Integrating Equations of Motion: Orbits

6 June 2020

PHY2004W KDSMIL001

Contents

1 Introduction

In this Assignment, we aim to simulate the Two- and Three-Body problems using vPython. These problems involve analysing the movement of 2 and 3 bodies interacting with each other, in this case at least, by the force of gravity. These are hard, in some cases impossible, to solve analytically so we use numerical methods to try and get an idea of the general qualities of these systems. Note that this does not give us an exact solution. In many cases our solution will diverge from the actual solution as the system is intrinsically chaotic, but there are a few tricks we can employ to reduce this effect.

2 Activity

1. **The Two-Body Problem**
   Before we get to our methods of numerical integration, we must first establish the theory behind the steps we will take. We will be using Newton's equation for gravitational force between two objects:

$$\vec{F}_{12} = -Gm_1m_2\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3} \tag{1}$$

   where $\vec{F}_{12}$ is the force on mass 1 due to mass 2, m is the mass of an object, and $\vec{r}$ is the position vector of an object.

Now we can move on to how we will simulate this system. We will use the Symplectic Euler Method, which involves using the force from the next time step in order to calculate the momentum and position for that time step. This is obviously a discrete method of integration and is prone to deviations from the actual value. This is the divergence we spoke about above. One of the tricks we can use to reduce this effect is, apart from using the Symplectic Euler Method, decreasing the size of our time step.
To start off, we'll run with the most simple initial conditions, even considering one of the masses to be much larger than the other (think of the Sun and the Earth). We mean this as the one mass doesn't move. For the sake of simplicity and size, their masses in the simulation are the same but the Star is fixed in place.

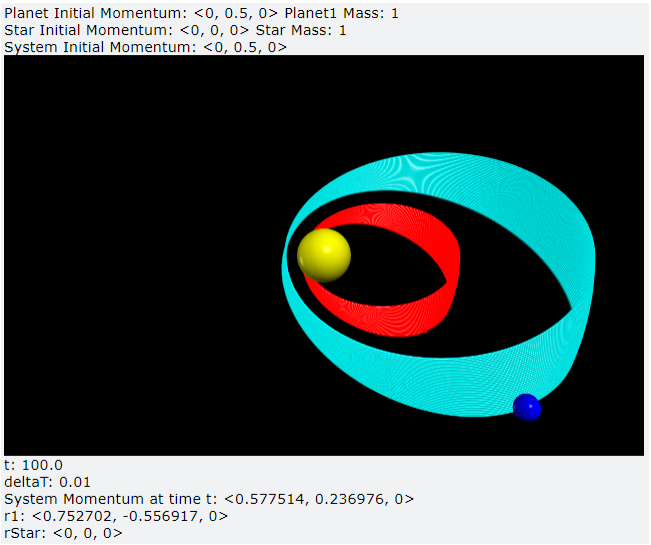


Figure 1: A Small Time Step

The red sphere and trail is the position of the Centre of Mass of the system. As you can see, in Figure 1 the "Earth's" orbit precesses. This happens because, once the system has finished an orbit, due to error in the approximation, the planet is not at exactly the same place as it was when it started. This continues every orbit.If we decrease the time step, we can see that that doesn't happen as much.

Clearly, Figure 2 is much more accurate. This makes sense as our method approximates the solution at each step as a straight line. With a larger time step, there's more room at each step for the approximation to diverge from the actual solution. This shortcoming of our method becomes even more apparent when we consider the system at high velocities. We can try to predict what will happen: Our method approximates the solution as a straight line. At relatively low ve-

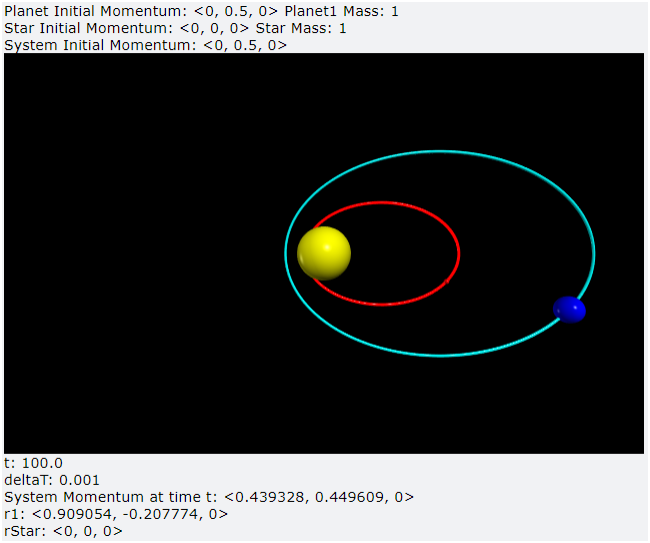


Figure 2: A Smaller Time Step

locities, this means that at each time step the distance covered is quite small, meaning the approximation is close to the actual solution as there isn't much that can go wrong. At higher velocities, the distance covered is much higher, meaning that our straight line approximation can diverge more over the same time step than for smaller distances. We can check this by running 2 identical simulations, apart from the starting momenta. Firstly we'll try a mildly eccentric orbit in Figure 3

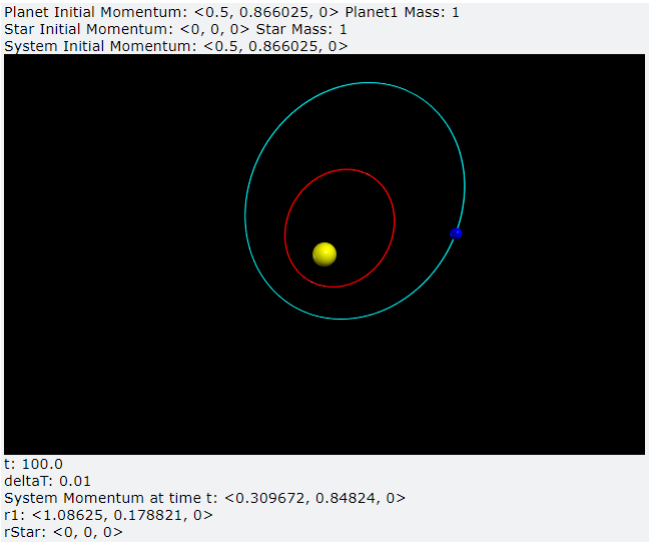


Figure 3: A Mildly Eccentric Orbit

We can see that this is a reasonable system. It doesn't precess, at least not noticeably, even after a long time. It even conserves momentum quite well. Compare that to Figure 4.

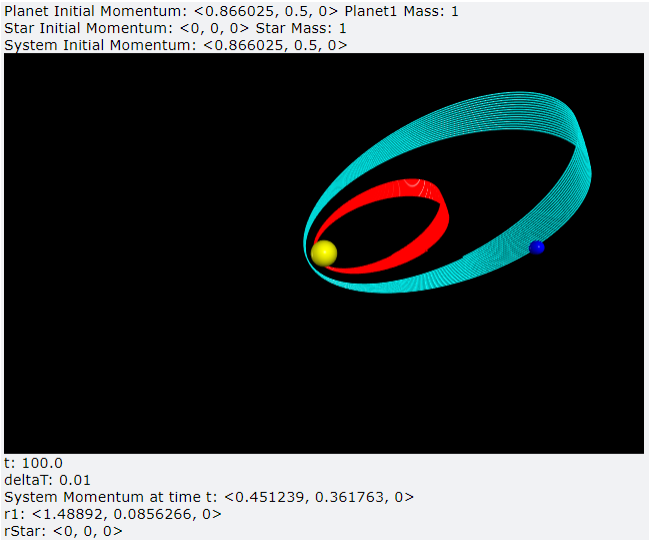


Figure 4: A Wildly Eccentric Orbit

After the same amount of time, the system has clearly precessed quite a bit and the total momentum at the end is completely different to the starting momentum in terms of magnitude. So we now know that our method has some limitations. Decreasing the time step will increase runtime but greatly increase accuracy. We must just consider these facts as we move forward.

Next up, we can see what would happen if, instead of employing a Symplectic Euler Method where we use the following step's momentum to calculate that step's change in position, we first update the position and *then* update the momentum. This is the plain old Euler's Method and it yields some interesting results to think about (Figure 5).

Clearly the program is overestimating the momentum at each time step, leading to growth of orbits.
Next up, we investigate a slight modification to the Potential Energy. It is known that gravitational force is the gradient of potential in the form

$$\vec{F_G} = -\boldsymbol{\nabla} U = \boldsymbol{\nabla} G m_1 m_2 \frac{1}{|\vec{r_2} - \vec{r_1}|} \tag{2}$$

If we change this potential to be in the form

$$U \propto -\frac{1}{r^{1+\epsilon}}$$

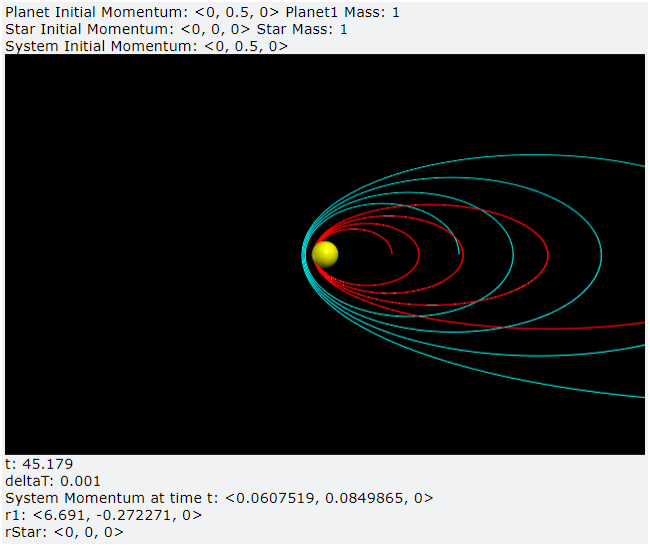


Figure 5: A Smaller Time Step with Euler's Method

where $\epsilon \in [0.1, 0.3]$, we will find the force to be

$$\vec{F_G} = -Gm_1m_2\frac{\epsilon + 1}{|r_2 - r_1|^{\epsilon+3}}\vec{r} \tag{3}$$

Implementing this equation into our program, we get the following results:

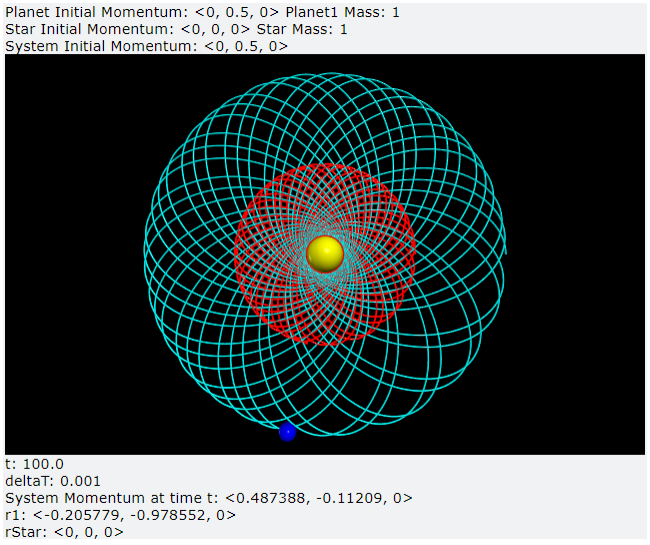


Figure 6: Adjusted Potential with $\epsilon = 0.1$

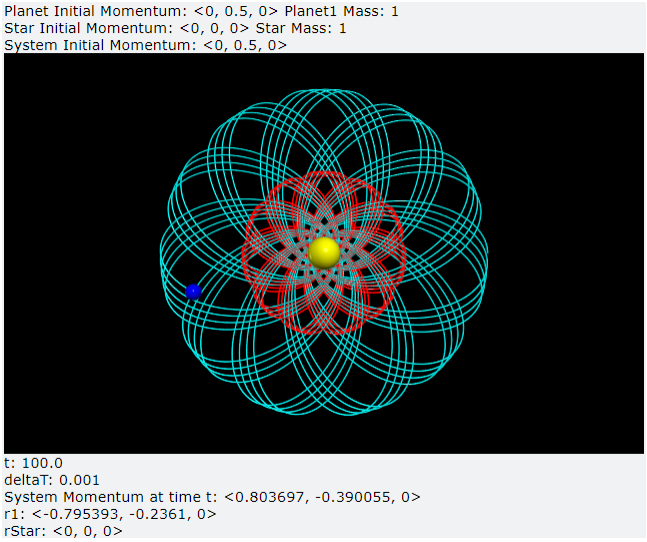


Figure 7: Adjusted Potential with $\epsilon = 0.3$

It seems that, predictably, with higher values of ϵ, the magnitude of the force increases. We expect this as the denominator from Equation 3 grows faster as ϵ grows, so the force should grow too. The precession would come from the $\epsilon + 1$ term.

Now we can finally get to the interesting stuff: Two bodies of similar mass orbiting each other. We shall return to the Symplectic Euler Method, with the potential as it should be. It's relatively simple to convert the program as we are already calculating the force between the bodies, so we need only reverse the direction to find the force on the star. If we let the star feel the force between the two bodies, with the same initial conditions as in Figure 2, we get Figure 8.

This seems to be stable, and is a reasonable system, but the system drifts as the starting momentum is non-zero. This can be fixed in 2 ways. The quick fix is to have starting conditions that give us a total momentum of 0. This is trivial and not really helpful in the grand scheme of things, so we can try another fix. If we adjust the coordinates as below we will always have the centre of mass stay still in our window. This is because it automatically adjusts the momenta to have a total momentum of 0.

```
12 vCoM = (p1+pStar)/(m1+mStar)
13 p1 -= m1*vCoM
14 pStar -= mStar*vCoM
15 # More constants, physical
```

Adjustment of Coordinates

This method with similar conditions gives us Figure 9

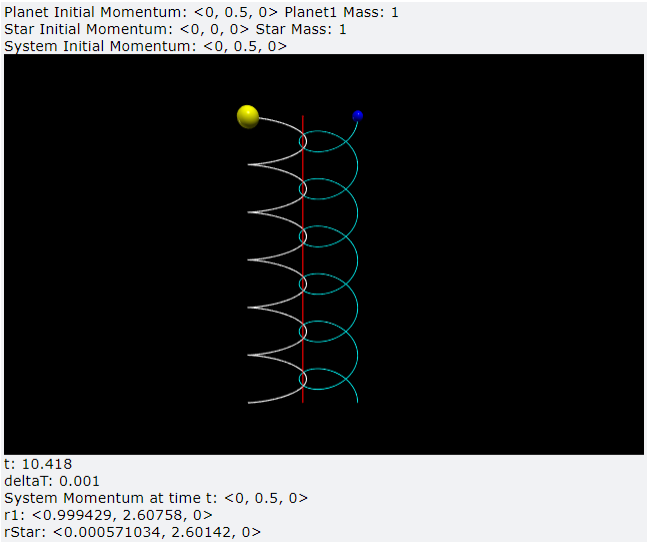


Figure 8: Two-Body System with Star able to move

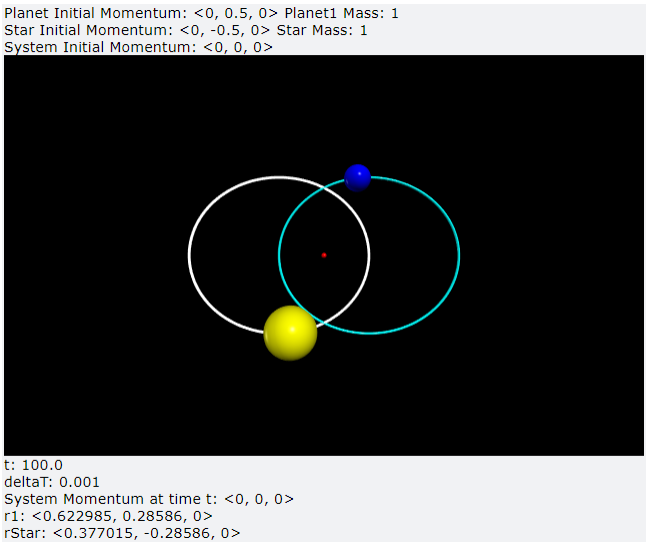


Figure 9: Two-Body System with Adjusted Coordinates

This is a very stable system. Actually, most initial conditions for the Two-Body problem result in stable motion. It's also clear that this system now perfectly conserves momentum, which is a comforting side effect.

2. **The Three-Body Problem**

Next up, we elevate the program to three bodies. On the surface, this seems like a small step up from before. We're only adding 1 body, how bad could it be? Well, trying to solve this analytically can prove to be extremely hard and in some cases impossible. The previous section is solvable by analytical means in a reasonable time period but this problem absolutely requires the use of numerical methods. Thankfully we have already investigated most of the shortcomings of these methods so we can move onto more interesting things.

To simulate this problem we only need to add a few lines of code. We obviously need to create this new body in the scene and give it a mass and momentum, however most importantly we need to add 2 more force calculations in the loop. As before, we could compute the forces for each pair of bodies and directions, but that's unnecessary as 2 bodies exert the same magnitude of force on each other. As a first test of our program, to make sure it's working correctly, we'll investigate a relatively simple system: A star with 2 planets orbiting around it at the same distance but half a period away from each other.

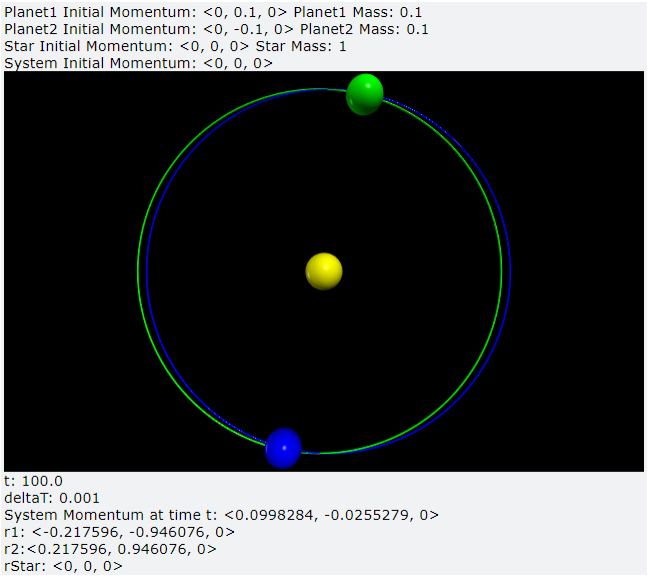


Figure 10: Three-Body Problem with Simple Conditions

This system is stable but that's because the initial conditions are very carefully chosen to not introduce too many opportunities for errors to show up, such as the bodies coming close together and thus moving at high velocities. They were also chosen in order to keep the system stable in another way. It turns out that the Three-Body Problem is what we call chaotic, meaning that it is extremely sensitive to initial conditions. This is why it's so hard to solve analytically. We

can see this in action by changing the mass of one of the planets by one part in 10^9:

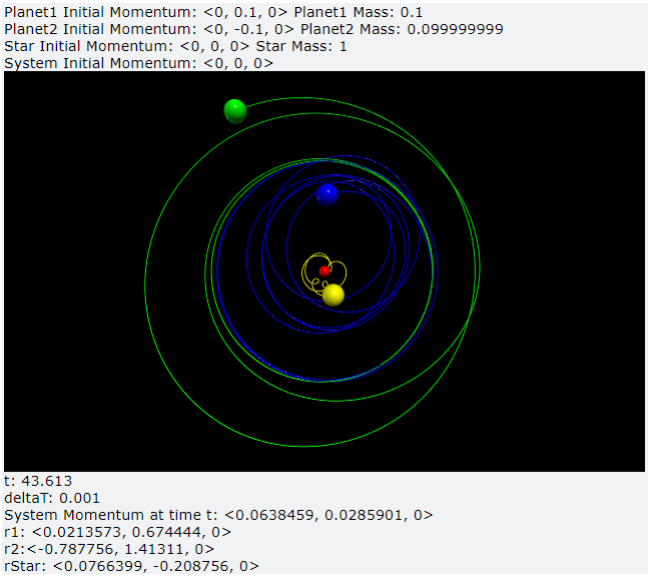
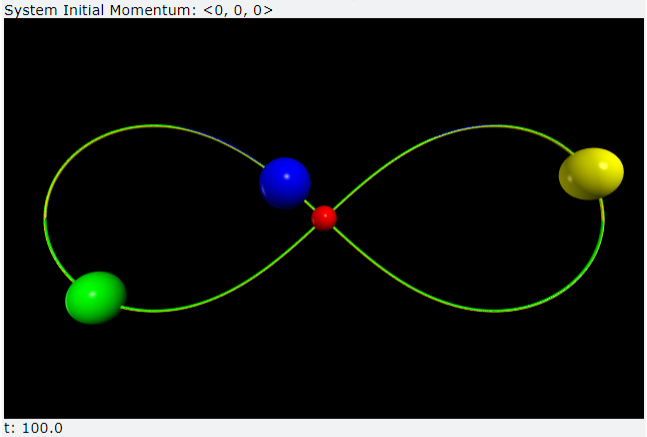


Figure 11: Slightly Adjusted Initial Conditions

After less than half the amount of time as before, this system has fully devolved into unpredictable and chaotic behaviour.
So we know that this system is incredibly sensitive to initial conditions, let's see what interesting things we can do. Figure 12 is quite cool and extremely hard to find, but surprisingly stable. Changing initial conditions even by one part in 10^3 doesn't do much to affect the system, at least in the time frame we're looking at. Finally, we'll look at a semi-realistic system, changing masses and momenta to something that we're likely to find in reality (Figure 13). This system is stable but any adjustment to something like the mass or momenta of the stars will greatly affect the stability, leading the system to devolve into something chaotic and wild.

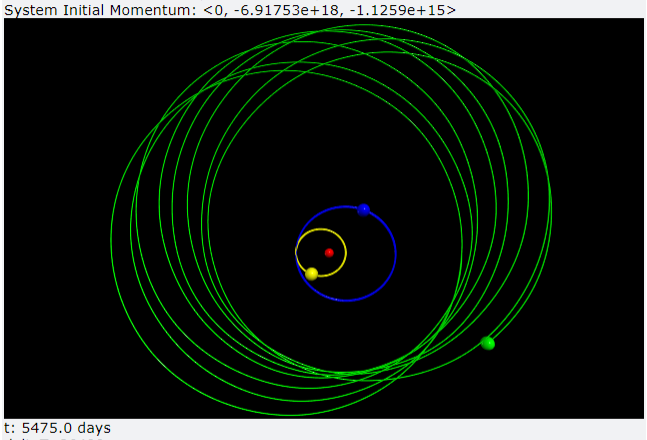
Planet1 Initial Momentum: <0.466204, 0.432366, -0> Planet1 Mass: 1
Planet2 Initial Momentum: <0.466204, 0.432366, -0> Planet2 Mass: 1
Star Initial Momentum: <-0.932407, -0.864731, 0> Star Mass: 1
System Initial Momentum: <0, 0, 0>



t: 100.0
deltaT: 0.001
System Momentum at time t: <0.678217, -0.351032, 0>
r1: <-0.149708, 0.133822, 0>
r2:<-0.880089, -0.305658, 0>
rStar: <1.0298, 0.171836, 0>

Figure 12: An Interesting System

Planet1 Initial Momentum: <0, 1.99574e+34, -3.99202e+30> Planet1 Mass: 1e+30
Planet2 Initial Momentum: <0, -2.28934e+32, 1.1976e+31> Planet2 Mass: 6e+27
Star Initial Momentum: <0, -1.97285e+34, -7.98403e+30> Star Mass: 2e+30
System Initial Momentum: <0, -6.91753e+18, -1.1259e+15>



t: 5475.0 days
deltaT: 86400
System Momentum at time t: <1.20821e+32, 1.0377e+32, 8.03285e+30>
r1: <9.99374e+10, 6.49771e+10, 1.0879e+08>
r2:<2.86595e+11, -1.36709e+11, 1.05936e+09>
rStar: <2.49215e+10, -3.20784e+10, -5.75729e+07>

Figure 13: A Realistic Binary Star System

3 Appendix

Appendix 1: Two-Body Problem Code

```
import numpy as np
from numpy import cos, pi, sin, sqrt, exp, random
from vpython import *
# Constants to be used later, such as start positions and start momenta
r1 = vector(1,0,0)
rStar = vector(0,0,0)
p1 = vector(0,1,0)
pStar = vector(0,0,0)
mStar = 1
m1 = 1
# Converting to CoM reference frame
vCoM = (p1+pStar)/(m1+mStar)
p1 -= m1*vCoM
pStar -= mStar*vCoM
# More constants, physical
deltaT = 0.001
t = 0
G = 1
rCoM = (m1*r1+mStar*rStar)/(m1+mStar)
# Creating the objects to be drawn on the canvas, as well as settings for the canvas
star = sphere(pos=rStar, radius=0.1, color=color.yellow)
planet1 = sphere(pos=r1, radius=0.05, color=color.blue)
CoM = sphere(pos=rCoM, radius=0.01, color=color.red)
starTrail = curve(color=vector(99,99,59), radius=0.005)
planet1Trail = curve(color=color.cyan, radius=0.005)
CoMTrail = curve(color=vector(99,0,0), radius=0.005)
scene.autoscale = 0
scene.title = 'Planet Initial Momentum: '+str(p1)+' Planet1 Mass: '+str(m1)+'\nStar Initial Momentum: '+str(pStar)+' Star Mass: '+str(mStar)\
    +'\nSystem Initial Momentum: '+str(p1+pStar)
scene.camera.follow(CoM)
# The loop that calculates new positions based on forces
while t < 100:
    rate(1000)
    # Updating positions and trails
    starTrail.append(pos=rStar)
    planet1Trail.append(pos=r1)
    CoMTrail.append(pos=rCoM)
    star.pos=rStar
    planet1.pos=r1
    CoM.pos=rCoM
    # Force calculations
    F1Star = -G*mStar*m1*(r1-rStar)*(1)/(mag(r1-rStar)**(3))
```

```
    # Updating star stuff
    pStar += deltaT*(-F1Star)
    rStar += pStar*deltaT/mStar
    # Updating planet1 stuff
    p1 += F1Star*deltaT
    r1 += p1*deltaT/m1
    # Finding new CoM position from position of masses
    rCoM = (m1*r1+mStar*rStar)/(m1+mStar)
    # Housekeeping
    t += deltaT
    scene.caption = 't: '+str(round(t,3))+'\ndeltaT: '+str(deltaT)+'\
        nSystem Momentum at time t: '+str(p1+pStar) \
         +'\nr1: '+str(r1)+'\nrStar: '+str(rStar)
```

Appendix 2: Three-Body Problem Code

```
import numpy as np
from numpy import cos, pi, sin, sqrt, exp, random
from vpython import *
# Constants to be used later, such as start positions and start
    momenta
r1=vector(1e11,0,0)
r2=vector(2.5e11,0,0)
rStar=vector(0,0,0)
m1=1e30
m2=6e27
mStar=2e30
G = 6.67e-11
p1=m1*vector(0,sqrt(G*mStar/mag(r1-rStar)),0)
p2=m2*vector(0,-sqrt(G*(mStar+m1)/mag(r2-rStar)),2000)
pStar=vector(0,0,0)
rCoM = ((m1*r1)+(m2*r2)+(mStar*rStar))/(m1+m2+mStar)
# Converting to CoM reference frame
vCoM = (p1+p2+pStar)/(m1+m2+mStar)
p1 -= m1*vCoM
p2 -= m2*vCoM
pStar -= mStar*vCoM
# More constants, physical
deltaT = 86400
t = 0
# Creating the objects to be drawn on the canvas, as well as settings
    for the canvas
planet1 = sphere(pos=r1, radius=1e10, color=color.blue)
planet2 = sphere(pos=r2, radius=1e10, color=color.green)
star = sphere(pos=rStar, radius=1e10, color=color.yellow)
CoM = sphere(pos=rCoM, radius=7e9, color=color.red)
planet1Trail = curve(color=color.blue, radius=1e9)
planet2Trail = curve(color=color.green, radius=1e9)
starTrail = curve(color=color.yellow, radius=1e9)
CoMTrail = curve(color=color.red, radius=1e9)
```

```
scene.autoscale = 0
scene.title = 'Planet1 Initial Momentum: '+str(p1)+' Planet1 Mass: '+
    str(m1)+'\nPlanet2 Initial Momentum: '+str(p2)+' Planet2 Mass: '+
    str(m2)+\
     '\nStar Initial Momentum: '+str(pStar)+' Star Mass: '+str(mStar)+
        '\nSystem Initial Momentum: '+str(p1+p2+pStar)
# The loop that calculates new positions based on forces
while t < deltaT*365*15:
    rate(100)
    # Updating positions and trails
    planet1Trail.append(pos=r1)
    planet2Trail.append(pos=r2)
    starTrail.append(pos=rStar)
    CoMTrail.append(pos=rCoM)
    planet1.pos=r1
    planet2.pos=r2
    star.pos=rStar
    CoM.pos=rCoM
    # Force calculations
    F1Star = -G*mStar*m1*(r1-rStar)/(mag(r1-rStar)**3)
    F12 = -G*m1*m2*(r1-r2)/(mag(r1-r2)**3)
    F2Star = -G*m2*mStar*(r2-rStar)/(mag(r2-rStar)**3)
    # Updating planet1 position and momentum
    p1 += (F1Star+F12)*deltaT
    r1 += p1*deltaT/m1
    # Updating planet2 position and momentum
    p2 += (F2Star-F12)*deltaT
    r2 += p2*deltaT/m2
    # Updating star position and momentum
    pStar += (-F1Star-F2Star)*deltaT
    rStar += pStar*deltaT/mStar
    # Finding new CoM position from positions of masses
    rCoM = ((m1*r1)+(m2*r2)+(mStar*rStar))/(m1+m2+mStar)
    # Housekeeping
    t += deltaT
    scene.caption = 't: '+str(round((t/86400),3))+' days'+'\ndeltaT:
        '+str(deltaT)+'\nSystem Momentum at time t: '+str(p1+pStar) \
         +'\nr1: '+str(r1)+'\nr2:'+str(r2)+'\nrStar: '+str(rStar)
```