

Solution to Newtonian Gravity problem with many bodies

Computational Physics-Phy905

Project 3

Crispin Contreras

May 8, 2016

Abstract

This paper discusses the numerical solution to Newtonian Gravity with different planets in our solar system. Two methods were used to solve the problem one was the Verlet method and the other one was the Runge-Kutta to the fourth order (RK4). From these two methods we found that the Verlet was more accurate and it was slightly faster in computational time.

1 Introduction

I start by solving the equations for Newtonian gravity. I begin with the simple Sun-Earth system and then move on to solve the Sun-Earth-Jupiter system with the Sun fixed and then the Sun-Earth-Jupiter with the Sun not fixed. Finally I model all the planets including Pluto. I used two different methods to solve this problem which are the Verlet and Runge-Kutta to the fourth order (RK4). These are just Taylor expansions and I discuss more about them in the Methods section. I test to see how stable the systems are by looking at different time steps and also increasing the time that the planets orbit. Finally include my results and the conclusions that I arrived to.

2 Theory

2.1 Earth-Sun System

I start with the Newtonian gravity which is given by $\mathbf{F} = -\frac{GM_1M_2}{r^2}\hat{\mathbf{r}}$. Here G is the gravitational constant, r is the distance between the bodies, and M represents the mass. I decided to work in Cartesian coordinates so I use

$$\begin{aligned}\hat{\mathbf{r}} &= \cos(\theta)\hat{\mathbf{x}} + \sin(\theta)\hat{\mathbf{y}} \\ x &= r\cos(\theta), \quad y = r\sin(\theta) \quad r = \sqrt{x^2 + y^2}\end{aligned}\tag{1}$$

Where θ is just the polar angle. Now using equation (1) I break the equation for gravitation into Cartesian coordinates

$$\begin{aligned}F_x &= -\frac{GM_1M_2}{r^3}x \\ F_y &= -\frac{GM_1M_2}{r^3}y\end{aligned}\tag{2}$$

With equation (2) now I can start setting up the equations of motions for the planets. I start with the Earth-Sun system where I treat the Sun as stationary so I use it as the origin. I start with the equations of motion, for simplicity I will only do the x coordinate since the only thing that changes in the others is the coordinate itself.

$$M_{Earth} \frac{d^2 x}{dt^2} = - \frac{GM_{Earth} M_{Sun}}{r^3} x$$

I will simplify this equation further by introducing a new set of units for the G and mass. The Earth's orbit around the sun is almost circular around the Sun so for this type of motion the force is given by

$$F = \frac{M_{Earth} v^2}{r} = \frac{GM_{Sun} M_{Earth}}{r^2}$$

from this equation we can get rid of the gravitational constant G and mass of the sun to replace it by

$$v^2 r = GM_{Sun} = 4\pi^2 \left(\frac{AU^3}{yr^2} \right) \quad (3)$$

with this transformation now we can use the astronomical units (AU) for length and year (yr) for time. The mass of the Sun is then one. Now the equation of motion for the earth reads

$$F_x = \frac{d^2 v_x}{dt^2} = - \frac{4\pi^2}{r^3} x$$

$$\frac{dx}{dt} = v_x \quad (4)$$

The equation for the y coordinate is the same except the x is replace by y.

2.2 Three Body Problem

I start with a simplified version of the three body problem. In this case I model the Sun, Earth, and Jupiter but I will keep the Sun fixed. For the Earth the only thing that changes now is that it feel the force from Jupiter. The force now reads

$$F_x^{Earth} = \frac{dv_{x_E}}{dt} = - \frac{4\pi^2}{r_{ES}^3} x_E - \frac{4\pi^2 (M_{Jupiter}/M_{Sun})}{r_{EJ}^3} (x_E - x_J) \quad (5)$$

where now include the coordinates for Jupiter, r_{EJ} is the distance between the Earth and Jupiter, and r_{ES} is the distance between the Earth and the Sun. Here x_J is the position of Jupiter and x_E is the position of the Earth and M represents the mass of the bodies. The same equation for (5) is obtained for y except the x is replace by either y. r_{EJ} is given by

$$r_{EJ} = \sqrt{(x_E - x_J)^2 + (y_E - y_J)^2}$$

For Jupiter I obtain the equation

$$F_x^{Jupiter} = \frac{dv_{x_J}}{dt} = - \frac{4\pi^2}{r_{JS}^3} x_J - \frac{4\pi^2 (M_{Earth}/M_{Sun})}{r_{EJ}^3} (x_J - x_E) \quad (6)$$

here r_{JS} is the distance between Jupiter and the Sun, in order to get the y coordinate I just replace x by y.

Now to do the full three body problem I will allow the Sun to move instead of being fixed. This means that the origin of the system is now the center of mass of the three bodies. I give the equation for the Sun

$$F_x^{Sun} = \frac{dv_{x_S}}{dt} = -\frac{4\pi^2(M_{Jupiter}/M_{Sun})}{r_{JS}^3}(x_S - x_J) - \frac{4\pi^2(M_{Earth}/M_{Sun})}{r_{SE}^3}(x_S - x_E) \quad (7)$$

for equations (5) and (6) the only thing that gets modified is the term x_J which goes to $(x_J - x_S)$ and x_E which goes to $(x_E - x_S)$, the same thing occurs for the y variable. With equations (5),(6),and (7) now I can solve the three body problem in two dimensions.

Finally to model the all the planets of the solar system and Pluto, I keep the sun fixed for simplicity. In general the equation is given by

$$F_x^j = -\frac{4\pi^2}{r_{jS}^3}(x_j - x_S) - 4\pi^2 \sum_{i=1, j \neq i}^9 \frac{M_i/M_{Sun}}{r_{ji}^3}(x_j - x_i) \quad (8)$$

here j and i can take on the values $j, i = 1, 2, 3...9$ and they represent how far the planet is from the Sun. For example Mercury will be 1, Venus 2, and so on. Now that I have all the equations setup I can move on to describe the methods.

3 Methods

Two methods were implemented to solve the equations above. I used the Verlet method which has an accuracy of h^5 . I also used the Runge-Kutta 4 which is more precise than the Verlet but has more floating point operations. The basic idea behind these two methods is to use a Taylor expansion and for each one have a different truncation. I take much of the derivations from [1].

3.1 Velocity Verlet Method

In order to derive the algorithm for the Verlet I will start by using Newton's second law in one dimension which reads

$$m \frac{d^2x}{dt^2} = F(x, t)$$

this can be rewritten in terms of couple equation such that

$$\frac{dx}{dt} = v_x \quad \text{and} \quad \frac{dv}{dt} = F(x, t)/m = a(x, t)$$

I also define the time step which I'm going to use which is $h = \frac{t_f - t_i}{n}$, here n is the step size. We want h to be small so we can perform a taylor expansion such that

$$x(t + h) = x(t) + hx'(t) + \frac{h^2}{2}x''(t) + \mathcal{O}(h^3)$$

now I will introduce the notation for discretized equation in terms of h.

$$x(t_i \pm h) = x_{i \pm 1} \quad x_i = x(t_i) \quad (9)$$

here i can range from 0, which is the initial time, to n . Now I will do the expansion in this discretized for the position and velocity

$$\begin{aligned}x_{i+1} &= x_i + hx'_i + \frac{h^2}{2}x''_i + \mathcal{O}(h^3) \\v_{i+1} &= v_i + hv'_i + \frac{h^2}{2}v''_i + \mathcal{O}(h^3)\end{aligned}\tag{10}$$

from Newton's second law then we have

$$v'_i = \frac{d^2x_i}{dt^2} = F(x_i, t_i)/m$$

From equation (10) we see that for the position we know all the variable up to the second order but for velocity we don't v'' . In order to address this problem we have to make another approximation given by

$$v'_{i+1} = v'_i + hv''_i + \mathcal{O}(h^2)$$

with this approximation I obtain a value for v''_i which is $v''_i \approx v'_{i+1} - v'_i$. Now I know the value of v'' in terms of v' , I substitute this into the velocity equation in (10) which give me

$$v_{i+1} = v_i + \frac{h}{2}(v'_{i+1} + v'_i) + \mathcal{O}(h^3)\tag{11}$$

with equation (10) and (11) we are now able to solve the problem. Equations (10) and (11) transform to

$$\begin{aligned}x_{i+1} &= x_i + hv_i - \frac{h^2}{2} \frac{4\pi^2}{r_i^3} x_i \\v_{i+1} &= v_i - \frac{h}{2} \left(\frac{4\pi^2}{r_{i+1}^3} x_{i+1} + \frac{4\pi^2}{r_i^3} x_i \right)\end{aligned}\tag{12}$$

Equation (12) is used for the calculation of the Earth-Sun system and it's only for the x coordinate. Using this algorithm we see that the order of precision is $\mathcal{O}(h^3)$. The same is done for the y coordinate and for the other systems this same algorithm is used but with different forces. This is shown in listing 1.

```

1 //This calculates the position of the planets using the Verlet Method
2 void solver::verlet(planet &N, int type)
3 {
4     double h = (N.t_f - N.t_i)/N.n;
5     double Fx, Fy, Fz, Fx1, Fy1, Fz1;
6     double acc[tot_p][3];
7     double Nacc[tot_p][3];
8     double rel_pos[3];
9     double time;
10
11     char Ve[30];
12     char Energy2[30]; //contains the Kinetic energy, potential energy, and
13         angular mom.
14         cout<<"Enter the name of the Verlet file: ";
15         cin>>Ve;
16     cout <<"Enter the name of the Energy file: ";

```

```

17  cin>>Energy2;

19  std::ofstream  ofile (Ve);
    std::ofstream  E_output (Energy2);
21  Header_Pos (type , ofile );
    Header_Energy ( type , E_output );

23

24  //Write initial values to the file
25  time= 0.0;
    Write_Pos ( ofile , time );
27  Write_Energy ( E_output , time , type );

29  //Start the clock
    clock_t  start_VV , finish_VV ;
31  start_VV = clock ();

33      for (int i=0; i< N.n; i++)
    {
35          time=(i+1)*h;
          for (int j=0; j<tot_p; j++)
37          {
              planet &este = all_planets[j];
39              Fx=Fy=Fz=Fx1=Fy1=Fz1=0;
              Force (Fx,Fy,Fz, este.pos[0] , este.pos[1] , este.pos[2] , 1 );
41              if (type>0)
              {
43                  for (int l=0; l<tot_p; l++)
                  {
45                      if (l == j)
                      {
47                          Fx+=0;
                          Fy+=0;
49                          Fz+=0;
                      }

51                      else
                      {
53                          planet &otro = all_planets[l];
                          for (int d =0; d<3; d++)
55                          {
                              rel_pos[d] = este.pos[d] - otro.pos[d];
57                          }
                              Force (Fx,Fy,Fz, rel_pos[0] , rel_pos[1] , rel_pos[2] , otro.mass );
59                      }
                  }

61              }

63              acc[j][0] = Fx;
              acc[j][1] = Fy;
65              acc[j][2] = Fz;

67              //Update the position .
              for (int l=0; l<3; l++)
69              {
                  este.pos[l] += h*este.vel[l] +0.5*h*h*acc[j][l];
71              }

73              //Values for the velocity

```

```

75     Force(Fx1,Fy1,Fz1,este.pos[0],este.pos[1],este.pos[2],1);
76     if(type>0)
77     {
78         for(int l=0; l<tot_p; l++)
79         {
80             if(l == j)
81             {
82                 Fx1+=0;
83                 Fy1+=0;
84                 Fz1+=0;
85             }
86             else
87             {
88                 planet &otro = all_planets[l];
89                 for(int d =0; d<3; d++)
90                 {
91                     rel_pos[d] = este.pos[d] - otro.pos[d];
92                 }
93                 Force(Fx1,Fy1,Fz1,rel_pos[0],rel_pos[1],rel_pos[2],otro
94                 .mass);
95             }
96         }
97     }
98     Nacc[j][0] = Fx1;
99     Nacc[j][1] = Fy1;
100     Nacc[j][2] = Fz1;
101     //Calculate new velocity
102     for(int l=0; l<3; l++)
103     {
104         este.vel[l] += 0.5*h*(acc[j][l] + Nacc[j][l]);
105     }
106 }
107 //Write the Updated the values
108 Write_Pos(ofile,time);
109 Write_Energy(E_output,time,type);
110 }

```

Listing 1: This shows how velocity Verlet method is implemented.

3.2 Runge-Kutta 4 method

The idea behind the Runge-Kutta method is similar to the velocity verlet methods in that again we use a Taylor approximation of the function we want to find. But it's more accurate since there are several more approximations made than in the velocity Verlet. I start with the a general function

$$\frac{dy}{dt} = f(t, y)$$

this type of function can be solve by integrating and the results with a discretized function as in the Verlet method is

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y) dt$$

the next critical approximation comes from the integral part of the equation. To approximate the integral we use Simpson's rule which is given by

$$\int_{t_i}^{t_{i+1}} f(t, y) dt \approx \frac{h}{6} [f(t_i, y_i) + 4f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})]$$

In this method we split the midpoint evaluation into two which gives us

$$\int_{t_i}^{t_{i+1}} f(t, y) dt \approx \frac{h}{6} [f(t_i, y_i) + 2f(t_{i+1/2}, y_{i+1/2}) + 2f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})]$$

To solve for the values of f at different at different times we use the predictor corrector methods which consists of finding the slope at the function t_i which is given by $k_1 = f(t_i, y_i)$. Then make a prediction for the solution using Euler's method and use this prediction to compute a new slope at this time. After finding the prediction for the slopes then we average the results. In the Runge-Kutta method to the fourth order (RK4) we do this four times. To help us visualize this better I will use figure 1 which shows where I will be taking the approximations.

We begin by looking at time t_i and finding the slope here, the slope here is given by $k_1 = hf(t_i, y_i)$. The next time step is then $t_{i+1/2}$. In this time step we will make two prediction for the slope which will be approximated using Euler's method. For the predictions we have

$$y_{(i+1/2), Prediction1} = y_i + \frac{h}{2} \frac{dy}{dt} = y_i + \frac{h}{2} f(t_i, y_i) = y_i + \frac{k_1}{2} \quad (13)$$

with equation (13) I can now find the second estimate for slope which is given by

$$k_2 = f(t_{i+1/2}, y_{(i+1/2), Prediction1}) = f(t_{i+1/2}, y_i + \frac{k_1}{2})$$

we make another prediction to find the next slope this is then

$$y_{(i+1/2), Prediction2} = y_i + \frac{h}{2} \frac{dy}{dt} = y_i + \frac{h}{2} f(t_{i+1/2}, y_{(i+1/2), Prediction1}) = y_i + \frac{k_2}{2} \quad (14)$$

and for the next slope we have

$$k_3 = hf(t_{i+1/2}, y_{(i+1/2), Prediction2}) = f(t_{i+1/2}, y_i + \frac{k_2}{2})$$

and finally for the last slope we have

$$k_4 = hf(t_{i+1}, y_{(i+1), Prediction3}) = f(t_{i+1}, y_i + k_3)$$

here we use the approximation for Prediction 3

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2})$$

which is know as the midpoint formula. Now we have all the values the slopes and we can use Simpson's rule above to arrive at the equation

$$y_{i+1} = y_i + \frac{h}{6} [k_1 + 2k_2 + 2k_3 + k_4] \quad (15)$$

with equation (16) we can now implement the RK4 method.

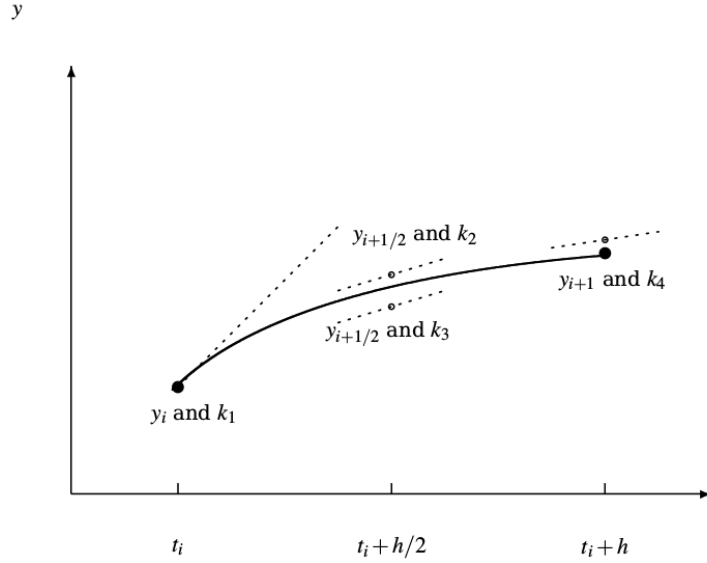


Figure 1: Plot of a general function and the different approximations taken in order to find the different slopes [1]

Equation (16) changes with the coordinate you are using and also the force that each planet feels so this has to be modified also. In our case we start with the equation for velocity which means we have to use the method twice. This means we find the k for the positions and the velocities. The way this algorithm is implemented is shown in the following listing.

```

// Calculates the position using the RK4 method.
2 void solver::RK4(planet &N, int type)
{
4     double h = (N.t_f - N.t_i)/N.n;
    double Fx, Fy, Fz;
6     double rel_pos[3];
    double time;
8
    //Setting up ks, the first [] is the total planets to be solved for and
    //the second is the number of dimensions
10    double k1_v[tot_p][3], k2_v[tot_p][3], k3_v[tot_p][3], k4_v[tot_p][3];
12    double k1_x[tot_p][3], k2_x[tot_p][3], k3_x[tot_p][3], k4_x[tot_p][3];
14
    //Initializes Writing
16    char RK4[30];
    char Energy[30]; //contains the Kinetic energy, potential energy, and
    angular mom.
18    cout<<"Enter the name of the RK4 file: ";
    cin>>RK4;
20    cout <<"Enter the name of the Energy file: ";
    cin>>Energy;
22
    std::ofstream ofile(RK4);
24    std::ofstream E_output(Energy);
    Header_Pos(type, ofile);
26    Header_Energy(type, E_output);

```



```

28 //Write initial values to the file
    time= 0.0;
30    Write_Pos(ofile ,time);
    Write_Energy(E_output,time ,type);
32
33 //Start Clock
34 clock_t start_RK, finish_RK;
    start_RK = clock();
36
37 //Setting up the k values
38 for(int i=0; i<N.n; i++)
39 {
40     time=(i+1)*h;
41
42     //Seting up K1
43     for(int j=0; j<tot_p; j++)
44     {
45         planet &este=all_planets[j];
46         Fx=Fy=0.0;
47
48         Force(Fx,Fy,Fz,este.pos[0],este.pos[1],este.pos[2],1);
49
50         if(type >0)
51         {
52             for(int l=0; l<tot_p; l++)
53             {
54                 if(j == l)
55                 {
56                     Fx+=0.0;
57                     Fy+=0.0;
58                     Fz+=0.0;
59                 }
60                 else
61                 {
62                     planet &otro=all_planets[l];
63                     for(int a=0; a<3;a++){rel_pos[a]=-(otro.pos[a]-este.pos[a]);}
64                     Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
65                 }
66             }
67         }
68
69         k1_v[j][0] = h*Fx;
70         k1_v[j][1] = h*Fy;
71         k1_v[j][2] = h*Fz;
72         for(int l= 0; l<3;l++)
73         {
74             k1_x[j][l] = h*este.vel[l];
75         }
76     } //End of loop
77
78 //Setting up K2
79 for(int j=0; j<tot_p; j++)
80 {
81     planet &este=all_planets[j];
82     Fx=Fy=0.0;
83
84

```

```

86     for (int a=0; a<3;a++)
87     {
88         rel_pos[a]=este.pos[a]+k1_x[j][a]/2.0;
89     }
90     Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
91
92     if (type >0)
93     {
94         for (int l=0; l<tot_p; l++)
95         {
96             if (j == 1)
97             {
98                 Fx+=0.0;
99                 Fy+=0.0;
100                 Fz+=0.0;
101             }
102             else
103             {
104                 planet &otro=all_planets[l];
105                 for (int a=0; a<3;a++)
106                 {
107                     rel_pos[a]=-((otro.pos[a]+k1_x[l][a]/2.0)-(este.pos[a]
108                     ]+k1_x[j][a]/2.0));
109                 }
110                 Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.
111                 mass);
112             }
113         }
114
115         k2_v[j][0] = h*Fx;
116         k2_v[j][1] = h*Fy;
117         k2_v[j][2] = h*Fz;
118         for (int l= 0; l<3;l++)
119         {
120             k2_x[j][l] = h*(este.vel[l]+ k1_v[j][l]/2.0);
121         }
122     } //End of loop
123
124     //Setting up K3
125     for (int j=0; j<tot_p; j++)
126     {
127         planet &este=all_planets[j];
128         Fx=Fy=0.0;
129
130         for (int a=0; a<3;a++)
131         {
132             rel_pos[a]=este.pos[a]+k2_x[j][a]/2.0;
133         }
134         Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
135
136         if (type>0)
137         {
138             for (int l=0; l<tot_p; l++)
139             {
140                 if (j == 1)
141                 {

```

```

140     Fx+=0.0;
141     Fy+=0.0;
142     Fz+=0.0;
143     }
144     else
145     {
146         planet &otro=all_planets[1];
147         for(int a=0; a<3;a++)
148         {
149             rel_pos[a]=-((otro.pos[a]+k2_x[1][a]/2.0)-(este.pos[
150 a]+k2_x[j][a]/2.0));
151         }
152         Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.
153 mass);
154     }
155 }
156
157     k3_v[j][0] = h*Fx;
158     k3_v[j][1] = h*Fy;
159     k3_v[j][2] = h*Fz;
160     for(int l= 0; l<2;l++)
161     {
162         k3_x[j][l] = h*(este.vel[l]+ k2_v[j][l]/2.0);
163     }
164     }//End of loop
165
166     //Setting up K4
167     for(int j=0; j<tot_p; j++)
168     {
169         planet &este=all_planets[j];
170         Fx=Fy=Fz=0.0;
171
172         for(int a=0; a<3;a++)
173         {
174             rel_pos[a]=este.pos[a]+k3_x[j][a];
175         }
176         Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
177     }
178     if (type>0)
179     {
180         for(int l=0; l<tot_p; l++)
181         {
182             if(j == l)
183             {
184                 Fx+=0.0;
185                 Fy+=0.0;
186                 Fz+=0.0;
187             }
188             else
189             {
190                 planet &otro=all_planets[l];
191                 for(int a=0; a<3;a++)
192                 {
193                     rel_pos[a]=-((otro.pos[a]+k3_x[l][a])-(este.pos[a]+
194 k3_x[j][a]));
195                 }
196                 Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.

```

```

194     mass);
195     }
196 }
197
198     k4_v[j][0] = h*Fx;
199     k4_v[j][1] = h*Fy;
200 k4_v[j][2] = h*Fz;
201     for(int l= 0; l<3;l++)
202     {
203         k4_x[j][l] = h*(este.vel[l]+ k3_v[j][l]);
204     }
205     }//End of loop
206
207     //This updates the functions
208     for(int j=0;j<tot_p;j++)
209     {
210         planet &este =all_planets[j];
211
212     for(int l=0; l<3; l++)
213     {
214         este.pos[l] += (k1_x[j][l] +2*(k2_x[j][l]+k3_x[j][l]) +k4_x[j][l])/6.0;
215         este.vel[l] += (k1_v[j][l] +2*(k2_v[j][l]+k3_v[j][l]) +k4_v[j][l])/6.0;
216     }
217     }
218
219     //Write updated values to the file
220     Write_Pos(ofile ,time);
221     Write_Energy(E_output,time,type);
222 }

```

Listing 2: This shows how velocity RK4 method is implemented.

4 Results

Here we begin to discuss the results of the program. We start by discussing the stability of both the RK4 and the Velocity Verlet method. To see whether it's stable we looked at the total energy and the angular momentum which should be conserved. Additionally we looked at their orbits and how much they varied. In the Discussion section I will only look closely at the Earth and Sun System which was enough to verify the program was working correctly. I next get the results for the Earth-Jupiter system and using the results from the Earth-Sun system I figure out the stability. In the Earth Jupiter system I look at how Jupiter effects Earth's orbit. First by looking at the original mass of Jupiter, then at 10 times of its original mass, and finally at 1000 times its original mass. Finally I look at the entire solar system, for this I use JPL's solar system data. To solve this problem I used two different classes one which was the plant and the solver. The planet class contained all the information for the planets and the solver contained the RK4 and the Velocity Verlet. This is given in the Code Attachment section.

5 Discussion

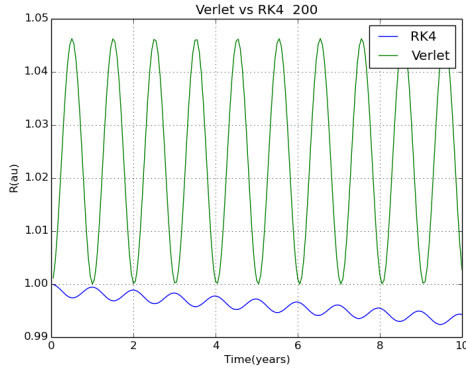
5.1 Earth Bound and Unbounded

Here I will look at the stability of the Earth and Sun system, computational speed, and also vary the initial speed of the Earth to see where the escape velocity occurs. From our results that the Velocity Verlet algorithm is the fastest on some occasions and also it has less error. This of course was expected since the relative error in the Velocity Verlet is h^4 and for the global error in the RK4 this is just h^3 .

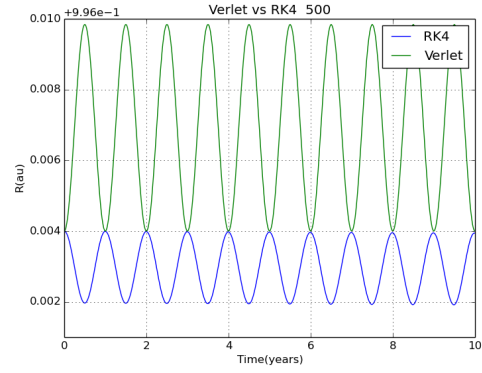
n	Verlet Time [s]	RK4 Time[s]
100	0.001393	0.001517
200	0.001385	0.003136
500	0.007372	0.06474
1000	0.014618	0.014609
2000	0.02704	0.027904
5000	0.068593	0.06544
10000	0.123608	0.128265

Table 1: This table gives the number of grid points used n , and the computational time. From here we see that the Velocity Verlet method is faster most of the time.

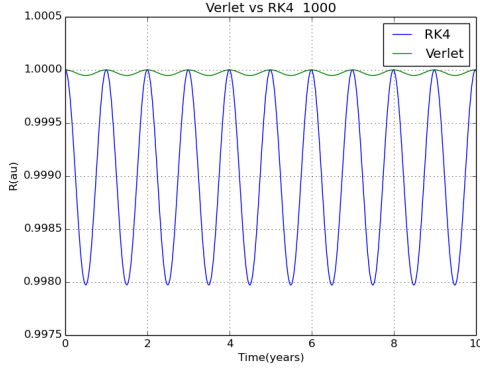
For the Earth to have a circular we found that the speed had to be $2\pi(AU/yr)$ at radius 1 AU. At this same distance we kept increasing the speed to find where the escape velocity. The escape velocity for this configuration is $2\sqrt{2}\pi(Au/yr)$. As we get close to this speed the orbit of the Earth reaches an ellipse and the finally it escapes. We can tell if the Earth is bounded by the potential if the total energy, which is the sum of the kinetic and potential, is positive.



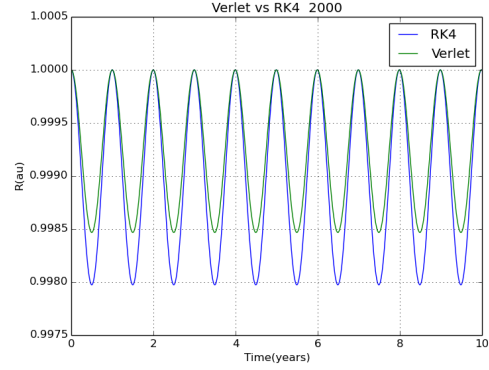
(a) Value of radius for $n=200$



(b) Value of radius for $n=500$

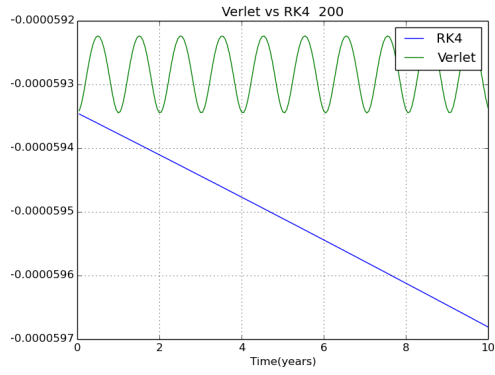


(c) Value of radius for $n=1000$

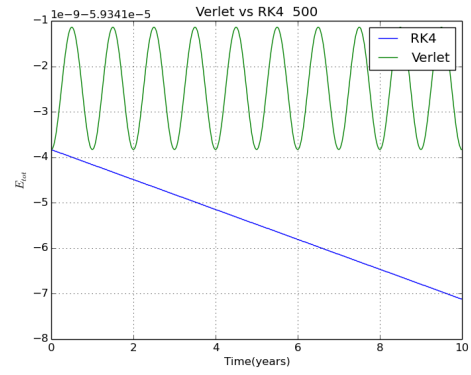


(d) Value of radius for $n=2000$

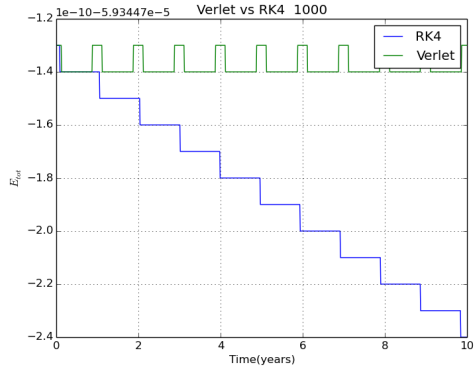
Figure 2: This shows the values of the radius for different values of n . From (a) and (b) we can see that the RK4 is a bit more stable than the Velocity Verlet. But as the value of n goes up the Velocity Verlet reaches stability at about $n=1000$. After this the Velocity Verlet is unstable and behaves similarly to RK4. From this we can gather that the Velocity Verlet is much more accurate.



(a) Total Energy $(Au/yr)^2$ for $n=200$

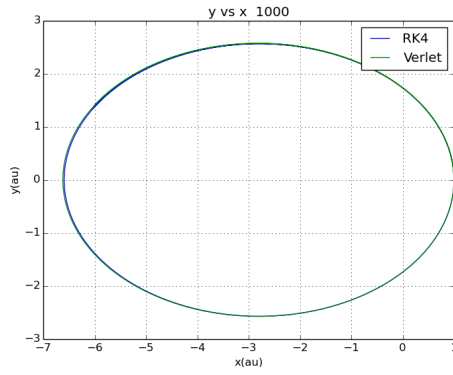


(b) Total Energy $(Au/yr)^2$ for $n=500$

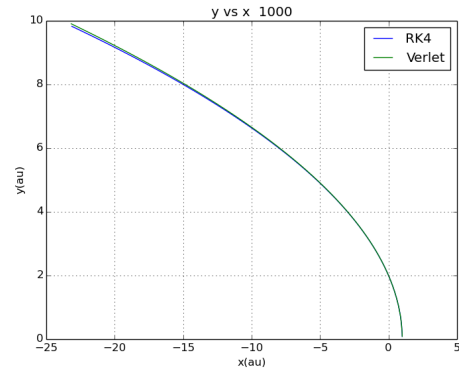


(c) Total Energy $(Au/yr)^2$ for $n=1000$

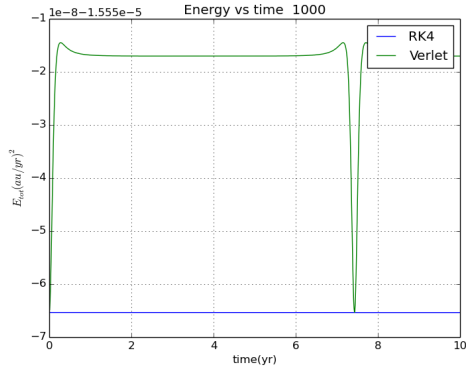
Figure 3: This shows the Values of the total energy for different values of n . Again we can clearly see that the Velocity Verlet method is more accurate.



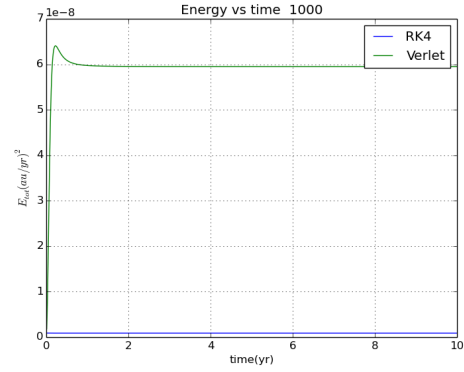
(a) Position of the Earth in x and y



(b) Posiion of the Earth in x and y



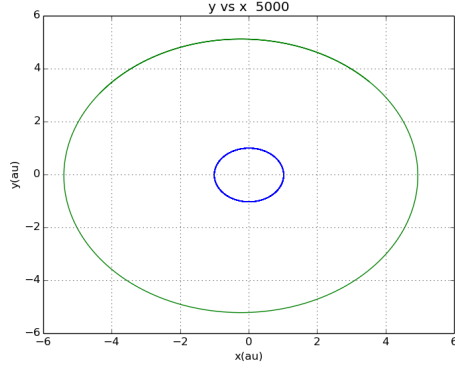
(c) Total Energy $(Au/yr)^2$ for $n=5000$



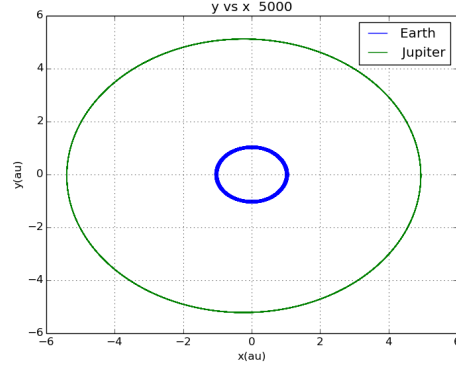
(d) Total Energy $(Au/yr)^2$ for $n=5000$

Figure 4: (a) and (b) are plot of the x and y coordinates of the Earth when the velocity will approach the escape velocity. (c) and (d) show their respective energies. This was done with $n=5000$.

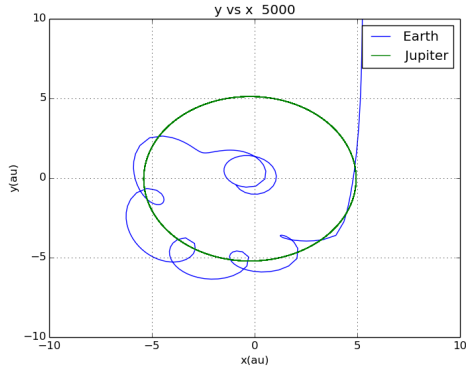
5.2 Earth Jupiter System



(a) Jupiter's original mass



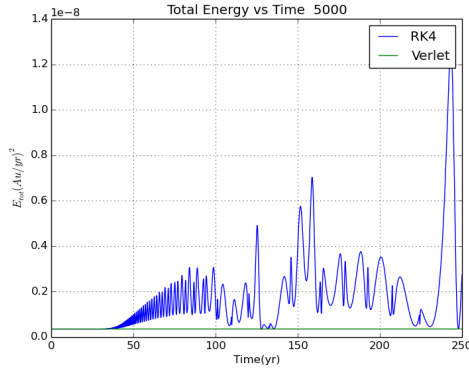
(b) Jupiter with 10 the original mass



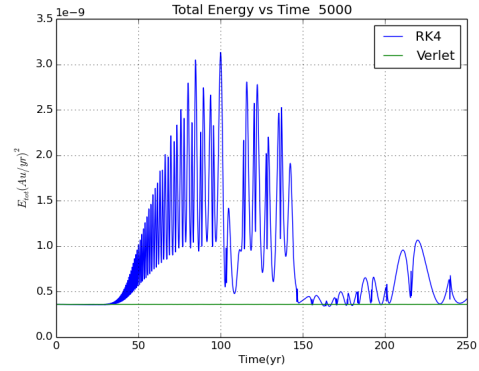
(c) Jupiter with 1000 the original mass

Figure 5: This shows the position of the Earth and Jupiter with different mass values for Jupiter. From this (a) and (b) we see that the mass of Jupiter alters Earth's orbit very little. This was done using the Verlet method at $n=5000$ since the RK4 was very unstable.

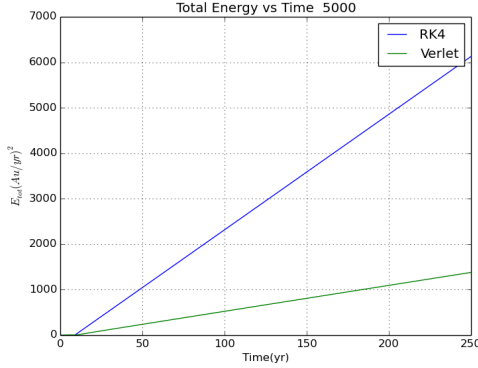
With the addition of Jupiter we see that Earth's orbit is only slightly perturbed. However as we increase the mass of Jupiter to ten times the original we start seeing a more noticeable deviation. It is until Jupiter's mass is increased by one thousand where we see the Earth's orbit is not stable anymore. This causes the Earth to get thrown out of the system. One additional thing to note is that the RK4 for this was very unstable and this can be seen in the values of the total energies.



(a) original mass



(b) 10 times the original mass

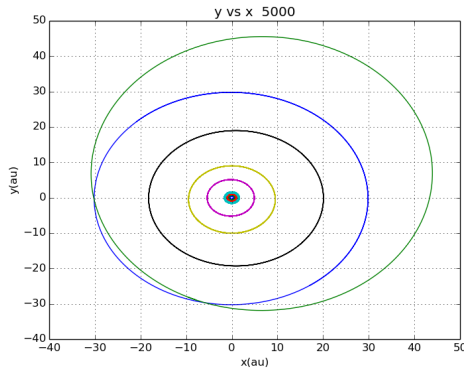


(c) 1000 times the original mass

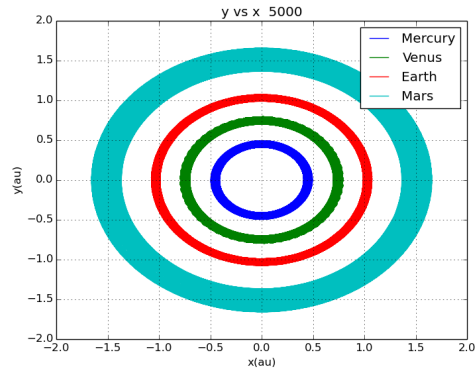
Figure 6: This shows the total energy for the Earth with different values of the mass of Jupiter. From this we can see that the RK4 method is very unstable and was not used to make plots of the positions. 5000 grid point were used to get the energies.

5.3 Solar System

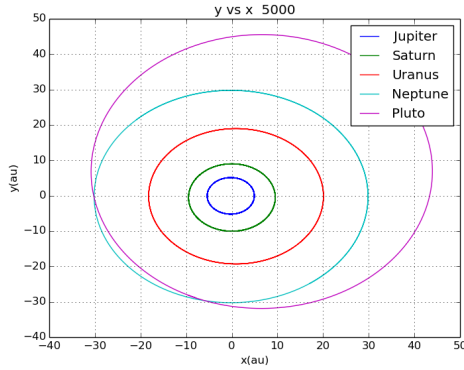
Here I include all of the planets and Pluto. From Figure 7 we see that the inner planets are perturbed more than the outer ones.



(a) Full Solar System



(b) Inner Solar System



(c) Outer Solar System

Figure 7: This shows the plane view of the orbits of all the planets and Pluto. To see what's really going on I included a view of the inner and outer solar system.

6 Conclusion

In setting up the equations of motion of the planets with the RK4 and Velocity Verlet(VV) method I conclude that the VV method is a better choice to solve this problem. The results show that the RK4 method is varying more compared to VV. This was expected though since the propagating error in VV is h^4 and for the RK4 is h^3 . We also found that the velocity needed for circular orbit and also escape the solar system. This are given by $2*\pi$ and $2*\sqrt{2}\pi$ respectively. When adding Jupiter we found that it has only small perturbation on Earth's orbit and this prompted us to hold the Sun in place when the we model all the planets.

7 References

- [1] M. Hjorth-Jensen. Computational Physics, Lecture Notes Spring 2016.
- [2] Solar System Dynamics. Jet Propulsion Laboratory: California Institute of Technology[accessed May 8 2016; ssd.jpl.nasa.gov/horizons.cgi]

8 Code Attachment

```
1 #ifndef PLANET_H
2 #define PLANET_H
3 #include <cmath>
4 #define _USE_MATH_DEFINES
5
6 class planet
7 {
8
9
10 public:
11     //Properties
12     int n;
13     double mass, t_i, t_f;
14     double pos[3];
15     double vel[3];
16     double L[3];
17
18     //Initializers
19     planet();
20     planet(double M, int step);
21
22     //Functions
23     double Kin_E();
24     double Ang_M();
25 };
26 #endif
```

Listing 3: This sets up the class for planets.

```
1 #include "/home/quetzalcoat1/Computational_Physics_work/Project3/Code/planet.h"
2 #include <iostream>
3
4 using namespace std;
5
6 planet::planet() {}
7
8 planet::planet(double M, int step)
9 {
10     n=step;
11     mass=M;
12
13     if(M == 1.0)
14     {
15         cout<<"Enter information for the SUN"<<endl;
16     }
17     else if(M ==1.6605e-7)
18     {
19         cout<<"Enter infomation for MERCURY"<<endl;
20     }
21     else if(M == 2.4483e-6)
22     {
23         cout<<"Enter information for the VENUS"<<endl;
24     }
25     else if(M ==3.0034e-6)
26     {
```

```

    cout<<"Enter infomation for EARTH"<<endl;
28     }
    else if(M ==3.2278e-7)
30     {
        cout<<"Enter infomation for MARS"<<endl;
32     }
    else if(M == 9.5449e-4)
34     {
        cout<<"Enter information for the JUPITER"<<endl;
36     }
    else if(M == 9.5449e-1)
38     {
        cout<<"Enter information for the JUPITER"<<endl;
40     }
    else if(M == 9.5449e-3)
42     {
        cout<<"Enter information for the JUPITER"<<endl;
44     }
        else if(M == 2.8580e-4)
46     {
            cout<<"Enter information for the SATURN"<<endl;
48     }
            else if(M == 4.3656e-5)
50     {
                cout<<"Enter infomation for URANUS"<<endl;
52     }
    else if(M ==5.1506e-5 )
54     {
        cout<<"Enter infomation for NEPTUNE"<<endl;
56     }
        else if(M == 6.5728e-8)
58     {
            cout<<"Enter information for the PLUTO"<<endl;
60     }
    }

62     int sepa;
    cout<<"Do you want to input intial conditions ,no(0) or yes(1): ";
64     cin>>sepa;
    if(sepa == 1)
66     {
        cout<<"Initial position x_i: ";
68         cin>>pos[0];
        cout<<"Initial position y_i: ";
70         cin>>pos[1];
        cout<<"Initial position z_i: ";
72         cin>>pos[2];
        cout<<"Initial velocity Vx_i: ";
74         cin>>vel[0];
        cout<<"Initial velocity Vy_i: ";
76         cin>>vel[1];
        cout<<"Initial velocity Vz_i: ";
78         cin>>vel[2];
    }
80     else
    {
82         if(M == 1.0)
            {

```

```

84 //Sun
85     pos[0]=-6.6275e-3;
86     pos[1]=-3.42121e-3;
87     pos[2]=1.96822e-4;
88     vel[0]=(6.12836e-6)*365;
89     vel[1]=(-6.7365e-6)*365;
90     vel[2]=(-1.17571e-7)*365;
91
92 }
93
94 if(M ==1.6605e-7)
95 {
96 //Mercury
97 pos[0]=-1.25329e-1;
98     pos[1]=-4.5394e-1;
99     pos[2]=-2.571179e-2;
100     vel[0]=(2.15648e-2)*365;
101     vel[1]=(-5.76037e-3)*365;
102 vel[2]=(-2.44891e-3)*365;
103 }
104 if(M == 2.4483e-6)
105 {
106 //Venus
107     pos[0]=5.7835e-1;
108     pos[1]=-4.3512e-1;
109     pos[2]=-3.94689e-2;
110     vel[0]=(1.18909e-2)*365;
111     vel[1]=(1.61875e-2)*365;
112     vel[2]=(-4.64774e-4)*365;
113 }
114
115 if(M ==3.0034e-6)
116 {
117 //Earth
118 pos[0]=-9.9140e-1;
119     pos[1]=-1.6994e-1;
120     pos[2]=1.98187e-4;
121     vel[0]=(2.5895e-3)*365;
122     vel[1]=(-1.70385e-2)*365;
123     vel[2]=(5.1364e-7)*365;
124 }
125 else if(M ==3.2278e-7)
126 {
127 //Mars
128     pos[0]=9.0233e-1;
129     pos[1]=1.1606;
130     pos[2]=2.2383e-3;
131     vel[0]=(-1.0490e-2)*365;
132     vel[1]=(9.797466e-3)*365;
133     vel[2]=(4.6327e-4)*365;
134 }
135 if(M == 9.5449e-4)
136 {
137 //Jupiter
138 pos[0]=3.55378;
139     pos[1]=3.47587;
140     pos[2]=-9.39615e-2;

```

```

142         vel[0]=(-5.3695e-3)*365;
143         vel[1]=(5.7509e-3)*365;
144         vel[2]=(9.6353e-5)*365;
145     }
146     if(M == 9.5449e-1)
147     {
148         //Jupiter
149         pos[0]=3.55378;
150         pos[1]=3.47587;
151         pos[2]=-9.39615e-2;
152         vel[0]=(-5.3695e-3)*365;
153         vel[1]=(5.7509e-3)*365;
154         vel[2]=(9.6353e-5)*365;
155     }
156     if(M == 9.5449e-3)
157     {
158         //Jupiter
159         pos[0]=3.55378;
160         pos[1]=3.47587;
161         pos[2]=-9.39615e-2;
162         vel[0]=(-5.3695e-3)*365;
163         vel[1]=(5.7509e-3)*365;
164         vel[2]=(9.6353e-5)*365;
165     }
166     if(M == 2.8580e-4)
167     {
168         //Saturn
169         pos[0]=6.01042;
170         pos[1]=6.9007;
171         pos[2]=-3.59215e-1;
172         vel[0]=(-4.49875e-3)*365;
173         vel[1]=(3.65260e-3)*365;
174         vel[2]=(1.15655e-4)*365;
175     }
176     if(M == 4.3656e-5)
177     {
178         //Uranus
179         pos[0]=1.4660e1;
180         pos[1]=-1.3499e1;
181         pos[2]=-2.40100e-1;
182         vel[0]=(2.635377e-3)*365;
183         vel[1]=(2.71037e-3)*365;
184         vel[2]=(-2.40986e-5)*365;
185     }
186     if(M == 5.1506e-5 )
187     {
188         //Neptune
189         pos[0]=1.70329e1;
190         pos[1]=-2.4836e1;
191         pos[2]=1.18927e-1;
192         vel[0]=(2.567966e-3)*365;
193         vel[1]=(1.79313e-3)*365;
194         vel[2]=(-9.6080e-5)*365;
195     }
196     if(M == 6.5728e-8)
197     {

```

```

198         //Pluto
200         pos[0]=-9.6135;
202         pos[1]=-2.80971e1;
204         pos[2]=5.78738;
206         vel[0]=(3.0528e-3)*365;
208         vel[1]=(-1.50824e-3)*365;
210         vel[2]=(-7.17669e-4)*365;
212     }
214 }
216
218 cout<<"Initial time t_i: ";
220 cin>>t_i;
222 cout<<"Final time t_f: ";
224 cin>>t_f;
226 }
228
230 double planet::Kin_E()
231 {
232     double Velocity=this->vel[0]*this->vel[0]+this->vel[1]*this->vel[1]+this->vel[2]*this->vel[2];
234     return 0.5*this->mass*Velocity;
236 }
238
240 double planet::Ang_M()
241 {
242     L[0]=this->pos[1]*this->vel[2]-this->pos[2]*this->vel[1];
244     L[1]=this->pos[2]*this->vel[0]-this->pos[0]*this->vel[2];
246     L[2]=this->pos[0]*this->vel[1]-this->pos[1]*this->vel[0];
248     return this->mass*this->mass*(L[0]*L[0]+L[1]*L[1]+L[2]*L[2]);
250 }

```

Listing 4: This gives commands for the class planets.

```

1  #ifndef SOLVER_H
2  #define SOLVER_H
3  #include <cmath>
4  #include "/home/quetzalcoat1/Computational_Physics_work/Project3/Code/planet.h"
5  #include <vector>
6  #include <iostream>
7  #include <fstream>
8  #include <iomanip>
9
10 using std::vector;
11
12 class solver
13 {
14
15 public:
16     friend class planet;
17
18     //Properties
19     int tot_p; //Total planets
20     vector<planet> all_planets;
21     double G;

```



```

23 //Initializers
    solver();
25
26 //Functions
27 void add(planet newplanet); //adds new planet
28 void verlet(planet &N,int type);
29 void RK4(planet &N,int type);
30 void Force(double &Fx, double &Fy, double &Fz, double x, double y, double z,
    double m);
31 void Header_Pos(int type, std::ofstream& ofile);
32 void Header_Energy(int type, std::ofstream& ofile);
33 void Write_Pos(std::ofstream& ofile, double time);
34 void Write_Energy(std::ofstream& ofile, double time, int type);
35 void Potential(double &pot, double x, double y, double z, double m1, double m2
    );
37 };
#endif

```

Listing 5: This sets up the class for the solver system which include the Velocity Verlet and RK4.

```

#include "/home/quetzalcoat1/Computational_Physics_work/Project3/Code/planet.h"
2 #include "/home/quetzalcoat1/Computational_Physics_work/Project3/Code/solver.h"
#include "time.h"
4 #define _USE_MATH_DEFINES
#include <cmath>
6 #include <iostream>
#include <fstream>
8 #include <iomanip>

10
using namespace std;

12
    solver::solver()
14 {
    tot_p = 0; //Total planets
16    G = 4*M_PI*M_PI;
    }

18
    void solver::add(planet newplanet)
20 {
    tot_p += 1;
22    all_planets.push_back(newplanet);
    }

24
//This calculates the position of the planets using the Verlet Method
26 void solver::verlet(planet &N,int type)
{
28    double h = (N.t_f - N.t_i)/N.n;
    double Fx, Fy, Fz, Fx1, Fy1, Fz1;
30    double acc[tot_p][3];
    double Nacc[tot_p][3];
32    double rel_pos[3];
    double time;
34

```

```

36 char Ve[30];
37 char Energy2[30]; //contains the Kinetic energy, potential energy, and
   angular mom.
38 cout<<"Enter the name of the Verlet file: ";
   cin>>Ve;
40 cout <<"Enter the name of the Energy file: ";
   cin>>Energy2;
42
43 std::ofstream ofile(Ve);
44 std::ofstream E_output(Energy2);
   Header_Pos(type, ofile);
46 Header_Energy(type, E_output);
48
49 //Write initial values to the file
   time= 0.0;
50 Write_Pos(ofile, time);
   Write_Energy(E_output, time, type);
52
53 //Start the clock
   clock_t start_VV, finish_VV;
   start_VV = clock();
56
57     for(int i=0; i< N.n; i++)
58     {
59         time=(i+1)*h;
60         for(int j=0; j<tot_p; j++)
61         {
62             planet &este = all_planets[j];
63             Fx=Fy=Fz=Fx1=Fy1=Fz1=0;
64             Force(Fx,Fy,Fz, este.pos[0], este.pos[1], este.pos[2], 1);
65             if(type>0)
66             {
67                 for(int l=0; l<tot_p; l++)
68                 {
69                     if(l == j)
70                     {
71                         Fx+=0;
72                         Fy+=0;
73                         Fz+=0;
74                     }
75                     else
76                     {
77                         planet &otro = all_planets[l];
78                         for(int d =0; d<3; d++)
79                         {
80                             rel_pos[d] = este.pos[d] - otro.pos[d];
81                         }
82                         Force(Fx,Fy,Fz, rel_pos[0], rel_pos[1], rel_pos[2], otro.mass);
83                     }
84                 }
85             }
86             acc[j][0] = Fx;
87             acc[j][1] = Fy;
88             acc[j][2] = Fz;
89
90             //Update the position .

```

```

92     for(int l=0; l<3; l++)
93     {
94         este.pos[l] += h*este.vel[l] +0.5*h*h*acc[j][l];
95     }
96
97     //Values for the velocity
98     Force(Fx1,Fy1,Fz1,este.pos[0],este.pos[1],este.pos[2],1);
99     if(type>0)
100    {
101        for(int l=0; l<tot_p;l++)
102        {
103            if(l == j)
104            {
105                Fx1+=0;
106                Fy1+=0;
107                Fz1+=0;
108            }
109            else
110            {
111                planet &otro = all_planets[l];
112                for(int d =0; d<3; d++)
113                {
114                    rel_pos[d] = este.pos[d] - otro.pos[d];
115                }
116                Force(Fx1,Fy1,Fz1,rel_pos[0],rel_pos[1],rel_pos[2],otro
117                .mass);
118            }
119        }
120
121        Nacc[j][0] = Fx1;
122        Nacc[j][1] = Fy1;
123        Nacc[j][2] = Fz1;
124
125        //Calculate new velocity
126        for(int l=0; l<3; l++)
127        {
128            este.vel[l] += 0.5*h*(acc[j][l] + Nacc[j][l]);
129        }
130    }
131
132    //Write the Updated the values
133    Write_Pos(ofile,time);
134    Write_Energy(E_output,time,type);
135
136    }
137
138    //stop clock and display time
139    finish_VV=clock();
140    cout<<"Total Time VV: "<<((double)(finish_VV - start_VV)/CLOCKS_PER_SEC)<<
141    endl;
142
143    ofile.close();
144    E_output.close();
145
146 }

```

```

148 //Calculates the position using the RK4 method.
void solver::RK4(planet &N,int type)
150 {
    double h = (N.t_f - N.t_i)/N.n;
152 double Fx,Fy,Fz;
    double rel_pos[3];
154 double time;

156 //Setting up ks, the first [] is the total planets to be solved for and
    //the second is the number of dimenstions
158 double k1_v[tot_p][3], k2_v[tot_p][3],k3_v[tot_p][3],k4_v[tot_p][3];
    double k1_x[tot_p][3], k2_x[tot_p][3],k3_x[tot_p][3],k4_x[tot_p][3];
160

162 //Initializes Writting
char RK4[30];
164 char Energy[30]; //contains the Kinetic energy, potential energy, and
    angular mom.
    cout<<"Enter the name of the RK4 file: ";
166 cin>>RK4;
    cout <<"Enter the name of the Energy file: ";
168 cin>>Energy;

170 std::ofstream ofile(RK4);
    std::ofstream E_output(Energy);
172 Header_Pos(type,ofile);
    Header_Energy(type,E_output);
174

176 //Write initial values to the file
    time= 0.0;
    Write_Pos(ofile,time);
178 Write_Energy(E_output,time,type);

180 //Start Clock
clock_t start_RK, finish_RK;
182 start_RK = clock();

184 //Setting up the k values
for(int i=0; i<N.n; i++)
186 {
    time=(i+1)*h;
188

    //Seting up K1
190 for(int j=0; j<tot_p; j++)
    {
192 planet &este=all_planets[j];
        Fx=Fy=0.0;
194

        Force(Fx,Fy,Fz,este.pos[0],este.pos[1],este.pos[2],1);
196

198 if(type >0)
    {
        for(int l=0; l<tot_p; l++)
200     {
            if( j == l)
202     {

```

```

204     Fx+=0.0;
206     Fy+=0.0;
208     Fz+=0.0;
210     }
212     else
214     {
216         planet &otro=all_planets[1];
218         for(int a=0; a<3;a++){rel_pos[a]=-(otro.pos[a]-este.pos[a]);}
220         Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
222     }
224 }
226 }
228
230     k1_v[j][0] = h*Fx;
232     k1_v[j][1] = h*Fy;
234     k1_v[j][2] = h*Fz;
236     for(int l= 0; l<3;l++)
238     {
240         k1_x[j][l] = h*este.vel[l];
242     }
244 }//End of loop
246
248 //Setting up K2
250 for(int j=0; j<tot_p; j++)
252 {
254     planet &este=all_planets[j];
256     Fx=Fy=0.0;
258
260     for(int a=0; a<3;a++)
262     {
264         rel_pos[a]=este.pos[a]+k1_x[j][a]/2.0;
266     }
268     Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
270
272     if(type >0)
274     {
276         for(int l=0; l<tot_p; l++)
278         {
280             if(j == l)
282             {
284                 Fx+=0.0;
286                 Fy+=0.0;
288                 Fz+=0.0;
290             }
292             else
294             {
296                 planet &otro=all_planets[l];
298                 for(int a=0; a<3;a++)
299                 {
300                     rel_pos[a]=-(otro.pos[a]+k1_x[l][a]/2.0)-(este.pos[a]
301 ]+k1_x[j][a]/2.0));
302                 }
303                 Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.
304 mass);
305             }
306         }
307     }
308 }

```

```

258     }
260         k2_v[j][0] = h*Fx;
261         k2_v[j][1] = h*Fy;
262     k2_v[j][2] = h*Fz;
263         for(int l= 0; l<3;l++)
264         {
265             k2_x[j][l] = h*(este.vel[l]+ k1_v[j][l]/2.0);
266         }
267     }//End of loop
268
269     //Setting up K3
270     for(int j=0; j<tot_p; j++)
271     {
272         planet &este=all_planets[j];
273         Fx=Fy=0.0;
274
275         for(int a=0; a<3;a++)
276         {
277             rel_pos[a]=este.pos[a]+k2_x[j][a]/2.0;
278         }
279         Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
280
281     if(type>0)
282     {
283         for(int l=0; l<tot_p; l++)
284         {
285             if(j == l)
286             {
287                 Fx+=0.0;
288                 Fy+=0.0;
289                 Fz+=0.0;
290             }
291             else
292             {
293                 planet &otro=all_planets[l];
294                 for(int a=0; a<3;a++)
295                 {
296                     rel_pos[a]=-((otro.pos[a]+k2_x[l][a]/2.0)-(este.pos[
297 a]+k2_x[j][a]/2.0));
298                 }
299                 Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.
300 mass);
301             }
302         }
303     }
304
305     k3_v[j][0] = h*Fx;
306     k3_v[j][1] = h*Fy;
307     k3_v[j][2] = h*Fz;
308         for(int l= 0; l<2;l++)
309         {
310             k3_x[j][l] = h*(este.vel[l]+ k2_v[j][l]/2.0);
311         }
312     }//End of loop
313
314     //Setting up K4

```

```

314     for(int j=0; j<tot_p; j++)
316     {
318         planet &este=all_planets[j];
320         Fx=Fy=Fz=0.0;
322         for(int a=0; a<3;a++)
324         {
326             rel_pos[a]=este.pos[a]+k3_x[j][a];
328             Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
330         }
332         if(type>0)
334         {
336             for(int l=0; l<tot_p; l++)
338             {
340                 if(j == l)
342                 {
344                     Fx+=0.0;
346                     Fy+=0.0;
348                     Fz+=0.0;
350                 }
352                 else
354                 {
356                     planet &otro=all_planets[l];
358                     for(int a=0; a<3;a++)
360                     {
362                         rel_pos[a]=-((otro.pos[a]+k3_x[l][a])-(este.pos[a]+
364                         k3_x[j][a]));
366                     }
368                     Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.
370                     mass);
372                 }
374             }
376         }
378         k4_v[j][0] = h*Fx;
380         k4_v[j][1] = h*Fy;
382         k4_v[j][2] = h*Fz;
384         for(int l= 0; l<3;l++)
386         {
388             k4_x[j][l] = h*(este.vel[l]+ k3_v[j][l]);
390         }
392     } //End of loop
394
396     //This updates the functions
398     for(int j=0;j<tot_p;j++)
400     {
402         planet &este =all_planets[j];
404
406         for(int l=0; l<3; l++)
408         {
410             este.pos[l] += (k1_x[j][l] +2*(k2_x[j][l]+k3_x[j][l]) +k4_x[j][l])/6.0;
412             este.vel[l] += (k1_v[j][l] +2*(k2_v[j][l]+k3_v[j][l]) +k4_v[j][l])/6.0;
414         }
416     }
418
420     //Write updated values to the file
422     Write_Pos(ofile ,time);

```

```

368         Write_Energy(E_output,time,type);
370     }
372     finish_RK = clock();
    cout<<"Total Time RK: "<<((double)(finish_RK -start_RK)/CLOCKS_PER_SEC)<<endl
    ;
374     ofile.close();
376     E_output.close();
378 }
380 void solver::Force(double &Fx, double &Fy,double &Fz, double x, double y,
    double z, double m)
    {
382     double r= sqrt(x*x+y*y+z*z);
384     Fx -= (G/(r*r*r))*x*m;
    Fy -= (G/(r*r*r))*y*m;
386     Fz -= (G/(r*r*r))*z*m;
    }
388 void solver::Header_Pos(int type,std::ofstream& ofile)
390 {
392     ofile<<setiosflags( ios::showpoint | ios::uppercase);
    ofile<<"#Time(Yr)";
394
    if(type == 0)
396     {
        ofile<<setw(20)<<"X_Ea"<<setw(20)<<"Y_Ea"<<setw(20)<<"Z_Ea";
398        ofile<<setw(20)<<"R_earth"<<endl;
    }
400    else if(type == 1)
    {
402        ofile<<setw(20)<<"X_Ea"<<setw(20)<<"Y_Ea"<<setw(20)<<"Z_Ea";
        ofile<<setw(20)<<"R_Earth";
404        ofile<<setw(20)<<"X_Ju"<<setw(20)<<"Y_Ju"<<setw(20)<<"Z_Ju";
        ofile<<setw(20)<<"R_Jupiter"<<endl;
406    }
    else if(type == 2)
408    {
        ofile<<setw(20)<<"X_Su"<<setw(20)<<"Y_Su"<<setw(20)<<"Z_Su";
410        ofile<<setw(20)<<"R_Sun";
        ofile<<setw(20)<<"X_Ea"<<setw(20)<<"Y_Ea"<<setw(20)<<"Z_Ea";
412        ofile<<setw(20)<<"R_Earth";
        ofile<<setw(20)<<"X_Ju"<<setw(20)<<"Y_Ju"<<setw(20)<<"Z_Ju";
414        ofile<<setw(20)<<"R_Jupiter"<<endl;
    }
416    else if(type == 3)
    {
418        ofile<<setw(20)<<"X_Me"<<setw(20)<<"Y_Me"<<setw(20)<<"Z_Me";
        ofile<<setw(20)<<"R_Mercury";
420        ofile<<setw(20)<<"X_Ve"<<setw(20)<<"Y_Ve"<<setw(20)<<"Z_Ve";
        ofile<<setw(20)<<"R_Venus";
422        ofile<<setw(20)<<"X_Ea"<<setw(20)<<"Y_Ea"<<setw(20)<<"Z_Ea";

```



```

424 ofile <<setw(20)<<"R_Earth";
425 ofile <<setw(20)<<"X_Ma"<<setw(20)<<"Y_Ma"<<setw(20)<<"Z_Ma";
426 ofile <<setw(20)<<"R_Mars";
427     ofile <<setw(20)<<"X_Ju"<<setw(20)<<"Y_Ju"<<setw(20)<<"Z_Ju";
428 ofile <<setw(20)<<"R_Jupiter";
429 ofile <<setw(20)<<"X_Sa"<<setw(20)<<"Y_Sa"<<setw(20)<<"Z_Sa";
430 ofile <<setw(20)<<"R_Saturn";
431     ofile <<setw(20)<<"X_Ur"<<setw(20)<<"Y_Ur"<<setw(20)<<"Z_Ur";
432 ofile <<setw(20)<<"R_Uranus";
433     ofile <<setw(20)<<"X_Ne"<<setw(20)<<"Y_Ne"<<setw(20)<<"Z_Ne";
434 ofile <<setw(20)<<"R_Neptune";
435     ofile <<setw(20)<<"X_Pu"<<setw(20)<<"Y_Pu"<<setw(20)<<"Z_Pu";
436 ofile <<setw(20)<<"R_Pluto"<<endl;
437 }
438 }
439
440 void solver::Header_Energy(int type, std::ofstream& ofile)
441 {
442     ofile <<setiosflags( ios::showpoint | ios::uppercase);
443     ofile <<"#Time(Yr)";
444
445     if(type == 0)
446     {
447         ofile <<setw(20)<<"K_Ea"<<setw(20)<<"U_Ea"<<setw(20)<<"Etot_Ea";
448         ofile <<setw(20)<<"L_Ea"<<endl;
449     }
450     else if(type == 1)
451     {
452         ofile <<setw(20)<<"K_Ea"<<setw(20)<<"U_Ea"<<setw(20)<<"Etot_Ea";
453         ofile <<setw(20)<<"L_Ea";
454         ofile <<setw(20)<<"K_Ju"<<setw(20)<<"U_Ju"<<setw(20)<<"Etot_Ju";
455         ofile <<setw(20)<<"L_Ju"<<endl;
456     }
457     else if(type == 2)
458     {
459         ofile <<setw(20)<<"K_Su"<<setw(20)<<"U_Su"<<setw(20)<<"Etot_Su";
460         ofile <<setw(20)<<"L_Su";
461         ofile <<setw(20)<<"K_Ea"<<setw(20)<<"U_Ea"<<setw(20)<<"Etot_Ea";
462         ofile <<setw(20)<<"L_Ea";
463         ofile <<setw(20)<<"K_Ju"<<setw(20)<<"U_Ju"<<setw(20)<<"Etot_Ju";
464         ofile <<setw(20)<<"L_Ju"<<endl;
465     }
466     else if(type == 3)
467     {
468         ofile <<setw(20)<<"K_Me"<<setw(20)<<"U_Me"<<setw(20)<<"Etot_Me";
469         ofile <<setw(20)<<"L_Me";
470         ofile <<setw(20)<<"K_Ve"<<setw(20)<<"U_Ve"<<setw(20)<<"Etot_Ve";
471         ofile <<setw(20)<<"L_Ve";
472         ofile <<setw(20)<<"K_Ea"<<setw(20)<<"U_Ea"<<setw(20)<<"Etot_Ea";
473         ofile <<setw(20)<<"L_Ea";
474         ofile <<setw(20)<<"K_Ma"<<setw(20)<<"U_Ma"<<setw(20)<<"Etot_Ma";
475         ofile <<setw(20)<<"L_Ma";
476         ofile <<setw(20)<<"K_Ju"<<setw(20)<<"U_Ju"<<setw(20)<<"Etot_Ju";
477         ofile <<setw(20)<<"L_Ju";
478         ofile <<setw(20)<<"K_Sa"<<setw(20)<<"U_Sa"<<setw(20)<<"Etot_Sa";
479         ofile <<setw(20)<<"L_Sa";
480         ofile <<setw(20)<<"K_Ur"<<setw(20)<<"U_Ur"<<setw(20)<<"Etot_Ur";

```

```

480     ofile <<setw(20)<<"L_Ur";
        ofile <<setw(20)<<"K_Ne"<<setw(20)<<"U_Ne"<<setw(20)<<"Etot_Ne";
482     ofile <<setw(20)<<"L_Ne";
        ofile <<setw(20)<<"K_Pl"<<setw(20)<<"U_Pl"<<setw(20)<<"Etot_pl";
484     ofile <<setw(20)<<"L_Pl"<<endl;
        }
486 }

488 void solver::Write_Pos(std::ofstream& ofile, double time)
{
490     double R[tot_p];

492     ofile <<time;
    for(int i=0; i<tot_p; i++)
494     {
        planet &este = all_planets[i];
496     for(int j=0; j<3; j++)
        {
498         ofile <<setw(20)<<este.pos[j];
        }

500     R[i] = sqrt(este.pos[0]*este.pos[0] + este.pos[1]*este.pos[1]+este.pos[2]*
        este.pos[2]);
502     ofile <<setw(20)<<R[i];
        }
504     ofile <<endl;
    }

506 void solver::Write_Energy(std::ofstream& ofile, double time, int type)
{
508     double pot;
    double Etot;
    double rel_p[3];
512     ofile <<time;

514     for(int i=0; i<tot_p; i++)
    {
516         pot= 0.0;
        Etot= 0.0;
518         planet &este = all_planets[i];

520         //Gets the Kinetic Energy
        ofile <<setw(20)<<este.Kin_E();

522         //Calculated the Potential energy
        Potential(pot, este.pos[0], este.pos[1], este.pos[2], este.mass, 1);

524         if(type == 0)
        {
526             ofile <<setw(20)<<pot;
        }

530         if(type > 0)
        {
532             for(int j=0; j<tot_p; j++)
            {
534                 planet &otro = all_planets[j];

```

```

536         if(i == j)
538         {
540             pot+=0.0;
542             }
544             else
546             {
548                 for(int l=0; l<3; l++)
550                 {
552                     rel_p[l] = este.pos[l] - otro.pos[l];
554                     Potential(pot, rel_p[0], rel_p[1], rel_p[2], este.mass, otro.mass);
556                     }
558                 ofile<<setw(20)<<pot;
560                 }
562                 //Writes total energy
564                 Etot = este.Kin_E() + pot;
566                 ofile<<setw(20)<<Etot;
568                 //Calculated the angular momentum
569                 ofile<<setw(20)<<setprecision(7)<<este.Ang_M();
570             }
571             ofile<<endl;
572         }
573
574         void solver::Potential(double &pot, double x, double y, double z, double m1,
575                                double m2)
576         {
577             double r = sqrt(x*x+y*y+z*z);
578             pot -= (G*m1*m2)/r;
579         }

```

Listing 6: This gives commands for the solver system.

```

1  #include <iostream>
2  #include "/home/quetzalcoat1/Computational_Physics_work/Project3/Code/planet.h"
3  #include "/home/quetzalcoat1/Computational_Physics_work/Project3/Code/solver.h"
4  #include <cmath>
5
6
7  using namespace std;
8
9  int main()
10 {
11     solver systemRK;
12     solver systemVV;
13     int type;
14     int n;
15
16     /*Chose what system will be solved.
17     type = 0 solves the Earth-Sun system with the sun stationary
18     type = 1 solves the Earth-Sun-Jupiter system with the sun stationary
19     type = 2 solves the Earth-Sun-Jupiter system with the sun, all with respect
20     to the center of mass
21     type = 3 solves the whole solar sytem with the sun stationary
22     */

```

```

21  cout<<"Which system will you like to be solve, type = ";
23  cin>>type;
25  cout<<"Enter the step size n: ";
27  cin>>n;

29  planet Earth(3.0034e-6, n);

31  if(type == 0)
33  {
35      systemRK.add(Earth);
37      systemVV.add(Earth);
39  }
41  else if(type == 1)
43  {
45      systemRK.add(Earth);
47      systemVV.add(Earth);
49  }
51  int typeJ;
53  cout<<"Mass of Jupiter; 0:normal, 1:10times, 2:E3times: ";
55  cin>>typeJ;
57  if(typeJ == 0)
59  {
61      planet Jupiter(9.5449e-4,100);
63      systemRK.add(Jupiter);
65      systemVV.add(Jupiter);
67  }
69  if(typeJ == 1)
71  {
73      planet Jupiter(9.5449e-3,100);
75      systemRK.add(Jupiter);
77      systemVV.add(Jupiter);
79  }
81  if(typeJ == 2)
83  {
85      planet Jupiter(9.5449e-1,100);
87      systemRK.add(Jupiter);
89      systemVV.add(Jupiter);
91  }
93  }
95  else if(type == 2)
97  {
99      planet Sun(1,100);
101      systemRK.add(Sun);
103      systemVV.add(Sun);
105      systemRK.add(Earth);
107      systemVV.add(Earth);
109      planet Jupiter(9.5449e-4,100);
111      systemRK.add(Jupiter);
113      systemVV.add(Jupiter);
115  }
117  else if(type == 3)
119  {
121      planet Mercury(1.6605e-7,100);
123      systemRK.add(Mercury);
125      systemVV.add(Mercury);

```

```

79  planet Venus(2.4483e-6,100);
    systemRK.add(Venus);
    systemVV.add(Venus);
81      systemRK.add(Earth);
    systemVV.add(Earth);
83  planet Mars(3.2278e-7,100);
    systemRK.add(Mars);
85  systemVV.add(Mars);
    planet Jupiter(9.5449e-4,100);
87      systemRK.add(Jupiter);
    systemVV.add(Jupiter);
89  planet Saturn(2.8580e-4,100);
    systemRK.add(Saturn);
91  systemVV.add(Saturn);
    planet Uranus(4.3656e-5,100);
93  systemRK.add(Uranus);
    systemVV.add(Uranus);
95  planet Neptune(5.1506e-5,100);
    systemRK.add(Neptune);
97  systemVV.add(Neptune);
    planet Pluto(6.5728e-8,100 );
99  systemRK.add(Pluto);
    systemVV.add(Pluto);
101 }

103 systemRK.RK4(Earth,type);
    systemVV.verlet(Earth,type);
105 }

```

Listing 7: Carries out the Calculation using all the classes set up before.