Chapter 17

Orbits

Dynamics of many-body systems.

Many mathematical models involve the dynamics of objects under the influence of both their mutual interaction and the surrounding environment. The objects might be planets, molecules, vehicles, or people. The ultimate goal of this chapter is to investigate the *n-body problem* in celestial mechanics, which models the dynamics of a system of planets, such as our solar system. But first, we look at two simpler models and programs, a bouncing ball and Brownian motion.

The `exm` program `bouncer` is a model of a bouncing ball. The ball is tossed into the air and reacts to the pull of the earth's gravitation force. There is a corresponding pull of the ball on the earth, but the earth is so massive that we can neglect its motion.

Mathematically, we let $v(t)$ and $z(t)$ denote the velocity and the height of the ball. Both are functions of time. High school physics provides formulas for $v(t)$ and $z(t)$, but we choose not to use them because we are anticipating more complicated problems where such formulas are not available. Instead, we take small steps of size δ in time, computing the velocity and height at each step. After the initial toss, gravity causes the velocity to decrease at a constant rate, g. So each step updates $v(t)$ with

$$v(t + \delta) = v(t) - \delta\,g$$

The velocity is the rate of change of the height. So each step updates $z(t)$ with

$$z(t + \delta) = z(t) + \delta\,v(t)$$

Here is the core of `bouncer.m`.

```
[z0,h] = initialize_bouncer;
g = 9.8;
c = 0.75;
delta = 0.005;
v0 = 21;
while v0 >= 1
    v = v0;
    z = z0;
    while all(z >= 0)
        set(h,'zdata',z)
        drawnow
        v = v - delta*g;
        z = z + delta*v;
    end
    v0 = c*v0;
end
finalize_bouncer
```

The first statement

```
[z0,h] = initialize_bouncer;
```

generates the plot of a sphere shown in figure 17.1 and returns z0, the z-coordinates
of the sphere, and h, the Handle Graphics "handle" for the plot. One of the exer-
cises has you investigate the details of initialize_bouncer. The figure shows the
situation at both the start and the end of the simulation. The ball is at rest and so
the picture is pretty boring. To see what happens during the simulation, you have
to actually run bouncer.

The next four statements in bouncer.m are

```
g = 9.8;
c = 0.75;
delta = 0.005;
v0 = 21;
```

These statements set the values of the acceleration of gravity g, an elasticity coef-
ficient c, the small time step delta, and the initial velocity for the ball, v0.

All the computation in bouncer is done within a doubly nested while loop.
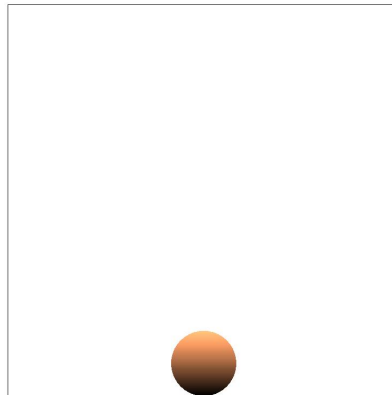The outer loop involves the initial velocity v0.

```
while v0 >= 1
    ...
    v0 = c*v0;
end
```

To achieve the bouncing affect, the initial velocity is repeatedly multiplied by $c =$
0.75 until it is less than 1. Each bounce starts with a velocity equal to 3/4 of the
previous one.

Within the outer loop, the statements

Figure 17.1. *Initial, and final, position of a bouncing ball. To see what happens in between, run bouncer.*

```
v = v0;
z = z0;
```

initialize the velocity `v` to `v0` and the height `z` to `z0`. Then the inner loop

```
while all(z >= 0)
   set(h,'zdata',z)
   drawnow
   v = v - delta*g;
   z = z + delta*v;
end
```

proceeds until the height goes negative. The plot is repeatedly updated to reflect the current height. At each step, the velocity `v` is decreased by a constant amount, `delta*g`, thereby affecting the gravitational deceleration. This velocity is then used to compute the change in the height `z`. As long as `v` is positive, the `z` increases with each step. When `v` reaches zero, the ball has reached its maximum height. Then `v` becomes negative and `z` decreases until the ball returns to height zero, terminating the inner loop.

After both loops are complete, the statement

```
finalize_bouncer
```

activates a pushbutton that offers you the possibility of repeating the simulation.

Brownian motion is not as obvious as gravity in our daily lives, but we do encounter it frequently. Albert Einstein's first important scientific paper was about Brownian motion. Think of particples of dust suspended in the air and illuminated

by a beam of sunlight. Or, diffusion of odors throughout a room. Or, a beach ball being tossed around a stadium by the spectators.

In Brownian motion an object – a dust particle, a molecule, or a ball – reacts to surrounding random forces. Our simulation of these forces uses the built-in MATLAB function `randn` to generate normally distributed random numbers. Each time the statement

```
randn
```

is executed a new, unpredictable, value is produced. The statement

```
randn(m,n)
```

produces an m-by-n array of random values. Each time the statement

```
hist(randn(100000,1),60)
```

is executed a histogram plot like the one in figure 17.2 is produced. Try executing this statement several times. You will see that different histograms are produced each time, but they all have the same shape. You might recognize the "bell-shaped curve" that is known more formally as the Gaussian or normal distribution. The histogram shows that positive and negative random numbers are equally likely and that small values are more likely than large ones. This distribution is the mathematical heart of Brownian motion.
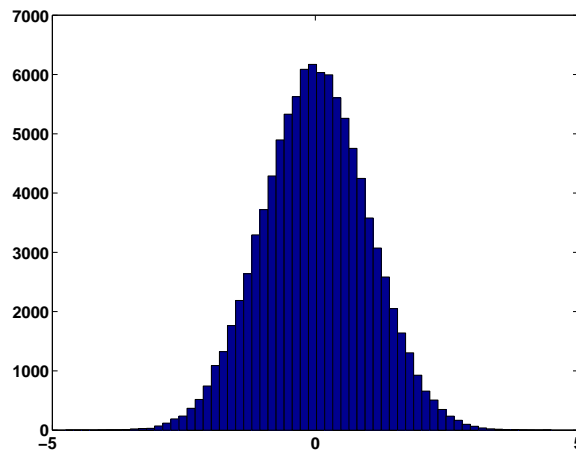
Figure 17.2. *Histogram of the normal random number generator.*

A simple example of Brownian motion known as a random walk is shown in figure 17.3. This is produced by the following code fragment.

```
m = 100;
x = cumsum(randn(m,1));
y = cumsum(randn(m,1));
```

```
plot(x,y,'.-')
s = 2*sqrt(m);
axis([-s s -s s]);
```
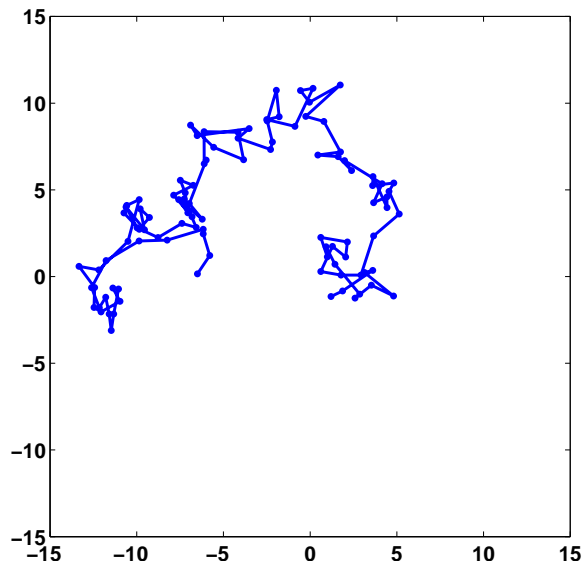

**Figure 17.3.** *A simple example of Brownian motion.*

The key statement is

```
x = cumsum(randn(m,1));
```

This statement generates the x-coordinates of the walk by forming the successive cumulative partial sums of the elements of the vector `r = randn(m,1)`.

$$
\begin{aligned}
x_1 &= r_1 \\
x_2 &= r_1 + r_2 \\
x_3 &= r_1 + r_2 + r_3 \\
&\dots
\end{aligned}
$$

A similar statement generates the y-coordinates. Cut and paste the code fragment into the MATLAB command window. Execute it several times. Try different values of `m`. You will see different random walks going off in different random directions. Over many executions, the values of `x` and `y` are just as likely to be positive as negative. We want to compute an axis scale factor `s` so that most, but not all, of the walks stay within the plot boundaries. It turns out that as m, the length of the walk, increases, the proper scale factor increases like $\sqrt{m}$.

A fancier Brownian motion program, involving simultaneous random walks of many particles in three dimensions, is available in `brownian3.m`. A snapshot of the evolving motion is shown in figure 17.4. Here is the core of `brownian3.m`.
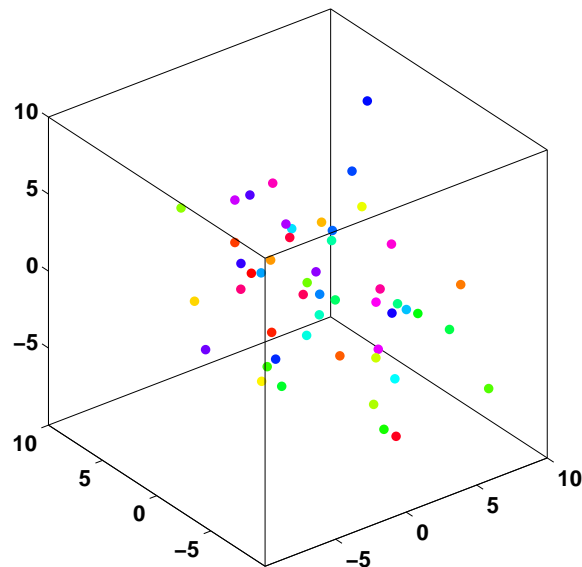
Figure 17.4. *A snapshot of the output from brownian3, showing simulta-neous random walks of many particules in three dimensions.*

```
n = 50;  % Default number of particles
P = zeros(n,3);
H = initialize_graphics(P);

while ~get(H.stop,'value')

   % Obtain step size from slider.
   delta = get(H.speed,'value');

   % Normally distributed random velocities.
   V = randn(n,3);

   % Update positions.
   P = P + delta*V;

   update_plot(P,H);

end
```

The variable `n` is the number of particles. It is usually equal to 50, but some other number is possible with `brownian3(n)`. The array P contains the positions of `n` particles in three dimensions. Initially, all the particles are located at the origin, $(0, 0, 0)$. The variable H is a MATLAB structure containing handles for all the user

interface controls. In particular, `H.stop` refers to a toggle that terminates the `while` loop and `H.speed` refers to a slider that controls the speed through the value of the time step `delta`. The array `V` is an `n`-by-3 array of normally distributed random numbers that serve as the particle velocities in the random walks. Most of the complexity of `brownian3` is contained in the subfunction `initialize_graphics`. In addition to the speed slider and the stop button, the GUI has pushbuttons or toggles to turn on a trace, zoom in and out, and change the view point.

We are now ready to tackle the *n-body problem* in celestial mechanics. This is a model of a system of planets and their interaction described by Newton's laws of motion and gravitational attraction. Over five hundred years ago, Johannes Kepler realized that if there are only two planets in the model, the orbits are ellipses with a common focus at the center of mass of the system. This provides a fair description of the moon's orbit around the earth, or of the earth's orbit around the sun. But if you are planning a trip to the moon or a mission to Mars, you need more accuracy. You have to realize that the sun affects the moon's orbit around the earth and that Jupiter affects the orbits of both the earth and Mars. Furthermore, if you wish to model more than two planets, an analytic solution to the equations of motion is not possible. It is necessary to compute numerical approximations.

Our notation uses vectors and arrays. Let n be the number of bodies and, for $i = 1, \ldots, n$, let p_i be the vector denoting the position of the i-th body. For two-dimensional motion the i-th position vector has components (x_i, y_i). For three-dimensional motion its components are (x_i, y_i, z_i). The small system shown in figure 17.5 illustrates this notation. There are three bodies moving in two dimensions. The coordinate system and units are chosen so that initially the first body, which is gold if you have color, is at the origin,

$$p_1 = (0, 0)$$

The second body, which is blue, is one unit away from the first body in the x direction, so

$$p_2 = (1, 0)$$

The third body, which is red, is one unit away from the first body in the y direction, so

$$p_3 = (0, 1)$$

We wish to model how the position vectors p_i vary with time, t. The *velocity* of a body is the rate of change of its position and the *acceleration* is the rate of change of its velocity. We use one and two dots over p_i to denote the velocity and acceleration vectors, $\dot{p}_i$ and $\ddot{p}_i$. If you are familiar with calculus, you realize that the dot means differentiation with respect to t. For our three body example, the first body is initially heading away from the other two bodies, so its velocity vector has two negative components,

$$\dot{p}_1 = (-0.12, -0.36)$$

The initial velocity of the second body is all in the y direction,

$$\dot{p}_2 = (0, 0.72)$$
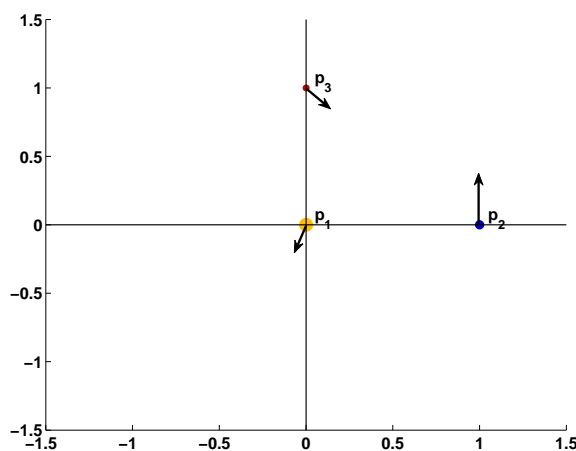
Figure 17.5. *Initial positions and velocities of a small system with three bodies in two-dimensional space.*

and the initial velocity of the third body is sending it towards the second body,

$$\dot{p}_3 = (0.36, -0.36)$$

Newton's law of motion, the famous $F = ma$, says that the mass of a body times its acceleration is proportional to the sum of the forces acting on it. Newton's law of gravitational says that the force between any two bodies is proportional to the product of their masses and inversely proportional to the square of the distance between them. So, the equations of motion are

$$m_i \ddot{p}_i = \gamma \sum_{j \neq i} m_i m_j \frac{p_j - p_i}{||p_j - p_i||^3}, \quad i = 1, \ldots, n$$

Here γ is the gravitational constant, m_i is the mass of the i-th body, $p_j - p_i$ is the vector from body i to body j and $||p_j - p_i||$ is the length or *norm* of that vector, which is the distance between the two bodies. The denominator of the fraction involves the cube of the distance because the numerator contains the distance itself and so the resulting quotient involves the inverse of the square of the distance.

Figure 17.6 shows our three body example again. The length of the vector $r_{23} = p_3 - p_2$ is the distance between p_2 and p_3. The gravitation forces between the bodies located at p_2 and p_3 are directed along r_{23} and $-r_{23}$.

To summarize, the position of the i-th body is denoted by the vector p_i. The instantaneous change in position of this body is given by its velocity vector, denoted by $\dot{p}_i$. The instantaneous change in the velocity is given by its acceleration vector, denoted by $\ddot{p}_i$. The acceleration is determined from the position and masses of all the bodies by Newton's laws of motion and gravitation.

The following notation simplifies the discussion of numerical methods. Stack the position vectors on top of each other to produce an n-by-d array where n is the
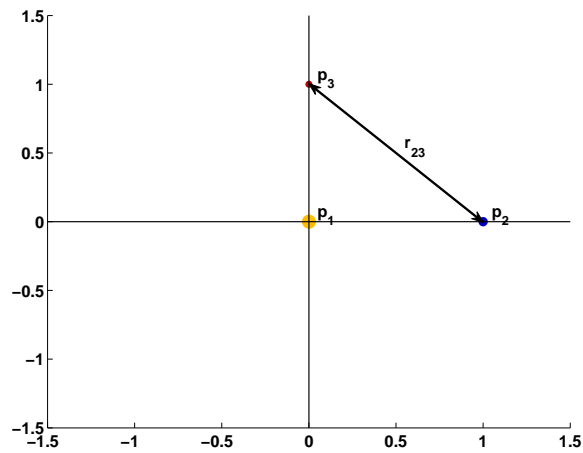
Figure 17.6. *The double arrow depicts the vectors $r_{23} = p_3 - p_2$ and $-r_{32}$. The length of this arrow is the distance between p_2 and p_3.*

number of bodies and $d = 2$ or 3 is the number of spatial dimensions..

$$P = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}$$

Let V denote a similar array of velocity vectors.

$$V = \begin{pmatrix} \dot{p}_1 \\ \dot{p}_2 \\ \vdots \\ \dot{p}_n \end{pmatrix}$$

And, let $G(P)$ denote the array of gravitation forces.

$$G(P) = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix}$$

where

$$g_i = \gamma \sum_{j \neq i} m_j \frac{p_j - p_i}{||p_j - p_i||^3}$$

With this notation, the equations of motion can be written

$$\dot{P} = V$$
$$\dot{V} = G(P)$$

For our three body example, the initial values of P and V are

$$P = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

and

$$V = \begin{pmatrix} -0.12 & -0.36 \\ 0 & 0.72 \\ 0.36 & -0.36 \end{pmatrix}$$

The masses in our three body example are

$$m_1 = 1/2, \quad m_2 = 1/3, \quad m_3 = 1/6$$

From these quantities, we can compute the initial value of the gravitation forces, $G(P)$.

We will illustrate our numerical methods by trying to generate a circle. The differential equations are

$$\dot{x} = y$$
$$\dot{y} = -x$$

With initial conditions $x(0) = 0, y(0) = 1$, the exact solution is

$$x(t) = \sin t, \quad y(t) = \cos t$$

The orbit is a perfect circle with a period equal to 2π.

The most elementary numerical method, which we will not actually use, is known as the *forward* or *explicit Euler* method. The method uses a fixed time step δ and simultaneously advances both the positions and velocities from time t_k to time $t_{k+1} = t_k + \delta$.

$$P_{k+1} = P_k + \delta\, V_k$$
$$V_{k+1} = V_k + \delta\, G(P_k)$$

The forward Euler's method applied to the circle generator problem becomes

$$x_{k+1} = x_k + \delta\, y_k$$
$$y_{k+1} = y_k - \delta\, x_k$$

The result for $\delta = 2\pi/30$ is shown in the first plot in figure 17.7. Instead of a circle we get a growing spiral. The method is unstable and consequently unsatisfactory, particularly for long time periods. Smaller time steps merely delay the inevitable. We would see more complicated, but similar, behavior with the n-body equations.

Another elementary numerical method is known as the *backward* or *implicit Euler* method. In general, it involves somehow solving a nonlinear system at each step.

$$P_{k+1} - \delta\, V_{k+1} = P_k$$
$$V_{k+1} - \delta\, G(P_{k+1}) = V_k$$
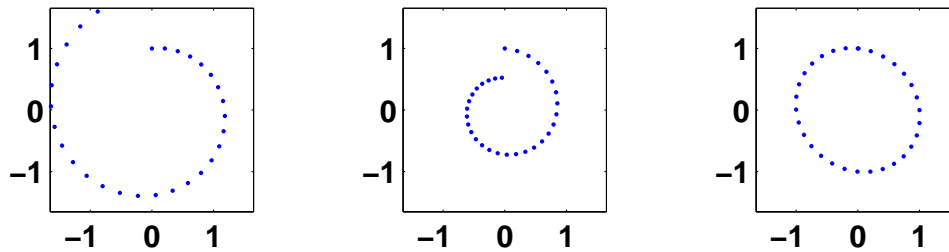
Figure 17.7. *Three versions of Euler's method for generating a circle. The first plot shows that the forward method is unstable. The second plot shows that the backward method has excessive damping. The third plot shows that symplectic method, which is a compromise between the first two methods, produces a nearly perfect circle.*

For our simple circle example the implicit system is linear, so x_{k+1} and y_{k+1} are easily computed by solving the 2-by-2 system

$$x_{k+1} - \delta\, y_{k+1} \; = \; x_k$$
$$y_{k+1} + \delta\, x_{k+1} \; = \; y_k$$

The result is shown in the second plot in figure 17.7. Instead of a circle we get a decaying spiral. The method is stable, but there is too much damping. Again, we would see similar behavior with the n-body equations.

The method that we actually use is a compromise between the explicit and implicit Euler methods. It is the most elementary instance of what are known as *symplectic* methods. The method involves two half-steps. In the first half-step, the positions at time t_k are used in the gravitation equations to update of the velocities.

$$V_{k+1} = V_k + \delta\, G(P_k)$$

Then, in the second half-step, these "new" velocities are used to update the positions.

$$P_{k+1} = P_k + \delta\, V_{k+1}$$

The novel feature of this symplectic method is the subscript $k+1$ instead of k on the V term in the second half-step.

For the circle generator, the symplectic method is

$$x_{k+1} \; = \; x_k + \delta\, y_k$$
$$y_{k+1} \; = \; y_k - \delta\, x_{k+1}$$

The result is the third plot in figure 17.7. If you look carefully, you can see that the orbit in not quite a circle. It's actually a nearly circular ellipse. And the final value does not quite return to the initial value, so the period is not exactly 2π . But the important fact is that the orbit is neither a growing nor a decaying spiral.
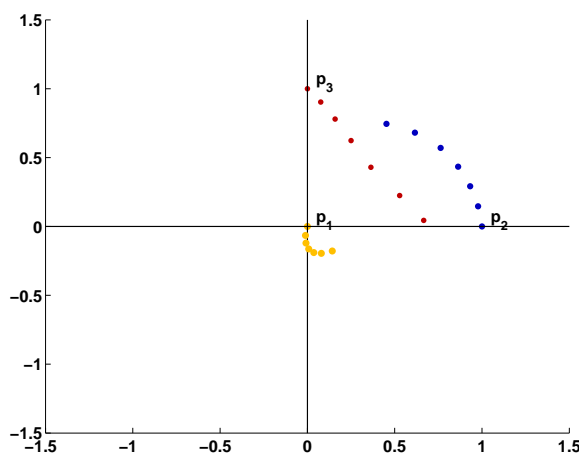
Figure 17.8. *The first few steps of our example system.*

There are more complicated symplectic algorithms that are much more accurate per step than this symplectic Euler. But the symplectic Euler is satisfactory for generating well behaved graphical displays. Most well-known numerical methods, including Runge-Kutta methods and traditional multistep methods, do not have this symplectic stability property and, as a result, are not as satisfactory for computing orbits over long time spans.

Figure 17.8 shows the first few steps for our example system. As we noted earlier, the initial position and velocity are

```
P =
         0          0
    1.0000          0
         0     1.0000


V =
   -0.1200    -0.3600
         0     0.7200
    0.3600    -0.3600
```

After one step with $\delta = 0.20$ we obtain the following values.

```
P =
   -0.0107    -0.0653
    0.9776     0.1464
    0.0767     0.9033


V =
   -0.0533    -0.3267
   -0.1118     0.7318
```

```
     0.3836     -0.4836
```

The three masses, $1/2$, $1/3$, and $1/6$, are not equal, but are comparable, so all three bodies have significant affects on each other and all three move noticeable distances. We see that the initial velocity of the first body causes it to move away from the other two. In one step, its position changes from $(0,0)$ to small negative values, $(-0.0107, -0.0653)$ The second body is initially at position $(1,0)$ with velocity $(0,1)$ in the positive y direction. In one step, its position changes to $(0.9776, 0.1464)$. The x-coordinate has changed relatively little, while the y-coordinate has changed by roughly $0.72\,\delta$. The third body moves in the direction indicated by the velocity vector in figure 17.5.

After a second step we have the following values. As expected, all the trends noted in the first step continue.

```
P =
    -0.0079    -0.1209
     0.9325     0.2917
     0.1589     0.7793

V =
     0.0136    -0.2779
    -0.2259     0.7268
     0.4109    -0.6198
```
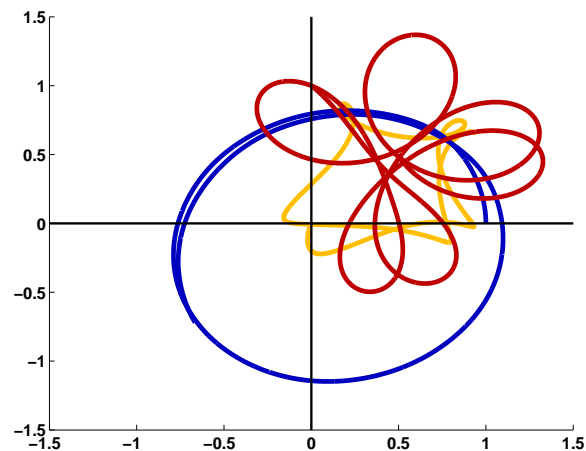

Figure 17.9. *The initial trajectories of our example system.*

Figure 17.9 shows an initial section of the trajectories. You should run our *Experiments* program `orbits(3)` to see the three bodies in motion. The small body and the large body orbit in a clockwise direction around each other while the medium-size body orbits in a counter-clockwise direction around the other two.
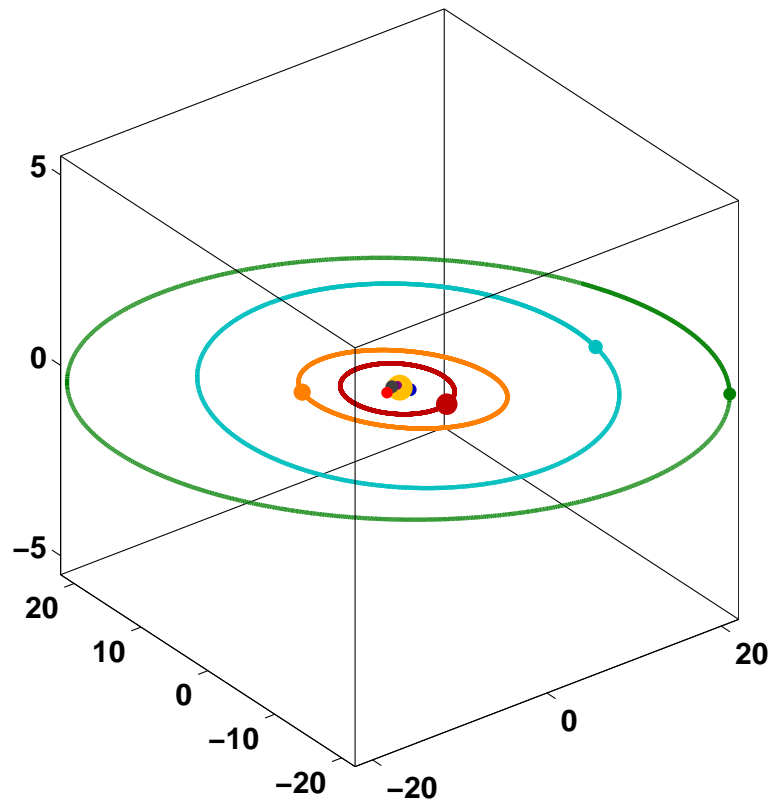
Figure 17.10. *The solar system, with the initial positions of all the planets and the orbits of the outer planets, Jupiter, Saturn, Uranus, and Neptune.*

Our *Experiments* program `orbits` models nine bodies in the solar system, namely the sun and eight planets. Figures 17.10 and 17.11 show snapshots of the output from `orbits` with two different zoom factors that are necessary to span the scale of the system. The orbits for all the planets are in the proper proportion. But, obviously, the symbols for the sun and the planets do not have the same scale. Web sources for information about the solar system are provided by the University Corporation for Atmospheric Research, the Jet Propulsion Laboratory, and the US National Air and Space Museum,

```
http://www.windows.ucar.edu
http://www.jpl.nasa.gov/solar_system
http://airandspace.si.edu:80/etp/ss/index.htm
```
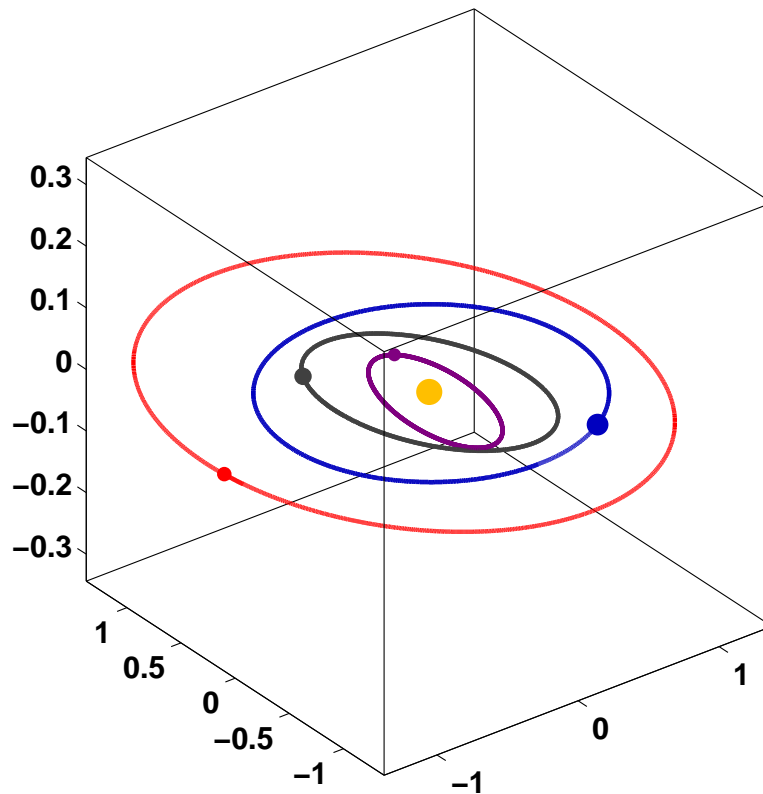
Figure 17.11. *Zooming in by a factor of 16 reveals the orbits of the inner planets, Mercury, Venus, Earth and Mars.*

Recap

```
%% Orbits Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Orbits Chapter of "Experiments in MATLAB".
% You can access it with
%
%    orbits_recap
%    edit orbits_recap
%    publish orbits_recap
%
% Related EXM programs
%
%    bouncer
%    orbits
```

```
%% Core of bouncer, simple gravity. no gravity

   % Initialize

   z0 = eps;
   g = 9.8;
   c = 0.75;
   delta = 0.005;
   v0 = 21;
   y = [];

   % Bounce

   while v0 >= 1
      v = v0;
      z = z0;
      while z >= 0
         v = v - delta*g;
         z = z + delta*v;
         y = [y  z];
      end
      v0 = c*v0;
   end

   % Simplified graphics

   close all
   figure
   plot(y)

%% Normal random number generator.

   figure
   hist(randn(100000,1),60)


%% Snapshot of two dimensional Brownian motion.

   figure
   m = 100;
   x = cumsum(randn(m,1));
   y = cumsum(randn(m,1));
   plot(x,y,'.-')
   s = 2*sqrt(m);
   axis([-s s -s s]);
```

```
%% Snapshot of three dimensional Brownian motion, brownian3

   n = 50;
   delta = 0.125;
   P = zeros(n,3);

   for t = 0:10000
      % Normally distributed random velocities.
      V = randn(n,3);
      % Update positions.
      P = P + delta*V;
   end

   figure
   plot3(P(:,1),P(:,2),P(:,3),'.')
   box on

%% Orbits, the n-body problem.

%{
% ORBITS  n-body gravitational attraction for n = 2, 3 or 9.
%   ORBITS(2), two bodies, classical elliptic orbits.
%   ORBITS(3), three bodies, artificial planar orbits.
%   ORBITS(9), nine bodies, the solar system with one sun and 8 planets.
%
%   ORBITS(n,false) turns off the uicontrols and generates a static plot.
%   ORBITS with no arguments is the same as ORBITS(9,true).

   % n = number of bodies.
   % P = n-by-3 array of position coordinates.
   % V = n-by-3 array of velocities
   % M = n-by-1 array of masses
   % H = graphics and user interface handles

   if (nargin < 2)
      gui = true;
   end
   if (nargin < 1);
      n = 9;
   end

   [P,V,M] = initialize_orbits(n);
   H = initialize_graphics(P,gui);

   steps = 20;     % Number of steps between plots
```

```
    t = 0;              % time

    while get(H.stop,'value') == 0

        % Obtain step size from slider.
        delta = get(H.speed,'value')/(20*steps);

        for k = 1:steps

            % Compute current gravitational forces.
            G = zeros(size(P));
            for i = 1:n
                for j = [1:i-1 i+1:n];
                    r = P(j,:) - P(i,:);
                    G(i,:) = G(i,:) + M(j)*r/norm(r)^3;
                end
            end

            % Update velocities using current gravitational forces.
            V = V + delta*G;

            % Update positions using updated velocities.
            P = P + delta*V;

        end

        t = t + steps*delta;
        H = update_plot(P,H,t,gui);
    end

    finalize_graphics(H,gui)
end
%}

%% Run all three orbits, with 2, 3, and 9 bodies, and no gui.

    figure
    orbits(2,false)

    figure
    orbits(3,false)

    figure
    orbits(9,false)
```

Exercises

17.1 *Bouncing ball.*
(a) What is the maximum height of the bouncing ball?
(b) How many times does the ball bounce?
(c) What is the effect of changing each of the four `bouncer` values `g`, `c`, `delta`, and `v0`.

17.2 *Pluto and Ceres.* Change `orbits` to `orbits11` by adding the erstwhile planet Pluto and the recently promoted dwarf planet Ceres. See Wikipedia:

```
http://en.wikipedia.org/wiki/Planet
http://en.wikipedia.org/wiki/Ceres_(dwarf_planet)
```

and

```
http://orbitsimulator.com/gravity/articles/ceres.html
```

17.3 *Comet.* Add a comet to `orbits`. Find initial conditions so that the comet has a stable, but highly elliptical orbit that extends well beyond the orbits of the planets.

17.4 *Twin suns.* Turn the sun in `orbits` into a twin star system, with two suns orbiting each other out of the plane of the planets. What eventually happens to the planetary orbits? For example, try

```
sun1.p = [1 0 0];
sun1.v = [0 0.25 0.25];
sun1.m = 0.5;
sun2.p = [-1 0 0];
sun2.v = [0 -0.25 -0.25];
sun2.m = 0.5;
```

Try other values as well.