Project 5

George Cowie and Jonas Thoen Faber

December 19, 2019

Abstract

In this project we have created a model that can predict planetary orbits using Newton's law of universal gravitation. The model utilizes Verlet integration. We found Verlet integration to be very well suited to this problem as it conserves energy, and has no significant difference in CPU time compared with Forward Euler integration. The model was able to predict physical phenomena, for example the perihelion values of the planets and escape velocity of Earth. We tested what would happen if we increased Jupiter's mass by a factor of 1000, this resulted in a highly chaotic system without stable orbits. With a correction term to the Newtonian force the model came close to predicting the observed perturbation of Mercury due to relativistic effects. The observed value is 43" per century while the model found 40.80".

https://github.com/JonasTFab/Project_5.2

1 Introduction

In this project we will look into different numerical solutions to modeling the solar system, namely the Foward Euler and Velocity Verlet integration schemes. With these integration schemes we will test different properties of the Solar System. We will look into questions like what happens to Earth's orbit if Jupiter's mass is changed, is Newton's law of motion due to gravitational pull enough to describe the orbits of the solar system and other problems. We will also look into the advantages and disadvantages of the two integration schemes, and look into stability of the solvers. As mentioned Newton's law of gravitation will be the formula mainly used in this project.

Many planetary systems, including the solar system contain multiple bodies orbiting around a center of mass [NASA [3]]. In creating a solar system solver our goal is to create a program which can add multiple planets to the system in a compact and easy way. This is where object oriented programming comes in as a useful tool and our project will be mainly based on developing a class that will solve these problems effectively. We could in theory do this in just two dimensions as the planets are close to co-planar but to make the simulation more realistic, we have used three dimensions. We will start off simple by assuming the Sun is stationary

and Earth is the only planet orbiting the Sun. Then gradually expand our program to handle more complex problems where more planets are included and let the Sun be in motion.

2 Method

2.1 Discretizing the Earth-Sun system

The first step in our solar system model will be to create a simple system of Earth orbiting the Sun. For this simple system we will develop the mathematics and algorithms needed to model the entire solar system. Assuming that the Earth's orbit around the Sun is almost circular we can write the force as

$$F_G = \frac{M_{Earth}v^2}{r} = \frac{GM_{\odot}M_{Earth}}{r^2} \tag{1}$$

Assuming that the orbit of the Earth and the Sun are co-planar in the xy-plane, using Newtons second law we have the equations

$$\frac{d^2x}{dt^2} = \frac{F_{G,x}}{M_{Earth}} \tag{2}$$

This can be rewritten as coupled differential equations

$$\frac{dx}{dt} = v(x,t)$$

$$\frac{dv}{dt} = \frac{F_G(x,t)}{M_{Earth}} = a(x,t)$$

with an initial value of x we can solve this equation iteratively by increasing x. This is known as Euler's method.

$$x_{i+1} = x(t_{i+1}) = x_i + v(t_i)dt$$

$$v_{i+1} = v(t_{i+1}) = v_i + a(t_i)dt$$

Eulers method is very simple but it has some limitations. Energy is not conserved using this method, meaning that it will not be a good choice for simulating the solar system. The order of the system will likely decay over large time-spans. We will therefore use a method that preserves energy, namely the velocity Verlet method. The derivation of this method can be found at [Hjorth Jensen [2]]

$$x_{i+1} = x_i + hv_i + \frac{h^2}{2}v_i^{(1)} + O(h^3) \tag{3}$$

$$v_{i+1} = v_i + \frac{h}{2}\left(v_{i+1}^{(1)} + v_i^{(1)}\right) + O(h^3) \tag{4}$$

For a circular orbit we have an equation that can be used to find the initial velocity that gives a circular orbit.

$$v^2 r = GM_\odot$$

$$v = \sqrt{\frac{GM_\odot}{r}}$$

For a single particle in circular motion we have the angular momentum:

$$\vec{L} = \vec{r} \times \vec{p} \tag{5}$$

where $\vec{L}$ is the angular momentum, $\vec{r}$ is the position vector and $\vec{p}$ is the momentum vector of the planet of interest.

2.2 Object orient the solar system

Since there are multiple planetary bodies in our solar system, it would be practical to add planets to our simulation in a easy and effective way. This is where object orient programming comes handy. We start with creating a class in the code called `object`. Declaring an object using this class requires 8 parameters as inputs which is the initial position and initial velocity in three dimensions, mass of the planet and the number of integration points like so

```
object planet(x0, y0, z0, vx0, vy0, vz0, mass, iterations);
```

and then obtain information about future motion of this particular planet. `object` will first of all create six armadillo arrays with space for the desired number of iterations. These vectors correspond to the position and velocities in three dimensions. The first element of each array is set to be equal to the input parameters. By stating that the Sun is fixed at the origin simplifies the code as we do not need to explicitly calculate the distance between the Sun and the planets, meaning less FLOPS. The distance is just the coordinates of the planet of interest from the origin

$$r = \sqrt{x^2 + y^2 + z^2} \tag{6}$$

where x, y and z are the three dimensional coordinates to that object. Now that an object has been declared as `planet`, we should tell the object what we want to do with it then it will be done automatically. We have included the following functionalities:

```
planet.velocity_verlet();
planet.euler();
planet.kinetic_energy();
planet.potential_energy();
planet.write_to_file(std::string filename);
```

So the two first functions solves the differential equation by the use of forward Euler or velocity Verlet integration schemes. Next both the kinetic and the potential energy is found based on the position and velocity. Since there are no external forces to slow down the motion of the planets, the total energy should be conserved. We will test if this is true for both algorithms. The last function `write_to_function` is called if we want to write the results to a separate data file. The data file is then used for plotting by using the programming language Python.

From subsection 2.1 we have found how each of the algorithms can be written. For the sake of not operating with large numbers that may cause loss in numerical precision, the units used in this project are year, astronomical unit and solar mass as time, distance and mass respectively. A simple two dimensional example of the Forward-Euler looks something like this

```
for (int i=1; i<N; i++){
  r = sqrt(x(i-1)*x(i-1) + y(i-1)*y(i-1));
  a = GM / (r*r*r);
  ax = -a*x(i-1);
  ay = -a*y(i-1);
  vx(i) = vx(i-1) + ax*dt;
  vy(i) = vy(i-1) + ay*dt;
  x(i) = x(i-1) + vx(i-1)*dt;
  y(i) = y(i-1) + vy(i-1)*dt;
}
```

This can easily be extended to three dimension by just adding a `z` parameter in the code. Here `GM` is the gravitational constant times the mass of the Sun. The acceleration is subtracted as it must always point towards the center of the system. `a` is divided by r^3 because we need to normalize the acceleration in all dimensions ($\vec{r}/|\vec{r}|^3$). Next we have the Euler-Verlet method which is slightly more complex than Forward-Euler. It goes like this

```
for (int i=1; i<N; i++){
  x(i) = x(i-1) + dt*vx(i-1) + 0.5*dt*dt*ax_prev;
  y(i) = y(i-1) + dt*vy(i-1) + 0.5*dt*dt*ay_prev;
  r = sqrt(x(i)*x(i) + y(i)*y(i));
  a = GM / (r*r*r);
  ax_new = -a * x(i);
  ay_new = -a * y(i);
  vx(i) = vx(i-1) + 0.5*dt*(ax_new + ax_prev);
  vy(i) = vy(i-1) + 0.5*dt*(ay_new + ay_prev);
  ax_prev = ax_new;
  ay_prev = ay_new;
```

This algorithm is also easily extended to three dimensions. In this algorithm we need to store the acceleration used to find the new position values. The next acceleration is found before the velocity is calculated using both new and old acceleration. This is the advantage of the velocity Verlet method, it is not very complicated but it is numerically quite stable. The data is stored in armadillo arrays after one of the algorithms has been run which makes it fairly easy to calculate the potential and kinetic energy. The potential energy in a gravitational field is defined as follows

$$U = -\frac{GM_{\odot}m}{r} \tag{7}$$

and the kinetic energy

$$K = \frac{1}{2}mv^2 \tag{8}$$

The energies are also declared as armadillo arrays with the same length as the position and velocity arrays are. It is then calculated in two separate functions when called to within the class `object` as so

```
void potential_energy() {
    for (int i; i<N; i++){
        pot_en(i) = -GM*mass/sqrt(x(i)*x(i)+y(i)*y(i)); }}

void kinetic_energy() {
    for (int i; i<N; i++){
        kin_en(i) = 0.5*mass*(vx(i)*vx(i)+vy(i)*vy(i)); }}
```

2.3 Escape Velocity

We can find the escape velocity of a planet from the system by adding the kinetic energy and potential energy together and set it to be zero using Equation 7 and Equation 8. This is true since the Sun can no longer hold on to a planet if the kinetic energy is larger than the gravitational potential energy of the planet. Think of it as a potential well where the planet needs more energy than the potential well to surpass it.

$$\frac{1}{2}M_{Earth}v^2 = \frac{GM_{\odot}M_{Earth}}{r}$$

$$v = \sqrt{\frac{2GM_{\odot}}{r}} \tag{9}$$

We will use this analytical velocity by comparing it to the escape velocity we find numerically. We will also explore what happens to the escape velocity when we change the gravitational force.

2.4 The three-body problem

In reality the all bodies with mass in our solar system will interact through gravity with all other massive bodies. This means that the sun will not remain in a fixed position, but will orbit around the center of mass of the total system. If as an example take a look at the system of Earth, the sun and Jupiter we have a three body problem. The three body problem has no analytical solution as it is a chaotic system, but it can be approximated numerically. We will use the Velocity Verlet method but with additional forces, that is the forces between the Jupiter-Earth system. This new force is calculated exactly as between the Earth and sun. It goes like this:

$$F_{Earth-Jupiter} = \frac{GM_{Jupiter}M_{Earth}}{R^2_{Earth-Jupiter}} \tag{10}$$

where G is the gravitational constant, $M_{Jupiter}$ is the mass of the Jupiter, M_{Earth} is the mass of the Earth and $R_{Earth-Jupiter}$ is the distance between Earth and Jupiter.

As the class `object` is restricted to a two-body system, such as the Earth-sun system where also the sun is fixed at origin, we will create a new class named `solar_system`. This class is made for adding multiple planets or suns to the solar system and also some additional features. One can add a planet to the solar system by calling the class function `add_planet` followed by some initial values of that planet. First the solar system has to be set up

```
solar_system system;
```

followed by the planets of interest that should be added and the Sun fixed at origin

```
system.add_planet(Earth, "Earth");
system.add_planet(jupiter, "Jupiter");
system.sun_fixed();
```

Now both `Earth` and `jupiter` have already been declared and initialized using the class `object`. The string input is just to identify the indices. Every time one planet is added using this new class, the planets will be counted up, mass will be stored in a armadillo array, the position and velocity array will be resized and initial values will be implemented. The Sun is included at a fixed position throughout the simulation.

So first of all we want to calculate the forces between the planets based on the their coordinates and mass. This is done by making a single armadillo array containing the coordinates of all the planets. The array will be resized as mentioned when planets are added. By setting up and indexing correctly in the arrays, we extract values corresponding to the planets of interest. This is shown in a simple example

```
arma::Col <double> position = arma::vec(3*number_of_planets)
for (int i=0; i<number_of_planets; i++){
position(3*i)   = x(i);
position(3*i+1) = y(i);
position(3*i+2) = z(i);
}
```

The size of the vector is three times the number of planets. That is because each planet has a coordinate in x, y and z direction. We will do this also with an velocity array that updates the velocities of the planets as time passes by using velocity Verlet. The values of the position are no longer stored in a array but rather stored directly in a data file. The first scenario using this class will only calculate the force between Jupiter and Earth as they are currently the only planets in the system. The force from the Sun is also included but as a separate force as it is stationary and should not be affected by the other planets.

What might be interesting when studying the three-body problem is to increase the mass of Jupiter and observe how the solar system may be affected by this. Since the Sun is stationary at all time, the only thing that would be affected by Jupiter is Earth. We will increase the mass of Jupiter with a factor of 10, 100 and 1000 and then observe how these changes affect the system.

2.5 Solar system model

Until this point we have assumed the Sun to be fixed at origin. We will extend the class `solar_system` such that the Sun also will be in motion and be affected by the gravitational pull of other planets. This is a rather simple task as we just need to discard the forces related to the fixed Sun and give it the same properties as we already have for the planets. By this we mean that the Sun should be declared as an object just like we did with Jupiter and Earth. The general solution to the forces that are present in the solar system is then given as

$$F_{i,j} = \frac{Gm_i m_j}{r_{i,j}} \tag{11}$$

where $F_{i,j}$ is the force between planet i and j, m_i and m_j are the mass of planet i and j respectively and $r_{i,j}$ is the distance between planet i and j. A unit test is created making sure that the distance between two planets are not to small i.e. not dividing by zero in Equation 11. So by running two for loops over i's and j's and making sure that $i \neq j$, we may find the forces between every planet in the solar system just by indexing correctly including the Sun. To make sure that the center-of-mass stays fixed, the total momentum of the system must be zero. We therefore need some initial values for Jupiter and Earth. The initial positions and velocity of Jupiter and Earth and eventually other planets will be gathered from

[NASA [4]] at A.D. 2019-Dec-09 00:00:00 TDB. The momentum is given as $\vec{p} = m\vec{v}$ so the total momentum in each dimension must be equal to zero

$$\sum_{k=1} m_k v_{k,dim} = 0, \qquad k = 1, 2..., \text{total number of objects}$$

where *dim* goes through all the dimensions. The formula is simply flipped so that the momentum of the Sun is isolated on one side of the equal sign. Also, the mass of the Sun should be divided as we are interested in the velocity

$$v_{\odot,dim} = -\frac{1}{m_\odot} \sum_{k=1} m_k v_{k,dim}, \qquad k = 1, 2..., \text{objects except the Sun}$$

An important factor here is to make sure that the indexing is always correct when declaring the initial velocity of the Sun. We will make sure that the Sun is the last object in every arrays meaning the last index will refer to the Sun. This is done by calling the function within the class `solar_system` after all the other planets are added like so

```
system.sun_included();
```

This function is more or less equal to `add_planet` but it will initialize the velocity of the Sun such that the total momentum in the system is equal to zero. To make sure that this stays true, we will implement a unit test which calculate the total momentum of the system. The test is active at every hundred iteration for the sake of saving FLOPS as the code may run for millions of iterations.

Now that every object has been declared in the solar system class with their initial values, the final task is then to solve the future motions of the objects. This is done as before by implementing the velocity Verlet method as previously. The method is a bit different in the `solar_system` class as it does not store the position and velocities as it did in the `object` class. The only parameter that is stored during the solving process is the previous acceleration. It is shown in a two dimensional example as follows

```
for (int iter=0; iter<N; iter++){
  for (int i=0; i<num_planets; i++){
    for (int j=0; j<num_planets; j++){
      if (i!=j){
        sum_forcex += force(i,j) * (pos(2*i) - pos(2*j));
        sum_forcey += force(i,j) * (pos(2*i+1) - pos(2*j+1));
      }
    }
    ax_prev = -sum_forcex/mass(i);
    ay_prev = -sum_forcey/mass(i);
```

```
    pos(2*i)   = pos(2*i) + dt*vel(2*i) + 0.5*dt*dt*ax_prev;
    pos(2*i+1) = pos(2*i+1) + dt*vel(2*i+1) + 0.5*dt*dt*ay_prev;

    sum_forcex=sum_forcey=0;
    for (int j=0; j<num_planets; j++){
      if (i!=j){
        sum_forcex += force(i,j) * (pos(2*i) - pos(2*j));
        sum_forcey += force(i,j) * (pos(2*i+1) - pos(2*j+1));
      }
    }
    vel(2*i) = vel(2*i) + 0.5*dt*(-sum_forcex/mass(i) + ax_prev);
    vel(2*i+1) = vel(2*i+1) + 0.5*dt*(-sum_forcey/mass(i) + ay_prev);

    sum_forcex=sum_forcey=0;
  }
}
```

We notice the difference using a triple for loop instead of just one as we did in the `object` class. That is because the forces between all the planets have to be taken into account. The first `iter` loop runs through the requested amount of iterations, the `i` loop goes though the planet we are currently "standing on" and the `j` for loop indicates the other planets applying a force on planet `i`. Also we notice that the position is not stored but continuously updated as the algorithm keeps going. The position values are then stored in a separate data file for each iteration. `force(i,j)` is a function within the `solar_system` class that calculates the force between planet i and j based on Equation 11. The class may solve the Solar System problem with any amount of planets. Additional moons may also be implemented if that should be of interest but one has to be cautious of the time step not being to small.

2.6 Testing the theory of general relativity

In most cases Newtonian mechanics is more than sufficient enough to predict the planetary orbits. However in mercury's case the strong gravitational field of the Sun will perturb the orbit in such a way that general relativity is needed to explain the observed orbit. We will add a correction term to the gravitational force

$$F_G = \frac{GM_{Sun}M_{Mercury}}{r^2}(1 + \frac{3l^2}{r^2c^2}) \tag{12}$$

Where l is the magnitude of mercury's angular momentum per unit mass, c is the speed of light and r is the radius. Since there is no external torque on the system the angular momentum should be conserved

$$\frac{dl}{dt} = \frac{d(r \times v)}{dt} = m(v \times v + \vec{r} \times \vec{a}) = 0 \tag{13}$$

l is therefore constant and only needs to be calculated once.

$$b = \frac{3l^2_{mercury}}{m^2_{mercury} r^2_{semi} c^2} = 7.33 \times 10^{-8} \tag{14}$$

The values used in the calculation can be found here: wof [1].
The equation we will use is then

$$F_G = \frac{GM_{Sun}M_{Mercury}}{r^2}(1 + b(r)) \tag{15}$$

The model has been run over one century with 10^8 iterations. One orbit of Mercury is 88 Earth days, the total number of orbits is then $365 * 100/88$ and the number of iterations per orbit is then about $2.4 * 10^5$

3 Results

3.1 Testing the algorithms

We will now show the results generated from the initial tests of our algorithms. We have tested both for forward Euler and Velocity Verlet.

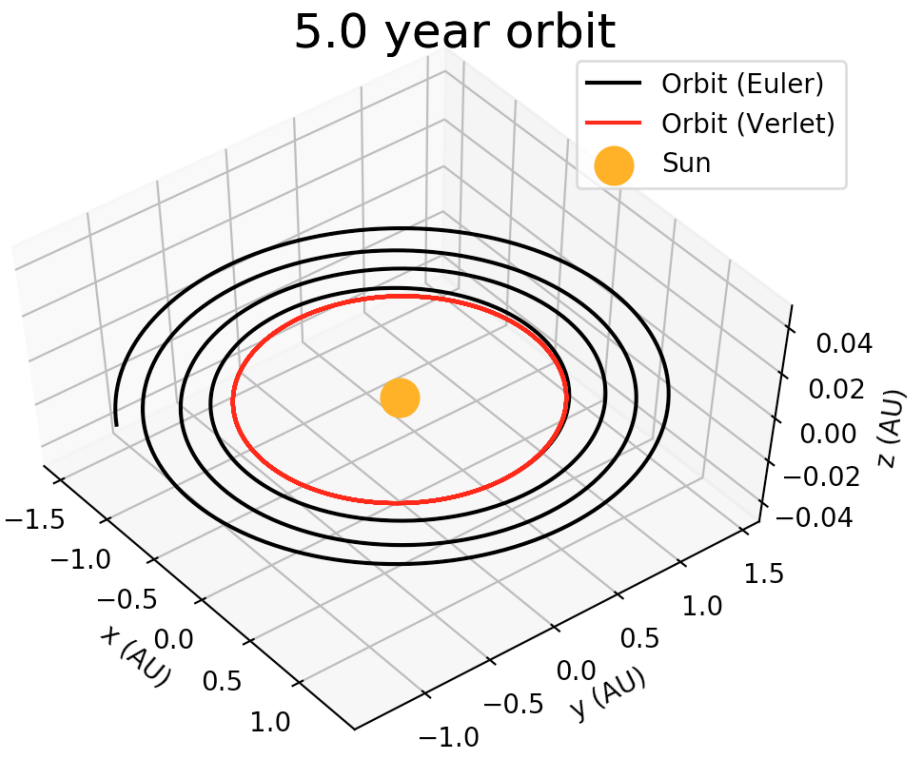

Figure 1: This plot shows the 5 year orbital path of a planet with Earths mass and initial position 1 AU from the Sun. The red orbit shows the results from the velocity Verlet algorithm and the black line shows results from the forward Euler algorithm. Both algorithms start with an initial velocity of $\sqrt{\frac{GM_\odot}{r}}$. The algorithms have been run with 1000 iterations.

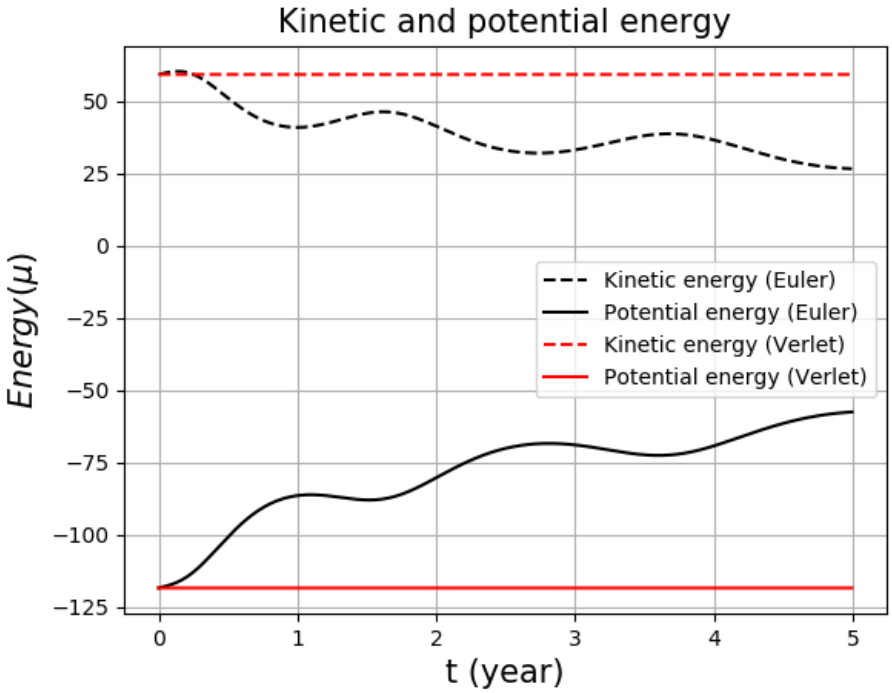

Figure 2: Here we see the kinetic and potential energy for the forward Euler(black line) and Velocity Verlet(red line). The plots show how the energies develop over a period of five years using 1000 steps.

Figure 2 shows that the fluctuations in energy for the forward Euler integration scheme are much larger than that of the Velocity Verlet. As time goes by the absolute value of the potential energy decreases using forward Euler. The kinetic energy decreases but not at the same rate as the potential energy. The consequence is that the kinetic energy will be greater than the potential energy after some time and eventually escape the gravitational pull from the Sun. Because of this, the total energy is not conserved. When zoomed in, we observe small fluctuations in the kinetic and potential energy in the Velocity Verlet algorithm. As opposed to the forward Euler algorithm the change in kinetic energy compensates for the change in potential leading to the net energy change $\Delta E = 0$.

For the Velocity verlet algorithm both the kinetic and potential energy remains more or less constant. This implies that both v and r remain constant. From Equation 5 we can conclude that this means the angular momentum will also remain constant as all the variables r, M and v are constant. This is not the case using forward Euler. We see from Figure 1 that the orbital path moves further and further away from the Sun, meaning that r grows. At the same time the kinetic energy changes. This causes the angular momentum to change as time goes by meaning

that the angular momentum is not conserved.
With our implementation of the forward Euler and velocity Verlet algorithms there is no big difference in the CPU time used to solve the same problem. For 10^7 iterations the total time was:
Forward Euler - 34.2s
Velocity Verlet - 34.5s

3.2 Escape Velocity

We tested different velocities by running our algorithm multiple times over a period of 10 000 years and checking if Earth returns into an orbit. We attempted the this using using a different number of integration points. What we found was that as the number of integration points increases the value of the escape velocity moves closer and closer to the analytical value found by using Equation 9 - 8.88577 $[\frac{AU}{yr}]$. The most accurate result we got using 10^7 iterations over 10 000 years we found that the escape velocity was 8.885 $[\frac{AU}{yr}]$.

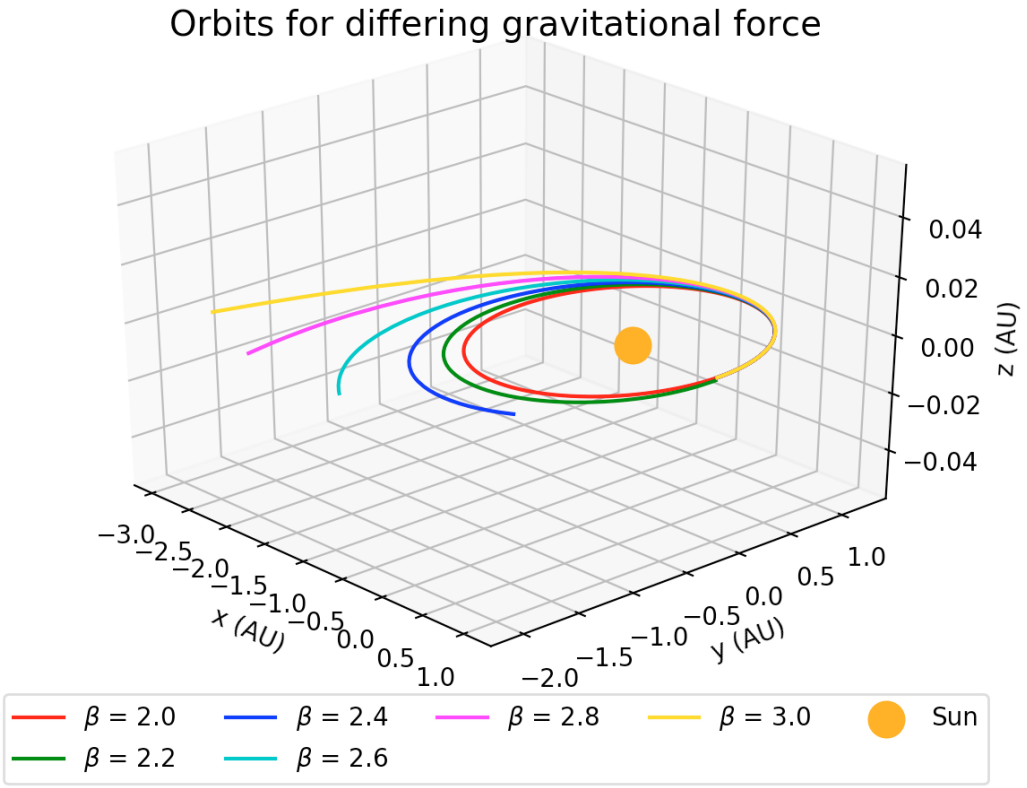

Figure 3: This figure shows how Earths orbit changes with differing gravitational force. The different colored lines show paths for differing β in the equation for gravitational force $F_G = \frac{GM_\odot M_{Earth}}{r^\beta}$. All paths have the same initial conditions: $[x, y, z] = [1, 0, 0]$, $[v_x, v_y, v_z] = [0, 6.7, 0]$.

Figure 3 we see how the orbits change as we change the parameter β between [2,3] in the gravitational force. As β moves towards 3 the orbits become more and more eccentric. The escape velocity also becomes smaller. We see that for $\beta = 2.8$ and 3 the planets do achieve escape velocity at 6.7 $[\frac{Au}{yr}]$

3.3 The three-body problem

These are the results from our simulations of the three body problem including Earth, Jupiter and the sun. In these simulations the sun remains fixed at the origin.

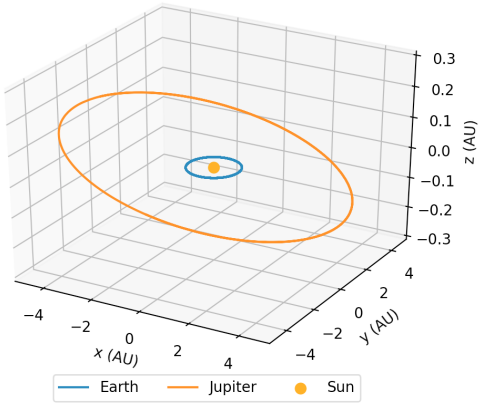

Figure 4: Jupiter mass: $m = m_{Jup}$

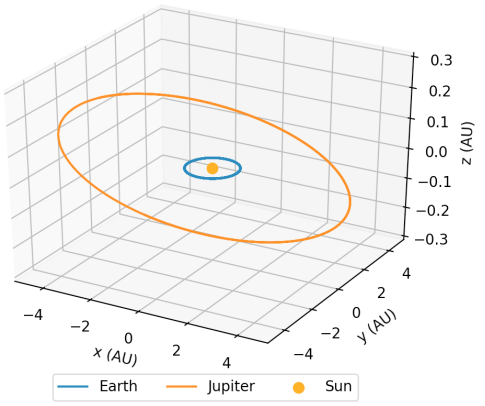

Figure 5: Jupiter mass: $m = 10m_{Jup}$

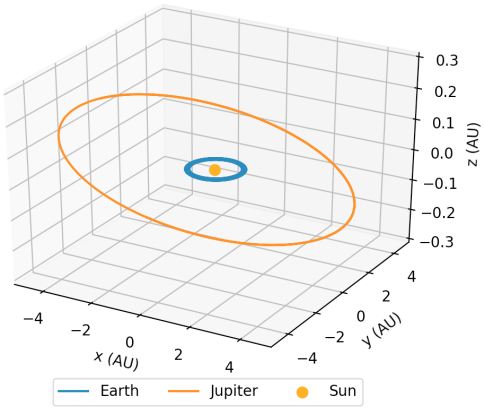

Figure 6: Jupiter mass: $m = 100m_{Jup}$

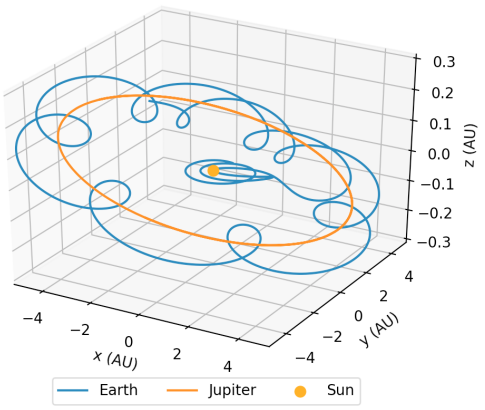

Figure 7: Jupiter mass: $m = 1000m_{Jup}$

Figure 8: The four figures show the orbits of Earth and Jupiter around the sun. Each figure shows how the orbits change as the mass of Jupiter increases. Notice that the Z axis is greatly exaggerated.

In Figure 8 we see how the mass of jupiter influence the orbit of Earth around the sun. The orbits of the normal Jupiter mass and $m_{jup} * 10$ are almost identical, the orbit of Earth is however very slightly perturbed. Whith $m_{jup} * 100$ Earths orbit starts becoming higly eccentric, with a difference of 0.257 AU between it's aphelion and perihelion. The last figure shows Jupiter with a mass roughly that of the sun. This leads to some interesting orbits where Earth leaves the sun orbit and begins orbiting Jupiter instead

3.4 Solar System Model

Here we will look at the results we got by applying our Verlet solver to the entire solar system (excluding pluto). The initial conditions of the planets are given by data from NASA [4].

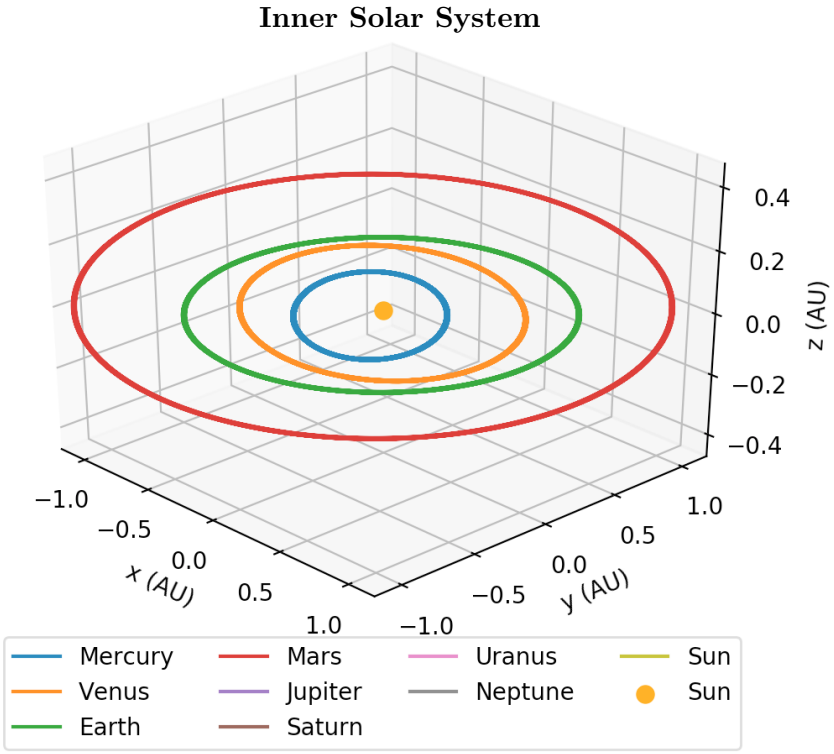

Figure 9: Model of the solar system showing the orbits of Mars, Earth, Venus and Mercury. The gas giant are also included in the calculations but are omitted from this plot.

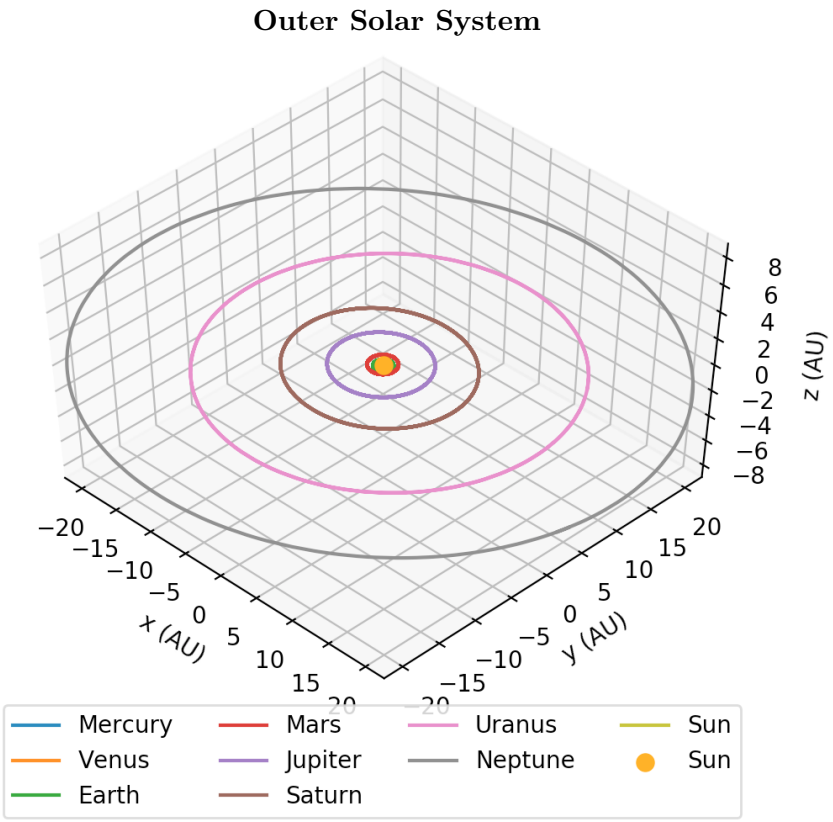

Figure 10: Model of the outer solar system, showing the orbits of Jupiter, Saturn, Uranus and Neptune

In Table 1 we have checked the accuracy of our simulations by comparing the perihelion value to planetary data from NASA. What we see from the results is that we get the best accuracy for the outer planets.

Planet	**Relative error**
Mercury	1.6%
Venus	2.4%
Earth	1.7%
Mars	0.4%
Jupiter	0.064%
Saturn	0.087%
Uranus	0.76%
Neptune	0.53%

Table 1: This table shows the relative error of our simulated perihelion values as compared with NASA data from Williams [5]

3.5 Testing the theory of general relativity

By running our model for a century with 10^8 iterations we found that on it's last orbit the perihelion value of mercury was perturbed by 409.81 arcseconds. This value was reached by using the smallest radius of the orbit. Due to numerical precision the smallest radius is in fact not the apsis of the orbit, but a point very close. By looking at the plot we manually picked the point where we would expect to see the apsis. We define this point as the place where the orbit shifts from a positive motion in the x direction to negative motion. By using this point we found that the orbit was perturbed by 40.80 arcseconds.

4 Discussion

4.1 Testing the algorithms

The energy should be conserved because there are no external forces acting upon the system. The results from our initial testing show that for planetary orbits we should not use the forward Euler algorithm. This is because the generated orbits are unstable. This can be seen in Figure 1 where the orbital radius becomes larger over time. This effect is magnified by the fact that we are using a small number of integration points. We observed that for a large number of integration points the orbit is more stable, but the energy is still not conserved. When dealing with time spans much longer than the five years we shown in our simulation, say the lifespan of the solar system, you would need an enormous amount of integration points to achieve anything approaching a stable orbit. The velocity Verlet algorithm is a much better choice, the orbits will remain stable over time because the energy is conserved. Looking at the orbit in Figure 1 we see that the orbit is more or less constant, even with a small amount of integration points. We also saw that the two algorithms are very close when it comes to CPU time. All in all this makes the velocity Verlet algorithm much more suited to this problem than the forward euler.

4.2 Escape velocity

Since we found that the escape velocity of our model matched the analytical escape velocity well, we gain more confidence in our model, as it is able to predict real world phenomena.
It seems quite obvious that reducing the gravitational force will reduce the velocity needed to escape the system, as there is less force pulling back. And indeed this is exactly what we found in Equation 9.

4.3 The three-body problem

In Figure 4 and 5 Earth's orbit is barely changed as the mass of Jupiter is increased with a factor of 10. The Sun's pull is still much greater. Recall Equation 1, objects

that are closer to Jupiter would be more affected by the change in Jupiter's mass such as Mars or the asteroid belt. When the mass of Jupiter is increased with a factor of 100 in Figure 6, Earth's motion is visually changing. Earth's orbit is disturbed, making it more eccentric. In Figure 7 the mass of Jupiter and the Sun is more or less the same which causes the Earth to orbit all over the place.

The three-body problem is a chaotic system meaning that small changes in any parameter would cause a very different output. Even if we were to try to simulate the system with same parameters but on a different computers, the difference in precision on each computer may cause the orbit of Earth to look quite different. The model is therefore not repeatable but rather unique for each computer. Even though a simulation may not be repeatable the total energy in the system should be conserved if the resolution is good enough. This is checked by storing the potential and kinetic energy of all the planet in a data file and then plotted using Python like so

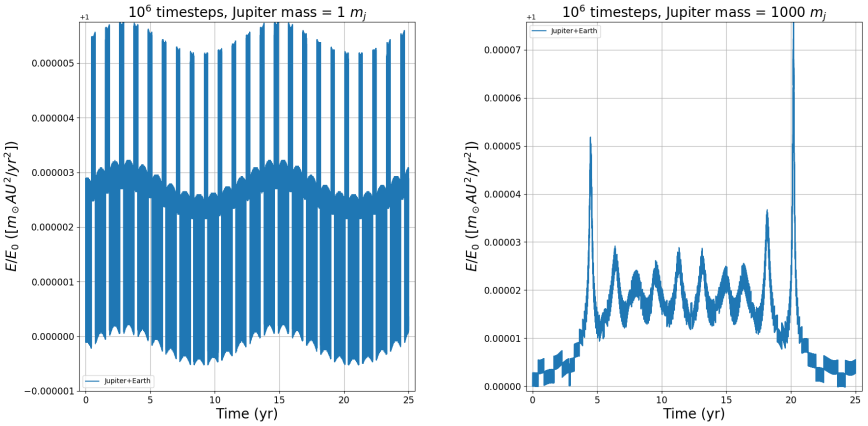

Figure 11: Total normalized energy of the solar system. Only Jupiter and Earth is present during this simulation.

We notice some fluctuations in the total energy in the system as seen from Figure 11 but these are minor when looking at the y-axis. The data is normalized so the values here show the change in energy with regards to the initial energy E_0. The greatest magnitude of error is just above 0.007 % when Jupiter's mass is enlarged with a factor of 1000. The error is even smaller when Jupiter's mass is normal. The errors may occur due to loss of numerical precision. However, as the fluctuations are very small we may safely assume that the total energy is conserved. This implicitly means that the chosen step size of $2.5 \cdot 10^{-5}$ should be good enough during the simulation.

4.4 Solar System Model

As seen from Figure 9 and 10 the model of the entire solar system performs quite well. Especially the more distant planets, as confirmed in Table 1. During the simulation the innermost planets do many more orbits than the outer planets. This may be part of the reason why they have a larger error than the outer planets. To get a better results for the inner planets we would need a higher resolution simulation. This is because in each timestep the inner planets will parse over a larger distance than the outer planets, leading to a coarser orbit and larger error. Our tests show that the energy of the system is conserved, at least within satisfactory limits. This means that the orbits should remain relatively stable, even over large time periods.

Moons may easily be included in the Solar system model. One needs only the initial condition of each of the moons. The drawbacks may cause the generated data files to be very large.

4.5 Testing the theory of general relativity

The results of our model including the correction term from general relativity seem to match well with the observed value. The model gave a precision of 40.80 arcseconds while the observed value is 43 arcseconds. This is quite close and is in any case better than the model without the correction term. The results we gained from this test should be taken with a large grain of salt as the selection process contains a high degree of subjectivity of where I "think" the perihelion should be. It would have been nice to run the simulation with a higher resolution, but our computers ran out of memory when going above 10^8 iterations. It would have been possible to run more simulations if our program was structured differently, for example by not using vectors but instead writing to a file for every iteration. This would mean that the computer would not have to allocate a large part of its memory to saving vectors.

References

[1] Wolfram alpha. 2019. URL https://www.wolframalpha.com/input/?i=3+%28mercury+orbital+momentum+%2F+mercury+mass%29%5E2+%2F+%28%28mercury+orbital+radius%29*%28speed+of+light%29%29%5E2+.

[2] Morten Hjorth Jensen. Computational physics, lecture notes. 2015. URL http://compphysics.github.io/ComputationalPhysics/doc/pub/ode/pdf/ode-print.pdf.

[3] NASA. Anatomy of an exosolar system. 2012. URL https://asd.gsfc.nasa.gov/blueshift/index.php/2012/08/03/jillians-blog-anatomy-of-an-exosolar-system/.

[4] NASA. Horizons web-interface. 2019. URL https://ssd.jpl.nasa.gov/horizons.cgi?s_target=1#top.

[5] Dr. David R. Williams. Planetary fact sheet. 2019. URL https://nssdc.gsfc.nasa.gov/planetary/factsheet/.