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 tha the Verlet was more accurate and it was slightly faster in computational time.

1 Introduction

I start by solving the 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 obit. 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

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

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.

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

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

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

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

$$\begin{aligned} F_x = \frac{d^2v_x}{dt^2} &= -\frac{4\pi^2}{r^3}x \\ \frac{dx}{dt} &= v_x \end{aligned} \tag{4}$$

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

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

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

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

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

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

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

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

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

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

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.

$$x(t_i \pm h) = x_{i\pm1} \quad x_i = x(t_i) \tag{9}$$

here i can range from 0, which is the initial time, to n. Now I will do the expansion in this discrtetized 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

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

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

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

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.

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


 char Ve[30];
 char Energy2[30]; //contains the Kinetic energy, potential energy, and
   angular mom.
       cout<<"Enter the name of the Verlet file: ";
       cin>>Ve;
 cout <<"Enter the name of the Energy file: ";
```

```
  cin>>Energy2;

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

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

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

        for(int i=0; i< N.n; i++)
        {
     time=(i+1)*h;
     for(int j=0; j<tot_p; j++)
     {
              planet &este = all_planets[j];
        Fx=Fy=Fz=Fx1=Fy1=Fz1=0;
        Force(Fx,Fy,Fz,este.pos[0],este.pos[1],este.pos[2],1);
        if(type>0)
        {
    for(int l=0; l<tot_p;l++)
    {
        if(l == j)
        {
      Fx+=0;
      Fy+=0;
      Fz+=0;
              }
        else
        {
          planet &otro = all_planets[l];
          for(int d =0; d<3; d++)
          {
        rel_pos[d] = este.pos[d] - otro.pos[d];
          }
          Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
        }
    }

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

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

        //Values for the velocity
```

```
        Force(Fx1,Fy1,Fz1,este.pos[0],este.pos[1],este.pos[2],1);
        if(type>0)
        {
          for(int l=0; l<tot_p;l++)
                {
                    if(l == j)
                    {
                        Fx1+=0;
                        Fy1+=0;
    Fz1+=0;
                    }
                    else
                    {
                        planet &otro = all_planets[l];
                        for(int d =0; d<3; d++)
                        {
                                rel_pos[d] = este.pos[d] - otro.pos[d];
                        }
                        Force(Fx1,Fy1,Fz1,rel_pos[0],rel_pos[1],rel_pos[2],otro
.mass);
                    }
                }
         }

        Nacc[j][0] = Fx1;
        Nacc[j][1] = Fy1;
        Nacc[j][2] = Fz1;

        //Calculate new velocity
        for(int l=0; l<3; l++)
        {
este.vel[l] += 0.5*h*(acc[j][l] + Nacc[j][l]);
        }

     }
    //Write the Updated the values
             Write_Pos(ofile,time);
             Write_Energy(E_output,time,type);

        }
```

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 is 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

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

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

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

To solve for the values of f at different at different times we us 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

$$y_{(i+1/2),Prediciton1} = y_i + \frac{h}{2}\frac{dy}{dt} = y_i + \frac{h}{2}f(t_i,y_i) = y_i + \frac{k_1}{2} \tag{13}$$

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

$$k_2 = f(t_{i+1/2}, y_{(i+1/2),Prediciton1}) = f(t_{i+1/2}, y_i + \frac{k_1}{2})$$

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

$$y_{(i+1/2),Prediciton2} = y_i + \frac{h}{2}\frac{dy}{dt} = y_i + \frac{h}{2}f(t_{i+1/2}, y_{(i+1/2),Prediciton1}) = y_i + \frac{k_2}{2} \tag{14}$$

and for the next slope we have

$$k_3 = hf(t_{i+1/2}, y_{(i+1/2),Prediciton2}) = f(t_{i+1/2}, y_i + \frac{k_2}{2})$$

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

$$y_{i+1} = y_i + \frac{h}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right] \tag{15}$$

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

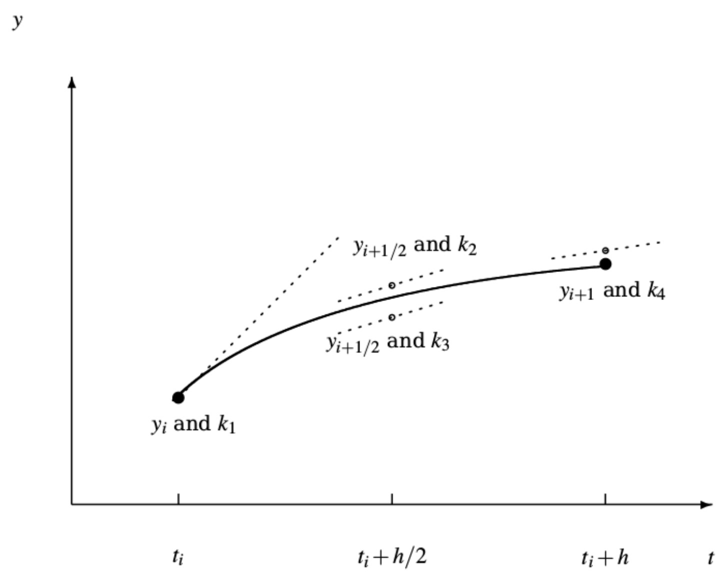


Figure 1: Plot of a general function and the differernt approximations taken in order to find the different slopes [1]

Equation (16) changes with the coordinate your 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 implimented is shown in the following listing.

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

 //Setting up ks, the first [] is the total planets to be solved for and
 //the second is the number of dimensions
 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];


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

        std::ofstream ofile(RK4);
        std::ofstream E_output(Energy);
        Header_Pos(type,ofile);
        Header_Energy(type,E_output);
```

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

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

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

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

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

  if(type >0)
  {
     for(int l=0; l<tot_p; l++)
     {
        if( j == l)
        {
    Fx+=0.0;
    Fy+=0.0;
    Fz+=0.0;
        }
        else
        {
          planet &otro=all_planets[l];
          for(int a=0; a<3;a++){rel_pos[a]=-(otro.pos[a]-este.pos[a]);}
                Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
        }
     }
  }


        k1_v[j][0] = h*Fx;
        k1_v[j][1] = h*Fy;
  k1_v[j][2] = h*Fz;
        for(int l= 0; l<3;l++)
  {
     k1_x[j][l] = h*este.vel[l];
  }
   }//End of loop

   //Setting up K2
   for(int j=0; j<tot_p; j++)
         {
              planet &este=all_planets[j];
              Fx=Fy=0.0;

```

```
      for(int a=0; a<3;a++)
                  {
                      rel_pos[a]=este.pos[a]+k1_x[j][a]/2.0;
                  }
                  Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);

      if(type >0)
      {
                      for(int l=0; l<tot_p; l++)
                      {
          if(j == l)
          {
        Fx+=0.0;
        Fy+=0.0;
        Fz+=0.0;
          }
          else
          {
                              planet &otro=all_planets[l];
                              for(int a=0; a<3;a++)
                              {
                                rel_pos[a]=-((otro.pos[a]+k1_x[l][a]/2.0)-(este.pos[a]+k1_x[j][a]/2.0));
                              }
                              Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
          }
                      }
      }

                  k2_v[j][0] = h*Fx;
                  k2_v[j][1] = h*Fy;
      k2_v[j][2] = h*Fz;
                  for(int l= 0; l<3;l++)
                  {
                      k2_x[j][l] = h*(este.vel[l]+ k1_v[j][l]/2.0);
                  }
            }//End of loop

        //Setting up K3
        for(int j=0; j<tot_p; j++)
            {
                  planet &este=all_planets[j];
                  Fx=Fy=0.0;


                  for(int a=0; a<3;a++)
                  {
                    rel_pos[a]=este.pos[a]+k2_x[j][a]/2.0;
                  }
                  Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
      if(type>0)
      {
                       for(int l=0; l<tot_p; l++)
                       {
           if(j == l)
           {
```

```
        Fx+=0.0;
        Fy+=0.0;
        Fz+=0.0;
           }
           else
           {
                          planet &otro=all_planets[l];
                          for(int a=0; a<3;a++)
                          {
                           rel_pos[a]=-((otro.pos[a]+k2_x[l][a]/2.0)-(este.pos[a]+k2_x[j][a]/2.0));
                          }
                          Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
           }
                    }
  }

            k3_v[j][0] = h*Fx;
            k3_v[j][1] = h*Fy;
  k3_v[j][2] = h*Fz;
            for(int l= 0; l<2;l++)
            {
               k3_x[j][l] = h*(este.vel[l]+ k2_v[j][l]/2.0);
            }
        }//End of loop

    //Setting up K4
    for(int j=0; j<tot_p; j++)
        {
            planet &este=all_planets[j];
            Fx=Fy=Fz=0.0;

            for(int a=0; a<3;a++)
            {
               rel_pos[a]=este.pos[a]+k3_x[j][a];
            }
            Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
  if(type>0)
  {
               for(int l=0; l<tot_p; l++)
               {
        if(j == l)
        {
    Fx+=0.0;
    Fy+=0.0;
    Fz+=0.0;
            }
        else
        {
                        planet &otro=all_planets[l];
                        for(int a=0; a<3;a++)
                        {
                           rel_pos[a]=-((otro.pos[a]+k3_x[l][a])-(este.pos[a]+k3_x[j][a]));
                        }
                        Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.
```

```
mass);
        }
                }
}

            k4_v[j][0] = h*Fx;
            k4_v[j][1] = h*Fy;
k4_v[j][2] = h*Fz;
            for(int l= 0; l<3;l++)
            {
                k4_x[j][l] = h*(este.vel[l]+ k3_v[j][l]);
            }
        }//End of loop

    //This updates the functions
        for(int j=0;j<tot_p;j++)
        {
            planet &este =all_planets[j];

for(int l=0; l<3; l++)
{
    este.pos[l] += (k1_x[j][l] +2*(k2_x[j][l]+k3_x[j][l]) +k4_x[j][l])/6.0;
    este.vel[l] += (k1_v[j][l] +2*(k2_v[j][l]+k3_v[j][l]) +k4_v[j][l])/6.0;
}
        }

      //Write updated values to the file
      Write_Pos(ofile,time);
      Write_Energy(E_output,time,type);

}
```

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

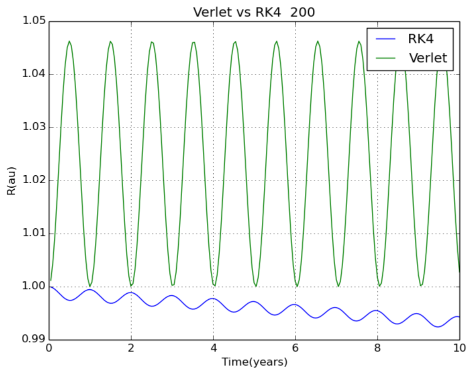
5.1 Earth Bound and Unbounded

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 ocassions 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.

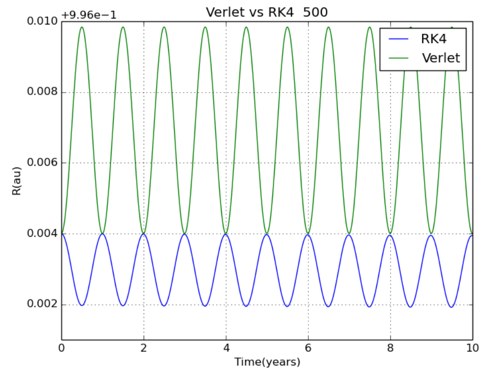
n	Verlet Time [s]	RK4 Time[s]
100	0.001393	0.001517
200	0.001385	0.0.003136
500	0.007372	0.06474
1000	0.014618	0.014609
2000	0.02704	0.027904
5000	0.068593	0.06544
10000	0.123608	0.128265

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.

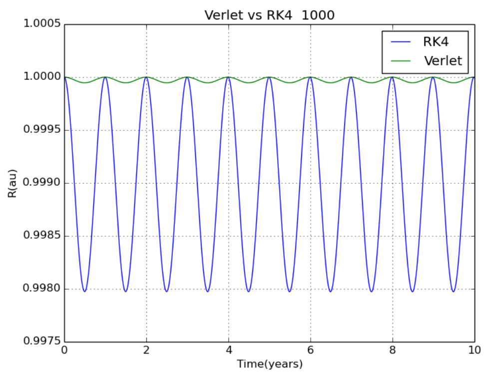
For the Earth to have a circular we found that the speed had to be $2\pi(AU/yr)$ at radius 1 AU. At this same distance we kept increasing the speed to find were the escape velocity. The escape velocity for this configuration is $2\sqrt{2}\pi(Au/yr)$. As we get close to this speed the orbit of the Earth 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.



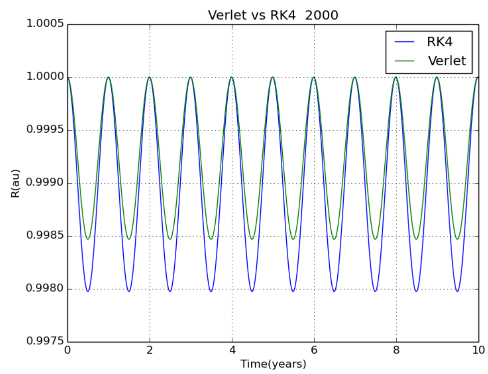
(a) Value of radius for n=200



(b) Value of radius for n=500

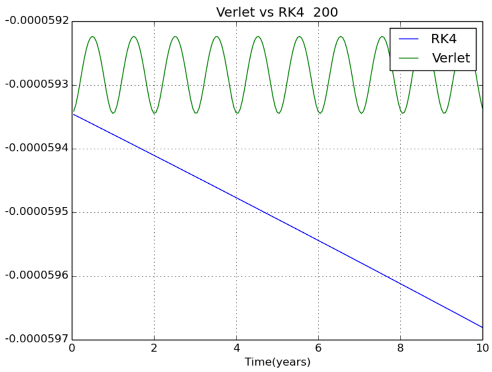


(c) Value of radius for n=1000

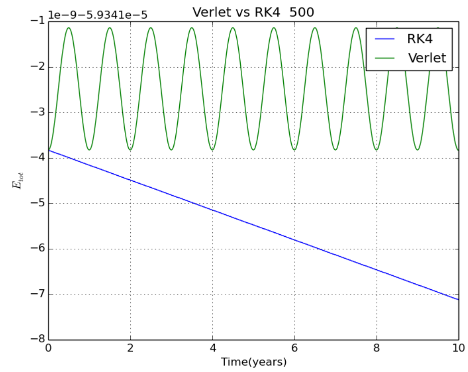


(d) Value of radius for n=2000

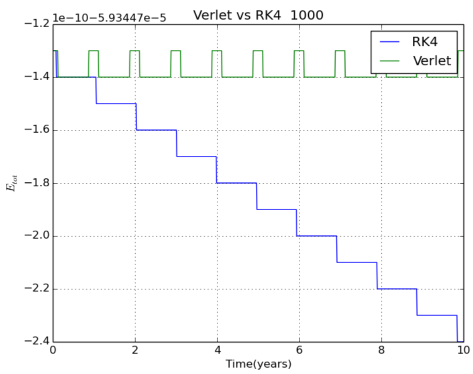
Figure 2: This shows the values of the radius for different values of n. From (a) and (b) we can see that the RK4 is a bit more stable than the Velocity Verlet. But as the value of n goes up the Velocity Verlet reaches stability at about n=1000. After this the Velocity Verlet is unstable and behaves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

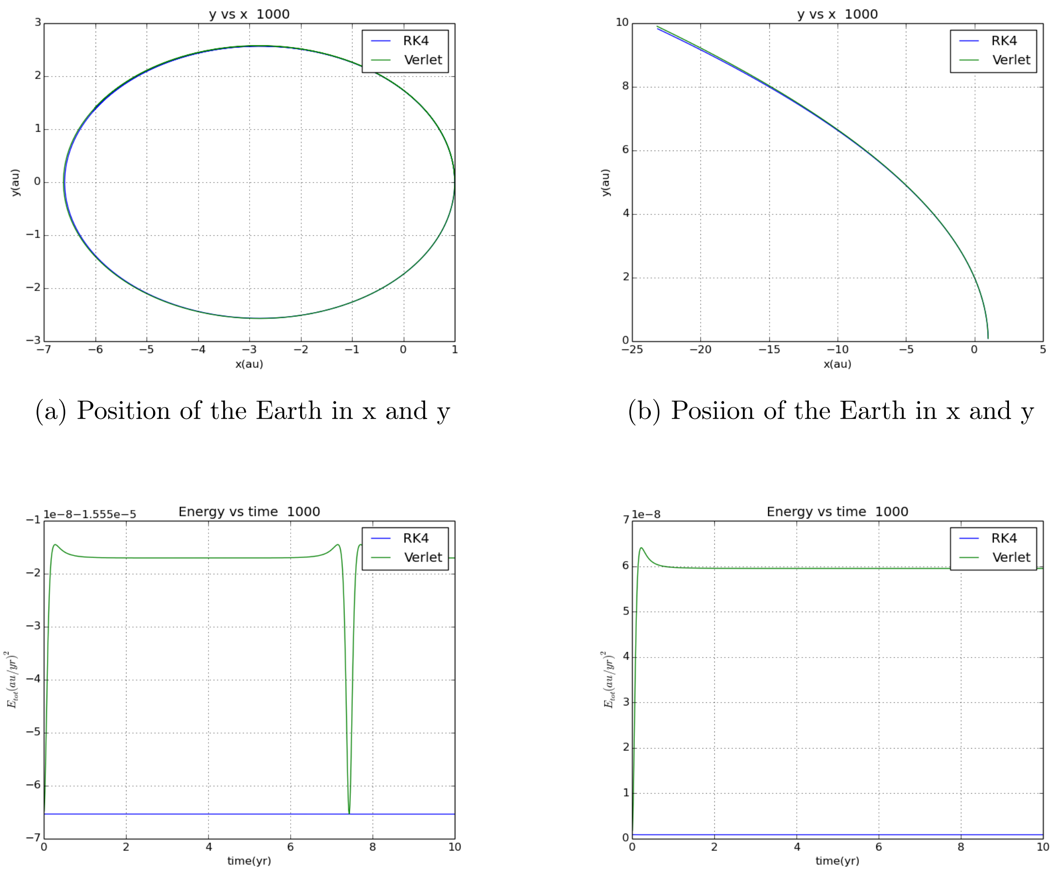


(b) Total Energy $(Au/yr)^2$ for n=500



(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

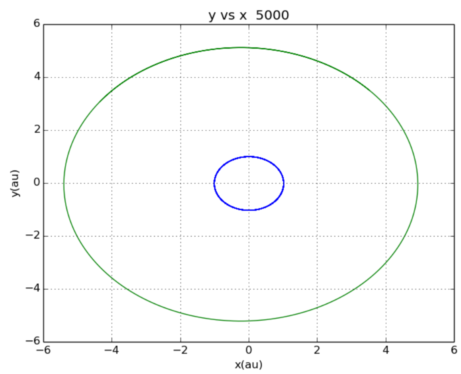
(b) Posiion of the Earth in x and y

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

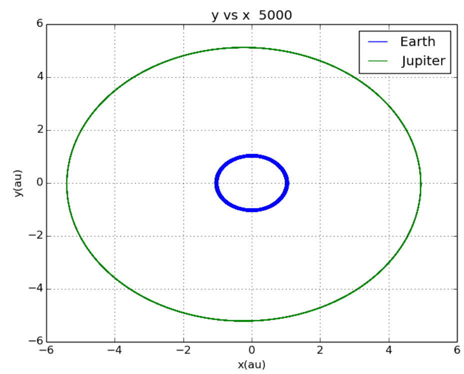
(d) Total Energy $(Au/yr)^2$ for n=5000

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.

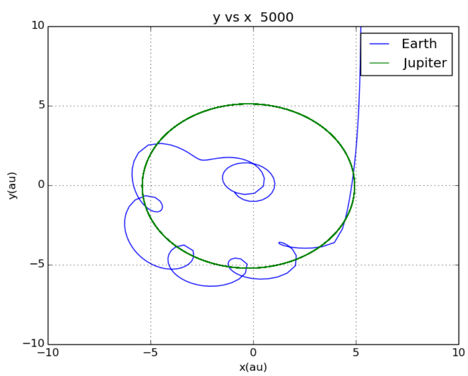
5.2 Earth Jupiter System



(a) Jupiter's original mass



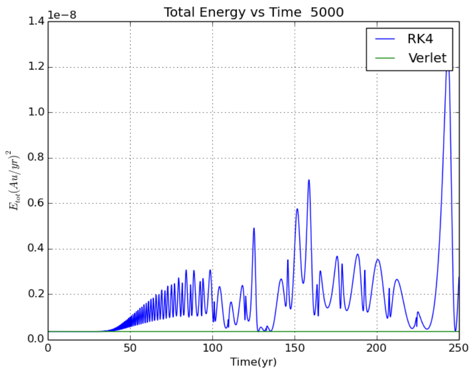
(b) Jupiter with 10 the original mass



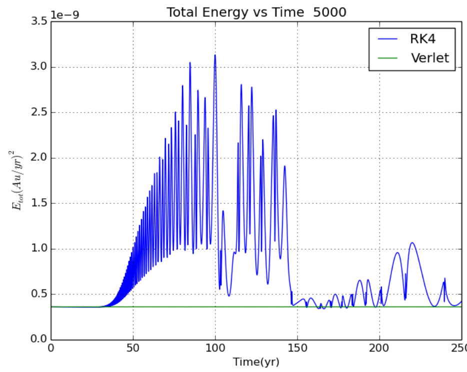
(c) Jupiter with 1000 the original mass

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 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.

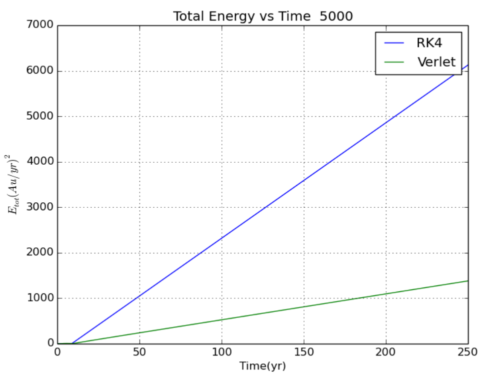
With the addition of Jupiter we see that Earth's obit is only slightly perturbed. However as we increase the mass of Jupiter to ten times the original we start seeing a more noticeable deviation. It is until Jupiter's mass is increase by one thousand where we see the Earth's obit is not stable anymore. 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

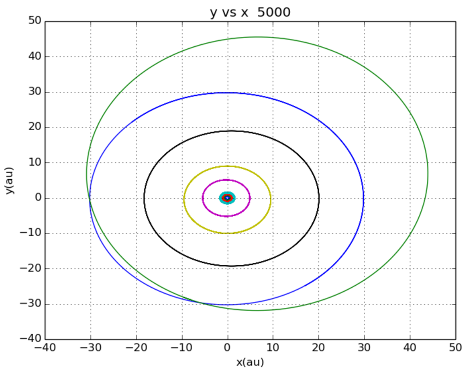


(c) 1000 times the original mass

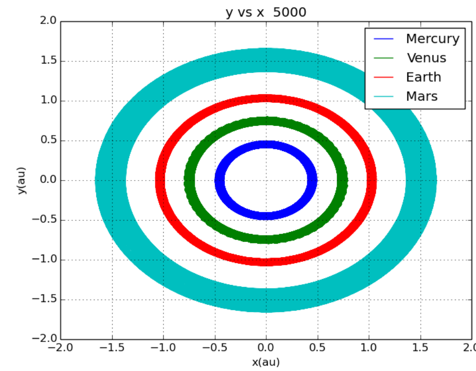
Figure 6: This shows the total energy for the Earth with different values of the mass of Jupiter. From this we can see that the RK4 method is very unstable and was not used to make plots of the positions. 5000 grid point were used to get the energies.

5.3 Solar System

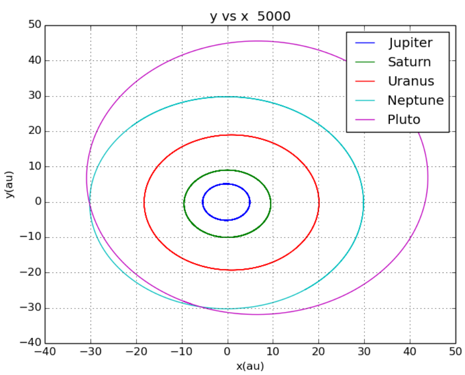
Here I include all of the planets and Pluto. From Figure 7 we see that the inner planets are perturbed more than the outer ones.



(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

```
#ifndef PLANET_H
#define PLANET_H
#include <cmath>
#define _USE_MATH_DEFINES

class planet
{


  public:
   //Properties
   int n;
   double mass,t_i,t_f;
   double pos[3];
   double vel[3];
   double L[3];

   //Initializers
   planet();
   planet(double M,int step);

   //Functions
   double Kin_E();
   double Ang_M();
};
#endif
```

Listing 3: This sets up the class for planets.

```
#include "/home/quetzalcoatl/Computational_Physics_work/Project3/Code/planet.h"
#include <iostream>

using namespace std;

 planet::planet(){}

 planet::planet(double M, int step)
 {
  n=step;
  mass=M;

   if(M == 1.0)
   {
      cout<<"Enter information for the SUN"<<endl;
   }
   else if(M ==1.6605e-7)
   {
      cout<<"Enter infomation for MERCURY"<<endl;
   }
   else if(M == 2.4483e-6)
          {
             cout<<"Enter information for the VENUS"<<endl;
          }
          else if(M ==3.0034e-6)
          {
```

```
         cout<<"Enter infomation for EARTH"<<endl;
      }
 else if(M ==3.2278e-7)
      {
         cout<<"Enter infomation for MARS"<<endl;
      }
else if(M == 9.5449e-4)
      {
         cout<<"Enter information for the JUPITER"<<endl;
      }
else if(M == 9.5449e-1)
      {
         cout<<"Enter information for the JUPITER"<<endl;
      }
else if(M == 9.5449e-3)
      {
         cout<<"Enter information for the JUPITER"<<endl;
      }
      else if(M == 2.8580e-4)
      {
         cout<<"Enter information for the SATURN"<<endl;
      }
      else if(M == 4.3656e-5)
      {
         cout<<"Enter infomation for URANUS"<<endl;
      }
 else if(M ==5.1506e-5 )
      {
         cout<<"Enter infomation for NEPTUNE"<<endl;
      }
      else if(M == 6.5728e-8)
      {
         cout<<"Enter information for the PLUTO"<<endl;
      }

int sepa;
cout<<"Do you want to input intial conditions,no(0) or yes(1): ";
cin>>sepa;
if(sepa == 1)
{
  cout<<"Initial position x_i: ";
  cin>>pos[0];
  cout<<"Intial position y_i: ";
  cin>>pos[1];
  cout<<"Initial position z_i: ";
  cin>>pos[2];
  cout<<"Initial velocity Vx_i: ";
  cin>>vel[0];
  cout<<"Initial velocity Vy_i: ";
  cin>>vel[1];
  cout<<"Initial velocity Vz_i: ";
  cin>>vel[2];
}
else
{
  if(M == 1.0)
        {
```

```
    //Sun
                pos[0]=-6.6275e-3;
                pos[1]=-3.42121e-3;
                pos[2]=1.96822e-4;
                vel[0]=(6.12836e-6)*365;
                vel[1]=(-6.7365e-6)*365;
                vel[2]=(-1.17571e-7)*365;

        }

        if(M ==1.6605e-7)
        {
    //Mercury
    pos[0]=-1.25329e-1;
                      pos[1]=-4.5394e-1;
                      pos[2]=-2.571179e-2;
                      vel[0]=(2.15648e-2)*365;
                      vel[1]=(-5.76037e-3)*365;
    vel[2]=(-2.44891e-3)*365;
        }
        if(M == 2.4483e-6)
        {
    //Venus
                      pos[0]=5.7835e-1;
                      pos[1]=-4.3512e-1;
                      pos[2]=-3.94689e-2;
                      vel[0]=(1.18909e-2)*365;
                      vel[1]=(1.61875e-2)*365;
                      vel[2]=(-4.64774e-4)*365;
        }

        if(M ==3.0034e-6)
        {
    //Earth
    pos[0]=-9.9140e-1;
                      pos[1]=-1.6994e-1;
                      pos[2]=1.98187e-4;
                      vel[0]=(2.5895e-3)*365;
                      vel[1]=(-1.70385e-2)*365;
                      vel[2]=(5.1364e-7)*365;
        }
        else if(M ==3.2278e-7)
        {
    //Mars
                      pos[0]=9.0233e-1;
                      pos[1]=1.1606;
                      pos[2]=2.2383e-3;
                      vel[0]=(-1.0490e-2)*365;
                      vel[1]=(9.797466e-3)*365;
                      vel[2]=(4.6327e-4)*365;
        }
  if(M == 9.5449e-4)
        {
            //Jupiter
    pos[0]=3.55378;
                      pos[1]=3.47587;
                      pos[2]=-9.39615e-2;
```

```
                    vel[0]=(-5.3695e-3)*365;
                    vel[1]=(5.7509e-3)*365;
                    vel[2]=(9.6353e-5)*365;

        }
        if(M == 9.5449e-1)
        {
   //Jupiter
                    pos[0]=3.55378;
                    pos[1]=3.47587;
                    pos[2]=-9.39615e-2;
                    vel[0]=(-5.3695e-3)*365;
                    vel[1]=(5.7509e-3)*365;
                    vel[2]=(9.6353e-5)*365;
        }
        if(M == 9.5449e-3)
        {
   //Jupiter
                    pos[0]=3.55378;
                    pos[1]=3.47587;
                    pos[2]=-9.39615e-2;
                    vel[0]=(-5.3695e-3)*365;
                    vel[1]=(5.7509e-3)*365;
                    vel[2]=(9.6353e-5)*365;
        }
        if(M == 2.8580e-4)
        {
  //Saturn
                    pos[0]=6.01042;
                    pos[1]=6.9007;
                    pos[2]=-3.59215e-1;
                    vel[0]=(-4.49875e-3)*365;
                    vel[1]=(3.65260e-3)*365;
                    vel[2]=(1.15655e-4)*365;
        }
        if(M == 4.3656e-5)
        {
  //Uranus
                    pos[0]=1.4660e1;
                    pos[1]=-1.3499e1;
                    pos[2]=-2.40100e-1;
                    vel[0]=(2.635377e-3)*365;
                    vel[1]=(2.71037e-3)*365;
                    vel[2]=(-2.40986e-5)*365;
        }
        if(M ==5.1506e-5 )
        {
  //Neptune
                    pos[0]=1.70329e1;
                    pos[1]=-2.4836e1;
                    pos[2]=1.18927e-1;
                    vel[0]=(2.567966e-3)*365;
                    vel[1]=(1.79313e-3)*365;
                    vel[2]=(-9.6080e-5)*365;
        }
        if(M == 6.5728e-8)
        {
```

```
        //Pluto
                              pos[0]=-9.6135;
                              pos[1]=-2.80971e1;
                              pos[2]=5.78738;
                              vel[0]=(3.0528e-3)*365;
                              vel[1]=(-1.50824e-3)*365;
                              vel[2]=(-7.17669e-4)*365;
             }


   }

   cout<<"Initial time t_i: ";
   cin>>t_i;
   cout<<"Final time t_f: ";
   cin>>t_f;

  }

  double planet::Kin_E()
  {
   double Velocity=this->vel[0]*this->vel[0]+this->vel[1]*this->vel[1]+this->vel
     [2]*this->vel[2];
   return 0.5*this->mass*Velocity;
  }

  double planet::Ang_M()
  {
   L[0]=this->pos[1]*this->vel[2]-this->pos[2]*this->vel[1];
   L[1]=this->pos[2]*this->vel[0]-this->pos[0]*this->vel[2];
   L[2]=this->pos[0]*this->vel[1]- this->pos[1]*this->vel[0];
   return this->mass*this->mass*(L[0]*L[0]+L[1]*L[1]+L[2]*L[2]);
  }
```

Listing 4: This gives commands for the class planets.

```
#ifndef SOLVER_H
#define SOLVER_H
#include <cmath>
#include "/home/quetzalcoatl/Computational_Physics_work/Project3/Code/planet.h"
#include <vector>
#include <iostream>
#include <fstream>
#include <iomanip>

using std::vector;

class solver
{

 public:
  friend class planet;

  //Properties
  int tot_p; //Total planets
  vector<planet> all_planets;
  double G;
```

```
  //Initializers
  solver();

  //Functions
  void add(planet newplanet); //adds new planet
  void verlet(planet &N,int type);
  void RK4(planet &N,int type);
  void Force(double &Fx, double &Fy,double &Fz, double x, double y,double z,
    double m);
  void Header_Pos(int type,std::ofstream& ofile);
  void Header_Energy(int type, std::ofstream& ofile);
  void Write_Pos(std::ofstream& ofile, double time);
  void Write_Energy(std::ofstream& ofile, double time, int type);
  void Potential(double &pot, double x, double y,double z, double m1, double m2
    );

};
#endif
```

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

```
#include "/home/quetzalcoatl/Computational_Physics_work/Project3/Code/planet.h"
#include "/home/quetzalcoatl/Computational_Physics_work/Project3/Code/solver.h"
#include "time.h"
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
#include <fstream>
#include <iomanip>


using namespace std;

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

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

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

```

```
char Ve[30];
char Energy2[30]; //contains the Kinetic energy, potential energy, and
  angular mom.
       cout<<"Enter the name of the Verlet file: ";
       cin>>Ve;
cout <<"Enter the name of the Energy file: ";
cin>>Energy2;

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

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

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

       for(int i=0; i< N.n; i++)
       {
    time=(i+1)*h;
    for(int j=0; j<tot_p; j++)
    {
             planet &este = all_planets[j];
       Fx=Fy=Fz=Fx1=Fy1=Fz1=0;
       Force(Fx,Fy,Fz,este.pos[0],este.pos[1],este.pos[2],1);
       if(type>0)
       {
   for(int l=0; l<tot_p;l++)
   {
       if(l == j)
       {
     Fx+=0;
     Fy+=0;
     Fz+=0;
             }
       else
       {
         planet &otro = all_planets[l];
         for(int d =0; d<3; d++)
         {
       rel_pos[d] = este.pos[d] - otro.pos[d];
         }
         Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
       }
   }

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

       //Update the position .
```

```
        for(int l=0; l<3; l++)
        {
    este.pos[l] += h*este.vel[l] +0.5*h*h*acc[j][l];
        }

        //Values for the velocity
        Force(Fx1,Fy1,Fz1,este.pos[0],este.pos[1],este.pos[2],1);
        if(type>0)
        {
          for(int l=0; l<tot_p;l++)
                  {
                      if(l == j)
                      {
                          Fx1+=0;
                          Fy1+=0;
      Fz1+=0;
                      }
                      else
                      {
                          planet &otro = all_planets[l];
                          for(int d =0; d<3; d++)
                          {
                                  rel_pos[d] = este.pos[d] - otro.pos[d];
                          }
                          Force(Fx1,Fy1,Fz1,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
                      }
                  }
         }

        Nacc[j][0] = Fx1;
        Nacc[j][1] = Fy1;
        Nacc[j][2] = Fz1;

        //Calculate new velocity
        for(int l=0; l<3; l++)
        {
    este.vel[l] += 0.5*h*(acc[j][l] + Nacc[j][l]);
        }

     }
    //Write the Updated the values
           Write_Pos(ofile,time);
           Write_Energy(E_output,time,type);

        }

  //stop clock and display time
  finish_VV=clock();
  cout<<"Total Time VV: "<<((double)(finish_VV - start_VV)/CLOCKS_PER_SEC)<<endl;


  ofile.close();
  E_output.close();

}
```

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

 //Setting up ks, the first [] is the total planets to be solved for and
 //the second is the number of dimensions
 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];


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

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

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

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

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

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

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

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

```
   Fx+=0.0;
   Fy+=0.0;
   Fz+=0.0;
       }
       else
       {
         planet &otro=all_planets[l];
         for(int a=0; a<3;a++){rel_pos[a]=-(otro.pos[a]-este.pos[a]);}
               Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
       }
    }
 }


       k1_v[j][0] = h*Fx;
       k1_v[j][1] = h*Fy;
 k1_v[j][2] = h*Fz;
       for(int l= 0; l<3;l++)
 {
    k1_x[j][l] = h*este.vel[l];
 }
  }//End of loop

  //Setting up K2
  for(int j=0; j<tot_p; j++)
        {
             planet &este=all_planets[j];
             Fx=Fy=0.0;

 for(int a=0; a<3;a++)
             {
                rel_pos[a]=este.pos[a]+k1_x[j][a]/2.0;
             }
             Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);

 if(type >0)
 {
                for(int l=0; l<tot_p; l++)
                {
       if(j == l)
       {
   Fx+=0.0;
   Fy+=0.0;
   Fz+=0.0;
       }
       else
       {
                    planet &otro=all_planets[l];
                    for(int a=0; a<3;a++)
                    {
                      rel_pos[a]=-((otro.pos[a]+k1_x[l][a]/2.0)-(este.pos[a]+k1_x[j][a]/2.0));
                    }
                    Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
       }
                }
```

```
    }

                 k2_v[j][0] = h*Fx;
                 k2_v[j][1] = h*Fy;
    k2_v[j][2] = h*Fz;
                 for(int l= 0; l<3;l++)
                 {
                    k2_x[j][l] = h*(este.vel[l]+ k1_v[j][l]/2.0);
                 }
             }//End of loop

       //Setting up K3
       for(int j=0; j<tot_p; j++)
             {
                 planet &este=all_planets[j];
                 Fx=Fy=0.0;


                 for(int a=0; a<3;a++)
                 {
                   rel_pos[a]=este.pos[a]+k2_x[j][a]/2.0;
                 }
                 Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
    if(type>0)
    {
                     for(int l=0; l<tot_p; l++)
                     {
             if(j == l)
             {
        Fx+=0.0;
        Fy+=0.0;
        Fz+=0.0;
             }
             else
             {
                               planet &otro=all_planets[l];
                               for(int a=0; a<3;a++)
                               {
                                rel_pos[a]=-((otro.pos[a]+k2_x[l][a]/2.0)-(este.pos[a]+k2_x[j][a]/2.0));
                               }
                               Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.mass);
             }
                     }
    }

                 k3_v[j][0] = h*Fx;
                 k3_v[j][1] = h*Fy;
    k3_v[j][2] = h*Fz;
                 for(int l= 0; l<2;l++)
                 {
                    k3_x[j][l] = h*(este.vel[l]+ k2_v[j][l]/2.0);
                 }
             }//End of loop

       //Setting up K4
```

```
        for(int j=0; j<tot_p; j++)
            {
                planet &este=all_planets[j];
                Fx=Fy=Fz=0.0;

                for(int a=0; a<3;a++)
                {
                    rel_pos[a]=este.pos[a]+k3_x[j][a];
                }
                Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],1);
    if(type>0)
    {
                    for(int l=0; l<tot_p; l++)
                    {
          if(j == l)
          {
      Fx+=0.0;
      Fy+=0.0;
      Fz+=0.0;
                }
          else
          {
                          planet &otro=all_planets[l];
                          for(int a=0; a<3;a++)
                          {
                              rel_pos[a]=-((otro.pos[a]+k3_x[l][a])-(este.pos[a]+
    k3_x[j][a]));
                          }
                          Force(Fx,Fy,Fz,rel_pos[0],rel_pos[1],rel_pos[2],otro.
    mass);
          }
                    }
    }

                k4_v[j][0] = h*Fx;
                k4_v[j][1] = h*Fy;
    k4_v[j][2] = h*Fz;
                for(int l= 0; l<3;l++)
                {
                    k4_x[j][l] = h*(este.vel[l]+ k3_v[j][l]);
                }
            }//End of loop

        //This updates the functions
            for(int j=0;j<tot_p;j++)
            {
                planet &este =all_planets[j];

    for(int l=0; l<3; l++)
    {
        este.pos[l] += (k1_x[j][l] +2*(k2_x[j][l]+k3_x[j][l]) +k4_x[j][l])/6.0;
        este.vel[l] += (k1_v[j][l] +2*(k2_v[j][l]+k3_v[j][l]) +k4_v[j][l])/6.0;
    }
            }

          //Write updated values to the file
          Write_Pos(ofile,time);
```

```
            Write_Energy(E_output,time,type);

    }

  finish_RK = clock();
  cout<<"Total Time RK: "<<((double)(finish_RK -start_RK)/CLOCKS_PER_SEC)<<endl;

  ofile.close();
  E_output.close();

}

void solver::Force(double &Fx, double &Fy,double &Fz, double x, double y, double z, double m)
{
  double r= sqrt(x*x+y*y+z*z);

  Fx -= (G/(r*r*r))*x*m;
  Fy -= (G/(r*r*r))*y*m;
  Fz -= (G/(r*r*r))*z*m;
}

void solver::Header_Pos(int type,std::ofstream& ofile)
{

  ofile<<setiosflags(ios::showpoint | ios::uppercase);
  ofile<<"#Time(Yr)";

  if(type == 0)
  {
      ofile<<setw(20)<<"X_Ea"<<setw(20)<<"Y_Ea"<<setw(20)<<"Z_Ea";
      ofile<<setw(20)<<"R_earth"<<endl;
  }
  else if(type == 1)
  {
      ofile<<setw(20)<<"X_Ea"<<setw(20)<<"Y_Ea"<<setw(20)<<"Z_Ea";
      ofile<<setw(20)<<"R_Earth";
      ofile<<setw(20)<<"X_Ju"<<setw(20)<<"Y_Ju"<<setw(20)<<"Z_Ju";
      ofile<<setw(20)<<"R_Jupiter"<<endl;
  }
  else if(type == 2)
  {
      ofile<<setw(20)<<"X_Su"<<setw(20)<<"Y_Su"<<setw(20)<<"Z_Su";
      ofile<<setw(20)<<"R_Sun";
      ofile<<setw(20)<<"X_Ea"<<setw(20)<<"Y_Ea"<<setw(20)<<"Z_Ea";
      ofile<<setw(20)<<"R_Earth";
            ofile<<setw(20)<<"X_Ju"<<setw(20)<<"Y_Ju"<<setw(20)<<"Z_Ju";
      ofile<<setw(20)<<"R_Jupiter"<<endl;
  }
  else if(type == 3)
  {
      ofile<<setw(20)<<"X_Me"<<setw(20)<<"Y_Me"<<setw(20)<<"Z_Me";
      ofile<<setw(20)<<"R_Mercury";
            ofile<<setw(20)<<"X_Ve"<<setw(20)<<"Y_Ve"<<setw(20)<<"Z_Ve";
      ofile<<setw(20)<<"R_Venus";
            ofile<<setw(20)<<"X_Ea"<<setw(20)<<"Y_Ea"<<setw(20)<<"Z_Ea";
```

```
        ofile <<setw(20)<<"R_Earth";
        ofile <<setw(20)<<"X_Ma"<<setw(20)<<"Y_Ma"<<setw(20)<<"Z_Ma";
        ofile <<setw(20)<<"R_Mars";
              ofile <<setw(20)<<"X_Ju"<<setw(20)<<"Y_Ju"<<setw(20)<<"Z_Ju";
        ofile <<setw(20)<<"R_Jupiter";
        ofile <<setw(20)<<"X_Sa"<<setw(20)<<"Y_Sa"<<setw(20)<<"Z_Sa";
        ofile <<setw(20)<<"R_Saturn";
              ofile <<setw(20)<<"X_Ur"<<setw(20)<<"Y_Ur"<<setw(20)<<"Z_Ur";
        ofile <<setw(20)<<"R_Uranus";
              ofile <<setw(20)<<"X_Ne"<<setw(20)<<"Y_Ne"<<setw(20)<<"Z_Ne";
        ofile <<setw(20)<<"R_Neptune";
              ofile <<setw(20)<<"X_Pu"<<setw(20)<<"Y_Pu"<<setw(20)<<"Z_Pu";
        ofile <<setw(20)<<"R_Pluto"<<endl;
   }
 }

void solver::Header_Energy(int type,std::ofstream& ofile)
 {
           ofile<<setiosflags(ios::showpoint | ios::uppercase);
           ofile<<"#Time(Yr)";

           if(type == 0)
           {
              ofile <<setw(20)<<"K_Ea"<<setw(20)<<"U_Ea"<<setw(20)<<"Etot_Ea";
        ofile <<setw(20)<<"L_Ea"<<endl;
           }
           else if(type == 1)
           {
              ofile <<setw(20)<<"K_Ea"<<setw(20)<<"U_Ea"<<setw(20)<<"Etot_Ea";
        ofile <<setw(20)<<"L_Ea";
              ofile <<setw(20)<<"K_Ju"<<setw(20)<<"U_Ju"<<setw(20)<<"Etot_Ju";
        ofile <<setw(20)<<"L_Ju"<<endl;
           }
           else if(type == 2)
           {
              ofile <<setw(20)<<"K_Su"<<setw(20)<<"U_Su"<<setw(20)<<"Etot_Su";
        ofile <<setw(20)<<"L_Su";
              ofile <<setw(20)<<"K_Ea"<<setw(20)<<"U_Ea"<<setw(20)<<"Etot_Ea";
        ofile <<setw(20)<<"L_Ea";
              ofile <<setw(20)<<"K_Ju"<<setw(20)<<"U_Ju"<<setw(20)<<"Etot_Ju";
        ofile <<setw(20)<<"L_Ju"<<endl;
           }
           else if(type == 3)
           {
              ofile <<setw(20)<<"K_Me"<<setw(20)<<"U_Me"<<setw(20)<<"Etot_Me";
        ofile <<setw(20)<<"L_Me";
              ofile <<setw(20)<<"K_Ve"<<setw(20)<<"U_Ve"<<setw(20)<<"Etot_Ve";
        ofile <<setw(20)<<"L_Ve";
              ofile <<setw(20)<<"K_Ea"<<setw(20)<<"U_Ea"<<setw(20)<<"Etot_Ea";
        ofile <<setw(20)<<"L_Ea";
              ofile <<setw(20)<<"K_Ma"<<setw(20)<<"U_Ma"<<setw(20)<<"Etot_Ma";
        ofile <<setw(20)<<"L_Ma";
              ofile <<setw(20)<<"K_Ju"<<setw(20)<<"U_Ju"<<setw(20)<<"Etot_Ju";
        ofile <<setw(20)<<"L_Ju";
              ofile <<setw(20)<<"K_Sa"<<setw(20)<<"U_Sa"<<setw(20)<<"Etot_Sa";
        ofile <<setw(20)<<"L_Sa";
              ofile <<setw(20)<<"K_Ur"<<setw(20)<<"U_Ur"<<setw(20)<<"Etot_Ur";
```

```
      ofile<<setw(20)<<"L_Ur";
            ofile<<setw(20)<<"K_Ne"<<setw(20)<<"U_Ne"<<setw(20)<<"Etot_Ne";
      ofile<<setw(20)<<"L_Ne";
            ofile<<setw(20)<<"K_Pl"<<setw(20)<<"U_Pl"<<setw(20)<<"Etot_pl";
      ofile<<setw(20)<<"L_Pl"<<endl;
         }
}

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

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

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

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

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

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

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

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

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

```
          if(i == j)
          {
        pot+=0.0;
                        }
          else
          {
        for(int l=0; l<3; l++)
        {
            rel_p[l] = este.pos[l] - otro.pos[l];
        }

        Potential(pot,rel_p[0],rel_p[1],rel_p[2],este.mass, otro.mass);
            }
      }
      ofile<<setw(20)<<pot;
       }

             //Writes total energy
       Etot = este.Kin_E() + pot;
       ofile<<setw(20)<<Etot;

       //Calculated the angular momentum
             ofile<<setw(20)<<setprecision(7)<<este.Ang_M();
  }
  ofile<<endl;
}

void solver::Potential(double &pot, double x, double y,double z, double m1,
   double m2)
{
  double r = sqrt(x*x+y*y+z*z);
  pot -= (G*m1*m2)/r;
}
```

Listing 6: This gives commands for the solver system.

```
#include <iostream>
#include "/home/quetzalcoatl/Computational_Physics_work/Project3/Code/planet.h"
#include "/home/quetzalcoatl/Computational_Physics_work/Project3/Code/solver.h"
#include <cmath>

using namespace std;

int main()
{
  solver systemRK;
  solver systemVV;
  int type;
  int n;

  /*Chose what system will be solved.
    type = 0 solves the Earth-Sun system with the sun stationary
    type =1 solves the Earth-Sun-Jupiter system with the sun stationary
    type = 2 solves the Earth-Sun-Jupiter system with the sun, all with repect
   to the center of mass
    type =3 solves the whole solar sytem with the sun stationary
  */
```

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


planet Earth(3.0034e-6, n);

if(type == 0)
{
systemRK.add(Earth);
systemVV.add(Earth);
}
else if(type ==1)
{
systemRK.add(Earth);
systemVV.add(Earth);
int typeJ;
cout<<"Mass of Jupiter; 0:normal, 1:10times, 2:E3times: ";
        cin>>typeJ;
if(typeJ == 0)
        {
    planet Jupiter(9.5449e-4,100);
    systemRK.add(Jupiter);
    systemVV.add(Jupiter);
        }
if(typeJ == 1)
        {
    planet Jupiter(9.5449e-3,100);
            systemRK.add(Jupiter);
            systemVV.add(Jupiter);
}
if(typeJ == 2)
   {
    planet Jupiter(9.5449e-1,100);
            systemRK.add(Jupiter);
            systemVV.add(Jupiter);
}

}
else if(type == 2)
{
planet Sun(1,100);
systemRK.add(Sun);
systemVV.add(Sun);
        systemRK.add(Earth);
systemVV.add(Earth);
        planet Jupiter(9.5449e-4,100);
        systemRK.add(Jupiter);
systemVV.add(Jupiter);
}
else if(type == 3)
{
planet Mercury(1.6605e-7,100);
systemRK.add(Mercury);
systemVV.add(Mercury);
```

```
  planet Venus(2.4483e-6,100);
  systemRK.add(Venus);
  systemVV.add(Venus);
        systemRK.add(Earth);
  systemVV.add(Earth);
  planet Mars(3.2278e-7,100);
  systemRK.add(Mars);
  systemVV.add(Mars);
        planet Jupiter(9.5449e-4,100);
        systemRK.add(Jupiter);
  systemVV.add(Jupiter);
  planet Saturn(2.8580e-4,100);
  systemRK.add(Saturn);
  systemVV.add(Saturn);
  planet Uranus(4.3656e-5,100);
  systemRK.add(Uranus);
  systemVV.add(Uranus);
  planet Neptune(5.1506e-5,100);
  systemRK.add(Neptune);
  systemVV.add(Neptune);
  planet Pluto(6.5728e-8,100 );
  systemRK.add(Pluto);
  systemVV.add(Pluto);
  }

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

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