Aristotle University of Thessaloniki

MSc Computational Physics

Master Thesis

Orbital Dynamics of Impact Ejecta around 65803 Didymos Binary (AIDA Mission)

Author
Michalis Gaitanas

Supervisors
George Voyatzis
Kleomenis Tsiganis

June 22, 2019

Contents

Abstract

In this thesis, we use the observations of 65803 Didymos binary asteroid in order to create a mechanical model, composed of point masses. The surface geometry of the binary's secondary component is currently not known to us, thus we assume it is a tri-axial ellipsoid. We provide the binary with a set of initial conditions that match the observations, then we integrate the equations of motion of the two asteroids using a Runge-Kutta numerical scheme and then we study the time varying orbital elements of the secondary for two months. After that, we incorporate the impact ejecta as an N-body cloud which is expected to be produced when NASA's space probe DART crashes on the surface of the secondary. We initialise the ejecta with a set of initial positions and velocities and then we integrate their equations of motion in space, again using a Runge-Kutta numerical scheme. Ultimately we study the dynamical evolution of the ejecta cloud near the binary's domain for one month.

Chapter 1

Introduction

1.1 Asteroids

Asteroids are the small bodies of our Solar system and could be briefly described as giant rocks that are in orbit around the Sun. Their size varies from a few tens of meters to hundreds of kilometers. Smaller bodies than asteroids are called meteoroids. Asteroids usually have irregular shape that reminds of a potato, but the bigger they grow, the more spherical tend to become due to their self gravity. According to the Internationl Astronomical Union, asteroids that are almost spherical and have not cleared their neighborhood of other material orbiting them, are from now on called dwarf planets. The majority of asteroids is concentrated in two circumstellar discs called belts: The *Main belt* and the *Kuiper belt*. Asteroids are probably remnants from the formation of the Solar System and it is estimated that there exist millions. Asteroids that belong to the Main belt are mainly composed of silicon rocks and metals. 1 Ceres is an exception, because a big part of it is iced water. On the other hand, the asteroids of the Kuiper belt are mainly composed of ice. As of October 2017, the Minor Planet Center had data on almost 745,000 objects in the inner and outer Solar System, of which almost 504,000 had enough information to be given numbered designations.

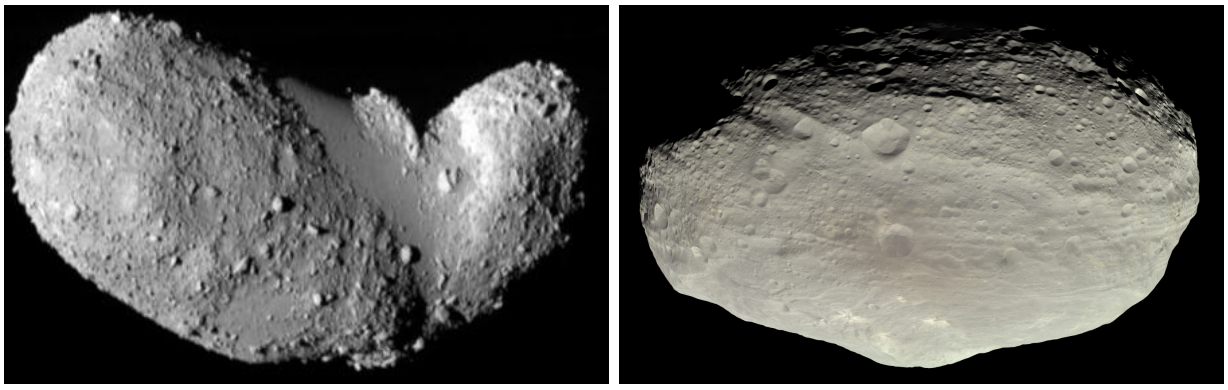


Figure 1.1: On the left we can see the asteroid 25143 Itokawa as seen by the Japanese Hayabusa space probe. On the right we can see the asteroid 4 Vesta as seen by NASA's Dawn space probe.

1.2 Position Distribution

The majority of the asteroids of the Main belt is concentrated in the *asteroid belt*, that is, a region between the orbits of Mars and Jupiter and in average distance ≈ 3 AU from the Sun. There exist other regions as well, in which we meet asteroids, like the Lagrange points of Mars and Jupiter, the motion of which the follow. These asteroids are called *Trojan bodies*. Some asteroids have themselves one or many orbiting satellites, making a double, triple or multiple asteroid system. Kuiper belt asteroids are farther Neptune's orbit and that is why they are known as *Transneptunian Objects* (TNOs). Lastly, we have a group of asteroids called *centaurs*, the orbit of which is between Jupiter and Neptune.

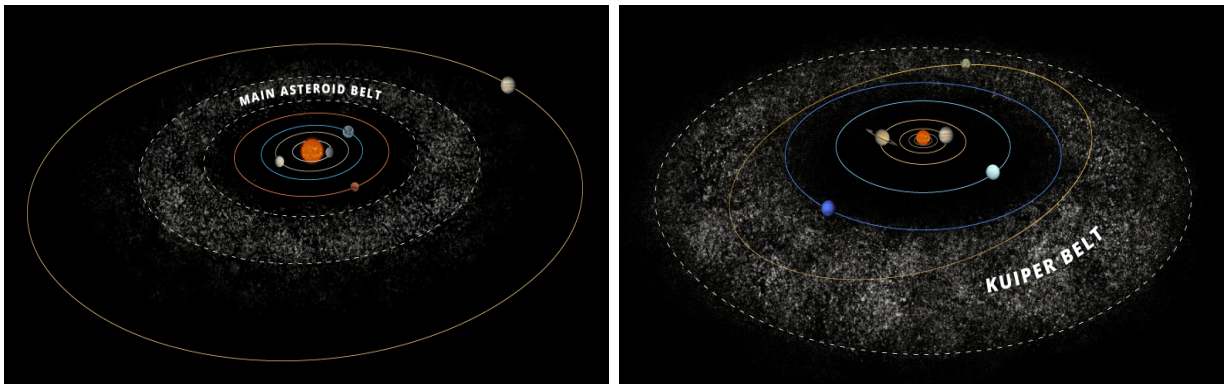


Figure 1.2: Main asteroid belt (left) and Kuiper belt (right). The size of the bodies and the distances are not scaled.

1.3 Mass Distribution

The total mass of the bodies that belong to the Main belt is not large enough. 1 Ceres (dwarf planet) is the biggest and the first body that was discovered in the Main belt and has approximately a diameter of 1000 km, whereas its mass is more or less equal to 40% of the whole mass of all the bodies of the Main belt which is estimated to be around $3 - 4\%$ of the Moon's mass. The mass of the seven biggest bodies of the Main belt is equal to 70% of its total. On the contrary, the total mass of the Kuiper belt is much larger and the biggest of asteroid of the belt has more mass and longer diameter than the biggest bodies of the Main belt. The biggest and most well known bodies of the Kuiper belt are: Eris (2003 UB_{313}) and the dwarf planet Pluto.

1.4 Origin

During the past years, we believed that the asteroids of the Main belt were debris of a planet which was crushed by a huge body. Today, the point of view that prevails is that the Main belt was the structural element of a small planet the size of Mars, but it was never formed due to Jupiter's gravitational perturbation. As far as the origin of the Kuiper belt is concerned, the belief is almost the same; fragments from the original protoplanetary disc around the Sun failed to fully coalesce into planets and instead formed into smaller bodies (asteroids).

Figure 1.3: Dwarf planet Pluto (left) as seen from New Horizons space probe, dwarf planet Ceres (right) as seen from Dawn space probe and asteroid Bennu (bottom) as seen from OSIRIS-REx spacecraft.

1.5 Discovery of Asteroids

The first (and the biggest) asteroid that was discovered was 1 Ceres (currently a dwarf planet). Its discovery occurred incidentally during the New Year's Eve (1801) from Giuseppe Piazzi, a monk who thought at the beginning that he had discovered a new star. The distance between Earth and that body was calculated by Gauss to be between Mars's and Jupiter's. During the next six years, three more asteroids were discovered: 2 Pallas, 3 Juno and 4 Vesta. Due to their small size and their nonsymmetric mass distribution, their discovery was pretty tough and so, after some years of failed attempts, the general search for asteroids was abandoned. 38 years later, Karl Ludwig Hencke who kept the search, discovered 5 Astraea and two years later, 6 Hebe. Therefore, the interest came up on the surface again. It is worth noting that the only year during which there were no new asteroid discoveries, was 1945. Until the end of the twentieth century, there were officially thousands of asteroid discoveries. After 1990, the interest for asteroids grew even stronger because of the concern of probable collision with Earth, which would be devastating. Therefore, a

new effort began for detailed tracking of their orbit. This happened using both ground telescopes and tracking devices placed in orbit around Earth. Research has currently mapped hundreds of thousands of asteroids. 600 of them have diameter more than 1 km and 3,353 of them are at such orbits that may approach Earth sometime.

Every asteroid, the existance of which is confirmed, receives a serial number. Until that, it is given a temporary number which consists of the year of discovery, a two-character code that denotes the week of the discovery year and finally one or two numbers if more asteroids were discovered the same week. After the confirmation, its code consists of the serial number in parenthesis, followed by the temporary number, e.g: (3360) 1981 VA, which was the first asteroid without a name. The serial number is usually used along with the asteroid's name, if that exists.

1.6 Binary Asteroids

A binary asteroid is a system of two asteroids orbiting around the center of mass of their system. 243 Ida was the first binary that was discovered in 1993 and since then numerous binary and even triple asteroid systems have been detected. Several theories have been posited to explain their formation. Many systems have significant macro porosity (a rubble-pile interior). The satellites orbiting large asteroids of the Main belt such as 22 Kalliope, 45 Eugenia or 87 Sylvia may have formed by disruption of a parent body after impact or fission after an oblique impact. Trans-Neptunian binaries may have formed during the formation of the Solar system by mutual capture or three-body interaction. NEAs most likely formed by spin-up and mass shedding, likely as a result of the YORP effect. Numercial simulations suggest that when solar energy spins a rubble-pile asteroid to a sufficiently fast rate by the YORP effect, material is thrown from the asteroid's equator. This process also exposes fresh material at the poles of the asteroid.

Figure 1.4: Left: Binary asteroid 243 Ida with its small minor-planet moon, Dactyl, as seen by Galileo. Right: (486958) 2014 MU_{69}, nicknamed Ultima Thule, composed of two planetesimals that are joined along their major axes.

1.7 Collision with Earth

The most accepted theory for dinosaurs extinction is an asteroid collision with Earth around 66 million years ago. This theory seems to be confirmed with the discovery of a huge crater that is located in the Gulf of Mexico. Such collisions seem to be rare according to human time, but they are actually very often in astronomical scale and the latter holds true not only for Earth, but for all the planets. Catastrophic collisions can occur every tenths of thousands of years, or millions of years. When meteorites reach Earth (which happens every day), they are burned during their descent due to the atmosphere's aerodynamic friction and they end up being dust in the air. Asteroids on the other hand are relative big in size and thus if one of them reached Earth, the atmosphere would barely touch it and thus it would collide with the surface. What would happen to Earth if such a collision occured? If the asteroid hit dry ground, then the shock wave would destroy everything in a radius of hundreds of kilometers and would trigger fires and unprecedented (Earth) quakes. The dust and the cinder which would be produced during the collision, would spread into the entire atmosphere, shading the Sunlight and creating a phenomenon similar to nuclear winter which would last for ages. Probably, all life on Earth would vanish... If on the other hand the asteroid hit an ocean, then the tsunami caused, would have a height of hundreds of meters and would entirely destroy all cities and villages at distance tenths of kilometers from the shores. In addition, the heat would boil the seawater, terminating all life near the impact place and releasing huge quantites of vapor in the atmosphere, perturbing the global climate.

1.8 Defense against a Collision

Collision between an asteroid and the Earth was seriously considered in the middle 1980s, along with some defense schemes. One of them would be to launch rockets armed with nuclear warheads meant to crash on the asteroid, not to smash it into dust, but to alter its orbit and therefore avoid collision. Another proposal is to paint one side of the asteroid with some bright color, so that the radiation pressure difference between the two sides would alter its obit. Another scenario would be to set a huge mass in orbit around the asteroid, so that the gravitational perturbation would drag the asteroid from Earth's path. Modern technology made possible to officially confirm hundreds of thousands asteroid orbits, 3,352 of which happen to be near Earth (Near Earth Asteroids (NEAs)). Those celestial bodies are constantly being observed by specialists that occupy themselves with the calculation of probability of collision. Luckily, until today (Summer 2019), none of these bodies have a high collision probability. The highest probability is given to the body (29075) 1950 DA and is $\approx 0.33\%$

1.9 65803 Didymos Binary

65803 Didymos is a sub-kilometer binary asteroid system and is the target of the proposed AIDA asteroid mission (see next section). Due to its binary nature, it was then named "Didymos", the Greek word for twin. The primary asteroid was discovered on 11 April 1996, by the University of Arizona Steward Observatory's Spacewatch survey using its 0.9-meter telescope at Kitt Peak National Observatory in Arizona, United States. The binary nature of the asteroid was discovered by others. Suspicions of binarity first arose in Goldstone delay-Doppler echoes, and these were

confirmed with an optical lightcurve analysis, along with Arecibo radar imaging on 23 November 2003. Didymos orbits the Sun at a distance of $1.0 - 2.3$ AU once every 2 years and 1 month (770 days). Its orbit has an eccentricity of ≈ 0.38 and an inclination of $\approx 3^o$ with respect to the ecliptic. Its approach to Earth in November 2003, was especially close with a distance of 7.18 million km. It will not come that near until November 2123, with a distance of 5.9 million km. It will also make a close approach to Mars: 4.69 million km in 2144. Didymos is classified as an S-type (figure 1.5) based on observations by DeLeon et al. and taxonomic classification scheme by Bus and DeMeo (2009), even though it was originally classified as an Xk-type (Binzel et al. 2004) due to limited wavelength coverage. The spectrum looks similar to that of Itokawa, which has a composition close to LL chondrites based on the analysis of the returned samples. The primary asteroid rotates rapidly, with a period of 2.26 hours and a brightness variation of 0.08 magnitude ($U = 3/3$), which indicates that the body has a nearly spheroidal shape. The secondary asteroid, moves in a mostly circular retrograde orbit with an orbital period of 11.9 hours. It measures approximately 0.163 kilometers in diameter compared to 0.775 kilometers of its primary (a mean-diameter-ratio of 0.21). Table 1.1 summarizes some important parameters of the binary that will help our study in the next chapters.

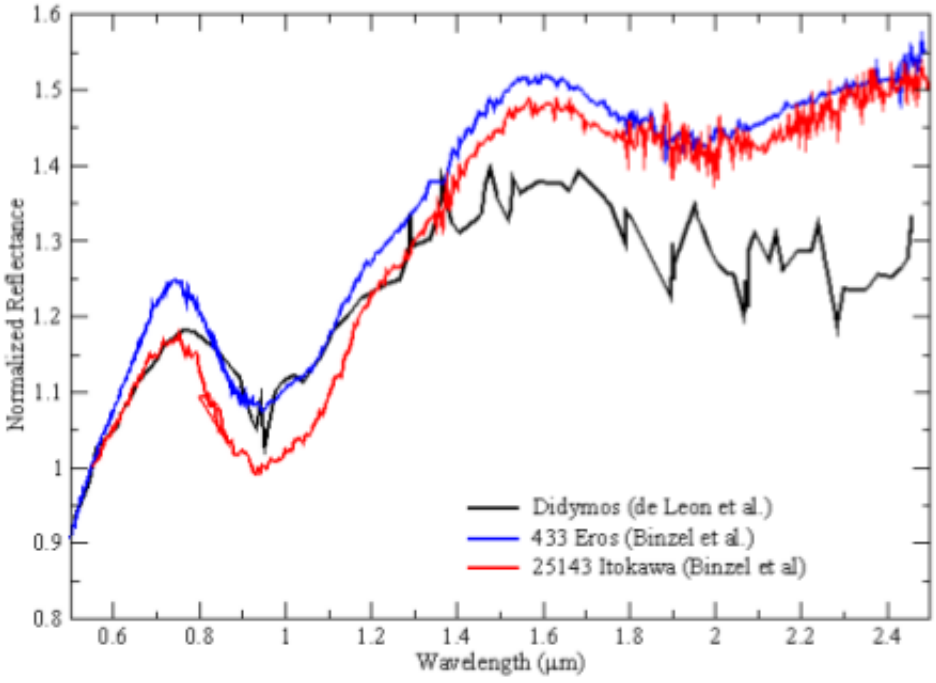


Figure 1.5: Spectrum of Didymos (de Leon et al.) compared with that of Eros (Binzel et al.) and Itokawa (Binzel et al.).

Parameter	Value
Primary's indicative size	0.775 km $+/-10\%$
Secondary's indicative size	0.163 km $+/-0.018$ km
Primary's bulk density	2146 kg $\cdot m^{-3}$ $+/-30\%$
Secondary's (ellipsoid) axes	$a_S = 103$ m $b_S = 79$ m $c_S = 66$ m
Distance between the center of primary and secondary	1.18 km $+0.04/-0.02$ km
Total mass of the system	$5.278 \cdot 10^{11}$ kg $+/- 0.54 \cdot 10^{11}$ kg
Geometric albedo	$0.15 + / - 0.04$
Radar albedo	$0.27 + / - 25\%$
Primary's rotation period	2.2600 h $+/-$ 0.0001 h
Heliocentric eccentricity	$0.383752501 + / - 7.7 \cdot 10^{-9}$
Heliocentric semi-major axis	$1.6444327821 + / - 9.8 \cdot 10^{-9}$ AU
Heliocentric inclination to the ecliptic	$3.4076499^o + / - 2.4 \cdot 10^{-6} o$
Mean absolute magnitude (whole system)	$18.16 + / - 0.04$
Obliquity to heliocentric orbit	$171^o + / - 9^o$
Diameter ratio	$0.21 + / - 0.01$
Secondary's orbital period	11.920 h $+0.004/-0.006$
Secondary's orbital eccentricity	Upper limit: 0.03
Secondary's orbital inclination (assumed)	0^o
Obliquity of the primary principal axis with respect to the mutual orbital plane (assumed)	0^o
Obliquity of the secondary principal axis with respect to the mutual orbital plane (assumed)	0^o

Table 1.1: Synopsis of some physical and orbital characteristics of Didymos binary.

It is worth noting that the only directly measured dynamical parameters by the observations are the orbital period of the secondary around the primary, their orbital separation, the rotation period of the primary and the size ratio of the secondary to the primary. All the other quantities (e.g. system's mass, etc) are derived from these measured parameters. A shape model of the primary is also derived from radar observations combined with optical lightcurve data (see chapter 2).

1.10 AIDA Mission

The Asteroid Impact and Deflection Assessment (AIDA) mission is a proposed pair of space probes which will study and demonstrate the kinetic effects of crashing an impactor spacecraft into an asteroid moon. The mission's main purpose is to test whether a spacecraft could successfully deflect an asteroid on a collision course with Earth. The concept proposes two spacecraft: Hera, built by ESA will orbit the binary and make multiple observations, and Double Asteroid Redirection Test (DART), built by NASA will impact the moon. Besides the observation of the change of orbital parameters of the asteroid moon, the observation of the plume, the crater, and the freshly exposed material will provide unique information for asteroid deflection, science and mining communities. Initially, Hera's role was to be realized by a much larger spacecraft called Asteroid Impact Mission (AIM), but In December 2016 the European Space Agency cancelled the development of the AIM

spacecraft after Germany decided to fund the ExoMars project only. NASA has continued on with the development of the DART spacecraft, replacing AIM's role of monitoring the effects of the impact with ground-based telescopes. As DART is currently planned to launch in 2021, Hera is currently intended to arrive at Didymos a few years after DART's impact. To maximize scientific outcome, the AIDA team proposes to delay DART's launch so that Hera will arrive at the asteroid first, enabling it to witness DART's impact. While most of the initial objectives of AIDA would still be met if Hera arrives after DART, as a drawback, data from direct observation of the impact and ejecta formation will not be obtained. The AIDA mission is a joint international collaboration of the European Space Agency (ESA), the German Aerospace Center (DLR), Observatoire de la Cote d'Azur (OCA), NASA, and Johns Hopkins University Applied Physics Laboratory (JHU/APL). The project was formed by joining two separate studies, DART, an asteroid impactor developed by NASA, and a monitoring spacecraft - ESA's Hera (formerly AIM). The μLidar instrument on board Hera will be provided by a consortium of teams from Portugal, Poland, and Ireland. Two CubeSats will be deployed by Hera while at Didymos. The APEX (Asteroid Prospection Explorer) CubeSat was developed by Sweden, Finland, Czech Republic and Germany. The Juventas CubeSat is developed by GomSpace and GMV's Romanian division. Along with monitoring DART's impact, Hera itself may also carry an impactor. As proposed by the Japanese Space Agency, this instrument will be a replica of the Small Carry-on Impactor (SCI), an explosively formed penetrator on board the Hayabusa2 asteroid sample return mission. The SCI will hit the asteroid's moon at a speed lower than that of DART. By performing a secondary impact, a comparison of the effects posed by two collisions of different nature on the same asteroid can be realized, helping validate numerical impact algorithms and scaling laws.

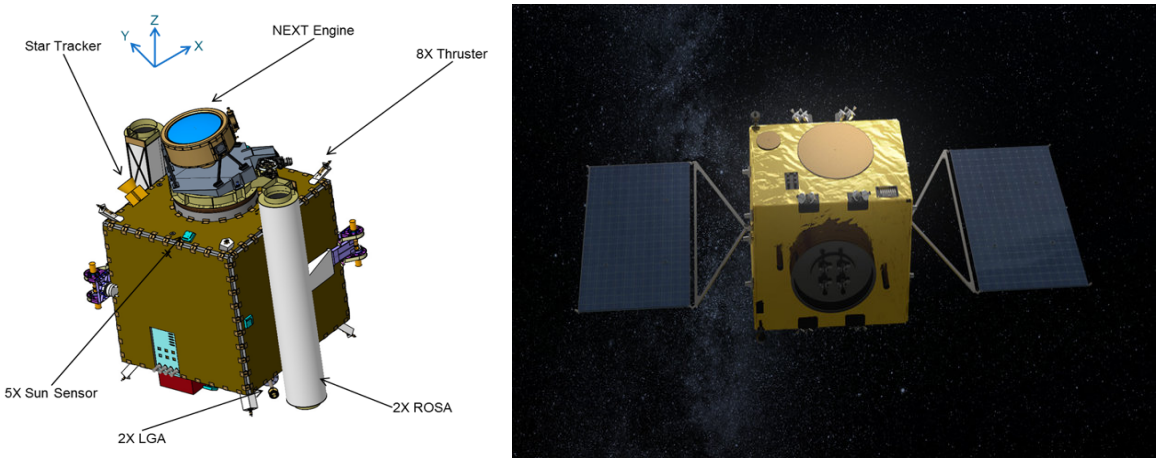


Figure 1.6: On the left we can see the kinetic impactor DART which is meant to crash on Didymoon. On the right we can see Hera spacecraft which is meant to perform numerous scientific observations when it arrives at the asteroids.

Chapter 2

Asteroid Modeling

In order to calculate the force being acted on a test particle in space provided its gravitational interaction with N bodies, one could apply Newton's laws of motion. In case we are dealing with point masses, Newton's second law is quite simple because we know the analytic form of the gravitational potential $V(x, y, z)$ of a point mass. Unfortunately, asteroids appear to be asymmetric rigids bodies and thus it is impossible to obtain an analytic function $V(x, y, z)$ (with finite number of terms) of their gravitational potential. So how could someone determine the force being acted on a test particle outside such a body? We could define a rigid body of finite volume as a collection of N point masses. The more point masses, the more precise the model will be. In such a case, one could directly find the potential energy of a test particle as the sum of N pontenials that are generated by those N point masses. After that, Newton's laws can be applied to obtain the differential equations of motion. So our first step is to make use of available surface data in order to create a numerical model of the binary so that we can proceed to our study. From now on, we will refer to the primary asteroid as Didymain and to the secondary asteroid as Didymoon.

2.1 Computational Space

In order to create a model of an asteroid, we have to define a geometric computational space inside of which the modeling calculations will take place. The latter will be a rectangular parallelepiped (box) as shown in figure 2.1, the sides of which will be determined by the asteroid's natural size:

$$\begin{aligned} \Delta x &= \Delta x' + 2h = x_{max} - x_{min} + 2h \\ \Delta y &= \Delta y' + 2h = y_{max} - y_{min} + 2h \\ \Delta z &= \Delta z' + 2h = z_{max} - z_{min} + 2h \end{aligned} \tag{2.1}$$

where $x_{max} - x_{min}$, $y_{max} - y_{min}$, $z_{max} - z_{min}$ are the maximum distances between the asteroid's surface along the x, y and z directions respectively and h is a small positive number which we will call computational step and will be used for the modeling procedure. In other words, through equations (2.1), we have defined a computational space of size $\Delta x', \Delta y', \Delta z'$ and then we have increased each side by $2h$ (one direction by h and the oppsite direction by h).

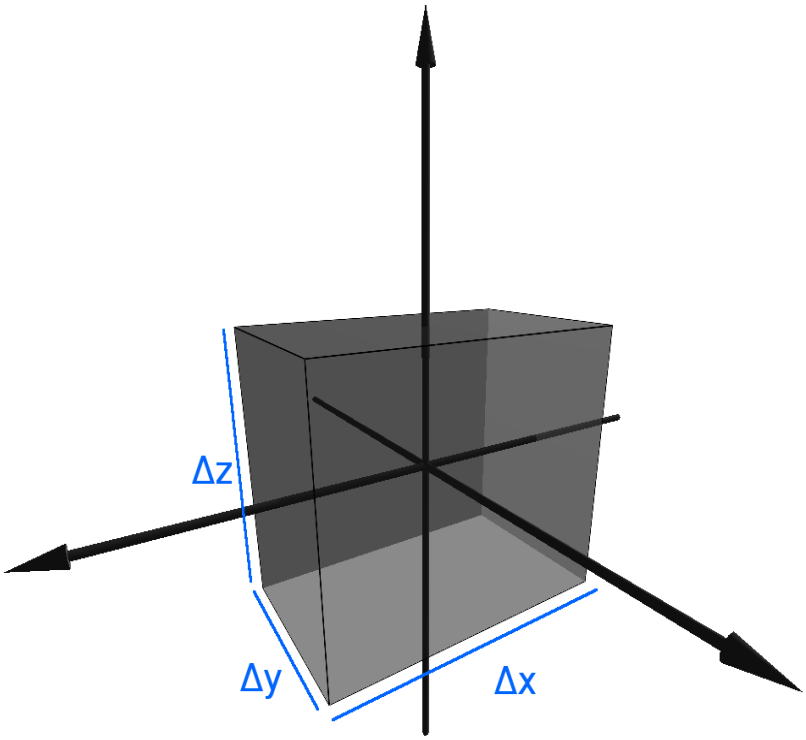


Figure 2.1: Computatonal space

2.2 Didymain Model

We have at our disposal two txt files that contain surface mapping data. The first one contains 1148 sampled surface vertices (points) of Didymain in Cartesian coordintes. The second file originates from the first and contains 2292 triads of unsigned integers that correspond to triads of vertices that form triangles (planes) on Didymain's surface. As far the asteroid's interior is concerned, it is not yet known to us, so we need an algorithm that will fill it with vertices (point masses). In the end we will end up with a full model. Below follows a graphical representation of Didymain's surface.

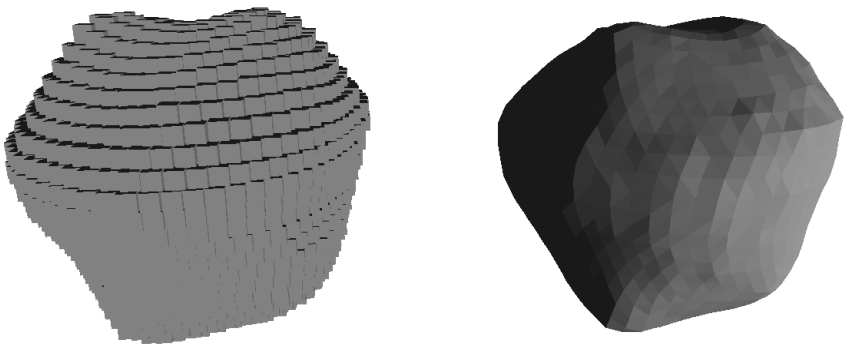


Figure 2.2: Didymain's surface model. On the left we have 1148 vertices, each enclosed with a voxel for better visualisation. On the right we have 2292 planes that form a smooth approximation of the surface.

The algorithm that creates our model could be as follows:

- Calculate the quantities x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max} from Didymain's vertices file.
- Determine a Cartesian step h with which you will move in the computational space along the directions x, y, z. The shorter h is, the preciser the final model will be (the more points it will consist of).
- Increase the computational space (i.e. each side of the box) by $2h$.
- Loop through all the computational space with step h.
  - ▷ For your current (x, y, z) position, determine if you are inside the asteroid's surface or not. If yes, mark your current (x, y, z) position as an interior point and print it to a file. Else proceed to the next point.

The process of determining whether a point lies inside or outside the surface is not as easy as it sounds. As we saw earlier, Didymain's surface is not given to us in an analytic form $f(x, y, z) = 0$, nor we can determine any symmetries from figure 2.2. In order to achieve our goal, we will develop a *ray casting algorithm* in the 3D space. We will briefly present the algorithm for the 2D space and then generalise it for the 3D one.

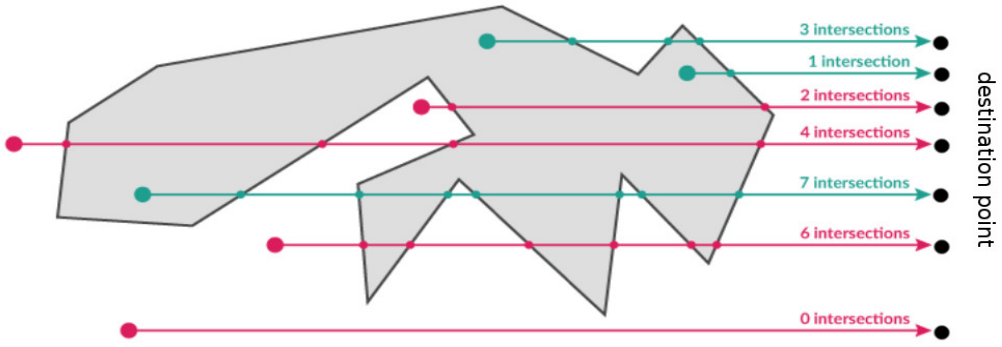


Figure 2.3: Ray casting algorithm representation.

Consider a simple polygon[1] in the 2D space (figure 2.3). One way of finding whether a point is inside or outside the polygon is to count how many times a ray, starting from the point and going in any fixed direction, intersects the edges of the polygon. If the point is outside the polygon, the ray will intersect its edges an even number of times. If the point is inside the polygon, the ray will intersect its edges an odd number of times. This method won't work if the point is on one of the edges of the polygon. The algorithm is based on the simple observation that if a point moves along a ray from infinity to a destination point and if it crosses the boundary of the polygon, possibly

[1]In geometry a simple polygon is a flat shape consisting of straight, non-self-intersecting line segments that are joined pair-wise to form a closed path.

several times, then it alternately goes from the outside to inside, then from the inside to outside, etc. As a result, provided that initially the point lies outside the polygon, then after every two border crossings, the moving point goes outside. Or provided that initially the point lies inside the polygon, then after two border crossings, the moving point goes inside. This observation can be mathematically proved using the Jordan curve theorem. The latter can be generalised for the 3D space. In such a case, instead of a simple polygon, we could have a polyhedron (figure 2.2 on the right), the planes of which are tested for intersection with a ray. So the problem ends up in the calculation of the intersection between a line (ray) and a plane in the 3D space.

Consider a Cartesian coordinate system $Oxyz$. Let $\vec{r}_0$ be the position vector of some known point $P_0(x_0, y_0, z_0)$ and let $\vec{n} = n_x\hat{x} + n_y\hat{y} + n_z\hat{z}$ be a nonzero vector. The plane determined by the point P_0 and the vector $\vec{n}$ consists of the points $P(x, y, z)$ with position vectors $\vec{r}$, such that the vector drawn from P_0 to P is perpendicular to $\vec{n}$. Recalling that two vectors are perpendicular if and only if their dot product is zero, it follows that the desired plane can be described as the set of all points P such that

$$\vec{n} \cdot (\vec{r} - \vec{r_0}) = 0 \Rightarrow$$

$$(n_x\hat{x} + n_y\hat{y} + n_z\hat{z}) \left[(x - x_0)\hat{x} + (y - y_0)\hat{y} + (z - z_0)\hat{z}\right] = 0 \Rightarrow$$

$$(x - x_0)n_x + (y - y_0)n_y + (z - z_0)n_z = 0 \Rightarrow$$

$$xn_x + yn_y + zn_z - (x_0n_x + y_0n_y + z_0n_z) = 0 \tag{2.2}$$

which reminds our familiar plane equation $ax + by + cz + d = 0$

Now let $\vec{r}_1$ and $\vec{r}_2$ be the position vectors of two known points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$. The line determined by P_1 and P_2 consists of the points $P(x, y, z)$ with position vectors $\vec{r}$, such that the vector drawn from P_1 to P is parallel to the one drawn from P_1 to P_2. Thus

$$\vec{r} - \vec{r}_1 = \lambda(\vec{r}_2 - \vec{r}_1) \Rightarrow \vec{r} = \vec{r}_1 + \lambda(\vec{r}_2 - \vec{r}_1) \Rightarrow$$

$$\begin{cases} x = x_1 + \lambda(x_2 - x_1) \\ y = y_1 + \lambda(y_2 - y_1) \\ z = z_1 + \lambda(z_2 - z_1) \end{cases} \tag{2.3}$$

which is the parametric equation of a straight line. The locus of points that is defined by the intersection of a plane and a line can be found by substituting equations (2.3) to (2.2) and in general is one of the following cases : 1) A point, 2) A line, 3) Void

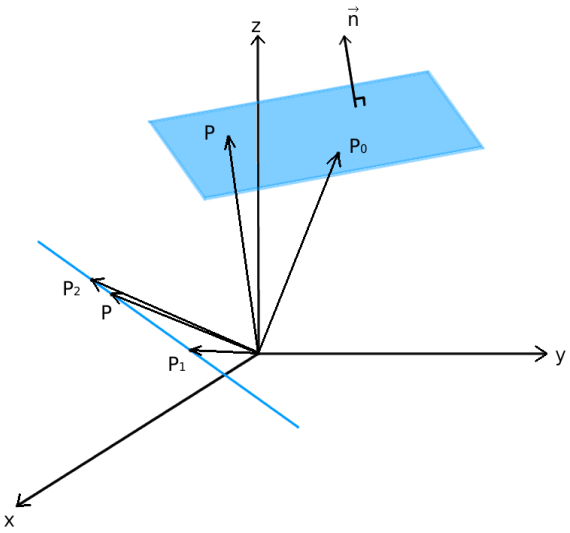


Figure 2.4: Gemoetrical representation of a line and a plane in the 3D space.

$$(2.3)\ \&\ (2.2) \Rightarrow [x_1+\lambda(x_2-x_1)]n_x+[y_1+\lambda(y_2-y_1)]n_y+[z_1+\lambda(z_2-z_1)]n_z-(x_0n_x+y_0n_y+z_0n_z)=0 \Rightarrow$$

$$x_1n_x+\lambda n_x(x_2-x_1)+y_1n_y+\lambda n_y(y_2-y_1)+z_1n_z+\lambda n_z(z_2-z_1)-(x_0n_x+y_0n_y+z_0n_z)=0 \Rightarrow$$

$$\lambda\left[n_x(x_2-x_1)+n_y(y_2-y_1)+n_z(z_2-z_1)\right]=(x_0n_x+y_0n_y+z_0n_z)-(x_1n_x+y_1n_y+z_1n_z) \Rightarrow$$

$$\lambda=\frac{(x_0-x_1)n_x+(y_0-y_1)n_y+(z_0-z_1)n_z}{(x_2-x_1)n_x+(y_2-y_1)n_y+(z_2-z_1)n_z}$$

By substituting the parameter λ in the line equation, we receive the coordinates of the intersection point.

$$\begin{aligned}
x_i &= x_1+\left[\frac{(x_0-x_1)n_x+(y_0-y_1)n_y+(z_0-z_1)n_z}{(x_2-x_1)n_x+(y_2-y_1)n_y+(z_2-z_1)n_z}\right](x_2-x_1)\\
y_i &= y_1+\left[\frac{(x_0-x_1)n_x+(y_0-y_1)n_y+(z_0-z_1)n_z}{(x_2-x_1)n_x+(y_2-y_1)n_y+(z_2-z_1)n_z}\right](y_2-y_1)\\
z_i &= z_1+\left[\frac{(x_0-x_1)n_x+(y_0-y_1)n_y+(z_0-z_1)n_z}{(x_2-x_1)n_x+(y_2-y_1)n_y+(z_2-z_1)n_z}\right](z_2-z_1)
\end{aligned}$$

Returning to the case of Didymain's surface, we have to check 2292 planes for possible intersection with each ray we form. The equations of the planes shall be defined from the coordinates of their 3 vertices and the ray shall be a line segment that starts from the computational space point $\mathrm{P}(x, y, z)$ and ends at the destination point $\mathrm{P}_{dest}(x_{max} + h, y, z)$. One can use a destination point at any direction (provided the point is on or out of the computational box), thus for simplicity we choose the direction of the $+x$ axis for all the rays.

There is still one more issue to solve. Calculating the intersection point between a ray and all the planes of the asteroid's surface will not yield desirable results. We must make sure that the intersection point is strictly limited on the triangle's surface that is defined by the 3 vertices and not on the extension of the triangle in the whole 3D space, otherwise all the rays casted will intersect with all the planes of Didymain's surface somewhere in the 3D space because non of the planes will be exactly parallel to any ray (we are dealing with floating point arithmetic). But how can we decide if a point is part of a 3D triangle's surface or not? Consider the following picture.

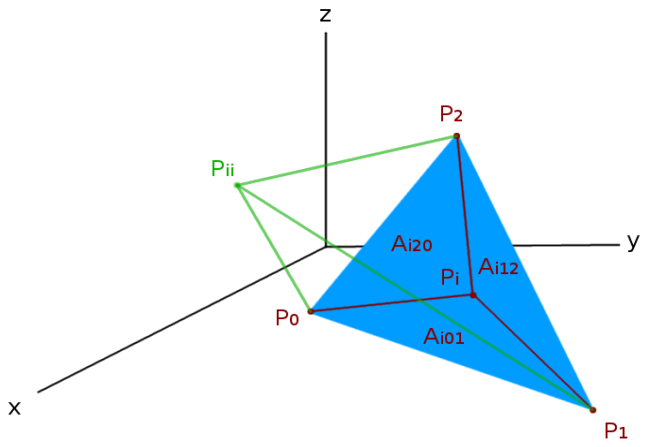


Figure 2.5: Deciding whether a point lies on the syrface of a triangle or not through the summation of the areas formed by 3 sub-triangles. P_i lies on the surface, while P_{ii} does not.

Let P_0, P_1, P_2 be the vertices of a triangle in the 3D space with area A and let P_i be an arbitrary point. Now form the triangles $(P_iP_0P_1)$, $(P_iP_1P_2)$, $(P_iP_2P_0)$ with corresponding areas A_{i01}, A_{i12}, A_{i20}. If $A_{i01} + A_{i12} + A_{i20} = A$, then P_i lies on the surface, otherwise not (e.g. point P_{ii} in figure 2.5). The area of an arbitrary triangle $(P_1P_2P_3)$ in the 3D space can easily be calculated as:

$$A_P = \frac{1}{2}|\overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}|$$

Finally, the complete algorithm to create Didymain model is the following:

- Calculate the quantities x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max} from Dydimain's vertices file.

- Determine a Cartesian step h with which you will move in the computational space. The shorter h is, the preciser the final model will be (the more points it will consist of).

- Increase the computational space (i.e. each side of the box) by h.

- Loop through all the computational space with step h.
  - ▷ Set the counter of intersection points to zero.
  - ▷ Form the line segment that connects your current position $P(x, y, z)$ with the point $P_{dest}(x_{max} + h, y, z)$
  - ▷ Loop through all the surface planes in search for intersection with the line segment.
    - ☐ Calculate the intersection point $P_i(x_i, y_i, z_i)$ between the current plane and the line segment formed.
    - ☐ Check if P_i sits on the triangle's surface. If yes, then count P_i as an intersection point.
  - ▷ Check whether the counter of intersection points is odd or even. If it is odd, then mark your current (x, y, z) position as an interior point and print it to a file. Else proceed to the next point.

When the above algorithm is executed, we receive the model of figure (2.6).

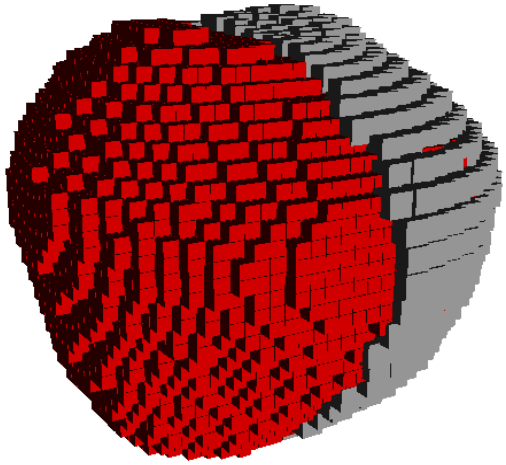


Figure 2.6: Complete model of Didymain. The gray voxels represent the asteroid's surface (the left half of the surface has been removed so that the interior can be visible), while the red ones represent its interior, that is, all the voxels that were produced from the previous algorithm. They are 17647 voxels in total (surface + interior), produced with a Cartesian step $h = 0.025$ km.

2.3 Didymoon Model

As far as Didymoon is concerned, we don't have any specific radar observations or detailed surface optical lightcurves available. Keep in mind that it is a body with an approximate size of only 160 m that is located millions of kilometers away from Earth. So we have decided to consider Didymoon a tri-axial ellipsoid with semi axes a, b, c. Although it is possible to obtain an analytic form of the gravitational potential of an ellipsoid, we choose again to create a model that consists of point masses, such that their macroscopic form match an ellipsoid. Rendering such a model is pretty easy beacause now we do know the analytic form of the equation of an ellipsoid:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$

So the algorithm that will create Didymoon model could be as follows:

- Input the semi axes a, b, c of the ellipsoid.
- Determine the computational box from the quantites $x_{min} = -a$, $x_{max} = a$, $y_{min} = -b$, $y_{max} = b$, $z_{min} = -c$, $z_{max} = c$.
- Determine a Cartesian step h with which you will move in the computational box. The shorter h is, the preciser the final model will be (the more points it will consist of).
- Increase each side of the computational box by $2h$.
- Loop through all the computational box with step h.
  - ▷ Check if your current space position (x, y, z) satisfies the inequality $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} \leq 1$
    If yes, then mark your current (x, y, z) position as an interior or surface point and print it to a file. Else proceed to the next point.

When the above algorithm is executed, we receive the model of figure (2.7).

Figure 2.7: Complete model of Didymoon. On the left we can see the model voxel by voxel and on the right we can see the very same model using a smooth surface. Totally 2323 voxels were produced with Cartesian step $h = 0.01$ km.

Chapter 3

The Binary in Orbit

Now that we have both asteroids modeled, we may proceed to the next step: Set the binary in orbit in the 3D space. To achieve that, we will make some assumptions. These concern the coordinate systems we will use, the way the asteroids translate and rotate in space, the numerical method we will use in order to solve the differential equations of motion, the units of measurement and and the initial conditions and parameters that we will incorporate in the system.

3.1 Coordinate Systems

We consider the three coordinate systems (F_0,F_1,F_2) of figure 3.1. The first one is a global inertial frame F_0, fixed at $O(0, 0, 0)$ and corresponds to the center of mass of the two asteroids. The second one is a non-inertial frame F_1, the origin of which always coincides with Didymain's center of mass and the frame's axes always coincide with Didymain's principal axes of inertia. So while Didymain moves through space, the origin of F_1 moves at the same path and while Didymain rotates, the axes of F_1 also rotate the same way. The third one is also a non-inertial frame F_2, the origin of which always coincides with Didymoon's center of mass and the frame's axes always coincide with Didymoon's principal axes of inertia. F_2 now follows Didymoon at its path and rotates the same way as Didymoon's principal axes do.

3.2 Translation and Rotation

In order to calculate the motion of the asteroids in space, we have to define how the two asteroids interact gravitationally. Both asteroids are considered to be rigid bodies, so each body would require 6 degrees of freedom in order for its configuration to be fully defined: 3 translational (position in space) and 3 rotational (orientation in space). Each degree of freedom would be calculated through the solution of a corresponding ordinary differential equation, which means that we would need 6 ODEs for each asteroid. However we choose to reduce our calculations by making use of table 1.1 and the fact that **observations show that Didymoon is tidally locked on Didymain.** The latter permits us to get away with the ODEs that describe the rotation of the

asteroids and instead, approximate Didymoon's rotation manually (we will explain how) without significant computational errors. Didymain's rotation ODEs can also be approximated assuming a constant angular velocity vector $\vec{\omega}_1$ towards the $+z$ direction of the F_1 coordinate system. So the "heavy" calculations end up being 3 ODEs that concern the translation of Didymain and 3 ODEs that concern the translation of Didymoon.

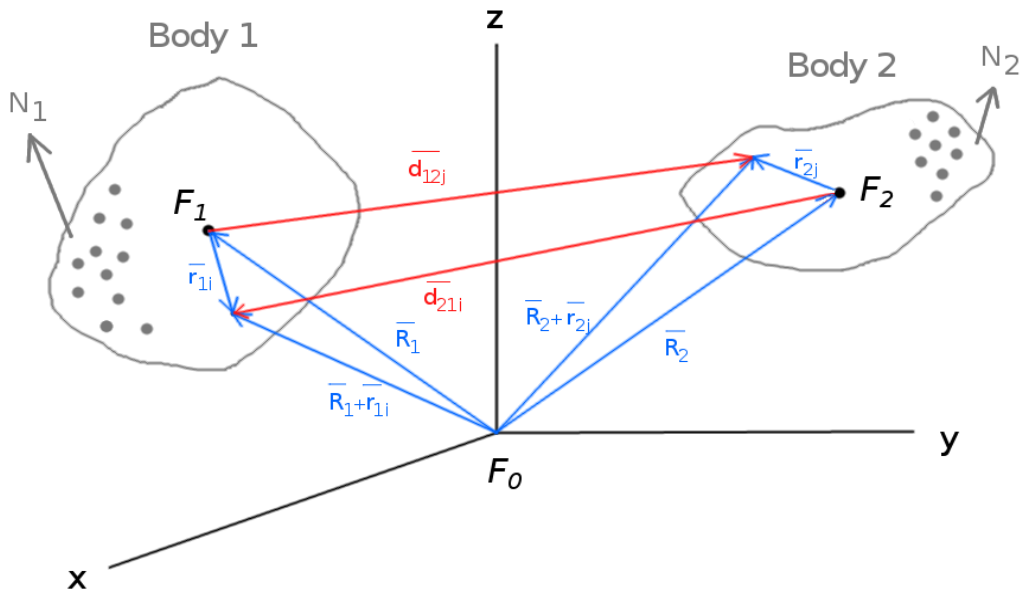


Figure 3.1: Two rigid bodies, each consisting of N_1 and N_2 point masses respectively, placed in space. The red and blue vectors are used to formulate their gravitational interaction.

Consider figure 3.1 which depicts two rigid bodies in space. Suppose Body 1 consists of N_1 point masses, while Body 2 consists of N_2 point masses. Let M_1 and M_2 be the total masses of the two bodies respectively. Assuming constant bulk density for each body, we can write:

$$m_{1i} = m_1, \quad \text{for} \quad i = 1, 2, \ldots, N_1$$

$$m_{2j} = m_2, \quad \text{for} \quad j = 1, 2, \ldots, N_2$$

Also the total mass of each body is the sum of its point masses:

$$M_1 = \sum_{i=1}^{N_1} m_{1i} \quad \text{and} \quad M_2 = \sum_{j=1}^{N_2} m_{2j}$$

Let $\vec{R}_1$ be the position vector of the center of mass of Body 1 (Didymain in our case) with respect to F_0 and $\vec{R}_2$ the position vector of the center of mass of Body 2 (Didymoon in our case) with respect to F_0. Let $\vec{r}_{1i}$ be the position vector of the i-point mass of Body 1 with respect to F_1 and

$\vec{r}_{2j}$ be the position vector of the j-point mass of Body 2 with respect to F_1. Finally we define the vector $\vec{d}_{12j}$ that connects the center of mass of Body 1 with an arbitrary point of Body 2 and the vector $\vec{d}_{21i}$ that connects the center of mass of Body 2 with an arbitrary point of Body 1. The latter is mathematically expressed as:

$$\vec{R}_1 + \vec{d}_{12j} = \vec{R}_2 + \vec{r}_{2j} \Rightarrow \vec{d}_{12j} = \vec{R}_2 - \vec{R}_1 + \vec{r}_{2j}$$

$$\vec{R}_2 + \vec{d}_{21i} = \vec{R}_1 + \vec{r}_{1i} \Rightarrow \vec{d}_{21i} = \vec{R}_1 - \vec{R}_2 + \vec{r}_{1i}$$

Provided the two bodies interact gravitationally, we want to write down the equations of their motion. As far as the translational part of their motion is concerned, we only need to know the motions of the two centers of mass. Application of Newton's second law, yields:

$$\cancel{M_1}\ddot{\vec{R}}_1 = \frac{G\cancel{M_1}m_2}{d_{121}^3}\vec{d}_{121} + \frac{G\cancel{M_1}m_2}{d_{122}^3}\vec{d}_{122} + \cdots + \frac{G\cancel{M_1}m_2}{d_{12N_2}^3}\vec{d}_{12N_2} \Rightarrow$$

$$\ddot{\vec{R}}_1 = Gm_2\sum_{j=1}^{N_2}\frac{\vec{d}_{12j}}{d_{12j}^3} \Rightarrow \ddot{\vec{R}}_1 = Gm_2\sum_{j=1}^{N_2}\frac{\vec{R}_2 - \vec{R}_1 + \vec{r}_{2j}}{d_{12j}^3} \Rightarrow$$

$$\begin{cases} \ddot{X}_1 = Gm_2\displaystyle\sum_{j=1}^{N_2}\frac{X_2 - X_1 + x_{2j}}{d_{12j}^3} \\ \ddot{Y}_1 = Gm_2\displaystyle\sum_{j=1}^{N_2}\frac{Y_2 - Y_1 + y_{2j}}{d_{12j}^3} \\ \ddot{Z}_1 = Gm_2\displaystyle\sum_{j=1}^{N_2}\frac{Z_2 - Z_1 + z_{2j}}{d_{12j}^3} \end{cases}$$

where $d_{12j} = \sqrt{(X_2 - X_1 + x_{2j})^2 + (Y_2 - Y_1 + y_{2j})^2 + (Z_2 - Z_1 + z_{2j})^2}$

In a similar way, we can extract the equations of motion for the center of mass of Body 2:

$$\cancel{M_2}\ddot{\vec{R}}_2 = \frac{G\cancel{M_2}m_1}{d_{211}^3}\vec{d}_{211} + \frac{G\cancel{M_2}m_1}{d_{212}^3}\vec{d}_{212} + \cdots + \frac{G\cancel{M_2}m_1}{d_{21N_1}^3}\vec{d}_{21N_1} \Rightarrow$$

$$\ddot{\vec{R}}_2 = Gm_1\sum_{i=1}^{N_1}\frac{\vec{d}_{21i}}{d_{21i}^3} \Rightarrow \ddot{\vec{R}}_2 = Gm_1\sum_{i=1}^{N_1}\frac{\vec{R}_1 - \vec{R}_2 + \vec{r}_{1i}}{d_{21i}^3} \Rightarrow$$

$$\begin{cases} \ddot{X}_2 = Gm_1 \sum_{i=1}^{N_1} \dfrac{X_1 - X_2 + x_{1i}}{d_{21i}^3} \\ \ddot{Y}_2 = Gm_1 \sum_{i=1}^{N_1} \dfrac{Y_1 - Y_2 + y_{1i}}{d_{21i}^3} \\ \ddot{Z}_2 = Gm_1 \sum_{i=1}^{N_1} \dfrac{Z_1 - Z_2 + z_{1i}}{d_{21i}^3} \end{cases} \tag{3.1}$$

where $d_{21i} = \sqrt{(X_1 - X_2 + x_{1i})^2 + (Y_1 - Y_2 + y_{1i})^2 + (Z_1 - Z_2 + z_{1i})^2}$

One could use a numerical scheme to solve the two systems of the ODEs (as we will see later). However we can skip some calculations by using Newton's third law. Applying the latter to the system of the centers of mass of the two bodies, we get:

$$\vec{F}_1 = -\vec{F}_2 \Rightarrow \vec{F}_1 + \vec{F}_2 = \vec{0} \Rightarrow M_1\ddot{\vec{R}}_1 + M_2\ddot{\vec{R}}_2 = \vec{0} \overset{\int}{\Rightarrow}$$

$$M_1\dot{\vec{R}}_1 + M_2\dot{\vec{R}}_2 = \vec{c_1} \overset{\int}{\Rightarrow} M_1\vec{R}_1 + M_2\vec{R}_2 = \vec{c}_1 t + \vec{c}_2 \Rightarrow$$

$$\frac{M_1\vec{R}_1 + M_2\vec{R}_2}{M_1 + M_2} = \frac{\vec{c}_1 t + \vec{c}_2}{M_1 + M_2}$$

The last expression states that the center of mass of the two centers of mass of the two bodies shall move on a straight line with constant velocity. We set $\vec{c}_1 = \vec{c}_2 = \vec{0}$

$$M_1\vec{R}_1 + M_2\vec{R}_2 = \vec{0} \Rightarrow \vec{R}_1 = -\frac{M_2\vec{R}_2}{M_1} \Rightarrow \begin{cases} X_1 = -\dfrac{M_2 X_2}{M_1} \\ Y_1 = -\dfrac{M_2 Y_2}{M_1} \\ Z_1 = -\dfrac{M_2 Z_2}{M_1} \end{cases} \tag{3.2}$$

That way, we have fixed the center of mass of the two centers of mass at in the origin of F_0 and instead of solving the ODEs for both bodies, we only need to solve for one of them (e.g equations (3.1) for Body 2) and then we can use equations (3.2) to find the motion of the other.

As far as the rotational motion of the asteroids is concerned, we set it manually in order to reduce our calculations. Specifically, we assume that Didymain rotates around the $+z$ axis with constant angular velocity $\omega_1 = \frac{2\pi}{T_1}$, where (according to table 1.1) $T_1 \approx 2.26$ h is the self rotation period. Didymain consists of N_1 point masses, each at posistion (x_{1i}, y_{1i}, z_{1i}) with respect to the frame F_1 at $t = 0$. The rotation is achieved by multiplying all Didymain's vertices with the rotation matrix R_z around the z axis.

$$\begin{bmatrix} x'_{1i} \\ y'_{1i} \\ z'_{1i} \end{bmatrix} = \underbrace{\begin{bmatrix} \cos\omega t & -\sin\omega t & 0 \\ \sin\omega t & \cos\omega t & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{R_z} \begin{bmatrix} x_{1i} \\ y_{1i} \\ z_{1i} \end{bmatrix}$$

where $(x'_{1i}, y'_{1i}, z'_{1i})$ is the position of the i-vertex with respect to F_1 after the rotation.

The (manual) rotation of Didymoon is slightly more complex. As we stated previously, Didymoon is assumed to be tidally locked on Didymain. In the case where the orbital eccentricity and the obliquity are nearly zero, tidal locking results in one side of Didymoon constantly facing its partner, an effect known as synchronous rotation. The last statement is sufficiently true in our case although there should be some libration because the Didymoon's orbit isn't perfectly circular. Consider the following figure:

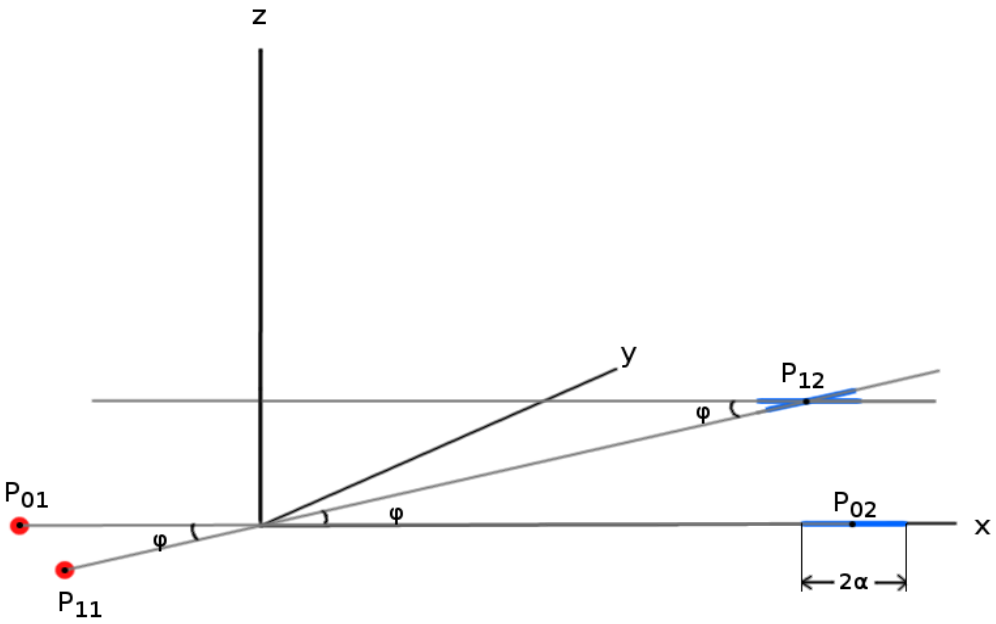


Figure 3.2: Representation of Didymoon's self rotation. The red dot depicts Didymain's center of mass at the moments t_0 and t_1 (P_{01} and P_{11} respectively). The blue stick depicts Didymoon's largest axis, again at the moments t_0 and t_1 (P_{02} and P_{12} respectively).

The red dot represents the center of mass of Didymain. The blue stick represents Didymoon's largest axis and its orientation in the 3D space During the time $\Delta t = t_1 - t_0$ Didymain's center of mass moves from P_{01} to P_{11}, while Didymoon's center of mass moves from P_{02} to P_{12}. Now form the vectors $\vec{a} = \overrightarrow{P_{01}P_{02}}$ and $\vec{b} = \overrightarrow{P_{11}P_{12}}$. The unit vector $\hat{u} = (\vec{a} \times \vec{b})/|\vec{a} \times \vec{b}|$ is perpendicular to the plane that the angle φ scans. In order for Didymoon to constantly face Didymain, Didymoon has to rotate at an angle φ around the vector $\vec{u}$. The angle φ can be calculated from the dot product of the vectors $\vec{a}$ and $\vec{b}$

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos\varphi \Rightarrow \cos\varphi = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$$

One can decompose the rotation in two sub-rotations (one azimuthal and one polar), but it can also be done in one rotation. We can use Rodrigues rotation formula to construct the rotation matrix that rotates by an angle φ around the predefined unit vector $\hat{u} = (u_x, u_y, u_z)$. Letting

$$W = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 1 \end{bmatrix}$$

the Rodrigues rotation matrix is constructed as

$$R_u = I + (\sin\varphi)W + (1 - \cos\varphi)W^2$$

where I is the 3×3 identity matrix. After that we can perform Didymoon's rotation, pretty much like we did with Didymain's case. Didymoon consists of N_2 point masses, each at posistion (x_{2i}, y_{2i}, z_{2i}) with respect to the frame F_2 at $t = 0$. The rotation is achieved by multiplying all Didymoon's vertices with the rotation matrix R_u

$$\begin{bmatrix} x'_{2i} \\ y'_{2i} \\ z'_{2i} \end{bmatrix} = \underbrace{\begin{bmatrix} R_{u11} & R_{u12} & R_{u13} \\ R_{u21} & R_{u22} & R_{u23} \\ R_{u31} & R_{u32} & R_{u33} \end{bmatrix}}_{R_u} \begin{bmatrix} x_{2i} \\ y_{2i} \\ z_{2i} \end{bmatrix}$$

where $(x'_{2i}, y'_{2i}, z'_{2i})$ is the position of the i-vertex with respect to F_2 after the rotation.

3.3 Numerical Method for the ODEs

The differential equations (3.1) cannot be solved analytically and therefore we have to implement a numerical method to serve our purpose. Runge-Kutta 4_{th} order method (RK4) combines sufficient accuracy and computational speed, hence we will use it for the solution of the differential equations. Below we present the RK4 method for the case of one differential equation with one independant variable. Afterwards, we adapt the method to our differential equations (3.1).

Let an initial value problem (IVP) be specified as follows:

$$\dot{x} = f(t, x), \quad x(t_0) = x_0$$

Here x is an unknown function (scalar or vector) of time t, which we would like to approximate. The function f and the numbers t_0 and x_0 are known. Provided a sufficiently short step $h > 0$, the sequence (t_ν, x_ν) that approximates the function $x(t)$ is given as:

$$x(t_{\nu+1}) \equiv x_{\nu+1} = x_\nu + \frac{1}{6}(k + 2l + 2m + n)$$

$$t_{\nu+1} = t_\nu + h$$

for $\nu = 0, 1, 2, \ldots$, where

$$\begin{aligned} k &= hf(t_\nu, x_\nu) \\ l &= hf(t_\nu + \frac{h}{2}, x_\nu + \frac{k}{2}) \\ m &= hf(t_\nu + \frac{h}{2}, x_\nu + \frac{l}{2}) \\ n &= hf(t_\nu + h, x_\nu + m) \end{aligned}$$

Now reconsider equations (3.1). It is a system of 3 ODEs of second order. In order to directly apply RK4 method we would like to reduce the order from 2 to 1. Thus we do the following substitution: $\dot{X}_1 = V_{X_1}$, $\dot{Y}_1 = V_{Y_1}$, $\dot{Z}_1 = V_{Z_1}$. So now we have the following systen of 6 ODEs of first order:

$$\begin{aligned} \dot{X}_2 &= V_{X_2} \\ \dot{Y}_2 &= V_{Y_2} \\ \dot{Z}_2 &= V_{Z_2} \\ \dot{V}_{X_2} &= Gm_1 \sum_{i=1}^{N_1} \frac{X_1 - X_2 + x_{1i}}{d_{21i}^3} \\ \dot{V}_{Y_2} &= Gm_1 \sum_{i=1}^{N_1} \frac{Y_1 - Y_2 + y_{1i}}{d_{21i}^3} \\ \dot{V}_{Z_2} &= Gm_1 \sum_{i=1}^{N_1} \frac{Z_1 - Z_2 + z_{1i}}{d_{21i}^3} \end{aligned} \tag{3.3}$$

We will write down the RK4 method for a general system of 6 ODEs of first order and then we will apply it to the equations (3.3). Consider the following initial value problem:

$$\begin{aligned} \dot{x}_1 &= f_1(t, x_1, x_2, \ldots, x_6), \quad x_1(t_0) = x_{10} \\ \dot{x}_2 &= f_2(t, x_1, x_2, \ldots, x_6), \quad x_2(t_0) = x_{20} \\ &\vdots \\ \dot{x}_6 &= f_6(t, x_1, x_2, \ldots, x_6), \quad x_6(t_0) = x_{60} \end{aligned}$$

Here $x_1, x_2, \ldots, x_6$ are unknown functions of time t, which we would like to approximate. The functions $f_1, f_2, \ldots, f_6$ and the numbers t_0 and $x_{10}, x_{20}, \ldots, x_{60}$ are known. Provided a sufficiently

short step $h > 0$, the sequences $(t_\nu, x_{1,\nu}), (t_\nu, x_{2,\nu}), \ldots, (t_\nu, x_{6,\nu})$ that approximate the functions $x_1, x_2, \ldots, x_6$ are given as:

$$x_1(t_{\nu+1}) \equiv x_{1,\nu+1} = x_{1,\nu} + \frac{1}{6}(k_1 + 2l_1 + 2m_1 + n_1)$$
$$x_2(t_{\nu+1}) \equiv x_{2,\nu+1} = x_{2,\nu} + \frac{1}{6}(k_2 + 2l_2 + 2m_2 + n_2)$$
$$\vdots$$
$$x_6(t_{\nu+1}) \equiv x_{6,\nu+1} = x_{6,\nu} + \frac{1}{6}(k_6 + 2l_6 + 2m_6 + n_6)$$

$$t_{\nu+1} = t_\nu + h$$

for $\nu = 0, 1, 2, \ldots$,6, where

$$k_1 = hf_1(t_\nu, x_{1,\nu}, x_{2,\nu}, \ldots, x_{6,\nu})$$
$$k_2 = hf_2(t_\nu, x_{1,\nu}, x_{2,\nu}, \ldots, x_{6,\nu})$$
$$\vdots$$
$$k_6 = hf_6(t_\nu, x_{1,\nu}, x_{2,\nu}, \ldots, x_{6,\nu})$$

$$l_1 = hf_1(t_\nu + \frac{h}{2}, x_{1,\nu} + \frac{k_1}{2}, x_{2,\nu} + \frac{k_2}{2}, \ldots, x_{6,\nu} + \frac{k_6}{2})$$
$$l_2 = hf_2(t_\nu + \frac{h}{2}, x_{1,\nu} + \frac{k_1}{2}, x_{2,\nu} + \frac{k_2}{2}, \ldots, x_{6,\nu} + \frac{k_6}{2})$$
$$\vdots$$
$$l_6 = hf_6(t_\nu + \frac{h}{2}, x_{1,\nu} + \frac{k_1}{2}, x_{2,\nu} + \frac{k_2}{2}, \ldots, x_{6,\nu} + \frac{k_6}{2})$$

$$m_1 = hf_1(t_\nu + \frac{h}{2}, x_{1,\nu} + \frac{l_1}{2}, x_{2,\nu} + \frac{l_2}{2}, \ldots, x_{6,\nu} + \frac{l_6}{2})$$
$$m_2 = hf_2(t_\nu + \frac{h}{2}, x_{1,\nu} + \frac{l_1}{2}, x_{2,\nu} + \frac{l_2}{2}, \ldots, x_{6,\nu} + \frac{l_6}{2})$$
$$\vdots$$
$$m_6 = hf_6(t_\nu + \frac{h}{2}, x_{1,\nu} + \frac{l_1}{2}, x_{2,\nu} + \frac{l_2}{2}, \ldots, x_{6,\nu} + \frac{l_6}{2})$$

$$n_1 = hf_1(t_\nu + \frac{h}{2}, x_{1,\nu} + m_1, x_{2,\nu} + m_2, \ldots, x_{6,\nu} + m_6)$$

$$n_2 = hf_2(t_\nu + \frac{h}{2}, x_{1,\nu} + m_1, x_{2,\nu} + m_2, \ldots, x_{6,\nu} + m_6)$$

$$\vdots$$

$$n_6 = hf_6(t_\nu + \frac{h}{2}, x_{1,\nu} + m_1, x_{2,\nu} + m_2, \ldots, x_{6,\nu} + m_6)$$

By substituting the theoretical functions $x_1, x_2, x_3, x_4, x_5, x_6$ with the functions of the position (X_2, Y_2, Z_2) and the velocity $(V_{X_2}, V_{Y_2}, V_{Z_2})$ of our problem, we solve the equations.

3.4 Units of Measurement

Instead of measuring our quantities in the SI, we define a new system of measurement, named TU. In the new system we assume $G = M = 1$, where G is the gravitational constant and $M = M_1 + M_2$ is the total mass of Didymos binary (M_1 is the mass of Didymain and M_2 is the mass of Didymoon). We also choose kilometers to be our unit of length because the size of the asteroids and the distances we will deal with are of the order of kilometer. Assuming the previous means that the time t_{TU} in the TU system is different than the one of the SI (t_{sec}). We want to find the relationship between t_{TU} and t_{sec}. We use the scalar form of Newton's gravtiational law:

$$\cancel{m}\frac{d^2r}{dt_{TU}^2} = \frac{G\cancel{m}M}{r^2} \Rightarrow r^2\frac{d^2r}{dt_{TU}^2} = GM \tag{3.4}$$

We choose $G = M = 1$ and km as a unit of length, thus equation (3.4) yields

$$\left([km]^2 \cdot 10^6\right)\frac{[km] \cdot 10^{-3}}{t_{TU}^2} = 1 \Rightarrow \frac{[km]^3}{t_{TU}^2} = 10^9 \tag{3.5}$$

In the SI, equation (3.5) takes the form

$$[m]^2\frac{[m]}{t_{sec}^2} = 6.67408 \cdot 10^{-11}\frac{[m]^3}{[kg][sec]^2} \cdot 5.278 \cdot 10^{11}[kg] \Rightarrow$$

$$\frac{[m]^3}{t_{sec}^2} = 35.22579424\frac{[m]^3}{[sec]^2} \Rightarrow \frac{[km]^3}{t_{sec}^2} = 35.22579424\frac{[km]^3}{[sec]^2} \tag{3.6}$$

$$(3.5)\&(3.6) \Rightarrow \frac{[km]^3/t_{TU}^2}{[km]^3/t_{sec}^2} = \frac{10^9}{35.22579424} \Rightarrow$$

$$\frac{t_{sec}}{t_{TU}} = +\sqrt{\frac{10^9}{35.22579424}} \Rightarrow t_{sec} \approx 5328.066\, t_{TU} \tag{3.7}$$

Equation (3.7) can be used to convert the time from the TU to the SI and the opposite. For example 1 TU = 5328.066 sec.

3.5 Initial Conditions and Parameters

We have to set some appropriate initial conditions for the binary to evolve. Table (1.1) informs us of some orbital elements of the system (though some error), which we will use to set the initial conditions in Cartesian form. The center of mass of the system will be at $\mathrm{O}(0,0,0)$ with respect to F_0. We know that the distance between the center of the primary and the secondary is 1.18 km $+0.04/-0.02$ km and that the orbital eccentricity is $e \leq 0.03$. Also we assume the following: 1) The orbital inclination of the secondary is $i = 0^o$ 2) The obliquity of both the primary and the secondary principal axes with respect to the mutual orbital plane is zero. Assuming the previous, the initial conditions of the center of mass of Didymoon can be written the following form:

$$\vec{R}_2 = (X_2, 0, 0) \quad \text{and} \quad \vec{V}_{R_2} = (0, V_{Y_2}, 0)$$

where $X_2 = 1.18$ km with respect to F_0 and V_{Y_2} has such a value that the orbital eccentricity of Didymoon that is calculated from the previous Cartesian coordiantes is $e \leq 0.03$. The simplest case we can think of, is $e = 0$ which corresponds to circular orbit. Theory suggests that the velocity of the circular orbit is calculated as $|V_{Y_2}| = |\sqrt{GM/X_2}| \approx 0.921$ km/TU. Didymoon is at a retrograde orbit, thus $V_{Y_2} \approx -0.921$ km/TU with respect to F_0. Provided Didymoon's aforementioned initial conditions and provided that the center of mass of the system is at $\mathrm{O}(0,0,0)$, we can find Didymain's center of mass initial conditions from equations (3.2). Also the initial orientations of the two asteroids must be set. Those will be such that all x-axes (consequently all y-axes and all z-axes) of the frames F_0,F_1,F_2 are parallel. Luckily, we don't need to incorporate new angle initial parameters to achieve that, because the asteroids are at those orientations from the modeling procedure. Didymain's rotational period is $T_1 = 2.26 \pm 0.0001$ h. In the SI it is $T_{1,sec} = 2.26 \cdot 3600 \approx 8136$ sec and in the TU it is $T_{1,TU} = T_{1,sec}/5328.066 = 1.5270081$ TU. Thus the angular velocity $\vec{\omega}_1 = -\omega_1 \hat{z}$ which we will use in order to rotate Didymain is:

$$\omega_1 = \frac{2\pi}{T_{1,TU}} = \frac{2\pi}{1.5270081} = 4.1147\, \frac{\mathrm{rad}}{\mathrm{TU}}$$

Under the assumption that the two asteroids have the same bulk density, the mass ratio of Didymoon (M_2) and Didymain (M_1) can be calculated as:

$$\frac{M_2}{M_1} = \frac{\rho V_2}{\rho V_1} = \frac{\frac{4\pi D_2^3}{3}}{\frac{4\pi D_1^3}{3}} = \frac{D_2^3}{D_1^3} \approx 0.0093$$

where D_1 and D_2 are the formal diameters of the two asteroids. We also assumed that $M_1+M_2 = 1$. Thus solving the system, we receive $M_1 = 0.9907$ and $M_2 = 0.0093$.

We perform three simulations. All simulate the system for a physical time $t_{max} = 2$ months (in terms of TU, this means $t_{max} = 972.96$ TU). The difference lies in the time step being used in the numerical integrations. The first simulation uses a relatively short time step $\Delta t = 2.66$ sec (in terms of TU, this means $\Delta t = 0.0005$ TU). The second simulation uses a longer time step $\Delta t = 26.64$ sec (in terms of TU, this means $\Delta t = 0.005$ TU). The third simulation uses a much longer time step $\Delta t = 266.40$ sec / 4.44 min (in terms of TU, this means $\Delta t = 0.05$ TU).

3.6 Simulation Results

Case: Short step

Figure (3.3) depicts the binary at the end of the simulation ($\Delta t = 2.66$ seconds and $t = t_{max} = 2$ months) from three different perspectives. Also, in figure (3.4) we present the plots of Didymoon's semi-major axis, eccentricity and inclination as functions of time. It seems that after two months, Didymoon's orbit remains quite stable. The functions $e(t)$ and $i(t)$ oscillate quite steadily, just as someone would expect in reality (remember that no matter how short the integration time step is, one should not expect absolutely constant eccentricity and inclination because we are not dealing with the classic two body problem, but instead two assymetric rigid bodies. Thus, the orbital elements are variables due to the nature of the problem). However the function $a(t)$ seems to suffer drag as well. This (which is not expected to happen in reality) is an error incorporated by the Runge-Kutta 4th order method.

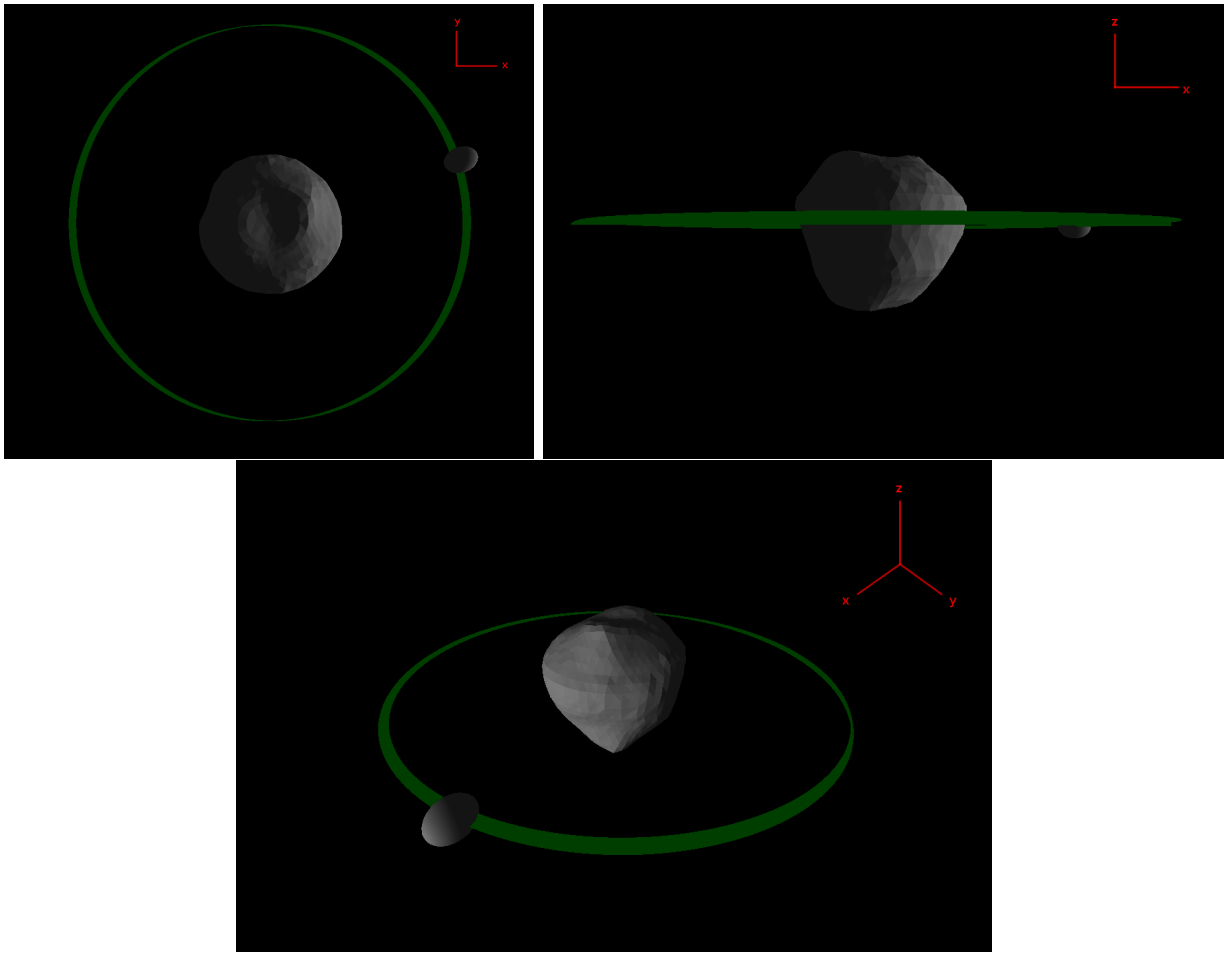


Figure 3.3: Orbit evolution of the binary at $t = t_{max} = 2$ months with step $\Delta t = 2.66$ sec.

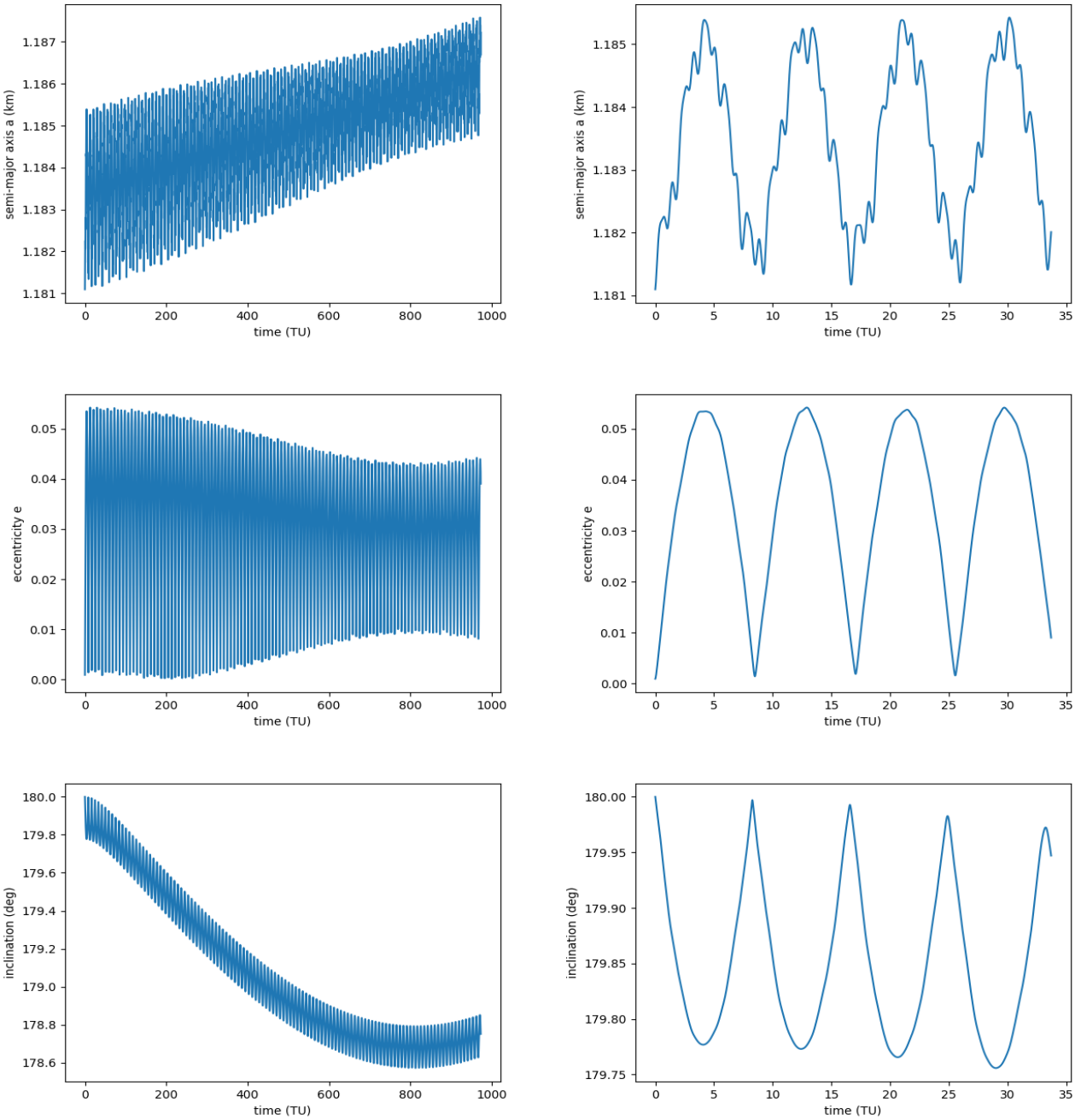


Figure 3.4: Semi-major axis a, eccentricity e and inclination i of Didymoon as functions of time for step $\Delta t = 2.66$ sec. The left panels show the evolution of the orbital elements for all the run time (2 months). The right panels show the evolution of the very same elements, but for a short amount of time (≈ 2.07 days).

Case: Middle step

Figure (3.5) depicts the binary at the end of the simulation ($\Delta t = 26.64$ seconds and $t = t_{max} = 2$ months) from three different perspectives. Also, in figure (3.6) we present the plots of Didymoon's semi-major axis, eccentricity and inclination as functions of time. Even though we have increased the method's time step by one order of magnitude, the orbit still remains quite stable. The functions $e(t)$ and $i(t)$ are almost the same with the ones of the previous case. The semi-major axis $a(t)$ has the same morphology with the corresponding one of the previous case, but now the drag is more obvious.

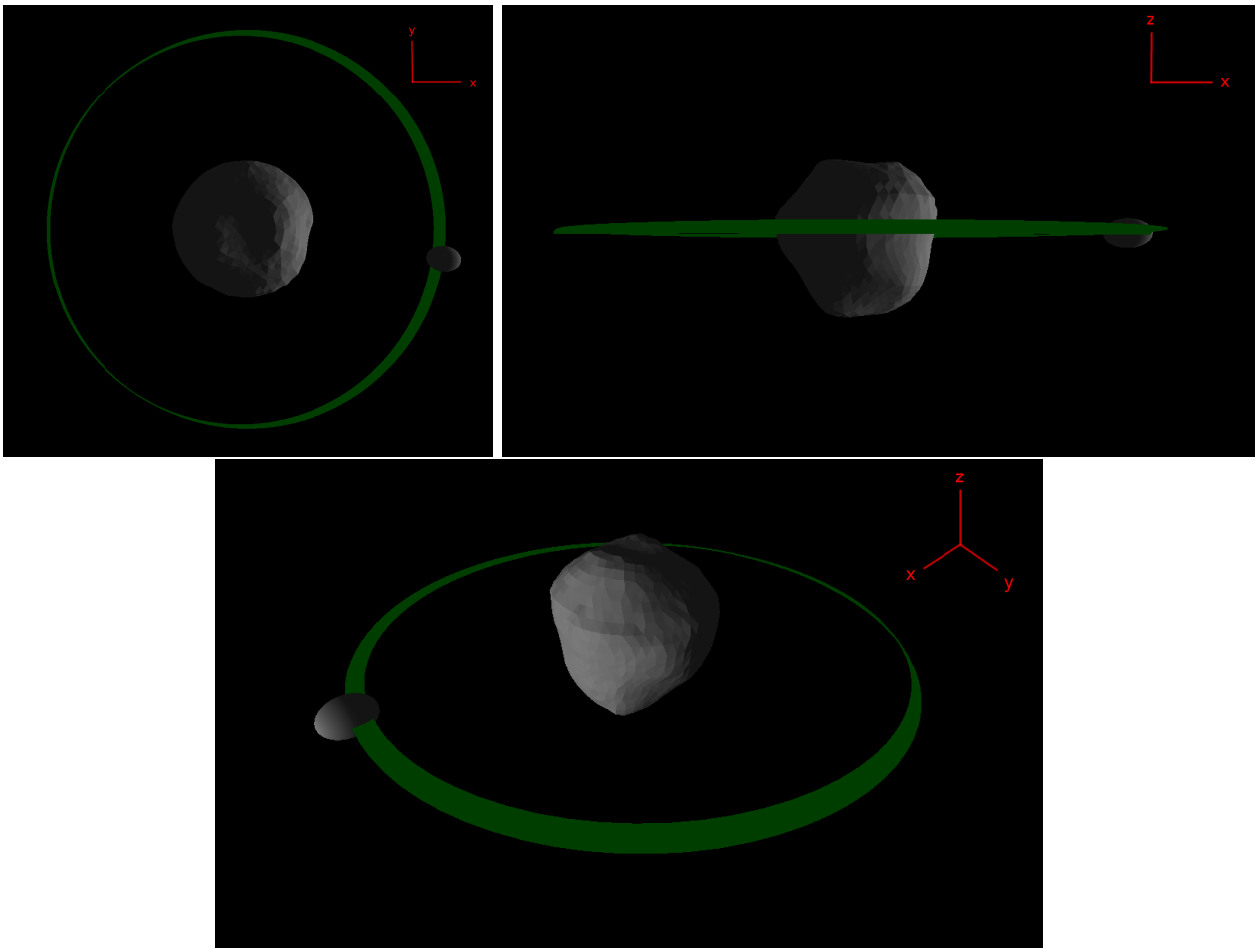


Figure 3.5: Orbit evolution of the binary at $t = t_{max} = 2$ months with step $\Delta t = 26.64$ sec.

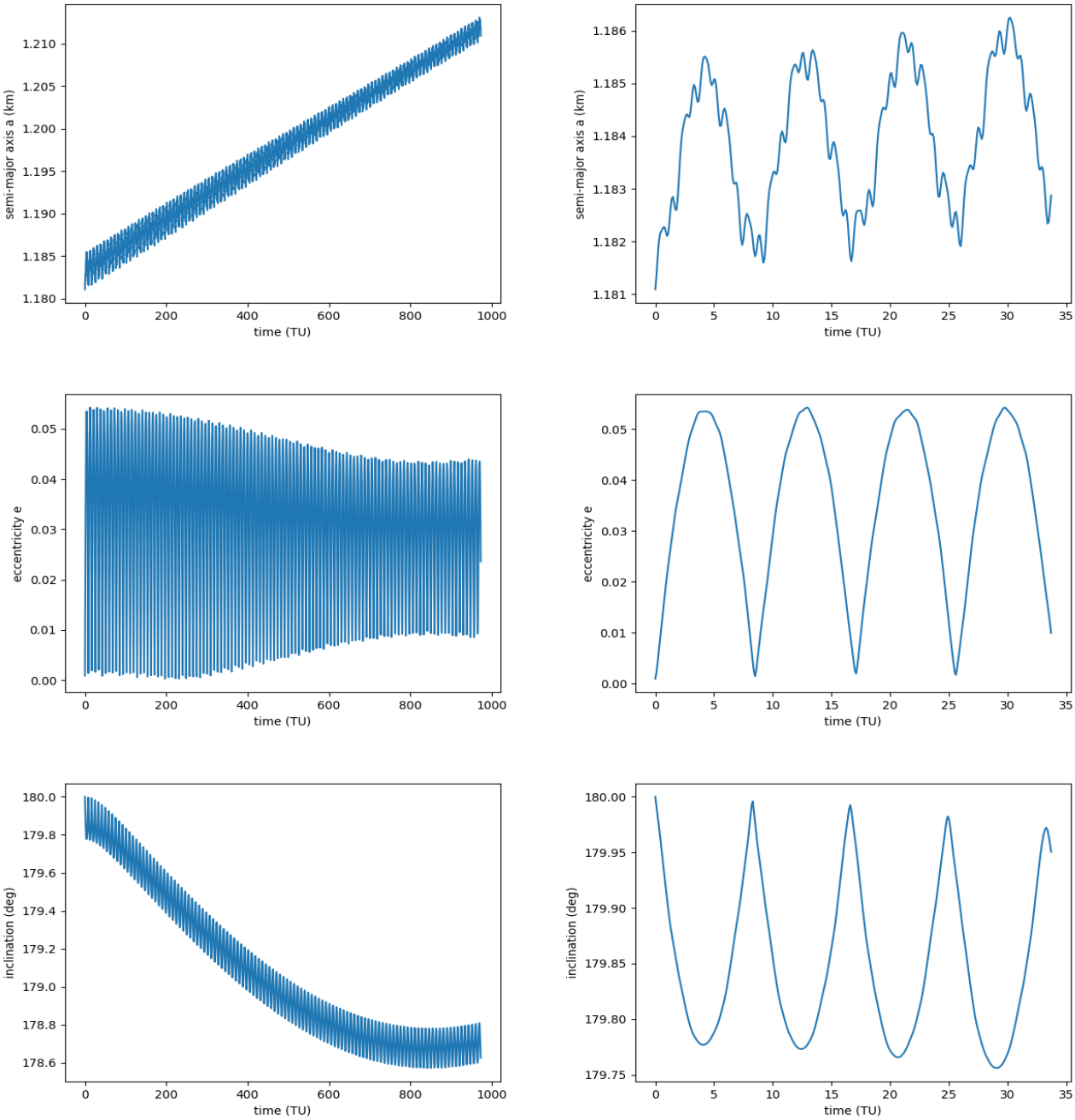


Figure 3.6: Semi-major axis a, eccentricity e and inclination i of Didymoon as functions of time for step $\Delta t = 26.64$ sec. The left panels show the evolution of the orbital elements for all the run time (2 months). The right panels show the evolution of the very same elements, but for a short amount of time (≈ 2.07 days).

Case: Long step

Figure (3.7) depicts the binary at the end of the simulation ($\Delta t = 266.4$ seconds and $t = t_{max} = 2$ months) from three different perspectives. Also, in figure (3.8) we present the plots of Didymoon's semi-major axis, eccentricity and inclination as functions of time. Again the functions $e(t)$ and $i(t)$ oscillate almost around the same values as in the previous two cases, but now we observe a significant drag of the semi-major axis $a(t)$.

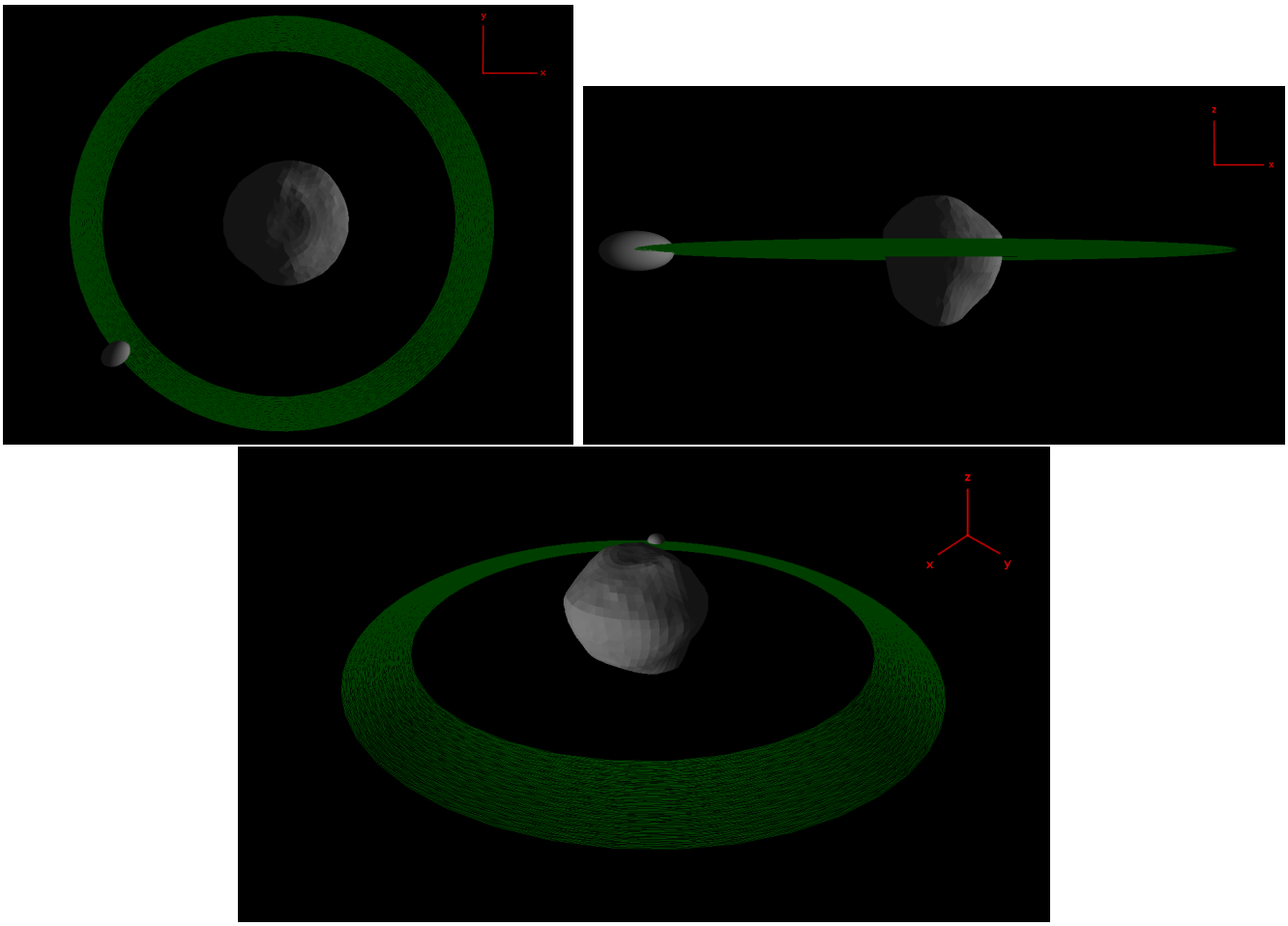


Figure 3.7: Orbit evolution of the binary at $t = t_{max} = 2$ months with step $\Delta t = 266.4$ sec.

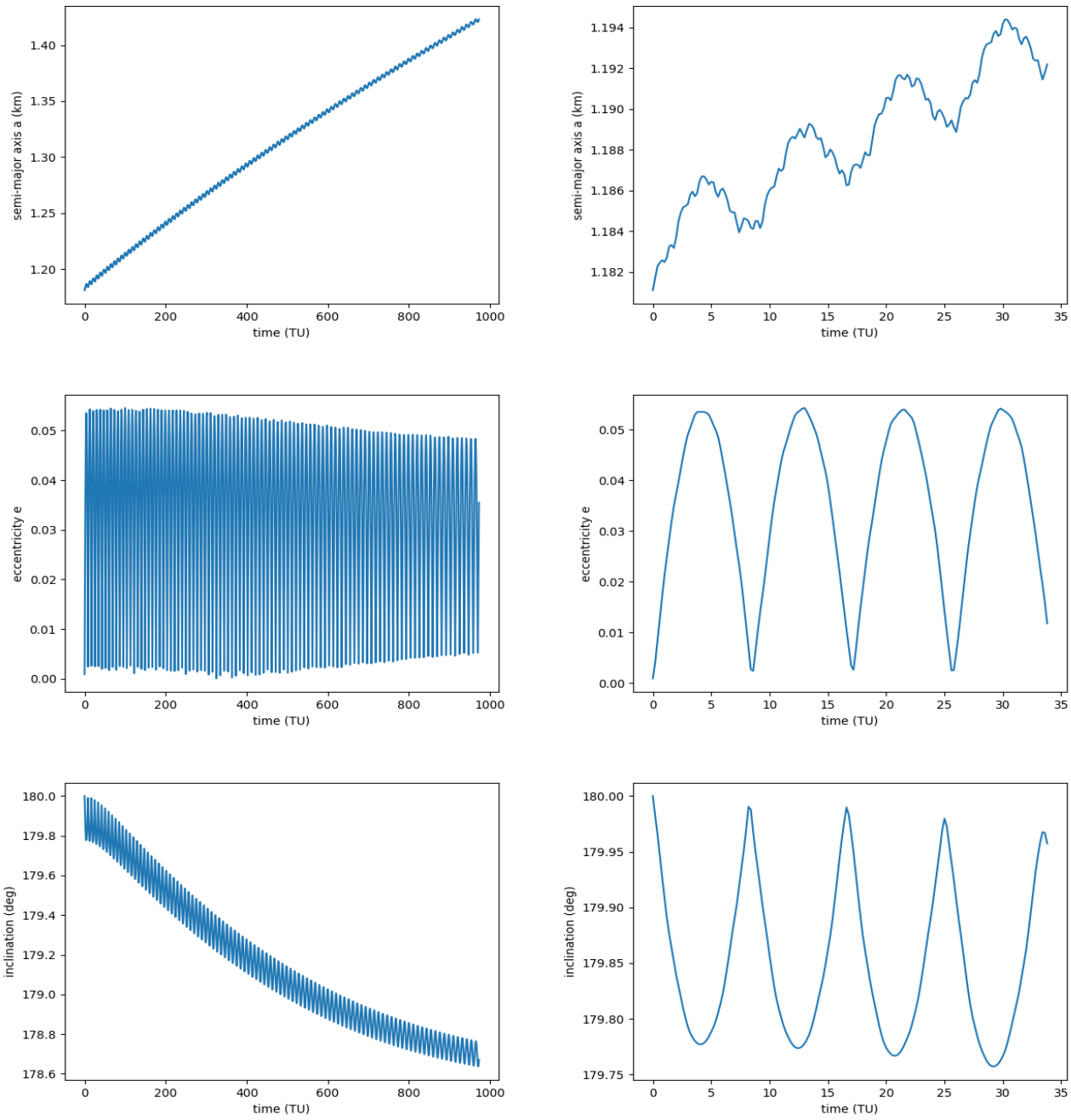


Figure 3.8: Semi-major axis a, eccentricity e and inclination i of Didymoon as functions of time for step $\Delta t = 266.4$ sec. The left panels show the evolution of the orbital elements for all the run time (2 months). The right panels show the evolution of the very same elements, but for a short amount of time (≈ 2.07 days).

Chapter 4

Impact Ejecta

Our final step is to incorporate in our simulation the impact ejecta that will be produced from Didymoon's surface due to DART's collsion. In our study we focus on the mechanical environment near the binary (up to 20 km radius from the center of mass of the system) **after** DART's collision. That is, we don't include DART's collision itself in our simulation, but only what happens after the collision. Besides, even if we did include in our simulation the collision, although there would be a slight change in the orbital elements of the system, it would barely affect the ejecta orbits. As soon as the collision occurs, we expect an ejecta cloud to be produced and spread in space. In our simulation, the cloud is assumed to consist of N point masses. Also we note that we are interested in the study of the **low velocity ejecta**, that is, the particles of which their ejection velocity is up to the escape velocity of Didymos binary (we explain further details about that in section (4.3)). Each ejected particle will be subjected to the combined gravity of Didymoon and Didymain and will follow its own trajectory in space. It is worth noting that in reality, not only the gravity of the two asteroids is applied to the ejecta, but also the gravity of other celestial bodies like the Sun and the other planets, as well as the radiation pressure of the Sun. In this thesis however, we reduce our study to the gravity of the two asteroids which is the dominant force. In this chapter we formulate the equations of motions of the ejecta, we implement collision and escape criterions and we set some initial conditions and parameters. Our goal is to observe the evolution of the ejecta cloud for some time and eventually conclude its fate.

4.1 Equations of Motion of the Ejecta

We assume that all ejected bodies are point masses that do not affect the motion of the binary. Each point mass undergoes the gravitational force of all the particles that make up Didymain plus the gravitational force of all the particles that make up Didymoon. Figure (4.1) illustrates the latter.

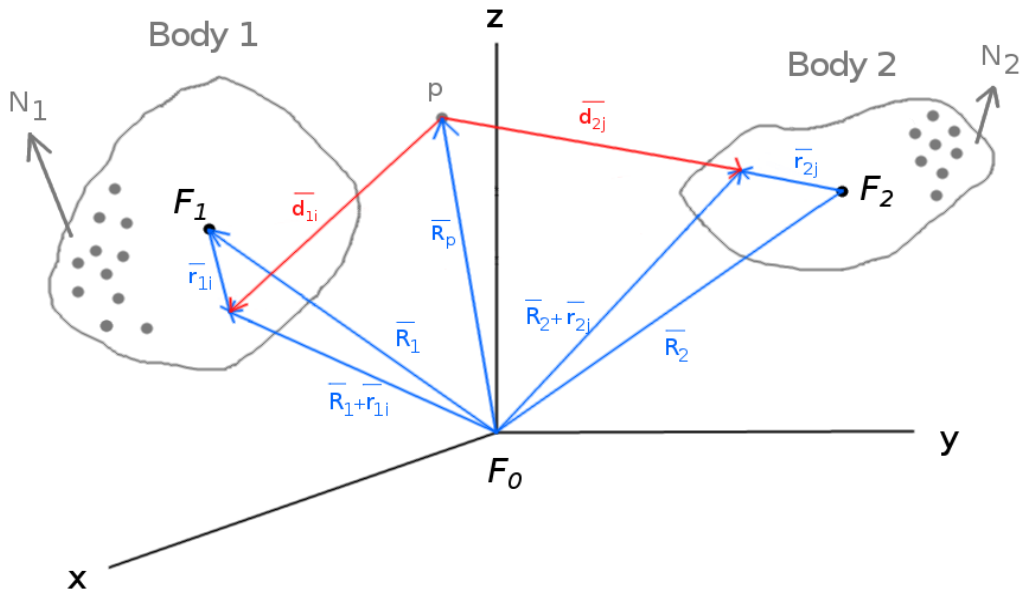


Figure 4.1: Representation of an ejected particle p, being attracted by the two asteroids.

Figure (4.1) is almost the same with figure (3.1), slightly changed to depict the mechanical environment of an ejected particle. The vectors $\vec{R}_1$, $\vec{R}_2$, $\vec{r}_{1i}$, $\vec{r}_{2j}$, represent the same with the ones of figure (3.1). Now let $\vec{R}_p$ be the position vector of an ejected particle with respect to F_0. We also define the vector $\vec{d}_{1i}$ that connects the ejected particle with an arbitrary point of Body 1 and the vector $\vec{d}_{2j}$ that connects the ejected particle with an arbitrary point of Body 2. The latter can be mathematically expressed as:

$$\vec{R}_p + \vec{d}_{1i} = \vec{R}_1 + \vec{r}_{1i} \Rightarrow \vec{d}_{1i} = \vec{R}_1 - \vec{R}_p + \vec{r}_{1i}$$

$$\vec{R}_p + \vec{d}_{2j} = \vec{R}_2 + \vec{r}_{2j} \Rightarrow \vec{d}_{2j} = \vec{R}_2 - \vec{R}_p + \vec{r}_{2j}$$

The equation of motion of an ejected particle (with negligible mass m) is:

$$\cancel{m}\ddot{\vec{R}}_p = \sum_{i=1}^{N_1} \frac{G\cancel{m}m_1}{d_{1i}^3}\vec{d}_{1i} + \sum_{j=1}^{N_2} \frac{G\cancel{m}m_2}{d_{2j}^3}\vec{d}_{2j} \Rightarrow$$

$$\ddot{\vec{R}}_p = G\left(m_1 \sum_{i=1}^{N_1} \frac{\vec{d}_{1i}}{d_{1i}^3} + m_2 \sum_{j=1}^{N_2} \frac{\vec{d}_{2j}}{d_{2j}^3}\right) \Rightarrow$$

$$\begin{cases} \ddot{X}_p = G\left(m_1 \sum_{i=1}^{N_1} \frac{X_1 - X_p + x_{1i}}{d_{1i}^3} + m_2 \sum_{j=1}^{N_2} \frac{X_2 - X_p + x_{2j}}{d_{2j}^3}\right) \\ \ddot{Y}_p = G\left(m_1 \sum_{i=1}^{N_1} \frac{Y_1 - Y_p + y_{1i}}{d_{1i}^3} + m_2 \sum_{j=1}^{N_2} \frac{Y_2 - Y_p + y_{2j}}{d_{2j}^3}\right) \\ \ddot{Z}_p = G\left(m_1 \sum_{i=1}^{N_1} \frac{Z_1 - Z_p + z_{1i}}{d_{1i}^3} + m_2 \sum_{j=1}^{N_2} \frac{Z_2 - Z_p + z_{2j}}{d_{2j}^3}\right) \end{cases}$$

where $d_{1i} = \sqrt{(X_1 - X_p + x_{1i})^2 + (Y_1 - Y_p + y_{1i})^2 + (Z_1 - Z_p + z_{1i})^2}$
and $d_{2j} = \sqrt{(X_2 - X_p + x_{2j})^2 + (Y_2 - Y_p + y_{2j})^2 + (Z_2 - Z_p + z_{2j})^2}$

We reduce the order of the ODEs from 2 to 1 in order to direclty apply RK4 method by substituting: $\dot{X}_p = V_{X_p}$, $\dot{Y}_p = V_{Y_p}$ $\dot{Z}_p = V_{Z_p}$. Thus, for one ejected particle p, we have the following system of 6 ODEs of first order:

$$\dot{X}_p = V_{X_p}$$

$$\dot{Y}_p = V_{Y_p}$$

$$\dot{Z}_p = V_{Z_p}$$

$$\dot{V}_{X_p} = G\left(m_1 \sum_{i=1}^{N_1} \frac{X_1 - X_p + x_{1i}}{d_{1i}^3} + m_2 \sum_{j=1}^{N_2} \frac{X_2 - X_p + x_{2j}}{d_{2j}^3}\right)$$

$$\dot{V}_{Y_p} = G\left(m_1 \sum_{i=1}^{N_1} \frac{Y_1 - Y_p + y_{1i}}{d_{1i}^3} + m_2 \sum_{j=1}^{N_2} \frac{Y_2 - Y_p + y_{2j}}{d_{2j}^3}\right)$$

$$\dot{V}_{Z_p} = G\left(m_1 \sum_{i=1}^{N_1} \frac{Z_1 - Z_p + z_{1i}}{d_{1i}^3} + m_2 \sum_{j=1}^{N_2} \frac{Z_2 - Z_p + z_{2j}}{d_{2j}^3}\right)$$

In our simulation we solve the differential equations of motion of the binary and at the same time we solve the differential equations of motion of the impact ejecta. Both systems will are solved with the Runge-Kutta 4th order method with the same time step Δt. The previous equations are integrated for N ejected particles. Think of it like we are integrating the motion of a cloud that consists of N bodies.

4.2 Escape, Chaos and Collision Detection

Suppose that a particle is ejected from Didymoon's surface. The particle will follow a trajectory based on its initial conditions. What happens to that particle afterwards? We enumerate 3 possible cases:

1) Escape from the binary

This case occurs either when a particle is ejected with very high initial velocity or when a particle's orbit evolves in such a way that its velocity becomes high enough to satisfy the criterion:

$$v_p \geq \underbrace{\sqrt{\frac{2G(M_1+M_2)}{r}}}_{v_{\text{esc}}} \quad \text{for} \quad r \gg 1$$

Specifically, we decided that $r_{max} = 20$ km from the center of mass of the binary. Thus, if a particle gains velocity $v_p \geq v_{\text{esc}}$ and at the same time $r \geq r_{max}$, then we consider that the particle escaped. Our algorithm constantly checks the previous criterion during each time step in order to decide whether a particle escaped from the binary or not.

2) Chaotic orbit near the binary's domain

This scenario is expected to happen when a particle is ejected with such initial conditions, that don't provide them with enough kinetic energy to escape the binary. These particles are expected to be trapped in chaotic orbits near the binary's domain of gravitational influence for who knows how long, until probably, either they collide with one of the two asteroids or escape the binary due to gravity assist. It is very important to know the long term behavior of the particles in such a case so that Hera space probe can be prepared for what it is going to meet there (whether the collision event happens before or after Hera's arrival).

3) Collision with Didymain or Didymoon

For an ejected particle orbiting the binary, it possible that after a finite amount of time, it will crash, either on Didymain's or on Didymoon's surface. We assume that after a collision between an ejected particle and an asteroid, the orbit of the particle ends at the exact point at which the collision took place. Realistically speaking, the last sentence is not true, because of possible ricochet and cloud regeneration. If for example an ejected body (a rock) has realatively big mass and velocity and it happens to crash on one of the asteroids almost tangentively, then a new ejecta cloud will rise. However, we do not include such details in our study. What we want are two collision detection criterions; one between the ejecta and Didymain and one between the ejecta Didymoon.

For Didymain, we assume a circumscribed sphere on its surface (i.e. a sphere centered at the center of mass of Didymain and radius equal to the longest distance between the center of mass of Didymain and its surface vertices). If an ejected particle crosses that (virtual) sphere, then we assume a collision with Didymain. The mathematical criterion for the last sentence is written as:

$$\sqrt{(X_1(t) - X_p(t))^2 + (Y_1(t) - Y_p(t))^2 + (Z_1(t) - Z_p(t))^2} \leq d_{max}$$

where $\vec{R}_1(t) = (X_1(t), Y_1(t), Z_1(t))$ is the position of the center of mass of Didymain at time t with respect to the global inertial frame F_0 (see figure (4.1)), $\vec{R}_p(t) = (X_p(t), Y_p(t), Z_p(t))$ is the position of an ejected particle at time t, again with respect to the global inertial frame F_0 and d_{max} is the longest distance between the center of mass of Didymain an its surface vertices.

For Didymoon, we can take advantage of the analytical expression of its surface (tri-axial ellipsoid) in order to write down a precise collision detection criterion. The equation of a tri-axial ellipsoid, centered at $O(0, 0, 0)$ with semi axes a, b, c is:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$

The equation of a self-translated tri-axial ellipsoid, centered at $P_0(x_0, y_0, z_0)$ with semi axes a, b, c is:

$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} + \frac{(z - z_0)^2}{c^2} = 1$$

The equation of a self-translated and self-rotated tri-axial ellipsoid, centered at $P_0(x_0, y_0, z_0)$, rotated at an angle φ around a vector $\vec{u}$, with semi axes a, b, c is:

$$\frac{(x' - x_0)^2}{a^2} + \frac{(y' - y_0)^2}{b^2} + \frac{(z' - z_0)^2}{c^2} = 1$$

where

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \underbrace{\begin{bmatrix} R_{u11} & R_{u12} & R_{u13} \\ R_{u21} & R_{u22} & R_{u23} \\ R_{u31} & R_{u32} & R_{u33} \end{bmatrix}}_{R_u} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Thus, a collision between Didymoon and an ejected particle p, occurs when the following criterion is satisfied

$$\frac{(X'_p - X_2)^2}{a^2} + \frac{(Y'_p - Y_2)^2}{b^2} + \frac{(Z'_p - Z_2)^2}{c^2} \leq 1$$

where

$$\begin{bmatrix} X'_p \\ Y'_p \\ Z'_p \end{bmatrix} = \underbrace{\begin{bmatrix} R_{u11} & R_{u12} & R_{u13} \\ R_{u21} & R_{u22} & R_{u23} \\ R_{u31} & R_{u32} & R_{u33} \end{bmatrix}}_{R_u} \begin{bmatrix} X_p \\ Y_p \\ Z_p \end{bmatrix}$$

and $\vec{R}_2 = (X_2, Y_2, Z_2)$ is the position vector of the center of mass of Didymoon at time t with respect to the global inertial frame F_0.

4.3 Initial Conditions of the Ejecta

We mentioned previously that we simulate the ejecta motions after the collision between DART and Didymoon. We also mentioned that we are interested in the particles that are ejected with relatively low velocities. If we were near Didymoon during the collision event, we would observe most of the ejected particles being catapulted in space with hypervelocities (depending on momentum gain parameter β) due to the violent exchange of momentum bewteen the spacecraft and Didymoon. However, there will be particles ejected in space with relatively low velocities; less than the escape velocity of the binary at the collision point. These are the ones we are interested in. Our algorithm provides the ejected particles with a set of initial conditions (positions and velocities), such that they don't escape the binary due to the initial ejection. Keep in mind that a particle can escape from the binary, not only due to an initial hypervelocity, but also due to chaotic evolution of its orbit near the binary.

We decided that all of the ejected particles shall have the same initial position. This position shall be at the impact point of Didymoon's surface and more specifically we decided that point to be at $(0, -b, 0)$ with respect to Didymoon's local coordinate system (frame F_2 of figure (4.1)). In our simulation, the initial position is not be placed exaclty on Didymoon's surface, but slightly further (1 meter). We do this in order to avoid infinities. If we do set the initial position of the ejected particles exactly on Didymoon's surface, it will result almost zero distance with some of Didymoon's model points which will lead to infinite forces. As far as the initial velocities are concerned, we make use of available experimental data (Johns Hopkins University / Applied Physics Laboratory) to see what happens when a hypervelocity aluminium projectile impacts a pumice target (figure (4.2)). We observe that the geometric distribution of the ejecta cloud right after the collision (left snapshot) is approximately a bunch of cone layers with different apex angles. After very few video frames, the cloud seems to occupy a wider region, due to the particles that are ejected with lower velocities and larger angles (right snapshot). The same principal we follow for the determination of the initial velocities in our simulation. First of all we assume that the ejecta cloud is generated in such a way that the velocity vectors of Didymoon and the impactor right before the impact, point to opposite directions. We also assume that the initial velocity vector components of all the ejecta are random numbers from the uniform distribution, but bounded in direction and magnitude so that the geometry of the ejecta approximates the right snapshot of figure (4.2). This way we achieve to cover a wide range of possible initial velocities. Figure (4.3) represents a sample of 100

initial velocity vectors calculated as predescribed. As far as the magnitudes of the initial velocities are concerned, we perform a search algorithm to calculate the initial critical velocity $v_{\rm crit}$ for which an ejected particle almost escapes the binary. This value is found to be $v_{\rm crit} \approx 1.4$ km/TU $= 0.262$ m/sec. The initial conditions of the binary are exactly the same with the ones described in section 3.5. We decided to simulate the system for a physical time $t_{max} = 1$ month with an integration time step $\Delta t = 26.64$ seconds (in terms of TU, this means $t_{max} = 486.48$ TU and $\Delta t = 0.005$ TU), which corresponds to the middle step case of section 3.6. We assume 4000 ejected particles totally.

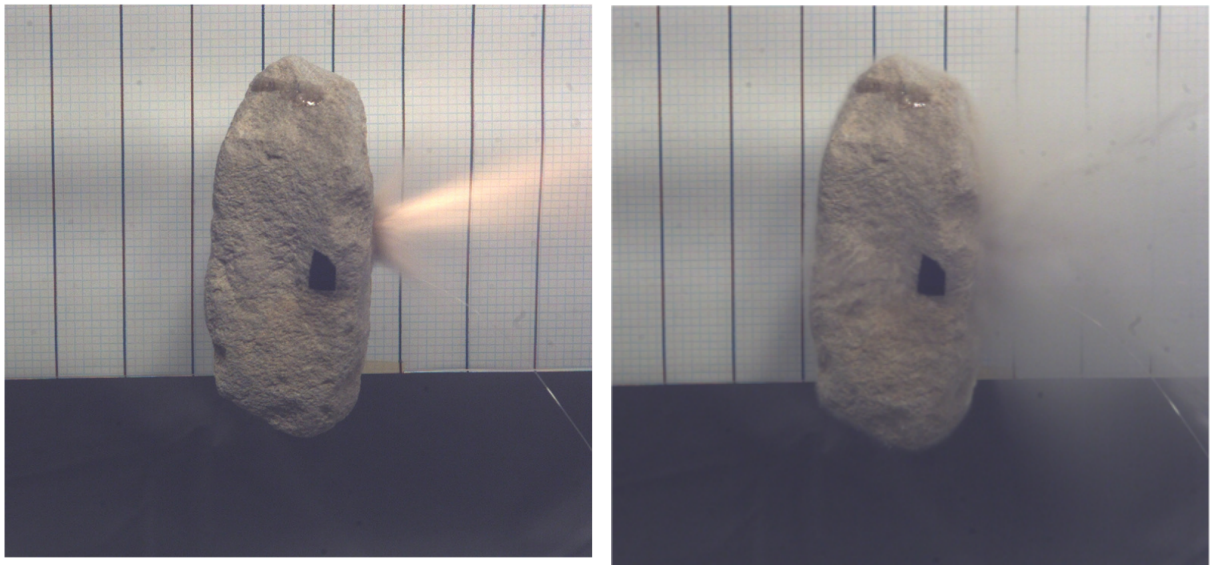


Figure 4.2: A pumice target of mass 297 gm being impacted by a $1/8^{\rm th}$ inch aluminium projectile of mass 0.0459 gm and speed 3.92 km/s. On the left picture we can see the ejecta cloud distribution right after the collision and on the right we can see the same cloud a few video frames later.

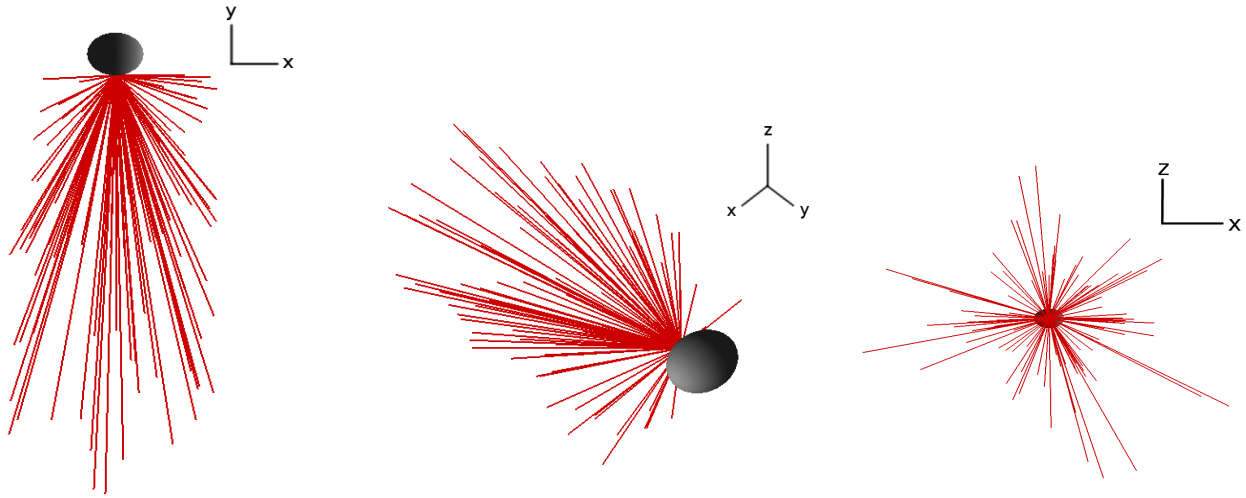
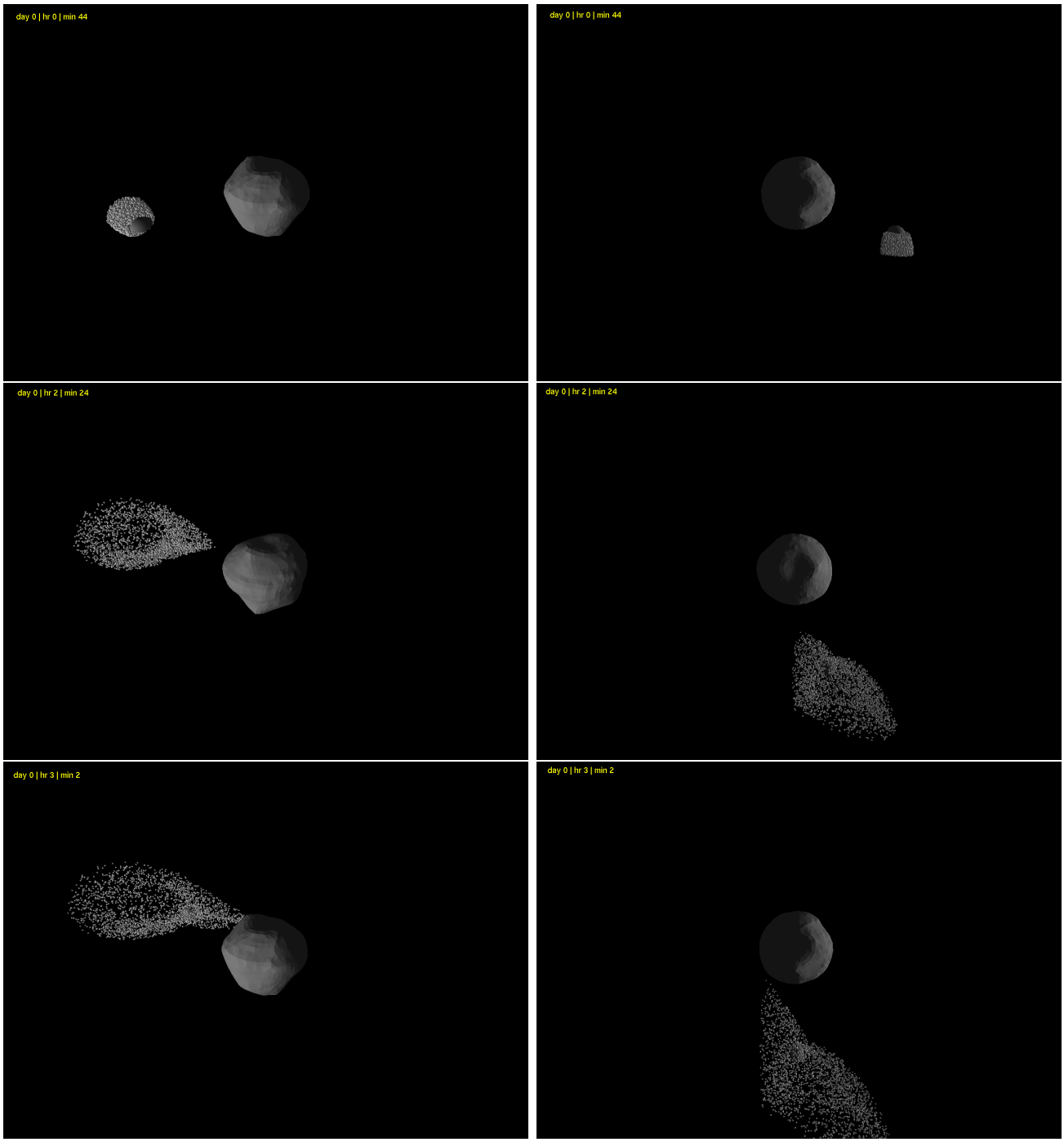
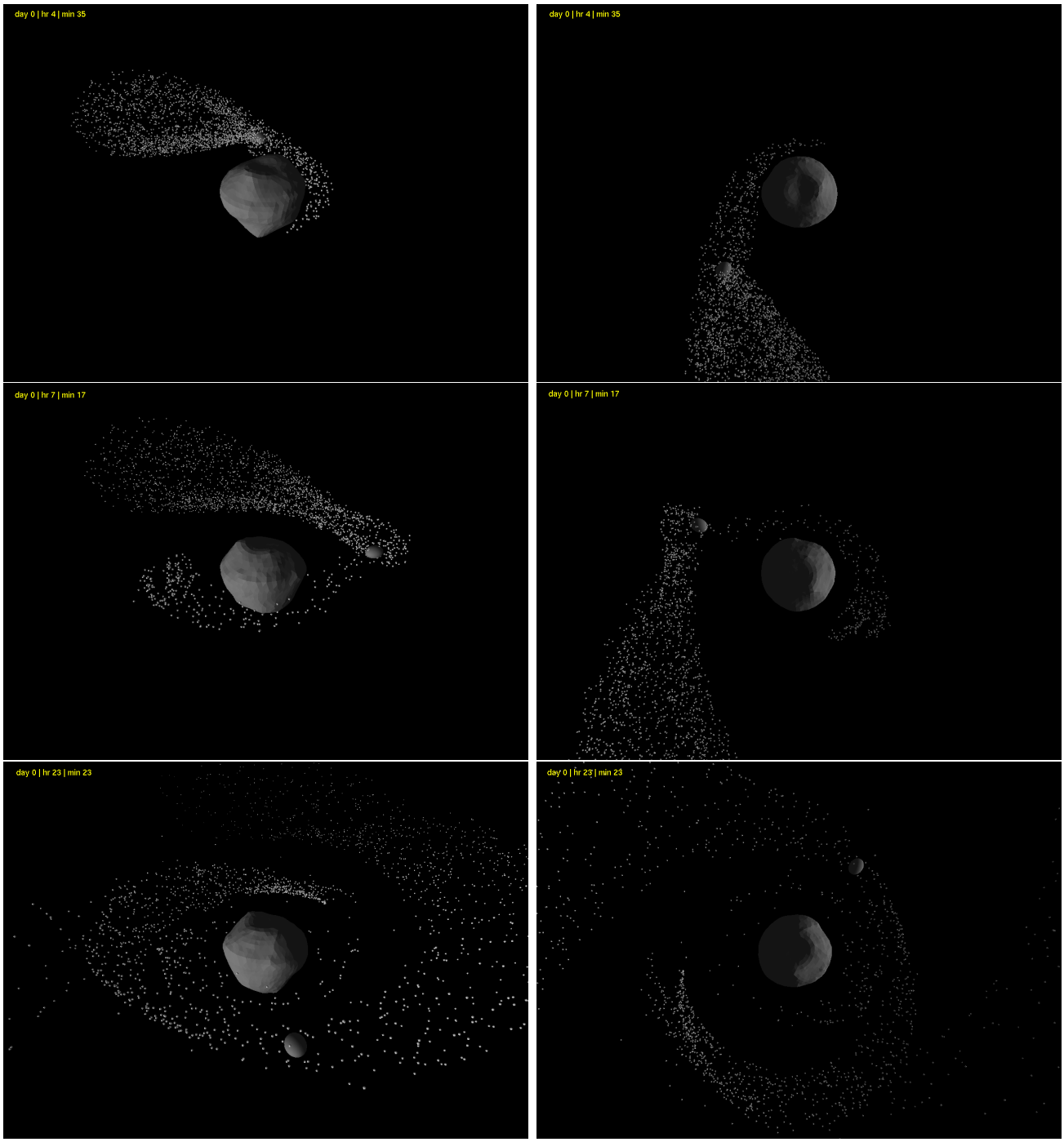


Figure 4.3: Simulated geometric distribution of 100 initial velocity vectors that correspond to low velocity ejecta. All the initial vectors are considered to be bounded in direction an magnitude in order to approximate the cloud distribution of figure (4.2). The elliposoid represents Didymoon.

4.4 Simulation Results

Below follows the evolution of the ejecta cloud at various times. Afterwards, we present the functions $r(t)$ and $v(t)$ for 21 randomly sampled ejected particles of all the 4000 that were simulated.




day 0 | hr 4 | min 35
day 0 | hr 4 | min 35
day 0 | hr 7 | min 17
day 0 | hr 7 | min 17
day 0 | hr 23 | min 23
day 0 | hr 23 | min 23

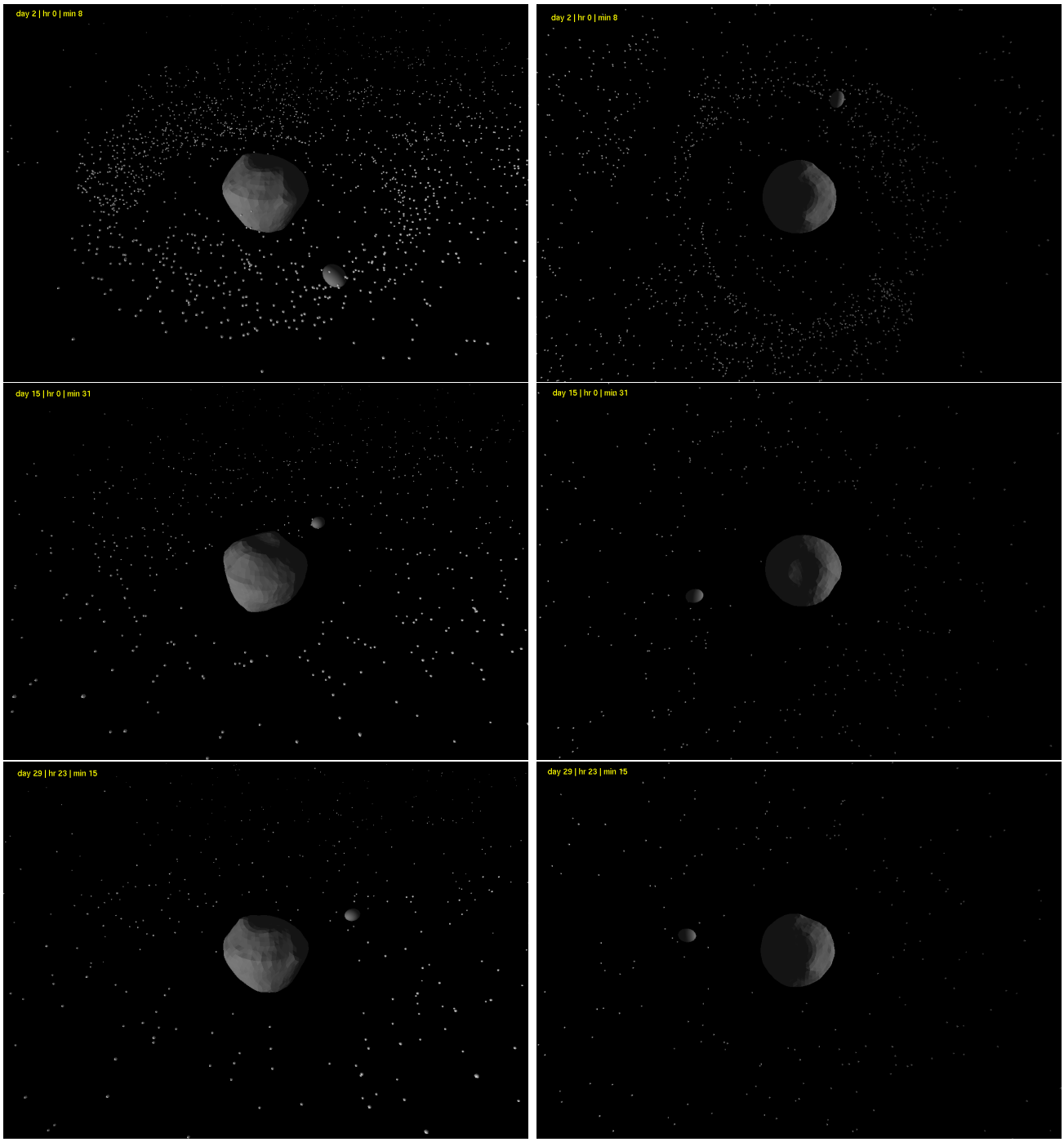
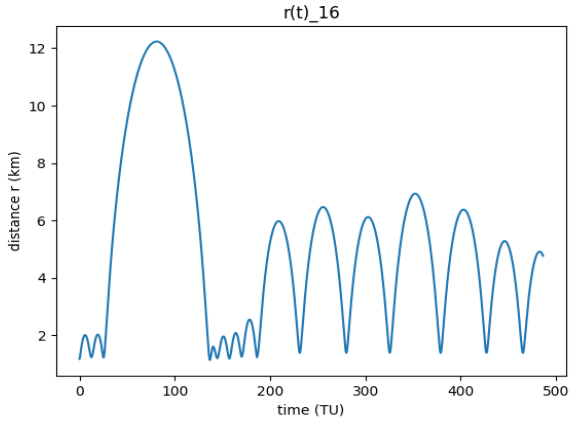
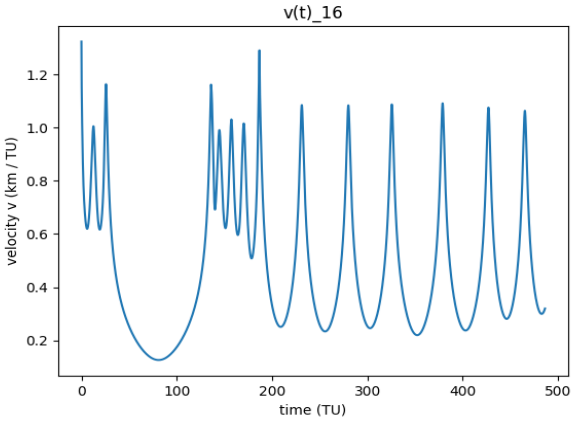
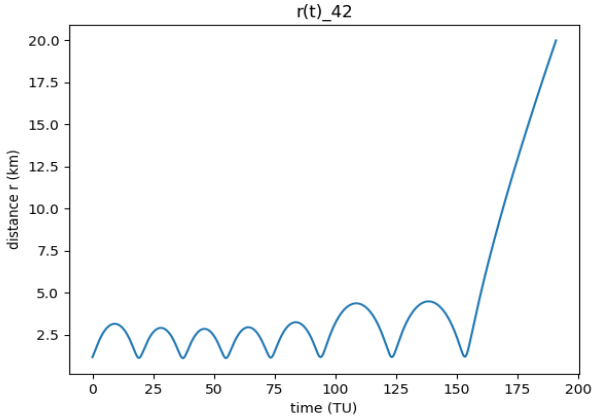
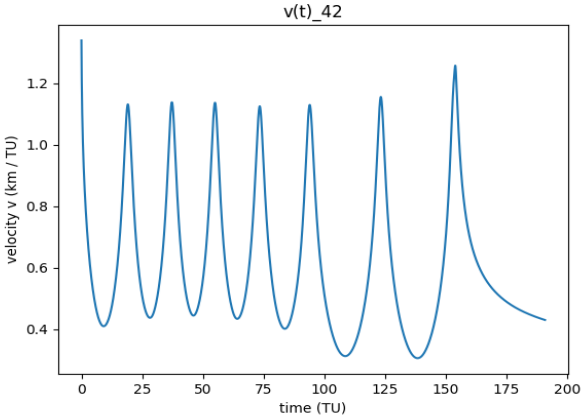


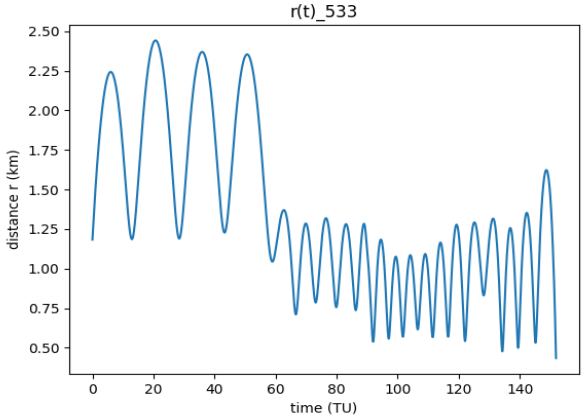
Figure 4.4: Orbital evolution of the low initial velocity ejecta for 1 month.

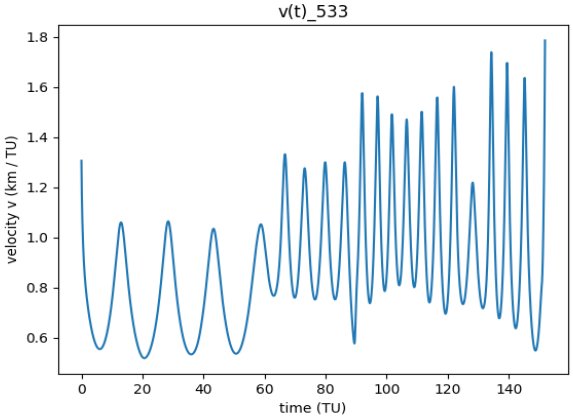

r(t)_16
distance r (km)
time (TU)

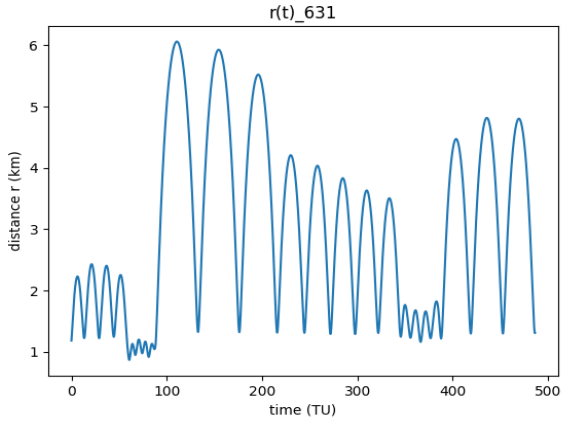

v(t)_16
velocity v (km / TU)
time (TU)

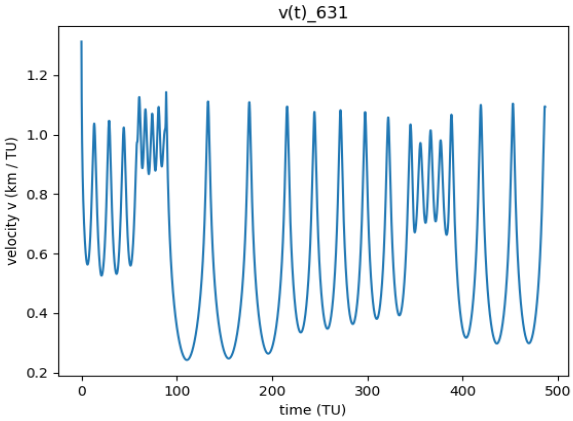

r(t)_42
distance r (km)
time (TU)

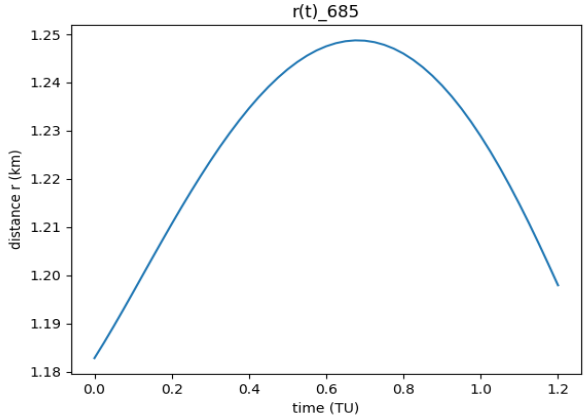

v(t)_42
velocity v (km / TU)
time (TU)

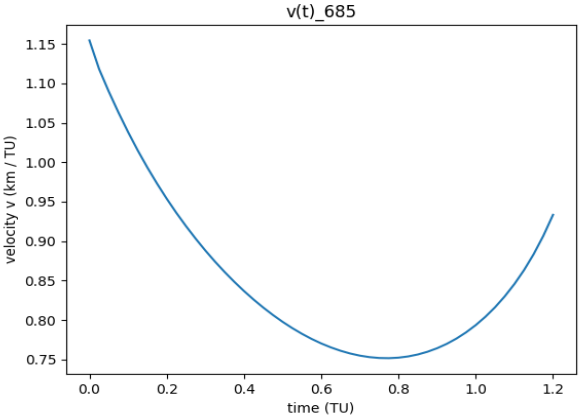

r(t)_533
distance r (km)
time (TU)

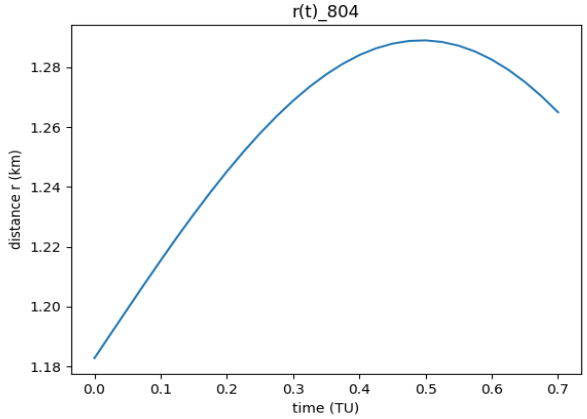

v(t)_533
velocity v (km / TU)
time (TU)

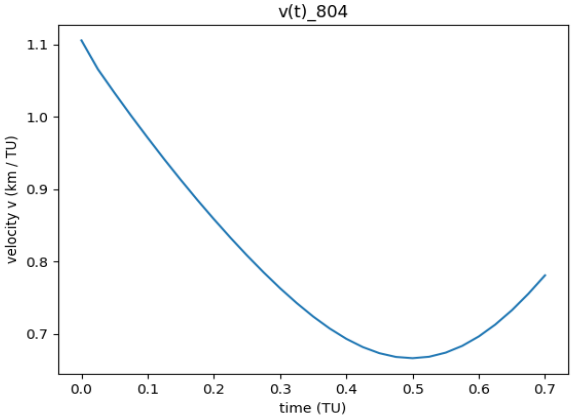

r(t)_631
distance r (km)
time (TU)

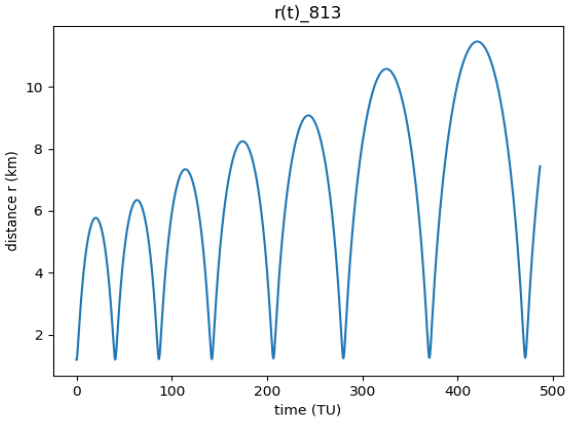

v(t)_631
velocity v (km / TU)
time (TU)

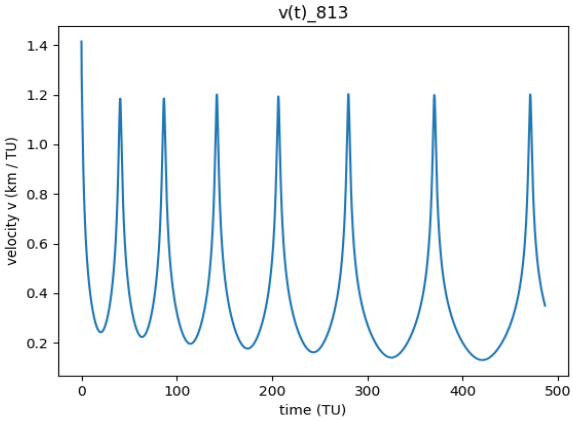

r(t)_685
distance r (km)
time (TU)

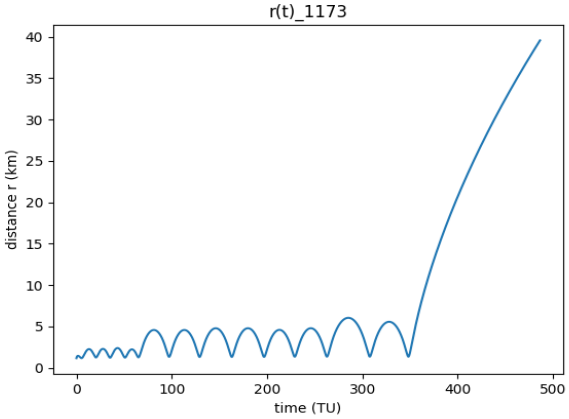

v(t)_685
velocity v (km / TU)
time (TU)

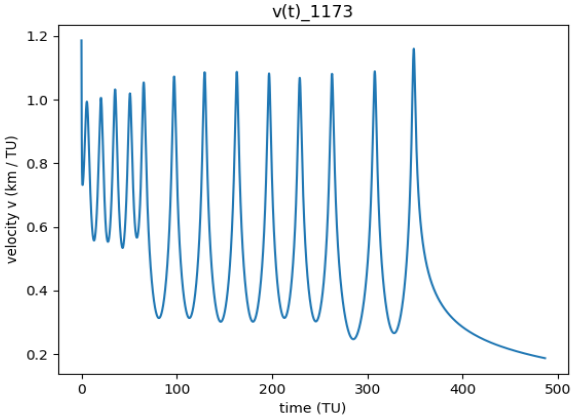

r(t)_804
distance r (km)
time (TU)

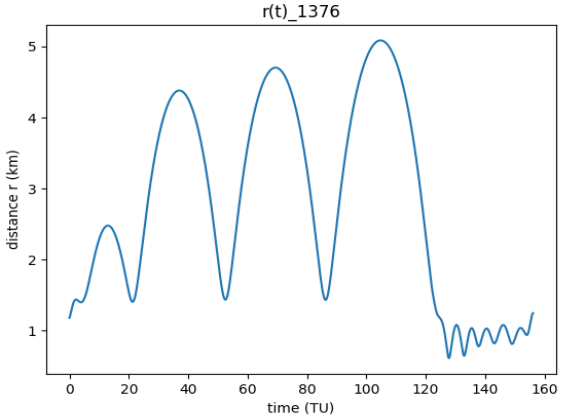

v(t)_804
velocity v (km / TU)
time (TU)

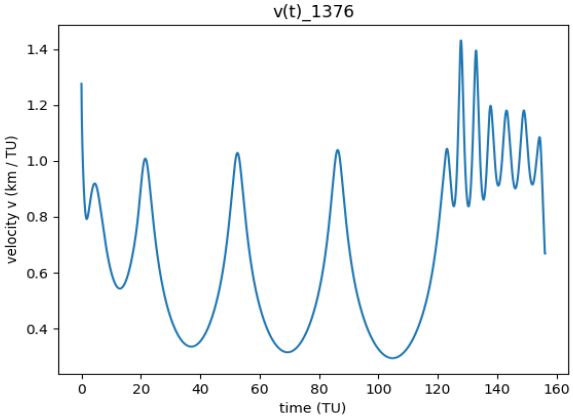

r(t)_813
distance r (km)
time (TU)

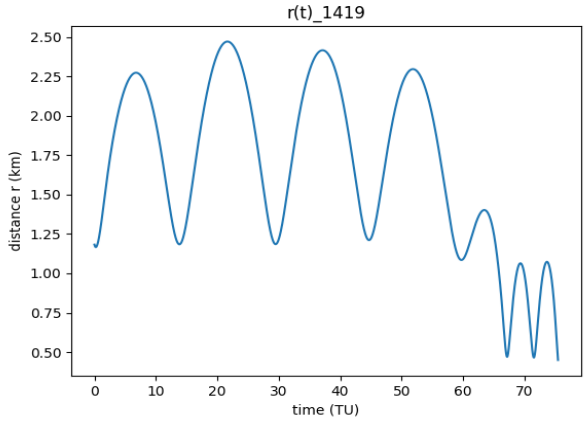

v(t)_813
velocity v (km / TU)
time (TU)

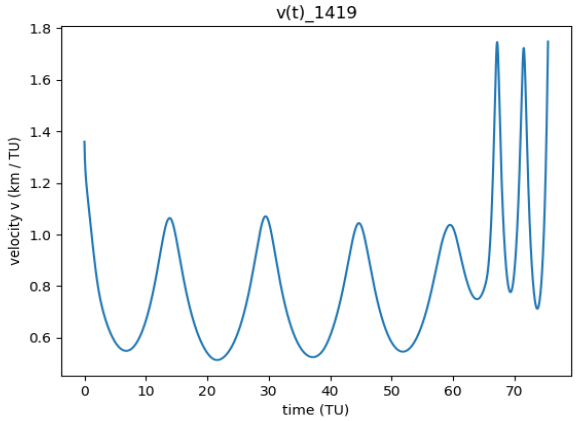

r(t)_1173
distance r (km)
time (TU)

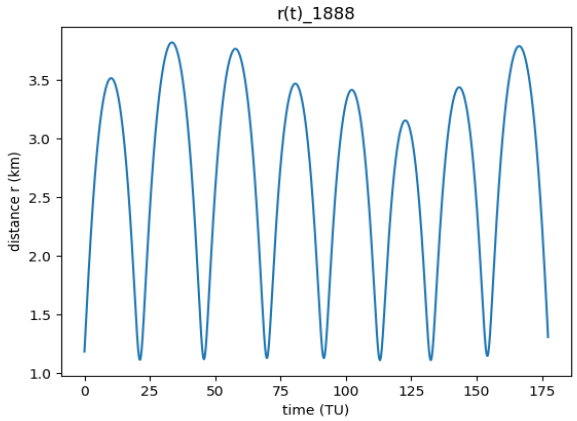

v(t)_1173
velocity v (km / TU)
time (TU)

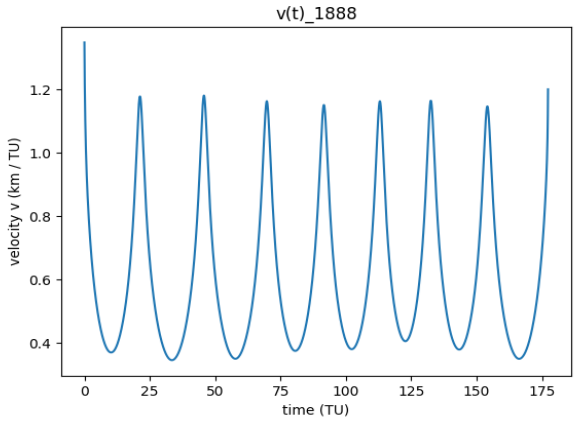

r(t)_1376
distance r (km)
time (TU)

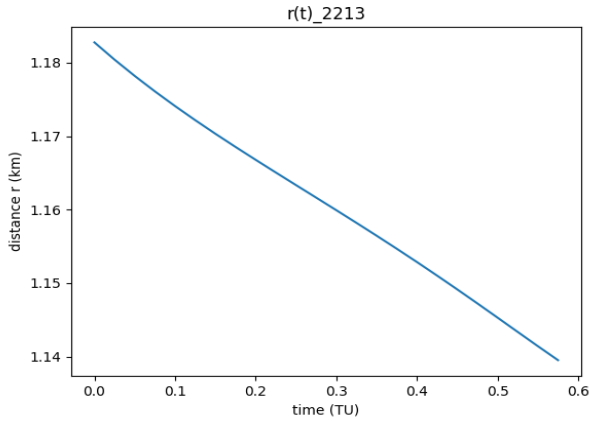

v(t)_1376
velocity v (km / TU)
time (TU)

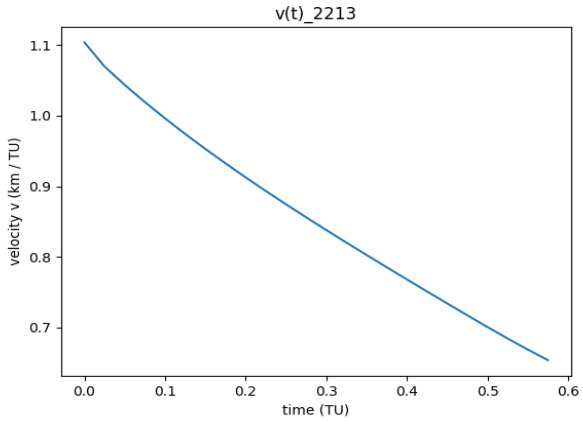

r(t)_1419
distance r (km)
time (TU)

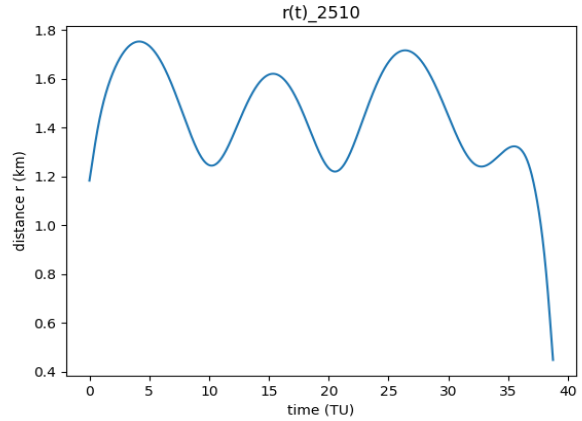

v(t)_1419
velocity v (km / TU)
time (TU)

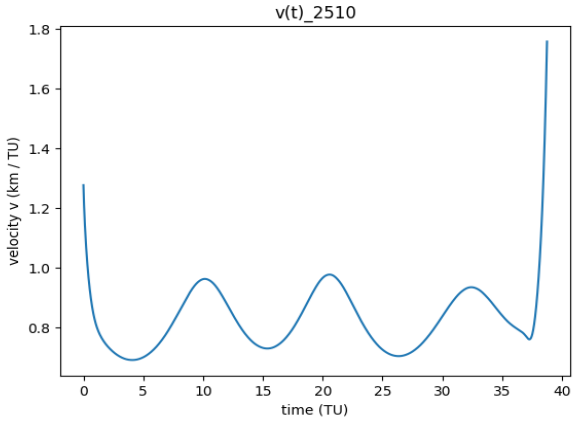

r(t)_1888
distance r (km)
time (TU)

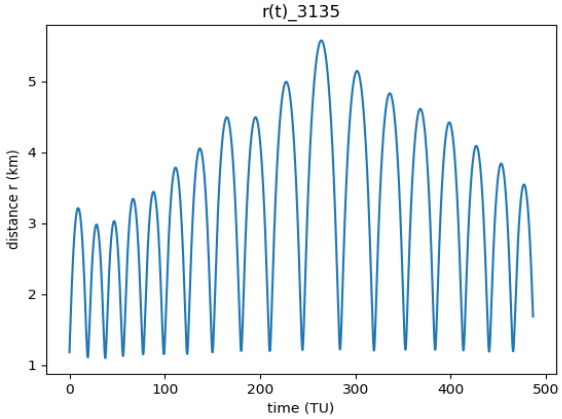

v(t)_1888
velocity v (km / TU)
time (TU)

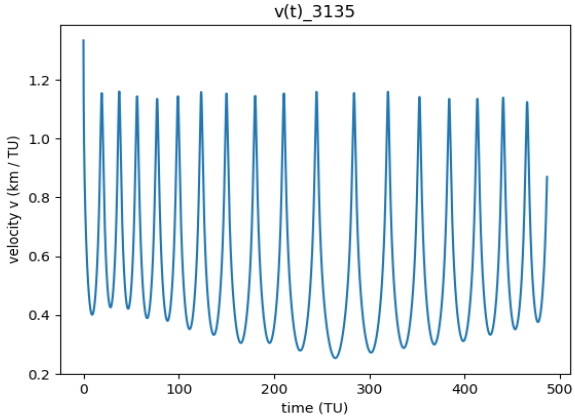

r(t)_2213
distance r (km)
time (TU)

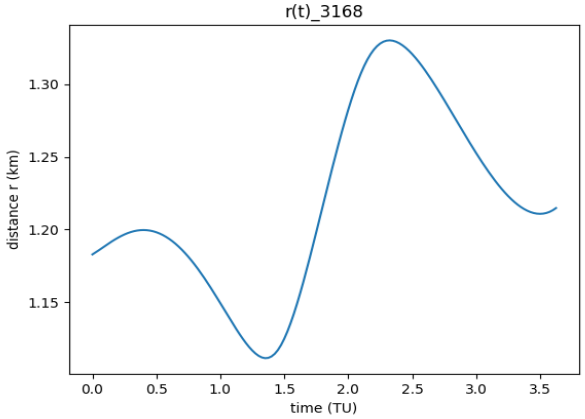

v(t)_2213
velocity v (km / TU)
time (TU)

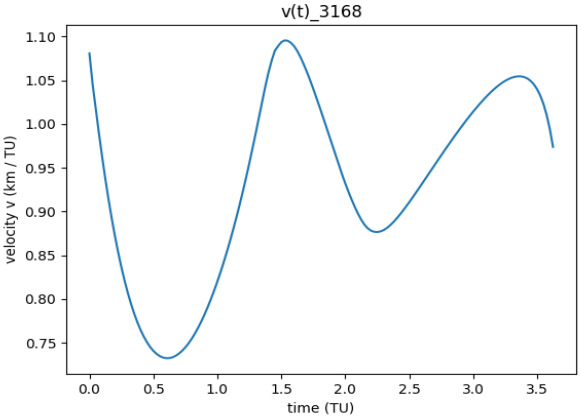

r(t)_2510
distance r (km)
time (TU)

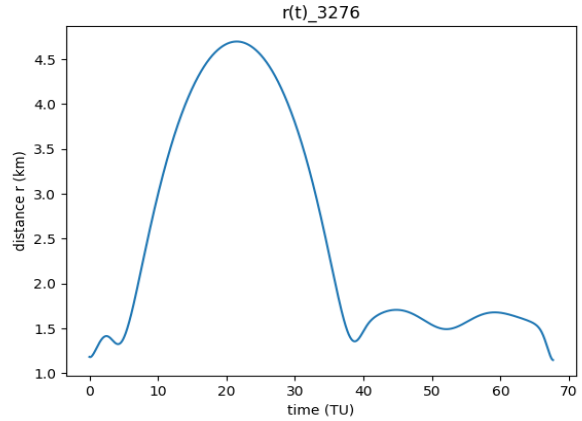

v(t)_2510
velocity v (km / TU)
time (TU)

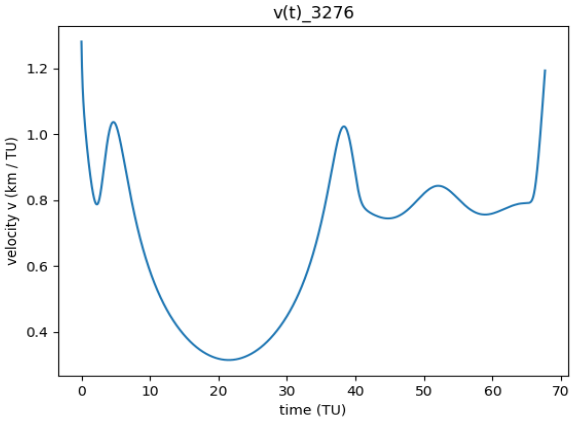

r(t)_3135
distance r (km)
time (TU)

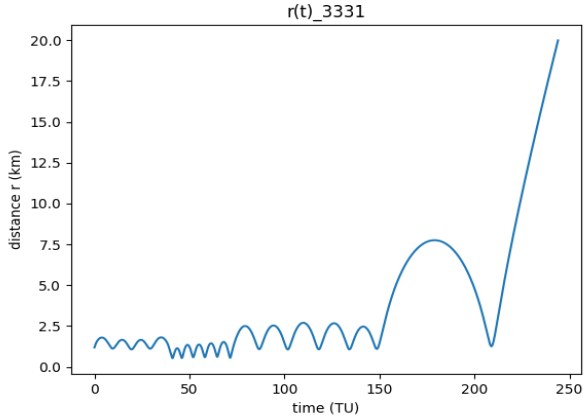

v(t)_3135
velocity v (km / TU)
time (TU)

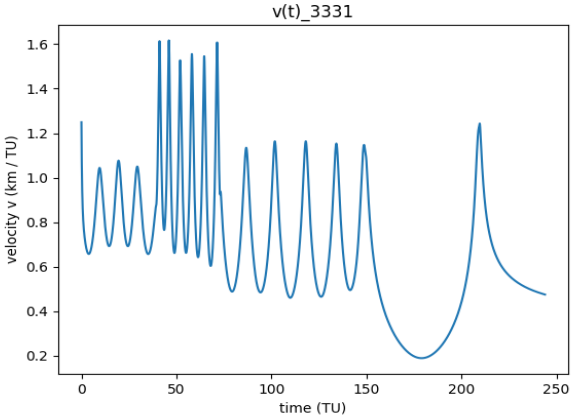

r(t)_3168
distance r (km)
time (TU)

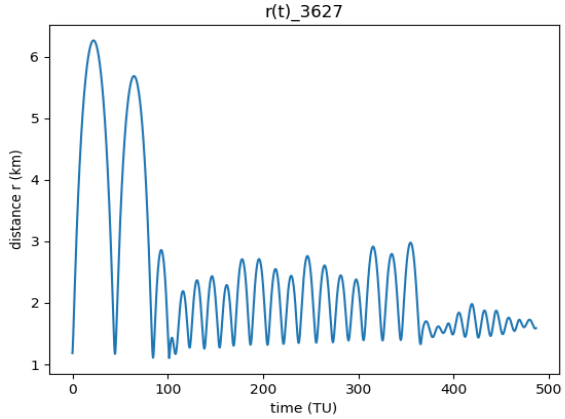

v(t)_3168
velocity v (km / TU)
time (TU)

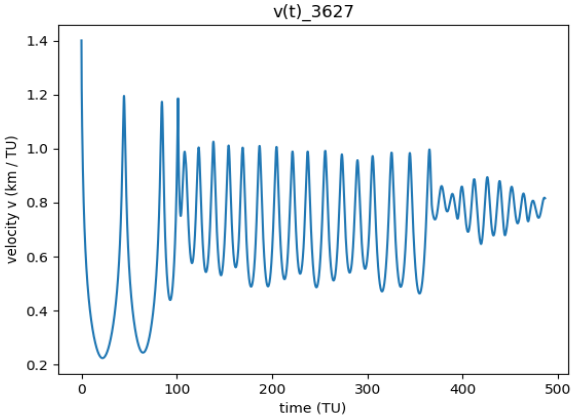

r(t)_3276
distance r (km)
time (TU)


v(t)_3276
velocity v (km / TU)
time (TU)


r(t)_3331
distance r (km)
time (TU)


v(t)_3331
velocity v (km / TU)
time (TU)


r(t)_3627
distance r (km)
time (TU)


v(t)_3627
velocity v (km / TU)
time (TU)

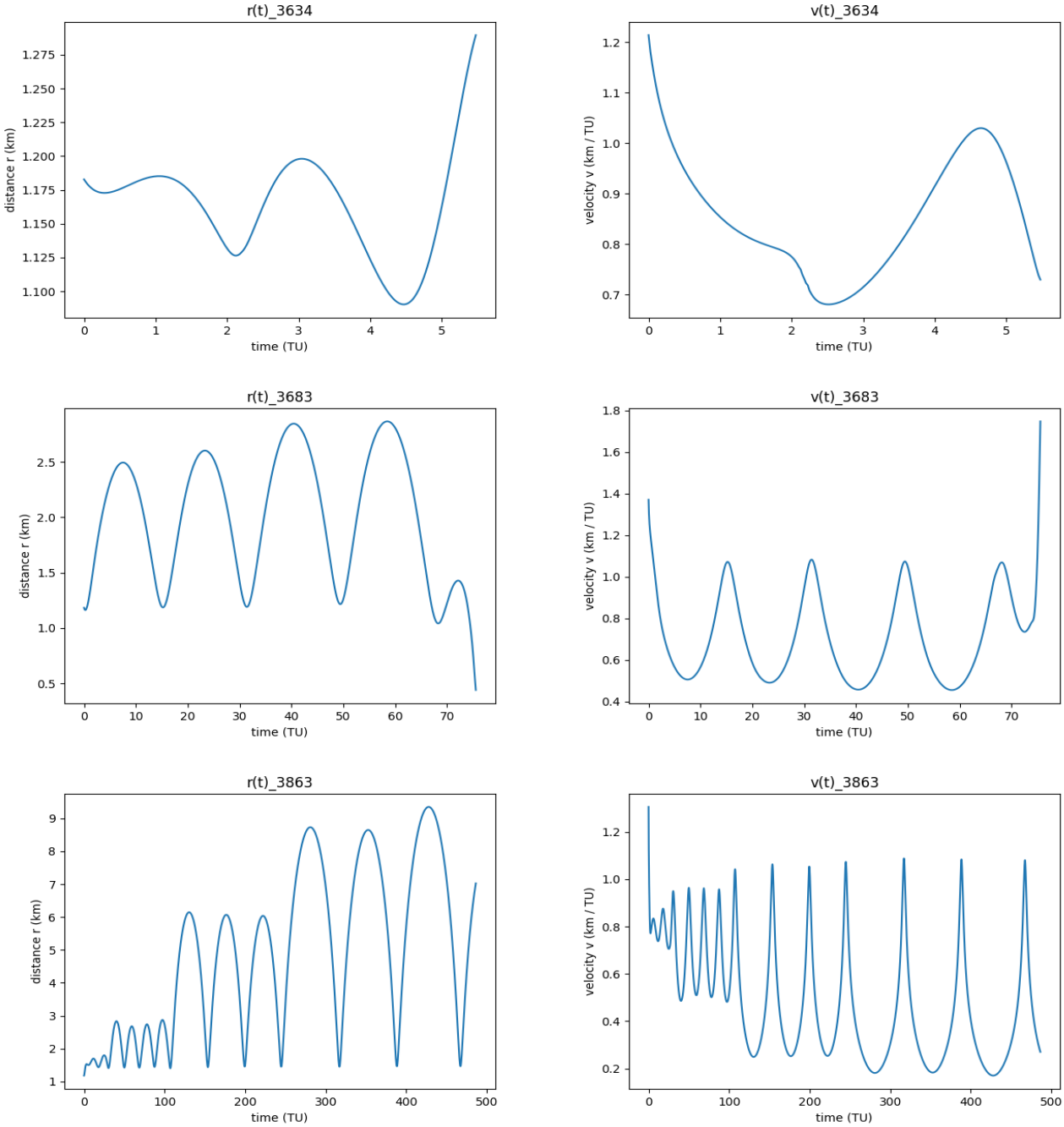
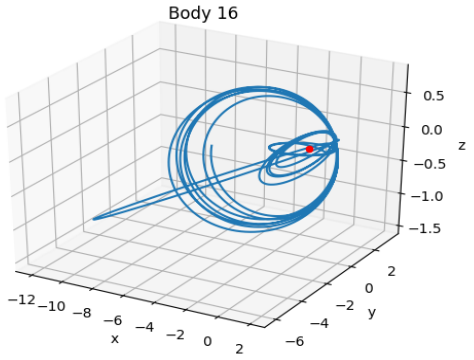
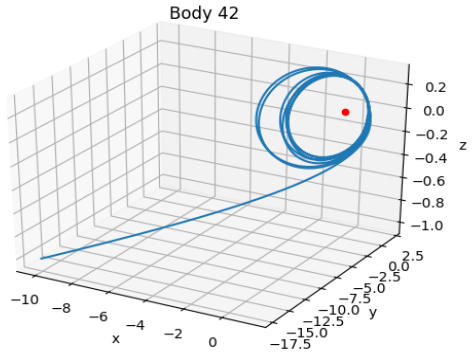
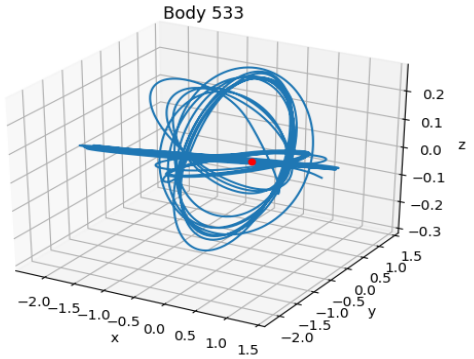
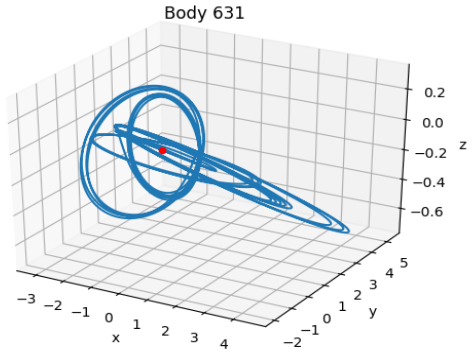


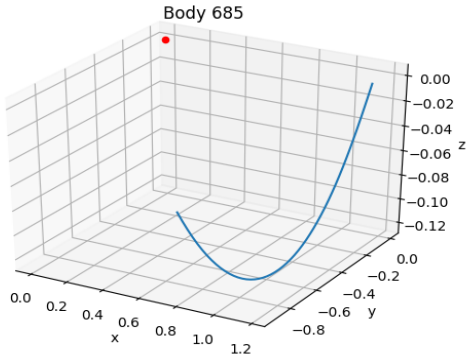
Figure 4.5: Evolution of the distance $r(t)$ and the velocity $v(t)$ of 21/4000 randomly sampled ejected particles.

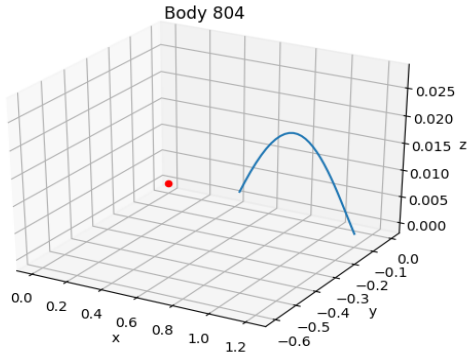

Body 16
x
y
z

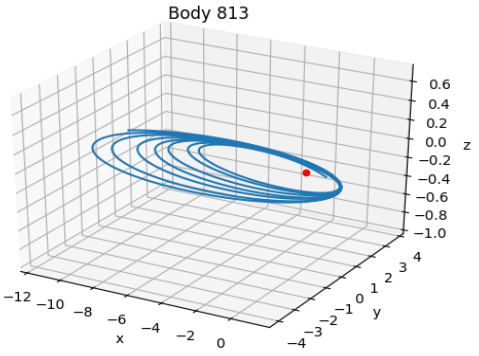

Body 42
x
y
z

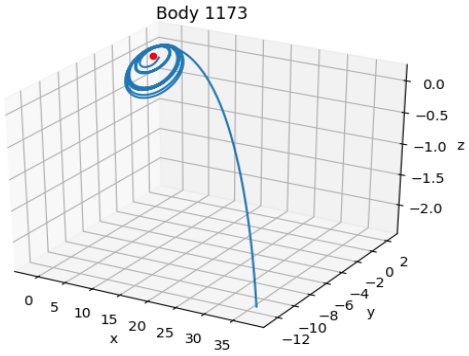

Body 533
x
y
z

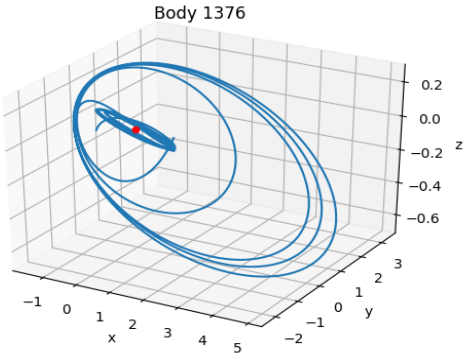

Body 631
x
y
z

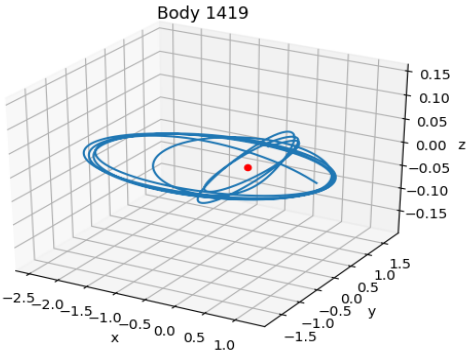

Body 685
x
y
z

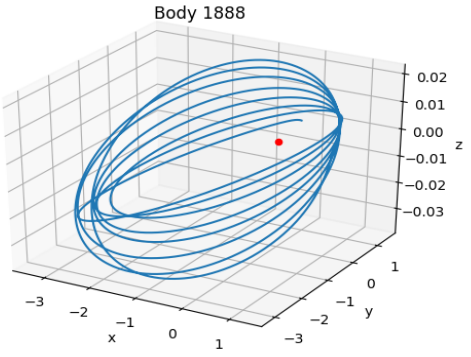

Body 804
x
y
z

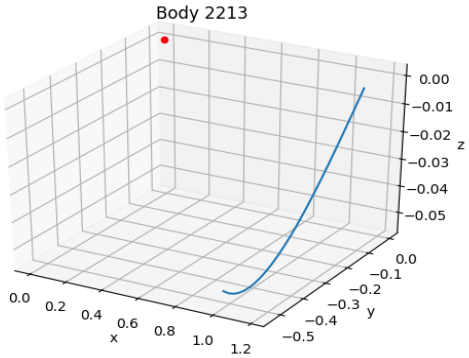

Body 813
x
y
z

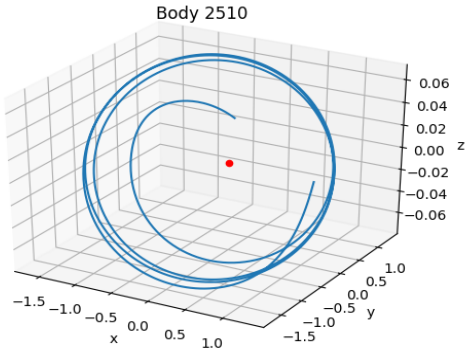

Body 1173
x
y
z

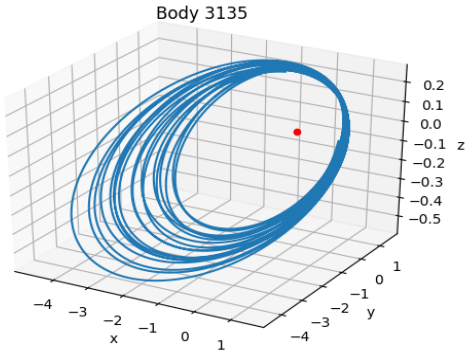

Body 1376
x
y
z

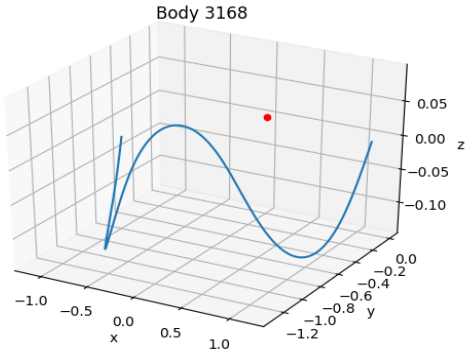

Body 1419
x
y
z

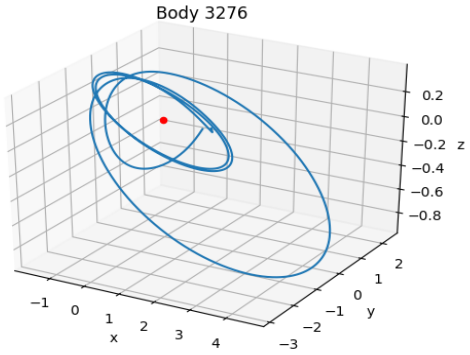

Body 1888
x
y
z

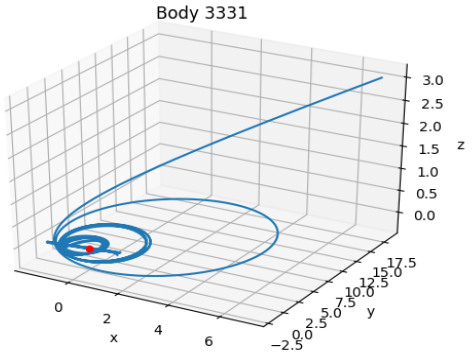

Body 2213
x
y
z

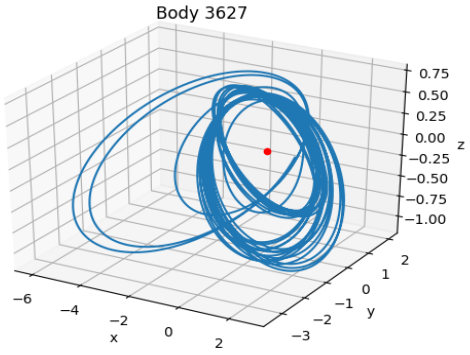

Body 2510
x
y
z


Body 3135
x
y
z


Body 3168
x
y
z


Body 3276
x
y
z


Body 3331
x
y
z


Body 3627
x
y
z

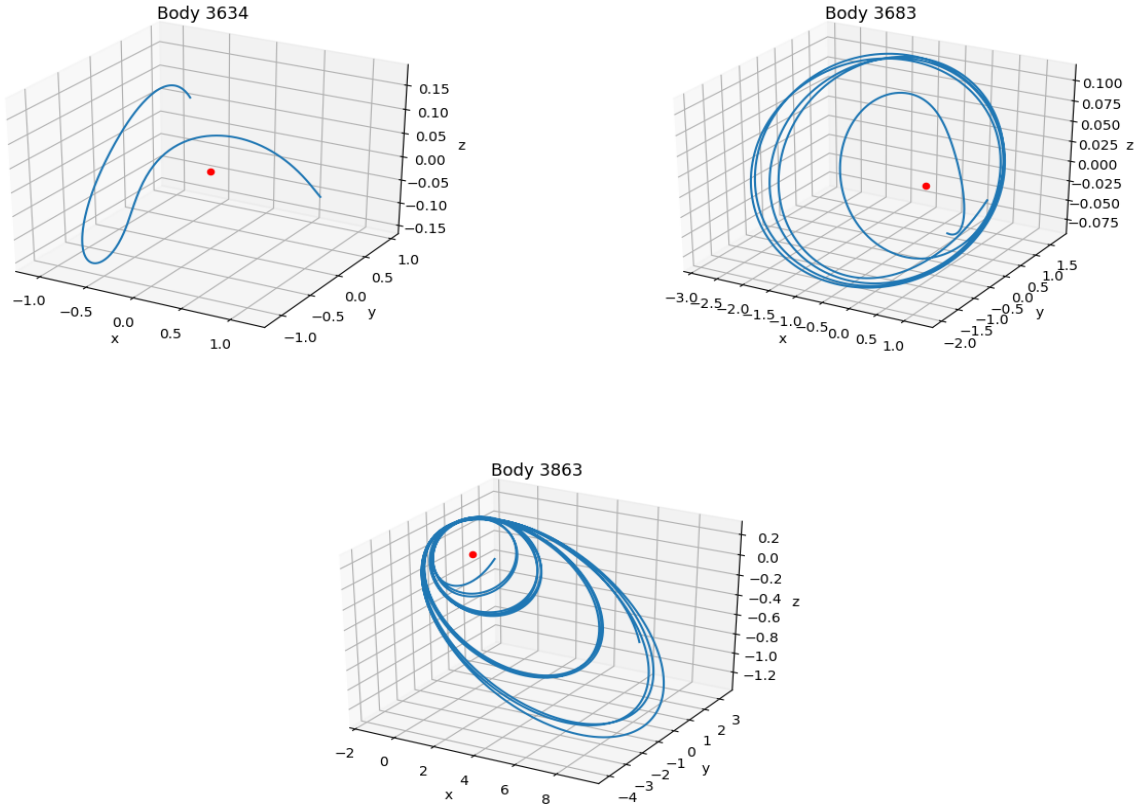


Figure 4.6: The orbit in space as a function of time $(x(t), y(t), z(t))$. The same 21/4000 particles were chosen as in figure (4.5). the red dot represents the center of mass of the system of the two asteroids.

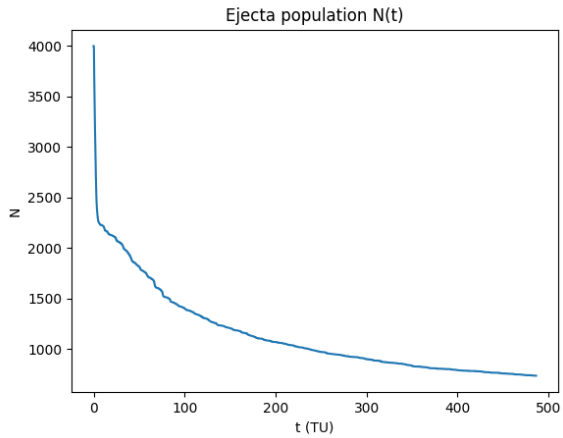


Figure 4.7: Number ejected particles that remain in orbit around the binary, as a function of time.

Figure (4.4) depicts the time evolution of the low velocity ejecta cloud for 1 month. Screenshots are taken at times: 44 min, 2 h 24 min, 3 h 2 min, 4 h 35 min, 7 h 17 min, 23 h 23 min, 2 days 8 min, 15 days 31 min and 29 days 23 h 15 min, showing the representative configurations of the cloud. We observe that the cloud spreads smoothly in space while time passes. Initially, the cloud surrounds Didymoon and many of its particles directly fall back and reaccumulate on Didymoon's surface due to their very low ejection velocity. After 3-4 hours, the cloud begins to surround Didymain, the same time at which many particles accrete on its surface. 2 days after, the cloud seems to have spread uniformly in the vicinity of the binary, until, after 15 days, it begins to dilute. After that, the rate at which the ejecta population reduces slows down. The latter can be observed from figure (4.7). Until 1 month, 11.75% of the ejected particles crash on Didymain's surface, 66.1% crash on Didymoon's surface and 3.75% escape the binary. In such a dynamical environment (low velocity particles), an escape occurs due to gravity assist from the asteroids. The remaining 18.4% is left in chaotic orbits around the binary. Apart from studying the ejecta cloud as a whole, one could study the behaviour of the ejected particles of the cloud as individuals. Figure (4.5) illustrates the distances $r(t)$ and the velocities $v(t)$ of 21/4000 randomly sampled ejected particles. Figure (4.6) illustrates the orbits $(x(t), y(t), z(t))$ of the same 21/4000 particles in space. One can observe the possible fates described in section (4.2). For example, the body 42, orbits the binary at a relatively short distance and at some point, due a gravity assist, it gains velocity greater than the corresponding escape one and ultimately flies away from the binary. Another case is the body 3683 which crashes on Didymain's surface after some time.

Appendices

Appendix A

Didymain Model (Source Code)

```
/*

This code uses the observed surface data of Didymain and fills its interior
with points. Ultimately a solid filled model is obtained.

Files used as input:
    1) main_surf_vertices.txt
    2) main_surf_indices.txt

Files produced as output:
    1) main_interior.txt
    2) main_complete_model.txt

*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

#define FILE_NAME_1 "main_surf_vertices.txt"
#define FILE_NAME_2 "main_surf_indices.txt"
#define FILE_NAME_3 "main_interior.txt"
#define FILE_NAME_4 "main_complete_model.txt"

const double h = 0.025; //3D grid step in km
int totalVertices = 0; //number of points that will form the asteroid

//Counts the number of rows of a file.
int FileRows(FILE *fp)
{
    int rows = 0;
    char c;
    while ((c = fgetc(fp)) != EOF)
    {
        if (c == '\n')
            rows++;
    }
    rows++;
    rewind(fp);
    return rows;
}

//Reads the observed vertices x,y,z of the asteroid from the file.
void InputVertices(FILE *fp, double *x, double *y, double *z)
{
    double tempx, tempy, tempz;
    int i = 0;
    while (fscanf(fp,"%lf %lf %lf",&tempx, &tempy, &tempz) != EOF)
    {
```

```
51          x[i] = tempx;
52          y[i] = tempy;
53          z[i] = tempz;
54          i++;
55      }
56  }
57
58  //Reads the observed indices of the asteroid from the file, that is,
59  //triads of points p1,p2,p3 that form triangles.
60  //Index i corresponds to the i-th row of the vertices file.
61  void InputIndices(FILE *fp, int *p[3])
62  {
63      int p1,p2,p3;
64      int i = 0;
65      while (fscanf(fp,"%d %d %d",&p1,&p2,&p3) != EOF)
66      {
67          //subtract 1 from all indices because the official file
68          //starts counting from 1, while I start from 0
69          p[i][0] = p1 - 1;
70          p[i][1] = p2 - 1;
71          p[i][2] = p3 - 1;
72          i++;
73      }
74  }
75
76  //Calculates the length of a vector.
77  double Len(double x, double y, double z)
78  {
79      return sqrt(x*x + y*y + z*z);
80  }
81
82  //Calculates the cross product components of two vectors that
83  //are formed by 3 points. Last argument is used to determine
84  //the component of the cross product that will be returned.
85  //Possible values of 'coordinate': 0 --> x, 1 --> y, 2 --> z
86  double CrossProduct(double x1, double y1, double z1,
87                      double x2, double y2, double z2,
88                      double x3, double y3, double z3, double coordinate)
89  {
90       if (coordinate == 0) return (y2-y1)*(z3-z2) - (z2-z1)*(y3-y2);
91       if (coordinate == 1) return (z2-z1)*(x3-x2) - (x2-x1)*(z3-z2);
92       if (coordinate == 2) return (x2-x1)*(y3-y2) - (y2-y1)*(x3-x2);
93       //else
94       printf("Error while calculating the normal vectors. Exiting...\n");
95       exit(EXIT_FAILURE);
96  }
97
98  //(nx[i],ny[i],nz[i]) --> coordinates of the i-th normal vector, that is,
99  //the vector which is perpendicular to the triangle formed from
100 //the i-th triad of the indices file.
101 void CalculateNormalVectors(double *x,  double *y, double  *z,
102                             double *nx, double *ny, double *nz,
103                             int *p[3], int N2)
104 {
105     for (int i = 0; i < N2; i++)
106     {
107         nx[i] = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
108                              x[p[i][1]], y[p[i][1]], z[p[i][1]],
109                              x[p[i][2]], y[p[i][2]], z[p[i][2]], 0);
110
111         ny[i] = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
112                              x[p[i][1]], y[p[i][1]], z[p[i][1]],
113                              x[p[i][2]], y[p[i][2]], z[p[i][2]], 1);
114
115         nz[i] = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
116                              x[p[i][1]], y[p[i][1]], z[p[i][1]],
117                              x[p[i][2]], y[p[i][2]], z[p[i][2]], 2);
118     }
119 }
```

```
120
121 //Calculates the min value of a 1D array.
122 double Min(double *array, int N1)
123 {
124     double min = array[0];
125     for (int i = 1; i < N1; i++)
126     {
127         if (array[i] < min)
128             min = array[i];
129     }
130     return min;
131 }
132
133 //Calculates the max value of a 1D array.
134 double Max(double *array, int N1)
135 {
136     double max = array[0];
137     for (int i = 1; i < N1; i++)
138     {
139         if (array[i] > max)
140             max = array[i];
141     }
142     return max;
143 }
144
145 //Creates a solid filled 3D model of the asteroid
146 void ComputationalSpace(FILE *fp3, FILE *fp4,
147                         double *x,  double *y,  double *z,
148                         double *nx, double *ny, double *nz,
149                         int *p[3], int N1, int N2)
150 {
151     //calculate the borders of the grid (box)
152     double x_min = Min(x,N1);
153     double x_max = Max(x,N1);
154     double y_min = Min(y,N1);
155     double y_max = Max(y,N1);
156     double z_min = Min(z,N1);
157     double z_max = Max(z,N1);
158     //increase the box's size a little bit
159     x_min -= h;
160     x_max += h;
161     y_min -= h;
162     y_max += h;
163     z_min -= h;
164     z_max += h;
165     //calculate the size of the box
166     double Dx = x_max - x_min;
167     double Dy = y_max - y_min;
168     double Dz = z_max - z_min;
169
170     printf("Computational space (box) size :\n");
171     printf("\tx_min = %lf\n",x_min);
172     printf("\tx_max = %lf\n",x_max);
173     printf("\t\tDx = %lf\n",Dx);
174     printf("\ty_min = %lf\n",y_min);
175     printf("\ty_max = %lf\n",y_max);
176     printf("\t\tDy = %lf\n",Dy);
177     printf("\tz_min = %lf\n",z_min);
178     printf("\tz_max = %lf\n",z_max);
179     printf("\t\tDz = %lf\n",Dz);
180     printf("Creating model. Please wait...\n");
181     //loop through all the points of the 3D grid with step h
182     for (double xx = x_min; xx <= x_max; xx += h)
183     {
184         for (double yy = y_min; yy <= y_max; yy += h)
185         {
186             for (double zz = z_min; zz <= z_max; zz += h)
187             {
188                 int intersections = 0; //counter
```

```
189                 //loop through all the triangles in search for intersection
190                 for (int i = 0; i < N2; i++)
191                 {
192                     //define the triangle i from its 3 points
193 
194                     //point p0
195                     double x0 = x[p[i][0]];
196                     double y0 = y[p[i][0]];
197                     double z0 = z[p[i][0]];
198 
199                     //point p1
200                     double x1 = x[p[i][1]];
201                     double y1 = y[p[i][1]];
202                     double z1 = z[p[i][1]];
203 
204                     //point p2
205                     double x2 = x[p[i][2]];
206                     double y2 = y[p[i][2]];
207                     double z2 = z[p[i][2]];
208 
209                     //intersection point pi
210                     double xi = x0 + (y0*ny[i] + z0*nz[i] - yy*ny[i] - zz*nz[i])/nx[i];
211                     double yi = yy;
212                     double zi = zz;
213 
214                     //Form the following 3 triangles and calculate their area :
215                     //1) pi p0 p1
216                     //2) pi p1 p2
217                     //3) pi p2 p0
218                     double A_i01 = 0.5*Len(CrossProduct(xi,yi,zi, x0,y0,z0, x1,y1,z1, 0),
219                                            CrossProduct(xi,yi,zi, x0,y0,z0, x1,y1,z1, 1),
220                                            CrossProduct(xi,yi,zi, x0,y0,z0, x1,y1,z1, 2));
221 
222                     double A_i12 = 0.5*Len(CrossProduct(xi,yi,zi, x1,y1,z1, x2,y2,z2, 0),
223                                            CrossProduct(xi,yi,zi, x1,y1,z1, x2,y2,z2, 1),
224                                            CrossProduct(xi,yi,zi, x1,y1,z1, x2,y2,z2, 2));
225 
226                     double A_i20 = 0.5*Len(CrossProduct(xi,yi,zi, x2,y2,z2, x0,y0,z0, 0),
227                                            CrossProduct(xi,yi,zi, x2,y2,z2, x0,y0,z0, 1),
228                                            CrossProduct(xi,yi,zi, x2,y2,z2, x0,y0,z0, 2));
229 
230                     double A     = 0.5*Len(CrossProduct(x0,y0,z0, x1,y1,z1, x2,y2,z2, 0),
231                                            CrossProduct(x0,y0,z0, x1,y1,z1, x2,y2,z2, 1),
232                                            CrossProduct(x0,y0,z0, x1,y1,z1, x2,y2,z2, 2));
233 
234                     //if the sum of the 3 areas is equal to the area of the main triangle,
235                     //then (xi,yi,zi) is indeed bounded from the triangle p0 p1 p2
236                     if ((float)(A_i01 + A_i12 + A_i20) == (float)A && xi > xx)
237                         intersections++;
238                 }
239                 //odd --> (xx,yy,zz) is inside the surface
240                 //even --> (xx,yy,zz) is outside the surface
241                 if (intersections%2 == 1)
242                 {
243                     fprintf(fp3,"%lf %lf %lf\n",xx,yy,zz);
244                     fprintf(fp4,"%lf %lf %lf\n",xx,yy,zz);
245                     totalVertices++;
246                 }
247             }
248         }
249     }
250     //append the initial surface vertices to obtain the complete model
251     for (int i = 0; i < N1; i++)
252     {
253         fprintf(fp4,"%lf %lf %lf\n",x[i],y[i],z[i]);
254         totalVertices++;
255     }
256 }
257 
```

```c
int main()
{
    FILE *fp1 = fopen(FILE_NAME_1,"r"); //vertices file
    if (fp1 == NULL)
    {
        printf("Error while reading file. Exiting...\n");
        exit(EXIT_FAILURE);
    }
    int N1  = FileRows(fp1); //number of rows of the vertices file
    //surface vertices coordinates
    double *x = (double*)malloc(N1*sizeof(double));
    double *y = (double*)malloc(N1*sizeof(double));
    double *z = (double*)malloc(N1*sizeof(double));
    InputVertices(fp1,x,y,z);

    FILE *fp2 = fopen(FILE_NAME_2,"r"); //indices file
    if (fp2 == NULL)
    {
        printf("Error while reading file. Exiting...\n");
        exit(EXIT_FAILURE);
    }
    int N2 = FileRows(fp2); //number of rows of the indices file (number of triangles)
    int **p = (int**)malloc(N2*sizeof(int*)); //p[][]: triads of indices that form triangles
    for (int i = 0; i < N2; i++)
        p[i] = (int*)malloc(3*sizeof(int));
    InputIndices(fp2,p);

    //coordinates of the normal vectors (perpendicular to each triangle)
    double *nx = (double*)malloc(N2*sizeof(double));
    double *ny = (double*)malloc(N2*sizeof(double));
    double *nz = (double*)malloc(N2*sizeof(double));
    CalculateNormalVectors(x,y,z,nx,ny,nz,p,N2);

    FILE *fp3 = fopen(FILE_NAME_3,"w"); //interior points
    FILE *fp4 = fopen(FILE_NAME_4,"w"); //surface AND interior points
    clock_t t1,t2;
    t1 = clock();
    ComputationalSpace(fp3,fp4,x,y,z,nx,ny,nz,p,N1,N2);
    t2 = clock();
    double cpuTime = (t2-t1)/(double)CLOCKS_PER_SEC;
    printf("Model created successfully.\n");
    printf("Total vertices: %d\n",totalVertices);
    printf("Estimated completion time: %lf sec\n",cpuTime);

    free(x);
    free(y);
    free(z);
    for (int i = 0; i < N2; i++)
        free(p[i]);
    free(p);
    free(nx);
    free(ny);
    free(nz);
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    fclose(fp4);
    return 0;
}
```

codes/MainCompSpace.c

Appendix B

Didymoon Model (Source Code)

```
/*
This code uses the analytic equation of a tri-axial ellipsoid
in order to create solid filled 3D model of Didymoon.

Files produced as output:
    1) moon_complete_model.txt

*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

#define FILE_NAME "moon_complete_model.txt"

//ellipsoid semi-axes in km
const double a = 0.100;
const double b = 0.080;
const double c = 0.070;

const double h = 0.01; //3D grid step in km
int totalVertices = 0; //number of points that will form the asteroid

//Creates a solid filled model of the asteroid
void ComputationalSpace(FILE *fp)
{
    //define the borders of the grid (box)
    double x_min = -a;
    double x_max =  a;
    double y_min = -b;
    double y_max =  b;
    double z_min = -c;
    double z_max =  c;
    //increase the box's size a little bit
    x_min -= h;
    x_max += h;
    y_min -= h;
    y_max += h;
    z_min -= h;
    z_max += h;
    //calculate the size of the box
    double Dx = x_max - x_min;
    double Dy = y_max - y_min;
    double Dz = z_max - z_min;

    printf("Computational space (box) size :\n");
    printf("\tx_min = %lf\n",x_min);
    printf("\tx_max = %lf\n",x_max);
    printf("\t\tDx = %lf\n",Dx);
```

```
51      printf("\ty_min = %lf\n",y_min);
52      printf("\ty_max = %lf\n",y_max);
53      printf("\t\tDy = %lf\n",Dy);
54      printf("\tz_min = %lf\n",z_min);
55      printf("\tz_max = %lf\n",z_max);
56      printf("\t\tDz = %lf\n",Dz);
57      printf("Creating model. Please wait...\n");
58      //loop through the computational space with step h
59      for (double x = x_min; x <= x_max; x += h)
60      {
61          for (double y = y_min; y <= y_max; y += h)
62          {
63              for (double z = z_min; z <= z_max; z += h)
64              {
65                  //check if you are inside or on the ellipsoid surface
66                  if ((x*x)/(a*a) + (y*y)/(b*b) + (z*z)/(c*c) <= 1.0)
67                  {
68                      fprintf(fp,"%.3lf %.3lf %.3lf\n",x,y,z);
69                      totalVertices++;
70                  }
71              }
72          }
73      }
74  }
75
76  int main()
77  {
78      FILE *fp = fopen(FILE_NAME,"w");
79      clock_t t1,t2;
80      t1 = clock();
81      ComputationalSpace(fp);
82      t2 = clock();
83      double cpuTime = (t2-t1)/(double)CLOCKS_PER_SEC;
84      printf("Model created successfully.\n");
85      printf("Total vertices: %d\n",totalVertices);
86      printf("Estimated completion time: %lf sec\n",cpuTime);
87      fclose(fp);
88      return 0;
89  }
```

codes/MoonCompSpace.c

Appendix C

Visualisation of the Models (Source Code)

```
/*

This code imports the two asteroid models and renders 3D visualisation.
Use the buttons '1', '2', '3', '4', '5', '6' to jump between models.

*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<stdbool.h>
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/freeglut.h>

#define FILE_NAME_1 "main_surf_vertices.txt"
#define FILE_NAME_2 "main_surf_indices.txt"
#define FILE_NAME_3 "main_interior.txt"
#define FILE_NAME_4 "main_complete_model.txt"
#define FILE_NAME_5 "moon_complete_model.txt"
#define FILE_NAME_6 "main_surf_interpolated_vertices.txt"

#define ESCAPE 27 //corresponding ASCII character
#define SPACEBAR 32 //corresponding ASCII character

//what you are going to see according to the button you press ('1',...,'6')
bool mainSurface = true; //didymain surface model
bool mainInterpSurface = false; //didymain's interpolated surface vertices
bool mainComplete = false; //didymain complete model
bool mainSurfaceAndInterior = false; //didymain surface and interior vertices
bool moonComplete = false; //didymoon's vertices
bool mainAndMoon = false; //didymain and didymoon surfaces

bool pause = false; //press SPACEBAR to pause/unpause

const float hMainSurf = 0.04; //didymain's surface voxel size
const float hMainInterpSurface = 0.024; //didymain's interpolated surface voxel size
const float hMainComplete = 0.024; //didymain's complete model voxel size
const float hMoonComplete = 0.01; //didymoon's complete model voxel size

const float axeDistance = 0.6f;

int N1,N2,N3,N4,N5,N6; //corresponding file rows

//didymain surface vertices
float *xx1 = NULL;
float *yy1 = NULL;
float *zz1 = NULL;

//didymain surface indices
```

```
51 int **p = NULL; //p[i][j]
52
53 //unit normal vector components (used for shading didymain)
54 float *ux = NULL;
55 float *uy = NULL;
56 float *uz = NULL;
57
58 //didymain interior
59 float *x3 = NULL;
60 float *y3 = NULL;
61 float *z3 = NULL;
62
63 //didymain complete model
64 float *x4 = NULL;
65 float *y4 = NULL;
66 float *z4 = NULL;
67
68 //didymoon complete model
69 float *x5 = NULL;
70 float *y5 = NULL;
71 float *z5 = NULL;
72
73 //didymain interpolated surface vertices
74 float *x6 = NULL;
75 float *y6 = NULL;
76 float *z6 = NULL;
77
78 //ellipsoid (didymoon) semi-axes in km
79 const float a = 0.103;
80 const float b = 0.079;
81 const float c = 0.066;
82
83 //graphics window size
84 GLsizei width  = 1000;
85 GLsizei height = 900;
86
87 //Counts the number of rows of a file.
88 int FileRows(FILE *fp)
89 {
90     int rows = 0;
91     char c;
92     while ((c = fgetc(fp)) != EOF)
93     {
94         if (c == '\n')
95             rows++;
96     }
97     rows++;
98     rewind(fp);
99     return rows;
100 }
101
102 //Reads the x,y,z data from the file.
103 void InputVertices(FILE *fp, float *x, float *y, float *z)
104 {
105     float tempx, tempy, tempz;
106     int i = 0;
107     while (fscanf(fp,"%f %f %f",&tempx, &tempy, &tempz) != EOF)
108     {
109         x[i] = tempx;
110         y[i] = tempy;
111         z[i] = tempz;
112         i++;
113     }
114 }
115
116 //Reads the observed indices of didymain from the file, that is,
117 //triads of points p1,p2,p3 that form triangles
118 //index i corresponds to the i-th row of the vertices file.
119 void InputIndices(FILE *fp)
```

```
120 {
121     int p1,p2,p3;
122     int i = 0;
123     while (fscanf(fp,"%d %d %d",&p1,&p2,&p3) != EOF)
124     {
125         //subtract 1 from all indices because the official file
126         //starts counting from 1, while I start from 0
127         p[i][0] = p1 - 1;
128         p[i][1] = p2 - 1;
129         p[i][2] = p3 - 1;
130         i++;
131     }
132 }
133
134 //Calculates the cross product components of two vectors that
135 //are formed by 3 points. Last argument is used to determine
136 //the component of the cross product that will be returned.
137 //Possible values of 'coordinate': 0 --> x, 1 --> y, 2 --> z
138 float CrossProduct(float x1, float y1, float z1,
139                    float x2, float y2, float z2,
140                    float x3, float y3, float z3, float coordinate)
141 {
142      if (coordinate == 0) return (y2-y1)*(z3-z2) - (z2-z1)*(y3-y2);
143      if (coordinate == 1) return (z2-z1)*(x3-x2) - (x2-x1)*(z3-z2);
144      if (coordinate == 2) return (x2-x1)*(y3-y2) - (y2-y1)*(x3-x2);
145      //else
146      printf("Error while calculating the normal vectors. Exiting...\n");
147      exit(EXIT_FAILURE);
148 }
149
150 //(nx[i],ny[i],nz[i]) --> coordinates of the i-th normal vector, that is,
151 //the vector which is perpendicular to the triangle formed from
152 //the i-th triad of the indices file.
153 void CalculateNormalVectors()
154 {
155     float nx,ny,nz; //i-th normal vector
156     for (int i = 0; i < N2; ++i)
157     {
158         nx = CrossProduct(xx1[p[i][0]], yy1[p[i][0]], zz1[p[i][0]],
159                           xx1[p[i][1]], yy1[p[i][1]], zz1[p[i][1]],
160                           xx1[p[i][2]], yy1[p[i][2]], zz1[p[i][2]], 0);
161
162         ny = CrossProduct(xx1[p[i][0]], yy1[p[i][0]], zz1[p[i][0]],
163                           xx1[p[i][1]], yy1[p[i][1]], zz1[p[i][1]],
164                           xx1[p[i][2]], yy1[p[i][2]], zz1[p[i][2]], 1);
165
166         nz = CrossProduct(xx1[p[i][0]], yy1[p[i][0]], zz1[p[i][0]],
167                           xx1[p[i][1]], yy1[p[i][1]], zz1[p[i][1]],
168                           xx1[p[i][2]], yy1[p[i][2]], zz1[p[i][2]], 2);
169
170         //i-th unit normal vector
171         ux[i] = nx/sqrt(nx*nx + ny*ny + nz*nz);
172         uy[i] = ny/sqrt(nx*nx + ny*ny + nz*nz);
173         uz[i] = nz/sqrt(nx*nx + ny*ny + nz*nz);
174     }
175 }
176
177 void setup()
178 {
179     glEnable(GL_DEPTH_TEST);
180     glEnable(GL_LIGHT0);
181     glEnable(GL_LIGHTING);
182     glEnable(GL_COLOR_MATERIAL);
183     glEnable(GL_NORMALIZE);
184     glClearColor(1.0,1.0,1.0,0.0);
185 }
186
187 void reshape(GLsizei w, GLsizei h)
188 {
```

```
189     if (h == 0) h = 1;
190     glViewport(0,0,w,h);
191     glMatrixMode(GL_PROJECTION);
192     glLoadIdentity();
193     float AspectRatio = (float)w/(float)h;
194     gluPerspective(60.0,AspectRatio,0.1,100.0);
195     glMatrixMode(GL_MODELVIEW);
196     glLoadIdentity();
197     gluLookAt(0,-1.9,0.75, 0,0,0, 0,1,0);
198 }
199
200 void display()
201 {
202     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
203
204     glPushMatrix();
205         float lightPosition[] = {1.0,-1.0,0.0,0.0};
206         glLightfv(GL_LIGHT0,GL_POSITION,lightPosition);
207     glPopMatrix();
208
209     static float angle = 0.0f;
210
211     glColor3f(0.5f,0.5f,0.5f);
212     if (mainSurface) //plot didymain's surface models (voxels and triangles)
213     {
214         //voxels
215         glPushMatrix();
216             glTranslatef(-axeDistance,0.0f,0.0f);
217             glRotatef(angle,0,0,1);
218             for (int i = 0; i < N1; ++i)
219             {
220                 glPushMatrix();
221                     glTranslated(xx1[i],yy1[i],zz1[i]);
222                     glutSolidCube(hMainSurf);
223                 glPopMatrix();
224             }
225         glPopMatrix();
226
227         //triangles
228         glPushMatrix();
229             glTranslatef(axeDistance,0.0f,0.0f);
230             glRotatef(angle,0,0,1);
231             glBegin(GL_TRIANGLES);
232                 for (int i = 0; i < N2; ++i)
233                 {
234                     glNormal3f(ux[i],uy[i],uz[i]); //for the shading
235                         glVertex3f(xx1[p[i][0]], yy1[p[i][0]], zz1[p[i][0]]);
236                         glVertex3f(xx1[p[i][1]], yy1[p[i][1]], zz1[p[i][1]]);
237                         glVertex3f(xx1[p[i][2]], yy1[p[i][2]], zz1[p[i][2]]);
238                 }
239             glEnd();
240         glPopMatrix();
241     }
242     else if (mainInterpSurface) //plot didymain (surface + interpolated)
243     {
244       //non interpolated
245         glPushMatrix();
246             glTranslatef(-axeDistance,0.0f,0.0f);
247             glRotatef(angle,0,0,1);
248             for (int i = 0; i < N1; ++i)
249             {
250                 glPushMatrix();
251                     glTranslated(xx1[i],yy1[i],zz1[i]);
252                     glutSolidCube(hMainSurf);
253                 glPopMatrix();
254             }
255         glPopMatrix();
256
257       //interpolated
```

```
258         glPushMatrix();
259       glTranslatef(axeDistance,0.0f,0.0f);
260             glRotatef(angle,0,0,1);
261             for (int i = 0; i < N6; ++i)
262             {
263                 glPushMatrix();
264                     glTranslated(x6[i],y6[i],z6[i]);
265                     glutSolidCube(hMainInterpSurface);
266                 glPopMatrix();
267             }
268         glPopMatrix();
269     }
270     else if (mainComplete) //plot complete model of didymain
271     {
272         glPushMatrix();
273             glRotatef(angle,0,0,1);
274             for (int i = 0; i < N4; ++i)
275             {
276                 glPushMatrix();
277                     glTranslated(x4[i],y4[i],z4[i]);
278                     glutSolidCube(hMainComplete);
279                 glPopMatrix();
280             }
281         glPopMatrix();
282     }
283     else if (mainSurfaceAndInterior) //demonstrate how the interior fits the surface
284     {
285         //surface
286         glPushMatrix();
287             glRotatef(angle,0,0,1);
288             for (int i = 0; i < N1; ++i)
289             {
290                 if (xx1[i] > 0)
291                 {
292                     glPushMatrix();
293                         glTranslated(xx1[i],yy1[i],zz1[i]);
294                         glutSolidCube(hMainSurf);
295                     glPopMatrix();
296                 }
297             }
298         glPopMatrix();
299 
300         //interior
301         glColor3f(0.7f,0.0f,0.0f);
302         glPushMatrix();
303             glRotatef(angle,0,0,1);
304             for (int i = 0; i < N3; ++i)
305             {
306                 glPushMatrix();
307                     glTranslated(x3[i],y3[i],z3[i]);
308                     glutSolidCube(hMainComplete);
309                 glPopMatrix();
310             }
311         glPopMatrix();
312     }
313     else if (moonComplete) //plot didymoon
314     {
315         glPushMatrix();
316             glTranslatef(-axeDistance/3,0.0f,0.0f);
317             glRotatef(angle,0,0,1);
318             for (int i = 0; i < N5; ++i)
319             {
320                 glPushMatrix();
321                     glTranslated(x5[i],y5[i],z5[i]);
322                     glutSolidCube(hMoonComplete);
323                 glPopMatrix();
324             }
325         glPopMatrix();
326 
```

```
327         glPushMatrix();
328             glTranslatef(axeDistance/3,0.0f,0.0f);
329             glRotatef(angle,0,0,1);
330             glScaled(a,b,c);
331             glutSolidSphere(1.0,20,20);
332         glPopMatrix();
333     }
334     else if (mainAndMoon) //plot didymain and didymoon
335     {
336         //didymain triangles
337         glPushMatrix();
338             glTranslatef(-1.18/2,0.0f,0.0f);
339             glRotatef(angle,0,0,1);
340             glBegin(GL_TRIANGLES);
341                 for (int i = 0; i < N2; ++i)
342                 {
343                     glNormal3f(ux[i],uy[i],uz[i]); //for the shading
344                         glVertex3f(xx1[p[i][0]], yy1[p[i][0]], zz1[p[i][0]]);
345                         glVertex3f(xx1[p[i][1]], yy1[p[i][1]], zz1[p[i][1]]);
346                         glVertex3f(xx1[p[i][2]], yy1[p[i][2]], zz1[p[i][2]]);
347                 }
348             glEnd();
349         glPopMatrix();
350 
351         //didymoon
352         glPushMatrix();
353             glTranslatef(1.18/2,0.0f,0.0f);
354             glRotatef(angle,0,0,1);
355             glScaled(a,b,c);
356             glutSolidSphere(1.0,20,20);
357         glPopMatrix();
358     }
359 
360     angle += 1.0f;
361     if (angle >= 360.0f)
362         angle = 0.0f;
363 
364     glutSwapBuffers();
365 }
366 
367 void keyboard(unsigned char key, int x, int y)
368 {
369     if (key == '1')
370     {
371         mainSurface = true;
372         mainInterpSurface = false;
373         mainComplete = false;
374         mainSurfaceAndInterior = false;
375         moonComplete = false;
376         mainAndMoon = false;
377     }
378     else if (key == '2')
379     {
380         mainSurface = false;
381         mainInterpSurface = true;
382         mainComplete = false;
383         mainSurfaceAndInterior = false;
384         moonComplete = false;
385         mainAndMoon = false;
386     }
387     else if (key == '3')
388     {
389         mainSurface = false;
390         mainInterpSurface = false;
391         mainComplete = true;
392         mainSurfaceAndInterior = false;
393         moonComplete = false;
394         mainAndMoon = false;
395     }
```

```
396     else if (key == '4')
397     {
398         mainSurface = false;
399         mainInterpSurface = false;
400         mainComplete = false;
401         mainSurfaceAndInterior = true;
402         moonComplete = false;
403         mainAndMoon = false;
404     }
405     else if (key == '5')
406     {
407         mainSurface = false;
408         mainInterpSurface = false;
409         mainComplete = false;
410         mainSurfaceAndInterior = false;
411         moonComplete = true;
412         mainAndMoon = false;
413     }
414     else if (key == '6')
415     {
416         mainSurface = false;
417         mainInterpSurface = false;
418         mainComplete = false;
419         mainSurfaceAndInterior = false;
420         moonComplete = false;
421         mainAndMoon = true;
422     }
423     else if (key == ESCAPE)
424         glutLeaveMainLoop();
425     else if (key == SPACEBAR)
426         pause = !pause;
427 }
428
429 void idle()
430 {
431     if (!pause)
432         glutPostRedisplay();
433 }
434
435 int main(int argc, char *argv[])
436 {
437     FILE *fp1 = fopen(FILE_NAME_1,"r"); //didymain surface vertices
438     if (fp1 == NULL)
439     {
440         printf("Error while reading file. Exiting...\n");
441         exit(EXIT_FAILURE);
442     }
443     N1 = FileRows(fp1);
444     xx1 = (float*)malloc(N1*sizeof(float));
445     yy1 = (float*)malloc(N1*sizeof(float));
446     zz1 = (float*)malloc(N1*sizeof(float));
447     InputVertices(fp1,xx1,yy1,zz1);
448
449     FILE *fp2 = fopen(FILE_NAME_2,"r"); //didymain surface indices
450     if (fp2 == NULL)
451     {
452         printf("Error while reading file. Exiting...\n");
453         exit(EXIT_FAILURE);
454     }
455     N2 = FileRows(fp2);
456     p = (int**)malloc(N2*sizeof(int*));
457     for (int i = 0; i < N2; i++)
458         p[i] = (int*)malloc(3*sizeof(int));
459     InputIndices(fp2);
460
461     //unit normal vectors. Used for the shading of didymain.
462     ux = (float*)malloc(N2*sizeof(float));
463     uy = (float*)malloc(N2*sizeof(float));
464     uz = (float*)malloc(N2*sizeof(float));
```

```
465     CalculateNormalVectors();
466
467     FILE *fp3 = fopen(FILE_NAME_3,"r"); //didymain surface vertices
468     if (fp3 == NULL)
469     {
470         printf("Error while reading file. Exiting...\n");
471         exit(EXIT_FAILURE);
472     }
473     N3 = FileRows(fp3);
474     x3 = (float*)malloc(N3*sizeof(float));
475     y3 = (float*)malloc(N3*sizeof(float));
476     z3 = (float*)malloc(N3*sizeof(float));
477     InputVertices(fp3,x3,y3,z3);
478
479     FILE *fp4 = fopen(FILE_NAME_4,"r"); //didymain surface vertices
480     if (fp4 == NULL)
481     {
482         printf("Error while reading file. Exiting...\n");
483         exit(EXIT_FAILURE);
484     }
485     N4 = FileRows(fp4);
486     x4 = (float*)malloc(N4*sizeof(float));
487     y4 = (float*)malloc(N4*sizeof(float));
488     z4 = (float*)malloc(N4*sizeof(float));
489     InputVertices(fp4,x4,y4,z4);
490
491     FILE *fp5 = fopen(FILE_NAME_5,"r"); //didymoon surface vertices
492     if (fp5 == NULL)
493     {
494         printf("Error while reading file. Exiting...\n");
495         exit(EXIT_FAILURE);
496     }
497     N5 = FileRows(fp5);
498     x5 = (float*)malloc(N5*sizeof(float));
499     y5 = (float*)malloc(N5*sizeof(float));
500     z5 = (float*)malloc(N5*sizeof(float));
501     InputVertices(fp5,x5,y5,z5);
502
503     FILE *fp6 = fopen(FILE_NAME_6,"r"); //didymain interpolated surface vertices
504     if (fp6 == NULL)
505     {
506         printf("Error while reading file. Exiting...\n");
507         exit(EXIT_FAILURE);
508     }
509     N6 = FileRows(fp6);
510     x6 = (float*)malloc(N6*sizeof(float));
511     y6 = (float*)malloc(N6*sizeof(float));
512     z6 = (float*)malloc(N6*sizeof(float));
513     InputVertices(fp6,x6,y6,z6);
514
515     glutInit(&argc,argv);
516     glutInitWindowSize(width,height);
517     glutInitWindowPosition(2000,50);
518     glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE|GLUT_DEPTH);
519     glutCreateWindow("Asteroid Models");
520     glutDisplayFunc(display);
521     glutReshapeFunc(reshape);
522     glutKeyboardFunc(keyboard);
523     glutIdleFunc(idle);
524     setup();
525     glutMainLoop();
526
527     free(xx1);
528     free(yy1);
529     free(zz1);
530
531     for (int i = 0; i < N2; ++i)
532         free(p[i]);
533     free(p);
```

```
534
535     free(x3);
536     free(y3);
537     free(z3);
538
539     free(x4);
540     free(y4);
541     free(z4);
542
543     free(x5);
544     free(y5);
545     free(z5);
546
547     fclose(fp1);
548     fclose(fp2);
549     fclose(fp3);
550     fclose(fp4);
551     fclose(fp5);
552     return 0;
553 }
```

codes/AsterModelOpenGL.c

Appendix D

The Binary in Orbit (Source Code)

```
/*

This code sets the two asteroids in orbits and calculates their evolution.
The primary (Didymain) is rotating with constant angular velocity around its z-axis,
while the secondary (Didymoon) is tidally locked to the primary.
Both asteroids are considered to consist of N1 and N2 particles respectively.
All N1 particles have mass (m1), while all N2 particles have
mass (m2). Runge-Kutta 4th order method is being used to solve the ODEs.

Case: Short time step (dt = 0.0005 TU = 2.66 sec)

Files used as input:
    1) main_complete_model.txt
    2) moon_complete_model.txt

Files produced as output:
    1) main_orbit.txt
    2) moon_orbit.txt
    3) orbital_elements.txt

*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

#define FILE_NAME_1 "main_complete_model.txt"
#define FILE_NAME_2 "moon_complete_model.txt"
#define FILE_NAME_3 "main_orbit.txt"
#define FILE_NAME_4 "moon_orbit.txt"
#define FILE_NAME_5 "orbital_elements.txt"

//system's parameters        2 months      2.66 sec | 0.04 min
const double t0 = 0.0, tmax = 972.96, dt = 0.0005; //t_sec = 5328.066*t_u
const double G  = 1.0; //gravitational constant
const double M1 = 0.9907; //total mass of Didymain
const double M2 = 0.0093; //total mass of Didymoon
const double w1 = -4.1147; //angular velocity of Didymain (w_u = 2*pi/T_u)
int N1,N2; //Didymain's and Didymoon's number of particles respectively
double m1,m2; //mass of each particle of Didymain and Didymoon

//decide how often are the data printed to the files
int printCounter = 0;
const int timeSkip = 50;

//Counts the number of rows of a file.
int FileRows(FILE *fp)
{
    int rows = 0;
```

```
51      char c;
52      while ((c = fgetc(fp)) != EOF)
53      {
54          if (c == '\n')
55              rows++;
56      }
57      rows++;
58      rewind(fp);
59      return rows;
60  }
61
62  //Reads the x,y,z vertices of the asteroid from the file.
63  void InputVertices(FILE *fp, double *x, double *y, double *z)
64  {
65      double tempx, tempy, tempz;
66      int i = 0;
67      while (fscanf(fp,"%lf %lf %lf",&tempx, &tempy, &tempz) != EOF)
68      {
69          x[i] = tempx;
70          y[i] = tempy;
71          z[i] = tempz;
72          i++;
73      }
74  }
75
76  //Shifts all the x,y,z vertices of Didymain, so that its COM coincides
77  //with O(0,0,0). This happens only once, before the calculation of the orbits.
78  void ShiftCOM(double *x, double *y, double *z, double m, double M, int N)
79  {
80      //calculate the COM
81      double X = 0.0, Y = 0.0, Z = 0.0;
82      for (int i = 0; i < N; i++)
83      {
84          X += m*x[i];
85          Y += m*y[i];
86          Z += m*z[i];
87      }
88      X /= M;
89      Y /= M;
90      Z /= M;
91      //shift the COM to O(0,0,0)
92      for (int i = 0; i < N; i++)
93      {
94          x[i] -= X;
95          y[i] -= Y;
96          z[i] -= Z;
97      }
98  }
99
100 //Calculates the length of a vector.
101 double Len(double x, double y, double z)
102 {
103     return sqrt(x*x + y*y + z*z);
104 }
105
106 //Calculates the distance between 2 points.
107 double Distance(double x1, double y1, double z1, double x2, double y2, double z2)
108 {
109     return sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) + (z2-z1)*(z2-z1));
110 }
111
112 //Calculates the maximum distance between the COM of an asteroid and all its
113 //other vertices. Both COMs are located at O(0,0,0) with respect to their frame.
114 double MaxDistance(double *x, double *y, double *z, int N)
115 {
116     double max = Len(x[0],y[0],z[0]);
117     for (int i = 1; i < N; i++)
118     {
119         if (Len(x[i],y[i],z[i]) > max)
```

```
120             max = Len(x[i],y[i],z[i]);
121     }
122     return max;
123 }
124
125 //Rotates Didymain to an angle w*dt around the z-axis (retrograde).
126 void RotateDidymain(double *x, double *y)
127 {
128     for (int i = 0; i < N1; i++)
129     {
130         double xx = x[i];
131         double yy = y[i];
132         x[i] = xx*cos(w1*dt) - yy*sin(w1*dt);
133         y[i] = xx*sin(w1*dt) + yy*cos(w1*dt);
134     }
135 }
136
137 //Rotates Didymoon so that it remains tidally locked to Didymain.
138 void RotateDidymoon(double *x,  double *y,  double *z,
139                     double X1,  double Y1,  double Z1,
140                     double X2,  double Y2,  double Z2,
141                     double X01, double Y01, double Z01,
142                     double X02, double Y02, double Z02)
143 {
144     double ax,ay,az,a_len; //vector a
145     ax = X02-X01;
146     ay = Y02-Y01;
147     az = Z02-Z01;
148     a_len = sqrt(ax*ax + ay*ay + az*az);
149     double bx,by,bz,b_len; //vector b
150     bx = X2-X1;
151     by = Y2-Y1;
152     bz = Z2-Z1;
153     b_len = sqrt(bx*bx + by*by + bz*bz);
154     double nx,ny,nz,n_len; //vector n (perpendicular to a and b)
155     nx = ay*bz - az*by;
156     ny = az*bx - ax*bz;
157     nz = ax*by - ay*bx;
158     n_len = sqrt(nx*nx + ny*ny + nz*nz);
159     double ux,uy,uz; //unit vector u <-- rotate Didymoon around this vector
160     ux = nx/n_len;
161     uy = ny/n_len;
162     uz = nz/n_len;
163
164     //f --> angle between vectors a and b (angle of rotation)
165     double cosf = (ax*bx + ay*by + az*bz)/(a_len*b_len);
166     double sinf = sqrt(1 - cosf*cosf);
167
168     double I[3][3]  = { {1,0,0}, {0,1,0}, {0,0,1} };
169     double W[3][3]  = { {0,-uz,uy}, {uz,0,-ux}, {-uy,ux,0} };
170     double W2[3][3] = { {-uz*uz - uy*uy, uy*ux, uz*ux},
171                         {ux*uy, -uz*uz - ux*ux, uz*uy},
172                         {ux*uz, uy*uz, -uy*uy - ux*ux} };
173     double R[3][3]; //rotation matrix
174     //calculate rotation matrix through Rodrigues formula
175     for (int i = 0; i < 3; i++)
176         for (int j = 0; j < 3; j++)
177             R[i][j] = I[i][j] + sinf*W[i][j] + (1-cosf)*W2[i][j];
178
179     //perform the rotation
180     for (int i = 0; i < N2; i++)
181     {
182         double xx = x[i];
183         double yy = y[i];
184         double zz = z[i];
185         x[i] = R[0][0]*xx + R[0][1]*yy + R[0][2]*zz;
186         y[i] = R[1][0]*xx + R[1][1]*yy + R[1][2]*zz;
187         z[i] = R[2][0]*xx + R[2][1]*yy + R[2][2]*zz;
188     }
```

```
189 }
190
191 double fX2(double vX2)
192 {
193     return vX2;
194 }
195
196 double fY2(double vY2)
197 {
198     return vY2;
199 }
200
201 double fZ2(double vZ2)
202 {
203     return vZ2;
204 }
205
206 ////////////////////////////////////////////////////////////////////////////
207
208 double fvX2(double *x1, double *y1, double *z1,
209             double  X1, double  X2, double  Y1, double Y2, double Z1, double Z2)
210 {
211     double sum = 0.0;
212     for (int i = 0; i < N1; i++)
213         sum += (X1-X2+x1[i])/pow((X1-X2+x1[i])*(X1-X2+x1[i]) +
214                                  (Y1-Y2+y1[i])*(Y1-Y2+y1[i]) +
215                                  (Z1-Z2+z1[i])*(Z1-Z2+z1[i]),3.0/2.0);
216     return G*m1*sum;
217 }
218
219 double fvY2(double *x1, double *y1, double *z1,
220             double  X1, double  X2, double  Y1, double Y2, double Z1, double Z2)
221 {
222     double sum = 0.0;
223     for (int i = 0; i < N1; i++)
224         sum += (Y1-Y2+y1[i])/pow((X1-X2+x1[i])*(X1-X2+x1[i]) +
225                                  (Y1-Y2+y1[i])*(Y1-Y2+y1[i]) +
226                                  (Z1-Z2+z1[i])*(Z1-Z2+z1[i]),3.0/2.0);
227     return G*m1*sum;
228 }
229
230 double fvZ2(double *x1, double *y1, double *z1,
231             double  X1, double  X2, double  Y1, double Y2, double Z1, double Z2)
232 {
233     double sum = 0.0;
234     for (int i = 0; i < N1; i++)
235         sum += (Z1-Z2+z1[i])/pow((X1-X2+x1[i])*(X1-X2+x1[i]) +
236                                  (Y1-Y2+y1[i])*(Y1-Y2+y1[i]) +
237                                  (Z1-Z2+z1[i])*(Z1-Z2+z1[i]),3.0/2.0);
238     return G*m1*sum;
239 }
240
241 void RK4(double *x1,  double *y1,  double *z1,
242          double  X1,  double *X2,  double  Y1, double *Y2, double Z1, double *Z2,
243          double *vX2, double *vY2, double *vZ2)
244 {
245     double kX2 = fX2(*vX2);
246     double kY2 = fY2(*vY2);
247     double kZ2 = fZ2(*vZ2);
248
249     double kvX2  = fvX2(x1,y1,z1, X1, *X2, Y1, *Y2, Z1, *Z2);
250     double kvY2  = fvY2(x1,y1,z1, X1, *X2, Y1, *Y2, Z1, *Z2);
251     double kvZ2  = fvZ2(x1,y1,z1, X1, *X2, Y1, *Y2, Z1, *Z2);
252
253     ////////////////////////////////////////////////////////////////
254
255     double lX2 = fX2(*vX2+(dt/2)*kvX2);
256     double lY2 = fY2(*vY2+(dt/2)*kvY2);
257     double lZ2 = fZ2(*vZ2+(dt/2)*kvZ2);
```

```
258
259     double lvX2 = fvX2(x1,y1,z1, X1, *X2+(dt/2)*kX2, Y1, *Y2+(dt/2)*kY2, Z1, *Z2+(dt/2)*kZ2);
260     double lvY2 = fvY2(x1,y1,z1, X1, *X2+(dt/2)*kX2, Y1, *Y2+(dt/2)*kY2, Z1, *Z2+(dt/2)*kZ2);
261     double lvZ2 = fvZ2(x1,y1,z1, X1, *X2+(dt/2)*kX2, Y1, *Y2+(dt/2)*kY2, Z1, *Z2+(dt/2)*kZ2);
262
263     ////////////////////////////////////////////////////////////////////
264
265     double mX2 = fX2(*vX2+(dt/2)*lvX2);
266     double mY2 = fY2(*vY2+(dt/2)*lvY2);
267     double mZ2 = fZ2(*vZ2+(dt/2)*lvZ2);
268
269     double mvX2 = fvX2(x1,y1,z1, X1, *X2+(dt/2)*lX2, Y1, *Y2+(dt/2)*lY2, Z1, *Z2+(dt/2)*lZ2);
270     double mvY2 = fvY2(x1,y1,z1, X1, *X2+(dt/2)*lX2, Y1, *Y2+(dt/2)*lY2, Z1, *Z2+(dt/2)*lZ2);
271     double mvZ2 = fvZ2(x1,y1,z1, X1, *X2+(dt/2)*lX2, Y1, *Y2+(dt/2)*lY2, Z1, *Z2+(dt/2)*lZ2);
272
273     ////////////////////////////////////////////////////////////////////
274
275     double nX2 = fX2(*vX2+dt*mvX2);
276     double nY2 = fY2(*vY2+dt*mvY2);
277     double nZ2 = fZ2(*vZ2+dt*mvZ2);
278
279     double nvX2 = fvX2(x1,y1,z1, X1, *X2+dt*mX2, Y1, *Y2+dt*mY2, Z1, *Z2+dt*mZ2);
280     double nvY2 = fvY2(x1,y1,z1, X1, *X2+dt*mX2, Y1, *Y2+dt*mY2, Z1, *Z2+dt*mZ2);
281     double nvZ2 = fvZ2(x1,y1,z1, X1, *X2+dt*mX2, Y1, *Y2+dt*mY2, Z1, *Z2+dt*mZ2);
282
283     ////////////////////////////////////////////////////////////////////
284
285     *X2  = *X2  + (dt/6.0)*(kX2 + 2*lX2 + 2*mX2 + nX2);
286     *Y2  = *Y2  + (dt/6.0)*(kY2 + 2*lY2 + 2*mY2 + nY2);
287     *Z2  = *Z2  + (dt/6.0)*(kZ2 + 2*lZ2 + 2*mZ2 + nZ2);
288
289     *vX2 = *vX2 + (dt/6.0)*(kvX2 + 2*lvX2 + 2*mvX2 + nvX2);
290     *vY2 = *vY2 + (dt/6.0)*(kvY2 + 2*lvY2 + 2*mvY2 + nvY2);
291     *vZ2 = *vZ2 + (dt/6.0)*(kvZ2 + 2*lvZ2 + 2*mvZ2 + nvZ2);
292 }
293
294 int main()
295 {
296     FILE *fp1 = fopen(FILE_NAME_1,"r"); //Didymain vertices
297     if (fp1 == NULL)
298     {
299         printf("Error while reading file. Exiting...\n");
300         exit(EXIT_FAILURE);
301     }
302     N1 = FileRows(fp1);
303     double *x1 = (double*)malloc(N1*sizeof(double));
304     double *y1 = (double*)malloc(N1*sizeof(double));
305     double *z1 = (double*)malloc(N1*sizeof(double));
306     InputVertices(fp1,x1,y1,z1);
307
308     FILE *fp2 = fopen(FILE_NAME_2,"r"); //Didymoon vertices
309     if (fp2 == NULL)
310     {
311         printf("Error while reading file. Exiting...\n");
312         exit(EXIT_FAILURE);
313     }
314     N2 = FileRows(fp2);
315     double *x2 = (double*)malloc(N2*sizeof(double));
316     double *y2 = (double*)malloc(N2*sizeof(double));
317     double *z2 = (double*)malloc(N2*sizeof(double));
318     InputVertices(fp2,x2,y2,z2);
319
320     m1 = M1/N1; //Didymain's particles mass
321     ShiftCOM(x1,y1,z1,m1,M1,N1);
322
323     m2 = M2/N2; //Didymoon's particles mass
324     //no need to shift the COM of Didymoon because by construction, the COM is
325     //located at O(0,0,0) at its coordinate system.
326
```

```
327     //Place the COM of the 2 asteroids at O(0,0,0) and set it to zero velocity
328     double X,Y,Z;
329     double vX,vY,vZ;
330     X = Y = Z = 0.0;
331     vX = vY = vZ = 0.0;
332
333     //initial conditions for the center of mass of Didymoon
334     double X2,Y2,Z2, X02,Y02,Z02;
335     double vX2,vY2,vZ2;
336     X2  = 1.18; Y2  = 0.0;  Z2  = 0.0;
337     vX2 = 0.0; vY2 = -0.921; vZ2 = 0.0;
338     X02 = X2;  Y02 = Y2;   Z02 = Z2; //auxiliary variables
339
340     //initial conditions for the center of mass of Didymain
341     double X1,Y1,Z1, X01,Y01,Z01;
342     double vX1,vY1,vZ1;
343     X1 = -M2*X2/M1; vX1 = -M2*vX2/M1;
344     Y1 = -M2*Y2/M1; vY1 = -M2*vY2/M1;
345     Z1 = -M2*Z2/M1; vZ1 = -M2*vZ2/M1;
346
347     double R2,V2,h2,hx2,hy2,hz2;
348     double a2,e2,i2; //semi major axis, eccentricity, plane inclination
349
350     //calculate the minimum allowed distance for the 2 center
351     //of masses to approach (collision detection)
352     const double maxDist1 = MaxDistance(x1,y1,z1, N1);
353     const double maxDist2 = MaxDistance(x2,y2,z2, N2);
354     const double minDist  = maxDist1 + maxDist2;
355
356     FILE *fp3 = fopen(FILE_NAME_3,"w"); //Didymain orbit
357     FILE *fp4 = fopen(FILE_NAME_4,"w"); //Didymoon orbit
358     FILE *fp5 = fopen(FILE_NAME_5,"w"); //orbital elements
359     printf("Calculating orbits. Please wait...\n");
360     clock_t t1,t2;
361     t1 = clock();
362     for (double t = t0; t <= tmax; t += dt)
363     {
364         printf("\r%.5lf %%",t/tmax*100);
365         fflush(stdout);
366
367         R2 = Len(X2,Y2,Z2);
368         V2 = Len(vX2,vY2,vZ2);
369         hx2 = Y2*vZ2 - Z2*vY2;
370         hy2 = Z2*vX2 - X2*vZ2;
371         hz2 = X2*vY2 - Y2*vX2;
372         h2 = Len(hx2,hy2,hz2);
373
374         a2 = 1.0/(2.0/R2 - V2*V2 /(G*(M1+M2)));
375         e2 = sqrt(1.0 - h2*h2/(G*(M1+M2)*a2));
376         i2 = acos(hz2/h2)*180.0/M_PI;
377
378         if (printCounter%timeSkip == 0) //print to the files for every timeSkip time steps
379         {
380             fprintf(fp3,"%lf %lf %lf %lf %lf %lf %lf\n",t,X1,Y1,Z1,vX1,vY1,vZ1);
381             fprintf(fp4,"%lf %lf %lf %lf %lf %lf %lf\n",t,X2,Y2,Z2,vX2,vY2,vZ2);
382             fprintf(fp5,"%lf %lf %lf %lf\n",t,a2,e2,i2);
383         }
384         printCounter++;
385
386         //collision detection criterion
387         if (Distance(X1,Y1,Z1, X2,Y2,Z2) < minDist)
388         {
389             printf("Collision detection at COM positions :\n");
390             printf("\t(X1,Y1,Z1) = (%.4lf, %.4lf, %.4lf)\n",X1,Y1,Z1);
391             printf("\t(X2,Y2,Z2) = (%.4lf, %.4lf, %.4lf)\n",X2,Y2,Z2);
392             printf("\tt = %lf\n",t);
393             break;
394         }
395
```

```
396         //update the COM coordinates of Didymoon through Runge-Kutta 4 method
397         RK4(x1,y1,z1, X1,&X2,Y1,&Y2,Z1,&Z2, &vX2,&vY2,&vZ2);
398         //update the COM coordinates of Didymain through the COM of the 2 asteroids
399         X1 = -M2*X2/M1; vX1 = -M2*vX2/M1;
400         Y1 = -M2*Y2/M1; vY1 = -M2*vY2/M1;
401         Z1 = -M2*Z2/M1; vZ1 = -M2*vZ2/M1;
402         //rotate Didymain to an angle w*dt around the z-axis
403         RotateDidymain(x1,y1);
404         //rotate Didymoon to an angle, so that it remains tidally locked
405         RotateDidymoon(x2,y2,z2, X1,Y1,Z1,X2,Y2,Z2, X01,Y01,Z01,X02,Y02,Z02);
406
407         //reset auxiliary variables for the next calculation
408         X01 = X1;
409         Y01 = Y1;
410         Z01 = Z1;
411
412         X02 = X2;
413         Y02 = Y2;
414         Z02 = Z2;
415     }
416     printf("\nOrbits calculated.\n");
417     t2 = clock();
418     double cpuTime = (t2-t1)/(double)CLOCKS_PER_SEC;
419     printf("Estimated completion time: %lf sec | %lf hrs\n",cpuTime,cpuTime/3600.0);
420     free(x1);
421     free(y1);
422     free(z1);
423     free(x2);
424     free(y2);
425     free(z2);
426     fclose(fp1);
427     fclose(fp2);
428     fclose(fp3);
429     fclose(fp4);
430     fclose(fp5);
431     return 0;
432 }
```

codes/MainMoonData.c

Appendix E

Visualisation of the Binary's Orbit (Source Code)

```
/*

This code creates a graphical 3D simulation of Didymos binary orbit.
Rotate the camera view by left clicking and moving the mouse.

Files used as input:
    1) main_surf_vertices.txt
    2) main_surf_indices.txt
    3) main_orbit.txt
    4) moon_orbit.txt

*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<stdbool.h>
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/freeglut.h>

#define FILE_NAME_1 "main_surf_vertices.txt"
#define FILE_NAME_2 "main_surf_indices.txt"
#define FILE_NAME_3 "main_orbit.txt"
#define FILE_NAME_4 "moon_orbit.txt"

#define ESCAPE 27 //escape ASCII character
#define SPACEBAR 32 //spacebar ASCII character

//mouse left click variables
int mousePreX = 0, mouseAftX;
int mousePreY = 0, mouseAftY;

bool pause = false;

////////////////////////////// Didymain ////////////////////////////////////////

int N1 = 0; //number of rows of the main_surf_vertices file
double *x = NULL;
double *y = NULL;
double *z = NULL;

int N2 = 0; //number of rows of the main_surf_indices file (number of triangles)
int **p = NULL;

//didymain COM
double *X1 = NULL;
```

```
48 double *Y1 = NULL;
49 double *Z1 = NULL;
50 //unit normal vector components (used for shading Didymain)
51 double *ux = NULL;
52 double *uy = NULL;
53 double *uz = NULL;
54
55 ////////////////////////////////////////////////////////////////////////////////
56
57 ////////////////////////////// Didymoon ////////////////////////////////////////
58
59 //ellipsoid (didymoon) semi-axes in km
60 const double a = 0.100;
61 const double b = 0.080;
62 const double c = 0.070;
63 //didymoon COM
64 double *X2 = NULL;
65 double *Y2 = NULL;
66 double *Z2 = NULL;
67
68 ////////////////////////////////////////////////////////////////////////////////
69
70 int Norb; //number of rows of the files 3,4
71 int j = 0; //current row of the files 3,4
72
73 //t_sec = 5328.066*t_u
74 double t = 0.0;
75 const double dt = 0.0005;
76 const double w = -4.1147; //angular velocity of didymain (w_u = 2*pi/T_u)
77 const int timeSkip = 50;
78
79 //graphics window size
80 GLsizei width  = 1000;
81 GLsizei height = 900;
82
83 //returns the angle in [0,2*pi]
84 double atan2pi(double b, double a)
85 {
86     double angle;
87     if (a > 0)
88         angle = atan(b/a);
89     else if (b >= 0 && a < 0)
90         angle = M_PI + atan(b/a);
91     else if (b < 0 && a < 0)
92         angle = -M_PI + atan(b/a);
93     else if (b > 0 && a == 0)
94         angle = M_PI/2;
95     else if (b < 0 && a == 0)
96         angle = -M_PI/2;
97
98     if (angle < 0)
99         angle += 2*M_PI;
100
101     return angle;
102 }
103
104
105 //Counts the number of rows of a file.
106 int FileRows(FILE *fp)
107 {
108     int rows = 0;
109     char c;
110     while ((c = fgetc(fp)) != EOF)
111     {
112         if (c == '\n')
113             rows++;
114     }
115     rows++;
116     rewind(fp);
```

```
117     return rows;
118 }
119
120 //Reads the (x,y,z) vertices of didymain from the file.
121 void InputVertices(FILE *fp)
122 {
123     double tempx, tempy, tempz;
124     int i = 0;
125     while (fscanf(fp,"%lf %lf %lf",&tempx, &tempy, &tempz) != EOF)
126     {
127         x[i] = tempx;
128         y[i] = tempy;
129         z[i] = tempz;
130         i++;
131     }
132 }
133
134 //Reads the observed indices of didymain from the file, that is,
135 //triads of points p1,p2,p3 that form triangles
136 //index i corresponds to the i-th row of the vertices file.
137 void InputIndices(FILE *fp)
138 {
139     int p1,p2,p3;
140     int i = 0;
141     while (fscanf(fp,"%d %d %d",&p1,&p2,&p3) != EOF)
142     {
143         //subtract 1 from all indices because the official file
144         //starts counting from 1, while I start from 0
145         p[i][0] = p1 - 1;
146         p[i][1] = p2 - 1;
147         p[i][2] = p3 - 1;
148         i++;
149     }
150 }
151
152 void InputDidymainOrbit(FILE *fp)
153 {
154     double t,x,y,z,vx,vy,vz;
155     int i = 0;
156     while (fscanf(fp,"%lf %lf %lf %lf %lf %lf %lf",&t,&x,&y,&z,&vx,&vy,&vz) != EOF)
157     {
158         X1[i] = x;
159         Y1[i] = y;
160         Z1[i] = z;
161         i++;
162     }
163 }
164
165 void InputDidymoonOrbit(FILE *fp)
166 {
167     double t,x,y,z,vx,vy,vz;
168     int i = 0;
169     while (fscanf(fp,"%lf %lf %lf %lf %lf %lf %lf",&t,&x,&y,&z,&vx,&vy,&vz) != EOF)
170     {
171         X2[i] = x;
172         Y2[i] = y;
173         Z2[i] = z;
174         i++;
175     }
176 }
177
178 //Calculates the cross product components of two vectors that
179 //are formed by 3 points. Last argument is used to determine
180 //the component of the cross product that will be returned.
181 //Possible values of 'coordinate': 0 --> x, 1 --> y, 2 --> z
182 double CrossProduct(double x1, double y1, double z1,
183                     double x2, double y2, double z2,
184                     double x3, double y3, double z3, double coordinate)
185 {
```

```
186       if (coordinate == 0) return (y2-y1)*(z3-z2) - (z2-z1)*(y3-y2);
187       if (coordinate == 1) return (z2-z1)*(x3-x2) - (x2-x1)*(z3-z2);
188       if (coordinate == 2) return (x2-x1)*(y3-y2) - (y2-y1)*(x3-x2);
189       //else
190       printf("Error while calculating the normal vectors. Exiting...\n");
191       exit(EXIT_FAILURE);
192 }
193
194 //(nx[i],ny[i],nz[i]) --> coordinates of the i-th normal vector, that is,
195 //the vector which is perpendicular to the triangle formed from
196 //the i-th triad of the indices file.
197 void CalculateNormalVectors()
198 {
199     double nx,ny,nz; //i-th normal vector
200     for (int i = 0; i < N2; i++)
201     {
202         nx = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
203                           x[p[i][1]], y[p[i][1]], z[p[i][1]],
204                           x[p[i][2]], y[p[i][2]], z[p[i][2]], 0);
205
206         ny = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
207                           x[p[i][1]], y[p[i][1]], z[p[i][1]],
208                           x[p[i][2]], y[p[i][2]], z[p[i][2]], 1);
209
210         nz = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
211                           x[p[i][1]], y[p[i][1]], z[p[i][1]],
212                           x[p[i][2]], y[p[i][2]], z[p[i][2]], 2);
213
214         //i-th unit normal vector
215         ux[i] = nx/sqrt(nx*nx + ny*ny + nz*nz);
216         uy[i] = ny/sqrt(nx*nx + ny*ny + nz*nz);
217         uz[i] = nz/sqrt(nx*nx + ny*ny + nz*nz);
218     }
219 }
220
221 void setup()
222 {
223     glEnable(GL_DEPTH_TEST);
224     glEnable(GL_LIGHT0);
225     glEnable(GL_LIGHTING);
226     glEnable(GL_COLOR_MATERIAL);
227     glShadeModel(GL_SMOOTH);
228     glEnable(GL_NORMALIZE);
229     glClearColor(0.0,0.0,0.0,0.0);
230 }
231
232 void reshape(GLsizei w, GLsizei h)
233 {
234     if (h == 0) h = 1;
235     glViewport(0,0,w,h);
236     glMatrixMode(GL_PROJECTION);
237     glLoadIdentity();
238     float AspectRatio = (float)w/(float)h;
239     gluPerspective(80.0,AspectRatio,0.1,100.0);
240     glMatrixMode(GL_MODELVIEW);
241     glLoadIdentity();
242     gluLookAt(1.5,1.5,1.0, 0,0,0, 0,0,1);
243 }
244
245 void Light()
246 {
247     glPushMatrix();
248         float lightPosition[] = {1.0,0.0,0.0,0.0};
249         glLightfv(GL_LIGHT0,GL_POSITION,lightPosition);
250     glPopMatrix();
251 }
252
253 void Didymain()
254 {
```

```
255     glColor3f(0.4f,0.4f,0.4f); //Didymain and Didymoon color
256     //draw didymain
257     glPushMatrix();
258         glTranslated(X1[j],Y1[j],Z1[j]);
259         glRotated(w*t*180/M_PI,0,0,1);
260         glBegin(GL_TRIANGLES);
261             for (int i = 0; i < N2; i++)
262             {
263                 glNormal3d(ux[i],uy[i],uz[i]); //for the shading
264                     glVertex3d(x[p[i][0]], y[p[i][0]], z[p[i][0]]);
265                     glVertex3d(x[p[i][1]], y[p[i][1]], z[p[i][1]]);
266                     glVertex3d(x[p[i][2]], y[p[i][2]], z[p[i][2]]);
267             }
268         glEnd();
269     glPopMatrix();
270 }
271
272 void Didymoon()
273 {
274     //draw didymoon
275     glPushMatrix();
276         glTranslated(X2[j],Y2[j],Z2[j]);
277         glRotated(atan2pi(Y2[j],X2[j])*180/M_PI, 0,0,1); //around z-axis
278         glRotated(acos(Z2[j]/sqrt(X2[j]*X2[j] + Y2[j]*Y2[j] + Z2[j]*Z2[j]))*180/M_PI + 90.0, 0,1,0);
279         glScaled(a,b,c);
280         glutSolidSphere(1.0,30,30);
281     glPopMatrix();
282
283     //draw didymoon's orbit
284     glColor3f(0.0f,0.8f,0.0f);
285     glPushMatrix();
286         glBegin(GL_LINE_STRIP);
287             for (int i = 0; i < j; i++)
288             {
289                 glVertex3d(X2[i],Y2[i],Z2[i]);
290                 glVertex3d(X2[i+1],Y2[i+1],Z2[i+1]);
291             }
292         glEnd();
293     glPopMatrix();
294 }
295
296 void RenderScene()
297 {
298     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
299
300     Light();
301     Didymain();
302     Didymoon();
303
304     t += timeSkip*dt;
305     j++;
306     if (j >= Norb)
307         glutLeaveMainLoop();
308     glutSwapBuffers();
309 }
310
311 void keyboard(unsigned char key, int x, int y)
312 {
313     if (key == ESCAPE)
314         glutLeaveMainLoop();
315     else if (key == SPACEBAR)
316         pause = !pause;
317 }
318
319 void motion(int x, int y)
320 {
321     mouseAftX = x;
322     if (mouseAftX - mousePreX > 0)
323         mousePreX = mouseAftX - 1;
```

```
324     else if (mouseAftX - mousePreX < 0)
325         mousePreX = mouseAftX + 1;
326     glRotatef(3*(mouseAftX - mousePreX), 0,0,1);
327     mousePreX = mouseAftX;
328 }
329
330 void idle()
331 {
332     if (!pause)
333         glutPostRedisplay();
334 }
335
336 int main(int argc, char *argv[])
337 {
338     FILE *fp1 = fopen(FILE_NAME_1,"r"); //didymain vertices
339     if (fp1 == NULL)
340     {
341         printf("Error while reading file. Exiting...\n");
342         exit(EXIT_FAILURE);
343     }
344     N1 = FileRows(fp1);
345     x = (double*)malloc(N1*sizeof(double));
346     y = (double*)malloc(N1*sizeof(double));
347     z = (double*)malloc(N1*sizeof(double));
348     InputVertices(fp1);
349
350     FILE *fp2 = fopen(FILE_NAME_2,"r"); //didymain indices
351     if (fp2 == NULL)
352     {
353         printf("Error while reading file. Exiting...\n");
354         exit(EXIT_FAILURE);
355     }
356     N2 = FileRows(fp2);
357     p = (int**)malloc(N2*sizeof(int*));
358     for (int i = 0; i < N2; i++)
359         p[i] = (int*)malloc(3*sizeof(int));
360     InputIndices(fp2);
361
362     //unit normal vectors. Used for the shading of didymain.
363     ux = (double*)malloc(N2*sizeof(double));
364     uy = (double*)malloc(N2*sizeof(double));
365     uz = (double*)malloc(N2*sizeof(double));
366     CalculateNormalVectors();
367
368     FILE *fp3 = fopen(FILE_NAME_3,"r"); //didymain orbit
369     if (fp3 == NULL)
370     {
371         printf("Error while reading file. Exiting...\n");
372         exit(EXIT_FAILURE);
373     }
374     Norb = FileRows(fp3);
375     X1 = (double*)malloc(Norb*sizeof(double));
376     Y1 = (double*)malloc(Norb*sizeof(double));
377     Z1 = (double*)malloc(Norb*sizeof(double));
378     InputDidymainOrbit(fp3);
379
380     FILE *fp4 = fopen(FILE_NAME_4,"r"); //didymoon orbit
381     if (fp4 == NULL)
382     {
383         printf("Error while reading file. Exiting...\n");
384         exit(EXIT_FAILURE);
385     }
386     X2 = (double*)malloc(Norb*sizeof(double));
387     Y2 = (double*)malloc(Norb*sizeof(double));
388     Z2 = (double*)malloc(Norb*sizeof(double));
389     InputDidymoonOrbit(fp4);
390
391     glutInit(&argc,argv);
392     glutInitWindowSize(width,height);
```

```
393     glutInitWindowPosition(100,50);
394     glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE|GLUT_DEPTH);
395     glutCreateWindow("Didymain - Didymoon orbit");
396     glutDisplayFunc(RenderScene);
397     glutReshapeFunc(reshape);
398     glutKeyboardFunc(keyboard);
399     glutMotionFunc(motion);
400     glutIdleFunc(idle);
401     setup();
402     glutMainLoop();
403 
404     free(x);
405     free(y);
406     free(z);
407     for (int i = 0; i < N2; i++)
408         free(p[i]);
409     free(p);
410     free(ux);
411     free(uy);
412     free(uz);
413     free(X1);
414     free(Y1);
415     free(Z1);
416     free(X2);
417     free(Y2);
418     free(Z2);
419     fclose(fp1);
420     fclose(fp2);
421     fclose(fp3);
422     fclose(fp4);
423     return 0;
424 }
```

codes/MainMoonOpenGL.c

Appendix F

Orbital Elements of the Binary Plots (Source Code)

```
#This script gathers the data of Didymain and Didymoon orbits
#for all integration cases (short, middle and long time step)
#and creates 2 pairs of plots of the orbital elements a,e,i as functions of time.
#The first pair depicts the evolution of the orbital elements for all the integration
#time. The second pair depicts exactly the same, but for little time. This
#enables us to observe the short time evolution of the functions a(t),e(t),i(t).

import numpy as np
import matplotlib.pyplot as plt

#path to directories for each time step case
path = ['Main_Moon_Short/','Main_Moon_Middle/','Main_Moon_Long/']
strType = ['short','middle','long']
critLines = np.array([1350,338,170])

figCounter = 0
for i in range(len(path)):
    fileName = path[i] + 'orbital_elements.txt'
    orbElem = np.loadtxt(fileName)
    str1 = strType[i]

    plt.figure(i + figCounter)
    plt.plot(orbElem[:,0],orbElem[:,1])
    plt.xlabel('time (TU)')
    plt.ylabel('semi-major axis a (km)')
    plt.title(strType[i])
    str2 = '_at_1.png'
    figName = str1 + str2
    plt.savefig(figName)

    figCounter += 1

    plt.figure(i + figCounter)
    plt.plot(orbElem[:,0],orbElem[:,2])
    plt.xlabel('time (TU)')
    plt.ylabel('eccentricity e')
    plt.title(strType[i])
    str2 = '_et_1.png'
    figName = str1 + str2
    plt.savefig(figName)

    figCounter += 1

    plt.figure(i + figCounter)
    plt.plot(orbElem[:,0],orbElem[:,3])
    plt.xlabel('time (TU)')
    plt.ylabel('inclination (deg)')
```

```
48      plt.title(strType[i])
49      str2 = '_it_1.png'
50      figName = str1 + str2
51      plt.savefig(figName)
52
53      figCounter += 1
54
55      plt.figure(i + figCounter)
56      plt.plot(orbElem[0:critLines[i],0],orbElem[0:critLines[i],1])
57      plt.xlabel('time (TU)')
58      plt.ylabel('semi-major axis a (km)')
59      plt.title(strType[i])
60      str2 = '_at_2.png'
61      figName = str1 + str2
62      plt.savefig(figName)
63
64      figCounter += 1
65
66      plt.figure(i + figCounter)
67      plt.plot(orbElem[0:critLines[i],0],orbElem[0:critLines[i],2])
68      plt.xlabel('time (TU)')
69      plt.ylabel('eccentricity e')
70      plt.title(strType[i])
71      str2 = '_et_2.png'
72      figName = str1 + str2
73      plt.savefig(figName)
74
75      figCounter += 1
76
77      plt.figure(i + figCounter)
78      plt.plot(orbElem[0:critLines[i],0],orbElem[0:critLines[i],3])
79      plt.xlabel('time (TU)')
80      plt.ylabel('inclination (deg)')
81      plt.title(strType[i])
82      str2 = '_it_2.png'
83      figName = str1 + str2
84      plt.savefig(figName)
85
86  plt.show()
```

codes/MainMoonPlots.py

Appendix G

Ejecta Cloud (Source Code)

```
/*

This code sets the two asteroids in orbit, calculates their orbital evolution and at the
same time calculates the orbits of 100 ejected particles from Didymoon's surface.
The code is meant to run in many processors, each calculating 100 orbits. To execute
the code, an argument (for the main() function) must be provided from the terminal. The
argument must be an unsigned integer which corresponds to the initial particle's
serial number. E.g. './EjectaData 400' will calculate 100 orbits: From particle 400
up to particle 499.

Three possible fates for each ejected particle: 1) The particle is trapped in orbit around
the binary for who knows how long. 2) The particle escapes the binary due to chaotic diffusion.
3) The particle crashes either on Didymain or Didymoon. Runge-Kutta 4th order method is being
used to solve all the ODEs.

Files used as input:
    1) main_complete_model.txt
    2) moon_complete_model.txt

Files produced as output:
    1) main_orbit.txt
    2) moon_orbit.txt
    100 .txt files that contain the orbits of 100 ejetced particles

*/

#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<math.h>
#include<time.h>
#include"mtwister.h" //meresenne twister RNG

#define FILE_NAME_1 "main_complete_model.txt"
#define FILE_NAME_2 "moon_complete_model.txt"
#define FILE_NAME_3 "main_orbit.txt"
#define FILE_NAME_4 "moon_orbit.txt"

//system's parameters          1 month     26.64 sec
const double t0 = 0.0, tmax = 486.48, dt = 0.005; //t_sec = 5328.066*t_u
const double G  = 1.0; //gravitational constant
const double M1 = 0.9907; //total mass of Didymain
const double M2 = 0.0093; //total mass of Didymoon
const double w1 = -4.1147; //angular velocity of Didymain (w_u = 2*pi/T_u)
int N1,N2; //Didymain's and Didymoon's number of particles respectively
double m1,m2; //mass o each particle of Didymain and Didymoon respectively
const double vEscInitial = 1.4; //initial escape velocity from the binary ~ 26 cm/s
const int numEjecta = 100; //number of ejected particles that will be calculated
const double escDistance = 20.0; //escape distance from the binary (in km)
const double ejectaConeAngle1 = M_PI/20.0;
```

```
51 const double ejectaConeAngle2 = M_PI/12.0;
52
53 //decide how often are the data printed to the files
54 int printCounter = 0;
55 const int timeSkip = 5;
56
57 //Didymoon's semi-axes (in km)
58 const double a = 0.100;
59 const double b = 0.080, phantom_b = 0.001;
60 const double c = 0.070;
61
62 //Didymoon's rotation matrix (initial state)
63 double R[3][3] = { {1.0, 0.0, 0.0},
64                    {0.0, 1.0, 0.0},
65                    {0.0, 0.0, 1.0} };
66
67 //Returns a random double in (a,b) from the uniform distribution.
68 double doubleRNG(double a, double b, mtRand *r)
69 {
70    double scale = RNG(r); //random number in (0,1)
71    return a + scale*(b-a);
72 }
73
74 //Counts the number of rows of a file.
75 int FileRows(FILE *fp)
76 {
77     int rows = 0;
78     char c;
79     while ((c = fgetc(fp)) != EOF)
80     {
81         if (c == '\n')
82             rows++;
83     }
84     rows++;
85     rewind(fp);
86     return rows;
87 }
88
89 //Reads the x,y,z vertices of the asteroid from the file.
90 void InputVertices(FILE *fp, double *x, double *y, double *z)
91 {
92     double tempx, tempy, tempz;
93     int i = 0;
94     while (fscanf(fp,"%lf %lf %lf",&tempx, &tempy, &tempz) != EOF)
95     {
96         x[i] = tempx;
97         y[i] = tempy;
98         z[i] = tempz;
99         i++;
100    }
101 }
102
103 //Shifts all the x,y,z vertices of Didymain, so that its COM coincides with O(0,0,0).
104 //This happens only once, before the calculation of the orbits.
105 void ShiftCOM(double *x, double *y, double *z, double m, double M, int N)
106 {
107     //calculate the COM
108     double X = 0.0, Y = 0.0, Z = 0.0;
109     for (int i = 0; i < N; i++)
110     {
111         X += m*x[i];
112         Y += m*y[i];
113         Z += m*z[i];
114     }
115     X /= M;
116     Y /= M;
117     Z /= M;
118     //shift the COM to O(0,0,0)
119     for (int i = 0; i < N; i++)
```

```
120     {
121         x[i] -= X;
122         y[i] -= Y;
123         z[i] -= Z;
124     }
125 }
126 
127 //Calculates the length of a vector.
128 double Len(double x, double y, double z)
129 {
130     return sqrt(x*x + y*y + z*z);
131 }
132 
133 //Calculates the distance between 2 points.
134 double Distance(double x1, double y1, double z1, double x2, double y2, double z2)
135 {
136     return sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) + (z2-z1)*(z2-z1));
137 }
138 
139 //Calculates the maximum distance between the COM of an asteroid and
140 //all its other vertices. Both COMs are located at O(0,0,0) in their frame.
141 double MaxDistance(double *x, double *y, double *z, int N)
142 {
143     double max = Len(x[0],y[0],z[0]);
144     for (int i = 1; i < N; i++)
145     {
146         if (Len(x[i],y[i],z[i]) > max)
147             max = Len(x[i],y[i],z[i]);
148     }
149     return max;
150 }
151 
152 //Rotates all the vertices of Didymain to an angle w*dt around the z-axis.
153 void RotateDidymain(double *x, double *y)
154 {
155     for (int i = 0; i < N1; i++)
156     {
157         double xx = x[i];
158         double yy = y[i];
159         x[i] = xx*cos(w1*dt) - yy*sin(w1*dt);
160         y[i] = xx*sin(w1*dt) + yy*cos(w1*dt);
161     }
162 }
163 
164 //Rotates Didymoon so that it remains tidally locked to Didymain.
165 void RotateDidymoon(double *x,  double *y,  double *z,
166                     double X1,  double Y1,  double Z1,
167                     double X2,  double Y2,  double Z2,
168                     double X01, double Y01, double Z01,
169                     double X02, double Y02, double Z02)
170 {
171     double ax,ay,az,a_len; //vector a
172     ax = X02-X01;
173     ay = Y02-Y01;
174     az = Z02-Z01;
175     a_len = sqrt(ax*ax + ay*ay + az*az);
176     double bx,by,bz,b_len; //vector b
177     bx = X2-X1;
178     by = Y2-Y1;
179     bz = Z2-Z1;
180     b_len = sqrt(bx*bx + by*by + bz*bz);
181     double nx,ny,nz,n_len; //vector n (perpendicular to a and b)
182     nx = ay*bz - az*by;
183     ny = az*bx - ax*bz;
184     nz = ax*by - ay*bx;
185     n_len = sqrt(nx*nx + ny*ny + nz*nz);
186     double ux,uy,uz; //unit vector u
187     ux = nx/n_len;
188     uy = ny/n_len;
```

```
189     uz = nz/n_len;
190
191     //f --> angle between vectors a and b
192     double cosf = (ax*bx + ay*by + az*bz)/(a_len*b_len);
193     double sinf = sqrt(1 - cosf*cosf);
194
195     double I[3][3] = { {1,0,0}, {0,1,0}, {0,0,1} };
196     double W[3][3] = { {0,-uz,uy}, {uz,0,-ux}, {-uy,ux,0} };
197     double W2[3][3] = { {-uz*uz - uy*uy, uy*ux, uz*ux},
198                         {ux*uy, -uz*uz - ux*ux, uz*uy},
199                         {ux*uz, uy*uz, -uy*uy - ux*ux} };
200
201     //calculate rotation matrix through Rodrigues formula
202     for (int i = 0; i < 3; i++)
203         for (int j = 0; j < 3; j++)
204             R[i][j] = I[i][j] + sinf*W[i][j] + (1-cosf)*W2[i][j];
205
206     //perform the rotation
207     for (int i = 0; i < N2; i++)
208     {
209         double xx = x[i];
210         double yy = y[i];
211         double zz = z[i];
212         x[i] = R[0][0]*xx + R[0][1]*yy + R[0][2]*zz;
213         y[i] = R[1][0]*xx + R[1][1]*yy + R[1][2]*zz;
214         z[i] = R[2][0]*xx + R[2][1]*yy + R[2][2]*zz;
215     }
216 }
217
218 double fX2(double vX2)
219 {
220     return vX2;
221 }
222
223 double fY2(double vY2)
224 {
225     return vY2;
226 }
227
228 double fZ2(double vZ2)
229 {
230     return vZ2;
231 }
232
233 ////////////////////////////////////////////////////////////////////////////
234
235 double fvX2(double *x1, double *y1, double *z1,
236             double  X1, double  X2, double  Y1, double Y2, double Z1, double Z2)
237 {
238     double sum = 0.0;
239     for (int i = 0; i < N1; i++)
240         sum += (X1-X2+x1[i])/pow((X1-X2+x1[i])*(X1-X2+x1[i]) +
241                                  (Y1-Y2+y1[i])*(Y1-Y2+y1[i]) +
242                                  (Z1-Z2+z1[i])*(Z1-Z2+z1[i]),3.0/2.0);
243     return G*m1*sum;
244 }
245
246 double fvY2(double *x1, double *y1, double *z1,
247             double  X1, double  X2, double  Y1, double Y2, double Z1, double Z2)
248 {
249     double sum = 0.0;
250     for (int i = 0; i < N1; i++)
251         sum += (Y1-Y2+y1[i])/pow((X1-X2+x1[i])*(X1-X2+x1[i]) +
252                                  (Y1-Y2+y1[i])*(Y1-Y2+y1[i]) +
253                                  (Z1-Z2+z1[i])*(Z1-Z2+z1[i]),3.0/2.0);
254     return G*m1*sum;
255 }
256
257 double fvZ2(double *x1, double *y1, double *z1,
```

```
258             double  X1, double  X2, double  Y1, double Y2, double Z1, double Z2)
259 {
260     double sum = 0.0;
261     for (int i = 0; i < N1; i++)
262         sum += (Z1-Z2+z1[i])/pow((X1-X2+x1[i])*(X1-X2+x1[i]) +
263                                  (Y1-Y2+y1[i])*(Y1-Y2+y1[i]) +
264                                  (Z1-Z2+z1[i])*(Z1-Z2+z1[i]),3.0/2.0);
265     return G*m1*sum;
266 }
267
268 void RK4_Moon(double *x1,  double *y1,  double *z1,
269               double  X1,  double *X2,  double  Y1, double *Y2, double Z1, double *Z2,
270               double *vX2, double *vY2, double *vZ2)
271 {
272     double kX2 = fX2(*vX2);
273     double kY2 = fY2(*vY2);
274     double kZ2 = fZ2(*vZ2);
275
276     double kvX2  = fvX2(x1,y1,z1, X1, *X2, Y1, *Y2, Z1, *Z2);
277     double kvY2  = fvY2(x1,y1,z1, X1, *X2, Y1, *Y2, Z1, *Z2);
278     double kvZ2  = fvZ2(x1,y1,z1, X1, *X2, Y1, *Y2, Z1, *Z2);
279
280     ////////////////////////////////////////////////////////////////////////
281
282     double lX2 = fX2(*vX2+(dt/2)*kvX2);
283     double lY2 = fY2(*vY2+(dt/2)*kvY2);
284     double lZ2 = fZ2(*vZ2+(dt/2)*kvZ2);
285
286     double lvX2 = fvX2(x1,y1,z1, X1, *X2+(dt/2)*kX2, Y1, *Y2+(dt/2)*kY2, Z1, *Z2+(dt/2)*kZ2);
287     double lvY2 = fvY2(x1,y1,z1, X1, *X2+(dt/2)*kX2, Y1, *Y2+(dt/2)*kY2, Z1, *Z2+(dt/2)*kZ2);
288     double lvZ2 = fvZ2(x1,y1,z1, X1, *X2+(dt/2)*kX2, Y1, *Y2+(dt/2)*kY2, Z1, *Z2+(dt/2)*kZ2);
289
290     ////////////////////////////////////////////////////////////////////////
291
292     double mX2 = fX2(*vX2+(dt/2)*lvX2);
293     double mY2 = fY2(*vY2+(dt/2)*lvY2);
294     double mZ2 = fZ2(*vZ2+(dt/2)*lvZ2);
295
296     double mvX2 = fvX2(x1,y1,z1, X1, *X2+(dt/2)*lX2, Y1, *Y2+(dt/2)*lY2, Z1, *Z2+(dt/2)*lZ2);
297     double mvY2 = fvY2(x1,y1,z1, X1, *X2+(dt/2)*lX2, Y1, *Y2+(dt/2)*lY2, Z1, *Z2+(dt/2)*lZ2);
298     double mvZ2 = fvZ2(x1,y1,z1, X1, *X2+(dt/2)*lX2, Y1, *Y2+(dt/2)*lY2, Z1, *Z2+(dt/2)*lZ2);
299
300     ////////////////////////////////////////////////////////////////////////
301
302     double nX2 = fX2(*vX2+dt*mvX2);
303     double nY2 = fY2(*vY2+dt*mvY2);
304     double nZ2 = fZ2(*vZ2+dt*mvZ2);
305
306     double nvX2 = fvX2(x1,y1,z1, X1, *X2+dt*mX2, Y1, *Y2+dt*mY2, Z1, *Z2+dt*mZ2);
307     double nvY2 = fvY2(x1,y1,z1, X1, *X2+dt*mX2, Y1, *Y2+dt*mY2, Z1, *Z2+dt*mZ2);
308     double nvZ2 = fvZ2(x1,y1,z1, X1, *X2+dt*mX2, Y1, *Y2+dt*mY2, Z1, *Z2+dt*mZ2);
309
310     ////////////////////////////////////////////////////////////////////////
311
312     *X2  = *X2  + (dt/6.0)*(kX2 + 2*lX2 + 2*mX2 + nX2);
313     *Y2  = *Y2  + (dt/6.0)*(kY2 + 2*lY2 + 2*mY2 + nY2);
314     *Z2  = *Z2  + (dt/6.0)*(kZ2 + 2*lZ2 + 2*mZ2 + nZ2);
315
316     *vX2 = *vX2 + (dt/6.0)*(kvX2 + 2*lvX2 + 2*mvX2 + nvX2);
317     *vY2 = *vY2 + (dt/6.0)*(kvY2 + 2*lvY2 + 2*mvY2 + nvY2);
318     *vZ2 = *vZ2 + (dt/6.0)*(kvZ2 + 2*lvZ2 + 2*mvZ2 + nvZ2);
319 }
320
321 ////////////////////////////////////////////////////////////////////////////
322
323 double fXp(double vXp)
324 {
325     return vXp;
326 }
```

```
327
328 double fYp(double vYp)
329 {
330     return vYp;
331 }
332
333 double fZp(double vZp)
334 {
335     return vZp;
336 }
337
338 ////////////////////////////////////////////////////////////////////////////
339
340 double fvXp(double *x1, double *x2, double *y1, double *y2, double *z1, double *z2,
341             double  X1, double  X2, double  Y1, double  Y2, double  Z1, double  Z2,
342             double  Xp, double  Yp, double  Zp)
343 {
344     double sum1 = 0.0; //force from Didymain
345     for (int i = 0; i < N1; i++)
346         sum1 += (X1-Xp+x1[i])/pow((X1-Xp+x1[i])*(X1-Xp+x1[i]) +
347                                   (Y1-Yp+y1[i])*(Y1-Yp+y1[i]) +
348                                   (Z1-Zp+z1[i])*(Z1-Zp+z1[i]),3.0/2.0);
349     double sum2 = 0.0; //force from Didymoon
350     for (int i = 0; i < N2; i++)
351         sum2 += (X2-Xp+x2[i])/pow((X2-Xp+x2[i])*(X2-Xp+x2[i]) +
352                                   (Y2-Yp+y2[i])*(Y2-Yp+y2[i]) +
353                                   (Z2-Zp+z2[i])*(Z2-Zp+z2[i]),3.0/2.0);
354     return G*(m1*sum1 + m2*sum2);
355 }
356
357 double fvYp(double *x1, double *x2, double *y1, double *y2, double *z1, double *z2,
358             double  X1, double  X2, double  Y1, double  Y2, double  Z1, double  Z2,
359             double  Xp, double  Yp, double  Zp)
360 {
361     double sum1 = 0.0; //force from Didymain
362     for (int i = 0; i < N1; i++)
363         sum1 += (Y1-Yp+y1[i])/pow((X1-Xp+x1[i])*(X1-Xp+x1[i]) +
364                                   (Y1-Yp+y1[i])*(Y1-Yp+y1[i]) +
365                                   (Z1-Zp+z1[i])*(Z1-Zp+z1[i]),3.0/2.0);
366     double sum2 = 0.0; //force from Didymoon
367     for (int i = 0; i < N2; i++)
368         sum2 += (Y2-Yp+y2[i])/pow((X2-Xp+x2[i])*(X2-Xp+x2[i]) +
369                                   (Y2-Yp+y2[i])*(Y2-Yp+y2[i]) +
370                                   (Z2-Zp+z2[i])*(Z2-Zp+z2[i]),3.0/2.0);
371     return G*(m1*sum1 + m2*sum2);
372 }
373
374 double fvZp(double *x1, double *x2, double *y1, double *y2, double *z1, double *z2,
375             double  X1, double  X2, double  Y1, double  Y2, double  Z1, double  Z2,
376             double  Xp, double  Yp, double  Zp)
377 {
378     double sum1 = 0.0; //force from Didymain
379     for (int i = 0; i < N1; i++)
380         sum1 += (Z1-Zp+z1[i])/pow((X1-Xp+x1[i])*(X1-Xp+x1[i]) +
381                                   (Y1-Yp+y1[i])*(Y1-Yp+y1[i]) +
382                                   (Z1-Zp+z1[i])*(Z1-Zp+z1[i]),3.0/2.0);
383     double sum2 = 0.0; //force from Didymoon
384     for (int i = 0; i < N2; i++)
385         sum2 += (Z2-Zp+z2[i])/pow((X2-Xp+x2[i])*(X2-Xp+x2[i]) +
386                                   (Y2-Yp+y2[i])*(Y2-Yp+y2[i]) +
387                                   (Z2-Zp+z2[i])*(Z2-Zp+z2[i]),3.0/2.0);
388     return G*(m1*sum1 + m2*sum2);
389 }
390
391 void RK4_Particle(double *Xp, double *Yp, double *Zp, double *vXp, double *vYp, double *vZp,
392                   double  X1, double  X2, double  Y1, double  Y2,  double  Z1,  double  Z2,
393                   double *x1, double *x2, double *y1, double *y2,  double *z1,  double *z2)
394 {
395     double kXp = fXp(*vXp);
```

```
396     double kYp = fYp(*vYp);
397     double kZp = fZp(*vZp);
398 
399     double kvXp  = fvXp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp,*Yp,*Zp);
400     double kvYp  = fvYp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp,*Yp,*Zp);
401     double kvZp  = fvZp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp,*Yp,*Zp);
402 
403     ////////////////////////////////////////////////////////////////////
404 
405     double lXp = fXp(*vXp+(dt/2)*kvXp);
406     double lYp = fYp(*vYp+(dt/2)*kvYp);
407     double lZp = fZp(*vZp+(dt/2)*kvZp);
408 
409     double lvXp = fvXp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*kXp, *Yp+(dt/2)*kYp, *Zp+(dt/2)*kZp);
410     double lvYp = fvYp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*kXp, *Yp+(dt/2)*kYp, *Zp+(dt/2)*kZp);
411     double lvZp = fvZp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*kXp, *Yp+(dt/2)*kYp, *Zp+(dt/2)*kZp);
412 
413     ////////////////////////////////////////////////////////////////////
414 
415     double mXp = fXp(*vXp+(dt/2)*lvXp);
416     double mYp = fYp(*vYp+(dt/2)*lvYp);
417     double mZp = fZp(*vZp+(dt/2)*lvZp);
418 
419     double mvXp = fvXp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*lXp, *Yp+(dt/2)*lYp, *Zp+(dt/2)*lZp);
420     double mvYp = fvYp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*lXp, *Yp+(dt/2)*lYp, *Zp+(dt/2)*lZp);
421     double mvZp = fvZp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*lXp, *Yp+(dt/2)*lYp, *Zp+(dt/2)*lZp);
422 
423     ////////////////////////////////////////////////////////////////////
424 
425     double nXp = fXp(*vXp+(dt/2)*mvXp);
426     double nYp = fYp(*vYp+(dt/2)*mvYp);
427     double nZp = fZp(*vZp+(dt/2)*mvZp);
428 
429     double nvXp = fvXp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*mXp, *Yp+(dt/2)*mYp, *Zp+(dt/2)*mZp);
430     double nvYp = fvYp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*mXp, *Yp+(dt/2)*mYp, *Zp+(dt/2)*mZp);
431     double nvZp = fvZp(x1,x2,y1,y2,z1,z2, X1,X2,Y1,Y2,Z1,Z2, *Xp+(dt/2)*mXp, *Yp+(dt/2)*mYp, *Zp+(dt/2)*mZp);
432 
433     ////////////////////////////////////////////////////////////////////////
434 
435     *Xp  = *Xp  + (dt/6.0)*(kXp + 2*lXp + 2*mXp + nXp);
436     *Yp  = *Yp  + (dt/6.0)*(kYp + 2*lYp + 2*mYp + nYp);
437     *Zp  = *Zp  + (dt/6.0)*(kZp + 2*lZp + 2*mZp + nZp);
438 
439     *vXp = *vXp + (dt/6.0)*(kvXp + 2*lvXp + 2*mvXp + nvXp);
440     *vYp = *vYp + (dt/6.0)*(kvYp + 2*lvYp + 2*mvYp + nvYp);
441     *vZp = *vZp + (dt/6.0)*(kvZp + 2*lvZp + 2*mvZp + nvZp);
442 }
443 
444 int main(int argc, char *argv[])
445 {
446     FILE *fp1 = fopen(FILE_NAME_1,"r"); //Didymain vertices
447     if (fp1 == NULL)
448     {
449         printf("Error while reading file. Exiting...\n");
450         exit(EXIT_FAILURE);
451     }
452     N1 = FileRows(fp1);
453     double *x1 = (double*)malloc(N1*sizeof(double));
454     double *y1 = (double*)malloc(N1*sizeof(double));
455     double *z1 = (double*)malloc(N1*sizeof(double));
456     InputVertices(fp1,x1,y1,z1);
457 
458     FILE *fp2 = fopen(FILE_NAME_2,"r"); //Didymoon vertices
459     if (fp2 == NULL)
460     {
461         printf("Error while reading file. Exiting...\n");
462         exit(EXIT_FAILURE);
463     }
464     N2 = FileRows(fp2);
```

```
465     double *x2 = (double*)malloc(N2*sizeof(double));
466     double *y2 = (double*)malloc(N2*sizeof(double));
467     double *z2 = (double*)malloc(N2*sizeof(double));
468     InputVertices(fp2,x2,y2,z2);
469
470     m1 = M1/N1; //Didymain's point masses
471     ShiftCOM(x1,y1,z1,m1,M1,N1);
472
473     m2 = M2/N2; //Didymoon's point masses
474     //no need to shift the COM of Didymoon because by construction, the COM is
475     //located at O(0,0,0) at its coordinate system.
476
477     //Place the COM of the 2 asteroids at O(0,0,0) and set it to zero velocity
478     double X,Y,Z;
479     double vX,vY,vZ;
480     X = Y = Z = 0.0;
481     vX = vY = vZ = 0.0;
482
483     //initial conditions for the center of mass of Didymoon
484     double X2,Y2,Z2, X02,Y02,Z02;
485     double vX2,vY2,vZ2;
486     X2  = 1.18; Y2  = 0.0; Z2  = 0.0;
487     vX2 = 0.0; vY2 = -0.921; vZ2 = 0.0;
488     X02 = X2;  Y02 = Y2;  Z02 = Z2; //auxiliary variables
489
490     //initial conditions for the center of mass of Didymain
491     double X1,Y1,Z1, X01,Y01,Z01;
492     double vX1,vY1,vZ1;
493     X1 = -M2*X2/M1; vX1 = -M2*vX2/M1;
494     Y1 = -M2*Y2/M1; vY1 = -M2*vY2/M1;
495     Z1 = -M2*Z2/M1; vZ1 = -M2*vZ2/M1;
496
497     //ejecta coordinates and velocitites
498     double *Xp  = (double*)malloc(numEjecta*sizeof(double));
499     double *Yp  = (double*)malloc(numEjecta*sizeof(double));
500     double *Zp  = (double*)malloc(numEjecta*sizeof(double));
501     double *vXp = (double*)malloc(numEjecta*sizeof(double));
502     double *vYp = (double*)malloc(numEjecta*sizeof(double));
503     double *vZp = (double*)malloc(numEjecta*sizeof(double));
504
505     //decides if ejecta i crashed
506     bool *crashed = (bool*)malloc(numEjecta*sizeof(bool));
507     //decides if ejecta i escaped
508     bool *escaped = (bool*)malloc(numEjecta*sizeof(bool));
509
510     double coneR1 = vEscInitial*tan(ejectaConeAngle1);
511     double coneR2 = vEscInitial*tan(ejectaConeAngle2);
512     FILE *fp3 = fopen(FILE_NAME_3,"w");
513     FILE *fp4 = fopen(FILE_NAME_4,"w");
514     FILE **fpEjecta = (FILE**)malloc(numEjecta*sizeof(FILE*));
515     char orbName[50]; //string buffer
516     mtRand r = seedRNG((unsigned int)time(NULL)); //seed the mersenne twister RNG
517     //initial conditions for the ejecta (Xp[i],Yp[i],Zp[i]) and (vXp[i],vYp[i],vZp[i])
518     for (int i = 0; i < numEjecta; ++i)
519     {
520         //all ejecta particles have the same initial position
521         Xp[i] = X2;
522         Yp[i] = -b - phantom_b;
523         Zp[i] = Z2;
524         for(;;) //validate that the initial velocity vector is be bounded by the 2 cones
525         {
526             //Monte Carlo the vx and vz components
527             vXp[i] = doubleRNG(-coneR2,coneR2, &r);
528             vZp[i] = doubleRNG(-coneR2,coneR2, &r);
529             double d = sqrt(vXp[i]*vXp[i] + vZp[i]*vZp[i]);
530             if (d > coneR1 && d < coneR2)
531                 break;
532         }
533         //Monte Carlo the vy component
```

```
534         vYp[i] = -doubleRNG(1.0,vEscInitial, &r);
535         crashed[i] = false; //assume no crashes initially
536         escaped[i] = false; //assume no escapes initially
537         sprintf(orbName,"orb_%d.txt", i + atoi(argv[1]));
538         fpEjecta[i] = fopen(orbName,"w");
539     }
540
541     //calculate the minimum allowed distance between
542     //ejecta and Didymain (for collision detection)
543     const double maxDist1 = MaxDistance(x1,y1,z1, N1);
544
545     //counters
546     int crashOnMain, crashOnMoon, escapes;
547     crashOnMain = crashOnMoon = escapes = 0;
548
549     printf("Calculating orbits. Please wait...\n");
550     clock_t t1,t2;
551     t1 = clock();
552     for (double t = t0; t <= tmax; t += dt)
553     {
554         printf("\r%.5lf %%",t/tmax*100);
555         fflush(stdout);
556
557         //print to the files for every timeSkip time steps
558         if (printCounter%timeSkip == 0)
559         {
560             fprintf(fp3,"%lf %lf %lf %lf %lf %lf %lf\n",t,X1,Y1,Z1,vX1,vY1,vZ1);
561             fprintf(fp4,"%lf %lf %lf %lf %lf %lf %lf\n",t,X2,Y2,Z2,vX2,vY2,vZ2);
562         }
563
564         //loop through all the particles and update their status
565         for (int i = 0; i < numEjecta; ++i)
566         {
567             if (crashed[i] || escaped[i])
568                 continue;
569
570             if (printCounter%timeSkip == 0) //print to the file for every timeSkip time steps
571                 fprintf(fpEjecta[i],"%lf %lf %lf %lf %lf %lf %lf\n",t,Xp[i],Yp[i],Zp[i],vXp[i],vYp[i],vZp[i]);
572
573             double XXp = R[0][0]*Xp[i] + R[0][1]*Yp[i] + R[0][2]*Zp[i];
574             double YYp = R[1][0]*Xp[i] + R[1][1]*Yp[i] + R[1][2]*Zp[i];
575             double ZZp = R[2][0]*Xp[i] + R[2][1]*Yp[i] + R[2][2]*Zp[i];
576
577             //collision detection between ejecta i and Didymain
578             if (Distance(Xp[i],Yp[i],Zp[i], X1,Y1,Z1) < maxDist1)
579             {
580                 crashed[i] = true;
581                 crashOnMain++;
582                 fclose(fpEjecta[i]);
583                 continue;
584             }
585             //collision detection between ejecta i and Didymoon
586             else if ((XXp-X2)*(XXp-X2)/(a*a) + (YYp-Y2)*(YYp-Y2)/(b*b) + (ZZp-Z2)*(ZZp-Z2)/(c*c) <= 1.0)
587             {
588                 crashed[i] = true;
589                 crashOnMoon++;
590                 fclose(fpEjecta[i]);
591                 continue;
592             }
593             //escape detection of ejecta i from the binary
594             else if ( Len(Xp[i],Yp[i],Zp[i]) > escDistance &&
595                       Len(vXp[i],vYp[i],vZp[i]) > sqrt(2.0*G*(M1+M2)/Len(Xp[i],Yp[i],Zp[i])) )
596             {
597                 escaped[i] = true;
598                 escapes++;
599                 fclose(fpEjecta[i]);
600                 continue;
601             }
602             //update the ejecta i coordinates through Runge-Kutta 4 method
```

```
603             RK4_Particle(&Xp[i],&Yp[i],&Zp[i], &vXp[i],&vYp[i],&vZp[i], X1,X2,Y1,Y2,Z1,Z2, x1,x2,y1,y2,z1,z2)
604         }
605
606         //update the COM coordinates of Didymoon through Runge-Kutta 4 method
607         RK4_Moon(x1,y1,z1, X1,&X2,Y1,&Y2,Z1,&Z2, &vX2,&vY2,&vZ2);
608         //update the COM coordinates of Didymain through the COM of the 2 asteroids
609         X1 = -M2*X2/M1; vX1 = -M2*vX2/M1;
610         Y1 = -M2*Y2/M1; vY1 = -M2*vY2/M1;
611         Z1 = -M2*Z2/M1; vZ1 = -M2*vZ2/M1;
612         //rotate Didymain to an angle w*dt around the z-axis
613         RotateDidymain(x1,y1);
614         //rotate Didymoon to an angle, so that it remains tidally locked to Didymain
615         RotateDidymoon(x2,y2,z2, X1,Y1,Z1,X2,Y2,Z2, X01,Y01,Z01,X02,Y02,Z02);
616         //reset auxiliary variables for the next calculation
617         X01 = X1;
618         Y01 = Y1;
619         Z01 = Z1;
620
621         X02 = X2;
622         Y02 = Y2;
623         Z02 = Z2;
624
625         printCounter++;
626     }
627     printf("\nOrbits calculated.\n");
628     t2 = clock();
629     double cpuTime = (t2-t1)/(double)CLOCKS_PER_SEC;
630     printf("Estimated completion time: %lf sec | %lf hrs\n",cpuTime,cpuTime/3600);
631
632     char ejecStatName[30];
633     sprintf(ejecStatName,"stat_%d_%d.txt",atoi(argv[1]),atoi(argv[1])+numEjecta);
634     FILE *fp5 = fopen(ejecStatName,"w");
635     fprintf(fp5,"%d %d %d",crashOnMain,crashOnMoon,escapes);
636
637     free(x1);
638     free(y1);
639     free(z1);
640     free(x2);
641     free(y2);
642     free(z2);
643     free(Xp);
644     free(Yp);
645     free(Zp);
646     free(vXp);
647     free(vYp);
648     free(vZp);
649     fclose(fp1);
650     fclose(fp2);
651     fclose(fp3);
652     fclose(fp4);
653     fclose(fp5);
654     for (int i = 0; i < numEjecta; ++i)
655     {
656         if ((!crashed[i]) && (!escaped[i]))
657             fclose(fpEjecta[i]);
658         //else the file was safely fclosed in the i-ejecta for loop
659     }
660     free(crashed);
661     free(escaped);
662     free(fpEjecta);
663     return 0;
664 }
```

codes/EjectaData.c

Appendix H

Visualisation of the Ejecta Cloud (Source Code)

```
/*

This code imports the orbits of Didymain and Didymoon along with the
orbits of 4000 impact ejecta and creates a 3D simulation.

Files used as input:
    1) main_surf_vertices.txt
    2) main_surf_indices.txt
    3) main_orbit.txt
    4) moon_orbit.txt
    and 4000 .txt files (ejecta orbits)

*/

#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<math.h>
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/freeglut.h>

#define FILE_NAME_1 "main_surf_vertices.txt"
#define FILE_NAME_2 "main_surf_indices.txt"
#define FILE_NAME_3 "main_orbit.txt"
#define FILE_NAME_4 "moon_orbit.txt"

#define ESCAPE 27 //escape ASCII character
#define SPACEBAR 32 //spacebar ASCII character

bool pause = false;

//camera viewing positions
bool cam_xyz = true;
bool cam_xz = false;
bool cam_xy = false;

//graphics window size
GLsizei winWidth  = 1200;
GLsizei winHeight = 900;
float aspectRatio; // winWidth/winHeight

////////////////////////////////// Didymain ////////////////////////////////////

int N1 = 0; //number of rows of the main_surf_vertices files
double *x = NULL;
double *y = NULL;
```

```
48 double *z = NULL;
49
50 int N2 = 0; //number of rows of the main_surf_indices file (number of triangles)
51 int **p = NULL;
52
53 //Didymain COM;
54 double *X1 = NULL;
55 double *Y1 = NULL;
56 double *Z1 = NULL;
57 //unit normal vector components (used for shading Didymain)
58 double *ux = NULL;
59 double *uy = NULL;
60 double *uz = NULL;
61
62 ////////////////////////////////////////////////////////////////////////////////
63
64 ////////////////////////////////// Didymoon ////////////////////////////////////
65
66 //ellipsoid (Didymoon) semi-axes in km
67 const double a = 0.100;
68 const double b = 0.080;
69 const double c = 0.070;
70 //Didymoon COM
71 double *X2 = NULL;
72 double *Y2 = NULL;
73 double *Z2 = NULL;
74
75 ////////////////////////////////////////////////////////////////////////////////
76
77 ////////////////////////////////// ejecta //////////////////////////////////////
78
79 //ejecta coordinates
80 double **Xp = NULL;
81 double **Yp = NULL;
82 double **Zp = NULL;
83 int *rowsInFile = NULL;
84 const int numEjecta = 4000;
85
86 ////////////////////////////////////////////////////////////////////////////////
87
88 int Norb; //number of rows of the files 3,4
89 int k = 0; //current row of the files 3,4
90
91 //t_sec = 5328.066*t_u
92 double t = 0.0;
93 const double dt = 0.005;
94 const double w = -4.1147; //angular velocity of Didymain (w_u = 2*pi/T_u)
95 const int timeSkip = 5;
96 int days = 0, hours = 0, minutes = 0; //physical time rendered on screen
97
98 //Returns the angle in [0,2*pi]
99 double atan2pi(double b, double a)
100 {
101     double angle;
102     if (a > 0)
103         angle = atan(b/a);
104     else if (b >= 0 && a < 0)
105         angle = M_PI + atan(b/a);
106     else if (b < 0 && a < 0)
107         angle = -M_PI + atan(b/a);
108     else if (b > 0 && a == 0)
109         angle = M_PI/2;
110     else if (b < 0 && a == 0)
111         angle = -M_PI/2;
112
113     if (angle < 0)
114         angle += 2*M_PI;
115
116     return angle;
```

```
117 }
118
119 //Counts the number of rows of a file.
120 int FileRows(FILE *fp)
121 {
122     int rows = 0;
123     char c;
124     while ((c = fgetc(fp)) != EOF)
125     {
126         if (c == '\n')
127             rows++;
128     }
129     rows++;
130     rewind(fp);
131     return rows;
132 }
133
134 //Reads the observed vertices x,y,z of the asteroid from the file.
135 void InputVertices(FILE *fp)
136 {
137     double tempx, tempy, tempz;
138     int i = 0;
139     while (fscanf(fp,"%lf %lf %lf",&tempx, &tempy, &tempz) != EOF)
140     {
141         x[i] = tempx;
142         y[i] = tempy;
143         z[i] = tempz;
144         i++;
145     }
146 }
147
148 //Reads the observed indices of Didymain from the file, that is,
149 //triads of points p1,p2,p3 that form triangles
150 //index i corresponds to the i-th row of the vertices file.
151 void InputIndices(FILE *fp)
152 {
153     int p1,p2,p3;
154     int i = 0;
155     while (fscanf(fp,"%d %d %d",&p1,&p2,&p3) != EOF)
156     {
157         //subtract 1 from all indices because the official file
158         //starts counting from 1, while I start from 0
159         p[i][0] = p1 - 1;
160         p[i][1] = p2 - 1;
161         p[i][2] = p3 - 1;
162         i++;
163     }
164 }
165
166 void InputDidymainOrbit(FILE *fp)
167 {
168     double t,x,y,z,vx,vy,vz;
169     int i = 0;
170     while (fscanf(fp,"%lf %lf %lf %lf %lf %lf %lf",&t,&x,&y,&z,&vx,&vy,&vz) != EOF)
171     {
172         X1[i] = x;
173         Y1[i] = y;
174         Z1[i] = z;
175         i++;
176     }
177 }
178
179 void InputDidymoonOrbit(FILE *fp)
180 {
181     double t,x,y,z,vx,vy,vz;
182     int i = 0;
183     while (fscanf(fp,"%lf %lf %lf %lf %lf %lf %lf",&t,&x,&y,&z,&vx,&vy,&vz) != EOF)
184     {
185         X2[i] = x;
```

```
186         Y2[i] = y;
187         Z2[i] = z;
188         i++;
189     }
190 }
191
192 void InputEjectaOrbit(FILE *fp, int i)
193 {
194     double t,x,y,z,vx,vy,vz;
195     int j = 0;
196     while (fscanf(fp,"%lf %lf %lf %lf %lf %lf %lf",&t,&x,&y,&z,&vx,&vy,&vz) != EOF)
197     {
198         Xp[i][j] = x;
199         Yp[i][j] = y;
200         Zp[i][j] = z;
201         j++;
202     }
203 }
204
205 //Calculates the cross product components of two vectors that are formed by 3 points.
206 //Last argument is used to determine the component of the cross product that will be returned.
207 //Possible values of 'coordinate': 0 --> x, 1 --> y, 2 --> z
208 double CrossProduct(double x1, double y1, double z1,
209                     double x2, double y2, double z2,
210                     double x3, double y3, double z3, double coordinate)
211 {
212      if (coordinate == 0) return (y2-y1)*(z3-z2) - (z2-z1)*(y3-y2);
213      if (coordinate == 1) return (z2-z1)*(x3-x2) - (x2-x1)*(z3-z2);
214      if (coordinate == 2) return (x2-x1)*(y3-y2) - (y2-y1)*(x3-x2);
215      //else
216      printf("Error while calculating the normal vectors. Exiting...\n");
217      exit(EXIT_FAILURE);
218 }
219
220 //(ux[i],uy[i],uz[i]) --> coordinates of the i-th unit normal vector, that is,
221 //the vector which is perpendicular to the triangle formed from the i-th triad of
222 //the indices file.
223 void CalculateNormalVectors()
224 {
225     double nx,ny,nz; //i-th normal vector
226     for (int i = 0; i < N2; i++)
227     {
228         nx = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
229                           x[p[i][1]], y[p[i][1]], z[p[i][1]],
230                           x[p[i][2]], y[p[i][2]], z[p[i][2]], 0);
231
232         ny = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
233                           x[p[i][1]], y[p[i][1]], z[p[i][1]],
234                           x[p[i][2]], y[p[i][2]], z[p[i][2]], 1);
235
236         nz = CrossProduct(x[p[i][0]], y[p[i][0]], z[p[i][0]],
237                           x[p[i][1]], y[p[i][1]], z[p[i][1]],
238                           x[p[i][2]], y[p[i][2]], z[p[i][2]], 2);
239
240         //i-th unit normal vector
241         ux[i] = nx/sqrt(nx*nx + ny*ny + nz*nz);
242         uy[i] = ny/sqrt(nx*nx + ny*ny + nz*nz);
243         uz[i] = nz/sqrt(nx*nx + ny*ny + nz*nz);
244     }
245 }
246
247 void setupGL()
248 {
249     glEnable(GL_DEPTH_TEST);
250     glEnable(GL_LIGHT0);
251     glEnable(GL_LIGHTING);
252     glEnable(GL_COLOR_MATERIAL);
253     glEnable(GL_NORMALIZE);
254     glPointSize(3.0f);
```

```
255     glClearColor(0.0,0.0,0.0,0.0);
256 }
257
258 void reshape(GLsizei w, GLsizei h)
259 {
260     if (h == 0) h = 1;
261     winWidth = w;
262     winHeight = h;
263     glViewport(0,0,w,h);
264     glMatrixMode(GL_PROJECTION);
265     glLoadIdentity();
266     aspectRatio = (float)w/h;
267     gluPerspective(60.0,aspectRatio,0.1,100.0);
268     glMatrixMode(GL_MODELVIEW);
269     glLoadIdentity();
270 }
271
272 //Places the camera in the 3D world (3 possible positions)
273 void Camera()
274 {
275     glLoadIdentity();
276     if (cam_xyz)
277         gluLookAt(2,2,2, 0,0,0, 0,0,1);
278     else if (cam_xy)
279         gluLookAt(0,0,4, 0,0,0, 0,1,0);
280     else if (cam_xz)
281         gluLookAt(0,-4,0, 0,0,0, 0,0,1);
282 }
283
284 //Renders the light
285 void Light()
286 {
287     glPushMatrix();
288         float lightPosition[] = {1.0,0.0,0.0,0.0};
289         glLightfv(GL_LIGHT0,GL_POSITION,lightPosition);
290     glPopMatrix();
291 }
292
293 //Renders Didymain
294 void Didymain()
295 {
296     glColor3f(0.4f,0.4f,0.4f);
297     glPushMatrix();
298         glTranslated(X1[k],Y1[k],Z1[k]);
299         glRotated(w*t*180/M_PI,0,0,1);
300         glBegin(GL_TRIANGLES);
301             for (int i = 0; i < N2; i++)
302             {
303                 glNormal3d(ux[i],uy[i],uz[i]); //for the shading
304                     glVertex3d(x[p[i][0]], y[p[i][0]], z[p[i][0]]);
305                     glVertex3d(x[p[i][1]], y[p[i][1]], z[p[i][1]]);
306                     glVertex3d(x[p[i][2]], y[p[i][2]], z[p[i][2]]);
307             }
308         glEnd();
309     glPopMatrix();
310 }
311
312 //Renders Didymoon
313 void Didymoon()
314 {
315     glColor3f(0.4f,0.4f,0.4f);
316     glPushMatrix();
317         glTranslated(X2[k],Y2[k],Z2[k]);
318         glRotated(atan2pi(Y2[k],X2[k])*180/M_PI, 0,0,1);
319         glRotated(acos(Z2[k]/sqrt(X2[k]*X2[k] +
320                                   Y2[k]*Y2[k] +
321                                   Z2[k]*Z2[k]))*180/M_PI + 90.0, 0,1,0);
322         glScaled(a,b,c);
323         glutSolidSphere(1.0,20,20);
```

```
324     glPopMatrix();
325 }
326
327 //Renders the ejecta
328 void Ejecta()
329 {
330     glColor3f(0.8f,0.8f,0.8f);
331     for (int i = 0; i < numEjecta; ++i)
332     {
333         if (k < rowsInFile[i])
334         {
335             //in case the program crashes, comment the commands: glPushMatrix() up to glPopMatrix()
336             //and uncomment the commands: glBegin(GL_POINTS) up to glEnd()
337             glPushMatrix();
338                 glTranslated(Xp[i][k],Yp[i][k],Zp[i][k]);
339                 glutSolidSphere(0.01,5,5);
340             glPopMatrix();
341             /*glBegin(GL_POINTS);
342                 glVertex3d(Xp[i][k],Yp[i][k],Zp[i][k]);
343             glEnd();*/
344         }
345     }
346 }
347
348 //Displays the physical time of the system's evolution on the window
349 void Timer()
350 {
351     glDisable(GL_LIGHTING); //disable light calculations
352     //switch to 2D orthographic view
353     glMatrixMode(GL_PROJECTION);
354     glLoadIdentity();
355     if (winWidth >= winHeight)
356         gluOrtho2D(-winWidth*aspectRatio,winWidth*aspectRatio,-winHeight,winHeight);
357     else
358         gluOrtho2D(-winWidth,winWidth,-winHeight*aspectRatio,winHeight*aspectRatio);
359     glMatrixMode(GL_MODELVIEW);
360     glLoadIdentity();
361
362     //elapsed time
363     days = (5328.066*t/(24.0*3600));
364     hours = (5328.066*t/3600.0);
365     minutes = (5328.066*t/60.0);
366     char tm[50]; //time string
367     sprintf(tm,"day %d | hr %d | min %d", days, hours%24, minutes%60);
368     glColor3f(1.0f,1.0f,0.0f);
369     //place the time on the top left of the window
370     if (winWidth >= winHeight)
371         glRasterPos2f((-winWidth+100)*aspectRatio, winHeight-100);
372     else
373         glRasterPos2f(-winWidth+100, (winHeight-100)*aspectRatio);
374     glutBitmapString(GLUT_BITMAP_HELVETICA_18,tm);
375
376     //switch back to perspective view
377     glViewport(0,0,winWidth,winHeight);
378     glMatrixMode(GL_PROJECTION);
379     glLoadIdentity();
380     gluPerspective(60.0,aspectRatio,0.1,100.0);
381     glMatrixMode(GL_MODELVIEW);
382     glLoadIdentity();
383     glEnable(GL_LIGHTING); //restore light calculations
384 }
385
386 void display()
387 {
388     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
389
390     Camera();
391     Light();
392     Didymain();
```

```
393     Didymoon();
394     Ejecta();
395     Timer();
396 
397     t += timeSkip*dt;
398     k++;
399     if (k >= Norb)
400         glutLeaveMainLoop();
401     glutSwapBuffers();
402 }
403 
404 void keyboard(unsigned char key, int x, int y)
405 {
406     if (key == '1')
407     {
408         cam_xyz = true;
409         cam_xy = false;
410         cam_xz = false;
411     }
412     else if (key == '2')
413     {
414         cam_xyz = false;
415         cam_xy = true;
416         cam_xz = false;
417     }
418     else if (key == '3')
419     {
420         cam_xyz = false;
421         cam_xy = false;
422         cam_xz = true;
423     }
424     else if (key == ESCAPE)
425         glutLeaveMainLoop();
426     else if (key == SPACEBAR)
427         pause = !pause;
428 }
429 
430 //Refresh the frame only if not paused
431 void idle()
432 {
433     if (!pause)
434         glutPostRedisplay();
435 }
436 
437 int main(int argc, char *argv[])
438 {
439     FILE *fp1 = fopen(FILE_NAME_1,"r"); //Didymain vertices
440     if (fp1 == NULL)
441     {
442         printf("Error while reading file. Exiting...\n");
443         exit(EXIT_FAILURE);
444     }
445     N1 = FileRows(fp1);
446     x = (double*)malloc(N1*sizeof(double));
447     y = (double*)malloc(N1*sizeof(double));
448     z = (double*)malloc(N1*sizeof(double));
449     InputVertices(fp1);
450 
451     FILE *fp2 = fopen(FILE_NAME_2,"r"); //Didymain indices
452     if (fp2 == NULL)
453     {
454         printf("Error while reading file. Exiting...\n");
455         exit(EXIT_FAILURE);
456     }
457     N2 = FileRows(fp2);
458     p = (int**)malloc(N2*sizeof(int*));
459     for (int i = 0; i < N2; i++)
460         p[i] = (int*)malloc(3*sizeof(int));
461     InputIndices(fp2);
```

```
462
463     //unit normal vectors. Used for the shading of Didymain.
464     ux = (double*)malloc(N2*sizeof(double));
465     uy = (double*)malloc(N2*sizeof(double));
466     uz = (double*)malloc(N2*sizeof(double));
467     CalculateNormalVectors();
468
469     FILE *fp3 = fopen(FILE_NAME_3,"r"); //Didymain orbit
470     if (fp3 == NULL)
471     {
472         printf("Error while reading file. Exiting...\n");
473         exit(EXIT_FAILURE);
474     }
475     Norb = FileRows(fp3);
476     X1 = (double*)malloc(Norb*sizeof(double));
477     Y1 = (double*)malloc(Norb*sizeof(double));
478     Z1 = (double*)malloc(Norb*sizeof(double));
479     InputDidymainOrbit(fp3);
480
481     FILE *fp4 = fopen(FILE_NAME_4,"r"); //Didymoon orbit
482     if (fp4 == NULL)
483     {
484         printf("Error while reading file. Exiting...\n");
485         exit(EXIT_FAILURE);
486     }
487     X2 = (double*)malloc(Norb*sizeof(double));
488     Y2 = (double*)malloc(Norb*sizeof(double));
489     Z2 = (double*)malloc(Norb*sizeof(double));
490     InputDidymoonOrbit(fp4);
491
492     FILE **fpEjecta = (FILE**)malloc(numEjecta*sizeof(FILE*)); //ejecta orbits
493     rowsInFile = (int*)malloc(numEjecta*sizeof(int));
494     Xp = (double**)malloc(numEjecta*sizeof(double*));
495     Yp = (double**)malloc(numEjecta*sizeof(double*));
496     Zp = (double**)malloc(numEjecta*sizeof(double*));
497     char orbName[15];
498     for (int i = 0; i < numEjecta; ++i)
499     {
500         printf("\rLoading files %d/%d",i,numEjecta-1);
501         fflush(stdout);
502         sprintf(orbName,"Ejecta_Orbits/orb_%d.txt",i);
503         fpEjecta[i] = fopen(orbName,"r");
504         if (fpEjecta[i] == NULL)
505         {
506             printf("\nError while reading file. Exiting...\n");
507             exit(EXIT_FAILURE);
508         }
509         rowsInFile[i] = FileRows(fpEjecta[i]);
510         Xp[i] = (double*)malloc(rowsInFile[i]*sizeof(double));
511         Yp[i] = (double*)malloc(rowsInFile[i]*sizeof(double));
512         Zp[i] = (double*)malloc(rowsInFile[i]*sizeof(double));
513         InputEjectaOrbit(fpEjecta[i],i);
514     }
515     printf("\n");
516
517     glutInit(&argc,argv);
518     glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE|GLUT_DEPTH);
519     glutInitWindowPosition((int)((glutGet(GLUT_SCREEN_WIDTH) - (int)winWidth)/2.0f),
520                            (int)((glutGet(GLUT_SCREEN_HEIGHT) - (int)winHeight)/2.0f));
521     glutInitWindowSize(winWidth,winHeight);
522     glutCreateWindow("Ejecta orbits");
523     glutDisplayFunc(display);
524     glutReshapeFunc(reshape);
525     glutKeyboardFunc(keyboard);
526     glutIdleFunc(idle);
527     setupGL();
528     glutMainLoop();
529
530     free(x);
```

```
531     free(y);
532     free(z);
533     for (int i = 0; i < N2; i++)
534         free(p[i]);
535     free(p);
536     free(ux);
537     free(uy);
538     free(uz);
539     free(X1);
540     free(Y1);
541     free(Z1);
542     free(X2);
543     free(Y2);
544     free(Z2);
545     for (int i = 0; i < numEjecta; ++i)
546     {
547         free(Xp[i]);
548         free(Yp[i]);
549         free(Zp[i]);
550     }
551     free(Xp);
552     free(Yp);
553     free(Zp);
554 
555     fclose(fp1);
556     fclose(fp2);
557     fclose(fp3);
558     fclose(fp4);
559     for (int i = 0; i < numEjecta; ++i)
560         fclose(fpEjecta[i]);
561     free(fpEjecta);
562     free(rowsInFile);
563     return 0;
564 }
```

codes/EjectaOpenGL.c

Appendix I

Ejecta Orbits Plots (Source Code)

```python
#This script picks a sample of ejected particles in random
#from the directory 'Ejecta_Orbits' and plots their functions
#r(t) and v(t) with respect to the global inertial frame.

import numpy as np
import matplotlib.pyplot as plt

totalParticles = 4000
sample = 21 #21 r(t) and 21 v(t) functions are to be plotted
orbIndex = np.random.randint(totalParticles, size = sample)
print('Particles selected:',orbIndex)
str1 = 'Ejecta_Orbits/orb_'
str2 = '.txt'
str3 = 'r(t)_'
str4 = 'v(t)_'
str5 = '.png'
figCounter = 0
for i in range(sample):
    strIndex = str(orbIndex[i])
    fileName = str1 + strIndex + str2
    orb = np.loadtxt(fileName)
    r = np.zeros((len(orb),1), dtype = float)
    v = np.zeros((len(orb),1), dtype = float)
    for j in range(len(orb)):
        #calculate distance and velocity from the x,y,z,vx,vy,vz components
        r[j] = np.sqrt( orb[j][1]**2 + orb[j][2]**2 + orb[j][3]**2 )
        v[j] = np.sqrt( orb[j][4]**2 + orb[j][5]**2 + orb[j][6]**2 )
    #plot the function r(t)
    plt.figure(i + figCounter)
    plt.plot(orb[:,0],r)
    plt.xlabel('time (TU)')
    plt.ylabel('distance r (km)')
    plt.title(str3 + strIndex)
    figName = str3 + strIndex + str5
    plt.savefig(figName)

    figCounter = figCounter + 1

    #plot the function v(t)
    plt.figure(i + figCounter)
    plt.plot(orb[:,0],v)
    plt.xlabel('time (TU)')
    plt.ylabel('velocity v (km / TU)')
    plt.title(str4 + strIndex)
    figName = str4 + strIndex + str5
    plt.savefig(figName)

plt.show()
```

codes/EjectaSamplePlots.py

Appendix J

Ejecta Population (Source Code)

```python
#This code calculates the population of the ejecta near Didymos binary
#as a function of time. The algorithm collects all the final moments
#t_final of calculations done at each of the 4000 particles and uses it
#in order to decide whether the particle i still exists (hasn't escaped or
#crashed) at time t or not.

import numpy as np
import matplotlib.pyplot as plt

totalParticles = 4000
tmax = 486.47
dt = 0.025
deathTime = [] #last time t of particle i
fp = open('population.txt','w') #it shall contain the data (t,N(t))

print('Collecting death times...')
for i in range(totalParticles): #loop through all 4000 files
    orb = np.loadtxt('Ejecta_Orbits/orb_' + str(i) + '.txt')
    lastLineTime = orb[len(orb[:,0]) - 1, 0] #last line, first column
    deathTime.append(lastLineTime)

print('Calculating the population N(t)')
deathTime = np.asarray(deathTime)
for t in np.arange(0,tmax,dt):
    population = 0
    for i in range(len(deathTime)):
        #if the last calculation time > t, then particle i exists at time t
        if deathTime[i] >= t:
            population = population + 1
    fp.write('%f %d\n'%(t,population))

fp.close()
```

codes/Population.py

Appendix K

Mersenne Twister RNG (Source Code)

```
#ifndef __MTWISTER_H
#define __MTWISTER_H

#define STATE_VECTOR_LENGTH 624
#define STATE_VECTOR_M      397
#define UPPER_MASK     0x80000000
#define LOWER_MASK     0x7fffffff
#define TEMPERING_MASK_B  0x9d2c5680
#define TEMPERING_MASK_C  0xefc60000

typedef struct mtRand
{
    unsigned long mt[STATE_VECTOR_LENGTH];
    int index;
} mtRand;

void m_seedRNG(mtRand *rand, unsigned long seed)
{
    rand -> mt[0] = seed & 0xffffffff;
    for(rand->index = 1; rand->index < STATE_VECTOR_LENGTH; rand->index++)
    {
        rand->mt[rand->index] = (6069*rand->mt[rand->index - 1]) & 0xffffffff;
    }
}

mtRand seedRNG(unsigned long seed)
{
    mtRand rand;
    m_seedRNG(&rand,seed);
    return rand;
}

unsigned long genRandLong(mtRand *rand)
{
    unsigned long y;
    static unsigned long mag[2] = {0x0, 0x9908b0df}; /* mag[x] = x * 0x9908b0df for x = 0,1 */
    if(rand->index >= STATE_VECTOR_LENGTH || rand->index < 0)
    {
        /* generate STATE_VECTOR_LENGTH words at a time */
        int kk;
        if(rand->index >= STATE_VECTOR_LENGTH + 1 || rand->index < 0)
        {
            m_seedRNG(rand, 4357);
        }
        for(kk = 0; kk < STATE_VECTOR_LENGTH - STATE_VECTOR_M; kk++)
        {
            y = (rand->mt[kk] & UPPER_MASK) | (rand->mt[kk+1] & LOWER_MASK);
            rand->mt[kk] = rand->mt[kk + STATE_VECTOR_M] ^ (y >> 1) ^ mag[y & 0x1];
        }
        for(; kk < STATE_VECTOR_LENGTH - 1; kk++)
```

```
51          {
52              y = (rand->mt[kk] & UPPER_MASK) | (rand->mt[kk + 1] & LOWER_MASK);
53              rand->mt[kk] = rand->mt[kk + (STATE_VECTOR_M - STATE_VECTOR_LENGTH)] ^ (y >> 1) ^ mag[y & 0x1];
54          }
55          y = (rand->mt[STATE_VECTOR_LENGTH - 1] & UPPER_MASK) | (rand->mt[0] & LOWER_MASK);
56          rand->mt[STATE_VECTOR_LENGTH - 1] = rand->mt[STATE_VECTOR_M - 1] ^ (y >> 1) ^ mag[y & 0x1];
57          rand->index = 0;
58      }
59      y = rand->mt[rand->index++];
60      y ^= (y >> 11);
61      y ^= (y << 7) & TEMPERING_MASK_B;
62      y ^= (y << 15) & TEMPERING_MASK_C;
63      y ^= (y >> 18);
64      return y;
65  }
66
67  //Returns a pseudorandom number in (0,1) from the uniform distro
68  double RNG(mtRand *rand)
69  {
70      return((double)genRandLong(rand) / (unsigned long)0xffffffff);
71  }
72
73  #endif /* #ifndef __MTWISTER_H */
```

codes/mtwister.h

Bibliography

[1] Yang Yu, Patrick Michel, Stephen R. Schwartz, Shantanu P. Naidu, Lance A. M. Benner. *Ejecta Cloud from a Kinetic Impact on the Secondary of a Binary Asteroid: I. Mechanical Environment and Dynamic Model.* Laboratoire Lagrange, Universite Cote d'Azur, Observatoire de la Cote d'Azur, CNRS, 06304 Nice, France, Jet Propulsion Laboratory, California Institute of Technology, 91109 Pasadena, CA, United States. March 24, 2016.

[2] J. Larson[1] and G. Sarid[2]. *ONE BODY, TWO BODY, SMALL BODY, N-BODY: EJECTA DYNAMICS IN THE ENVIRONMENT OF SINGLE AND BINARY ASTEROIDS.* [1]Department of Physics, University of Central Florida, [2]Florida Space Institute, University of Central Florida. 2018.

[3] Patrick Michel et al. *European component of the AIDA mission to a binary asteroid: Characterization and interpretation of the impact of the DART mission.* 12 December 2017.

[4] DR. PATRICK MICHEL (OCA, FRANCE), LEAD. *ASTEROID IMPACT MISSION: DIDYMOS REFERENCE MODEL.* VERSION 10, OCTOBER 22, 2015.

[5] G. J Flynn[1], D. D. Durda[2], E. B. Patmore[3], A. N. Clayton[3], S. J. Jack[3], M. D. Lipman[3] and M. M. Strait[3]. *MOMENTUM TRANSFER IN IMPACT CRATERING OF A POROUS TARGET*, 2014.

[6] www.nasa.gov/planetarydefense/dart

[7] http://dart.jhuapl.edu

[8] www.esa.int/Our_Activities/Space_Safety/Hera

[9] C.D. Murray and S.F. Dermott. *Solar System Dynamics.* 1999.

[10] The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide, Seventh Edition.* 2009.

[11] www.wikipedia.org