PLANET

9

Daniel Boutros

Christiaan Goossens

PWS
PLANET NINE

Written by

DANIEL BOUTROS
CHRISTIAAN GOOSSENS

Stedelijk Gymnasium Nijmegen
Nijmegen, the Netherlands
2016

Supervised by

DRS. PETER RENNSPIES

2016
STEDELIJK GYMNASIUM NIJMEGEN

Contents

List of Figures

Chapter 1

Abstract

In (Batygin and Brown, 2016b) the existence of a hypothetical ninth planet is predicted. In this thesis we will find out why this planet has been predicted and we will build a computer model which will calculate and simulate whether or not the existence of a ninth planet is really needed and if it is indeed, determine its mass (to a certain degree of precision). We also discuss some other possible solutions to the existing situation in the solar system and also if planet nine is the best solution.

Part I

Introduction

Chapter 2

Introduction

> Imagination governs the world.
>
> Napoleon Bonaparte

On the 20th of January 2016, Konstantin Batygin and Michael E. Brown published an article in The Astronomical Journal called: "Evidence for a distant giant planet in the solar system" in which they predict the existence of a huge planet of multiple earth masses in the far outer solar system (Oort cloud). We will from know on refer to this predicted planet as planet nine.

At that time, we were in the process of choosing a subject for our high school thesis (PWS). We wanted to choose a subject within planetary science and preferably concerning the solar system and suddenly, our opportunity arrived. We chose planetary science as a starting point because we wanted a subject with a good mix of computer science (because of Christiaan's excellence in that subject), mathematics, physics (because we both like physics) and a little scientific history.

Figure 2.1: An artist's impression of planet nine.[2]

We will list a few conventions which will be used in this thesis. Angles are (mostly) measured in degrees. Vectors are boldfaced. This thesis was typeset and produced in LaTeX, which is the standard way of writing documents in the sciences.[3] [4] [5]

We would like thank the following people for all their help and time which helped us a great deal along the way to develop this thesis:

- First, our 1st supervisor drs. Peter Rennspies for the time he took to give us good advice on writing the parts of this thesis and giving us an idea about the physical basis of the computer model. We would also like to thank him for the careful reading of the manuscript.

- Our 2nd supervisor drs. Wine van Huijzen for helping Daniel with difficult mathematics such as differential equations.

- Prof. Dr. S. Portegies Zwart (who specialises in computational astrophysics) who gave us immensely useful guidance on the building of the computer model and the way of

---

[2](Wikipedia, 2016p)
[3](Wikipedia, 2016n)
[4](Wikipedia, 2016o)
[5](van Oostrum, 2016)

determining a position of planet nine. He also helped us with the statistics and made us really think about the existence of planet nine.

- Prof. Dr. V. Icke (who is specialised in theoretical astronomy) from the university of Leiden. He put the article (Batygin and Brown, 2016b) (for us) into another perspective and made us realize that the existence of planet nine isn't at all certain.

- Prof. Dr. G.J. Heckman (specialised in mathematical physics) from Radboud University (in Nijmegen) who gave Daniel explanation of his own proof of the Kepler's first law and helped him with proofs of the others as well.

- Prof. Dr. P.J. Mulders (who is specialised in theoretical physics) from the NIKHEF Theory Group and the VU (Amsterdam). He brought us into contact with professors Icke and Portegies Zwart and also gave us a wonderful tour of the NIKHEF institute.

- Prof. Dr. F. Verbunt (specialised in high-energy astrophysics) also from the Radboud University for the helpful explanations about the prediction of the existence of planet nine and the other scientific articles about this subject.

- Prof. Dr. N de Groot (specialised in experimental high-energy physics) from Radboud University for making lecture notes on analytical mechanics available to us.

- Prof. Dr. G. Nelemans (specialised in double white dwarfs) from Radboud University for directing us towards the people who are knowledgeable about the subject.

- Prof. Dr. H. Zantema (specialized in mathematical computer science) from the Eindhoven University of Technology for taking the time to talk with Christiaan about the technical side of the computer model.

- Dr. A. van den Essen from Radboud university for giving both of us a course in linear algebra in the academic year 2015-2016 at the Radboud University in linear algebra which helped smoothen the way for this project.

- Dr. W. Bosma from Radboud university for the extra courses on Linear Algebra which Daniel took in 2016 and all the time he provided to explain mathematical concepts such as definitions of number systems, vector spaces, markov chains, Dedekind cuts.

- Prof. Dr. H.T. Koelink from Radboud university for the courses of calculus that both Christiaan and Daniel took in the autumn of 2016 and for his answers to our questions about numerical methods, differential equations etc.

- (Graphical designer and our friend) Floris Thoonen for designing the wonderful cover.

Christiaan Goossens & Daniel Boutros
May 2016

Chapter 3

The Aims of our Thesis

> You can tell whether a man is clever by his answers. You can tell whether a man is wise by his questions.
>
> Naguib Mahfouz

Our goal in this project is to determine whether the predicted (but still hypothetical) planet nine is likely to exist. It leads to the following research question:

Is planet nine the best possible solution to the unexplained situation in the solar system?

We will split this question into the following five sub-questions:

1. *How was planet nine predicted?*

2. *Is its existence really needed to explain the present situation of our solar system?*

3. *If the answer to question 2 is yes, what is its mass (determined to a certain degree)?*

4. *Is there another solution by which the present situation in our solar system can be explained?*

We want to answer these questions by building a computer model which simulates the solar system accurately enough to really assess if the problem of explaining the current alignment of objects in the solar system requires a big planet in the outer solar system to stabilize it and if so, to say something about its likelihood. We want to build this model without any bias and will also give some attention to its precision.

Chapter 4

Methods

> Therefore measure in terms of five things, use these assesments to make comparisons, and thus find out what the conditions are. The five things are the way, the weather, the terrain, the leadership, and discipline.
>
> Sun Tzu

We are going to answer this question in this thesis:

Is planet nine the best possible solution to the unexplained situation in the solar system?

In this chapter we will discuss the way and methods by which we are going to answer the sub-questions and eventually the main question. The first sub-question is:

How was planet nine predicted?

We are going to answer this by reading the literature on the subject. Here the main article (Batygin and Brown, 2016b) is of course of great importance because this is the article that brought us to this subject in the first place and the whole thesis is about the prediction made in this article. The article leads us to other important research articles on this subject which together form the source of research information for this thesis. Of course we are also going to use books, lecture notes, articles on other 'subjects', websites etc. as background literature on mathematical, physical, astronomical and computer science subjects which are gaps in our background knowledge and which are need for a full understanding of the research on planet nine. We will treat all of the missing necessary background knowledge for students at pre-university level in this thesis, so that will be the subject of the next few chapters.

The second sub-question was:

Is its existence really needed to explain the present situation of our solar system?

Like we have mentioned in the previous chapter, we are planning to build a computer model to simulate the solar system. This model is planned to have several components (all programmed in Java):

- First, it has a part to simulate the solar systems with calculations.

- Second, it has another part which calculates certain orbital elements.

We will use the model (especially the first part) to simulate the solar system without a planet nine to see if the observed effects/properties on which the prediction of existence of planet nine was based are maintained/constant. If that is the case, there is no need for a planet nine, because the known present situation (by which the known objects such as planets are meant) in the solar system cause the effects observed. Otherwise we can conclude that a planet nine is needed (there are also other solutions, which we will discuss later).

The third sub-question is:

If the answer to question 2 is yes, what is its mass (determined to a certain degree)?

Again the model will be very important here. Because of the (probable) great uncertainty, we will not be able to approximate planet nine's mass with great precision. We will take a possible position of planet nine from the literature and will then test it using different amounts of mass to say roughly how much mass planet nine has.

The fourth sub-question is:

Is there another solution by which the present situation in our solar system can be explained?

We will once more turn to the literature for this question. Maybe we will also think up an original solution.

Now we will talk about the main question:

Is planet nine the best possible solution to the unexplained situation in the solar system?

We will attempt to answer this question by (again) looking at the vast (scientific) literature on this subject. But this not only includes the (current) research articles, we will also makes use of several books (and other sources of information) on subjects like celestial mechanics, advanced classical mechanics etc. (which all will be discussed later). We will then use our own judgement and maybe the computer model to determine which several proposed (or maybe one of our own) solutions can be possible. An important role plays the sub-question about the origins of planet nine. If it is hard to explain where planet nine came from (or for example why it has not been detected yet because of great brightness), than it will be less likely that planet nine exists.

Chapter 5

Hypothesis

> I have no religion, and at times I wish all religions at the bottom of the sea. He is a weak ruler who needs religion to uphold his government; it is as if he would catch his people in a trap. My people are going to learn the principles of democracy, the dictates of truth and the teachings of science.
>
> Mustafa Kemal Atatürk

In this chapter, we will discuss our hypothesis of the various sub-questions and finally the research question. If the reader is not well-versed in the needed background knowledge for this kind of planetary research we urge him/her to first read the part 'Introduction to the Different Subjects'

5.1 How was planet nine predicted?

You cannot give a real hypothesis to this question, but you can suspect the prediction is due to a clustering of some orbital feature in all the inner Oort cloud objects.

5.2 Is its existence really needed to explain the present situation of our solar system?

Yes, otherwise the argument of the argument of perihelion of an object such as Sedna would change costantly due to the Kozai mechanism, which means that it will be shepherded by Neptune. The argument of perihelion of these ETNO's is conserved so one can conclude that another large body is needed to preserve the present situation. If for all the objects modelled in the computer model (without a planet nine) the argument of perihelion shifts a significant number of degrees, we can say that the answer to the question is yes. If it remains clustered around a certain value, the answer is no, because no planet nine is required to preserve the situation as it is, because the known solar system does it already for us.[1] [2]

---

[1](de Pater and Lissauer, 2016)
[2](Wikipedia, 2016h)

5.3 If the answer to question 2 is yes, what is its mass (determined to a certain degree)?

You cannot give a hypothesis to this question, but due to the literature we suspect that its mass is probably between 5 and 20 Earth masses.[3] [4] [5]

5.4 Is there another solution by which the present situation in our solar system can be explained?

We think so. It could still be coincidence because there are still too few objects discovered (just six). It could also be due to a stellar encounter for example.

5.5 Is planet nine the best possible solution to the unexplained situation in the solar system?

We earlier concluded that planet nine is needed to conserve the situation. But one can also conclude that there are restrictions on the variance of its mass. It cannot be too big, otherwise it would have been easily observed already. However, it can also not be too small, then it would have little effect on Oort cloud objects. It is also a problem to explain how it got there in the first place (which is addressed by the last sub-question). For these reasons we think that another solution is a better explanation because we think that the existence of planet nine is unlikely because there are so many conditions to must be true if planet nine exists.

---

[3](Batygin and Brown, 2016b)
[4](de la Fuente Marcos et al., 2016)
[5](Batygin and Brown, 2016a)

Chapter 6

The Build-up of this thesis

> Most of us seldom take the trouble to think. It is a troublesome and fatiguing process and often leads to uncomfortable conclusions.
>
> Jawaharlal Nehru

For this thesis, we assume that the reader has basic high school background knowledge. Particularly, basic knowledge of mathematics and physics is very helpful. We mean knowledge of basic differential and integral calculus, basic arithmetic skills, $F = ma$ style physics etc. All the other background knowledge is covered in this thesis. We will now discuss the build-up of the document.

In the first chapters, we are going to provide the necessary background for the rest of the thesis. We will start off with the definition of basic terms in the planetary sciences, such as a planet, dwarf planet, Oort cloud and so on. Then a small historical chapter will follow to put this scientific development (around planet nine) into perspective. After that we will discuss essential mathematics, physics and astronomy. In the case of mathematics that will include linear algebra, calculus, geometry etc. In the physics that will be mostly classical mechanics. In the astronomical chapter, We will begin with some basic astronomical conventions and then we will mostly cover orbital mechanics.

After this necessary introduction, we begin our real research. We first start with summarizing the current research on planet nine, to make sure that the reader knows the background and the place of this thesis. After that we will state the goal of our computer model an then give an introduction to scientific computing (including various types of numerical integration) and then we will describe the way our model works. We will first do that in terms of physics and in a later chapter in terms of computer science. We will then show and discuss the results of our computer model. We will conclude with a discussion and a conclusion. After the bibliography we include summaries of our meetings with several professors and the entire code of the computer model.

Part II

Introduction into the Different Subjects

Chapter 7

The Build-up of the Solar System

Since 2006 the solar system consists of eight planets, the sun, several (there is no definite number yet) dwarf planets, KBO's (Kuiper Belt Objects), asteroids, comets, meteoroids, moons etc. Before we even start talking about the research we have done we would like to precisely define what some of these terms mean so that there can be no confusion later on.[1]

7.1 The Sun & (dwarf)planets

First, by far the biggest object of our solar system is the sun, an average star on the main sequence. It contains 99,8 percent of the total mass of the solar system. Secondly, as of November 2016, we have eight planets in our solar system, they are (in order of increasing distance from the sun): Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune. According to the IAU (the International Astronomical Union), the precise definition of a planet is:[4] [5]

"A planet is a celestial body that (a) is in orbit around the Sun, (b) has sufficient mass for its self-gravity to overcome rigid body forces so that it assumes a hydrostatic equilibrium (nearly round) shape, and (c) has cleared the neighbourhood around its orbit."[6]

Figure 7.1: The solar system, the sizes of the planets and the sun are scaled, the distances are not.[3]

---

[1](Watson et al., 2007)
[4](Watson et al., 2007)
[5](Wikipedia, 2016c)
[5](Wikipedia, 2016m)
[6](Wikipedia, 2016c)

In other words, it means that each planet has to have the following properties:[7] [8]

1. It has to orbit the sun and not another body in the solar system. Because of this property, the sun and several big moons in the solar system are not planets.

2. It has to have enough mass to make itself a (nearly perfect) sphere. However, the precise definition of a nearly perfect sphere is still unclear.

3. Clearing the neighbourhood around an orbit means that an object clears its own orbit of small bodies such as small asteroids etc. Again, as with 2, this is not clearly defined either. It can be argued that the Earth, Jupiter and Neptune for example don't meet this criterion either. For example if Neptune had cleared its orbit, Pluto wouldn't be there (because their orbits cross), although Earth, Jupiter and Neptune are clearly planets according to the IAU.

Dwarfplanets are bodies who meet criteria 1 and 2, but not 3 (and the IAU also states that it is not a satellite, but that also indirectly follows from 1). Examples of dwarf planets are Pluto, Ceres (who is in the asteroid belt between Mars and Jupiter) and Eris (who is even bigger than Pluto and which was also discovered by Mike Brown). Note that objects which meet the IAU criteria but orbit another star instead of the sun are called exoplanets.[9] [10] [11]

7.2 Other Objects

Another important object is a moon, we're not just talking about Earth's moon (the Moon) but also about other planet's moons. The criterion for a body to be a moon is that it has to orbit another body in the solar system which is not the sun. Note that not only planets, but also dwarf planets, asteroids etc. can have moons. There are no bounds for the mass, diameter etc. that a moon needs to have. There also exist other types of objects in our solar systems, such as asteroids, comets, meteorites, meteors etc. For the precise explanation of these terms, we would like to direct you to other literature (Ridpath, 2012), (Kutner, 2003), (Maran, 2012) & (Schilling, 2012).[12]

7.3 The different regions of the solar system

First, at the centre of the solar system, you have of course the sun. Then come the terrestrial planets (in order of increasing distance): Mercury, Venus, Earth and Mars. After Mars comes the asteroid belt, which contains small bodies made of rock. Then come the giant gas planets: Jupiter, Saturn, Uranus and Neptune. Uranus and Neptunus are also collectively known as the ice giants because they gathered also a lot of icy material in their formation next to rocky material and gas. After that comes the Kuiper belt (which contains Pluto) and in the far outer reaches of the solar system exists the Oort cloud which consists of comets

---

[7](Watson et al., 2007)
[8](Wikipedia, 2016c)
[9](Watson et al., 2007)
[10](Wikipedia, 2016c)
[11](Brown et al., 2005)
[12](Wikipedia, 2016c)

and other icy bodies. Objects in these region are sometimes called: ETNO's (Extreme Trans Neptunian Objects).[13] [14] [15] [16] [17]

---

[13](Watson et al., 2007)

[14](Maran, 2012)

[15]Schilling (2012)

[16](Baker, 2011)

[17](de Pater and Lissauer, 2016)

Chapter 8

Historical Background

> I did not intend to kill Pluto when I
> started out. I was actually looking for a
> tenth planet.
>
> Mike Brown

8.1   Ancient World

Figure 8.1: Ptolemy (c. AD 100 - c. AD 170)[2]

For two millennia the ancient Greeks dictated the way we thought about and approached the universe and especially our solar system. The existing astronomical knowledge was first gathered by Aristotle, the famous Greek philospher. It was then refined by Ptomely a Greek scientist living in the then very important city of Alexandria, in Egypt.

Ptolemy believed in the geocentric model, meaning that the earth was at the centre of our universe. He also thought that planetary orbits were perfect circles (which was later undermined by Johannes Kepler), however his for that time very accurate observations showed that the planetary orbits are not perfect circles. Ptolemy solved this problem by putting circles on circles (also called epicycles). But the retrograde movement of the planet Mars in the sky, meaning that Mars would change direction, from the perspective of people on earth, and then change direction once again, could not be explained by the geocentric model. Even in ancient times, scientists considered the addition of epicycles to the model as a last rescue. 1700 years before Copernicus, the heliocentric model (see the next section) had already been proposed. It then was already clear that the geocentric model had to be changed somehow.[3] [4]

8.2   Copernicus

In 1543, on his deathbed, Nicolas Copernicus (1473-1543) published a book (De Revolutionibus Orbium Coelestium) that would change the face of science forever. In this book he

---

[2](Wikipedia, 2016k)
[3](Limburg et al., 2015)
[4](Boutros, 2013)
[5](Hyperphysics, 2016)
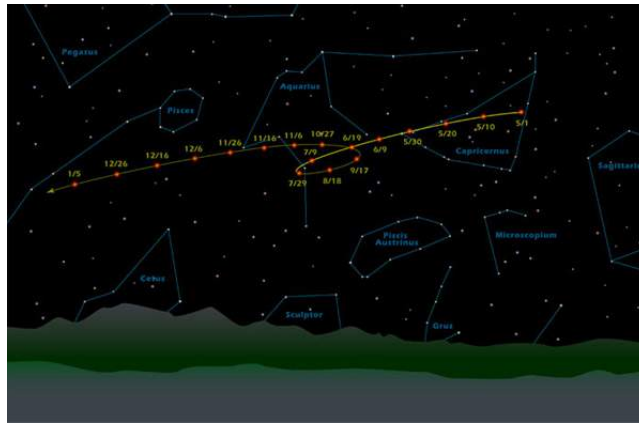
Figure 8.2: The retrograde motion of Mars, as seen from earth.[5]

proposed a new model to explain the movement of heavenly bodies, called the heliocentric model. It was definitely simpler and more elegant than the geocentric model. In the new model, Earth was no longer at the centre of the universe or even the solar system, but the sun was. Copernicus himself said in the introduction to his book that it was just a mathematical model and it wasn't realistic to prevent being attacked by the catholic church. But the advantages of the model were clear, and scientists would continue to support and expand it.[6] [7]

8.3   Brahe, Kepler and Galilei

Copernicus' hypothesis was not readily accepted by the scientific community. Tycho Brahe (1546-1601) made, like Ptolemy, for his time very accurate observations. With an increase in accuracy of 300% compared to his predecessors. He thought he had disproven Copernicus' model with his experimental work. His research assistant, Johannes Kepler (1571-1630) based three laws of planetary motion on Brahe's observations after Brahe's death. He had become Brahe's research assistant after he had moved from Graz (in Austria) to Prague because of religious reasons. After Brahe died he became the Imperial Mathematician himself (he stayed at this post until 1612). His conclusions from the research were that planetary are not prefect circles, but ellipses. He concluded this from studying the orbit of Mars. To some, his laws may seem be obvious, but it must be remembered that he discovered them without any calculus, analytic geometry, newtonian mechanics etc. Slowly, but gradually problems began to arise with the geocentric model. Kepler's contributions are significant because he was among the first to introduce the concept of a natural law, which is a physical law that is uniform (on earth as well as in the rest of the universe). The work of Kepler formed part of the groundwork for Newton's laws (including the law of gravitation). During Kepler's time, a man named Galileo Galilei lived in Italy. After the telescope had been invented in Holland around 1610 he built one himself and started to do observations with it. He discovered Jupiter's four big moons and also observed our own moon. Because of these observations he became convinced that Copernicus was right and publicized several books on it. He became one of the most important exponents of the heliocentric model. Eventually, he was

---

[6](Limburg et al., 2015)
[7](Boutros, 2013)

put under housearrest for the rest of his life.[8] [9] [10] [11]

8.4   Newton

The work of Copernicus, Brahe, Kepler, Galilei was combined into one great theory and model by sir Isaac Newton. He stated his famous three laws of motion and the well-known law of universal gravitation. These laws were discussed in one of the most famous books of science: Philosophiae Naturalis Principia Mathematica (Latin for Mathematical Principles of Natural Philosophy). It is commonly referred to as the Principia. In this book Newton derived the Kepler laws from his own laws. Newtonian mechanics remained the best physical description of the world until Einstein appeared over 200 years later at the beginning of the twentieth century.[12] [13] [14] [15] [16]

8.5   The Discoveries of Uranus and Neptune

In ancient times, the Greeks and Romans knew only the five planets which are visible with the naked eye from Earth: Mercury, Venus, Mars, Jupiter and Saturn. Uranus was discovered in 1781 by William Herschel, a British astronomer and composer of German origin. Herschel made observations of the night sky and discovered several moons in our solar system, nebulae, galaxies and Uranus.

More than half a century later, John Couch Adams (a British astronomer) and Urbain Le Verrier (a French astronomer) calculated the predicted orbit of Uranus and found that it differs from Uranus' real (observed) orbit. They concluded that there must exist another planet which causes Uranus to move into the observed orbit. They proceeded to calculate what the mass and position of this eight planet must be and they published their predictions simultaneously. Then, German astronomer Johann Gottfried Galle observed the predicted position and found Neptune.

8.6   The Discovery of Pluto

After the discovery of Neptune, astronomers continued to calculate the orbit of Uranus and they found that the effect of Neptunus was 'not enough' to explain the difference between the observed and predicted orbit. Even Neptune itself didn't follow its predicted orbit. Thus a new planet was hypothesized. So Percival Lowell, an American astronomer, founded an observatory to search for the predicted planet. Lowell passed but Clyde Tombaugh found Pluto, which was named the ninth planet, in 1930. However, Pluto was far too small and

---

[8](Limburg et al., 2015)

[9](Boutros, 2013)

[10](Byrne, 2014)

[11](Wikipedia, 2016g)

[12](Heckman, 2015)

[13](Limburg et al., 2015)

[14](Boutros, 2013)

[15](Byrne, 2014)

[16](Wikipedia, 2016j)

didn't have enough mass to perturb Uranus and Neptune enough to explain the observed orbits. Later it turned out that the masses of Neptune and Uranus that were used in the calculations differed very much from their real masses.

8.7 The Degradation of Pluto

In 2005 Mike Brown and his team discovered Eris, a body in the Kuiper Belt which turned out to be bigger than Pluto. Now the IAU held a congress to determine whether Eris would be identified as the tenth planet or that Pluto and Eris would be put in a different class. The vote was taken and Pluto was demoted and became a dwarf planet. At this congress the definition of a planet (which was cited in a previous chapter) was established for the first time.

As you can see, it is not the first time that a new planet has been hypothesized. Maybe Mike Brown wants to make up for 'the planet he killed"(he wrote about it) by hypothesizing another one...

Chapter 9

Mathematical Introduction

> "Do you know what a mathematician is?" Lord Kelvin asked his class. He then stepped to the board and wrote:
>
> $$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$$
>
> Then he put his finger on the board and said: "A mathematician is one that is as obvious as twice two makes four to you."
>
> — William Thomson Kelvin

In this thesis we will make extended use of linear algebra (also called vector space theory), it helps if the reader has some familiarity with it. However, all the mathematical concepts new to high school students used in this thesis are explained here.[1]

9.1   Matrices

In this thesis we will make extended use of matrices, so we will define those first:

Definition 9.1.1. *A matrix is a rectangular array of the form:*

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}$$

The entries a_{ij} are from a given field. A field is a set where the result with any two of the four operations of addition, subtraction, multiplication and division (not by zero) on any two elements (from the field) is an element of the field and where the operations satisfy certain axioms which we will not mention here. [2] [3] [4] The number of rows is called m and the number of columns is called n, so this matrix is called a $m \times n$ matrix.[5] The definition of matrix multiplication is as follows:

Definition 9.1.2. *The product of the two matrices A and B, which has as result AB, can only exist if and if only the number of columns of A is the same as the number of rows of B, this means that*

---

[1](Terwijn, 2014)

[2](Lenstra jr. et al., 2014)

[3](van den Essen, 2015a)

[4](Bosma, 2016)

[5](Friedberg et al., 2003)

A has to be an m×n matrix and B has to be an n×p where m and p don't necessarily have to be the same as each other. Note that this only applies to the product AB and not BA (the multiplication of matrices is normally not commutative). When a product is commutative it means that $x \cdot y = y \cdot x$. In the case of matrices it is not necessarily the case that $AB = BA$. [6]), you will see now why that is true. The entry in the row i and column j in the product of matrices A and B is defined as:[7] [8]

$$(AB)_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \ldots + A_{in}B_{nj} = \sum_{k=1}^{n} A_{ik}B_{kj} \tag{9.1}$$

$(AB)_{ij}$ denotes the entry in the i-th row and the j-th column from the matrix AB (and the same goes on for the other variables). So you multiply the entries in row i (from matrix A) with the entries from column j (from matrix B) with each other. We will now prove some rules about matrix multiplication:

Theorem 9.1.1. *Let A be an $m \times n$ matrix, B an $n \times p$ matrix and C an $p \times q$ matrix. λ is a certain scalar[9] [10]. Then the following statements are always true (which will be important later):*

   i)

$$\lambda AB = A\lambda B = AB\lambda \tag{9.2}$$

   ii)

$$(AB)C = A(BC) \tag{9.3}$$

Proof. Proving i) is just writing out the multiplication, we will leave that to the reader. We will discuss the proof of ii) (also called the associativity[11] of the multiplication) in the next section: $\qquad \square$

9.2   Proof of theorem 9.1.1ii)

We will prove this statement by writing the statement out:

Proof.

$$(AB)C_{ij} = \sum_{k=1}^{p}\left(\sum_{z=1}^{n}(a_{iz}b_{zk})c_{kj}\right) = (a_{i1}b_{11} + \ldots + a_{in}b_{n1})c_{1j} + \ldots + (a_{i1}b_{1p} + \ldots + a_{in}b_{np})c_{pj} \tag{9.4}$$

$$A(BC)_{ij} = \sum_{k=1}^{n}\left(a_{ik}\sum_{z=1}^{p}(b_{kz}c_{zj})\right) = a_{i1}(b_{11}c_{1j} + \ldots + b_{1p}c_{pj}) + \ldots + a_{in}(b_{n1}c_{1j} + \ldots + b_{np}c_{pj}) \tag{9.5}$$

$$a_{i1}(b_{11}c_{1j} + \ldots + b_{1p}c_{pj}) + \ldots + a_{in}(b_{n1}c_{1j} + \ldots + b_{np}c_{pj}) = (a_{i1}b_{11} + \ldots + a_{in}b_{n1})c_{1j} + \ldots + (a_{i1}b_{1p} + \ldots + a_{in}b_{np})c_{pj} \tag{9.6}$$

It follows from the calculations that:

$$(AB)C_{ij} = A(BC)_{ij} \tag{9.7}$$

---

[6](Friedberg et al., 2003)
[7](van den Essen, 2015b)
[8](H.van Gendt and Dames, 2008)
[9]A scalar is a number or coefficient which has only a magnitude and no direction (a vector has a direction).
[10](Friedberg et al., 2003)
[11]van den Essen (2015a)

This means that $(AB)C$ and $A(BC)$ have the same number on each position (i, j) and therefore, the two matrices are the same. This proves theorem 1.1.1 ii)[12]. □

Now we move on to another set of important mathematical objects, which is closely related to matrices.

9.3 Vectors

Definition 9.3.1. *A vector in the three-dimensional space ($\mathbb{R}^3$) is normally written as a 3×1 matrix like this:*

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Where x denotes the x-component of its position, y denotes the y-component and so on.

One of the properties of a vector is its length, this leads to the definition of the length of a vector in the three-dimensional space:

Definition 9.3.2. *The length of a vector v is denoted with $\|v\|$, it is defined in the $\mathbb{R}^3$ as:*[13]

$$\|\mathbf{v}\| = \sqrt{x^2 + y^2 + z^2} \tag{9.8}$$

Later in the physics part of this thesis we will make use of a matrix to calculate the sum of all the gravitational forces on a certain (planetary) body (object), but first will have to define mathematically what we are going to do:

Theorem 9.3.1 (Pythagoras)**.** *This theorem is the well-know Pythagorean theorem: Let ABC be a right triangle where c is the length of the hypotenuse, which is the side opposite the right angle and a and b are the lengths of the other two. Of course the following is always true:*[14]

$$a^2 + b^2 = c^2 \tag{9.9}$$

Proof. We will leave the obvious proof (there are of course multiple ones) of this to the reader. □

After we have calculated the gravitational force on an object in the different components (x, y, z) we will have to calculate the resulting total gravitational force on the object in the three-dimensional space ($\mathbb{R}^3$). Using the Pythagorean theorem, we will now prove the following theorem:

Theorem 9.3.2. *Let a, b and c denote certain sides (in the physical case x-, y- or z-coordinates), then the resulting length of the vector with these coordinates is equal to:*

$$\sqrt{a^2 + b^2 + c^2} \tag{9.10}$$

---

[12]van den Essen (2015b)
[13](Friedberg et al., 2003)
[14](Wikipedia, 2016l)

Proof. Let a and b be coordinates in the xy-plane. Then the vector $\mathbf{x}$ with these coordinates has the following length according to theorem 9.3.1:

$$\|\mathbf{x}\| = \sqrt{a^2 + b^2} \tag{9.11}$$

Let the vector $\mathbf{d}$ be formed by $\mathbf{x}$ and another component c. Its total length is then equal to:

$$\|\mathbf{d}\| = \sqrt{\|\mathbf{x}\|^2 + c^2} \tag{9.12}$$

But it follows immediately that:

$$\sqrt{\|\mathbf{x}\|^2 + c^2} = \sqrt{\left(\sqrt{a^2+b^2}\right)^2 + c^2} = \sqrt{a^2 + b^2 + c^2} \tag{9.13}$$

$\square$

Definition 9.3.3. *Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be vectors so that:*

$$\boldsymbol{a} = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}$$

$$\boldsymbol{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

Then the inner product (also called scalar product) of $\boldsymbol{a}$ and $\boldsymbol{b}$, denoted as $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$ is defined as:[15] [16]

$$\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=1}^{n} a_i b_i \tag{9.14}$$

This is often called the standard inner product and the result of this operation is a scalar. Now we can generalize the concept of length to a lot of other cases:
Definition 9.3.4. *Let $\boldsymbol{a}$ be a certain vector, then its length is defined as:*

$$\|\boldsymbol{a}\| = \sqrt{\langle \boldsymbol{a}, \boldsymbol{a} \rangle} \tag{9.15}$$

This definition makes clear why theorem 9.3.2 is true. The geometric idea behind the inner product is as follows:[17]
Theorem 9.3.3. *Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be certain non-zero vectors with an angle θ which is between 0 and π between the two vectors. Than the inner product has the following geometrical meaning:*

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \|\boldsymbol{u}\|\|\boldsymbol{v}\|\cos\theta \tag{9.16}$$

Proof. Let $\mathbf{u}$ and $\mathbf{v}$ be certain vectors in the $\mathbb{R}^2$. The angle between them is called α. If you connect the endpoints of the two vectors you get the difference vector $\mathbf{v} - \mathbf{u}$, as is shown in

---

[15](Friedberg et al., 2003)
[16](Heckman, 2015)
[17](Heckman, 2015)

the picture. Together, these three sides form a triangle. The law of cosines states for this triangle:[18] [19] [20] [21]

$$\|\mathbf{v} - \mathbf{u}\|^2 = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 - 2\|\mathbf{u}\|\|\mathbf{v}\|\cos\alpha \tag{9.17}$$

But according to the previous definition, the following equation also holds (because of the properties of the inner product, which is a bilinear form):[22]

$$\|\mathbf{v} - \mathbf{u}\|^2 = \langle \mathbf{v} - \mathbf{u}, \mathbf{v} - \mathbf{u} \rangle = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 - 2\langle \mathbf{u}, \mathbf{v} \rangle \tag{9.18}$$

Because the two previous equations are equal, it follows that:

$$-2\|\mathbf{u}\|\|\mathbf{v}\|\cos\alpha = -2\langle \mathbf{u}, \mathbf{v} \rangle \tag{9.19}$$

$$\langle \mathbf{u}, \mathbf{v} \rangle = \|\mathbf{u}\|\|\mathbf{v}\|\cos\alpha \tag{9.20}$$

$\square$

You also have another vector operation:

Definition 9.3.5. *Let a and b be certain vectors in the three-dimensional space ($\mathbb{R}^3$):*

$$a = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Then the vector product, denoted by $a \times b$ has a vector as result, is:[23]

$$a \times b = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix} \tag{9.21}$$

It has the following properties:

Theorem 9.3.4. *It follows immediately that the vector product is not commutative and that the following properties are always true (λ is a certain scalar):*

Figure 9.1: Figure belonging to the proof of theorem 9.3.3.

$$a \times b = -b \times a \tag{9.22}$$

$$a \times (b + c) = a \times b + a \times c \tag{9.23}$$

$$(\lambda a) \times b = a \times (\lambda b) = \lambda(a \times b) \tag{9.24}$$

$$a \times a = 0 \tag{9.25}$$

Proof. We will leave the proof by writing out the coordinates to the reader. $\square$

---

[18](Heckman, 2015)
[19](Friedberg et al., 2003)
[20](Bouwens et al., 2013)
[21](Wikipedia, 2016f)
[22](Friedberg et al., 2003)
[23](Heckman, 2015)

You can even extend the last theorem, but first we need the following definition and theorem.

Definition 9.3.6. *Two vectors a and b are proportional whenever:*[24]

$$\langle a, b \rangle^2 = \|a\|^2 \|b\|^2 \tag{9.26}$$

We will now make this geometrically clear:

Theorem 9.3.5. *Two vectors a and b are proportional if and only if they point in the same direction or opposite directions.*

Proof. According to an earlier proven theorem, the geometric meaning of the inner product is:

$$\langle \mathbf{a}, \mathbf{b} \rangle = \|\mathbf{a}\|\|\mathbf{b}\| \cos \theta \tag{9.27}$$

It then follows that:

$$\langle \mathbf{a}, \mathbf{b} \rangle^2 = \|\mathbf{a}\|^2 \|\mathbf{b}\|^2 \cos^2 \theta \tag{9.28}$$

The following equation has to hold whenever vectors are proportional:

$$\langle \mathbf{a}, \mathbf{b} \rangle^2 = \|\mathbf{a}\|^2 \|\mathbf{b}\|^2 \tag{9.29}$$

So you can conclude that:

$$\cos^2 \theta = 1 \tag{9.30}$$

$$\cos \theta = \pm 1 \tag{9.31}$$

It follows that the angle (θ) between the two vectors has to be 0 or 180 degrees, which means that the two vectors are in the same or opposite directions, which is exactly what we wanted to prove. You can reason backwards to prove it the other way round. $\square$

Lemma 9.3.1. *Whenever two vectors are proportional, one convert one into the other by multiplying the vector with a certain scalair.*

Proof. Drawing a picture will make this immediately clear. $\square$

Now we can finally prove this theorem:

Theorem 9.3.6. *Whenever vectors a and b are proportional, their vector product is 0.*

Proof. According to the previous lemma, the following equations holds (for a certain scalar):

$$\mathbf{a} \times \mathbf{b} = \mathbf{a} \times (\lambda \mathbf{a}) = \lambda (\mathbf{a} \times \mathbf{a}) \tag{9.32}$$

According to an earlier theorem, this is obviously equal to zero. $\square$

We will have to make an important note on vector calculus before we move on (this will be important later). Suppose you have a vector which is dependant on a certain variable t (which does not necessarily mean time):[25] [26]

$$\mathbf{r}(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix} \tag{9.33}$$

[24](Heckman, 2015)
[25](Byrne, 2014)
[26](Cowley, 2000)

Then the derivative of this vector is:

$$\frac{d\mathbf{r}(t)}{dt} = \begin{pmatrix} \frac{dx(t)}{dt} \\ \frac{dy(t)}{dt} \\ \frac{dz(t)}{dt} \end{pmatrix} \tag{9.34}$$

So when you differentiate a vector, you differentiate with respect to each of its components. A dot above a vector denotes its derivative with respect to time.

9.4 Geometry

When we will talk about planet orbits, which are ellipses, we will need some geometric terminology. So first we have to define what an ellipse is:[27] [28]

You can define an ellipse in terms of a semimajor (usually denoted by a) and a semiminor axis (usually denoted by b). An ellipse is then defined by:

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1 \tag{9.35}$$

Please note that if a and b are 1, then the equation has a circle as a result. If you use this definition, you can think of an ellipse as a circle which has been horizontally multiplied with a and vertically with b.[29] [30]

The eccentricity e of an ellipse is defined as:

$$e = \sqrt{1 - \frac{b^2}{a^2}} \tag{9.36}$$

How higher the eccentricity of an orbit how more elliptical the orbit is. The observant reader has already noticed that for a circle its eccentricity is 0. For an ellipse it is between 0 and 1.[31] [32]

This was all the necessary mathematics for this PWS, we will move on to physics and astronomy.

---

[27] (Heckman, 2015)
[28] (Bouwens et al., 2013)
[29] (Heckman, 2015)
[30] (Bouwens et al., 2013)
[31] (Bouwens et al., 2013)
[32] (Heckman, 2015)

Chapter 10

Physical Introduction

10.1 Newtonian Mechanics

10.1.1 Gravity

In Newtonian mechanics, the magnitude of the force of gravity is defined by:[1] [2]

$$F_g = G\frac{m_1 m_2}{r^2} \tag{10.1}$$

In this formula F_g is the force of gravity that two objects exert upon each other (in newton), G is the constant of Cavendish[3], m_1 and m_2 are the masses of the objects (in kilograms) and r is the distance between them. The result is F_g, which is a scalar, and not a vector. Because the computer model works with vectors, we want to make a vector which is the gravitational force.[4] [5]

Call $\mathbf{r}_1$ the position vector of mass one and the same goes for $\mathbf{r}_2$. Then the direction of the vector of the exerted gravitational force on mass one is equal to $\mathbf{r}_2 - \mathbf{r}_1$. And for mass two it is $\mathbf{r}_1 - \mathbf{r}_2$. Then the gravity vector is:

$$\mathbf{F}_{1on2} = G\frac{m_1 m_2}{(\|\mathbf{r}_1 - \mathbf{r}_2\|)^2} \cdot \frac{\mathbf{r}_1 - \mathbf{r}_2}{\|\mathbf{r}_1 - \mathbf{r}_2\|} = G\frac{m_1 m_2}{(\|\mathbf{r}_1 - \mathbf{r}_2\|)^3} \cdot (\mathbf{r}_1 - \mathbf{r}_2) \tag{10.2}$$

$\frac{\mathbf{r}_1 - \mathbf{r}_2}{\|\mathbf{r}_1 - \mathbf{r}_2\|}$ denotes a vector with a length of one because it has been divided by its length, the direction of this vector is the direction in which gravity is pulling on the object (as mentioned before), in this case a planet. $G\frac{m_1 m_2}{(\|\mathbf{r}_1 - \mathbf{r}_2\|)^2}$ denotes the magnitude of the gravitational force because nothing has really changed compared with the formula of Newton's law of gravity mentioned earlier. So the result is a vector in the direction in which the gravity works on the object multiplied by its magnitude/size, so that is indeed the gravitational vector. Because of Newton's famous third law of motion, which states:[6] [7]

$$\mathbf{F}_{1on2} = -\mathbf{F}_{2on1} \tag{10.3}$$

---

[1] (de Pater and Lissauer, 2016)

[2] (Serway and Jewett Jr., 2014)

[3] Cavendish' constant appears in Newton's law of universal gravitation and Einstein's theory of general relativity. It is approximately $6,67 \cdot 10^{-11} \text{Nm}^2/\text{kg}^2$.

[4] (Serway and Jewett Jr., 2014)

[5] (Bouwens et al., 2013)

[6] (Serway and Jewett Jr., 2014)

[7] (de Pater and Lissauer, 2016)

We can now conclude that:

$$G\frac{m_1 m_2}{(\|\mathbf{r}_1 - \mathbf{r}_2\|)^3} \cdot (\mathbf{r}_1 - \mathbf{r}_2) = -G\frac{m_1 m_2}{(\|\mathbf{r}_2 - \mathbf{r}_1\|)^3} \cdot (\mathbf{r}_2 - \mathbf{r}_1) \tag{10.4}$$

Note that $\|\mathbf{r}_1 - \mathbf{r}_2\|$ and $\|\mathbf{r}_2 - \mathbf{r}_1\|$ are the same except that the vectors are pointing in opposite direction (so the length is the same). This is true because of Newton's third law and because of that the last equation follows.

And so it follows that:

$$\mathbf{F}_{2on1} = G\frac{m_1 m_2}{(\|\mathbf{r}_1 - \mathbf{r}_2\|)^3} \cdot (\vec{r_2} - \vec{r_1}) \tag{10.5}$$

10.1.2 Momentum

First you can define a quantity called linear momentum ($\mathbf{p}$) which is equal to:[8] [9]

$$\mathbf{p} = m\mathbf{v} \tag{10.6}$$

With linear momementum, you can define another quantity called angular momentum ($\mathbf{L}$) which is defined as:

$$\mathbf{L} = \mathbf{r} \times \mathbf{p} = \mathbf{r} \times (m\mathbf{v}) = m(\mathbf{r} \times \mathbf{v}) \tag{10.7}$$

Recall that $\times$ denotes a vector product. If you differentiate the angular momentum vector with respect to time, you get (provided that the mass doesn't change with time):

$$\dot{\mathbf{L}} = m(\dot{\mathbf{r}} \times \mathbf{v}) + m(\mathbf{r} \times \dot{\mathbf{v}}) = m(\mathbf{v} \times \mathbf{v}) + m(\mathbf{r} \times \mathbf{a}) \tag{10.8}$$

For the proof that the product rule is also true for the vector product, we would like to direct you to (Heckman, 2015). So if you simply the previous equation, you get:

$$\dot{\mathbf{L}} = m(\mathbf{r} \times \mathbf{a}) = \mathbf{r} \times (m\mathbf{a}) = \mathbf{r} \times \mathbf{F} \tag{10.9}$$

The quantity $\mathbf{r} \times \mathbf{F}$ is called torque and it is denoted by the letter τ. If $\mathbf{F}$ and $\mathbf{r}$ are proportional, the torque is zero which means that angular momentum is conserved.

---

[8](Serway and Jewett Jr., 2014)
[9](Goldstein et al., 2000)

Chapter 11

Astronomical Introduction

> The stars that decorate the sky, though we rightly regard them as the finest and most perfect of visible things, are far inferior, just because they are visible, to the true realities;
>
> Plato

11.1 Astronomical Units

On the scale of the solar system, one unit has proven to be particularly handy, namely the astronomical unit, which is nearly always shortened to AU. The AU is defined as the average distance between the sun and the earth during its object (which is around 150 million kilometres). The Kuiper belt stretches from 30 to 50 AU from the sun and the Oort cloud is beyond that.[1]

11.2 Planetary Orbits and Orbital Elements

11.2.1 The N-body problem

The problem we are focussing on in the entire PWS is called the N-body problem. The N-body problem is the question how N bodies (which in our simulator are modelled as points) interact under a certain force (in the case of the solar system, that force is gravity). The most simple of these problem is the two-body problem (which concerns the motion of two bodies), which can be solved entirely analtyically. But more difficult however, is the three-body problem. A general three-body problem cannot be solved analytically, because there are too many degrees of freedom. But, if you make certain restrictions an analytical solution can be obtained. An example of this is the circular restricted three-body problem. In this situation there are two (comparatively) massive bodies moving in circular orbits around their common centre of mass and the third one has negligible mass compared to the other two (this third body is commonly referred to as the test particle). Without restrictions such as these, little progress can be made without numerical methods and simulations.[2]

The case of planet nine is not a circular restricted three-body problem, because planet nine is in the Oort cloud and most likely has a highly elliptical orbit (and therefore not circular). So we can't apply things like Lagrange points or the Jacobi constant to calculate or restrict

---

[1](Kutner, 2003)
[2](de Pater and Lissauer, 2016)

the position of planet nine (to find out more about these subjects please go to (de Pater and Lissauer, 2016)).

Finally, there exists the N-body problem, which concerns the interaction of N bodies. You need a lot of restrictions to solve an N-body problem. Fortunately, the case of planet nine can be modelled as three-body problem, because the distance between the Oort cloud (where planet nine most likely resides and where the perturbed objects are) and the known planets is so big that the gravitational interaction between them is very small. We have to model the situation using secular perturbation theory, because the interaction between the perturbed objects and the planet nine is very important.

11.2.2   Argument of perihelion

Since we are now talking about orbits, it is important to understand how you should define an orbit. An orbit is defined by six scalar quantities, which are called orbital elements. We will discuss the most common set of six orbital element, but first we will introduce some new terminology.

The perihelion in an orbit around the sun is the point where the orbiting object is closest to the sun. The opposite is the aphelion, the point where the object is at the greatest distance from the sun. Please note that the terms perihelion and aphelion are for sun-centered orbits. In the case of earth-centered orbit one uses perigee and apogee. For orbits around other stars than our own you use periastron and apastron, for lunar orbits pericynthion and apocynthion. The general terms for this are pericenter and apocenter and periapsis and apsis. Now, the six quantites are:[3] [4] [5]

- The semimajor axis (a), the major axis of an ellipse is the greatest possible diameter of the ellipse. The semimajor axis is half the major axis.

- The eccentricity (e) defines 'how much stretched' the ellipse is.

- The inclination (i), in figure 11.2 you can see that the inclination is the angle between the orbital plane and the reference plane.

- The argument of perihelion (ω), this is the angle between the ascending node and the perihelion of the orbit.[8]

- The longitude of the ascending node (Ω) is the angle between the reference direction and the direction of the ascending node (where the planet passes its reference upwards).

- The true anomaly (f or ν) specifies the angle between a planet's perihelion and its instantaneous position.

---

[3](de Pater and Lissauer, 2016)
[4](Wikipedia, 2016i)
[5](Wikipedia, 2016a)
[7](Wikipedia, 2016b)
[8](Wikipedia, 2016i)
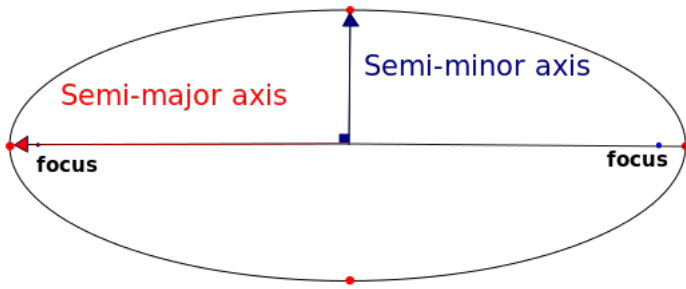
Figure 11.1: A two dimensional view of an orbit.[7]

a, e and i are often called the principal orbital elements because they determine size, shape and tilt of the orbit. Together with the other three they complete determine a planet's postion and orbit. One can also define a planetary orbit with the start position in a system and the start velocity. These two are vectors and are called orbital state vectors, so in the normal Cartesian Space these vectors are equal to six scalar quantities.[10]

In the model we will use the orbital state vectors as input instead of orbital elements, because we don't know which orbits the objects are going to move in, so we can't calculate the orbital elements. We do know their start positions and velocities. So we need to calculate the argument of perihelion with the orbital state vectors, how we are going to do that, will be explained now:[11]

Figure 11.2: The several orbital elements[9]

First you compute the orbital momentum vector, denoted by **h**, which is defined as:

$$\mathbf{h} = \mathbf{r} \times \dot{\mathbf{r}} \tag{11.1}$$

The observant reader has already noticed that:

$$\mathbf{h} = \frac{\mathbf{L}}{m} \tag{11.2}$$

You can then calculate the eccentricity vector **e**, which points from the aphelion to the perihelion of an orbit, with the following formula:

$$\mathbf{e} = \frac{\dot{\mathbf{r}} \times \mathbf{h}}{\mu} - \frac{\mathbf{r}}{\|\mathbf{r}\|} \tag{11.3}$$

---

[9](Wikipedia, 2016i)
[10](de Pater and Lissauer, 2016)
[11](Schwarz, 2016)

Here μ is the standard gravitational paramter, which has a value of $1,32712440018 \cdot 10^{20}$ ($\pm 8 \cdot 10^9$) m^3 s^{-2}. You will have to calculate the vector pointing to the ascending node $\mathbf{n}$ (whose z-component is zero) with:

$$\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \times \mathbf{h} = \begin{pmatrix} -h_y \\ h_x \\ 0 \end{pmatrix} \tag{11.4}$$

You can now calculate the argument of perihelion (or periapsis) with the following formula:[12]

$$\omega = \arccos\left(\frac{\langle \mathbf{n}, \mathbf{e} \rangle}{\|\mathbf{n}\|\|\mathbf{e}\|}\right) \tag{11.5}$$

We will now show the derivation:[13] [14] [15]

To calculate the argument of perihelion/periapsis you are in fact looking for the angle between the eccentricity vector and the vector pointing towards the ascending node. According to a theorem proven earlier in the mathematical introduction section, the following equation holds:

$$\langle \mathbf{n}, \mathbf{e} \rangle = \|\mathbf{n}\|\|\mathbf{e}\| \cos\omega \tag{11.6}$$

Because the argument of perihelion is in fact (as mentioned before) the angle between the two vectors, the only thing you have to do is solve this equation to find ω. So the following equations follow:

$$\cos\omega = \frac{\langle \mathbf{n}, \mathbf{e} \rangle}{\|\mathbf{n}\|\|\mathbf{e}\|} \tag{11.7}$$

You just use arccosine to get:

$$\arccos(\cos\omega) = \omega = \arccos\left(\frac{\langle \mathbf{n}, \mathbf{e} \rangle}{\|\mathbf{n}\|\|\mathbf{e}\|}\right) \tag{11.8}$$

This gives the already mentioned formula.

11.2.3 The Kozai Mechanism

The Kozai or Lidov-Kozai mechanism is a mechanism that changes the argument of perihelion of a small object under the influence of two much more massive objects. It was named after the Soviet scientist Michael Lidov and the Japanese astronomer Yoshihide Kozai. Under some approximations, there exists a conserved quantity, which is:[16] [17] [18]

$$\mathbf{L}_z = \sqrt{1 - e^2} \cos i \tag{11.9}$$

Here e is the eccentricity, i is the inclination and $\mathbf{L}_z$ is the z-component of the angular moment of the body which is perturbed. You can see that eccentricity and inclination can be exchanged for one another. Which means that high-inclination nearly circular orbits can become low-inclinitation orbits with great eccentricity. This happens periodically. For high

---

[12](Wikipedia, 2016i)
[13](Heckman, 2015)
[14](Wikipedia, 2016i)
[15](Wikipedia, 2016d)
[16](Trujillo and Sheppard, 2014)
[17](de Pater and Lissauer, 2016)
[18](Wikipedia, 2016h)

angles of inclination (i.e. $\cos^2 i < \frac{3}{5}$ which is approximately equal to 39,2 degrees) the mechanism ensures the argument of perihelion to remain constant. If the inclination is lower than that, the argument of perihelion librates (oscillates) around 90 or 270 degrees. Large periodic variations and exchanges in eccentricity and inclination are produced. The timescale associated with these exchanges is:[19] [20]

$$T_{Kozai} = 2\pi \frac{\sqrt{Gm} a_2^3}{GM a^{3/2}} (1 - e_2^2)^{3/2} = \frac{m P_2^2}{MP} (1 - e_2^2)^{3/2} \tag{11.10}$$

m is the mass of the perturbed object and M is the mass of the perturber. Quantities with a 2 as subjects are the properties of the perturber and otherwise, they are properties of the small object. The period of oscillation of all the three variables (ω, e, i) is the same.

---

[19](de Pater and Lissauer, 2016)
[20](Wikipedia, 2016h)

Part III

The Current Research on Planet Nine

Chapter 12

Current Research on the Subject

> On my return home, it occurred to me in 1837, that something might perhaps be made out on this question by patiently accumulating and reflecting on all sorts of facts which could possibly have any bearing on it.
>
> Charles Darwin

12.1   The Discovery of Sedna

In 2004 a new object was discovered by Michael Brown (the same scientist who later predicted the existence of planet nine), Chad Trujillo and David Rabinowitz in (Brown et al., 2004), the object (minor planet 90377) was named Sedna. At that time it was the most distant known object in the solar system. In this article they predicted its orbit with the discovery and the prediscovery images of the object from 2001, 2003 and 2004. The resulting orbit from the calculations has a semimajor axis of 480±40 AU, a perihelion of 76±4 AU and a high eccentricity (please see the mathematical and astronomical section for an explanation for these terms). The high eccentricity is very interesting because the then known objects in the Kuiper belt had nearly circular orbits. Already in 2004, they say in the article:[1]

"Such an orbit [Sedna's orbit] is unexpected in our current understanding of the solar system but could be the result of scattering by a yet-to-be-discovered planet, perturbation by an anomalously close stellar encounter, or formation of the solar system within a cluster of stars. In all of these case a significant additional population is likely present, and in the two most likely cases Sedna is best considered a member of the inner Oort Cloud, which then extends to much smaller semimajor axes than previously expected. Continued discovery and orbital characterization of objects in this inner Oort Cloud will verify the genesis of this unexpected population."[2]

The orbit of Sedna is unexpected because of the very high perihelion. At that point, the Kuiper Belt with the greatest perihelion had one at 46,6 AU. It was thought that the Oort cloud region started at much greater distance from the sun. They ask in the article how the object was scattered so far from the sun. They calculated the orbit in a very precise way by searching Sedna in data made prior to the discovery. They predicted by calculation in which position in the sky Sedna would have been in the previous years and search it in the data. In 2001, 2002 and 2003 they found Sedna and the chance that it was a randomly occurring event (such as a supernova)with the same brightness as Sedna in its position appears to be

---

[1](Brown et al., 2004)
[2](Brown et al., 2004)

44

in all the cases used for the calculation of the orbit less than 0.01 percent. They then fitted the orbit with techniques described in (Bernstein and Khushalani, 2000). After they had also used another algorithm for fitting the orbit, they also found approximate values for Sedna's eccentricity and inclination. They subsequently say that such an object must have been perturbed by unknown forces in the solar system or beyond.[3]

In the article, they discuss three explanations for Sedna's high perihelion:

- Scattering by planet nine (although they don't call the planet by that name yet). They concluded then already that Neptune couldn't be responsible for Sedna's high perihelion. They suggest that a planet at around 80 AU of 1 or 2 earth masses might do the trick, but they deem it unlikely.

- A single stellar encounter, talking about the possibility they say: "As an example, simple orbital integrations show that an encounter of a solar mass star moving at 30 km s^{-1} perpendicular to the ecliptic at a distance of 500 AU will perturb an orbit with a perihelion of 30 AU and semimajor axis of 480 AU to one with a perihelion of 76 AU, like that seen." It is important to note that such an encounter would massively affect our solar system and there would probably be many more scars visible to us.

- Formation in the sun's birth cluster. They conclude that several slow encounters can produce orbits like Sedna's. They then considered it to be the most likely scenario of the three. They also mention that the distribution of the orbital elements in the Oort cloud will say something about the size of the sun's birth cluster, if this hypothesis is true.

12.2 The Discovery of several more Oort cloud objects

12.2.1 The Discovery of 2012 VP$_{113}$

Sedna was the first discovered object in the inner Oort cloud, ten years later, in (Trujillo and Sheppard, 2014) a second one was reported, namely 2012 VP$_{113}$ (when you assume that it has a moderate albedo[4], it has possibly the size of a dwarf planet). Already then they recognized that Sedna and the new objects are not alone and hypothetically part of the greatest dynamical system in the solar system (when you measure it in terms of numbers of bodies). The perihelion of this object was 80 AU, which is roughly similar to Sedna's 76 AU. The authors suggest that the collection of these objects formed in circular orbits and were then perturbed in eccentric orbits. They define an inner Oort cloud object as an object whose orbit is not shaped by the known mass in the solar system (this typically means not shaped by the gas giants such as Neptune). They estimate a perihelion of such an object to be greater than 50 AU (which is beyond the range of a significant perturbartion by Neptune) and a semimajor axis between 150 and 1.500 AU, beyond 1.500 AU objects can be considered part of the outer Oort cloud. However, their formation is an entirely other matter because galactic tides start to become important in the process, some attention to them will be paid later on.[5]

In the article they introduce two main models for the formation of the inner Oort cloud objects. As mentioned before, the outer Oort cloud can have formed and sustained because

---

[3](Brown et al., 2004)

[4]Which is a reflection coefficient.

[5](Trujillo and Sheppard, 2014)

of galactic tides. However, according to them this cannot be true concerning the inner Oort cloud. The first one is that a large planet-sized object (such as planet nine), or several of them, perturbs objects in the Kuiper belt and the inner Oort cloud. The second one is that, close stellar encounters (when the solar system passed another star at a very small distance) can also have put the inner oort cloud objects in the current positions. It could even have happened within the first ten million years of the Sun's life, when it still resided in its birth cluster. That way it seems likely that one (or several) stellar encounters happened. They even suggest a third model that the inner oort cloud objects are captured planetisimals from other stars. They conclude to say that is as more inner Oort cloud objects are discovered, more limits are put on the different formation models and we will eventually see which one seems to be the most likely scenario.[6]

In the research, Trujillo and Sheppard have checked whether observational biases would determine which inner Oort cloud objects we would discover first and if they can account for the properties of the objects we have really found. Three results followed from this analysis:[7]

Figure 12.1: The perihelia of around a thousand minor planets set out against their eccentricity. It is immediately clear that Sedna and 2012 VP$_{113}$ are outliers[9].
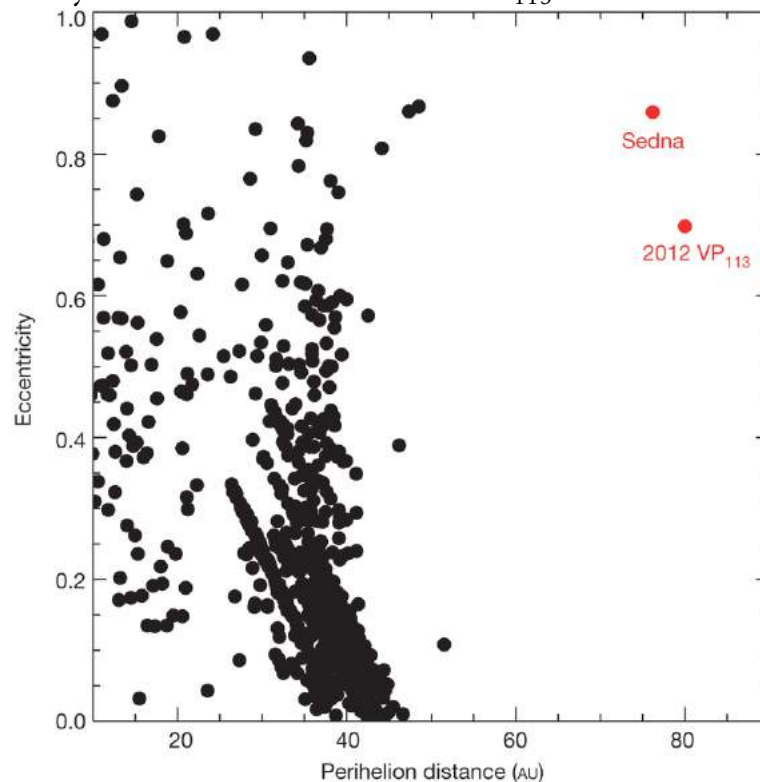
1. There appears to be very little objects with a perihelion between 50 to 75 AU. This suggests that the inner Oort cloud population has perihelia only greater than 75 AU. Please see figure 12.1 for this. It is strange that there are no objects discovered in the range from 50 (the end of the Kuiper belt) to 75 AU, these would obviously be brighter and consequently easier to detect than the inner Oort cloud objects we have already seen. The surveys of (Brown et al., 2004) and (Trujillo and Sheppard, 2014) were sensitive to objects from 50 AU to 300 AU, even then, they found no objects in the

---

[6](Trujillo and Sheppard, 2014)
[7](Trujillo and Sheppard, 2014)
[9](Trujillo and Sheppard, 2014)

range 50-75 AU. If there was a normal amount of objects in that range (i.e. if the innner Oort cloud followed one of the normal known small-body reservoir distibutions), they note that there would only be a 1% of finding Sedna and 2012 VP$_{113}$ and no objects with a perihelion smaller than 75 AU. But we must note already, that a chance of 1% is by no means scientifically certain, it could just have been coincidence or there can of course been other explanations for it than planet nine. They note however, that the inner Oort cloud objects may have increasing numbers when the distance increases. They say that some stellar encounter models which incorporate the capture of extrasolar planetesimals predict a strong inner edge to the perihelion distribution of these objects. They say that the model is consistent with their observations.[10]

2. The existance of 2012 VP$_{113}$ means that the semimajor axes of inner Oort cloud objects must stretch down to around 250 AU.

3. No observational bias can explain the clustering of the argument of perihelion. Sedna has an argument of perihelion of 311 degrees and 2012 VP$_{113}$ one of 293. Surprisingly this clustering is in all the known objects with semimajor axes larger than 150 AU and perihelia greater than Neptune's perihelion. The clustering can be seen in figure 17.2 (from the article): For most of these objects ω can be explained by resonant interactions with Neptune. But for several of these objects (such as Sedna and 2012 VP$_{113}$) that is not possible. They conclude that the ω-clustering is a real effect and not because of observational bias for two reasons:

   - If there were any bias for ω, it would be around 0 or 180 degrees because then the peihelion is at the heavily observed eccliptic (which is the eccliptical plane observed from earth). But the observed argument of perihelia is not around either 0 or 180 degrees.

   - The surveys that found the two objects were unbiased to 0 or 180 degrees because they were off-eccliptic or all-sky surveys.

Then they mention that the Lidov-Kozai effect is the best known dynamical mechanism for constraining the ω of a minor body. Like mentioned before, this is a three-body interaction that could have created this clustering in the Sun's early days. But it cannot explain the clustering today, because ω circulates due to the presence of the giant gas planets. They have already built a simulation that simulates the effect of the known bodies (especially the giant planets of course) on the inner Oort cloud objects (which we are also going to do, to check if we are going to find the same result) and they found that the inneer Oort cloud objects should have random ω. They conclude to say that a "massive perturber" (or a planet nine) may exist and restrict ω. They then simulated the effect a perturber of multiple earth masses on ω and found that it remains restriced for several billions of years. They note that their configuration (a planet nine at 250 AU) is not the unique possibility and remark that a perturber of this size at that distance with a low albedo would be fainter than all our detection limits.[11] [12]

They estimate the total mass of the inner Oort cloud to be around 1/80th of an Earth mass, the prediction of planet nine's mass of 4 to 5 Earth masses seems to be very strange in this context.

[10](Trujillo and Sheppard, 2014)
[11](Trujillo and Sheppard, 2014)
[12](de Pater and Lissauer, 2016)
[14](Trujillo and Sheppard, 2014)

Figure 12.2: The arguments of perihelia of distant objects plotted against their semimajor axis. It is clear that for distant objects ω is clustered[14].
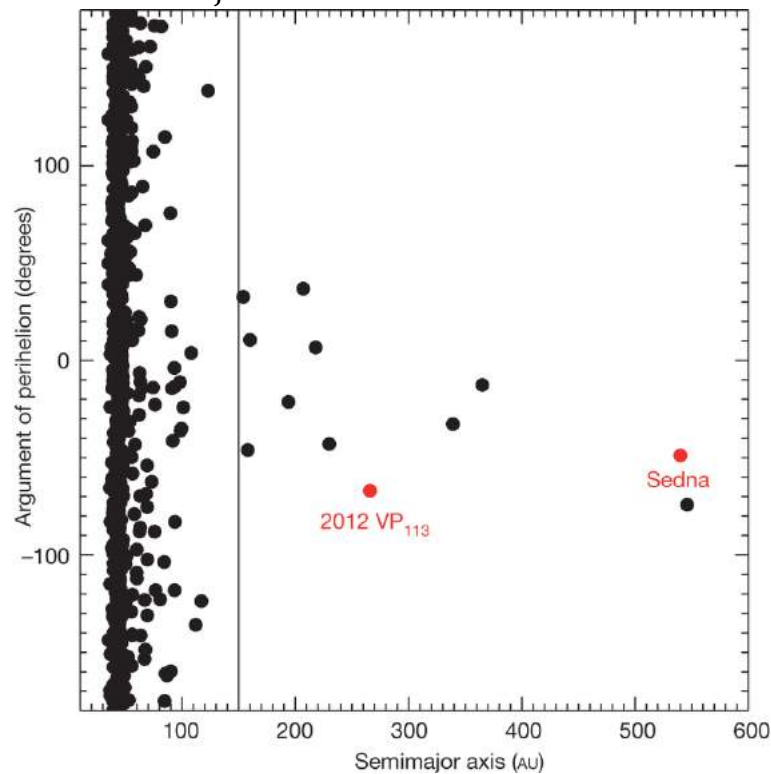
12.2.2 The response

Freelance astronomers and brothers Carlos and Raúl de la Fuente Marcos responoded in a new article: (de la Fuente Marcos and de la Fuente Marcos, 2014). In that article they confirmed the clustering in ω by using monte carlo techniques (which will we not go into in this thesis) and say that it is not a statistical coincidence. They say that the few known objects signal a very large population. They also mention (as one of the first) that there are likely at least two perturbers.

They find, like mentioned before a clustering of $\omega \approx 20°$ but what's new is that they say that for these objects also $i \approx 20°$.[15]

12.3 The article itself

At the time of the article, there were six known objects in the inner Oort Cloud. And they all had a clustering in one particular orbital feature, namely the argument of perihelion. The six objects are:

- Sedna
- 2012 VP$_{113}$
- 2004 VN$_{112}$
- 2007 TG$_{422}$

---

[15](de la Fuente Marcos and de la Fuente Marcos, 2014)

- 2013 RF$_{98}$

- 2010 GB$_{174}$

(de la Fuente Marcos and de la Fuente Marcos, 2014) had also mentioned a clustering in inclination, but now (Batygin and Brown, 2016b) showed that the six objects are not only clustered in ω, but also in physical space. Their perihelion positions and orbital planes are confined closely. They report that there is a probability of only 0,007% that this is due to chance (this is getting closer to the scientif certainty of 5σ). They say that a planet nine of more than ten Earth masses with an orbital plane which approximately the same of the six ETNO's can cause the observed orbital alignment.

They explain that the clustering around $\omega = 0°$ is expectable because the ecliptic is being observed heavily. However, then a clustering around $\omega = 180°$ would also be expected. Remarkably, that clustering is absent. They then say that no observational bias can explain the clustering around $\omega = 0°$ which is suprising because the gravitational forces exerted by the giant planets are expected to randomize ω. They predicted that a planet nine with a certain set of orbital elements allows the existence of an extra population of ETNO's that do not have this orbital clustering. Time will tell if their hypothesis is true or not.[16]

12.3.1 The Response

So far, the response to (Batygin and Brown, 2016b) has been tremendous. The number of follow-up studies is almost reaching 30, according to Konstantin Batygin. We will discuss some of these articles here, but to discuss them all in detail is far beyond the scope of this thesis.

First, a graduate student from Batygin published an article on solar obliquity (the obliquity is the angle between a body's spinning axis and its orbital plane). For a long time the sun's six degree solar obliquity has been a mystery. Now it has been suggested that its cause could have been planet nine during the formation of the solar system. It was shown analytically in the article that an object the way planet nine is generally thought to be can in some conditions create almost exactly the observed properties (including the sun's polar alignment).[17] [18]

Also the already mentioned De la Fuente Marcos brothers also published some new articles on the subject. In (de la Fuente Marcos and de la Fuente Marcos, 2016) they did some of the numerical calculations of (Batygin and Brown, 2016b) themselves with a focus on the perihelion and pole positions of the ETNO's. They found that planet nine is likely to be at aphelion (which seems logical because otherwise an object of that size would be 'too easily' visible from earth and would have long been detected). They also say that there are likely to be at least two massive perturbers in the Oort cloud (if not more). They also collaborated with Sverre Aarseth (an astronomer from Cambridge) in (de la Fuente Marcos et al., 2016) where they have determined likely sets of orbital elements of planet nine using N-body calculations (which are Mr. Aarseth's expertise). Mr. Brown and Mr. Batygin too have established likely sets of orbital elements in (Batygin and Brown, 2016a).[19]

---

[16](Batygin and Brown, 2016b)
[17](Bailey et al., 2016)
[18](de Pater and Lissauer, 2016)
[19](de la Fuente Marcos and de la Fuente Marcos, 2016)

Part IV

The Computer Model

Chapter 13

The Goal of the Computer Model

> There is nothing impossible to him who will try.
>
> ———————————————————
>
> Alexander the Great

Originally, the Computer Model was planned to have two parts (also mentioned in the 'Methods' part): a simulator (with seven input parameters) and a program tasked with guessing these seven parameters (the mass, position and speed of planet nine) by means of statistical analysis (monte carlo markov chains).

Nearing the end of our time window for handing in this thesis - after just having finished the simulator - we spoke to prof. Portegies Zwart again (to ask him about how we should have implemented the markov chains) and he told us to change plans. He also made some important recommendations for changing the simulator and suggested some experiments to answer the questions we asked in the Introduction.

13.1   The Original Plans

The original plan was to determine the position of planet nine, and also predict its motion. Therefore we would have had to determine several parameters. This would be done using a monte carlo algoritm, in which the values which produce the best result in simulation would be chosen. We would use a monte carlo markov chain algoritm to prevent getting stuck in a local minimum.

The position of a celestial body is completely defined by the object's start velocity, starting position and its mass. We now have a total of seven parameters for the monte carlo algoritm to 'estimate'. The algoritm will ultimately produce the most likely values for these parameters. We will then (like mentioned before) determine whether the existence of a planet nine with these values is very likely to exist. If the answer to this question is yes, we will then have to determine where in the sky the planet must be and calculate the magnitude of the object so that possibly telescopes could go looking for it.

One can also formulate it differently: The model is numerically looking for the planet which together with the sun has the six inner Oort cloud objects as lagrange points. The model could also find orbital resonances (but you find more on these topics in the astronomical introduction).

13.2 The Flaws with our Implementation

The basis of our simulator worked great, but it had some small flaws (as does any program). When looking at the original results for the argument of periapsis for Sedna and 2012 VP_{113}, some strange atifacts appeared.

Figure 13.1: The argument of periapsis of Sedna over time in radians (1 million years)[1]
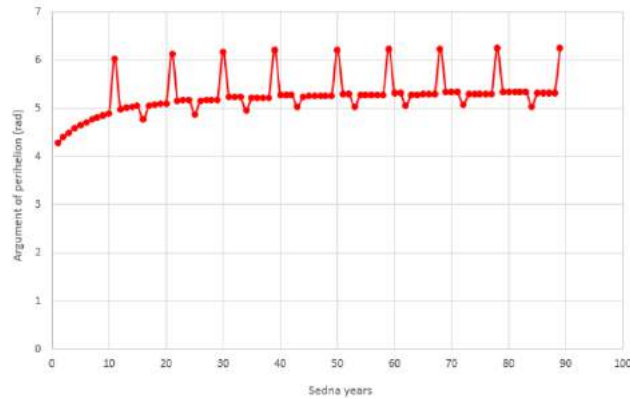
Figure 13.2: The argument of periapsis of 2012 VP_{113} over time in radians (1 million years)[1]
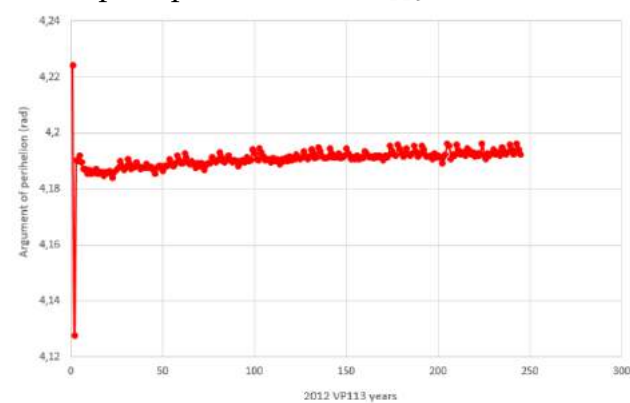
As can be seen in figures 13.1 and 13.2, the argument of periapsis of Sedna had strange fluctuations with strange points near 6 radians, instead of near 5 radians as the rest of the points were. The argument also has a strange wave like form at the beginning before it stabilises after a few orbits. This form is noticable in both Sedna and 2012 VP_{113}, which seems to suggest problems with the simulator.

Also, a sine curve seems to appear in the argument of 2012 VP_{113}. This suggests that there is a problem with the simulator itself, instead of a glitch.

---

[1]The original runtime was 1 million years, but these graphs don't represent those full 1 million years, because some years are missing, and therefore - incorrectly - hidden. This has been fixed in later graphs, but these were the originals we presented to Prof. Protegies Zwart.

13.3 Changes Recommended by Prof. Portegies Zwart

The technical changes required by the changes recommended by Prof. Portegies Zwart, will not be discussed in this section. It will be described in more detail in the chapter about the Technical Side of the Computer model, however.

Prof. Portegies Zwart proposed a rather simple solution: instead of the way we calculated the orbital elements by tracking when a full rotation has been reached and then calculating the nodes to calculate the argument of periapsis, we could also transform the position and speed vectors into the orbital elements directly. The way this can be done is described in more detail in the section about the Orbital Elements in the Astronomical Introduction.

This way of calculating is less error prone, because it involves a lot less calculations.

Chapter 14

Numerical and Scientific Computing

> The power of a consequence is determined by the power of its cause, because its being is explained or defined by the being of the cause.
>
> Spinoza

When you make a model of a physical system such as the solar system, you are essentialy numerically solving differential equations. Now there are several types of integration possible, such as the Euler integration, which is used in the program Coach 6 which will be discussed later. Thanks to professors Portegies Zwart and Icke we found out that this kind of integration is flawed for our purpose and we found out that leapfrog integration works much better, we will now pay some attention to the different numerical methods and their algorithms because they form the basis of our computer model:

14.1   Euler Integration

First we will discuss the Euler integration. Any Euler integration can be brought down to the following formulas (if you are using h as size of the timestep):[1] [2]

$$y_{n+1} = y_n + hf(x,y) \tag{14.1}$$

$$x_{n+1} = x_n + h \tag{14.2}$$

Here the n denotes the timestep number. These formulas may seem very abstract to some people, but we will now explain them in the context of our thesis. Let $\mathbf{r}_n$ be the position vector at time step n, $\mathbf{r}_n$ has the function of y_n from the above example. Let t_n be the time at timestep n and fulfil the function of x_n. Now let $f(\mathbf{r},t)$ be the function that calculates the velocity at time t and position $\mathbf{r}$. Then the equations would become (using timestep Δt):

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \Delta t \cdot f(\mathbf{r},t) \tag{14.3}$$

$$t = t + \Delta t \tag{14.4}$$

These equations make sense if you want to calculate the position of an object over time. So if we were to determine the displacement of an object and we were to use Euler integration, we wouldn't just use a random function f, our equations would be like (assuming that the mass doesn't change over time):

$$\mathbf{a}_{n+1} = \frac{\mathbf{F}_{n+1}}{m} \tag{14.5}$$

---

[1](Adams and Essex, 2013)
[2](van der Laan, 2015)

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{a}_{n+1} \cdot \Delta t \tag{14.6}$$

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \mathbf{v}_{n+1} \cdot \Delta t \tag{14.7}$$

The function f that calculates velocity will then be:

$$f(\mathbf{r}, t) = \mathbf{v}_{n+1} = \mathbf{v}_n + \frac{\mathbf{F}_{n+1}}{m} \cdot \Delta t \tag{14.8}$$

14.2 Leapfrog Integration

Now the problem is that this method gives a large deviation. Another numerical method was suggested to us, namely the leapfrog integration. This type of integration is suitable for solving differential equations of the form:[3]

$$\mathbf{F}(\mathbf{r}, \dot{\mathbf{r}}) = m\frac{d^2\mathbf{r}}{dt^2} \tag{14.9}$$

Which is Newton's famous second law of motion, one can conclude from this that this integrator is particularly handy for simulating dynamical systems.

If we were to use the leapfrog integration, we would use the following equations (i denotes timestep i):

$$\mathbf{r}_i = \mathbf{r}_{i-1} + \mathbf{v}_{i-\frac{1}{2}} \cdot dt \tag{14.10}$$

$$\mathbf{a}_i = \frac{\mathbf{F}(\mathbf{r}_i)}{m} \tag{14.11}$$

$$\mathbf{v}_{i+\frac{1}{2}} = \mathbf{v}_{i-\frac{1}{2}} + \mathbf{a}_i dt \tag{14.12}$$

This is why it is called leapfrog integration, because the positions and velocities 'leap frog' past one another. There is another algorithm that is worthwile to discuss:

14.3 The Runge-Kutta method

This is a much more precise numerical method, if you write it down, and you again use timestep h):[4]

$$p_n = f(x_n, y_n) \tag{14.13}$$

$$q_n = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}p_n) \tag{14.14}$$

$$r_n = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}q_n) \tag{14.15}$$

$$s_n = f(x_n + h, y_n + hr_n) \tag{14.16}$$

$$y_{n+1} = y_n + h\frac{p_n + 2q_n + 2r_n + s_n}{6} \tag{14.17}$$

$$x_{n+1} = x_n + h \tag{14.18}$$

The reader can already see the vast amount of calculations that must be done each timestep, which makes the computer model considerably slower. This is the reason why we didn't implement this method instead of the leapfrog, despite its greater accuracy.

---

[3](van der Laan, 2015)
[4](Adams and Essex, 2013)

Chapter 15

The Basis of the Computer model

> I have not failed. I've just found 10.000
> ways that won't work.
>
> —— Thomas Edison

In this chapter we are going to define what the model we have used to find answer to our research questions does in mathematical and physical terms (and not in computer science terminology). Please read the next chapter for more information about the way the model is programmed.

15.1   Orbital state vectors and orbital elements

In the astronomical section has already been explained what orbital state vectors and orbital elements are. But one of the challenges of building the computer model is converting orbital state vectors into orbital elements. The model uses the NASA (North American Space Association) data of the objects simulated in the model, to be precise, the object's position and velocity vectors are put into the model and then the simulation starts. However, the real challenge is to calculate the argument of perihelion using the simulated orbits. The model has to find the ascending node and perihelium on its own. To make sure it works properly has been quite a challenge.

15.2   CoachTaal (Coach 6)

Originally, we intended to write the model like we do in school. In the physics class, we use a programme called Coach 6 (which is used nationwide in the physics curriculum in the Netherlands), the programming language used in that programme is called CoachTaal. The examples in the national physics exams make use of an Euler integration (which was explained in the previous chapter), so originally we wrote it with an Euler integration. The programme would (roughly) simulate the solar system like this:[1]

First you would calculate the gravitational force with the formula from the physics section, then:

$$a_x := \frac{F_x}{m} \tag{15.1}$$

This has of course has been derived from Newton's second law of motion $\mathbf{F} = m \cdot \mathbf{a}$, a_x is here the acceleration of the object in the x-component (in all physical quantities which are

---

[1](Dorenbos and Kedzierska, 2011)

57

used in this section where the x, y or z appears as a subscript, it means the quantity in that direction), the next step is:[2] [3]

$$dv_x := a_x \cdot dt \tag{15.2}$$

dt denotes an infinitesimal[4] amount of time and dv_x denotes a change in the speed of the planet, asteroid etc. Then:

$$v_x := v_x + dv_x \tag{15.3}$$

Which means that the speed of the object increases or decreases with dv_x. Then you can calculate the new position of the object with the following formulas:

$$dx := v_x \cdot dt \tag{15.4}$$

$$x := x + dx \tag{15.5}$$

And in the model, of course, the amount of time passed increases:

$$t := t + dt \tag{15.6}$$

More information about CoachTaal can be found on the CMA (the publisher) website or at http://www.cma-science.nl/downloads/nl/software/coach6/c6_3_handboek_coachtaal.pdf (only available in Dutch).

15.3   Matrix Calculations (related to the physical simulation)

Now we will discuss how the model calculates the force on the different objects. Like we have seen in the physics section you can calculate the gravitational vector on a specific object with the following formula:[5] [6]

$$\mathbf{F}_{1on2} = G \frac{m_1 m_2}{(\|\mathbf{r}_1 - \mathbf{r}_2\|)^3} \cdot (\mathbf{r}_1 - \mathbf{r}_2) \tag{15.7}$$

You can create a matrix A when you are using n objects in the model, position a_{ij} is defined as (we will show this matrix and the following big matrices in the relevant appendix because of the clarity of the explanations):

$$a_{ij} = \frac{1}{(\|\mathbf{r}_i - \mathbf{r}_j\|)^3} \cdot (\mathbf{r}_i - \mathbf{r}_j) \tag{15.8}$$

In which $\|\mathbf{r}_i - \mathbf{r}_j\|$ the distance between objects i and j (whichever objects are named i and j). $(\mathbf{r}_i - \mathbf{r}_j)$ is the vector in the direction of the gravitational force exerted on object j by object i (and the same goes for all the other combinations of objects). Note that a_{ii} is 0 for all i because a body obviously doesn't exert a gravitational force on itself. If you multiply this matrix with a certain matrix B (m of course denotes the mass of the certain body in kilograms):

$$B = \begin{pmatrix} m_1 & 0 & \dots & 0 \\ 0 & m_2 & \dots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \dots & 0 & m_n \end{pmatrix}$$

---

[2](Serway and Jewett Jr., 2014)

[3](Wikipedia, 2016e)

[4]An infinitely small number, but larger than zero. It is used in differential and integral calculus.

[5](H.van Gendt and Dames, 2008)

[6](van den Essen, 2015b)

The resulting product of B and A is then:

$$(BA)_{ij} = \frac{m_i}{(\|\mathbf{r}_i - \mathbf{r}_j\|)^3} \cdot (\mathbf{r}_i - \mathbf{r}_j) \tag{15.9}$$

Note again that a_{ii} is 0 for all i. Name this resulting matrix C, if you multiply C on the right with the same matrix B as used before the resulting matrix is:

$$(CB)_{ij} = (BAB)_{ij} = \frac{m_i m_j}{(\|\mathbf{r}_i - \mathbf{r}_j\|)^3} \cdot (\mathbf{r}_i - \mathbf{r}_j) \tag{15.10}$$

Finally you have to multiply the matrix with the constant G (mentioned before), so that the result is:

$$(G(BAB))_{ij} = G \cdot \frac{m_i m_j}{(\|\mathbf{r}_i - \mathbf{r}_j\|)^3} \cdot (\mathbf{r}_i - \mathbf{r}_j) \tag{15.11}$$

Some readers may ask if it's important to multiply the matrices like:

$$G(B(AB)) \tag{15.12}$$

Or in any other order. However, the position of G is not important due to theorem 9.1.1i), so it can be in any place and in theorem 9.1.1ii) it has been proved that matrix multiplication is associative so that the order in which the matrices are multiplied is not important (please see the "Mathematical Introduction" section for a more detailed explanation).

The observant reader has already noticed that this last matrix is equal to:

$$\begin{pmatrix} 0 & \mathbf{F}_{1on2} & \cdots & \mathbf{F}_{1on(n)} \\ \mathbf{F}_{2on1} & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \mathbf{F}_{(n-1)on(n)} \\ \mathbf{F}_{(n)on1} & \cdots & \mathbf{F}_{(n)on(n-1)} & 0 \end{pmatrix}$$

Again, the diagonal is obviously zero because a body doesn't exert a gravitational force on itself (on a celestial scale). The sum of the i-th column is the total gravitational force on the i-th object, so if you notate the resulting gravitational force on object i (like the computer will calculate it from the final matrix), it will be:

$$\mathbf{F}_{totalon(i)} = \sum_{i=1}^{n} a_{ij} \tag{15.13}$$

Here a_{ij} denotes the entry in the final matrix in the i-th row and the j-th column.

Chapter 16

The Technical Side of the Computer Model

> Nothing in life is to be feared, it is only
> to be understood. Now it is time to
> understand more, so that we may fear
> less.
>
> Marie Curie

In this section the technical side of the computer model will be covered. Our computer model is written in Java to ensure compatibility with all operating systems and to fit in well with our schools courses in computer science (Java supports Windows, OS X and most variants of Linux).

The program(s) are available under a MIT license at http://pws.christiaangoossens.nl and have been included fully in the Appendices.

As described, the original plan involved a model in two parts, but this was eventually reduced to only the simulator, and not the statistical part.

16.1   The Original Simulator

The simulator itself was the original first part. It calculates the orbits of (given) objects using leapfrog integration with Newtonian mechanics. The process of determening the position and speed vectors at different times with simulation is already described in CoachTaal in the last chapter, so we'll focus on the analysis that's done by the simulator after the orbit is simulated.

Because we want to calculate the argument of periapsis, we need the eccentricity vector and the ascending node. The argument of periapsis is calculated each rotation (because only after a full rotation the nodes are known). Therefore we have a function to detect the distance to the starting position.

When a full rotation is reached, the aphelion & perihelion, and the ascending & descending nodes are determined.

16.1.1   Rotation Check

In the original simulator, the orbital elements could only be calculated upon a full rotation, because the global minimum and maximum height have to be determined.

A rotation check is preformed by checking the difference to the final position of the last rotation or by checking the difference to the starting position of the object to the current position. If this round's distance is higher than last round's and the round before that

had an higher distance, the point with the least distance to the starting position has been reached.

When the least distance to the starting position is reached, they object has made a full rotation around the star.

The following code is used[1]:

```java
public boolean processRoundCheck() {
    double startDistance =
        this.thisObject.getDistance(this.startingPosition).length();
    boolean fullRotation = false;

    if (beforeLastStartDistance != -1 && lastStartDistance != -1) {
        if (beforeLastStartDistance > lastStartDistance && startDistance >
            lastStartDistance) {
            // Last point was the closest to the starting position overall!
            fullRotation = true;

            // REDACTED => Print some information to the console
        }

        beforeLastStartDistance = lastStartDistance;
        lastStartDistance = startDistance;
    }

        /**
         * REDACTED => Check if all the variables are not undefined and/or set them.
         */

    if(fullRotation) {
        return true;
    } else {
        return false;
    }
}
```

This function is called every simulator round (timestep).

16.1.2   Calculating the Maximum and Minimum of the z-axis graph

The following code is an excerpt from ObjectProcessor.java and describes the way the global minima and maxima are determined.

This function is also called every timestep.

```java
public void calculateTops() {
    if (this.absoluteMax == null || this.absoluteMax.empty()) {
        this.absoluteMax = new Node(this.thisObject.position);
        this.absoluteMax.setRound(Simulator.round);
```

---

[1]Please note this is not the full code but an excerpt. The full code can be found in ObjectProcessor.java.

```
    }

    if (this.absoluteMin == null || this.absoluteMin.empty()) {
        this.absoluteMin = new Node(this.thisObject.position);
        this.absoluteMin.setRound(Simulator.round);
    }

    if (this.thisObject.position.getZ() > this.absoluteMax.getZ()) {
        this.absoluteMax = new Node(this.thisObject.position);
        this.absoluteMax.setRound(Simulator.round);
    }

    if (this.thisObject.position.getZ() < this.absoluteMin.getZ()) {
        this.absoluteMin = new Node(this.thisObject.position);
        this.absoluteMin.setRound(Simulator.round);
    }
}
```

The code can be devided in two parts: the first part defines the maximum and minimum as the current position if they are not set yet and the second part compares the z component of the current position vector to the z component of the stored position vector.

After a full rotation, this function has set the highest position as the maximum, and the lowest position as the minimum and the vectors will be reset.

16.1.3   Determening the Positions of the Nodes

After a full rotation has been completed, the ascending and descending node can be calculated. All positions of the finished rotation are compared to find the positions closest to the average height between the maximum and minimum height.

```
private Node findNode(Node min, Node max) {
    this.referenceZ = (min.getZ() + max.getZ()) / 2;

    /**
     * REDACTED => A check to find undefined values or errors with calculation (for
         example if there would have been more tops to the graph than only the
         maximum and minimum in this round
     */

    for (Map.Entry<Integer, Vector3d[]> entry : this.history.entrySet()) {
        // REDACTED => Some definitions and a check related to the three tops issue
        if (vectorArray[0].getZ() < referenceZ && this.history.get(round +
            1)[0].getZ() > referenceZ) {
            returnNode = new Node(vectorArray[0]);
            returnNode.setRound(round);
        } else if (vectorArray[0].getZ() > referenceZ && this.history.get(round +
            1)[0].getZ() < referenceZ) {
            returnNode = new Node(vectorArray[0]);
            returnNode.setRound(round);
        }
```

```
        }

    return returnNode; // SIMPLIFIED RETURN, REDACTED empty check
}
```

Determening the nodes was quite hard, because in the NASA data[2] the planets don't tran-scent the ecliptic (at z = 0) every rotation, as they should do according to theory.

Figure 16.1: Exported NASA Data (CSV Export from the HORIZIONS Web-Interface) for the z component of Earth's position vector over a timespan of 50 years
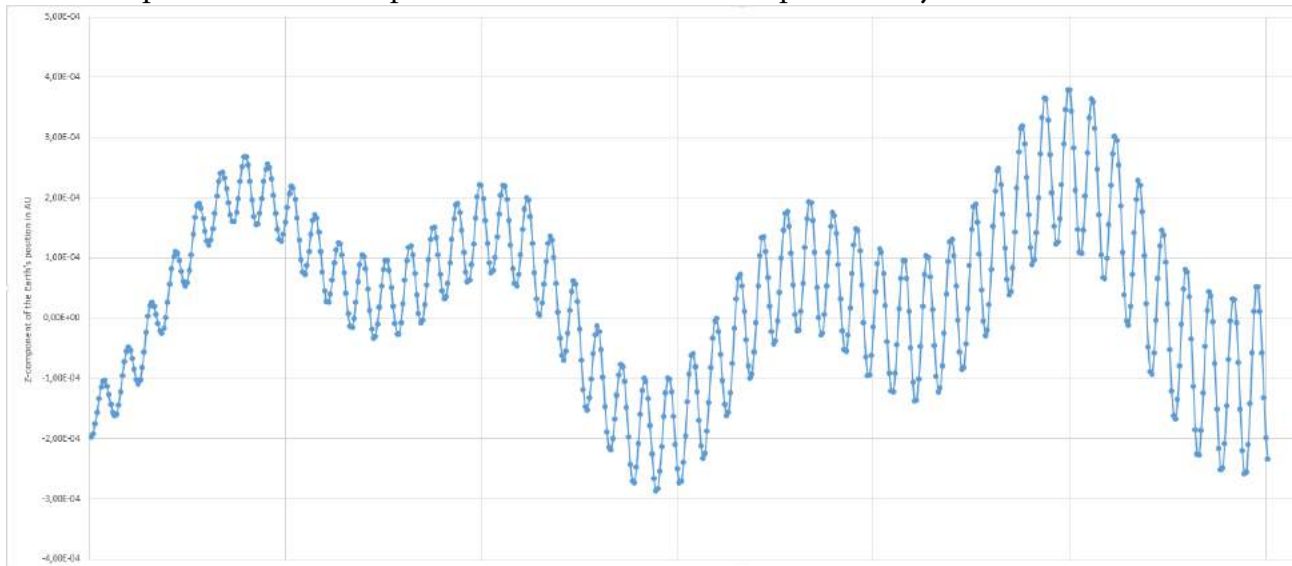
In figure 16.1 it's clear that the z component graph isn't a sin function around z = 0 as you would expect, but that the average height between two tops keeps changing over time.

There's also a problem with the seventh crest in this figure, because in that rotation both troughs (the one left and the one right of the crest) are within the rotation, while in all other rounds, there is only one trough and one crest. Therefore a check was put in place to remove those rounds from the argument calculation, because the results were incorrectly calculated. (The computer can't find both troughs because the graph is actually made up of very small sine waves instead of straight lines between the tops and the computer would therefore have to find all local maxima and minima (which would include the little (invisible) ones) or the global maxima and minima (which would only find the highest and lowest, but fail to find the one but lowest)).

16.1.4   Calculating the argument of periapsis

After determening the nodes, the argument of periapsis for this rotation can be calculated with the following formula (from the Mathematical Introduction):

$$\omega = \arccos\left(\frac{\langle \mathbf{n}, \mathbf{e} \rangle}{\|n\|\|e\|}\right) \qquad (16.1)$$

---

[2]We got our initial starting positions and speeds from the NASA HORIZIONS Web-Interface at http://ssd.jpl.nasa.gov/horizons.cgi

After the entire simulation has finished, the simulator outputs a list of all calculated arguments and a score, defined as the sum of the absolute differences between the arguments in radians (for example the following list of arguments would result in a score of 0.3: [0.45, 0.55, 0.35].

16.2   The Changed Simulator

As described in the chapter about the Goal of the Simulator, we changed the simulator to calculate the orbital elements directly from the position and speed at a certain point in time, instead of using the functions above to determine the ascending and descending node.

The formula used here are already described in the Argument of Perihelion section in the Astronomical Introduction.

First the orbital momentum vector is calculated, from which you can then calculate the eccentricity vector and derive the vector pointing to the ascending node. Then the normal formula for the argument of perihelion can be used (formula 16.1).

The above is translated into code as the following:

```
public static double calculate(Vector3d pos, Vector3d speed) {
    // ORBITAL MOMENTUM VECTOR
   Vector3d orbitalMomentum = new Vector3d(0,0,0);
   orbitalMomentum.cross(speed, pos);

    // ACCENDING NODE VECTOR
   Vector3d ascendingNode = new Vector3d(0,0,0);
   ascendingNode.cross(new Vector3d(0,0,1), orbitalMomentum);

   // ECCENTRICITY VECTOR
   double mu = 1.32712440018E20;

   Vector3d upCross = new Vector3d(0,0,0);
   upCross.cross(speed, orbitalMomentum);
   upCross.scale(1/mu);
   double posLength = pos.length();
   Vector3d rightPos = new Vector3d(pos);
   rightPos.scale(1/posLength);

   Vector3d eccentricity = new Vector3d(0,0,0);
   eccentricity.sub(upCross, rightPos);

   // AOP
   double aop;
   if (eccentricity.getZ() < 0) {
     aop = (2 * Math.PI) - ascendingNode.angle(eccentricity);
   } else {
     aop = ascendingNode.angle(eccentricity);
   }

    return aop;
}
```

16.3   Comparison between the Old and New Simulators

Figures 16.2, 16.3 and 16.4 are comparision graphs between the resulting data from the old and the new simulator. Both simulators were configured to use the following settings (and the vectors and masses as described in the Appendix about the Settings):

- Earth: amount of total rounds: 525948000 (approximately based on 500 Earth years) , timestep (per round): 30 seconds

- Sedna & 2012 VP$_{113}$: amount of total rounds: 184000000 (approximately based on 1 million Earth years) , timestep (per round): 172800 seconds (approximately based on 2 Earth days)

The settings for the old simulator differ from the graphs in the chapter about the flaws of the old simulator. Previously we used timesteps with a length of approximately one Earth month, but these steps were too large, which was also the cause of most of the problems with the old simulator.

After we changed the timestep to 2 Earth days, the objects (Sedna & 2012 VP$_{113}$) made the correct orbits around the Sun (instead of flying away with a timestep of a month).

The old simulator calculates the argument once every rotation (see the chapter about how the old simulator works), and therefore the same logic is applied to the new simulator (which can also calculate the argument during a rotation) to create a fair comparison.

Figure 16.2: The argument of periapsis of 2012 VP$_{113}$
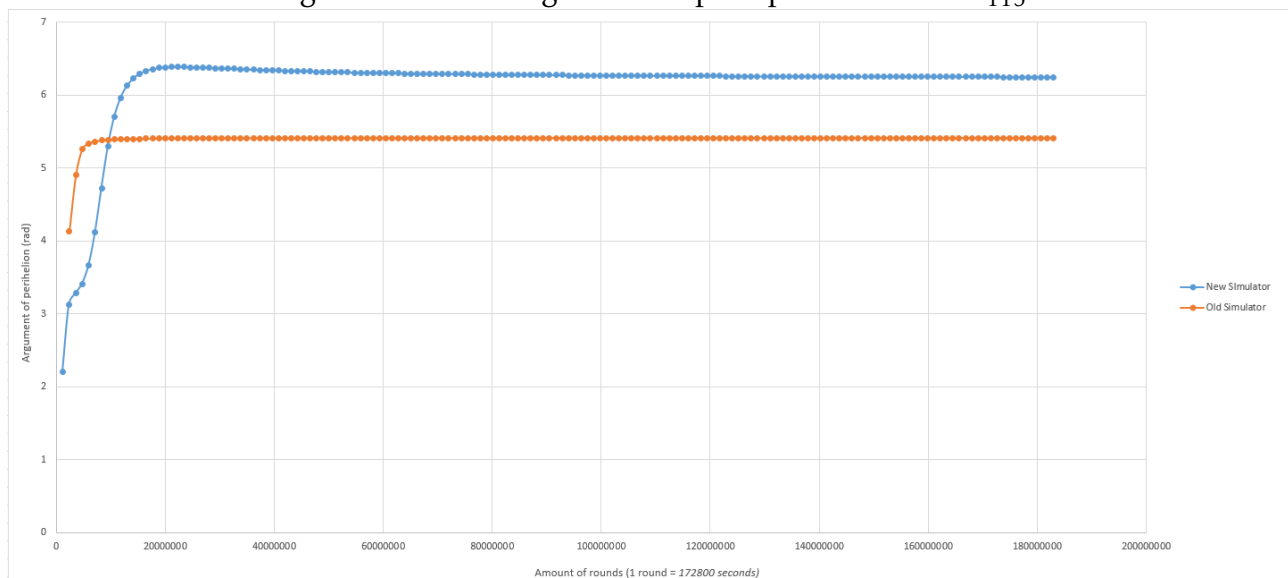
After reviewing figures 16.2, 16.3 and 16.4 a few things become clear:

- Most of the problems with the old simulator were caused by incorrect settings

- The strange 'warm-up' effect at the beginning of every graph is apparent in both the old simulator and the new simulator. It is therefore most likely caused by our integrator, instead of the argument calculation.

- The old simulator seems to be more accurate, based on the results for the argument of periapsis of the Earth, which is averaging around 5,03 radians with the old simu-

Figure 16.3: The argument of periapsis of Sedna
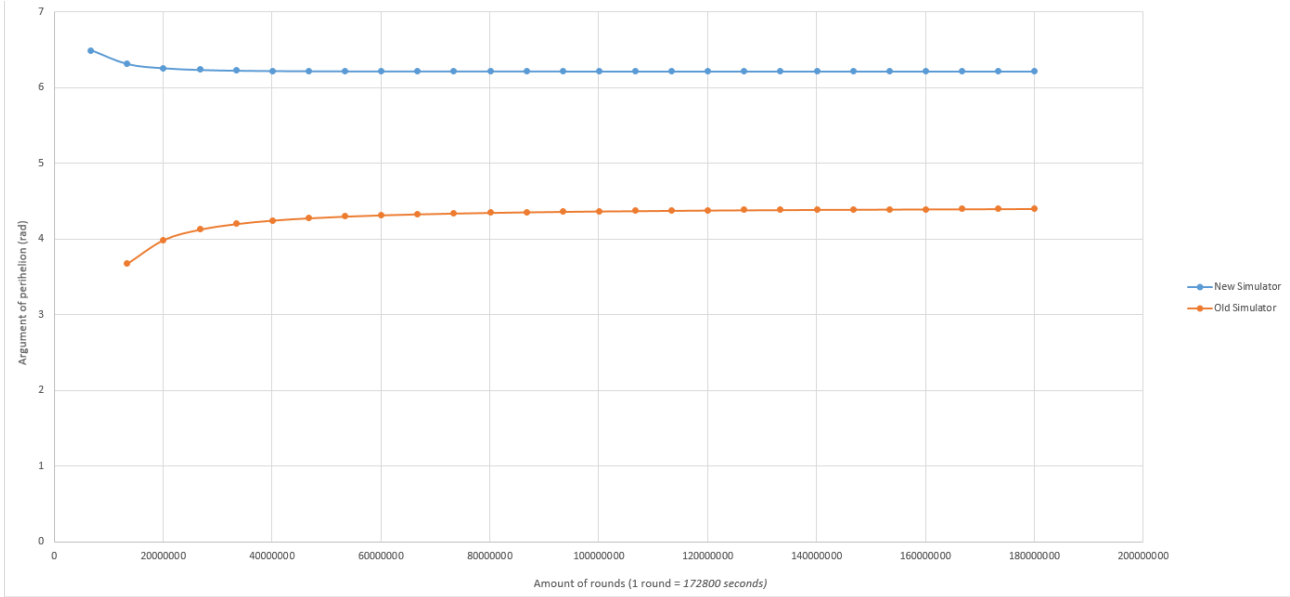
Figure 16.4: The argument of periapsis of the Earth
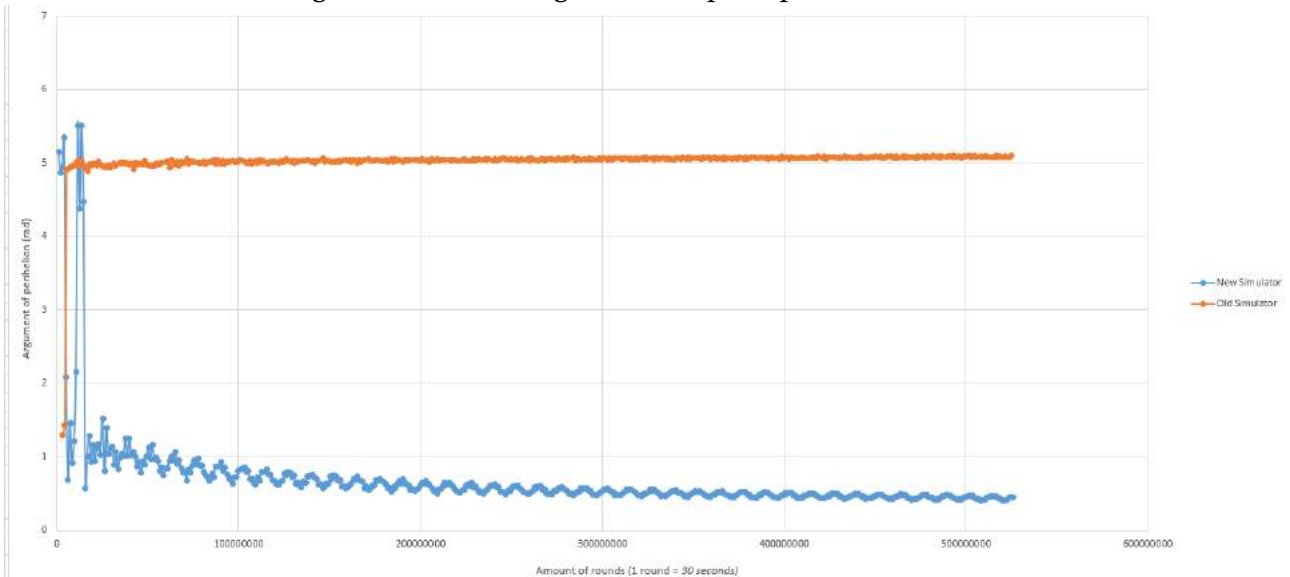
lator, and behaving like a sine curve near 0.5 radians with the new simulator.[3] The difference in exact value, however, shouldn't be taken as a measure of accuracy, as the argument of periapsis is only calculated at the end of the rotation - it can change during a rotation -, and the way of determening the end of the rotation is pretty inaccurate (but sufficient).

- (Nearly) all graphs result in a straight horizontal line, with the notable exception of the Earth graph with the new simulator.

Because the new simulator seems to have more graphs that exceed 6,283 (2 times pi) and therefore should return to values below 6,283 (because the angle is calculated modulo 2 pi),

---

[3](Simon et al., 1994) lists the longitude of perihelion of the Earth as 102,937°, which gives us a value of 288,064° or 5,027 radians for the argument of perihelion (by subtracting the node longitude of 174,873° and adding 360°)

which the graphs above were adjusted for[4], and the overall results are very similar[5], - except for the Earth graph which isn't a straight line as you would expect, but a sine wave - , all resulting graphs in the Results section will be generated with the old simulator.

---

[4]The original resulting graphs had a very steep drop from 6,283 to 0 somewhere in the graph, which made it difficult to see the resulting (near) straight line. Therefore the graphs have been adjusted to exceed 6,283 and have been limited at a maximum value of 7 instead

[5]The exact values also aren't the subject of this article. We only look at the bigger picture, for instance if the graph is centered around a higher value than the last graph

Part V

Our Own Results

Chapter 17

Results

17.1   How was planet nine predicted?

When the first Oort cloud object, Sedna, was discovered in 2004. Already then it was noted that Sedna exhibited unlikely characteristics, namely a very high perihelion. Then, a stellar encounter was deemed to be the most likely cause of this anomaly.[1] Then in 2012, with the discovery of a second object, 2012 VP_{113}, an orbital clustering was found in ω. It was also noted that (since Sedna has a perihelion of 76 AU and 2012 VP_{113} one of 80 AU) there seems to be a strange absence of objects in the 50-75 AU range.[2] It was reported later on there is also a clustering in i and that the Kozai mechanism might be responsible for the clustering in ω. (Batygin and Brown, 2016b) noted a clustering in orbital plane and perihelion positions. They calculated that there is only a chance 0,007% that the clustering is a coincidence.[3]

Now we know planet nine was predicted because it can explain the unexpected clustering in the argument of perihelion, inclination, orbital plane and perihelion position in near Oort cloud objects. There are now six known objects in the inner Oort cloud and they all have roughly the same argument of perihelion. It has been proposed that these objects have been perturbed by a distant giant planet through the Kozai mechanism. This mechanism causes smaller objects to oscillate around a certain argument of perihelion, eccentricity and inclination under the influence of a perturber. This was suggested by (Trujillo and Sheppard, 2014) and (Batygin and Brown, 2016b). However (Trujillo and Sheppard, 2014) lists it as one of the solutions. They also say that this clustering cannot be explained by either observational bias and it is unlikely that the clustering is due to chance.[4] [5] [6] [7]

---

[1](Brown et al., 2004)
[2](Trujillo and Sheppard, 2014)
[3](de la Fuente Marcos and de la Fuente Marcos, 2014)
[4](de la Fuente Marcos and de la Fuente Marcos, 2014)
[5](Batygin and Brown, 2016b)
[6](Trujillo and Sheppard, 2014)
[7](de la Fuente Marcos et al., 2016)

17.2 Is its existence really needed to explain the present situation of our solar system?

Figure 17.1: The argument of periapsis of 2012 VP$_{113}$ as calculated by the simulator over more than 1 million years[9].
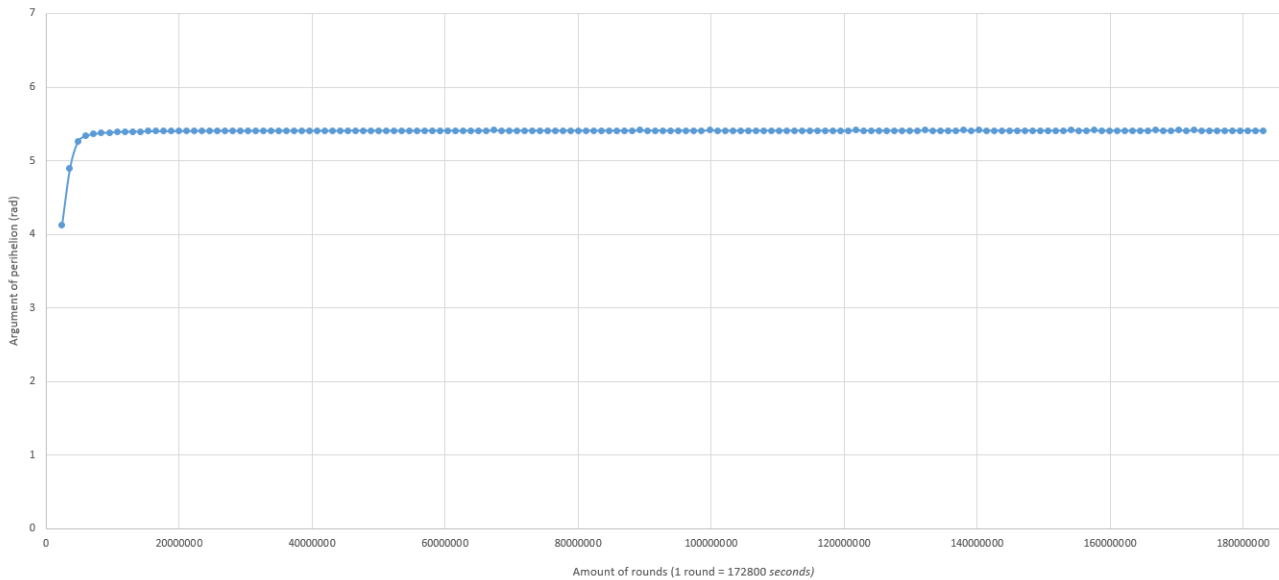
Figure 17.2: The argument of periapsis of Sedna as calculated by the simulator over more than 1 million years[11].
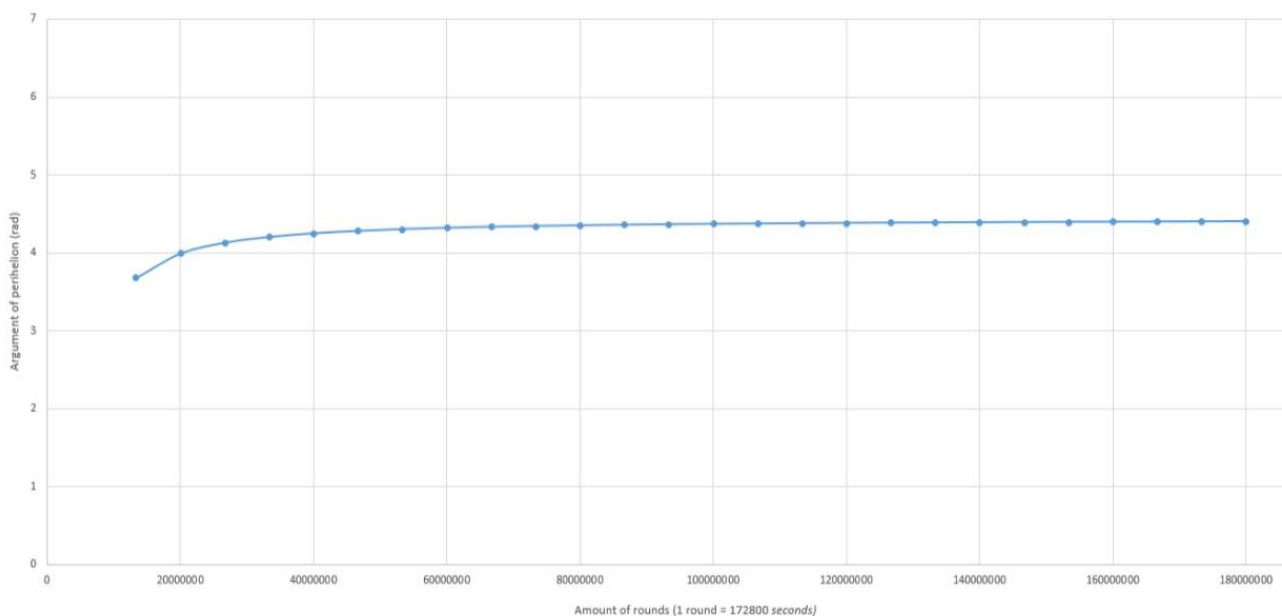
If you look at following graphs, you see that at the start (in the first few orbits) there is a (relatively) big increase in argument of perihelion. This is probably due to some startup flaw

---

[9] The simulation included the Sun, Jupiter, Saturn, Neptune, Uranus and 2012 VP$_{113}$, with a timestep of 172800 seconds. More information about the settings can be found in the Appendix.

[11] The simulation included the Sun, Jupiter, Saturn, Neptune, Uranus and Sedna, with a timestep of 172800 seconds. More information about the settings can be found in the Appendix.

in the simulator. But after that, you can see that in both cases the argument of perihelion slowly but surely increases, which was what we expected. But it is logical that the argument of the perihelion of the objects concerned is not in- or decreasing at the same speed which means that there will be difference and the clustering around the argument of perihelion. One can conclude that the observed clustering is caused by something special and planet nine is indeed a possible candidate.

17.2.1   More objects

While 2012 VP_{113} and Sedna are the only mentioned objects with a known mass, we also conducted tests on the other mentioned objects using the mass of 2012 VP_{113}. An overview of these tests can be seen in figure 17.3.

Figure 17.3: The argument of periapsis of the known ETNOs as calculated by the simulator over more than 1 million years
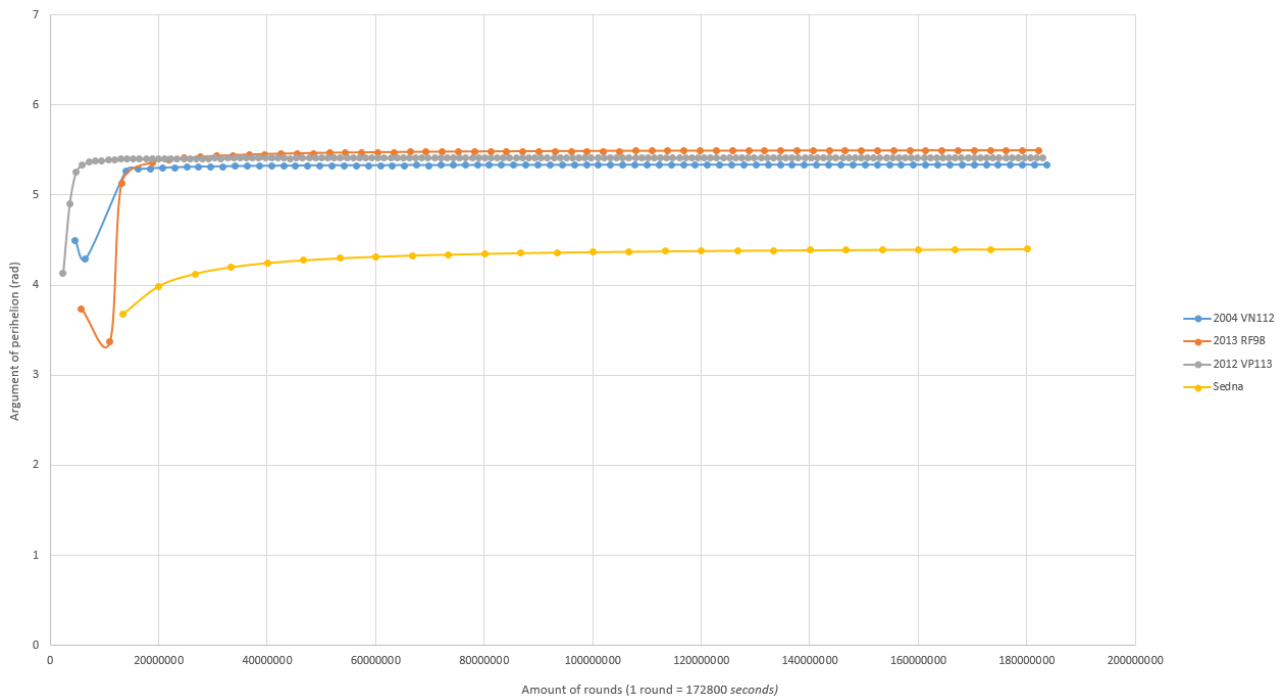
It can be seen in figure 17.3 that three ETNO's remain relativtely clustered (although their respective difference in ω gets larger over time). Sedna acquires an entirely different argument of perihelion. For this reason (like was concluded before) planet nine is, according to our own results, needed 'to conserve the situation'. The simulator used the settings described in the Appendix, with a timestep of 172800 and included the Sun, Jupiter, Saturn, Neptune, Uranus and the tested object (all were tested individually). Also, the following vectors were used for the ETNOs (with exception of Sedna & 2012 VP_{113} whose vectors[12] are already in the Appendix):

2004 VN$_{112}$

position: (3.338469440683407E+01, 3.296760926256486E+01, -8.176834813898699E+00)
speed: (-1.830443771273609E-03, 2.551493797427650E-03, 1.295080364913495E-03)

---

[12]The position vectors are in AU, the speed vectors are in AU/day.

2007 TG$_{442}$

position: (-2.216102118938070E+00, -5.957656766688118E-01, -9.228532887388547E-03)
speed: (1.973707536998759E-03, -1.106231446142322E-02, -1.188438173809993E-04)

2013 RF$_{98}$

position: (2.809064890818173E+01, 2.117251775628629E+01, -1.015547278525787E+01)
speed: (-1.408524658517317E-03, 3.354634129283988E-03, 1.461376116722572E-03)

2010 GB$_{174}$

position: (-6.661904379651325E+01, -8.411238128232725E+00, 2.212233193483758E+01)
speed: (-9.610782795963537E-04, -2.406268777135870E-03, 9.081217152229448E-04)

Sadly, these tests failed on 2007 TG$_{442}$ and 2010 GB$_{174}$ who both started flying towards the Sun with a perihelion and aphelion between 0.05 and 2 AU. Therefore they are not included in figure 17.3.[13]

17.3 If the answer to question 2 is yes, what is its mass (determined to a certain degree)?

Originally, we wanted to determine a possible position of planet nine on our own using a markov chain monte carlo. We didn't succeed to find an answer to this question for several reasons. First and foremost, the orbits of the objects that have led to the prediction of the existence of planet nine in the first place have not been observed for long enough to perform such as a calculation with any precision. Moreover, we didn't succeed in comprehending the markov chain monte carlo at even nearly a sufficient level to implement it in the simulator, because we don't know enough about statistics and our time was up. For these reasons we couldn't predict the postion of planet nine.

We did however (on the advice of professor Portegies Zwart) take a possible position from the literature[14] and tested several possible masses of planet nine to get a rough idea of its mass.

17.3.1 The effect of adding Planet Nine

In the literature, (de la Fuente Marcos et al., 2016), the following two sets of possible orbital elements for planet nine are mentioned:

Set 1

- $\alpha = 700$ AU
- $e = 0.6$

- $i = 30°$
- $\Omega = 113°$
- $\omega = 150°$

---

[13]Both of these objects started much closer to the solar system than the others (they were near their perihelion), and therefore the guessed mass and big timestep mattered a lot and propably caused their weird orbits. Their masses are 'not known' so we had to use a rough estimate. More thoughts about the accuracy of these graphs can be found in the Discussion.

[14](de la Fuente Marcos et al., 2016)

- $f = 117.8°$

Set 2

- $\alpha = 701$ AU
- $e = 0.6$

- $i = 33°$
- $\Omega = 89°$
- $\omega = 142°$
- $f = 180°$

For the simulator to work, these need to be converted into state vectors. The corresponding vectors for Set 1 are: position: (504.87295629, 188.36337907, -310.80926724) and speed: (0.00810759, 0.03933163, -0.01318158) and for Set 2: position: (594.46056469, 873.59114602, -376.08736567) and speed: (-0.01227576, 0.01184265, 0.00810498).[15]

For the mass, we took the estimate of about 10 Earth masses, or $5.97219 \cdot 10^{25}$ kg.

When simulated with 10 Earth masses[16] on the position given by the vectors corresponding to Set 1, planet nine wasn't attracted to the Sun and drifted away from the solar system.

Therefore we tried a mass of 25 Earth masses, or $1.4930475 \cdot 10^{26}$ kg. The result was just the same.

At last, we tried a mass of 5 Earth masses, or $2.986095 \cdot 10^{25}$ kg. This last test made planet nine escape the solar system even faster than the first two did.

We can only conclude for this that 2 days is too big of a timestep (which is unlikely) or that these positions for planet nine are not likely according to our simulator, because the planet would escape the solar system, unfortunately we cannot say anything about its mass.

17.4 Is there another solution by which the present situation in our solar system can be explained?

There are several possible solutions, namely:

- A stellar encounter may have put these inner Oort cloud objects into the current position and configuration. During such an encouter they may have been perturbed by the Kozai mechanism into the current situation. But we know from question 2 that the argument of perihelion changes without the constant presence of a perturber, so the stellar encounter must not have occurred to long ago because then the effect wouldn't be observable any more. Also, it has been mentioned in the literature that an object at the distance of Sedna would have its perihelion modified by less than one percent over its lifetime. It would require a very very close stellar encounter which is not expectable because of the current position of our solar system in the Milky Way. Combined with the fact that the stellar encounter must have happened recently, makes it an unlikely explanation.[17]

- These objects are captured planetisimals from other stars. If they were captured from the same star at the same time, that could explain the configuration. It must be noted that there are a lot of conditions that have to be satisfied in order for this to occur which makes it unlikely.

---

[15]These were calculated using a converter at https://janus.astro.umd.edu/orbits/elements/convertframPle.html

[16]Including the Sun, Jupiter, Saturn, Neptune, Uranus and 2012 VP$_{(}$113) with timesteps of 172800 seconds and using the settings described in the Appendix for the other objects.

[17](Brown et al., 2004)

- The galactic tide may have influenced the inner Oort cloud objects. This means that the argument of perhelion of the different objects assumed a constant value because of the influence of the Milky Way. But in (Brown et al., 2004) it was mentioned that such effect tend to become important at distances like $\approx 10^4$ AU, because the objects concerned aren't nearly at such distance. This explanation is hardly plausible.

- It can also be the case that there are several distant giant planets in the outer solar system. Which were all perturbed into the Oort cloud in the formation of the solar system or maybe captured from another star.

- Their configuration could have been formed during the time when the sun was still in its birth cluster. However, this explanation still needs an additional reason why the argument of perihelion did not change.

- Being by moved out by Neptune's migration. Neptune and Uranus migrated farther from the Sun due to the influence of Jupiter and Saturn in the solar system's early history. This could have created the current configuration.

Each of these solutions has a different orbital configuration of the dynamical system as a result and should be tested individually to see if they can provide the current situation in our solar system. That is a new topic for further research.

Part VI

Discussion and Conclusion

Chapter 18

Discussion

> The crucial thing is what comes at the end.
>
> ———————————
>
> Helmut Kohl

18.1 Thoughts on the Process

Christiaan is a professional computer scientist and entrepreneur - he runs his own software company (Verictas) - with an interest in understanding the universe through physics. Daniel has a great interest in physics, mathematics and astronomy, but computer science is not his forte. Sometimes it was very difficult to communicate between us two what the problem was or to suggest a solution. Ultimately, the two disciplines worked very well together and really complemented one another.

In this thesis we have gained a lot of new knowledge about different subjects, such as numerical methods, celestial mechanics, geometry etc. If we had had this background knowledge at the start, we would have of course approached this thesis differently. We would for instance have had a much clearer idea of how you could go about approximating the orbital elements of planet nine. Also we went into a lot of different subjects that didn't turn out very useful for what our thesis turned out to be in the end, these include complex numbers and quaternions, the Kepler laws, Hamiltonian and Lagrangian mechanics etc. But that is probably the way it always goes in research.

One final thing that has to be mentioned is the change in the simulator. Like was mentioned in the chapter 'The technical side of the computer model' initially we had a very different way of calculating the argument of perihelion, for example using rotation checks. We found out again thanks to professor Portegies Zwart that there is also an analytical way (which you can by hand calculations) of calculating it. But in the end, we found that our original model was more accurate than our new one, which is strange since our original model is much more prone to errors than our new one.

18.2 Suggestions for further research

Of course someone else with much more computing power and more knowledge about N-body simulations must redo our calculcations to make sure they are accurate.Also, because of a lack of time, we could have studied motion resonances, hamiltonian systems etc. in much more detail to get an analytical idea of the orbit of planet nine so that we have a better idea of how the computer model should be modified, because we then understand the situation much better theoretically.

Due to a shortage of time, we had no time to really understand the markov chain monte carlo algorithm and to implement it into our own model so that we could get a real idea of the position of planet nine and to calculate where it should approximately be in the night sky. Also more study of numerical methods and differential equations would have bettered our own model.

Also, Daniel would have liked to really study more the analytical side of this subject so that he can understand the analytical parts of articles like (Bailey et al., 2016) and (Bernstein and Khushalani, 2000).

18.3   Some Thoughts upon the Accuracy of the Simulator

To understand more of the accuracy of the simulator, we urge the reader to read the technical explanation of both the Old and New Simulators, described in the 'The Technical Side of the Computer Model' chapter. In this section we'll discuss some tests with the accuracy of the old simulator, as that's the one we're using to produce the results[1].

18.3.1   The aphelion and perihelion

The aphelion and perihelion are important measures of accuracy, as they can be determined quite explicitly. They do depend on the way of determening when a full rotation around the star has occurred, and can therefore be one or two timesteps behind, or in front of the actual timestep, in which the aphelion or perihelion occurs.

When calculating ten years worth of Earth positional data, the following averages for the aphelion and perihelion are reached.

- Aphelion: $152,098 \cdot 10^6$ km
- Perihelion: $147,089 \cdot 10^6$ km

Compared to the following data from (Simon et al., 1994)[2]:

- Aphelion: $152,10 \cdot 10^6$ km
- Perihelion: $147,10 \cdot 10^6$ km

After comparison it becomes clear that the positional data from the simulator almost exacly matches the data from (Simon et al., 1994).[3].

The aphelion and perihelion of the ETNOs

To measure the accuracy of the orbits of the simulated ETNOs[4] they are compared aphelion & perihelion data from their respective Wikipedia pages[5]

---

[1]It's worth mentioning that the old and new simulator are the exact same integrator, and will thus produce the same results for the positions of the objects

[2]aphelion = a × (1 + e); perihelion = a × (1 − e), where a is the semi-major axis and e is the eccentricity

[3]The aphelion & perihelion calculated by the simulator were listed with three digets, instead of a possibly more fitting two digets. The data in (Simon et al., 1994) was only listed with two digets.

[4]Sedna, 2012 VP_{113}, 2004 VN_{112}, 2013 RF_{98}

[5](?), (?), (?) and (?)

Sedna

Average as generated with the simulator:

- Aphelion: $3,18 \cdot 10^{11}$ km

- Perihelion: $1,14 \cdot 10^{10}$ km

Average as listed by (**?**):

- Aphelion: $1,40 \cdot 10^{11}$ km

- Perihelion: $1,14 \cdot 10^{10}$ km

2012 VP$_{113}$

Average as generated with the simulator:

- Aphelion: $9,03 \cdot 10^{10}$ km

- Perihelion: $1,22 \cdot 10^{10}$ km

Average as listed by (**?**):

- Aphelion: $6,43 \cdot 10^{10} \pm 1,95 \cdot 10^{9}$ km

- Perihelion: $1,20 \cdot 10^{10} \pm 7,48 \cdot 10^{7}$ km

2004 VN$_{112}$

Average as generated with the simulator:

- Aphelion: $1,52 \cdot 10^{11}$ km

- Perihelion: $7,13 \cdot 10^{9}$ km

Average as listed by (**?**):

- Aphelion: $9,08 \cdot 10^{10} \pm 2,70 \cdot 10^{9}$ km

- Perihelion: $7,08 \cdot 10^{9}$ km

2013 RF$_{98}$

Average as generated with the simulator:

- Aphelion: $1,87 \cdot 10^{11}$ km

- Perihelion: $5,47 \cdot 10^{9}$ km

Average as listed by (**?**):

- Aphelion: $9,95 \cdot 10^{10} \pm 3,14 \cdot 10^{9}$ km

- Perihelion: $5,48 \cdot 10^{9}$ km

For all objects the perihelion match the listed value on Wikipedia very well, but the aphelion for all objects are larger than the listed values. This is likely due to the timestep, but these aphelion are also not known very precisely.

18.3.2 The effect of changing settings on 2012 VP$_{113}$

In figure 18.1 the effect of changing the settings of the simulator is displayed. We ran 7 tests, and a control (with the normal settings of 172800 seconds per timestep). The tests are described below:

- Test 1: Changing the timestep from 172800 seconds (about 2 days) to 432000 seconds (about 5 days). This resulted in a failure, in which 2012 VP$_{113}$ departed the solar system and flew away into the abyss.

---

[7]The simulation included the Sun, Jupiter, Saturn, Neptune, Uranus and 2012 VP$_{113}$. More information about the settings can be found in the Appendix.

Figure 18.1: The argument of periapsis of 2012 VP$_{113}$ as calculated by the simulator over almost 250.000 years[7].
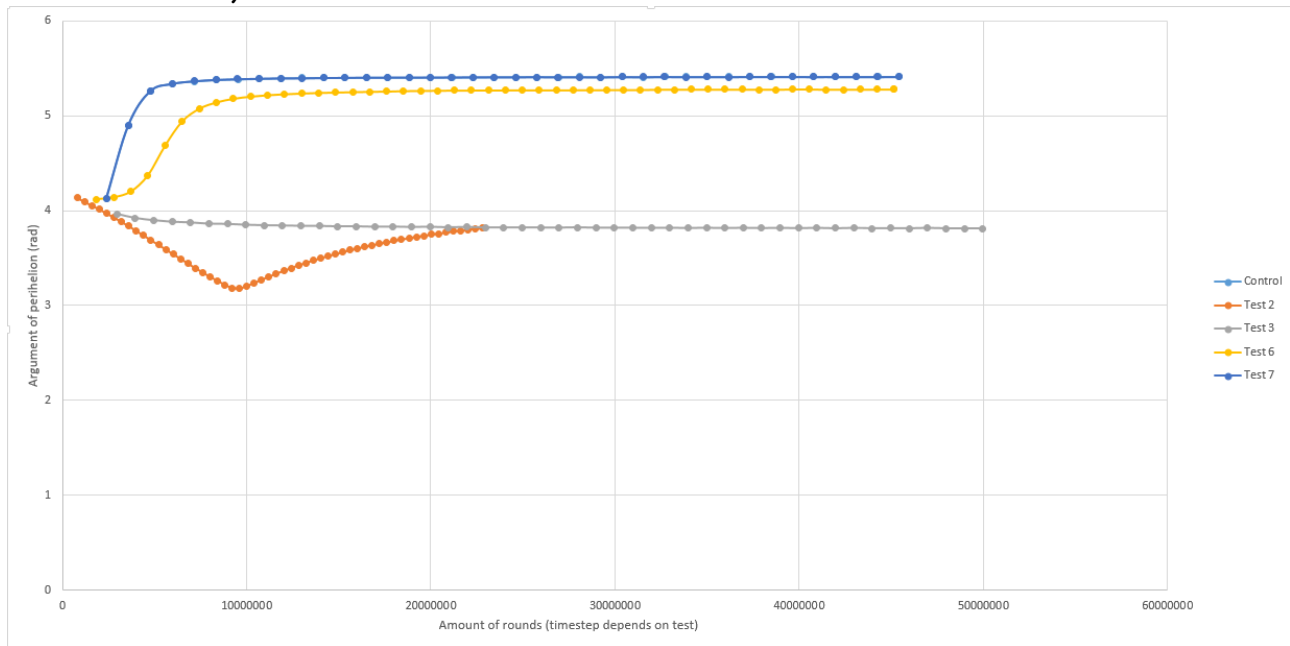
- Test 2: Changing the timestep from 172800 seconds (about 2 days) to 345600 seconds (about 4 days). This resulted in a weird orbit, with a weird graph for the argument of perihelion, as can be seen in the figure.

- Test 3: Changing the timestep from 172800 seconds (about 2 days) to 86400 seconds (about a day). The results actually extended over a longer period (twice as long as the control), but that has been cut of for clarity in the graph. Therefore it doesn't run for the same length of time as the other tests.

- Test 4: Changing the timestep from 172800 seconds (about 2 days) to 43200 seconds (half a day). This resulted in a weird orbit, possibly due to calculation errors, and did not result in any arguments.

- Test 5: Changing the mass from $2,7 \cdot 10^{18}$ kg to $2,7 \cdot 10^{17}$ kg. This resulted in the exact same values as the control did. It is therefore not shown in the figure.

- Test 6: Changing the mass from $2,7 \cdot 10^{18}$ kg to $2,7 \cdot 10^{10}$ kg.

- Test 7: Changing the mass from $2,7 \cdot 10^{18}$ kg to $2,7 \cdot 10^{25}$ kg. This also resulted in the same values as the control, but Test 7 is shown in the figure, at the cost of the control not being visible.

Some conclusions can be drawn from these tests:

1. The mass doesn't matter as much. The difference between $2,7 \cdot 10^{18}$ kg and $2,7 \cdot 10^{17}$ kg on 2012 VP$_{113}$ wasn't noticable, as was the difference between $2,7 \cdot 10^{18}$ kg and $2,7 \cdot 10^{25}$ kg, despite there being a $1 \cdot 10^{7}$ kg difference. Making the object too light however, seems to matter a bit, because Test 6 has different values.

2. The timestep is very important and can totally change the results. The difference between the tests with different time steps is much more apparent than between the tests with different masses. This also explains why the old simulator seemed to produce weird results before (with an incorrect timestep)

Chapter 19

Conclusion

One can conclude that planet nine is a possible (because of our simulation) but hardly a likely solution to the problem stated in this thesis. However, none of the other proposed explanations is really likely and simple too. There are several reasons why we don't deem it likely:

1. We performed a test with a likely position (according to the literature), it is discussed in the 'Results' chapter. We found that with each mass we did a test, planet nine disappeared from the solar system and therefore didn't fit as a solution with these orbital elements.

2. Several articles[1] [2] say that it is likely that there exist at least two planets in the Oort cloud in order to account for the observations. We don't have the statistical tools and knowledge to calculate the chance that two big planets in the outer solar system exist and do not have been detected yet. But we know that is very low and probably considerably lower than the chance that one planet exists. If there are two planets 'needed', that makes the planet nine hypothesis even less likely.

3. Thanks to profesor Icke, our attention was drawn to another important fact. It is very hard to explain how planet nine got there in the first place. We recognize that there are three main options to explain where planet nine came from:[3] [4] [5]

   • It comes from the inner solar system and was later perturbed outwards by the other gas planets. If it formed in the inner solar system and was ejected into the Oort cloud, then the question is: Why weren't Uranus and Neptune ejected so far? Why is there such a big distance between Neptune and planet nine? The answer to the second question that there are more undiscovered giant planets, which leads us back to reason 2.

   • It formed in the Oort cloud. We don not consider it likely that it has formed in the Oort cloud itself because there is simply not enough mass there[6] and there probably never was.

   • It was captured from another planetary system or it was a lone planet drifting through the galaxy that was then 'caught'. If it was captured from another star, then it is likely that more objects were captured. The scars of this encounter would probably be much more visible, because these captured objects would also have entered the inner solar system.

---

[1](de la Fuente Marcos et al., 2016)
[2](de la Fuente Marcos and de la Fuente Marcos, 2014)
[3](?)
[4](?)
[5](?)
[6](Batygin and Brown, 2016b)

As you can see, each of the three origin possibilities has its own troubles, which makes planet nine's origins a difficult question.

4. Finally, we will give a historical reason. In the decades before the discovery of Pluto astronomers became convinced that there was a Planet X (a big distant giant planet) in the outer solar system. That was a big hype on the time, but it was grounded in real scientific evidence (just like today). Pluto didn't turn out to be a big planet and eventually it was found that there had been calculation and estimation errors, they were fortunately errors in good faith (see 'the Historical Background' chapter for details), and Planet X was never found. We are not saying that anyone is doing bad research, we are just saying that maybe like last time, we're all doing something wrong and planet nine does not exist.

For all these reasons, we think that planet nine does not exist.

Chapter 20

Bibliography

R.A. Adams and C. Essex. *Calculus: A Complete Course*. Prentice-Hall, Toronto, 2013.

E. Bailey, K. Batygin, and M.E. Brown. Solar obliquity induced by planet nine. *The Astronomical Journal*, pages 126–133, 2016.

J. Baker. *50 inzichten universum*. Veen Magazines, Zutphen, 2011.

K. Batygin and M.E. Brown. Observational constraints on the orbit and location of planet nine in the outer solar system. *The Astrophysical Journal*, page 824:2, 2016a.

K. Batygin and M.E. Brown. Evidence for a distant giant planet in the solar system. *The Astronomical Journal*, page 151:22, 2016b.

G. Bernstein and B. Khushalani. Orbit fitting and uncertainties for kuiper belt objects. *Astronomical Journal*, pages 3323–3332, 2000.

W. Bosma. *Lichamen*. Radboud University, Nijmegen, 2016.

D.W. Boutros. *Belangrijke Astronomen*. Stedelijk Gymnasium Nijmegen, Nijmegen, 2013.

R.E.A. Bouwens, P.A.M. de Groot, W. Kranendonk, and J.P. van Lune et al. *BiNaS*. Noordhoff Uitgevers, Groningen, 2013.

M.E. Brown, C. Truijllo, and D. Rabinowitz. Discovery of a candidate inner oort cloud planetoid. *The Astrophysical Journal*, pages 645–649, 2004.

M.E. Brown, C. Truijllo, and D. Rabinowitz. Discovery of a planetary-sized object in the scattered kuiper belt. *The Astrophysical Journal*, pages 97–100, 2005.

C. Byrne. *Kepler's Laws of Planetary Motion*. University of Massachusetts Lowell, Lowell, 2014.

S.J. Cowley. *Mathematics Tripos: 1A Vector Calculus*. University of Cambridge, Cambridge, 2000.

C. de la Fuente Marcos and R. de la Fuente Marcos. Extreme trans-neptunian objects and the kozai mechanism: signalling the presence of trans-plutonian planets. *Monthly Notices of the Royal Astronomical Society*, pages 59–63, 2014.

C. de la Fuente Marcos and R. de la Fuente Marcos. Finding planet nine: apsidal anti-alignment monte carlo results. *Monthly Notices of the Royal Astronomical Society*, pages 1972–1977, 2016.

C. de la Fuente Marcos, R. de la Fuente Marcos, and Sverre J. Aarseth. Dynamical impact of the planet nine scenario: N-body experiments. *Monthly Notices of the Royal Astronomical Society*, pages 460–464, 2016.

I. de Pater and J.J. Lissauer. *Planetary Sciences*. Cambridge University Press, Cambridge, 2016.

V. Dorenbos and E. Kedzierska, 2011. URL `http://www.cma-science.nl/downloads/nl/software/coach6/c6_3_handboek_coachtaal.pdf`.

S. Friedberg, A. Insel, and L. Spence. *Linear Algebra*. Pearson Education Limited, Harlow, 2003.

H. Goldstein, C.P. Poole jr., and J.L. Safko. *Classical Mechanics*. Addison Wesley, New York, 2000.

G.J. Heckman. *On the Shoulders of Giants*. Radboud University, Nijmegen, 2015.

H.van Gendt and R. Dames. *Lineaire algebra toegepast*. Stedelijk Gymnasium Nijmegen, Nijmegen, 2008.

Hyperphysics, 2016. URL `http://hyperphysics.phy-astr.gsu.edu/hbase/Solar/picsol/retro2003.jpg`.

M.L. Kutner. *Astronomy: A Physical Perspective*. Cambridge University Press, Cambridge, 2003.

H.W. Lenstra jr., F. Oort, and B.J.J. Moonen. *Ringen en Lichamen*. Radboud University, Nijmegen, 2014.

E. Limburg, N. de Boer, M. de Haas, A. Verbrugge, and T. Tersteeg. *Algemene Natuurwetenschappen - De Wetenschappelijke Methode - Syllabus Periode 1*. Stedelijk Gymnasium Nijmegen, Nijmegen, 2015.

S.P. Maran. *Sterrenkunde voor Dummies*. Pearson, Amsterdam, 2012.

I. Ridpath. *Oxford Dictionary of Astronomy*. Oxford University Press, Oxford, 2012.

G. Schilling. *Handboek Sterrenkunde*. Fontaine Uitgevers, Hilversum, 2012.

R. Schwarz, 2016. URL `https://downloads.rene-schwarz.com/download/M002-Cartesian_State_Vectors_to_Keplerian_Orbit_Elements.pdf`.

R.A. Serway and J.W. Jewett Jr. *Physics for Scientists and Engineers*. Cengage Learning, Boston, 2014.

J.L. Simon, P. Bretagnon, J. Chapront, M. Chapront-Touzé, G.Francou, and J. Laskar. Numerical expressions for precession formulae and mean elements for the moon and planets. *Astronomy and Astrophysics*, page 663–683, 1994.

S.A. Terwijn. *Inleiding in de Wiskunde*. Radboud University, Nijmegen, 2014.

C.A. Trujillo and S.S. Sheppard. A sedna-like body with a perihelion of 80 astronomical units. *Nature*, pages 471–474, 2014.

A. van den Essen. *Crash Course Lineaire Algebra 2*. Radboud University, Nijmegen, 2015a.

A. van den Essen. *Lineaire Afbeeldingen en Matrices*. Radboud University, Nijmegen, 2015b.

Muriel van der Laan. *Stable Configurations of Planetary Systems*. KNAW, Amsterdam, 2015.

P. van Oostrum, 2016. URL `http://www.win.tue.nl/latex/documentation/manual.pdf`.

F. Watson, M. Anderson, C. Burgess, and L. Dalrymple et al. *Astronomica*. Millennium House, Hilversum, 2007.

Wikipedia, 2016a. URL `https://en.wikipedia.org/wiki/Apsis`.

Wikipedia, 2016b. URL `https://en.wikipedia.org/wiki/Semi-major_and_semi-minor_axes`.

Wikipedia, 2016c. URL `https://en.wikipedia.org/wiki/IAU_definition_of_planet`.

Wikipedia, 2016d. URL `https://en.wikipedia.org/wiki/Eccentricity_vector`.

Wikipedia, 2016e. URL `https://nl.wikipedia.org/wiki/Infinitesimaal`.

Wikipedia, 2016f. URL `https://nl.wikipedia.org/wiki/Inwendig_product`.

Wikipedia, 2016g. URL `https://nl.wikipedia.org/wiki/Johannes_Kepler`.

Wikipedia, 2016h. URL `https://en.wikipedia.org/wiki/Kozai_mechanism`.

Wikipedia, 2016i. URL `https://en.wikipedia.org/wiki/Argument_of_periapsis`.

Wikipedia, 2016j. URL `https://en.wikipedia.org/wiki/Philosophi%C3%A6_Naturalis_Principia_Mathematica`.

Wikipedia, 2016k. URL `https://en.wikipedia.org/wiki/Ptolemy`.

Wikipedia, 2016l. URL `https://en.wikipedia.org/wiki/Pythagorean_theorem`.

Wikipedia, 2016m. URL `https://en.wikipedia.org/wiki/Solar_System`.

Wikipedia, 2016n. URL `https://en.wikibooks.org/wiki/LaTeX`.

Wikipedia, 2016o. URL `https://nl.wikibooks.org/wiki/LaTeX`.

Wikipedia, 2016p. URL `https://en.wikipedia.org/wiki/Planet_Nine`.

Part VII

Appendices

Appendix A

The Matrices

In this appendix, we will write down the entire matrices used in the section 'Matrix Calculations'. Note that:

$$\|\vec{r_i} - \vec{r_j}\| = \|\vec{r_j} - \vec{r_i}\| \tag{A.1}$$

$$A = \begin{pmatrix} 0 & \frac{1}{(\|\vec{r_1}-\vec{r_2}\|)^3} \cdot (\vec{r_1} - \vec{r_2}) & \cdots & \frac{1}{(\|\vec{r_1}-\vec{r_n}\|)^3} \cdot (\vec{r_1} - \vec{r_n}) \\ \frac{1}{(\|\vec{r_2}-\vec{r_1}\|)^3} \cdot (\vec{r_2} - \vec{r_1}) & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \frac{1}{(\|\vec{r_{n-1}}-\vec{r_n}\|)^3} \cdot (\vec{r_{n-1}} - \vec{r_n}) \\ \frac{1}{(\|\vec{r_n}-\vec{r_1}\|)^3} \cdot (\vec{r_n} - \vec{r_1}) & \cdots & \frac{1}{(\|\vec{r_n}-\vec{r_{n-1}}\|)^3} \cdot (\vec{r_n} - \vec{r_{n-1}}) & 0 \end{pmatrix} \tag{A.2}$$

$$B = \begin{pmatrix} m_1 & 0 & \cdots & 0 \\ 0 & m_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \cdots & 0 & m_n \end{pmatrix} \tag{A.3}$$

$$C = \begin{pmatrix} 0 & \frac{m_1}{(\|\vec{r_1}-\vec{r_2}\|)^3} \cdot (\vec{r_1} - \vec{r_2}) & \cdots & \frac{m_1}{(\|\vec{r_1}-\vec{r_n}\|)^3} \cdot (\vec{r_1} - \vec{r_n}) \\ \frac{m_2}{(\|\vec{r_2}-\vec{r_1}\|)^3} \cdot (\vec{r_2} - \vec{r_1}) & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \frac{m_{n-1}}{(\|\vec{r_{n-1}}-\vec{r_n}\|)^3} \cdot (\vec{r_{n-1}} - \vec{r_n}) \\ \frac{m_n}{(\|\vec{r_n}-\vec{r_1}\|)^3} \cdot (\vec{r_n} - \vec{r_1}) & \cdots & \frac{m_n}{(\|\vec{r_n}-\vec{r_{n-1}}\|)^3} \cdot (\vec{r_n} - \vec{r_{n-1}}) & 0 \end{pmatrix} \tag{A.4}$$

$$CB = \begin{pmatrix} 0 & \frac{m_1 m_2}{(\|\vec{r_1}-\vec{r_2}\|)^3} \cdot (\vec{r_1} - \vec{r_2}) & \cdots & \frac{m_1 m_n}{(\|\vec{r_1}-\vec{r_n}\|)^3} \cdot (\vec{r_1} - \vec{r_n}) \\ \frac{m_2 m_1}{(\|\vec{r_2}-\vec{r_1}\|)^3} \cdot (\vec{r_2} - \vec{r_1}) & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \frac{m_{n-1} m_n}{(\|\vec{r_{n-1}}-\vec{r_n}\|)^3} \cdot (\vec{r_{n-1}} - \vec{r_n}) \\ \frac{m_n m_1}{(\|\vec{r_n}-\vec{r_1}\|)^3} \cdot (\vec{r_n} - \vec{r_1}) & \cdots & \frac{m_n m_{n-1}}{(\|\vec{r_n}-\vec{r_{n-1}}\|)^3} \cdot (\vec{r_n} - \vec{r_{n-1}}) & 0 \end{pmatrix} \tag{A.5}$$

$$G \cdot CB = \tag{A.6}$$

$$
\begin{pmatrix}
0 & G \cdot \frac{m_1 m_2}{(\|\overrightarrow{r_1} - \overrightarrow{r_2}\|)^3} \cdot (\overrightarrow{r_1} - \overrightarrow{r_2}) & \cdots & G \cdot \frac{m_1 m_n}{(\|\overrightarrow{r_1} - \overrightarrow{r_n}\|)^3} \cdot (\overrightarrow{r_1} - \overrightarrow{r_n}) \\
G \cdot \frac{m_2 m_1}{(\|\overrightarrow{r_2} - \overrightarrow{r_1}\|)^3} \cdot (\overrightarrow{r_2} - \overrightarrow{r_1}) & 0 & \cdots & \vdots \\
\vdots & \vdots & \ddots & G \cdot \frac{m_{n-1} m_n}{(\|\overrightarrow{r_{n-1}} - \overrightarrow{r_n}\|)^3} \cdot (\overrightarrow{r_{n-1}} - \overrightarrow{r_n}) \\
G \cdot \frac{m_n m_1}{(\|\overrightarrow{r_n} - \overrightarrow{r_1}\|)^3} \cdot (\overrightarrow{r_n} - \overrightarrow{r_1}) & \cdots & G \cdot \frac{m_n m_{n-1}}{(\|\overrightarrow{r_n} - \overrightarrow{r_{n-1}}\|)^3} \cdot (\overrightarrow{r_n} - \overrightarrow{r_{n-1}}) & 0
\end{pmatrix}
$$

Appendix B

The Simulator Data/Settings

The following vectors were used with the simulator[1]:

Object Name	Positional Vector (AU)	Speed Vector (AU/day)
Sun	(3.7378817113150281E-03,1.402397586692506E-03,-1.612700291840256E-04)	(8.619338996535534E-07,6.895607793642275E-06,-2.794074909231784E-08)
Earth	(-1.630229002588497E-01,9.704723344534316E-01,-1.955367328932975E-04)	(-1.723383356491747E-02,-2.969134550063944E-03,-4.433758674928828E-07)
The Moon	(-1.657103868749121E-01,9.706382026425473E-01,-1.879812512691582E-04)	(-1.728100931961937E-02,-3.525371122447976E-03,4.909148618073602E-05)
Jupiter	(-5.172279968303672E+00,1.591564562098799E+00,1.090553487095606E-01)	(-2.306423668033420E-03,-6.856869314900905E-03,8.012916249248967E-05)
Saturn	(-3.710637850378867E+00,-9.289569433157130E+00,3.091990731378936E-01)	(4.874750391005278E-03,-2.086615906689840E-03,-1.574898601194673E-04)
Venus	(-7.130901319004951E-01,-5.719763212192740E-02,4.040076577877051E-02)	(1.525993024372452E-03,-2.024175581604569E-02,-3.656582385749146E-04)
Mars	(-1.644664047074283E+00,1.714211195991345E-01,4.385749324150048E-02)	(-9.128062787682906E-04, -1.271783289037382E-02, -2.442517367300464E-04)
Neptune	(2.795458622849629E+01,-1.077602237438394E+01,-4.223299945454949E-01)	(1.108107308612818E-03, 2.948021656576779E-03, -8.584675894389943E-05)
Uranus	(1.887206485673029E+01,6.554830107743496E+00,-2.201473388797619E-01)	(-1.319173006464416E-03, 3.532006412470987E-03, 3.002475806591822E-05)
Sedna	(4.831201219703945E+01, 6.863113643822504E+01, -1.773001247239095E+01)	(-2.401309021644802E-03, 7.269559406640982E-04, 1.704114106899654E-04)
2012 VP$_{113}$	(5.074554081273273E+01, 6.194684521116067E+01, -2.303377758579428E+01)	(-1.390042223661063E-03, 1.919356165611094E-03, 6.083057470436023E-04)

The following masses (in kg) were used:

- Sun: 1.988544E30

- Earth: 5.97219E24

- The Moon: 734.9E20

- Jupiter: 1898.13E24

- Saturn: 5.68319E26

- Venus: 48.685E23

- Mars: 6.4185E23

- Neptune: 102.41E24

- Uranus: 86.8103E24

- Sedna: 4E21

- 2012 VP$_{113}$: 2.7E18

[1]We got our initial starting positions and speeds from the NASA HORIZIONS Web-Interface at http://ssd.jpl.nasa.gov/horizons.cgi

Appendix C

The Meetings

Because our research encapsulates several subjects at a higher level than our own we chose to meet with several professors across mathematics, physics, astronomy and computer science. The coming appendices are summaries of our meetings with them. They have all been very helpful to us in making several of the concepts clear that were important for our thesis. Please note that the summaries are not quote verbatim and have been translated. In this chapter we will first give a summary of the first two meetings.

Daniel went to Gert Heckman first. Gert Heckman is a professor of mathematical physics at the Radboud University (in Nijmegen). He works in group theory, representation theory, hypergeometric functions and algebraic, hyperbolic and sympletic geometry. In the first year of the mathematics bachelor, he teaches a course on the Kepler laws, Daniel has read (parts) the lecture notes of this course which is the reason he went to him. Mr. Heckman himself even discovered a new proof for Kepler's first law.

In the meeting Mr. Heckman explained the proof to Daniel. It gave Daniel a feel of the mathematical language used when talking about orbits.

Daniel then went to Frank Verbunt. Frank Verbunt is a professor of high-energy astrophysics at the Radboud University in Nijmegen. He has worked on X-ray astronomy, neutron stars and white dwarfs in binaries and the history of astronomy. Daniel came into contact with him because he originally emailed professor Gijs Nelemans (also from Radboud university and one of the discovers of gravitational waves) because he teaches a course on planetary systems. Subsequently, professor Nelemans directed Daniel to professor Verbunt.

In the meeting the current research on planet nine was discussed, which included the way it was predicted, how likely the hypothesis is etc. Also orbital elements and monte carlo methods were briefly discussed.

Appendix D

Summary of the meeting with professor Zantema

Hans Zantema is a professor of computer science at the Radboud University in Nijmegen (and the Eindhoven University of Technology). Software and algorithms are a few of his specialisations and he promoted on algebraic number theory.

How long will a full simulation take approximately?

The calculation time will be somewhere around one second, but that will depend on how many objects you want to simulate and how much they interact with eachother.

What do you think is the best way to transform this problem into an algorithm?

I would start with a few objects (ten or twenty) and then try different starting positions and speeds for planet nine. Calculating the first 4000 years of these objects far from the sun will take something in the order of miliseconds.

The parameters for planet nine could be guessed with some AI. Then you can calculate some sort of score with a pre-determined formula to determine which begin parameters are the most likely. Eventually you also can simulate those with your visualisation tool.

Appendix E

Summary of the meeting with professor Icke

Vincent Icke is a professor of theoretical astronomy at the university of Leiden. He is also a professor by special appointment of Cosmology at the university of Amsterdam (UvA). His research interests include cosmology and the hydrodynamics of the high-energy and relativistic flow of gases around dying stars and compact objects.

Can we determine the position of planet nine exactly?

Icke showed us a simulation of an enormous asteroid belt around a star with a giant planet orbiting it on his Macbook.

Look at this simulation, it takes a very long time before anything happens. And even then, to determine the position of this giant perturber requires you to examine hundreds of objects. We now know only six of them, so it is for now impossible to say anything accurate.

Do you think planet nine exists?

No, if such an object exists in the Oort cloud, one must ask: "How did it get there?" There is not enough mass in the Oort cloud to form such a big object, and if it was kicked out by the other gas planets that far, there must be several objects between the known planets and the hypothesized planets.

Furthermore the chance that this alignment in argument of perihelion is a mere coincidence is equal to:

$$\frac{1}{\sqrt{n-1}} \tag{E.1}$$

You can calculate it for yourself, the researchers based their hypothesis on the six known objects in the Oort cloud, so the chance that this is coincidence is vast. We have to wait and see that once more objects have been discovered, they will exhibit the same properties. I understand that the authors of the article have to earn a living doing science, but I find it not entirely scientific to call out now: We predict a new planet!

Appendix F

Summary of the meeting with professors Portegies Zwart and Icke

After we had met professor Icke, we went (together with him) to Portegies Zwart's office. Simon Portegies Zwart is a professor of computational astrophysics at the university of Leiden.

How did Batygin derive these equations (Daniel showed Zwart and Icke the article (Batygin and Brown, 2016b))?)

Portegies Zwart: This is all classical mechanics, but I have to warn you: The derivation of these equations is not simple. You have to look it in up in (Goldstein et al., 2000), but it will take you a lot of time.

What is the best statistical method to determine a possible position of planet nine because there is a relatively large number of parameters?

Portegies Zwart: The best way to get a good estimation is a monte carlo method. However, this has a problem. There is a chance that you get stuck in a local minimum, while there exists a much more likely set of solutions. A standard monte carlo method works like this: You have a set of values of the different parameters, you let them change a little bit randomly and look if they are more likely. If they are more likely, you take them as your new values. If not, you retain your old values. But like I said, this could get you stuck in a local minimum. A markov chain monte carlo works the same way except that when the new values are less likely, you accept them with a certain chance. That way you can get out of a local minimum.

You should also try to first program a big planet and see if it has any effect. If such a planet has no effect, one can be sure that smaller ones also have little effect and that it is unlikely that planet nine exists.

Icke: You also have to think about the time you have left. Programming all of it can take a lot of work.

Portegies Zwart: Yes, he is right. Programming a markov chain monte carlo generator can take a full week of work, 40 hours or so. I know it feels good that you do it yourself. But several other have done it already so it is best that you take it from the internet. It will save you a lot of time which you can then use for better purposes. Can I have a look at the basis of the computer model you already have?

We handed them our printed PWS (which was what we had at that time).

Portegies Zwart: Ah! You have made use of Euler integration. It is very good that I have noticed it. Euler integration is a way of numerically evaluating an integral or differential euqations. In other words, it is a way of calculating the motion of bodies. But the problem is that it will result in a very big deviation from the real trajectory of a body. It is much better to use leapfrog integration. That will correct the error. I have made a website together with several other colleagues, on this site I have written a leapfrog integrator in a lot of different

programming languages, I will give you the URL later. I am glad I have seen this 'mistake' because otherwise your model would have been useless.

Icke: At the start of each academic year I show the new first-year students both the Euler and leapfrog integration so that they can see the difference between the two with their own eyes.

Did you know that national physics curriculum in secondary school contains some modelling in a programme called Coach 6 which makes use of the Euler integration?

Portegies Zwart: I find this very strange, because the Euler integration is an algorithm which a big deviation so it is strange that they use Coach 6 in the national curriculum and subsequently not tell the students that it gives a big deviation over time..

Are relativistic effects important in the outer solar system?

Portegies Zwart: No, it plays a role in the orbit of the planet Mercury (which had kept astronomers puzzled for a long time) and which ultimately formed evidence for Einstein's theory of general relativity. But once you go farther away from the sun, the relativistic are negligible.

So no need for general relativity?

Portegies Zwart: Fortunately not.

Do you believe planet nine exists?

Portegies Zwart: I don't think so. The question that one must himself concerning planet nine is: How did it get there? That is very hard to explain. It is very unlikely that it formed in the Oort Cloud because the predicted total mass of the Oort cloud is less than one Earth mass. If it formed in the inner solar system and it was kicked out by the other gas planets, why weren't Uranus and Neptune kicked out then? Or were there more planets then? Where have they gone? You notice from all these questions of course, that it is very hard to explain the origin of planet nine.

Appendix G

Summary of the second meeting with professor Portegies Zwart

Portegies Zwart: Tell me what you have done so far.

We have built a computer model which simulates the solar system and which subsequently calculates the argument of perihelion.

My goodness! How have you done that?

We have built in a rotation check so that we can see when an orbit is finished and subsequently we can find the ascending node and the eccentricity vector.

Ah! That was very smart of you, you should keep it like that, your method is good enough for your purpose. You must know that there are algorithms to convert cartesian state vectors into Keplerian orbital elements.

Could you tell us more about that?

Well, it goes beyond the scope of this meeting. It is no rocket science, but you need some mathematical tools in your arsenal which you two probably do not have yet.

But have you any results from your simulation? Can you show them to me?

Christiaan showed him our PWS with graphs of the argument of perihelion of both Sedna and 2012 VP$_{113}$ simulated over time (figures 13.1 and 13.2).

Do these graphs seem even a bit realistic to you?

I can already see that there are several flaws. Firstly, the graph of Sedna seems like a periodic function, which is obviously not the case. Secondly, the growth of ω is much to large than I would expect. Finally, there is some startup flaw in the model because of the big sweep at the beginning.

What do you think we should do next?

In your PWS you should talk about the precision of the simulation. You should simulate the Earth and Jupiter separately over some time (because you know that their orbits are stable) and put graphs into your thesis from these experiments and compare your results with established results so that you can say something about the accuracy of your model. Also you should talk about all the odd things in the graphs in your PWS.

After that, you should take a possible position from planet nine from the literature and try different masses in that position to get a very rough idea of its mass. I must already say that, you have done impressive work so far. You can put it in your PWS that I have said that.

The problem cannot be solved by Lagrange points?

No, because Lagrange points only apply to circular orbits and not very eccentric ones like planet nine's orbit probably is.

Appendix H

The Full Code of the Old Simulator

H.1 Main.java

```java
package com.verictas.pos.simulator;

import com.verictas.pos.simulator.mathUtils.AU;

import javax.vecmath.*;

public class Main {
    /**
     * PLANETARY ORBIT SIMULATOR
     * Data Simulation Tool
     *
     * Programmed for the PWS "Planeet Negen" for the Stedelijk Gymnasium Nijmegen,
     *   the Netherlands.
     *
     * =================================
     *
     * The MIT License (MIT)
     * Copyright (c) 2016 Christiaan Goossens (Verictas) & Daniel Boutros
     *
     * The full license is included in the git repository as LICENSE.md
     */

    public static int version = 1;

    public static void main(String[] args) {
        /**
         * Object definitions
         */

        /**
         * Definitions for the ecliptic plane (by 1st of january 2016)
         */
        Object sun = new Object("Sun", 1.988544E30, AU.convertToMeter(new
            Vector3d(3.737881713150281E-03,1.402397586692506E-03,-1.612700291840256E-04)),
            AU.convertToMetersPerSecond(new
            Vector3d(8.619338996535534E-07,6.895607793642275E-06,-2.794074909231784E-08)));
        Object earth = new Object("Earth", 5.97219E24, AU.convertToMeter(new
            Vector3d(-1.630229002588497E-01,9.704723344534316E-01,-1.955367328932975E-04)),
            AU.convertToMetersPerSecond(new
            Vector3d(-1.723383356491747E-02,-2.969134550063944E-03,-4.433758674928828E-07)));
```

```java
Object moon = new Object("The_Moon", 734.9E20, AU.convertToMeter(new
    Vector3d(-1.657103868749121E-01,9.706382026425473E-01,-1.879812512691582E-04)),
    AU.convertToMetersPerSecond(new
    Vector3d(-1.728100931961937E-02,-3.525371122447976E-03,4.909148618073602E-05)));
Object jupiter = new Object("Jupiter", 1898.13E24, AU.convertToMeter(new
    Vector3d(-5.172279968303672E+00,1.591564562098799E+00,1.090553487095606E-01)),
    AU.convertToMetersPerSecond(new
    Vector3d(-2.306423668033420E-03,-6.856869314900905E-03,8.012916249248967E-05)));
Object saturn = new Object("Saturn", 5.68319E26, AU.convertToMeter(new
    Vector3d(-3.710637850378867E+00,-9.289569433157130E+00,3.091990731378936E-01)),
    AU.convertToMetersPerSecond(new
    Vector3d(4.874750391005278E-03,-2.086615906689840E-03,-1.574898601194673E-04)));
Object venus = new Object("Venus", 48.685E23, AU.convertToMeter(new
    Vector3d(-7.130901319004951E-01,-5.719763212192740E-02,4.040076577877051E-02)),
    AU.convertToMetersPerSecond(new
    Vector3d(1.525993024372452E-03,-2.024175581604569E-02,-3.656582385749146E-04)));
Object mars = new Object("Mars", 6.4185E23, AU.convertToMeter(new
    Vector3d(-1.644664047074283E+00,1.714211195991345E-01,4.385749324150048E-02)),
    AU.convertFromMetersPerSecond(new Vector3d(-9.128062787682906E-04,
    -1.271783289037382E-02, -2.442517367300464E-04)));
Object pluto = new Object("Pluto", 1.307E22, AU.convertToMeter(new
    Vector3d(8.535178336776600E+00,-3.187687983153820E+01,9.421570822362236E-01)),
    AU.convertFromMetersPerSecond(new Vector3d(3.105916866228581E-03,
    1.759704223757070E-04, -9.146208184741589E-04)));
Object neptune = new Object("Neptune", 102.41E24, AU.convertToMeter(new
    Vector3d(2.795458622849629E+01,-1.077602237438394E+01,-4.223299945454949E-01)),
    AU.convertFromMetersPerSecond(new Vector3d(1.108107308612818E-03,
    2.948021656576779E-03, -8.584675894389943E-05)));
Object uranus = new Object("Uranus", 86.8103E24, AU.convertToMeter(new
    Vector3d(1.887206485673029E+01,6.554830107743496E+00,-2.201473388797619E-01)),
    AU.convertFromMetersPerSecond(new Vector3d(-1.319173006464416E-03,
    3.532006412470987E-03, 3.002475806591822E-05)));
Object charon = new Object("Charon", 1.53E21, AU.convertToMeter(new
    Vector3d(8.535206843097511E+00,-3.187692375327401E+01,9.420370068039806E-01)),
    AU.convertFromMetersPerSecond(new Vector3d(3.015458707073605E-03,
    8.495285732817140E-05, -9.028237165874783E-04)));

// PWS Objects
Object object1 = new Object("Sedna", 4E21, AU.convertToMeter(new
    Vector3d(4.831201219703945E+01, 6.863113643822504E+01,
    -1.773001247239095E+01)), AU.convertToMetersPerSecond(new
    Vector3d(-2.401309021644802E-03, 7.269559406640982E-04,
    1.704114106899654E-04)));
Object object2 = new Object("2012_VP113", 2.7E18, AU.convertToMeter(new
    Vector3d(5.074554081273273E+01, 6.194684521116067E+01,
    -2.303377758579428E+01)), AU.convertToMetersPerSecond(new
    Vector3d(-1.390042223661063E-03, 1.919356165611094E-03,
    6.083057470436023E-04)));
Object object3 = new Object("2004_VN112", 0, AU.convertToMeter(new
    Vector3d(3.338469440683407E+01, 3.296760926256486E+01,
    -8.176834813898699E+00)), AU.convertToMetersPerSecond(new
    Vector3d(-1.830443771273609E-03, 2.551493797427650E-03,
    1.295080364913495E-03)));
```

```java
        Object object4 = new Object("2007_TG442", 0, AU.convertToMeter(new
            Vector3d(-2.216102118938070E+00, -5.957656766688118E-01,
            -9.228532887388547E-03)), AU.convertToMetersPerSecond(new
            Vector3d(1.973707536998759E-03, -1.106231446142322E-02,
            -1.188438173809993E-04)));
        Object object5 = new Object("2013_RF98", 0, AU.convertToMeter(new
            Vector3d(2.809064890818173E+01, 2.117251775628629E+01,
            -1.015547278525787E+01)), AU.convertToMetersPerSecond(new
            Vector3d(-1.408524658517317E-03, 3.354634129283988E-03,
            1.461376116722572E-03)));
        Object object6 = new Object("2010_GB174", 0, AU.convertToMeter(new
            Vector3d(-6.661904379651325E+01, -8.411238128232725E+00,
            2.212233193483758E+01)), AU.convertFromMetersPerSecond(new
            Vector3d(-9.610782795963537E-04, -2.406268777135870E-03,
            9.081217152229448E-04)));

        /**
         * Object listing
         */

        Object[] objects = {}; // Fill in the objects to be simulated


        /**
         * Run the simulator for the specified objects
         */
        Simulator.run(objects);
    }
}
```

H.2   Node.java

```java
package com.verictas.pos.simulator;

import javax.vecmath.Vector3d;

/**
 * Storage object for storing nodes on the graph
 */
public class Node extends Vector3d {
    public int round;

    /**
     * Constructor for casting
     * @param vector
     */
    public Node(Vector3d vector) {
        this.set(vector);
    }

    /**
```

```java
     * Constructor for empty creation
     */
    public Node() {
        this.set(new Vector3d(0,0,0));
    }

    /**
     * Sets the stored round associated with this node
     * (It will most likely be the round when this node is reached)
     * @param round
     */
    public void setRound(int round) {
        this.round = round;
    }

    public boolean empty() {
        if (this.getX() == 0 && this.getY() == 0 && this.getZ() == 0) {
            return true;
        }
        return false;
    }
}
```

H.3 Object.java

```java
package com.verictas.pos.simulator;
import javax.vecmath.*;
import java.lang.*;

public class Object {
    public double mass;
    public Vector3d position;
    public Vector3d speed;

    public Vector3d acceleration;
    public Vector3d oldAcceleration;

    public String name;

    private double gravitationalConstant = 6.67384E-11;

    /**
     * Constructs an object
     * @param mass The mass of the object
     * @param position The position vector of the object
     * @param speed The speed vector of the object
     */
    public Object(String name, double mass, Vector3d position, Vector3d speed) {
        this.name = name;
        this.mass = mass;
        this.position = position;
```

```java
        this.speed = speed;
        this.oldAcceleration = new Vector3d(0,0,0);
        this.acceleration = new Vector3d(0,0,0);
    }

    /**
     * Sets the speed vector of an object
     * @param speed Current speed vector
     */
    public void setSpeed(Vector3d speed) {
        this.speed = speed;
    }
    public void setSpeed(double[] speed) {
        this.speed = new Vector3d(speed[0], speed[1], speed[2]);
    }

    /**
     * Gets the speed into a double[3]
     * @return double[3]
     */
    public double[] getSpeed() {
        double[] v = new double[3];
        this.speed.get(v);
        return v;
    }

    /**
     * Sets the position vector of an object.
     * @param position Current position vector
     */
    public void setPosition(Vector3d position) {
        this.position = position;
    }
    public void setPosition(double[] position) {
        this.position = new Vector3d(position[0], position[1], position[2]);
    }

    /**
     * Gets the position into a double[3]
     * @return double[3]
     */
    public double[] getPosition() {
        double[] r = new double[3];
        this.position.get(r);
        return r;
    }

    /**
     * Sets the acceleration vector of an object
     * @param acceleration Current acceleration vector
     */
    public void setAcceleration(Vector3d acceleration) { this.acceleration =
        acceleration; }
```

```java
    public void setAcceleration(double[] acceleration) {
        this.acceleration = new Vector3d(acceleration[0], acceleration[1],
            acceleration[2]);
    }

    /**
     * Gets the acceleration into a double[3]
     * @return double[3]
     */
    public double[] getAcceleration() {
        double[] a = new double[3];
        this.acceleration.get(a);
        return a;
    }

    /**
     * Sets the acceleration vector of an object
     * @param acceleration Current acceleration vector
     */
    public void setOldAcceleration(Vector3d acceleration) { this.acceleration =
        acceleration; }
    public void setOldAcceleration(double[] acceleration) {
        this.oldAcceleration = new Vector3d(acceleration[0], acceleration[1],
            acceleration[2]);
    }

    /**
     * Gets the acceleration into a double[3]
     * @return double[3]
     */
    public double[] getOldAcceleration() {
        double[] a = new double[3];
        this.oldAcceleration.get(a);
        return a;
    }

    /**
     * Changes an object into readable form
     * @return String
     */
    public String toString() {
        return "Mass:␣" + this.mass + "␣&␣Position:␣" + this.position + "␣&␣Speed:␣
            " + this.speed;
    }

    /**
     * Calculates the force of the passed object on the current object.
     * @param secondObject The passed object
     * @return Vector3d The gravitational force
     */
    public Vector3d getForceOnObject(Object secondObject) {
        double scale = gravitationalConstant * ((this.mass * secondObject.mass) /
            Math.pow(getDistance(secondObject).length(), 3.0));
```

```java
        Vector3d force = getDistance(secondObject);
        force.scale(scale);
        return force;
    }


    /**
     * Get the vector distance between the current position vector and the position
         vector of the passed object.
     * @param secondObject The passed object.
     * @return Vector3d The distance vector
     */
    public Vector3d getDistance(Object secondObject) {
        Vector3d distance = new Vector3d(0,0,0); // Empty
        distance.sub(this.position, secondObject.position);
        return distance;
    }


    /**
     * Get the vector distance between the current position vector and a given
         position.
     * @param position The position vector you want the distance to.
     * @return Vector3d The distance vector
     */
    public Vector3d getDistance(Vector3d position) {
        Vector3d distance = new Vector3d(0,0,0); // Empty
        distance.sub(this.position, position);
        return distance;
    }


    /**
     * Updates the position based on dt
     * @param dt The difference in time
     */
    public void updatePosition(double dt) {
        // Write the vectors to double[3]
        double[] r = this.getPosition();
        double[] v = this.getSpeed();
        double[] a = this.getAcceleration();

        for (int i = 0; i != 3; i++){
            double dt2 = dt * dt;
            r[i] += v[i] * dt + 0.5 * a[i] * dt2;
        }

        // Write the doubles into the vectors to save them
        setPosition(r);
        setSpeed(v);
        setAcceleration(a);
    }


    /**
     * Updates the speed based on dt
     * @param dt The difference in speed
```

```java
    */
    public void updateSpeed(double dt) {
        // Write the vectors to double[3]
        double[] v = this.getSpeed();
        double[] a = this.getAcceleration();
        double[] aold = this.getOldAcceleration();

        for (int i = 0; i != 3; i++){
            v[i] += 0.5 * dt *(a[i] + aold[i]);
        }

        setSpeed(v);
        setAcceleration(a);
        setOldAcceleration(aold);
    }

    /**
     * Updates the acceleration based on dt
     */
    public void updateAcceleration() {
        this.oldAcceleration = this.acceleration;
    }

    /**
     * Enacts a certain force on the object
     * @param force The force in N.
     */
    public void enactForceOnObject(Vector3d force) {
        double factor = 1/this.mass;
        Vector3d acceleration = force;
        acceleration.scale(factor);
        this.acceleration = acceleration;
    }
}
```

H.4   Simulator.java

```java
package com.verictas.pos.simulator;
import javax.vecmath.*;

import com.verictas.pos.simulator.dataWriter.WritingException;
import com.verictas.pos.simulator.mathUtils.Vector3dMatrix;
import com.verictas.pos.simulator.processor.ProcessingException;
import com.verictas.pos.simulator.processor.Processor;

public class Simulator {
    public static int round = 0; // Stores an global integer value with the
        current round (as a timestamp)

    /**
     * Run method for the Simulator
```

```java
     * @param objects
     */
    public static void run(Object[] objects) {

        /**
         * Get variables from the config
         */

        int rounds = SimulatorConfig.rounds;
        double time = SimulatorConfig.time;

        /**
         * Log a debug message to the console to signal the simulation has started
         */
        System.out.println("========== Simulation Started ==========\n");

        /**
         * Create a time to measure runtime
         */
        long startTime = System.currentTimeMillis();

        /**
         * Define the forces matrix and the DataWriter
         */
        Vector3dMatrix matrix = new Vector3dMatrix(objects.length, objects.length);

        try {
            Processor processor = new Processor(objects);

            /**
             * Start the leap frog integration!
             */

            accelerate(objects, matrix);

            /**
             * Start the rounds loop
             */
            for(int t = 0; t != rounds; t++) {
                // Set round
                Simulator.round++;

                /**
                 * The round has started
                 */
                if(SimulatorConfig.logConsole) {
                    if(SimulatorConfig.skipConsole == -1 || Simulator.round %
                        SimulatorConfig.skipConsole == 0 || Simulator.round == 1) {
                        System.out.println("Round " + Simulator.round + " started!");
                    }
                }

                for(int i = 0; i < objects.length; i++) {
```

```java
                objects[i].updatePosition(time);
                objects[i].updateAcceleration();
            }

            accelerate(objects, matrix);

            for(int i = 0; i < objects.length; i++) {
                objects[i].updateSpeed(time);
            }

            /**
             * Do the processing on the objects
             */
            processor.process(objects);

            /**
             * The round has ended
             */
        }

        /**
         * Log that the simulation has finished and save info to file
         */
        processor.close();
        System.out.println("========= Simulation Finished =========");

        /**
         * Display information about the program runtime
         */
        long stopTime = System.currentTimeMillis();
        System.out.println("Simulation took: " + (stopTime - startTime) + "ms");
    } catch(ProcessingException e) {
        System.out.println("\nERROR:: Processing failed.");
        e.printStackTrace();
    } catch(WritingException e) {
        System.out.println("\nERROR:: Writing to file failed.");
        e.printStackTrace();
    }
}

/**
 * Accelerates the given objects, puts the results in the given matrix and
     enacts forces
 * @param objects
 * @param matrix
 */
private static void accelerate(Object[] objects, Vector3dMatrix matrix) {
    // Loop
    for(int i = 0; i < objects.length; i++) {
        /**
         * For every object: calculate the force upon it.
         */
```

```java
            // Reset acceleration
            objects[i].setAcceleration(new Vector3d(0, 0, 0));

            for (int o = 0; o < objects.length; o++) {
                /**
                 * Loop through all other objects
                 */
                if (o == i) {
                    break;
                }

                Vector3d force = objects[i].getForceOnObject(objects[o]);
                matrix.setPosition(force, i, o);

                /**
                 * Also put in the opposite force
                 */
                force.scale(-1);
                matrix.setPosition(force, o, i);
            }
        }

        for(int i = 0; i < objects.length; i++) {
            /**
             * Progress forces on the object
             */
            Vector3d forceOnI = matrix.getColumnTotal(i);
            objects[i].enactForceOnObject(forceOnI);
        }
    }
}
```

H.5   SimulatorConfig.java (empty)

```java
package com.verictas.pos.simulator;

public class SimulatorConfig {

    /**
     * (Example) Settings for the EARTH
     * Rounds: 1051896 * (amount of years to run)
     * Time: 30
     */

    /**
     * (Example) Settings for SEDNA
     * Rounds: 184000000 (approx. 1 million years)
     * Time: 172800 (2 days)
     */

    /**
```

```java
     * (Example) Settings for 2012 VP113
     * Rounds: 184000000 (approx. 1 million years)
     * Time: 172800 (2 days)
     */
    /**
     * Time settings
     */

    public static int rounds = 0; // Amount of rounds to run the simulator for
    public static double time = 0; // Time steps in seconds

    /**
     * Object settings
     */

    public static String sunName = "Sun"; // The name of the sun to calculate
        values TO
    public static String[] objectNames = {}; // The name of the object(s) your
        want to calculate the values OF

    /**
     * Output preferences
     */

    public static String outputUnit = "AU"; // Preferred output unit preference
        (AU => AU/day, m => m/s)
    public static int outputNumbers = 0; // Preferred way of outputting numbers:
        (0 => comma for decimals, dot in large numbers OR 1 => comma for large
        numbers, dot with decimals)
    public static int skipLines = 1; // Set the skipLines integer to skip lines
        (for example: every 5th line is written) in the output file (for smaller
        files), if this is set to 1, it has no effect and all lines will be written.
    public static boolean skipUnnecessary = true; // Skip the unnecessary objects
        in the export

    /**
     * Console settings
     */
    public static boolean logConsole = true;
    public static int skipConsole = 1;
}
```

H.6 dataWriter/DataWriter.java

```java
package com.verictas.pos.simulator.dataWriter;

import com.verictas.pos.simulator.Main;
import com.verictas.pos.simulator.SimulatorConfig;

import java.io.File;
import java.io.FileWriter;
```

```java
import java.io.IOException;
import java.text.*;
import java.util.Date;

public class DataWriter {
    protected FileWriter writer = null;

    /**
     * Set global variables, such as the delimiter and the new line character
     */
    protected static final String DELIMITER = "\t";
    protected static final String NEW_LINE = "\n";

    protected int counter = 0;

    /**
     * Decimal formatter
     */

    public DecimalFormat formatter = new DecimalFormat();

    /**
     * Constructor
     * @throws WritingException
     */
    public DataWriter(String filenameAppendix) throws WritingException {

        /**
         * Prepare the locale
         */

        try {
            /**
             * Define the save path
             */
            String directory = System.getProperty("user.home") + File.separator +
                "simulatorExports";
            File directoryPath = new File(directory);

            String path = directory + File.separator + "v" + Main.version + "-" +
                getCurrentTimeStamp() + "-" + filenameAppendix + ".txt";
            System.out.println("WRITING␣DATA␣TO:␣" + path);

            /**
             * Check if the saving directory exists to prevent IOException
             */
            if (!directoryPath.exists()) {
                directoryPath.mkdirs();
            }

            /**
             * Open a file to write to and write the header
             */
```

```java
        this.writer = new FileWriter(path);

        /**
         * Configure the decimal formatter
         */
        DecimalFormatSymbols symbols = new DecimalFormatSymbols();
        if (SimulatorConfig.outputNumbers == 0) {
            symbols.setDecimalSeparator(',');
            symbols.setGroupingSeparator('.');
        } else {
            symbols.setDecimalSeparator('.');
            symbols.setGroupingSeparator(',');
        }
        this.formatter.setDecimalFormatSymbols(symbols);
        this.formatter.setMinimumFractionDigits(0);
        this.formatter.setMaximumFractionDigits(25);
    } catch(IOException e) {
        throw new WritingException("The destination file couldn't be created.");
    } catch(Exception e) {
        throw new WritingException("Some unknown error occurred while writing to
            the file!");
    }
}


/**
 * Writes a string to the file
 * @param string
 * @throws WritingException
 */
public void write(String string) throws WritingException {
    if (this.writer == null) {
        throw new WritingException("The writer isn't defined yet");
    } else {
        try {
            if (this.counter % SimulatorConfig.skipLines == 0) {
                this.writer.append(string);
            }
            this.counter++;
        } catch (Exception e) {
            e.printStackTrace();
            throw new WritingException("An error occurred while writing to the
                file!");
        }
    }
}

protected String decimalFormatter(double input) {
    return this.formatter.format(input);
}

/**
 * Saves the file to disk
 * @throws WritingException
```

```java
         */
    public void save() throws WritingException {
        if (this.writer == null) {
            throw new WritingException("The writer isn't defined yet");
        } else {
            try {
                this.writer.flush();
                this.writer.close();
            } catch (IOException e) {
                throw new WritingException("Whoop! Save error!");
            }
        }
    }

    /**
     * Gets the current line count
     * @return int
     */
    public int getLines() {
        return this.counter;
    }

    /**
     * Gets the current filestamp for file naming
     * @return String
     */
    private String getCurrentTimeStamp() {
        return new SimpleDateFormat("yyyy-MM-dd_HH-mm-ss").format(new Date());
    }
}
```

H.7 dataWriter/AOPDataWriter.java

```java
package com.verictas.pos.simulator.dataWriter;

import java.util.*;

public class AOPDataWriter extends DataWriter {
    public AOPDataWriter() throws WritingException {
        super("arguments");
        try {

            /**
             * Write the lines with information about the columns
             */

            this.writer.write("OBJECT" + DELIMITER + "ROUND" + DELIMITER + "ARGUMENT
                (RAD)" + NEW_LINE);
            this.counter++;
        } catch (Exception e) {
            e.printStackTrace();
```

```java
                throw new WritingException("An error occurred while writing to the
                    file!");
        }
    }

    public void write(String object, TreeMap<Integer, Double> arguments) throws
        WritingException {
        try {
            for (Map.Entry<Integer, Double> entry : arguments.entrySet()) {
                Integer key = entry.getKey();
                Double value = entry.getValue();
                this.writer.append(object + DELIMITER + key + DELIMITER +
                    decimalFormatter(value) + NEW_LINE);
                this.counter++;
            }
        } catch (Exception e) {
            e.printStackTrace();
            throw new WritingException("An error occurred while writing to the
                file!");
        }
    }
}
```

H.8 dataWriter/PosDataWriter.java

```java
package com.verictas.pos.simulator.dataWriter;

import com.verictas.pos.simulator.Object;
import com.verictas.pos.simulator.SimulatorConfig;
import com.verictas.pos.simulator.mathUtils.AU;

import javax.vecmath.Vector3d;

public class PosDataWriter extends DataWriter {
    public PosDataWriter() throws WritingException {
        super("position");
        try {

            /**
             * Write the lines with information about the columns
             */

            if (SimulatorConfig.outputUnit.equals("AU")) {
                this.writer.write("Object" + DELIMITER + "X (AU)" + DELIMITER + "Y
                    (AU)" + DELIMITER + "Z (AU)" + DELIMITER + "VX (AU/day)" +
                    DELIMITER + "VY (AU/day)" + DELIMITER + "VZ (AU/day)" + NEW_LINE);
            } else {
                this.writer.write("Object" + DELIMITER + "X (m)" + DELIMITER + "Y
                    (m)" + DELIMITER + "Z (m)" + DELIMITER + "VX (m/s)" + DELIMITER +
                    "VY (m/s)" + DELIMITER + "VZ (m/s)" + NEW_LINE);
            }
```

```java
                this.counter++;
        } catch (Exception e) {
            e.printStackTrace();
            throw new WritingException("An error occurred while writing to the
                file!");
        }
    }


    /**
     *
     * @param object The object you want to write data about
     * @throws WritingException
     */
    public void write(Object object) throws WritingException {
        String id = object.name;
        Vector3d position = object.position;
        Vector3d speed = object.speed;
        Vector3d AUposition = AU.convertFromMeter(position);
        Vector3d AUspeed = AU.convertFromMetersPerSecond(speed);

        if (this.writer == null) {
            throw new WritingException("The writer isn't defined yet");
        } else {
            try {
                if (this.counter % SimulatorConfig.skipLines == 0) {
                    if (SimulatorConfig.outputUnit.equals("AU")) {
                        this.writer.append(id + DELIMITER +
                            decimalFormatter(AUposition.getX()) + DELIMITER +
                            decimalFormatter(AUposition.getY()) + DELIMITER +
                            decimalFormatter(AUposition.getZ()) + DELIMITER +
                            decimalFormatter(AUspeed.getX()) + DELIMITER +
                            decimalFormatter(AUspeed.getY()) + DELIMITER +
                            decimalFormatter(AUspeed.getZ()) + NEW_LINE);
                    } else {
                        this.writer.append(id + DELIMITER +
                            decimalFormatter(position.getX()) + DELIMITER +
                            decimalFormatter(position.getY()) + DELIMITER +
                            decimalFormatter(position.getZ()) + DELIMITER +
                            decimalFormatter(speed.getX()) + DELIMITER +
                            decimalFormatter(speed.getY()) + DELIMITER +
                            decimalFormatter(speed.getZ()) + NEW_LINE);
                    }
                }
                this.counter++;
            } catch (Exception e) {
                e.printStackTrace();
                throw new WritingException("An error occurred while writing to the
                    file!");
            }
        }
    }
}
```

H.9 dataWriter/WritingException.java

```java
package com.verictas.pos.simulator.dataWriter;

public class WritingException extends Exception {
    public WritingException() { super(); }
    public WritingException(String message) { super(message); }
    public WritingException(String message, Throwable cause) { super(message,
        cause); }
    public WritingException(Throwable cause) { super(cause); }
}
```

H.10 mathUtils/AOP.java

```java
package com.verictas.pos.simulator.mathUtils;

import javax.vecmath.Vector3d;

public class AOP {
    /**
     * Helper class for calculating the argument of periapsis
     */
    public static double calculate(Vector3d ascendingNode, Vector3d perihelion,
        Vector3d aphelion) {
        Vector3d eccentricity = new Vector3d(0,0,0);
        eccentricity.sub(perihelion, aphelion);

        if (eccentricity.getZ() < ascendingNode.getZ()) {
            return (2 * Math.PI) - ascendingNode.angle(eccentricity);
        } else {
            return ascendingNode.angle(eccentricity);
        }
    }
}
```

H.11 mathUtils/AU.java

```java
package com.verictas.pos.simulator.mathUtils;

import javax.vecmath.Vector3d;

public class AU {
    /**
     * Helper class for working with astronomical units
     */
```

```java
/**
 * Converts AU to meters
 * @param input Vector3d with values in AU
 * @return Vector3d with values in meter
 */
public static Vector3d convertToMeter(Vector3d input) {
    Vector3d output = new Vector3d(input);

    // Convert AU to m by NASA
    output.scale(149597870.700); // Number to large when multiplied with 1000
    output.scale(1000);

    return output;
}

/**
 * Converts AU/day to m/s
 * @param input Vector3d with values in AU/day
 * @return Vector3d with values in m/s
 */
public static Vector3d convertToMetersPerSecond(Vector3d input) {
    Vector3d output = new Vector3d(input);

    // 1 AU/day to M/s
    output.scale(1731456.84);

    return output;
}

/**
 * Converts meters to AU for data collection
 * @param input Vector3d with values in meters
 * @return Vector3d with values in AU
 */
public static Vector3d convertFromMeter(Vector3d input) {
    Vector3d output = new Vector3d(input);

    // Convert m to AU by NASA
    output.scale(6.6845871E-12);

    return output;
}

public static double convertFromMeter(double input) {
    return input * 6.6845871E-12;
}

/**
 * Converts m/s to AU/day for data collection
 * @param input Vector3d with values in m/s
 * @return Vector3d with values in AU/day
 */
public static Vector3d convertFromMetersPerSecond(Vector3d input) {
```

```java
        Vector3d output = new Vector3d(input);

        // Convert seconds to days by NASA
        output.scale(5.77548327E-7);

        return output;
    }
}
```

H.12   mathUtils/Vector3dMatrix.java

```java
package com.verictas.pos.simulator.mathUtils;

import javax.vecmath.GMatrix;
import javax.vecmath.Vector3d;

public class Vector3dMatrix extends GMatrix {
    /**
     * Creates a new matrix with some helper functions for use with Vector3f. The
         created matrix will be empty.
     * @param n The number of rows.
     * @param m The number of columns.
     */
    public Vector3dMatrix(int n, int m) {
        // Change the size to suit Vector3d
        super(n, m * 3);
        this.setZero();
    }

    /**
     * Set the size of the matrix in the amount of vectors (e.g. a 1 x 3 vector
         matrix gets converted to a 1 x 9 storage matrix).
     * @param n The amount of rows
     * @param m The amount of columns expressed in vectors (1 vector = 3 values).
     */
    public void setSizeInVectors(int n, int m) {
        this.setSize(n, m * 3);
    }

    /**
     * Provides a function for putting the matrix into String form for
         visualisation.
     * @return String
     */
    public String toString() {
        StringBuffer buffer = new StringBuffer(this.getNumRow() * this.getNumCol()
            * 8);

        for(int n = 0; n < this.getNumRow(); ++n) {
            for(int m = 0; m < this.getNumCol(); ++m) {
                if ((m + 1) == 1 || m % 3 == 0) {
```

```java
                // If m is 1 or a multiple of 4, begin the bracket.
                buffer.append("(").append(this.getElement(n, m)).append(",␣");
            } else if ((m + 1) % 3 == 0) {
                // If m is a multiple of 3, close the bracket
                buffer.append(this.getElement(n, m)).append(")\t\t");
            } else {
                buffer.append(this.getElement(n, m)).append(",␣");
            }
        }

        buffer.append("\n");
    }

    return buffer.toString();
}


/**
 * Provides a translator from the vector positions (e.g. the second vector
 *   starts at position 1) to the matrix positions (the second vector starts at
 *   position 3).
 * @param n The vector positions row
 * @param m The vector positions column
 * @return void
 */
private int[] translatePosition(int n, int m) {
    return new int[]{n, m * 3};
}


/**
 * Provides a way to set a vector into a certain position in the matrix
 * @param settable The vector you want to put in the matrix
 * @param n The row to insert into
 * @param m The column to insert into
 */
public void setPosition(Vector3d settable, int n, int m) {
    int[] position = translatePosition(n, m);
    n = position[0];
    m = position[1];

    this.setElement(n, m, settable.x);
    this.setElement(n, m + 1, settable.y);
    this.setElement(n, m + 2, settable.z);
}


/**
 * Provides a way to get a vector from a certain position in the matrix
 * @param n The row to get from
 * @param m The column to get from
 * @return Vector3d The vector in that position
 */
public Vector3d getPosition(int n, int m) {
    int[] position = translatePosition(n, m);
    n = position[0];
```

```java
        m = position[1];

        double x = this.getElement(n, m);
        double y = this.getElement(n, m + 1);
        double z = this.getElement(n, m + 2);
        return new Vector3d(x, y, z);
    }

    /**
     * Provides a way to calculate the result vector of a certain row
     * @param row The row to calculate the total of
     * @return Vector3d
     */
    public Vector3d getRowTotal(int row) {
        double[] rowTotal = new double[this.getNumCol()];
        this.getRow(row, rowTotal);

        // Create an empty vector to store the result
        Vector3d resultVector = new Vector3d(0,0,0);

        for(int i = 0; i < this.getNumCol(); i = i + 3) {
            // For every third entry (including 0).
            double x = this.getElement(row, i);
            double y = this.getElement(row, i + 1);
            double z = this.getElement(row, i + 2);
            resultVector.add(new Vector3d(x, y, z));
        }

        return resultVector;
    }

    /**
     * Provides a way to calculate the result vector of a certain column
     * @param column The column to calculate the total of
     * @return Vector3d
     */
    public Vector3d getColumnTotal(int column) {
        double[] columnTotal = new double[this.getNumRow()];

        // Translate the column number to the correct vector column
        int[] position = translatePosition(0, column);
        column = position[1];

        this.getColumn(column, columnTotal);

        // Create an empty vector to store the result
        Vector3d resultVector = new Vector3d(0,0,0);

        for(int i = 0; i < this.getNumRow(); i++) {
            // For every entry (including 0).
            double x = this.getElement(i, column);
            double y = this.getElement(i, column + 1);
            double z = this.getElement(i, column + 2);
```

```
            resultVector.add(new Vector3d(x, y, z));
        }

        return resultVector;
    }
}
```

H.13 processor/ObjectProcessor.java

```java
package com.verictas.pos.simulator.processor;

import com.verictas.pos.simulator.Node;
import com.verictas.pos.simulator.Object;
import com.verictas.pos.simulator.Simulator;
import com.verictas.pos.simulator.SimulatorConfig;

import javax.vecmath.Vector3d;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class ObjectProcessor {
    public Node aphelion;
    public Node perihelion;
    public double aphelionDistance = -1;
    public double perihelionDistance = -1;

    private Object thisObject;
    private Object referenceObject;

    private Vector3d startingPosition;
    private double lastStartDistance = -1;
    private double beforeLastStartDistance = -1;

    public Node ascendingNode;
    public Node descendingNode;

    public Node absoluteMax;
    public Node absoluteMin;

    private Node carryOverNode;
    private int carryOverBit;

    public double referenceZ;

    private HashMap<Integer, Vector3d[]> history = new HashMap<>();

    private double lastMaxRound = -1;
    private double lastMinRound = -1;

    private boolean skipNodes = false;
```

```java
public void setStartingPosition(Vector3d position) {
    this.startingPosition = position;
}

public void setObjectData(Object object) {
    this.thisObject = object;
}

public void setReferenceObjectData(Object object) {
    this.referenceObject = object;
}

/**
 * Keep an history of the object position and speed (for logging and further
   processing)
 */

public void processHistory() {
    this.history.put(Simulator.round, new Vector3d[] {this.thisObject.position,
        this.thisObject.speed});
}

/**
 * Processes the aphelion & perihelion
 */
public void processAphelionAndPerihelion() {
    double sunDistance =
        this.thisObject.getDistance(this.referenceObject).length();

    /**
     * Set the defaults
     */

    if (this.aphelionDistance == -1) {
        this.aphelionDistance = sunDistance;
    }

    if (this.perihelionDistance == -1) {
        this.perihelionDistance = sunDistance;
    }

    /**
     * Check if the aphelion or perihelion should be changed
     */

    if (sunDistance > aphelionDistance) {
        this.aphelion = new Node(this.thisObject.position);
        this.aphelion.setRound(Simulator.round);
        this.aphelionDistance = sunDistance;
    }

    if (sunDistance < perihelionDistance) {
```

```java
            this.perihelion = new Node(this.thisObject.position);
            this.perihelion.setRound(Simulator.round);
            this.perihelionDistance = sunDistance;
        }
    }


    /**
     * Get the absolute maximum and minimum positions (max z and min z)
     */

    public void calculateTops() {
        if (this.absoluteMax == null || this.absoluteMax.empty()) {
            this.absoluteMax = new Node(this.thisObject.position);
            this.absoluteMax.setRound(Simulator.round);
        }

        if (this.absoluteMin == null || this.absoluteMin.empty()) {
            this.absoluteMin = new Node(this.thisObject.position);
            this.absoluteMin.setRound(Simulator.round);
        }

        if (this.thisObject.position.getZ() > this.absoluteMax.getZ()) {
            this.absoluteMax = new Node(this.thisObject.position);
            this.absoluteMax.setRound(Simulator.round);
        }

        if (this.thisObject.position.getZ() < this.absoluteMin.getZ()) {
            this.absoluteMin = new Node(this.thisObject.position);
            this.absoluteMin.setRound(Simulator.round);
        }
    }
    /**
     * Process the nodes
     */

    public void processNodes() {
        /**
         * Determine how the starting positions are
         */

        /**
         * Carry out carry over checking
         */

        if (this.carryOverNode != null) {
            // There is a node present in memory from last round. We should check
                what the carryOverBit is, to see if it's a maximum or a minimum
            if (this.carryOverBit == 1) {
                // Last rounds node is a maximum, we're searching for a descending
                    node
                Node result = this.findNode(this.absoluteMin, this.carryOverNode);
```

```java
        if (!result.empty()) {
            if (SimulatorConfig.logConsole) {
                System.out.println("INFO::␣Found␣descending␣node␣in␣round␣" +
                    result.round + "\n");
            }
            this.descendingNode = result;
        }
    } else {
        // Last rounds node is a minimum, we're searching for an ascending
            node
        Node result = this.findNode(this.carryOverNode, this.absoluteMax);

        if (!result.empty()) {
            if (SimulatorConfig.logConsole) {
                System.out.println("INFO::␣Found␣ascending␣node␣in␣round␣" +
                    result.round + "\n");
            }
            this.ascendingNode = result;
        }
    }

    // Cleaning up
    this.carryOverNode = null;
    this.carryOverBit = -1;
}

/**
 * Carry out the normal checking
 */

double minRound = this.absoluteMin.round;
double maxRound = this.absoluteMax.round;

if (minRound < maxRound) {
    // The minimum came before the maximum node, we're expecting to find the
        ascending node between the two
    // The maximum node should remain in memory to find the descending node
        next round

    Node result = this.findNode(this.absoluteMin, this.absoluteMax);

    if (!result.empty()) {
        if (SimulatorConfig.logConsole) {
            System.out.println("INFO::␣Found␣ascending␣node␣in␣round␣" +
                result.round + "\n");
        }
        this.ascendingNode = result;
    }

    this.carryOverNode = this.absoluteMax;
    this.carryOverBit = 1;
    this.cleanHistory(this.absoluteMax.round);
} else {
```

```java
        // The maximum came before the minimum node, we're expecting to find the
        //    descending node between the two
        // The minimum node should remain in memory to find the ascending node
        //    next round

        Node result = this.findNode(this.absoluteMin, this.absoluteMax);

        if (!result.empty()) {
            if (SimulatorConfig.logConsole) {
                System.out.println("INFO:: Found descending node in round " +
                    result.round + "\n");
            }
            this.descendingNode = result;
        }

        this.carryOverNode = this.absoluteMin;
        this.carryOverBit = 0;
        this.cleanHistory(this.absoluteMin.round);
    }

}

private Node findNode(Node min, Node max) {
    this.referenceZ = (min.getZ() + max.getZ()) / 2;

    if (SimulatorConfig.logConsole) {
        System.out.println("INFO:: Called node finder with min: " + min + " 
            (round " + min.round + ") and max: " + max + " (round " + max.round
            + ") and a reference height of " + referenceZ);
    }

    Node returnNode = new Node();

    if (lastMaxRound == -1 || lastMinRound == -1) {
        lastMinRound = min.round;
        lastMaxRound = max.round;
    } else {
        // You should compare these values to check.
        if (lastMaxRound < min.round && max.round < min.round && min.round ==
            lastMinRound) {
            // max2 > max1 > (min1 = min2)
            System.out.println("WARNING:: This round's values for the nodes 
                shouldn't be trusted. They are calculated incorrectly.");
            this.skipNodes = true;
        }

        if (lastMinRound < max.round && min.round < max.round && max.round ==
            lastMaxRound) {
            // (max1 = max2) > min1 > min2
            System.out.println("WARNING:: This round's values for the nodes 
                shouldn't be trusted. They are calculated incorrectly.");
            this.skipNodes = true;
        }
```

```java
        }

        for (Map.Entry<Integer, Vector3d[]> entry : this.history.entrySet()) {
            Integer round = entry.getKey();
            Vector3d[] vectorArray = entry.getValue();

            boolean roundCheck;

            if (min.round < max.round) {
                roundCheck = min.round < round && round < max.round;
            } else {
                roundCheck = max.round < round && round < min.round;
            }

            if ((this.history.get(round + 1) != null) && roundCheck) {
                // There is a next key and this key is within logical bounds

                if (vectorArray[0].getZ() < referenceZ && this.history.get(round +
                    1)[0].getZ() > referenceZ) {
                    returnNode = new Node(vectorArray[0]);
                    returnNode.setRound(round);
                } else if (vectorArray[0].getZ() > referenceZ &&
                    this.history.get(round + 1)[0].getZ() < referenceZ) {
                    returnNode = new Node(vectorArray[0]);
                    returnNode.setRound(round);
                }
            }
        }

        if (!returnNode.empty()) {
            return returnNode;
        } else {
            return new Node();
        }
    }

    public boolean checkNodes() {
        return !this.skipNodes;
    }

    /**
     * Processes the round check
     */
    public boolean processRoundCheck() {
        double startDistance =
            this.thisObject.getDistance(this.startingPosition).length();
        boolean fullRotation = false;

        /**
         * Check if all are set and shuffle!
         */

        if (beforeLastStartDistance != -1 && lastStartDistance != -1) {
```

```java
            // Ready to go!
            if (beforeLastStartDistance > lastStartDistance && startDistance >
                lastStartDistance) {
                // Last point was the closest to the starting position overall!
                fullRotation = true;
                if (SimulatorConfig.logConsole) {
                    System.out.println("INFO::_Object_" + this.thisObject.name + "_
                        has_made_a_full_rotation_last_round.");
                }
            }

            beforeLastStartDistance = lastStartDistance;
            lastStartDistance = startDistance;
        }

        /**
         * Check if 1st distance is set and 2nd isn't set
         */

        if (beforeLastStartDistance != -1 && lastStartDistance == -1) {
            lastStartDistance = startDistance;
        }

        /**
         * Check if the 1st distance isn't set
         */
        if (beforeLastStartDistance == -1) {
            beforeLastStartDistance = startDistance;
        }

        if(fullRotation) {
            return true;
        } else {
            return false;
        }
    }

    public void reset() {
        aphelionDistance = -1;
        perihelionDistance = -1;
        lastStartDistance = -1;
        beforeLastStartDistance = -1;
        aphelion = new Node();
        perihelion = new Node();
        ascendingNode = new Node();
        descendingNode = new Node();
        absoluteMax = new Node();
        absoluteMin = new Node();
        referenceZ = -1;
        lastMaxRound = -1;
        lastMinRound = -1;
        skipNodes = false;
    }
```

```java
    /**
     * Clears all entries from history before the given key
     * @param key
     */
    public void cleanHistory(int key) {
        for(Iterator<Map.Entry<Integer, Vector3d[]>> it =
            this.history.entrySet().iterator(); it.hasNext(); ) {
            Map.Entry<Integer, Vector3d[]> entry = it.next();
            if(entry.getKey() < key) {
                it.remove();
            }
        }
    }
}
```

H.14   processor/ProcessingException.java

```java
package com.verictas.pos.simulator.processor;

public class ProcessingException extends Exception {
    public ProcessingException() { super(); }
    public ProcessingException(String message) { super(message); }
    public ProcessingException(String message, Throwable cause) { super(message,
        cause); }
    public ProcessingException(Throwable cause) { super(cause); }
}
```

H.15   processor/Processor.java

```java
package com.verictas.pos.simulator.processor;

import com.verictas.pos.simulator.Object;
import com.verictas.pos.simulator.Simulator;
import com.verictas.pos.simulator.SimulatorConfig;
import com.verictas.pos.simulator.dataWriter.AOPDataWriter;
import com.verictas.pos.simulator.dataWriter.DataWriter;
import com.verictas.pos.simulator.dataWriter.PosDataWriter;
import com.verictas.pos.simulator.dataWriter.WritingException;
import com.verictas.pos.simulator.mathUtils.AOP;
import com.verictas.pos.simulator.mathUtils.AU;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.TreeMap;

public class Processor {
    private PosDataWriter writer;
```

```java
private AOPDataWriter aopWriter;
public HashMap<String, Object> initialObjectValues = new HashMap<>();
public HashMap<String, ObjectProcessor> objects = new HashMap<>();
public HashMap<String, TreeMap<Integer, Double>> arguments = new HashMap<>();

public Processor(Object[] objects) throws ProcessingException,
    WritingException {
    /**
     * Initialize DataWriter
     */
    this.writer = new PosDataWriter();
    this.aopWriter = new AOPDataWriter();

    /**
     * Store the initial values of all the objects in memory (and to a file)
         for later use
     */
    this.initialObjectValues = objectArrayToHashMap(objects);

    // Write initial values to file
    this.writePos(initialObjectValues);

    /**
     * Create the object processing array
     */
    for (Object object : initialObjectValues.values()) {
        this.objects.put(object.name, new ObjectProcessor());
        this.objects.get(object.name).setStartingPosition(object.position);
    }
}

public void process(Object[] objectArray) throws ProcessingException,
    WritingException {
    HashMap<String, Object> objects = objectArrayToHashMap(objectArray);

    /**
     * Only do the processing for the asked planet(s)
     */
    for(String objectName : SimulatorConfig.objectNames) {
        ObjectProcessor object = this.objects.get(objectName);

        object.setObjectData(objects.get(objectName));
        object.setReferenceObjectData(objects.get(SimulatorConfig.sunName));
        object.processHistory();

        // Check if the object has gone round last round

        boolean round = object.processRoundCheck();
        if (round) {
            // Process the nodes
            object.processNodes();

            // ECHO:: Object has gone full circle last round!
```

```java
System.out.println("\n\n==============␣ROTATION␣DATA:␣" +
    objectName.toUpperCase() + ",␣ROUND␣" + (Simulator.round - 1) + "␣
    =============");

if (SimulatorConfig.outputUnit.equals("AU")) {
    if (object.ascendingNode != null) {
        System.out.println("Ascending␣node␣(AU):␣" +
            AU.convertFromMeter(object.ascendingNode));
    } else {
        System.out.println("WARNING::␣Ascending␣node␣not␣found.");
    }

    if (object.descendingNode != null) {
        System.out.println("Descending␣node␣(AU):␣" +
            AU.convertFromMeter(object.descendingNode) + "\n");
    } else {
        System.out.println("WARNING::␣Descending␣node␣not␣found.\n");
    }

    System.out.println("Distance␣from␣(the)␣" +
        SimulatorConfig.sunName + "␣during␣apastron␣in␣km:␣" +
        object.aphelionDistance / 1000 + "\n");
    System.out.println("Distance␣from␣(the)␣" +
        SimulatorConfig.sunName + "␣during␣periastron␣in␣km:␣" +
        object.perihelionDistance / 1000 + "\n");
} else {
    if (object.ascendingNode != null) {
        System.out.println("Ascending␣node␣(m):␣" +
            object.ascendingNode);
    } else {
        System.out.println("WARNING::␣Ascending␣node␣not␣found.");
    }

    if (object.descendingNode != null) {
        System.out.println("Descending␣node␣(m):␣" +
            object.descendingNode + "\n");
    } else {
        System.out.println("WARNING::␣Descending␣node␣not␣found.\n");
    }
    System.out.println("Distance␣from␣(the)␣" +
        SimulatorConfig.sunName + "␣during␣apastron␣in␣km:␣" +
        object.aphelionDistance / 1000);
    System.out.println("Distance␣from␣(the)␣" +
        SimulatorConfig.sunName + "␣during␣periastron␣in␣km:␣" +
        object.perihelionDistance / 1000 + "\n");
}

if (object.ascendingNode != null) {
    System.out.println("Argument␣of␣periapsis␣(radians):␣" +
        AOP.calculate(object.ascendingNode, object.perihelion,
        object.aphelion));

    if (object.checkNodes()) {
```

```java
                    // Add the node to the list
                    if (arguments.get(objectName) == null) {
                        // If not defined
                        TreeMap<Integer, Double> agmnts = new TreeMap<>();
                        arguments.put(objectName, agmnts);
                    }


                    arguments.get(objectName).put(Simulator.round,
                        AOP.calculate(object.ascendingNode, object.perihelion,
                        object.aphelion));
                }

            } else {
                System.out.println("ERROR::␣Can't␣calculate␣the␣argument␣of␣
                    periapsis␣because␣the␣ascending␣node␣is␣missing.");
            }

            System.out.println("===================================================

            object.reset();

            // Reset starting position
            this.objects.get(objectName).setStartingPosition(objects.get(objectName).positio
        }

        // Process values for this round
        object.processAphelionAndPerihelion();
        object.calculateTops();

        this.objects.put(objectName, object);
    }

    this.writePos(objects);
}

private void writePos(HashMap<String, Object> objects) throws
    ProcessingException, WritingException {
    if (SimulatorConfig.skipUnnecessary) {
        for (String name : SimulatorConfig.objectNames) {
            this.writer.write(objects.get(name));
        }
    } else {
        for (Object object : objects.values()) {
            this.writer.write(object);
        }
    }
}

private HashMap<String, Object> objectArrayToHashMap(Object[] objects) {
    // Create the return map
    HashMap<String, Object> objectMap = new HashMap<>();
```

```java
        for(int i = 0; i < objects.length; i++) {
            objectMap.put(objects[i].name, objects[i]);
        }

        return objectMap;
    }

    public void close() throws ProcessingException {
        try {
            this.writer.save();
            System.out.println("");
            for(String objectName : SimulatorConfig.objectNames) {
                TreeMap<Integer, Double> arguments = this.arguments.get(objectName);

                this.aopWriter.write(objectName, arguments);

                double score = 0;

                Double[] empty = new Double[arguments.size()];
                Double[] agmnts = arguments.values().toArray(empty);

                // Calculate score
                for(int i = 1; i < agmnts.length - 1; i++) {
                    score = score + Math.abs(agmnts[i-1] - agmnts[i]);
                }

                System.out.println("SCORE␣(" + objectName + "):␣" + score);

                // CALCULATE AVERAGE
                double sum = 0;
                for (int i = 0; i < agmnts.length; i++){
                    sum = sum + agmnts[i];
                }
                // calculate average
                double average = sum / agmnts.length;

                System.out.println("AVERAGE␣(" + objectName + ")␣(degrees):␣" +
                    Math.toDegrees(average));
                System.out.println("");
            }

            this.aopWriter.save();
        } catch(WritingException e) {
            throw new ProcessingException("An␣error␣occurred␣during␣creation␣of␣the␣
                file␣writer:␣" + e.toString());
        }
    }
}
```

Appendix I

The Full Code of the New Simulator

I.1 Main.java

```java
package com.verictas.pos.simulator;

import com.verictas.pos.simulator.mathUtils.AU;

import javax.vecmath.*;

public class Main {
    /**
     * PLANETARY ORBIT SIMULATOR
     * Data Simulation Tool
     *
     * Programmed for the PWS "Planeet Negen" for the Stedelijk Gymnasium Nijmegen,
     *   the Netherlands.
     *
     * =================================
     *
     * The MIT License (MIT)
     * Copyright (c) 2016 Christiaan Goossens (Verictas) & Daniel Boutros
     *
     * The full license is included in the git repository as LICENSE.md
     */

    public static int version = 2;

    public static void main(String[] args) {
        /**
         * Object definitions
         */

        /**
         * Definitions for the ecliptic plane (by 1st of january 2016)
         */
        Object sun = new Object("Sun", 1.988544E30, AU.convertToMeter(new
            Vector3d(3.737881713150281E-03,1.402397586692506E-03,-1.612700291840256E-04)),
            AU.convertToMetersPerSecond(new
            Vector3d(8.619338996535534E-07,6.895607793642275E-06,-2.794074909231784E-08)));
        Object earth = new Object("Earth", 5.97219E24, AU.convertToMeter(new
            Vector3d(-1.630229002588497E-01,9.704723344534316E-01,-1.955367328932975E-04)),
            AU.convertToMetersPerSecond(new
            Vector3d(-1.723383356491747E-02,-2.969134550063944E-03,-4.433758674928828E-07)));
```

```java
Object moon = new Object("The_Moon", 734.9E20, AU.convertToMeter(new
    Vector3d(-1.657103868749121E-01,9.706382026425473E-01,-1.879812512691582E-04)),
    AU.convertToMetersPerSecond(new
    Vector3d(-1.728100931961937E-02,-3.525371122447976E-03,4.909148618073602E-05)));
Object jupiter = new Object("Jupiter", 1898.13E24, AU.convertToMeter(new
    Vector3d(-5.172279968303672E+00,1.591564562098799E+00,1.090553487095606E-01)),
    AU.convertToMetersPerSecond(new
    Vector3d(-2.306423668033420E-03,-6.856869314900905E-03,8.012916249248967E-05)));
Object saturn = new Object("Saturn", 5.68319E26, AU.convertToMeter(new
    Vector3d(-3.710637850378867E+00,-9.289569433157130E+00,3.091990731378936E-01)),
    AU.convertToMetersPerSecond(new
    Vector3d(4.874750391005278E-03,-2.086615906689840E-03,-1.574898601194673E-04)));
Object venus = new Object("Venus", 48.685E23, AU.convertToMeter(new
    Vector3d(-7.130901319004951E-01,-5.719763212192740E-02,4.040076577877051E-02)),
    AU.convertToMetersPerSecond(new
    Vector3d(1.525993024372452E-03,-2.024175581604569E-02,-3.656582385749146E-04)));
Object mars = new Object("Mars", 6.4185E23, AU.convertToMeter(new
    Vector3d(-1.644664047074283E+00,1.714211195991345E-01,4.385749324150048E-02)),
    AU.convertFromMetersPerSecond(new Vector3d(-9.128062787682906E-04,
    -1.271783289037382E-02, -2.442517367300464E-04)));
Object pluto = new Object("Pluto", 1.307E22, AU.convertToMeter(new
    Vector3d(8.535178336776600E+00,-3.187687983153820E+01,9.421570822362236E-01)),
    AU.convertFromMetersPerSecond(new Vector3d(3.105916866228581E-03,
    1.759704223757070E-04, -9.146208184741589E-04)));
Object neptune = new Object("Neptune", 102.41E24, AU.convertToMeter(new
    Vector3d(2.795458622849629E+01,-1.077602237438394E+01,-4.223299945454949E-01)),
    AU.convertFromMetersPerSecond(new Vector3d(1.108107308612818E-03,
    2.948021656576779E-03, -8.584675894389943E-05)));
Object uranus = new Object("Uranus", 86.8103E24, AU.convertToMeter(new
    Vector3d(1.887206485673029E+01,6.554830107743496E+00,-2.201473388797619E-01)),
    AU.convertFromMetersPerSecond(new Vector3d(-1.319173006464416E-03,
    3.532006412470987E-03, 3.002475806591822E-05)));
Object charon = new Object("Charon", 1.53E21, AU.convertToMeter(new
    Vector3d(8.535206843097511E+00,-3.187692375327401E+01,9.420370068039806E-01)),
    AU.convertFromMetersPerSecond(new Vector3d(3.015458707073605E-03,
    8.495285732817140E-05, -9.028237165874783E-04)));

// PWS Objects
Object object1 = new Object("Sedna", 4E21, AU.convertToMeter(new
    Vector3d(4.831201219703945E+01, 6.863113643822504E+01,
    -1.773001247239095E+01)), AU.convertToMetersPerSecond(new
    Vector3d(-2.401309021644802E-03, 7.269559406640982E-04,
    1.704114106899654E-04)));
Object object2 = new Object("2012_VP113", 2.7E18, AU.convertToMeter(new
    Vector3d(5.074554081273273E+01, 6.194684521116067E+01,
    -2.303377758579428E+01)), AU.convertToMetersPerSecond(new
    Vector3d(-1.390042223661063E-03, 1.919356165611094E-03,
    6.083057470436023E-04)));
Object object3 = new Object("2004_VN112", 0, AU.convertToMeter(new
    Vector3d(3.338469440683407E+01, 3.296760926256486E+01,
    -8.176834813898699E+00)), AU.convertToMetersPerSecond(new
    Vector3d(-1.830443771273609E-03, 2.551493797427650E-03,
    1.295080364913495E-03)));
```

```java
        Object object4 = new Object("2007⎵TG442", 0, AU.convertToMeter(new
            Vector3d(-2.216102118938070E+00, -5.957656766688118E-01,
            -9.228532887388547E-03)), AU.convertToMetersPerSecond(new
            Vector3d(1.973707536998759E-03, -1.106231446142322E-02,
            -1.188438173809993E-04)));
        Object object5 = new Object("2013⎵RF98", 0, AU.convertToMeter(new
            Vector3d(2.809064890818173E+01, 2.117251775628629E+01,
            -1.015547278525787E+01)), AU.convertToMetersPerSecond(new
            Vector3d(-1.408524658517317E-03, 3.354634129283988E-03,
            1.461376116722572E-03)));
        Object object6 = new Object("2010⎵GB174", 0, AU.convertToMeter(new
            Vector3d(-6.661904379651325E+01, -8.411238128232725E+00,
            2.212233193483758E+01)), AU.convertFromMetersPerSecond(new
            Vector3d(-9.610782795963537E-04, -2.406268777135870E-03,
            9.081217152229448E-04)));


        /**
         * Object listing
         */

        Object[] objects = {}; // Fill in the objects to be simulated


        /**
         * Run the simulator for the specified objects
         */
        Simulator.run(objects);
    }
}
```

I.2   Node.java

```java
package com.verictas.pos.simulator;

import javax.vecmath.Vector3d;

/**
 * Storage object for storing nodes on the graph
 */
public class Node extends Vector3d {
    public int round;

    /**
     * Constructor for casting
     * @param vector
     */
    public Node(Vector3d vector) {
        this.set(vector);
    }

    /**
```

```
     * Constructor for empty creation
     */
    public Node() {
        this.set(new Vector3d(0,0,0));
    }

    /**
     * Sets the stored round associated with this node
     * (It will most likely be the round when this node is reached)
     * @param round
     */
    public void setRound(int round) {
        this.round = round;
    }

    public boolean empty() {
        if (this.getX() == 0 && this.getY() == 0 && this.getZ() == 0) {
            return true;
        }
        return false;
    }
}
```

I.3 Object.java

```
package com.verictas.pos.simulator;
import javax.vecmath.*;
import java.lang.*;

public class Object {
    public double mass;
    public Vector3d position;
    public Vector3d speed;

    public Vector3d acceleration;
    public Vector3d oldAcceleration;

    public String name;

    private double gravitationalConstant = 6.67384E-11;

    /**
     * Constructs an object
     * @param mass The mass of the object
     * @param position The position vector of the object
     * @param speed The speed vector of the object
     */
    public Object(String name, double mass, Vector3d position, Vector3d speed) {
        this.name = name;
        this.mass = mass;
        this.position = position;
```

```java
        this.speed = speed;
        this.oldAcceleration = new Vector3d(0,0,0);
        this.acceleration = new Vector3d(0,0,0);
    }


    /**
     * Sets the speed vector of an object
     * @param speed Current speed vector
     */
    public void setSpeed(Vector3d speed) {
        this.speed = speed;
    }
    public void setSpeed(double[] speed) {
        this.speed = new Vector3d(speed[0], speed[1], speed[2]);
    }


    /**
     * Gets the speed into a double[3]
     * @return double[3]
     */
    public double[] getSpeed() {
        double[] v = new double[3];
        this.speed.get(v);
        return v;
    }


    /**
     * Sets the position vector of an object.
     * @param position Current position vector
     */
    public void setPosition(Vector3d position) {
        this.position = position;
    }
    public void setPosition(double[] position) {
        this.position = new Vector3d(position[0], position[1], position[2]);
    }


    /**
     * Gets the position into a double[3]
     * @return double[3]
     */
    public double[] getPosition() {
        double[] r = new double[3];
        this.position.get(r);
        return r;
    }


    /**
     * Sets the acceleration vector of an object
     * @param acceleration Current acceleration vector
     */
    public void setAcceleration(Vector3d acceleration) { this.acceleration =
        acceleration; }
```

```java
    public void setAcceleration(double[] acceleration) {
        this.acceleration = new Vector3d(acceleration[0], acceleration[1],
            acceleration[2]);
    }

    /**
     * Gets the acceleration into a double[3]
     * @return double[3]
     */
    public double[] getAcceleration() {
        double[] a = new double[3];
        this.acceleration.get(a);
        return a;
    }

    /**
     * Sets the acceleration vector of an object
     * @param acceleration Current acceleration vector
     */
    public void setOldAcceleration(Vector3d acceleration) { this.acceleration =
        acceleration; }
    public void setOldAcceleration(double[] acceleration) {
        this.oldAcceleration = new Vector3d(acceleration[0], acceleration[1],
            acceleration[2]);
    }

    /**
     * Gets the acceleration into a double[3]
     * @return double[3]
     */
    public double[] getOldAcceleration() {
        double[] a = new double[3];
        this.oldAcceleration.get(a);
        return a;
    }

    /**
     * Changes an object into readable form
     * @return String
     */
    public String toString() {
        return "Mass:␣" + this.mass + "␣&␣Position:␣" + this.position + "␣&␣Speed:␣
            " + this.speed;
    }

    /**
     * Calculates the force of the passed object on the current object.
     * @param secondObject The passed object
     * @return Vector3d The gravitational force
     */
    public Vector3d getForceOnObject(Object secondObject) {
        double scale = gravitationalConstant * ((this.mass * secondObject.mass) /
            Math.pow(getDistance(secondObject).length(), 3.0));
```

```java
        Vector3d force = getDistance(secondObject);
        force.scale(scale);
        return force;
    }


    /**
     * Get the vector distance between the current position vector and the position
     *   vector of the passed object.
     * @param secondObject The passed object.
     * @return Vector3d The distance vector
     */
    public Vector3d getDistance(Object secondObject) {
        Vector3d distance = new Vector3d(0,0,0); // Empty
        distance.sub(this.position, secondObject.position);
        return distance;
    }


    /**
     * Get the vector distance between the current position vector and a given
     *   position.
     * @param position The position vector you want the distance to.
     * @return Vector3d The distance vector
     */
    public Vector3d getDistance(Vector3d position) {
        Vector3d distance = new Vector3d(0,0,0); // Empty
        distance.sub(this.position, position);
        return distance;
    }


    /**
     * Updates the position based on dt
     * @param dt The difference in time
     */
    public void updatePosition(double dt) {
        // Write the vectors to double[3]
        double[] r = this.getPosition();
        double[] v = this.getSpeed();
        double[] a = this.getAcceleration();

        for (int i = 0; i != 3; i++){
            double dt2 = dt * dt;
            r[i] += v[i] * dt + 0.5 * a[i] * dt2;
        }

        // Write the doubles into the vectors to save them
        setPosition(r);
        setSpeed(v);
        setAcceleration(a);
    }


    /**
     * Updates the speed based on dt
     * @param dt The difference in speed
     */
```

```java
     */
    public void updateSpeed(double dt) {
        // Write the vectors to double[3]
        double[] v = this.getSpeed();
        double[] a = this.getAcceleration();
        double[] aold = this.getOldAcceleration();

        for (int i = 0; i != 3; i++){
            v[i] += 0.5 * dt *(a[i] + aold[i]);
        }

        setSpeed(v);
        setAcceleration(a);
        setOldAcceleration(aold);
    }

    /**
     * Updates the acceleration based on dt
     */
    public void updateAcceleration() {
        this.oldAcceleration = this.acceleration;
    }

    /**
     * Enacts a certain force on the object
     * @param force The force in N.
     */
    public void enactForceOnObject(Vector3d force) {
        double factor = 1/this.mass;
        Vector3d acceleration = force;
        acceleration.scale(factor);
        this.acceleration = acceleration;
    }
}
```

I.4 Simulator.java

```java
package com.verictas.pos.simulator;
import javax.vecmath.*;

import com.verictas.pos.simulator.dataWriter.WritingException;
import com.verictas.pos.simulator.mathUtils.Vector3dMatrix;
import com.verictas.pos.simulator.processor.ProcessingException;
import com.verictas.pos.simulator.processor.Processor;

public class Simulator {
    public static int round = 0; // Stores an global integer value with the
        current round (as a timestamp)

    /**
     * Run method for the Simulator
```

```java
 * @param objects
 */
public static void run(Object[] objects) {

    /**
     * Get variables from the config
     */

    int rounds = SimulatorConfig.rounds;
    double time = SimulatorConfig.time;

    /**
     * Log a debug message to the console to signal the simulation has started
     */
    System.out.println("========== Simulation Started ==========\n");

    /**
     * Create a time to measure runtime
     */
    long startTime = System.currentTimeMillis();

    /**
     * Define the forces matrix and the DataWriter
     */
    Vector3dMatrix matrix = new Vector3dMatrix(objects.length, objects.length);

    try {
        Processor processor = new Processor(objects);

        /**
         * Start the leap frog integration!
         */

        accelerate(objects, matrix);

        /**
         * Start the rounds loop
         */
        for(int t = 0; t != rounds; t++) {
            // Set round
            Simulator.round++;

            /**
             * The round has started
             */
            if(SimulatorConfig.logConsole) {
                if(SimulatorConfig.skipConsole == -1 || Simulator.round %
                    SimulatorConfig.skipConsole == 0 || Simulator.round == 1) {
                    System.out.println("Round " + Simulator.round + " started!");
                }
            }

            for(int i = 0; i < objects.length; i++) {
```

```java
                objects[i].updatePosition(time);
                objects[i].updateAcceleration();
            }

            accelerate(objects, matrix);

            for(int i = 0; i < objects.length; i++) {
                objects[i].updateSpeed(time);
            }

            /**
             * Do the processing on the objects
             */
            processor.process(objects);

            /**
             * The round has ended
             */
        }

        /**
         * Log that the simulation has finished and save info to file
         */
        processor.close();
        System.out.println("=========_Simulation_Finished_=========");

        /**
         * Display information about the program runtime
         */
        long stopTime = System.currentTimeMillis();
        System.out.println("Simulation_took:_" + (stopTime - startTime) + "ms");
    } catch(ProcessingException e) {
        System.out.println("\nERROR::_Processing_failed.");
        e.printStackTrace();
    } catch(WritingException e) {
        System.out.println("\nERROR::_Writing_to_file_failed.");
        e.printStackTrace();
    }
}

/**
 * Accelerates the given objects, puts the results in the given matrix and
    enacts forces
 * @param objects
 * @param matrix
 */
private static void accelerate(Object[] objects, Vector3dMatrix matrix) {
    // Loop
    for(int i = 0; i < objects.length; i++) {
        /**
         * For every object: calculate the force upon it.
         */
```

```java
            // Reset acceleration
            objects[i].setAcceleration(new Vector3d(0, 0, 0));

            for (int o = 0; o < objects.length; o++) {
                /**
                 * Loop through all other objects
                 */
                if (o == i) {
                    break;
                }

                Vector3d force = objects[i].getForceOnObject(objects[o]);
                matrix.setPosition(force, i, o);

                /**
                 * Also put in the opposite force
                 */
                force.scale(-1);
                matrix.setPosition(force, o, i);
            }
        }

        for(int i = 0; i < objects.length; i++) {
            /**
             * Progress forces on the object
             */
            Vector3d forceOnI = matrix.getColumnTotal(i);
            objects[i].enactForceOnObject(forceOnI);
        }
    }
}
```

I.5   SimulatorConfig.java (empty)

```java
package com.verictas.pos.simulator;

public class SimulatorConfig {

    /**
     * (Example) Settings for the EARTH
     * Rounds: 1051896 * (amount of years to run)
     * Time: 30
     * Mod arg: 1051896 (1 Earth year)
     */

    /**
     * (Example) Settings for SEDNA
     * Rounds: 184000000 (approx. 1 million years)
     * Time: 172800 (2 days)
     * Modulo argument: 2101968 (1 Sedna year)
     */
```

```java
    /**
     * (Example) Settings for 2012 VP113
     * Rounds: 184000000 (approx. 1 million years)
     * Time: 172800 (2 days)
     * Modulo argument: 788923 (1 2012VP113 year)
     */

    /**
     * Time settings
     */

    public static int rounds = 0; // Amount of rounds to run the simulator for
    public static double time = 0; // Time steps in seconds

    /**
     * Object settings
     */

    public static String sunName = "Sun"; // The name of the sun to calculate
        values TO
    public static String[] objectNames = {}; // The name of the object(s) your
        want to calculate the values OF

    /**
     * Output preferences
     */

    public static String outputUnit = "AU"; // Preferred output unit preference
        (AU => AU/day, m => m/s)
    public static int outputNumbers = 0; // Preferred way of outputting numbers:
        (0 => comma for decimals, dot in large numbers OR 1 => comma for large
        numbers, dot with decimals)
    public static int skipLines = 1; // Set the skipLines integer to skip lines
        (for example: every 5th line is written) in the output file (for smaller
        files), if this is set to 1, it has no effect and all lines will be written.
    public static boolean skipUnnecessary = true; // Skip the unnecessary objects
        in the export

    /**
     * Console settings
     */
    public static boolean logConsole = true;
    public static int skipConsole = 1;

    /**
     * Processor settings
     */
    public static boolean autoModulo = true;
    public static int moduloArgument = 1;
}
```

I.6 dataWriter/AOPDataWriter.java

```java
package com.verictas.pos.simulator.dataWriter;

import java.util.*;

public class AOPDataWriter extends DataWriter {
    public AOPDataWriter() throws WritingException {
        super("arguments");
        try {

            /**
             * Write the lines with information about the columns
             */

            this.writer.write("OBJECT" + DELIMITER + "ROUND" + DELIMITER + "ARGUMENT␣
                (RAD)" + NEW_LINE);
            this.counter++;
        } catch (Exception e) {
            e.printStackTrace();
            throw new WritingException("An␣error␣occurred␣while␣writing␣to␣the␣
                file!");
        }
    }

    public void write(String object, TreeMap<Integer, Double> arguments) throws
        WritingException {
        try {
            for (Map.Entry<Integer, Double> entry : arguments.entrySet()) {
                Integer key = entry.getKey();
                Double value = entry.getValue();
                this.writer.append(object + DELIMITER + key + DELIMITER +
                    decimalFormatter(value) + NEW_LINE);
                this.counter++;
            }
        } catch (Exception e) {
            e.printStackTrace();
            throw new WritingException("An␣error␣occurred␣while␣writing␣to␣the␣
                file!");
        }
    }
}
```

I.7 dataWriter/DataWriter.java

```java
package com.verictas.pos.simulator.dataWriter;

import com.verictas.pos.simulator.Main;
import com.verictas.pos.simulator.SimulatorConfig;
```

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.text.*;
import java.util.Date;

public class DataWriter {
    protected FileWriter writer = null;

    /**
     * Set global variables, such as the delimiter and the new line character
     */
    protected static final String DELIMITER = "\t";
    protected static final String NEW_LINE = "\n";

    protected int counter = 0;

    /**
     * Decimal formatter
     */

    public DecimalFormat formatter = new DecimalFormat();

    /**
     * Constructor
     * @throws WritingException
     */
    public DataWriter(String filenameAppendix) throws WritingException {

        /**
         * Prepare the locale
         */

        try {
            /**
             * Define the save path
             */
            String directory = System.getProperty("user.home") + File.separator +
                "simulatorExports";
            File directoryPath = new File(directory);

            String path = directory + File.separator + "v" + Main.version + "-" +
                getCurrentTimeStamp() + "-" + filenameAppendix + ".txt";
            System.out.println("WRITING␣DATA␣TO:␣" + path);

            /**
             * Check if the saving directory exists to prevent IOException
             */
            if (!directoryPath.exists()) {
                directoryPath.mkdirs();
            }
```

```java
        /**
         * Open a file to write to and write the header
         */
        this.writer = new FileWriter(path);

        /**
         * Configure the decimal formatter
         */
        DecimalFormatSymbols symbols = new DecimalFormatSymbols();
        if (SimulatorConfig.outputNumbers == 0) {
            symbols.setDecimalSeparator(',');
            symbols.setGroupingSeparator('.');
        } else {
            symbols.setDecimalSeparator('.');
            symbols.setGroupingSeparator(',');
        }
        this.formatter.setDecimalFormatSymbols(symbols);
        this.formatter.setMinimumFractionDigits(0);
        this.formatter.setMaximumFractionDigits(25);
    } catch(IOException e) {
        throw new WritingException("The destination file couldn't be created.");
    } catch(Exception e) {
        throw new WritingException("Some unknown error occurred while writing to
            the file!");
    }
}


/**
 * Writes a string to the file
 * @param string
 * @throws WritingException
 */
public void write(String string) throws WritingException {
    if (this.writer == null) {
        throw new WritingException("The writer isn't defined yet");
    } else {
        try {
            if (this.counter % SimulatorConfig.skipLines == 0) {
                this.writer.append(string);
            }
            this.counter++;
        } catch (Exception e) {
            e.printStackTrace();
            throw new WritingException("An error occurred while writing to the
                file!");
        }
    }
}

protected String decimalFormatter(double input) {
    return this.formatter.format(input);
}
```

```java
    /**
     * Saves the file to disk
     * @throws WritingException
     */
    public void save() throws WritingException {
        if (this.writer == null) {
            throw new WritingException("The writer isn't defined yet");
        } else {
            try {
                this.writer.flush();
                this.writer.close();
            } catch (IOException e) {
                throw new WritingException("Whoop! Save error!");
            }
        }
    }

    /**
     * Gets the current line count
     * @return int
     */
    public int getLines() {
        return this.counter;
    }

    /**
     * Gets the current filestamp for file naming
     * @return String
     */
    private String getCurrentTimeStamp() {
        return new SimpleDateFormat("yyyy-MM-dd_HH-mm-ss").format(new Date());
    }
}
```

I.8 dataWriter/PosDataWriter.java

```java
package com.verictas.pos.simulator.dataWriter;

import com.verictas.pos.simulator.Object;
import com.verictas.pos.simulator.SimulatorConfig;
import com.verictas.pos.simulator.mathUtils.AU;

import javax.vecmath.Vector3d;

public class PosDataWriter extends DataWriter {
    public PosDataWriter() throws WritingException {
        super("position");
        try {

            /**
             * Write the lines with information about the columns
```

```java
          */

         if (SimulatorConfig.outputUnit.equals("AU")) {
             this.writer.write("Object" + DELIMITER + "X␣(AU)" + DELIMITER + "Y␣
                 (AU)" + DELIMITER + "Z␣(AU)" + DELIMITER + "VX␣(AU/day)" +
                 DELIMITER + "VY␣(AU/day)" + DELIMITER + "VZ␣(AU/day)" + NEW_LINE);
         } else {
             this.writer.write("Object" + DELIMITER + "X␣(m)" + DELIMITER + "Y␣
                 (m)" + DELIMITER + "Z␣(m)" + DELIMITER + "VX␣(m/s)" + DELIMITER +
                 "VY␣(m/s)" + DELIMITER + "VZ␣(m/s)" + NEW_LINE);
         }

         this.counter++;
     } catch (Exception e) {
         e.printStackTrace();
         throw new WritingException("An␣error␣occurred␣while␣writing␣to␣the␣
             file!");
     }
}


/**
 *
 * @param object The object you want to write data about
 * @throws WritingException
 */
public void write(Object object) throws WritingException {
    String id = object.name;
    Vector3d position = object.position;
    Vector3d speed = object.speed;
    Vector3d AUposition = AU.convertFromMeter(position);
    Vector3d AUspeed = AU.convertFromMetersPerSecond(speed);

    if (this.writer == null) {
        throw new WritingException("The␣writer␣isn't␣defined␣yet");
    } else {
        try {
            if (this.counter % SimulatorConfig.skipLines == 0) {
                if (SimulatorConfig.outputUnit.equals("AU")) {
                    this.writer.append(id + DELIMITER +
                        decimalFormatter(AUposition.getX()) + DELIMITER +
                        decimalFormatter(AUposition.getY()) + DELIMITER +
                        decimalFormatter(AUposition.getZ()) + DELIMITER +
                        decimalFormatter(AUspeed.getX()) + DELIMITER +
                        decimalFormatter(AUspeed.getY()) + DELIMITER +
                        decimalFormatter(AUspeed.getZ()) + NEW_LINE);
                } else {
                    this.writer.append(id + DELIMITER +
                        decimalFormatter(position.getX()) + DELIMITER +
                        decimalFormatter(position.getY()) + DELIMITER +
                        decimalFormatter(position.getZ()) + DELIMITER +
                        decimalFormatter(speed.getX()) + DELIMITER +
                        decimalFormatter(speed.getY()) + DELIMITER +
                        decimalFormatter(speed.getZ()) + NEW_LINE);
```

```
                    }
                }
                this.counter++;
            } catch (Exception e) {
                e.printStackTrace();
                throw new WritingException("An error occurred while writing to the
                    file!");
            }
        }
    }
}
```

I.9   dataWriter/WritingException.java

```
package com.verictas.pos.simulator.dataWriter;

public class WritingException extends Exception {
    public WritingException() { super(); }
    public WritingException(String message) { super(message); }
    public WritingException(String message, Throwable cause) { super(message,
        cause); }
    public WritingException(Throwable cause) { super(cause); }
}
```

I.10   mathUtils/AOP.java

```
package com.verictas.pos.simulator.mathUtils;
import javax.vecmath.Vector3d;

public class AOP {
    public static double calculate(Vector3d pos, Vector3d speed) {
        // ORBITAL MOMENTUM VECTOR
        Vector3d orbitalMomentum = new Vector3d(0,0,0);
        orbitalMomentum.cross(speed, pos);

        // ACCENDING NODE VECTOR
        Vector3d ascendingNode = new Vector3d(0,0,0);
        ascendingNode.cross(new Vector3d(0,0,1), orbitalMomentum);

        // ECCENTRICITY VECTOR
        double mu = 1.32712440018E20;

        Vector3d upCross = new Vector3d(0,0,0);
        upCross.cross(speed, orbitalMomentum);
        upCross.scale(1/mu);
        double posLength = pos.length();
        Vector3d rightPos = new Vector3d(pos);
        rightPos.scale(1/posLength);
```

```java
        Vector3d eccentricity = new Vector3d(0,0,0);
        eccentricity.sub(upCross, rightPos);

        // AOP
        double aop;
        if (eccentricity.getZ() < 0) {
            aop = (2 * Math.PI) - ascendingNode.angle(eccentricity);
        } else {
            aop = ascendingNode.angle(eccentricity);
        }

        return aop;
    }
}
```

I.11   mathUtils/AU.java

```java
package com.verictas.pos.simulator.mathUtils;

import javax.vecmath.Vector3d;

public class AU {
    /**
     * Helper class for working with astronomical units
     */

    /**
     * Converts AU to meters
     * @param input Vector3d with values in AU
     * @return Vector3d with values in meter
     */
    public static Vector3d convertToMeter(Vector3d input) {
        Vector3d output = new Vector3d(input);

        // Convert AU to m by NASA
        output.scale(149597870.700); // Number to large when multiplied with 1000
        output.scale(1000);

        return output;
    }

    /**
     * Converts AU/day to m/s
     * @param input Vector3d with values in AU/day
     * @return Vector3d with values in m/s
     */
    public static Vector3d convertToMetersPerSecond(Vector3d input) {
        Vector3d output = new Vector3d(input);

        // 1 AU/day to M/s
```

```java
        output.scale(1731456.84);

        return output;
    }

    /**
     * Converts meters to AU for data collection
     * @param input Vector3d with values in meters
     * @return Vector3d with values in AU
     */
    public static Vector3d convertFromMeter(Vector3d input) {
        Vector3d output = new Vector3d(input);

        // Convert m to AU by NASA
        output.scale(6.6845871E-12);

        return output;
    }

    public static double convertFromMeter(double input) {
        return input * 6.6845871E-12;
    }

    /**
     * Converts m/s to AU/day for data collection
     * @param input Vector3d with values in m/s
     * @return Vector3d with values in AU/day
     */
    public static Vector3d convertFromMetersPerSecond(Vector3d input) {
        Vector3d output = new Vector3d(input);

        // Convert seconds to days by NASA
        output.scale(5.77548327E-7);

        return output;
    }
}
```

I.12 mathUtils/Vector3dMatrix.java

```java
package com.verictas.pos.simulator.mathUtils;

import javax.vecmath.GMatrix;
import javax.vecmath.Vector3d;

public class Vector3dMatrix extends GMatrix {
    /**
     * Creates a new matrix with some helper functions for use with Vector3f. The
     *     created matrix will be empty.
     * @param n The number of rows.
     * @param m The number of columns.
```

```java
 */
public Vector3dMatrix(int n, int m) {
    // Change the size to suit Vector3d
    super(n, m * 3);
    this.setZero();
}

/**
 * Set the size of the matrix in the amount of vectors (e.g. a 1 x 3 vector
     matrix gets converted to a 1 x 9 storage matrix).
 * @param n The amount of rows
 * @param m The amount of columns expressed in vectors (1 vector = 3 values).
 */
public void setSizeInVectors(int n, int m) {
    this.setSize(n, m * 3);
}

/**
 * Provides a function for putting the matrix into String form for
     visualisation.
 * @return String
 */
public String toString() {
    StringBuffer buffer = new StringBuffer(this.getNumRow() * this.getNumCol()
        * 8);

    for(int n = 0; n < this.getNumRow(); ++n) {
        for(int m = 0; m < this.getNumCol(); ++m) {
            if ((m + 1) == 1 || m % 3 == 0) {
                // If m is 1 or a multiple of 4, begin the bracket.
                buffer.append("(").append(this.getElement(n, m)).append(",␣");
            } else if ((m + 1) % 3 == 0) {
                // If m is a multiple of 3, close the bracket
                buffer.append(this.getElement(n, m)).append(")\t\t");
            } else {
                buffer.append(this.getElement(n, m)).append(",␣");
            }
        }

        buffer.append("\n");
    }

    return buffer.toString();
}

/**
 * Provides a translator from the vector positions (e.g. the second vector
     starts at position 1) to the matrix positions (the second vector starts at
     position 3).
 * @param n The vector positions row
 * @param m The vector positions column
 * @return void
 */
```

```java
    private int[] translatePosition(int n, int m) {
        return new int[]{n, m * 3};
    }

    /**
     * Provides a way to set a vector into a certain position in the matrix
     * @param settable The vector you want to put in the matrix
     * @param n The row to insert into
     * @param m The column to insert into
     */
    public void setPosition(Vector3d settable, int n, int m) {
        int[] position = translatePosition(n, m);
        n = position[0];
        m = position[1];

        this.setElement(n, m, settable.x);
        this.setElement(n, m + 1, settable.y);
        this.setElement(n, m + 2, settable.z);
    }

    /**
     * Provides a way to get a vector from a certain position in the matrix
     * @param n The row to get from
     * @param m The column to get from
     * @return Vector3d The vector in that position
     */
    public Vector3d getPosition(int n, int m) {
        int[] position = translatePosition(n, m);
        n = position[0];
        m = position[1];

        double x = this.getElement(n, m);
        double y = this.getElement(n, m + 1);
        double z = this.getElement(n, m + 2);
        return new Vector3d(x, y, z);
    }

    /**
     * Provides a way to calculate the result vector of a certain row
     * @param row The row to calculate the total of
     * @return Vector3d
     */
    public Vector3d getRowTotal(int row) {
        double[] rowTotal = new double[this.getNumCol()];
        this.getRow(row, rowTotal);

        // Create an empty vector to store the result
        Vector3d resultVector = new Vector3d(0,0,0);

        for(int i = 0; i < this.getNumCol(); i = i + 3) {
            // For every third entry (including 0).
            double x = this.getElement(row, i);
            double y = this.getElement(row, i + 1);
```

```java
            double z = this.getElement(row, i + 2);
            resultVector.add(new Vector3d(x, y, z));
        }

        return resultVector;
    }

    /**
     * Provides a way to calculate the result vector of a certain column
     * @param column The column to calculate the total of
     * @return Vector3d
     */
    public Vector3d getColumnTotal(int column) {
        double[] columnTotal = new double[this.getNumRow()];

        // Translate the column number to the correct vector column
        int[] position = translatePosition(0, column);
        column = position[1];

        this.getColumn(column, columnTotal);

        // Create an empty vector to store the result
        Vector3d resultVector = new Vector3d(0,0,0);

        for(int i = 0; i < this.getNumRow(); i++) {
            // For every entry (including 0).
            double x = this.getElement(i, column);
            double y = this.getElement(i, column + 1);
            double z = this.getElement(i, column + 2);
            resultVector.add(new Vector3d(x, y, z));
        }

        return resultVector;
    }
}
```

I.13 processor/ObjectProcessor.java

```java
package com.verictas.pos.simulator.processor;

import com.verictas.pos.simulator.Node;
import com.verictas.pos.simulator.Object;
import com.verictas.pos.simulator.Simulator;
import com.verictas.pos.simulator.SimulatorConfig;

import javax.vecmath.Vector3d;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class ObjectProcessor {
```

```java
public Node aphelion;
public Node perihelion;
public double aphelionDistance = -1;
public double perihelionDistance = -1;

private Object thisObject;
private Object referenceObject;

private Vector3d startingPosition;
private double lastStartDistance = -1;
private double beforeLastStartDistance = -1;

public void setStartingPosition(Vector3d position) {
    this.startingPosition = position;
}

public void setObjectData(Object object) {
    this.thisObject = object;
}

public void setReferenceObjectData(Object object) {
    this.referenceObject = object;
}

/**
 * Processes the aphelion & perihelion
 */
public void processAphelionAndPerihelion() {
    double sunDistance =
        this.thisObject.getDistance(this.referenceObject).length();

    /**
     * Set the defaults
     */

    if (this.aphelionDistance == -1) {
        this.aphelionDistance = sunDistance;
    }

    if (this.perihelionDistance == -1) {
        this.perihelionDistance = sunDistance;
    }

    /**
     * Check if the aphelion or perihelion should be changed
     */

    if (sunDistance > aphelionDistance) {
        this.aphelion = new Node(this.thisObject.position);
        this.aphelion.setRound(Simulator.round);
        this.aphelionDistance = sunDistance;
    }
```

```java
        if (sunDistance < perihelionDistance) {
            this.perihelion = new Node(this.thisObject.position);
            this.perihelion.setRound(Simulator.round);
            this.perihelionDistance = sunDistance;
        }
    }

    /**
     * Processes the round check
     */
    public boolean processRoundCheck() {
        double startDistance =
            this.thisObject.getDistance(this.startingPosition).length();
        boolean fullRotation = false;

        /**
         * Check if all are set and shuffle!
         */

        if (beforeLastStartDistance != -1 && lastStartDistance != -1) {
            // Ready to go!
            if (beforeLastStartDistance > lastStartDistance && startDistance >
                lastStartDistance) {
                // Last point was the closest to the starting position overall!
                fullRotation = true;
                if (SimulatorConfig.logConsole) {
                    System.out.println("INFO:: Object " + this.thisObject.name + " 
                        has made a full rotation last round.");
                }
            }

            beforeLastStartDistance = lastStartDistance;
            lastStartDistance = startDistance;
        }

        /**
         * Check if 1st distance is set and 2nd isn't set
         */

        if (beforeLastStartDistance != -1 && lastStartDistance == -1) {
            lastStartDistance = startDistance;
        }

        /**
         * Check if the 1st distance isn't set
         */
        if (beforeLastStartDistance == -1) {
            beforeLastStartDistance = startDistance;
        }

        if(fullRotation) {
            return true;
        } else {
```

```java
            return false;
        }
    }

    public void reset() {
        aphelionDistance = -1;
        perihelionDistance = -1;
        lastStartDistance = -1;
        beforeLastStartDistance = -1;
        aphelion = new Node();
        perihelion = new Node();
    }
}
```

I.14   procesor/ProcessingException.java

```java
package com.verictas.pos.simulator.processor;

public class ProcessingException extends Exception {
    public ProcessingException() { super(); }
    public ProcessingException(String message) { super(message); }
    public ProcessingException(String message, Throwable cause) { super(message,
        cause); }
    public ProcessingException(Throwable cause) { super(cause); }
}
```

I.15   processor/Processor.java

```java
package com.verictas.pos.simulator.processor;

import com.verictas.pos.simulator.Object;
import com.verictas.pos.simulator.Simulator;
import com.verictas.pos.simulator.SimulatorConfig;
import com.verictas.pos.simulator.dataWriter.AOPDataWriter;
import com.verictas.pos.simulator.dataWriter.PosDataWriter;
import com.verictas.pos.simulator.dataWriter.WritingException;
import com.verictas.pos.simulator.mathUtils.AOP;
import com.verictas.pos.simulator.mathUtils.AU;

import javax.vecmath.Vector3d;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.TreeMap;

public class Processor {
    private PosDataWriter writer;
    private AOPDataWriter aopWriter;
    public HashMap<String, Object> initialObjectValues = new HashMap<>();
```

```java
public HashMap<String, ObjectProcessor> objects = new HashMap<>();
public HashMap<String, TreeMap<Integer, Double>> arguments = new HashMap<>();

public Processor(Object[] objects) throws ProcessingException,
    WritingException {
    /**
     * Initialize DataWriter
     */
    this.writer = new PosDataWriter();
    this.aopWriter = new AOPDataWriter();

    /**
     * Store the initial values of all the objects in memory (and to a file)
         for later use
     */
    this.initialObjectValues = objectArrayToHashMap(objects);

    // Write initial values to file
    this.writePos(initialObjectValues);

    /**
     * Create the object processing array
     */
    for (Object object : initialObjectValues.values()) {
        this.objects.put(object.name, new ObjectProcessor());
        this.objects.get(object.name).setStartingPosition(object.position);
    }
}

public void process(Object[] objectArray) throws ProcessingException,
    WritingException {
    HashMap<String, Object> objects = objectArrayToHashMap(objectArray);
    this.writePos(objects);


    /**
     * Calculate AOP for specified objects
     */
    for(String objectName : SimulatorConfig.objectNames) {
        // Process the aphelion & perihelion for reference
        ObjectProcessor object = this.objects.get(objectName);

        object.setObjectData(objects.get(objectName));
        object.setReferenceObjectData(objects.get(SimulatorConfig.sunName));

        // Check if the object has gone round last round

        boolean round = object.processRoundCheck();
        if (round) {
            System.out.println("\n\n==============ROTATION DATA: " +
                objectName.toUpperCase() + ", ROUND " + (Simulator.round - 1) + " 
                ============");
            System.out.println("Distance from (the) " + SimulatorConfig.sunName
```

```java
                + "␣during␣apastron␣in␣km:␣" + object.aphelionDistance / 1000);
        System.out.println("Distance␣from␣(the)␣" + SimulatorConfig.sunName
                + "␣during␣apastron␣in␣AU:␣" +
                AU.convertFromMeter(object.aphelionDistance));
        System.out.println("Distance␣from␣(the)␣" + SimulatorConfig.sunName
                + "␣during␣periastron␣in␣km:␣" + object.perihelionDistance /
                1000);
        System.out.println("Distance␣from␣(the)␣" + SimulatorConfig.sunName
                + "␣during␣periastron␣in␣AU:␣" +
                AU.convertFromMeter(object.perihelionDistance));
        System.out.println("==========================================================

        object.reset();

        // Reset starting position
        this.objects.get(objectName).setStartingPosition(objects.get(objectName).positio
    }

    object.processAphelionAndPerihelion();
    this.objects.put(objectName, object);


    /**
     * Calculate AOP
     */
    if (SimulatorConfig.autoModulo && round) {
        if (arguments.get(objectName) == null) {
            // If not defined
            TreeMap<Integer, Double> agmnts = new TreeMap<>();
            arguments.put(objectName, agmnts);
        }

        // Calculate AOP and put it in the array
        Object AOPobject = objects.get(objectName);
        Vector3d pos = new Vector3d(AOPobject.position);
        Vector3d speed = new Vector3d(AOPobject.speed);
        arguments.get(objectName).put(Simulator.round, AOP.calculate(pos,
                speed));

        System.out.println("INFO::␣Last␣rounds␣AOP:␣" + AOP.calculate(pos,
                speed));
    } else if (!SimulatorConfig.autoModulo) {
        if (Simulator.round % SimulatorConfig.moduloArgument == 0) {
            if (arguments.get(objectName) == null) {
                // If not defined
                TreeMap<Integer, Double> agmnts = new TreeMap<>();
                arguments.put(objectName, agmnts);
            }

            // Calculate AOP and put it in the array
            Object AOPobject = objects.get(objectName);
            Vector3d pos = new Vector3d(AOPobject.position);
            Vector3d speed = new Vector3d(AOPobject.speed);
```

```java
                arguments.get(objectName).put(Simulator.round, AOP.calculate(pos,
                    speed));
            }
        }
    }
}

private void writePos(HashMap<String, Object> objects) throws
    ProcessingException, WritingException {
    if (SimulatorConfig.skipUnnecessary) {
        for (String name : SimulatorConfig.objectNames) {
            this.writer.write(objects.get(name));
        }
    } else {
        for (Object object : objects.values()) {
            this.writer.write(object);
        }
    }
}

private HashMap<String, Object> objectArrayToHashMap(Object[] objects) {
    // Create the return map
    HashMap<String, Object> objectMap = new HashMap<>();

    for(int i = 0; i < objects.length; i++) {
        objectMap.put(objects[i].name, objects[i]);
    }

    return objectMap;
}

public void close() throws ProcessingException {
    try {
        this.writer.save();
        System.out.println("");
        for(String objectName : SimulatorConfig.objectNames) {
            TreeMap<Integer, Double> arguments = this.arguments.get(objectName);

            this.aopWriter.write(objectName, arguments);

            double score = 0;

            Double[] empty = new Double[arguments.size()];
            Double[] agmnts = arguments.values().toArray(empty);

            // Calculate score
            for(int i = 1; i < agmnts.length - 1; i++) {
                score = score + Math.abs(agmnts[i-1] - agmnts[i]);
            }

            System.out.println("SCORE␣(" + objectName + "):␣" + score);

            // CALCULATE AVERAGE
```

```java
            double sum = 0;
            for (int i = 0; i < agmnts.length; i++){
                sum = sum + agmnts[i];
            }
            // calculate average
            double average = sum / agmnts.length;

            System.out.println("AVERAGE␣(" + objectName + ")␣(degrees):␣" +
                Math.toDegrees(average));
            System.out.println("");
        }

        this.aopWriter.save();
    } catch(WritingException e) {
        throw new ProcessingException("An␣error␣occurred␣during␣creation␣of␣the␣
            file␣writer:␣" + e.toString());
    }
  }
}
```