Simulating Central-Force Problems

6.1 Stating the general problem and a possible line of attack

In Chapter 4, we learned how we could use an iterative procedure (called the Euler method) to advance the position of an object in small increments based on its current velocity. This allowed us to numerically obtain $x(t)$ given $v(t)$. In other words, we performed a numerical integration: given the known velocity-time graph, we generated the position-time graph.

The task in many physics problems is slightly different, however. Very often, we know the force acting on an object at all points in space, and we would like to somehow calculate the object's trajectory in response to the forces it encounters. How do we do that?

So, let's assume that we know the force that is acting on an object as a function of the object's location. What this means is that no matter where the object happens to be, we can calculate the force that it experiences there. Mathematically speaking, what we are given is the force-function, $\vec{F}(\vec{r})$. This notation communicates that we can evaluate a force vector, $\vec{F}$, by inserting a position vector, $\vec{r}$, into a function. Mathematicians would call such a function a map from R^3 to R^3. A good example are the so-called *central-force problems*, such as the gravitational force a comet feels in the vicinity of the sun.

We also know that the force on the object is related to the acceleration of the object via Newton's second law:

$$\vec{F} = m\vec{a} \tag{6.1}$$

This means that simply dividing the force-function by the object's mass gives us the acceleration function $\vec{a}(\vec{r})$.

Now that we have the acceleration, how do we get the trajectory? Recall that we faced a somewhat similar task in Chapter 4. There we knew the velocity and were able to compute the position. But now we are one step further removed. We know the acceleration, not velocity. What's more, we know the acceleration not as a function of time, but as a function of position.

You may know that in these types of problems, in order to compute the object's trajectory, we must know how the object was initialized. At what position was it released and what velocity did it have at that moment? These two things must be known to us if we are to find the unique trajectory - they are called the *initial conditions.*

So here is the basic idea. Let start from this position and velocity in our code, call them: $\vec{x}_1$ and $\vec{v}_1$. From $\vec{x}_1$ we can compute the force at that location and thus the acceleration $\vec{a}_1$. Using this $\vec{a}_1$, we now update the velocity. Here we make use of the formula for average acceleration,

$$\vec{a}_{avg} = \frac{\Delta\vec{v}}{\Delta t}. \tag{6.2}$$

As the time interval gets very small (and, in the limit, infinitesimal), the average acceleration becomes the instantaneous acceleration, and so,

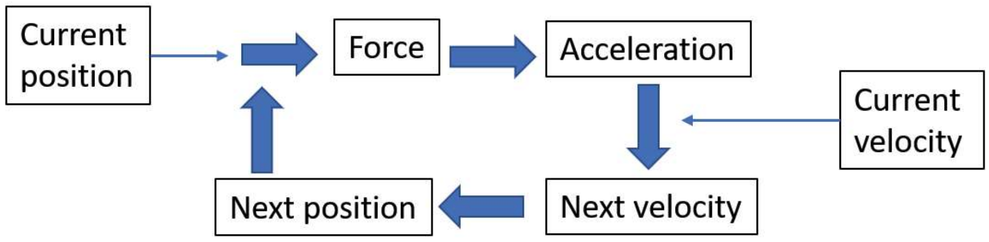
$$\vec{v}_2 = \vec{v}_1 + \vec{a} * \Delta t. \tag{6.3}$$

This completes the second step.

In the third and final step, we now use this new velocity to update the position, using the vector-verson of Equation (4.2), namely:

$$\vec{x}_2 = \vec{x}_1 + \vec{v}_2 * \Delta t. \tag{6.4}$$

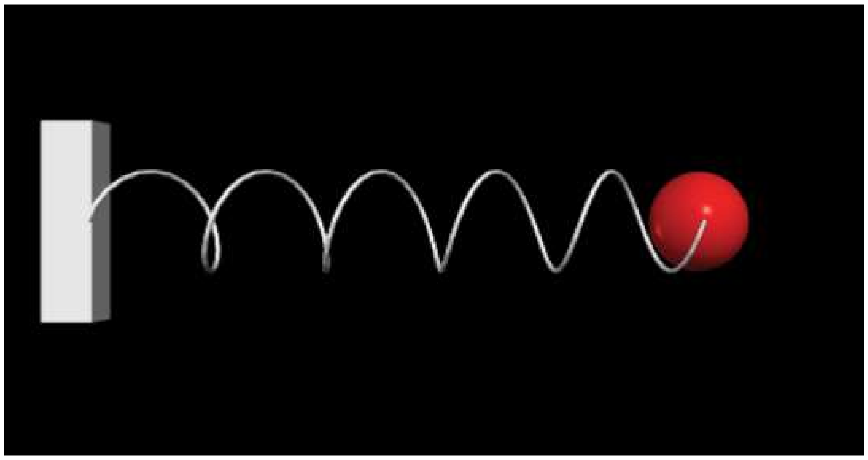
Now that we have computed the new position vector $\vec{x}_2$, we can start the whole procedure over again by first computing the new force and acceleration ($\vec{F}_2, \vec{a}_2$), then the new velocity ($\vec{v}_3$), and then finally the new position ($\vec{x}_3$). Schematically, we are proceeding in steps given by the following flow chart:



In summary, we basically apply the Euler method twice, once for the velocity and then for the position. Notice also that we update the velocity first and then use the new velocity to update position second. It turns out that this order of operation is often superior to the reverse order in terms of computational accuracy. It is called the *Euler-Cromer* method [1].

[1]A. Cromer, Stable solutions using the Euler Approximation, *American Journal of Physics*, **49**, 455 (1981).

6.2 A first example: The mass-spring problem



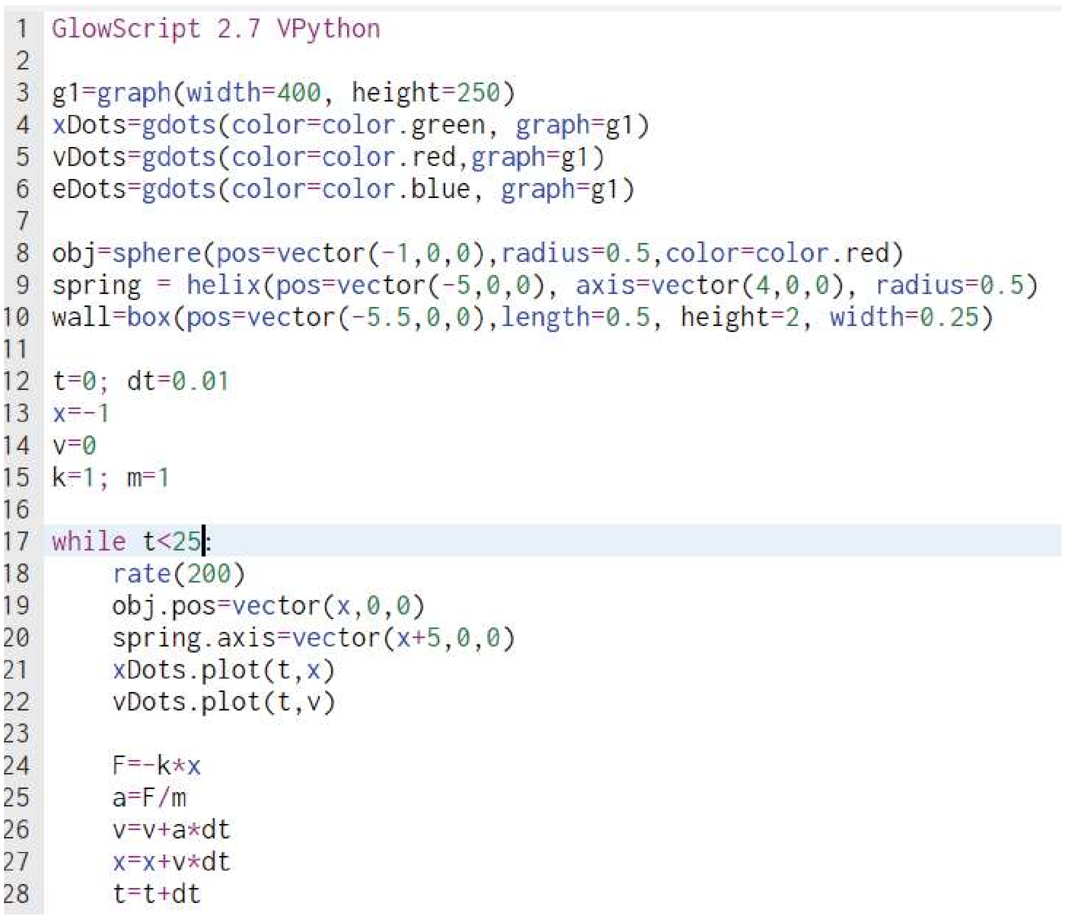
To illustrate the method, let's get our feet wet with a relatively simple problem - a mass on a horizontal spring. All we have to remember is Hooke's law for ideal springs,

$$F(x) = -kx, \tag{6.5}$$

where k is called the spring constant which corresponds to the stiffness of the spring, and x represents the amount of stretch (if positive) or compression (if negative) of the spring. We should appreciate this formula as the one-dimensional version of $\vec{F}(\vec{r})$ from before. Dividing Equation (6.5) by the mass of the object that we attach to the end of the spring yields the object's acceleration.

Let's look at the basic VPython code, shown in Figure 6.1, to see how the Euler-Cromer method gets implemented in practice here. The basic steps (outlined in the flowchart in the previous section) are contained in lines 24 through 27. Line 28 is not absolutely necessary, but it is included for the purpose of making position and velocity-time graphs.

Notice also how easy it is in VPython to draw a spring-mass setup - we basically select a sphere for the end-mass and a helix for the



```
GlowScript 2.7 VPython

g1=graph(width=400, height=250)
xDots=gdots(color=color.green, graph=g1)
vDots=gdots(color=color.red,graph=g1)
eDots=gdots(color=color.blue, graph=g1)

obj=sphere(pos=vector(-1,0,0),radius=0.5,color=color.red)
spring = helix(pos=vector(-5,0,0), axis=vector(4,0,0), radius=0.5)
wall=box(pos=vector(-5.5,0,0),length=0.5, height=2, width=0.25)

t=0; dt=0.01
x=-1
v=0
k=1; m=1

while t<25:
    rate(200)
    obj.pos=vector(x,0,0)
    spring.axis=vector(x+5,0,0)
    xDots.plot(t,x)
    vDots.plot(t,v)

    F=-k*x
    a=F/m
    v=v+a*dt
    x=x+v*dt
    t=t+dt
```

Figure 6.1: The basic code simulating a mass at the end of a spring

spring. The only thing we have to take care of is to deform the helix in dependence upon the position of the end-mass.

6.3 Exercises

- Run the program in Figure 6.1 and examine the position-time graph. What does the graph look like? What math function does it seem to follow? What about the velocity-time graph?
- What can you say about the relationship between the position-time graph and the velocity-time graph. Is there

a phase difference between them, and if so, approximately what is it?

- Find the place in the code where the initial conditions (x_0 and v_0) are specified. Try running the program with a few different sets of initial conditions. Does the period of oscillation seem to depend on this choice?

- Find the place in the code where system's physical parameters are specified, namely the spring constant and mass. Change one of these parameters at a time, and observe qualitatively how it changes the period of oscillation.

- What if you double (or triple) both the mass and the spring constant? Do the graphs change?

- Are your observations above consistent with the well-known formula for the period of oscillation, T, given below?

$$T = 2\pi\sqrt{\frac{m}{k}}$$

6.4 A second example: Projectile motion with air drag

As a second example - one that is slightly more difficult while also showing off some of VPython's strengths - let us consider a projectile launched through the atmosphere. Here we don't want to neglect air drag, as is customary in introductory physics, and as we did in Chapter 5. Instead, we recall that a good formula for the drag force (under certain assumptions) is given by,

$$\vec{F}_D = \frac{1}{2}\rho A C_D v^2(-\hat{v}), \tag{6.6}$$

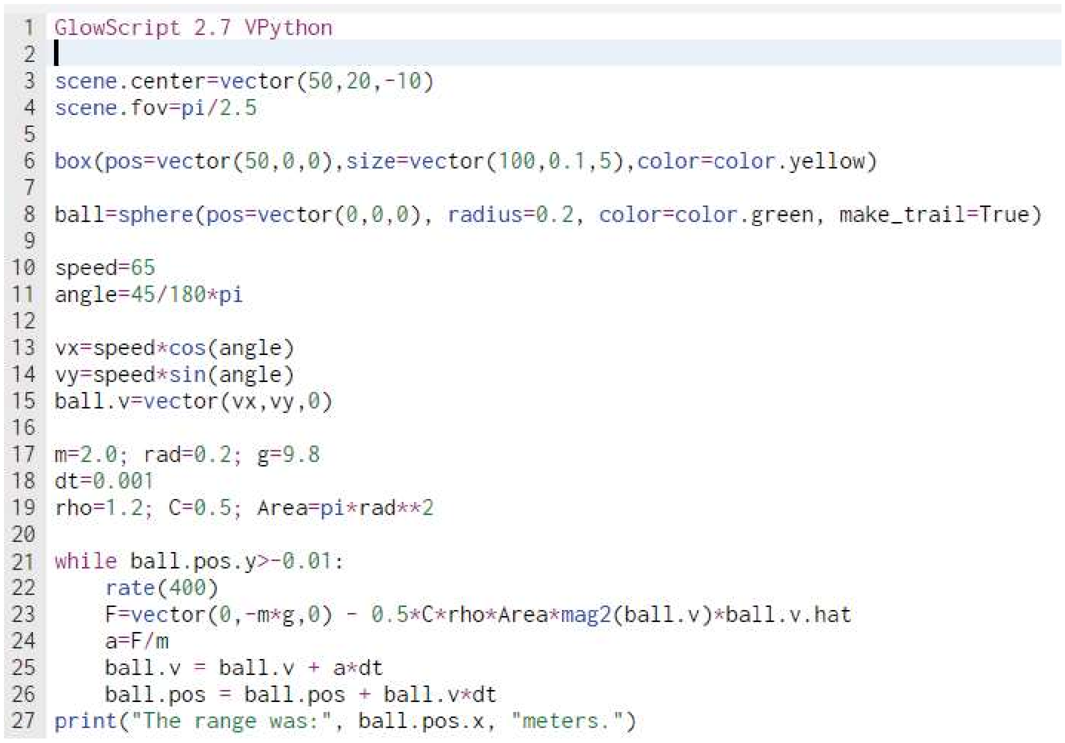
where ρ is the density the medium (here, air), A the cross-sectional area of the projectile, and C_D the coefficient of drag. We see that the magnitude of the drag force is proportional to the square of the speed, and that its direction is opposite to the motion. This latter point is mathematically represented by the last term in Equation (6.6), where $\hat{v}$ is the unit vector in the direction of instantaneous motion, or $\hat{v} = \vec{v}/|v|$.

You might be thinking that the changing direction of the drag force would be difficult to "handle", and ordinarily you would be right. In fact, this facet of the problem coupled with the quadratic dependence on speed makes this problem impossible to solve in closed form with pencil and paper. So here we now encounter our first instance of a problem that has no closed-form analytical solution and where we rely on a computer to give us the solution numerically.

We said "ordinarily" because within the VPython environment there exist high-level commands that will make this problem substantially easier. Particularly nice are the built-in vector operations that include easy evaluations of *dot* and *cross*-products between vectors, as well as evaluations of a vector's magnitude and direction. For our purposes here, we want to highlight two operations we can perform on a vector **A**:

- **mag(A)** and **mag2(A)**, where the output yields the length and length-squared of that vector, respectively.
- **A.hat**, which produces from the vector **A** its unit vector (by dividing it by its length). This syntax treats the directionality (given by the unit vector) as a *property* of the vector. It is just like any other property of a vector, such as **A.x**, which yields the x-component of the vector **A**.

Armed with these commands, implementing Equation (6.6) should seem a lot more straightforward. Let's look at the basic code first (see Figure 6.2).



```
 1 GlowScript 2.7 VPython
 2 
 3 scene.center=vector(50,20,-10)
 4 scene.fov=pi/2.5
 5 
 6 box(pos=vector(50,0,0),size=vector(100,0.1,5),color=color.yellow)
 7 
 8 ball=sphere(pos=vector(0,0,0), radius=0.2, color=color.green, make_trail=True)
 9 
10 speed=65
11 angle=45/180*pi
12 
13 vx=speed*cos(angle)
14 vy=speed*sin(angle)
15 ball.v=vector(vx,vy,0)
16 
17 m=2.0; rad=0.2; g=9.8
18 dt=0.001
19 rho=1.2; C=0.5; Area=pi*rad**2
20 
21 while ball.pos.y>-0.01:
22     rate(400)
23     F=vector(0,-m*g,0) - 0.5*C*rho*Area*mag2(ball.v)*ball.v.hat
24     a=F/m
25     ball.v = ball.v + a*dt
26     ball.pos = ball.pos + ball.v*dt
27 print("The range was:", ball.pos.x, "meters.")
```

Figure 6.2: The basic code simulating a the trajectory of a projectile with air drag

One thing to point out before we delve in is that everything here is in SI-uits. This means that whenever you encounter a number (assigned to a variable), that number should be considered to have the appropriate SI-unit. So, for example, Line 10 states: **speed=10**. This means that we set the speed of the projectile to 10 m/s. Similarly, Line 12 defines the density, **rho = 1.2**, as 1.2 kg/m^3 (the density of air). Since the SI-unit system is closed, any calculations that the code performs will automatically be in the appropriate SI-unit for that variable.

We can start by looking at Lines 13 through 15. Here we first compute the x- and y- components of the velocity vector from the speed and launch angle. Line 15 is interesting: **ball.v** = **vector(vx,vy,0)**. How should we interpret this statement? First, it is an assignment. The velocity vector on the right of the equal sign is assigned to a quantity called **ball.v**. The notation on the left side of the equal sign suggests that **v** is a property of **ball**. In fact, with this statement we define the velocity to be a property of the object we introduced earlier in the code and called **ball**. So now in addition to the properties **pos**, **size**, and **color** by virtue of the **ball** being defined as a sphere, we have added the property **v**.

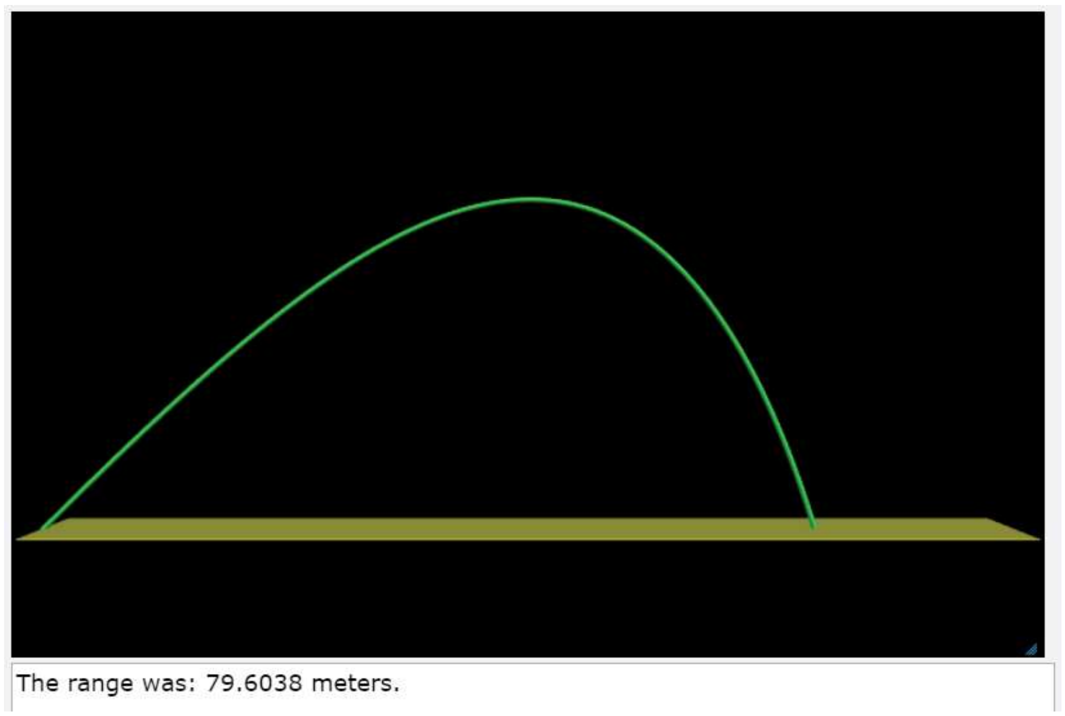
What is interesting is that **ball.v** has itself all the properties that a vector can have, and so, for instance **ball.v.x** would refer to the x-component of the ball's velocity vector. Similarly, **ball.v.hat** would give the unit vector in the direction of motion.

Finally, let's examine Line 23:

F=vector(0,-m*g,0) - 0.5*C*rho*Area*mag2(ball.v)*ball.v.hat.

You should recognize this as the net force acting on the projectile. The first part is the gravitational force near the surface of the earth and the second is the drag force. Notice that VPython automatically adds these two parts as vectors.

When you run the program you should get something like the following:



```
The range was: 79.6038 meters.
```

Notice that the code returns the approximate range of the projectile in the print statement. Also observe the shape of the trajectory. It clearly deviates from the parabola - the highest point in the trajectory is not reached at the horizontal half-way point, and the descent is steeper then the ascent.

6.5 Exercises

- Run the program with different launch angles and determine the angle that gives you the largest range. Without air-drag, it is fairly easy to prove that that angle is 45 degrees. What is it now?
- Let's play a game. The object is to make the range as close to 100 meters as possible. You are only allowed to change the initial launch angle and speed. What combination gets you closest to 100 meters? There may be more than one solution.

- Instead of modifying the initial conditions, as in the previous exercise, now examine the role of the projectile properties. The shape of the projectile enters the problem via the drag coefficient C_D; it can vary from as low as 0.05 (for a very aerodynamic shape) to about 1.0 (for a cube). Adjust first the mass and then the drag coefficient, and describe how these two parameters affect the trajectory.

- Add to the code given above so that you can get two trajectories on screen simultaneously. These two trajectories should correspond to two different projectiles (differentiated either by mass or drag coefficient).

6.6 Third Example: Trajectories of Planets and Comets

Our final example is also the most famous in the sense of historical significance - the Keplerian orbits of planets and comets around the sun. The perfectly circular orbit is one special solution to Newton's second law. This is usually demonstrated in introductory physics by setting the formula for the centripetal force equal to the gravitational force,

$$\frac{mv^2}{r} = \frac{GMm}{r^2}, \tag{6.7}$$

We set them equal because gravitation actually provides us with the centripetal force necessary for moving in a circle. Solving Equation (6.7) for v yields the orbital speed as a function of the orbital radius,

$$v = \sqrt{\frac{GM}{r}} \tag{6.8}$$

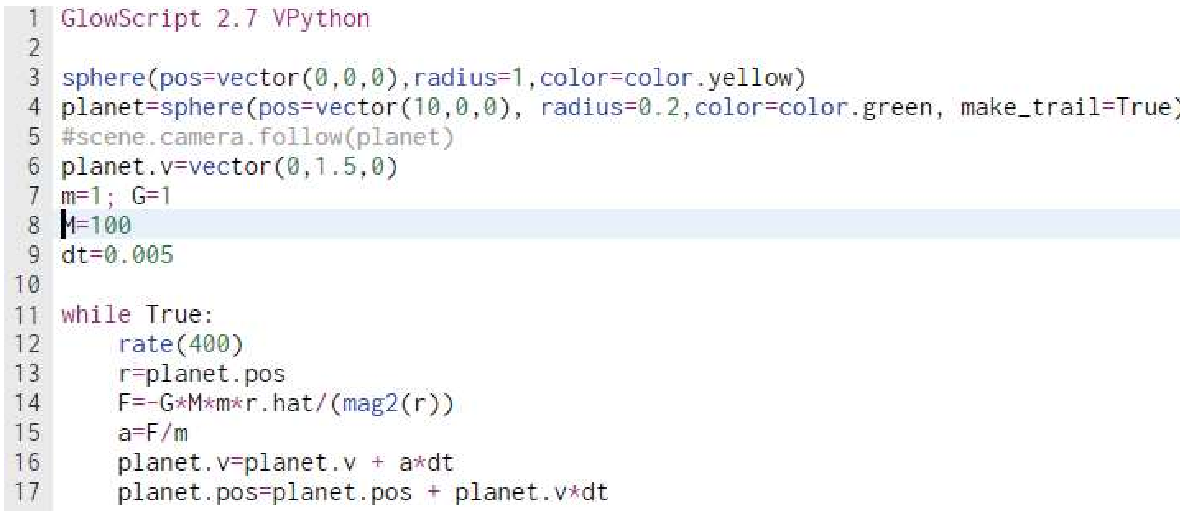
Proving that elliptical orbits also satisfy the governing equations is much more difficult and usually reserved for an junior-level course in classical dynamics. The same goes for parabolic and hyperbolic orbits - the remaining cone sections. However, we can explore those orbits numerically using VPython without any advanced knowledge of physics.

Imagine for a moment that we had the power to launch a comet at a particular distance from the sun, call it r_0, as well as with a certain initial velocity. It is not hard to see that if we chose a velocity of zero, the comet would head straight for the sun; it would accelerate in a straight line toward the sun and be swallowed up by it. In fact, many people unfamiliar with physics think that this scenario is the only one possible and that circular orbits are only feasible due to other planets or stars in the picture, or by some other magic (which is, of course, ludicrous).

But what if we chose an initial velocity at right angles to the line connecting the comet to the sun? We would at least have a chance of obtaining a circular orbit, but only if the speed in this direction matched Equation (6.8). Indeed, this set of initial conditions would produce a circular orbit.

Now imagine what would happen if the speed that we impart to the comet at right angles did not match that speed. What if it were much smaller or larger than what Equation (6.8) demands? It stands to reason that we would then not recover a circular orbit, but what do we get instead? Let's find out by running a numerical simulation!

The basic code is surprisingly short. In this version we decided for simplicity to set the universal gravitational constant to 1, but you can change it to the actual value in SI-units. In that case, however, you should also make the masses and distances involved realistically large. The basic code, then, is shown in Figure 6.6. Feel free to make the parameters more realistic; the mass ratio that appears here is only 1:100, for instance. Nonetheless, we can use this simplified code to explore the possible orbits.



```
GlowScript 2.7 VPython

sphere(pos=vector(0,0,0),radius=1,color=color.yellow)
planet=sphere(pos=vector(10,0,0), radius=0.2,color=color.green, make_trail=True)
#scene.camera.follow(planet)
planet.v=vector(0,1.5,0)
m=1; G=1
M=100
dt=0.005

while True:
    rate(400)
    r=planet.pos
    F=-G*M*m*r.hat/(mag2(r))
    a=F/m
    planet.v=planet.v + a*dt
    planet.pos=planet.pos + planet.v*dt
```

Figure 6.3: The basic code simulating gravitational orbits.

6.7 Exercises

- Examine the code provided, make sure that you understand what each line does, and annotate.
- Use Equation (6.8) with the parameters given in the code to find the orbital speed for a circular orbit. Enter this speed as the starting speed in the code at the appropriate place. Do you get a circular orbit?
- If you doubled the initial distance of the comet from the sun, what speed would be necessary now for a circular orbit? Try it in the code.

- Now select a starting speed that is smaller than the one you chose above. Observe the kind of orbit you obtain now. Does the orbit trace out the same path after each revolution? Astronomers call this scenario a closed orbit.

- What can you say about the orbital speed? Is it constant or does the comet appear to speed up at certain points? Explain!

- If you make the initial speed too small, you will see that the simulation eventually breaks down and returns nonsensical results. Why is that and what can you adjust to remedy the situation? [Hint: When the comet gets very close to the sun, the acceleration it experiences becomes very large. What line in the code is going to be adversely affected?]

- Let's be even more creative. Instead of the normal law of universal gravitation that decreases with the square of distance, what would happen in a universe ruled by a gravitational force proportional to $1/r$? Make this change in the code, and start with a circular (or near circular) orbit by using Equation (6.7) but with the new gravity to solve for orbital speed.

- Now lower the speed and observe the orbits that result. What is the most obvious qualitative difference about the orbits that we get now in this alternative universe?