ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Σχολή Θετικών Επιστημών
Τμήμα Φυσικής

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μελέτη κινήσεων γύρω από αστεροειδείς ανώμαλου σχήματος - Εφαρμογή στον αστεροειδή Δίδυμο

Νικολαΐδης Συμεών ΑΕΜ: 13755

Επιβλέπων: Βουγιατζής Γεώργιος

Οκτώβριος 2018

Περίληψη

Η ανάγκη μελέτης παραγήινων πλανητικών σωμάτων, όπως οι αστεροειδείς γίνεται ολοένα και πιο επιτακτική. Η πιθανότητα ανεπιθύμητων συγκρούσεων τέτοιων σωμάτων, δεν περνά πλέον απαρατήρητη από την επιστημονική κοινότητα. Σκοπός αυτής τη εργασίας, είναι η μελέτη του βαρυτικού πεδίου που δημιουργεί ένας αστεροειδής με ομοιογενή κατανομή μάζας αλλά ανώμαλο σχήμα. Για να πραγματοποιηθεί αυτή η μελέτη κατασκευάζεται ένα υπολογιστικό μοντέλο του αστεροειδή 65803 Δίδυμος από δεδομένα χαρτογράφησης της $NASA$. Στη συνέχεια, αναλύεται ο τρόπος με τον οποίο το ελκτικό βαρυτικό πεδίο του αστεροειδή επηρεάζει σώμα που βρίσκεται σε προσεγγιστικά κυκλική τροχιά γύρω του, μέσω αριθμητικής ολοκλήρωσης με τη μέθοδο $Bulirsch - Stoer$ των σχετικών διαφορικών εξισώσεων της κίνησης.

The need for study of near earth planetary bodies, such as asteroids is gradually becoming imperative. The chances of undesirable crushes of these objects, are no longer being treated as a trivial matter by the scientific community. The purpose of this thesis, is the study of the gravitational field created by a homogeneous but anomalous asteroid. In order to realize this study, a computational model of the asteroid 65803 Didymos is constructed, based on cartography data provided by NASA. Afterwards, there is an analysis of the way the pulling gravitational field of the asteroid affects a body in an approximately circular orbit, through the Bulirsch - Stoer method of numerical integration of the relevant differential equation of motion.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή κύριο Γεώργιο Βουγιατζή, για όλες τις συμβουλές και τη βοήθεια που παρείχε για την πραγματοποίηση αυτής της εργασίας.

Περιεχόμενα

1 Εισαγωγή **4**
- 1.1 Αστεροειδείς . . . 4
  - 1.1.1 Γενικά . . . 4
  - 1.1.2 Κεπλέρια τροχιακά στοιχεία . . . 5
- 1.2 Βαρυτικές δυνάμεις . . . 8
  - 1.2.1 Γενικά . . . 8
  - 1.2.2 Βαρυτικές δυνάμεις από κατανομές μάζας . . . 9
- 1.3 Ο Δίδυμος . . . 11
  - 1.3.1 Γενικά χαρακτηριστικά . . . 11
  - 1.3.2 Αποστολή DART . . . 12

2 Θεωρητική Εισαγωγή **14**
- 2.1 Μέθοδοι ανάλυσης βαρυτικού πεδίου . . . 14
  - 2.1.1 Γενικά . . . 14
  - 2.1.2 Μέθοδοι για ανώμαλους αστεροειδείς . . . 15
- 2.2 Εξισώσεις σχετικής κίνησης . . . 16
  - 2.2.1 Γενικά . . . 16
  - 2.2.2 Κανονικοποιήσεις . . . 17
  - 2.2.3 Υπολογισμός τροχιάς . . . 20
  - 2.2.4 Μετασχηματισμοί συστημάτων αναφοράς . . . 20
- 2.3 Κύριοι άξονες αδράνειας . . . 22
- 2.4 Μέθοδος αριθμητικής ολοκλήρωσης . . . 25
  - 2.4.1 Βελτιωμένη μέθοδος μέσου σημείου . . . 25
  - 2.4.2 Μέθοδος $Bulirsch - Stoer$ . . . 26

3 Περιγραφή του Αλγορίθμου **27**
- 3.1 Διάγραμμα ροής . . . 27
- 3.2 Interpolation δεδομένων χαρτογράφησης . . . 29
- 3.3 Κατασκευή υπολογιστικού χώρου . . . 32
- 3.4 Διόρθωση συντεταγμένων . . . 36
- 3.5 Υπολογισμός δυναμικής ενέργειας στο επίπεδο $z = 0$ . . . 38
- 3.6 Υπολογισμός τροχιών . . . 39
- 3.7 Γραφική απεικόνιση αστεροειδή . . . 42

4 Αποτελέσματα **44**
4.1 Υπολογιστικός χώρος και διορθώσεις συντεταγμένων 44
4.2 Αποτελέσματα υπολογισμού ενέργειας στο επίπεδο $z = 0$ 46
4.3 Αποτελέσματα υπολογισμού τροχιών 48
4.3.1 Υπολογισμός κυκλικών τροχιών 49
4.3.2 Αλλαγή σχετικού προσανατολισμού 60
4.3.3 Υπολογισμός τροχιάς με βάση τα Κεπλέρια στοιχεία . . 62

5 Συμπεράσματα **65**

Κεφάλαιο 1

Εισαγωγή

1.1 Αστεροειδείς

Με τον όρο αστεροειδείς αναφερόμαστε σε σχετικά μικρά σώματα πλανητικής ύλης που περιφέρονται γύρω από τον Ήλιο. Αν και η ονομασία ελάσσονες πλανήτες θεωρείται πιο ορθή, ιστορικά τα ουράνια αυτά σώματα έχει επικρατήσει να αποκαλούνται αστεροειδείς.

1.1.1 Γενικά

Η πρώτη επίσημη παρατήρηση αστεροειδή τοποθετείται στις αρχές του 19ου αιώνα, με την αναζήτηση τους να έχει ξεκινήσει ήδη από την εποχή που διατυπώθηκε η αριθμητική ακολουθία του νόμου Bode. Η ακολουθία αυτή αντιπροσωπεύει την απόσταση των πλανητών από τον Ήλιο σε αστρονομικές μονάδες AU. Ωστόσο, η αδυναμία εντοπισμού κάποιου πλανήτη στην περιοχή ανάμεσα στον Άρη και το Δία που προέβλεπε ο νόμος του Bode οδήγησε σε όλο και πιο διεξοδικές αναζητήσεις μέχρι το 1801 όταν και ο Giuseppe Piazzi ανακάλυψε τυχαία έναν "πλανήτη" στην περιοχή ενδιαφέροντος , που ονομάστηκε Δήμητρα. Την ανακάλυψη αυτήν ακολούθησε μια πληθώρα καταγραφών σωμάτων που περιφέρονταν γύρω από τον Ήλιο και μεταξύ των πλανητών Άρη και Δία σχηματίζοντας μια ζώνη μεταξύ των 2.0 και 4.2 AU. Τα μη αξιόπιστα τηλεσκόπια της εποχής οδήγησαν στον ατυχή χαρακτηρισμό αυτών των σωμάτων με τον όρο αστεροειδείς, καθώς έμοιαζαν με αστέρια.

Αστεροειδείς συναντάμε γύρω από τα δύο σημεία ισορροπίας Lagrange του συστήματος Ηλίου – Δία, οι οποίοι αποκαλούνται Τρωϊκοί. Επιπλέον, μια περιοχή που συγκεντρώνει ένα μεγάλο ποσοστό των αστεροειδών του ηλιακού μας συστήματος είναι η ζώνη Kuiper και βρίσκεται πέραν της τροχιάς του Ποσειδώνα.

Από τους αστεροειδείς του ηλιακού συστήματος μόλις το 5% παρουσιάζει τροχιές με τέτοια εκκεντρότητα ώστε να τέμνουν τις τροχιές ενός ή και περισσότερων πλανητών όπως του Άρη, του Δία, του Κρόνου, του Ουρανού αλλά και της Γης. Για το λόγο αυτό η μελέτη της κίνησης των αστεροειδών κρίνεται μεγάλου ενδιαφέροντος και σημασίας εν όψει πιθανής τους σύγκρουσης με τη Γη ή των κοντινών πλανητών. Αξίζει βέβαια να σημειωθεί πως ένα τέτοιο γεγονός έχει πολύ μικρές πιθανότητες. Όσοι μάλιστα αστεροειδείς πλησιάζουν τη Γη χαρακτηρίζονται και ως παραγήινοι αστεροειδείς, όπως ο 433 Έρως, ο Δίδυμος, ο Άδωνις κ.α.

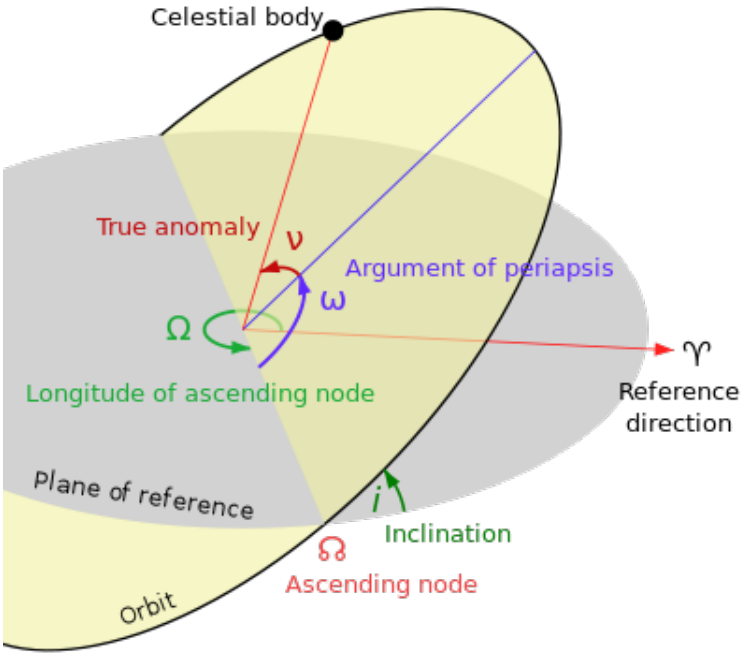
Οι περισσότεροι αστεροειδείς μοιάζουν με μικρούς βράχους με την πλειονότητα τους να έχει ανώμαλο σχήμα ή να είναι επιμήκη σώματα όπως οι δορυφόροι του Άρη, Φόβος και Δείμος. Παρουσιάζουν κρατήρες και σχισμές στην επιφάνεια τους που προκλήθηκαν από συγκρούσεις με άλλα μικρότερα μεσοπλανητικά υπολείμματα.

Τέλος, όσον αφορά την δημιουργία τους η επικρατέστερη θεωρία σήμερα επισημαίνει πως οι αστεροειδείς, της κύριας ζώνης, πιθανότατα δημιουργήθηκαν από κάποιο τμήμα του ίδιου νέφους που δημιούργησε το πλανητικό μας σύστημα. Ωστόσο, εξαιτίας παρέλξεων του Δία, που ήταν από τα πρώτα σώματα που είχαν σχηματιστεί στο ηλιακό σύστημα, δεν συνεχίστηκε η διαδικασία προσαύξησης μάζας προς τον σχηματισμό κάποιου πλανήτη. Αντίθετα, όλη η πλανητική μάζα παρέμεινε διάσπαρτη στο χώρο. Η μικρή πυκνότητα των αστεροειδών, περίπου $1gr/cm^3$ αλλά και η έλλειψη ομοιογένειας τους φαίνεται να συνηγορούν σε ένα τέτοιο σενάριο.

1.1.2 Κεπλέρια τροχιακά στοιχεία

Προκειμένου να προσδιοριστεί μια συγκεκριμένη τροχιά ενός ουράνιου σώματος είναι απαραίτητη η γνώση μιας σειράς παραμέτρων, οι οποίες είναι γνωστές ως Κεπλέρια τροχιακά στοιχεία[1].

Στην ουράνια μηχανική τα στοιχεία αυτά θεωρούνται γενικά στο κλασσικό σύστημα των δύο σωμάτων, όπου χρησιμοποιείται κάποια Κεπλέρια τροχιά. Μια πραγματική τροχιά (μαζί με τα στοιχεία της) μεταβάλλεται συνεχώς λόγω των διάφορων βαρυτικών διαταραχών που προκαλούν άλλα σώματα, αλλά και λόγω επιπτώσεων της σχετικότητας. Η Κεπλέρια τροχιά αποτελεί την μαθηματική έκφραση μιας εξιδανικευμένης τροχιάς μια δεδομένη χρονική στιγμή.



Σχήμα 1.1: Κεπλέρια τροχιακά στοιχεία.

Τα παραδοσιακά τροχιακά στοιχεία είναι τα έξι Κεπλέρια στοιχεία, που φαίνονται στο Σχήμα 1 και περιγράφονται συνοπτικά παρακάτω.

Σε ένα αδρανειακό σύστημα αναφοράς, δύο σώματα σε τροχιά ακολουθούν διακριτές πορείες. Κάθε μια από αυτές, έχει την εστίαση της στο κοινό κέντρο μάζας. Από ένα μη αδρανειακό σύστημα του ενός σώματος ωστόσο, μπορεί να παρατηρηθεί μόνο η τροχιά του άλλου σώματος. Τα Κεπλέρια τροχιακά στοιχεία επιτρέπουν την περιγραφή αυτών των μη αδρανειακών τροχιών. Μια τροχιά έχει δύο σετ Κεπλεριανών στοιχείων ανάλογα με το ποιο σώμα χρησιμοποιείται σαν σημείο αναφοράς.

Το σώμα αναφοράς αποκαλείται πρωτεύον, και το άλλο σώμα δευτερεύον, χωρίς να υπάρχει κάποιο κριτήριο για το πως αυτά θα επιλεγούν.

Τα δύο βασικά στοιχεία που καθορίζουν το σχήμα της έλλειψης είναι:

• **Η εκκεντρότητα e :** Αποτελεί ένα δείκτη για τη μορφή της έλλειψης, περιγράφοντας πόσο επιμηκυμένη είναι σχετικά με έναν κύκλο.

• **Ο κύριος ημιάξονας α :** Πρόκειται για το ήμι-άθροισμα των αποστάσεων περίαψης και απόαψης. Στην περίπτωση κυκλικών τροχιών, είναι η απόσταση μεταξύ των κέντρων των σωμάτων.

Τα δύο στοιχεία που ορίζουν τον προσανατολισμό του τροχιακού επιπέδου στο οποίο βρίσκεται η έλλειψη:

• **Η κλίση i :** Είναι η κλίση του επιπέδου της έλλειψης σε σχέση με το επίπεδο αναφοράς , μετρούμενη στο ανερχόμενο σημείο, δηλαδή εκεί όπου η τροχιά αρχίζει να ανεβαίνει μέσα από το επίπεδο αναφοράς.

• **Το μήκος του ανερχόμενου σημείου Ω :** Δίνει τον οριζόντιο προσανατολισμό στο ανερχόμενο σημείο (κόμβο) σε σχέση με την εαρινή ισημερία του συστήματος αναφοράς.

Επιπλέον έχουμε τα στοιχεία:

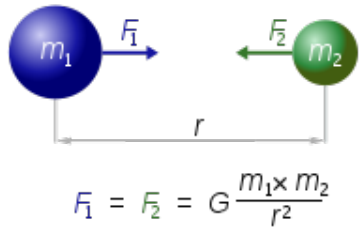
• **Το όρισμα του περικέντρου ω :** Ορίζει τον προσανατολισμό της έλλειψης στο επίπεδο της τροχιάς, και μετριέται ως γωνία ξεκινώντας από το ανερχόμενο σημείο μέχρι την αψίδα προς το περίκεντρο (δηλαδή το κοντινότερο σημείο που το δευτερεύον σώμα πλησιάζει στο πρωτεύον.

• **Η μέση ανωμαλία κατά την εποχή ν :** Πρόκειται για μια γωνία που ορίζει τη θέση του περιφερόμενου σώματος κατά μήκος της έλλειψης σε κάποια χρονική στιγμή (εποχή).

1.2 Βαρυτικές δυνάμεις

Σύμφωνα με τον νόμο της παγκόσμιας έλξης που διατυπώθηκε από τον Νεύτωνα, όταν δύο σώματα με μάζες m_1 και m_2 βρίσκονται σε απόσταση r ασκείται μεταξύ τους ελκτική δύναμη. Η δύναμη αυτή ονομάζεται βαρυτική και μαθηματικά δίνεται από τη διανυσματική σχέση:

$$\vec{F_{21}} = -G\frac{m_1 m_2}{|\vec{r_{12}}|^2}\hat{r_{12}} \tag{1.1}$$



Σχήμα 1.2: Βαρυτική αλληλεπίδραση δύο σωμάτων.

όπου $\vec{F_{21}}$ είναι η δύναμη που ασκεί το σώμα 1 στο σώμα 2 , $G = 6.674 \times 10^{-11} Nkg^{-2}m^2$ η παγκόσμια σταθερά της βαρύτητας σε μονάδες SI και $|\vec{r_{12}}|$ η απόσταση μεταξύ των μαζών m_1 και m_2. Είναι προφανές ότι $\vec{F_{21}} = -\vec{F_{12}}$.

1.2.1 Γενικά

Επειδή η βαρυτική δύναμη F έχει ευθεία δράσης την ευθεία που ενώνει τις δύο μάζες και το μέτρο της εξαρτάται από την απόσταση τους r, λέγεται κεντρική δύναμη[2]. Μαθηματικά οι κεντρικές δυνάμεις εκφράζονται ως:

$$\vec{F} = F(r)\hat{r} \tag{1.2}$$

Οι κεντρικές δυνάμεις είναι συντηρητικές, δηλαδή το έργο W που παράγεται από αυτές κατά την μετατόπιση ενός σώματος από θέση A σε θέση B είναι ανεξάρτητο της διαδρομής και εξαρτάται αποκλειστικά από την αρχική και τελική θέση.

Ένα σώμα που βρίσκεται σε πεδίο βαρυτικών δυνάμεων αποκτά δυναμική ενέργεια η οποία γενικά δίνεται από τη σχέση:

$$U_B - U_A = W_{(B\to A)} = -\int_A^B \vec{F} \bullet d\vec{r} \tag{1.3}$$

η οποία με κατάλληλη εκλογής της στάθμης μηδενικής ενέργειας $U_A = U_\infty = 0$ γράφεται για την περίπτωση των κεντρικών δυνάμεων στη μορφή:

$$U = -\int_\infty^r F(r)dr \tag{1.4}$$

Γίνεται λοιπόν, προφανές ότι το μέτρο μιας κεντρικής δύναμης $F(r)$ μπορεί να εκφραστεί ως η παράγωγος μιας χρονικά ανεξάρτητης συνάρτησης δυναμικής ενέργειας $U(r)$:

$$F = -\frac{dU}{dr} \tag{1.5}$$

Τέλος, αναφέρουμε τη βασική ενεργειακή εξίσωση που περιγράφει πεδία συντηρητικών δυνάμεων, σύμφωνα με την οποία το άθροισμα κινητικής και δυναμικής ενέργειας διατηρείται:

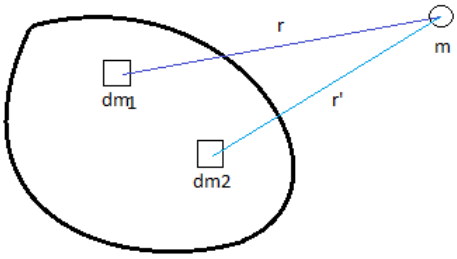
$$E = \frac{1}{2}m|\dot{\vec{r}}|^2 + U(r) = \sigma\tau\alpha\theta \tag{1.6}$$

όπου $\dot{\vec{r}}$ είναι η πρώτη παράγωγος του διανύσματος θέσης ως προς το χρόνο και m η μάζα του σώματος.

1.2.2 Βαρυτικές δυνάμεις από κατανομές μάζας

Συχνά κατά την μελέτη της αλληλεπίδρασης δύο σωμάτων, τουλάχιστον ένα από αυτά αντιμετωπίζεται όχι ως υλικό σημείο αλλά ως κατανομή μάζας (Σχήμα 1.3). Θεωρούμε ότι το σύστημα που μελετάμε έχει συνολική μάζα M_{total}. Έστω ότι η κατανομή είναι ομογενής, δηλαδή ότι το σώμα μάζας M (αστεροειδής) αποτελείται από N στοιχειώδεις μάζες dm με $dm = M_{total}/N$. Τότε κάθε μία από αυτές τις στοιχειώδεις μάζες θα αλληλεπιδρά με το σώμα μάζας m (δορυφόρος) έλκοντας το με μια βαρυτική δύναμη μέτρου F_i. Με την ίδια λογική, κάθε στοιχειώδης μάζα συνεισφέρει δυναμικό U_i. Σύμφωνα με την αρχή της επαλληλίας μπορούμε να αθροίσουμε τα επιμέρους U_i στη θέση του σώματος m. Δηλαδή θα είναι:

$$U = -\sum_{i=1}^{N} \frac{Gdm_i}{r_i} \tag{1.7}$$



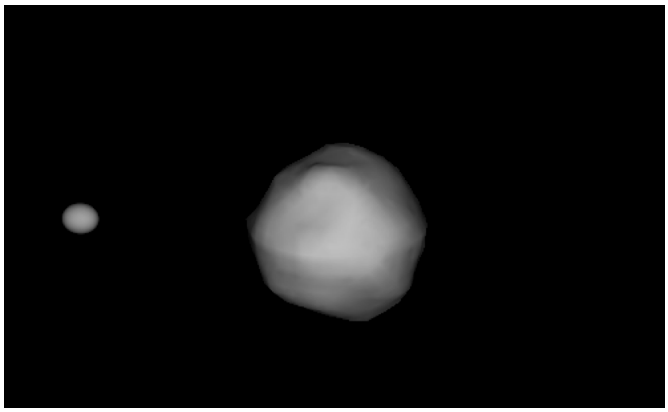
Σχήμα 1.3: Επιμέρους επιδράσεις των στοιχειωδών μαζών πάνω στο σώμα m.

όπου dm_i η εκάστοτε στοιχειώδης μάζας και r_i η απόσταση της κάθε στοιχειώδης μάζας από το σώμα m και G η σταθερά της παγκόσμιας έλξης με τιμή $6.674 \times 10^{-11} m^3 kg^{-1} s^{-2}$.

1.3 Ο Δίδυμος

Ο 65803 Δίδυμος είναι ένας παραγήινος αστεροειδής με διαστάσεις μικρότερες του χιλιομέτρου, ο οποίος μαζί με τον μικρότερο σε διαστάσεις δορυφόρο του γνωστό ως *Didymoon* αποτελεί ένα διπλό σύστημα αστεροειδών[3]. Η ανακάλυψη του έγινε στις 11 Απριλίου 1996, από το παρατηρητήριο *Steward* του πανεπιστημίου της Αριζόνα και έχει κατηγοριοποιηθεί ως ένα πιθανώς επικίνδυνο για σύγκρουση με την Γη σώμα. Ανήκει στα σύνολα αστεροειδών *Apollo* και *Amor*.

1.3.1 Γενικά χαρακτηριστικά



Σχήμα 1.4: Προσομοίωση του του διπλού συστήματος του αστεροειδή Δίδυμος. Η διάμετρος του φεγγαριού του υπολογίζεται στα $150m$. Τα δύο σώματα βρίσκονται σε απόσταση περίπου ενός χιλιομέτρου, ενώ η περίοδος περιφοράς του φεγγαριού είναι 11.9 ώρες.

Ο Δίδυμος βρίσκεται σε τροχιά γύρω από τον Ήλιο σε απόσταση που κυμαίνεται από 1 εως 2.3 *AU* με περίοδο 770 ημέρες. Η τροχιά του παρουσιάζει εκκεντρότητα 0.38 και κλίση 3° ως προς την εκλειπτική. Τον Νοέμβριο του 2003 πλησίασε την Γη σε απόσταση $7.18*10^6 km$, ενώ υπολογίζεται ότι η επόμενη φορά που θα την προσεγγίσει τόσο κοντά θα είναι περί το Νοέμβριο του 2123.

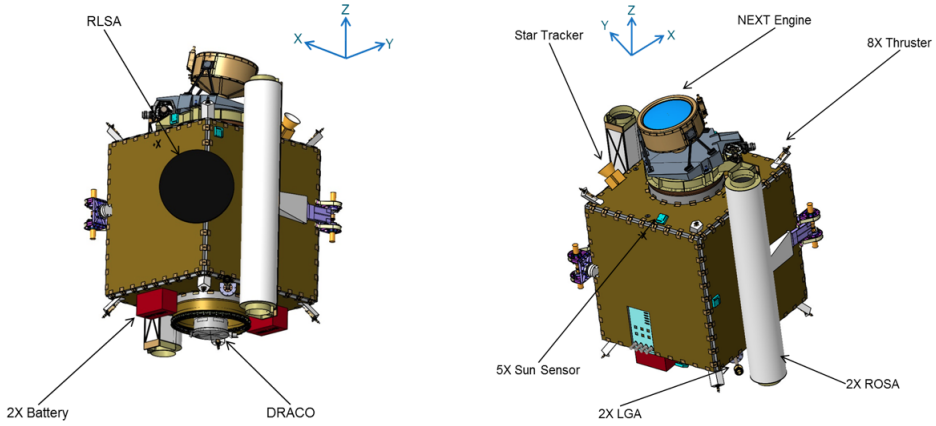
Κατά την κατηγοριοποίηση *SMASS*, ο Δίδυμος ανήκει στην ομάδα *Xk* αστεροειδών. Περιστρέφεται γρήγορα, με περίοδο 2.26 ωρών και παρουσιάζει διακύμανση φωτεινότητας μεγέθους 0.08, γεγονός που μαρτυρά τη σφαιροειδή μορφή του.

Φυσικές παράμετροι Δίδυμου			
Παράμετρος	Σύμβολο	Τιμή	Μονάδες
Απολυτο μέγεθος	H	18 ±0.03	mag
Διάμετρος	$diameter$	0.78 ±0.08	km
Πυκνότητα	ρ	2.1 ±0.6	g/cm^3
Περίοδος περιστροφής	T	2.2593 ±0.0002	h

Πίνακας 1.1: Φυσικές παράμετροι του αστεροειδή 65803 Δίδυμος

1.3.2 Αποστολή DART

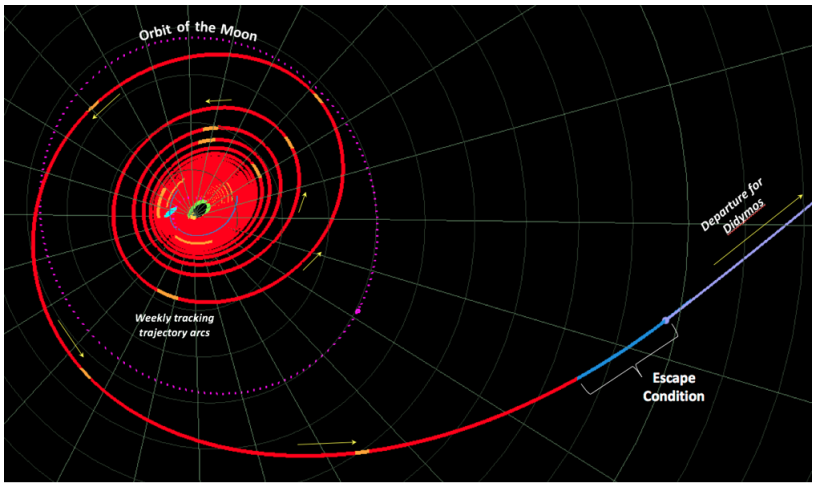
Ο Δίδυμος αποτελεί τον στόχο μιας μη επανδρωμένης διαστημικής αποστολής, υπό την ονομασία *AIDA* (Asteroid Impact and Deflection mission). Πρόκειται για μια συνεργασία μεταξύ των διαστημικών οργανισμών της Ευρώπης και Αμερικής (*ESA* και *NASA*). Σκοπός της είναι η διεκπερέωση μιας ολοκληρωμένης μελέτης για το αν μέσω μιας πρόσκρουσης ενός τεχνητού σώματος πάνω σε αστεροειδή, μπορεί να αλλιωθεί επαρκώς η τροχιά του, που υπο κανονικές συνθήκες θα ήταν επιζήμια την Γη και τις ανθρώπινες δραστηριότητες.



Σχήμα 1.5: Το διαστημικό όχημα *Dart* από δύο διαφορετικές οπτικές γωνίες.

Την αποστολή αυτή καλείται να φέρει εις πέρας το διαστημικό όχημα *DART*, υπο την επίβλεψη της *NASA* [4]. Το *DART* εξοπλισμένο με κατάλληλους προωθητήρες που θα του εξασφαλίσουν την ταχύτητα των $6km/s$, θα εμβολίσει το φεγγάρι του Δίδυμου. Η σύγκρουση αυτή αναμένεται να μεταβάλλει την ταχύτητα του *Didymoon* κατά ένα κλάσμα του ενός τοις εκατό, αρκετό ώστε να παρατηρηθεί από επίγεια τηλεσκόπεια.

Αξίζει να σημειωθεί ότι, το υπερσύγχρονο σύστημα προώθησης με το οποίο έχει εφοδιαστεί το *DART*, με την ονομασία NASA Evolutionary Xenon Thruster - Commercial , NEXT - C προσφέρει υψηλή ευελιξία στα χρονικά και οικονομικά όρια ενός τέτοιου εγχειρήματος λόγω της υψηλής αποδοτικότητας του.



Σχήμα 1.6: Η τροχιά της αποστολής του διαστημικού οχήματος *DART* .

Η αποστολή *DART* αναμένεται να πραγματοποιηθεί τον Δεκέμβρη του 2020 με ημερομηνία πρόσκρουσης τον Οκτώβριο του 2022.

Κεφάλαιο 2

Θεωρητική Εισαγωγή

2.1 Μέθοδοι ανάλυσης βαρυτικού πεδίου

2.1.1 Γενικά

Η μελέτη των τροχιών πλανητικών σωμάτων, βασίζεται εν γένει στον υπολογισμό των βαρυτικών πεδίων που αυτά δημιουργούν καθώς και στον τρόπο που αλληλεπιδρούν με άλλα πεδία δυνάμεων[5]. Ο υπολογισμός ενός τέτοιου βαρυτικού πεδίου παρουσιάζει δυσκολίες, οι οποίες μάλιστα εξαρτώνται σε μεγάλο βαθμό από το είδος της συμμετρίας που μπορεί να εμφανίζει το υπο μελέτη σώμα.

Τα πλανητικά σώματα παρουσιάζουν συχνά υψηλή σφαιρικότητα. Αυτός είναι και ο λόγος που μέθοδοι ανάλυσης που εκμεταλλεύονται τέτοιες συμμετρίες όπως οι σφαιρικές αρμονικές επιλέγονται για την περιγραφή του πεδίου. Σε τέτοιες περιπτώσεις, οι σφαιρικές αρμονικές παρουσιάζουν μεγάλη σύγκλιση γύρω από το σώμα και δίνουν μια ιδιαίτερα ακριβή εικόνα του βαρυτικού πεδίου ακόμα και πολύ κοντά στην επιφάνεια του.

Κατα ανάλογο τρόπο έχει χρησιμοποιηθεί και η προσέγγιση του *Ivory*, σε περιπτώσεις που απαιτείται η αναλυτική έκφραση της βαρυτικής έλξης ενός ομοιογενούς τριαξονικού ελλειψοειδούς.

Ωστόσο, στην πέριπτωση που εξετάζονται αστεροειδείς σαν τον Δίδυμο, οι δυσκολίες αυξάνονται. Ο Δίδυμος αν και παρουσιάζει σχετική σφαιρικότητα, παραμένει ένα ακανόνιστο σώμα του οποίου η μορφή δεν μπορεί να διατυπωθεί από τυπικές μαθηματικές εκφράσεις.

2.1.2 Μέθοδοι για ανώμαλους αστεροειδείς

Μια προσέγγιση που βρήκε μέγαλη εφαρμογή στο παραπάνω πρόβλημα, είναι αυτή που προτάθηκε το 1996 απο τους *Scheeres* και *Werner*. Η αναλυτική λύση τους, είναι ένας ακριβής υπολογισμός του βαρυτικού πεδίου που δημιουργεί ένα τυχαίο πολύεδρο, το οποίο όμως αποτελείται από τριγωνικές στοιχειώδεις επιφάνειες.

Ένα τέτοιο μοντέλο, αποδείχθηκε ότι παρουσιάζει γραμμική πολυπλοκότητα, που εξαρτάται από το πλήθος των τριγωνικών επιφανειών και των ακμών που αυτές ορίζουν στο πολύεδρο. Ειδικά στην περίπτωση πολυέδρου που αυτές οι επιφάνειες δεν τέμνονται μεταξύ τους, η γνώση απλώς του πλήθους των κορυφών του σώματος είναι αρκετή για τον υπολογισμό της γραμμικής πολυπλοκότητας του προβλήματος.

Για τις περιπτώσεις αστεροειδών ανώμαλου σχήματος έχει επίσης εφαρμοστεί η μέθοδος μοντελοποίησης του σώματος από έναν αριθμό, συνήθως μεγάλο, ομοιογενών σφαιρών. Αυτό το μοντέλο που είναι γνωστό στην βιβλιογραφία ως "*mascons*", προσφέρει μια πολύ απλή έκφραση για το βαρυτικό δυναμικό και την επιτάχυνση.

Στην παρούσα εργασία, η μοντελοποίηση του Δίδυμου θα γίνει με μια σειρά από ομοιογενείς κυβικές κυψελίδες, οι οποίες όπως οι σφαίρες του μοντέλου "*mascons*", ασκούν η κάθεμια μια έλξη πάνω σε σώμα που βρίσκεται κοντά στον αστεροειδή και καθορίζουν την κίνηση του.

2.2 Εξισώσεις σχετικής κίνησης

2.2.1 Γενικά

Η εξίσωση που περιγράφει την κίνηση ενός σωματίου γύρω από από έναν περιστρεφόμενο αστεροειδή, είναι μια συνήθης διαφορική εξίσωση δεύτερης τάξης:

$$\ddot{\vec{r}} + 2\vec{\omega} \times \dot{\vec{r}} + \vec{\omega} \times (\vec{\omega} \times \vec{r}) + \dot{\vec{\omega}} \times \vec{r} + \frac{\partial U(\vec{r})}{\partial \vec{r}} = 0 \tag{2.1}$$

όπου $\vec{r}$ είναι το διάνυσμα θέσης του σώματος σε σύστημα συντεταγμένων με αρχή το κέντρο μάζας του αστεροειδή, $\dot{\vec{r}}$ και $\ddot{\vec{r}}$ η πρώτη και η δεύτερη παράγωγος του διανύσματος $\vec{r}$ ως προς τον χρόνο στο αδρανειακό σύστημα, $\vec{\omega}$ είναι το διάνυσμα της γωνιακής ταχύτητας περιστροφής του αστεροειδή στο αδρανειακό σύστημα και $U(\vec{r})$ είναι το βαρυτικό δυναμικό του αστεροειδή[6].

Η εξίσωση 2.1 αποτελεί την κλασσική μορφή της εξίσωσης κίνησης ενός σωματίου στον τρισδιάστατο Ευκλείδιο χώρο. Η μηχανική ενέργεια Ε του σώματος θα δίνεται από την σχέση:

$$E = \frac{1}{2}\vec{v_I} \cdot \vec{v_I} + U(\vec{r}) \tag{2.2}$$

όπου $\vec{v_I} = \dot{\vec{r}} + \vec{\omega} \times \vec{r}$ είναι ταχύτητα του σώματος στο αδρανειακό σύστημα. Η κινητική ενέργεια του σώματος θα δίνεται ως:

$$T = \frac{1}{2}(\dot{\vec{r}} + \vec{\omega} \times \vec{r}) \cdot (\dot{\vec{r}} + \vec{\omega} \times \vec{r}) \tag{2.3}$$

και το ενεργό δυναμικό θα είναι:

$$V(\vec{r}) = -\frac{1}{2}(\vec{\omega} \times \vec{r}) \cdot (\vec{\omega} \times \vec{r}) + U(\vec{r}) \tag{2.4}$$

Αντικαθιστώντας την 2.4 στην 2.1 γράφουμε:

$$\ddot{\vec{r}} + 2\vec{\omega} \times \dot{\vec{r}} + \dot{\vec{\omega}} \times \vec{r} + \frac{\partial V(\vec{r})}{\partial \vec{r}} = 0 \tag{2.5}$$

ενώ το σχετικό ολοκλήρωμα της ενέργειας γράφεται ως εξής:

$$H = \frac{1}{2}\dot{\vec{r}} \cdot \dot{\vec{r}} + V(\vec{r}) \tag{2.6}$$

2.2.2 Κανονικοποιήσεις

Στο πρόβλημα του Δίδυμου που εξετάζουμε, θεωρούμε αφενός ότι το κέντρο μάζας του αστεροειδή που μελετάμε ταυτίζεται με την αρχή των συντεταγμένων και οτι η περιστροφή του γίνεται πέρι τον άξονα z, με φορά περιστροφής αυτή των δεικτών του ρολογιού και σταθερή γωνιακή ταχύτητα μέτρου ω. Επιπλέον, θα εφαρμόσουμε μια σειρά από κανονικοποιήσεις όσον αφορά την μέτρηση μήκους και αυτή του χρόνου. Συγκεκριμένα, θεωρούμε τους παρακάτω μετασχηματισμούς:

$$l' = \frac{l}{a} = \frac{l}{400m} \tag{2.7}$$

όπου ως το a ισούται με το μισό της διαμέτρου του Δίδυμου δηλαδή με $400m$. Σύμφωνα λοιπόν με αυτόν τον μετασχηματισμό, μια μονάδα μήκους στο σύστημα συντεταγμένων που θα χρησιμοποιήσουμε ισοδυναμεί με $400m$.

Επιπρόσθετα θεωρούμε:

$$\tau = \omega t = \frac{2\pi t}{T} \tag{2.8}$$

όπου $T = 2.26h = 8 \times 10^3 sec$ είναι η περίοδος περιστροφής του Δίδυμου. Επομένως, για το σύστημα μας μια μονάδα χρόνου ισοδυναμεί με τον λόγο $2.26h/2\pi = 0.36h = 1287.3sec$.

Η εκλογή των παραπάνω παραμέτρων μπορεί να είναι αυθαίρετη και εξαρτάται από τα φυσικά χαρακτηριστικά του προβλήματος που εξετάζουμε.

Αν τώρα σε ένα σημείο του χώρου $\Sigma(x, y, z)$ θεωρήσουμε ότι βρίσκεται υλικό σημείο μάζας m , αμελητέας σε σχέση με τη μάζα του κυρίως σώματος - αστεροειδή που συμβολίζουμε με M, ώστε να θεωρούμε $M_{total} = M$ για τη συγκεκριμένη περίπτωση. Τότε σύμφωνα με την **1.7** στο σημείο **Σ** γράφουμε για το δυναμικό:

$$U = -\sum_{i=1}^{N} \frac{Gdm_i}{r_i} = GM_{total}(-\sum_{i=1}^{N} \frac{\frac{dm_i}{M_{total}}}{r_i}) = GMU'(r_i') = \mu U' \tag{2.9}$$

όπου με U' συμβολίζουμε το δυναμικό μετά τις κανονικοποιήσεις των **2.7** και **2.8**.

Στη συνέχεια θα προχωρήσουμε στην ανάλυση της διανυσματικής εξίσωσης της κίνησης **2.1** σε κάθε διάσταση υπολογίζοντας την εκάστοτε παράγωγο του δυναμικού με βάση την **2.9**. Έτσι για τον άξονα x έχουμε:

$$\ddot{x} = 2\omega\dot{y} + x\omega^2 - \frac{GM}{a}\frac{\partial U'}{\partial x} = 2\omega\dot{y} + x\omega^2 - \frac{GM}{a^2}\frac{\partial U'}{\partial x'} \tag{2.10}$$

όμως επειδή:

$$\ddot{x} = \frac{d^2x}{dt^2} = \frac{d^2(ax')}{d(\tau/\omega)^2} = a\omega^2\frac{d^2x'}{d\tau^2} \tag{2.11}$$

και

$$\dot{x} = \frac{dx}{dt} = a\omega\frac{dx'}{d\tau} \tag{2.12}$$

αντικαθιστώντας τις **2.11** και **2.12** στην **2.10** παίρνουμε:

$$\frac{d^2x'}{d\tau^2} = 2\frac{dy'}{d\tau} + x' - \frac{GM}{a^3\omega^2}\frac{\partial U'}{\partial x'} \tag{2.13}$$

την οποία γράφουμε πιο απλά ως:

$$\ddot{x} = 2\dot{y} + x - \frac{GM}{a^3\omega^2}\frac{\partial V'}{\partial x} \tag{2.14}$$

όπου τα αντίστοιχα μεγέθη είναι πλέον κανονικοποιημένα. Δουλεύοντας με την ίδια μεθοδολογία για y, z βρίσκουμε τις αντίστοιχες διαφορικές εξισώσεις.

$$\ddot{y} = -2\dot{x} + y - \frac{GM}{a^3\omega^2}\frac{\partial V'}{\partial y} \tag{2.15}$$

$$\ddot{z} = -\frac{GM}{a^3\omega^2}\frac{\partial V'}{\partial z} \tag{2.16}$$

Συμφωνα με τις εξισώσεις **2.14** εως **2.16** μπορούμε να διατυπώσουμε πλεον την εξίσωση που δίνει την βαρυτική δυναμική ενέργεια U, μετά την εφαρμογή όλων των μετασχηματισμών στο συστημα μας:

$$U = -\frac{1}{2}(x^2 + y^2) + \delta V'(x, y, z) \tag{2.17}$$

Στην σχέση **2.17** αξίζει αφενός να σημειωθεί ότι λόγω περιστροφής γύρω από τον άξονα z η δυναμική ενέργεια είναι ανεξάρτητη του z και εξαρτάται μόνο από την θέση του σώματος σχετικά με τον αστεροειδή. Αφετέρου, ο όρος του κανονικοποιημένου δυναμικού $V'(x, y, z)$ και υπολογίστηκε σύμφωνα με τη σχέση:

$$V' = -\sum_{i=1}^{N} \frac{dm_i}{r_i} \tag{2.18}$$

όπου $dm_i = M_{total}/N$, ώστε το συνολικό τους άθροισμα να είναι μονάδα. Όσον αφορά την αδιάστατη σταθερά δ της **2.17**, θα ισχύει:

$$\delta = \frac{GM}{\omega^2 a^3} = \frac{\mu}{\omega^2 a^3} \approx 0.845 \tag{2.19}$$

όπου $M = 5 \times 10^{11} kg$ η μάζα του **Δίδυμου**. Η σταθερά αυτή διευκολύνει την επίλυση του προβλήματος χωρίς να αλλάζει την φύση του, ενώ εφαρμόζει ουσιαστηκά μια κλίμακα στα μετρούμενα μεγέθη. Σύμφωνα με τον *Scheeres* [7], το δ αντιπροσωπεύει την αναλογία της βαρυτικής επιτάχυνσης ως προς την κεντρομόλο επιτάχυνση που ασκείται σε κάποιο σώμα στο πιο απομακρυσμένο σημείο απο το κέντρο του αστεροειδή, με την προυπόθεση ότι ο αστεροειδής έχει συγκεντρωμένη όλη τη μάζα στο κέντρο του.

2.2.3 Υπολογισμός τροχιάς

Αντικαθιστώντας πλέον και την τιμή της σταθεράς δ από την 2.19 μπορούμε πλεον να αναδιατυπώσουμε τις διαφορικές εξισώσεις (2.14 εως 2.16) της κίνησης ενός σώματος - δορυφόρου που θα κινείται υπο την επίδραση της βαρυτικής έλξης του περιστρεφόμενου περι τον άξονα z Δίδυμου, ως εξής :

$$\ddot{x} = 2\dot{y} + x - \delta\frac{\partial V'}{\partial x} \tag{2.20}$$

$$\ddot{y} = -2\dot{x} + y - \delta\frac{\partial V'}{\partial y} \tag{2.21}$$

$$\ddot{z} = -\delta\frac{\partial V'}{\partial z} \tag{2.22}$$

2.2.4 Μετασχηματισμοί συστημάτων αναφοράς

Για τους διάφορους υπολογισμούς όπως για τα Κεπλέρια τροχιακά στοιχεία μιας τροχιάς γύρω από τον Δίδυμο θα χρειαστούν μετατροπές του συστήματος αναφοράς από περιστρεφόμενο σε αδρανειακό (και αντίστροφα).

Για να επιτευχθούν αυτοί οι μετασχηματισμοί θεωρούμε αρχικά τις μεταβλητές θέσης και ταχύτητας στο αδρανειακό σύστημα αναφοράς $(X, \Upsilon, Z, V_x, V_y, V_z)$. Επιπλέον, έστω ότι τα αντίστοιχα μεγέθη στο μη αδρανειακό σύστημα αναφοράς, συμβολίζονται με τις μεταβλητές (x, y, z, u_x, u_y, u_z). Τότε με πίνακα περιστροφής τον R_z γύρω από τον άξονα z

$$R_z = \begin{pmatrix} cos\omega t & -sin\omega t & 0 \\ sin\omega t & cos\omega t & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.23}$$

έχουμε διαδοχικα:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = R_z \begin{pmatrix} x \\ y \\ z \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \end{pmatrix} = R_z^{-1} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{2.24}$$

$$\begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} = R_z \begin{pmatrix} u_x - \omega y \\ u_y - \omega x \\ u_z \end{pmatrix} \rightarrow \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} = R_z^{-1} \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} + \omega \begin{pmatrix} y \\ -x \\ 0 \end{pmatrix} \tag{2.25}$$

δηλαδή οι ζητούμενοι μετασχηματισμοί από περιστρεφόμενο σε αδρανειακό θα είναι:

$$\begin{aligned} X &= xcos\omega t - ysin\omega t \\ Y &= xsin\omega t + ycos\omega t \\ Z &= z \\ V_x &= (u_x - y\omega)cos\omega t - (u_y + x\omega)sin\omega t \\ V_y &= (u_y - y\omega)sin\omega t + (u_y + x\omega)cos\omega t \\ V_z &= z \end{aligned} \tag{2.26}$$

ενώ ακολουθώντας την αντίστροφη πορεία, βρίσκουμε τους αντίστοιχους μετασχηματισμούς από το αδρανειακό στο περιστρεφόμενο:

$$\begin{aligned} x &= Xcos\omega t + Ysin\omega t \\ y &= -Xsin\omega t + Ycos\omega t \\ z &= Z \\ u_x &= V_xcos\omega t + V_ysin\omega t + \omega y \\ u_y &= -V_xsin\omega t + V_ycos\omega t - \omega x \\ u_z &= V_z \end{aligned} \tag{2.27}$$

2.3 Κύριοι άξονες αδράνειας

Οι κύριοι άξονες αδράνειας είναι διευθύνσεις τέτοιες ώστε καθώς το στερεό (ο Δίδυμος, στο πρόβλημα μας) περιστρέφεται γύρω από έναν άξονα που διέρχεται από το κέντρο μάζας του (ή ένα οποιοδήποτε ακίνητο σημείο O) και είναι παράλληλος προς αυτή τη διεύθυνση, η αντίστοιχη στροφορμή να είναι επίσης παράλληλη προς το διάνυσμα της γωνιακής ταχύτητας.

Έστω $\vec{\omega_0} = (\omega_0 1, \omega_0 2, \omega_0 3)$ **ένα διάνυσμα γωνιακής ταχύτητας που είναι παράλληλο προς μια τέτοια διεύθυνση που περιγράψαμε νωρίτερα, τότε:**

$$\vec{L} = \lambda \vec{\omega_0} \tag{2.28}$$

όπου $\vec{L}$ **η στροφορμή ως πρός το κέντρο μάζας (ή το ακίνητο σημείο) και** λ **μια σταθερά.**

Γενικά όμως, με τη βοήθεια του τανυστή αδράνειας J γράφουμε:

$$\vec{L} = \mathbf{J}\vec{\omega_0} \tag{2.29}$$

επομένως από τις 2.21 και 2.22 προκύπτει:

$$(\mathbf{J} - \lambda \vec{I})\vec{\omega_0} = 0 \tag{2.30}$$

όπου ο ταυτοτικός τελεστής $\vec{I}$ **, ισοδυναμεί με έναν μοναδιαίο πίνακα** 3×3 **στην συγκεκριμένη περίπτωση. Αναλύοντας την 2.23 στις επιμέρους συνιστώσες της, παίρνουμε το γραμμικό και ομογενές σύστημα:**

$$\begin{aligned} (J_{11} - \lambda)\vec{\omega_{01}} + J_{12}\vec{\omega_{02}} + J_{13}\vec{\omega_{03}} &= 0 \\ J_{21}\vec{\omega_{01}} + (J_{22} - \lambda)\vec{\omega_{02}} + J_{23}\vec{\omega_{03}} &= 0 \\ J_{31}\vec{\omega_{01}} + J_{32}\vec{\omega_{02}} + (J_{33} - \lambda)\vec{\omega_{03}} &= 0 \end{aligned} \tag{2.31}$$

όπου για τις συνιστώσες J_{ij} **έχουμε:**

$$\begin{aligned} J_{11} &= \sum m_i(y_i^2 + z_i^2) \\ J_{22} &= \sum m_i(z_i^2 + x_i^2) \\ J_{33} &= \sum m_i(x_i^2 + y_i^2) \end{aligned} \tag{2.32}$$

$$\begin{aligned} J_{12} = J_{21} &= -\sum m_i x_i y_i \\ J_{13} = J_{31} &= -\sum m_i x_i z_i \\ J_{23} = J_{32} &= -\sum m_i y_i z_i \end{aligned} \tag{2.33}$$

Το σύστημα 2.24 έχει μη μηδενική λύση όταν η ορίζουσα του είναι ίση με μηδέν, δηλαδή:

$$\begin{vmatrix} J_{11}-\lambda & J_{12} & J_{13} \\ J_{21} & J_{22}-\lambda & J_{23} \\ J_{31} & J_{32} & J_{33}-\lambda \end{vmatrix} = 0 \tag{2.34}$$

Λαμβάνοντας υπόψιν ότι ο πίνακας του τανυστή αδράνειας, όπως φαίνεται από τις 2.25 και 2.26 είναι συμμετρικός, η 2.27 θα έχει τρεις πραγματικές ρίζες τις $\lambda_1, \lambda_2, \lambda_3$. Σε κάθεμια από αυτές τις ρίζες (ιδιοτιμές) αντιστοιχεί και μια μη μηδενική λύση του συστήματος 2.24. Υπολογίζονται λοιπόν, τρια διαφορετικά ιδιοδιανύσματα, τα $\vec{\omega_0'}, \vec{\omega_0''}, \vec{\omega_0'''}$ τα οποία θα είναι μεταξύ τους ανα δύο κάθετα και επιπλέον έχουν την ιδιότητα το αντίστοιχο διάνυσμα της στροφορμής να είναι παράλληλο προς αυτά, ώστε:

$$L_1 = \lambda_1 \vec{\omega_0'}, L_2 = \lambda_2 \vec{\omega_0''}, L_3 = \lambda_3 \vec{\omega_0'''} \tag{2.35}$$

Φτάνουμε λοιπόν στο συμπέρασμα, ότι από το σημείο Ο του στερεού σώματος διέρχονται τρεις διευθύνσεις, μεταξύ τους κάθετες , οι οποίες καθώς το στερεό περιστρέφεται γύρω από μια εξ αυτών το διάνυσμα της στροφορμής να διατηρείται παράλληλο προς αυτήν τη διεύθυνση. Οι διευθύνσεις αυτές, ονομάζονται κύριες διευθύνσεις του τανυστή αδράνειας σε αυτό το σημείο και είναι διαφορετικές για διαφορετικά σημεία του στερεού.

Ας θεωρήσουμε ένα ορθογώνιο καρτεσιανό σύστημα συντεταγμένων $Ox_1x_2x_3$, στερεά συνδεδεμένο με το στερεό σώμα, με άξονες παράλληλους στις κύριες διευθύνσεις του τανυστή αδράνειας στο σημείο Ο. Αν το διάνυσμα της γωνιακής ταχύτητας είναι παράλληλο προς τον άξονα Ox_3 θα είναι:

$$\vec{\omega_3} = \omega_3 \hat{e_3} \tag{2.36}$$

με στροφορμή σε αυτόν τον άξονα:

$$L_3 = \lambda_3 \vec{\omega_3} = \lambda_3 \omega_3 \hat{e_3} \tag{2.37}$$

καθώς το $\vec{\omega}$ είναι παράλληλο προς μια κύρια διέυθυνση και συνεπώς λόγω της 2.28, αντίστοιχες σχέσεις ισχύουν και για τη στροφορμή στους άλλους άξονες. Επιπλέον, επειδή ισχύει

με βάση τα προηγούμενα ότι:

$$\begin{aligned} L_1 &= J_{11}\omega_1 + J_{12}\omega_2 + J_{13}\omega_3 \\ L_2 &= J_{21}\omega_1 + J_{22}\omega_2 + J_{23}\omega_3 \\ L_3 &= J_{31}\omega_1 + J_{32}\omega_2 + J_{33}\omega_3 \end{aligned} \tag{2.38}$$

προκύπτει ότι στο διάνυσμα της γωνιακής ταχύτητας $\vec{\omega} = (\omega_1, \omega_2, \omega_3)$ **θα αντιστοιχεί η στροφορμή** $\vec{L} = (\lambda_1\omega_1, \lambda_2\omega_2, \lambda_3\omega_3)$ **, επομένως έχουμε:**

$$\begin{aligned} L_1 &= \lambda_1\omega_1 \\ L_2 &= \lambda_2\omega_2 \\ L_2 &= \lambda_3\omega_3 \end{aligned} \tag{2.39}$$

Τέλος από τη σύγκριση των σχέσεων 2.31 και 2.32 έχουμε:

$$\begin{aligned} J_{11} &= \lambda_1 \\ J_{22} &= \lambda_2 \\ J_{33} &= \lambda_3 \\ J_{12} = J_{21} = J_{13} = J_{31} = J_{23} = J_{32} &= 0 \end{aligned} \tag{2.40}$$

Οι τιμές $J_{11} = \lambda_1$, $J_{22} = \lambda_2$ **και** $J_{33} = \lambda_3$ **ονομάζονται κύριες ροπές αδράνειας.**

2.4 Μέθοδος αριθμητικής ολοκλήρωσης

Σε αυτό το σημείο θα αναφερθούμε στη μέθοδο που θα εφαρμοστεί για την αριθμητική ολοκλήρωση των εξισώσεων 2.13 εως 2.15, δηλαδή των διαφορικών εξισώσεων που περιγράφουν την τροχιά του σώματος. Θεωρούμε αρχικά το σύστημα διαφορικών εξισώσεων $\dot{y} = f(x, y)$ όπου $y = (y_1, y_2, ...)$.

2.4.1 Βελτιωμένη μέθοδος μέσου σημείου

Η βελτιωμένη μέθοδος μέσου σημείου είναι μια αριθμητική μέθοδος ολοκλήρωσης δεύτερης τάξης, η οποία προσφέρει το πλεονέκτημα της απαίτησης ενός μόνο υπολογισμού για κάθε βήμα h , σε αντίθεση με την μέθοδο $Runge - Kutta$ δεύτερης τάξης που απαιτεί δύο υπολογισμούς ανά βήμα.

Η μέθοδος αυτή βρίσκει την πραγματική εφαρμογή της μέσω της τεχνικής $Bulirsch - Stoer$ [8], την οποία και θα εφαρμόσουμε στην επίλυση του προβλήματος αυτής της εργασίας.

Συνοπτικά, η βελτιωμένη μέθοδος μέσου σημείου θεωρεί αρχικά ένα διάνυσμα από εξαρτημένες μεταβλητές $y(x)$, και το μετακινεί από τη θέση x στη θέση $x + H$ μέσω μιας σειράς n βημάτων, μεγέθους h όπου:

$$h = H/n \tag{2.41}$$

Το πλήθος των υπολογισμών στην δεξιά μεριά που απαιτούνατι συνολικά από αυτή τη μέθοδο είναι $n + 1$. Οι αναδρομικές σχέσεις τις μεθόδου είναι οι εξής:

$$\begin{aligned} z_0 &\equiv y(x) \\ z_1 &= z_0 + hf(x, z_0) \\ z_{m+1} &= z_{m-1} + 2hf(x + mh, z_m) \\ y(x + H) \approx y_n &\equiv \frac{1}{2}[z_n + z_{n-1} + hf(x + H, z_n)] \end{aligned} \tag{2.42}$$

όπου $m = 1, 2, ..., n-1$. Στις προηγούμενες αναδρομικές σχέσεις τα z αποτελούν ενδιάμεσους υπολογισμούς καθώς προχωρούμε κατά βήμα h, ενώ το y_n είναι η τελική προσέγγιση του $y(x+H)$.

2.4.2 Μέθοδος $Bulirsch - Stoer$

Η τεχνική $Bulirsch-Stoer$ **βασίζεται στους προηγούμενους αναδρομικούς τύπους, και προτείνει την προσέγγιση του σημείου** H **με μεταβαλλόμενο βήμα. Συγκεκριμένα, η μεταβολή επιτυγχάνεται με σταδιακή αύξηση του πλήθους των υποβημάτων** n **κατα μια ακολουθία η οποία προτάθηκε από τους** $Bulirsch$ **και** $Stoer$ **:**

$$\begin{gathered} n = 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96... \\ n_j = 2n_{j-2} \end{gathered} \tag{2.43}$$

και η οποία πιο πρόσφατα βελτιώθηκε από τον $Deuflhard$ **ώστε να πάρει τη μορφή:**

$$\begin{gathered} n = 2, 4, 6, 8, 10, 12, 14... \\ n_j = 2(j+1) \end{gathered} \tag{2.44}$$

Σε κάθε βήμα δεν γνωρίζουμε εκ των προτέρων πόσο μακριά θα φτάσει η ακολουθία αυτή. Αφού κάθε διαδοχικό n **έχει δοκιμαστεί, εφαρμόζεται μια πολυωνυμική παρεμβολή. Η παρεμβολή δίνει τόσο τιμές του εκτιμούμενου μεγέθους όσο και τα αντίστοιχα σφάλματα. Εαν οι τιμές των σφαλμάτων κρίνονται αρκετά μεγάλες, προχωράμε τους υπολογισμούς σε μεγαλύτερες τιμές του** n **της ακολουθίας.**

Βέβαια, θα πρέπει να οριστεί κάποιο ανώτατο όριο στην προηγούμενη διαδικασία , πέρα από το οποίο θεωρείται πως υπάρχει κάποιο εμπόδιο προσεγγίζοντας το H**. Τότε, συνεχίζουμε με μείωση του** H **και όχι περαιτέρω υποδιαιρέσεις του βήματος μέσω του** n**. Επιπρόσθετα, η ακρίβεια των υπολογισμών αποτελεί και αυτή κριτήριο για την περάτωση της αριθμητικής διαδικασίας.**

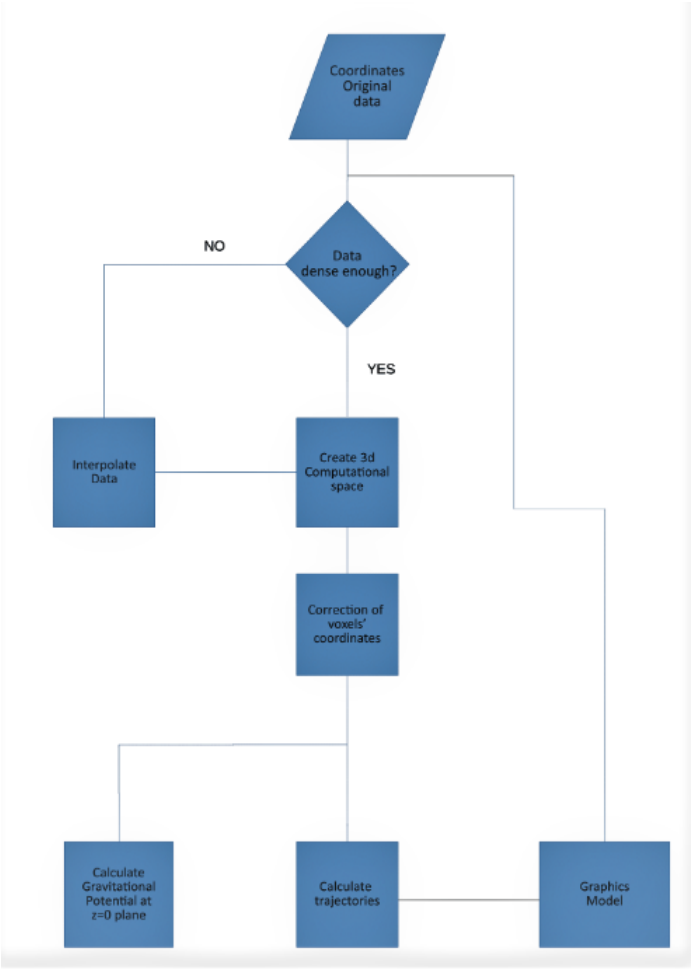
Κεφάλαιο 3

Περιγραφή του Αλγορίθμου

Στο κεφάλαιο που ακολουθεί, θα γίνει αναλυτική περιγραφή των προγραμμάτων που χρησιμοποιήθηκαν για την υπολογιστική μοντελοποίηση της κίνησης ενός μικρού σώματος - δορυφόρου υπό την επίδραση του βαρυτικού πεδίου του Δίδυμου. Για την πραγματοποίηση τους χρησιμοποιήθηκε η προγραμματιστική γλώσσα $C++$, σε συνδυασμό με βιβλιοθήκες όπως η *Eigen* [9] (υπολογισμός ιδιοτιμών και ιδιοδιανυσμάτων), η *ODESBS.h* που περιέχει συναρτήσεις που εφαρμόζουν την μέθοδο ολοκλήρωσης *Bulirsch – Stoer* (Βουγιατζής Γεώργιος) αλλά και το *framework* της *OpenGL* για την κατασκευή μιας τρισδιάστατης προσομοίωσης του αστεροειδή και την απεικόνιση των τροχιών που υπολογίστηκαν.

3.1 Διάγραμμα ροής

Για την κατασκευή του μοντέλου και τους υπολογισμούς των τροχιών, δημιουργήθηκαν μια σειρά από προγράμματα καθένα από τα οποία κατέχει σημαντικό ρόλο στην συνολική διαδικασία. Παρακάτω παρουσιάζουμε γραφικά ένα διάγραμμα ροής του αλγορίθμου, ο οποίος δίνει μια συνοπτική αλλά σαφή εικονά της μεθοδολογίας μας.



Σχήμα 3.1: Διάγραμμα ροής του αλγορίθμου μοντελοποίησης της μελέτης του βαρυτικού πεδίου του Δίδυμου.

Στην συνέχεια θα περιγράψουμε κάθε βήμα της διαδικασίας που αναπαριστάται στο Σχήμα 3.1 και τον τρόπο υλοποίησης της.

3.2 Interpolation δεδομένων χαρτογράφησης

Για την κατασκευή ενός ρεαλιστικού μοντέλου του Δίδυμου χρησιμοποίηθηκαν δεδομένα χαρτογράφησης του αστεροειδή από παρατηρήσεις με ειδικά τηλεσκόπια, τα οποία έχουν δημοσιευτεί σε ψηφιακή βιβλιοθήκη της *NASA* [10]. Συγκεκριμένα, πρόκειται για τα αρχεία *didymosl*162_*vertices.txt* και *didymosl*162_*indices.txt*.

Στο αρχείο *didymosl*162_*vertices.txt* βρίσκουμε έναν πίνακα που περιέχει τις συνταγμένες σημείων της επιφάνειας του αστεροειδή, σε ορθογώνιο σύστημα συντεταγμένων, στερεά συνδεδεμένο με το στερεό, με την αρχή των αξόνων του να συμπίπτει με το κέντρο του αστεροειδή με σφάλμα της τάξης των $10m$. Οι μονάδες των συντεταγμένων είναι σε km. Στο αρχείο υπάρχουν 1148 καταχωρήσεις σημείων της επιφάνειας (*vertices*) του Δίδυμου.

Το αρχείο *didymosl*162_*indices.txt* περιέχει τριάδες σημείων τις επιφάνειας, τα οποία ορίζουν τρίγωνα στην επιφάνεια του αστεροειδή. Το αρχείο περιέχει 2292 σετ τριάδων. Συνολικά, από όλα τα τρίγωνα που σχηματίζονται σύμφωνα με τις πληροφορίες που εμπεριέχονται στα δύο αρχεία, μπορεί να σχηματιστεί ένα πολύεδρο. Όσο πιο πυκνές είναι οι καταγραφές σημείων της επιφάνειας του αστεροειδή, τόσο πιο ρεαλιστικό είναι το μοντέλο του αστεροειδή που προκύπτει.

Επειδή ο Δίδυμος, παρουσιάζει έντονες σχηματικές ανωμαλίες κρίθηκε σκόπιμο στα πλαίσια αυτής της εργασίας να εφαρμοστεί μια μέθοδος γραμμικής παρεμβολής (*LinearInterpolation*) προκειμένου να μην αντιμετωπίσουμε δυσκολίες προσδιορισμού των ακριβών ορίων της επιφάνειας του στην υπολογιστική προσομοίωση του αργότερα.

Για την παρεμβολή σημείων στα δεδομένα χαρτογράφησης, αρχικά υπολογίστηκαν όλες οι αποστάσεις μεταξύ γειτονικών σημείων. Από αυτές δημιουργήσαμε μια λίστα με τις ελάχιστες τιμές. Από τη λίστα αυτή κρατήσαμε τη μέγιστη τιμή d_{max}. Θεωρώντας σαν ελάχιστη απόσταση το $d_{max}/2$, πήραμε διαδοχικά τις αποστάσεις μεταξύ γειτονικών σημείων και όπου αυτές ξεπερνούσαν το όριο $d_{max}/2$ τότε προσθέταμε ένα σημείο ακριβώς στο μέσο της απόστασης των εν λόγω σημείων.

Ακολουθούν οι βασικές συναρτήσεις που χρησιμοποιήθηκαν στο πρόγραμμα *Linear_Interpolation.cpp*:

```
void vertexdata(vector<vector<double> > &vec, int *N)
```

Η παραπάνω συνάρτηση διαβάζει το αρχείο *didymosl*162*_vertices.txt* και αποθηκεύει σε μια δυναμική δομή μνήμης *vector* τις καρτεσιανές συντεταγμένες των σημείων, ενώ αποθηκεύεται παράλληλα στην μεταβλητή N το πλήθος τους.Ο δυναμικός πίνακας *vec* που προκύπτει έχει τρεις στήλες για τις αντίστοιχες συντεταγμένες (x, y, z) κάθε στοιχείου και N γραμμές. Γνωρίζοντας λοιπόν, τον ακέραιο αριθμό που αντιστοιχεί στην εκάστοτε γραμμή του *vector vec* μπορούμε να έχουμε άμεση πρόσβαση στις συντεταγμένες του, κάτι το οποίο εκμεταλλευόμαστε παρακάτω για να υπολογίσουμε την απόσταση σημείου που βρίσκεται στη θέση m του πίνακα από σημείο που βρίσκεται στη θέση n.

```
void indexdata(vector<vector<int> > &ind, int *M)
```

Αντίστοιχα με την προηγούμενη συνάρτηση, η παραπάνω ρουτίνα αποθηκεύει τα σετ γειτονικών σημείων , που σχηματίζουν τρίγωνα από το αρχείο *didymosl*162*_indices.txt*, και το πλήθος τους στην μεταβλητή M. Με άλλα λόγια γνωρίζουμε πλέον σε ποιές γραμμές του πίνακα *vec* που δημιούργησε η προηγούμενη συνάρτηση βρίσκονται γειτονικά σημεία της επιφάνειας του αστεροειδή.

```
double Distance(int n, int m)
```

Η παραπάνω συνάρτηση υπολογίζει την απόσταση δύο σημείων του πίνακα *vec* στο χώρο.

```
void find_adjacent_distances(vector <double>
&adjacent_distances, int *L)
```

Η παραπάνω συνάρτηση υπολογίζει την απόσταση μεταξύ των γειτονικών σημείων και την καταχωρεί σε δυναμική δομή μνήμης. Επίσης, αποθηκεύεται το μέγεθος αυτής της δομής L.

```
double minDistance(int i, int N)
```

Η παραπάνω συνάρτηση υπολογίζει την ελάχιστη απόσταση του σημείου i από σύνολο N σημείων του πίνακα *vec*.

```
double max_minDistance(int N)
```

Η παραπάνω συνάρτηση βρίσκει τη μέγιστη τιμή απόστασης από σύνολο N σημείων.

```
void distance_to_coordinates(double distance, int N,
 int *max1, int *max2)
```

Η παραπάνω συνάρτηση παίρνει σαν όρισμα την αριθμιτική τιμή μιας απόστασης *distance*, το πλήθος σημείων N, και δύο *pointers* τους $*max1$ και $*max2$. Η συνάρτηση βρίσκει τα σημεία που απέχουν απόσταση ίση με *distance* και τα επιστρέφει μέσω των $*max1$ και $*max2$.

```
void linear_interpolation(int i , int j , int &N)
```

Η παραπάνω συνάρτηση παρεμβάλει σημείο μεταξύ των σημείων i και j και αναπροσαρμόζει το συνολικό πλήθος των σημείων του αστεροειδή N.

```
void output(int N)
```

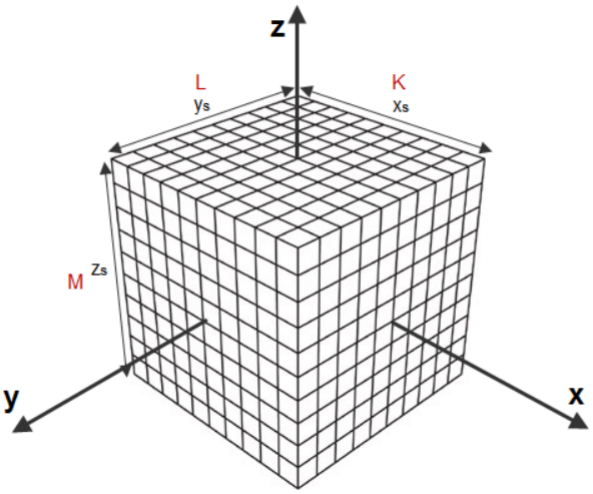
Τέλος, η παραπάνω συνάρτηση αποθηκεύει εκ νέου στο αρχείο *didymosl*162*_vertices.txt* τον πίνακα με τα *interpolated* δεδομένα συντεταγμένων.

3.3 Κατασκευή υπολογιστικού χώρου

Μετά την διαδικασία παρεμβολής, τα δεδομένα μας είναι αρκετά πυκνά για να προχωρήσουμε με ασφάλεια στην κατασκευή ενός τρισδιάστατου μοντέλου του αστεροειδή μέσα σε έναν κυβικό υπολογιστικό χώρο[11].

Οι διαστάσεις του υπολογιστικού χώρου μας ορίζονται με βάσει τις διαστάσεις του Δίδυμου. Υπολογίσαμε λοιπόν για κάθε άξονα τις διαστάσεις του σώματος ως εξής:

$$\begin{aligned} x_s &= x_{max} - x_{min} \\ y_s &= y_{max} - y_{min} \\ z_s &= z_{max} - z_{min} \end{aligned} \tag{3.1}$$



Σχήμα 3.2: Τρισδιάστατος υπολογιστικός χώρος που αποτελείται από κυβικές κυψελίδες.

όπου x_s, y_s, z_s είναι οι άξονες του υπολογιστικού χώρου, με τις αποστάσεις να μετρώνται σε km. Ακολούθως, κάθε μια από τις τρεις διαστάσεις διαιρείται από την πλευρά μιας κυβικής κυψελίδας. Έστω λοιπόν, ότι ο άξονας x χωράει K κυψελίδες, ο άξονας y χωράει L κυψελίδες και ο z χωράει M. Τότε ο υπολογιστικός μας χώρος είναι ένα κυβικό πλέγμα που αποτελείται από $K \times L \times M$ κυψελίδες, όπως φαίνεται στο **Σχήμα 3.2.** Κάθε κυψελίδα θα διακρίνεται από έναν ακέραιο αριθμό $voxel_index$ και ένα σετ δεικτών - συντεταγμένων (i, j, k).

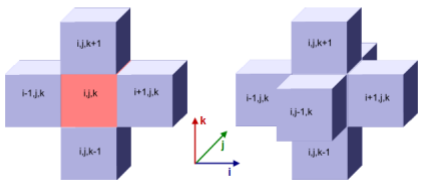
Σε αυτό το σημείο πρέπει να σημειωθεί ότι για να γνωρίζουμε που ακριβώς βρίσκεται κάθε κυψελίδα στο χώρο υπάρχει ανάγκη για μια σειρά από συναρτήσεις που επιτρέπουν την αντιστοίχηση των πραγματικών καρτεσιανών συντεταγμένων στου συστήματος αναφοράς μας , σε οποιαδήποτε κυψελίδα του υπολογιστικού χώρου η οποία θα φέρει τους δείκτες (i, j, k). Έτσι, ξεκινώντας από την αρχή του συστήματος συντεταγμένων $O(0, 0, 0)$, και προχωρώντας προς οποιαδήποτε κατεύθυνση είναι θεμιτό ανα πάσα στιγμή να ξέρουμε σε ποιούς δείκτες (i, j, k) (και άρα σε ποιά κυψελίδα) αντιστοιχούν οι εκάστοτε συντεταγμένες (x, y, z) και αντίστροφα.

Όσον αφορά την διάσταση της κυψελίδας, προκειμένου να την αρχικοποιήσουμε ακολουθούμε την παρακάτω διαδικασία:

Αρχικά, βρίσκουμε την ελάχιστη απόσταση κάθε σημείου της επιφάνειας του αστεροειδή από όλα τα άλλα και αποθηκεύουμε τις τιμές αυτές σε μια λίστα.

Στη συνέχεια, από τις τιμές αυτής της λίστας βρίσκουμε την μέγιστη, έστω α_{max_min}. Αρχικοποιούμε την πλευρά της κυψελίδας με βάση αυτήν την τιμή.

Εφόσον έχουμε πλεον μια αρχική εκτίμηση για την πλευρά των κυψελίδων, μπορούμε να διαιρέσουμε τους άξονες x_s, y_s, z_s με αυτήν για να βρούμε το πλήθος που αντιστοιχεί σε καθέναν απο αυτούς (K, L, M).



Σχήμα 3.3: Αναπαράσταση γειτονιάς κυψελίδας στο μοντέλο

Ξεκινώντας την κατασκευή του μοντέλου, θα πρέπει να αρχικοποιήσουμε όλες τις κυψελίδες του χώρου ως άδειες, δίνοντας τους την ακέραια τιμή 0. Ακολούθως βρίσκουμε τις συντεταγμένες των κυψελίδων που αντιστοιχούν στα σημεία της επιφάνειας του Δίδυμου (αρχείο *didymosl*162_*vertices.txt*) και σε

αυτά αναθέτουμε τον ακέραιο 2 που αντιστοιχεί σε επιφανειακή κυψελίδα. Γνωρίζοντας πλέον τα όρια του σώματος, ξεκινάμε από εσωτερικό σημείο του αστεροειδή, όπως το κέντρο μάζας O(0,0,0), στο οποίο αναθέτουμε την τιμή 1, που εκπροσωπεί τα εσωτερικά σημεία. Σταδιακά τώρα, ελέγχουμε τις γειτονικές κυψελίδες (βλ. Σχήμα 3.3 , εξετάζοντας την ιδιότητα τους, δηλαδή αν είναι άδειες (0) ή επιφανειακές (2). Με τον έλεγχο αυτό μπορούμε να γεμίσουμε τον αστεροειδή με κυψελίδες ιδιότητας 1 , χωρίς να ξεφύγουμε από τα όρια του .

Κατασκευάζουμε, λοιπόν το κυβικό πλέγμα με βάση τις προηγούμενες παραμέτρους. Σε περίπτωση που οι κυψελίδες δεν συγκρατούνται από την κλειστή επιφάνεια του αστεροειδή, προχωράμε σε αύξηση της πλευράς της κυψελίδας. Εφόσον οι διαστάσεις του υπολογιστικού χώρου παραμένουν σταθερές, εξετάζουμε αν οι κυψελίδες με μεγαλύτερες πλεον διαστάσεις μπορούν να γεμίσουν το εσωτερικό του αστεροειδή χωρίς να έχουμε περαιτέρω διαρροές. Ο έλεγχος αυτός πραγματοποιείται εντός μιας επαναληπτικής διαδικασίας μέχρι να παραχθεί ένα μοντέλο δίχως διαρροές.

Παρακάτω παρατίθενται οι συναρτήσεις που χρησιμοποίηθηκαν για την υλοποίηση του αλγορίθμου που περιγράψαμε:

```
void VoxelGridCoord(double x, double y, double z,
 int *i, int *j, int *k)
```

Η παραπάνω συνάρτηση δέχεται ως όρισμα τις καρτεσιανές συντεταγμένες (x, y, z) ενός σημείου στον υπολογιστικό χώρο και επιστρέφει τους δείκτες (i, j, k) που δίνουν τη θέση της κυψελίδας στο πλέγμα.

```
int VoxelIndex(int i, int j, int k)
```

Η παραπάνω συνάρτηση υπολογίζει τον αύξοντα ακέραιο αριθμό της κυψελίδας με δείκτες (i, j, k).

```
int  VoxelGridPos(int voxel_index, int *i, int *j, int *k)
```

Η παραπάνω συνάρτηση είναι η αντίστροφη της προηγούμενης και βρίσκει τα στοιχεία (i, j, k) από άυξοντα αριθμό της κυψελίδας *voxel_index*.

```
void CartesianGridCoord(int i, int j, int k,
 double *x, double *y, double *z)
```

Η παραπάνω συνάρτηση υπολογίζει τις καρτεσιανές συντεταγμένες του κέντρου της κυψελίδας με δείκτες (i, j, k).

```
void GetMinMax(int N, double *mx, double *my, double *mz
, double *Mx, double *My, double *Mz)
```

Η παραπάνω συνάρτηση βρίσκει τα όρια του πλέγματος από τις συντεταγμένες των σημείων της επιφάνειας του Δίδυμου που βρίσκονται στο αρχείο *didymosl*162_*vertices.txt* **.**

```
int GetWorkingGrid(int k, int rows)
```

Η παραπάνω συνάρτηση υπολογίζει τις παραμέτρους K, L, M του πλέγματος καθώς και την πλευρά της κυβικής κυψελίδας.

```
int GetModelCoord()
```

Η παραπάνω συνάρτηση περιέχει όλα τα βήματα και ελέγχους που είναι απαραίτητοι για την κατασκευή του υπολογιστικού χώρου.

Τρέχοντας το πρόγραμμα *computational_space.cpp* **προκύπτουν δύο αρχεία. Το** *coeffiecients.txt* **που περιέχει τις τιμές των παραμέτρων K, L, M καθώς και το πλευρά της κυψελίδας που εκλέχθηκε για την κατασκευή του μοντελου. Επίσης, στο αρχείο** *voxel_coordinates.txt* **περιέχονται οι καρτεσιανές συντεταγμένες των κέντρων των κύψελίδων, αποκλειστικά του αστεροειδή, και όχι των κενών κυψελίδων που προέκυψαν κατά την κατασκευή του πλέγματος. Μια επιπλέον στήλη στο ίδιο αρχείο φέρει τους ακέραιους αριθμούς 1 για κυψελίδες εσωτερικές του αστεροειδή και 2 για επιφανειακές κυψελίδες.**

3.4 Διόρθωση συντεταγμένων

Αν και οι συντεταγμένες στο αρχικό *didymosl*162*_vertices.txt* είχαν δοθεί σε σύστημα κυρίων αξόνων αδράνειας, για τις συντεταγμένες των κυψελίδων του μοντέλου θα πρέπει να γίνουν κατάλληλες διορθώσεις. Ακολουθούμε το εξής σκεπτικό:

Αρχικά μετακινούμε όλα τα σημεία κατά τέτοιο τρόπο ώστε η αρχή του συστήματος συντεταγμένων να αντιστοιχεί στο κέντρο μάζας του Δίδυμου με ακρίβεια $10^{-6}m$.

Στη συνέχεια υπολογίζονται τανυστή οι κύριοι άξονες αδράνειας με τη μεθοδολογία που αναφέρθηκε στη θεωρητική εισαγωγή.

Γνωρίζοντας πλεον τις ζητούμενες διευθύνσεις στρέφουμε το σύστημα συντεταγμένων κατά τέτοια γωνία ώστε οι άξονες του νέου συστήματος να συμπίπτουν με τους κύριους άξονες αδράνειας.

Παρακάτω παρατίθενται όλες οι συναρτήσεις που χρησιμοποιήθηκαν για την πραγματοποίηση των διορθώσεων των συντεταγμένων με το πρόγραμμα *main_inertia_axes.cpp* .

```
void calculate_center_mass(double *mcx, double *mcy,
 double *mcz, int Vox)
```

Η παραπάνω συνάρτηση υπολογίζει τις συντεταγμένες του κέντρου μάζας του αστεροειδή.

```
void correct_Center(double mcx, double mcy,
 double mcz, int vox)
```

Η παραπάνω συνάρτηση διορθώνει τις συντεταγμένες των κυψελίδων του υπολογιστικού χώρου βάσει του κέντρου μάζας του αστεροειδή.

```
void shift_coord_to_cm()
```

Η παραπάνω συνάρτηση περιέχει την επαναληπτική διαδικασία διόρθωσης των συντεταγμένων μέχρι να επιτευχθεί η επιθυμητή ακρίβεια.

```
void rotate_to_m_axes()
```

Η παραπάνω συνάρτηση περιστρέφει το σύστημα κατά τέτοιο τρόπο ώστε οι άξονες του νέου συστήματος να συμπίπτουν με τους κύριους άξονες αδράνειας.

```
void output(vector< vector <double> > voxels, double mass)
```

Η παραπάνω συνάρτηση αποθηκεύει στο αρχείο $shifted_V_coord_w_mass.txt$ **τις συντεταγμένες των διορθωμένων κατά τους κύριες άξονες μαζί με την μάζα που αντιστοιχεί σε κάθε κυψελίδα. Επειδή αντιμετωπίζουμε τον αστεροειδή σαν ένα ομοιογενές σώμα, σε κάθε κυψελίδα αντιστοιχίζουμε μάζα με τιμή** $m_i = 1/N$ **(κανονικοποίηση της μάζας του αστεροειδή).**

3.5 Υπολογισμός δυναμικής ενέργειας στο επίπεδο $z = 0$

Λόγω περιστροφής του Δίδυμου γύρω από τον άξονα z ο οποίος διέρχεται από το κέντρο μάζας του, ο υπολογισμός της βαρυτικής δυναμικής ενέργειας στο επίπεδο $z = 0$ θα μας δώσει μια σαφή εικόνα για το βαρυτικό πεδίο που δημιουργείται γύρω από τον αστεροειδή.

Για το σκοπό αυτό δημιουργήθηκε το πρόγραμμα *num_cal.cpp* στο οποίο χρησιμοποιώντας τη σχέση 2.17 που δίνει την βαρυτική δυναμική ενέργεια στο κανονικοποιημένο μας σύστημα.

Παρακάτω παρουσιάζονται οι επιμέρους συναρτήσεις του προγράμματος.

```
void voxeldata(vector< vector <double> > &vox, int *N)
```

Η παραπάνω συνάρτηση αποθηκεύει από το αρχείο *shifted_V_coord_w_mass.txt* τις καρτεσιανές συντεταγμένες, την ιδιότητα (εσωτερική ή επιφανειακή) και την μάζα (σταθερή $1/N$) σε μια δυναμική δόμη. Επιπλέον συγκρατείται το πλήθος των ζητούμενων κυψελίδων N.

```
double z0_distance(double x, double y, int i)
```

Η παραπάνω συνάρτηση χρησιμοποιεί τα δεδομένα της θέσης κάθε κυψελίδας για να υπολογίσει την απόσταση της από σημεία του επιπέδου $z = 0$

```
void plane_points(vector< vector <double> > &plane,
 double size)
```

Η παραπάνω συνάρτηση παράγει μια δυναμική λίστα με σημεία της επιφάνειας $z = 0$ μεταξύ επιθυμητών ορίων και με συγκεκριμένο βήμα. Επιπλέον, αρχικοποιεί για κάθε σημείο την ενέργεια και το δυναμικό με τιμή 0.

```
void calculate_U(vector< vector <double> > &plane)
```

Η παραπάνω συνάρτηση εκτελεί τον αριθμητικό υπολογισμό της δυναμικής ενέργειας για όλα τα επιθυμητά σημεία της επιφάνειας $z = 0$ που είναι αποθηκευμένα στην δυναμική λίστα *plane*.

```
void output(vector< vector <double> > plane, int size)
```

Η παραπάνω συνάρτηση αποθηκεύει στο αρχείο *num_calc_U_z0.txt* **τις τιμές του U και τις αντίστοιχες συντεταγμένες (x, y)**

3.6 Υπολογισμός τροχιών

Για τον υπολογισμό τροχιών σε μια σειρά περιπτώσεων, δημιουργήθηκε το πρόγραμμα - *project NumericalIntegrations*, **το οποίο αποτελείται από μια σειρά υποπρογραμμάτων. Η μεθοδολογία που ακολουθήσαμε αποτελεί ουσιαστηκά εφαρμογή των διαφορικών εξισώσεων της κίνησης 2.13 εως 2.16.**

Επιπλέον, με κατάλληλες μετατροπές του περιστρεφόμενου συστήματος σε αδρανειακό, επιτεύχθηκε ο υπολογισμός των Κεπλέριων τροχιακών στοιχείων για κάθε τροχιά. Αντίστροφα, ξεκινώντας από παραμέτρους σε αδρανειακό σύστημα κάναμε παρόμοιους υπολογισμούς για να πάρουμε αποτελέσματα σε περιστρεφόμενο σύστημα. Οι μετασχηματισμοί που χρησιμοποιήθηκαν αναφέρονται στις 2.19 και 2.20 , όπου θεωρήσαμε γωνιακή ταχύτητα χρονικά ανεξάρτητη με μέτρο τη μονάδα.

Ακολουθούν σχετικές συναρτήσεις στο πρόγραμμα *dev1gv.cpp* **:**

```
void Initialize(double acc)
```

Η παραπάνω συνάρτηση αρχικοποιεί βασικές παραμέτρους όπως η ακρίβεια των υπολογισμών, για τις αριθμιτικές ολοκληρώσεις.

```
double distance(double x, double y, double z, int i)
```

Η παραπάνω συνάρτηση υπολογίζει την απόσταση σημείου με συντεταγμένες (x, y, z) από την κυψελίδα i του αστεροειδή.

```
void Sum(int N, double x, double y, double z,
 double *sumx, double *sumy, double *sumz)
```

Η παραπάνω συνάρτηση υπολογίζει τις μερικές παραγώγους του δυναμικού $sumx, sumy, sumz$ σε θέση (x, y, z).

```
void DEQS(double t, double X[], double Y[])
```

Η παραπάνω συνάρτηση δέχεται σαν όρισμα τον χρόνο t, και τον πίνακα $X[\ ]$ που περιέχει τις συνιστώσες του διανύσματος της θέσης και ταχύτητας εκείνη τη χρονική στιγμή, του σώματος που βρίσκεται σε τροχιά. Επιστρέφει το πίνακα $Y[\ ]$ που περιέχει τις αντίστοιχες συνιστώσες του διανύσματος της ταχύτητας και επιτάχυνσης, όπως θεωρήθηκαν στις σχέσεις 2.13 εως 2.15

```
void TransformationRtoI(double x[], double X[])
```

Η παραπάνω συνάρτηση παίρνει σαν όρισμα τις συνιστώσες του διανύσματος θέσης και αυτού της ταχύτητας από πίνακα $x[\ ]$ σε περιστρεφόμενο σύστημα αναφοράς και τις μετασχηματίζει σε ισοδύναμες ποσότητες σε αδρανειακό σύστημα (πίνακας $X[\ ]$).

```
void TransformationItoR(double X[], double Y[])
```

Η παραπάνω συνάρτηση είναι η αντίστροφη της προηγούμενης. Παίρνει σαν όρισμα τις συνιστώσες του διανύσματος θέσης και αυτού της ταχύτητας από πίνακα $X[\ ]$ σε αδρανειακό σύστημα αναφοράς και τις μετασχηματίζει σε ισοδύναμες ποσότητες σε περιστρεφόμενο σύστημα (πίνακας $Y[\ ]$)

```
void RotationZ(double theta, double X[], double Y[])
```

Η παραπάνω συνάρτηση έχει σαν όρισμα τις συνιστώσες των διανυσμάτων θέσης και ταχύτητας του σώματος την χρονική στιγμή t (πίνακας $X[\ ]$) και την γωνία *theta* (μετρούμενη σε *radians*) . Η λειτουργία της είναι να στρέψει το σώμα από τη θέση του κατά γωνία ίση με *theta* και να επιστρέψει μέσω του πίνακα $Y[\ ]$ τις νέες συντεταγμένες του σώματος.

```
double Calc_Energy(double X[])
```

Με την παραπάνω συνάρτηση γίνεται υπολογισμός της συνολικής ενέγειας του σώματος με συντεταγμένες που δίνονται μέσω του πίνακα $X[\ ]$ συγκεκριμένη χρονική στιγμή.

```
void GetApproxCircular(double R, double X0[])
```

Με την παραπάνω συνάρτηση υπολογίζονται οι αρχικές συνθήκες (πίνακας $X0[\ ]$)μιας κατά προσέγγιση κυκλικής κίνησης του σώματος, σε περιστρεφόμενο σύστημα συντεταγμένων και σε απόσταση R από το κέντρο μάζας του αστεροειδή.

```
void GetApproxCircular(double R, double ThetaRot,
 double X0[]))
```

Η παραπάνω συνάρτηση επιτελεί την ίδια λειτουργία με την προηγούμενη με την διαφορά ότι στρέφει επιπλεόν το σώμα κατα γωνία *ThetaRot*.

```
int Crush(double X[])
```

Με την παραπάνω συνάρτηση γίνεται έλεγχος στα δεδομένα συντεταγμένων του σώματος που περιέχει ο πίνακας $X[\ ]$ σε δεδομένη χρονική στιμή t για το άν το σώμα έχει προσκρούσει πάνω στον αστεροειδή (απόσταση από το κέντρο $<= 0.45km$).

Χρησιμοποιώντας τις παραπάνω συναρτήσεις σε συνδυασμό με εκείνες της βιβλιοθήκης *restroe3.h* αλλά και αυτές της *ODESBS.h* εκτελούμε μια σειρά από υπολογισμούς τροχιών οι οποίες παρουσιάζονται αναλυτικά στο κεφάλαιο των αποτελεσμάτων.

3.7 Γραφική απεικόνιση αστεροειδή

Σε αυτήν την ενότητα θα αναφερθούμε στο πρόγραμμα που κατασκευάστηκε για την γραφική απεικόνιση του αστεροειδή. Αρχικά, τα δεδομένα που χρησιμοποιήθηκαν είναι τα αυθεντικά δεδομένα του αρχείου *didymosl*162_*vertices.txt* χωρίς την εφαρμογή της γραμμικής παρεμβολής επιπλέον σημείων, όπως και τα δεδομένα του αρχείου *didymosl*162_*indices.txt*. Η παραγωγή των γραφικών έγινε μέσω του *framework* της *OpenGL*, για την γλώσσα $C++$. Ακολουθεί η περιγραφή του προγράμματος:

```
void initGL()
```

Με την συνάρτηση πραγματοποιείται μια σειρά από βασικές αρχικοποιήσεις, απαραίτητες για τον σχεδιασμό αντικειμένων από το πρόγραμμα μας. Επιπλέον, δηλώνονται οι δομές που θα σχηματίσουν τους άξονες του συστήματος αναφοράς (*axes_list*), τις ακμές του πολυέδρου - αστεροειδή (*ind_list*), οι τριγωνικές επιφάνειες που ορίζουν τα επιφανειακά σημεία (*faces_list*) καθώς και η τροχιά σώματος γύρω από τον αστεροειδή που θέλουμε να απεικονίσουμε στην εκάστοτε περίπτωση (*graph_list*).

```
void display()
```

Με την παραπάνω συνάρτηση φορτώνονται στο *buffer* οι δομές που έχουν νωρίτερα αρχικοποιηθεί προκειμένου να γίνει η απεικόνιση τους.

```
void processSpecialKeys(int key, int xx, int yy)
```

Η παραπάνω συνάρτηση δέχεται σαν όρισμα είσοδο από τον χρήστη με το πάτημα των δεικτών του πληκτρολογίου, οι οποίοι μετακινούν την κάμερα για να παρατηρηθεί ο αστεροειδής υπο διαφορετική οπτική γωνία.

```
void timer(int value)
```

Συνάρτηση καταγραφής του χρόνου.

```
void reshape(GLsizei width, GLsizei height)
```

Συνάρτηση προσαρμογής του μεγέθους του πάραθυρου της γραφικής απεικόνισης.

```
void create_data_arrays()
```

Με την παραπάνω συνάρτηση γίνεται αποθήκευση σε δυναμικές δομές των δεδομένων από τα αρχεία *didymosl*162*_vertices.txt* **,** *didymosl*162*_indices.txt* **και από το αρχείο που περιέχει τις συντεταγμένες της εκάστοτε τροχιάς.**

Κεφάλαιο 4

Αποτελέσματα

Στο κεφάλαιο αυτό, θα ακολουθήσει η ανάλυση των αποτελεσμάτων που πάρθηκαν από την εκτέλεση του αλγορίθμου για την μελέτη του βαρυτικού πεδίου του Δίδυμου.

4.1 Υπολογιστικός χώρος και διορθώσεις συντεταγμένων

Αρχικά θα πρέπει να επισημανθούν οι βασικές τροποποιήσεις των αρχικών δεδομένων από την λειτουργία των προγραμμάτων.

Από την εκτέλεση του κώδικα *Linear_Interpolation.txt* προέκυψε το αρχείο *didymosl*162*_vertices.txt* με τις καρτεσιανές συντεταγμένες 8024 επιφανειακών σημείων του αστεροειδή (αναφέρεται ότι το αυθεντικό αρχείο, πριν τις παρεμβολές περιείχε μόλις 1148 επιφανειακά σημεία).

Από τον κώδικα *computational_space* λάβαμε σαν έξοδο τα αρχεία: *voxel_coordinates.txt* με τις καρτεσιανές συντεταγμένες 14,316 κυψελίδων που απαρτίζουν το μοντέλο του αστεροειδή με αναφορά στην ιδιότητα κάθεμιας (εσωτερική ή επιφανειακή) , και το αρχείο $coefficiants.txt$ με τις παραμέτρους του πλέγματος που φαίνονται στον Πίνακα 4.1

K	L	M	$voxelside(km)$
33	33	31	0.0271871

Πίνακας 4.1: Αριθμός κυψελίδων ανά άξονα (K, L, M) και πλευρά κυψελίδας στο μοντέλο του Δίδυμου.

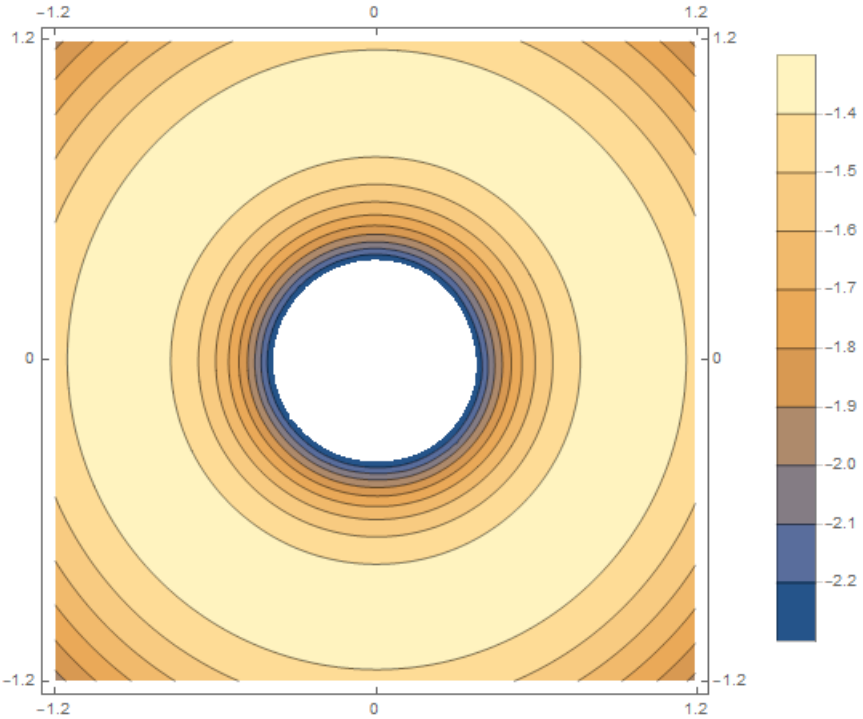
```
 Initial calculation of center of mass:
(2.00925e-005 , -0.000163624 , -0.000190286)
Iteration:  0
Center of mass coordinates after correction:
(-5.37235e-017 , -3.25142e-016 , -1.45058e-014)
--------------------------
voxel mass: 6.98519e-005
N: 14316

Eigenvalue1 = 0.0667692
Eigenvalue2 = 0.0680305
Eigenvalue3 = 0.0688407
Axis z is the max inertia axis
The first eigenvector of the Inertia tensor J is v1 : ( -0.998331, 0.0562881, 0.0129225)
The second eigenvector of the Inertia tensor J is v2 : ( 0.0545667, 0.992633, -0.108175)
The third eigenvector of the Inertia tensor J is v3 : ( -0.0189162, -0.107289, -0.994048)
v1.v2 = 1.80411e-016
v2.v3 = 8.29675e-015
v3.v1 = -2.70617e-016
Voxel information and its mass are now stored in the file:shifted_V_coord_w_mass.txt
```

Σχήμα 4.1: Η έξοδος στο τερματικό από τον κώδικα *inertia_axes_coord.cpp*. Το κέντρο μάζας υπολογίζεται με ικανοποιητική ακρίβεια στην αρχή (0,0,0). Είναι επίσης εμφανές, από τις απειροστά μικρές τιμές των αριθμητικών γινομένων των ιδιοδιανυσμάτων πως εξασφαλίζεται η καθετότητα τους.

Ακολούθως, από την διόρθωση των συντεταγμένων κατά τους κύριους άξονες από τον κώδικα του προγράμματος *inertia_axes_coord.cpp* **, προκύπτει το αρχείο** $shifted_V_coord_w_mass.txt$ **που φέρει πέραν των διορθώσεων και την τιμή στοιχειώδης μάζας που αντιστοιχεί σε κάθε κυψελίδα το συνολικό του άθροισμα να ισούται με μονάδα.**

4.2 Αποτελέσματα υπολογισμού ενέργειας στο επίπεδο $z = 0$



Σχήμα 4.2: Τιμές της βαρυτκής δυναμικής ενέργειας στο επίπεδο $z = 0$. Η περιοχή του αστεροειδή φαίνεται λευκή. Μόναδα μήκους είναι τα km. Σε απόσταση περίπου ενός χιλιομέτρου παρατηρείται μια περιοχή όπου η ενέργεια παίρνει τις μεγαλύτερες τιμές. Σε αυτήν την απόσταση περιφέραται ο δορυφόρος *Didymoon* του Δίδυμου σύμφωνα με παρατηρησιακά δεδομένα.

Από την εκτέλεση του κώδικα *num_calc.cpp* **προέκυψε το αρχείο** *num_calc_U_z0.txt* **στο οπόιο έχει υπολογιστεί η τιμής της βαρυτικής δυναμικής ενέργειας, λόγω του βαρυτικού πεδίου του Δίδυμου σε σημεία του επιπέδου $z = 0$. Τα σημεία επιλέχθηκαν μεταξύ των ορίων -1.2 εως 1.2 km , με βήμα 0.01km, για τους άξονες x και y στο σύστημα συντεταγμένων που έχει αρχή το κέντρο μάζας του Δίδυμου. Συνολικά προέκυψαν 57600 καταχωρήσεις τις οποίες απεικονίσαμε στο διάγραμμα του Σχήματος 4.2.**

Η εικόνα του πεδίου που παίρνουμε από το Σχήμα 4.2 συμβαδίζει με την εξίσωση 2.17. Καθώς απομακρυνόμαστε από την

αρχή των αξόνων, ο αρνητικός όρος $-\frac{1}{2}(x^2+y^2)$ παίρνει όλο και μικρότερες (αρνητικότερες) τιμές ενώ ο επίσης αρνητικός όρος του δυναμικού $-\delta\sum_{i=1}^{N} dm_i/r_i$ φαίνεται να γίνεται μεγαλύτερος (θετικότερος). Σαν αποτέλεσμα προκύπτει μια δισκοειδής περιοχή, όπου φαίνεται η ενέργεια να παίρνει τις μεγαλυτερες της τιμές. Η αδυναμία εντοπισμού συγκεκριμένων μεγίστων ή αντίστοιχα ελαχίστων του δυναμικού καθιστά δύσκολη την εξαγωγή συμπεράσματος ως προς την ύπαρξη σημείων ισορροπίας γύρω από τον αστεροειδή. Η ακρίβεια των αποτελεσμάτων εξαρτάται από το πόσο πυκνά είναι τα σημεία που λήφθηκαν στην επιφάνεια $z = 0$.

Είναι σημαντικό να σημειωθεί ότι σε μεγάλες αποστάσεις από τον αστεροειδή, το πρόβλημα μας ανάγεται από μελέτη του βαρυτικού πεδίου ενός ομοιογενούς στερεού ανώμαλου σχήματος στη μελέτη της βαρυτικής έλξης ενός σφαιρικού σώματος. Αυτό συμβαίνει διότι οι ανωμαλίες του βαρυτικού πεδίου του Δίδυμου εντοπίζονται σε κοντινές αποστάσεις από αυτόν.

4.3 Αποτελέσματα υπολογισμού τροχιών

Σε αυτήν την παράγραφο θα παρουσιάσουμε τα αποτελέσματα από τους αριθμητικούς υπολογισμούς κατά προσέγγιση κυκλικών τροχιών γύρω από τον αστεροειδή Δίδυμο.

Για το σκοπό αυτό, χρησιμοποιήθηκαν οι κανονικοποιημένες διαφορικές εξισώσεις 2.20 εως 2.22, οι οποίες περιγράφουν την κίνηση του σώματος στο περιστρεφόμενο σύστημα. Στο σώμα δόθηκαν αρχικές συνθήκες τέτοιες ώστε άν βρισκόταν σε ένα αδρανειακό σύστημα η τροχιά του να ήταν κατα προσεγγιση κυκλική. Συγκεκριμένα, εξετάστηκαν οι παρακάτω περιπτώσεις:

1. Σε αδρανειακό σύστημα συντεταγμένων θεωρούμε ότι το σώμα βρίσκεται σε απόσταση R από την αρχή O(0,0,0) και πάνω στον άξονα x με συνιστώσα ταχύτητας τέτοια ώστε να εκτελεί κατά προσέγγιση κυκλική κίνηση γύρω από τον αστεροειδή. Επειδή οι διαφορικές εξισώσεις αναφέρονται στο περιστρεφόμενο σύστημα, είναι απαραίτητος ο μετασχηματισμός των αρχικών συνθηκών από αδρανειακό σε περιστρεφόμενο. Το σώμα τοποθετήθηκε στις θέσεις $A(0.48km, 0, 0)$, $B(0.6km, 0, 0)$, $\Gamma(0.8km, 0, 0)$ και τέλος $\Delta(1.18km, 0, 0)$.

2. Στην περίπτωση που το σώμα βρίσκεται στη θέση $\Gamma(0.8km, 0, 0)$ αλλάξαμε το σχετικό προσανατολισμό του σε σχέση με τον αστεροειδή κατά γωνία $theta$ για να εξετάσουμε αν θα παρουσιαστεί κάποια αλλαγή στην τροχιά του.

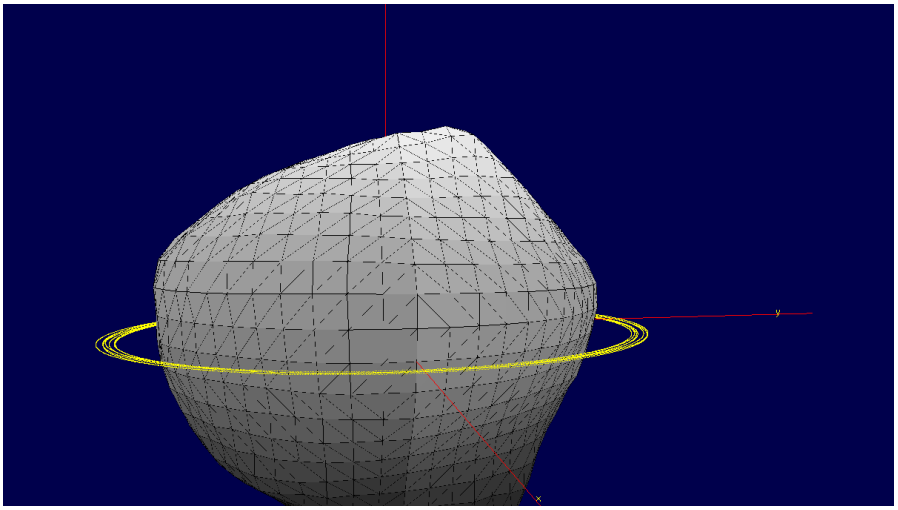
3. Τέλος δίνοντας ως αρχικές συνθήκες τις κάτάλληλες τιμές των Κεπλέριων τροχιακών στοιχείων μιας κυκλικής κίνησης, εξετάσαμε την τροχιά του σώματος που ξεκινάει από το $\Gamma(0.8km, 0, 0)$.

4.3.1 Υπολογισμός κυκλικών τροχιών

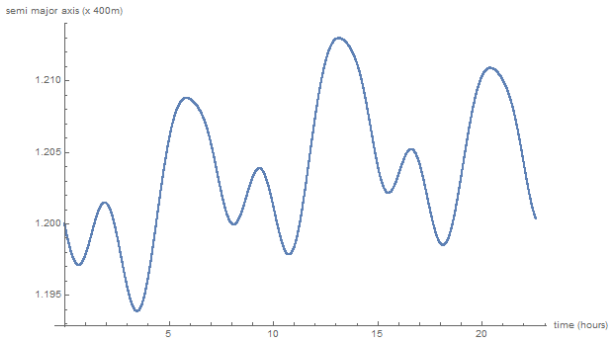
Για να πραγματοποιήσει το σώμα κυκλική κίνηση στη θέση $(R, 0, 0)$ σε ένα αδρανειακό σύστημα θα είναι απαραίτητο η συνιστώσα της ταχύτητας του κατά τον άξονα y να δίνεται από τη σχέση:

$$u_y = \sqrt{\delta / R} \tag{4.1}$$

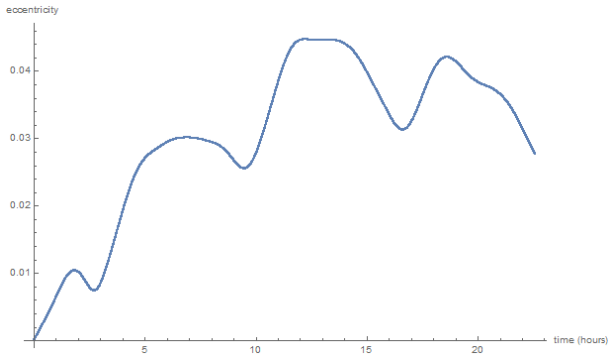
Με βάση του μετασχηματισμούς 2.27 βρίσκουμε τις ζητούμενες αρχικές συνθήκες στο περιστρεφόμενο σύστημα. Τα αποτελέσματα των υπολογισμών έχουν ως μονάδα μήκους τα $400m$, ενώ η μονάδα χρόνου είναι $0.36h$, ωστόσο κατά την αποθήκευση των δεδομένων σε λίστες, μετασχηματίσαμε τον χρόνο σε κανονικές μονάδες (όπως φαίνεται και στα παρακάτω διαγράμματα). Ακολουθούν τα αποτελέσματα (Κεπλέρια τροχιακά στοιχεία και απεικόνιση τροχιάς σε αδρανειακό σύστημα) για τις επιμέρους περιπτώσεις.



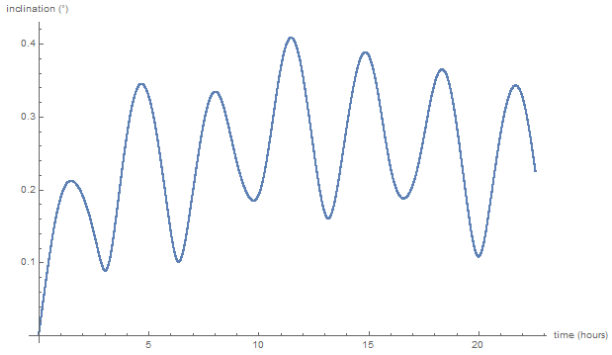
Σχήμα 4.3: Η τροχιά του σώματος όπως την αντιλαμβάνεται ο αδρανειακός παρατηρητής για προσεγγιστικά κυκλική κίνηση με αρχικές συνθήκες ξεκινώντας από τη θέση $x = 0.48km$.



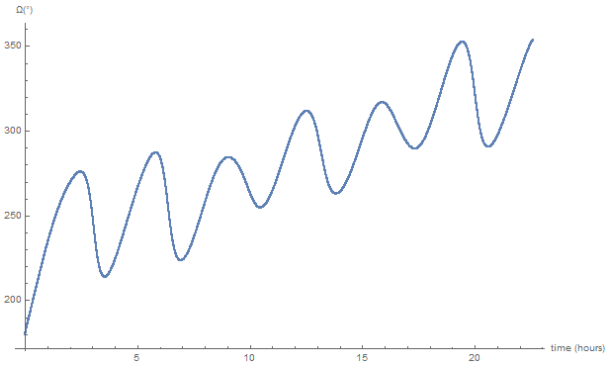
Σχήμα 4.4: Τιμές κύριου ημιάξονα $\alpha(\times 400m)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.48km$.



Σχήμα 4.5: Τιμές εκκεντρότητας e συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.48km$.



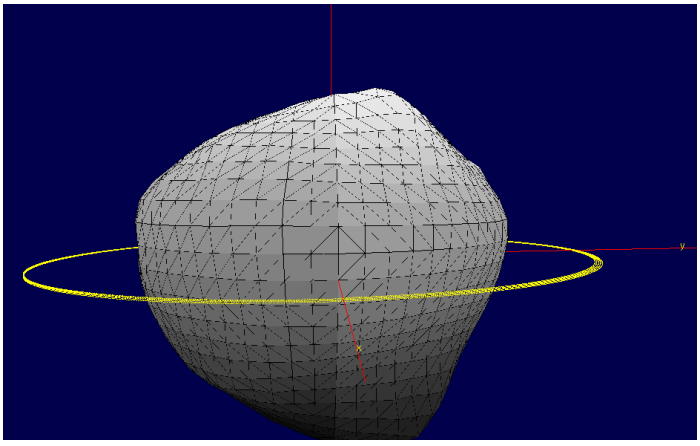
Σχήμα 4.6: Τιμές κλίσης i $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.48km$.



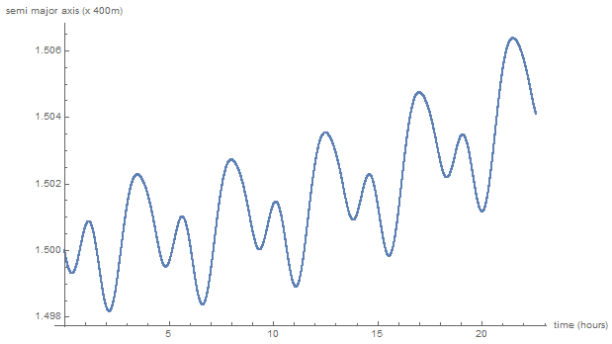
Σχήμα 4.7: Μήκος του ανερχόμενου σημείου Ω ($degrees$) συναρτήσει χρόνου ($hours$) για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.48km$.

Από τα διαγράμματα 4.4 εως 4.7 γίνεται αντιληπτό ότι σε μια τόσο κοντινή απόσταση $(0.48km)$ **από τον αστεροειδή τα τροχιακά στοιχεία παρουσιάζουν έντονες μεταβολές και η τροχιά του σώματος είναι ελλειπτική.**

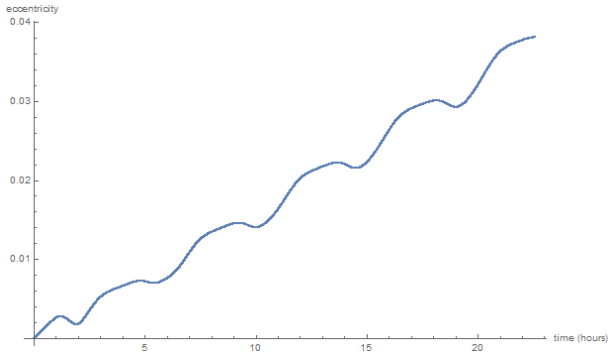
Όμοια με την προηγούμενη περίπτωση τα διαγράμματα 4.9 εως 4.12 δείχουν ότι και σε απόσταση $0.6km$ **τα τροχιακά χαρακτηριστικά της κίνησης παρουσιάζουν έντονες μεταβολές. Για να ληφθεί μια πιο ακριβής εικόνα της κίνησης κρίθηκε απαραίτητη η αριθμητική ολοκλήρωση των τροχιών για πολύ μεγαλύτερο χρονικό διάστημα (λήψη 200 χιλιάδων σημείων τροχιάς).**



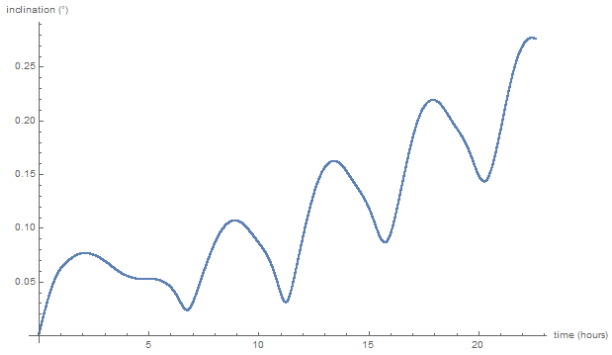
Σχήμα 4.8: Η τροχιά του σώματος όπως την αντιλαμβάνεται ο αδρανειακός παρατηρητής για προσεγγιστικά κυκλική κίνηση με αρχικές συνθήκες ξεκινώντας από τη θέση $x = 0.6km$.



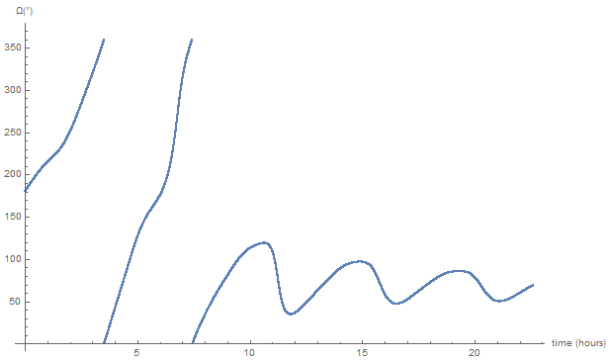
Σχήμα 4.9: Τιμές κύριου ημιάξονα $\alpha(\times 400m)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.6km$.



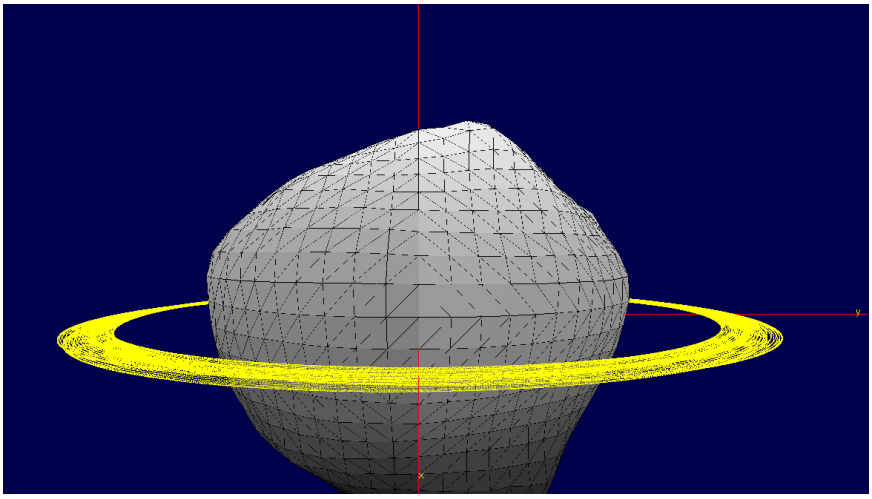
Σχήμα 4.10: Τιμές εκκεντρότητας e συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.6km$.



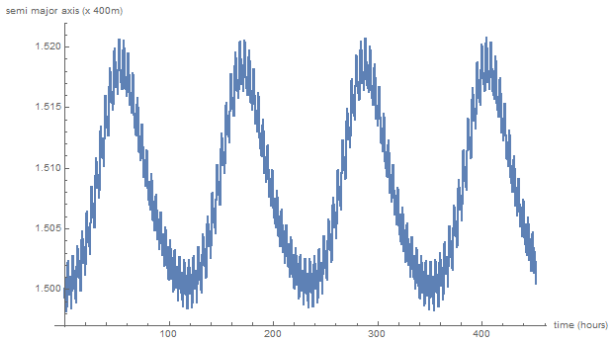
Σχήμα 4.11: Τιμές κλίσης i $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.6km$.



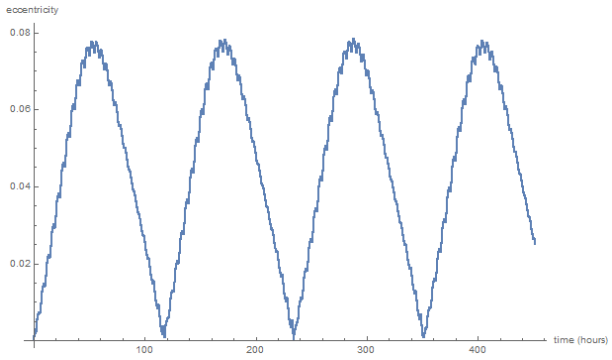
Σχήμα 4.12: Μήκος του ανερχόμενου σημείου Ω $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.6km$.



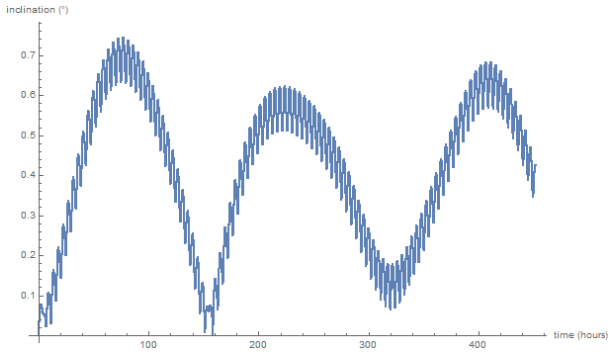
Σχήμα 4.13: Η τροχιά του σώματος όπως την αντιλαμβάνεται ο αδρανειακός παρατηρητής για προσεγγιστικά κυκλική κίνηση με αρχικές συνθήκες ξεκινώντας από τη θέση $x = 0.6km$ (200 χιλιάδες σημεία).



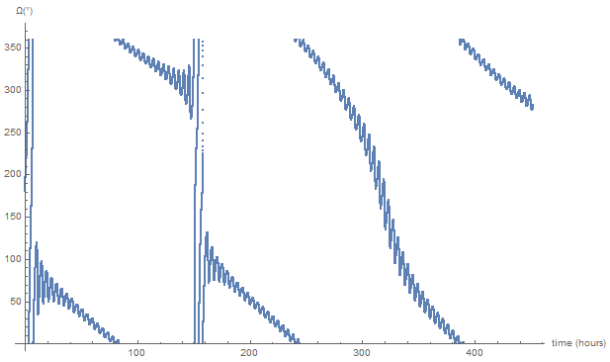
Σχήμα 4.14: Τιμές κύριου ημιάξονα $\alpha(\times 400m)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.6km$.



Σχήμα 4.15: Τιμές εκκεντρότητας e συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.6km$.



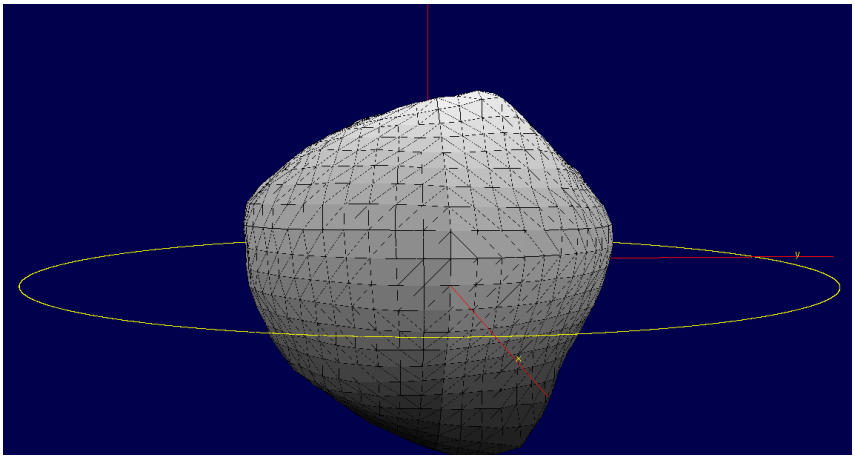
Σχήμα 4.16: Τιμές κλίσης i $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.6km$.



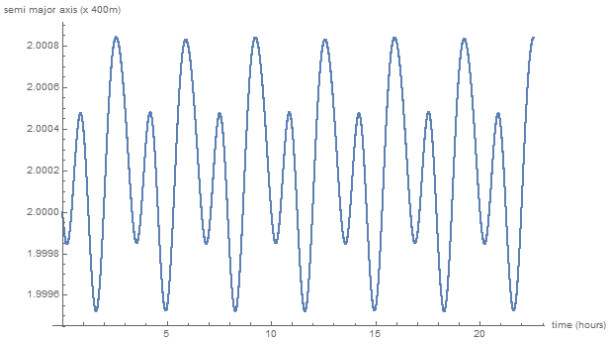
Σχήμα 4.17: Μήκος του ανερχόμενου σημείου Ω ($degrees$) συναρτήσει χρόνου ($hours$) για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.6km$.

Εξετάζοντας λοιπόν την κίνηση σε ένα αρκετά μεγαλύτερο χρόνικό διάστημα, λαμβάνουμε μια πιο σαφή εικόνα. Αρχικά, μελετώντας την μεταβολή του κύριου ημιάξονα της τροχιάς (Σχήμα 4.14) φαίνεται μια περιοδική αυξομοίωση, σταθερής περιόδου, περίπου ίσης με $240hours$, κατά την οποία ο κύριος ημιάξονας α, διακυμαίνεται μεταξύ των τιμών $0.59km(1.495 \times 400m)$ εώς $0.608km$ $(1.520 \times 400m)$. Όσον αφορά την εκκεντρότητα (Σχήμα 4.15), η μεταβολή της επίσης είναι περιοδική, με σταθερή περίοδο περίπου ίση με $118hours$ και παρουσιάζει μέγιστη τιμή περίπου 0.079 . Η κλίση της τροχιάς i (Σχήμα 4.16) φαίνεται να μεταβάλεται με μη περιοδικό τρόπο, ενώ η μέγιστη τιμή της που υπολογίστηκε μέσα στο χρονικό διάστημα των ολοκληρώσεων μας ήταν 0.74 $degrees$. Γίνεται επομένως αντιληπτό, ότι είναι πολύ δύσκολο σε αυτή την περιοχή να λάβουμε μια κυκλική τροχιά.

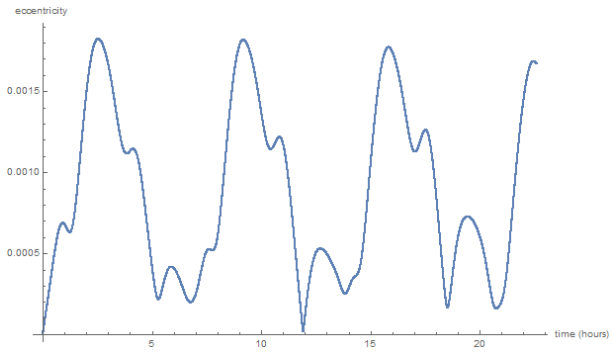
Στη συνέχεια, προχωρήσαμε σε παρόμοιους υπολογισμούς για τις θέσεις $\Gamma(0.8km, 0, 0)$ και $\Delta(1.18km, 0, 0)$. Το σημείο $\Delta(1.18km, 0, 0)$ βρίσκεται στην περιοχή που σύμφωνα με τα παρατηρησιακά δεδομένα περιφέρεται ο δορυφόρος του Δίδυμου.



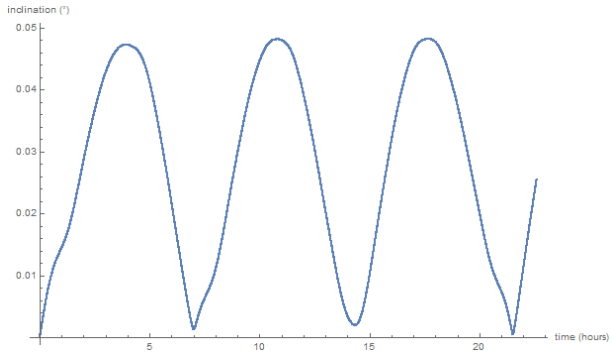
Σχήμα 4.18: Η τροχιά του σώματος όπως την αντιλαμβάνεται ο αδρανειακός παρατηρητής για προσεγγιστικά κυκλική κίνηση με αρχικές συνθήκες ξεκινώντας από τη θέση $x = 0.8km$ (10 χιλιάδες σημεία).



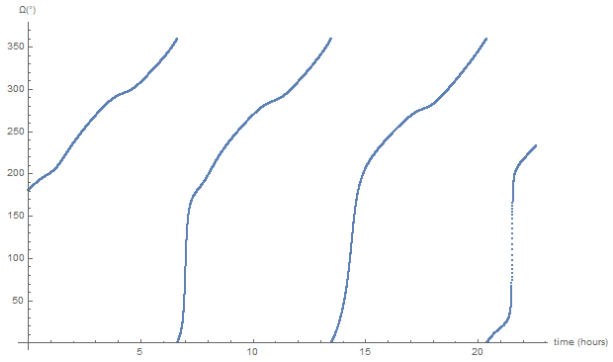
Σχήμα 4.19: Τιμές κύριου ημιάξονα $\alpha(\times 400m)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



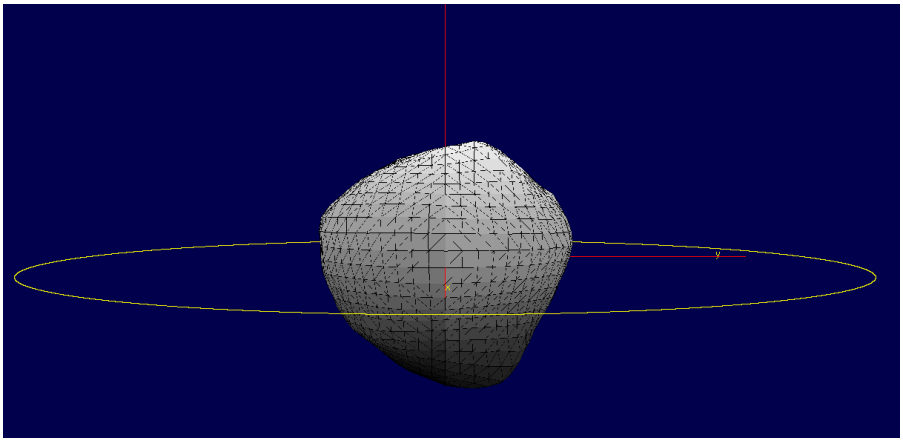
Σχήμα 4.20: Τιμές εκκεντρότητας e συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



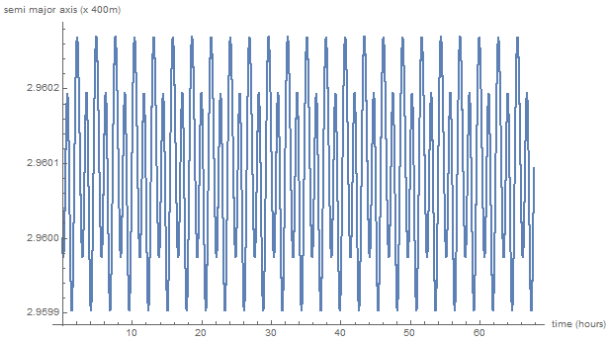
Σχήμα 4.21: Τιμές κλίσης i $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



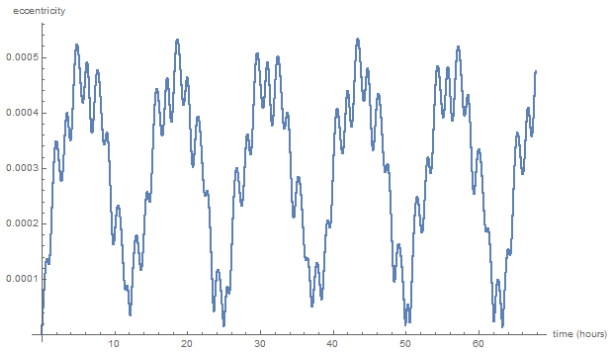
Σχήμα 4.22: Μήκος του ανερχόμενου σημείου Ω $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



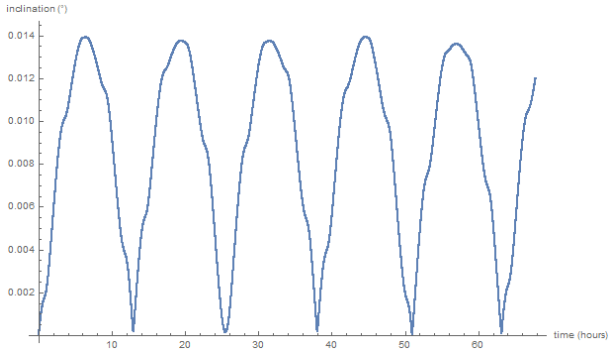
Σχήμα 4.23: Η τροχιά του σώματος όπως την αντιλαμβάνεται ο αδρανειακός παρατηρητής για προσεγγιστικά κυκλική κίνηση με αρχικές συνθήκες ξεκινώντας από τη θέση $x = 1.18km$ (30 χιλιάδες σημεία).



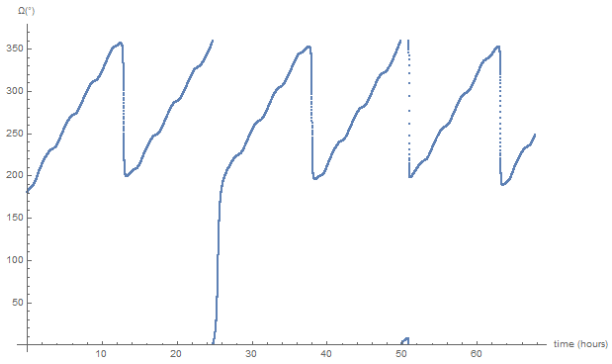
Σχήμα 4.24: Τιμές κύριου ημιάξονα $\alpha(\times 400m)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



Σχήμα 4.25: Τιμές εκκεντρότητας e συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



Σχήμα 4.26: Τιμές κλίσης i $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



Σχήμα 4.27: Μήκος του ανερχόμενου σημείου Ω $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.

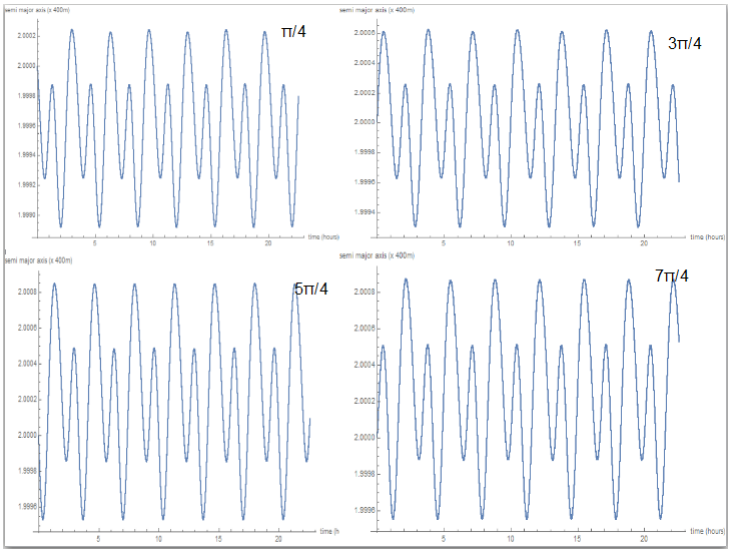
Εξετάζοντας τις κατα προσέγγιση κυκλικές τροχιές σε μεγαλύτερες αποστάσεις από τον αστεροειδή, παρατηρούμε ότι τα τροχιακά χαρακτηρηστικά των κινήσεων παρουσιάζουν πλέον ελάχιστες διακυμάνσεις. Στην περίπτωση που το σώμα ξεκίνησε από το σημείο $\Gamma(0.8km, 0, 0)$ η τροχιά του προσεγγίζει σε πολύ μεγαλύτερο βαθμό μια κυκλική κίνηση γύρω από τον Δίδυμο, που θα έκανε ένα σώμα με τις ίδιες αρχικές συνθήκες σε ένα αδρανειακό σύστημα.

Όπως βλέπουμε στο Σχήμα 4.19 οι σταθερά περιοδικές μεταβολές του κύριου ημιάξονα περιορίζονται στο τεταρτο δεκαδικό ψηφίο, επομένως μπορούν να θεωρηθούν αμελητέες. Ακόμη, στο σχήμα 2.20 φαίνεται ότι η εκκεντρότητα παρουσιάζει πάρα πολύ μικρές τιμές που προσεγγίζουν το μηδέν και επιβεβαιώνουν μια κυκλική τροχιά. Η κλίση της τροχιάς μεταβάλλεται περιοδικά με σταθερή περίοδο και μεταξύ των τιμών **0** και 0.05 $degrees$.

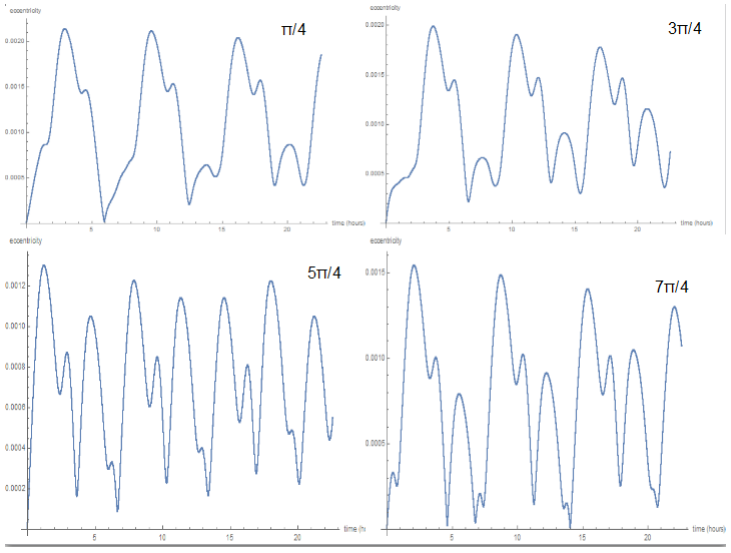
Παρόμοια εικόνα λάβαμε για την κίνηση του σώματος, όταν αυτό ξεκινά από την πιο απομακρυσμένη θέση $\Delta(1.18km, 0, 0)$ (υπολογίστηκαν **30** χιλιάδες σημεία τροχιάς). Οι μεταβολές των τροχιακών μεγεθών είναι εξίσου αμελητέες, γεγονός που επιβεβαιώνει ότι, καθώς απομακρυνόμαστε από τον αστεροειδή, η βαρυτική έλξη που ασκεί ο Δίδυμος σε σώμα προσεγγίζει εκείνη που θα ασκούσε ένας τέλεια σφαιρικός αστεροειδής.

4.3.2 Αλλαγή σχετικού προσανατολισμού

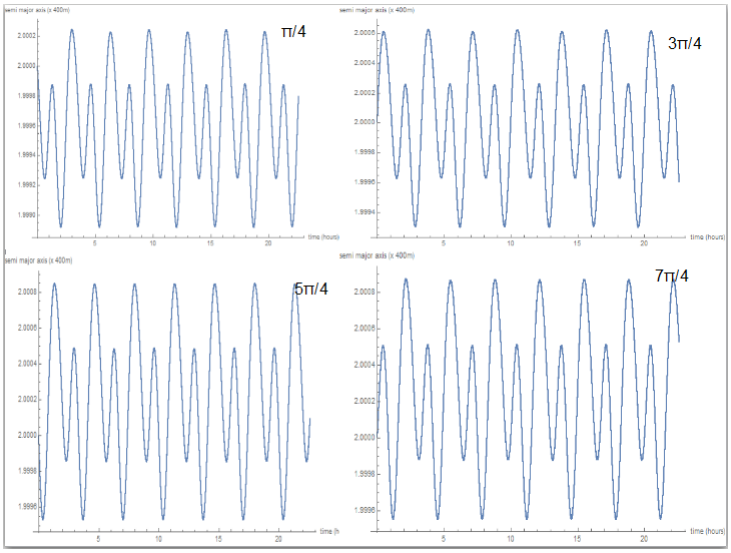
Σε αυτήν την παράγραφο οι τροχιές που υπολογίστηκαν λαμβάνουν υπόψιν τον σχετικό προσανατολισμό του αστεροειδή σε σχέση με το σώμα. Με βάση μια γωνία $theta$ περιστρέψαμε τον αστεροειδή περί τον άξονα z.Το σώμα τοποθετήθηκε στη θέση $\Gamma(0.8km, 0, 0)$ με κατάλληλη συνιστώσα ταχύτητας κατά τον άξονα y για να πραγματοποιήσει κατά προσεγγιση κυκλική κίνηση σε αδρανειακό σύστημα, όμοια με τις προηγούμενες περιπτώσεις. Οι γωνίες που λήφθηκαν υπόψιν είναι οι: $\pi/4, 3\pi/4, 5\pi/4, 7\pi/4$. Παρακάτω παρουσιάζονται τα διαγράμματα με τα τροχιακά στοιχεία για κάθε περίπτωση.



Σχήμα 4.28: Διαγράμματα μεταβολής του κύριου ημιάξονα α συναρτήσει του χρόνου για διαφορετικές γωνίες $theta$ των κατά προσέγγιση κυκλικών τροχιών από το σημείο Γ.



Σχήμα 4.29: Διαγράμματα μεταβολής της εκκεντρότητας e συναρτήσει του χρόνου για διαφορετικές γωνίες $theta$ των κατά προσέγγιση κυκλικών τροχιών από το σημείο Γ.

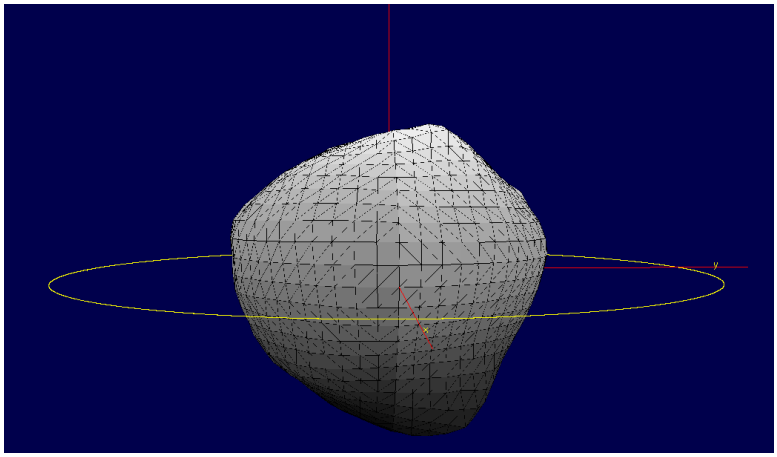


Σχήμα 4.30: Διαγράμματα της κλίσης i συναρτήσει του χρόνου για διαφορετικές γωνίες $theta$ των κατά προσέγγιση κυκλικών τροχιών από το σημείο Γ.

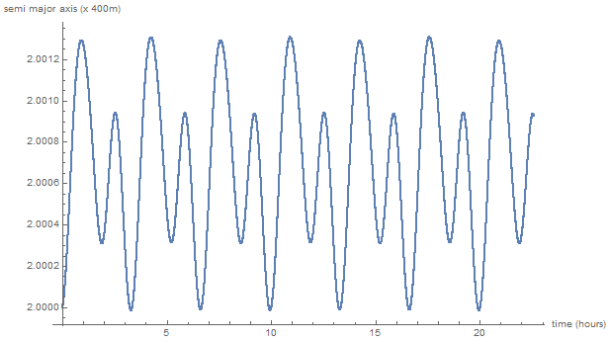
Συγκρίνοντας τα παραπάνω μεγέθη για κάθε περίπτωση, είναι σαφές ότι οποιεσδήποτε διακυμάσεις λόγω διαφορετικού προσανατολισμού είναι αμελητέες και η τροχιά του σώματος εξακολουθεί να προσεγγίζει εκείνη της κυκλικής γύρω από τον Δίδυμο.

4.3.3 Υπολογισμός τροχιάς με βάση τα Κεπλέρια στοιχεία

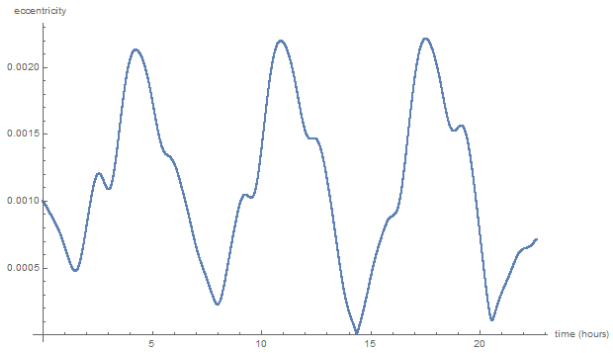
Σε αυτό το σημείο, παρουσιάζεται η τροχιά που υπολογίστηκε ξεκινώντας από το σημείο $\Gamma(0.8km, 0, 0)$ **με εκκεντρότητα σχεδόν μηδενική και κλίση μηδέν. Συγκεκριμένα η τιμές που δόθηκαν ήταν** $\alpha = 2 = 0.8km$, $e = 0.001$, $i = 0degrees$, $\Omega = 0degrees$. **Η τροχιά που πήραμε είναι κατά προσέγγιση κυκλική και δεν παρουσιάζει κάποια ουσιαστική διαφοροποίηση από αυτήν που είδαμε στις προηγούμενες περιπτωσεις(βλ. Σχήματα 4.18 εως 4.22):**



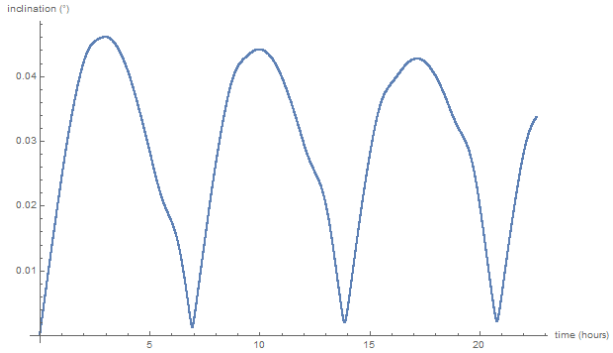
Σχήμα 4.31: Η τροχιά του σώματος όπως την αντιλαμβάνεται ο αδρανειακός παρατηρητής για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$ (10 χιλιάδες σημεία).



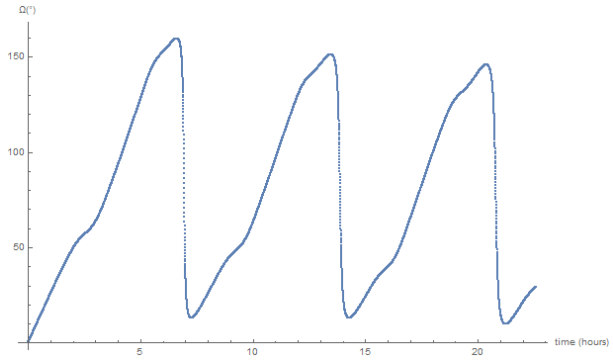
Σχήμα 4.32: Τιμές κύριου ημιάξονα $\alpha(\times 400m)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



Σχήμα 4.33: Τιμές εκκεντρότητας e συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



Σχήμα 4.34: Τιμές κλίσης i $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.



Σχήμα 4.35: Μήκος του ανερχόμενου σημείου Ω $(degrees)$ συναρτήσει χρόνου $(hours)$ για προσεγγιστικά κυκλική κίνηση ξεκινώντας από τη θέση $x = 0.8km$.

Κεφάλαιο 5

Συμπεράσματα

Μέσα από τη μοντελοποίηση του αστεροειδή 65803 Δίδυμος, ως ένα ομοιογενές σύνολο κυψελίδων, επιτεύχθει με μεγάλη ακρίβεια η αριθμητική ολοκλήρωση μιας σειράς τροχιών που εκτελεί σώμα υπο την επίδραση της έλξης του αστεροειδή. Η μελέτη επικεντρώθηκε συγκεκριμένα στην εύρεση κατα προσέγγιση κυκλικών τροχιών γύρω από τον Δίδυμο. Από τα αποτελέσματα αυτών των υπολογισμών φάνηκε πως τέτοιες τροχιές είναι δεν είναι εφικτό να πραγματοποιηθούν σε πολυ κοντινές αποστάσεις από τον αστεροειδή. Το ανώμαλο του σχήμα προκαλεί διαταραχές στο βαρυτικό πεδίο σε αυτές τις περιοχές, με αποτέλεσμα έντονες μεταβολές στα τροχιακά τους χαρακτηριστικά. Ωστόσο, σε μεγαλύτερες αποστάσεις ήταν εφικτή η εύρεση τροχιών που προσέγγισαν σε πολύ μεγάλο βαθμό κυκλικές.

Όσον αφορά την μελέτη του βαρυτικού πεδίου του Δίδυμου, έγινε εμφανές ότι η έλλειψη συμμετριών και το ανώμαλο σχήμα του, καθιστούν δύσκολη την εύρεση σημείων ισορροπίας. Μελλοντικά, θα μπορούσαν να πραγματοποιηθούν πιο λεπτομερείς υπολογισμοί , με μεγαλύτερη αριθμητική ακρίβεια για τον εντοπισμό ενδεχόμενων σημείων ισορροπίας.

Αξίζει τέλος, να σημειωθεί ότι με κατάλληλες τροποποιήσεις στον κώδικα αυτής της εργασίας θα ήταν δυνατόν, δοθέντων αντίστοιχων δεδομένων χαρτογράφησης να γίνουν άμεσα ανάλογοι υπολογισμοί για άλλους αστεροειδείς.

Παράρτημα

Παράρτημα 1. Κώδικας *Linear_Interpolation.cpp*

```
#include <iostream>
#include <fstream>
#include <vector>
#include <math.h>
#include <cmath>
using namespace std;

#define FILENAME1 "didymos1162_vertices.txt"
#define FILENAME2 "didymos1162_indices.txt"
#define COLS 3
int N,M,L; // number of vertices and indices
vector< vector <double> > vec;
vector<vector<int> > ind;
vector <double> adjacent_distances;


//Coordinates of vertices
void vertexdata(vector<vector<double> > &vec, int *N){

    /* FILE1 vec */
    fstream file;
    vector <double> rowVector(COLS);                    // vector to add into 'array' (represents a row)
    int row = 0;                                        // Row counter

    file.open(FILENAME1 , ios::in);                     // Open file1
    if (file.is_open()) {

        cout << "vec file correctly opened" << endl;


        while (file.good()) {                           // check for errors
            vec.push_back(rowVector);                   // add a new row
            for (int col=0; col<COLS; col++) {
                file >> vec[row][col];                  // fill the row with col elements
            }
            row++;                                      // number of rows in my file a.k.a number of vec
        }
    }
    else cout << "Unable to open vec file" << endl;
    file.close();
   *N = row;
}

//Adjacent vertices
void indexdata(vector<vector<int> > &ind, int *M){

    /* FILE2 INDICES */
```

```
    fstream file;
    vector <int> rowVector(COLS);                          // vector to add into 'array' (represents a row)
    int row = 0;                                           // Row counter

    file.open(FILENAME2 , ios::in);                        // Open file2
    if (file.is_open()) {

        cout << "Indices file correctly opened" << endl;

        while (file.good()) {                              // check for errors
            ind.push_back(rowVector);                      // add a new row
            for (int col=0; col<COLS; col++) {
                file >> ind[row][col];                     // fill the row with column elements
            }
            row++;                                         // number of rows in my file a.k.a number of indices
        }
    }
    else cout << "Unable to open indices file" << endl;
    file.close();
    *M = row;
}

//Point distance
double Distance(int n, int m){
        if (n == m)
                return 0;
        return sqrt( (vec[m][0] - vec[n][0])*(vec[m][0] - vec[n][0]) +
        (vec[m][1] - vec[n][1])*(vec[m][1] - vec[n][1]) + (vec[m][2] - vec[n][2])*(vec[m][2] - vec[n][2]) );
}

//Calculate the distances of all adjacent vertices
void find_adjacent_distances(vector <double> &adjacent_distances, int *L){
        int j =0;
        for (int i = 0; i < M; i++){
        adjacent_distances.push_back(1);
        adjacent_distances[j] = Distance(ind[i][0], ind[i][1]);
        j++;
        adjacent_distances.push_back(1);
        adjacent_distances[j] = Distance(ind[i][1], ind[i][2]);
        j++;
        adjacent_distances.push_back(1);
        adjacent_distances[j] =  Distance(ind[i][2], ind[i][0]);
        j++;
        }
        *L = j - 1;
}

//Minimum point distance
double minDistance(int i, int N){
        double minDist;
        if (i < N - 1) minDist = Distance(i, i+1);
        else minDist = Distance(i, i-1);
        for (int j = 0; j < N; j++){
                if (j==i) continue;
                double dist = Distance(i, j);
                if (dist < minDist)     minDist = dist;
        }
        return minDist;
}

//Max value of minimum distances
double max_minDistance(int N){
        double dmax = 0;
        for (int i = 0; i < N; i++){
                double d = minDistance(i, N);
```

```
                if (d > dmax){
                        dmax = d;
                }
        }
        return dmax;
}

//Reverse function takes as input the distance and finds the two elements
void distance_to_coordinates(double distance, int N , int *max1, int *max2){
        for(int i = 0; i < N; i++){
                for (int j = 1; j < N; j++){
                        if (Distance(i,j) == distance){
                                *max1 = i;
                                *max2 = j;

                                i = N;
                                break;
                        }
                }
        }
}

//Find the middle point of the max minimum distance and add that value
//as the new last element of the vector : vec
void linear_interpolation(int i , int j , int &N){
        vector < double > rowVector(COLS);
        vec.push_back(rowVector);
        vec[N][0] = (vec[i][0] + vec[j][0])/2;
        vec[N][1] = (vec[i][1] + vec[j][1])/2;
        vec[N][2] = (vec[i][2] + vec[j][2])/2;
        N++;

}// after real N = N - 1

//Output in original file new interpolated data
void output(int N){
        fstream file(FILENAME1);
        if (file.is_open()){
                for (int i = 0; i < N; i++){
                        file  << vec[i][0] << "   " << vec[i][1] << "   " << vec[i][2] << endl;
                }
                file.close();
        }

}

//Main function
int main(int argc, char** argv) {
        int max1=0 , max2=0;
        double dmax;
        vertexdata(vec, &N);
        indexdata(ind, &M);
        dmax = max_minDistance(N);;
    distance_to_coordinates(dmax, N, &max1, &max2);

        //Interpolate max distance
        linear_interpolation(max1,max2,N);

        //Calculate adjacent distances
        find_adjacent_distances(adjacent_distances, &L);
        int n  = N;
        //Continue to interpolate with new points any time the distance between two vertices
        //is greater than dmax/2
        for(int i =0; i < L; i++){
                if (adjacent_distances[i] > dmax/2){
```

```
                distance_to_coordinates(adjacent_distances[i], n, &max1, &max2);
                linear_interpolation(max1,max2,N);
                }
        }

        //Add new vertices' coordinates in the original data
        output(N);
        cout  << "new max min distance is    " <<  max_minDistance(N);


}
```

Παράρτημα 2. Κώδικας *computational_space.cpp*

```
#include <iostream>
#include <fstream>
#include <vector>
#include <math.h>
#include <cmath>
using namespace std;

#define FILENAME1 "didymos1162_vertices.txt"
#define FILENAME2 "coefficients.txt"
#define FILENAME3 "voxel_coordinates.txt"
#define COLS 3

//global variables & basic memory storage unit
int K, L, M, N;
double VoxelL;
double rminx , rminy, rminz, rmaxx, rmaxy, rmaxz;
double startx = 0, starty = 0, startz = 0;
vector< vector <double> > vec;

//Basic operations between (x,y,z) , (i,j,k) and voxel_index parameters for each voxel
//find (i,j,k) from (x,y,z)
void VoxelGridCoord(double x, double y, double z, int *i, int *j, int *k){
                *i = int ((x-rminx)/VoxelL) + 1;
                *j = int ((y-rminy)/VoxelL) + 1;
                *k = int ((z-rminz)/VoxelL) + 1;
        }

// find voxel_index from (i,j,k)
int VoxelIndex(int i, int j, int k){
                return (k - 1)*(K*L) + (K*(j - 1)) + i;
        }

// find (i,j,k) from voxel_index
int  VoxelGridPos(int voxel_index, int *i, int *j, int *k){
                if (voxel_index < 1 || voxel_index > K*L*M)
                return 0;
                *k = (voxel_index - 1 )/(K*L) + 1 ;
                int dl = voxel_index - (*k - 1)*K*L;
                *j = (dl-1)/K + 1;
                *i = (dl-1)%K + 1;
                return 1;
        }

// find (x,y,z) from (i,j,k)
void CartesianGridCoord(int i, int j, int k, double *x, double *y, double *z){
                *x = i * VoxelL - 0.5 * VoxelL + rminx ;
                *y = j * VoxelL - 0.5 * VoxelL + rminy ;
                *z = k * VoxelL - 0.5 * VoxelL + rminz ;
        }
```

```
// Input FUNCTION
void vertexdata(vector< vector <double> > &vec , int *N){

   /* FILE1 VERTICES */
   fstream file;
   vector <double> rowVector(COLS);              // vector to add into 'array' (represents a row)
   int row = 0;                                  // Row counter

   file.open(FILENAME1 , ios::in);               // Open file1
   if (file.is_open()) {
       cout << "File correctly opened" << endl;
       while (file.good()) {                     // check for errors
           vec.push_back(rowVector);             // add a new row
           for (int col=0; col<COLS; col++) {
               file >> vec[row][col];            // fill the row with col elements
           }
           row++;                                // number of rows in my file a.k.a number of vertices
       }
   }
   else cout << "Unable to open file" << endl;
   file.close();
   *N = row;                                                    // pass row's value in N variable
}

// find max and min values from the vector that contains the coord of the vertices
void GetMinMax(int N, double *mx, double *my, double *mz, double *Mx, double *My, double *Mz){
        double min_x = vec[0][0], min_y = vec[0][1], min_z = vec[0][2];
        double max_x = vec[0][0], max_y = vec[0][1], max_z = vec[0][2];

        for (int i = 1; i < N; i++){                            // min-max search 2D vector
                if (min_x > vec[i][0]) min_x = vec[i][0];
                if (min_y > vec[i][1]) min_y = vec[i][1];
                if (min_z > vec[i][2]) min_z = vec[i][2];
                if (max_x < vec[i][0]) max_x = vec[i][0];
                if (max_y < vec[i][1]) max_y = vec[i][1];
                if (max_z < vec[i][2]) max_z = vec[i][2];
        }

        *mx = min_x;
        *my = min_y;
        *mz = min_z;
        *Mx = max_x;
        *My = max_y;
        *Mz = max_z;
}

//distance between points with indices n , m
double Distance(int n, int m){
        if (n == m)
                return 0;
        return sqrt (pow(vec[m][0] - vec[n][0],2) +
        pow(vec[m][1] - vec[n][1],2) + pow(vec[m][2] - vec[n][2], 2) );
}

// minimum point distance
double minDistance(int i, int N){
        double minDist;
        if (i < N - 1 )  minDist = Distance(i, i+1);
        else  minDist = Distance(i, i-1);
        for (int j = 0; j < N; j++){
                if (j==i) continue;
                double dist = Distance(i, j);
                if (dist < minDist)  minDist = dist;
        }
        return minDist;
```

```
}

// max value of min distances
double max_minDistance(int N){
        double dmax = 0;
        for (int i = 0; i < N; i++){
                double d = minDistance(i, N);
                if (d > dmax) dmax =d;
        }
        return dmax;
}

//CALCULATE  borders  , K , L , M , AND THE CUBIC VOXEL'S SIDE
int GetWorkingGrid(int k, int rows){
        double min_x, max_x, min_y, max_y, min_z, max_z;
        GetMinMax(rows, &min_x, &min_y, &min_z, &max_x, &max_y, &max_z);
        if (k < 3) return 0; // grid too small
        K = k;
        VoxelL = (max_x - min_x)/(K-2)  ;
        rminx = min_x - VoxelL; rminy = min_y - VoxelL; rminz = min_z - VoxelL;
        rmaxx = max_x + VoxelL; rmaxy = max_y + VoxelL; rmaxz = max_z + VoxelL;
        L = int((rmaxy - rminy)/ VoxelL) + 1 ;
        M = int((rmaxz - rminz)/ VoxelL) + 1 ;
        return 1;
}

//create the voxel model
int GetModelCoord(){
        cout << endl << "Calculating the cartesian coordinates of the INSIDE voxels..." << endl;

        int comp_space = K * L * M ;
        int *comp_space_voxels;
        int *InnerVoxels;

        comp_space_voxels = new int [comp_space + 1];
        InnerVoxels = new int [comp_space + 1];

        int i, j, k;
        double x0, y0, z0;

        for (int a = 0; a <= comp_space; a++){
                InnerVoxels[a] = 0;
                comp_space_voxels[a] = 0; //empty
        }

        for (int b = 0; b < N; b++){
        VoxelGridCoord(vec[b][0], vec[b][1], vec[b][2], &i, &j, &k);
        comp_space_voxels[VoxelIndex(i,j,k)] = 2; //crust
        }

        //coordinates of the start
        VoxelGridCoord(startx, starty, startz, &i, &j, &k);
        int n = VoxelIndex(i,j,k);
        //cout << endl << n << "  " << i << "  " << j << "  " << k << "   "  << "CHECK" << endl;
        int InVoxelCounter = 1;
        int VoxelCounter = 1;
        VoxelGridPos(n, &i, &j, &k);

        InnerVoxels[InVoxelCounter] = n;
        comp_space_voxels[VoxelCounter] = 1;  // Checked and into the object ,filled

do{
        if (comp_space_voxels[n] == 3){
                comp_space_voxels[n] = 1;  // if the n element is marked as checked then mark it as filled
        }
```

```
//left
if (comp_space_voxels[VoxelIndex(i-1, j, k)] == 0){
        InVoxelCounter++;
        InnerVoxels[InVoxelCounter] = VoxelIndex(i-1, j, k);
        comp_space_voxels[VoxelIndex(i-1, j, k)] = 3; // checked
        }


//right
if (comp_space_voxels[VoxelIndex(i+1, j, k)] == 0){
        InVoxelCounter++;
        InnerVoxels[InVoxelCounter] = VoxelIndex(i+1, j, k);
        comp_space_voxels[VoxelIndex(i+1, j, k)] = 3; // checked
        }

//front
if (comp_space_voxels[VoxelIndex(i, j-1, k)] == 0){
        InVoxelCounter++;
        InnerVoxels[InVoxelCounter] = VoxelIndex(i, j-1, k);
        comp_space_voxels[VoxelIndex(i, j-1, k)] = 3; // checked
        }

//back
if (comp_space_voxels[VoxelIndex(i, j+1, k)] == 0){
        InVoxelCounter++;
        InnerVoxels[InVoxelCounter] = VoxelIndex(i, j+1, k);
        comp_space_voxels[VoxelIndex(i, j+1, k)] = 3; // checked
        }

//bottom
if (comp_space_voxels[VoxelIndex(i, j, k-1)] == 0){
        InVoxelCounter++;
        InnerVoxels[InVoxelCounter] = VoxelIndex(i, j, k-1);
        comp_space_voxels[VoxelIndex(i, j, k-1)] = 3; // checked
        }

//top
if (comp_space_voxels[VoxelIndex(i, j, k+1)] == 0){
        InVoxelCounter++;
        InnerVoxels[InVoxelCounter] = VoxelIndex(i, j, k+1);
        comp_space_voxels[VoxelIndex(i, j, k+1)] = 3; // checked
        }

VoxelCounter++;
n = InnerVoxels[VoxelCounter];
VoxelGridPos(n, &i, &j, &k);
//cout << endl <<   " i = " << i << "    j = " << j << "    k = "   << k;
if (i <= 1 || i >= K   || j <= 1 || j >= L   || k <= 1 || k >= M  ){
        cout << endl << " Voxel side is " << VoxelL;
        cout << endl << "FAILURE. Calculating voxel of greater size." ;
        delete InnerVoxels;
        delete comp_space_voxels;
        return 1;
        break;
        }

}while(InnerVoxels[VoxelCounter]!=0);

ofstream myfile2(FILENAME2);
if (myfile2.is_open()){
        myfile2 << K << '\t' << L << '\t' << M << '\t' << VoxelL ;
        myfile2.close();
        cout << endl << " K  L M VoxelL values succesfully stored in file:" << FILENAME2
```

```
<< endl;
        }

ofstream myfile3(FILENAME3);
if (myfile3.is_open()){
        for (int l = 1; l <= comp_space; l++){
                if (comp_space_voxels[l] == 0){ continue; } //don't keep empty voxels
                VoxelGridPos(l, &i, &j, &k);
                CartesianGridCoord(i, j, k, &x0, &y0, &z0);
                myfile3 << x0 << "    " << y0 << "    " << z0 << "    " << comp_space_voxels[l] << endl;
        }

        myfile3.close();
        cout << "The cartesian coordinates of the center of each  voxel "
                        "are now stored in the file:" << FILENAME3 << endl;
}

cout << "Voxel side is: " << VoxelL << "    " << "SUCCESS!" << endl;
return 0;
}

// main function
int main (int argc, char* argv[]){

double min_x , max_x, min_y, max_y, min_z, max_z;
vertexdata(vec, &N);
GetMinMax( N, &min_x, &min_y, &min_z, &max_x, &max_y, &max_z );
double MaxMinDistance = max_minDistance(N);
int h = ceil(abs(max_x-min_x)/MaxMinDistance);
GetWorkingGrid(h,N);
int iteration = 0;
while (GetModelCoord()==1){
        cout << endl << " K before new calculation is   " << K ;
        GetWorkingGrid(K-1, N);
        cout << endl << " K = " << K ;
        cout << endl << "iteration number: " << iteration++;
        }

return 0;
}
```

Παράρτημα 3. Κώδικας *inertia_axes_coord.cpp*

```
/* PROGRAM TO SHIFT COORDINATES OF THE SYSTEM TO MAIN INERTIA AXES COORDINATES */
#include <iostream>
#include <cmath>
#include <math.h>
#include <fstream>
#include <complex>
#include <vector>
#include  "Eigen/Eigenvalues"
#include  "Eigen/Dense"

using namespace std;
using namespace Eigen;


#define accuracy 10e-9 //accuracy of calculations for center of mass
#define FILENAME1  "voxel_coordinates.txt"
#define FILENAME2  "shifted_V_coord_w_mass.txt"

int N;                  //Number of asteroid voxels (count rows in coord_file)
double Mass;
```

```
// array that contains the x,y,z coordinates of each voxel
//of the asteroid (only read 3 first columns (x,y,z))
vector< vector <double> > voxels;

/* PART A of the program: Calculating the center of mass of the asteroid
  and correcting the (x,y,z) coordinates */

//function to find the coordinates of the center of mass
void calculate_center_mass(double *mcx, double *mcy, double *mcz, int Vox){
        *mcx=0;
        *mcy=0;
        *mcz=0;
        for (int i = 0; i < Vox; i++){
    *mcx += Mass * voxels[i][0];
    *mcy += Mass * voxels[i][1];
    *mcz += Mass * voxels[i][2];
        }
}

//function to correct the x,y,z coordinates with respect to center of mass
void correct_Center(double mcx, double mcy, double mcz, int vox){
        for (int i = 0; i < vox; i++){
                voxels[i][0] -= mcx;
                voxels[i][1] -= mcy;
                voxels[i][2] -= mcz;
        }
}

//function that reads the coordinates of the asteroid voxels and counts
//the number of the asteroid voxels
void voxeldata(vector< vector <double> > &voxels, int *N){
   fstream file;
   vector <double> rowVector(4);                    // vector to add into 'array' (represents a row)
   int row = 0;                                     // Row counter

   file.open(FILENAME1 , ios::in);                  // Open file1
   if (file.is_open()) {
       while (file.good()){                         // check for errors
           voxels.push_back(rowVector);             // add a new row
           for (int col=0; col<4; col++) {
               file >> voxels[row][col];            // fill the row with col elements
           }
           row++;                                   // number of rows in my file a.k.a number of voxels
       }
   }
   file.close();

   *N = row - 1;                                    // last row empty

}

void output(vector< vector <double> > voxels, double mass){  //output file
ofstream myfile2(FILENAME2);
if(myfile2.is_open()){
        for (int i = 0; i < N; i++){
                myfile2 << voxels[i][0] << "    " << voxels[i][1] << "    " << voxels[i][2]
                << "    " << voxels[i][3]  << "    " << mass << endl;
        }
        myfile2.close();
                cout << "Voxel information and its mass are now stored in the file:"
            << FILENAME2 << endl;
  }

}
//function shifting coordinates to center of mass
```

```
void shift_coord_to_cm(){
        double mcx = 0, mcy = 0, mcz = 0;                                   //initialize coordinates
        calculate_center_mass(&mcx, &mcy, &mcz, N);                         //calculate the initial

        cout << " Initial calculation of center of mass: " << endl;
        cout << "(" << mcx << " , " << mcy << " , " << mcz << ")" << endl;
        int iteration = 0;
        while (mcx > accuracy || mcy > accuracy || mcz > accuracy){  //correcting the coordinates of c.m.
                correct_Center(mcx, mcy, mcz, N);                           //applying corrections
                calculate_center_mass(&mcx, &mcy, &mcz, N);                 //re-calculating c.m. coordinates
                cout << "Iteration:  " << iteration << endl;
                cout << "Center of mass coordinates after correction: " << endl;
                cout << "(" << mcx << " , " << mcy << " , " << mcz << ")" << endl;
                iteration++;
        }

}

/* PART B of the program: Inertia axes correction */

//function rotating coordinates to the main inertia axes coordinate system
void rotate_to_m_axes(){
   //Inertia tensor
   MatrixXd J(3,3);
   J << MatrixXd::Zero(3,3);
   //calculating the inertia tensor
        for (int i = 0; i < N; i++){
                J(0,0) += Mass*(voxels[i][1]*voxels[i][1] + voxels[i][2]*voxels[i][2]);
                J(1,1) += Mass*(voxels[i][2]*voxels[i][2] + voxels[i][0]*voxels[i][0]);
                J(2,2) += Mass*(voxels[i][0]*voxels[i][0] + voxels[i][1]*voxels[i][1]);

                J(0,1) += Mass*(voxels[i][0]*voxels[i][1]);
                J(0,2) += Mass*(voxels[i][0]*voxels[i][2]);
                J(1,2) += Mass*(voxels[i][1]*voxels[i][2]);
        }
   J(1,0) = J(0,1);
   J(2,0) = J(0,2);
   J(2,1) = J(1,2);



  //calculate eigenvalues and eigenvectors

  EigenSolver <MatrixXd> es(J);

  //The Inertia tensor is real and symmetric so the eigenvalues will be real numbers

  //Eigenvalues
  double eig1 = (double)(es.eigenvalues()[0]).real();
  double eig2 = (double)(es.eigenvalues()[1]).real();
  double eig3 = (double)(es.eigenvalues()[2]).real();

  //Eigenvectors
  VectorXcd v1 = es.eigenvectors().col(0);
  VectorXcd v2 = es.eigenvectors().col(1);
  VectorXcd v3 = es.eigenvectors().col(2);

  //Real part of Eigenvectors
  double eigv1x = (double)v1(0).real();
  double eigv1y = (double)v1(1).real();
  double eigv1z = (double)v1(2).real();

  double eigv2x = (double)v2(0).real();
  double eigv2y = (double)v2(1).real();
  double eigv2z = (double)v2(2).real();
```

```
double eigv3x = (double)v3(0).real();
double eigv3y = (double)v3(1).real();
double eigv3z = (double)v3(2).real();

//complete Eigenvectors
Vector3d eigv1; eigv1 << eigv1x, eigv1y, eigv1z;
Vector3d eigv2; eigv2 << eigv2x, eigv2y, eigv2z;
Vector3d eigv3; eigv3 << eigv3x, eigv3y, eigv3z;

//Direction vectors of axes before rotation
Vector3d x0; x0 << 1, 0, 0;
Vector3d y0; y0 << 0, 1, 0;
Vector3d z0; z0 << 0, 0, 1;

//Rotation tensor
Matrix3d R, tR;

//calculate the angles between axes direction vectors and eigenvectors
//Rij = cos(k) = a.b/(|a||b|) , k in radians

//angle between x and x'
R(0,0) = x0.dot(eigv1)/( x0.norm()*eigv1.norm() );
//angle between x and y'
R(0,1) = x0.dot(eigv2)/( x0.norm()*eigv2.norm() );
//angle between x and z'
R(0,2) = x0.dot(eigv3)/( x0.norm()*eigv3.norm() );

//angle between y and x'
R(1,0) = y0.dot(eigv1)/( y0.norm()*eigv1.norm() );
//angle between y and y'
R(1,1) = y0.dot(eigv2)/( y0.norm()*eigv2.norm() );
//angle between y and z'
R(1,2) = y0.dot(eigv3)/( y0.norm()*eigv3.norm() );


//angle between z and x'
R(2,0) = z0.dot(eigv1)/( z0.norm()*eigv1.norm() );
//angle between z and y'
R(2,1) = z0.dot(eigv2)/( z0.norm()*eigv2.norm() );
//angle between z and z'
R(2,2) = z0.dot(eigv3)/( z0.norm()*eigv3.norm() );


//Rotate all points of the asteroid according to the previous angles
Vector3d r, rt;

for (int i = 0; i < N; i++){
        r(0) = voxels[i][0];
        r(1) = voxels[i][1];
        r(2) = voxels[i][2];
        rt = R * r;
        voxels[i][0] = rt(0);
        voxels[i][1] = rt(1);
        voxels[i][2] = rt(2);
}

//find transverse of R
tR = R.transpose();

cout << endl;
cout << "Eigenvalue1 = " << eig1 << endl;
cout << "Eigenvalue2 = " << eig2 << endl;
cout << "Eigenvalue3 = " << eig3 << endl;
```

```
  //find the max inertia axis
  if (eig1 > eig2 && eig1 > eig3){
          cout << "Axis x is the max inertia axis" << endl;
  }
  if (eig2 > eig1 && eig2 > eig3){
          cout << "Axis y is the max inertia axis" << endl;
  }
  if (eig3 > eig1 && eig3 > eig2){
          cout << "Axis z is the max inertia axis" << endl;
  }


  cout << "The first eigenvector of the Inertia tensor J is v1 : "  << "( "  <<
                  eigv1x << ", " << eigv1y << ", " << eigv1z << ")" <<endl;
  cout << "The second eigenvector of the Inertia tensor J is v2 : " << "( "  <<
                  eigv2x << ", " << eigv2y << ", " << eigv2z << ")" <<endl;
  cout << "The third eigenvector of the Inertia tensor J is v3 : "  << "( "  <<
                  eigv3x << ", " << eigv3y << ", " << eigv3z << ")" <<endl;

  //Verticality verification
  cout << "v1.v2 = "  << eigv1.dot(eigv2) << endl;
  cout << "v2.v3 = "  << eigv2.dot(eigv3) << endl;
  cout << "v3.v1 = "  << eigv3.dot(eigv1) << endl;


  //Save new coordinates in new file
  output(voxels, Mass);       //new file has 5 columns shifted(x,y,z) tag and mass = constant

}

//main function
int main(int argc, char* argv[]){
        voxeldata(voxels, &N);
        Mass = 1.0/N;                       //calculate mass for each voxel of the asteroid
        shift_coord_to_cm();
        cout <<"--------------------------- " << endl;
        cout << "voxel mass: " << Mass << endl;
        cout << "N: "    << N << endl;
        rotate_to_m_axes();
}
```

Παράρτημα 4. Κώδικας *num_calc.cpp*

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <math.h>
#include <vector>

using namespace std;

#define FILENAME1 "shifted_V_coord_w_mass.txt"  //input file with the coordinates of the asteroid voxels
                                                //and the mass of each one
#define FILENAME3 "num_calc_U_z0.txt"           //output file with U for each point of the plane z=0 outside
                                                //of the asteroid
#define COLS1  5                                //number of columns in the input file
#define delta  0.845                            // (G*M/(pow(a,3)*(w*w)))

int N;                              //number of voxels of the asteroid
vector< vector <double> > vox;      //array of vectors to store the voxel information (x,y,z,tag,mass)

//function that reads into memory the input data
void voxeldata(vector< vector <double> > &vox, int *N){
   fstream file;
```

```
    vector <double> rowVector(COLS1);    // vector to add into 'array' (represents a row)
    int row = 0;                         // Row counter

    file.open(FILENAME1 , ios::in);
    if (file.is_open()) {
        while (file.good()) {
            vox.push_back(rowVector);
            for (int col=0; col<COLS1; col++) {
                file >> vox[row][col];
            }
            row++;
        }
    }
    file.close();
    *N = row - 1;
}

//function to calculate the distance between two points in space
double z0_distance(double x, double y, int i){
        double distance = sqrt((x-vox[i][0])*(x-vox[i][0]) +
        (y-vox[i][1])*(y-vox[i][1]) + ( 0 - vox[i][2])*( 0 - vox[i][2]));
        return distance;
}

//function to calculate all the points in a range
void plane_points(vector< vector <double> > &plane , double size){  // vector containing 4 columns x|y|V|U

        vector < double > rowVector(4);            // vector to add into array of vectors
        int row = 0;                               // Row counter
        for(double x = -1.2; x <= 1.2 ; x = x + size){
                for(double y =-1.2; y <= 1.2; y = y + size){
                        plane.push_back(rowVector);
                        plane[row][0] = x;
                        plane[row][1] = y;
                        plane[row][2] = 0;                  // initialize V = 0 for all points
                        plane[row][3] = 0;                  // initialize U = 0 for all points
                        row++;
                }
        }
}

//function to calculate V(x,y,0) from the data of an area
void calculate_U(vector< vector <double> > &plane){
        int points = plane.size();
        for (int i = 0; i < points; i++){
                for (int j = 0; j < N; j++){
                        plane[i][2] += -(vox[j][4])/ (z0_distance(plane[i][0], plane[i][1], j));
                }
        }

        for (int i = 0; i < points; i++){
                plane[i][3] = -0.5*(plane[i][0]*plane[i][0] + plane[i][1]*plane[i][1])
                + delta*plane[i][2];
        }

}

//function to export data in file3
void output(vector< vector <double> > plane, int size){
        ofstream myfile3(FILENAME3);
        if (myfile3.is_open()){
                for (int i = 0; i < size; i++){
                        myfile3 << plane[i][0] << "    "
                        << plane[i][1]  <<  "    " << plane[i][3]   << endl; // x|y|U
                }
```

```
                myfile3.close();
                                cout << "x|y|U information  are now stored in the file:"
                                << FILENAME3 << endl;
        }
}

//main function
int main(int argc, char* argv[]){
        vector< vector <double> > plane;
        voxeldata(vox,&N);
        plane_points(plane, 0.01);
        calculate_U(plane);
        output(plane,plane.size());
}
```

Παράρτημα 5. Κώδικας *main1.cpp* και *de1gv.cpp*

main1.cpp

```
//Source code: main1.cpp//
#include "de1gv.h"
#include "restroe3.h"

OEPOS Q;                                        //Data structure for integrations

int main()
{
        double ENERGY = 0.0;                    //Initialize Energy value
        FILE *fil1 = fopen("testR.dat", "wt");  //Create output file for orbits in the rotating frame
        FILE *fil2 = fopen("testI.dat", "wt");  //Create output file for orbits in the inertial frame
        FILE *fil3 = fopen("testO.dat", "wt");  //Create output file for keplerian elements

        fprintf(fil3, "t(hr)          a          e          i          "
        " om            OM          obar      n\n"); //column titles for file 3
        double dt=0.001;
        Initialize(1.0e-10);
        if (!voxeldata(vox, &N)){                       //Checking for errors in data input
                printf("Error reading voxel data\n"); getchar(); return 0;
        }
        InitRestroeH(0, 0, DELTA);                      //Initialize restroe function
        double X0[NEQ], X[NEQ], Y[NEQ], YI[NEQ];        //Declare data structures
        X0[0] = 0;                                  //Initialize array of main variables

        //Case A: The satellite starts in a specific position and given
        //the appropriate velocity to perform approximately circular orbits
        //get circular motion (in central field approximation)
        double R = 1.5;
        GetApproxCircular(R, X0);

        //Case B: Calculate the circular orbit but with regard to the
        //orientation of the asteroid and the satellite
        //double R = 2;
        //double ThetaRot = 7*pi/4; //rad
        //GetApproxCircular(R, ThetaRot, X0);

        //Case C: Calculation of the orbits starting from specific
        //Keplerian elements' values
        //OEDEF Q0;
        //Q0.a = 2; Q0.e = 0.001; Q0.i = 0.0; Q0.omega = Q0.Omega = 0;
        //Q0.elpos = 180; Q0.inputangle = 1; Q0.modedisplay = 1;
        //GetInertialFromOrbital(Q0, YI);
        //TransformationItoR(YI, X0);
        //-------------------------------------------------
        /* Calculation of Keplerian elements at the starting point  of the orbit */
```

```
        TransformationRtoI(X0, YI);             //Transformation of rotating frame to Inertial
        GetOrbitalElements(YI, &Q, 1);          //Calculation of Keplerian elements
        printf("Rotating      Inertial\n");     //Print initial values of inertial and rotating frame
        for (int i = 1; i < NEQ; i++) printf("%f     %f\n", X0[i], YI[i]);

        TransformationItoR(YI, X0);             //Reverse transformation
        printf("-----------------------------\n");
        printf("Rotating      Inertial\n");
        for (int i = 1; i < NEQ; i++) printf("%f     %f\n", X0[i], YI[i]);
        printf("-----------------------------\n");
        printf("----Orbital elements-------\n");
        printf("a=%f    e=%f    i=%f    omegabar=%f  "
                        " Period=%f\n", Q.a, Q.e, Q.i, Q.omegabar,pi2/Q.meanmotion);
        printf("-------------------------------------------------------"
                        "----------------------------------------\n");
        getchar();
        //-------------------------------------------------
        //---------------Integration Loops-----------------

        for (int i = 0; i < NEQ; i++) X[i] = X0[i];
        int K = 10000; double DT = 0.00628318;

        for (int k = 1; k < K; k++){
                bsintegrate.DTstep(X, DT, &dt, Y);
                ENERGY =  Calc_Energy(X);       //calculate orbit's energy
                TransformationRtoI(Y, YI);
                GetOrbitalElements(YI, &Q, 1);
                for (int i = 0; i < NEQ; i++) X[i] = Y[i];
                        if (!Crush(X)){                                         //check for crush
                        printf("%d  %f    %f    %f    %f     %12.11f\n", k, Y[0], Y[1], Y[2], Y[3], ENERGY);
                        fprintf(fil1,"%f   %f    %f    %f  %f  %f  %f  \n", Y[0], Y[1], Y[2],
                        Y[3], Y[4], Y[5], Y[6]);
                        fprintf(fil2, "%f  %f    %f    %f  %f  %f  %f  \n", YI[0],
                        YI[1]*0.4, YI[2]*0.4, YI[3]*0.4, YI[4], YI[5], YI[6]);  //rescale (x,y,z)
                        double th = 2.26*YI[0] / pi2; //rescale time
                        fprintf(fil3, "%f  %f  %f   %f    %f   %f     %f   %f\n", th,
                        Q.a, Q.e, Q.i, Q.omega, Q.Omega, Q.omegabar, Q.meanmotion);
                }
                        else break;
        }
        fclose(fil1); fclose(fil2); fclose(fil3);
        getchar();
}
```

de1gv.cpp

```
//Source code: delgv.cpp//
#include "delgv.h"

//Global variables
int N;                                                  //number of voxels of the asteroid
vector< vector <double> > vox;          //array of vectors to store the voxel information (x,y,z,tag,mass)
void DEQS(double t, double X[], double Y[]);
ODESBS bsintegrate;

//Initialize accuracy for integrations
void Initialize(double acc){
        bsintegrate.open(7, DEQS, 0.000001, 0.1, acc, 0);
        bsintegrate.dt_start = 0.0001;
}

//function that reads into memory the input data
int voxeldata(vector< vector <double> > &vox, int *N){
   fstream file;
   vector <double> rowVector(COLS1);
```

```
    int row = 0;

    file.open(FILENAME1 , ios::in);
    if (file.is_open()) {
        while (file.good()) {
            vox.push_back(rowVector);
            for (int col=0; col<COLS1; col++) {
                file >> vox[row][col];
            }
            row++;
        }
    }
    else return 0;
    file.close();
    *N = row - 1;
    return 1; // last row empty
}

//Function to calculate the distance between a point
//in space and (each) voxel of the asteroid (normalized)
double distance(double x, double y, double z, int i){
        double distance = sqrt((x-vox[i][0])*(x-vox[i][0])
        + (y-vox[i][1])*(y-vox[i][1]) + ( z - vox[i][2])*( z - vox[i][2]));
        return distance;
}

//Function to calculate Sum of V'
void Sum(int N, double x, double y, double z, double *sumx, double *sumy, double *sumz){
        *sumx = *sumy = *sumz = 0;
        for (int i = 0; i < N; i++) {
                double d = distance(x, y, z, i); double d3 = d*d*d;
                *sumx += vox[i][4]*(x - vox[i][0]) / d3;
                *sumy += vox[i][4]*(y - vox[i][1]) / d3;
                *sumz += vox[i][4]*(z - vox[i][2]) / d3;
        }

}

//Function that calculates the acceleration for each direction (x,y,z)
void DEQS(double t, double X[], double Y[]){
// ROTATING SYSTEM parameters, X[0] = time, X[1] = x, X[2] = y , X[3] = z
        X[0] = t;
        Y[1] = X[4];                    // Y[1] = x'
        Y[2] = X[5];                    // Y[2] = y'
        Y[3] = X[6];                    // Y[3] = z'
        double SUMX, SUMY, SUMZ;
        Sum(N, X[1], X[2], X[3], &SUMX, &SUMY, &SUMZ);
        Y[4] = 2*X[5]*ROT + X[1]*ROT*ROT - DELTA*SUMX;
        Y[5] = -2*X[4]*ROT + X[2]*ROT*ROT - DELTA*SUMY;
        Y[6] = -DELTA*SUMZ;
}

//Function that transforms the Initial variables of
//the rotating system to those of an Inertial system
//Inertial system transformations (x,y,z,x',y',z') -> (X,Y,Z,V_x,V_y,V_z)
void TransformationRtoI(double x[], double X[]){
        double t = x[0];   //initialy array x[] contains the values from the rotating system
        X[1] = x[1]*cos(t) - x[2]*sin(t);
        X[2] = x[1]*sin(t) + x[2]*cos(t);
        X[3] = x[3];

        X[4] = (x[4] - x[2])*cos(t) - (x[5] + x[1])*sin(t);
        X[5] = (x[4] - x[2])*sin(t) + (x[5] + x[1])*cos(t);
        X[6] = x[6];
        X[0] = t;
```

```
}
//Transformation function for pos and vel variables from inertial to rotating frame
void TransformationItoR(double X[], double Y[]){
        double t = X[0];     //initialy array X[] contains the values from the inertial system
        double ct = cos(t); double st = sin(t);
        Y[0] = X[0];
        Y[1] = X[1] * ct + X[2] * st;
        Y[2] = -X[1] * st + X[2] * ct;
        Y[3] = X[3];
        Y[4] = X[4] * ct + X[5] * st + Y[2];
        Y[5] = -X[4] * st + X[5] * ct - Y[1];
        Y[6] = X[6];
}
//Transformation function that rotates an object around z axis, given an angle theta
void RotationZ(double theta, double X[], double Y[]){  //Rotate inertial pos and vel vectors by theta
        double t = -theta;  //rotangle
        double ct = cos(t); double st = sin(t);
        Y[0] = X[0];
        Y[1] = X[1] * ct + X[2] * st;
        Y[2] = -X[1] * st + X[2] * ct;
        Y[3] = X[3];
        Y[4] = X[4] * ct + X[5] * st;
        Y[5] = -X[4] * st + X[5] * ct;
        Y[6] = X[6];
}

// calculate total energy at any given point in the trajectory
double Calc_Energy(double X[]){
    double energy = 0.0;
    double u2 = 0.0;
    double V = 0.0;

    for (int i = 0; i < N; i++){
        double d = distance(X[1], X[2], X[3], i);
        V +=  - vox[i][4]/d;
    }
    u2 = X[4]*X[4]+ X[5]*X[5] + X[6]*X[6];
    energy = 0.5*u2  - 0.5 * ROT * ROT * (X[1]*X[1] + X[2]*X[2]) + DELTA*V;
    return energy;
}

//Find initial conditions for approximate circular motion in inertial frame and transform
//them to rotational frame
void GetApproxCircular(double R, double X0[]){
        double X[NEQ];
        X[0] = 0;
        X[1] = R; X[2] = 0; X[3] = 0;
        X[4] = 0; X[5] = sqrt(DELTA / R); X[6] = 0;
        TransformationItoR(X, X0);
}

//Find initial conditions for approximate circular motion in inertial frame and transform
//them to rotational frame - also rotate around z
void GetApproxCircular(double R, double ThetaRot, double X0[]){
        double X[NEQ], Y[NEQ];
        X[0] = 0;
        X[1] = R; X[2] = 0; X[3] = 0;   X[4] = 0;       X[5] = sqrt(DELTA / R); X[6] = 0;
        RotationZ(ThetaRot, X, Y);
        TransformationItoR(Y, X0); X0[0] = 0.0;
}

//error checking if the object crashes onto the asteroid (if true then -> crush)
int Crush(double X[]){
        double d = sqrt(X[1]*X[1] + X[2]*X[2] + X[3]*X[3]);  //distance from the center of coordinates 0
        if (d <= 1.1) return 1;          // d = 1 corresponds to 400m
```

```
        else return 0;
}
```

Παράρτημα 6. Κώδικας *didymos_graphics.cpp* και *data_input.h*

didymos_graphics.cpp

```
/*-----------------------------------------------include necessary libraries-----------------------------------*/
#include <windows.h>                                // for MS Windows
#include <GL/glut.h>                                // GLUT, include glu.h and gl.h
#include "data_input.h"                             // contains data reading function, and definitions
#include <math.h>
#define FACES 2292.0f                               //only for didymos, float for colour
/*-----------------------------------------------Global variables-----------------------------------------*/
char title[] = "3D Graphical Simulation of Asteroid's Didymos Surface";
GLfloat spin = 0.0f;                                // Rotational angle for the object
int refreshMills = 15;                      // refresh interval in milliseconds
/*-----------------------------------------------final object arrays--------------------------------------*/
GLfloat fixed_varray[1148][3];              //only for didymos
GLuint  fixed_indarray[2292][3];            //only for didymos
GLfloat fixed_trajectory_array[10000][3];  //number of trajectory points - variable in each case
/*-----------------------------------------------lists for ready to render objects-----------------------*/
static GLuint axes_list;
static GLuint faces_list;
static GLuint ind_list;
static GLuint graph_list;
/*-----------------------------------------camera position variables-----------------------------------------*/
float cam_angle=0.0;                                // angle of rotation for the camera direction
float lx=0.0f,lz=-1.0f;                             // vector pointing to the camera's direction
float x=0.0f,z=1.0f;                                // XZ position of the camera
/*-------------------------------------------- Initialize OpenGL Graphics-----------------------------------*/
void initGL() {
   glClearColor(0.0f, 0.0f, 0.3f, 0.0f);                // Set background color
   glClearDepth(10.0f);                                                 // Set background depth to farthest
   glEnable(GL_DEPTH_TEST);                                             // Enable depth testing for z-culling
   glDepthFunc(GL_LEQUAL);                                              // Set the type of depth-test
   glShadeModel(GL_SMOOTH);                                             // Enable smooth shading
   glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);   // Nice perspective corrections
/*-----------------------------------------------create the 3D axes list-----------------------------------*/
   axes_list = glGenLists(1);
       glNewList(axes_list, GL_COMPILE);

       glColor4ub(255, 0, 0, 255);

       glBegin(GL_LINE_STRIP);
       glVertex3f(0.0f, 0.0f, 0.0f);
       glVertex3f(1.0f, 0.0f, 0.0f);
       glEnd();

       glBegin(GL_LINE_STRIP);
       glVertex3f(0.0f, 0.0f, 0.0f);
       glVertex3f(0.0f, 1.0f, 0.0f);
       glEnd();

       glBegin(GL_LINE_STRIP);
       glVertex3f(0.0f, 0.0f, 0.0f);
       glVertex3f(0.0f, 0.0f, 1.0f);
       glEnd();

       glColor4ub(255, 255, 0, 255);
       glRasterPos3f(0.9f, 0.0f, 0.0f);
```

```
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, 'y');
        glRasterPos3f(0.0f, 0.9f, 0.0f);
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, 'z');
        glRasterPos3f(0.0f, 0.0f, 0.9f);
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, 'x');

        glEndList();
/*-------------------------------------------create the faces list-------------------------------------------*/

    faces_list = glGenLists(1);
        glNewList(faces_list, GL_COMPILE);
        int i;
        glBegin(GL_TRIANGLES);
        for (i = 0; i < FACES ; i++){
            glColor3f((FACES-i)/FACES, (FACES-i)/FACES, (FACES-i)/FACES); // from dark to bright
              glVertex3fv(&fixed_varray[fixed_indarray[i][0]][0]);
              glVertex3fv(&fixed_varray[fixed_indarray[i][1]][0]);
              glVertex3fv(&fixed_varray[fixed_indarray[i][2]][0]);
        }
        glEnd();
        glEndList();
/*-------------------------------------------create the lines list-------------------------------------------*/
    ind_list = glGenLists(1);
        glNewList(ind_list, GL_COMPILE);
        glColor3f(0.0f, 0.0f, 0.0f);
        glBegin(GL_LINES);
        for (i = 0; i < FACES; i++){
          glVertex3fv(&fixed_varray[fixed_indarray[i][0]][0]);    // connect vert[i][0] to vert[i][1]
          glVertex3fv(&fixed_varray[fixed_indarray[i][1]][0]);
        }

        for (i=0; i < FACES; i++){
          glVertex3fv(&fixed_varray[fixed_indarray[i][0]][0]);    // connect vert[i][0] to vert[i][2]
          glVertex3fv(&fixed_varray[fixed_indarray[i][2]][0]);
        }

        for (i=0; i < FACES; i++){
          glVertex3fv(&fixed_varray[fixed_indarray[i][1]][0]);    // connect vert[i][1] to vert[i][2]
          glVertex3fv(&fixed_varray[fixed_indarray[i][2]][0]);
        }
        glEnd();
        glEndList();
/*-------------------------------------------create the trajectory line-------------------------------------------*/
    graph_list =  glGenLists(1);
        glNewList(graph_list, GL_COMPILE);
        glColor3f(1.0f, 1.0f, 0.0f);
        glBegin(GL_POINTS);
        for(int i = 0; i < 10000; i++){                                          //change according to the number of p
            glVertex3f(fixed_trajectory_array[i][1],fixed_trajectory_array[i][2],fixed_trajectory_array[i][3]);
        }
        glEnd();
        glEndList();
}

/*-----window-repaint: Called when the window first appears and  whenever the window needs to be re-painted-----*/
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);    // Clear color and depth buffers
    glMatrixMode(GL_MODELVIEW);                                                  // To operate on model-view matrix

// Render object
/*---------------------basic opengl functions for object position , and any movement or rotation----------------*/
    glLoadIdentity();
// Reset the model-view matrix
    gluLookAt(    x, 0.15f, z,                                                   // Set the camera
                        x+lx, 0.15f,  z+lz,
```

```
                        0.0f, 0.15f,  0.0f);
    glTranslatef(0.0f, 0.0f, -2.5f);                                    // Move
into the screen more than max distance  (zoom out)

/*-------------------------------------------------render the trajectory-------------------------------------------*/
    glPushMatrix();
    glCallList(graph_list);
    glPopMatrix();
    //glRotatef(spin, 0.0f, 1.0f, 0.0f);  // Enable to Rotate about the y-axis in computational space , (z axis)

/*-----------------------------------------------render primitives: lines-----------------------------------------*/
    glPushMatrix();
    glCallList(ind_list);
    glPopMatrix();

/*-----------------------------------------------render primitives: triangles-------------------------------------*/
    glPushMatrix();
    glCallList(faces_list);
    glPopMatrix();

/*-------------------------------------------------render the xyz axes----------------------------------------------*/
    glPushMatrix();                                                     // put them before glRotate for
no rotation
    glCallList(axes_list);
    glPopMatrix();

    glutSwapBuffers();   // Swap the front and back frame buffers (double buffering)
    spin += 0.5f;                // rotational angle after each refresh
}
/*-----------------------------------------------keys input for camera-------------------------------------------*/
void processSpecialKeys(int key, int xx, int yy) {
        float fraction = 0.2f;
        switch (key) {
                case GLUT_KEY_LEFT :
                        cam_angle -= 0.01f;
                        lx = sin(cam_angle);
                        lz = -cos(cam_angle);
                        break;
                case GLUT_KEY_RIGHT :
                        cam_angle += 0.01f;
                        lx = sin(cam_angle);
                        lz = -cos(cam_angle);
                        break;
                case GLUT_KEY_UP :
                        x += lx * fraction;
                        z += lz * fraction;
                        break;
                case GLUT_KEY_DOWN :
                        x -= lx * fraction;
                        z -= lz * fraction;
                        break;
        }
}

/*-----------------------------------Timer function, it is called back when it expires-----------------------------*/
void timer(int value) {
    glutPostRedisplay();                                    // Post re-paint request to activate display()
    glutTimerFunc(refreshMills, timer, 0);
}

/*--------Window re-size event. Called back when the window first appears and whenever the window is re-sized-----*/
void reshape(GLsizei width, GLsizei height)     {               // GLsizei for non-negative integer
    if (height == 0) height = 1;                                // aspect ratio of new window , error: divide by 0
    GLfloat aspect = (GLfloat)width / (GLfloat)height;
```

```
    // Set the viewport
    glViewport(0, 0, width, height);

    // Set the aspect ratio of the clipping volume to match the viewport
    glMatrixMode(GL_PROJECTION);                        // To operate on the Projection matrix
    glLoadIdentity();                                   // Reset
    gluPerspective(45.0f, aspect, 0.1f, 100.0f);
}
/*-----------------------------All file input needed for rendering the vertex_array/asteroid-------------------------*/
void create_data_arrays(){
        int number_of_vertices ;
        int number_of_indices ;
        int n;
        vector< vector <float> > vertices;
        data(vertices, &number_of_vertices,3,1);
        for (int row = 0; row < number_of_vertices; row++){             //(x,y,z) -> (z,x,y) -> (y,z,x)
                fixed_varray[row][0]=vertices[row][1];
                fixed_varray[row][1]=vertices[row][2];
                fixed_varray[row][2]=vertices[row][0];
        }

        vector< vector <int> > indices;
        data(indices, &number_of_indices,3,2);
     for (int row = 0; row < number_of_indices; row++){
        for (int col =0; col < 3; col++){
                fixed_indarray[row][col]= indices[row][col] - 1;//change values (from 1 to 1148) to (from 0 to 1147)
        }
     }

     vector< vector <float> > points;
     data(points, &n,7, 3);
     for (int row = 0; row < n; row++){
        fixed_trajectory_array[row][0]=points[row][1];
        fixed_trajectory_array[row][1]=points[row][2];
        fixed_trajectory_array[row][2]=points[row][3];
     }

}
/*---------------Main function:GLUT runs as a console application starting at main()-------------------------------*/
int main(int argc, char** argv) {
    create_data_arrays();
/*-------------------------------------------------------------GRAPHICS-----------------------------------------------*/
    glutInit(&argc, argv);                          // Initialize GLUT
    glutInitDisplayMode(GLUT_DOUBLE);               // Enable double buffered mode
    glutInitWindowSize(1280, 720);                  // Set the window's initial width & height
    glutInitWindowPosition(50, 50);                 // Position the window's initial top-left corner
    glutCreateWindow(title);                        // title
    glutDisplayFunc(display);                       // window re-paint event
    glutReshapeFunc(reshape);                       // window re-size event
    initGL();                                       // initialize
    glutTimerFunc(0, timer, 0);                     // timer call
    glutSpecialFunc(processSpecialKeys);            // move around the camera
    glutMainLoop();                                 // event loop
    return 0;
}
```

data_input.h

```
//Header for data input functions: data_input.h//
#ifndef DATA_INPUT_H_
#define DATA_INPUT_H_

#include <vector>
#include <fstream>
#include <iostream>
```

```
using namespace std;

#define FILENAME1 "didymosl162_vertices.txt"
#define FILENAME2 "didymosl162_indices.txt"
#define FILENAME3 "testI.dat"

template <typename  T>  //Using a template in order to create an instance of the function for integers and one
for floats
void data(vector<vector< T > > &vec, int *N, int COLUMNS, int filenumber){
        fstream file;
    vector < T > rowVector(COLUMNS);     // vector to add into 'array' (represents a row)
    int row = 0;
    if (filenumber == 1)
        file.open(FILENAME1 , ios::in); // read vertices
    else if (filenumber == 2)
        file.open(FILENAME2 , ios::in); // read indices
    else
        file.open(FILENAME3 , ios::in); //contains data of the trajectory

    if (file.is_open()) {
        while (file.good()) {   // check for errors
                vec.push_back(rowVector);        // add a new row
                for (int col=0; col<COLUMNS; col++) {
                        file >> vec[row][col];  // fill the row with col elements
            }
                row++;  // number of rows in my file
        }
    }
    else cout << "Unable to open file" << endl;
    file.close();
    *N = row;    //counting total rows recorded in the memory
}


#endif /* DATA_INPUT_H_ */
```

Bibliography

[1] Orbital Elements, *https://en.wikipedia.org/wiki/Orbital_elements*

[2] Ι.Χατζηδημητρίου,*Θεωρητική Μηχανική*, Τόμος Α,Β, 2000

[3] 65803 Didymos, *https://en.wikipedia.org/wiki/65803_Didymos*

[4] Double Asteroid Redirection Test (DART) Mission, *https://www.nasa.gov/planetarydefense/dart*

[5] S.Tardivel, *The limits of the mascons approximation of the homogeneous polyhedron*, Jet Propulsion Laboratory, California Institute of Technology, California, 2016

[6] Yu Jiang, Hexi Baoyin, *Orbital Mechanics near a Rotating Asteroid*, State Key Laboratory of Astronautic Dynamics, China, 2014

[7] D.J. Scheeres, *Dynamics about Uniformly Rotating Triaxial Ellipsoids: Application to Asteroids*,Jet Propulsion Laboratory, California Institute of Technology, California, 1994

[8] William H. Press et All, *Numerical Recipes*, Cambridge University press, 2007

[9] Eigen library, *hhtp://eigen.tuxfamily.org/dox/*

[10] PDS Asteroid/Dust Archive, *https://sbn.psi.edu/pds/*

[11] Ε. Δασκαλάκης, *Προσομοίωση τροχιών στο πεδίο βαρύτητας του αστεροειδή 433 Έρως*, Διπλωματική Εργασία, Τμήμα Φυσικής, Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης, 2014