Numerical Solutions to the Problem of the Solar System

Elizabeth Drueke

April 12, 2016

Abstract

The study of the solar system is a good window into how planetary bodies orbit and interact. As a result, determining numerical methods for solving for the orbits of planets in the Milky Way have broad applications in many areas of astrophysics. Here, we work on developing two such methods – the Verlet and 4th-order Runge-Kutta algorithms – and apply them to the problem of our solar system, focusing particularly on stability of orbits and computation time required to do the calculations.

1 Introduction

For as long as humans have been on the earth, we have been looking to the sky. Ancient peoples relied on the stars and planets to tell when the seasons were changing, so they would know when was a good time to plant crops for food. A little later, the stars were used by mariners as a compass, guiding them across the seas.

Today still we look towards the skies. However, we arguably know more about them. We know that our solar system is not the center of the universe, nor are we the center of our solar system. We know that our sun is a middle-sized star, and without its light and heat there would be no life on earth. We know that the rotation of the planets around the sun is dictated by Newton's Second Law, and, with this, we can calculate the trajectory of the planetary bodies around us.

Here, we present two such calculations. Using the Verlet and 4^{th}-Order Runge-Kutta (RK4) Methods, we investigate the motion of the planets around the sun. We begin by presenting the simple theory of planetary motion in Section 2. Then we discuss the Verlet and RK4 methods in Sections 2.1 and 2.2. After this, we discuss the framework and the algorithm developed particularly in this project in Section 3. Finally, results and benchmarks for the code are discussed in Section 4.

2 Theory

As mentioned in Section 1, the movements of planets are dictated by Newton's Second Law, which states

$$\vec{F} = m\vec{a}.$$

Thus, we have a second order differential equation:

$$m\frac{d^2\vec{x}}{dt^2} = \vec{F}$$

where $\vec{F}$ is the sum of the forces on the planet in question. In particular, the force of one planet on another is given by

$$\vec{F} = -\frac{Gm_1m_2}{r^2}\hat{r} \tag{2.1}$$

where $G = 6.67 \times 10^{-11}m^3kg^{-1}s^{-2}$ is the gravitational constant, $m_{1,2}$ are the masses of the two planets, and r is the distance between the planets.

Now, we want to be able to use some sort of discretized version of this in order to use a computer to approximate a numerical solution to this problem. Our first step is then to look at Eq. 2.1 component-wise (ie. look at the x- and y- components separately). In particular, we should have

$$m\frac{d^2x}{dt^2} = F_x \text{ and } m\frac{d^2y}{dt^2} = F_y$$

Noting that $\vec{r} = x\hat{x} + y\hat{y}$, this gives us that

$$F_x = -\frac{Gm_1m_2x}{r^3} \text{ and } F_y = -\frac{Gm_1m_2y}{r^3}$$

Thus we have two coupled second-order differential equations:

$$\frac{d^2x}{dt^2} = -\frac{Gm_1x}{r^3} \text{ and } \frac{d^2y}{dt^2} = -\frac{Gm_1y}{r^3} \tag{2.2}$$

or, alternatively, four coupled first-order differential equations:

$$\begin{aligned} \frac{dx}{dt} &= v_x, \quad & \frac{dv_x}{dt} &= -\frac{Gm_1x}{r^3}, \\ \frac{dy}{dt} &= v_y, \quad & \frac{dv_y}{dt} &= -\frac{Gm_1y}{r^3}. \end{aligned} \tag{2.3}$$

In this analysis, we first investigate the unperturbed earth-sun system. In this case, Eq. 2.1 becomes

$$\vec{F} = \frac{GM_\odot m_E}{r^2}\hat{r} \tag{2.4}$$

where m_E is the mass of the earth and $M_\odot$ is the mass of the sun. Assuming a circular orbit, we can say that

$$a = \frac{mc^2}{r},$$

and so we have

$$\frac{mv^2}{r} = \frac{GM_\odot m_E}{r^2}$$

or

$$v^2 r = GM_\odot = 4\pi^2 AU^2 yr^{-2}.$$

Thus, for the unperturbed earth-sun system, we wish to investigate

$$\frac{d^2x}{dt^2} = -\frac{4\pi^2 x}{r^3} \text{ and } \frac{d^2y}{dt^2} = -\frac{4\pi^2 y}{r^3}. \tag{2.5}$$

We will also want to look at adding other planets to our solar system. After all, the unperturbed earth-sun system is really to simplistic to be a reasonable approximation for how the solar system. Noting that

$$Gm_p = GM_\odot \frac{m_p}{M_\odot} = 4\pi^2 \frac{m_p}{M_\odot},$$

we have, for planet $p\prime$,

$$\begin{aligned} a_x = \frac{dv_x}{dt} = & -\frac{4\pi^2}{r_{p\prime\odot}^3}\left(x_{p\prime} - x_\odot\right) - \frac{4\pi^2}{M_\odot}\sum_{p\neq p\prime}\frac{m_p\left(x_{pprime} - x_p\right)}{r_{pp\prime}^3}, \\ a_y = \frac{dv_y}{dt} = & -\frac{4\pi^2}{r_{p\prime\odot}^3}\left(y_{p\prime} - y_\odot\right) - \frac{4\pi^2}{M_\odot}\sum_{p\neq p\prime}\frac{m_p\left(y_{p\prime} - y_p\right)}{r_{pp\prime}^3}. \end{aligned} \tag{2.6}$$

The solution of Eq. 2.5 is fairly straightforward (we are only really looking at two coupled second-order differential equations), although still not simple by any means. However, the solution to Eq. 2.6 is impossible to get by hand. The number of couple equations will be twice the number of planets in the solar system. Thus, to solve either system, it is useful to turn to numerical approximations and computer algorithms. In particular, we look into the Verlet and RK4 methods as means by which to solve the system.

2.1 Verlet Method

It is a common practice in creating computer algorithms to solve complex problems to discretize the equations in order to get something more concrete to work with. In this case, we will discretize using the Taylor Series expansion. That is, we will have

$$x\left(t+h\right) = x\left(t\right) + hx^{'}\left(t+h\right) + \frac{h^2}{2}x^{''}\left(t+h\right) + O\left(h^3\right) \tag{2.7}$$

Thus, we can say, letting $x_i = x\left(t_0 + hi\right)$, that, for planet $p\prime$ in the multi-planet system,

$$\begin{aligned} x_{i+1} &= x_i + hv_i + \frac{h^2}{2}v_i^{'} + O\left(h^3\right) \\ &= x_i + hv_i + \frac{h^2}{2}\left(-\frac{4\pi^2}{r_{p\prime\odot i}^{\,3}}\left(x_{p\prime} - x_\odot\right)_i - \frac{4\pi^2}{M_\odot}\sum_{p\neq p\prime}\frac{m_p\left(x_{p\prime} - x_p\right)_i}{r_{pp\prime i}^{\,3}}\right) + O\left(h^3\right). \end{aligned} \tag{2.8}$$

We can similarly discretize the velocity of planet $p\prime$ to find

$$
\begin{aligned}
v_{i+1} &= v_i + \frac{h}{2}\left(v_{i+1}^{'} + v_i^{'}\right) + O\left(h^2\right) \\
&= v_i + \frac{h}{2}\left(-4\pi^2\left(\frac{\left(x_{p\prime} - x_\odot\right)_i}{r_{p\prime\odot i}^{\,3}} + \frac{\left(x_{p\prime} - x_\odot\right)_{i+1}}{r_{p\prime\odot i+1}^{\,3}}\right)\right. \\
&\left. - \frac{4\pi^2}{M_\odot}\sum_{p\neq p\prime} m_p\left(\frac{\left(x_{p\prime} - x_p\right)_i}{r_{pp\prime i}^{\,3}} + \frac{\left(x_{p\prime} - x_p\right)_{i+1}}{r_{pp\prime i+1}^{\,3}}\right)\right) + O\left(h^3\right).
\end{aligned} \tag{2.9}
$$

In the case of the unperturbed earth-sun system, Eqs. 2.8 and 2.9 simplify to

$$
x_{i+1} = x_i + hv_i + \frac{h^2}{2}\left(-\frac{4\pi^2}{r_{p\prime\odot i}^{\,3}}\left(x_{p\prime} - x_\odot\right)_i\right) + O\left(h^3\right)
$$

and

$$
v_{i+1} = v_i + \frac{h}{2}\left(-4\pi^2\left(\frac{\left(x_{p\prime} - x_\odot\right)_i}{r_{p\prime\odot i}^{\,3}} + \frac{\left(x_{p\prime} - x_\odot\right)_{i+1}}{r_{p\prime\odot i+1}^{\,3}}\right)\right) + O\left(h^3\right).
$$

Together, Eqs. 2.8 and 2.9 make up what is known as the Verlet method [4]. With the introduction of the velocity Verlet method, this method is self-starting.

2.2 Fourth-Order Runge-Kutta

The RK4 method is a bit more precise than the Verlet method discussed in Section 2.1. It is based on the observation that, for

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

we can say

$$
y\left(t\right) = \int f\left(t, y\right) dt.
$$

Discretizing, this yields

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

Letting $y_{i+1/2} = y\left(t_i + h/2\right)$, and using the midpoint formula for the integral, we find

$$
\int_{t_i}^{t_{i+1}} f\left(t, y\right) dt \approx hf\left(t_{i+1/2}, y_{i+1/2}\right) + O\left(h^3\right).
$$

Thus, we have

$$
y_{i+1} = y_i + hf\left(t_{i+1/2}, y_{i+1/2}\right) + O\left(h^3\right).
$$

However, it is clear that, in order to use this method, we must have some idea of what $y_{i+1/2}$ is. To get this quantity, we use Euler's method to approximate it:

$$y_{i+1/2} \approx y_i + \frac{h}{2} f\left(t_i, y_i\right).$$

This leads us to the 2^{nd}-Order Runge-Kutta Method, or RK2, which says that, for

$$\begin{aligned} k_1 &= hf\left(t_i, y_i\right) \\ k_2 &= hf\left(t_{i+1/2}, y_i + k_1/2\right), \end{aligned} \tag{2.10}$$

we have

$$y_{i+1} \approx y_i + k_2 + O\left(h^2\right). \tag{2.11}$$

We can go through another similar sequence of steps to get to RK4, culminating in the following definitions:

$$\begin{aligned} k_1 &= hf\left(t_i, y_i\right) \\ k_2 &= hf\left(t_i + h/2, y_i + k_1/2\right), \\ k_3 &= hf\left(t_i + h/2, y_i + k_2/2\right), \\ k_4 &= hf\left(t_i + h, y_i + k_3\right), \\ y_{i+1} &\approx y_i + (1/6)\left(k_1 + 2k_2 + 2k_3 + k_4\right) + O\left(h^4\right). \end{aligned} \tag{2.12}$$

3 The Algorithm

The code developed for this project is written in C++ using the ROOT [1] framework for plotting. Code from [2] and [3] was reused, particularly the thevec and themat classes defined in classes.C. However, several new classes and functions were designed specifically for the problem of solving the solar system.

In header file `odesolvers.h` we define the `Verlet` and `RK4` functions, each of which return two `thevec`s, one of the discretized position solution and the other of the discretized velocity solution.

- The `Verlet` method begins by determining a step size `h` according to certain inputs (the initial time `t0` and final time `tf`, as well as the number of steps `nsteps`) as

  ```
  h = (1.0*tf-1.0*t0)/(1.0*nsteps);
  ```

  and then sets the first available position point to be

  ```
  pos[0] = x[0];
  pos[1] = pos[0]+h*v0
  ```

  for some given initial position `x0`. It then iterates over integers less than `nsteps` in order to completely solve the position vector:

```
for(int i=2;i<nsteps+1;i++){
     pos[i] = 2*pos[i-1]-pos[i-2]-a*pow(h,2)*pos[i-1]/r;
}
```

where `a` is the $4\pi^2$ factor. It then solves for the velocity vector by setting

```
vel[0] = v0;
```

for some given initial velocity `v0` and then iterating again over integers less than `nsteps` as follows:

```
for(int i=1;i<nsteps;i++){
     vel[i] = vel[i-1]+(h/2)*a*pos[i]/pow(r,3)-a*pos[i-1]/pow(r,3);
}
```

where `r` is the averate distance between the planet and the sun.

- The `RK4` algorithm computes the step size `h` and initializes the position and velocity vectors as the `Verlet` algorithm does, and then iterates over integers less than `nsteps` as follows:

```
for(int i=0;i<nsteps;i++){
     double k1p = h*vel[i-1];
     double k1v = h*a*pos[i-1]/pow(r,3);

     double k2p = h*vel([i-1]+k1v/2);
     double k2v = -h*a*(pos[i-1]+k1p/2)/pow(r,3);

     double k3p = h*(vel[i-1]+k2v/2);
     double k3v = -h*a*(pos[i-1]+k2p/2)/pow(r,3);

     double k4p = h*(vel[i-1]+k3v);
     double k4v = -h*a*(pos[i-1]+k3p)/pow(r,3);

     pos[i] = pos[i-1]+(1/6)*(k1p+2*k2p+2*k3p+k4p);
     vel[i] = vel[i-1]+(1/6)*(k1v+2*k2v+2*k3v+k4v);
}
```

These algorithms clearly are only designed to solve one direction (x or y) at a time, and are designed only to handle the earth-sun system with the earth in an assumed circular orbit. The multi-body problem was solved by a similar code in `solar_system.C`.

We also, for this project, found it prudent to define two new classes: `planet` and `solar_system`. The `planet` class is declared and defined in `planets.h` and `planets.C`. Each object of type `planet` has associated it with it a `mass` (`double` of mass in kilograms),

`dist_sun` (`double` of the average distance between the planet and the sun, `name` (`string` of the name of the planet), `acc` (`double` of the acceleration the planet would have in a pure, idealized sun-planet system with a circular orbit), `v0` (`double` of the average velocity of the planet around the sun), and eight `thevec` which are used to hold the positions and velocities in the x- and y-directions after they are solved for by the `RK4` and `Verlet` algorithms. These last objects, however, are not directly used by the `planet` class. It only became obvious after the class had been designed, during the designing stage of the `solar_system` class, that it would be convenient for the `planet` objects to have these components. Thus they are used solely by the `solar_system` class as discussed below.

The `planet` class has several constructors as well as functions:

- `print()`, which returns a string one might use to print the planet information to screen,
- `kinetic()`, which calculates the kinetic energy of the planet from a velocity which is taken as an argument,
- `potential()`, which calculates the potential energy of the planet from a distance from the sun which is taken as an argument,
- `ang_mom()`, which calculates the angular momentum of the planet from a distance from the sun and a velocity, which are both taken as arguments.

These algorithms are all fairly straightforward and do not merit discussion here.

The `solar_system` class is really the meat of the code. Defined and declared in `solar_system.h` and `solar_system.C`, this class was designed to solve the multi-body problem in full. Each object of type `solar_system` has associated with it `planets` (`vector<planet*>` of the planets to be included in the solar system), `nsteps` (`int` of the number of steps to be used by the Verlet and RK4 algorithms discussed below), `tf` (`double` of the final time to be used by the RK4 and Verlet algorithms), and `originx` and `originy` (`double` which describe the x- and y-components of the position of the origin to be used in the calculations). This class also has several constructors, as well as the following functions:

- `Add()`, which adds a planet, taken as an argument, to the solar system
- `Solve_Verlet()`, which solves the multi-body solar system using the Verlet algorithm. In particular, after calculating the step size as

  ```
  double h = tf/nsteps;
  ```

  and defining the $-4\pi^2$ factor as `fact` and the mass of the sun as `msun`, this algorithm creates a dynamic array of the planets in `planets` and initializes their position and velocity `thevec` objects by using their `v0` and `dist_sun` and the initial x-velocity and y-positions, respectively. Then, there is a `for` loop which iterates over integers `i` which are less than `nsteps`. For each `i`, it goes calculates the both the position at i for every planet in the solar system:

```
for(unsigned int it=0;it<planets.size();it++){

   planet myplan = theplanets[it];

   double vx_prev = myplan.velocitiesx_v[i-1];
   double vy_prev = myplan.velocitiesy_v[i-1];
   double x_prev = myplan.positionsx_v[i-1];
   double y_prev = myplan.positionsy_v[i-1];

   double r_prev = sqrt(pow(x_prev-originx,2)
                           +pow(y_prev-originy,2));

   double x = x_prev+h*vx_prev;
   double y = y_prev+h*vy_prev;

   for(unsigned int m = 0;m<planets.size();m++){

      planet plan = theplanets[m];

      if(myplan != plan){

         double xp_prev = plan.positionsx_v[i-1];
         double yp_prev = plan.positionsy_v[i-1];

         double rp_prev = sqrt(pow(x_prev-xp_prev,2)
                                    +pow(y_prev-yp_prev,2));

         x += (pow(h,2)/2)*fact*plan.mass
                  *(x_prev-xp_prev)/(msun*pow(rp_prev,3));
         y += (pow(h,2)/2)*fact*plan.mass
                  *(y_prev-yp_prev)/(msun*pow(rp_prev,3));

      }
   }

theplanets[it].positionsx_v[i]=x;
theplanets[it].positionsy_v[i]=y;
```

Then, the algorithm solves for the velocity at i for each of the planets:

```
for(unsigned int it = 0;it<planets.size();it++){

   planet myplan = theplanets[it];

   double vx_prev = myplan.velocitiesx_v[i-1];
```

```
      double vy_prev = myplan.velocitiesy_v[i-1];
      double x_prev = myplan.positionsx_v[i-1];
      double y_prev = myplan.positionsy_v[i-1];

      double r_prev = sqrt(pow(x_prev-originx,2)+pow(y_prev-originy,2));

      double x = myplan.positionsx_v[i];
      double y = myplan.positionsy_v[i];
      double newr = sqrt(pow(x-originx,2)+pow(y-originy,2));

      double vx = vx_prev;
      double vy = vy_prev;

      for(unsigned int m = 0;m<planets.size();m++){

         planet plan = theplanets[m];

         if(myplan != plan){

            double xp_prev = plan.positionsx_v[i-1];
            double yp_prev = plan.positionsy_v[i-1];

            double xp_cur = plan.positionsx_v[i];
            double yp_cur = plan.positionsy_v[i];

            double rp_cur = sqrt(pow(x-xp_cur,2)+pow(y-yp_cur,2));
            double rp_prev = sqrt(pow(x_prev-xp_prev,2)
                                  +pow(y_prev-yp_prev,2));

            vx += ((h/2)*fact*plan.mass/msun)*((x-xp_cur)/pow(rp_cur,3)
                      +(x_prev-xp_prev)/pow(rp_prev,3));
            vy += ((h/2)*fact*plan.mass/msun)*((y-yp_cur)/pow(rp_cur,3)
                      +(y_prev-yp_prev)/pow(rp_prev,3));

         }
      }

      theplanets[it].velocitiesx_v.point[i] = vx;
      theplanets[it].velocitiesy_v.point[i] = vy;

   }
```

Then, the algorithm redefines the planets in the `planets` vector so that their position and velocity Verlet `thevecs` are filled appropriately.

- `Solve_RK4()`, which uses the RK4 algorithm to solve the multi-body problem. In particular, after finding the step size `h` and defining `fact` and `msun`, and setting the initial positions and velocities as in the `Solve_Verlet()` algorithm, this function loops over integers `i` less than `nsteps` and computes the positions and velocities in the x- and y-directions simultaneously:

```
for(unsigned int it=0;it<planets.size();it++){

   planet myplan = theplanets[it];

   thevec posx = myplan.positionsx_r;
   thevec posy = myplan.positionsy_r;
   thevec velx = myplan.velocitiesx_r;
   thevec vely = myplan.velocitiesy_r;

   double prev_r = sqrt(pow(posx[i-1]-originx,2)
                            +pow(posy[i-1]-originy,2));

   double k1vx = 0;
   double k1vy = 0;
   double k1x = h*velx[i-1];
   double k1y = h*vely[i-1];

   for(unsigned int n=0;n<planets.size();n++){

      planet plan = theplanets[n];

      if(myplan!=plan){

         double prev_rp = sqrt(pow(posx[i-1]-plan.positionsx_r[i-1],2)
               +pow(posy[i-1]-plan.positionsy_r[i-1],2));
         k1vx+=h*fact*plan.mass
                *(posx[i-1]-plan.positionsx_r[i-1])/pow(prev_rp,3)/msun;
         k1vy+=h*fact*plan.mass
                *(posy[i-1]-plan.positionsy_r[i-1])/pow(prev_rp,3)/msun;

      }
   }

   double k2vx = 0;
   double k2vy = 0;
   double k2x = h*(velx[i-1]+k1vx/2);
   double k2y = h*(vely[i-1]+k1vy/2);

   for(unsigned int n=0;n<planets.size();n++){
```

```
   planet plan = theplanets[n];

   if(myplan!=plan){

      double prev_rp = sqrt(pow(posx[i-1]-plan.positionsx_r[i-1],2)
           +pow(posy[i-1]-plan.positionsy_r[i-1],2));
      k2vx+=h*fact*plan.mass
           *(posx[i-1]+k1x/2-plan.positionsx_r[i-1])/pow(prev_rp,3)/msun;
      k2vy+=h*fact*plan.mass
           *(posy[i-1]+k1y/2-plan.positionsy_r[i-1])/pow(prev_rp,3)/msun;

   }
}

double k3vx = 0;
double k3vy = 0;
double k3x = h*(velx[i-1]+k2vx/2);
double k3y = h*(vely[i-1]+k2vy/2);

for(unsigned int n=0;n<planets.size();n++){

   planet plan = theplanets[n];

   if(myplan!=plan){

      double prev_rp = sqrt(pow(posx[i-1]-plan.positionsx_r[i-1],2)
           +pow(posy[i-1]-plan.positionsy_r[i-1],2));
      k3vx+=h*fact*plan.mass
           *(posx[i-1]+k2x/2-plan.positionsx_r[i-1])/pow(prev_rp,3)/msun;
      k3vy+=h*fact*plan.mass
           *(posy[i-1]+k2y/2-plan.positionsy_r[i-1])/pow(prev_rp,3)/msun;

   }
}

double k4vx = 0;
double k4vy = 0;
double k4x = h*(velx[i-1]+k3vx);
double k4y = h*(vely[i-1]+k3vy);

for(unsigned int n=0;n<planets.size();n++){

   planet plan = theplanets[n];
```

```
            if(myplan!=plan){

               double prev_rp = sqrt(pow(posx[i-1]-plan.positionsx_r[i-1],2)
                    +pow(posy[i-1]-plan.positionsy_r[i-1],2));
               k4vx+=h*fact*plan.mass
                     *(posx[i-1]+k3x-plan.positionsx_r[i-1])/pow(prev_rp,3)/msun;
               k4vy+=h*fact*plan.mass
                     *(posy[i-1]+k3y-plan.positionsy_r[i-1])/pow(prev_rp,3)/msun;

            }
         }

         theplanets[it].positionsx_r[i] = posx[i-1]
                        +(1.0/6)*(k1x+2.0*k2x+2.0*k3x+k4x);
         theplanets[it].positionsy_r[i] = posy[i-1]
                        +(1.0/6)*(k1y+2.0*k2y+2.0*k3y+k4y);
         theplanets[it].velocitiesx_r[i] =velx[i-1]
                        +(1.0/6)*(k1vx+2.0*k2vx+2.0*k3vx+k4vx);
         theplanets[it].velocitiesy_r[i] =vely[i-1]
                        +(1.0/6)*(k1vy+2.0*k2vy+2.0*k3vy+k4vy);

      }
```

Then, the algorithm redefines the planets in the `planets` vector so that their position and velocity RK4 `thevec`s are filled appropriately.

- `Draw_Verlet()`, which makes paper-quality plots of the RK4 positions and velocities.
- `Draw_RK4()`, which makes paper-quality plots of the RK4 positions and velocities.
- `Set_COM()`, which sets the origin of the solar system to the center of mass of the planets.
- `Set_O()`, which sets the origin for the solar system to the origin (ie. the position of the sun).

These classes are used together in `project3.C`, which defines individual planets according to their masses, average distances from the sun, and average velocities [6], adds them to an object of type `solar_system`, and calls the appropriate `solar_system` methods to solve the multi-body problem. This code also does some benchmark checking and other smaller calculations.

4 Results and Benchmarks

The first test of the algorithms was performed using the ideal earth-sun two-body problem. It is solved in `project3.C` in function `partb()`. Assuming an ideal, circular orbit, we would have

$$\frac{GM_\odot m_E}{r^2} = m_E a$$
$$= \frac{mv^2}{r}$$

This implies that

$$v = \sqrt{\frac{GM_\odot}{r}}, \tag{4.1}$$

So we take this to be the initial velocity of the earth around the stationary sun centered at the origin. We began by looking at the orbit of the earth over one year with 100 steps. The results can be found in Fig. 1. We solved this system using both the RK4 and Verlet algorithms, as shown.

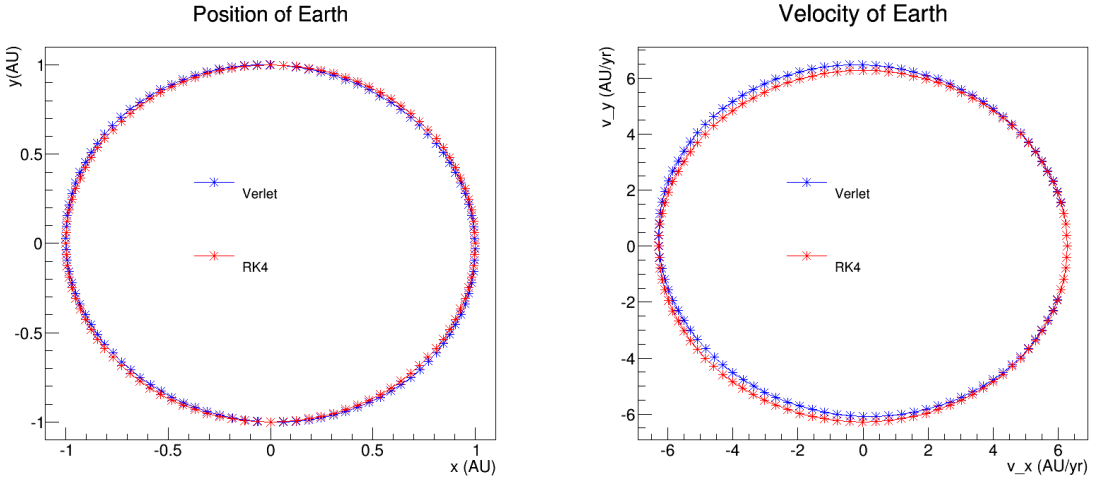

Figure 1: The position and velocity over 1 year with 100 steps as calculated byt he RK4 and Verlet algorithms.

We also tested the stability over long amounts of time. We saw that the algorithm was quite stable in position for as many years as was checked (up to 3000), but somewhere around 1000 years, we found that the velocity Verlet algorithm started to become unstable. This can be seen clearly in Fig. 2, which shows the velocity and positions as calculated by the two algorithms over 3000 years with step size 1,000,000 (ie. 333 steps per year). We believe, however, that this instability at higher times may indicate that the Verlet method requires more steps per year at higher years. For example, we tested also 300 years with 10,000 steps (ie. 33 steps per year) and saw that the instability was more pronounced in this case (see Fig. 3).

Another test performed was designed to check the stability of both arguments for various step sizes or number of steps. For this test, we looked at the performance of both the Verlet and RK4 algorithms over the course of one year for various step sizes between 2 and 50

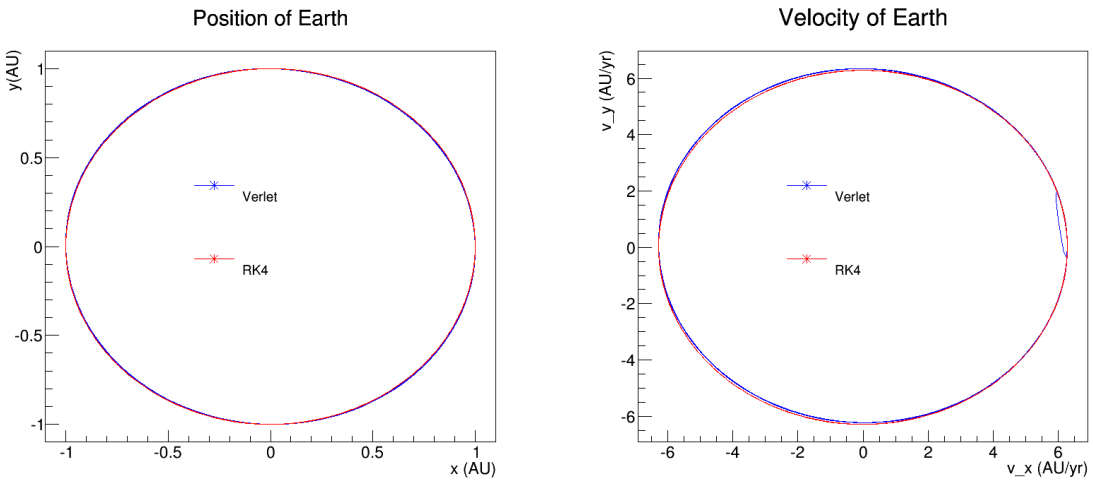

Figure 2: The position and velocity over 3000 years with 1,000,000 steps. A slight instability is observed in the Verlet velocity solutions.

(noting from Fig. 1 that both the Verlet and RK4 algorithms are stable over one year for step size 100. In Fig. 4, we see that, for small step size, the position Verlet algorithm is very unstable. The orbits are not actually even closed over the course of one year. However, at 5 steps, the orbit closes (although is not particularly nice), and the orbit finally stabilizes around 10 steps. The velocity Verlet algorithm forms a closed curve even at 2 steps, but does not become fully stable until between 20 and 50 steps. This indicates that the velocity Verlet algorithm may require more steps in order to be accurate than the position algorithm (which further validates our hypothesis that more steps may be required for longer times).

The RK4 algorithm has a slightly different step number dependence. The position RK4 algorithm is not as wildly non-elliptical as the Verlet algorithm even at very small numbers of steps, but doesn't fully stabilize until about 10 steps. A very similar behavior is observed for the velocity RK4 algorithm. This similarity (compared with the distinctions between the Verlet algorithm velocity and position results) is likely due to the fact that the velocity and position RK4 algorithms are identical (excepting the function being used to evaluate), whereas the velocity and position Verlet algorithms are slightly different and cannot be evaluated simultaneously.

A final test which was performed was designed to check that each algorithm conserved potential, kinetic, and total energy and angular momentum. Because we assume a circular orbit, we should have a constant radius, which implies that the potential and kinetic energies should be individually conserved (the potential depends only on the radius of the planet from the sun and thus, because total energy must always be conserved, kinetic energy must, in this case, be independently conserved). Angular momentum should also be conserved because there is no linear motion in the system, and total momentum is always conserved.

We checked these expected conserved quantities again over the course of one year for 100 steps. We found that generally the RK4 algorithm had better results. In particular, as shown in Table 1, the RK4 algorithm is off only by a fraction of a percent whereas the Verlet algorithm is off by about 10% in every quantity. The variation in these quantities

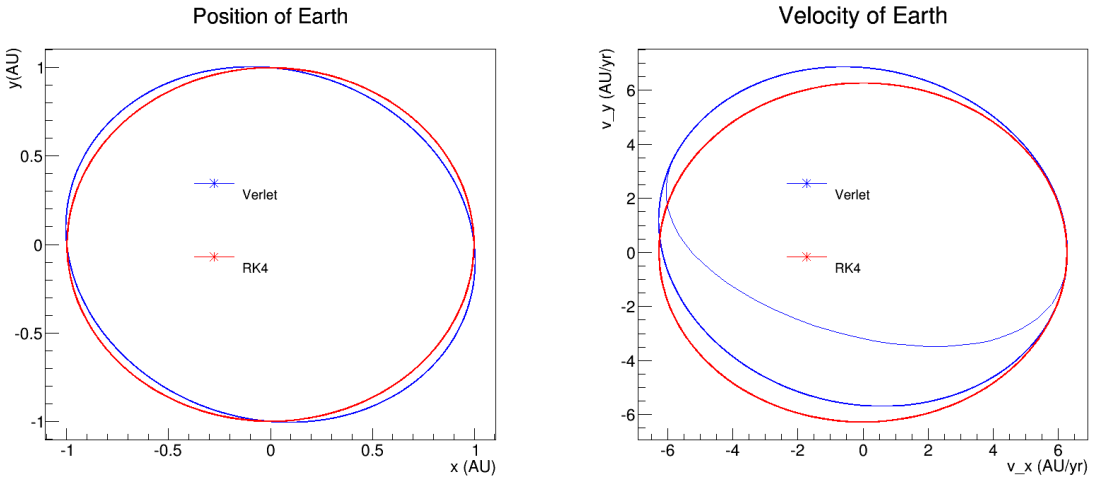

Figure 3: The position and velocity over 300 years with 10000 steps. We see that the instability with fewer steps per year is much more pronounced than in Fig. 2.

is shown in Fig. 5 However, as the time and number of steps increased (particularly as the number of steps increased), it was found that the Verlet algorithm became more accurate. For example, at 300 years and 100000 steps, the largest percent difference of the conserved quantities was under 5%.

	Kinetic Energy	Potential Energy	Total Energy	Angular Momentum
Verlet	15.068%	3.092%	7.104%	3.100%
RK4	8.456×10^{-6}	4.228×10^{-6}	2.514×10^{-13}	8.456×10^{-6}

Table 1: The percent difference between the maximum and minimum of the expected conserved quantities for the Verlet and RK4 algorithms in the case of the idealized earth-sun model.

We next tried to use our algorithms to calculate a convenient quantity – the escape velocity – of a planet at a distance of 1 AU from the sun (eg. earth). To do this, we started with our ideal earth-sun system with a circular orbit. We then increased our velocity by a small amount until the planet was not at the same distance from the sun after a year of revolution (see `partd()` and `partd_precise()` in `project3.C`). We found that the escape velocity was 100.531 AU/yr, and that it was independent of the mass of the planet. This is because the escape velocity occurs where the potential energy is equal to the kinetic energy. That is,

$$\frac{GM_\odot m_p}{r} = \frac{1}{2}m_p v_e^2$$

or

$$v_e = \sqrt{\frac{2GM_\odot}{r}}. \tag{4.2}$$

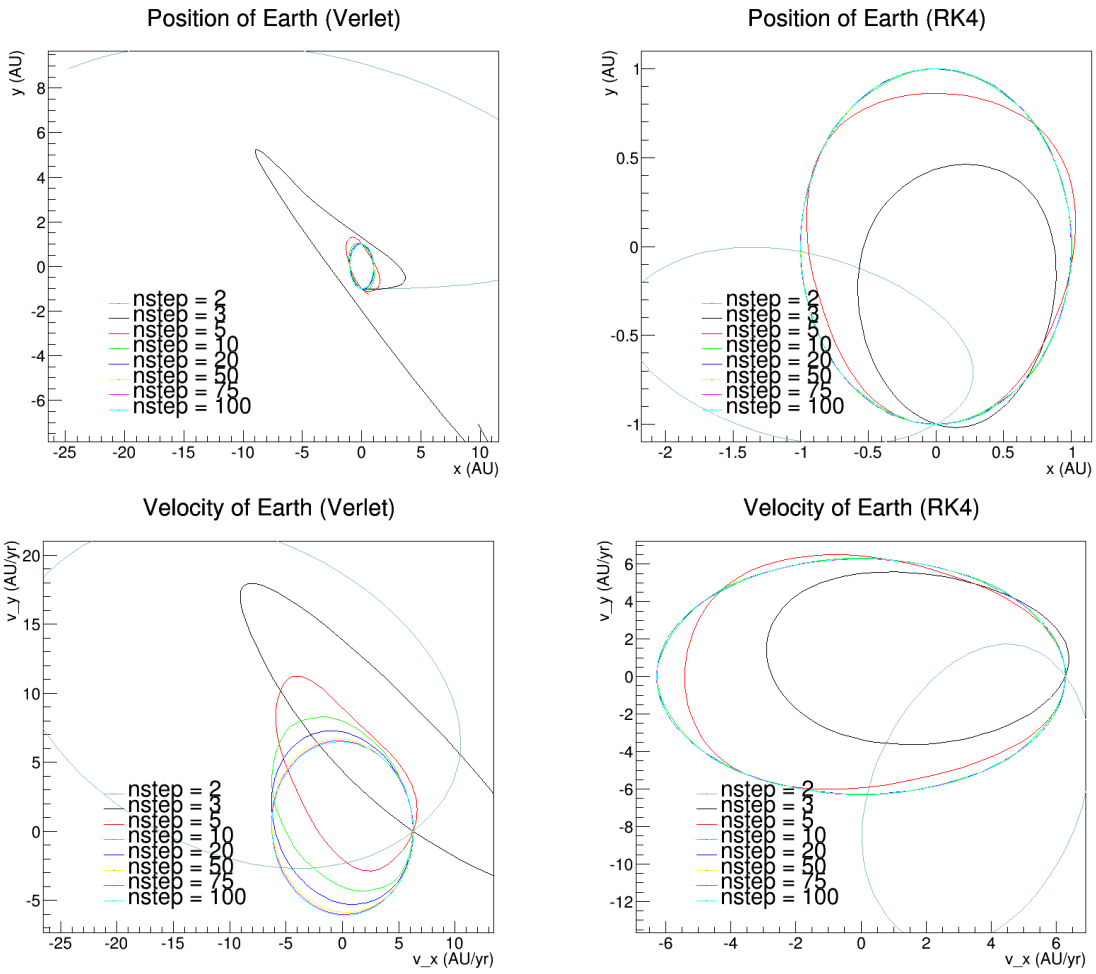

Figure 4: Checking the stability of the RK4 and Verlet algorithms with varying step sizes over the course of one year.

From Eq. 4.2, it is clear that the escape velocity should not be dependent on the mass of the planet (or other object) in orbit around the sun, but just on the mass of the sun and the distance from the sun. In this case, then, we have

$$v_e = \sqrt{\frac{8\pi^2}{1}} = 8.89\frac{km}{AU}$$

Clearly, then, our calculation of the final escape velocity was incorrectly done within the code. This is something which it would be nice to investigate further.

Once we were satisfied that the basic algorithms were working, we decided to add a second planet – Jupiter – to the system. Assuming the orbits of the earth and Jupiter to be coplanar (in the x-y direction), we determined the orbits and velocities of Earth and Jupiter assuming the sun to be at the origin in `parteandf()` of `project3.C`. The results for 500 steps over 10 years are shown in Fig. 6. We can see that all of the results are quite reasonable – elliptical/circular orbits around a stationary sun at the origin. With a longer time as well

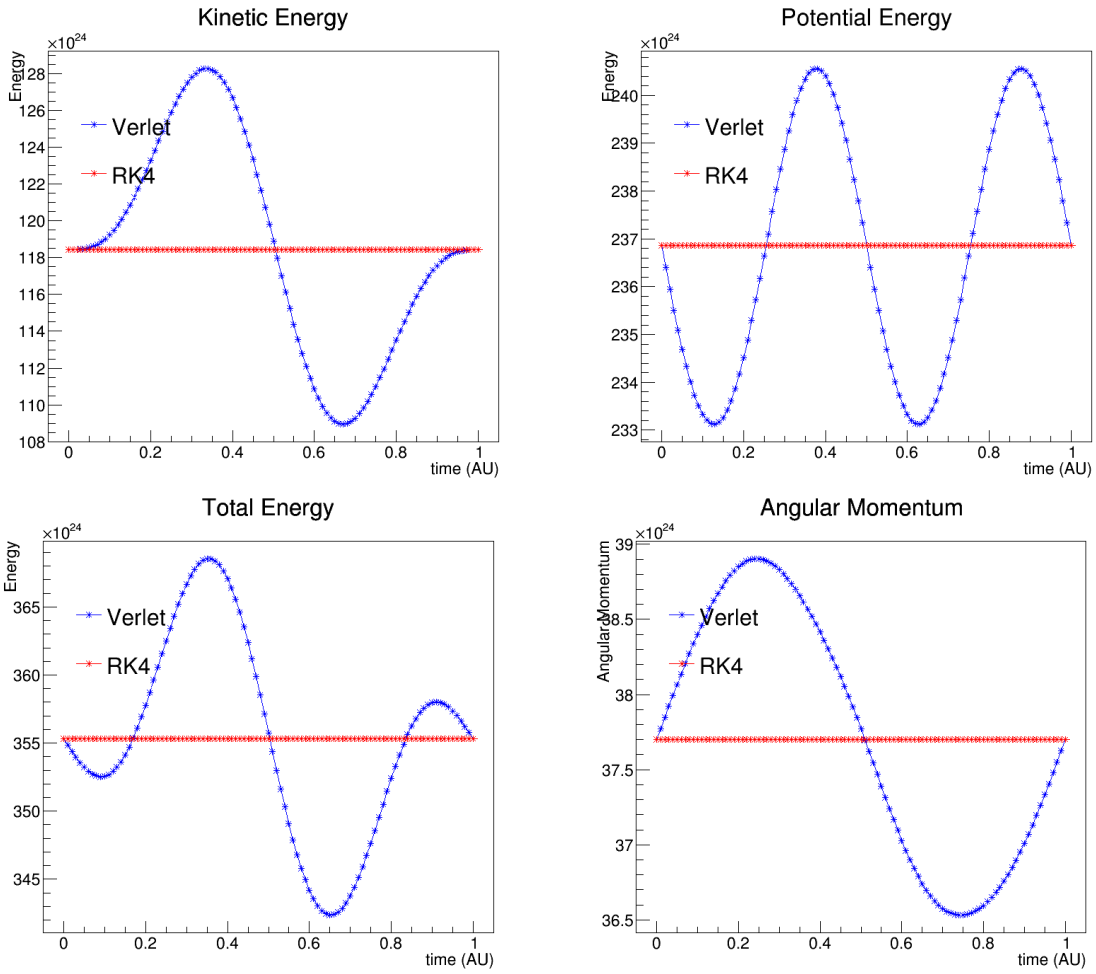

Figure 5: Checking that various expected conserved quantities are actually conserved over the course of one year with 100 steps for the RK4 and Verlet algorithms.

we see that the algorithms are fairly stable. The plots in Fig. 7 are made with 100000 steps over 100 years. We see that the velocity Verlet algorithm is quite stable in spite of some oscillatory motion from the position algorithm. The same holds true for the RK4 algorithm.

We next investigated the stability of the algorithms for various step sizes. We looked specifically at 50 years with numbers of steps between 500 and 1000. We found that, for 500 steps, the position and velocity of Earth using RK4 was highly unstable while those for Verlet were more stable (see Fig. 8). By 1000 steps, however, the orbits seem quite stable for both methods (although there did seem to be some instability still in the velocity Verlet algorithm). Instability in the RK4 algorithm persisted until around 700 steps.

We next looked at what happened when we made Jupiter heavier. We started by making Jupiter 10 times its actual mass. Starting with 750 steps (Fig. 9), we slowly increased the number of steps until we had achieved a stable orbit. With this approach, we found that we tried as many as 10000 steps between 0 and 50 years, and never achieved a stable orbit. It was impractical to attempt more steps due to computing time, and so we assume that no

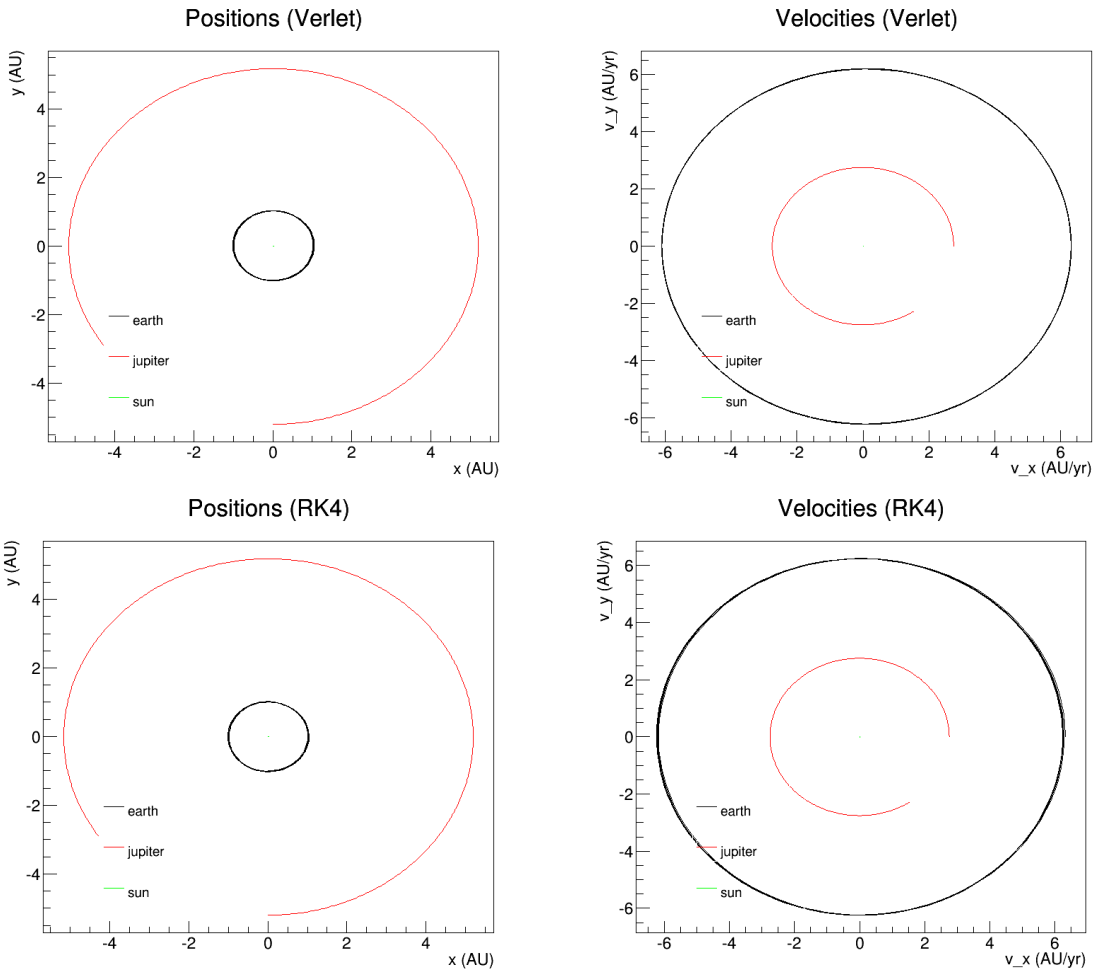

Figure 6: Adding Jupiter to the Earth-sun system and computing the positions and velocities using the RK4 and Verlet algorithms for 500 steps over 10 years.

such stable orbit exists. We can make a similar statement about the orbit for the planets if Jupiter is 1000 times its actual mass, shown in Fig. 10, which we checked with 2000 steps over 50 years.

We next investigated the movement of the planets about the center of mass of the system. To do this, we computed the center of mass and set this as the origin. We wanted the center of mass to be a stationary point, so we needed to adjust the velocity of the sun. We know that the velocity v_{com} of the center of mass is given by

$$v_{com} = \frac{\sum_p m_p v_p}{\sum_p m_p}$$

This indicates that the velocity of the sun should be

$$v_{sun} = -\frac{\sum_{p \neq sun} m_p v_p}{m_{sun}}. \tag{4.3}$$

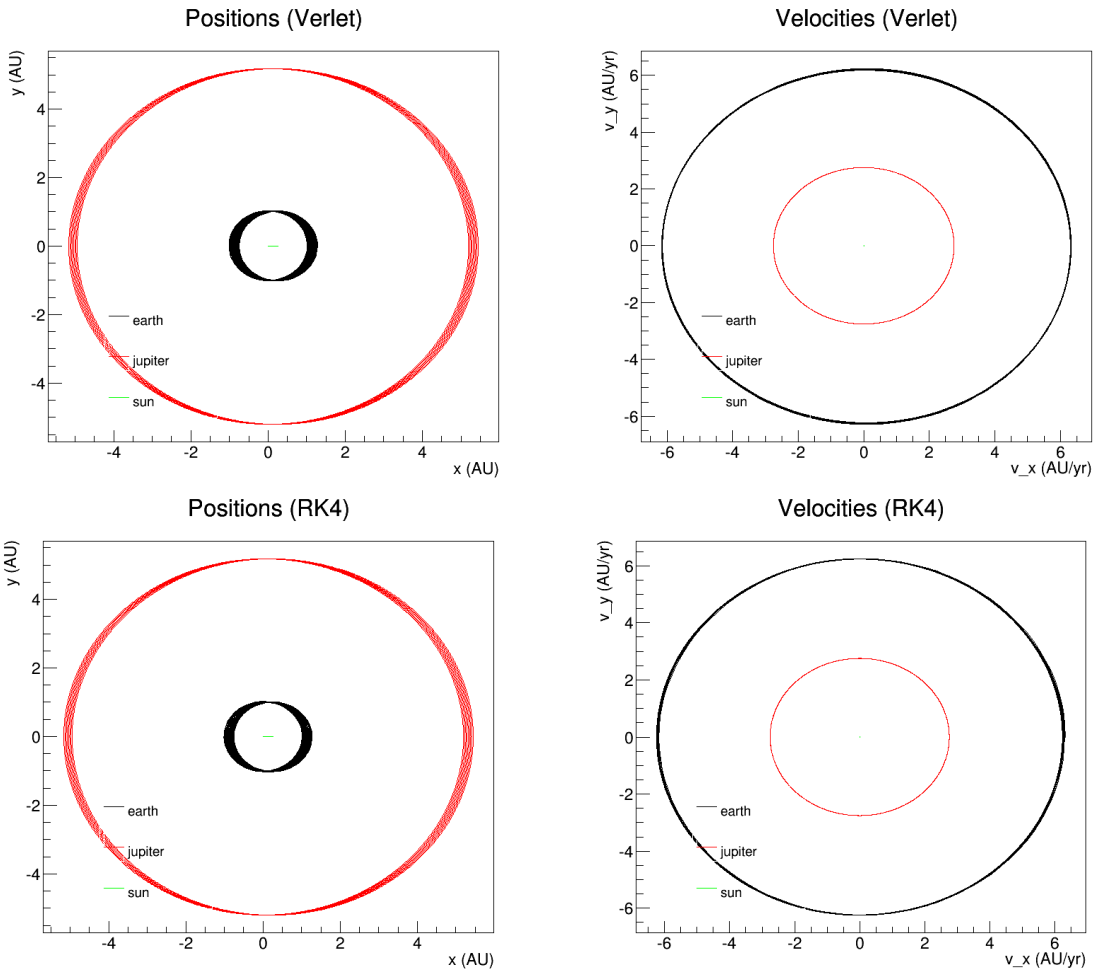

Figure 7: Adding Jupiter to the Earth-sun system and computing the positions and velocities using the RK4 and Verlet algorithms for 100000 steps over 100 years to check the stability over time.

So we set the velocity of the sun to be as in Eq. 4.3 and looked at the movement of the planets for 1000 steps over 10 years as shown in Fig. 11. We see that in fact the sun moves very little – an almost negligible amount – with these restrictions.

Finally, we looked at solving the full solar system problem. Here we investigated the motion over 100 years with 10000 steps, first with the sun as the origin, as shown in Fig. 12, and then with the center of mass at the origin and the sun's velocity set as in Eq. 4.3 **plots**. In both cases, the initial velocities for each planet were assumed to be the average velocity of the planet around the sun. These numbers and the masses and average distances from the sun were taken from [6].

We can see that, in Fig. 12, the orbits seem stable and elliptical, as expected. The only small instability comes from the RK4 algorithm, where we see that the velocity of mercury has some fluctuations as time progresses (as evidenced by the thicker velocity ring). This is somewhat surprising as some sources [5] suggest that only a step size of 0.5 days should be

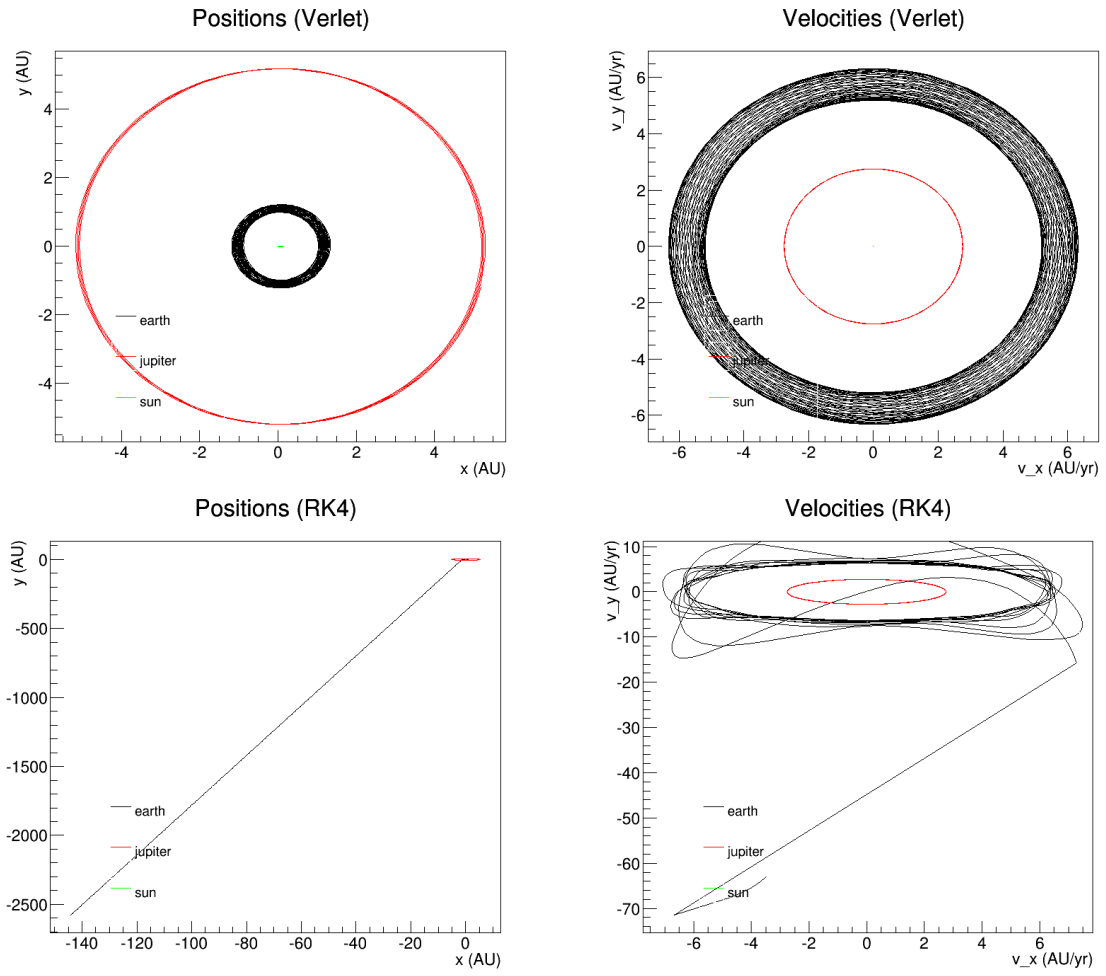

Figure 8: Adding Jupiter to the Earth-sun system and computing the positions and velocities using the RK4 and Verlet algorithms for 500 steps over 50 years to check the stability over time.

needed to observe stability, and our step size was ~ 0.2 days. However, it is not clear which integrating technique this source used and, as our Verlet algorithm does indicate stability, it is fairly safe to say that this instability observed may be a step size problem or a small bug in the code.

In **plots**,

5 Conclusions

Overall, this project was fairly successful. We were able to create a fully object-oriented C++ code which could work with planets and solar systems and calculate the positions and velocities of those planets in orbit around the sun. We were able to find relative stability in the orbits of the planets, as expected, in both the cases of the sun stationary at the center of the system and with the center of mass of the system as the origin and the sun moving

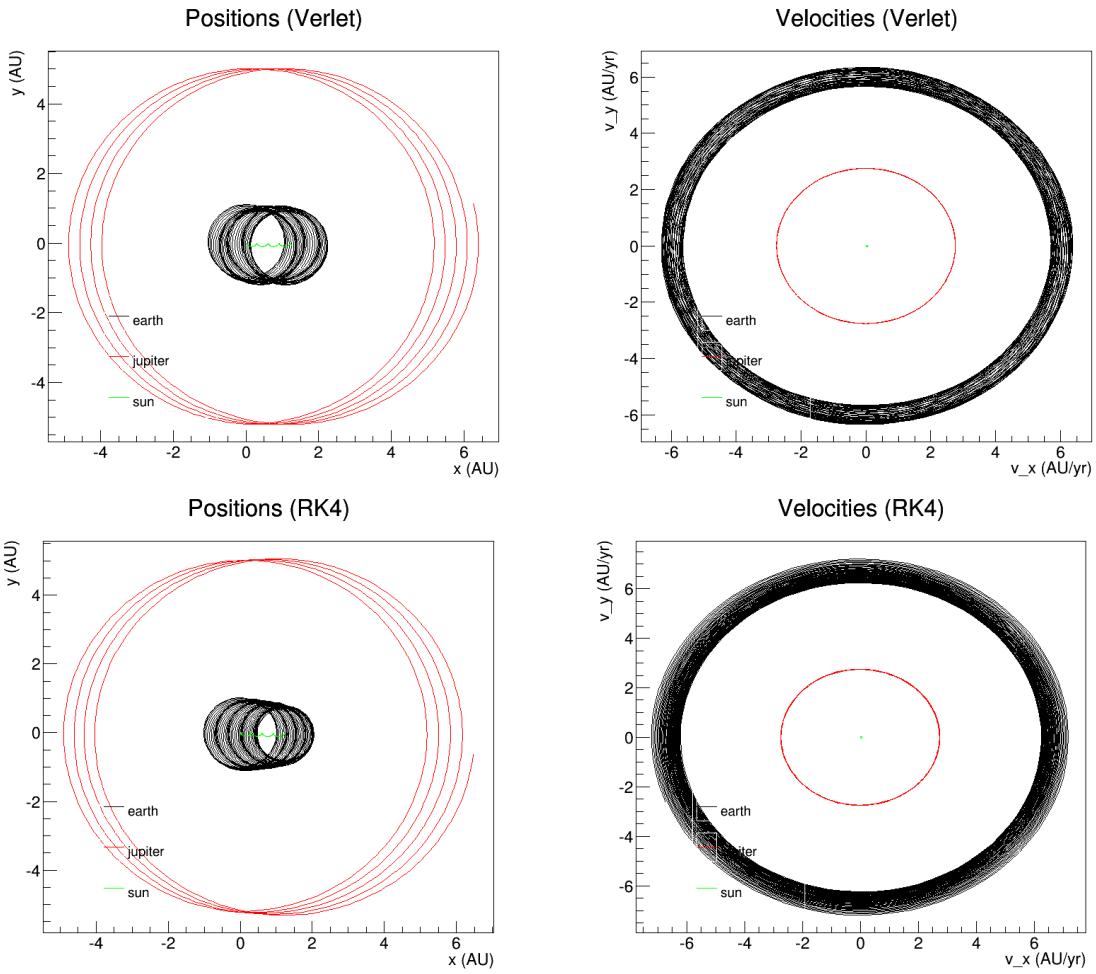

Figure 9: Investigating the effect of making Jupiter 10 times more massive than it actually is on the orbit of earth around the sun in the three-body system.

in such a way (Eq. 4.3) as to keep the center of mass stationary. However, there are several further tests which may be performed on the algorithms and which may shed more light on the apparent fluctuations in Mercury's velocity as it orbits around the sun as part of the full solar system.

The first and probably most enlightening study we might perform is a further investigation into the dependence of the stability of the Verlet and RK4 algorithms on step size and length of time. It was observed throughout the course of the project that both algorithms, but particularly the RK4 algorithm, required a very small step size in order to accurately compute the positions and velocities of the planets. It would be interesting to do a study on what the minimum number of steps would be for integration for larger systems specifically.

Along these lines, it would also be useful to work on optimizing the code. One of the main limitations in what was studied in this project was computation time. In particular, the RK4 algorithm took significantly longer to run than the Verlet algorithm. This makes sense because, for each step, the RK4 algorithm calculates four floating point quantites

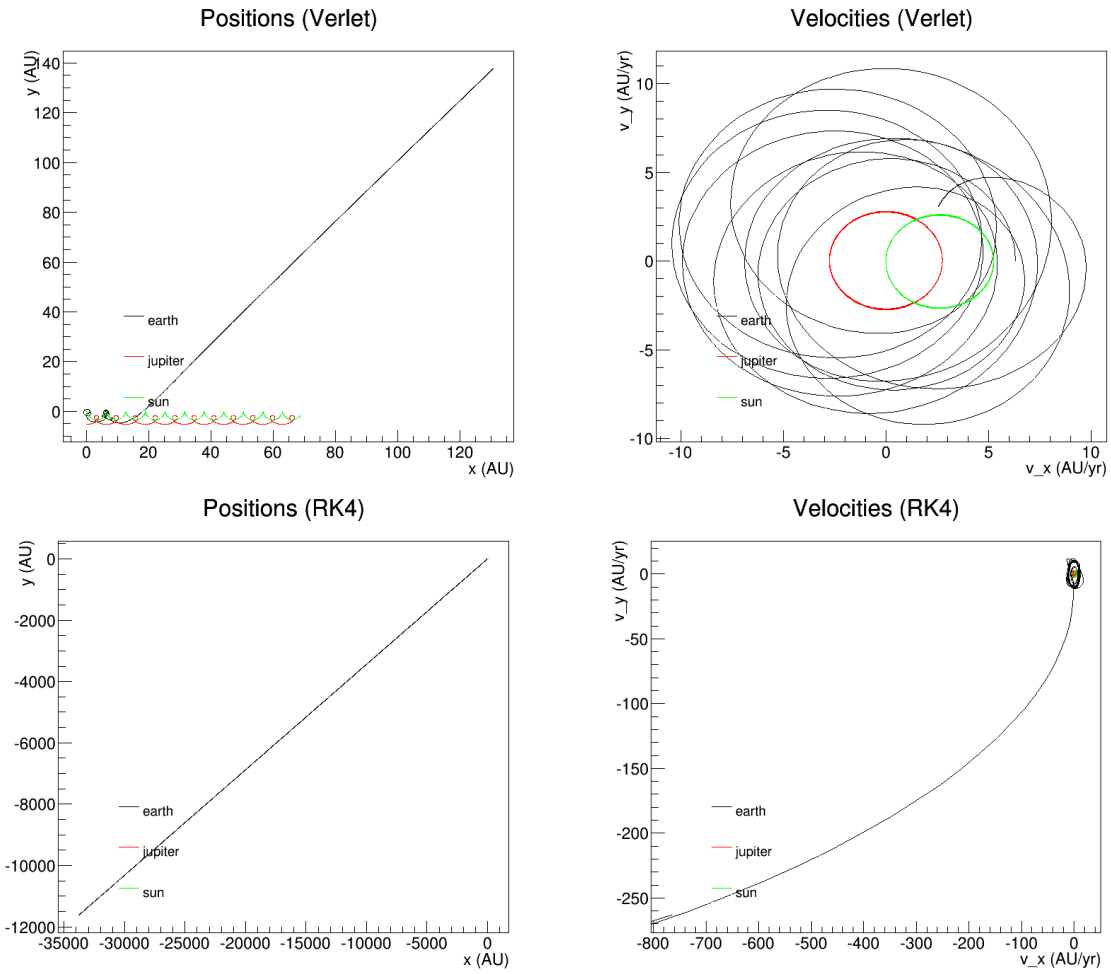

Figure 10: Investigating the effect of making Jupiter 1000 times more massive than it actually is on the orbit of earth around the sun in the three-body system.

which must be further combined into the solution. This gave us only 2 flops per quantity (ie. 8 flops to calculate the k's) for the positions. However, this number was quite a bit larger for the velocities because each planet required 6 flops per k based on the function used in the integration. With 11 planets, we had 66 flops per k. This means that. due to computational time, we were unable to do any more than what was discussed in Section 4.The Verlet algorithm is not as flop-heavy, but still requires ~ 9 flops per planet for the position calculations and about 11 for the velocity calculations. This requires a very high computation time as well.

We wiykd aksi kuje to spend more time working on the calculation of the escape velocity. As discussed in Section 4, our calculation of this quantity was inconsistent with expected values. It would be very useful to have more time to devote to this problem, as it provides a very convenient check that the algorithms are working properly.

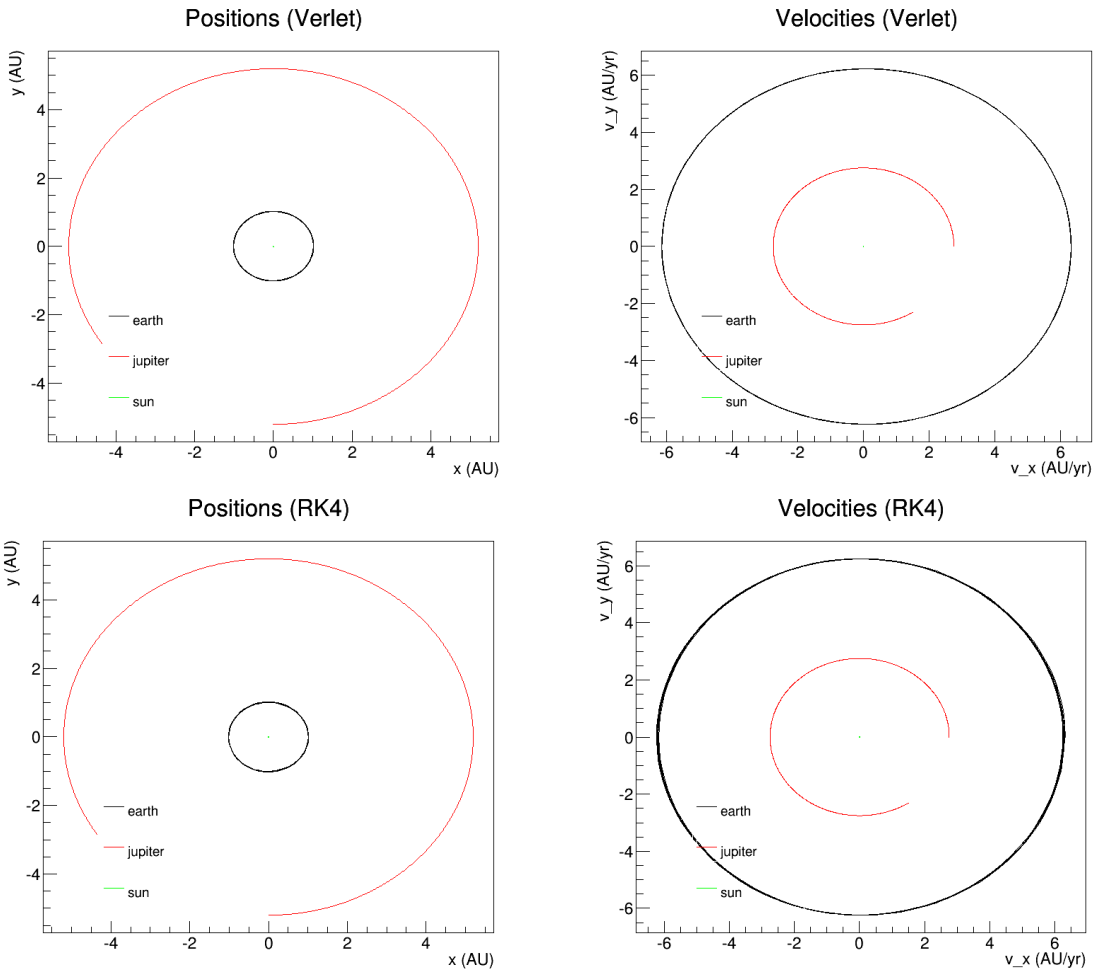

Figure 11: The three-body problem solved over 10 years with 1000 steps with the origin at the center of mass of the system and the initial velocity of the sun set as in Eq. 4.3.

6 Bibliography

1. Brun, Rene, and Fons Rademakers. "ROOT - An Object Oriented Data Analysis Framework." *Nuclear Instruments and Methods in Physics Research A* 389 (1997): 81-86. Web.

2. Drueke, Elizabeth A. *Programming a Linear Algebra Solution to the Poisson Equation.* Rep. N.p.: n.p., n.d. Print.

3. Drueke, Elizabeth A. *Programming a Linear Algebra Solution to the Problem of Two Electrons Trapped in a Spherical Harmonic Oscillator Well.* Rep. N.p.: n.p., n.d. Print.

4. Hjorth-Jensen, Morten. *Computational Physics.* N.p.: Department of Physics, U of Oslo, n.d. Print.

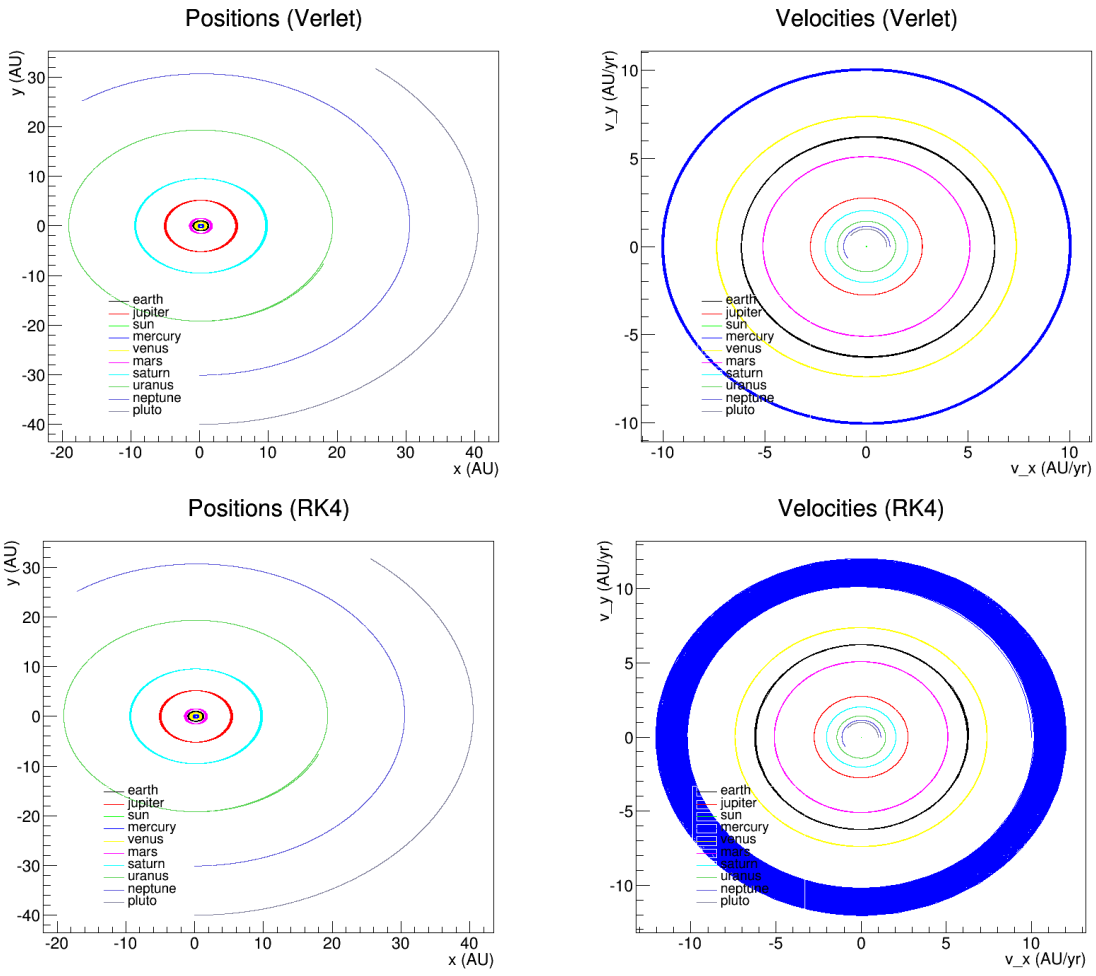

Figure 12: The solution to the solar system over 100 years with 10000 steps with the sun at the origin.

5. Laskar, Jacques. "Stability of the Solar System." *Scholarpedia.* Scholarpedia, 31 Oct. 2014. Web. 12 Apr. 2016. <http://www.scholarpedia.org/article/Stability_of_the_solar_system>.

6. "Planetary Fact Sheet - Metric." *Planetary Fact Sheet.* NASA, n.d. Web. 12 Apr. 2016. <http://nssdc.gsfc.nasa.gov/planetary/factsheet/>.