On the simulation of gravitational interaction in n-body systems using the forward Euler and velocity Verlet methods of numerical integration

Eivind Støland, Anders P. Åsbø
(Dated: October 29, 2020)

We derived and implemented in C++ the forward Euler and velocity Verlet methods of numerical integration. Firstly, we applied our implementations to simulate variations of a Sun-Earth 2-body system, and found that the velocity Verlet method, while slower, converged faster than forward Euler. Velocity Verlet was found to be symplectic as expected. For simulations with gravity proportional to $1/r^\beta$, instead of $1/r^2$, we found that β must be close to 2 to avoid significant precession of non-circular orbits. For simulations of the Sun's escape velocity at 1 AU, the analytic value of $2\pi\sqrt{2}$ AU/year gave a net energy $E_{net}/E_{max} = -3.33 \times 10^{-2} < 0$ likely due to truncation error.

We simulated variations of a Sun-Earth-Jupiter 3-body system, finding that the Sun's apoapis was approximatly 1.1 solar radii from the center of mass. Increasing Jupiter's mass by a factor 10^3 led to significant orbital motion of the Sun. We also simulated a system with the Sun, all eight planets and Pluto with stable orbits over 248 years using a timestep of 2.48×10^{-5} years. Finally we simulated the perihelion precession of Mercury with and without a relativistic correction of Newtonian gravity, over 100 years with timestep 10^{-7}. This gave a precession per century of $42.7924''$ with a relative error 0.483% from the observed $43''$. The non-relativistic precession approached $0''$ as expected for a 2-body system.

CONTENTS

I. INTRODUCTION

Differential equations are the mathematical description of continuous change, and thus one of the most important cornerstones of physics since their invention. In spite of how often they appear in physics, few differential equations have analytic solutions, and as such it is vital to be able to approximate a solution numerically.

In this report, we will derive two different methods of numerical integration, namely the forward Euler method and the velocity Verlet method. We will implement these methods in C++ and perform several tests of the validity of the implementations. To test important properties of the methods, such as conservation of energy, conservation of angular momentum and numerical stability, we will apply the methods to solve the differential equations that arise in celestial mechanics when simulating the motion of objects in our Solar System. Some of these tests will be performed automatically as unit-tests, but results pertaining to these tests will be presented as well. In order to test the efficiency of the methods we will also perform benchmarks, and compare the results to an expected amount of floating point operations needed to perform the algorithms.

We firstly apply both methods to solve the Sun-Earth two-body system with only gravitational interaction. The simulations are performed with varying precision to ascertain the stability and convergence of each method.

Furthermore, we use the velocity Verlet method to simulate orbits of an Earth-like object with initial velocity close to the Sun's escape velocity at a 1 AU distance. We also simulate the Sun-Earth system for both circular and non-circular orbits, using variations of Newtonian gravity that are not pure inverse-square laws. To test our implementation's ability to solve a system with no analytic solution, we apply the velocity Verlet method to simulate a 3-body system consisting of the Sun, Earth and Jupiter, with different masses for Jupiter per simulation.

To test the generality of our implementation, we apply the velocity Verlet method to simulate an n-body system consisting of the Sun with eight planets and Pluto.

Finally, we apply the velocity Verlet method together with a relativistic correction of Newton's law of gravitation to simulate the perihelion precession of Mercury's orbit. A simulation with no relativistic correction is also done for comparison.

II. FORMALISM

A. Two methods for numerical integration

To solve our equations of motion we require methods for numerical integration. To derive our first method, we consider the generic, unknown function f that depends on a variable t, and a known function g that depends on t and f such that:

$$\frac{d}{dt}f(t) = g(t, f(t)). \tag{1}$$

Given an arbitrary point $f(t_0)$, we wish to approximate the point $f(t_1)$. Thus we can integrate each side of (1), such that:

$$\int_{t_0}^{t_1} \frac{d}{dt}f(t)dt = \int_{t_0}^{t_1} g(t, f(t))dt.$$

Applying the fundemental theorem of calculus, we get that:

$$\begin{aligned} f(t_1) - f(t_0) &= \int_{t_0}^{t_1} g(t, f(t))dt, \\ f(t_1) &= f(t_0) + \int_{t_0}^{t_1} g(t, f(t))dt. \end{aligned} \tag{2}$$

To complete the approximation, we define a step-length $\Delta t = t_1 - t_0$, and approximate the area under the curve of $g(t, f(t))$ between $g(t_0, f(t_0))$ and $g(t_1, f(t_1))$ as the area of a rectangle with height $g(t_0, f(t_0))$. Inserting the rectangle approximation into (2) yields:

$$f(t_0 + \Delta t) \approx f(t_0) + \Delta t g(t_0, f(t_0)).$$

As we neglect all terms of second order or higher in this approximation we have that the error goes as $\mathcal{O}(\Delta t^2)$. If we want to approximate a value $f(t_{k+1})$, we can simply start with a known value at a $t_k < t_{k+1}$ and take a step with our approximation such that:

$$f(t_{k+1}) \approx f(t_k) + \Delta t g(t_k, f(t_k)).$$

Finally we can simplify the notation by setting $f_{k+1} = f(t_k)$ resulting in

$$f_{k+1} = f_k + \Delta t g(t_k, f_k). \tag{3}$$

This method of numerical integration is known as Euler's method, henceforth referred to as the forward Euler method.

Since we are approximating the integral between each pair of $g(t_k, f_k)$ and $g(t_{k+1}, f_{k+1})$ as a single rectangle with width Δt, our approximation will be more accurate for smaller values of Δt. Thus to approximate the curve of the function f, we create a set of f_k values for a given f_0 and Δt by repeatedly taking steps using the forward Euler method.

Many physical problems can be represented by a second order differential equation:

$$\frac{d^2}{dt^2}\vec{x}(t) = \vec{a}(t, \vec{x}(t)), \tag{4}$$

where $\vec{x}(t)$ is the function we want to find and $\vec{a}(t, \vec{x}(t))$ is a function dependent on $\vec{x}(t)$. As the parts of this equation are vectors, this is actually a set of D differential equations, where D is the length of the vectors. This differential equation can be split into $2D$ first order differential equations:

$$\begin{aligned} \frac{d}{dt}\vec{v}(t) &= \vec{a}(t, \vec{x}(t)) \\ \frac{d}{dt}\vec{x}(t) &= \vec{v}(t)\,, \end{aligned} \tag{5}$$

When split into first order differential equations like this we can easily solve this system using methods such as the forward Euler method given a set of initial conditions.

Another method used to solve the system of equations in (5) is the velocity Verlet algorithm. We will now derive this algorithm. First we Taylor expand $\vec{x}(t + \Delta t)$:

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \Delta t\vec{v}(t) + \frac{\Delta t^2}{2}\vec{a}(t) + \mathcal{O}(\Delta t^3) \tag{6}$$

Using the discretization described earlier, along with defining $\vec{x}(t_k) = \vec{x}_k$ and $\vec{v}(t_k) = \vec{v}_k$ we can rewrite this equation:

$$\vec{x}_{k+1} \approx \vec{x}_k + \Delta t\vec{v}_k + \frac{\Delta t^2}{2}\vec{a}(t_k, \vec{x}_k) \tag{7}$$

Similarly we can also Taylor expand $\vec{v}(t + \Delta t)$:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \Delta t\vec{a}(t) + \frac{\Delta t^2}{2}\frac{d}{dt}\vec{a}(t) + \mathcal{O}(\Delta t^3) \tag{8}$$

We do not know what $\frac{d}{dt}\vec{a}(t)$ is, but we can approximate it:

$$\begin{aligned} \frac{d}{dt}\vec{a}(t) &= \frac{\vec{a}(t + \Delta t) - \vec{a}(t)}{\Delta t} + \mathcal{O}(\Delta t^2) \\ \Delta t\frac{d}{dt}\vec{a}(t) &= \vec{a}(t + \Delta t) - \vec{a}(t) + \mathcal{O}(\Delta t^3) \end{aligned}$$

Inserting this into (8) gives:

$$\begin{aligned} \vec{v}(t + \Delta t) &= \vec{v}(t) + \Delta t\vec{a}(t) + \frac{\Delta t}{2}(\vec{a}(t + \Delta t) - \vec{a}(t)) + \mathcal{O}(\Delta t^3) \\ &= \vec{v}(t) + \frac{\Delta t}{2}(\vec{a}(t + \Delta t) + \vec{a}(t)) + \mathcal{O}(\Delta t^3) \end{aligned}$$

Discretizing this gives:

$$\vec{v}_{k+1} \approx \vec{v}_k + \frac{\Delta t}{2}\Big(\vec{a}(t_{k+1}, \vec{x}_{k+1}) + \vec{a}(t_k, \vec{x}_k)\Big)$$

Combining this with (7) gives us the velocity Verlet algorithm:

$$\begin{aligned} \vec{x}_{k+1} &= \vec{x}_k + \Delta t\vec{v}_k + \frac{1}{2}\Delta t^2\vec{a}(t_k, \vec{x}_k) \\ \vec{v}_{k+1} &= \vec{v}_k + \frac{1}{2}\Delta t\Big(\vec{a}(t_{k+1}, \vec{x}_{k+1}) + \vec{a}(t_k, \vec{x}_k)\Big)\,. \end{aligned} \tag{9}$$

In this algorithm we first move the position one step, and then use this when calculating $\vec{v}$ in the next step. This algorithm has two advantages over the forward Euler method. Firstly it is symplectic, meaning that if it is used to solve a set of equations given by Newton's second law there is no energy drift coming from the algorithm itself. Secondly the error of the approximation goes as $\mathcal{O}(\Delta t^3)$ compared to $\mathcal{O}(\Delta t^2)$ for the forward Euler method.

B. A generalized gravitational force

For any two bodies a and b, the gravitational force acting on body a from body b is:

$$\vec{F}_a = -G\frac{m_a m_b}{|\vec{r}_a - \vec{r}_b|^3}\left(\vec{r}_a - \vec{r}_b\right), \tag{10}$$

where m_a and m_b are the masses of a and b respectivly, $\vec{r}_a$ and $\vec{r}_b$ are the position vectors of a and b respectivly, and G is the gravitational constant. The potential energy and the force are connected by the relation $\vec{F} = -\boldsymbol{\nabla}U$. This gives us that the potential energy has to be:

$$U = -G\frac{m_a m_b}{|\vec{r}_a - \vec{r}_b|} \tag{11}$$

For a circular motion of body a around b this should remain constant as the distance between the objects is always the same. The total energy in such a two-body system is also conserved, and this means that the kinetic energy also has to be conserved, as the total energy is the sum of the kinetic and potential energy. At this point this is just an assumption, but we will prove it in the next section. In an elliptical motion however, only the total energy is conserved, and we will see oscillations in the kinetic and potential energy.

If we consider a body in a n-body system where the only interaction is gravitational, the net force on any body i becomes the sum of all the forces on body i from every other body in the system. Using (10) and summing over all bodies j, where $j \neq i$, we get:

$$\vec{F}_i = -Gm_i\sum_{j\neq i} m_j\frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^3}. \tag{12}$$

We then find the acceleration of body i at time t by dividing the net force by m_a resulting in:

$$\vec{a}_i(t) = \frac{\vec{F}_i(t)}{m_i} = -G\sum_{j\neq i} m_j \frac{\vec{r}_i(t) - \vec{r}_j(t)}{|\vec{r}_i(t) - \vec{r}_j(t)|^3}, \tag{13}$$

which can be written as the differential equation:

$$\frac{d\vec{v}_i(t)}{dt} = -G\sum_{j\neq i} m_j \frac{\vec{r}_i(t) - \vec{r}_j(t)}{|\vec{r}_i(t) - \vec{r}_j(t)|^3}. \tag{14}$$

In addition we have the differential equation:

$$\frac{d\vec{r}_i(t)}{dt} = \vec{v}_i(t). \tag{15}$$

Solving (14) and (15) for every body $i = 1, ..., n$, allows us to find the internal motion of a given n-body system with only gravitational interaction. In this system we can also define total potential energy:

$$U_{\text{tot}} = \sum_i -Gm_i \sum_{j>i} \frac{m_j}{|\vec{r}_i - \vec{r}_j|}\,, \tag{16}$$

and the total kinetic energy:

$$K_{\text{tot}} = \sum_i \frac{1}{2} m_i \vec{v}_i^2 \tag{17}$$

The sum of the total kinetic and potential energy of the system should be constant (this will be proven in the next section). This can be used to test the algorithms.

We will also be experimenting with a slightly different form of the force in (10):

$$\vec{F}_a = -G\frac{m_a m_b}{|\vec{r}_a - \vec{r}_b|^{\beta+1}}(\vec{r}_a - \vec{r}_b)\,, \tag{18}$$

where $\beta \in [2, 3]$ is a real constant. Choosing $\beta = 2$ corresponds to the normal gravitational force, and choosing a β larger than this means that the force will be stronger for objects that are closer to each other and weaker for those that are far from each other. In a n-body system such as in (12) we have that the total force an object experiences is:

$$\vec{F}_i = -Gm_i \sum_{j\neq i} m_j \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^{\beta+1}}. \tag{19}$$

We can also write down the total potential energy in this case:

$$U_{\text{tot}} = \sum_i -\frac{Gm_i}{\beta - 1} \sum_{j>i} \frac{m_j}{|\vec{r}_i - \vec{r}_j|^{\beta-1}}\,, \tag{20}$$

which is found by using the relation between the force and the potential energy $\vec{F} = -\boldsymbol{\nabla} U$. The total kinetic energy remains the same in this case.

C. Conservation of energy

We have that the force $\vec{F}$ is related to the potential energy as follows:

$$\vec{F} = -\boldsymbol{\nabla} U \tag{21}$$

This will be useful later. That the total energy E is conserved can be written mathematically as:

$$\frac{dE}{dt} = 0$$
$$\frac{d}{dt}(K_{\text{tot}} + U_{\text{tot}}) = 0\,,$$

where we have used that the total energy is the sum of the total kinetic energy K_{tot} and the total potential energy U_{tot}. We now assume that we are in an n-body system, where the potential energy is velocity independent. The potential energy is determined by the interaction between the bodies and a potential time dependent part. We define $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ as the distance vector between the position of body i ($\vec{r}_i$) and body j ($\vec{r}_j$). This means that we can write the total potential energy as:

$$U_{\text{tot}} = \frac{1}{2}\sum_i \sum_{j\neq i} U(\vec{r}_{ij}, t)\,,$$

where U here is the potential energy experienced by body i because of body j. We know that $U(\vec{r}_{ij}, t) = U(\vec{r}_{ji}, t)$ is the same potential energy, and so as to avoid counting every part twice we add the factor $1/2$ to the equation. This also means that we can write the force experienced by body i from body j as:

$$\vec{F}_{ij} = -\boldsymbol{\nabla}_{ij} U(\vec{r}_{ij}, t)\,,$$

where $\boldsymbol{\nabla}_{ij}$ differentiates with respect to $\vec{r}_{ij}$. We can then write the time derivative of the potential energy as:

$$\begin{aligned}
\frac{d}{dt}U_{\text{tot}} &= \frac{1}{2}\sum_i \sum_{j\neq i} \frac{d}{dt} U(\vec{r}_{ij}, t)\,, \\
&= \frac{1}{2}\sum_i \sum_{j\neq i} \left(\boldsymbol{\nabla}_{ij} U(\vec{r}_{ij}, t) \cdot \frac{d\vec{r}_{ij}}{dt} + \frac{\partial}{\partial t} U(\vec{r}_{ij}, t) \right) \\
&= \frac{\partial}{\partial t} U_{\text{tot}} + \frac{1}{2}\sum_i \sum_{j\neq i} \boldsymbol{\nabla}_{ij} U(\vec{r}_{ij}, t) \cdot \frac{d\vec{r}_{ij}}{dt} \\
&= \frac{\partial}{\partial t} U_{\text{tot}} - \frac{1}{2}\sum_i \sum_{j\neq i} \vec{F}_{ij} \cdot \frac{d\vec{r}_{ij}}{dt}
\end{aligned} \tag{22}$$

The total kinetic energy can be given as the sum of the kinetic energy of each body:

$$K_{\text{tot}} = \sum_i \frac{1}{2} m \vec{v}_i^2$$

The time derivative of this is:

$$\begin{aligned}
\frac{d}{dt} K_{\text{tot}} &= \sum_i m \vec{v}_i \cdot \vec{a}_i \\
&= \sum_i \frac{d\vec{r}_i}{dt} \cdot \vec{F}_i \\
&= \frac{1}{2} \sum_i \sum_{i \neq j} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij} - \frac{1}{2} \sum_i \sum_{i \neq j} \frac{d\vec{r}_j}{dt} \cdot \vec{F}_{ij} \\
&= \frac{1}{2} \sum_i \sum_{i \neq j} \frac{d\vec{r}_{ij}}{dt} \cdot \vec{F}_{ij}
\end{aligned} \tag{23}$$

where $\vec{F}_i$ is the total force body i experiences given as $\vec{F}_i = \sum_{j \neq i} \vec{F}_{ij}$ as long as there are no external forces. The second to last equality is non-trivial, which means we need to prove it. We have that:

$$\sum_i \frac{d\vec{r}_i}{dt} \cdot \vec{F}_i = \sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij}$$

We can also recognize by using Newton's third law ($\vec{F}_{ij} = -\vec{F}_{ji}$) that:

$$\sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij} = -\sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ji}$$

By changing the index labels on the right-hand side we get:

$$\sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij} = -\sum_j \sum_{i \neq j} \frac{d\vec{r}_j}{dt} \cdot \vec{F}_{ij}$$

As the intention of the double summation is to sum over every combination of i and j where $i \neq j$ it is arbitrary in which order we sum over the variables, as long as the inequality is kept as part of the inner sum:

$$\sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij} = -\sum_i \sum_{j \neq i} \frac{d\vec{r}_j}{dt} \cdot \vec{F}_{ij}$$

Putting this all together, we can write:

$$\begin{aligned}
\sum_i \frac{d\vec{r}_i}{dt} \cdot \vec{F}_i &= \sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij} \\
&= \frac{1}{2} \sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij} + \frac{1}{2} \sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij} \\
&= \frac{1}{2} \sum_i \sum_{j \neq i} \frac{d\vec{r}_i}{dt} \cdot \vec{F}_{ij} - \frac{1}{2} \sum_i \sum_{j \neq i} \frac{d\vec{r}_j}{dt} \cdot \vec{F}_{ij} \,,
\end{aligned} \tag{24}$$

which is the relation we wanted to show. Our requirement for the energy to be conserved was that:

$$\frac{d}{dt}(K_{\text{tot}} + U_{\text{tot}}) = 0$$

Using (22) and (23) in this gives us:

$$\begin{aligned}
\frac{1}{2} \sum_i \sum_{i \neq j} \frac{d\vec{r}_{ij}}{dt} \cdot \vec{F}_{ij} + \frac{\partial}{\partial t} U_{\text{tot}} - \frac{1}{2} \sum_i \sum_{j \neq i} \vec{F}_{ij} \cdot \frac{d\vec{r}_{ij}}{dt} = 0 \\
\frac{\partial}{\partial t} U_{\text{tot}} = 0 \\
\frac{1}{2} \sum_i \sum_{j \neq i} \frac{\partial}{\partial t} U(\vec{r}_{ij}, t) = 0
\end{aligned} \tag{25}$$

In other words for an n-body system to have conserved energy we need the potential energy to be independent of time. Another assumption made along the way was that there are no external forces, which means that this is only for a closed system. This proves that total energy is conserved both in n-body systems where the force interaction is given by (12) or (19) as there are no external forces and the potential energies ((16) and (20) respectively) are time independent.

D. Conservation of angular momentum

Kepler's second law states that the position vector between a planet and the Sun sweeps out a constant area for a given interval of time, no matter the starting position of the planet in its orbit. Mathematically this can be formulated as follows:

$$\frac{dA}{dt} = \frac{r^2}{2} \frac{d\theta}{dt} \,, \tag{26}$$

where A is the area swept by the position vector, r is the length of the position vector and θ is the angle determining the direction of the position vector. It is implicit here that we are looking at the 2D plane that the planet orbits the Sun in as the xy-plane, as this simplifies the equation significantly.

The angular momentum is given by:

$$\ell = |\vec{r} \times \vec{p}| \,, \tag{27}$$

where $\vec{r}$ is the position vector, and $\vec{p} = m\vec{v}$ is the momentum vector with $\vec{v}$ being the velocity vector and m being the mass. Now we can make some assumptions. Assuming that the position vector rotates in the xy-plane the only contribution to the cross product comes from the tangential component of the velocity v_θ. If we assume the tangential velocity to be in positive θ-direction

(we can always transform the chosen plane so that this is the case) we get:

$$\ell = mrv_\theta \tag{28}$$

Using that $\vec{v} = \frac{d}{dt}\vec{r}$ we can write:

$$v_\theta = r\frac{d\theta}{dt}$$

We can insert this along with (28) into (26):

$$\begin{aligned}\frac{dA}{dt} &= \frac{r^2}{2}\frac{d\theta}{dt} \\ \frac{dA}{dt} &= \frac{r}{2}v_\theta \\ \frac{dA}{dt} &= \frac{1}{2}\frac{\ell}{m}\end{aligned} \tag{29}$$

Since we already know that the left-hand side of this equation is constant this directly implies that the size of the angular momentum is a conserved quantity as we also assume the mass to be constant.

For a system of n bodies such as in the previous sections we can define the total angular momentum of a system as:

$$\vec{L}_{\text{tot}} = \sum_i \vec{r}_i \times m\vec{v}_i \tag{30}$$

This quantity should be conserved as long as there are no external forces, and the forces are conservative. We can prove that this is the case for both the forces we use in this report ((12) and (19)). That the angular momentum is conserved means that:

$$\frac{d}{dt}\vec{L} = 0\,,$$

where 0 here corresponds to the zero vector technically. We find that the left-hand side of this can be written as:

$$\begin{aligned}\frac{d}{dt}\vec{L} &= \frac{d}{dt}(\sum_i \vec{r}_i \times m\vec{v}_i) \\ &= \sum_i \vec{v}_i \times m\vec{v}_i + \vec{r}_i \times m\vec{a}_i \\ &= \sum_i \vec{r}_i \times \vec{F}_i\,,\end{aligned} \tag{31}$$

where $\vec{F}_i$ is the force on body i, $\vec{a}_i$ is the acceleration on body i and all other quantities are as previously defined. The cross product of a vector with itself is the zero vector, so that is why the $\vec{v} \times \vec{v}$ term disappeared. Now we can use a result similar to the one found in (24):

$$\begin{aligned}\sum_i \vec{r}_i \times \vec{F}_i &= \sum_i \sum_{j\neq i} \vec{r}_i \times \vec{F}_{ij} \\ &= \frac{1}{2}\sum_i \sum_{j\neq i} \vec{r}_i \times \vec{F}_{ij} + \frac{1}{2}\sum_i \sum_{j\neq i} \vec{r}_i \times \vec{F}_{ij} \\ &= \frac{1}{2}\sum_i \sum_{j\neq i} \vec{r}_i \times \vec{F}_{ij} - \frac{1}{2}\sum_i \sum_{j\neq i} \vec{r}_j \times \vec{F}_{ij} \\ &= \frac{1}{2}\sum_i \sum_{j\neq i} \vec{r}_{ij} \times \vec{F}_{ij}\,,\end{aligned} \tag{32}$$

where $\vec{r}_{ij}$ is the distance vector between body i and j. In both the forces we use in this report ((12) and (19)) the force is in the direction of $\vec{r}_{ij}$. This means we can write $\vec{F}_{ij} = f_{ij}\vec{r}_{ij}$ where f_{ij} is a scalar. Using this along with (32) in (31) gives:

$$\begin{aligned}\frac{d}{dt}\vec{L} &= \sum_i \vec{r}_i \times \vec{F}_i \\ &= \frac{1}{2}\sum_i \sum_{j\neq i} \vec{r}_{ij} \times \vec{F}_{ij} \\ &= \frac{1}{2}\sum_i \sum_{j\neq i} \vec{r}_{ij} \times f_{ij}\vec{r}_{ij} \\ &= 0\,,\end{aligned} \tag{33}$$

where we again used that the cross-product of a vector with itself is the zero-vector. This shows that the angular momentum is conserved.

E. Escape velocity of object in gravitational field

An object has a velocity large enough to escape a gravitational field when the total energy of the object is larger than zero. In other words we need the velocity v to fulfill the following requirement:

$$\begin{aligned}K + U &\geq 0 \\ \frac{1}{2}mv^2 - \frac{GMm}{r} &\geq 0 \\ v &\geq \sqrt{\frac{2GM}{r}}\,,\end{aligned}$$

where K is the kinetic energy, U the potential energy, G the gravitational constant, m the mass of the object, M the mass of the object causing the gravitational field, r the distance between them, and v the size of the velocity of the object relative to the object causing the field. The escape velocity v_e is the lowest velocity that allows escape in this way, and is thus given by the formula:

$$v_e = \sqrt{\frac{2GM}{r}} \tag{34}$$

If we consider a planet at a distance of 1 AU from the Sun, we can calculate its escape velocity as

$$v_{sun,1} = \sqrt{\frac{2 \cdot 4\pi^2 \cdot 1}{1}} \text{ AU/year},$$

using the Sun's mass as our unit mass, and $G = 4\pi^2$. Thus we get

$$v_{sun,1} = 2\pi\sqrt{2} \text{ AU/year}, \tag{35}$$

as the escape velocity of a planet starting at a distance of 1 AU from the Sun.

F. Perihelion precession of Mercury

The observed precession of Mercury's perihelion is about 43" per century when all contributions from classical effects are subtracted. By adding a relativistic correction term (as per [1, p. 5]) we can verify that we obtain the same result. The gravitational force between Mercury and the Sun with the relativistic correction term is as follows:

$$F_G = \frac{GM_{\text{Mercury}}M_{\text{Sun}}}{r^2}\left[1 + \frac{3l^2}{r^2c^2}\right], \tag{36}$$

where M_{Mercury} is the mass of Mercury, M_{Sun} the mass of the Sun, r the distance between Mercury and the Sun, c the speed of light, and l the size of the angular momentum per unit mass.

If we simulate the orbit of the Mercury around the Sun as an isolated system, using (36) as the force that the Sun exerts on Mercury, we can use the resulting data to find the perihelion precession of Mercury. This can be compared with the measured precession due to relativistic effects listed earlier to test the accuracy of our numerical solution in this case.

To measure the precession of Mercury's perihelion we can use the following process. First we initialize a system with only Mercury and the Sun. We choose the reference frame of the Sun with Mercury orbiting in the xy-plane, as this means we only have to calculate the force that the Sun exerts on Mercury and reduces the problem from a three-dimensional one to a two-dimensional one. In order to simplify some calculations earlier we also initialize Mercury along the x-axis in its perihelion, and with initial velocity in positive orientation. Secondly we simulate the system for a century (simulation time). Thirdly we process the data and identify the first and last positions of the perihelion. Now we can calculate the precession of Mercury's perihelion. Since our system is generally in cartesian coordinates, we can find the angular position as:

$$\theta_p = \arctan\left(\frac{x_p}{y_p}\right), \tag{37}$$

where θ_p is the angular position of the perihelion, and x_p and y_p are the cartesian coordinates of the position of the perihelion.

Given the first and last position of the perihelion $\theta_{p,1}$ and $\theta_{p,2}$ with the second coming after the first, we define the change in the angular position as $\Delta\theta_p = \theta_{p,2} - \theta_{p,1}$. The precession per century is approximately equal to this quantity, as we have simulated for a century. In other words the precession p is:

$$p = \Delta\theta_p \tag{38}$$

G. Center of mass

In an n-body system, where body i has mass m_i, position $\vec{r}_i$, and velocity $\vec{v}_i$ we have that the position of the center of mass $\vec{R}$ is:

$$\vec{R} = \frac{1}{M}\sum_i m_i\vec{r}_i\,, \tag{39}$$

where:

$$M = \sum_i m_i \tag{40}$$

We can also find the velocity of the center of mass $\vec{V}$:

$$\vec{V} = \frac{1}{M}\sum_i m_i\vec{v}_i \tag{41}$$

We can always transform to a reference frame where the origin is placed at the center of mass, and all the velocities of the objects making up the system are relative to the velocity of the center of mass. If there are no external forces the center of mass reference frame does not change, and is thus ideal when simulating systems such as the ones we are modelling. To transform our reference frame into the center of mass reference frame it is enough to subtract the center of mass position from the positions of all the bodies, and the center of mass velocity from the velocity of all the bodies:

$$\begin{aligned}\vec{r}_i' &= \vec{r}_i - \vec{R}\\ \vec{v}_i' &= \vec{v}_i - \vec{V}\,,\end{aligned} \tag{42}$$

where $\vec{r}_i'$ is the position of body i in the center of mass reference frame, and $\vec{v}_i$ the velocity of body i in the center of mass reference frame.

This only works in the classical limit, as when we are considering relativistic effects we also have to consider that the frame may be accelerated due to the center of mass potentially having a non-zero velocity.

III. METHOD

A. Implementation

In this section we will discuss the layout of the code and what each part does. As there is a lot of code we will only be including some snippets of code directly, and in some instances we will make references to functions and variables not shown in the code snippets. In these cases the reader should look to the source code if he or she wishes to know more. Links to the source code can be found in appendix A. The units chosen can be found Table 1.

Table I. Table containing the units used in the programs. See section III.A for more details on these programs.

	Unit
Length	AU
Time	Year
Mass	Mass of the Sun

We have written a set of object oriented programs in C++ that together simulate a set of point objects with a gravitational interaction. We use these programs to simulate the motion of objects in the solar system. In order to better interface with these programs we have automated running them using a Python script, which we will get back to once we have explained the general layout and implementation of the algorithms.

1. *A class to store celestial bodies as objects*

The most basic part of the program is the `CelestialBody` class. The `CelestialBody` object is used to store the position, velocity, mass, and force an object is experiencing. No calculation in general is done in this class. See appendix A.5 for the full code. The vectors (position, velocity and force) are stored as Armadillo [2] vectors, as these have built-in functionality for the mathematical operations we need.

2. *A class to store a set of celestial bodies and perform calculations on them*

The level above the `CelestialBody` class is the `SolarSystem` class. This class creates `CelestialBody` objects and stores them in the standard library vector object `m_bodies`. In this class we also calculate the forces that each object feels with one of two methods. The first one, called `calculateForcesAndEnergy()`, is as follows:

```
void SolarSystem::calculateForcesAndEnergy() {
  // Clear variables
  m_kinetic_energy = 0;
  m_potential_energy = 0;
  m_angular_momentum = arma::zeros(3);
  for (CelestialBody &body: m_bodies) {
    body.force.zeros();
  }

  // Iterate over every body (body1)
  for (int i = 0; i<numberOfBodies();++i){
    CelestialBody &body1 = m_bodies[i];
    // Second iteration over every body after body1 (body2)
    for (int j = i+1; j<numberOfBodies();++j) {
      CelestialBody &body2 = m_bodies[j];

      // Distance vector
      arma::vec dr_vec = body2.position - body1.position;

      // Length of distance vector
      double dr = arma::norm(dr_vec);

      // Potential energy between body1 and body2
      double potential_energy = m_G*body1.mass*body2.mass
                                /std::pow(dr,m_beta_1);

      // Force vector between the bodies (sign adjusted when
      // adding to the bodies)
      arma::vec gravforce = dr_vec*potential_energy/(dr*dr);
      body1.force += gravforce;
      body2.force -= gravforce;

      // Adding potential energy to total
      m_potential_energy -= potential_energy/m_beta_1;
    }

    // Adding kinetic energy of body1 to total
    double v = arma::norm(body1.velocity);
    m_kinetic_energy += 0.5*body1.mass*v*v;

    // Adding angular momentum of body1 to total angular momentum
    m_angular_momentum += arma::cross(body1.position,body1.mass
                                      *body1.velocity);
  }
}
```

The method uses (19) to calculate the force an object experiences, (20) to calculate the total potential energy, (17) to calculate total kinetic energy, and (30) to calculate the total angular momentum. All are stored in internal variables in the class, and can be retrieved using methods built into the class. The variable `m_beta_1` is the parameter β (from (19)) minus 1, and is set upon instantiating the class, which we will get back to.

The second method is `calculateForcesWithRelativisticCorrection()`:

```
void SolarSystem::calculateForcesWithRelativisticCorrection() {
  // Clear variables
  for (CelestialBody &body: m_bodies) {
    body.force.zeros();
  }

  // Load Mercury (body2)
  CelestialBody &body2 = m_bodies[1];

  // Distance vector
  arma::vec dr_vec = body2.position;

  // Length of distance vector
  double dr = arma::norm(dr_vec);
  double dr2 = dr*dr;

  // Potential energy between body1 and body2
  double potential_energy = m_G*body2.mass/dr;

  // Calculate force vector with relativistic correction factor
  // (sign adjusted when adding to the bodies)
  arma::vec gravforce = dr_vec*(potential_energy/dr2)
                        * (1 + m_rel_constant*l2/dr2 );
  body2.force -= gravforce;
}
```

This is a specialized method, and should only be used in a two-body system where one of the bodies is the

Sun. The force from the Sun onto the second body is calculated using (36), and we assume the reference frame to be chosen such that the Sun is at rest. As we have no expression for what the potential energy is in this case we have chosen to omit all calculation of potential energy. The magnitude of the angular momentum per unit mass is used as a constant in the relativistic correction term for the gravitational force, and needs to be precalculated by manually calling the `calcAngMom()` method of the `SolarSystem` class. Both this method and `calculateForcesWithRelativisticCorrection()` are mainly used in estimating the perihelion precession of Mercury, as outlined in section II.F.

There are four ways to instantiate the `SolarSystem` class. The first constructor takes no arguments. In this case the internal variables for potential and kinetic energy are set to 0, the angular momentum set to the zero-vector, and `m_beta` (the internal variable for β in (18)) is set to 2. The second constructor takes one double as an argument and `m_beta` is set to be equal to this double. Otherwise this functions similarly to the first constructor. When instantiated in both these ways the `CelestialBody` objects that make up the system must be instantiated manually using the `createCelestialBody()` method. This method takes two Armadillo [2] three-component vectors and a double as input (in that order). The first vector should contain the initial position of the object instantiated, the second should contain the velocity, and the double should be the mass of the object. The base units of these quantities should be as in Table 1.

The third way to instantiate the class is with a string as an argument. The string is taken to be the path leading to a file with initial conditions for the system. Each line in such a file should contain first the three cartesian coordinates (x,y,z) of an object, then the three components of the velocity (v_x,v_y,v_z), and then lastly the mass of the object, all with a single whitespace between them. The file can be of arbitrary length as long as it is formatted correctly. This constructor first calls the first constructor, then instantiates all the objects in the file using the `createCelestialBody()` method. The fourth way of instantiating class works very similarly to the third one, in that it takes the path to a file of initial conditions as an argument, but now it also takes a double, which is taken to be `m_beta`. It functions in exactly the same way as the third one, except that instead of calling the first constructor, it calls the second one so that `m_beta` is stored correctly.

The class also contains the `moveToCOFMFrame()` method. As the name says, this transforms the coordinates and velocities of all the objects of the system so that the origin is the center of mass in the simulation. The method uses (39), (41) and (42) to perform the transformation. In general it is recommended to call this method before starting the simulation. If we later want to see things in terms of the reference system of the Sun or any other body for that matter we can transform to these reference systems when processing the data after running the simulation.

There are also some methods that serve as ways to retrieve private variables from the class. We list them all here. The `bodies()` method returns the standard library vector containing the `CelestialBody` objects, `m_bodies`, `potentialEnergy()` and `kineticEnergy()` returns the total potential and kinetic energy of the system respectively (these are zero if `calculateForcesAndEnergy()` has not been ran yet), `angularMomentum()` returns the total angular momentum vector (this is the zero-vector if `calculateForcesAndEnergy()` has not been ran yet), and `numberOfBodies()` which returns the amount of `CelestialBody` objects currently stored in `m_bodies`.

The three last methods of the class that we want to cover are `initiateDataFile()`, `writeToFile()` and `writeEnergyToFile()`. These are used together to write data generated during the simulation to files. The `initiateDataFile()` is overloaded so that it takes either one or two strings as arguments. If one is given, it is taken to be the path leading to the file that the positions of all the objects should be written to. If two are given, the first one is taken to be the path leading to the file that the positions of the objects should be written to, and the second is taken to be the path leading to the file that potential energy, kinetic energy and angular momentum should be written to.

The `writeToFile()` method takes no arguments, but any time it is called, the current positions of all the objects are written on a new line in the file. Each line contains first the x, y and z coordinates of the first object (in that order), then the same coordinates for the second object, and so on until all the positions have been written to the file specified with `initiateDataFile()`. All the coordinates are separated by a single whitespace. If `initiateDataFile()` has not been ran, this method will not work.

The `writeEnergyToFile()` method takes no arguments, but any time it is called, the current kinetic energy, potential energy, and the components of the angular momentum are written to a new line (in that order, separated by a single whitespace) in the datafile specified with `initiateDataFile()`. If `initiateDataFile()` has not been ran, or if it has only been ran with one argument this method will not work.

3. *A class used to integrate the system*

The next class we need to talk about is the `solar_integrator` class. When instantiating this class, two arguments are required: a double containing the timestep to be used for the simulation, and a string specifying which integration method should be used. Currently there are two methods of integration to choose

from: the forward Euler (3) and the velocity Verlet (9) methods. We will get back to how these are implmented once we have outlined the general structure of the class.

The main method of the class is the `integrateOneStep()` method. This takes a `SolarSystem` object as an argument and moves the system one timestep forward. Depending on what was specified when instantiating the class, this uses either the forward Euler method or the velocity Verlet algorithm. The method is also overloaded so that it may take a string as an argument as well. In this case the string is a flag that tells the integrator to use either `calculateForcesAndEnergy()` or `calculateForcesWithRelativisticCorrection()` to get the forces each body in the system is experiencing.

The forward Euler method is implemented as follows:

```
void solar_integrator::Euler(SolarSystem& system,
                             std::string relOrNonRel) {
  // Calculate forces and energy
  if (relOrNonRel == "rel") {
    system.calculateForcesWithRelativisticCorrection();
  } else if (relOrNonRel == "nonrel") {
    system.calculateForcesAndEnergy();
  }

  // Integrate every body one step
  for (CelestialBody &body: system.bodies()) {
    body.position += body.velocity*m_dt;
    body.velocity += m_dt * body.force / body.mass;
  }
}
```

The string `relOrNonRel` is the flag that tells which method the integrator should use to calculate the forces. The main part of the implmentation is in the for loop. We loop over all the bodies stored in the class, and update their positions and velocities using (3) on the two differential equations outlined in (14) and (15). This is done in a vectorized fashion, as Armadillo [2] vectors supports these kinds of mathematical operations on vectors.

The velocity Verlet algorithm is implemented as follows:

```
void solar_integrator::VelocityVerlet(SolarSystem& system,
                                      std::string relOrNonRel) {
  // Calculate forces and energy
  if (relOrNonRel == "rel") {
    system.calculateForcesWithRelativisticCorrection();
  } else if (relOrNonRel == "nonrel") {
    system.calculateForcesAndEnergy();
  }

  // Initializing matrix to store the acceleration in current
  // timestep
  arma::mat prev_acceleration(3,system.numberOfBodies());

  // Integrates the position of the bodies one step
  int i = 0;
  for (CelestialBody &body: system.bodies()) {
    prev_acceleration.col(i) = body.force / body.mass;
    body.position += m_dt * body.velocity
                   + m_dt_sqr_2 * prev_acceleration.col(i);
    ++i;
  }

  // Calculate forces and energy again (to get acceleration in
  // next timestep)
  if (relOrNonRel == "rel") {
    system.calculateForcesWithRelativisticCorrection();
  } else if (relOrNonRel == "nonrel") {
    system.calculateForcesAndEnergy();
  }

  // Integrates the velocity of the bodies one step using the
  // acceleration in the current and the next timestep
  i = 0;
  for (CelestialBody &body: system.bodies()) {
    body.velocity += m_dt_2*( prev_acceleration.col(i)
                   + body.force / body.mass);
    ++i;
  }
}
```

The `relOrNonRel` flag works in the same way as in the forward Euler method. Note that the key difference here is that there are two loops instead of one. As seen in (9) we need to first update the position in order to update the velocity, as the velocity requires the acceleration in both the current timestep and the one we are integrating to. The acceleration is given by the force (13), and so we need to calculate the force again before we can calculate the velocity. In order to improve the efficiency of the code it is thus better to split the integration into two loops and call the method to update the forces once in between those loops. We also need to store the acceleration in the current timestep, as calling the methods that update the forces overwrite the forces of the current timestep stored in the `SolarSystem` object we are integrating. Thus we initialize and use the Armadillo [2] matrix `prev_acceleration`. The second loop then uses the acceleration in the current timestep, and the next, to update the velocity of each `CelestialBody` object stored in the `SolarSystem` object we are integrating. The variable `m_dt_sqr_2` used when updating the position is the timestep squared and divided by two. Similarly the variable `m_dt_2` is the timestep divided by two. These calculations are done when instantiating the class in order to reduce the amount of FLOPs. While the compiler should be able to move these calculations out of the loops automatically when using flags for optimizing, we have chosen to do it explicitly where we can.

The `Euler()` and `VelocityVerlet()` methods shown in the previous two code snippets cannot be called directly as they are private methods of the class. Instead the `integrateOneStep()` method should be called, as it calls either of these methods depending on what was specified when instantiating the class.

4. A main program used to interface with the classes

Although at this point these classes can be instanced directly we have also automated interfacing with them in `main.cpp` (see section A.2). When running the compiled version of this script it takes a total of ten command line arguments, where all of them have default values defined so that they are technically optional. We will discuss them all in order. The first command line argument is taken to be an integer and is the number of timesteps for which to simulate, and the default value is 1000. The second command line argument is taken to be a double and should be the timestep in years, and the default value is 0.001. The third command line argument should be an

integer. This integer is used as a way to limit the size of the resulting data files as it modifies the simulation such that it will only write to file every k steps where k is the integer supplied. The default value of this is 1.

The fourth command line argument is taken to be a string, which should specify which integration method that is to be used. It should only be either `VelocityVerlet` or `Euler` (no quotation marks, it is automatically stored as a single string as long as there are no whitespaces). By default this is set to `VelocityVerlet`. The fifth command line argument should be a string giving the path to a file containing initial conditions for the system to be simulated (see the paragraph on the third and fourth way of instantiating the `SolarSystem` class for information about this file). The default value here is `../data/sun-earth.init`. This assumes that there is a folder called data one level above the current folder containing the file `sun-earth.init`.

The sixth command line argument is taken to be a string containing the path to the file in which to store the positions of the objects from the simulation in (see the paragraph on the `writeToFile()` method for information about this file). The default value here is `../data/positions.xyz`, which assumes that there exists a folder one level above the current one called data. The seventh command line argument is taken to be a string containing the path to the file in which the kinetic and potential energy and the angular momentum should be stored in (see the paragraph on the `writeEnergyToFile()` method for more information about this file). The default value here is `../data/energies.dat`, which again assumes that there exists a folder one level above the current one called data.

The eigth command line argument is also taken to be a string. This string is a flag that tells whether to use `calculateForcesAndEnergy()` or `calculateForcesWithRelativicsticCorrection()` should be called to calculate the forces. Using `nonrel` correposnds to the first one and `rel` corresponds to the second one. The default value here is `nonrel`. The ninth command line argument is taken to be a double equal to the factor β in (18) and (20). The default value of this is 2. The tenth and final command line argument is taken to be a string. If it is specified to be `sm` it will run a specialized simulation which is intended to be used to calculate the precession of Mercurys perihelion per century.

5. *A Python script automating some simulations*

Now we have outlined the main structure of the C++ parts of our implementation. We admit that there may be more information than necessary given here, but we want to be precise in our discussion of the implementation as this will allow others to reuse our code with more ease. We have written a Python script, `project.py`, which can be used to run all the simulations used in this report in a way that is easier to interface with. All the program files and files containing initial conditions can be found in the GitHub repository linked to in appendix A. When running `project.py` in a terminal it will prompt the user for several inputs, outlining what they may be (we will not outline all the options specifically here, see the README.md file in the repository for specifics). There are some systems that we have created files with initial conditions for and these are listed as options when running the Python script. If these files themselves are of interest they can be found in the data folder in the GitHub repository. We have also implemented a secondary program used to benchmark the algorithms, which we cover in section III.E.

B. Floating point operations

It is of interest to count the amount of FLOPs (floating point operations) needed to perform the algorithm. We denote the amount of timesteps in the simulations N, and the amount of bodies in the system k. There are several options that we may choose from which impacts the total amount of FLOPs. When using the `calculateForcesAndEnergy()` method to find the forces we need $31((k-1)!) + 18k$ FLOPs. Some of these will likely be culled by the compiler when compiled with optimization flags. We do not round this off immediately as we don't use very large k in our simulations. This means that the $18k$ term is not necessarily dominated by the $31((k-1)!)$ term. When using the `calculateForcesWithRelativisticCorrection()` method to find the forces we need 30 FLOPs. This is independent of k as this method assumes a specific system (the system with only the Sun and Mercury).

When using the forward Euler method we can see that there are $15k$ FLOPs in the loop. When `calculateForcesAndEnergy()` is also called, this adds another $31((k-1)!) + 18k$ FLOPs for a total of $31((k-1)!) + 33k$ FLOPs per iteration. To complete a full simulation with these options we would need $(31((k-1)!) + 33k)N$ FLOPs. When the `calculateForcesWithRelativisticCorrection()` method is called this adds another 30 FLOPs so that the total amount of FLOPs needed are $15k + 30$ per iteration. To complete a full simulation with these options we would need $(15k + 30)N$ FLOPs.

When using the `VelocityVerlet()` method a total of $27k$ FLOPs are needed in this method itself. It also calls either of the force calculation methods twice. If `calculateForcesAndEnergy()` is used this adds another $62((k-1)!) + 36k$ FLOPs for a total of $62((k-1)!) + 63k$ FLOPs per iteration. To complete a full simulation with these options we would need $(62((k-1)!) + 63k)N$ FLOPs. When `calculateForcesWithRelativisticCorrection()` is used this adds another 60 FLOPs for a total of $27k + 60$

FLOPs per iteration. To complete a full simulation with these options we would need $(27k + 60)N$ FLOPs.

All of these values are listed in table II for ease of access.

Table II. Table showing the amount of floating point operations needed to complete a simulation with k bodies and N steps for four sets of options. "Non-relativistic" and "relativistic" refers to the calculation of the forces in the system. What these options are is detailed in section III.A.

	Forward Euler	Velocity Verlet
Non-relativistic	$(31((k-1)!) + 33k)N$	$(62((k-1)!) + 63k)N$
Relativistic	$(15k + 30)N$	$(27k + 60)N$

C. Error sources

The sources of errors in these simulations are quite simple to get a grasp of as there are only two major ones, aside from potential errors in the implementation. These are the errors made from approximation when deriving the numerical methods of integration used, and machine precision.

When using the forward Euler method there is an error $\mathcal{O}(\Delta t^2)$ (where Δt is the timestep) from the approximation made when deriving the method, and when using the velocity Verlet algorithm there is an error $\mathcal{O}(\Delta t^3)$. Thus we expect the forward Euler method to be less precise than the velocity Verlet algorithm.

It is also important to note that, analytically, the total energy of the system should be a conserved quantity. The forward Euler method does not conserve energy, but the velocity Verlet algorithm does (in theory). This means that we should see a small drift in the total energy when using the Euler method that we will not see when using the velocity Verlet algorithm. This is a simple criterion that we can use the verify that our implementation of the velocity Verlet algorithm works as intended. While this is true in theory it is not always true in practice, as machine precision also plays a part. Due to this we will also probably see a drift (or more likely an oscillation) in the total energy when using the velocity Verlet algorithm. The size of this drift or oscillation can be used to tell us whether or not the chosen timestep is a good one, as it tells us something about the machine error when using the velocity Verlet method, and something about the total error when using the forward Euler method.

The total angular momentum is also a conserved quantity (see section II.C) in theory. While there are errors for this as well from the same sources as for the energy, this gives another easy criterion that we can use to determine whether our implementation works as it should, as there should be little to no drift with the velocity Verlet algorithm and only a small drift with the forward Euler method.

D. Unit-tests

To test our implementations of the forward Euler and velocity Verlet methods, we implemented 4 unit-tests in `test_functions.cpp` (A.7) using the CATCH2 unit-test framework. The tests are specifically written to test the energy conservation and angular momentum conservation of both the velocity Verlet and forward Euler methods respectivly. The tests do this by setting up the SolarSystem and solar_integrator classes for an approximated Sun-Earth system, where the Sun is set to the origin with no initial velocity, and Earth is set to $\vec{r}_0 = (1, 0, 0)$ and $\vec{v}_0 = (0, 2\pi, 0)$ as initial position and velocity (units AU and AU/year). Earth is assumed to have a mass of $3 \times 10^{-6} M_{\text{Sun}}$. All the tests then integrate the given system for $N = 10^4$ time steps, with a step length of $dt = 1 \times 10^{-3}$ years. The energy conservation tests stores the total energy of the system at the first and last timestep, then compares the two energies. Similarly, the angular momentum tests integrates with the same number of time steps and step length as the energy tests. The angular momentum tests then compare both the magnitude of the angular momentum for the first and final timestep, and the individual components of the angular momentum. We expect these tests to pass for the velocity Verlet method, but not the forward Euler method.

E. Benchmark

To time our implementations of the forward Euler and velocity Verlet methods. The source code for the benchmark can be found in `benchmark.cpp` (A.8). The benchmark sets up the SolarSystem and solar_integrator classes for the file containing initial conditions for the Sun, all eight planets and Pluto. The benchmark integrates the given system for $N = 10^6$ time steps and a step length of $dt = 2.48 \times 10^{-4}$ s. The integration is timed, and performed 50 times for both the velocity Verlet and forward Euler methods. The benchmark writes the results to file. Running `project.py` (A.1) with the benchmark keyword will write the mean result, with standard deviation, to a file `benchmark_sun_and_friends.dat` in the data directory.

The benchmark only measures time spent on integration. Time spent on writing position data to file is not accounted for, but is expected to significantly slow down the integration loops, as storage media is significantly slower than RAM. We choose to omit timing storage operations, as these are not relevant to the subject of interest in this report, which is numerical integration.

IV. RESULTS

A. Benchmark results

We ran 50 simulations of the Solar System (more specifically the Sun, the planets and Pluto) using both the forward Euler and velocity Verlet methods of numerical integration. In these runs we measured the amount of time the simulations took, which will allow us to compare the performance of the two methods. The mean time spent for both, and the standard deviation in these measurements are listed in Table III.

Table III. Table showing the mean time and standard deviation (σ) of 50 runs integrating our Solar System for $N = 10^6$ time steps and a step length of $dt = 2.48 \times 10^{-4}$ years using the forward Euler and velocity Verlet methods (see III.E for more details).

Method	Mean time [seconds]	σ [seconds]
Forward Euler	1.8331 s	4.2967×10^{-3} s
Velocity Verlet	3.6346 s	1.5126×10^{-2} s

B. Simulations of Earth and the Sun for a few selected timesteps

In order to test the convergence of our solvers we ran simulations of Earth and the Sun with three choices of timestep, $\Delta t_1 = 10^{-3}$, $\Delta t_2 = 10^{-4}$ and $\Delta t_3 = 10^{-5}$. The amount of timesteps was chosen so that the simulations span 1 year in simulation time. These simulations were all ran twice, one using the forward Euler method of integration, and the other using the velocity Verlet method. Plots of Earth's orbit around the Sun in 2D for the forward Euler method are shown in figure 1. Similar plots for the velocity Verlet method are shown in figure 2. We also measured the standard deviation of the total energy in these simulations and they are listed as percentages of the mean total energy in table IV. We also include plots of the magnitude of the total angular momentum in the simulations with timestep $\Delta t = 10^{-5}$ in figure 3.

As for the initial positions we start with the Sun in the origin and Earth at a distance 1 AU along the x-axis away from the Sun. The average distance between Earth and the Sun is 1 AU so this mimics real life. Earth makes a full orbit around the Sun in a year, so in order for the orbit to circular we need the initial velocity to be tangential the position, and we need it to be such that it can make a full orbit in one year. The circumference of a circle with radius 1 is 2π so we need the velocity of Earth to be 2π in the y-direction in order for the orbit to be circular. These are the initial conditions we have chosen.

The simulations were all ran in three dimensions, but there is no motion in the z-direction because of the choice of initial conditions, and thus we only show 2D plots of the orbits. They were also ran in the center of mass frame (see section II.G) and transformed back to reference frame of the Sun when processing the data.

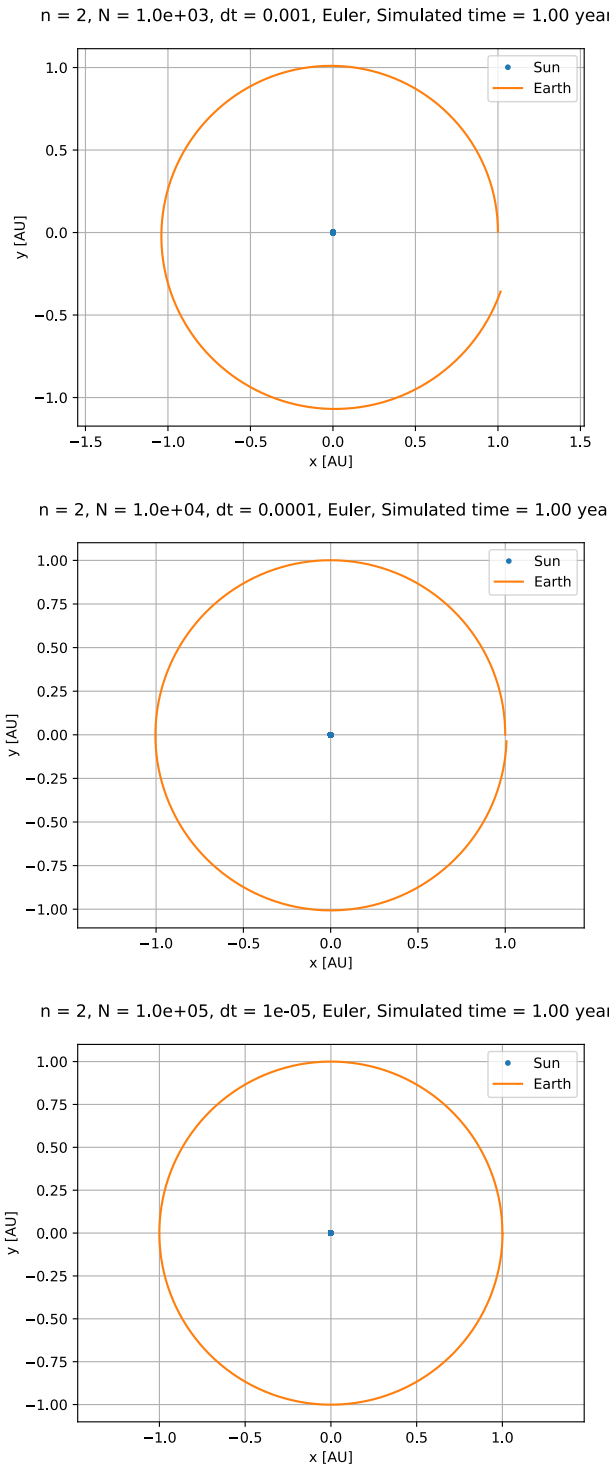


Figure 1. This figure contains plots of simulated orbits of Earth around the Sun for three different timesteps (denoted dt in the plots). These three simulations used the forward Euler method of integration (see section II for details on the method and section III for details on the simulation). In the title of the plot some variables are listed: n is the number of bodies, N the number of timesteps, dt the timestep, and the rest is self-explanatory.

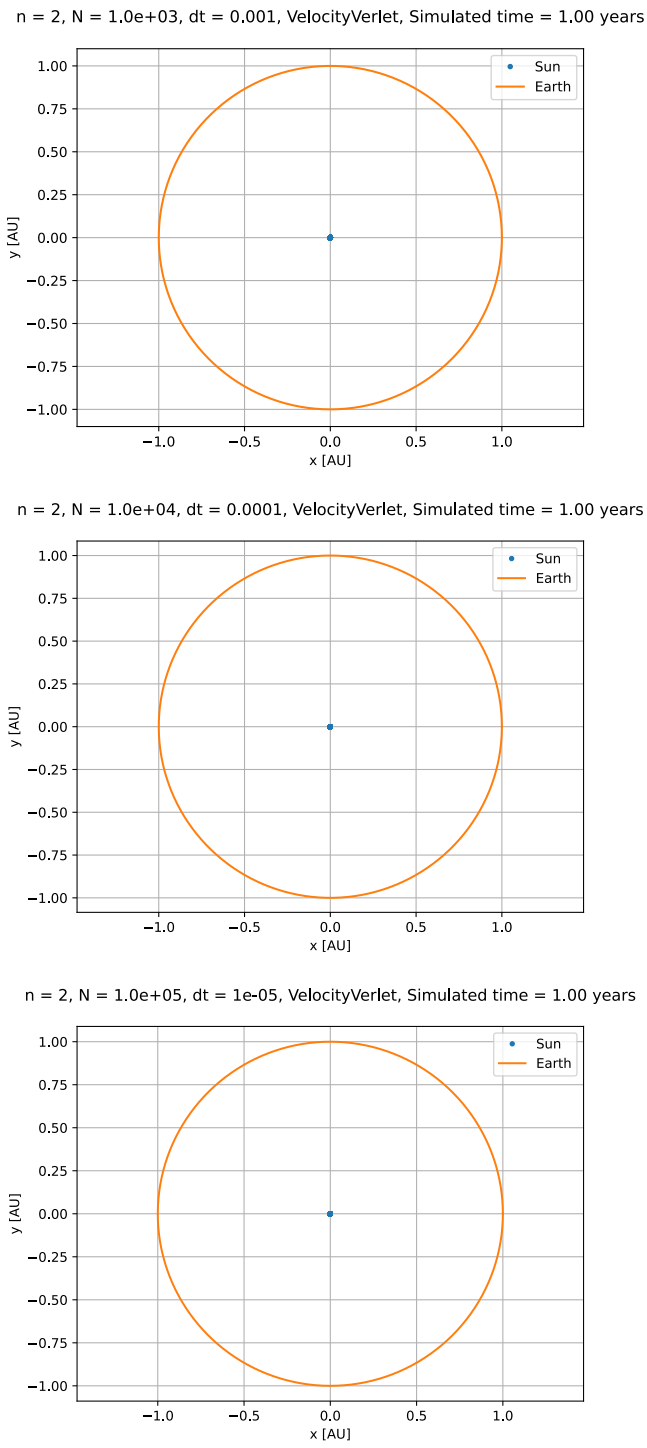


Figure 2. This figure contains plots of simulated orbits of Earth around the Sun for three different timesteps (denoted dt in the plots). These three simulations used the velocity Verlet method of integration (see section II for details on the method and section III for details on the simulation). In the title of the plot some variables are listed: n is the number of bodies, N the number of timesteps, dt the timestep, and the rest is self-explanatory.

Table IV. Table containing standard deviation of the total energy as a percentage of the mean total energy in the simulations described in section IV.B. The column labeled "Forward Euler" corresponds to the simulations performed with the forward Euler algorithm, and the column labeled "Velocity Verlet" corresponds to the simulationgs performed with the velocity Verlet algorithm.

Timestep [years]	Forward Euler	Velocity Verlet
10^{-3}	1.97 %	4.28×10^{-4} %
10^{-4}	0.224 %	5.18×10^{-4} %
10^{-5}	2.28×10^{-2} %	5.10×10^{-4} %

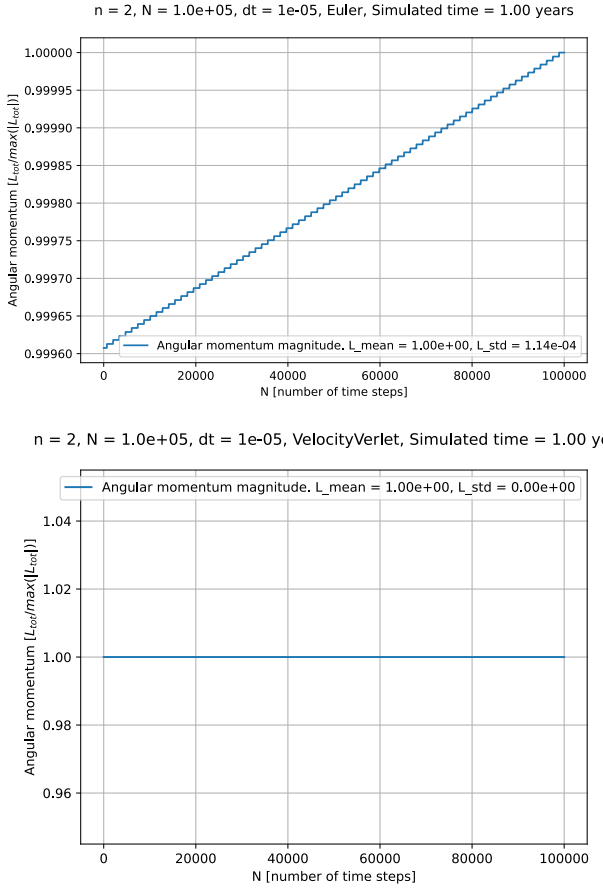


Figure 3. This figure contains two plots showing the magnitude of the total angular momentum L, scaled so that it's maximum value is 1, as it develops during two simulations with Earth and the Sun. Both simulations were ran with a timestep $\Delta t = 10^{-5}$, but one of them used the forward Euler algorithm, and the other the velocity Verlet algorithm (as indicated in the plot). The mean and standard deviation of L is listed in the legend text.

C. Simulations of Earth and the Sun with another kind of force

We ran simulations of the system with Earth and the Sun with the force given (19), for different choices of the parameter β. These simulations were all performed with the velocity Verlet algorithm, the timestep set to $\Delta t = 10^{-6}$ and the amount of timesteps $N = 10^7$. First we ran simulations with the same initial conditions as in the previous section. In this case all the plots of the orbits remained roughly the same for any choice of β between 2 and 3 and thus we only include a plot of the orbit with $\beta = 2.667$ which can be seen along with a plot showing potential, kinetic, and total energy in figure 4.

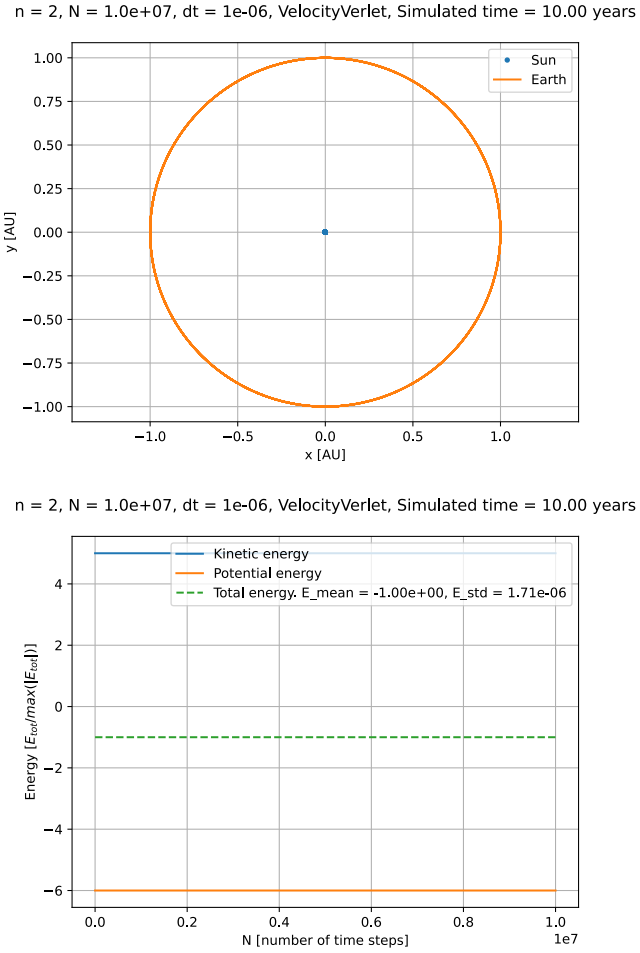
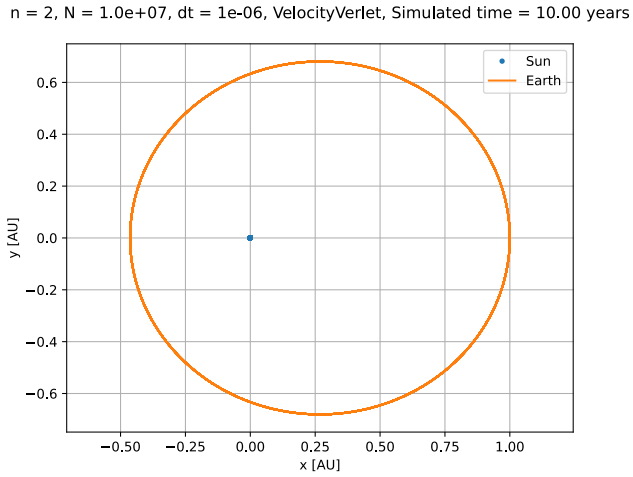


Figure 4. Figure containing two plots. The first plot shows the orbit of Earth around the Sun with the force interaction given by (19), where the parameter $\beta = 2.667$. The second plot shows potential, kinetic, and total energy in the same simulation. In the plot titles we have listed the number of bodies, n, the numer of timesteps, N, and the timestep, dt.

We also ran simulations with the same initial conditions, except that we set the initial velocity of Earth to be 5 AU/year instead of 2π AU/year. With a regular gravitational interaction this corresponds to elliptical orbits. We ran these simulations with $\beta = 2$, $\beta = 2.333$, $\beta = 2.667$ and $\beta = 3$. Setting $\beta = 2$ is the same as a regular gravitational interaction. We show plots of the orbit and the energy in this case in figure 5. The angular mo-

mentum remains a constant and is thus not shown. Plots of the orbit and energies in the case where $\beta = 2.333$ are shown in figure 6. The angular momentum remained a constant here as well and is thus not shown. Plots of the orbit and energies in the case where $\beta = 2.667$ are shown in figure 7. The angular momentum remained constant here as well. Plots of the orbits, energies, and angular momentum in the case where $\beta = 3$ are shown in figure 8. The plot of the energies and the plot of the orbits was zoomed in around the relevant portion of the plots.



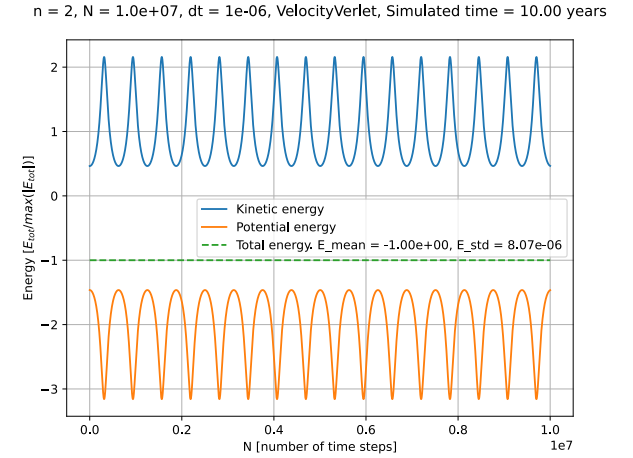
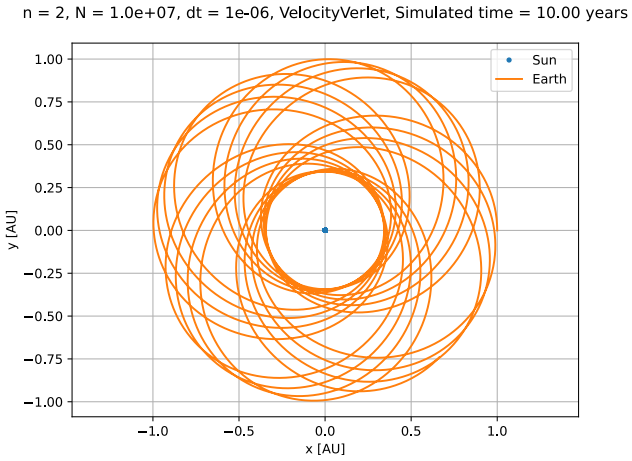


Figure 5. Figure contains a plot showing the simulated orbit of an Earth-like object orbiting around the Sun, and a plot showing potential, kinetic, and total energy for the system. The initial velocity has been altered from Earth's initial velocity so that the orbit is more elliptical. N is the amount of timesteps, dt is the timestep, and n is the number of bodies in the simulation.



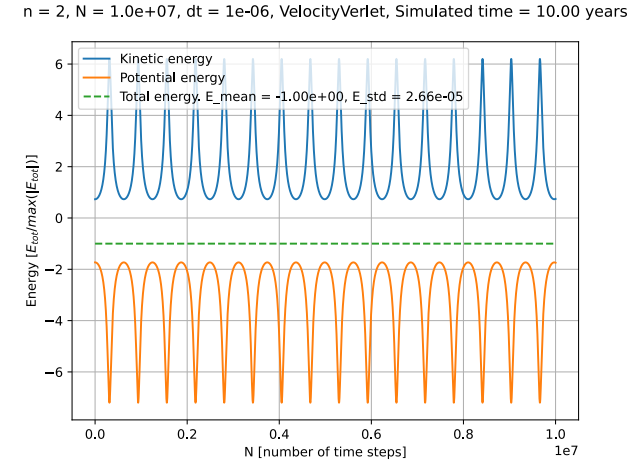


Figure 6. Figure contains a plot showing the simulated orbit of an Earth-like object orbiting around the Sun with a modified gravitational interaction ((19) with $\beta = 2.333$), and a plot of the potential, kinetic, and total energy for this system. The initial velocity has been altered from Earth's initial velocity. N is the amount of timesteps, dt is the timestep, and n is the number of bodies in the simulation.

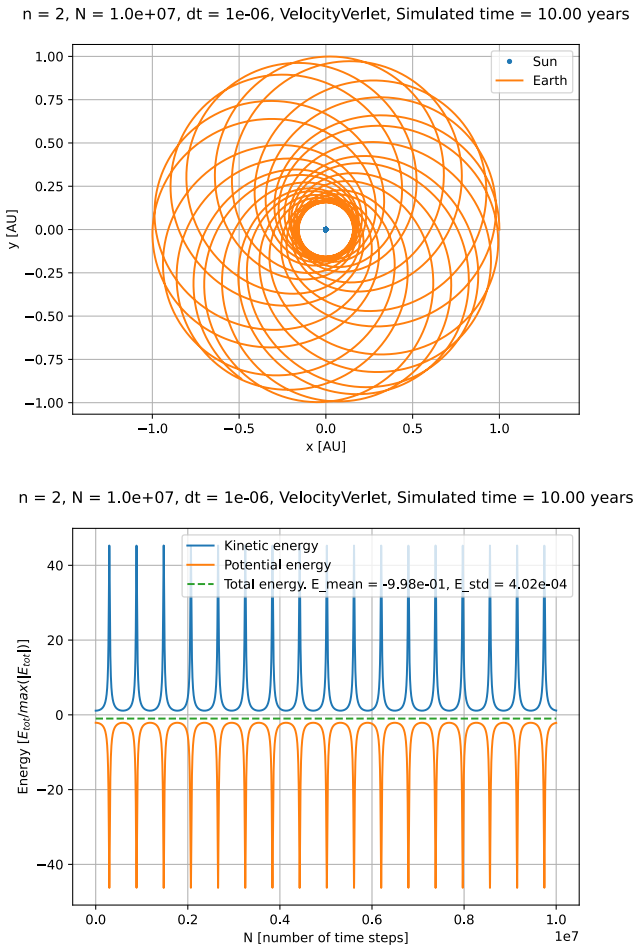


Figure 7. Figure contains a plot showing the simulated orbit of an Earth-like object orbiting around the Sun with a modified gravitational interaction ((19) with $\beta = 2.667$), and a plot of the potential, kinetic, and total energy for this system. The initial velocity has been altered from Earth's initial velocity. N is the amount of timesteps, dt is the timestep, and n is the number of bodies in the simulation.

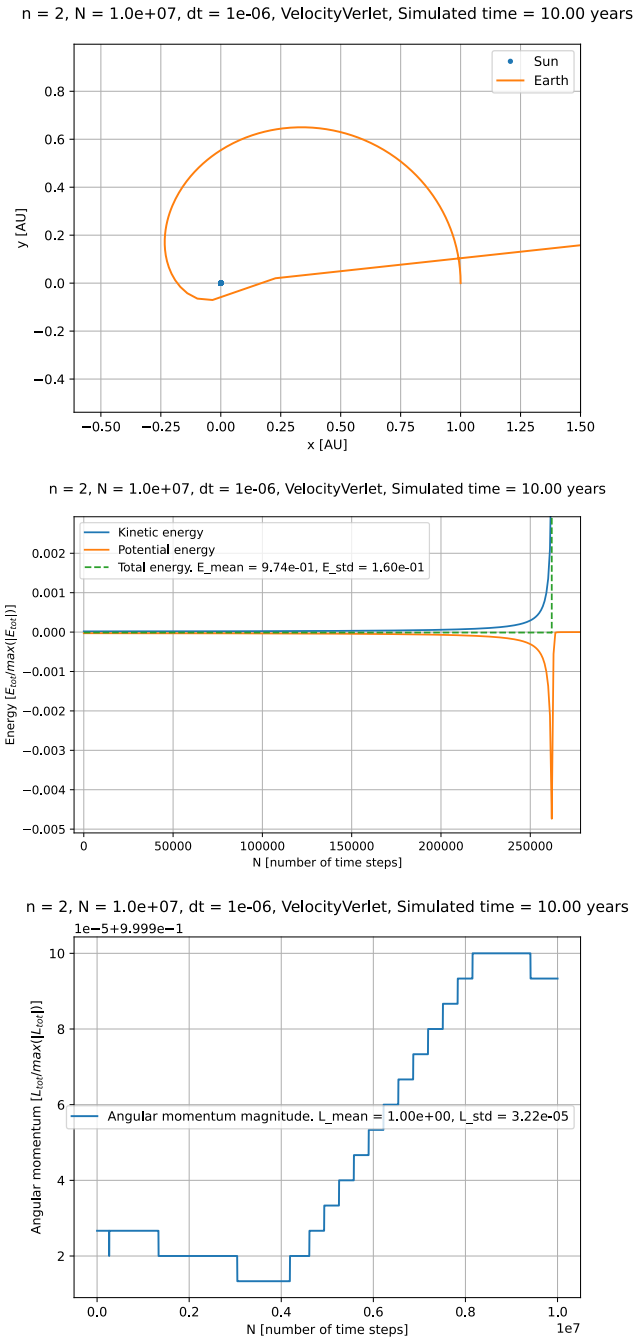


Figure 8. Figure contains a plot showing the simulated orbit of an Earth-like object orbiting around the Sun with a modified gravitational interaction ((19) with $\beta = 3$), a plot of the potential, kinetic, and total energy, and a plot of the magnitude of the total angular momentum of this system. Note that the plot of the energies and the plot of the orbits are zoomed in around their respeective relevant parts, as the rest of the plots would be unreadable. The initial velocity has been altered from Earth's initial velocity. N is the amount of timesteps, dt is the timestep, and n is the number of bodies in the simulation.

D. Simulations with velocity close to escape velocity

We ran simulations with Earth and the Sun, where we set Earth's velocity to values close to the escape velocity in the Sun's gravitational field. The escape velocity of Earth in this case was found in section II.E and the escape velocity itself is listed in (35). All the simulations were ran with $N = 10^7$ timesteps and timestep $\Delta t = 2.48 \times 10^{-5}$. We needed to simulate for a sufficient amount of time so that we could verify that Earth would or would not escape the gravitational field of the Sun, and we chose then a set of parameters such that the simulation time is approximately equal to the time it takes Pluto to complete a full orbit around the Sun, as that makes sense in terms of objects in our Solar System. We chose this as a baseline set of parameters in our simulations. We used the velocity Verlet algorithm as the method of numerical integration in these simulations.

The Sun is set to the origin of the system, and Earth starts 1 AU away from the Sun along the x-axis, and a varying velocity in the y-direction. We showcase three simulations here where the velocity was chosen such that it was just below, exactly at, or just above the escape velocity. We ran a simulation with the initial velocity set to $v = 8.8$ AU/year, one with $v = 2\pi\sqrt{2} \approx 8.886$, and the last with $v = 9$ AU/year. Plots showing the orbit and energies in these simulations can be found in figures 9, 10 and 11 respectively.

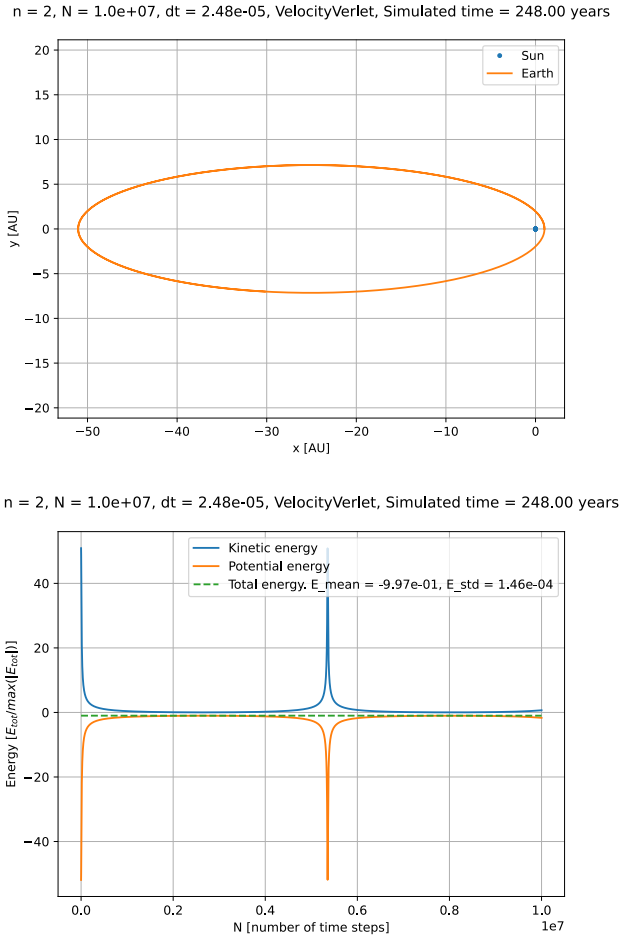


Figure 9. This figure contains plots of the path Earth took in a simulation with Earth and the Sun and the energy in this system. The initial velocity of Earth in the y-direction was set to 8.8 AU/year, and the other initial conditions used are listed in section IV.D. N is the number of timesteps, dt the timestep, and n the number of bodies in the system.

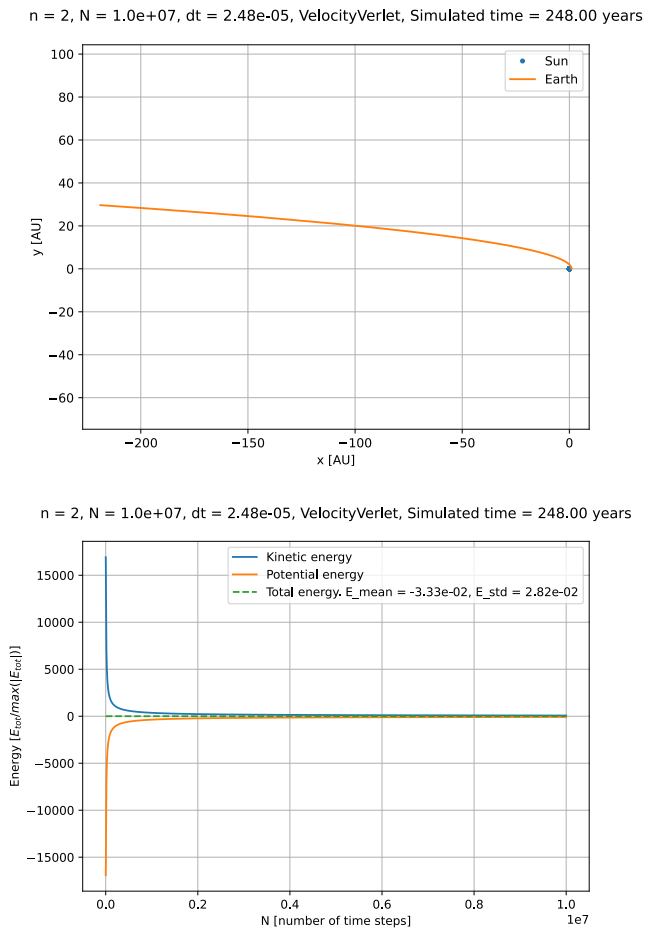


Figure 10. This figure contains plots of the path Earth took in a simulation with Earth and the Sun and the energy in this system. The initial velocity of Earth in the y-direction was set to $2\sqrt{2}\pi$ AU/year, and the other initial conditions used are listed in section IV.D. N is the number of timesteps, dt the timestep, and n the number of bodies in the system.

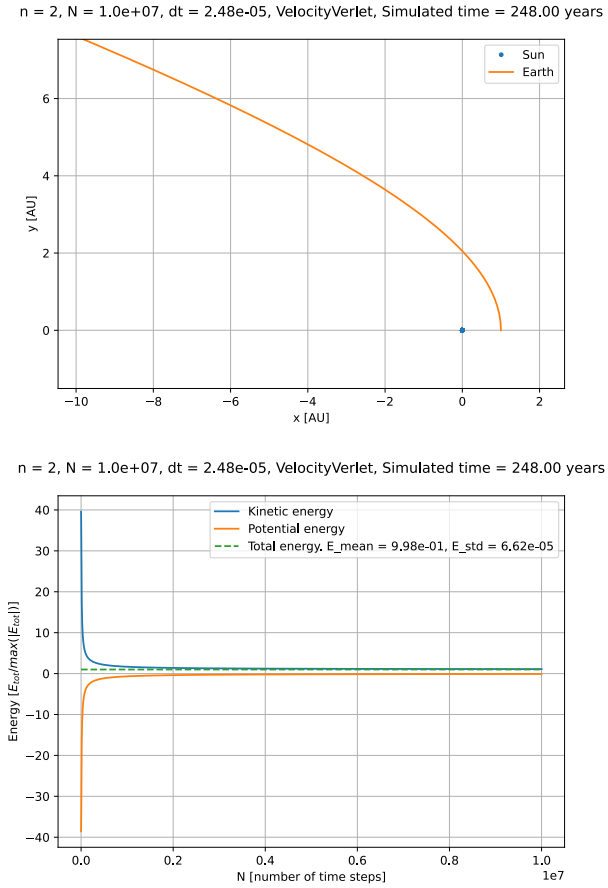


Figure 11. This figure contains plots of the path Earth took in a simulation with Earth and the Sun and the energy in this system. The initial velocity of Earth in the y-direction was set to 9 AU/year, and the other initial conditions used are listed in section IV.D. N is the number of timesteps, dt the timestep, and n the number of bodies in the system.

E. Simulations of Earth, Jupiter and the Sun

We ran simulations with Earth, Jupiter and the Sun. The initial conditions were obtained using [3]. Three simulations were performed with this system, one with the standard initial conditions, one with the mass of Jupiter being a factor 10 larger, and the last one with it being a factor 10^3 larger. All the simulations were ran with $N = 10^7$ timesteps, with steplength $\Delta t = 1.2 \times 10^{-6}$. This set of parameters was chosen so that Jupiter would complete a full orbit around the Sun. Plots of the resulting orbits can be found in figures 12, 13 and 14. Which plots are shown is determined by readability. All of the plots in these figures are in the reference system of the Sun. The simulations were ran in the center of mass frame, and then transformed to the reference frame of the Sun when processing the data. In most cases the 3D plots are the best, as the simulations were all ran in three dimensions, but in some other cases the plots in the xy-

plane are much easier to read, and give a better overview of the results. The angular momentum and total energy remained constant in all of these simulations, and thus plots of these were not included. We also included plots from two of these simulations in the center of mass frame, and these can be seen in figures 15, 16 and 17.

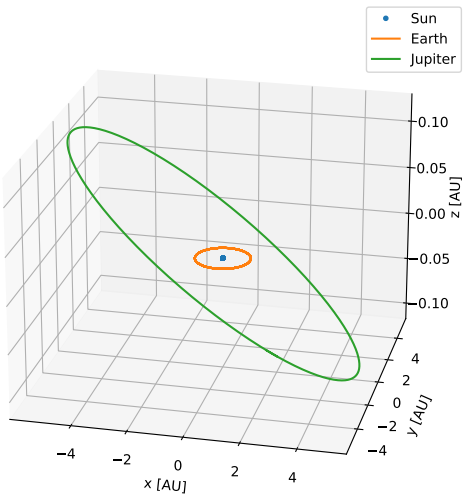


Figure 12. This figure contains a 3D plot of the simulated orbits of Earth and Jupiter around the Sun. The simulation was ran with $N = 10^7$ timesteps and steplength $\Delta t = 1.2 \times 10^{-6}$ years. Note that the axes in the plot are not equal, so the orbit of Jupiter is not as tilted as it may seem to be at first glance.

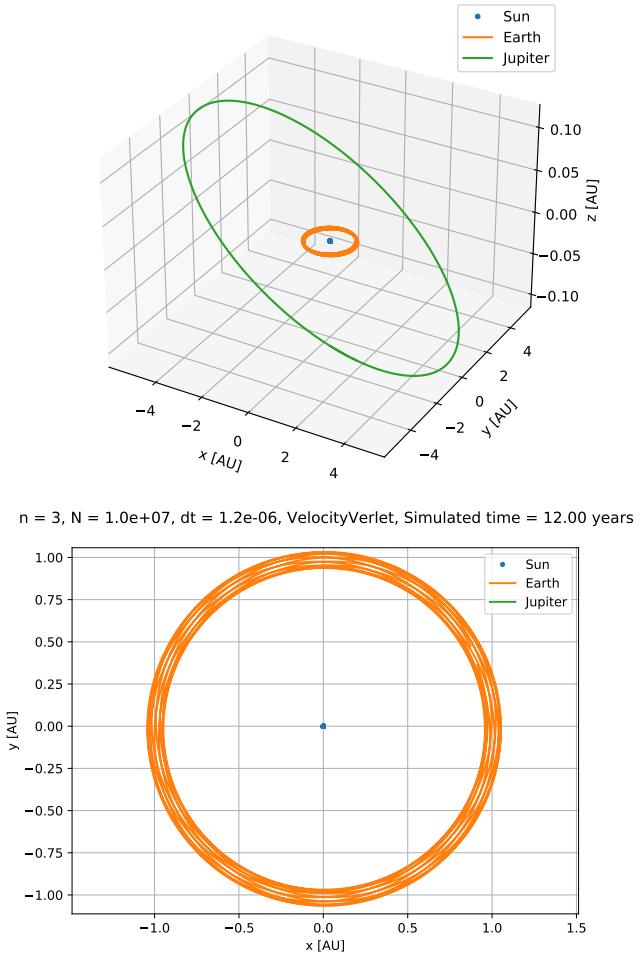


Figure 13. This figure contains a 3D plot of the simulated orbits of Earth and Jupiter (with Jupiter's mass being set to be 10 times larger than it actually is) around the Sun, and another plot of the same simulation but zoomed in to better show Earth's orbit in the xy-plane. The simulation was ran with $N = 10^7$ timesteps and steplength $\Delta t = 1.2 \times 10^{-6}$ years. Note that the axes in the plot are not equal, so the orbit of Jupiter is not as tilted as it may seem to be at first glance.

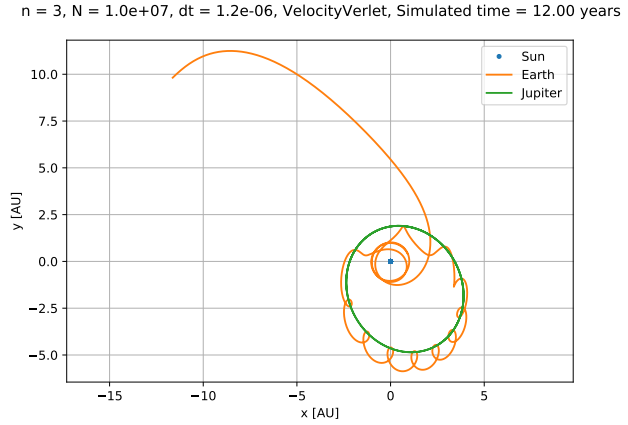


Figure 14. This figure contains a plot of the simulated orbits of Earth and Jupiter (with Jupiter's mass being set to be 10^3 times larger than it actually is) around the Sun in the xy-plane. The simulation was ran with $N = 10^7$ timesteps and steplength $\Delta t = 1.2 \times 10^{-6}$ years.

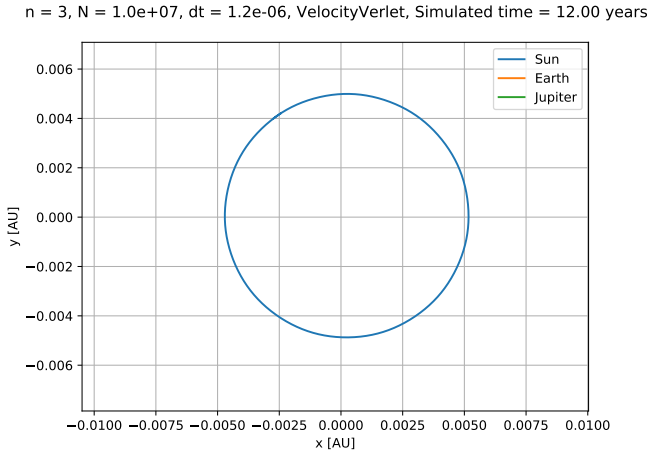


Figure 16. This figure contains a 2D plot of the simulated orbit of the Sun around the center of mass (origin of the plane), for the Sun-Earth-Jupiter system. The simulation was ran with $N = 10^7$ timesteps and steplength $\Delta t = 1.2 \times 10^{-6}$ years.

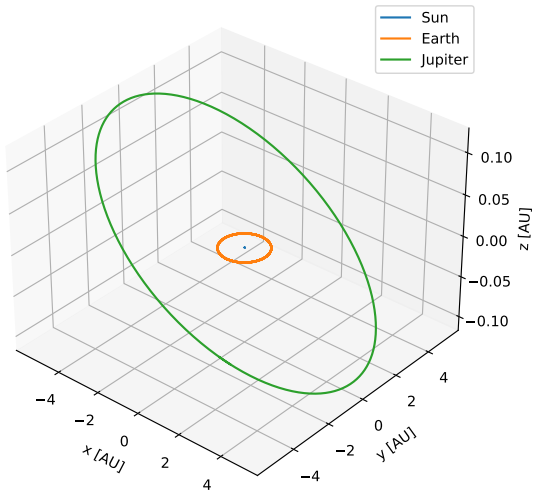


Figure 15. This figure contains a 3D plot of the simulated orbits of Earth, Jupiter and Sun around the center of mass (origin of the plot). The simulation was ran with $N = 10^7$ timesteps and steplength $\Delta t = 1.2 \times 10^{-6}$ years without transforming to the Sun's frame of reference. Note that the axes in the plot are not equal, so the orbit of Jupiter is not as tilted as it may seem to be at first glance.

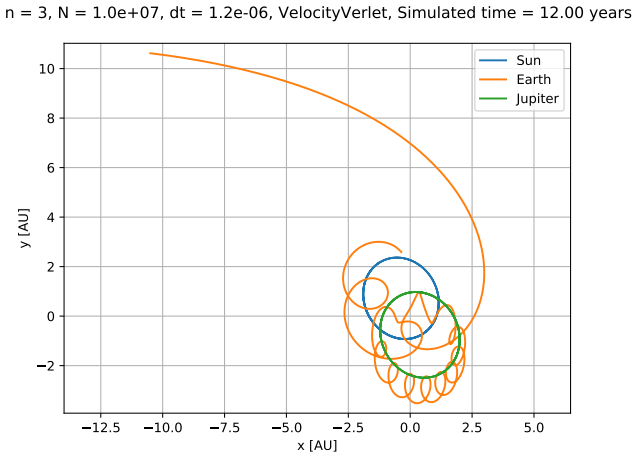


Figure 17. This figure contains a plot of the simulated orbits of Earth, Jupiter (with Jupiter's mass being set to be 10^3 times larger than it actually is) and the Sun around the center of mass (origin of the plane) in the xy-plane. The simulation was ran with $N = 10^7$ timesteps and steplength $\Delta t = 1.2 \times 10^{-6}$ years.

F. Simulation of the Solar System

We ran a simulation of the planets of the Solar System, the Sun and Pluto. Hereby we refer to this system as just "the Solar System", even though we have not included all objects that are in the actual Solar System. The simulation had $N = 10^7$ timesteps and steplength $\Delta t = 2.48 \times 10^{-5}$ years. This set of parameters was chosen so that Pluto would complete a full revolution around the Sun. Plots of the resulting orbits is shown in figure

18. The initial conditions for this system was generated using [3].

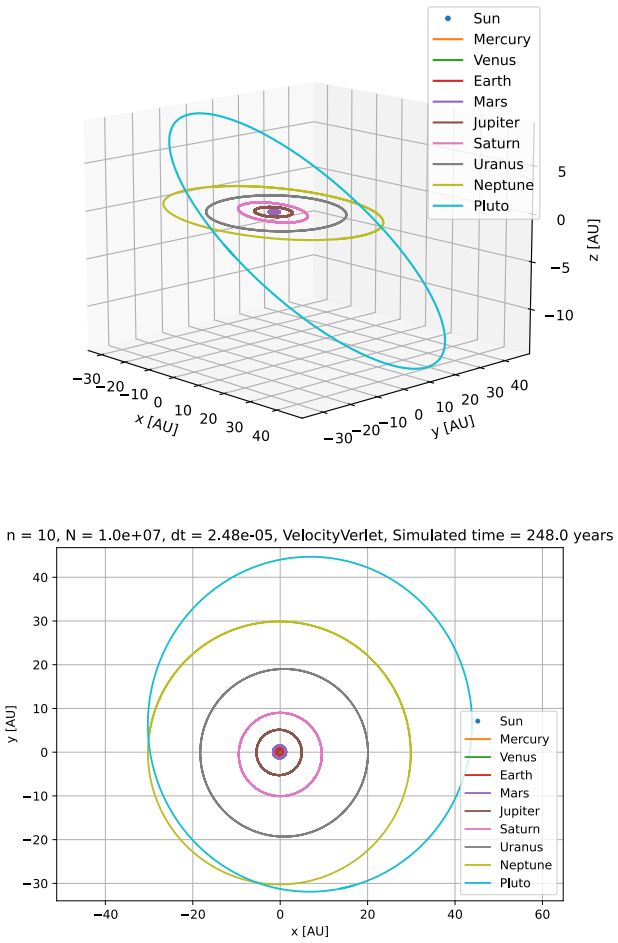


Figure 18. This figure contains a 3D plot of the simulated orbits of the planets (and Pluto) in the Solar System and the Sun around the center of mass. The simulation had $N = 10^7$ timesteps and steplength $\Delta t = 2.48 \times 10^{-5}$ years. Note that the axes are not equal in the 3D plot. The figure also contains a plot of the orbits in the xy-plane.

G. Calculating the perihelion precession of Mercury

We ran a simulation with Mercury and the Sun with the modified gravitational interaction given in (36). The simulation was ran with several sets of timestep and amount of timesteps, all chosen such that the simulation time would reach 100 years. An easy relation to use here is that we need the amount of timesteps to be $N = 10^{a+2}$ when the timestep is $\Delta t = 10^{-a}$. Mercury was initialized in its perihelion of $x_0 = 0.307491008$ AU with a linear velocity of $v_{y,0} = 12.433287$ AU/year [3]. The force was calculated only on Mercury so that the Sun would remain at rest during the simulation. We then calculated the position of Mercury's perihelion in the final revolution around the Sun in the simulation. The precession per century of Mercurys perihelion is then directly equal to the angular coordinate of this position. We also ran simulations with the same set of timestep and amount of timesteps with a normal gravitational interaction (as given in (12)). In this case the simulation was performed as normal, in the center of mass frame, with the results being transformed into the reference fram of the Sun when processing data. The precession of Mercurys perihelion was calculated in this case as well, so we could find the difference between the calculated values with the modified force in (36) and the regular gravitational interaction shown in (12). We also calculated the relative error in this value compared to the expected result of 43". These results are all shown in table V.

Table V. The table contains measured value of the precession of Mercurys perihelion per century for different sets of timesteps $N = 10^{a+2}$ and steplength $\Delta t = 10^{-a}$ years. The parameter a determines both and is listed in the first column of the table. The precession was calculated both with a standard gravitational interaction (listed in the second column) and with the modified force given in (36) (listed in the third column). The difference between these values is listed in the fourth column. We also calculated the relative error in the difference compared to the expected result of 43", and this is listed in the last column.

a	Classical	Relativistic	Difference	Relative error
4	-1122.17"	-912.284"	209.886"	388 %
5	-7.04534"	34.1085"	41.1538"	4.29 %
6	0.679516"	43.0873"	42.4078"	1.38 %
7	0.173463"	42.9659"	42.7924"	0.483 %

V. DISCUSSION

A. Benchmarks of the numerical methods

We benchmarked the two numerical methods of integration we have presented. In order to do this we ran simulations of the solar system using the forward Euler method (3) and the velocity Verlet method (9). The benchmark results can be found in table III. We also counted the amount of FLOPs needed (shown in table II), and from this we can see that the algorithms perform as expected. The amount of FLOPs needed to perform a simulation with the velocity Verlet algorithm is approximately twice the amount needed with the forward Euler method, and the measured times also differ by an approximate factor of 2. The velocity Verlet algorithm is slower than the forward Euler method, which is generally a compromise made for the improved accuracy of the former. As these time measurements were averaged over several runs, we were also able to calculate the standard deviation in the time measured. From these we can see that the velocity Verlet method is slighly more unstable in terms of the time it takes to run the simulation. While the mean time spent by the methods differ by about a factor 2, the standard deviation differs by larger factor of about 3.5. This is generally of no consequence, but it is an interesting fact to take note of.

B. Simulations of Earth and the Sun

We ran several simulations of Earth and the Sun, with the results of these being shown in sections IV.A, IV.B, IV.C and IV.D.

1. *Simulations with different choices of timesteps*

Simulations using both the forward Euler and velocity Verlet methods of integration were done for different choices of timesteps. This was done so as to compare the accuracy of the methods, and to see which choices of timestep gave the most stable results. The sets of step length and amounts of timesteps were all chosen so that the simulation time would reach 1 year, so that we expect Earth to have completed a full revolution around the Sun. The initial conditions was chosen such that we expect the orbit of Earth around the Sun to be circular, and that one revolution would take 1 year.

The resulting orbits from the simulations using the forward Euler method can be seen in figure 1. In general we see that the results improve the shorter the timestep is. Importantly, we note that Earth isn't able to complete a full orbit with timesteps $\Delta t_1 = 10^{-3}$ years and $\Delta t_2 = 10^{-4}$ years even though the simulation time is a year. We can also see from the trajectory of Earth that the orbit is not stable in these two simulation, although the simulations didn't last long enough to clearly show this. In other words, after a full orbit Earth is clearly not returning to the original starting point. The simulation with $\Delta t_3 = 10^{-5}$ years has no visible error in the orbits, but from what we've seen so far it is not unreasonable to expect that after running the simulation for a sufficient amount of time we would see the orbit diverge from the expected result of a stable circular orbit. As the timestep gets shorter, we expect to eventually encounter problems with machine precision, but we obviously do not encounter this as an issue in these simulations.

The resulting orbits from using the velocity Verlet method with the same sets of steplength and amount of timesteps can be seen in figure 2. There is no visual difference in the orbit of Earth around the Sun in this case for any of the choices of timestep. This seems to indicate that the velocity Verlet method is better suited for this kind of simulation.

The total energy was also measured, and the standard deviation of the total energy from the mean in these simulations can be seen in table IV. As seen in section II.C energy is supposed to be a conserved quantity. We can see that the calculated energy is generally not as stable with the forward Euler method as with the velocity Verlet method, which is as expected. The velocity Verlet method is supposed to be symplectic, but some error always arises in any kind of numerical calculation. The fact that it seems to be approximately the same irrespective of timestep is a good sign, as it indicates that the method or the approximations used in the method is not what is causing the error. If that was the case the error would scale with the timestep and amount of timesteps. The forward Euler method does not, however, conserve energy, and this is pretty clear from our results as well. As the accuracy increases inversely with the step length, we expect better results for shorter step length when using this method of integration. With these differences in mind, these results also seem to support that the velocity Verlet method is better suited for this kind of problem.

We also measured the total angular momentum of the system in the center of mass frame. This is supposed to be a conserved quantity of the system (as we have shown in section II.D) and so we expect this to be the case in these simulations as well. In general we saw similar behaviour in all of the simulations and so only included plots of the case where the timestep was the shortest ($\Delta t_3 = 10^{-5}$). These plots can be seen in figure 3. In these plots we can see the magnitude of the total angular momentum, scaled against the maximum value measured of the same quantity so that it is dimensionless. Using the forward Euler method seems to cause a gradual increase in the magnitude of the total angular momentum, and using the velocity Verlet method seems to conserve it. In general this indicates that the forward Euler method will not conserve this quantity, and that the velocity Verlet will. As it should be a conserved quantity, this result also implies that the velocity Verlet method is the better suited method.

All in all, our results indicate that for this kind of

problem the velocity Verlet method works better than the forward Euler method as the method of numerical integration. In all the simulations that followed these ones, we chose to use the velocity Verlet method because of these results. As some of these results used here (conservation of energy and angular momentum) are easy to verify numerically, these were also implemented as unit-tests (see section III.D).

2. *Modified force*

We used the modified force given in (19) during some simulations of Earth and the Sun. First we ran these simulations, with initial conditions corresponding to a circular orbit with $\beta = 2$, for different choices of β (always between 2 and 3). The orbit of Earth around the sun stayed circular for all of these simulations, and so we only included a plot of the orbit and the energy of the system with $\beta = 2.667$ in figure 4. We showed in section II.C that the total energy is supposed to be conserved, and this is the case in this simulation, and all the others that we did not explicitly include plots of. The orbit however is of more interest, as it is circular for any choice of β. In the case of the system with Earth and the Sun however, it is quite easy to see that this is the case. As the distance between the two objects is initialized as 1 AU, it is clear that $r^\beta = 1^\beta = 1$ is the same independent of what β is as long as the distance remains a constant. This means that the force is independent of β in this case. In general a change of units should not affect the system, and the only thing that is truly special in this case is that the initial conditions corresponds to circular orbits when $\beta = 2$. This means that this is the case for any circular orbit. In other words, as long as the orbit is circular when $\beta = 2$, it remains circular for any other choice of β as well.

Now that we found that circular orbits are a special case that simplify the solutions, we also decided to run simulations with initial conditions corresponding to an elliptical orbit. In order to do this, we simply altered the velocity of Earth in the previous set of initial conditions, so that it was less than what it used to be. We ran the simulation first with $\beta = 2$ in order to verify that the orbit was elliptical as expected. This can be seen in figure 5. Here we see that the orbit is elliptical, and that the total energy is conserved as expected. The potential and kinetic energy vary. As the distance between Earth and the Sun varies when the orbit is elliptical, this means that the velocity also has to vary in order for Kepler's second law (26) to hold true. This implies a change in kinetic energy due to the change in velocity (17) and a change in the potential energy due to the change in positions (20), and so it is expected that they vary in this simulation. We also ran the simulations with $\beta = 2.333$, $\beta = 2.667$ and $\beta = 3$, and the results of these simulations can be seen in figures 6, 7 and 8 respectively.

When $\beta = 2.333$ and $\beta = 2.667$ we see similar results. The orbit of Earth around the Sun is no longer stable, but it still oscillates between a maximal and a minimal distance away from the Sun, and in that sense this kind of orbit could be viewed as one where the perihelion and aphelion precesses as time passes. With this taken into consideration the energy behaves as expected. The total energy is conserved, and the potential and kinetic energy vary, and how much they vary is dependent on β. This makes sense, as there seems to be a clear trend that the orbits extremes move further away from each other as β increases. In our case it is the perihelion of the orbit of Earth that moves closer to the Sun, as we initialize the motion in aphelion. In both of these simulations the angular momentum remained pretty much constant, with no gradual drift.

With $\beta = 3$, however, we see more weird results. While the plots do look weird if we consider this to be what would happen in a real physical case with a force such as this, it is not that weird when numerical effects are taken into consideration. In this case several of the regularities we have experienced earlier, such as conserved total energy and angular momentum seem to break. This makes us doubt the accuracy of the numerical solution more than anything else, as these quantities should be conserved per our results in sections II.C and II.D. The other thing we must take into consideration when evaluating these plots are that not all the data points of the simulation are plotted, as the plots contain only 10^4 evenly spaced points irrespective of the amount timesteps used during the simulation. All this taken into consideration it seems that what happened here was that Earth got too close to the Sun, the error term got large because the step length was not short enough, and then Earth got even closer to the Sun than it was supposed to, resulting in the force and thus the calculated velocity blowing up.

The plots shown in figure 8 are zoomed in (except for the plot of the magnitude of the total angular momentum) around the relevant portions, and we can see that this explanation fits in all cases. In the plot of the orbit we can see that as Earth gets closer to the Sun during the first orbit, it suddenly shoots off to the side. This seems to fit with the assumption that the velocity blows up. As for the total energy, we can see that it remains a constant. The potential and kinetic energy both increase in size, as it did for other β as well, and as it seems to reach a maximum suddenly the potential energy tends to zero as the total energy and kinetic energy blow up. This also fits the explanation that Earth got too close to the Sun, and that the numerical error caused the velocity to blow up.

The magnitude of the angular momentum does not really tell us anything very useful, as it is expected that the conservation of this quantity breaks when everything else breaks as well. The interesting part here is that the change in angular momentum seems to be discrete, in a way. This might be because of loss of significant digits. There are subtractions performed when calculating the cross product (which is necessary to calculate the

angular momentum (27) in general), and as some of the components of the velocity and position gets very large as other don't when Earth drifts off, it is not unexpected that this can cause a loss significant digits in the angular momentum. If there is a loss of significant digits here, this also carries over to the magnitude of the total angular momentum. All the problems in this simulation could probably have been avoided if we chose a shorter step length.

All in all, the results discussed in this section are as expected, with the only outlier being the simulation where $\beta = 3$ and we had initial conditions corresponding to an elliptical orbit when $\beta = 2$. We chose to include this simulation anyway, as it highlights key difficulties when running simulations such as these. This shows that while we only change β, if we are not careful we can still get large numerical errors if we don't change the step length as well. As the pattern in how the motion changes was apparent from only looking at the cases where $\beta = 2.333$ and $\beta = 2.667$, we thought it more relevant to highlight where these simulations can go wrong rather than rerunning the simulation with a better choice of step length.

From observations of actual planetary orbits we see some deviation from perfect orbits, with the extremes precessing such as they did in these simulations. This might mean that Nature differs slightly from a perfect inverse-square law, but this is only when we don't consider other effects as well. We know that some precession of the orbits occur when more bodies are introduced, so inherently the idea of looking at a two-body system to evaluate whether or not Nature deviates from a perfect inverse-square law is wrong. We also know that Nature does not follow a perfect inverse-square law already when relativistic effects are taken into account. This cannot be expressed simply by adding the factor β such as we have, and so while we might be able to fit our results to real orbits by appropriate choices of β, there would be no universal choice of β that fits all observations.

3. *Escape velocity*

We ran simulations in the system with Earth and the Sun, where we set the velocity of Earth to something close to, or exactly equal to the escape velocity of Earth. The escape velocity of earth was calculated in section II.E, and we found it to be $v_e = 2\pi\sqrt{2} \approx 8.886$ AU/year. Thus we ran three simulations, one with the velocity set just below the escape velocity ($v = 8.8$ AU/year), one with the velocity to the escape velocity, and one just above it ($v = 9$ AU/year). These are shown in figures 9, 10 and 11 respectively. In general there is nothing unexpected showing up in any of these simulations. The energy is conserved in all of them as expected. In the simulation where the velocity is just below the escape velocity we can see that Earth is still orbiting around the Sun, just in a very elliptical orbit, and in the other two Earth escapes the Sun's gravitational field as expected.

It is also worth noting that when we found the escape velocity analytically we used the the total energy needed to be larger than or equal to zero in order for Earth to escape. We see in the simulation where the velocity is just below the escape velocity that the total energy is negative, that it is close to zero in the one with Earth being at escape velocity and it being positive when Earth's velocity is just above escape velocity. Note that the mean of the total energy in the case where Earth is at escape velocity is not zero. This is most likely due to a truncation error in the velocity of Earth. We cannot represent the number exactly, and as $\pi\sqrt{2}$ is an irrational number this means there will be a truncation or rounding unless we express the system in units of this number when solving.

As the energy in this case deviates slightly to negative values however, we would expect that the velocity might also be truncated in such a way that Earth shouldn't escape the gravitational field of the Sun. This doesn't seem to be the case from looking at the plot however, but it might be that we just didn't run the simulation long enough to see Earth start turning around. However, when the distance gets large and the velocity and force are small, as it does when Earth moves away, the simulation is prone to more numerical error. When integrating we use the velocity Verlet method (9), and this method contains additions in order to move the system forward in time. When we add small values to large ones, this can generally cause significant errors due to loss significant digits. All in all this makes it hard to determine exactly what will happen in a simulation where the velocity of Earth is set extremely close to its escape velocity as this is very prone to numerical errors.

C. Simulations of many-body systems

We ran several simulations of systems with $n \geq 3$ number of bodies. The results of which is shown in sections IV.E for the Sun-Earth-Jupiter system, and IV.F for the entire Solar System.

1. *Simulations of Earth, Jupiter and the Sun*

Firstly, 12 shows that there is no meaningful perturbation of Earth's orbit caused by the presence of Jupiter, which is as expected given the relatively short simulation time of 12 years.

Furthermore, 13 shows that increasing Jupiter's mass by a factor of 10, causes visible perturbations in Earth's orbit, but does not cause major instabilities over a 12 year time frame. This is perhaps to be expected given that the minimum distance between Jupiter and Earth is 3.934 AU [4], which is further than the 1 AU distance between Earth and the Sun. Even at its closest approach, the gravitational force of Jupiter on Earth, with 10x Jupiter's mass, is still expected to be four orders of magnitude

weaker than the average force of the Sun on Earth at 1 AU distance.

Finally, 14 shows a significant alteration in Earth's orbit, resulting in a potential ejection from the system. Such change in Earth's orbit is expected, as Jupiter with 1000x its normal mass, would have approximatly 95% the mass of the Sun.

As mentioned in IV.E, the total angular momentum and energy was conserved in all three simulations. However, there might arise error in the simulation for significantly close approaches between Jupiter and Earth. This is because Earth could potentially accelerate to velocities at which our temporal resolution is insufficient for approximating motion with non-uniform acceleration. The timestep used of 1.2×10^{-6} years, corresponds to approximatly 37.84 seconds. Given that energy and angular momentum remained conserved, we assume our temporal resolution to be sufficient.

We have also plotted the Sun-Earth-Jupiter system from the reference frame of the systems center of mass. As seen in 15, this does not significantly change the shape of the orbits when Jupiter has its normal mass. This is to be expected as the Sun is still the dominant mass in the system. We can see from 16 that the Sun's orbit around the center of mass has an apoapsis of 0.005 AU, which corresponds to approximatly 1.1 solar radii [5].

With Jupiter at 1000x its normal mass, we can se that the Sun's orbit around the center of mass is significant with an apoapsis of more than 2 AU, since Jupiter effectively functions as a second star.

2. *Simulations of the solar system*

We ran a simulation of the planets of the solar system, Pluto, and the Sun, which we henceforth refer to as just the solar system. The resulting orbits from this simulation are shown in figure 18. In this figure we show two plots, one being a 3D plot of the orbits, and the other showing the orbits in the xy-plane. We chose to show both as the orbits are mainly in the xy-plane other than the notable exception being Pluto. Both total energy and angular momentum remained approximately constant in this simulation, and so we have not included plots of these.

All in all this system behaves as expected, and it shows that our solver is capable of handling systems such as these as well. We must note that in cases as these were some of the bodies are much farther from each other than others, that the step length chosen must cater to the bodies that are the closest. In this case it is important that the step length is chosen such that there is no significant error in Mercurys orbit, as it is the closest to the Sun, and thus probably the body being most susceptible to numerical error from a step length that is too large. This can inadvertedly cause the amount of steps necessary to simulate a full orbit for the farthest objects (Pluto in this case) to get very large. When we calculated the precession of Mercurys perihelion (see section V.D for a full discussion on this) we saw that timesteps of an order of magnitude larger than 10^{-5} cause significant error in the orbit of Mercury, and thus we chose something of this order of magnitude as the timestep in this simulation as well. In the end we chose the timestep $\Delta t = 2.48 \times 10^{-5}$ and $N = 10^7$ amount of timesteps as this equates to a simulation time of 248 years, which is the time it takes Pluto to complete a revolution around the Sun.

D. Precession of Mercurys perihelion

For the simulation of relativistic and non-relativistic orbits of Mercury around the Sun, we see from table V that the simulated precession caused by the relativistic correction of gravity approaches the observed value of $43''$ per century. We also see that our most accurate result is $42.7924''$ per century with a relative error of only 0.483%, achieved with a timestep of 10^{-7} years. This suggests that the perihelion precession of Mercury can be reliably explained by General Relativity. While we obtained our result as the difference between the precession from the relativistic and classical calculations, we can also see that the classical result approaches zero while the relativistic result approaches 43" faster than the difference between these two as the timestep gets shorter. This might mean that it would be better to look at the value from the relativistic simulation directly instead of the difference between the values from the relativistic and classical simulations as we did.

Idealy the non-relativistic precession should be zero, since a pure inverse-square law such as Newtonian gravity should give closed elliptical orbits [1, p. 5], but we expect some precession due to the accumulation of numerical errors during integration. Furthermore, we see from table V the Newtonian precession approach zero as the timestep decreases. This, combined with the relativistc precession approaching the observed value, suggests that our implementation of the velocity Verlet method is behaving as expected. There is also a large jump in the error when the timestep is 10^{-4} years, compared to the other simulations. This indicates that simulations using this system require a timestep of, at most, order 10^{-5} years to function properly, which is a result we also used in the previous section (V.C.2).

VI. CONCLUSION

In this report we presented the forward Euler (3) and velocity Verlet (9) methods of numerical integration. Our main objective was to verify some key properties and differences in these two methods. Systems based on celestial mechanics are particularly well suited to this purpose, and so we wrote a set of programs that simulate gravitational systems, using the two methods, which can be seen in section III. Accompanying these programs we wrote a Python script automating the simulations presented in this report and a C++ program which can be used to interface with the implementations. We also wrote unittests based on conservation of energy and angular momentum (which are conserved per our results in sections II.C and II.D).

We counted the amount of FLOPs necessary to perform simulations with the two methods (shown in table II), and performed a benchmark in order to verify that they behave as expected. The benchmark results in table III differ by about the same factor as the amount of FLOPs needed, meaning the solvers behave as expected in this regard.

In order to verify the numerical errors caused by the methods, we performed similar simulations of the Sun-Earth system with different choices of timesteps. The results from these simulations can be seen in figures 1, 2, 3 and table IV. These results verified that the error was larger when using the forward Euler method (should be $\mathcal{O}(\Delta t^2)$) than the velocity Verlet method (should be $\mathcal{O}(\Delta t^3)$). The simulations using the forward Euler method generally displayed orbits that were not closed, and that total energy and angular momentum were not conserved quantities unless the timestep was significantly shorter compared to the time steps that yielded similar results with the velocity Verlet method. The velocity Verlet method is supposed to be symplectic, and that also seemed to be the case in our simulations. Some error arises from machine precision, but these were so small as to not cause any relevant disturbance in the results. In the end we concluded that the velocity Verlet algorithm was better suited for simulations of systems in celestial mechanics, and so we used this method for all the other simulations performed in the report.

We also performed simulations where we used force interactions that differ from the pure inverse square form that Newtonian gravity has. The results from these simulations can be seen in figures 4, 5, 6, 7 and 8. This force is given in (19). Simulations of the Sun-Earth system using this force gave the result that energy and angular momentum was still conserved, which should be the case per our results in sections II.C and II.D. With initial conditions corresponding to circular orbits with a standard gravitational interaction, we observed that the orbits also remained circular irrespective of the choice of β. This is expected, as we reasoned in section V.B.2. Initial conditions corresponding to elliptical orbits, however, displayed major differences with other choices of β. There is a precession of the orbits, and as β gets larger, the perihelion and aphelion of Earth move farther away from each other. Among these simulations we also presented a failed one in figure 8 were we concluded that the choice of timestep was inadequate. We still chose to include this in the report, as it highlights that the adequacy of a given timestep depends on β.

Lastly for the Sun-Earth system we calculated the escape velocity of the Sun at 1 AU (35), and ran three simulations with the initial velocity of Earth close to this. The results of the simulations can be seen in figure 9, 10, and 11. Generally they all behaved as expected, since Earth escaped the gravitational field when the velocity was larger than the escape velocity, and stayed trapped when it was smaller. Some difficulties arose when running the simulation with the velocity set to the escape velocity, which we concluded were most likely due to a truncation error of the input velocity. To perform a simulation with Earth exactly at escape velocity, we would have to define our units in terms of this number, which we did not do. This is might be of interest for future research, as it can say something about how the numerical error of the solver behaves in terms of sign specifically.

In order to truly test the capabilities of our solver and implementation we found it necessary to simulate systems with more than 2 bodies. First we simulated the motion of Earth, Jupiter, and the Sun, and the results from these simulations can be seen in figures 12, 13, 14, 15, 16 and 17. The mass of Jupiter was increased for each simulation, in order to see what effect this would have on the orbit of Earth, and the Sun. As expected, the closer the mass of Jupiter got to the mass of the Sun, the more disrupted Earth's orbit became. We also viewed the system in the center of mass frame, where we could see the effect of Jupiter on the Sun as well. As the mass of Jupiter increased, the orbit of the Sun was altered more, and the apoapsis of its motion increased as expected. Total energy and angular momentum were conserved in all of these simulations, which seem to indicate that our solver is capable of simulating three-body problems without issues.

Following this we simulated the planets of the solar system along with the Sun and Pluto. The results of this simulation can be seen in 18. These results are fully as expected. All of the orbits are as expected in the center of mass frame, and there is no reason to believe there is any major numerical failing in this simulation, as the total energy and angular momentum remained conserved. This indicates that our implementation and solver is able to simulate many-body system, as long as the choice of timestep is adequate for the given system.

As a final test for our implementation of the velocity Verlet method, we attempted using it to find the precession of Mercury's perihelion due to relativistic effects. In order to do this we calculated the precession per century numerically both in the relativistic and non-relativistic case for different choices of timesteps, found the difference between these results and compared this to the ob-

served value of 43". The results are shown in table V. As expected the calculated precession per century due to relativistic effects tend towards the observed value as the timestep gets smaller. At $\Delta t = 10^{-7}$ years the relative error was at just 0.4834 %. The error spikes as the timestep gets larger than 10^{-5} and thus we conclude that the timestep necessary to simulate this system should not be of a magnitude larger than this. As the results corresponded well with the observed value, we conclude that our solver is capable of solving this kind of problem as well.

[1] M. Hjorth-Jensen, *Project 3, deadline October 26 (Monday)*, Tech. Rep. (Department of Physics, University of Oslo, Norway, Oslo, 2020).
[2] C. Sanderson and R. Curtin, Journal of Open Source Software **1**, 26 (2016).
[3] NASA, "Horizons web-interface," `https://ssd.jpl.nasa.gov/horizons.cgi`, accessed: 19/10-2020.
[4] NASA, "Jupiter Fact Sheet," (2014).
[5] M. Emilio, J. R. Kuhn, R. I. Bush, and I. F. Scholl, Astrophysical Journal **750**, 135 (2012), arXiv:1203.4898.

Appendix A: Source code

All code for this report was written in C++ and Python 3.8, and the complete set of files can be found at https://github.com/eivinsto/FYS3150_Project2.git.

1. project.py

Main script for running project, plotting and data analysis.
https://github.com/eivinsto/FYS3150_Project_3/blob/main/project.py

2. main.cpp

Main cpp file for running algorithms.
https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/main.cpp

3. solar_integrator

Source and header file containing class for integrating a velocity and position for a system of bodies.
Source file solar_integrator.cpp: https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/solar_integrator.cpp
Header file solar_integrator.hpp: https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/solar_integrator.hpp

4. solar_system

Source and header file containing class for creating system of bodies, and calculating gravitational force between them, as well as kinetic and gravitational-potential energy of system.
Source file solar_system.cpp: https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/solar_system.cpp
Header file solar_system.hpp: https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/solar_system.hpp

5. celestial_body

Source and header file containing class defining gravitational bodies.
Source file celestial_body.cpp: https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/celestial_body.cpp
Header file celestial_body.hpp: https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/celestial_body.hpp

6. test_main.cpp

File running unit-tests using CATCH2 framework.
https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/test_main.cpp

7. test_functions.cpp

File containing the unit-tests for the system.
https://github.com/eivinsto/FYS3150_Project_3/blob/master/src/test_functions.cpp

8. benchmark.cpp

Source code for benchmark program.
https://github.com/eivinsto/FYS3150_Project_3/blob/main/src/benchmark.cpp

Appendix B: Selected results

Here is a folder of selected results from running our code.
https://github.com/eivinsto/FYS3150_Project_3/tree/master/data

Appendix C: System specifications

All results included in this report were achieved by running the implementation on the following system:

- CPU: AMD Ryzen 9 3900X
- RAM: $2 \times 8\,\text{GB}$ Corsair Vengeance LPX DDR4 $3200\,\text{MHz}$