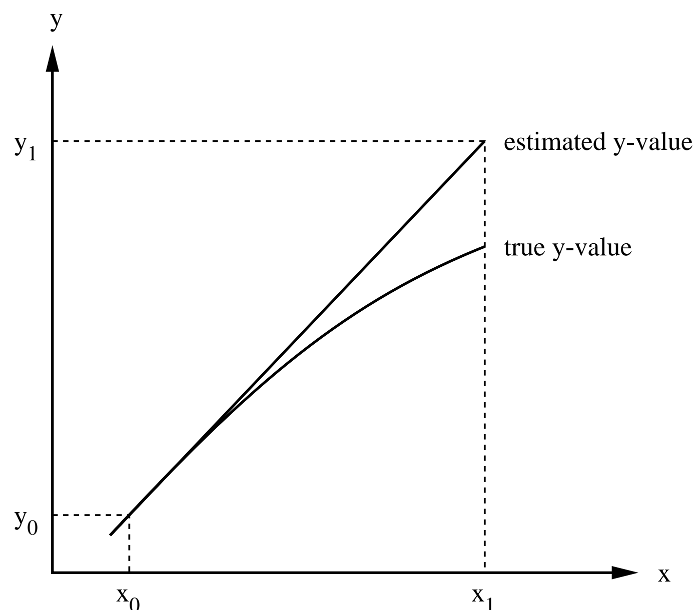

Figure 5.1: Graphical illustration of the Euler algorithm. The slope is evaluated at the beginning of the interval. The results of the Euler algorithm and the true function are represented by a straight line and a curve respectively.

This procedure is called the constant slope or *Euler* algorithm. We expect that (5.8) will yield a good approximation to the exact value of y if Δx is sufficiently small; the degree of "smallness" of Δx depends on our requirements and must be left vague until we consider specific applications.

The Euler algorithm assumes that the rate of change of y is constant over the interval x_n to x_{n+1}, and that the rate of change can be evaluated at the *beginning* of the interval. The graphical interpretation of (5.8) is shown in Figure 5.1. We see that if the slope changes during an interval, a discrepancy occurs between the numerical solution and the exact solution. Nonetheless, the discrepancy can be made smaller if we choose a smaller value of Δx.

5.3 A Simple Example

We first use the Euler algorithm to compute the numerical solution of the differential equation $dy/dx = 2x$ with the initial condition $y_0 = 1$ at $x_0 = 1$. Suppose that we wish to find the approximate value of y at $x = 2$. We choose $\Delta x = 0.1$, so that the number of steps is $n = (x_n - x_0)/\Delta x = (2 - 1)/0.1 = 10$.

The calculation can be arranged as in Table 5.1. The initial condition $x_0 = 1$ determines the initial slope $f(x_0) = 2x_0 = 2$. The value of y at the end of the interval, y_1, is obtained from y_0,

n	x_n	y_n	$f(x) = 2x$	$y_n + \text{slope} \times 0.10$
0	1.00	1.00	2.00	$1.00 + 2.00 \times 0.10 = 1.20$
1	1.10	1.20	2.20	$1.20 + 2.20 \times 0.10 = 1.42$
2	1.20	1.42	2.40	$1.42 + 2.40 \times 0.10 = 1.66$
3	1.30	1.66	2.60	$1.66 + 2.60 \times 0.10 = 1.92$
4	1.40	1.92	2.80	$1.92 + 2.80 \times 0.10 = 2.20$
5	1.50	2.20	3.00	$2.20 + 3.00 \times 0.10 = 2.50$
6	1.60	2.50	3.20	$2.50 + 3.20 \times 0.10 = 2.82$
7	1.70	2.82	3.40	$2.82 + 3.40 \times 0.10 = 3.16$
8	1.80	3.16	3.60	$3.16 + 3.60 \times 0.10 = 3.52$
9	1.90	3.52	3.80	$3.52 + 3.80 \times 0.10 = 3.90$
10	2.00	3.90		

Table 5.1: Iterated solution of the differential equation $dy/dx = 2x$ with $y = 1$ at $x = 1$ using the Euler algorithm. The step size is $\Delta x = 0.1$. Three significant figures are shown.

the value of y at the beginning of the interval, by the relation

$$y_1 = y_0 + \text{slope} \times \Delta x = 1 + 2 \times 0.1 = 1.2. \tag{5.9}$$

This value of y is then transferred to the second line of Table 5.1 and the process is repeated. In this way we find that $y = 3.90$ at $x = 2$. For comparison, the true solution is $y = x^2 = 4$; the error is 2.5%. Convince yourself that a smaller value of Δx improves the accuracy of the solution by redoing Table 5.1 using $\Delta x = 0.05$.

We will now discuss the program in Listing 5.1 that reproduces the steps in Table 5.1. The most important code is in method `calculate` which implements the Euler algorithm using a `while` loop.

Listing 5.1: Implementation of the Euler algorithm.

```
package org.opensourcephysics.sip.ch5;
import java.awt.Color;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.display.*;

public class EulerTestApp implements Calculation {
   Control myControl;
   PlottingPanel plottingPanel;
   DatasetCollection datasetCollection;
   DrawingFrame plottingFrame;
   DataTable dataTable;
   DataTableFrame tableFrame;
   double x0 = 1.0;
   // initial value of x
   double y0 = 1.0;
   // initial value of y
   double xmax = 2.0;                    // x value that stops the calculation
```

```
  double dx = 0.10;                    // step size
  int datasetIndex;

  public EulerTestApp() { // EulerTestApp constructor
    // set up the plotting panel
    plottingPanel = new PlottingPanel("x", "y", "dy/dx");
    plottingFrame = new DrawingFrame(plottingPanel);
    dataTable = new DataTable();               // set up table
    datasetCollection = new DatasetCollection();
    dataTable.add(datasetCollection);
    tableFrame = new DataTableFrame(dataTable);
    plottingPanel.addDrawable(datasetCollection);      // add new datasetCollection to table
  }

  public void setControl(Control c) {
    myControl = c;
    resetCalculation ();
  }

  public double getRate(double x) {
    return 2*x;                          // rate of change of y)
  }

  public void calculate() { // read values from control
    x0 = myControl.getDouble("x0");
    y0 = myControl.getDouble("y0");
    xmax = myControl.getDouble("xmax");
    dx = myControl.getDouble("dx");
    double x = x0;
    double y = y0;
    datasetIndex++;            // set to -1 in reset
    datasetCollection.append(datasetIndex, x, y);      // add initial values
    while (x < xmax) {
      y = y + getRate(x)*dx;   // iterate y
      x = x + dx;              // increase x
      datasetCollection.append(datasetIndex, x, y);      // store result
    }
    plottingPanel.repaint ();              // redraw panel including datasetCollection
    dataTable.refreshTable ();             // redraw the table
  }

  public void resetCalculation() {
    datasetIndex = -1;
    datasetCollection.clear ();
    myControl.setValue("x0", 1);          // default initial value of x
    myControl.setValue("y0", 1);          // default initial value of y
    myControl.setValue("xmax", 2);        // default maximum value of x
    myControl.setValue("dx", 0.1);        // default step size
    plottingPanel.repaint ();             // redraw panel including datasetCollection
    dataTable.refreshTable ();            // redraw table
```

```
    }

    public static void main(String[] args) {
        EulerTestApp app = new EulerTestApp();
        CalculationControl c = new CalculationControl(app);
        app.setControl(c);
    }
}
```

The remaining methods do the usual housekeeping chores of creating display objects, setting default parameter values, and registering the calculation with a control. The `Reset` button restores the default values of the parameters and removes all data sets. The `Calculate` button creates a new data set each time it is pressed and enables the user to run and compare the calculation with different parameters.

Problem 5.1 requires that you study, compile, and modify the `EulerApp` program. This program can be downloaded from the Chapter 5 package on the `OSP` website.

Problem 5.1. Euler algorithm

a. Compile and test the `EulerApp` program. Verify that the output is identical to what is shown in Table 5.1.

b. The choice of a `for` loop or a `while` loop is usually a matter of taste or convenience. Modify the program to replace the `while` loop by a `for` loop in the `calculate` method.

c. Show that the analytical solution of $dy/dx = 2x$ can be written in the form

$$y(x) = y_0 e^{kx}. \tag{5.10}$$

Note that $y(x = 0) = y_0$. What is the value of k in this case?

d. Create a second `datasetCollection` that displays the analytical solution (5.10) together with the numerical solution in the `dataTable` and the graph.

e. Does the step size Δx have any significance? Create a third data set that displays the difference between the analytical solution and the numerical solution in the data table. Show that the error decreases as the step size Δx is decreased. What value of Δx is sufficiently small so that it does not affect your numerical results? Explain.

f. Can the error of the numerical solution be decreased to an arbitrarily small value by decreasing the step size indefinitely? Determining the accuracy of a solution is important and will be discussed in Section 5.8.

5.4 Nuclear Decay

The power of mathematics when applied to physics comes in part from the fact that frequently seemingly unrelated problems can 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 discharge of a capacitor in an RC circuit, and nuclear decay can all be formulated in equivalent ways.

Consider a large number of radioactive nuclei. Although the number of nuclei is discrete, we can often treat this number as a continuous variable. Using this approach, the fundamental law of radioactive decay is that the rate of decay is proportional to the number of nuclei. Thus we can write

$$\frac{dN}{dt} = -\lambda N, \tag{5.11}$$

where N is the number of nuclei and λ is the decay constant. Note that the form of (5.11) is identical to (5.9). Of course, we do not need to use a computer to solve this decay equation, and it can be solved analytically as

$$N(t) = N_0 e^{-\lambda t}, \tag{5.12}$$

where N_0 is the initial number of particles.

Although units are not specified in (5.11) or (5.12), units are important. However, computer programs store numbers and numbers do not have units. For example, as we saw in Section 2.2, we can write the speed of light as

```
public static final double SPEED_OF_LIGHT = 3.0e8;
```

and no explicit units are specified. Because it is awkward to treat very large or very small numbers on a computer, it is convenient to choose units such that the calculated values of the variables are not too far from unity. For example, an astronomical calculation might use a time unit of one year (see Chapter 7) in contrast to the choice of a second as the time unit if we were simulating the motion of a body falling near the earth's surface (see Section 5.5). The different choices can cause confusion because the time units can be anything we want them to be.

In the context of nuclear decay, we want to measure time in terms of the natural time scale in the problem. A commonly used 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 characteristic time, τ, the time it takes for $1/e$ of the original nuclei to decay.

Problem 5.2. Single nuclear species decay

a. Modify `EulerApp` so that nuclear decay notation is used in your program. Add a variable to the control so that can input the decay constant, λ. 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 $t = 2$.

b. Use your modified program to verify that the half life, $T_{1/2}$, is $\ln 2/\lambda$. How long does it take for $1/e$ of the original nuclei to decay?

c. Determine the decay constant λ in units of s^{-1} for $^{238}\text{U} \rightarrow\ ^{234}\text{Th}$ if the half-life is 4.5×10^9 years. What time step would be appropriate for the numerical solution to the rate equation if this rate were used? How would these values change if the particle being modeled where a muon with a lifetime of 2.2×10^{-6} second?

d. Modify your program so that the time t is measured in terms of the half-life. That is, at $t = 1$ half the particles should have decayed and at $t = 2$, one quarter of the particles have decayed. Use your program to determine the time (in years) for 100 atoms of ^{238}U to decay to 20% of their original number. Does the graph change if you calculate the decay of muons?

5.5 Constant Acceleration

In the absence of air resistance, all particles, regardless of size, mass, or composition, have the same acceleration at the same point near the earth's surface. This idealized motion is called "free fall." According to (5.3), constant acceleration implies that the force per unit mass, F/m, is a constant. This constant is commonly denoted by the symbol g. Near the earth's surface, the magnitude of g is approximately 9.8 N/kg or 9.8 m/s^2. Let us adopt the coordinate system shown in Figure 5.2 with the positive direction upward. In this case $a = -g$ and the solution of (5.4) can be written as

$$v(t) = v_0 - gt, \tag{5.13a}$$

and

$$y(t) = y_0 + v_0 t - \frac{1}{2} g t^2, \tag{5.13b}$$

where y_0 and v_0 are the initial position and velocity of the particle respectively.

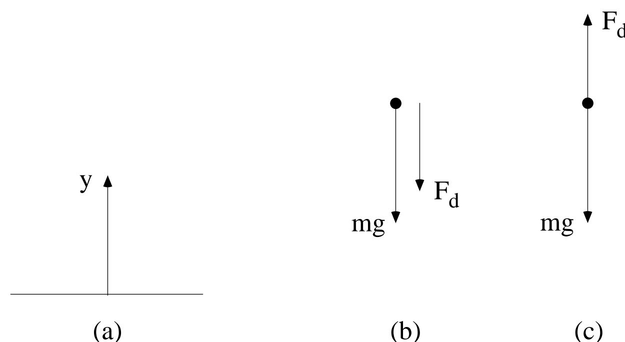

Figure 5.2: (a) Coordinate system with y measured positive upwards from the ground. (b) The force diagram for upward motion. (c) The force diagram for downward motion.

The analysis of free fall under the influence of a constant net force is straightforward, and there is no need to use a computer. However, to ease our introduction to the solution of Newton's second law using numerical methods, we will use a computer in this familiar context. We begin by encapsulating the analytical solution in Listing 5.2:

Listing 5.2: Analytical solution of freely falling particle.

```
public class ParticleFree extends Particle {
```

```
    public ParticleFree (double _y0, double _v0) {
        y = _y0;
        v = _v0;
    }

    // analytical solution of freely falling particle
    public void move (double dt) {
        a = −g;                // included here to emphasize that acceleration is constant
        y = y + v∗dt + 0.5∗a∗dt∗dt;
        double v = v + a∗dt;
    }
}
```

We next apply Euler's algorithm to Newton's equations of motion. We write Newton's second law (5.4) as a system of coupled first-order differential equations and apply the Euler algorithm to each variable. The variables of interest are the particle's vertical position y and velocity v, and the time. The algorithm can be written as:

$$y_{n+1} = y_n + v_n \Delta t \tag{5.14a}$$

$$v_{n+1} = v_n + a_n \Delta t \tag{5.14b}$$

$$t_{n+1} = t_n + \Delta t. \tag{5.14c}$$

Note that we have added an explicit rate equation for time, $dt/dt = \dot{t} = 1$, to the differential equations for y and v. By placing the independent variable, time, on an equal footing with the other dynamical quantities will allow us to easily introduce time-dependent forces in later chapters.

Rather than including the algorithm (5.14) into the `calculate` method, we encapsulate it in a separate class named `ParticleFreeEuler` as follows:

Listing 5.3: Implementation of Euler algorithm for freely falling particle.

```
public class ParticleFreeEuler extends Particle {

    // compute position and velocity of particle in free fall using Euler algorithm
    public void move (double dt) {
        a = −g;            // How would statement be changed if drag resistance present?
        y = y + v∗dt;
        v = v + a∗dt;
    }
}
```

We next list the code for the target class, `ParticleFreeApp`, which instantiates the classes `ParticleFreeEuler` and `ParticleFree` and compares the numerical and analytical solutions. The most important method in `ParticleFreeApp` is the `calculate` method. This method calculates the Euler solution using an instance of `ParticleFreeEuler` and compares it to the analytical solution using an instance of `ParticleFree`. Note that its structure is similar to calculate methods in the previous programs. It is coded as follows:

Listing 5.4: Implementation of the Calculation interface for the freely falling particle.

```
package org.opensourcephysics.ch5;
```

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

public class ParticleFreeApp implements Calculation {
    Control myControl;
    PlottingPanel plottingPanel;
    DrawingFrame plottingFrame;
    DataTable dataTable;
    DataTableFrame tableFrame;
    ParticleFreeEuler euler;
    ParticleFree exact;        // no truncation error
    DatasetCollection datasetCollection;
    double tmin = 0;
    double tmax = 100;
    double y0 = 10.0;          // initial y position
    double v0 = 0;             // initial y velocity
    int numberOfPoints;

    public ParticleFreeApp() {
        plottingPanel = new PlottingPanel("Time t (s)", "Height y (m)", "y vs. t");
        plottingFrame = new DrawingFrame(plottingPanel);
        dataTable = new DataTable();
        tableFrame = new DataTableFrame(dataTable);
        datasetCollection = new DatasetCollection();
        datasetCollection.setXYColumnNames(0, "time", "exact");
        datasetCollection.setXYColumnNames(1, "time", "euler");
        dataTable.add(datasetCollection);
        plottingPanel.addDrawable(datasetCollection);
        // don't display x values of euler data
        dataTable.setColumnVisible(datasetCollection, 2, false);

    }

    public void setControl(Control control) {
        myControl = control;
        resetCalculation();
    }

    public void calculate() {
        y0 = myControl.getDouble("initial y");
        v0 = myControl.getDouble("initial v");
        tmax = myControl.getDouble("tmax (s)"); // time when calculation should stop
        numberOfPoints = myControl.getInt("number of points");
        plottingPanel.setPreferredMinMaxX(tmin, tmax);
        euler.y = y0;                // set initial state
        euler.v = v0;
        exact.y = y0;
        exact.v = v0;
        double dt = (tmax - tmin)/(numberOfPoints - 1);
```

```
        double time = 0;
        for (int i = 0; i < numberOfPoints; i++) {
            exact.move(dt);
            datasetCollection.append(0, time, exact.y);   // add data to dataset
            euler.move(dt);
            datasetCollection.append(1, time, euler.y);
            time += dt;
        }
        plottingPanel.repaint();                        // redraw datasets
        dataTable.refreshTable();                       // redraw table
    }

    public void resetCalculation() {
        plottingPanel.clear();
        dataTable.clear();
        datasetCollection.clear();
        y0 = 10;
        v0 = 0;
        tmax = 100;
        numberOfPoints = 20;
        myControl.setValue("initial y", y0);      // store initial height
        myControl.setValue("initial v", v0);      // store initial velocity
        myControl.setValue("tmax (s)", tmax);  // store time when calculation should stop
        myControl.setValue("number of points", numberOfPoints); // store number of points in the plot
        plottingPanel.repaint();                        // redraw datasets
        dataTable.refreshTable();                       // redraw table
    }

    public static void main(String[] args) {
        Calculation app = new ParticleFreeApp();
        Control c = new CalculationControl(app);
        app.setControl(c);
    }
}
```

After reading the input parameters, the `calculate` method uses a `while` loop to compute y and v until the particle hits the ground at $y = 0$. The only difference is that each Euler data point is generated by iterating the difference equation rather than by evaluating a function as in the move method in `ParticleFree`. Exercise 5.3 asks you to run the `ParticleFreeApp` program to see if it works as expected.

Exercise 5.3. Free Fall Test

a. Compile and run `ParticleFreeApp`. Test the correctness of the program by comparing the results of the calculation of the analytical solution to a pencil and paper calculation.

b. Using the default values $y = 20$ and $v = 0$, what value of Δt is required so that the computed value of y at $t = 1$ using the Euler algorithm is accurate to one percent? Does the required value of Δt change at $t = 2$ or does the numerical solution retain the same one percent accuracy for all times?

5.6 Some Other Simple Algorithms

The algorithm for obtaining a numerical solution of a differential equation is not unique, and there are many algorithms that reduce to the same differential equation in the limit $\Delta t \to 0$. For example, a simple variation of (5.14) is to determine y_{n+1} using v_{n+1}, the velocity at the *end* of the interval rather than at the beginning. We write this modified Euler algorithm as

$$v_{n+1} = v_n + a_n \Delta t \tag{5.15a}$$
$$y_{n+1} = y_n + v_{n+1}\Delta t. \qquad \text{(Euler-Cromer algorithm)} \tag{5.15b}$$

We refer to (5.15) as the Euler-Cromer algorithm. Is there any *a priori* reason to prefer the Euler algorithm over the Euler-Cromer algorithm?

It might occur 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 of the interval. 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 $y_{\rm mid}$ and velocity $v_{\rm mid}$ at a time $t_{\rm mid} = t + \Delta t/2$. Then we compute the force, $F(y_{\rm mid}, v_{\rm mid}, t_{\rm mid})$ and the acceleration $a_{\rm mid}$ at $t_{\rm mid}$. The new position y_{n+1} and velocity v_{n+1} at time t_{n+1} is found using $v_{\rm mid}$ and $a_{\rm mid}$. We summarize the Euler-Richardson algorithm as follows:

$$a_n = F(y_n, v_n, t_n)/m \tag{5.16a}$$
$$v_{\rm mid} = v_n + \frac{1}{2} a_n \Delta t, \tag{5.16b}$$
$$y_{\rm mid} = y_n + \frac{1}{2} v_n \Delta t, \tag{5.16c}$$
$$a_{\rm mid} = F(y_{\rm m}, v_{\rm m}, t + \frac{1}{2}\Delta t)/m, \tag{5.16d}$$

and

$$v_{n+1} = v_n + a_{\rm mid}\Delta t. \tag{5.17a}$$
$$y_{n+1} = y_n + v_{\rm mid}\Delta t. \qquad \text{(Euler-Richardson algorithm)} \tag{5.17b}$$

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 5A. You are asked to implement the Euler-Cromer and Euler-Richardson algorithms in Exercise 5.4.

Exercise 5.4. Algorithm Test

a. Write a class named `ParticleFreeEulerCromer` that encapsulates the Euler-Cromer algorithm and modify the `ParticleFreeApp` program to test this class. What time step is needed to achieve one percent accuracy after an elapsed time of one second?

b. Repeat part (a) using the Euler-Richardson algorithm. Which algorithm works best for the case of a freely falling particle?

5.7 Forces on Falling Objects

The analytical solution for free fall near the earth's surface, (5.13), is well known. However, 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

$$F = \frac{GMm}{(R+y)^2} = \frac{GMm}{R^2(1+y/R)^2} = mg\big(1 - 2\frac{y}{R} + \cdots\big), \tag{5.18}$$

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 5.5. Position-dependent force

Modify `ParticleFreeApp` to simulate the fall of a particle with the position-dependent force law (5.18). 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 this impact velocity differs by one percent from its value with a constant acceleration $g = 9.8\,\mathrm{m/s}^2$. Take the radius of the earth to be $6.37 \times 10^6\,\mathrm{m}$. Make sure that the one percent error is due to the physics of the force law and not the accuracy of the algorithm.

For particles near the earth's surface, a more important modification of the free fall problem is the retarding drag force due to air resistance. The direction of the drag force is opposite to the velocity of the particle. We first discuss the motion of a falling body. The direction of the drag force F_d is opposite to the motion and is upward as shown in Figure 5.2c. Hence, the total force F on the particle can be written as

$$F = -mg + F_d. \tag{5.19}$$

In general, it is necessary to determine the velocity dependence of F_d empirically over a limited range of conditions. One way to find the form of $F_d(v)$ is to measure y as a function of t and to determine the velocity and acceleration as a function of t. From this information it is possible to find the acceleration as a function of v and to extract $F_d(v)$ from (5.19). However, this algorithm is not useful in general, because errors are introduced by taking the derivatives needed to find the velocity and acceleration. A better method 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 $y(t)$ is consistent with the experimental values of $y(t)$, then the assumed v dependence of $F_d(v)$ is justified empirically.

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

$$F_d(v)/m = C_1 v \tag{5.20a}$$

and

$$F_d(v)/m = C_2 v^2, \tag{5.20b}$$

where the parameters C_1 and C_2 depend on the properties of the medium and the shape of the object. We stress that the forms (5.20) are not exact laws of physics, but instead 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 $F_d = mg$ and the acceleration equals zero. This terminal speed can be found from (5.19) and (5.20) and is given by

$$v_t = \frac{g}{C_1} \tag{5.21a}$$

or

$$v_t = \left(\frac{g}{C_2}\right)^{1/2} \tag{5.21b}$$

for the linear and quadratic cases, respectively. It frequently is convenient to measure velocities in terms of the terminal velocity. We can use (5.20) and (5.21) to write F_d in the linear and quadratic cases as

$$F_{1,d}/m = C_1 v_t \big(\frac{v}{v_t}\big) = mg\frac{v}{v_t} \tag{5.22a}$$

and

$$F_{2,d}/m = C_2 {v_t}^2 \big(\frac{v}{v_t}\big)^2 = mg\big(\frac{v}{v_t}\big)^2. \tag{5.22b}$$

Hence, we can write the net force on a falling object in the convenient form

$$F_1(v) = -mg\big(1 - \frac{v}{v_t}\big), \tag{5.23a}$$

and

$$F_2(v) = -mg\big(1 - \frac{v^2}{{v_t}^2}\big). \tag{5.23b}$$

To determine if the effects of air resistance are important in 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. From (5.21b) 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 time of about 3 s, we expect that the effects of air resistance would be appreciable for comparable times and distances. Hence, many of the textbook problems we encounter in elementary mechanics courses are not realistic. More realistic physics is treated in Problems 5.7 and 5.6.

Problem 5.6. The fall of a styrofoam ball

a. Use the empirical data for the displacement $y(t)$ in Table 5.2 to estimate the velocity $v(t)$ using the central difference approximation given in (3.10):

$$v(t) \approx \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t}. \qquad \text{(central difference approximation)} \tag{5.24}$$

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

$$v(t) \approx \frac{y(t+\Delta t) - y(t)}{\Delta t}, \qquad \text{(forward difference approximation)} \tag{5.25}$$

t (s)	position (m)
-0.132	0.0
0.0	0.075
0.1	0.260
0.2	0.525
0.3	0.870
0.4	1.27
0.5	1.73
0.6	2.23
0.7	2.77
0.8	3.35

Table 5.2: Results by Greenwood, Hanna, and Milton (see references) for the vertical fall of a styrofoam ball of mass 0.254 gm and radius 2.54 cm. Note that the initial time is negative and not an integer multiple of 0.1.

we can express the acceleration as

$$a(t) \approx \frac{y(t + \Delta t) - 2y(t) + y(t - \Delta t)}{(\Delta t)^2}. \tag{5.26}$$

Use (5.24) and (5.26) to estimate the velocity and acceleration. Estimate the terminal velocity from the data given in Table 5.2. This estimation is nontrivial, in part because the terminal velocity has not yet been reached during the time interval shown in the table. 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.

b. Adopt 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.

c. Use your program with the net force given by either (5.23a) or (5.23b). Choose the terminal velocity as one of the input parameters, and take as your first guess for the terminal velocity the value you found in part (a). Make sure that your computed results for the displacement of the particle, $y(t) - y(0)$, do not depend on Δt. Compare your plot of the computed values of $y(t) - y(0)$ for the linear and quadratic form of the drag resistance with the empirical values of $y(t) - y(0)$ and visually determine which form of the drag force yields the best overall fit. What is your criteria for the "best" fit? Should you match your results with the experimental data at early times or at later times? Or should you adopt another strategy? What are the qualitative differences between the two computed forms of $y(t) - y(0)$?

Problem 5.7. Effect of air resistance on the ascent and descent of a pebble

a. Verify the claim made in Section 5.7 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 performing a numerical 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 decent?

c. Perform a simulation to determine if your qualitative answers are correct. Assume that the drag force is proportional to v^2. From (5.22) we see that 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. Suggestions: It is a good idea to choose an initial velocity that allows the pebble to remain in the air for a total time such that the time of ascent differs appreciably from the time of descent. A reasonable choice is $v(t = 0) = 50\,$m/s. Choose the coordinate system shown in Figure 5.2 with y positive upward. What is the net force for $v > 0$ and $v < 0$? You might find it convenient to the drag force in the form $F_d = -v*\texttt{Math.abs}(v)$. One way to determine the maximum height of the pebble is to use the statement

```
if (v*vold < 0)
   System.out(println("maximum height = " + y);
```

where $v = v_{n+1}$ and vold $= v_n$.

Multiple nuclear decays can also produce systems of first-order differential equations. Problem 5.8 asks you to model such a system using the techniques that we have developed for falling particles.

Problem 5.8. Multiple nuclear decays

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

2. ^{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?

3. ^{211}Rn decays via two branches as shown in Figure 5.3. 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.

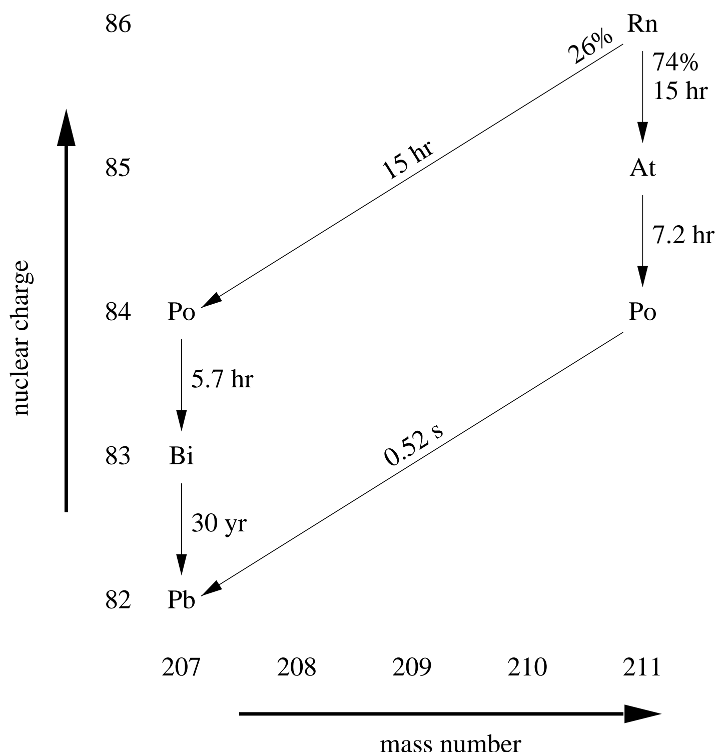

Figure 5.3: The decay scheme of ^{211}Rn. Note that ^{211}Rn decays via two branches, and the final product is the stable isotope ^{207}Pb. All vertical transitions are by electron capture (a form of beta decay), and all diagonal transitions are by alpha decay. The times represent half-lives.

5.8 Accuracy and Stability

Now that we have learned how to use the numerical methods to find a numerical solution to simple first-order differential equations, we need to develop some practical guidelines to help us estimate the accuracy of various methods. Because we have replaced a differential equation by a difference equation, our numerical solution cannot be identically equal, in general, to the "true" solution of the original differential equation. In general, the discrepancy between the two solutions has two causes. One cause is that computers do not store numbers with infinite precision, but rather to a maximum number of digits that is hardware and software dependent. Most programming languages allow the programmer to distinguish between *floating point* or *real* numbers, that is, numbers with decimal points, and *integer* or *fixed point* numbers. Arithmetic with numbers represented by integers is exact. However in general, we cannot solve a differential equation using integer arithmetic. Hence, arithmetic operations such as addition and division, which involve real numbers, can introduce an error, called the *roundoff error*. For example, if a computer only stored real numbers to two significant figures, the product 2.1×3.2 would be stored as 6.7 rather than the exact value 6.72. The

significance of roundoff errors is that they accumulate as the number of mathematical operations increases. Ideally we should choose algorithms that do not significantly magnify the roundoff error, for example, we should avoid subtracting numbers that are nearly the same in magnitude.

The other source of the discrepancy between the true answer and the computed answer is the error associated with the choice of algorithm. This error is called the *truncation error*. A truncation error would exist even on an idealized computer that stored floating point numbers with infinite precision and hence had no roundoff error. Because this error depends on the choice of algorithm and hence can be controlled by the programmer, you should be motivated to learn more about numerical analysis and the estimation of truncation errors. However, there is no general prescription for the best method for obtaining numerical solutions of differential equations. We will find in later chapters that each method has advantages and disadvantages and the proper selection depends on the nature of the solution, which you might not know in advance, and on your objectives. How accurate must the answer be? Over how large an interval do you need the solution? What kind of computer are you using? How much computer time and personal time do you have?

In practice, we usually can determine the accuracy of our numerical solution by reducing the value of Δt until the numerical solution is unchanged at the desired level of accuracy. Of course, we have to be careful not to choose Δt too small, because too many steps would be required and the total computation time and roundoff error would increase.

In addition to accuracy, another important consideration is the stability of an algorithm. For example, it might happen that the numerical results are very good for short times, but diverge from the true solution for longer times. This divergence might occur if small errors in the algorithm are multiplied many times, causing the error to grow geometrically. Such an algorithm is said to be *unstable* for the particular problem. We consider the accuracy and the stability of the Euler method in Problems 5.9 and 5.10.

Problem 5.9. Accuracy of the Euler method

a. Use the modified version of the Euler program that you created for Problem 5.1 to compute the value of y at $x = 2\,\text{min}$ with $\Delta x = 0.1$, 0.05, 0.025, 0.01, and 0.005. Your program should already have a table column showing the difference between the exact solution

$$y(x) = y_0 e^{kx} \qquad \text{where} \qquad y(x=0) = y_0 \tag{5.27}$$

and the numerical solution. Is the difference between these solutions a decreasing function of Δx? That is, if Δx is decreased by a factor of two, how does the difference change? Plot the difference as a function of Δx. If your points fall approximately on a straight line, then the difference is proportional to Δx (for $\Delta x \ll 1$). A numerical method is called nth order, if the difference between the analytical solution and the numerical solution is proportional to $(\Delta x)^n$ at a fixed value of x. What is the order of the Euler method?

b. One way to determine the accuracy of a numerical solution is to repeat the calculation with a smaller step size and compare the results. If the two calculations agree to p decimal places, we can reasonably assume that the results are correct to p decimal places. What value of Δt is necessary for 0.1% accuracy at $x = 2$? What value of Δt is necessary for 0.1% accuracy at $x = 4$?

Problem 5.10. Stability of the Euler method

a. Consider the differential equation

$$R\frac{dQ}{dt} = V - \frac{Q}{C}, \tag{5.28}$$

with $Q = 0$ at $t = 0$. This equation represents the charging of a capacitor in an RC circuit with an applied voltage V. Measure t in seconds and choose $R = 2000\,\Omega$, $C = 10^{-6}$ farads, and $V = 10$ volts. Do you expect $Q(t)$ to increase with t? Does $Q(t)$ increase indefinitely or does it reach a steady-state value? Write a program to solve (5.28) numerically using the Euler method. What value of Δt is necessary to obtain three decimal accuracy at $t = 0.005$ s?

b. What is the nature of your numerical solution to (5.28) at $t = 0.005$ s for $\Delta t = 0.005$, 0.0025, and 0.001? Does a small change in Δt lead to a large change in the computed value of Q? Is the Euler method stable in this calculation for any value of Δt?

Problem 5.11. Second-order algorithm

As we saw in the Problem 5.9, the Euler method is only first-order in accuracy. We can increase the accuracy of Equation 5.9 to second-order by the following reasoning. Expand $y(x)$ in a Taylor series:

$$y(x + \Delta x) = y(x) + \frac{dy(x)}{dx}\Delta x + \frac{1}{2}\frac{d^2y(x)}{dx^2}(\Delta x)^2 + \cdots, \tag{5.29}$$

and use (5.9) to write $d^2y(x)/dx^2 = k, dy(x)/dx$. We also use (5.9) to write $dy(x)/dx$ in terms of $y(x)$ and express (5.29) as

$$y(x + \Delta x) = y(x) + ky(x)\Delta x + \frac{k^2}{2}y(x)(\Delta x)^2 + \cdots \tag{5.30}$$

If we write the Euler estimate as $y_e = y(x) + ky(x)\Delta x$, we can rewrite (5.30) as

$$y(x + \Delta x) = y(x) + \frac{k}{2}(y(x) + y_e)\Delta x. \tag{5.31}$$

From the form of (5.31), we see that a second-order algorithm can be obtained by using the average of the Euler estimate for the value of y at $x+\Delta x$ and the value of y at x in an Euler-like expression. Modify your program so that the second-order algorithm (5.31) is used and repeat Problem 5.9.

5.9 Levels of Simulation

So far we have done simulations in which the microscopic complexity of the system 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. Then we reduced the number of degrees of freedom even more by representing the collisions of the pebble with the billions of molecules in the air by a velocity-dependent friction term. It is remarkable that 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 in a more microscopic manner. For example, we can 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 particle. How accurately do we need to represent the potential energy of interaction between the fluid particles? Clearly the level of detail that is needed in a model 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 words *simulation* and *modeling* are frequently used interchangeably and their precise meaning is not important, especially because people who work with models and who do simulations do not use them precisely. Most practitioners would say that in Chapter 5 we have solved several mathematical models numerically. Beginning with Chapter 6, we will be able to say that we actually 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 5A: The Euler-Richardson Method

We motivate the Euler-Richardson method (5.16) in the following. We write $y(t+\Delta t)$ as a Taylor series to second-order in Δt:

$$y_1 = y(t+\Delta t) = y(t) + v(t)\Delta t + \frac{1}{2}a(t)(\Delta t)^2, \tag{5.32}$$

where $a(t) = a(y(t), v(t), t)$. The notation y_1 implies that $y(t+\Delta t)$ is related to $y(t)$ by one time step. We also divide the step Δt into half steps and write the first half step, $y(t+\frac{1}{2}\Delta t)$, as

$$y(t+\frac{1}{2}\Delta t) = y(t) + v(t)\frac{\Delta t}{2} + \frac{1}{2}a(t)\big(\frac{\Delta t}{2}\big)^2. \tag{5.33}$$

The second half step, $y_2(t+\Delta t)$, can be written as

$$y_2(t+\Delta t) = y(t+\frac{1}{2}\Delta t) + v(t+\frac{1}{2}\Delta t)\frac{\Delta t}{2} + \frac{1}{2}a(t+\frac{1}{2}\Delta t)\big(\frac{\Delta t}{2}\big)^2. \tag{5.34}$$

We substitute (5.33) into (5.34) and obtain

$$y_2(t+\Delta t) = y(t) + \frac{1}{2}\big[v(t) + v(t+\frac{1}{2}\Delta t)\big]\Delta t + \frac{1}{2}\big[a(t) + a(t+\frac{1}{2}\Delta t)\big]\big(\frac{1}{2}\Delta t\big)^2. \tag{5.35}$$

Now $a(t+\frac{1}{2}\Delta t) = a(t) + \frac{1}{2}a'(t)\Delta t + \ldots$ Hence to order $(\Delta t)^2$, (5.35) becomes

$$y_2(t+\Delta t) = y(t) + \frac{1}{2}\big[v(t) + v(t+\frac{1}{2}\Delta t)\big]\Delta t + \frac{1}{2}\big[2a(t)\big]\big(\frac{1}{2}\Delta t\big)^2. \tag{5.36}$$

We can create an approximation that is accurate to order $(\Delta t)^3$ by combining (5.32) and (5.36) so that the terms to order $(\Delta t)^2$ cancel. The combination that works is $2y_2 - y_1$, which gives the Euler-Richardson result:

$$y_{\rm er}(t+\Delta t) = 2y_2(t+\Delta t) - y_1(t+\Delta t) = y(t) + v(t+\frac{1}{2}\Delta t)\Delta t + O(\Delta t)^3. \tag{5.37}$$

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

$$v_{\rm er} = 2v_2(t+\Delta t) - v_1(t+\Delta t) = v(t) + a(t+\frac{1}{2}\Delta t)\Delta t + O(\Delta t)^3. \tag{5.38}$$

A bonus of the Euler-Richardson method is that the quantities $|y_2 - y_1|$ and $|v_2 - v_1|$ give an estimate for the error in the procedure. We can use these estimates to change the time step so that the error is always within some desired level of precision.

References and Suggestions for Further Reading

William R. Bennett, *Scientific and Engineering Problem-Solving with the Computer*, Prentice Hall (1976). One of the first books to incorporate computer problem solving. Many one- and two-dimensional falling body problems are considered in Chapter 5.

Byron L. Coulter and Carl G. Adler, “Can a body pass a body falling through the air?,” *Am. J. Phys.* **47**, 841 (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 (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 6).

R. M. Eisberg, *Applied Mathematical Physics with Programmable Pocket Calculators*, McGraw-Hill (1976). Chapter 3 of this handy paperback is similar in spirit to the present discussion.

Richard P. Feynman, Robert B. Leighton and Matthew Sands, *The Feynman Lectures on Physics, Vol. 1*, Addison-Wesley (1963). Feynman discusses the numerical solution of Newton’s equations in Chapter 9.

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 (1994). The author discusses the Euler-Richardson algorithm.

Margaret Greenwood, Charles Hanna, and John Milton, “Air resistance acting on a sphere: numerical analysis, strobe photographs, and videotapes,” *Phys. Teacher* **24**, 153 (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 (1980).

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

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

Nancy Roberts, David Andersen, Ralph Deal, Michael Jaret, and William Shaffer, *Introduction to Computer Simulation: The System Dynamics Approach*, Addison-Wesley (1983). A book on computer simulation in the social sciences.

R. R. Rogers, *A Short Course in Cloud Physics*, Pergamon Press (1976).

Emilio Segré, *Nuclei and Particles*, second edition, W. A. Benjamin (1977). Chapter 5 discusses decay cascades. The decay schemes described briefly in Problem 5.8 are taken from C. M. Lederer, J. M. Hollander, and I. Perlman, *Table of Isotopes*, sixth edition, John Wiley & Sons (1967).

It is impossible to list all the excellent books on numerical analysis. We list a few of our favorites.

Forman S. Acton, *Numerical Methods That Work*, Harper & Row (1970); corrected edition, Mathematical Association of America (1990). A somewhat advanced, but clearly written text.

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

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes*, second edition, Cambridge University Press (1992).