Exploring N-Body Simulations and Their Applications

by

James Peters (201122706)

Supervisor: Dr. Matt Darnley

A thesis submitted in partial fulfillment for the degree of Master of Physics

in the
Faculty of Science and Engineering
Department of Physics

May 11, 2020

Contents

Declaration of Authorship

I, James Peters, declare that this thesis titled, 'Exploring N-Body Simulations and Their Applications' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____

Date: ____

University of Liverpool

Abstract

Faculty of Science and Engineering

Department of Physics

Master of Physics

Exploring N-Body Simulations and Their Applications

by **James Peters** (201122706)

A study on the effectiveness of N-Body simulations and their uses. A complete N-body simulator has been constructed from first-physical principles which has then been benchmarked against the JPL HORIZONS data-set. Four integrators were investigated: a 1st, 2nd and two 4th-order methods - the Euler, Leapfrog, Yoshida and Runge-Kutta methods respectively. The 4th-order Yoshida method was found to provide the most accurate results at large time-steps during long time-scale simulations however at short time-scales there exists a small range of time-steps in which the Runge-Kutta provides less accumulated error. The simulation was then applied to predict Solar and Lunar eclipses and was accurate to within ± 10.2 minutes over a 20 year period. The transits of the Trappist-One system were also simulated and compared to observations, the use of limb darkening and a polynomial fit was applied to reduce χ^2 and provide a closer fit. The discrepancies between the simulated results and the JPL Horizons and Trappist observations are discussed and the need for additional models on top of Newtonian mechanics.

Chapter 1

Introduction

1.1 Origins of the Laws of Gravitation

The dynamics of gravitational systems have been a curiosity of astrophysicists since we first started looking at, and measuring, the positions of the stars. Very early models of the solar system described the Earth as being at the centre of the universe, with the Sun, the Moon and the stars all moving around it - the geocentric model. This theory prevailed for centuries, largely due to religious beliefs which stated that the Earth was at the centre of all creation. It wasn't until Nicolaus Copernicus published the heliocentric model in 1543 that beliefs began to change, the model switched the positions of the Earth and the Sun therefore stating that the Earth and several other planets all instead moved around a central body, the Sun [1].

1.1.1 Kepler and Galileo's Laws

The heliocentric theory was then further developed by Kepler. He was the assistant of Tycho Brahe who painstakingly spent almost his entire lifetime recording planetary motion with exceptional precision, accurate to less than half a minute of arc, more than twenty times better than Copernicus' data. After his death in 1601, Kepler continued Brahe's observations and the theoretical interpretations of them. He found that Brahe's observations of Mars did not fit the Copernican model as it placed the orbit eight minutes of arc outside of that recorded by Brahe and Kepler - this was within Copernicus' limit of precision so the deviation was not seen in his data. When examining the paths the planets took Kepler found that instead of the orbits being perfectly circular as previously thought they were instead, to his surprise, slightly elliptical. The properties of ellipses had been known since the second century B.C, this led to Kepler's first law *"Planets move in elliptical paths, with the sun at one focus of the ellipse."* [2].

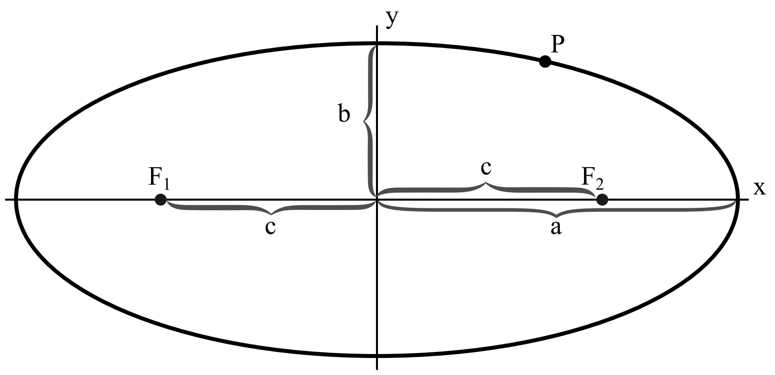

Figure 1.1: *Ellipse showing foci at F_1 and F_2.*

Kepler also derived two more laws known as his second and third laws of planetary motion. His second law was deduced from several incorrect assumptions but which at the time were reasonable to make. The first assumption was that the planets are pushed around the orbit by a force radiating from the sun. He also assumed that the strength of this force was inversely proportional to the planets distance from the sun and that the speed of that planet was proportional to that force, hence inversely proportional to the distance from the sun. These assumptions led to his idea that the time taken for a planet to travel along a segment of the orbit was proportional to area swept out by the straight line drawn from the sun to that planet. Despite the unconvincing reasoning for his assumptions to a modern physicist used to derive the law, it describes exactly the motion of any planet around the sun which is more commonly known today as the *"Law of Equal Areas"*. The law states that: *"During a given time interval a line drawn from the planet to the sun sweeps out an equal area anywhere along its path."* [2].

It wasn't until 10 years later in 1619 that Kepler published his third and final law of planetary motion. After his first and second laws were published Kepler still wasn't satisfied with his description of the solar system. He felt that there should be a connection between the motion of the different planets, currently there was no law relating the speeds, shape or periods of each one. Kepler was obsessed with the idea of a simplistic relationship between the properties of each orbit, which is why he stated after his discovery of his third law that he *"first believed [he] was dreaming and was presupposing the object of [his] search among the principles."* [3]. After an unthinkable amount of time and effort Kepler found his law he'd been searching for; as a relationship between the mean radius and the period of the orbit. He found that, in modern terms, the mean radius cubed was directly proportional to the orbital period squared: $\frac{T^2}{R^3} = Constant.$

Around the same time Galileo Galilei was investigating the concept of inertia with falling bodies. He proposed that a falling body would move with uniform acceleration in a vacuum, or in a medium with low resistance. He also deduced from his experiments that a body in motion would remain in motion unless acted upon by an external force, such as friction. This is his concept of *"inertia"* which states that any object in motion possesses such a property that causes it to remain in that state of motion, unless an external force acts upon it. The key to all of these concepts is the idea of force as a cause for motion, something that was crucial for the modern idea of the laws of motion.

It wasn't until around 50 years later that Sir Isaac Newton published his theory of universal gravitation (1687) which combined the planetary laws that Kepler defined and Galileo's laws of motion under the influence of gravity. Such a theory was a revelation in physics and is known as the *"first great unification"* of laws [4].

1.1.2 Newton's First Law of Motion

Newton's first law of motion was derived directly from Galileo's work stating that:

> *"Every body perseveres in its state of rest, or of uniform motion in a right line, unless it is compelled to change that state by forces impressed thereon [5]."*

which in modern terms means that in any *inertial frame of reference*, a body which is acted on by no net force at rest will remain at rest. If it is moving, it will continue to move with constant velocity. An *inertial frame of reference* is by definition a reference frame in which Newton's first law of motion is valid [6].

1.1.3 Newton's Second Law of Motion

Newton's second law of motion states:

> *"The alteration of motion is ever proportional to the motive force impressed; and is made in the direction of the right line in which that force is impressed [5]."*

which, again in modern terms, means that the force on any mass is directly proportional to the acceleration on such object:

$$a = F/m \tag{1.1}$$

where **F** is measured in Newtons (N), **a** is measured in meters per second per second (ms^{-2}) and **m** is measured in kilograms.

1.1.4 Newton's Third Law of Motion

Newton's third law of motion states that:

> *"To every action there is always opposed an equal reaction: or the mutual actions of two bodies upon each other are always equal, and directed to contrary parts [5]."*

this law is one of Newton's most well known in the modern day. Any object that exerts a force on a second body will receive a force equal in magnitude and opposite in direction from that body.

1.1.5 Newton's Law of Universal Gravitation

Using these laws Newton wanted to explain how and why the planets obeyed Kepler's Laws in terms of his theory of gravity. It was already known that Kepler's third law could be explained by an attractive force between the sun and the planet that was inversely proportional to their separation squared $F \propto r^{-2}$ which Newton attributes to in his book *Principia* [5]. A few other mathematicians at the time were also eager to unify the theories but Newton had the edge due to his ability to conceptualise the mechanical problems and describe their motions mathematically by various laws - as described above. More importantly he had already developed *infinitesimal calculus* but hadn't published it yet, this gave him the ability to test any type of force against Kepler's laws which led to him demonstrating that all three of them follow an inverse-square law for gravity as follows:

$$F = \frac{Gm_1m_2}{r^2} \tag{1.2}$$

where m_1 and m_2 are the masses of the two bodies, r is the separation between them and G is the Gravitational Constant.

This single equation allows us to calculate the force on any planet by summing for every other object in the solar system - then combining this with Newton's second law (1.1) the acceleration of the planet can be calculated, thus a change in velocity and displacement. This is the basis of how we can describe the motions of the planets.

1.2 N-Body Techniques

N-body techniques can be used to solve the n-body problem - predicting the motion of celestial bodies that interact with each other gravitationally. Using Newton's Universal Law of gravitation (1.2) it is possible to solve the problem analytically using a system of two bodies ($N = 2$) meaning the motions of both bodies is known exactly at any given time, this was solved by Kepler and showed that the general solution was an ellipse. However for $N \geqslant 3$ there is no closed form solution as the system becomes chaotic, meaning the problem can not be solved analytically but instead must be solved numerically. Such numerical methods are what make up N-body techniques each of which has a computational cost versus accuracy.

1.3 Project Outline

The aim of this project is to build a complete N-body simulator from first-physical principles, this will be used to achieve a full understanding of the techniques used to build such simulations and to quantify their limitations. The simulation will use the Euler method and then be extended to higher order methods such as the Runge-Kutta methods - the effectiveness of each method with varying time-steps will be investigated and then applied to a range of different scenarios.

The simulation will start with the Solar system and then be used to predict the Solar and Lunar eclipses and again measure the effectiveness of the simulation. Once this is complete the whole simulation will be transferred to a planetary system we know much less about, Trappist-1, and the transits of the system will be predicted and compared to observation.

Chapter 2

Theory

2.1 Numerical Methods

Numerical integration is used to evaluate a definite integral when there is no closed-form expression for it, it is also used to describe the method of finding numerical solutions of ordinary differential equations (ODEs). Many ODEs cannot be solved analytically, such as Newton's equations of motion, therefore we must use an iterative method that predicts the values of the function given initial conditions.

Newtons second law, Equation 1.1, can be combined with his law of universal gravitation, Equation 1.2, to produce a second order differential equation, in terms of the acceleration on any body, in vector form:

$$\vec{\boldsymbol{a}}(\vec{\boldsymbol{r}}) = -G \sum_{n=1}^{N} m_n \frac{\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_n}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_n|^3} \tag{2.1}$$

where $\vec{\boldsymbol{a}}$ is the resultant acceleration vector of the body due to the sum of gravitational forces from the other N bodies, m_n is the mass of the n_{th} body, and the vector $\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_n$ is the distance from each of the n_{th} bodies to the body being calculated.

Given initial conditions for the velocity and position of each body two ODEs can be constructed to be solved by numerical integration:

$$\frac{\mathrm{d}\vec{\boldsymbol{r}}}{\mathrm{d}t} = \vec{\boldsymbol{v}}\ , \ \frac{\mathrm{d}\vec{\boldsymbol{v}}}{\mathrm{d}t} = -G \sum_{n=1}^{N} m_n \frac{\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_n}{|\vec{\boldsymbol{r}} - \vec{\boldsymbol{r}}_n|^3} \tag{2.2}$$

2.1.1 Euler Method - First Order

The most basic of numerical integrators is the Euler method, it is a first order method derived by Euler in his book *Institutionum calculi integralis* [7]. It is most commonly used as the basis for higher order methods due to its limited accuracy and is rarely ever used for anything complex. It is a good starting point to show how numerical integration works.

For any given ODE, with initial conditions, of the form:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y(t)) \ , \ y(t_0) = y_0 \tag{2.3}$$

where $f(t, y(t))$ is a known function of the variables from the derivative, and t_0 and y_0 are the initial conditions.

Euler's method gives the iterative solution as follows:

$$y_{i+1} = y_i + f(t_i, y_i) \cdot \Delta t \tag{2.4}$$

where Δt is the size of the time step so that $t_{i+1} = t_i + \Delta t$ and $f(t_i, y_i)$ is the value of the known function at the previous values of t and $y(t)$.

Applying this method to the ODEs from Equation 2.2 yields the numerical approximations for the equations of motion as follows:[1]

$$\vec{\boldsymbol{v}}_{i+1} = \vec{\boldsymbol{v}}_i + \vec{\boldsymbol{a}}_i \cdot \Delta t \tag{2.5}$$

$$\vec{\boldsymbol{r}}_{i+1} = \vec{\boldsymbol{r}}_i + \vec{\boldsymbol{v}}_i \cdot \Delta t \tag{2.6}$$

Geometrically Equation 2.3 can be thought of as a method of calculating the gradient of the tangent at any point of the function, given that point is known - such as the initial condition given. This is shown in Figure 2.1 where the initial conditions are $(0, 0)$ and the blue line is the tangent to curve, which is already known in this instance for reference. At small scales this tangent is almost flush with the curve and follows the same direction, as shown on the right of Figure 2.1. Therefore if you were to move a small amount $(\mathrm{d}x)$ along the tangent you would be in a position very close to the true curve - this position would then have a different tangent and the same process can be applied to approximate each value of the curve.

[1]Note that the semi-implicit Euler method was used in practice which substitutes $\vec{\boldsymbol{v}}_i$ for $\vec{\boldsymbol{v}}_{i+1}$ when calculating the position $\vec{\boldsymbol{r}}_{i+1}$. It is a symplectic integrator and so yields much better results than the standard Euler method.

This is exactly how Euler's method, Equation 2.4, works: The next value is the previous value plus some multiple of the gradient calculated from the differential Equation 2.3 at that point.

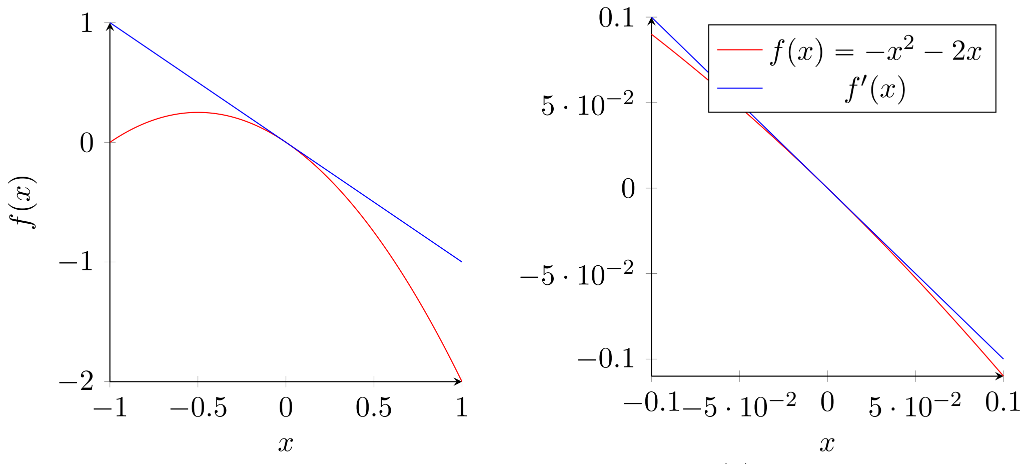

Figure 2.1: *Plots showing the tangent to the function $f(x)$ at different scales.*

Theoretically this method tends to the exact solution as the time-step tends to zero, but then the computational time tends to infinity as it is inversely proportional to the time-step:

$$\tau_c \propto \frac{1}{\Delta t} \tag{2.7}$$

where τ_c is the computational time taken and Δt is the time-step.

Given this relationship it's also possible to have an extremely small computational time given a large time-step, but this leads to poor accuracy. This can be seen geometrically on Figure 2.1 where the tangent tends away from the curve the further from the point of contact at $(0, 0)$ - so the larger the time-step the further along the tangent you move for each iteration. Also note that for functions with rapidly changing elements a smaller time-step is needed, for example at the top of a sine curve the tangent may be pointed upwards and then the next time-step it should be moving downwards again, as shown in Figure 2.2, so if too large of a time-step is used the approximation will keep overestimating the solution.

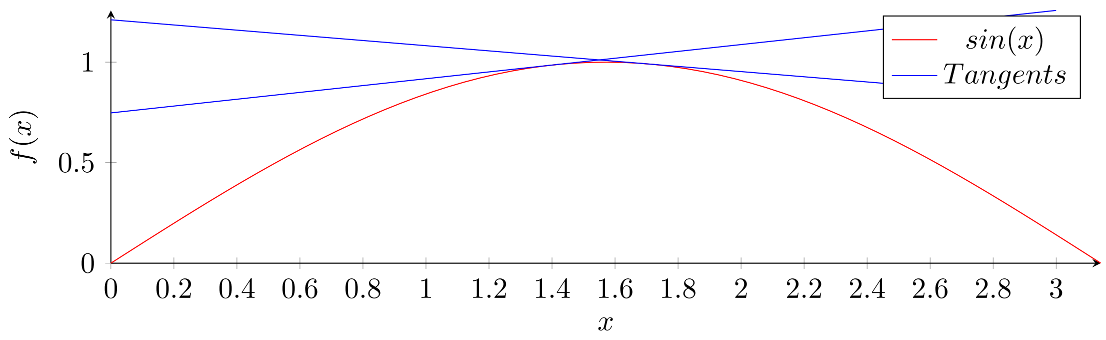

Figure 2.2: *Plots showing the tangents to the function $sin(x)$ in close proximity at $x = 1.4$ and $x = 1.7$.*

The slight error on each iteration accumulates over time and a large time-step will lead to large errors and too small of a time-step leads to extreme computational times - leaving only a small range of acceptable time-steps. Therefore it would be preferable to have a method that is more accurate using the same size time-step, this is possible using higher order methods such as Runge-Kutta.

2.1.2 Runge-Kutta - Higher Order Approximations

Higher order methods can be used to calculate a more accurate approximation of a function in the same time-step. The idea behind such methods is to take a weighted average of the gradient at different points of the time-step and using that gradient to make the full step in the same way as the Euler method, Equation 2.4. Note, the Euler method is a first order method and is part of the family of numerical methods called the *Runge-Kutta* methods.

The higher order methods can be derived from the Taylor expansion. The second-order expansion for $y(t + \Delta t)$ from Equation 2.3 is given as follows: [2]

$$y(t+\Delta t) = y(t) + \Delta t \frac{\mathrm{d}y(t)}{\mathrm{d}t} + \frac{\Delta t}{2}\frac{\mathrm{d}^2 y(t)}{\mathrm{d}t^2} \tag{2.8}$$

Next the equation needs to be in terms of the function of the derivative $f(t, y(t))$, this is simple for the first derivative of $y(t)$ but a bit more work for the second, by taking the multi-variable derivative of the first:

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}y}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}f(t,y) = \frac{\partial f}{\partial t}\frac{\mathrm{d}t}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t} = f_t(t,y) + f_y(t,y)f(t,y) \tag{2.9}$$

where $f_n(t, y)$ is the partial derivative of $f(t, y)$ with respect to n, giving the Taylor expansion in terms of $f(t, y)$ as:

$$\begin{aligned} y(t+\Delta t) &= y(t) + f(t,y)\Delta t + \frac{\Delta t}{2}\big[f_t(t,y) + f_y(t,y)f(t,y)\big] \\ &= y(t) + \frac{\Delta t}{2}\big[2f(t,y) + \Delta t f_t(t,y) + \Delta t f_y(t,y)f(t,y)\big] \\ &= y(t) + \frac{\Delta t}{2}f(t,y) + \frac{\Delta t}{2}\big[f(t,y) + \Delta t f_t(t,y) + \Delta t f_y(t,y)f(t,y)\big] \end{aligned} \tag{2.10}$$

Then using the Multivariate Taylor Expansion:

$$\begin{aligned} f(t+\Delta t, y+k) &= f(t,y) + \Delta t f_t(t,y) + k f_y(t,y) \\ f(t+\Delta t, y+f(t,y)) &= f(t,y) + \Delta t f_t(t,y) + f_y(t,y)f(t,y) \end{aligned} \tag{2.11}$$

[2]Note, at this point if we truncated the last term we would end up with the first order Runge-Kutta/Euler method.

Therefore giving:

$$y(t+\Delta t) = y(t) + \frac{\Delta t}{2} f(t,y) + \frac{\Delta t}{2} f(t+\Delta t, y + f(t,y)) \tag{2.12}$$

Which in the iterative form can be written as:

$$\begin{aligned} y_{n+1} &= y_n + \frac{\Delta t}{2}\big[f(t_n,y_n) + f(t_n+\Delta t, y_n + \Delta t f(t_n,y_n)\big] \\ y_{n+1} &= y_n + \frac{\Delta t}{2}(k_1+k_2) \end{aligned} \tag{2.13}$$

where $k_1 = f(t_n, y_n)$ and $k_2 = f(t_n + \Delta t, y_n + \Delta t f(t_n, y_n)$.

This is the second-order Runge-Kutta method and higher order methods can be derived in a similar way. The Fourth-order method is the most commonly used since it provides the best balance between the number of evaluations needed and the order of the method as shown in Table 2.1 - where the fifth order method requires two extra evaluations per time-step.

Evaluations per time-step	2	3	4	5	6	7	8	9	10	11
Runge-Kutta order	2	3	4	4	5	6	6	7	7	8

Table 2.1: *Efficiency of the Runge-Kutta methods [8].*

The iterative form of the fourth order method is then given as follows:

$$y_{n+1} = y_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.14}$$

where:

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{\Delta t}{2}, y_n + k_1\frac{\Delta t}{2}) \\ k_3 &= f(t_n + \frac{\Delta t}{2}, y_n + k_2\frac{\Delta t}{2}) \\ k_4 &= f(t_n + \Delta t, y_n + k_3\Delta t) \end{aligned} \tag{2.15}$$

Therefore if we apply this method to the ODEs from Equation 2.2 the 4th-order Runge-Kutta numerical approximations for the equations of motion are as follows:

$$\begin{aligned} \vec{\boldsymbol{v}}_{i+1} &= \vec{\boldsymbol{v}}_i + \frac{\Delta t}{6}\Big(\vec{\boldsymbol{k}}_{1_{v_{i+1}}} + 2\vec{\boldsymbol{k}}_{2_{v_{i+1}}} + 2\vec{\boldsymbol{k}}_{3_{v_{i+1}}} + \vec{\boldsymbol{k}}_{4_{v_{i+1}}}\Big) \\ \vec{\boldsymbol{r}}_{i+1} &= \vec{\boldsymbol{r}}_i + \frac{\Delta t}{6}\Big(\vec{\boldsymbol{k}}_{1_{r_{i+1}}} + 2\vec{\boldsymbol{k}}_{2_{r_{i+1}}} + 2\vec{\boldsymbol{k}}_{3_{r_{i+1}}} + \vec{\boldsymbol{k}}_{4_{r_{i+1}}}\Big) \end{aligned} \tag{2.16}$$

where the $\vec{\boldsymbol{k}}$ coefficients for the velocity are given as follows:

$$\begin{aligned}
\vec{\boldsymbol{k}}_{1_{v_{i+1}}} &= \vec{\boldsymbol{a}}(\vec{\boldsymbol{r}}_i) \\
\vec{\boldsymbol{k}}_{2_{v_{i+1}}} &= \vec{\boldsymbol{a}}\Big(\vec{\boldsymbol{r}}_i + \vec{\boldsymbol{k}}_{1_{r_{i+1}}}\frac{\Delta t}{2}\Big) \\
\vec{\boldsymbol{k}}_{3_{v_{i+1}}} &= \vec{\boldsymbol{a}}\Big(\vec{\boldsymbol{r}}_i + \vec{\boldsymbol{k}}_{2_{r_{i+1}}}\frac{\Delta t}{2}\Big) \\
\vec{\boldsymbol{k}}_{4_{v_{i+1}}} &= \vec{\boldsymbol{a}}\Big(\vec{\boldsymbol{r}}_i + \vec{\boldsymbol{k}}_{3_{r_{i+1}}}\Delta t\Big)
\end{aligned} \tag{2.17}$$

and for the positions:

$$\begin{aligned}
\vec{\boldsymbol{k}}_{1_{r_{i+1}}} &= \vec{\boldsymbol{v}}_i \\
\vec{\boldsymbol{k}}_{2_{r_{i+1}}} &= \vec{\boldsymbol{v}}_i \cdot \vec{\boldsymbol{k}}_{1_{v_{i+1}}}\frac{\Delta t}{2} \\
\vec{\boldsymbol{k}}_{3_{r_{i+1}}} &= \vec{\boldsymbol{v}}_i \cdot \vec{\boldsymbol{k}}_{2_{v_{i+1}}}\frac{\Delta t}{2} \\
\vec{\boldsymbol{k}}_{4_{r_{i+1}}} &= \vec{\boldsymbol{v}}_i \cdot \vec{\boldsymbol{k}}_{3_{v_{i+1}}}\Delta t
\end{aligned} \tag{2.18}$$

Note that the $\vec{\boldsymbol{k}}$ coefficients for each of the velocity and position depend on each other and so each iteration must be calculated together.

These equations provide all that is needed to perform a simple simulation of a gravitational N-body system and their implementation and applications will be discussed further in the next chapters.

2.1.3 Leapfrog Integrator

The Leapfrog integrator [9] is another method for numerically integrating differential equations. It is a symplectic integrator meaning that energy is conserved as long as the time-step is small enough, and so is stable for oscillatory motion. This is compared to the Euler and Runge-Kutta methods which don't conserve energy and allow the system to drift over time. It is also a second-order method but requires the same number of function evaluations per step as the Euler method which is first order. The method gets it's name due to the *"leapfrog"* nature of updating the positions and velocities at interleaved time points.

The equations take the following form:

$$\vec{\boldsymbol{v}}_{i+1/2} = \vec{\boldsymbol{v}}_{i-1/2} + \vec{\boldsymbol{a}}_i\Delta t \tag{2.19}$$

$$\vec{\boldsymbol{r}}_{i+1} = \vec{\boldsymbol{r}}_i + \vec{\boldsymbol{v}}_{i+1/2}\Delta t \tag{2.20}$$

This can be converted into an integer step [10] form by replacing the half integer steps with equivalent terms that only depend on integer steps:

$$\vec{\boldsymbol{v}}_{i+1/2} = \vec{\boldsymbol{v}}_i + \vec{\boldsymbol{a}}_i \frac{\Delta t}{2} \tag{2.21}$$

The position and velocity equations are then given as follows:

$$\vec{\boldsymbol{v}}_{i+1} = \vec{\boldsymbol{v}}_i + (\vec{\boldsymbol{a}}_i + \vec{\boldsymbol{a}}_{i+1}) \frac{\Delta t}{2} \tag{2.22}$$

$$\vec{\boldsymbol{r}}_{i+1} = \vec{\boldsymbol{r}}_i + \vec{\boldsymbol{v}}_i \Delta t + \vec{\boldsymbol{a}}_i \frac{\Delta t^2}{2} \tag{2.23}$$

2.1.4 4th Order Yoshida Integrator

The leapfrog integrator can be extended to higher orders using techniques from Haruo Yoshida. The method applies the leapfrog technique over a number of different time-steps and weights them in a way so that the errors cancel out. The 4th order Yoshida integrator is therefore defined as follows:

$$\begin{aligned}
\vec{\boldsymbol{r}}_i^1 &= \vec{\boldsymbol{r}}_i + c_1\, \vec{\boldsymbol{v}}_i\, \Delta t \\
\vec{\boldsymbol{v}}_i^1 &= \vec{\boldsymbol{v}}_i + d_1\, \vec{\boldsymbol{a}}\,(\vec{\boldsymbol{r}}_i^1)\, \Delta t \\
\vec{\boldsymbol{r}}_i^2 &= \vec{\boldsymbol{r}}_i^1 + c_2\, \vec{\boldsymbol{v}}_i^1\, \Delta t \\
\vec{\boldsymbol{v}}_i^2 &= \vec{\boldsymbol{v}}_i^1 + d_2\, \vec{\boldsymbol{a}}(\vec{\boldsymbol{r}}_i^2)\, \Delta t \\
\vec{\boldsymbol{r}}_i^3 &= \vec{\boldsymbol{r}}_i^2 + c_3\, \vec{\boldsymbol{v}}_i^2\, \Delta t \\
\vec{\boldsymbol{v}}_i^3 &= \vec{\boldsymbol{v}}_i^2 + d_3\, \vec{\boldsymbol{a}}(\vec{\boldsymbol{r}}_i^3)\, \Delta t \\
\vec{\boldsymbol{r}}_{i+1} \equiv \vec{\boldsymbol{r}}_i^4 &= \vec{\boldsymbol{r}}_i^3 + c_4\, \vec{\boldsymbol{v}}_i^3\, \Delta t \\
\vec{\boldsymbol{v}}_{i+1} \equiv \vec{\boldsymbol{v}}_i^4 &= \vec{\boldsymbol{v}}_i^3
\end{aligned} \tag{2.24}$$

Where:

$$\begin{aligned}
&\omega_0 \equiv -\frac{\sqrt[3]{2}}{2 - \sqrt[3]{2}}, \\
&\omega_1 \equiv \frac{1}{2 - \sqrt[3]{2}}, \\
&c_1 = c_4 \equiv \frac{\omega_1}{2}, c_2 = c_3 \equiv \frac{\omega_0 + \omega_1}{2}, \\
&d_1 = d_3 \equiv \omega_1, d_2 \equiv \omega_0
\end{aligned} \tag{2.25}$$

2.2 Eclipse & Transit Calculations

In this section the different applications of the simulation will be discussed including derivations for each of the methods so that they can be implemented later on in code. The first application that will be explored is the prediction of eclipses and later on this will be extended to predicting the transits of distant systems.

2.2.1 Condition for an eclipse

The condition for an eclipse is dependent on the position of the Moon which is used to decide if it's between or inline with both the Sun and the Earth. This will be achieved by modelling the area between the Sun and Earth as a conical frustum as seen in Figure 2.3.

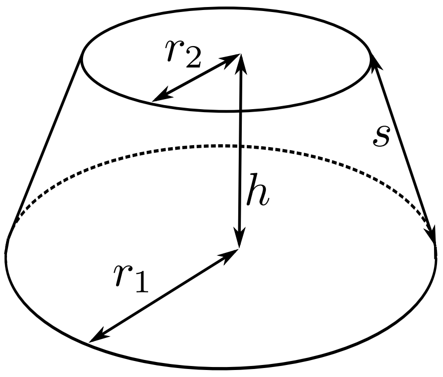

Figure 2.3: *Diagram of a Conical frustum [11].*

In this example r_2 would be the radius of the Earth $R_\oplus$ and r_1 the radius of the sun $R_\odot$ with h being the distance between the two. Therefore if the position of the moon lays in this area of the conical frustum then there is an eclipse (note if it is in the area behind the earth it will be a Lunar eclipse).

This problem can be solved by using some vector geometry and some simple trigonometry. To start, a line vector between the Earth and the Sun is defined as follows:

$$\vec{\mathbf{P}} = \vec{\mathbf{P}}_\oplus + \lambda \vec{\mathbf{P}}_{\oplus\to\odot} \tag{2.26}$$

where $\vec{\mathbf{P}}_\oplus$ is the position vector of the Earth and $\vec{\mathbf{P}}_{\oplus\to\odot}$ is the direction vector from the Earth to the Sun. A projection of the Moon's position, $\vec{\mathbf{P}}_{☾}$, onto Equation 2.26 can then be calculated - this gives the point along the line that is orthogonal to the direction vector $\vec{\mathbf{P}}_{\oplus\to\odot}$. This point is calculated as follows:

$$\vec{\mathbf{P}'} = \frac{(\vec{\mathbf{P}}_{☾} - \vec{\mathbf{P}}_\oplus) \cdot \vec{\mathbf{P}}_{\oplus\to\odot}}{\left\|\vec{\mathbf{P}}_{\oplus\to\odot}\right\|^2} \vec{\mathbf{P}}_{\oplus\to\odot} + \vec{\mathbf{P}}_\oplus \tag{2.27}$$

Using the point $\vec{\mathbf{P}}'$ the radius, $R_{\mathbf{P}'}$, along the conical frustum at that point can be calculated - this is the radius the Moon must be inside in order for there to be an eclipse.

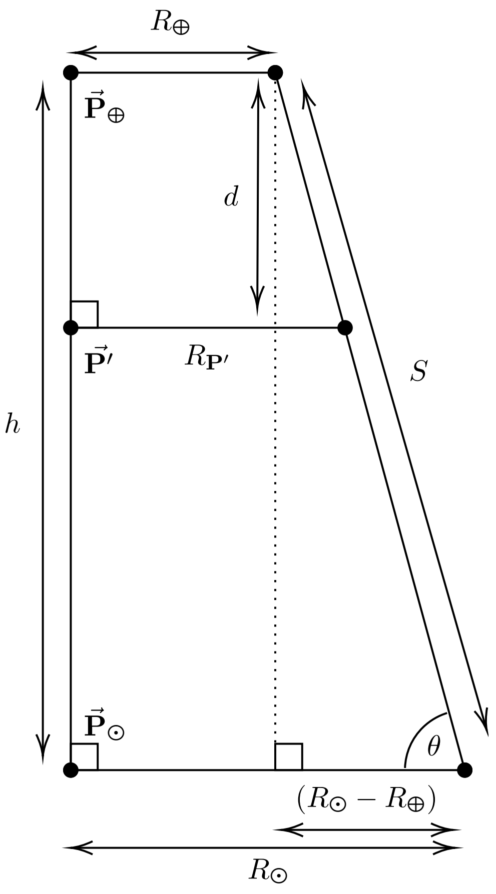

Figure 2.4: *Cross section of the conical frustum formed between the Earth and the Sun.*

Then using some simple trigonometry the value for $R_{\mathbf{P}'}$ can be calculated:

$$\tan(\theta) = \frac{d}{R_{\mathbf{P}'} - R_\oplus} \tag{2.28}$$

$$R_{\mathbf{P}'} = \frac{d}{\tan(\theta)} + R_\oplus \tag{2.29}$$

However this equation still includes a θ term so some substitutions need to be made:

$$\sin(\theta) = \frac{h}{S}\ ,\ \cos(\theta) = \frac{R_\odot - R_\oplus}{S} \tag{2.30}$$

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)} = \frac{h}{S} \times \frac{S}{R_\odot - R_\oplus} = \frac{h}{R_\odot - R_\oplus} \tag{2.31}$$

Therefore giving the final equation for $R_{\mathbf{P}'}$ as:

$$R_{\mathbf{P}'} = \frac{d}{h}(R_{\odot} - R_{\oplus}) + R_{\oplus} \tag{2.32}$$

where $d = \left|\vec{\mathbf{P}}' - \vec{\mathbf{P}}_{\oplus}\right|$ and $h = \left|\vec{\mathbf{P}}_{\odot} - \vec{\mathbf{P}}_{\oplus}\right|$.

Therefore the condition needed for the moon to be inside the area and thus an eclipse is:

$$\left|\vec{\mathbf{P}}' - \vec{\mathbf{P}}_{☾}\right| - R_{☾} < R_{\mathbf{P}'} \tag{2.33}$$

This is checking if the distance to the center of the moon minus it's radius is less than the radius $R_{\mathbf{P}'}$ or inside of the conical frustum.

2.2.2 Intensity of Eclipse Calculations

Now that the conditions for the eclipse are known the intensity of it must be calculated. This can be achieved by taking the ratio of the area covered by the moon against the total area of the circle from the conical frustum where $R = R_{\mathbf{P}'}$. Therefore the intensity is calculated using the area of the intersection between a circle of radius $R_{☾}$ centered at the Moon's origin and a circle of radius $R_{\mathbf{P}'}$ centered at $\mathbf{P}'$.

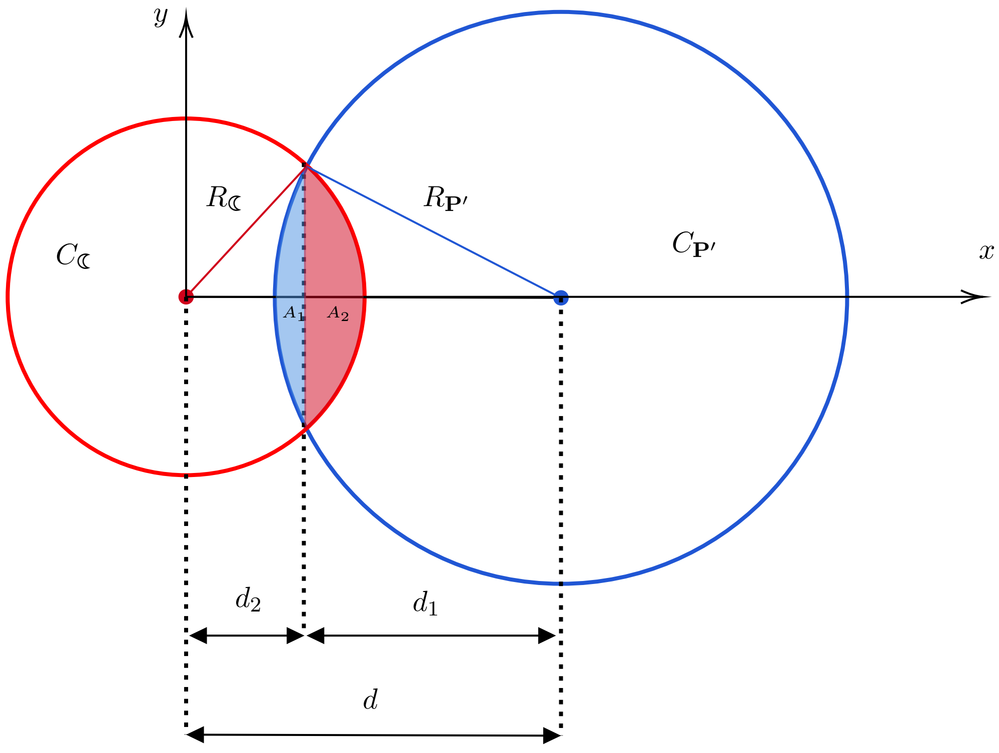

Figure 2.5: *Diagram showing the intersection of two circles with radius $R_{☾}$ and $R_{\mathbf{P}'}$.*

The value of d_1 from Figure 2.5 can be derived as follows using the equation of a circle:

$$\begin{aligned} R_{\mathbf{P'}}^2 &= y^2 + x^2 \\ R_{☾}^2 &= y^2 + (x-d)^2 \end{aligned} \tag{2.34}$$

Combing both equations from Equation 2.34 and solving for x, assuming $d_1 = x$ and $d_2 = d - d_1$, gives the following:

$$d_1 = \frac{d^2 - R_{☾}^2 + R_{\mathbf{P'}}^2}{2d} \tag{2.35}$$

where d is the distance $\left|\vec{\mathbf{P'}} - \vec{\mathbf{P}}_{☾}\right|$.

To calculate the areas A_1 and A_2 the formula for a circular segment must be used. To start, the angle enclosing both the segments in Figure 2.5 will be given as θ_i and the radii as R_i so that the arc length of the segment is:

$$s_i = R_i\theta_i \tag{2.36}$$

and the distance d_i is given by:

$$d_i = R_i \cos\left(\frac{1}{2}\theta_i\right) \tag{2.37}$$

The length a of the chord is also given as:

$$\begin{aligned} a_i &= 2R_i \sin\left(\frac{1}{2}\theta_i\right) = 2d_i \tan\left(\frac{1}{2}\theta_i\right) = 2\sqrt{R_i^2 - d_i^2} \\ &= R_i \sin(\theta_i)/\cos\left(\frac{1}{2}\theta_i\right) \\ &= \frac{R_i^2}{d_i}\sin(\theta_i) \end{aligned} \tag{2.38}$$

The area A_i is then calculated as the area of the sector minus an isosceles triangle of length R_i and angle θ_i as follows:

$$\begin{aligned} A_i &= A_{sector} + A_{triangle} \\ &= \frac{1}{2}R_i^2(\theta_i - \sin(\theta_i)) \\ &= \frac{1}{2}(R_i s_i - a_i d_i) \\ &= R_i^2 \cos^{-1}\left(\frac{d_i}{R_i}\right) - d_i\sqrt{R_i^2 - d_i^2} \end{aligned} \tag{2.39}$$

Therefore giving the total area of intersection as follows:

$$\begin{aligned} A &= A_1 + A_2 \\ &= R_{☾}^2 \cos^{-1}\left(\frac{d_1}{R_{☾}}\right) - d_1\sqrt{R_{☾}^2 - d_1^2} + R_{\mathbf{P}'}^2 \cos^{-1}\left(\frac{d_2}{R_{\mathbf{P}'}}\right) - d_2\sqrt{R_{\mathbf{P}'}^2 - d_2^2} \end{aligned} \tag{2.40}$$

There are some limits on Equation 2.40 such as when the smaller circle is entirely encompassed by the larger circle the area of intersection is simply just its total area - and when they don't intersect at all the area is zero.

Condition	Area
$d \geqslant R_{\mathbf{P}'} + R_{☾}$	0
$d \leqslant R_{\mathbf{P}'} - R_{☾}$	$\pi R_{☾}^2$
All other	Equation 2.40

Table 2.2: *Conditions for Equation 2.40.*

Finally, to calculate the intensity of light blocked the ratio of the two areas is taken:

$$I_{blocked} = \frac{A}{\pi R_{\mathbf{P}'}^2} \tag{2.41}$$

So the total intensity ratio of the sun is given as:

$$\frac{I}{I_0} = 1 - I_{blocked} = 1 - \frac{A}{\pi R_{\mathbf{P}'}^2} \tag{2.42}$$

2.2.3 Modifications for Transits

When applying this method to a distant system where d and h are extremely large, so that $d \approx h$, some approximations can be made since:

$$\frac{d}{h} \approx 1 \tag{2.43}$$

Thus Equation 2.32 is modified so that:

$$R_{\mathbf{P}'} = R_\star - R_{planet} + R_{planet} = R_\star \tag{2.44}$$

This is useful since it removes the dependence of the system's distance from Earth whilst also reducing the number of calculations made.

Chapter 3

Code Implementations

3.1 Simulation Workflow

This section will outline the basic workflow required in order to set up a simple simulation. Starting with the data containing classes and how they are initialised, then how a system is defined in code and the different classes that need to be implemented and finally the corrections that are made to the simulation will be discussed.

All code is written in Java with the use of CSV and JSON files for data input/output. Note that essential classes will be included in the Appendix with a link to the full project.

3.1.1 Data Processing

The most crucial class for the simulation is the `Body.java` class, it holds all the information on each Body in the simulation and provides methods to interact with that information, here are the properties the class has:

```java
    /** BODIES PROPERTIES **/
    private transient Vector3D velocity, nextVelocity; // (A.U/day)
    private transient Vector3D position, nextPos, tempPos; //Position in Kilometers (A.U)
    private transient List<Body> bodies;  //List of bodies to interact with.
    private transient List<Body> exclusiveBodies; //List of bodies without this body.
    private transient Universe universe;  //Universe this Body belongs too.
    private transient HashMap<Integer, Vector3D> tempAccelMap, tempVeloMap, tempPosMap;
    public  transient TreeMap<Integer, Vector3D>  positions; //History of positions. Key - time.
```

Listing 1: *Properties of* `Body.java`

The class holds the information for the objects current position and velocity and stores a history of positions at each time interval to be used later. It also includes maps for temporary accelerations, velocities and positions which will be used when implementing the integrators.

The Body class is abstract and so a subclass must be created that implements the methods of the parent class which are:

```java
    /**
     * IMPLEMENTATIONS
     */
    public abstract Vector3D getInitialPosition();
    public abstract Vector3D getInitialVelocity();
    public abstract String getName();
    public abstract double getMass();
    public abstract double getGM();
    public abstract double getBodyRadius();
    public abstract TreeMap<Double, Vector3D> getJPLPositions();
    public abstract TreeMap<Double, Vector3D> getJPLVelocities();
    public abstract boolean isOrigin();
    public abstract LocalDateTime getStartDate();
```

Listing 2: *Implementations for Body.java*

This is useful as it reduces the amount of code repetition and enables different types of systems to be implemented as long as they can provide values for each of the abstract methods - for example each system may have the input data in different formats or require extra calculations to provide the initial conditions.

Each system can then use the class `BodyBuilder.java` class to create an instance of a body and set each of it's values. The class uses a builder pattern in order to create an instance and set each of the Bodies values:

```java
    public static BodyBuilder getInstance(){
        return new BodyBuilder();
    }
    private BodyBuilder(){}

    public BodyBuilder setName(String name){
        this.name = name;
        return this;
    }

    public BodyBuilder setInitPos(Vector3D initPos) {
        this.initPos = initPos;
        return this;
    }
```

Listing 3: *Builder pattern used in BodyBuilder.java*

Once all the values are set the `create()` method is called which creates an instance of `Body` and returns it as seen in Listing 4.

The builder pattern is useful since it provides a short and simple way to initiate the class making it easier to implement new systems later and results in cleaner code.

An example of how a Body is initiated is shown in Listing 6.

```
    public Body create(){
        Body body = new Body() {
            @Override
            public Vector3D getInitialPosition() { return initPos; }
            @Override
            public Vector3D getInitialVelocity() { return initVelocity; }

            @Override
            public boolean isOrigin() { return isOrigin; }
            @Override
            public LocalDateTime getStartDate() { return startDate;}
        };
        body.setColor(color);
        return body;
```

Listing 4: *Section of the* `create()` *method used in BodyBuilder.java*

```
Body body = BodyBuilder.getInstance()
                .setInitPos(initialPos)
                .setInitVelocity(initialVelocity)
                .setName(bodyName)
                .setPositions(positions)
                .setVelocities(velocities)
                .setStartDate(initDateTime)
                .create();
```

Listing 5: *How to initiate a Body using the BodyBuilder class.*

The next class that must be implemented is `UniverseBuilder.java`, this holds simple information about the system and is where the initialisation for each Body happens. It's an abstract class where it's only abstract method to be implemented is one that creates a list of Bodies.

```
public abstract class UniverseBuilder {

    protected String name;
    protected double dt;

    /**
     * SETTERS & GETTERS
     */
    public void setName(String name) {
        this.name = name;
    }
    public void setDt(double dt) {
        this.dt = dt;
    }

    public abstract List<Body> createBodies();
    public String getName() {
        return name;
    }
    public double getDt() {
        return dt;
    }
```

Listing 6: `UniverseBuilder.java`

This concludes the data containing classes that the simulation uses and the next section will discuss how a system is defined in the code.

3.1.2 Defining a System

To define a system the `Universe.java` class must be extended and it's methods implemented. The Universe class is where all the different aspects of the simulation are called and provides the methods to interact with it during and post simulation.

```java
    /**
     *  ABSTRACT METHODS
     **/
    public abstract List<Body> createBodies();
    public abstract String getName();
    public abstract double dt();
    public abstract double runningTime();
    public abstract int resolution(); //Number of steps between each point saved.

    //Called at the end of the simulation.
    protected abstract void onFinish() throws IOException;
    //Called every tick.
    protected abstract void loop() throws InterruptedException;
```

Listing 7: `Universe.java` *abstract methods.*

Therefore each system must provide the name of the *"Universe"* or system, a list of bodies in the system, the time-step Δt, the running time and a resolution which is simply just the number of steps saved to memory spaced by the resolution value - for example a resolution of 5 would mean 20 steps are stored for a running time of 100, this enables a simulation to run for a longer running time by storing less data in memory.

The methods `void loop()` and `void onFinish()` must also be implemented. The loop method gets called at every time-step, it is here where each system must manipulate each Body. The onFinish method is simply called after the simulation has finished and can be used to perform any data analysis on the system.

That is all a system needs to provide to produce a working simulation and the Universe class hides all of the initialisation and keeps track of the simulation. To start the simulation the Universe's `void init()` method must be called, and then the `void start()` method is called to begin the simulation:

```java
    Universe universe = new UniverseImpl();
    universe.init();
    universe.start();
```

The start() method includes most of the logic for the loop of the simulation as seen in Listing 9. It begins by creating a Runnable that includes a while() loop that is set to run until the "running" variable is set to false - this is so the simulation can be paused and also tells the loop to break once the simulation is finished. Inside the loop the `void loop()` method is called and then the total execution time is added to the total "cpuTime" and the current step is incremented by one. Checks are then made to see if one real-time second has passed and then information about the current state of the simulation is printed to the console. The final part of the loop checks if the current Universe time has completed by reaching the running time implemented by the system, if true it prints out the statistics for the simulation and calls the `void onFinish()` method implemented by the system as discussed in Listing 7.

The Runnable task is then run on a background thread as seen in lines 85-88. This is so multiple simulations can be run at the same time on a multi-core CPU.

```
39      public void start(){
40          lastTime = System.nanoTime();
41          Runnable task = () -> {
42              int loops  = 0;
43              performanceTracker.startTracker();
44
45              while(running){
46                  long now = System.nanoTime();
47
48                  //Try to perform a loop.
49                  try { loop(); }
50                  catch (InterruptedException e) { e.printStackTrace(); }
51
52                  cpuTime += (now-lastTime);
53                  lastTime = now;
54                  universeStep++;
55
56                  //Print to console every second the current state of the simulation.
57                  if(cpuTime > TimeUnit.SECONDS.toNanos(1)){
58                      System.out.println("Universe Time: "+(getUniverseTime())+" Days
                    ↪  "+(100*getUniverseTime()/runningTime())+"%");
59                      cpuTime = 0;
60                  }
61
62                  // Every 100 timesteps
63                  if(loops >= 100){
64                      loops = 0;
65                      energyShift.put(getUniverseTime(),getTotalEnergy());
66                  }
67                  loops++;
68
69                  //Simulation is finished
70                  if(getUniverseTime() >= runningTime()){
71                      performanceTracker.finishTracker();
72                      System.out.println("-------------------------");
73                      System.out.println("--Finished Simulation!--");
74                      System.out.println("-------------------------");
75                      MemoryCalculator.printMemoryUsed();
76                      performanceTracker.printStats();
77
78                      try { onFinish(); }
79                      catch (IOException e) { e.printStackTrace(); }
80                      running = false;
81                  }
82              }
83          };
84
85          Thread backgroundThread = new Thread(task);
86          backgroundThread.setDaemon(false);
87          backgroundThread.setName("Background Thread");
88          backgroundThread.start();
```

Listing 8: `void start()` *method from* `Universe.java`

3.1.3 Corrections

During the simulations post and initialisation phase some corrections are made to the system. The first correction is made during the initialisation of the Universe class. For any given N-body system being simulated there will be a drift in the direction of the total momentum of the bodies, to correct for this the average velocity of the system must be subtracted from every body:

$$\begin{aligned} \langle \vec{\mathbf{v}} \rangle &= \frac{\sum \rho_i}{\sum m_i} \\ \vec{\mathbf{v}}_{i_{corrected}} &= \vec{\mathbf{v}}_i - \langle \vec{\mathbf{v}} \rangle \end{aligned} \tag{3.1}$$

This is implemented in the `MomentumCorrector.java` class as follows:

```java
        Vector3D momentum = Vector3D.ZERO;
        double totalMass = 0;

        for(Body body : universe.bodies){
            momentum = momentum.add(body.getVelocity().multiply(body.getMass()));
            totalMass += body.getMass();
        }

        Vector3D velocity = momentum.multiply(1/totalMass);

        for(Body body : universe.bodies){
            body.setVelocity(body.getVelocity().subtract(velocity));
        }
```

Listing 9: `MomentumCorrector.java`

Note this makes use of the Vector3D class which will be discussed in the next section.

The next correction is made when processing the data for output, since the system will naturally rotate around it's centre of mass this leads to the data being difficult to interpret especially as the amount of bodies increases. Instead each system has an origin body assigned and this is the body that all other bodies coordinates are with respect to.

This is achieved very simply using the Universe's `Vector3D getOriginBody()` method which sets each bodies position relative to the origin at the given time.

```java
body.positions.forEach((time, point3D) -> {
    Vector3D origin = body.getUniverse().getOriginBody().positions.get(time);
    point3D = point3D.subtract(origin);
}
```

3.2 Integrators

In this section each of the Integrators from Section 2.1 will be implemented in code along with a discussion on how each of the velocities and positions are stored as vectors.

3.2.1 Vectors

The first essential class needed to implement the integrators and to store each bodies velocity and position is a vector class. This class will store the coordinates of the vector and provide utility methods to add, subtract, dot product, cross product, normalise, calculate magnitudes and rotate the vector. The vector class is called `Vector3D.java` and is based upon the `Point3D.java` class from the OpenJFX library [12]. It adds in some methods in order to be able to rotate a vector around an axis but the base class provided the basic methods for everything else.

A Vector can be initialised in two different ways:

```
Vector3D vector1 = Vector3D.ZERO;
Vector3D vector2 = new Vector3D(1.0,2.0,3.0);
```

Where `vector1` is a vector at (0,0,0) and `vector2` at (1,2,3). Vectors can then be manipulated as follows:

```
Vector3D vector3 = vector1.subtract(vector2);
Vector3D vector4 = vector1.add(vector2);
double dotProduct = vector3.dotProduct(vector4);
```

Noting that the utility methods return a new instance of a vector and don't directly manipulate the original vector.

3.2.2 Integrator Implementations

Now that the storage of positions and velocities are possible through the Vector3D class, each of the different integrators can be implemented. The design goal for the integrator system was to provide a quick and simple way to switch the "Universe" between each integrator and also provide an easy way to implement new integrators.

The first class that a new Integrator must implement then is the `Integrator.java` class seen in Listing 10. This provides a method for calculating the resultant acceleration vector on a given body using Equation 2.1, it then requires two methods to be implemented one to provide the name of the integrator and another that is called at each time-step.

```java
public abstract class Integrator {

    //Integrator can decide how to evolve the system at each step.
    public abstract void step(Universe universe);

    public abstract String getIntegratorName();

    //Calculates Acceleration of given body using the temp pos parameter, e.g index 1 = x1
    public Vector3D accel(Body body, int positionIndex){
        Vector3D accel = Vector3D.ZERO;
        for(Body body2 : body.getExclusiveBodies()) {
            Vector3D delta = body.getTempPos(positionIndex).subtract(body2.getTempPos(positionIndex));
            double distance = delta.magnitude();
            // a(t)
            double accelMagnitude = -(body2.getGMAU()) / (distance * distance * distance);
            accel = accel.add(delta.multiply(accelMagnitude));
        }
        return accel;
    }
}
```

Listing 10: `Integrator.java`

The methods of integration used in Section 2.1 all require the storage of multiple temporary positions, velocities and accelerations in order to perform each of the intermediate steps involved in the higher order methods. To achieve this the Body class includes 3 HashMaps to store a temporary vector at a given index which correspond to the k_i values of the integrator, this can be seen on line 9 of Listing 1. Note that `Vector3D accel()` takes the position index as an argument to calculate the acceleration of a body at the given temporary positions.

Therefore the first-order semi-implicit Euler method is implemented as follows:

```java
public class EulerIntegrator extends Integrator {

    @Override
    public void step(Universe universe) {
        double dt = universe.dt();

        for(Body body : universe.getBodies()){
            body.setTempPos(1,body.getPosition());                //x1
        }

        for(Body body : universe.getBodies()){
            Vector3D v = body.getVelocity().add(accel(body,1).multiply(dt));
            Vector3D x = body.getTempPos(1).add(v.multiply(dt));
            body.setNextVelocity(v);
            body.setNextPosition(x);
        }
    }

    @Override
    public String getIntegratorName() {
        return "Euler";
    }

}
```

Listing 11: `EulerIntegrator.java`

And then the 4th-order Runge Kutta method is implemented similarly making use of four temporary positions and velocities. Listing 12 shows the first loop of the step that calculates $a1$, $v2$ and $x2$ and stores them as temporary values for use in the next loop and to calculate the final value - note that $v1$ is simply just the body's current velocity which is the $\vec{\boldsymbol{k}}_1$ value for position and $a1$ is the $\vec{\boldsymbol{k}}_1$ value for velocity from Equations (2.18) and (2.17) respectively.

```java
        for(Body body : universe.getBodies()){
            Vector3D a1 = accel(body,1);
            Vector3D v1 = body.getVelocity();

            Vector3D x2 = body.getPosition().add(v1.multiply(dt/2));
            Vector3D v2 = body.getVelocity().add(a1.multiply(dt/2));

            body.setTempPos(2,x2);
            body.setTempVelocity(2,v2);
            body.setTempAccel(1,a1);
        }
```

Listing 12: *Section from* `RK4Integrator.java`

This pattern repeats for each of the $\vec{\boldsymbol{k}}_i$ values except in the last loop where the final calculation is made:

```java
        for(Body body : universe.getBodies()){
            Vector3D a4 = accel(body,4);
            Vector3D dx = sumRK4(body.getVelocity(), body.getTempVelocity(2), body.getTempVelocity(3),
            ↪  body.getTempVelocity(4)).multiply(dt);
            Vector3D dv = sumRK4(body.getTempAccel(1), body.getTempAccel(2), body.getTempAccel(3),
            ↪  a4).multiply(dt);
            body.setNextPosition(body.getPosition().add(dx));
            body.setNextVelocity(body.getVelocity().add(dv));
        }
```

```java
    private Vector3D sumRK4(Vector3D k1, Vector3D k2, Vector3D k3, Vector3D k4){
        Vector3D sumdx = Vector3D.ZERO;
        sumdx = sumdx.add(k1);
        sumdx = sumdx.add(k2.multiply(2));
        sumdx = sumdx.add(k3.multiply(2));
        sumdx = sumdx.add(k4);
        return sumdx.multiply((double) 1 / (double) 6);
    }
```

Listing 13: *Final calculation of the step from* `RK4Integrator.java`

This final loop is used to calculate the final acceleration value and then to perform the summation of each of the $\vec{\boldsymbol{k}}_i$ values for the positions and velocities using Equation 2.16. The final step is to call the Body's `void setNextPosition()` and `void setNextVelocity()` methods. This is done because the position and velocity cannot be set directly since that would affect the other bodies calculations in the loop. The implementation of `Universe` should deal with this and call the `void postUpdate()` method on each body after the step is complete.

3.3 Eclipse & Transit Implementations

In this section the implementations of both the eclipse and transit calculator is discussed and the differences between them.

3.3.1 Eclipse Calculator

The eclipse calculator implements the method given in Section 2.2.2 and iterates over each step of the simulation saved to memory and then for each step calculates whether or not an eclipse has occured. The class used for this is called `EclipseCalculator.java` and to use it `void findEclipses(Universe universe)` simply has to be called after the simulation has finished in the `void onFinish()` method of the Universe implementation. The eclipse calculator is specifically hard-coded to work with the Solar System, but it would be simple to modify it to a new system. It starts by extracting the Sun, Earth and Moon from the list of bodies in the system.

```
        Body sun = null, earth = null, moon = null;
        for(Body body : universe.getBodies()){
            if(body.isOrigin()) sun = body;
            if(body.getName().equals("Earth")) earth = body;
            if(body.getName().equals("Moon")) moon = body;
        }
```

and then loops over every step and calculates the eclipse conditions using the positions of the Sun, Earth and Moon at that step:

```
            for(Integer step: sun.positions.keySet()) {
                //Simulated data
                Vector3D MoonPos = moon.positions.get(step);
                Vector3D SunPos = sun.positions.get(step);
                Vector3D EarthPos = earth.positions.get(step);
                calc(step*universe.dt(), EarthPos, SunPos, MoonPos, sun, earth, moon, ConeRadius,
                ↪  EdgeOfMoonDist, Lambda, Area);
            }
```

Where the `void calc()` method takes the parameters of the current universe time, the current positions of each body, the instance of those bodies and then 4 `TreeMap<Double,Double>()` objects - these store the parameters from Section 2.2.1 for each step and can be used for debugging and graphical processes.

`void calc()` is the method where the eclipse algorithm described in Section 2.2.2 takes place. It starts by calculating the point $\vec{\mathbf{P}}'$ from Equation 2.27:

```java
        //Direction vector.
        Vector3D SunEarthVector = directionVector(EarthPos,SunPos);
        //Lambda < 0 means Moon is behind Earth from Sun's perspective.
        double lambda = (MoonPos.subtract(EarthPos).dotProduct(SunEarthVector)) /
        ↪  (SunEarthVector.dotProduct(SunEarthVector));
        //Point perpendicular to line
        Vector3D P = SunEarthVector.multiply(lambda).add(EarthPos);
```

Then the radius of the cone at the point perpendicular to the moon $R_{\mathbf{P}'}$ from Equation 2.32 is calculated as follows:

```java
        double SEdist = dist(EarthPos,SunPos).magnitude();
        double PEdist = dist(EarthPos, P).magnitude();
        // Radius of circle at point of cone perpendicular to Moon.
        double r1 = (PEdist / SEdist) * (sun.getBodyRadiusAU() - earth.getBodyRadiusAU()) +
        ↪  earth.getBodyRadiusAU();

```

Finally the total area of the eclipse can be calculated using Equation 2.40 and Equation 2.42 and the conditions from Table 2.2:

```java
        //Calculate Area of eclipse
        double r2 = moon.getBodyRadiusAU();
        double d = dist(MoonPos, P).magnitude();

        double A;
        if(d >= r1 + r2) { // No intersection = 0 area;
            A = 0;
        } else if(d <= r1 - r2){ // If moon is inside cone area it's just the moons total area.
            A = Math.PI*r2*r2;
        } else { // Otherwise it's the intersection area between the two.
            double d1 = (r1 * r1 - r2 * r2 + d * d) / (2 * d);
            double d2 = d - d1;
            A = (r1 * r1) * Math.acos(d1 / r1) - d1 * Math.sqrt(r1 * r1 - d1 * d1)
                   + (r2 * r2) * Math.acos(d2 / r2) - d2 * Math.sqrt(r2 * r2 - d2 * d2);
        }

        double areaRatio = (1 - (A) / (Math.PI * r1 * r1));
```

The end of the method simply stores the different properties into each TreeMap.

The calculator will then perform the same calculations on the "true" data-set to be compared against and then finally calculate the start and finish points of the eclipse using the `HashMap<Integer, EclipseInfo> calculateEclipseFeatures()` method. This method simply calculates the point after which the intensity drops below 100% and marks that as the point at which the eclipse starts, the same is then done to calculate the end of the eclipse. This method can be found at the end of `EclipseCalculator.java` at line 89.

3.3.2 Transit Calculator

The Transit calculator is very similar to the Eclipse calculator but makes some modifications including those discussed in Section 2.2.3.

The first change is how the calculator decides which body the transit is calculated for, noting that the modifications discussed in Section 2.2.3 remove the dependence on Earth and only requires the direction vector that points directly at it. Therefore the `void findEclipses(Universe universe)` becomes `TreeMap<Double, Double> findTransits(Universe universe, Body planet, Vector3D direction, boolean plot)` where the TreeMap returned is an ordered list of the intensity from the star at the given step due to that body. The rest of the method is the same and loops over each step to calculate the transit.

The next change is in the `void calc()` method which makes use of the modifications discussed in Section 2.2.3. First, the point $\vec{\mathbf{P}}'$ is calculated using the direction vector provided:

```
         //Lambda < 0 means Moon is behind Earth from Sun's perspective.
        double lambda = (planetPos.subtract(starPos).dotProduct(direction)) /
↪   (direction.dotProduct(direction));
        //Point perpendicular to line
        Vector3D P = direction.multiply(lambda).add(starPos);
```

The next modification is for the $r1$ value since it is now simply just the stars radius, and $r2$ is the planets radius, and then adding a condition for λ to be greater than zero - since for transits we only care about when the planet moves in front of the star not behind.

```
        // Radius of circle at point of cone perpendicular to Moon.
        double r1 = star.getBodyRadiusAU();
        double d = dist(planetPos, P).magnitude();
        double areaRatio = 1;

        if(lambda > 0) {
            //Calculate Area of eclipse
            double r2 = planet.getBodyRadiusAU();
```

The next addition to the class is a method that simply loops over each body in the system and calls the `findTransits()` method and stores the results into two separate Maps. One with the individual transit intensities and a second that stores the summation of all the intensities - it achieves this by summing the difference from full intensity for each transit and then offsets the final result by 1, where 1 is 100% intensity.

3.4 Data Output

3.4.1 CSV Files

To create CSV output files a new class was created called `CSVWriter.java` - this is a utility class that contains a range of methods to create different types of CSV files. The first method of the file is used to create a `PrintWriter` to output data to a given file:

```java
    private static PrintWriter getOutputFile(String filenamePath){
        try {
            Path path = Paths.get("outputs/"+filenamePath);
            Files.createDirectories(path.getParent());
           FileWriter file = new FileWriter(String.valueOf(path));     // this creates the file with the
           ↪  given name
            return new PrintWriter(file); // this sends the output to file
        } catch (IOException e) {
            System.err.println("File couldn't be accessed it may be being used by another process!");
            System.err.println("Close the file and press Enter to try again!");
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            try { reader.readLine(); } catch (IOException ex) { ex.printStackTrace(); }
            return getOutputFile(filenamePath);
        }
    }

```

A CSV file is then processed by printing comma separated values for each line, for example the method to write the position of a Body to file is as follows:[1]

```java
    // Resolution = Number of points to discard between each CSV record.
    public static void writeBody(Body body, int resolution) {
        PrintWriter outputFile = getOutputFile(body.getUniverse().getName()
        ↪  +"/"+body.getUniverse().getIntegrator().getIntegratorName() +"/"+body.getName()
        ↪  +"/"+body.getName()+"_data_" +body.getUniverse().dt()+".csv");

        if(outputFile != null) {
            // Write the file as a comma seperated file (.csv) so it can be read it into EXCEL
            outputFile.println("time (days), x, y, z, magnitude (A.U)");
            double dt = body.getUniverse().dt();

            // now make a loop to write the contents of each step to disk, one number at a time
            AtomicInteger points = new AtomicInteger(resolution);
            body.positions.forEach((time, point3D) -> {
                points.incrementAndGet();
                if (points.get() >= resolution) {
                    // comma separated values
                    Vector3D origin = body.getUniverse().getOriginBody().positions.get(time);
                    point3D = point3D.subtract(origin);
                    outputFile.println(time * dt + "," + point3D.getX() + "," + point3D.getY() + "," +
                    ↪  point3D.getZ() + "," + point3D.magnitude());
                    points.set(0);
                }
            });
            outputFile.close(); // close the output file
            System.out.println("Written CSV Data for: " + body.getName());
        }
```

Listing 14: *Section from* `CSVWriter.java`

[1]Note that this method makes use of the corrections from Section 3.1.3 to set the systems origin to the star's (or origin body's) position.

All other methods make use of the same basic principle to write eclipse, transit, χ^2 fits and other types of data.

An example CSV output is shown in Table 3.1:

Time (Days)	X	Y	X	Magnitude (A.U)
0	0.930661	0.358601	-1.61E-05	0.997359
1	0.924061	0.37453	-1.71E-05	0.997076
2	0.917185	0.390349	-1.82E-05	0.996795
3	0.910037	0.406052	-1.94E-05	0.996516
4	0.902616	0.421635	-2.07E-05	0.996239
5	0.894926	0.437094	-2.20E-05	0.995964

Table 3.1: *Example output using the CSVWriter class.*

3.4.2 Graphing

To analyse data on the fly and to help fix bugs in the program 2 graphing libraries were used. The first of which was the *Matplotlib for Java* [13] library which makes use of the widely used Matplotlib for Python. It provides a simple interface that converts the parameters into Python code that is then passed into the Matplotlib library to output an interactive graph all at runtime in Java.

To create a Plot the library first needs to know where the Python install is located:

```java
public static Plot getPlot(){
    PythonConfig config = PythonConfig.pythonBinPathConfig("...
    ↪  Programs\\Python\\Python37\\python.exe");
    return Plot.create(config);
}
```

A basic plot can then be made using the following code:

```java
Plot plt = getPlot();
plt.figure("Title");

List<Number>
        x = Arrays.asList(1,2,3,4,5),
        y = Arrays.asList(2,4,6,8,10);
String fmt = "ro";

plt.plot()
        .add(x,y,fmt)
        .label("label")
        .linestyle("solid");
plt.xlabel("X");
plt.ylabel("Y");
plt.legend().loc("upper right");

openPlot(plt);
```

The `void openPlot()` method simply calls the `plt.show()` in a background thread so that multiple plots can be opened at the same time. This produces the graph shown in Figure 3.1:

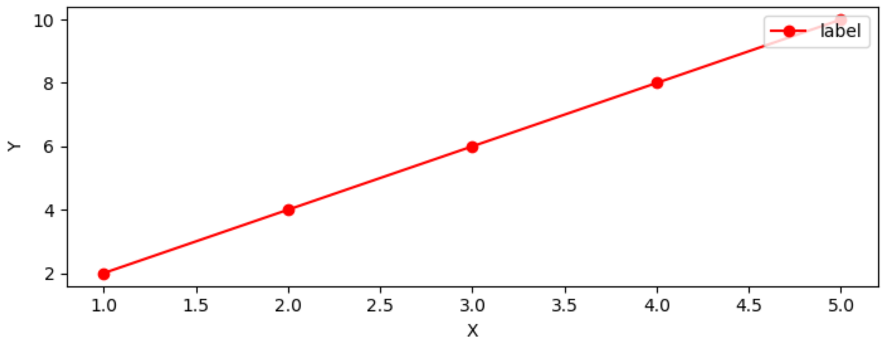

Figure 3.1: *Example plot using the Matplotlib for Java library.*

The Jzy3d [14] library was also used to make 3D plots. This is useful for making quick plots of the trajectories of each body from a system to debug code and also to see the influences of different time-steps and other changes to the system.

A basic plot can be created using the folowing code:

```
Coord3d[] coords = new Coord3d[body.positions.size()];
int i=0;
for(Vector3D vector3D : body.positions.values()) {
    coords[i] = Utils.fromPoint3D(vector3D);
    i++;
}
Scatter scatter = new Scatter(coords);
Chart chart = AWTChartComponentFactory.chart(Quality.Fastest, "newt");
chart.getScene().add(scatter);
ChartLauncher.openChart(chart, new Rectangle(200, 200, 1280, 720), body.getName());
```

And Figure 3.2 shows an example 3D plot using the Jzy3d library:

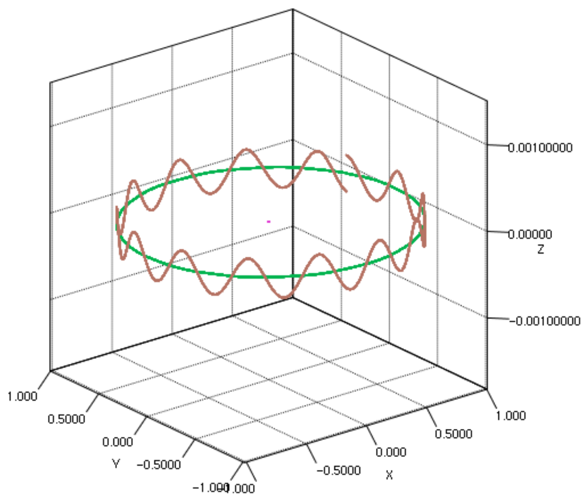

Figure 3.2: *Example plot of a 3 body system stretched along the Z axis.*

Chapter 4

Applications

In this chapter, using all the previous work from the preceding sections, the simulation will be applied to two different systems. The first being the Solar System which will be used to benchmark the simulation against other datasets/simulations and to investigate the best conditions for the system such as time-step and resolution of the output data, it will also use the eclipse The second system to be simulated is Trappist-one and will include a discussion on how it's initial conditions were calculated and the source of such data and then the transit calculator will be used to predict and compare against observational data.

4.1 Solar System

4.1.1 Data Collection

Since the Solar System is already well defined in terms of N-body simulations and lots of sources of data are readily available it makes for a suitable system to test and benchmark each integrator and the simulation as a whole. The data-set chosen to be used for comparison was JPL's HORIZONS System which provides highly accurate ephemerides for solar system objects including the Sun and the 8 planets (and dwarf planets) along with thousands of comets and asteriods [15].

The system allows for a range of different settings for data output and Table 4.1 shows the chosen configuration which was kept consistent throughout the whole simulation. These settings produced a file which contained an embedded CSV file that would provide the positions and velocities of the chosen body in the units of AU/day.

Setting	Value
Ephemeris Type	VECTORS
Coordinate Origin	Sun (body center) [500@10]
Time Span	Start=2019-10-15, Step=1 day
Table Settings	CSV format=YES
Display/Output	download/save (plain text file)

Table 4.1: *Table showing the settings used for the JPL HORIZONS System.*

4.1.2 Defining the Solar System

Once each of the selected bodies were chosen to be simulated an implementation of `Universe.java` must be created to provide the list of bodies to the simulation and then to provide a method to be performed every loop. To import the list of bodies each of the files from JPL HORIZONS needed to be parsed and information about the bodies properties provided. To achieve this a JSON file was used that included a value for the name of the system, the time-step and then an array of each body - which was comprised of it's name, the path to the JPL file, and then values for it's mass, radius and standard gravitational parameter (these were contained in the JPL files but were in an inconsistent format for each body so it was quicker to manually set these values.)

```json
{
  "name": "Solar System",
  "dt": 0.01,
  "JPLInfo": [
    {
      "name": "Earth",
      "filename": "horizon_data/earth.txt",
      "mass": 5.97219E24,
      "radius": 6371.01,
      "GM": 398600.435436
    }
  ]
}
```

Listing 15: *Example JSON file format containing one body.*

To read this data and deserialize it the GSON library from Google [16] was used. An abstract class called `UniverseBuilder.java` was created that contained variables for the name and time-step and then an abstract method to create the bodies - these variables get set by the gson library when a file is deserialized that contains the same field names as that class.

```java
protected String name;
protected double dt;
public abstract List<Body> createBodies();
```

For the Solar System an implementation called `UniverseBuilderJPL.java` was used that contained a list of a class called `JPLInfo.java` this is the class that contains the information about each body as seen in the JSON file in Listing 17. It provides methods to return the `BodyBuilder` class and an instance of `Body`. The BodyBuilder method was returned by a class that parsed the JPL file class `JPLHorizonsParser.java`

The JPL parser class would read the file into memory and then extract the CSV section from within it. The CSV file was contained within the two strings "$$SOE" and "$$EOE" and so the CSV string is read and extracted as follows:

```
URI pathname = Utils.getResource(filename);
Path path = Paths.get(pathname);
String content = Files.readString(path, StandardCharsets.US_ASCII);
String csvString = StringUtils.substringBetween(content, "$$SOE", "$$EOE");
```

To parse the CSV file the Apache Commons CSV library [17] was used and this would provide a list of each row as a CSVRecord:

```
CSVParser parser = CSVParser.parse(csvString, CSVFormat.DEFAULT);
List<CSVRecord> records = parser.getRecords();
```

Each row was parsed to read the date, time, position and velocities. And the first row was taken as the initial position and velocity. An instance of `BodyBuilder` was then be returned by using these values:

```
return BodyBuilder.getInstance()
    .setInitPos(initialPos)
    .setInitVelocity(initialVelocity)
    .setName(bodyName)
    .setPositions(positions)
    .setVelocities(velocities)
    .setStartDate(initDateTime);
```

And then the JPLInfo class would add the extra fields provided from the JSON file:

```
return JPLHorizonsParser.parse(filename,name)
        .setMass(mass)
        .setRadius(radius)
        .setGM(GM)
        .setOrigin(originBody);
```

The SolarSystem class then just had to return `builder.createBodies()` for the `createBodies()` method. The other methods to be implemented are show in Listing 7 where `getName()` and `dt()` are provided from the builder. The running-time and resolution methods are set based on how long the simulation is to be run and then the `loop()` and `onFinish()` methods are implemented as follows:

```
@Override
protected void loop() {
    integrator.step(this);
    bodies.forEach(Body::update);
}
```

```java
    @Override
    protected void onFinish() throws IOException {
        onFinish.forEach(Runnable::run);
    }
```

The loop method simply just calls the integrator to perform a step and then update the bodies and the onFinish method calls a list of runnables that are set before the simulation is run. Therefore a SolarSystem instance can be created as follows:

```java
    SolarSystem universe = new SolarSystem("/Body.json");
    universe.setRunningTime(0.25*365);
    universe.setIntegrator(IntegratorFactory.getRK4Integrator());
    universe.overrideTimeStep(0.0001);
    universe.addOnFinishListener(() -> {
        universe.getBodies().forEach(body -> CSVWriter.writeBody(body,10));
        Graph.plotTrajectory(universe,1);
    });
    universe.start();
```

Listing 16: *How to create a SolarSystem instance and start the simulation.*

This allows for a clean and simple way to create a simulation and also to run multiple simulations at the same time with different time-steps, running times, integrators and different onFinish listeners.

4.1.3 Eclipses

The Solar System simulation will be used to predict solar and lunar eclipses and measure the effectiveness of the method described in Section 2.2. The initial data will be compared to readily available data online such as "timeanddate.com" [18] which provides start and end times for both types of eclipses. A more thorough comparison will be made using a dataset from a NASA [19] - it provides the date, time, and location of every eclipse in five thousand years

4.2 Trappist-One

TRAPPIST-1 is a planetary system, located 12 parsecs away from the Solar system. It contains at least seven planets in orbit around a star which is 12 times less massive than the Sun. The initial discovery was made by TRAPPIST (the TRAnsiting Planets and PlanetesImals Small Telescope) and further planets were identified using the *Spitzer space telescope*, the *Very Large Telescope*, *UKIRT*, the *Liverpool Telescope* and the *William Herschel Telescope* [20].

4.2.1 Data Collection

Since the Trappist system doesn't have any accurate ephemerides for its planets their trajectories and starting positions must be inferred from the information we know due to the observations of the transits of the system. These include the planets orbital inclination, period, semi-major axis and eccentricities along with their mass and radius ratios. This information was found in the paper titled *"The nature of the TRAPPIST-1 exoplanets"* from 2018 [21].

The last piece of information needed to calculate the initial conditions is the phase of each of the planets orbits. Initially an estimate of this was made by calculating the percentage each planet had completed of it's orbit with respect to Planet 1C. This was calculated by taking the time difference between the midpoint of the transit of each planet and dividing by it's time period as seen in Table 4.2. The phase difference was then calculated by multiplying by 2π - making the assumption that the orbits were almost circular, since it is just an estimate.

Planet	First Transit	ΔT (Days)	P (Days)	Orbit (%)	Phase Diff
1c	7650.920787	0	2.42182	0	0
1b	7651.888331	-0.96754	1.51087	-0.640388	-4.023678401
1d	7653.944251	-3.02346	4.04961	-0.746606	-4.691065209
1e	7654.284258	-3.36347	6.09904	-0.551475	-3.465020917
1f	7652.972457	-2.05167	9.20559	-0.222872	-1.400348028

Table 4.2: *Calculation of the phase differences for each planet.*

4.2.2 Initial Condition Calculations

The initial conditions for each planet can now be calculated. First the position of the planet from it's semi-major axis, eccentricity, phase and inclination. The distance from the foci at the given phase first needs to be calculated:

$$r(\theta) = \frac{a(1-e^2)}{1+e\cos(\theta)} \tag{4.1}$$

Since the direction that Earth has been modelled to be at $(1, 0, 0)$ this means that the peak of each transit will occur along this direction too. So a vector is constructed at $(r(\theta), 0, 0)$ and then that vector is rotated around the z axis by θ. Then it is rotated by it's inclination along the y axis where 90° would be at $(1, 0, 0)$.

In Java this is implemented as follows:

```java
private Vector3D calculateInitialPosition(){
    if(originBody) return Vector3D.ZERO;

    //Calculate distance from foci (Star)
    double r = distFromFoci();
    Vector3D pos = new Vector3D(r,0,0);
    return pos.rotateAroundZ(startPhase).rotateAroundY(Math.toRadians(90-inclination));
}
```

```java
private double distFromFoci(){
    return semiMajorAxis*(1-eccentricity*eccentricity) / (1+eccentricity*Math.cos(startPhase));
}
```

The same method is applied to calculate the initial velocity of each planet. The magnitude of the velocity is calculated using the vis-viva equation:

$$\|\vec{\mathbf{v}}\| = \sqrt{GM\left(\frac{2}{r} - \frac{1}{a}\right)} \tag{4.2}$$

and then a vector is constructed as $(0, \|\vec{\mathbf{v}}\|, 0)$, where the initial vector is along the y axis so that the body is moving perpendicular to the starting position, it's then rotated along the z axis by the phase angle and then the y axis by the inclination in the same way the position vector is transformed:

```java
private Vector3D calculateInitialVelocity(TrappistBody origin){
    if(originBody) return Vector3D.ZERO;
    double GM = origin.calculateGM()*Constants.CONVERSIONS.GM_to_AU;
    double r = distFromFoci();
    double v2 = GM*((2/r)-(1/semiMajorAxis));
    double v = Math.sqrt(v2);

    Vector3D velo = new Vector3D(0,v,0); //Start with velocity at x=0 in y direction. z=0
    velo = velo.rotateAroundZ(startPhase).rotateAroundY(Math.toRadians(90-inclination));
    return velo;
}
```

4.3 Defining the Trappist System

To implement the Trappist system the `SolarSystem` class was extended by the class `TrappistSystem` and overridden a few methods, the first being the getBuilder() method which returns a different instance of `UniverseBuilder` that parses the Trappist JSON file which takes a slightly different format but is deserialized in the same way:

```json
{
  "name": "Trappist System",
  "dt": 0.001,
  "trappistBodies": [
    {
      "name": "Trappist 1a (Star)",
      "sunMassRatio": 0.089,
      "sunRadiusRatio": 0.121,
      "originBody": true
    },
    {
      "name": "Trappist 1b",
      "earthMassRatio": 1.017,
      "inclination": 89.56,
      "orbitalPeriod": 1.51087637,
      "semiMajorAxis": 0.01154775,
      "startPhase": -4.023678401,
      "earthRadiusRatio": 1.121,
      "eccentricity": 0.00622,
      "color": "#b04796",
      "periapsisArgument": 336.86,
      "meanAnomaly": 203.12,
      "originBody": false
    }
  ],
  "startDate": {
  "date": {"year": 2016, "month": 9, "day": 19},
  "time": {"hour": 10, "minute": 6, "second": 14, "nano": 0}
}
}
```

Listing 17: *Example JSON file format containing one body.*

The Trappist parser reads in the data from the JSON file, calculates the initial conditions and returns the list of bodies for the simulation to use. The direction vector that points in the direction of Earth is also defined in the `TrappistSystem` class and is set as $(1, 0, 0)$ so that the transits of the system match the observational data. The rest of the `TrappistSystem` is identical to the Solar System and so is initiated in the same way as Listing 16.

4.3.1 Transits

The Trappist System simulation will be used to produce a simulated data-set for the transits which will then be compared to the original observations made. The simulation will then be tuned so that the observational data is matched and the effectiveness of the method at reproducing the system will be analysed.

Chapter 5

Results & Analysis

5.1 Accuracy of Numerical Methods

In order to select the most optimal method of simulation there are multiple factors that must be considered. First, the accuracy of the integrator must be compared across a range of time-steps so that the deviation from the true value is minimal. This can then be compared for each integrator to see which produces the most accurate results at varying time-steps.

For any given system the size of the time-step is constrained to the body with the smallest orbital period. For example a 1 day time-step for the moon is almost 4% of it's orbit per step, but only 0.0017% of Neptune's orbit. Therefore the simulation is going to use the Moon's orbit as a benchmark for the accuracy of the system. If the integrator is sufficiently accurate at reproducing the smallest orbit of a system it should also be sufficient for all larger orbits (As long as the time-step of the system isn't too small).

Each integrator will also be tested for their computational time so that a balance between both the time taken to compute the simulation and it's accuracy can be made. The first test to be made is a comparison between the four integrators at a range of different time-steps. The first test was run between $\Delta T = 0.4$ days and $\Delta T = 0.05$ days and plots of the deviation from the JPL Horizons data-set against time can be seen in Figure 5.1 and then the total accumulated error against time-step Figure 5.2.

An extended plot of Figure 5.1 with a log y scale can be seen in Figure B.2.

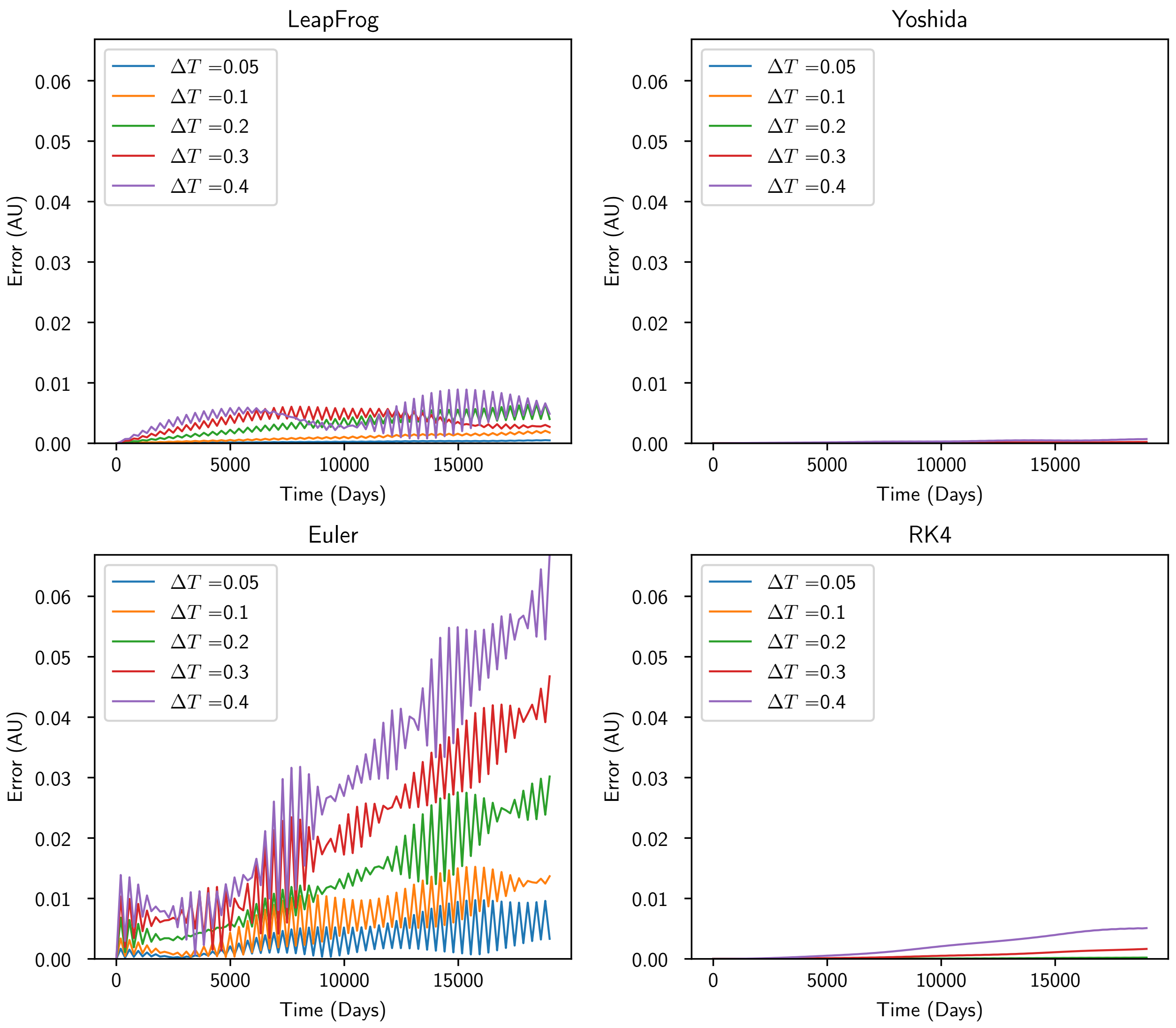

Figure 5.1: *Plot of error against time where the error is the deviance from the JPL Horizons data-set. The comparison is between the magnitude of the Moon's position vector relative to the sun.*

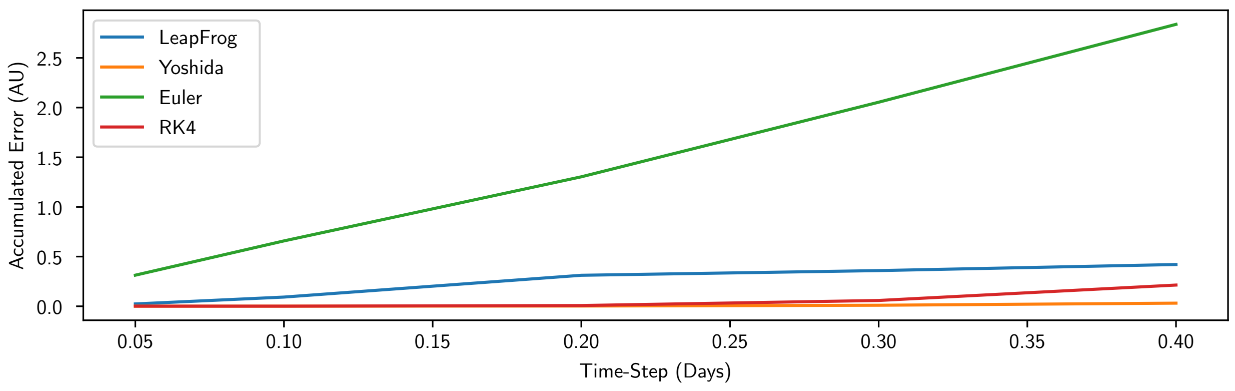

Figure 5.2: *Plot showing the total accumulated error against time-step for each of the 4 integrators being tested.*

5.1.1 Effects of Different Time-Steps

Figure 5.1 clearly shows the drawbacks of the Euler method and it's poor local truncation error, which is proportional to the square of the time-step $\mathcal{O}(h^2)$. This leads to the method quickly accumulating it's global error of $\mathcal{O}(h)$ over n iterations.

Figure 5.2 shows a linear relationship between the time-step and accumulated error for the Euler method which is exactly what is expected from the first-order method. Since the Euler method has such poor accuracy, and offers no benefits, it will be excluded in the next graph and the higher order methods will be explored at larger time-steps.

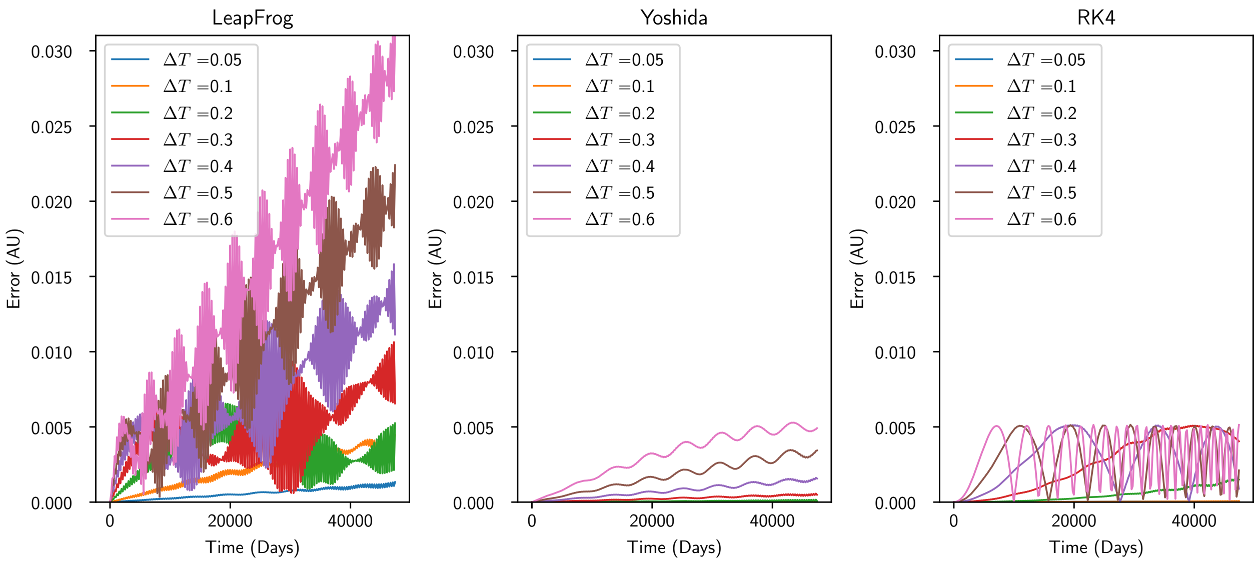

Figure 5.3: *Plot of error against time excluding the Euler method.*

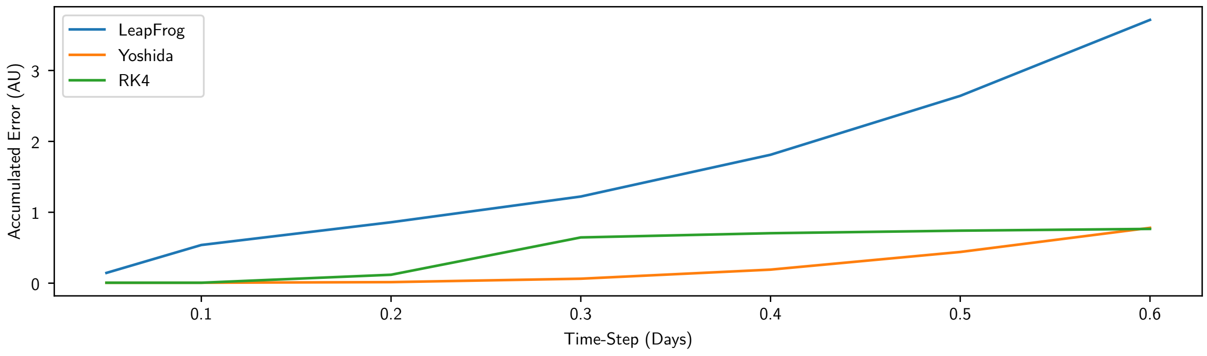

Figure 5.4: *Plot showing the total accumulated error over 120 years against time-step for the Runge-Kutta, LeapFrog and Yoshida integrators.*

Figure 5.3 shows the same error deviation graph as before excluding the Euler integrator. This time the simulation was ran for 120 years and up to $\Delta T = 0.6$ days. This time the second-order leapfrog method is the least accurate and accumulates it's error the quickest, by $\Delta T = 0.2$ it's not comparable to either of the 4th Order methods.

Also note that the Runge-Kutta method over/underestimates the position quicker than the Yoshida method, which leads to the large oscillations between around $0.005AU$ and 0 - this is the diameter of the Moon's orbit. It is clear that the 4th order methods are much more accurate at larger time-steps which is expected compared to the 2nd order leapfrog method.

If the error is split up into the 3 vector components most of the error is due to the Z-component. The Z-component is relative to the J2000 ecliptic meaning in the year 2000 it would be zero - so the ecliptic is always measured relative to what it was in the year 2000 on January 1st 12:00 UT. The precession of this ecliptic is dependant on all the other bodies in the system which is why this is the dominant error when compared to the JPL data-set. This will be explored further in Section 5.1.3.

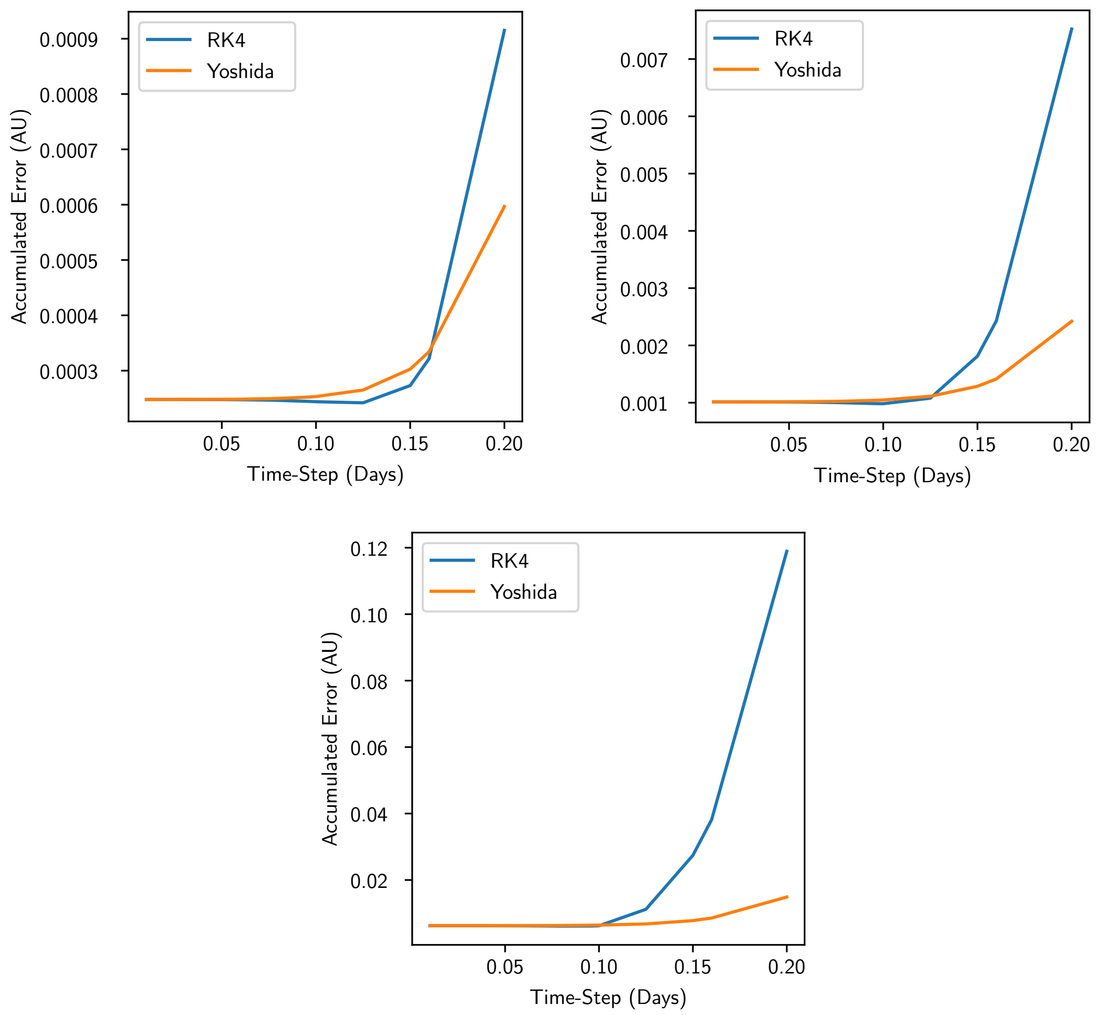

Figure 5.5: *Plot showing the total accumulated error against time-step for the Runge-Kutta and Yoshida integrators. Left to right the running times are 9600, 19200 and 48000 days.*

The next test will compare just the 4th order methods at small time-steps and also at varying running times. The simulation will be run from $\Delta T = 0.2$ down to $\Delta T = 0.01$

days each for 3 different running times of 9600, 19200 and 48000 days.

Figure 5.5 presents a couple of interesting results. The first being at long time-scale simulations the Yoshida method accumulates the least error at the same time-step but at smaller time-scale simulations the Runge-Kutta method has a period of lower accumulated error. Note that the two methods cannot be directly compared by their time-steps without taking into account their computational times which will be discussed in the next section.

The second interesting result from Figure 5.5 is the section between $\Delta T = 0.15$ and 0.05 on the first graph, where the error appears to increase with smaller time-steps. This effect can be seen in more detail in Figure 5.6 where the error increases and plateaus, this seems strange since smaller time-steps should produce less error.

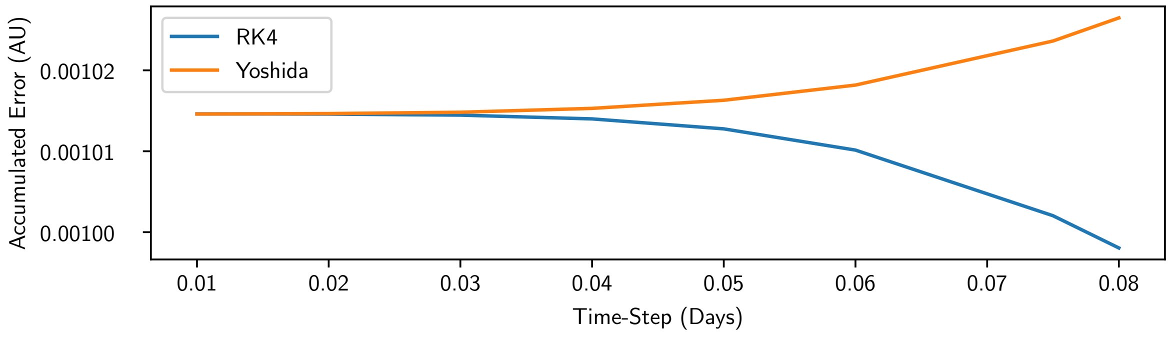

Figure 5.6: *Plot showing the increasing accumulated error of the Runge-Kutta as the time-steps decreases at small values.*

This phenomenon occurs due to rounding error from the use of double-precision arithmetic, which is accurate to 10^{-16}. This means that any information less than 10^{-16} is lost when that number is stored as a double in memory. This is a problem when working with small time-steps relative to a body's orbital period because it leads to minimal change in the body's position and velocity. If this change is of the order of 10^{-16} then that information will be lost and the body's position/velocity will suffer with reduced accuracy.

This means that making less calculations leads to less rounding error but less calculations means a larger time-step which leads to worse accuracy - this is the reason for the dip in accumulated error in Figure 5.5 and why each system has an optimal time-step for each running time. This also means that rounding errors occur in the 4th order Yoshida integrator at a smaller time-step than the Runge-Kutta method because it contains one less force evaluation even-though it is of the same order. And it is also the reason why higher order methods lead to less rounding error when the required accuracy is large or the time-scale is long.

5.1.2 Computational Times

In order to measure the viability of each of the integrators their computational times also need to be calculated. As previously discussed it is expected that higher order methods will have longer computational times than lower order methods since they make more force evaluations per step; smaller time-steps also lead to longer computational times as this requires more steps to be made for the same simulation running time.

The next set of results will show the computational times of each integrator at different running times of 4800, 9600, 19200 and 38400 days with time-steps ranging from $\Delta T =$ 0.03 to 0.3 days.

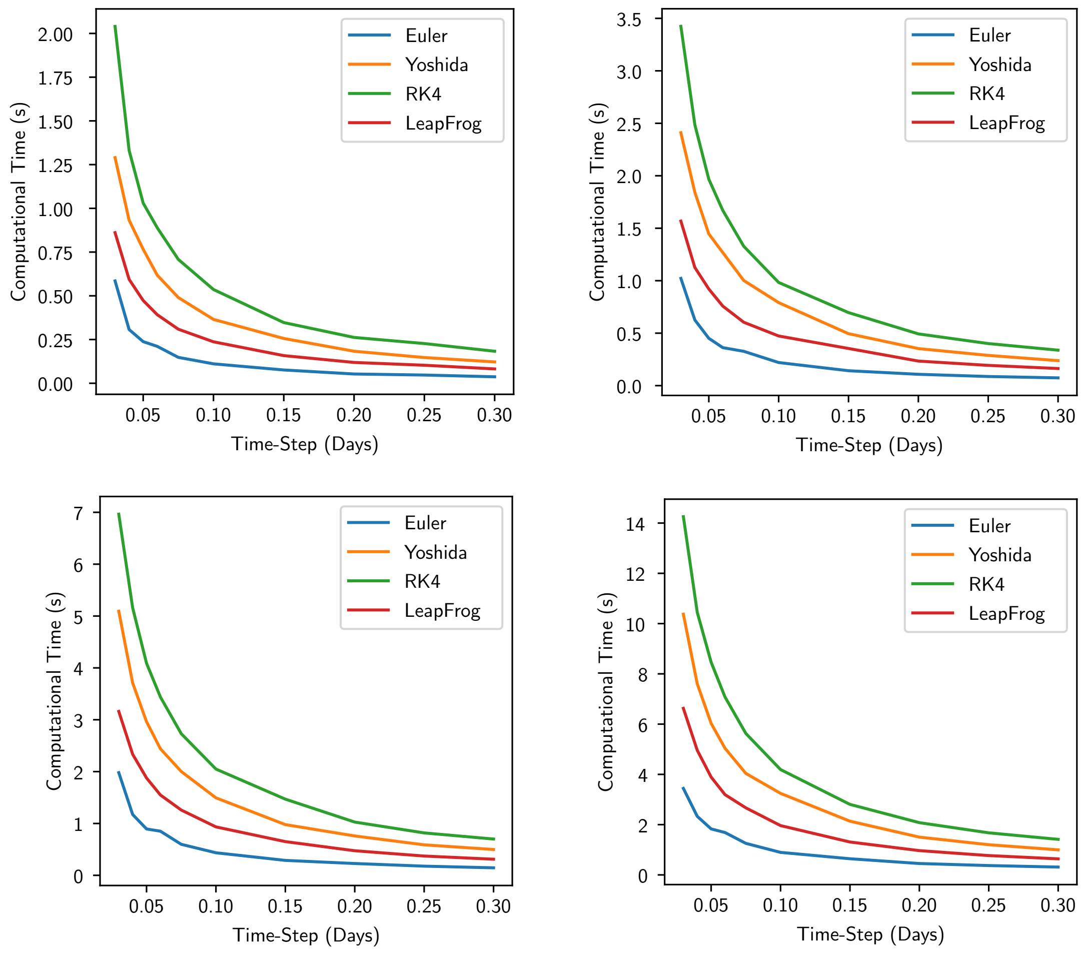

Figure 5.7: *Plots showing the computational times vs time-step for each of the 4 integrators. Left to right the running times are 4800, 9600, 19200 and 38400 days.*

Figure 5.7 shows clearly the inverse proportionality, discussed in Equation 2.7, between the computational time and time-step. It also shows the increased computational time required for the higher order methods that make more evaluations per step but also demonstrates the clear advantage those methods have at larger time-steps where the computational times are similar to those of the lower-order methods.

The computational times, however, are more useful when compared to the total accumulated error for each of the integrators at different time-steps. Figure 5.8 shows a plot of accumulated error vs the computational times from Figure 5.7 again with the time-steps ranging from $\Delta T = 0.03$ to 0.3 days and the same running times of 4800, 9600, 19200 and 38400 days.

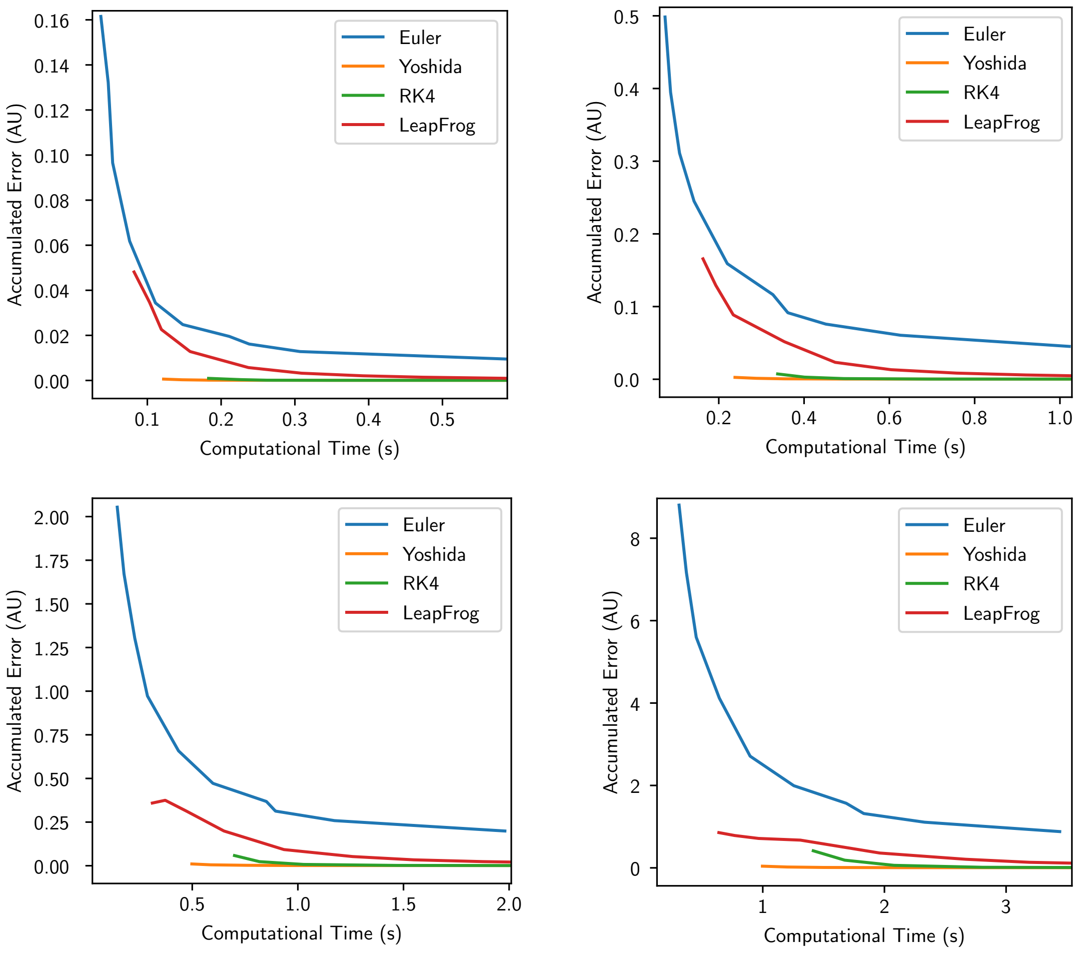

Figure 5.8: *Plots showing accumulated error vs computational time for each of the 4 integrators. Left to right the running times are 4800, 9600, 19200 and 38400 days.*

The best way to interpret this graph is to look at the vertical separation of each integrator, for example the Euler method has no computational time that produces the same or lower accumulated error of the 4th-order methods. And the leapfrog can produce the same accumulated error in a smaller computational time.

These results also confirm the benefits of the Yoshida integrator at longer running-times, as the Runge-Kutta method starts to accumulate more error with larger time-steps (shorter computational time). Not only is the Yoshida method more accurate at larger time-steps but it is also quicker. This is due to a combination of it's symplectic behaviour and the less force evaluations it makes per step. Also note that there is a period where the leapfrog integrator produces similar accumulated error to the Runge-Kutta method at the 38400 day running time.

The Yoshida integrator isn't always the most accurate integrator however, since there is still a small period where the Runge-Kutta achieves accuracy the Yoshida method never achieves. The additional accuracy is useful for short time-scale simulations since it requires a specific time-step but at longer time-scale simulations the Yoshida integrator may be preferable due to it's much greater performance at larger time-steps.

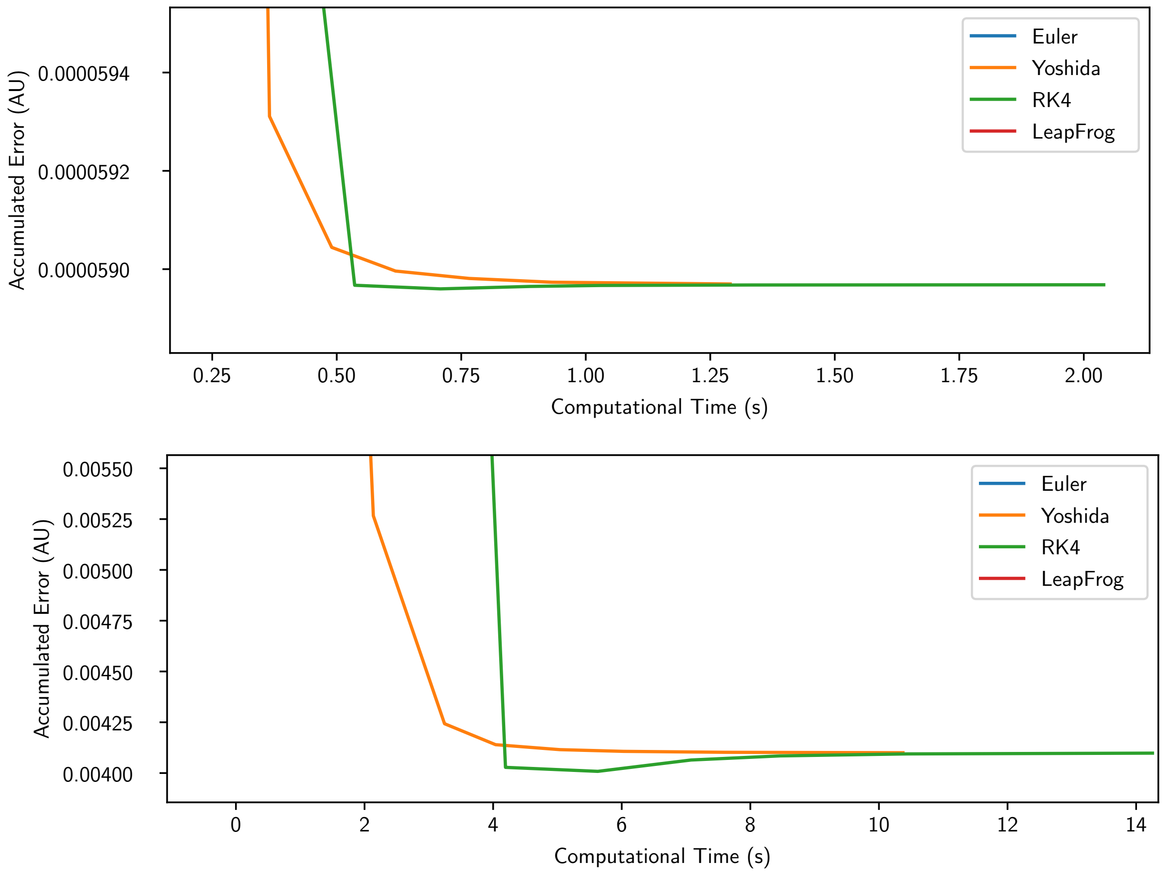

Figure 5.9: *Plot showing the period of reduced accumulated error for the 4th-order Runge-Kutta integrator vs the 4th-order Yoshida integrator. The running times are 4800 days and 38400 days top to bottom.*

5.1.3 Influences of Planets on Earth's Position

One of the aspects that an N-body simulation provides is the ability to change different properties of a system and see what effect it has compared to observational or previous results. When building the simulation and testing against the solar system there were some very noticeable effects when adding each of the other 7 planets.

If we start with the most basic two body system with the Sun and the Earth and plot the Z-coordinate with time from 1990 to 2010 and compare against the JPL Horizons data-set there is a large discrepancy between the two as seen in Figure 5.10

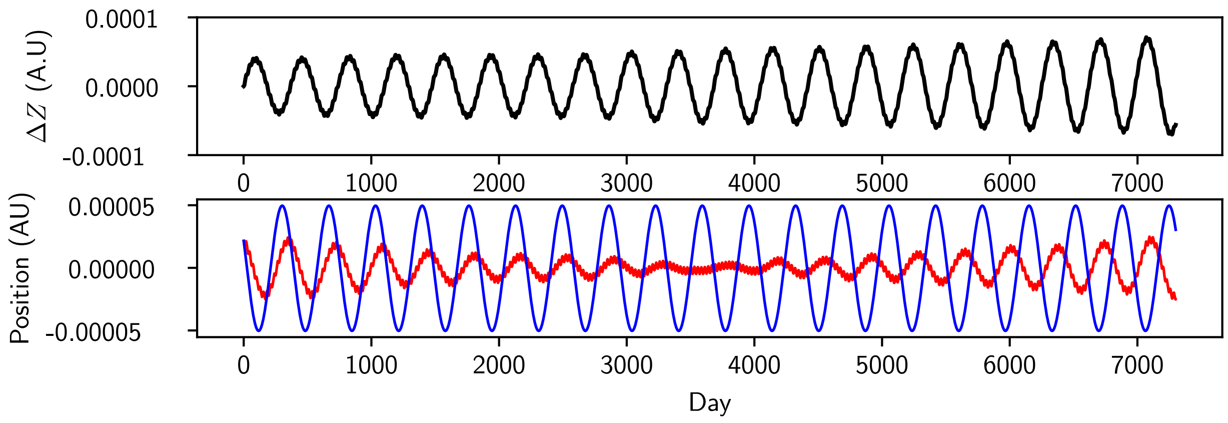

Figure 5.10: *Plot showing the two body system of the Sun and Earth compared to the complete JPL Horizons data-set from 1990 to 2010. Where red is the JPL data-set and blue simulated.*

The discrepancy shown is due to the J2000 ecliptic as previously discussed (It can be seen in the centre of Figure 5.10 where the Z-coordinate reaches zero) and without any other planets in the system the Earth's orbit doesn't precess at all. The effect each planet has on this precession will be explored next.

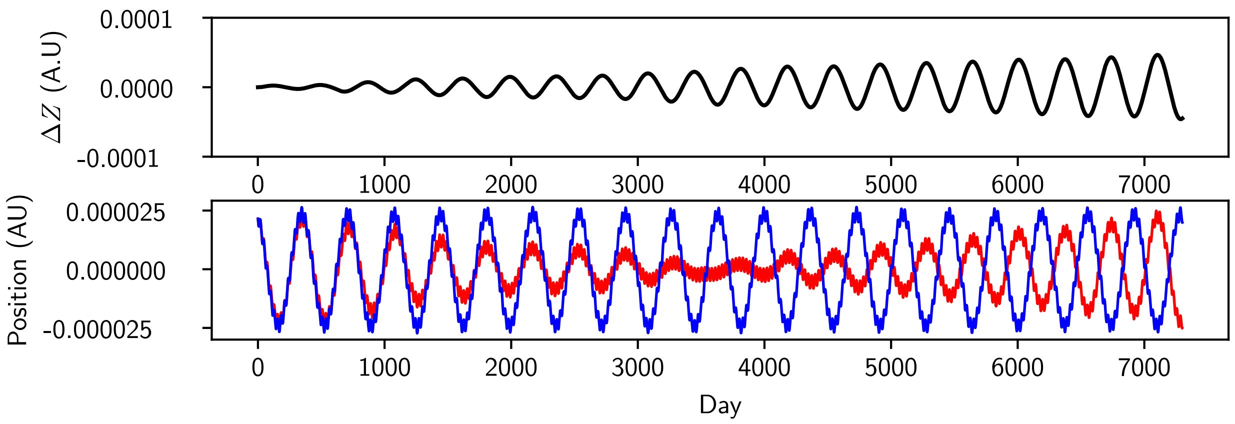

Figure 5.11: *Plot showing the system containing the Sun, Earth and the Moon compared to the complete JPL Horizons data-set from 1990 to 2010.*

First, if the moon is added the noisy curve is seen due to the Earth and the Moon's gravitational pull on each other seen in Figure 5.11 and the amplitude is much more in line but it ends up out of phase after the J2000 ecliptic.

Out of the three lighter planets only Venus produces a noticeable effect on Earth with Mars and Mercury having little to no effect at all seen in Figure 5.12.

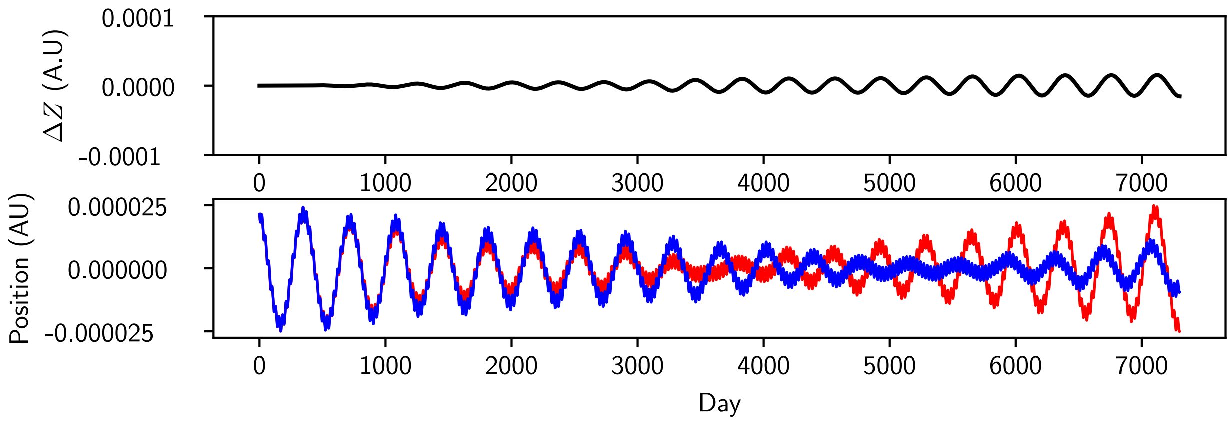

Figure 5.12: *Plot showing the system containing the Sun, Earth, Moon, Venus, Mars and Mercury compared to the complete JPL Horizons data-set from 1990 to 2010.*

Venus produces the decrease in amplitude expected until the ecliptic after which there is no increase, this must be caused by the larger planets.

Adding Uranus and Neptune produces no noticeable effects and Saturn and Jupiter are the planets that have the greatest effect on Earth with Jupiter having the most, which is understandable with their incredible mass. This is seen in Figure 5.13 and the residual plot shows the close match now all 8 planets are included.

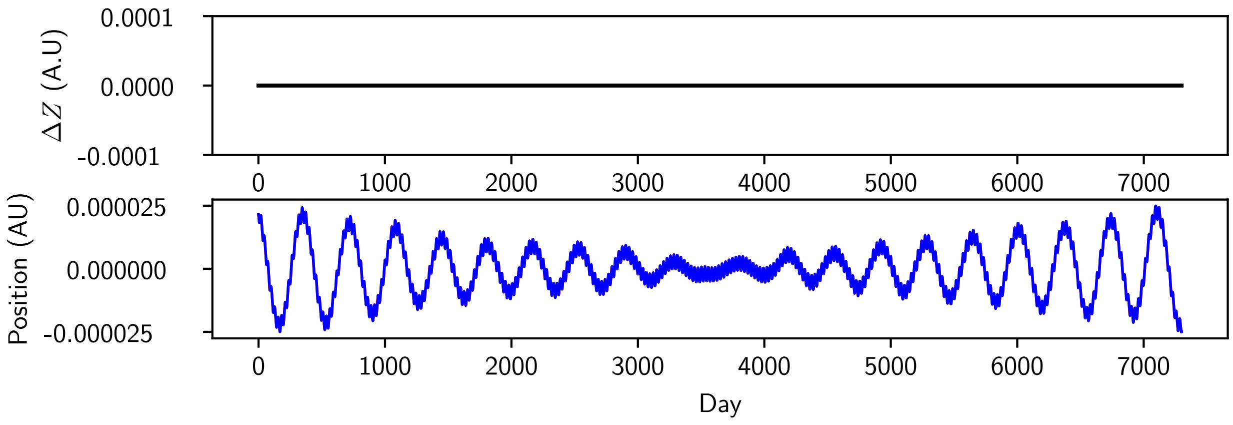

Figure 5.13: *Plot showing the system containing all 8 planets compared to the complete JPL Horizons data-set from 1990 to 2010.*

This means that the effect can be reproduced very closely with only the Moon, Venus, Saturn and Jupiter as part of the system including the Earth and the Sun:

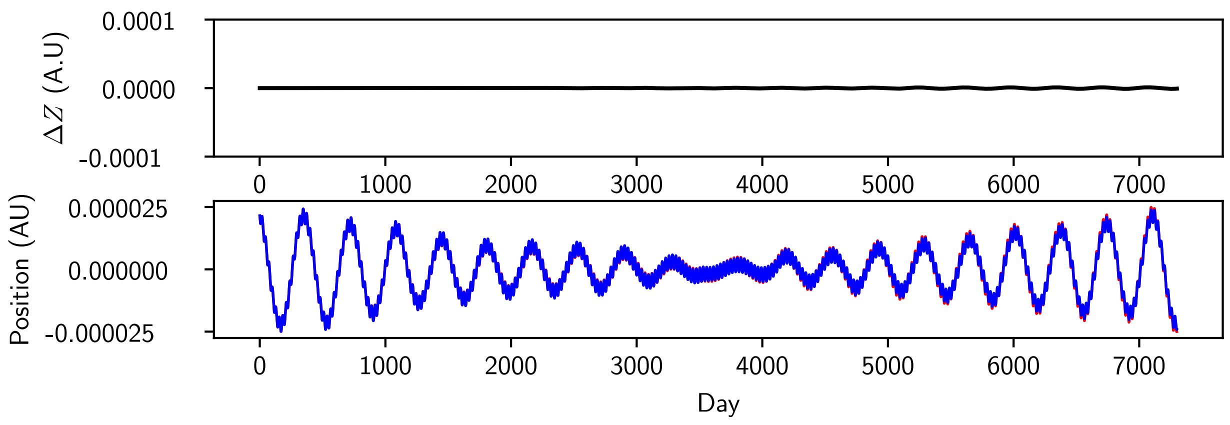

Figure 5.14: *Plot showing the system containing the Sun, Earth, the Moon, Venus, Saturn and Jupiter compared to the complete JPL Horizons data-set from 1990-2010.*

Figure 5.15 shows the Root Mean Square error of the residual plots that each additional planet provides - this provides the average magnitude from the true value. And Figure 5.16 shows the change in RMS as that each of the additional bodies provides, note that the Moon, Venus and Jupiter provide the largest reduction in RMS error.

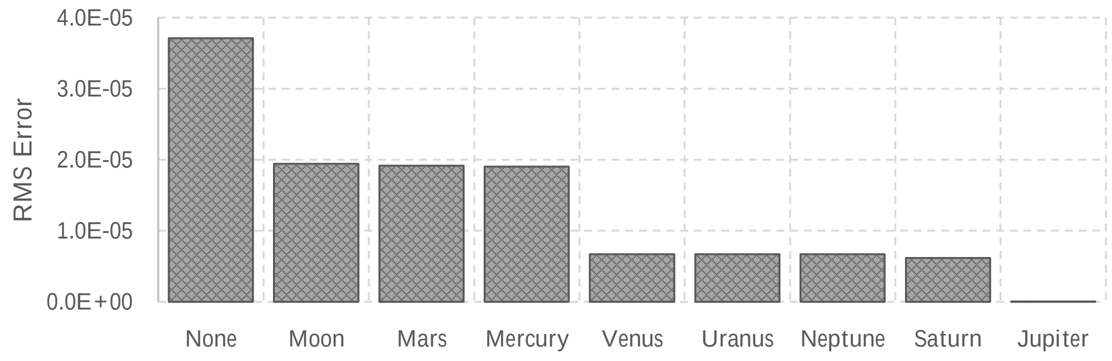

Figure 5.15: *Plot showing the reduction of RMS error as each additional planet is added to the two body system of the Earth and Sun.*

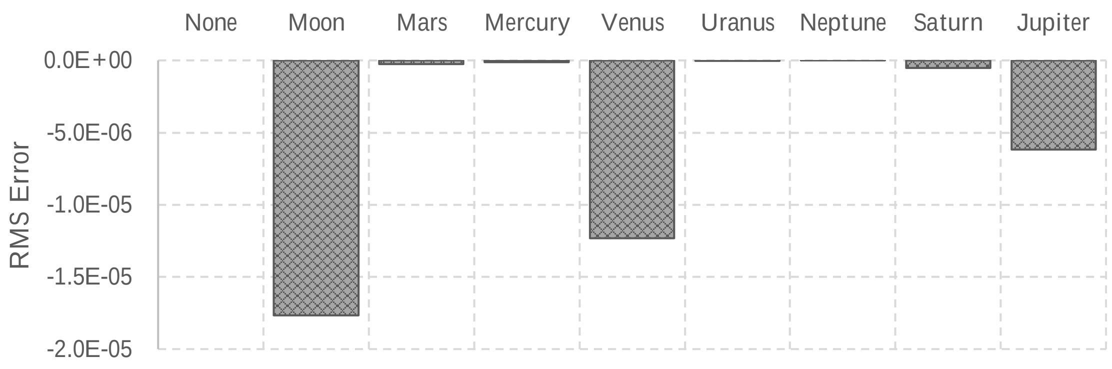

Figure 5.16: *Plot showing the individual reduction in RMS each planet provides.*

Accumulated Planet	RMS Error (10^{-6})	Change (10^{-6})
None	37.109	-
Moon	19.425	-17.685
Mars	19.158	-0.266
Mercury	19.041	-0.118
Venus	6.708	-12.332
Uranus	6.706	-0.002
Neptune	6.706	0.000
Saturn	6.169	-0.537
Jupiter	0.003	-6.166
Most Effective	0.478	-36.632

Table 5.1: *Table showing the accumulated RMS error as each additional planet is added to the two body system of the Earth and Sun. Along with the change in RMS error each individual planet provides as well as the most effective group of Jupiter, Saturn, Venus and the Moon*

5.1.4 Differences to JPL Data-set

From the results so far the effects of smaller time-steps and higher order integrators have been explored and shown to be effective at minimising error, however even with the smallest possible time-step and either the 4th order Runge-Kutta or Yoshida integrators there is still a slight error compared to the JPL Horizons data-set.

There are a few reasons for this and the most obvious is the limited accuracy of the 4th-order methods. The numerical integration used by JPL is much more complex and involves higher order integrators coupled with the method of special perturbations. This means that additional laws are implemented that affect the accelerations of each planet, for example the tidal forces on the Earth due to the moon are modelled which causes a change in it's acceleration, the shape and size of the Earth is also modelled rather than the spherical Newtonian assumption, a model of lunar librations is also included - these additions alone would likely improve the error accumulation for the moon.

Another law that is missing from this simulation is the addition of general relativity and the relativistic corrections made. JPL also include over 343 asteroids from the main asteroid belt [22]. All these differences and corrections lead to a much more accurate simulation which is the main reason for the discrepancy between results even at small time-steps with a 4th order integrator.

5.2 Eclipse Predictions

Now that the simulation has been benchmarked and the most accurate and efficient methods available are known, it can be used to predict the eclipses using the methods and code described in Section 2.2 and Section 3.3. An initial run of the simulation and analysing the data for eclipses using a time-step of $\Delta T = 0.01$ days, from 1990 January 1st to 2010 January 1st produces the graph seen in Figure 5.17.

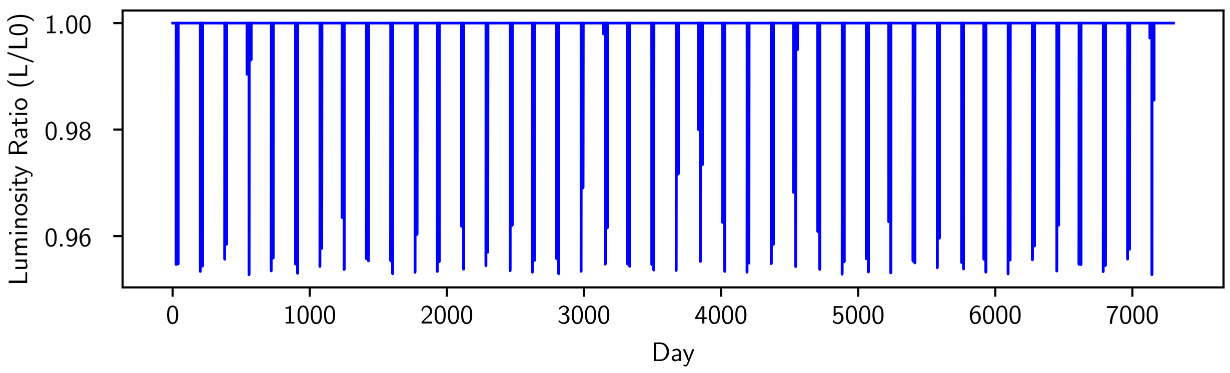

Figure 5.17: *Plot showing the Luminosity Ratio for both the Sun and Moon from 1990 to 2010.*

This shows the luminosity ratio for both the Sun and the Moon and the periodic nature of the dips in luminosity is a good sign the simulation is working. If the first dip is plotted on it's own as seen in Figure 5.18 it shows the features of an eclipse, the smooth descent as the partial eclipse begins as the edge of the Moon starts to move between the Sun and the Earth; and then the trough at the bottom when the entire moon is blocking a portion of the Sun's light and then the Moon's exit as the luminosity begins to increase again.

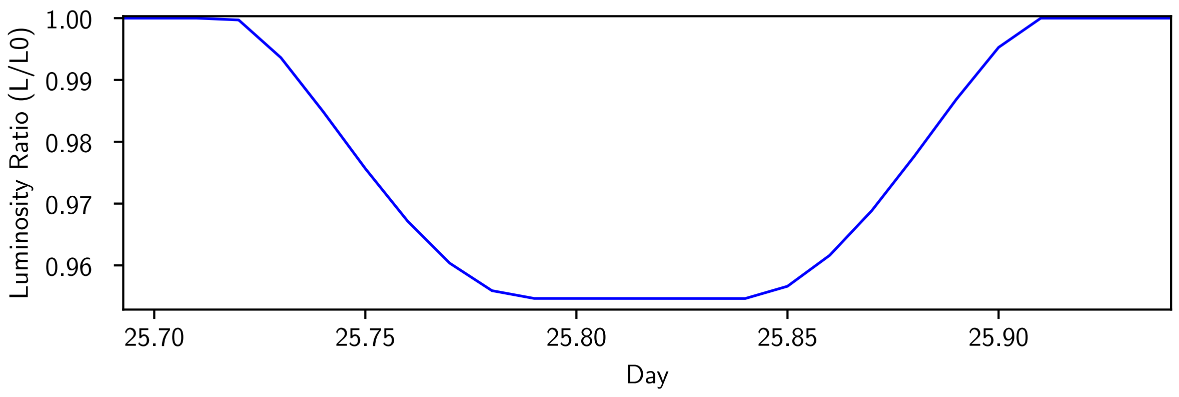

Figure 5.18: *Plot showing the simulated Luminosity Ratio for the Solar Eclipse of January 26th 1990.*

To measure the accuracy of the method the start and end of the eclipse was measured by taking the steps before and after the luminosity dips. These steps have an error equal to the time-step ΔT so for this simulation the start and end times are known to within 14.4 minutes. This error can be slightly improved by measuring the midpoint of the eclipse using both the start and end times and calculating the average of the two. This results in an error on the midpoint of $14.4/\sqrt{2}$ minutes or 10.2 minutes.

Start Date	Start Time	End Date	End Time	Duration (s)	Peak Time	Type
26/01/1990	17:02:24	26/01/1990	21:50:24	17280	19:26:24	SOLAR
09/02/1990	16:19:12	09/02/1990	22:04:48	20736	19:12	LUNAR
22/07/1990	00:28:48	22/07/1990	05:31:12	18144	03:00	SOLAR
06/08/1990	11:31:12	06/08/1990	17:02:24	19872	14:16:48	LUNAR
15/01/1991	20:52:48	16/01/1991	03:07:12	22464	00:00	SOLAR
30/01/1991	03:50:24	30/01/1991	08:09:36	15552	06:00	LUNAR
27/06/1991	01:55:12	27/06/1991	04:48	10368	03:21:36	LUNAR
11/07/1991	16:19:12	11/07/1991	21:50:24	19872	19:04:48	SOLAR
26/07/1991	16:48	26/07/1991	19:26:24	9504	18:07:12	LUNAR
21/12/1991	08:24	21/12/1991	12:43:12	15552	10:33:36	LUNAR

Table 5.2: *Table showing the simulated Solar and Lunar eclipse in 1990 and 1991.*

Table 5.2 shows each of the simulated eclipses with both the start and end times along with the midpoint peak for 1990 and 1991. When compared to data from timeand-date.com [18] for an initial comparison all start and date times are accurate to within one time-step which confirms the methods ability to predict eclipses.

Next a more in-depth test will be performed over a longer running-time and compared to a NASA data-set [19] that provides the midpoint date and time for thousands of eclipses from 2000 B.C to 3000 A.D. By plotting the difference in seconds between the simulated midpoints and the Nasa data-set the reliability of the method can be analysed as well as measuring the change in error over time.

NASA		Simulated		
Date	Time	Date	Time	ΔT (s)
26/01/1990	19:31:24	26/01/1990	19:26:24	-300
22/07/1990	03:03:07	22/07/1990	03:00	-187
15/01/1991	23:53:51	16/01/1991	00:00	369
11/07/1991	19:07:01	11/07/1991	19:04:48	-133
04/01/1992	23:05:37	04/01/1992	23:02:24	-193
30/06/1992	12:11:22	30/06/1992	12:14:24	182
24/12/1992	00:31:41	24/12/1992	00:36	259
21/05/1993	14:20:15	21/05/1993	14:24	225
13/11/1993	21:45:51	13/11/1993	21:50:24	273
10/05/1994	17:12:27	10/05/1994	17:16:48	261
03/11/1994	13:40:06	03/11/1994	13:40:48	42

Table 5.3: *Table containing the NASA and simulated peak times for Solar eclipses.*

Table 5.3 shows the midpoint date and time from the NASA data-set, between 1990 and 1994, along with the simulated values and then the difference in seconds between them. All values are within the midpoint error for the time-step of $\Delta T = 0.01$ days. A residual plot can be seen in Figure 5.19 which ranges from 1990 to 2030.

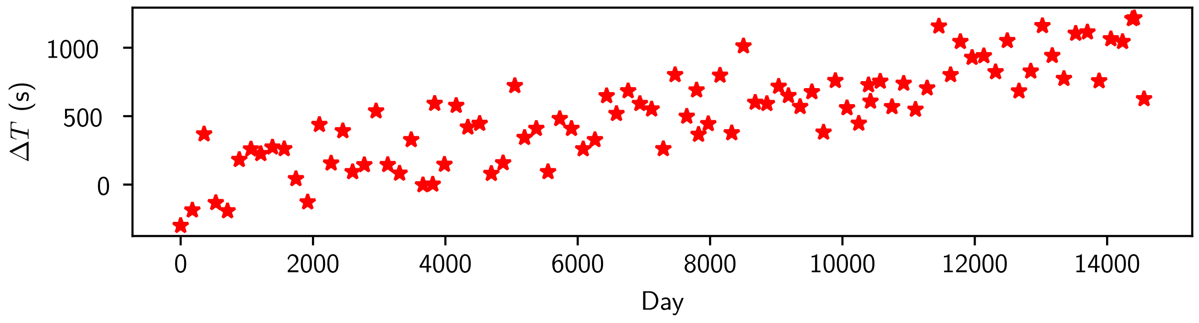

Figure 5.19: *Plot of ΔT against time for simulated Solar eclipses compared to the NASA data-set.*

There is a linear trend between ΔT and the simulated date as the accumulated error increases. This is expected due the inefficiencies and differences the simulation has at long time-scale simulations when compared to the JPL Horizons simulation. However it is sufficient at predicting eclipses within a 20 year time-scale at an accuracy of ± 10.2 minutes.

The data from the simulation however could be used in a least-squares fit method to fit a curve to the luminosity ratio function. This would provide a much smaller error due to the increased amount of data points being used in the calculation.

5.3 Transit Simulations for Trappist-One

Now that the eclipse method has been confirmed to work to a relatively good degree of accuracy it can be applied to the Trappist-One system and produce simulated transit light curves and compare them to observational data.

The time-step for this simulation was chosen based upon the results from the Solar System. Since the smallest time period in the Trappist system is around 1.5 days the time-step needed to be much smaller. Given a time-step of $\Delta T = 0.01$ days was used for the Solar System a time-step of $\Delta T = 0.005$ to $\Delta T = 0.001$ days was selected.

The initial run of the simulation was made with estimated values for the phases of the planets as described in Section 4.2.1 and then compared to the observational data-set starting from 19th September 2016.

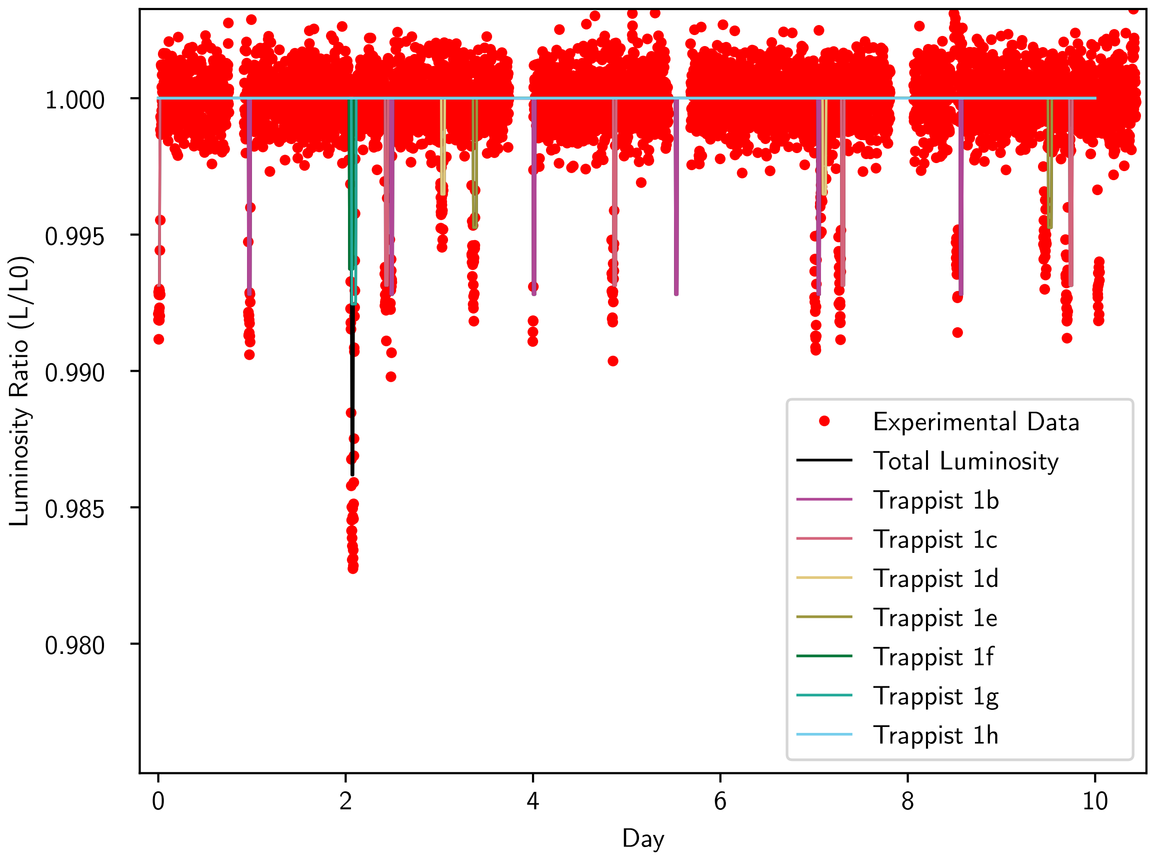

Figure 5.20: *Plot of the Luminosity ratio from the Trappist-One Star 1a, it contains simulated and observational data for the transits of the system from 19th September 2016.*

For an initial run the graph produced in Figure 5.20 is promising as each of the planets produce a transit at a reasonable time relative to the observational data but at short time-scales some of the transits are at the wrong times and the shape of the light curve is incorrect which can be seen

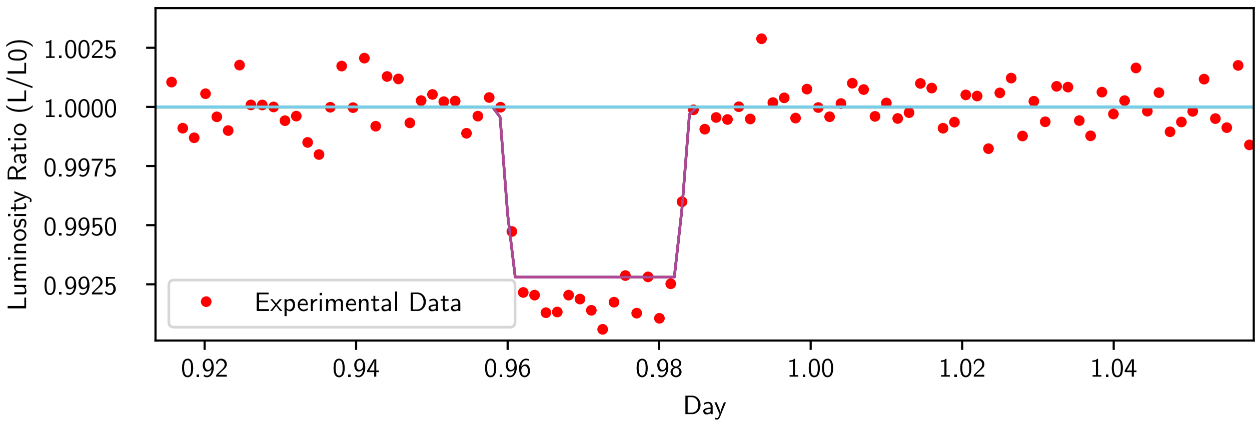

Figure 5.21: *Plot of the simulated transit of planet 1B without Limb Darkening.*

To fix the problem with the shape of the transit limb darkening can be applied to the model. This takes the distance of the planet from the centre of the star and calculates the intensity of light for the star at that position.

The equation takes the following form:

$$\frac{I(\theta)}{I(\theta = 0)} = \frac{a + b\cos\theta}{a + b} \tag{5.1}$$

where $a = 2/5$ and $b = 3/5$ [23].

$\cos\theta$ can be substituted using some trigonometry with the radius of the star and the perpendicular distance from the centre of the star:

$$\frac{I(\theta)}{I(\theta = 0)} = \frac{2}{5} + \frac{3}{5}\left(\frac{R_\star}{\sqrt{R_\star^2 + d^2}}\right) \tag{5.2}$$

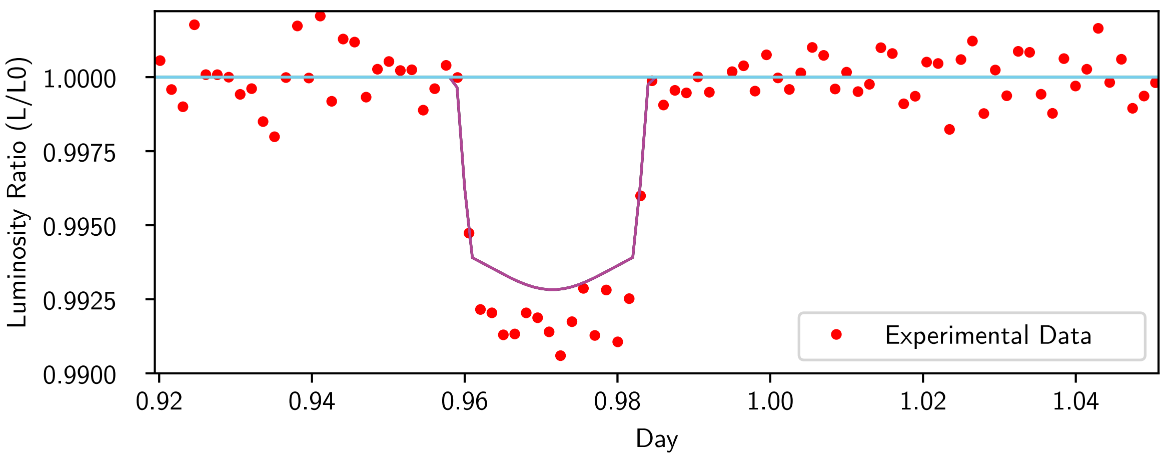

Figure 5.22: *Plot of the simulated transit of planet 1B with Limb Darkening.*

Figure 5.22 shows the same transit of planet 1B with the limb darkening model applied, which fits the observations more closely. Next the starting phase of each of the transits need to be fit to observations. This was achieved by fitting a high degree polynomial to the base of the transit for the simulated data-set. A χ^2 fit can then be applied to the model with the free parameter along the x axis.

Figure 5.23 shows the planets 1D and 1E before any fitting is made to the system. The original estimations for the starting phase for both planets puts both of their transits out by around an 15 minutes compared to observations.

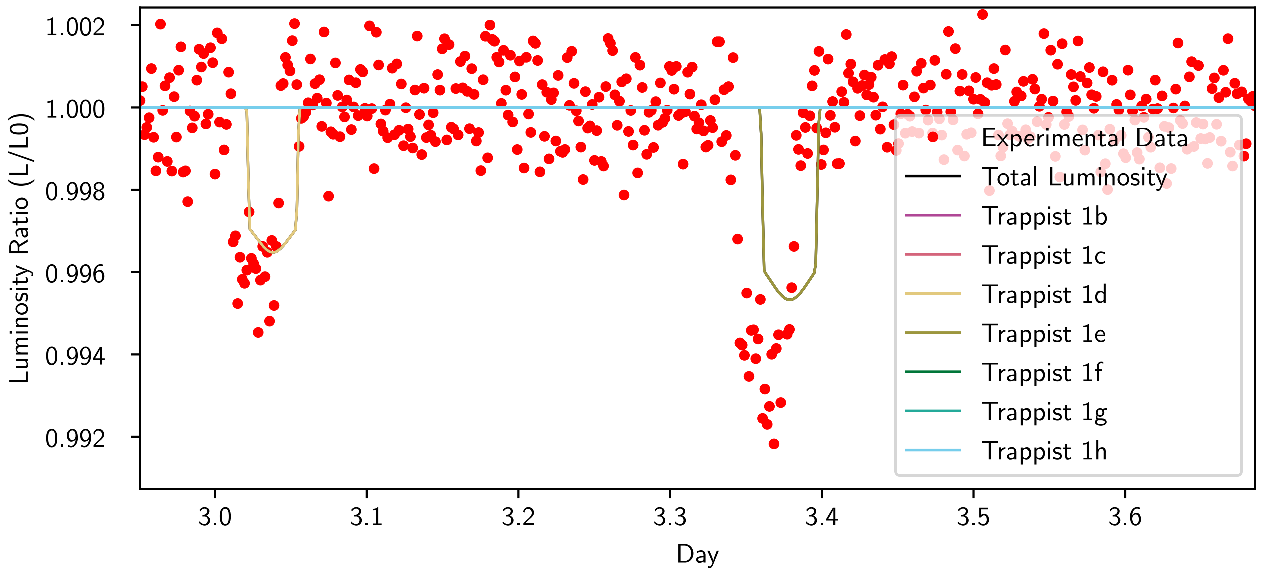

Figure 5.23: *Plot of the simulated transits of planets 1D and 1E with Limb Darkening.*

A 15 degree polynomial was used to model the trough of each of the transit light curves - such a high degree polynomial was chosen to provide an accurate representation of the curve and also due to the minimal computational power needed. This provided a fit to within 10^{-11} of the Luminosity ratio. Figure 5.24 shows the fit for planet 1E.

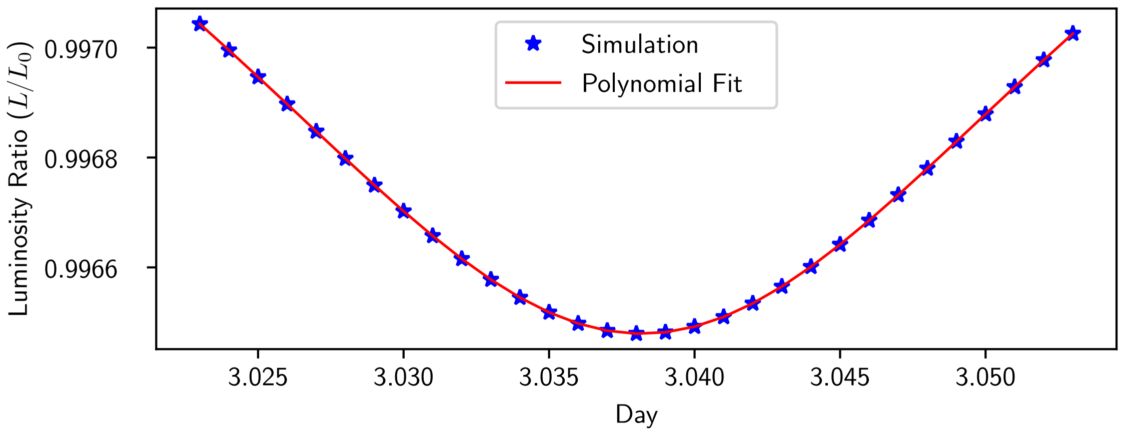

Figure 5.24: *Polynomial fit for the transit of planet 1D.*

A χ^2 fit was then applied to the model by translating the model along the x axis and calculating the reduced χ^2 over a range of offsets. The minimum χ^2 value was selected as the best fitting offset. Figure 5.25 shows the χ^2 plot for the transit of planet 1D and the calculated offset is around -0.01 days. Note that the steps seen between different χ^2 levels are due to the step size of observations.

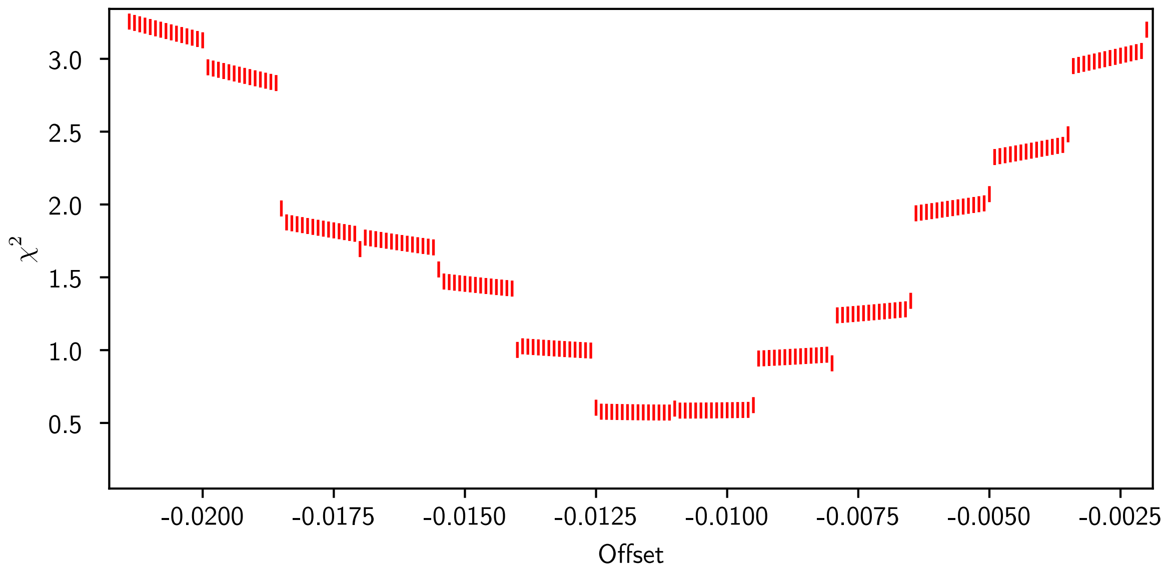

Figure 5.25: *Reduced χ^2 plot for the polynomial fit to observations for planet 1D.*

After applying the fit to planet 1D the resulting offset was -0.0159. Resulting in a new starting phase of -4.6660 radians. This can be seen in Figure 5.26 showing the improved fit. An extended view of this graph can be seen in the appendix with error bars applied in Figure B.1. The fits are not perfect due to the model only using the trough of the dip but they produce χ^2 values below 2 around the transit and move it to within minutes of the observations.

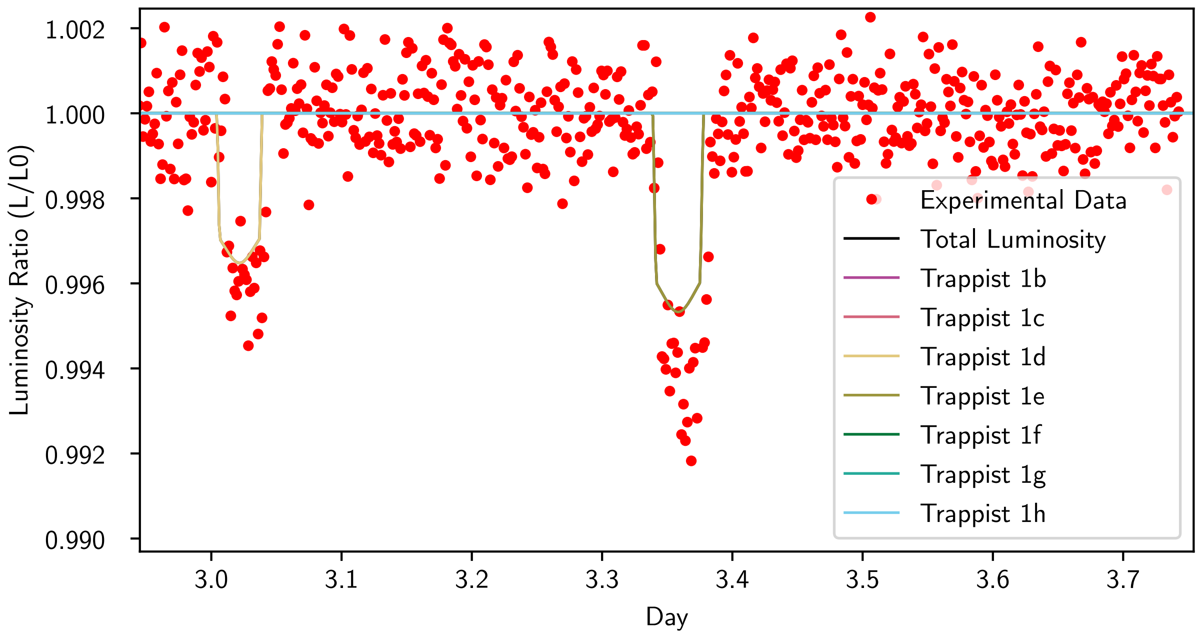

Figure 5.26: *Plot of the simulated transits of planets 1D and 1E after χ^2 fitting.*

Simulation	χ^2/D.O.F
No Limb Darkening or χ^2 Fitting	2.798
Limb Darkening	2.691
χ^2 Fit	2.458
χ^2 Fit & Limb Darkening	2.367

Table 5.4: *Table showing the decrease in χ^2 as each of the corrections are made.*

Once all of the planets have been fitted the reduced χ^2 of the whole data-set over a 10 day running time is reduced, however it still isn't perfect. Table 5.4 shows the effects both the limb darkening and polynomial fitting had on the simulation.

After each planet has orbited once it's second transit appears to be slightly delayed in the simulation and this effect is present for all planets at varying levels. This effect can be seen in Figure 5.27.

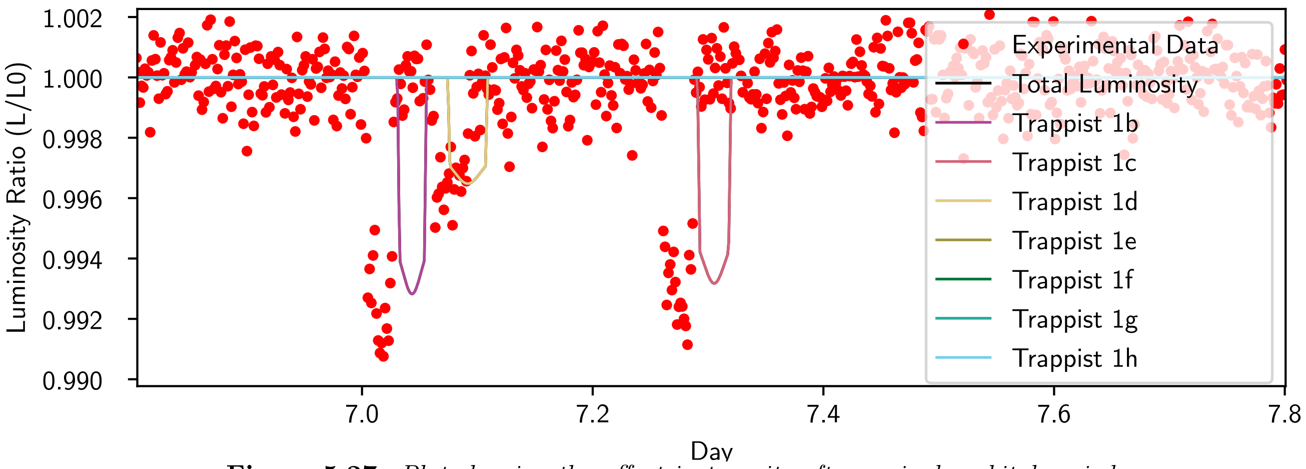

Figure 5.27: *Plot showing the offset in transits after a single orbital period.*

This is because of a variety of different reasons, the most likely is due to the simplicity of the simulation as discussed previously combined with the large uncertainties on the initial conditions. There could be an improvement in the initial condition estimations to make use of the mean anomalies and then a numerical method to solve for the true anomaly which would provide more accurate starting conditions.

The lack of general relativity, tidal deformation and rotational flattening models likely have influences on the simulation accuracy; especially given the features of the Trappist-one system involving a large number of planets close to their star that likely contain liquid oceans or liquid magma oceans [24]. The planets also have large influences on each other's orbits which leads even more free parameters.

Chapter 6

Summary and Conclusions

In this project an N-body simulator was developed from first-physical principles using the laws of motion derived by Newton. Numerical methods such as the first order Euler method were explored to solve the equations of motion for systems of $N > 2$ and then extended to higher order methods - the leapfrog method, the 4th-order Runge-Kutta method and the 4th-order Yoshida integrator. Each of these methods were bench-marked against the JPL Horizons data-set to assess their accuracy and computation times.

From initial runs of the simulation between $\Delta T = 0.4$ days and $\Delta T = 0.05$ days the Euler method was found to be less accurate by orders of magnitude compared to the Leapfrog, Yoshida and Runge-Kutta methods. This was expected due to the higher orders of these methods - the Euler method had an accumulated error of over 2.5 A.U with a time-step of 0.4 days, whilst all 3 of the higher order methods were below 0.6 A.U over a 50 year period. The same was found when comparing the leapfrog, a second order method, to the 4th order methods with it performing an order of magnitude worse in terms of accumulated accuracy over a 120 year period, at $\Delta T = 0.2$ days the leapfrog method accumulated almost 1 A.U of error compared to the 4th-order methods that accumulated below 0.1 A.U. A comparison between the two 4th-order methods was then made over a range of running times and it was found that the Yoshida integrator accumulated less error during long-time scale simulations at the same time-step as the Runge-Kutta method - with the Runge-Kutta accumulating 8x as much error at a time-step of $\Delta T = 0.2$ days with a running time of 48,000 days. Further investigation found that the accumulated error actually began to increase with smaller time-steps, this was due to rounding error from the use of double-precision arithmetic - one of the reasons that higher order methods are more accurate since larger time-steps can be used leading to less rounding-error.

Next the computational times of each method was investigated over a range of time-steps and running times. As expected, higher order methods had longer computational times, especially at small time-steps, but at larger time-steps the difference in computational time was less prevalent - this is the strength of higher order methods, providing additional accuracy with similar or minimal difference in computational time. This is seen in plots of accumulated error against computational time where the Euler method cannot produce the same accumulated error as the 4th-order methods at any computational times. It was also shown that the Leapfrog method could produce the same amount of accumulated error as the Euler method in a smaller computational time. The symplectic behaviour of the Yoshida and Leapfrog integrators was observed during long time-scale simulations where they accumulated error at slower rates to the Runge-Kutta method - this is due to these methods conserving energy much more accurately. Overall it was found that for short time-scale simulations the Runge-Kutta method may be preferable due to the small dip in accumulated error at small time-steps - but for any long time-scale simulations the Yoshida method is always preferable since it provides less accumulated error at larger time-steps and also performs faster.

The influences each planet has on the Earth's position was investigated by comparing to the JPL data-set between 1990 and 2010. In the most basic two body system involving just the Earth and the Sun there was a large discrepancy found in the amplitude of the Earth's Z-coordinate. This was due to the lack of precession in the Earth's orbit which is caused by the other planets. As each of the planets and the Moon were added to the simulation the curve began to align with the JPL data-set with the Moon, Venus, Saturn and Jupiter having the greatest effect. The Moon reduced the RMS error by 47.7%, Venus 33.2%, Jupiter 16.6% and Saturn 1.45%.

Using the simulation of the Solar System Lunar and Solar eclipses could be predicted. Initial tests shown that the simulation predicted the start and end time of eclipses between 1990 and 1991 (A 2 year running time) within one time-step: $\Delta T = 0.01$ days. A more in depth test using the midpoint of the eclipse reduced the error by a factor of $1/\sqrt{2}$. The midpoints were compared to a NASA data-set and their residuals were analysed - the simulation was found to be accurate to within one time-step over a 20 year period. The accuracy of the method could be improved by taking a least squares approach but ultimately at long time-scales the simulation suffered due to the lack of tidal force and lunar libration models as well as the use of a Newtonian model for the shape and size of the Earth.

Finally the same methods were applied to Trappist-One system to predict the transits it's planets. The simulation used the information learned from the previous results to obtain a reasonable time-step based on the time-periods of the planets from the system. The lack of limb darkening was noticeable when compared to observations and so a limb darkening model was applied to the simulation, the transits of each of the planets were then also fit using a polynomial function and a χ^2 fit applied to estimate an improved value for the phase for each of the planets. This improved the models fit by around 15-20% over a 10 day running time with a reduced χ^2 of 2.367. However after a single orbit the transits appear to be delayed slightly and out of phase - this is likely due to the uncertainties in the calculate of the initial conditions for the simulation combined with the lack of general relativity, tidal deformation and rotational flattening models.

Overall the simulation was successful at constructing an N-body simulator from first physical principles. The investigation into different integrators found the 4th-order Yoshida integrator to be the most effective at long time-scale simulations and some applications for N-body simulators were explored. To improve the methodology and the simulation as a whole if this project was repeated I would investigate the use additional models specific to the application being explored - such as a tidal force model between the Earth and Moon, this would likely have improved the accuracy of the eclipse predictions for long time-scale simulations. The prediction of transits from the Trappist system would benefit from more precise initial conditions and the method of calculating the initial positions and velocities suffers from approximations such as assuming circular orbits when calculating phases and rotating the positions and velocity vectors.

Appendix A

Code

A repository of all the code for the simulation is available at github.com/JamesPeters98/Java-NBody-Simulation

A.1 Body.java

```java
public abstract class Body implements Callable<Boolean>, Comparable<Body> {

    /** BODIES PROPERTIES **/
    private transient Vector3D velocity, nextVelocity; // (A.U/day)
    private transient Vector3D position, nextPos, tempPos; //Position in Kilometers (A.U)
    private transient List<Body> bodies;  //List of bodies to interact with.
    private transient List<Body> exclusiveBodies; //List of bodies without this body.
    private transient Universe universe;  //Universe this Body belongs too.
    private transient HashMap<Integer, Vector3D> tempAccelMap, tempVeloMap, tempPosMap;
    public  transient TreeMap<Integer, Vector3D>  positions; //History of positions. Key - time.

    private int loop = -1;
    private double GMinAU;
    private Color color = Utils.randomColor();

    public Body(){
        super();
        position = getInitialPosition();
        velocity = getInitialVelocity();
        positions = new TreeMap<>();
        tempAccelMap = new HashMap<>();
        tempVeloMap = new HashMap<>();
        tempPosMap = new HashMap<>();
        GMinAU = getGM()* Constants.CONVERSIONS.GM_to_AU;
    }

    /**
     * Call this after all bodies have been updated.
     */
    public void update(){
        if(loop >= universe.resolution() || loop == -1) {
            positions.put(universe.getUniverseStep(), position);
            loop = 0;
        }
        loop++;

        velocity = nextVelocity;
        position = nextPos;
    }

    /**
     * IMPLEMENTATIONS
     */
    public abstract Vector3D getInitialPosition();
    public abstract Vector3D getInitialVelocity();
    public abstract String getName();
    public abstract double getMass();
    public abstract double getGM();
    public abstract double getBodyRadius();
    public abstract TreeMap<Double, Vector3D> getJPLPositions();
    public abstract TreeMap<Double, Vector3D> getJPLVelocities();
    public abstract boolean isOrigin();
    public abstract LocalDateTime getStartDate();

    @Override
    public String toString() {
        return "["+getName()+"] " + "\n Pos:"+getInitialPosition()+ "\n Velocity:"+velocity;
    }

    @Override
    public Boolean call() {
        update();
        return true;
    }

    @Override
    public int compareTo(Body o) {
        return getName().compareTo(o.getName());
    }

    /**
     * GETTERS & SETTERS
     **/
    public void setBodies(List<Body> bodies){
        this.bodies = bodies;
        this.exclusiveBodies = new ArrayList<>(bodies);
        exclusiveBodies.remove(this);
    }

    public void setUniverse(Universe universe){ this.universe = universe;}

    public Color getColor(){ return color;}
    public Body setColor(Color color) {
        this.color = color;
        return this;
    }

    public double getBodyRadiusAU(){ return getBodyRadius()/ Constants.KILOMETERS.AU;}
    public double getGMAU() { return GMinAU;}
    public Vector3D distTo(Body body){ return body.position.subtract(position);}
    public double getEnergy(){ return kinteticEnergy()+potentialEnergy();}
    private double kinteticEnergy(){ return 0.5*getMass()*getVelocity().magnitude();}
    private double potentialEnergy(){
        return bodies.stream().filter(body -> body != this).mapToDouble(body -> -getGM() * (body.getMass() /
        ↪  distTo(body).magnitude())).sum();
    }

    public Vector3D getVelocity() { return velocity; }
    public Universe getUniverse() { return universe; }
    public List<Body> getBodies() { return bodies; }
    public List<Body> getExclusiveBodies(){ return exclusiveBodies;}
        public Vector3D getPosition() { return position; }
    public Body setPosition(Vector3D position) {
        this.position = position;
        return this;
    }

    public void setVelocity(Vector3D velocity){ this.velocity = velocity; }
    public void addPosition(Vector3D pos){ position.add(pos); }
    public void addVelocity(Vector3D velocity){ this.velocity.add(velocity); }

    // Set velocity to be updated after the integrator has made all steps. (after update() method is called)
    public void setNextPosition(Vector3D position){ this.nextPos = position; }
    public void setNextVelocity(Vector3D velo){ this.nextVelocity = velo; }
    public void setTempAccel(int index, Vector3D accel){ tempAccelMap.put(index,accel); }
    public Vector3D getTempAccel(int index){ return tempAccelMap.get(index); }
    public void setTempVelocity(int index, Vector3D velocity){ tempVeloMap.put(index,velocity); }
    public Vector3D getTempVelocity(int index){ return tempVeloMap.get(index); }
    public void setTempPos(int index, Vector3D position){ tempPosMap.put(index,position); }
    public Vector3D getTempPos(int index){ return tempPosMap.get(index); }
}
```

A.2 BodyBuilder.java

```java
public class BodyBuilder {

    private String name;
    private Vector3D initPos, initVelocity;
    private double mass, radius, GM;
    private TreeMap<Double, Vector3D> truePositions, trueVelocities;
    private boolean isOrigin = false;
    private LocalDateTime startDate;
    private Color color;

    public static BodyBuilder getInstance(){
        return new BodyBuilder();
    }
    private BodyBuilder(){}

    public BodyBuilder setName(String name){
        this.name = name;
        return this;
    }

    public BodyBuilder setInitPos(Vector3D initPos) {
        this.initPos = initPos;
        return this;
    }

    public BodyBuilder setInitVelocity(Vector3D initVelocity) {
        this.initVelocity = initVelocity;
        return this;
    }

    public BodyBuilder setMass(double mass) {
        this.mass = mass;
        return this;
    }

    public BodyBuilder setRadius(double radius) {
        this.radius = radius;
        return this;
    }

    public BodyBuilder setPositions(TreeMap<Double, Vector3D> positions) {
        this.truePositions = positions;
        return this;
    }

    public BodyBuilder setVelocities(TreeMap<Double, Vector3D> velocities) {
        this.trueVelocities = velocities;
        return this;
    }

    public BodyBuilder setOrigin(boolean origin) {
        isOrigin = origin;
        return this;
    }

    public BodyBuilder setGM(double GM) {
        this.GM = GM;
        return this;
    }

    public BodyBuilder setStartDate(LocalDateTime startDate) {
        this.startDate = startDate;
        return this;
    }

    public BodyBuilder setColor(Color color) {
        this.color = color;
        return this;
    }

    public Body create(){
        Body body = new Body() {
            @Override
            public Vector3D getInitialPosition() { return initPos; }
            @Override
            public Vector3D getInitialVelocity() { return initVelocity; }
            @Override
            public String getName() { return name; }
            @Override
            public double getMass() { return mass; }
            @Override
            public double getGM() { return GM; }
            @Override
            public double getBodyRadius() { return radius; }
            @Override
            public TreeMap<Double, Vector3D> getJPLPositions() { return truePositions; }
            @Override
            public TreeMap<Double, Vector3D> getJPLVelocities() { return trueVelocities; }
            @Override
            public boolean isOrigin() { return isOrigin; }
            @Override
            public LocalDateTime getStartDate() { return startDate;}
        };
        body.setColor(color);
        return body;
    }

    public String serialise(Gson gson){
        return gson.toJson(this);
    }
    public BodyBuilder fromString(Gson gson, String json){
        return gson.fromJson(json, BodyBuilder.class);
    }
}
```

A.3 UniverseBuilder.java

```java
public abstract class UniverseBuilder {

    protected String name;
    protected double dt;

    /**
     * SETTERS & GETTERS
     */
    public void setName(String name) {
        this.name = name;
    }
    public void setDt(double dt) {
        this.dt = dt;
    }

    public abstract List<Body> createBodies();
    public String getName() {
        return name;
    }
    public double getDt() {
        return dt;
    }

    /**
     *  SERIALIZER
     */
    public String serialise(Gson gson){
        return gson.toJson(this);
    }

}
```

A.4 Universe.java

```java
public abstract class Universe {

    //Simulation objects.
    protected List<Body> bodies;
    protected Body originBody;
    protected Integrator integrator;
    private SimulationPerformanceTracker performanceTracker;
    public transient TreeMap<Double, Double> energyShift;

    //Simulation variables.
    private long cpuTime = 0;
    private long lastTime = 0;
    private int universeStep = 0;
    private boolean running = true;
    private boolean output = true;


    public Universe(){
        energyShift = new TreeMap<>();
    }

    /**
     *  MUST BE CALLED BEFORE SIMULATION STARTS
     */
    public void init(){
        bodies = createBodies();

        for(Body body : bodies){
            if(body.isOrigin()) originBody = body;
            body.setUniverse(this);
            body.setBodies(bodies);
        }

        MomentumCorrector.correct(this);
        energyShift.put(0.0,getTotalEnergy());
        performanceTracker = new SimulationPerformanceTracker(this);
    }

    public void start(){
        lastTime = System.nanoTime();
        Runnable task = () -> {
            int loops  = 0;
            performanceTracker.startTracker();

            while(running){
                long now = System.nanoTime();

                //Try to perform a loop.
                try { loop(); }
                catch (InterruptedException e) { e.printStackTrace(); }

                cpuTime += (now-lastTime);
                lastTime = now;
                universeStep++;

                //Print to console every second the current state of the simulation.
                if(cpuTime > TimeUnit.SECONDS.toNanos(1)){
                    System.out.println("Universe Time: "+(getUniverseTime())+" Days
                    ↪  "+(100*getUniverseTime()/runningTime())+"%");
                    cpuTime = 0;
                }

                // Every 100 timesteps
                if(loops >= 100){
                    loops = 0;
                    energyShift.put(getUniverseTime(),getTotalEnergy());
                }
                loops++;

                //Simulation is finished
                if(getUniverseTime() >= runningTime()){
                    performanceTracker.finishTracker();
                    System.out.println("------------------------");
                    System.out.println("--Finished Simulation!--");
                    System.out.println("------------------------");
                    MemoryCalculator.printMemoryUsed();
                    performanceTracker.printStats();

                    try { onFinish(); }
                    catch (IOException e) { e.printStackTrace(); }
                    running = false;
                }
            }
        };

        Thread backgroundThread = new Thread(task);
        backgroundThread.setDaemon(false);
        backgroundThread.setName("Background Thread");
        backgroundThread.start();
    }

    private double getTotalEnergy(){
        return bodies.stream().mapToDouble(Body::getEnergy).sum();
    }

    public void stop(){ running = false; }
    public boolean hasFinished(){ return !running; }
    public double getUniverseTime() { return universeStep*dt }
    public int getUniverseStep() { return universeStep; }
    public List<Body> getBodies() { return bodies; }
    public Body getOriginBody() { return originBody; }
    public Integrator getIntegrator() { return integrator; }
    public void setIntegrator(Integrator integrator){ this.integrator = integrator; }
    public void setOutput(boolean output){ this.output = output; }

    /**
     *  ABSTRACT METHODS
     **/
    public abstract List<Body> createBodies();
    public abstract String getName();
    public abstract double dt();
    public abstract double runningTime();
    public abstract int resolution(); //Number of steps between each point saved.

    //Called at the end of the simulation.
    protected abstract void onFinish() throws IOException;
    //Called every tick.
    protected abstract void loop() throws InterruptedException;
}
```

A.5 MomentumCorrector.java

```java
public class MomentumCorrector {

    public static void correct(Universe universe){
        Vector3D momentum = Vector3D.ZERO;
        double totalMass = 0;

        for(Body body : universe.bodies){
            momentum = momentum.add(body.getVelocity().multiply(body.getMass()));
            totalMass += body.getMass();
        }

        Vector3D velocity = momentum.multiply(1/totalMass);

        for(Body body : universe.bodies){
            body.setVelocity(body.getVelocity().subtract(velocity));
        }

        System.out.println("TOTAL MOMENTUM = "+momentum.magnitude());
        System.out.println("TOTAL MASS = "+totalMass);
        System.out.println("VELOCITY CORRECT = "+velocity.magnitude());
        System.out.println("VELOCITY CORRECT = "+velocity);

    }
}
```

A.6 Integrator.java

```java
public abstract class Integrator {

    //Integrator can decide how to evolve the system at each step.
    public abstract void step(Universe universe);

    public abstract String getIntegratorName();

    //Calculates Acceleration of given body using the temp pos parameter, e.g index 1 = x1
    public Vector3D accel(Body body, int positionIndex){
        Vector3D accel = Vector3D.ZERO;
        for(Body body2 : body.getExclusiveBodies()) {
            Vector3D delta = body.getTempPos(positionIndex).subtract(body2.getTempPos(positionIndex));
            double distance = delta.magnitude();
            // a(t)
            double accelMagnitude = -(body2.getGMAU()) / (distance * distance * distance);
            accel = accel.add(delta.multiply(accelMagnitude));
        }
        return accel;
    }
}
```

A.7 EulerIntegrator.java

```java
public class EulerIntegrator extends Integrator {

    @Override
    public void step(Universe universe) {
        double dt = universe.dt();

        for(Body body : universe.getBodies()){
            body.setTempPos(1,body.getPosition());                  //x1
        }

        for(Body body : universe.getBodies()){
            Vector3D v = body.getVelocity().add(accel(body,1).multiply(dt));
            Vector3D x = body.getTempPos(1).add(v.multiply(dt));
            body.setNextVelocity(v);
            body.setNextPosition(x);
        }
    }

    @Override
    public String getIntegratorName() {
        return "Euler";
    }

}
```

A.8 RK4Integrator.java

```java
public class RK4Integrator extends Integrator {

    @Override
    public void step(Universe universe) {
        double dt = universe.dt();

        for(Body body : universe.getBodies()){
            body.setTempPos(1,body.getPosition());          //x1
        }

        for(Body body : universe.getBodies()){
            Vector3D a1 = accel(body,1);
            Vector3D v1 = body.getVelocity();

            Vector3D x2 = body.getPosition().add(v1.multiply(dt/2));
            Vector3D v2 = body.getVelocity().add(a1.multiply(dt/2));

            body.setTempPos(2,x2);
            body.setTempVelocity(2,v2);
            body.setTempAccel(1,a1);
        }

        for(Body body : universe.getBodies()){
            Vector3D a2 = accel(body,2);
            Vector3D v2 = body.getTempVelocity(2);

            Vector3D x3 = body.getPosition().add(v2.multiply(dt/2));
            Vector3D v3 = body.getVelocity().add(a2.multiply(dt/2));

            body.setTempPos(3,x3);
            body.setTempVelocity(3,v3);
            body.setTempAccel(2,a2);
        }

        for(Body body : universe.getBodies()){
            Vector3D a3 = accel(body,3);
            Vector3D v3 = body.getTempVelocity(3);

            Vector3D x4 = body.getPosition().add(v3.multiply(dt));
            Vector3D v4 = body.getVelocity().add(a3.multiply(dt));

            body.setTempPos(4,x4);
            body.setTempVelocity(4,v4);
            body.setTempAccel(3,a3);
        }

        for(Body body : universe.getBodies()){
            Vector3D a4 = accel(body,4);
            Vector3D dx = sumRK4(body.getVelocity(), body.getTempVelocity(2), body.getTempVelocity(3),
            ↪  body.getTempVelocity(4)).multiply(dt);
            Vector3D dv = sumRK4(body.getTempAccel(1), body.getTempAccel(2), body.getTempAccel(3), a4).multiply(dt);
            body.setNextPosition(body.getPosition().add(dx));
            body.setNextVelocity(body.getVelocity().add(dv));
        }
    }

    @Override
    public String getIntegratorName() {
        return "RK4";
    }

    private Vector3D sumRK4(Vector3D k1, Vector3D k2, Vector3D k3, Vector3D k4){
        Vector3D sumdx = Vector3D.ZERO;
        sumdx = sumdx.add(k1);
        sumdx = sumdx.add(k2.multiply(2));
        sumdx = sumdx.add(k3.multiply(2));
        sumdx = sumdx.add(k4);
        return sumdx.multiply((double) 1 / (double) 6);
    }
}
```

A.9 EclipseCalculator.java

```java
public class EclipseCalculator {

    //This is just for Earth, Sun and Moon (Luna). Needs to be adapted for more general scenario.
    public static void findEclipses(Universe universe){
        Body sun = null, earth = null, moon = null;
        for(Body body : universe.getBodies()){
            if(body.isOrigin()) sun = body;
            if(body.getName().equals("Earth")) earth = body;
            if(body.getName().equals("Moon")) moon = body;
        }
        if(sun != null && earth != null && moon != null) {
            TreeMap<Double,Double> ConeRadius = new TreeMap<>(), EdgeOfMoonDist = new TreeMap<>(), Lambda = new
            ↪ TreeMap<>(), Area = new TreeMap<>(), JPLArea = new TreeMap<>();

            for(Integer step: sun.positions.keySet()) {
                //Simulated data
                Vector3D MoonPos = moon.positions.get(step);
                Vector3D SunPos = sun.positions.get(step);
                Vector3D EarthPos = earth.positions.get(step);
                calc(step*universe.dt(), EarthPos, SunPos, MoonPos, sun, earth, moon, ConeRadius, EdgeOfMoonDist,
                ↪ Lambda, Area);
            }

            for(Double time: sun.getJPLPositions().keySet()){
                Vector3D MoonPos = moon.getJPLPositions().get(time);
                Vector3D SunPos = sun.getJPLPositions().get(time);
                Vector3D EarthPos = earth.getJPLPositions().get(time);
                calc(time, EarthPos, SunPos, MoonPos, sun, earth, moon, null, null, null, JPLArea);
            }

            HashMap<Integer,EclipseInfo> eclipseInfoHashMap = calculateEclipseFeatures(sun.getStartDate(),Area);
            Graph.plotEclipse("Eclipse Plot",Area,JPLArea);
            CSVWriter.writeEclipseData("SolarSystem",ConeRadius,EdgeOfMoonDist,Lambda,Area);
            CSVWriter.writeEclipseInfo("SolarSystem",eclipseInfoHashMap,universe);
        }
    }

    private static void calc(double time, Vector3D EarthPos, Vector3D SunPos, Vector3D MoonPos, Body sun, Body earth,
    ↪  Body moon, TreeMap<Double,Double> ConeRadius, TreeMap<Double,Double> EdgeOfMoonDist,TreeMap<Double,Double>
    ↪  Lambda,TreeMap<Double,Double> Area){
        //Direction vector.
        Vector3D SunEarthVector = directionVector(EarthPos,SunPos);
        //Lambda < 0 means Moon is behind Earth from Sun's perspective.
        double lambda = (MoonPos.subtract(EarthPos).dotProduct(SunEarthVector)) /
        ↪  (SunEarthVector.dotProduct(SunEarthVector));
        //Point perpendicular to line
        Vector3D P = SunEarthVector.multiply(lambda).add(EarthPos);

        double SEdist = dist(EarthPos,SunPos).magnitude();
        double PEdist = dist(EarthPos, P).magnitude();
        // Radius of circle at point of cone perpendicular to Moon.
        double r1 = (PEdist / SEdist) * (sun.getBodyRadiusAU() - earth.getBodyRadiusAU()) + earth.getBodyRadiusAU();

        //Calculate Area of eclipse
        double r2 = moon.getBodyRadiusAU();
        double d = dist(MoonPos, P).magnitude();

        double A;
        if(d >= r1 + r2) { // No intersection = 0 area;
            A = 0;
        } else if(d <= r1 - r2){ // If moon is inside cone area it's just the moons total area.
            A = Math.PI*r2*r2;
        } else { // Otherwise it's the intersection area between the two.
            double d1 = (r1 * r1 - r2 * r2 + d * d) / (2 * d);
            double d2 = d - d1;
            A = (r1 * r1) * Math.acos(d1 / r1) - d1 * Math.sqrt(r1 * r1 - d1 * d1)
                    + (r2 * r2) * Math.acos(d2 / r2) - d2 * Math.sqrt(r2 * r2 - d2 * d2);
        }

        double areaRatio = (1 - (A) / (Math.PI * r1 * r1));

        double EdgeOfMoon = d - moon.getBodyRadiusAU();
        if(ConeRadius != null) ConeRadius.put(time, r1);
        if(EdgeOfMoonDist != null) EdgeOfMoonDist.put(time, EdgeOfMoon);
        if(Lambda != null) Lambda.put(time, lambda);
        if(Area != null) Area.put(time, areaRatio);
    }

    private static Vector3D dist(Vector3D from, Vector3D to){
        return to.subtract(from);
    }
    private static Vector3D directionVector(Vector3D from, Vector3D to){
        return dist(from,to).normalize();
    }

    private static HashMap<Integer, EclipseInfo> calculateEclipseFeatures(LocalDateTime startDate,
    ↪  TreeMap<Double,Double> area){
        double prevL = 1;
        HashMap<Integer,EclipseInfo> eclipseList = new HashMap<>();
        int eclipses = 0;
        for(Map.Entry<Double,Double> entry : area.entrySet()){
            double time = entry.getKey();
            double L = entry.getValue();
            long offset = (long) (time*86400);

            if(prevL == 1 && L < 1){
                //Start of eclipse.
                EclipseInfo info = new EclipseInfo();
                info.startDate = startDate.plusSeconds(offset);
                eclipseList.put(eclipses,info);
            }
            if(prevL < 1 && L == 1){
                //End of eclipse.
                EclipseInfo info = eclipseList.get(eclipses);
                info.endDate = startDate.plusSeconds(offset);
                eclipses++;
            }
            prevL = L;
        }
        return eclipseList;
    }
}
```

A.10 TransitCalculator.java

```java
public class TransitCalculator {

    public static TransitInfo plotTotalTransits(Universe universe, Vector3D direction) throws IOException {
        TreeMap<Body, TreeMap<Double,Double>> transits = new TreeMap<>();
        TreeMap<Double,Double> totalTransit = new TreeMap<>();
        for(Body body : universe.getOriginBody().getExclusiveBodies()){
            TreeMap<Double,Double> transit = findTransits(universe,body,direction,false);
            transits.put(body,transit);
            for(Map.Entry<Double,Double> entry : transit.entrySet()){
                double value = entry.getValue()-1;
                Utils.addToTreeMapValue(totalTransit,entry.getKey(),value);
            }
        }

        for(Map.Entry<Double,Double> entry : totalTransit.entrySet()){
            Utils.addToTreeMapValue(totalTransit,entry.getKey(),1.0);
        }

        CSVWriter.writeTransitData("Trappist",transits,totalTransit);
        Graph.plotEclipse("Total Transits",transits,totalTransit);
        return new TransitInfo(totalTransit,transits);
    }

    //This will calculate the Transits for the given body in the given direction.
    public static TreeMap<Double, Double> findTransits(Universe universe, Body planet, Vector3D direction, boolean
    ↪  plot){
        Body star = universe.getOriginBody();

        TreeMap<Double,Double> ConeRadius = new TreeMap<>();
        TreeMap<Double,Double> EdgeOfMoonDist = new TreeMap<>();
        TreeMap<Double,Double> Lambda = new TreeMap<>();
        TreeMap<Double,Double> Area = new TreeMap<>();

        for(Integer step: star.positions.keySet()) {
            //Simulated data
            Vector3D planetPos = planet.positions.get(step);
            Vector3D starPos = star.positions.get(step);
           calc(step*universe.dt(),direction,starPos,planetPos,star,planet, ConeRadius, EdgeOfMoonDist, Lambda, Area);
        }

        HashMap<Integer,EclipseInfo> eclipseInfoHashMap = calculateEclipseFeatures(star.getStartDate(),Area);
        if(plot) Graph.plotEclipse("Transits for "+planet.getName(),Area,null);
        CSVWriter.writeEclipseData("Trappist/"+planet.getName(),ConeRadius,EdgeOfMoonDist,Lambda,Area);
        CSVWriter.writeEclipseInfo("Trappist/"+planet.getName(),eclipseInfoHashMap,universe);
        return Area;
    }

    private static void calc(double time, Vector3D direction, Vector3D starPos, Vector3D planetPos, Body star, Body
    ↪  planet, TreeMap<Double,Double> ConeRadius, TreeMap<Double,Double> EdgeOfMoonDist,TreeMap<Double,Double>
    ↪  Lambda,TreeMap<Double,Double> Area){
         //Lambda < 0 means Moon is behind Earth from Sun's perspective.
        double lambda = (planetPos.subtract(starPos).dotProduct(direction)) / (direction.dotProduct(direction));
        //Point perpendicular to line
        Vector3D P = direction.multiply(lambda).add(starPos);

        // Radius of circle at point of cone perpendicular to Moon.
        double r1 = star.getBodyRadiusAU();
        double d = dist(planetPos, P).magnitude();
        double areaRatio = 1;

        if(lambda > 0) {
            //Calculate Area of eclipse
            double r2 = planet.getBodyRadiusAU();
            double A;
            double ratio = LimbDarkening.intensityRatio(star.getBodyRadiusAU(),d,planet.getBodyRadiusAU());

            if (d >= r1 + r2) { // No intersection = 0 area;
                A = 0;
            } else if (d < r1 - r2) { // If moon is inside cone area it's just the moons total area.
                A = Math.PI * r2 * r2;
            } else { // Otherwise it's the intersection area between the two.
                //double totalA =  Math.PI * r2 * r2;
                double d1 = (r1 * r1 - r2 * r2 + d * d) / (2 * d);
                double d2 = d - d1;
                A = (r1 * r1) * Math.acos(d1 / r1) - d1 * Math.sqrt(r1 * r1 - d1 * d1)
                        + (r2 * r2) * Math.acos(d2 / r2) - d2 * Math.sqrt(r2 * r2 - d2 * d2);

            }
            areaRatio = (1 - (A*ratio) / (Math.PI * r1 * r1));
        }

        double EdgeOfPlanet = d - planet.getBodyRadiusAU();
        if(ConeRadius != null) ConeRadius.put(time, r1);
        //if(EdgeOfMoonDist != null) EdgeOfMoonDist.put(time, EdgeOfMoon);
        if(Lambda != null) Lambda.put(time, lambda);
        if(Area != null) Area.put(time, areaRatio);
    }

/*
 * THE REST OF CLASS IS IDENTICAL TO EclipseCalculator.java
 */
```

A.11 CSVWriter.java

```java
public class CSVWriter {

    private static PrintWriter getOutputFile(String filenamePath){
        try {
            Path path = Paths.get("outputs/"+filenamePath);
            Files.createDirectories(path.getParent());
            FileWriter file = new FileWriter(String.valueOf(path));     // this creates the file with the given name
            return new PrintWriter(file); // this sends the output to file
        } catch (IOException e) {
            System.err.println("File couldn't be accessed it may be being used by another process!");
            System.err.println("Close the file and press Enter to try again!");
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            try { reader.readLine(); } catch (IOException ex) { ex.printStackTrace(); }
            return getOutputFile(filenamePath);
        }
    }

    // Resolution = Number of points to discard between each CSV record.
    public static void writeBody(Body body, int resolution) {
        PrintWriter outputFile = getOutputFile(body.getUniverse().getName()
        ↪  +"/"+body.getUniverse().getIntegrator().getIntegratorName() +"/"+body.getName()
        ↪  +"/"+body.getName()+"_data_" +body.getUniverse().dt()+".csv");

        if(outputFile != null) {
            // Write the file as a comma seperated file (.csv) so it can be read it into EXCEL
            outputFile.println("time (days), x, y, z, magnitude (A.U)");
            double dt = body.getUniverse().dt();

            // now make a loop to write the contents of each step to disk, one number at a time
            AtomicInteger points = new AtomicInteger(resolution);
            body.positions.forEach((time, point3D) -> {
                points.incrementAndGet();
                if (points.get() >= resolution) {
                    // comma separated values
                    Vector3D origin = body.getUniverse().getOriginBody().positions.get(time);
                    point3D = point3D.subtract(origin);
                   outputFile.println(time * dt + "," + point3D.getX() + "," + point3D.getY() + "," + point3D.getZ() +
                    ↪  "," + point3D.magnitude());
                    points.set(0);
                }
            });
            outputFile.close(); // close the output file
            System.out.println("Written CSV Data for: " + body.getName());
        }
    }

    public static void writeEnergyShift(Universe universe) {
        PrintWriter outputFile =
        ↪  getOutputFile(universe.getName()+"/Energyshift/"+universe.dt()+"-dt/"+universe.getIntegrator().getIntegratorName()+"-"+unive
        double initEnergy = universe.energyShift.get(0.0);
        if(outputFile != null) {
            // Write the file as a comma seperated file (.csv) so it can be read it into EXCEL
            outputFile.println("dt, " + universe.dt());
            outputFile.println("universe, " + universe.getName());
            outputFile.println("initial Energy (J), " + initEnergy);

            outputFile.println(",");
            outputFile.println("time, energy change (J)");

            universe.energyShift.forEach((time, energy) -> {
                outputFile.println(time + "," + (initEnergy - energy));
            });

            outputFile.close(); // close the output file
            System.out.println("Written CSV Data for: Energy Shift");
        }
    }

    public static void writeEclipseData(String folderPath, TreeMap<Double,Double> coneRadius, TreeMap<Double,Double>
    ↪  edgeOfMoonDist,TreeMap<Double,Double> lambda,TreeMap<Double,Double> area) throws IOException {
        PrintWriter outputFile = getOutputFile("eclipse/"+folderPath+"/data.csv"); // this sends the output to file1
        if (outputFile != null) {
            outputFile.println("time, Cone Radius, Edge Of Moon Distance, Lambda, Area");
            for (Double time : coneRadius.keySet()) {
                outputFile.println(time + "," + coneRadius.get(time) + "," + edgeOfMoonDist.get(time) + "," +
                ↪  lambda.get(time) + "," + area.get(time));
            }
            outputFile.close(); // close the output file
            System.out.println("Saved CSV Data for: Eclipse - " + folderPath);
        }
    }

    public static void writeTransitData(String folderPath, TreeMap<Body, TreeMap<Double,Double>> transits,
    ↪  TreeMap<Double,Double> totalTransits) {
        PrintWriter outputFile = getOutputFile("eclipse/"+folderPath+"/totalTransits.csv");
        if(outputFile != null) {
            outputFile.print("time");
            transits.keySet().forEach(body -> outputFile.print(", " + body.getName()));
            outputFile.println(", Total");

            Set<Double> times = transits.get(transits.firstKey()).keySet();
            times.forEach(time -> {
                outputFile.print(time);
                transits.forEach((body, map) -> {
                    outputFile.print(", " + map.get(time));
                });
                outputFile.println(", " + totalTransits.get(time));
            });
            outputFile.close(); // close the output file
            System.out.println("Saved CSV Data for: Total Transits - " + folderPath);
        }
    }

    public static void writeChi2Data(String folderPath, TreeMap<Double,ChiSquareValue> values) {
        PrintWriter outputFile = getOutputFile("eclipse/"+folderPath+"/chi2fit.csv");
        if (outputFile != null) {
            outputFile.println("Time, Model Luminosity, Observed Luminosity, Error, Chi^2");
            values.forEach((time, chiSquareValue) -> {
                outputFile.println(time + ", " + chiSquareValue.model + ", " + chiSquareValue.observed + ", " +
                ↪  chiSquareValue.error + ", " + chiSquareValue.getChi2());
            });
            outputFile.close(); // close the output file
            System.out.println("Saved CSV Data for: Chi^2 fits - " + folderPath);
        }
    }

    public static void writeEclipseInfo(String folderPath, HashMap<Integer, EclipseInfo> eclipseInfo, Universe
    ↪  universe) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("uuuu-MMM-dd");
        PrintWriter outputFile = getOutputFile("eclipse/"+folderPath +"/eclipseInfo-"
        ↪  +universe.getOriginBody().getStartDate().format(formatter) +"-dt-"+universe.dt()
        ↪  +"-"+universe.getIntegrator().getIntegratorName()+".csv");
        if (outputFile != null) {
            outputFile.println("Eclipse, Start Date, End Date, Duration (Seconds)");
            for (Map.Entry<Integer, EclipseInfo> entry : eclipseInfo.entrySet()) {
               outputFile.println(entry.getKey() + "," + entry.getValue().startDate + "," + entry.getValue().endDate +
                ↪  "," + entry.getValue().getDuration().toSeconds());
            }
            outputFile.close(); // close the output file
            System.out.println("Saved CSV Data for: Eclipse Info - " + folderPath);
        }
    }
}
```

Appendix B

Extended Figures

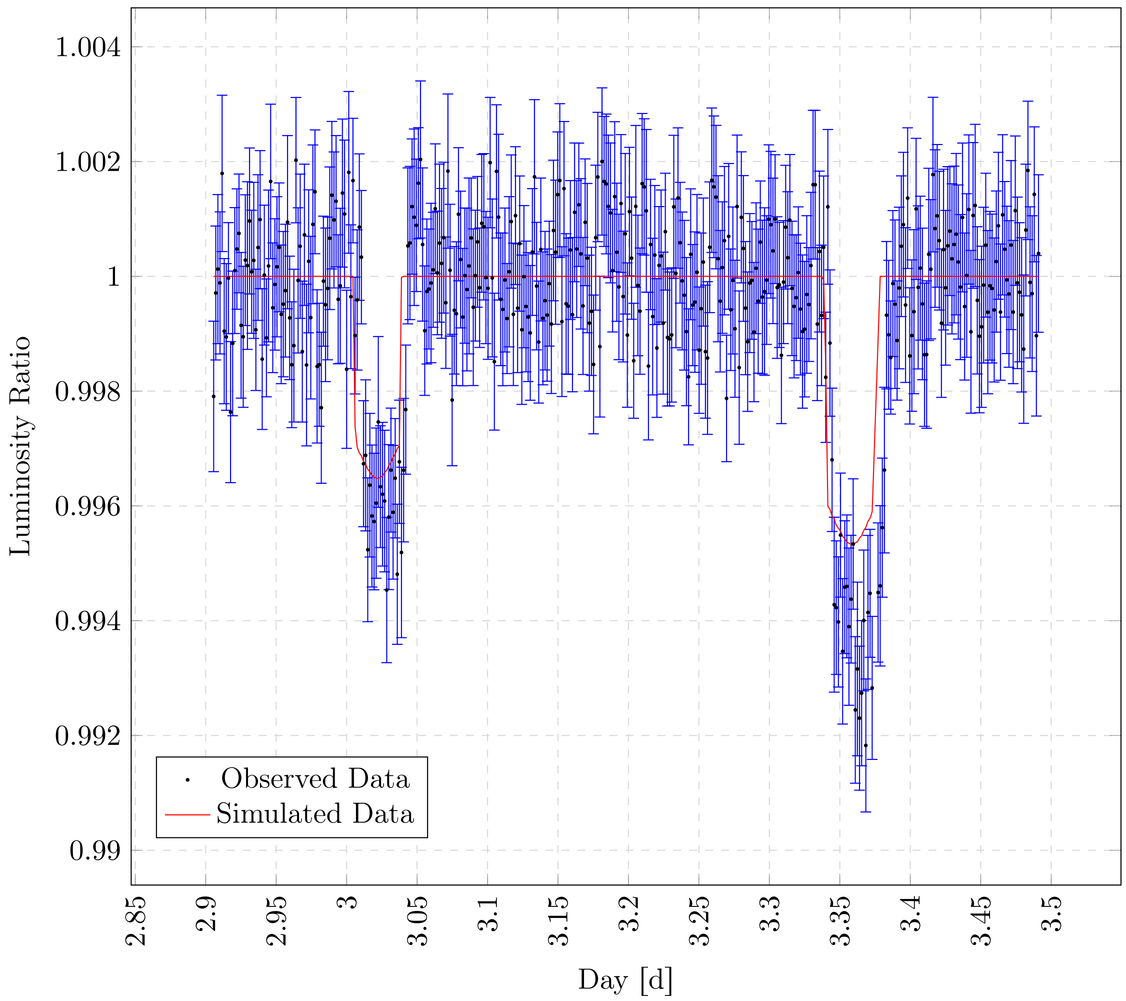

Figure B.1: *Plot of the simulated transits of planets 1D and 1E after χ^2 fitting with error bars on observational data.*

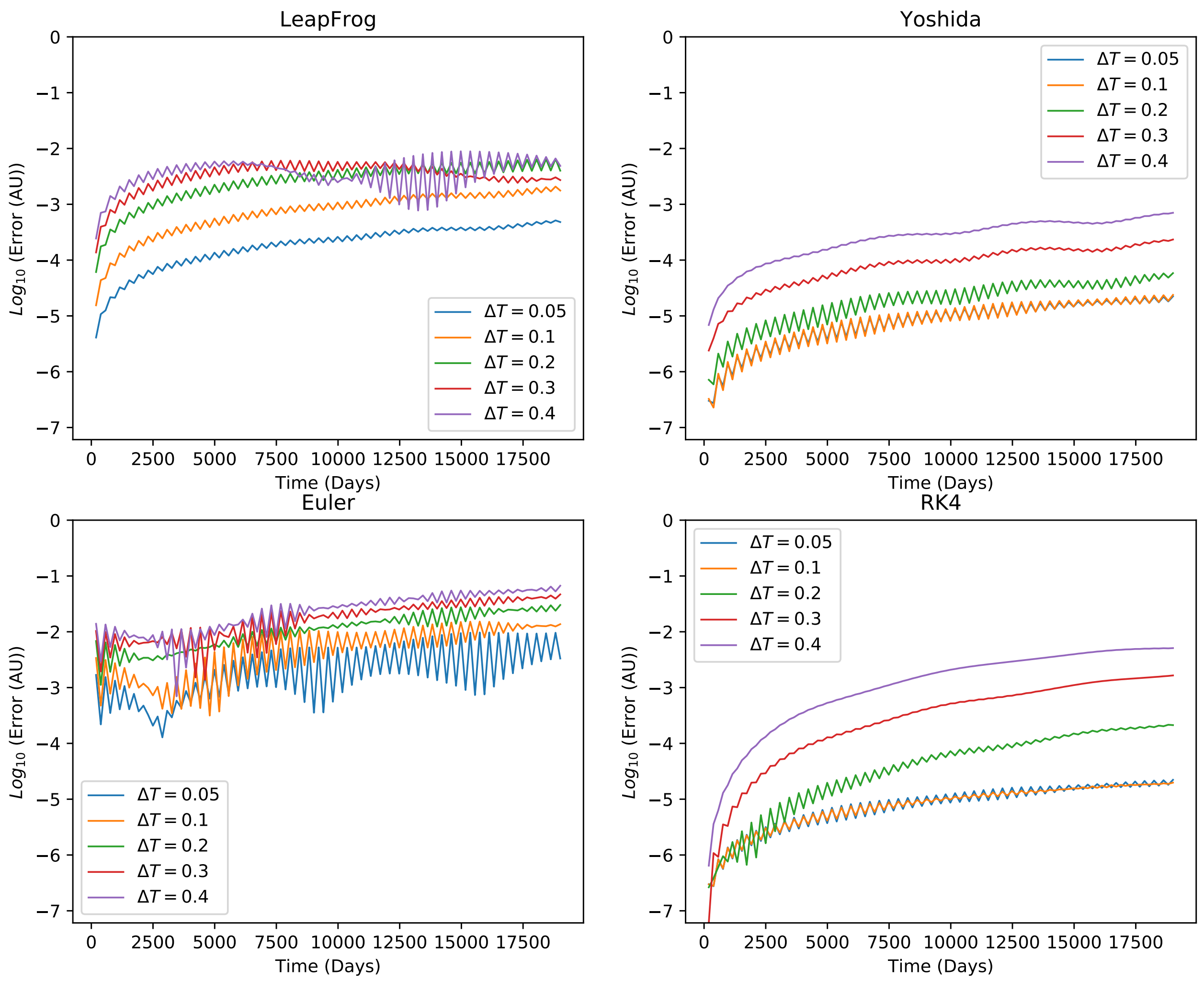

Figure B.2: *Log plot of error against time where the error is the deviance from the JPL Horizons data-set. The comparison is between the magnitude of the Moon's position vector relative to the sun.*

List of Figures

List of Tables

Bibliography

[1] Bernard R. Goldstein. Copernicus and the origin of his heliocentric system. *Journal for the History of Astronomy*, 33(3):219–235, 2002. doi: 10.1177/002182860203300301. URL https://doi.org/10.1177/002182860203300301.

[2] Stephen G. Brush. *Physics, the Human Adventure: From Copernicus to Einstein and Beyond*. Rutgers University Press, mar 2001. ISBN 0813529085. URL https://www.xarg.org/ref/a/0813529085/.

[3] Johannes Kepler. *Harmonices mundi*. Culture et Civilisation, 1968.

[4] F Rohrlich. *From paradox to reality : our basic concepts of the physical world*. Cambridge University Press, Cambridge New York, 1989. ISBN 9780521376051.

[5] Isaac Newton, N. W. Chittenden, and Andrew Motte. *Newtons Principia: The mathematical principles of natural philosophy*. 1729.

[6] Michael E. Browne. *Schaums outlines physics for egineering and science*. McGraw-Hill, 1999.

[7] Leonhard Euler. *Institutionum calculi integralis*. 1768.

[8] John Charles Butcher. *Numerical methods for ordinary differential equations*. Wiley, 2008.

[9] Wikipedia. Leapfrog integration — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Leapfrog%20integration&oldid=949509207, 2020.

[10] Nikos Drakos and Ross Moore. Two ways to write the leapfrog, Jan 2004. URL http://www.artcompsci.org/vol_1/v1_web/node34.html.

[11] Dirk Hünniger. Cropped cone, Jun 2008. URL https://commons.wikimedia.org/wiki/File:CroppedCone.svg.

[12] Oracle. Open JFX 9. https://github.com/teamfx/openjfx-9-dev-rt/, 2016.

[13] Sho Nakamura. Matplotlib for Java. https://github.com/sh0nk/matplotlib4j, 2017.

[14] Martin Pernollet. Scientific 3d plotting. URL http://www.jzy3d.org/.

[15] NASA & JPL. Horizons system. URL https://ssd.jpl.nasa.gov/?horizons.

[16] Google. GSON - A Java serialization/deserialization library. https://github.com/google/gson, 2008.

[17] Apache Commons. Apache commons csv. URL https://commons.apache.org/proper/commons-csv/.

[18] All you need to know about eclipses. URL https://www.timeanddate.com/eclipse/.

[19] Nasa. Solar and lunar eclipses, Feb 2017. URL https://www.kaggle.com/nasa/solar-eclipses.

[20] Amaury Triaud and Michael Gillon. Trappist-1. URL http://www.trappist.one/#about.

[21] Simon L. Grimm, Brice-Olivier Demory, Michaël Gillon, Caroline Dorn, Eric Agol, Artem Burdanov, Laetitia Delrez, Marko Sestovic, Amaury H. M. J. Triaud, Martin Turbet, and et al. The nature of the trappist-1 exoplanets. *Astronomy & Astrophysics*, 613:A68, May 2018. ISSN 1432-0746. doi: 10.1051/0004-6361/201732233. URL http://dx.doi.org/10.1051/0004-6361/201732233.

[22] William M Folkner, James G Williams, Dale H Boggs, Ryan S Park, and Petr Kuchynka. The planetary and lunar ephemerides de430 and de431.

[23] Bradley W. Carroll and Dale A. Ostlie. *An introduction to modern astrophysics.* Cambridge University Press, 2018.

[24] E. Bolmont, B.-O. Demory, S. Blanco-Cuaresma, E. Agol, S. L. Grimm, P. Auclair-Desrotour, F. Selsis, and A. Leleu. Impact of tides on the transit-timing fits to the trappist-1 system. *Astronomy & Astrophysics*, 635:A117, Mar 2020. ISSN 1432-0746. doi: 10.1051/0004-6361/202037546. URL http://dx.doi.org/10.1051/0004-6361/202037546.