Project 3 - Solar System

Sam Edwards,[1] Eric Aboud,[1] and Parker Brue[1]

[1] *Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48823*

We present an algorithm that allows us to solve coupled differential equations. Using Euler's forward algorithm and the Verlet algorithm, we are able to solve for many body systems. Comparisons between the methods for few body models allows us to verify results. Ultimately, we explicitly plot out a partially and fully populated planetary solar system. The methods covered allow for us to extend our models to varied many body systems.

I. INTRODUCTION

Planetary motion, while decidedly very complex in reality, can be simulated with knowledge of the gravitational force between objects and its impact on orbital motion. This project serves as an engaging stepping stone into the finer nuances of how bodies interact with each other. We present multiple algorithms within a class structure that allows for us to solve coupled ordinary differential equations. We first compare Euler's forward algorithm with the Verlet algorithm in simple two and three body cases. Further focusing on the Verlet algorithm, we are able to specialize it to build a working model for the planetary motion of the solar system. The integration of a generic class structure allows for not only streamlined code implementation in the main program, but also acts as a possible extension into similar many-body problems such as a molecular or atomic system. Chronologically, our report first discusses the theory, algorithms and methods behind our project, then transition into a structured analysis of the results to the posed questions. Finally, we summarize, and consider the consequences of our model and how we can improve our methods.

II. THEORY, ALGORITHMS AND METHODS

A. Ordinary Differential Equations

We can first start with the force described by Newton's law of gravitation between the Sun and Earth [1]:

$$F_G = \frac{GM_\odot M_{\text{Earth}}}{r^2}$$

where we can define $M_\odot$ as the solar mass, M_{Earth} the mass of the earth, G the gravitational constant, and r the mean distance.

Our knowledge of the solar system allows us to make a simple assumption, that we can estimate the motion of the planets in a single plane which allows us to constrain our differential equations to the x-y plane. Since $F_x = m\ddot{x}$ and $F_y = m\ddot{y}$ we can relate the gravitational force to the position, velocity, and acceleration of the bodies in the form of coupled differential equations:

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

[2]

B. Verlet Algorithm

Pseudo-code is included below, with important information discussed in the captions.

```
   void solver::VelocityVerlet(int dimension, int
       integration_points, double final_time, int
       print_number, double epsilon)
{  ...
    // PLANET CALCULATIONS
    time += time_step;
    while(time < final_time){
        lostPlanets[n] = 0;

        // Loop over all planets
        for(int nr1=0; nr1<total_planets; nr1++){
            planet &current = all_planets[nr1]; //
                Current planet we are looking at

            Fx = Fy = Fz = Fxnew = Fynew = Fznew =
                0.0; // Reset forces before each run

        // Calculate forces in each dimension
            for(int nr2=nr1+1; nr2<total_planets;
                nr2++){
                planet &other = all_planets[nr2];
                GravitationalForce(current,other,Fx,Fy,Fz,
                epsilon);
               }

            // Acceleration in each dimension for
                current planet
            acceleration[nr1][0] = Fx/current.mass;
            acceleration[nr1][1] = Fy/current.mass;
            acceleration[nr1][2] = Fz/current.mass;

            // Calculate new position for current
                planet
            for(int j=0; j<dimension; j++) {
                current.position[j] +=
                    current.velocity[j]*time_step+
                    0.5*time_step*time_step
                *acceleration[nr1][j];
```

```
        }

        // Loop over all other planets
            for(int nr2=nr1+1;
                nr2<total_planets; nr2++){
                planet &other = all_planets[nr2];
                GravitationalForce(current,other,
                Fxnew,Fynew,Fznew,epsilon);
            }

        // Acceleration each dimension exerted
            for current planet
        acceleration_new[nr1][0] =
            Fxnew/current.mass;
        acceleration_new[nr1][1] =
            Fynew/current.mass;
        acceleration_new[nr1][2] =
            Fznew/current.mass;

        // Calculate new velocity for current
            planet
        for(int j=0; j<dimension; j++)
            current.velocity[j] +=
            0.5*time_step*(acceleration[nr1][j]
            + acceleration_new[nr1][j]);
    }...

}
```

FIG. 1: The Verlet algorithm that was used in solving for new accelerations and positions. The algorithm begins with resetting the forces to initial conditions (F=0). It then calculated the gravitational forces between all of the planets and calculated the acceleration in the x and y directions. The algorithm then proceeds to calculate the position and velocity of each planet.

```
void solver::GravitationalForce(planet
    &current,planet &other,double &Fx,double
    &Fy,double &Fz,double epsilon)
   {   // Function that calculates the
       gravitational force between two objects,
       component by component.

   // Calculate relative distance between current
       planet and all other planets
   double relative_distance[3];

   for(int j = 0; j < 3; j++) relative_distance[j]
       = current.position[j]-other.position[j];
   double r = current.distance(other);
   double smoothing = epsilon*epsilon*epsilon;

   // Calculate the forces in each direction
   Fx -= this->G*current.mass*other.mass
   *relative_distance[0]/((r*r*r) + smoothing);
   Fy -= this->G*current.mass*other.mass
   *relative_distance[1]/((r*r*r) + smoothing);
   Fz -= this->G*current.mass*other.mass
   *relative_distance[2]/((r*r*r) + smoothing);
}
```

FIG. 2: A key part of the Verlet algorithm is the calculation of the gravitational forces. The gravitational force is calculated by using the distance and masses of the involved planets. This part of the algorithm calculates the gravitational force in all three dimensions, while only the x and y dimensions are used in further analysis.

Interesting pieces of the code to note are the appearance of variables like "total_planets" and the vector "all_planets". Through object orientation and planet class files (discussed in section II D), the gravitational forces used in the ODEs can be calculated for any specified number of planets that are appended to the vector "all_planets".

Also helpful was the argument "print_number" included in both algorithms. This number corresponds to the number of planets' position and velocity information outputted into a text file. Importing these text files into python made plotting the orbits very simple.

C. Euler Algorithm

The major difference between the Euler algorithm and the Verlet algorithm is how the position and velocity are updated. Pseudo-code highlighting those differences is shown below.

```
    for(int j=0; j<dimension; j++) {
              current.position[j] +=
                  current.velocity[j]*time_step;
          }
```

FIG. 3: Notice that the position's update does not depend on the acceleration, like in the Verlet algorithm.

```
    for(int j=0; j<dimension; j++)
        current.velocity[j] +=
        (-4*M_PI*M_PI*current.position[j]*time_step);
```

FIG. 4: The $-4\pi^2$ is actually equivalent to the acceleration of Earth. Normally this term is divided by r^3, but for Earth's case, r is 1 AU. Note that for this reason, this would not work for other planets

D. Object Orientation

In order to handle a number of bodies that behave the same way, a class for planets was set up to streamline how these objects interact according to Newton's laws. In the figures below (figure 5), certain attributes of the planet class are shown. The user can easily initialize the

planet's initial mass, position, and velocities, its distance to other planets, and the affect of gravity of larger bodies on smaller bodies.

Initially in the project the critical parameters were simply the Earth's attributes relative to the sun. It was important to be able to easily change the mass of bodies when running simulations on how a much larger Jupiter could affect Earth's orbit.

```
   planet::planet(double M, double x, double y,
       double z, double vx, double vy, double vz)
{
    mass = M;
    position[0] = x;
    position[1] = y;
    position[2] = z;
    velocity[0] = vx;
    velocity[1] = vy;
    velocity[2] = vz;
    potential = 0.;
    kinetic = 0.;
}
double planet::distance(planet otherPlanet)
{
    double x1,y1,z1,x2,y2,z2,xx,yy,zz;

    x1 = this->position[0];
    y1 = this->position[1];
    z1 = this->position[2];

    x2 = otherPlanet.position[0];
    y2 = otherPlanet.position[1];
    z2 = otherPlanet.position[2];

    xx = x1-x2;
    yy = y1-y2;
    zz = z1-z2;

    return sqrt(xx*xx + yy*yy + zz*zz);
 }

double planet::GravitationalForce(planet
     otherPlanet,double Gconst)
{
    double r = this->distance(otherPlanet);
    if(r!=0) return
        Gconst*this->mass*otherPlanet.mass/(r*r);
    else return 0;
}
```

FIG. 5: Example of the object orientation code used in the current work. This sample code displays how we can define objects that require certain inputs and how we can define the inputs. Objects allow us to easily input new planets and objects into our code.

III. ANALYSIS

A. Velocity for a circular orbit

We were initially tasked with finding a value for the Earth's velocity that gives a circular orbit. We checked the values in ref. [3] for initial conditions before calculating for the orbital velocity. The orbital velocity can be solved for by equating the Earth's centripetal force with the gravitational force. In units of AU/year, this velocity simply comes out to 2π. We tested the Verlet algorithm for different time steps and also plotted with Earth's position with its proper orbital velocity.

Time Step [years]	Final Distance [AU]
0.005	1.000
0.01	1.000
0.05	1.037
0.1	1.187
0.5	5.23E4
1.0	2.055E5

TABLE I: Time steps being varied in the Verlet method. The final distance is found using the final x and y coordinates of Earth.

As you can see, as the time step gets larger, the Earth begins to deviate from a circular path and travel much farther than 1 AU away from the Sun. This is because the Earth's position and velocity are not getting updated at fine enough intervals, decreasing the accuracy of the final position as the algorithm iterates.

The plot of the Earth's x and y positions using the proper orbital velocity and a time step of 0.005 can be seen below. A python notebook was used to import the text file that the C++ file outputted containing the positions and velocities at each step, and the positions were then plotted.

Kinetic and potential energy is conserved in this plot as well; the "EnergyLoss" function determines this and prints out the output as zero when the Verlet method is ran. Angular momentum is also conserved. In the same python notebook used to make the above plot, the initial and final velocities can be examined.

Defining angular momentum L as mvr, one can see that if mass and distance from the sun do not change, then only the change in velocity is necessary to examine to determine a change in L. We know that velocity did not change because there the EnergyLoss function returned zero, implying kinetic energy was conserved. For the above bound system, L did not change (as expected; energy and angular momentum should be conserved unless an outside force acts on the system). The calculation to prove this is shown explicitly in the python notebook that produces the plots.

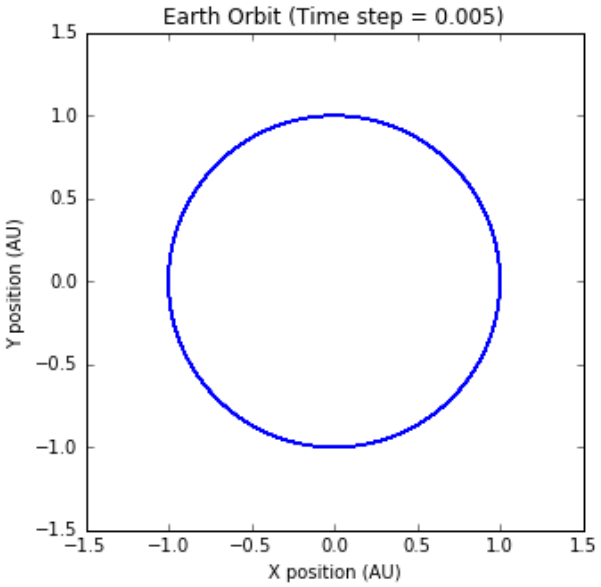

FIG. 6: Earth's orbital path, centered on the origin (Sun). Time step is 0.005 and v = 2π

B. Comparison to Euler's

The Euler algorithm is asymmetric in time as it calculates its positions and velocities; the Verlet algorithm is also, but it uses 'half' steps in time that allow closer, more accurate updating of the position. We expect the Verlet algorithm to be more accurate with larger, more inaccurate time steps than the Euler algorithm.

To see how the number of FLOPs varies between algorithms, one just needs to look at the number of operations each algorithm spends on updating both the position and the velocity. Verlet's position and velocity updates take six and four respectively; Euler's position and velocity updates take two and five respectively. In timing the algorithms, this difference should be reflected.

A table of the timings for different final times (and therefore, different time intervals) is shown below.

Final Time [years]	Verlet Time [sec]	Euler Time [sec]
50	0.1591	0.1516
100	0.1546	0.1527
500	0.1569	0.1704
1000	0.1566	0.1615
5000	0.1631	0.1441
10000	0.1860	0.1516

TABLE II: The times of the Verlet and Euler algorithms for different final times. A larger final time gives larger time steps, which gives the while loop in the solvers less iterations to complete. It appears that on average, the Euler algorithm is quicker, but not by much.

IV. RESULTS AND DISCUSSION

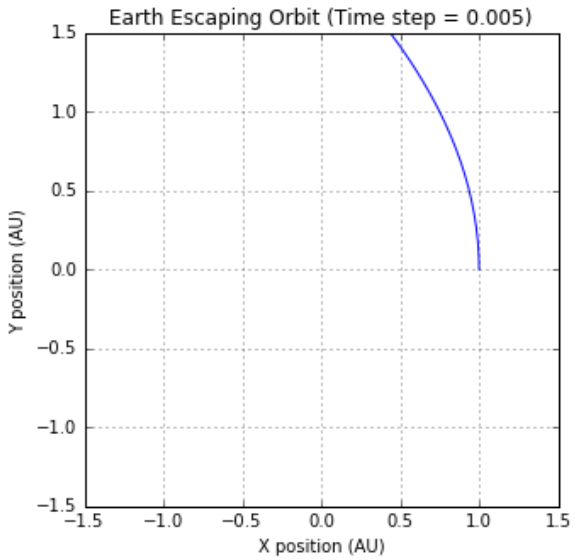

FIG. 7: Earth escaping its orbital path. Time step is 0.005 and v = $\sqrt{2}$*2π.

A. Escape velocity

Setting kinetic energy equal to gravitational potential energy gives the second cosmic speed, or 'escape velocity', of an object in orbit. This only differs from the orbital velocity by a factor of $\sqrt{2}$. A plot is shown in Figure 7.

B. Three-Body Problem

Jupiter Mass	Deviation	
	Final Distance	Final Velocity
[Relative mass]	[AU]	[AU per Year]
1	0.00060	0.00350
10	0.00318	0.01808
100	0.00240	0.06054

TABLE III: The deviations of both the velocity and distance from its initial values of 1 AU and 2π AU/year found in the final earth orbit.

While the deviations in distance aren't very large, nor does there seem to be a clear pattern, the relative deviations in the velocity indicate a clear trend in increased velocity corresponding to increased Jupiter mass (Figure 8 and Table IV B). Extrapolation of this trend would indicate a very unstable Earth orbit as Jupiter approached the mass of the sun

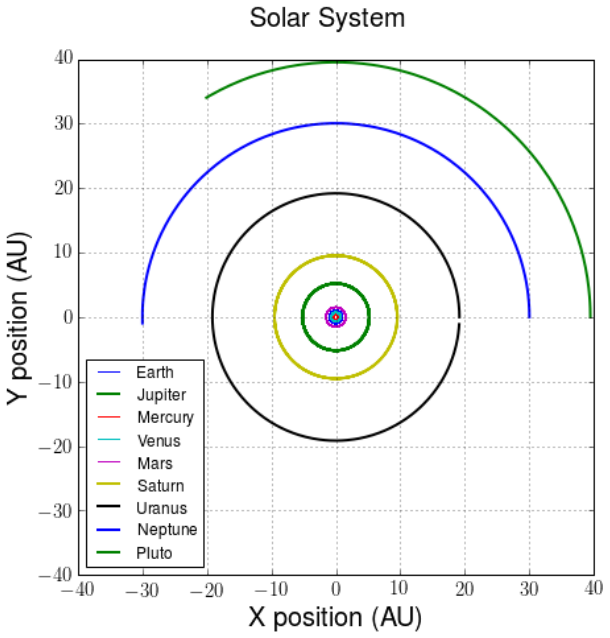

FIG. 9: An x-y planar view of the motion of the planets. Data points for complete orbits for the outer three planets was not available due to a short simulation time set in the code. Full orbits can clearly be extrapolated to show the full, circular orbits of all planets.

Planet	Mass [Solar Mass Units]	Radius [Au]
Sun†	1.0	0
Mercury	1.66E-7	0.39
Venus	2.46E-6	0.72
Earth	3.0E-6	1.0
Mars	3.32E-7	1.52
Jupiter	9.55E-4	5.20
Saturn	2.77E-4	9.45
Uranus	4.42E-5	19.19
Neptune	5.18E-5	30.06
Pluto†	6.59E-9	39.53

TABLE IV: The planets used in the final simulation of the solar system. Included are masses and distances from the sun, which is placed at the origin. The † represents objects that are not defined as planets.

C. Complete Solar System

Having a class that defines planets made it very simple to add more bodies to our system that was already shown to work with just Earth, Jupiter, and the sun. All that was necessary was typing in each planet's mass (in solar mass units) and radius of orbit (in AU), which was provided in our project description (this data can be seen in Table IV B.) Similar to the simpler system, the energy loss was zero and ten bound objects (including the sun) were found.

It should also be noted that each of the planet's initial velocities is simply 2π divided by the square root of the orbital radius in AU, according to the orbital speed equation derived earlier for Earth.

We plotted the orbits of each of the planets. However, in this plot, we re-framed our program in a center-of-mass system. To make the momentum zero, we assumed that every planet started out in a line initially, the x axis, and gave the sun an initial velocity in the opposite direction of each of the planets to balance. This calculation is done in our main cpp file. You can see the simulation of the orbits in figure 9.

V. CONCLUSION

We were able to construct an algorithm and class structure that produces the orbits of Earth-Sun, Earth-Jupiter-Sun, and all planet systems. We found, for the Earth-Sun system, that for small times (∼50 years) both the Verlet and Euler algorithms take a comparable time to compute the positions and velocities of the planets. As we increased the time (∼10000 years) the Verlet algorithm slowed compared to the Euler algorith. While both computational times differ, they still remain very small (Figure III B). Figure 7 shows the position of earth during its escape after we found the escape velocity and applied it to the algorithm. We then were able to take the Earth-Jupiter-Sun system and determine the effect that Jupiter had on Earths orbit (Figure 8 and Table IV B). We were able to solve for the whole solar system and determine the orbits of the planets (Figure 9).

Possible deeper exploration upon our project in the future would be the implementation of non-planetary bodies such as all of the moons and some major comets and asteroids. Another possibility is expansion of the code to account for different interactive forces. While our code functions on the assumption that all of the objects orbit in the 2d plane, a option for refinement is accounting for objects that orbit in aberration to the 2d solar plane.

ACKNOWLEDGEMENTS

The authors would like to thank Morten for teaching us how to do C++ stuff.

[1] "Project 3," https://compphysics.github.io/ComputationalPhysicsMSU/doc/Projects/2017/Project3/pdf/Project3.pdf, accessed: 2017-04-08.

[2] "The orbital velocities of the planets," http://www.sjsu.edu/faculty/watkins/orbital.htm, accessed: 2017-04-08.

[3] "Horizons web-interface," https://ssd.jpl.nasa.gov/horizons.cgi#top, accessed: 2017-04-08.

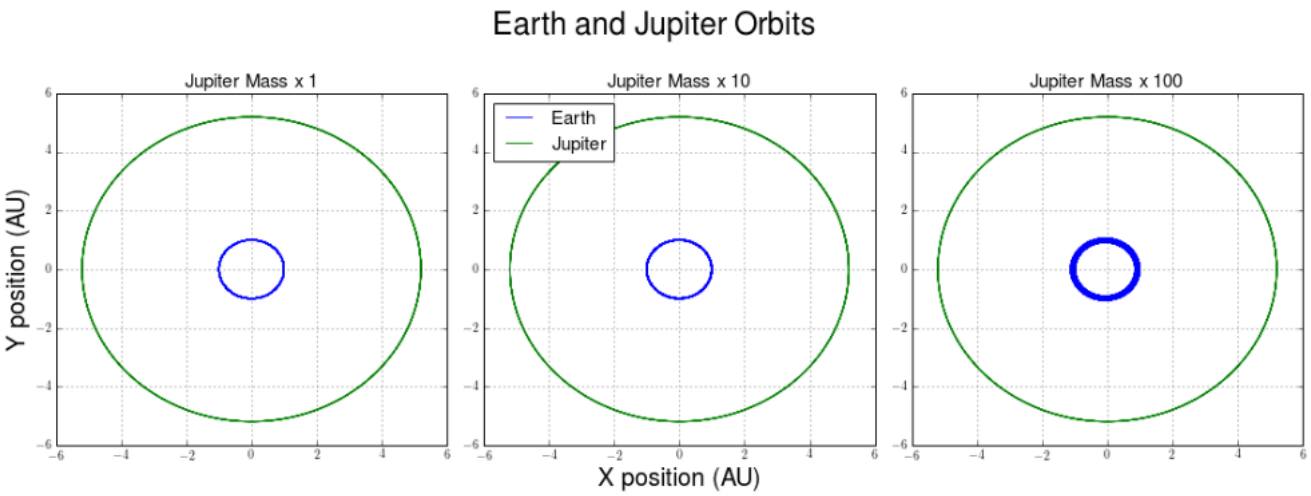

FIG. 8: Earth and Jupiter orbits for increasing values of Jupiter's mass. From left to right, Jupiter has its original mass, its mass times ten, and its mass times one hundred.