POLITECHNIKA POZNAŃSKA

WYDZIAŁ INŻYNIERII LĄDOWEJ I TRANSPORTU
Instytut Energetyki Cieplnej

Praca dyplomowa inżynierska

OPRACOWANIE NARZĘDZIA DO ZAUTOMATYZOWANEGO WYZNACZANIA TRANSFERÓW ORBITALNYCH DO I Z ASTEROID

Alan Baker, 141589

Promotor
dr. inż. Przemysław Grzymisławski

POZNAŃ 2022

Tutaj będzie karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

Streszczenie (Abstract)

Praca zawiera narzędzie własne do zautomatyzowanego wyznaczania transferów orbitalnych do i z asteroid oraz opisuje działanie poszczególnych jej funkcji. Trajektorie między obiektami wyznaczono za pomocą algorytmu Izzo, służącego do rozwiązywania problemu Lamberta. Narzędzie napisano w języku programowania Python 3.7. Do obliczeń wymagane są parametry, określające obiekty odlotu i przylotu oraz zadany okres czasu analizy.

Przedstawiono też przykładowy transfer z Ziemi do asteroidy 1996FG3 oraz transfer powrotny w przedziale czasowym od 2027-01-01 00:00:00 do 2033-01-01 00:00:00. Z przeprowadzonej analizy wyników wybrano misję o najmniejszym delta-v wynoszącym 51.67 [km/s], przy czasie odlotu z Ziemi 2029-09-14, czasie przylotu na asteroidę 2030-03-26, czasie odlotu z asteroidy 2032-02-28 oraz czasie powrotu na Ziemię 2032-11-04.

This dissertation describes the operation of the various functions of a self-created software tool that automates the orbital transfer calculations for journeys between Earth and a selected asteroid. The trajectories between the astronomical bodies were determined using the Izzo algorithm to solve the Lambert problem. The tool is written in the Python 3.7 programming language. As input, the program requires the user to specify the parameters of the departure and arrival bodies and the time for analysis.

Various transfers from the Earth to the asteroid 1996FG3 and back for the time interval 2027-01-01 to 2033-01-01 are calculated. The tool found that the minimum value of delta-v was 51.67 [km/s] for a mission with a departure time from Earth on 2029-09-14 and an arrival time on the asteroid of 2030-03-26. The departure time from the asteroid was 2032-02-28, and the arrival time on Earth was 2032-11-04.

Rozdział 1

Wstęp

Przyszłe uprzemysłowienie przestrzeni kosmicznej spowoduje poszukiwanie zasobów kosmicznych, zarówno w zakresie masy konstrukcyjnej, jak i materiałów pędnych. Szczególnie, rozwój wiedzy o asteroidach bliskich Ziemi, dokonany w ostatnich dziesięcioleciach czyni je potencjalnie lukratywnymi obiektami zainteresowania. [19] Projektowanie, więc, narzędzi do analizy trajektorii lotu misji międzyplanetarnych stanowi pożądane i ważne zadanie.

Modelowanie trajektorii lotu jest jednym z najważniejszych aspektów projektowania każdej misji kosmicznej. Pierwszym krokiem do realizacji tych misji jest właściwe obliczenie okna startowego, czasu przelotu, zmiany prędkości oraz trajektorii. Celem niniejszej pracy jest opracowanie algorytmów do wyznaczania transferów orbitalnych statków kosmicznych od planety do asteroidy i z powrotem. Poszczególne funkcje algorytmów przetwarzają i zwracają wymienione powyżej elementy składowe misji.

Kluczową pozycję w materiałach źródłowych zajmuje podręcznik akademicki Howarda Curtisa pt. *Orbital mechanics for engineering students*, który zawiera dogłębną analizę zagadnień z zakresu dynamiki orbitalnej ze szczególnym uwzględnieniem problematyki niniejszej pracy, czyli manewrów orbitalnych i trajektorii międzyplanetarnych. Dostępne online opracowanie Roberta Baeuninga [15] w sposób zwięzły wyjaśnia niezbędne kwestie mechaniki lotu i stanowi praktyczne kompendium referencyjne.

Algorytmy zostały napisane w języku programowania Python (wersja 3.7). Do poprawnego działania wymagane są następujące biblioteki:

- math
- Astropy
- SciPy
- poliastro
- NumPy

Struktura pracy jest następująca:

Rozdział pierwszy ma charakter wprowadzający.

W rozdziale 2 przedstawiono zagadnienia teoretyczne związane z podstawowymi prawami astrodynamiki, właściwościami orbit i transferami międzyplanetarnymi.

Rozdział 3 zawiera opisy algorytmów służących do wyznaczania transferów orbitalnych do i z asteroid. Określono w nim warunki, jakie powinny spełniać dane wejściowe i wyniki oraz konieczne zmienne. Dokładnie opisano działanie kolejnych funkcji.

W rozdziale 4 zestawiono wygenerowane wyniki algorytmów.

Rozdział 5 stanowi podsumowanie pracy, w którym zaprezentowano wnioski i rekomendacje do przyszłych badań.

Rozdział 2

Trajektorie międzyplanetarne i prawa nimi rządzące

2.1 Prawa Keplera

Jan Kepler odkrył i sformułował w następujący sposób trzy prawa rządzące ruchem ciał niebieskich krążących po orbicie Słońca:

I Prawo - Każda planeta Układu Słonecznego porusza się wokół Słońca po orbicie w kształcie elipsy, w której w jednym z ognisk jest Słońce. Ekscentryczność elipsy (stopień jej wydłużenia) e równa się stosunkowi długości odcinka l między środkiem elipsy a jednym z jej ognisk do długości wielkiej półosi a.

$$e = \frac{l}{a} \tag{2.1}$$

II Prawo - W równych odstępach czasu promień wodzący planety, poprowadzony od Słońca, zakreśla równe pola.

Prawo to ilustruje poniższy wzór:

$$\frac{\Delta A}{\Delta t} = const \tag{2.2}$$

III Prawo - Stosunek kwadratu okresu obiegu planety wokół Słońca do sześcianu wielkiej półosi jej orbity jest stały dla wszystkich planet w Układzie Słonecznym.

III prawo wyraża następujący wzór:

$$\frac{T_1^2}{a_1^3} = \frac{T_2^2}{a_2^3} = const \tag{2.3}$$

T_1,T_2 – okresy obiegów dwóch planet
a_1, a_2 – wielkie półosie tych planet

2.2 Parametry orbitalne (elementy Keplerowskie)

Położenie każdego obiektu na orbicie definiuje się przy użyciu sześciu parametrów orbitalnych, zwanych także elementami Keplerowskimi. Należą do nich ekscentryczność (inaczej mimośród), półoś wielka orbity, anomalia prawdziwa, argument perycentrum (długość perycentrum), inklinacja oraz długość węzła wstępującego.

1. Ekscentryczność – to parametr określający kształt orbity. Jest to własność krzywej stożkowej, który opisany jest okręgiem, elipsą, parabolą lub hiperbolą. Dla okręgu ekscentryczność wynosi 0. W przypadku elipsy jest mniejsza od wartości 1 . Dla paraboli równa się 1, a hiperboli osiąga wartość większą niż 1.

2. Półoś wielka orbity – w zależności od stożka przyjmuje się ją jako połowę sumy długości perycentrum i apocentrum dla ekscentryczności mniejszej niż 1. Dla ekscentryczności większej od 1 (czyli orbity hiperbolicznej) półoś wielką orbity można obliczyć za pomocą poniższego równania:
$$v^2 = GM(\frac{2}{r} - \frac{1}{a}) \tag{2.4}$$
gdzie:

   v - prędkość na orbicie

   G - stała grawitacji

   r - promień orbity

   a - długości wielkiej półosi

3. Anomalia prawdziwa – to wielkość określająca czas od przejścia przez perygeum. Przyjmuje wartości od 0 do 2π. Używa się także parametru kątowego wiążącego położenie ciała z czasem zwanego anomalią średnią. W jej przypadku za punkt odniesienia przyjmuje się epokę w celu prawidłowego ustalenia położenia orbitującego ciała.

4. Argument perycentrum - to kąt między kierunkiem węzła wstępującego a kierunkiem perygeum. Określa orientację orbity w jej płaszczyźnie. Może wynosić od 0 do π.

5. Inklinacja – kąt pomiędzy płaszczyzną orbitalną a płaszczyzną odniesienia. Przykładową płaszczyzną odniesienia może być płaszczyzna równika ciała centralnego. Inklinacja orbity przyjmuje wartości od 0 do π.

6. Długość węzła wstępującego – to kąt pozycyjny liczony w wybranej płaszczyźnie od pewnego ustalonego kierunku do punktu, w którym poruszające się po orbicie ciało przekracza tę płaszczyznę ze strony południowej na północną (węzeł wstępujący). Długość węzła wstępującego zawiera się między 0 a 2 π włącznie.

Parametry orbitalne przedstawiono na rysunku 2.1:

Do obliczeń orbitalnych stosuje się także anomalię średnią.

Anomalia średnia wskazuje, gdzie satelita znajdował się na swojej orbicie w danej epoce. Średnią anomalię w dowolnym momencie t, $M(t)$, można wyznaczyć dodając ostatnią znaną średnią anomalię, M_0, do średniego ruchu orbity pomnożonego przez czas, który upłynął $(t - t_0)$:
$$M(t) = M_0 + n(t - t_0) \tag{2.5}$$
gdzie:

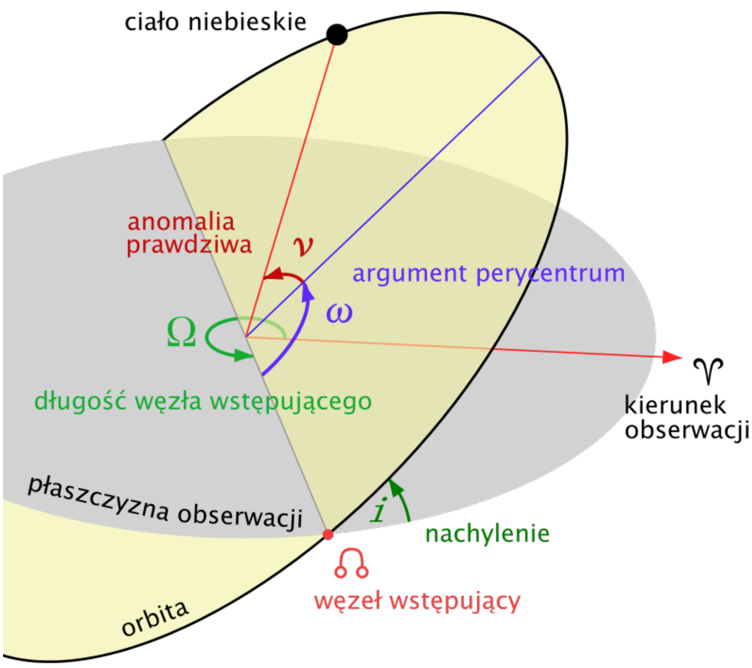
$M(t)$ = średnia anomalia w czasie t

M_0 = średnia anomalia w chwili t=0

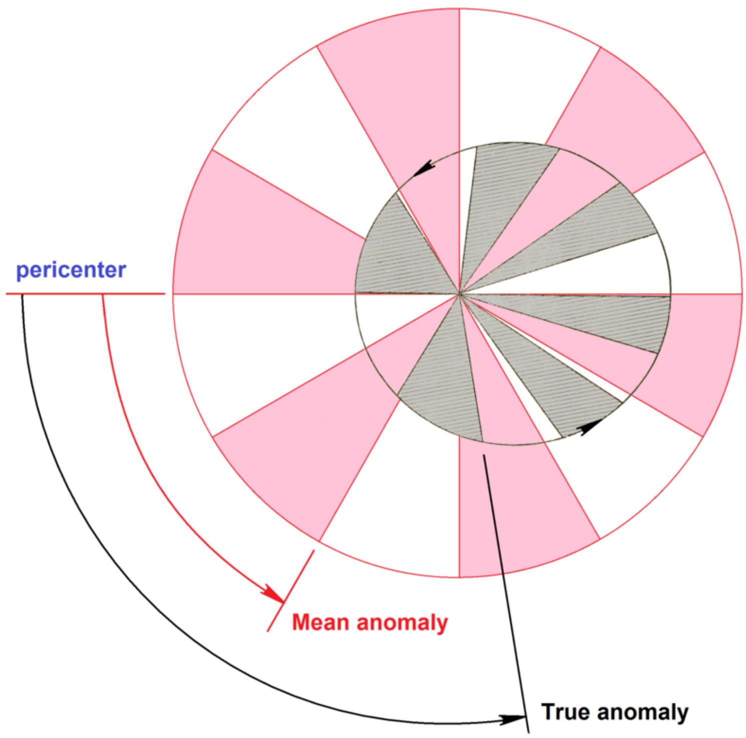
n = średnia ruchliwość orbity satelitarnej

t = wybrana godzina

t_0 = czas ostatniej znanej średniej anomalii



Rysunek 2.1: Parametry orbitalne [4]



Rysunek 2.2: Anomalia średnia (pericenter – perycentrum, mean anomaly – anomalia średnia, true anomaly – anomalia prawdziwa) [6]

Dla orbity idealnie kołowej (ekscentryczność 0), średnia anomalia jest dokładnie równa anomalii prawdziwej na całej orbicie. Średnia anomalia jest związana z anomalią mimośrodową (E) za pomocą równania Keplera. Anomalia średnia może mieć zakres od 0 do 360°[7].

2.3 Strefa wpływów

W zależności od położenia na orbicie przyciąganie grawitacyjne ciał ulega zmianie. W myśl prawa powszechnego ciążenia Newtona siła tego przyciągania jest odwrotnie proporcjonalna do kwadratu odległości między środkami tych ciał. Problem oddziaływania wielu ciał można uprościć przyjmując założenie, że gdy ciało niebieskie jest odpowiednio blisko w stosunku do ciała centralnego, to siły przyciągania innych, bardziej oddalonych ciał można pominąć. Obliczenie, zatem, odległości, przy której wpływ innych ciał można zaniedbać przedstawia się następująco:

$$Strefa\ wpływów = a(\frac{m}{M})^{\frac{2}{5}} \tag{2.6}$$

a – półoś wielka ciała
m – masa mniejszego ciała orbitującego np. Ziemi
M – masa centralnego ciała np. Słońca [20]

2.4 Prędkość ucieczki (druga prędkość kosmiczna)

Mimo faktu, że każde ciało ma swoją strefę wpływów, przyciąganie grawitacyjne można pokonać dostarczając ciału odpowiednią ilość energii, co spowoduje uwolnienie od przyciągania ciała centralnego. Aby ciało orbitujące centralną planetę opuściło zatem jej strefę wpływów prędkość ucieczki można opisać równaniem:

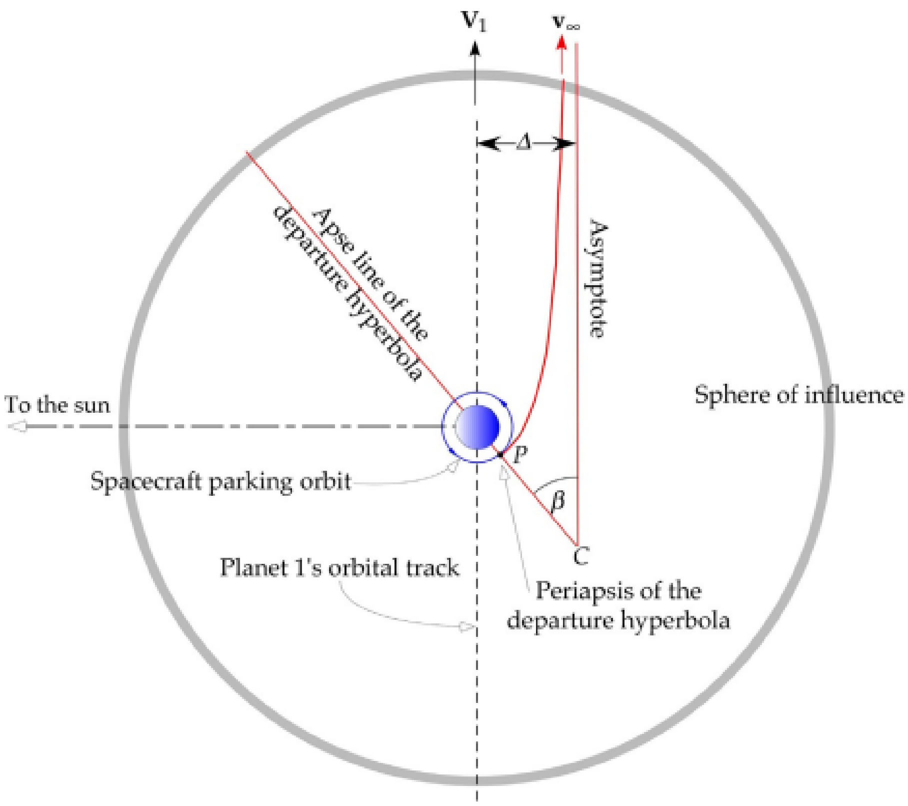
$$V_{esc} = \sqrt{\frac{2GM}{r}} \tag{2.7}$$

M – masa centralnego ciała
r - odległość ciała orbitującego od centralnego
G – stała grawitacji

Gdy nada się ciału prędkość równą lub większą od prędkości ucieczki, nie znajduje się już ono na orbicie wokół planety centralnej, ale ucieka po parabolicznej lub hiperbolicznej trajektorii. Przykładowo w celu dotarcia do Marsa, statek kosmiczny powinien osiągnąć prędkość ucieczki co najmniej większą niż prędkość ucieczki dla planety Ziemi.[20]

2.5 Opuszczenie strefy wpływów planety

Aby uciec przed przyciąganiem grawitacyjnym planety, statek kosmiczny musi przebyć trajektorię hiperboliczną względem tej planety, docierając do brzegu jej sfery wpływów z prędkością względną v_∞ (nadmiarowa prędkość hiperboliczna) większą od zera.



RYSUNEK 2.3: Odlot statku kosmicznego na trajektorii od planety wewnętrznej do planety zewnętrznej. (sphere of influence – strefa wpływów, periapsis of the departure hyperbola – perycentrum hiperboli, planet 1's orbital track – tor orbity pierwszej planety, spacecraft parking orbit – orbita parkingowa statku kosmicznego, asymptote – asymptota, apse line of the departure hyperbola – linia absyd hiperboli, to the Sun – w kierunku Słońca) [16]

Do wyznaczenia v_∞ przedstawionej na rys.2.3 należy skorzystać ze wzoru:

$$v_\infty = \sqrt{\frac{\mu_{sun}}{r_1}(\sqrt{\frac{2r_2}{r_1+r_2}}-1)} = \frac{\mu_1}{h} esin\theta_\infty = \frac{\mu_1}{h}\sqrt{e^2-1} \tag{2.8}$$

gdzie:

r_1 - promień orbity planety odlotu

r_2 - promień orbity planety docelowej

μ_{sun} - parametr grawitacyjny słońca

h - właściwy moment pędu hiperboli odlotu (względem planety)

e – ekscentryczność hiperboli

μ_1 - parametr grawitacyjny planety odlotu

Pojazd kosmiczny wystrzeliwuje się zwykle na trajektorię międzyplanetarną z parkingowej orbity kołowej. Promień tej orbity parkingowej jest równy promieniowi perycentrum r_p hiperboli odlotu. Promień perycentrum można otrzymać przy pomocy następującego równania:

$$r_p = \frac{h^2}{\mu_1}\frac{1}{1+e} \tag{2.9}$$

gdzie:

μ_1 - parameter grawitacyjny planety

Ponieważ nadmiarowa prędkość hiperboliczna jest określona przez wymagania misji (równanie 2.8), wybór perycentrum odlotu r_p wyznacza parametry e i h hiperboli odlotu. Aby wyznaczyć deltę-v potrzebną do umieszczenia statku kosmicznego na hiperboli odlotu można skorzystać ze wzoru:

$$\Delta v = v_c(\sqrt{2 + (\frac{v_\infty}{v_c})^2} - 1) \tag{2.10}$$

gdzie:

v_c - prędkość statku na orbicie parkowania

Prędkość v_c wyraża się równaniem:

$$v_c = \sqrt{\frac{\mu_1}{r_p}} \tag{2.11}$$

Położenie perycentrum, w którym musi nastąpić manewr delta-v, można znaleźć korzystając z równania:

$$\beta = cos^{-1}(\frac{1}{e}) = cos^{-1}(\frac{1}{1 + \frac{r_p {v_\infty}^2}{\mu_1}}) \tag{2.12}$$

gdzie β określa orientację linii apsyd hiperboli względem heliocentrycznego wektora prędkości planety.

Gdy statek kosmiczny znajduje się już na orbicie parkingowej, możliwość wystrzelenia go na trajektorię odlotową pojawia się raz w każdym obiegu orbitalnym.

Jeśli misja polega na wysłaniu statku kosmicznego z planety zewnętrznej na planetę wewnętrzną, jak na rys. 2.4, to prędkość heliocentryczna statku kosmicznego przy starcie musi być mniejsza niż prędkość planety. Oznacza to, że statek kosmiczny musi wyjść z tylnej części sfery wpływów z wektorem prędkości względnej skierowanym przeciwnie do V_1, jak pokazano na rys. 2.4.

W przypadku transferów nie-Hohmannowskich, zaprezentowanych na rys. 2.5 powyższe parametry oblicza się z zastosowaniem następujących wzorów.

Wektor hiperbolicznej prędkości odlotu:

$$\boldsymbol{v_\infty} = \boldsymbol{V_D} - \boldsymbol{V_1} \tag{2.13}$$

gdzie:

$\boldsymbol{V_1}$ - wektor prędkości planety odlotu

$\boldsymbol{V_D}$ - wektor prędkości statku w odniesieniu do Słońca

Hiperboliczna prędkość odlotu:

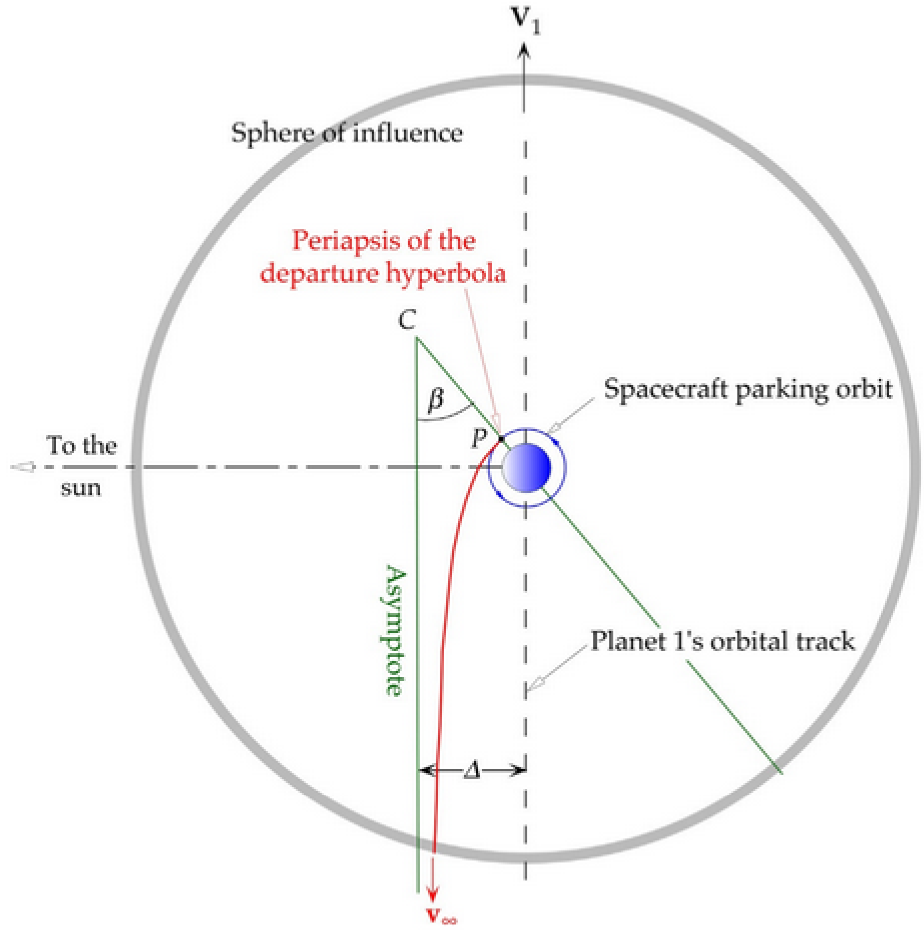
$$v_\infty = |\boldsymbol{V_D} - \boldsymbol{V_1}| \tag{2.14}$$

Prędkość statku na perycentrum hiperboli odlotu:

$$v_{odlotu} = \sqrt{v_\infty^2 + \frac{2\mu_1}{r_p}} \tag{2.15}$$

Delta-v odlotu:

$$\Delta v = v_{odlotu} - v_c \tag{2.16}$$



RYSUNEK 2.4: Odlot statku kosmicznego na trajektorii od planety zewnętrznej do planety wewnętrznej (sphere of influence – strefa wpływów, periapsis of the departure hyperbola – perycentrum hiperboli, planet 1's orbital track – tor orbity pierwszej planety, spacecraft parking orbit – orbita parkingowa statku kosmicznego, asymptote – asymptota, apse line of the departure hyperbola – linia absyd hiperboli, to the Sun – w kierunku Słońca) [16]

2.6 Wejście w strefę wpływów

Wchodząc w strefę wpływów planet, statek kosmiczny porusza się z prędkością względną v_∞ (nadmiarowa prędkość hiperboliczna) większą od zera. Prędkość tą trzeba zniwelować, aby wejść na orbitę okołoplanetarną.

Wektor hiperbolicznej prędkości przylotu:

$$\boldsymbol{v_\infty} = \boldsymbol{V_A} - \boldsymbol{V_2} \tag{2.17}$$

gdzie:

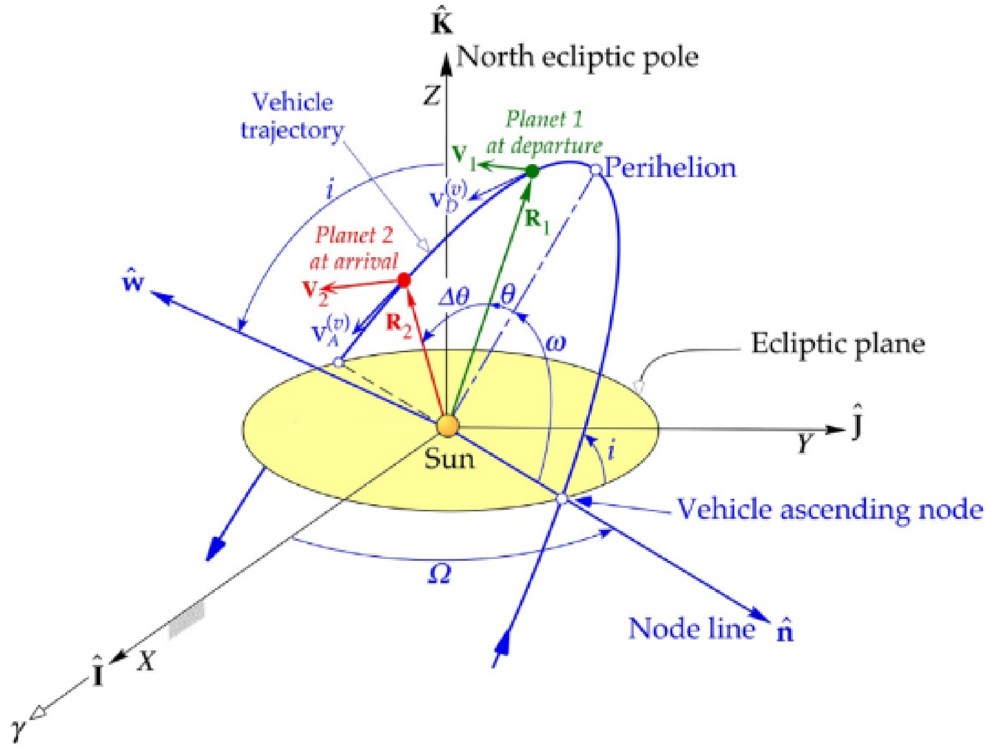
$\boldsymbol{V_2}$ - wektor prędkości planety przylotu

$\boldsymbol{V_A}$ - wektor prędkości statku w odniesieniu do Słońca

Hiperboliczną prędkość przylotu można obliczyć z równania:

$$v_\infty = |\boldsymbol{V_A} - \boldsymbol{V_2}| \tag{2.18}$$

Prędkość statku na perycentrum hiperboli przylotu:



RYSUNEK 2.5: Heliocentryczne elementy orbitalne trójwymiarowej trajektorii transferu z planety 1 do planety 2 (Planet 1 at departure - położenie planety 1 przy odlocie, Planet 2 at arrival - położenie planety 2 przy przylocie, Vehicle trajectory - trajektoria statku, Perihelion - Peryhelium, Vehicle ascending node - węzeł wstępujący statku, Node line - linia węzłów, Sun - Słońce, North ecliptic pole - Północny biegun ekliptyczny, Eliptic plane - Płaszczyzna Elipsy) [16]

$$v_{przylotu} = \sqrt{v_\infty^2 + \frac{2\mu_2}{r_p}} \tag{2.19}$$

gdzie:

μ_2 - parametr grawitacyjny planety przylotu

r_p - perycentrum hiperboli przylotu

Prędkość statku na eliptycznej orbicie parkingowej:

$$v_{el} = \sqrt{\frac{\mu_2}{r_p}(1+e)} \tag{2.20}$$

gdzie:

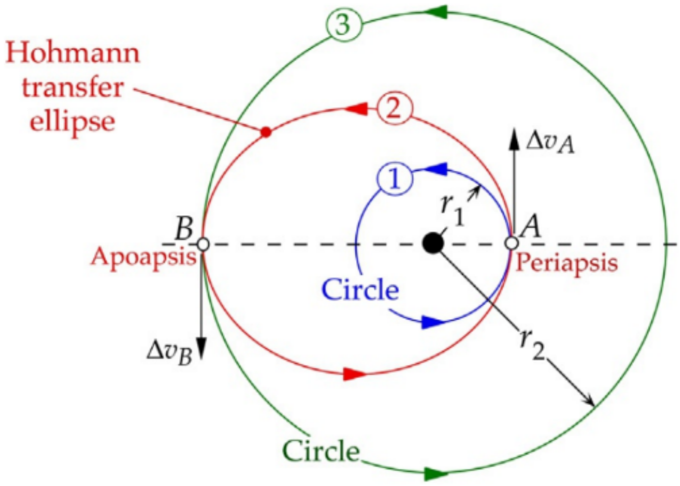
e - ekscentryczność orbity parkingowej:

Aby deltę-v przylotu powinno się skorzystać ze wzoru:

$$\Delta v = v_{przylotu} - v_{el} \tag{2.21}$$

2.7 Transfer Hohmanna

Transfer Hohmanna to manewr, który przenosi statek kosmiczny z jednej orbity kołowej na drugą, gdy posiadają one wspólne ognisko i są współnopłaszczyznowe. Wymagana delta-V do takiego manewru (zmiana prędkości) ma najniższą wartość całkowitą, co sprawia, że jest on bardzo wydajny energetycznie i tym samym ekonomiczny w zużyciu paliwa. Aby poprawnie wykonać transfer konieczne są dwa uruchomienia silnika (transfer dwuimpulsowy). Sam transfer jest więc orbitą eliptyczną styczną do obu okręgów na jej linii apsyd. Perycentrum oraz apocentrum elipsy transferowej stanowią promienie odpowiednio wewnętrznego i zewnętrznego. okręgu. W trakcie manewru zakreślana jest połowa elipsy, a manewr może być wykonywany w obu kierunkach, od okręgu wewnętrznego do zewnętrznego lub odwrotnie.



Rysunek 2.6: Transfer Hohmanna (Periapsis - perycentrum, apoapsis - apocentrum, Hohmann transfer ellipse – elipsa transferu Hohmanna)[16]

Rysunek 2.6 przedstawia reprezentację graficzną transferu Hohmanna. Począwszy od punktu A na wewnętrznym okręgu (1), wymagany jest przyrost prędkości Δv_A w kierunku lotu, aby skierować pojazd na trajektorię eliptyczną o wyższej energii (2). Po przejściu z punktu A do B, kolejny przyrost prędkości Δv_B umieszcza pojazd na jeszcze bardziej energetycznej, zewnętrznej orbicie kołowej (3).

Całkowity wydatek energetyczny odzwierciedla się w całkowitym zapotrzebowaniu delta-v i można je wyrazić równaniem:

$$\Delta v_{total} = \Delta v_A + \Delta v_B \tag{2.22}$$

Wymagana delta-v statku kosmicznego w punkcie A wyrazić można następującym przekształceniem:

$$\Delta v_A = \sqrt{\frac{\mu_{sun}}{r_1}}\left(\sqrt{\frac{2r_2}{r_1+r_2}} - 1\right) \tag{2.23}$$

Podobnie deltę-v w punkcie B przedstawia wyrażenie:

$$\Delta v_B = \sqrt{\frac{\mu_{sun}}{r_2}}\left(1 - \sqrt{\frac{2r_1}{r_1+r_2}}\right) \tag{2.24}$$

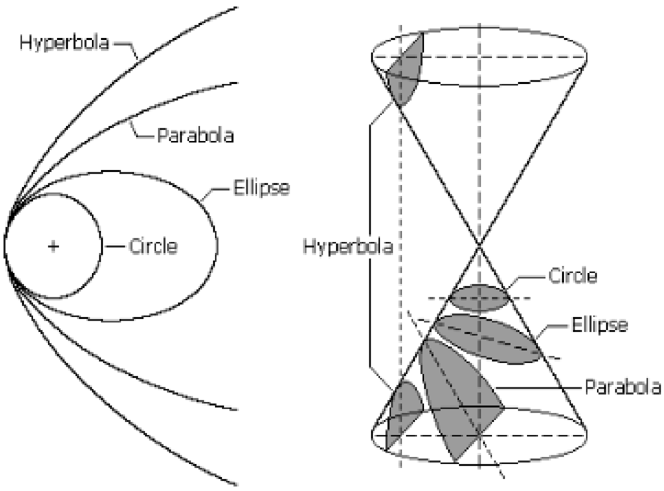
Jeśli początek transferu stanowi punkt B na zewnętrznej orbicie kołowej, konieczne jest takie samo zapotrzebowanie delta-V. W tym przypadku deltę-V osiąga się wskutek zapłonu silników hamujących, gdyż przejście na wewnętrzny krąg wymaga obniżenia energii statku kosmicznego. Ciąg statku skierowany jest przeciwnie do kierunku lotu. Przejście więc na wyższą orbitę i odwrotnie odbywa się przy tym samym wydatku paliwa.

2.8 Kształty orbit

Orbita to tor, po którym porusza się ciało niebieskie w polu sił grawitacji wokół wspólnego środka masy. Orbity można ogólnie podzielić na otwarte i zamknięte. W przypadku orbity otwartej ciało nie powraca do punktu wyjścia, natomiast gdy orbita jest zamknięta ciało wraca do punktu początkowego.

Kształty orbit:

- zamknięte – koło oraz elipsa
- otwarte – parabola i hiperbola



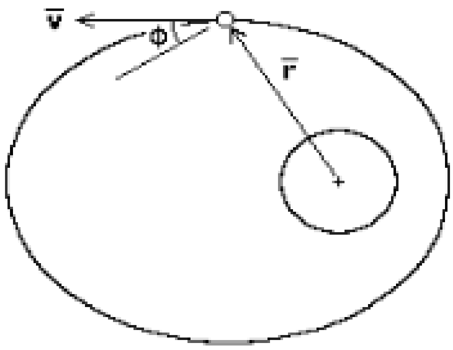
Rysunek 2.7: Przekroje stożka (hyperbola – hiperbola, circle – koło, ellipse – elipsa) [15]

Orbity określa się w zależności od ich ekscentryczności. Od tego parametru zależy również energia orbity. W tabeli poniżej zawarto właściwości otwartych i zamkniętych orbit.

Przekrój stożka	**Ekscentryczność**	**Półoś wielka**	**Energia**
Koło	0	= promień	<0
Elipsa	0 <e <1	>0	<0
Parabola	1	nieskończoność	0
Hiperbola	>1	<0	>0

Tablica 2.1: Właściwości różnych typów orbit [15]

Najczęściej używane orbity, począwszy od wystrzeliwania satelitów, a skończywszy na transferach w obrębie Układu Słonecznego to orbity eliptyczne.



Rysunek 2.8: Typowy transfer orbity eliptycznej [15]

Trzy następujące równania mają zastosowanie do obliczania elementów orbitalnych:

$$(\frac{r_p}{r}) = \frac{-C \pm \sqrt{C^2 - 4(1-C)(-cos^2\phi)}}{2(1-C)} \tag{2.25}$$

$$C = \frac{2GM}{rv^2} \tag{2.26}$$

gdzie:

$$e = \sqrt{(\frac{rv^2}{GM} - 1)cos^2\phi + sin^2\phi} \tag{2.27}$$

$$tanv = \frac{(\frac{rv^2}{GM})cos\phi sin\phi}{(\frac{rv^2}{GM})cos^2\phi - 1} \tag{2.28}$$

Pierwsze równanie kwadratowe ma dwa rozwiązania odpowiadające promieniowi perygeum i apogeum. Zatem dla orbity eliptycznej półoś wielką można obliczyć biorąc połowę z sumy perygeum i apogeum lub z poniższego równania:

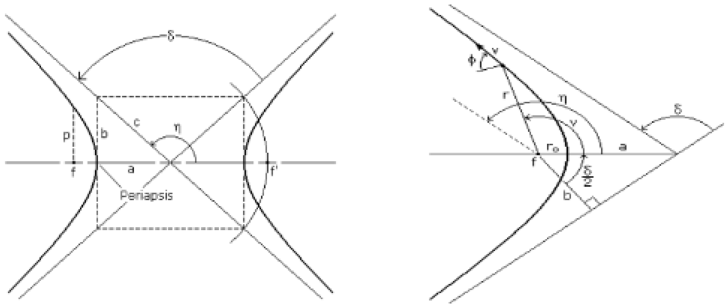
$$v^2 = GM(\frac{2}{r} - \frac{1}{a}) \tag{2.29}$$

Po wyznaczeniu półosi wielkiej, ekscentryczności i prawdziwej anomalii orbity, kąt nachylenia toru lotu i położenie orbitującego ciała można rekurencyjnie obliczyć za pomocą poniższych wzorów:

$$\phi = arctan(\frac{esinv}{1 + ecosv}) \tag{2.30}$$

$$r = \frac{a(1-e^2)}{1 + ecosv} \tag{2.31}$$

Poniżej przedstawiono obraz typowej orbity hiperbolicznej. Wynika z niego, że hiperbola ma dwa ramiona, które są asymptotyczne do przecinających się prostych. Przy założeniu, że ciało centralne znajduje się w lewym ognisku, można pominąć prawe ramię hiperboli, gdyż siła grawitacji nie działa odpychająco.



Rysunek 2.9: Właściwości orbity hiperbolicznej [15]

Hiperbola stanowi przekrój stożkowy. Jej ekscentryczność można określić korzystając z własności kierownicy.

$$e = \frac{c}{a} \tag{2.32}$$

Drogę statku kosmicznego poruszającego się po trajektorii hiperbolicznej obraca się o kąt równy kątowi asymptot δ, gdy napotyka ciało centralne. Kąt skrętu oblicza się następującym równaniem:

$$sin(\frac{\delta}{2}) = \frac{1}{e} \tag{2.33}$$

Jeśli znane są długość promienia, prędkość chwilowa i kąt nachylenia toru lotu w określonym czasie, można obliczyć ekscentryczność i półoś wielką orbity hiperbolicznej za pomocą wskazanych już wcześniej równań:

$$e = \sqrt{(\frac{rv^2}{GM} - 1)^2 cos^2\phi + sin^2\phi} \tag{2.34}$$

oraz

$$v = arccos(\frac{a(1-e^2) - r)}{e \cdot r}) \tag{2.35}$$

Jako że anomalia prawdziwa może przyjmować wartość między 0 a 2π, podczas gdy arccos generuje wartości między 0 a π, wykorzystuje się kąt toru nachylenia lotu ϕ, aby określić kwadrant. Jeśli ϕ jest wartością dodatnią, to v też jest dodatnie i odwrotnie. Jeżeli pomiędzy statkiem kosmicznym a ciałem centralnym nie ma odchylenia grawitacyjnego, to wprowadza się nowy parametr zwany parametrem zderzenia b, który określa odległość najbliższego zbliżenia (odległość między środkami obiektów, gdy są one styczne). Wzór na b wynosi:

$$b = \frac{-a}{tan(\frac{\delta}{2})} \tag{2.36}$$

Jeśli natomiast występuje odchylenie grawitacyjne, to statek kosmiczny i ciało centralne oddzielone są o odległość perygeum, co wyraża równanie:

$$r_p = a(1-e) \tag{2.37}$$

Parametr krzywej stożkowej można obliczyć wykorzystując poniższe równanie:

$$a = \frac{1}{2}(r_{min} + r_{max}) = \frac{1}{2}(\frac{p}{1+e} + \frac{1}{1-e}) = \frac{p}{1-e^2} = \frac{h^2}{GM(1-e^2)} \tag{2.38}$$

Po otrzymaniu prawdziwej anomalii wektor promienia, kąt nachylenia toru lotu i prędkość można obliczyć w sposób analogiczny jak w przypadku transferu eliptycznego. Czas lotu obliczono przy użyciu anomalii mimośrodowej:

$$T = \frac{2\pi}{n} = 2\pi\sqrt{\frac{a^3}{GM}} \tag{2.39}$$

Można zatem wprowadzić podobny parametr zwany hiperboliczną anomalią mimośrodową – F:

$$coshF = \frac{e + cosv}{1 + ecosv} \tag{2.40}$$

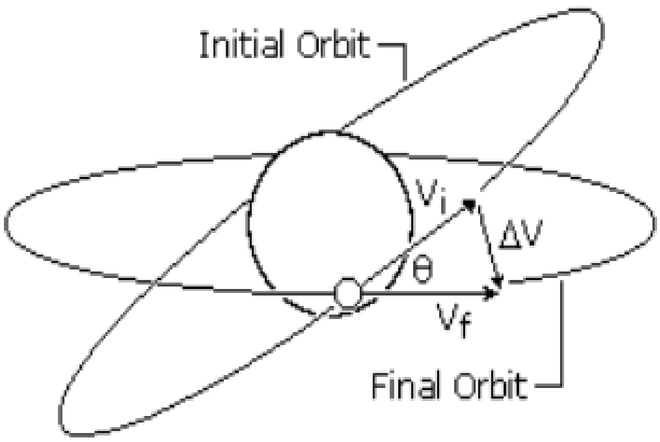
A z F można wyprowadzić czas lotu na orbicie hiperbolicznej w następujący sposób:

$$t - t_o = \sqrt{\frac{(-a)^3}{GM}}[(esinhF - F) - (esinhF_O - F_O)] \tag{2.41}$$

Elementy orbitalne orbity hiperbolicznej oblicza się w podobny sposób do orbity eliptycznej.

2.9 Zmiany nachylenia (inklinacji) płaszczyzny orbitalnej

Orbity ciał niebieskich w Układzie Słonecznym o tym samym ognisku zazwyczaj nie leżą na tej samej płaszczyźnie. Z tego powodu konieczne jest wykonywanie manewru zmiany płaszczyzny.



Rysunek 2.10: Zmiana płaszczyzny orbitalnej (initial orbit – orbita początkowa, final orbit – orbita końcowa) [15]

Zmiany płaszczyzny orbitalnej przeprowadza się zmieniając kierunek wektora prędkości. Kierunek prędkości ciała orbitującego jest zawsze styczny do orbity. Z tego powodu wszelkie zmiany płaszczyzny wymagają składowej delta-V prostopadłej do płaszczyzny orbity lub wektora prędkości. W sytuacji prostej zmiany płaszczyzny, w której żadne parametry orbitalne oprócz inklinacji nie ulegają zmianie oraz jeżeli długość początkowego i końcowego wektora prędkości jest równa, prawo cosinusów przyjmuje następującą postać:

$$\Delta v = 2V_i sin(\frac{\theta}{2}) \tag{2.42}$$

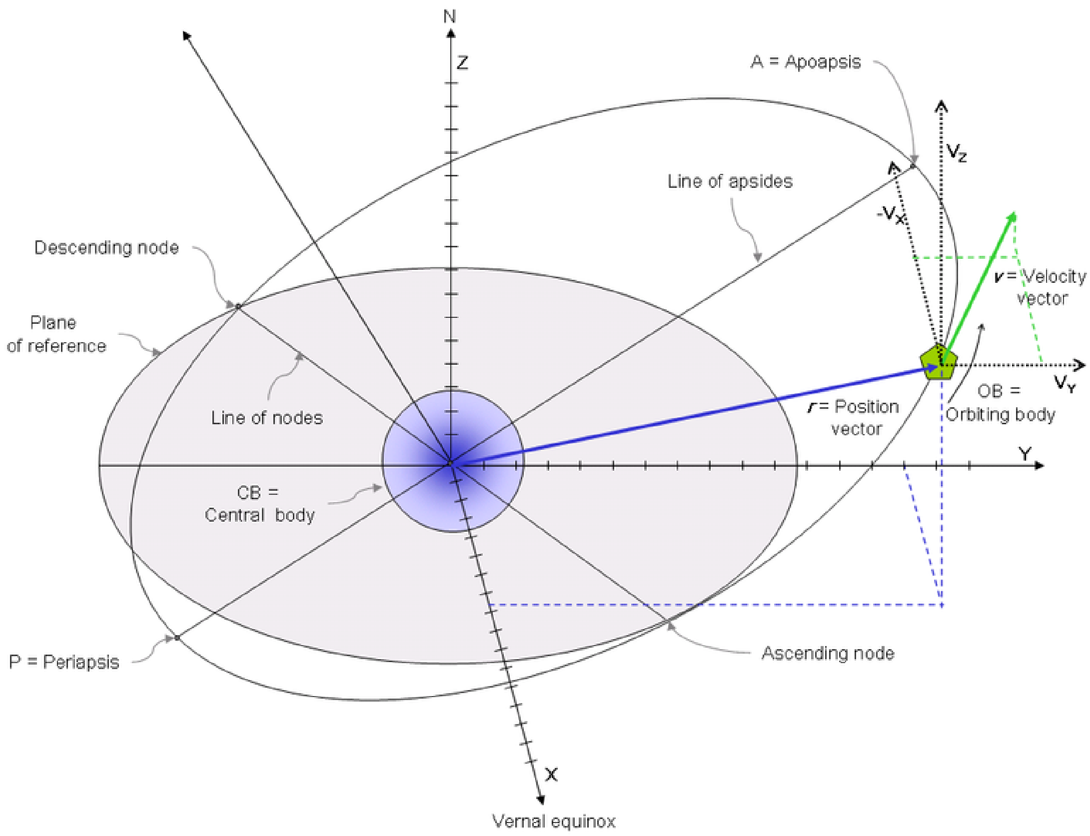
W przypadku zmian płaszczyzny, gdy wszystkie parametry orbitalne są różne, a także prędkości początkowa i końcowa nie są takie same, całkowitą wymaganą zmianę prędkości oblicza się następująco:

$$\Delta v = \sqrt{v_i^2 + v_f^2 - 2v_iv_fcos\theta)} \tag{2.43}$$

Z powyższych równań wynika, że manewry zmiany płaszczyzny są bardzo kosztowne i np. prosta zmiana płaszczyzny o $60°$ wymaga już delty V równej prędkości początkowej [15], dlatego manewry orbitalne powinny być wykonywane w apogeum, gdyż prędkość początkowa jest w tym punkcie minimalna.

2.10 Wektory stanu orbitalnego i właściwy moment pędu

Wektory stanu orbitalnego są kartezjańskimi wektorami położenia r i prędkości v. Wraz z czasem (epoka) t określają stan orbitującego ciała. [16] Wektory stanu definiuje się w oparciu o jakiś układ odniesienia. W tej pracy układ wyśrodkowany jest na Słońcu.



Rysunek 2.11: Wektory stanu orbitalnego (position vector – wektor położenia, velocity vector – wektor prędkości, plane of reference – płaszczyzna odniesienia, periapsis – perycentrum, apoapsis – apocentrum, central body – ciało centralne, orbiting body - ciało orbitujące, line of apsides – linia apsyd, descending node – węzeł zstępujący, ascending node – węzeł wstępujący, line of nodes – linia węzłow) [10]

Wektor położenia **r** opisuje położenie ciała w wybranym układzie odniesienia, a wektor prędkości opisuje jego prędkość w tym samym układzie w tym samym czasie.

Wektorów stanu można używać do obliczenia właściwego momentu pędu **h** (*specific angular momentum*). W przypadku dwóch ciał orbitujących jest to iloczyn wektora względnego położenia **r** i wektora względnej prędkości **v** podzielony przez masę ciała:

$$\mathbf{h} = \mathbf{r} \times \mathbf{v} = \frac{\mathbf{L}}{m} \tag{2.44}$$

gdzie $\mathbf{L}$ jest wektorem momentu pędu i wynosi $\mathbf{r}\times m\mathbf{v}$. Właściwy moment pędu można wykorzystać do obliczenia długości węzła wstępującego Ω w następujący sposób:

$$n = k \times h = (-h_y, h_x, 0) \tag{2.45}$$

$$\Omega = \begin{cases} arccos\frac{n_x}{|n|}, n_y \geq 0; \\ 2\pi - arccos\frac{n_x}{|n|}, n_y < 0. \end{cases} \tag{2.46}$$

$n =< n_x, n_y, n_z >$ jest tutaj wektorem skierowanym w stronę węzła wstępującego. Przyjmuje się, że płaszczyzną odniesienia jest płaszczyzna xy, a początkiem długości jest dodatnia oś x. k to wektor jednostkowy (0,0,1), który stanowi wektor normalny płaszczyzny odniesienia xy. Dla orbit nienachylonych (o nachyleniu równym zero) Ω jest niezdefiniowana. Do obliczeń przyjmuje się, że jest ona równa zeru; oznacza to, że węzeł wstępujący jest umieszczony w kierunku odniesienia, co jest równoznaczne z tym, że n jest skierowany w stronę dodatniej osi x [5]

2.11 Epoka

Epoka to moment w czasie używany jako punkt odniesienia dla pewnej zmiennej w czasie wielkości astronomicznej, takiej jak współrzędne niebieskie lub elementy orbity eliptycznej ciała niebieskiego, ponieważ podlegają one perturbacjom i zmieniają się w czasie. Głównym zastosowaniem tak określonych wielkości astronomicznych jest obliczanie innych istotnych parametrów ruchu, w celu przewidywania przyszłego położenia i prędkości [12] Epokę zwykle podaje się wraz ze współrzędnymi równikowymi, ponieważ współrzędne równikowe obiektów niebieskich zmieniają się powoli wraz z upływem czasu. Obecnie najczęściej używaną epoką jest J2000.0 (rok juliański 2000.0), która (prawie dokładnie) odpowiada 1 stycznia 2000r., godz. 12:00 uniwersalnego czasu koordynowanego (UTC). W tej pracy jako punkt odniesienia przyjęto zmodyfikowaną wersję daty juliańskiej zwaną MJD (Modified Julian Date), którą otrzymano przez odjęcie 2 400 000,5 dni od daty juliańskiej.

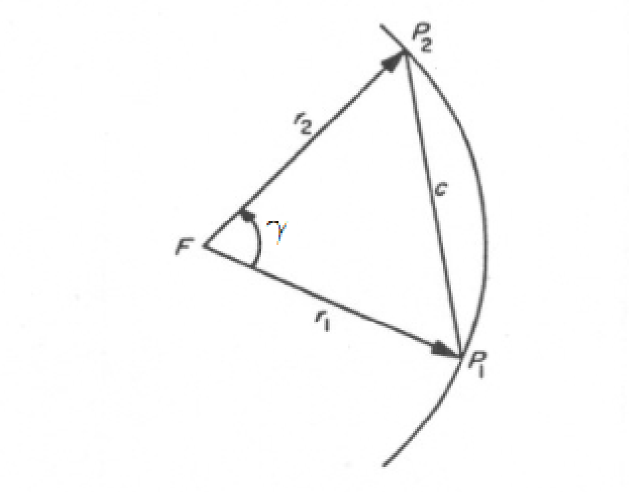
$$MJD \equiv JD - 2400000,5 \tag{2.47}$$

MJD podaje zatem liczbę dni, które upłynęły od północy 17 listopada 1858 roku. Data ta odpowiada 2400000,5 dnia po dniu 0 kalendarza juliańskiego. Należy jednak zachować ostrożność przy konwersji na inne jednostki czasu, ze względu na przesunięcie o pół dnia (w przeciwieństwie do daty juliańskiej zmodyfikowana data juliańska jest odnoszona do północy, a nie do południa) oraz z powodu wstawiania półrocznych sekund przestępnych (które są wstawiane o północy) [8]

2.12 Problem Lamberta

W założeniu, że ciało pod wpływem siły grawitacji ciała centralnego przebywa drogę z punktu P_1 na swojej eliptycznej trajektorii do punktu P_2 w czasie T, czas lotu tego ciała opisuje twierdzenie Lamberta, które mówi, że:

Czas transferu ciała, poruszającego się między dwoma punktami na trajektorii eliptycznej jest funkcją jedynie sumy odległości obu punktów od początku działania siły, odległości liniowej między punktami oraz półosi wielkiej stożka.[17]



Rysunek 2.12: Przedstawienie geometryczne twierdzenia Lamberta (γ - kąt przebyty w czasie T) [17]

Wyraża się ono funkcją:

$$T = T(r_1 + r_2, c, a) \tag{2.48}$$

gdzie:
a – półoś wielka
r_1 – wektor FP_1
r_2 – wektor FP_2
c – odległość między P_1 i P_2

Algorytm do rozwiązywania problemu Lamberta używany jest głównie do projektowania misji międzyplanetarnych oraz do wyznaczania (oszacowania) orbit obiektów w przestrzeni kosmicznej.

Rozdział 3

Algorytmy do wyznaczania transferów orbitalnych

3.1 Dane wejściowe

W celu zainicjowania procesu wyznaczania transferów orbitalnych należy określić następujące parametry planety startu oraz docelowej asteroidy:

- ciało centralne - ciało, wokół którego orbituje obiekt
- apocentrum - maksymalna odległość ciała na orbicie od ciała centralnego wyrażona w [km]
- perycentrum - minimalna odległość ciała na orbicie od ciała centralnego wyrażona w [km]
- ekscentryczność - określa kształt orbity [-]
- inklinacja orbity - kąt pomiędzy płaszczyzną orbitalną a płaszczyzną odniesienia wyrażony w [°]
- argument perycentrum - kąt między płaszczyzną odniesienia a perycentrum orbity wyrażony w [°]
- długość węzła wstępującego - kąt między kierunkiem obserwacji a węzłem wstępującym wyrażony w [°]
- epoka definiująca anomalię średnią - czas, w którym zmierzono anomalię średnią wyrażony w [MJD]
- anomalia średnia - kąt określający położenie asteroidy wyrażony w [°]
- masa planety startu - wyrażona w [kg], stała ta dla Ziemi zawarta jest w bibliotece astropy[3].

Parametry te pozwalają określić położenie obiektu w przestrzeni w danym momencie w czasie.

Masę asteroidy docelowej można pominąć. W porównaniu z masą ciała centralnego jest ona nieistotna - jej strefa wpływów jest zbyt mała, by statek kosmiczny został przez nią przejęty.

Do analizy wymagana jest również masa ciała centralnego, stałą tą dla Słońca można znaleźć w bibliotece astropy.

Następnie trzeba zdefiniować przedział czasowy, czyli czas rozpoczęcia i zakończenia analizy podane w TimeISOT według biblioteki astropy[13], "RRRR-MM-DDTGG:MM:SS.sss" (rok, miesiąc, dzień, godzina, minuta, sekunda, ułamek sekundy).

3.2 Układ odniesienia

Jako środek układu odniesienia przyjęto ciało centralne. Kierunek obserwacji (oś x) określa perycentrum planety startu. Płaszczyznę obserwacji (płaszczyzna xy) stanowi płaszczyzna orbity planety startu. Założono, że obiekty poruszają się po orbitach w kierunku przeciwnym do ruchu wskazówek zegara.

3.3 Lokalna pozycja w czasie - anomalia prawdziwa

Znając położenie obiektu w przestrzeni w danym momencie w czasie - stan odniesienia obiektu, można wyznaczyć położenie tego obiektu w innym momencie w czasie. Korzystając Praw Keplera, opisanych w podrozdziale 2.1, stworzono algorytm find_true_anomaly(), aby obliczyć anomalię prawdziwą oraz lokalny wektor położenia obiektu.

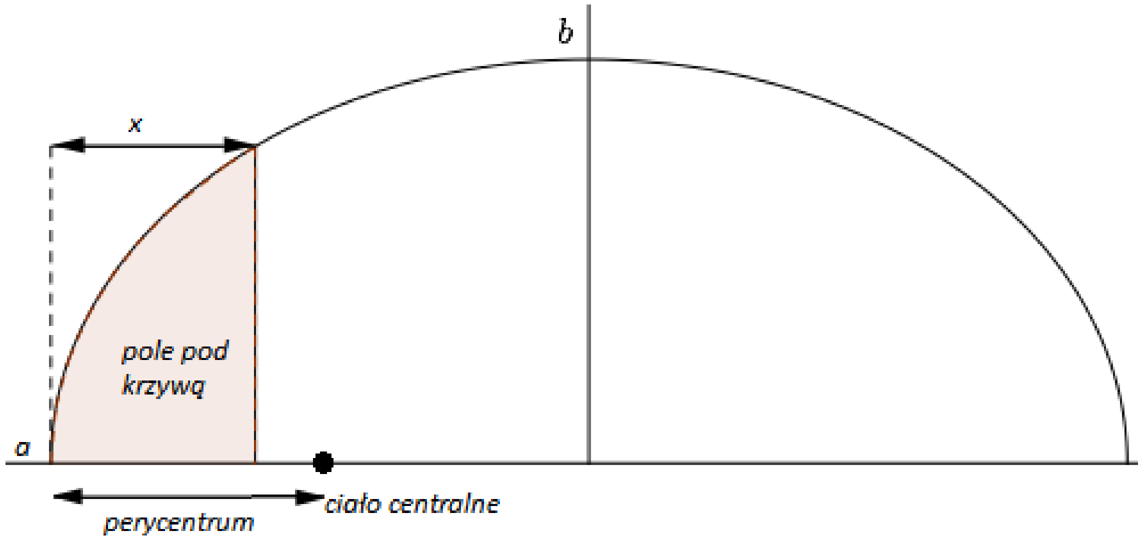
Jako środek lokalnego układu odniesienia przyjęto perycentrum. Oś x leży wzdłuż linii osi wielkiej orbity. Kierunek obrotu obiektu jest zgodny z ruchem wskazówek zegara.

W liniach kodu od 193 do 219 zdefiniowano parametry początkowe:

- time_current - aktualny czas
- half_ellipse_area - powierzchnia półelipsy
- P - okres orbity
- time_at_periapsis - czas w perycentrum
- unit_time - reszta z dzielenia przez okres różnicy czasu aktualnego od czasu w perycentrum
- time0 - czas przebycia połowy orbity
- which_half_of_orbit - precyzuje, która połowa orbity (pierwsza połowa - od perycentrum do apocentrum, druga - od apocentrum do perycentrum)
- swept_area - pole zakreślone w czasie przebytym od perycentrum
- X_max - długość osi wielkiej - definiuje badany zakres współrzędnej x
- rotation_point - punkt obrotu (punkt określający położenie ciała centralnego na osi x)
- error - zapisuje zmienną x, jej dokładność oraz po której stronie prawdziwej wartości x znajduje się przewidziana wartość x
- accuracy - zadana dokładność wartości x
- x_previous - poprzednia wartość x

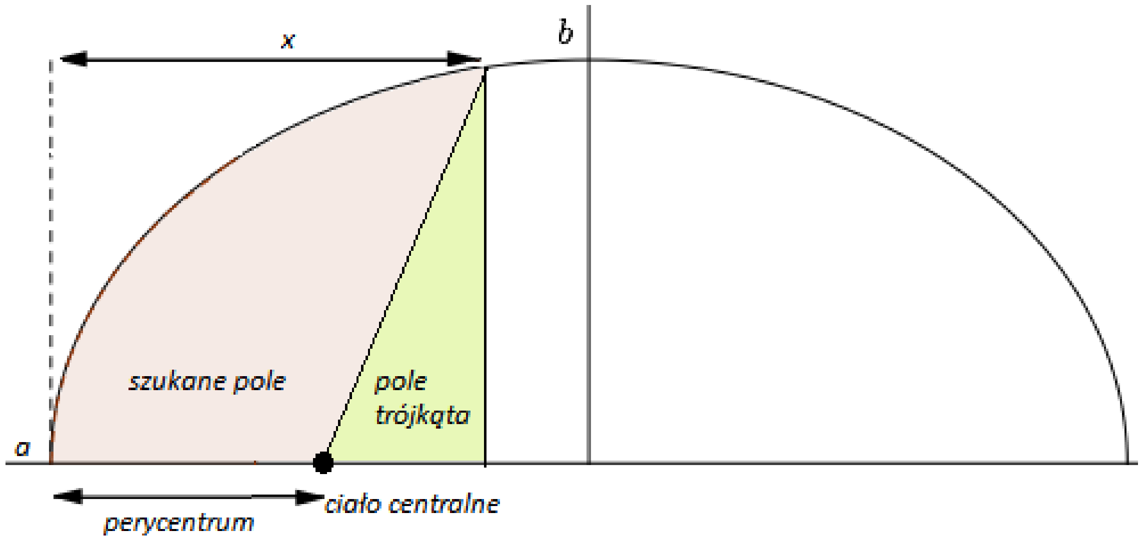
Pętla od 210 do 248 linii kodu oblicza wartość współrzędnej x w następujących krokach:

1. Wywołuje funkcję, która dla wyznaczonej wartości x oblicza pole pod krzywą, będącą częścią połowy elipsy orbity jak na rys. 3.1.

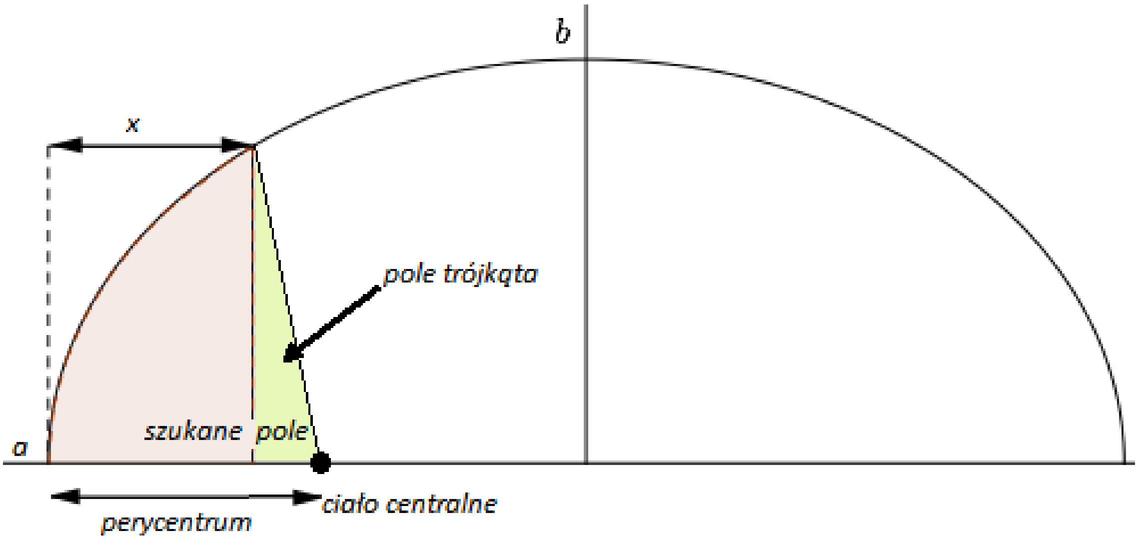


Rysunek 3.1: Pole pod krzywą (a - półoś wielka, b - półoś mała, x - przewidywana współrzędna) [11]

2. Jeżeli przewidywana współrzędna x leży po prawej stronie punktu obrotu (opcja 1), to obliczane jest pole trójkąta pokazanego na rys. 3.2 W przeciwnym przypadku współrzędna x leży po lewej stronie (opcja 2) więc obliczane jest pole trójkąta przedstawionego na rys. 3.3



Rysunek 3.2: Szukane zakreślone pole oraz pole trójkąta - x > punkt obrotu (a - półoś wielka, b - półoś mała, x - przewidywana współrzędna)



Rysunek 3.3: Szukane zakreślone pole oraz pole trójkąta - x < punkt obrotu (a - półoś wielka, b - półoś mała, x - przewidywana współrzędna)

3. W opcji 1 od wyznaczonego w kroku 1 pola pod krzywą odejmuje się pole trójkąta oznaczone na rys. 3.2 oraz pole zakreślone w czasie przebytym od perycentrum.

   W opcji 2 do wyznaczonego w kroku 1 pola pod krzywą dodaje się pole trójkąta oznaczone na rys. 3.3 a następnie odejmuje się pole zakreślone w czasie przebytym od perycentrum.

   Jeżeli wynik działań w obu przypadkach jest większy od 0, to zwiększana jest dokładność zmiennej x i przewidywana współrzędna przesuwana jest w lewo.

   W przeciwnym wypadku, gdy wynik działań w obu przypadkach jest mniejszy od 0, to po zwiększeniu dokładności zmiennej x przesuwa się przewidywaną współrzędną w prawo.

4. Jeżeli wartość bezwzględna z różnicy poprzedniej oraz przewidywanej wartości współrzędnej x jest mniejsza od zadanej dokładności, przerywa się pętlę. W sytuacji przeciwnej powraca się do punktu 1.

W liniach kodu od 192 do 289 dokonywane są obliczenia anomalii prawdziwej, lokalnych współrzędnych x i y oraz promienia orbity.

Jeżeli wyznaczona w pętli i umiejscowiona w pierwszej połowie orbity wartość współrzędnej znajduje się po prawej stronie punktu obrotu, to:

1. Anomalia prawdziwa wyraża się wzorem:

$$\nu = \pi - arctan(\frac{y}{x}) \tag{3.1}$$

2. Funkcja find_true_anomaly() zwraca wyliczone ν i r (wzór 3.5), lokalne wartości x i y oraz czas, który upłynął od przejścia przez perycentrum.

Jeżeli wyznaczona w pętli i umiejscowiona w drugiej połowie orbity wartość współrzędnej znajduje się po prawej stronie punktu obrotu, to:

1. Anomalia prawdziwa wyraża się wzorem:

$$\nu = 2\pi - arctan(\frac{y}{x}) \tag{3.2}$$

2. Funkcja find_true_anomaly() zwraca wyliczone ν i r, lokalne wartości x i y, gdzie y przyjmuje wartość ujemną oraz czas, który upłynął od przejścia przez perycentrum.

Jeżeli wyznaczona w pętli i umiejscowiona w pierwszej połowie orbity wartość współrzędnej znajduje się po lewej stronie punktu obrotu, to:

1. Anomalia prawdziwa wyraża się wzorem:

$$\nu = arctan(\frac{y}{x}) \tag{3.3}$$

2. Funkcja find_true_anomaly() zwraca wyliczone ν i r, lokalne wartości x i y oraz czas, który upłynął od przejścia przez perycentrum.

Jeżeli wyznaczona w pętli i umiejscowiona w drugiej połowie orbity wartość współrzędnej znajduje się po lewej stronie punktu obrotu, to:

1. Anomalia prawdziwa wyraża się wzorem:

$$\nu = \pi + arctan(\frac{y}{x}) \tag{3.4}$$

2. Funkcja find_true_anomaly() zwraca wyliczone ν i r, lokalne wartości x oraz y, gdzie y przyjmuje wartość ujemną oraz czas, który upłynął od przejścia przez perycentrum.

Promień orbity opisuje równanie:

$$r = \frac{x}{cos(\nu)} \tag{3.5}$$

3.4 Globalna pozycja w czasie

3.4.1 Wektor położenia

Funkcja position_vector() zwraca wektor położenia w zadanym momencie czasu.

Wykorzystując lokalne współrzędne x i y (związane z linią apsyd orbity oraz jej płaszczyzną) otrzymane z funkcji find_true_anomaly() z podrozdziału 3.3 dokonano ich obrotu o kolejne kąty, aby przekształcić je do globalnego układu współrzędnych opisanego w podrozdziale 3.2. Najpierw wykonano obrót o argument perycentrum wokół osi z, następnie o inklinację wokół osi, wyznaczonej przez linię węzłów.

3.4.2 Wektor prędkości

Funkcja velocity_vector() zwraca wektor prędkości dla danego wektora położenia.

Aby wyznaczyć wektor prędkości, wymagane są dwie pozycje obiektu: pierwsza - w zadanym momencie czasu, a druga - po niewielkim odstępie czasowym. Między tymi punktami w przestrzeni wyznaczono wektor o punkcie zaczepienia w położeniu pierwszym. Ostatecznie podzielono ten wektor przez odstęp czasowy.

3.4.3 Właściwy moment pędu i węzeł wstępujący

Funkcja specific_angular_momentum() zwraca właściwy moment pędu.

Jest on iloczynem wektorowym wektora położenia i wektora prędkości.

Funkcja position_of_ascending_node() zwraca położenie następnego węzła wstępującego.

Stanowi on iloczyn wektorowy skierowanego w górę (składowa z jest dodatnia) wektora prostopadłego powierzchni odniesienia i wektora właściwego momentu pędu.

Funkcja angle_to_ascending_node() zwraca kąt między aktualną pozycją a pozycją następnego węzła wstępującego [18] .

3.5 Znalezienie czasu po obrocie o zadany kąt

Funkcja time_after_travel_by_angle() zwraca czas (datę), w którym obiekt znajdzie się po obrocie o zadany kąt.

Aktualną anomalię prawdziwą otrzymano z funkcji find_true_anomaly(). Wywołano funkcję time_after_travel_by_angle_find_angle(), która zwraca czas, wymagany do obrotu o zadany kąt. Otrzymany czas dodaje się do czasu aktualnego, co stanowi czas po obrocie o zadany kąt.

Poniżej przedstawiono działanie funkcji time_after_travel_by_angle_find_angle().

1. Jeżeli kąt jest mniejszy od 90 [°], to należy wyznaczyć szukane pole według rys. 3.3 (x otrzymano mnożąc odległość od ciała centralnego przez cosinus kąta)
2. Jeżeli kąt jest większy od 90 [°] a mniejszy od 180 [°], to należy wyznaczyć szukane pole według rys. 3.2.
3. Jeżeli kąt jest większy od 180 [°] a mniejszy od 270 [°], to należy wyznaczyć szukane pole według rys. 3.2.
4. Jeżeli kąt jest większy od 270 [°] a mniejszy od 360 [°], to należy wyznaczyć szukane pole według rys. 3.3.
5. Jeżeli kąt jest większy od 360 [°], to wymagany czas obrotu należy powiększyć o jeden okres orbitalny i wywołać na nowo funkcję z zadanym kątem, pomniejszonym o 360 [°].
6. Funkcja zwraca czas, otrzymany z II prawa Keplera, mnożąc szukane pole przez połowę okresu i dzieląc przez połowę pola orbity.

3.6 Znalezienie dat przejścia przez węzły wstępujące

Funkcja find_times_of_ascending_node() zwraca daty przejścia obiektu przez węzeł wstępujący w zadanym okresie czasu, czyli okna transferowe, korzystając z funkcji position_of_ascending_node(), angle_to_ascending_node() oraz time_after_travel_by_angle().

Każda kolejna data jest wyznaczana w sposób następujący:

1. Za pomocą funkcji position_of_ascending_node() należy obliczyć odległość następnego węzła wstępującego od ciała centralnego.
2. Używając funkcji angle_to_ascending_node() należy obliczyć kąt między aktualnym położeniem obiektu a położeniem następnego węzła wstępującego.
3. Podstawiając wyliczone wyżej wartości do funkcji time_after_travel_by_angle() otrzymano datę przejścia obiektu przez następny węzeł wstępujący.

3.7 Wyznaczenie transferu między dwoma obiektami o zadanym czasie odlotu

Funkcja transfer() przy użyciu funkcji time_of_transfer(), planetary_departure() i planetary_rendezvous() zwraca wszystkie dane transferowe:

- czas odlotu
- β odlotu [°] (2.12)
- Δv hiperboli odlotu [km/s]
- ΔV odlotu [km/s]
- czas przylotu
- β przylotu [°] (2.12)
- Δv hiperboli przylotu [km/s]
- ΔV przylotu [km/s]
- Δv transferu [km/s]
- czas transferu [dni]

Korzystając z funkcji lambert(), zawartej w bibliotece poliastro [14] funkcja time_of_transfer() zwraca całkowity czas transferu o wartości bliskiej najmniejszemu Δv w zadanym okresie czasu.

Wywołując wielokrotnie tą funkcję z progresywnie zmniejszającym się zadanym okresem następuje zbliżenie do całkowitego czasu transferu z minimalnym Δv.

Po podstawieniu znalezionego czasu ponownie do funkcji lambert() otrzymano parametry orbity transferowej.

Jeżeli obiekt startu posiada strefę wpływów, której nie można pominąć, to należy wywołać funkcję planetary_departure(), która zwraca Δv hiperboli odlotu oraz β odlotu. Następnie trzeba wyznaczyć Δv, wymagane do zmiany nachylenia płaszczyzny orbity.

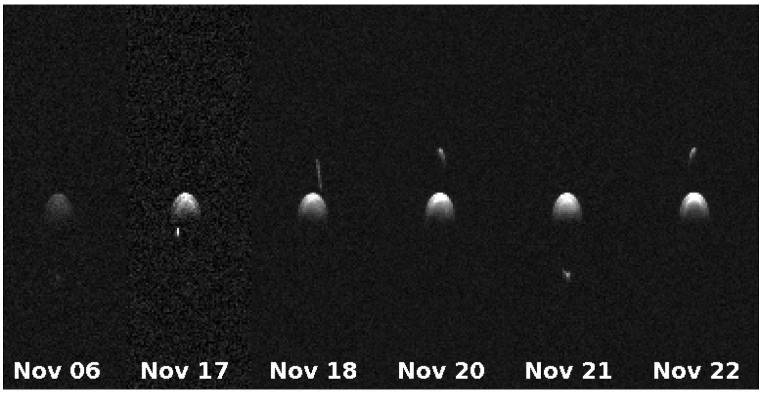
W przeciwnym wypadku, za Δv odlotu należy podstawić Δv odlotu, wyznaczoną przez funkcję lambert().

Jeżeli obiekt docelowy posiada strefę wpływów, której nie można pominąć, to należy wywołać funkcję planetary_rendezvous(), która zwraca Δv hiperboli przylotu oraz β przylotu.

W przeciwnym wypadku, za Δv przylotu należy podstawić Δv przylotu, wyznaczoną przez funkcję lambert().

3.8 Analiza misji Ziemia - asteroida 1996FG3

175706 (1996 FG3) to średniej wielkości asteroida, której orbita przecina orbitę Ziemi. Krąży wokół Słońca co 395 dni (1,08 roku), zbliżając się na odległość 0,69 AU i oddalając od Słońca na odległość 1,42 AU. 1996 FG3 ma około 1,2 kilometra średnicy, co czyni ją większą od 99% planetoid. Zaobserwowano ruch obrotowy 1996 FG3. Wykonuje ona obrót wokół własnej osi co 3,59 godziny. Z analizy spektralnej wynika, że prawdopodobnie występują tam woda, żelazo, nikiel, kobalt, azot i amoniak. NASA JPL sklasyfikowała 1996FG3 jako "Potencjalnie niebezpieczną asteroidęź powodu jej przewidywanego zbliżenia do Ziemi [1].



Rysunek 3.4: 1996FG3 i jej księżyc - obrazy radarowe Delay-Doppler z Obserwatorium Arecibo z 2011 r. [2]

Wartości parametrów 1996FG3 [9]:

- ciało centralne - Słońce
- apocentrum - 212728172 [km]
- perycentrum - 102474541 [km]
- ekscentryczność - 0.35 [-]
- inklinacja orbity - 2 [°]
- argument perycentrum - 24.08 [°]
- długość węzła wstępującego - 299.6764 [°]
- epoka definiująca anomalię średnią - 59600 [MJD]
- anomalia średnia - 202.32 [°]

W tabeli 3.1 przedstawiono potencjalne transfery z Ziemi do asteroidy 1996FG3 w przedziale czasowym od 2027-01-01 00:00:00 do 2033-01-01 00:00:00.

Natomiast tabela 3.2 przedstawia potencjalne transfery z asteroidy 1996FG3 do Ziemi w przedziale czasowym od 2027-10-18 00:00:00 do 2033-11-01 00:00:00.

Z przeprowadzonej analizy wynika, że misją o najmniejszym Δv jest misja o parametrach zaprezentowanych w rzędach zaznaczonych kolorem zielonym.

TABLICA 3.1: Transfer z Ziemia do 1996FG3 (wynik „None" oznacza: „nie dotyczy")

czas odlotu	β **odlotu** [°] **2.12**	Δv **hiperboli odlotu** [km/s]	ΔV **odlotu** [km/s]	**czas przylotu**	β **przylotu** [°] **2.12**	Δv **hiperboli przylotu** [km/s]	ΔV **przylotu** [km/s]	Δv **transferu** [km/s]	**czas transferu** [dni]
2033-04-08 16:54:27.355	89.15	56.01	[-26.58, 4.51, -32.45]	2033-09-25 05:00:37.155	None	None	[28.12, 16.6, 3.54]	131.05	169.5
2032-06-30 21:38:42.773	84.96	19.46	[-0.0, 0.23, -0.38]	2033-06-23 08:14:17.801	None	None	[6.15, -0.21, -1.53]	26.23	357.44
2031-09-15 04:44:50.535	86.75	25.61	[0.06, -0.01, -0.0]	2032-05-24 10:38:59.851	None	None	[6.11, 1.95, 0.85]	32.13	252.25
2031-06-12 22:51:21.372	89.53	78.15	[-0.04, 0.13, -2.84]	2031-11-25 05:25:08.020	None	None	[0.83, -2.92, 2.08]	84.68	165.27
2031-04-09 04:38:29.915	89.12	55.02	[-28.4, 4.82, -30.61]	2031-07-28 07:03:21.199	None	None	[25.14, -0.99, 1.53]	122.25	110.1
2030-07-01 09:22:45.333	85.01	19.59	[-0.0, 0.22, -0.34]	2031-04-25 12:21:15.520	None	None	[3.11, -0.68, -1.6]	23.57	298.12
2029-09-14 16:28:53.095	86.13	22.99	[0.06, -0.01, -0.07]	2030-03-26 11:06:09.033	None	None	[1.36, -1.42, 0.68]	25.16	192.78
2029-06-12 10:35:23.933	89.51	76.03	[-0.0, 0.0, -0.11]	2029-11-13 07:30:11.481	None	None	[4.14, 1.72, 1.04]	80.74	153.87
2029-04-08 16:22:32.476	89.1	54.48	[-30.22, 5.13, -28.76]	2029-05-29 09:05:57.005	None	None	[26.92, 9.53, 0.15]	125.07	50.7
2028-06-30 21:06:47.894	90.0	69236.12	[-0.0, 0.0, -0.76]	2028-10-31 05:35:51.644	None	None	[5.52, 8.88, 1.06]	69247.39	122.35
2027-12-08 18:17:50.414	86.4	24.07	[-0.0, -0.0, 0.0]	2028-10-16 05:49:42.914	None	None	[4.62, -7.75, 1.17]	33.17	312.48
2027-05-02 22:29:58.556	89.54	78.49	[-0.0, 0.0, -0.97]	2027-10-17 18:39:51.378	None	None	[1.62, 1.89, 0.77]	82.07	167.84

TABLICA 3.2: Transfer z 1996FG3 do Ziemia (wynik „None” oznacza: „nie dotyczy”)

czas odlotu	β **odlotu** [°] **2.12**	Δv **hiperboli odlotu [km/s]**	ΔV **odlotu [km/s]**	**czas przylotu**	β **przylotu** [°] **2.12**	Δv **hiperboli przylotu [km/s]**	ΔV **przylotu [km/s]**	Δv **transferu [km/s]**	**czas transferu [dni]**
2033-07-13 15:38:37.606	None	None	[-14.74, -75.78, -2.59]	2033-11-15 19:30:58.433	85.68	21.47	None	98.71	125.16
2032-12-08 01:28:06.814	None	None	[-25.19, -4.91, -2.11]	2033-12-13 01:27:49.510	86.69	25.36	None	51.11	370.0
2032-02-28 23:43:02.422	None	None	[-4.58, -1.32, -0.01]	2032-11-04 10:35:29.126	85.77	21.75	None	26.51	249.45
2031-05-15 17:41:21.786	None	None	[-29.9, -44.0, -0.4]	2031-07-01 16:05:32.750	63.21	6.13	None	59.34	46.93
2030-10-10 03:30:50.993	None	None	[11.72, 4.43, -2.6]	2030-12-30 21:33:21.121	86.79	25.8	None	38.6	81.75
2029-12-31 01:45:46.602	None	None	[-2.52, 0.27, -0.02]	2030-10-31 14:17:35.586	86.27	23.53	None	26.07	304.52
2029-03-16 19:44:05.965	None	None	[-34.66, -38.3, -0.95]	2029-07-01 07:33:53.422	81.2	13.48	None	65.15	106.49
2028-08-11 05:33:35.172	None	None	[17505.66, 7868.59, 667.15]	2028-08-11 11:21:19.528	90.0	17269.71	None	36474.08	0.24
2027-09-12 09:14:15.688	None	None	[-2.23, 7.44, -1.05]	2028-06-30 21:16:30.820	86.66	25.2	None	33.03	292.5
2027-03-15 09:53:13.071	None	None	[-55.24, -12.15, -1.15]	2027-12-14 11:26:45.656	86.42	24.15	None	80.72	274.06

Rozdział 4

Konkluzje i rekomendacje

Głównym celem niniejszej pracy było stworzenie narzędzia do analizy trajektorii lotu misji między ciałami niebieskimi, a w szczególności do wyznaczania transferów orbitalnych statków kosmicznych od planety do asteroidy i z powrotem. Stworzony program w swoim zastosowaniu wykracza poza założone ramy. Oprócz wyznaczania transferów Ziemia - asteroida, umożliwia wyznaczanie transferów międzyplanetarnych oraz między asteroidami. Po wprowadzaniu parametrów innych układów słonecznych, pozwala również wyznaczać transfery w tych układach.

Porównanie wyników, zwróconych przez program prowadzi do praktycznych wniosków i zastosowań. Z wyszukanych wartości delta-v można wybrać najniższą, którą w konsekwencji wykorzystuje się do określenia kosztu paliwa. Możliwe jest także dobranie najkrótszego czasu transferu. Zależnie od wymagań misji można dostosować długość pobytu na docelowym ciele niebieskim w celu realizacji zamierzonych zadań.

Zaprojektowany w pracy algorytm nie ma wszystkich możliwości profesjonalnego oprogramowania, może być jednak skutecznie używany do wstępnej analizy misji.

Program generuje szerokie pole dalszego rozwoju i możliwych przyszłych zastosowań, między innymi:

- Stworzenie interfejsu graficznego w celu uproszczenia procesu wprowadzania danych i odczytu wyników.
- Graficzna reprezentacja transferu.
- Umożliwienie zapisu i wczytywania parametrów orbitalnych obiektów.
- Zaimplementowanie bardziej złożonych manewrów transferowych.
- Skrócenie czasu przetwarzania danych wyjściowych.
- Stworzenie funkcji obliczającej ilość wymaganego paliwa dla misji.
- Opcja i zautomatyzowanie procesu planowania podróży wieloetapowych.

Literatura

[1] 175706 (1996 fg3). [on-line] https://www.spacereference.org/asteroid/175706-1996-fg3.

[2] (175706) 1996 fg3. [on-line] https://en.wikipedia.org/wiki/(175706)_1996_FG3.

[3] astropy.constants. [on-line] https://docs.astropy.org/en/stable/constants/index.html.

[4] Elementy orbitalne. [on-line] https://boyce-astro.org/wp-content/uploads/BRIEF-Video-Lesson-TIME-What-is-an-Epoch.pdf.

[5] Longitude of the ascending node. [on-line] https://en.wikipedia.org/wiki/Longitude_of_the_ascending_node.

[6] Mean anomaly. [on-line] https://en.wikipedia.org/wiki/Mean_anomaly.

[7] Mean anomaly. [on-line] http://www.castor2.ca/03_Mechanics/02_Elements/06_Mean_Anom/index.html.

[8] Modified julian date. [on-line] https://scienceworld.wolfram.com/astronomy/ModifiedJulianDate.html).

[9] Near-earth objects coordination centre. [on-line] https://neo.ssa.esa.int/search-for-asteroids.

[10] Orbital state vectors. [on-line] https://en.wikipedia.org/wiki/Orbital_state_vectors#/media/File:Orbital_state_vectors.png.

[11] Pole pod krzywą. [on-line] https://math.stackexchange.com/questions/1906439/ellipse-segment-area-of-tank.

[12] Time what is an epoch. [on-line] https://boyce-astro.org/wp-content/uploads/BRIEF-Video-Lesson-TIME-What-is-an-Epoch.pdf.

[13] Timeisot. [on-line] https://docs.astropy.org/en/stable/api/astropy.time.TimeISOT.html#astropy.time.TimeISOT.

[14] Traveling through space: solving the lambert problem. [on-line] https://docs.poliastro.space/en/stable/quickstart.html#traveling-through-space-solving-the-lambert-problem.

[15] Braeunig, R. A. Orbital mechanics. [on-line] http://www.braeunig.us/space/index.htm.

[16] Curtis., H. D. *Orbital Mechanics for Engineering Students*. ButterworthHeinemann, 2020.

[17] Jordo, J. F. *The Application of Lambert's Theorem to the Solution of Interplanetary Transfer Problems*. Jet Propulsion Laboratory, 1964.

[18] Liang, X. Find angle between two vectors along a given direction (anti-clockwise or clockwise). [on-line] https://adndevblog.typepad.com/manufacturing/2012/06/find-angle-between-two-vectors-along-a-given-direction-anti-clockwise-or-clockwise.html, 2012.

[19] Sonter, M. J. The technical and economic feasibility of mining the near-earth asteroids, master of science (hons.) thesis. [on-line] http://ro.uow.edu.au/theses/2862, 1996.

[20] Vallado, D. A. *Fundamentals of Astrodynamics and Applications*. Microcosm Press, 2013.

Dodatek A

Kod źródłowy

```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 13 13:55:29 2021

@author: Szogunator
"""

import numpy as np
import math
import sys
from scipy.integrate import quad
from astropy.time import Time
from astropy import units
from astropy.constants import G  #Gravitational consatant
from astropy.constants import M_earth, R_earth
from astropy.constants import M_sun, R_sun
from poliastro.twobody import Orbit
from poliastro.maneuver import Maneuver
from poliastro.bodies import Sun


def days_to_seconds(time):
    return time*24*60*60
def seconds_to_days(time):
    return time/60/60/24

def rotate_point(point_vector, rotation_axis_unit_vector, angle):
    costau = np.cos(angle)
    sintau = np.sin(angle)
    rotation_matrix = np.array([
            [costau+rotation_axis_unit_vector.x**2*(1-costau), rotation_axis_unit_vector.x*rotation_axis_unit_vector.y*(1-costau)-rotation_axis_unit_vector.z*sintau, rotation_axis_unit_vector.x*rotation_axis_unit_vector.z*(1-costau)+rotation_axis_unit_vector.y*sintau],
            [rotation_axis_unit_vector.x*rotation_axis_unit_vector.y*(1-costau)+rotation_axis_unit_vector.z*sintau, costau+rotation_axis_unit_vector.y**2*(1-costau), rotation_axis_unit_vector.y*rotation_axis_unit_vector.z*(1-costau)-rotation_axis_unit_vector.x*sintau],
            [rotation_axis_unit_vector.z*rotation_axis_unit_vector.x*(1-costau)-rotation_axis_unit_vector.y*sintau, rotation_axis_unit_vector.z*rotation_axis_unit_vector.y*(1-costau)+rotation_axis_unit_vector.x*sintau, costau+rotation_axis_unit_vector.z**2*(1-costau)]
    ])
```

```python
    point_vector = rotation_matrix @ point_vector.to_np_vector()
    return np_vector_to_standard_vector(point_vector)

class data:
    def __init__(self, name):
        self.name = name
        self.beta_arrival = None
        self.beta_departure = None
        self.time_of_arrival = None
        self.time_of_departure = None
        self.delta_v_hyperbolic_departure = None
        self.delta_v_hyperbolic_arrival = None
        self.dv_departure = None
        self.dv_arrival = None
    def travel_time(self):
        return self.time_of_arrival - self.time_of_departure
    def dv_total(self):
        dv_t = 0
        if self.delta_v_hyperbolic_departure != None:
            dv_t = dv_t + self.delta_v_hyperbolic_departure
        if self.delta_v_hyperbolic_arrival != None:
            dv_t = dv_t + self.delta_v_hyperbolic_arrival
        if self.dv_departure != None:
            dv_t = dv_t + self.dv_departure.vector_length()
        if self.dv_arrival != None:
            dv_t = dv_t + self.dv_arrival.vector_length()
        return dv_t

class vector:
    #https://tomaszgolan.github.io/js-python/wyklady/js-python_w10/#wektor-iloczyn-
    skalarny
    def __init__(self, x=0.0, y=0.0, z=0.0):
        self.x = x
        self.y = y
        self.z = z
    def __str__(self):
        return "[{}, {}, {}]".format(self.x, self.y, self.z)
    def __mul__(self, w): # operator mnozenia, iloczyn skalarny
        return self.x*w.x + self.y*w.y + self.z*w.z
    def vector_mul(self, w):#iloczyn wektorowy
        return vector(self.y*w.z-self.z*w.y, self.z*w.x-self.x*w.z, self.x*w.y-self
    .y*w.x)
    def vector_add(self, w):
        return vector(self.x+w.x, self.y+w.y, self.z+w.z)
    def vector_subtract(self, w):
        #print(type(w.x),type(self.x))
        return vector(self.x-w.x, self.y-w.y, self.z-w.z)
    def vector_mul_scalar(self, s):
        return vector(self.x*s, self.y*s, self.z*s)
    def vector_div_scalar(self, s):
        return vector(self.x/s, self.y/s, self.z/s)
    def angle_between_vectors(self, w):#https://www.omnicalculator.com/math/angle-
    between-two-vectors#angle-between-two-3d-vectors-example
        return math.acos((self.x*w.x+self.y*w.y+self.z*w.z)/(math.sqrt(self.x**2+
    self.y**2+self.z**2)*math.sqrt(w.x**2+w.y**2+w.z**2)))
    def vector_length(self):
        return math.sqrt(self.x**2+self.y**2+self.z**2)
    def to_np_vector(self):
        return np.array([[self.x, self.y, self.z]]).T
```

```
    def to_np_1d_vector(self):
        return np.array([self.x, self.y, self.z])
    def vector_dot_prodcut(self, w):
        a = self.to_np_vector()
        a = np.transpose(a)
        b = w.to_np_vector()
        return np.dot(a,b)[0].item()
    def get_unit_vector(self):
        return self.vector_div_scalar(self.vector_length())
    def vector_round(self, decimal_places):
        return vector(round(self.x, decimal_places), round(self.y, decimal_places), round(self.z, decimal_places))


def np_vector_to_standard_vector(np_array):
    return vector(np_array[0].item(), np_array[1].item(), np_array[2].item())


class param:
    k = vector(0,0,1)
    def __init__(self, name, has_mass_bool):
        self.name = name
        self.has_mass = has_mass_bool


    def body(self, mass, equatorial_radius):
        self.mass = mass #kg
        self.equatorial_radius = equatorial_radius#km

    def mu(self):
        return self.mass*G.value/1000000000 #km^3*s^-2

    def orbit(self, apoapsis, periapsis, eccentricity, inclination, arg_of_periapsis, orbited_body, mean_anomaly, epoch_MJD, ascending_node):#, epoch_MJD
        self.periapsis  = periapsis #km
        self.apoapsis = apoapsis #km
        self.eccentricity = eccentricity #[-]
        self.inclination = math.radians(inclination)#input in degrees #math.pi/2-
        self.orbited_body = orbited_body#class param of the body being orbited
        self.mean_anomaly = math.radians(mean_anomaly)#input in degrees
        self.epoch_MJD = epoch_MJD#input modified Julian Date - I don't think this is neaded any more
        self.arg_of_periapsis = math.radians(arg_of_periapsis)#input in degrees
        self.ascending_node = math.radians(ascending_node)#input in degrees

    def parking_orbit(self, parking_orbit_periapse_radius):
        self.parking_orbit_periapse_radius = parking_orbit_periapse_radius + self.equatorial_radius

    def semi_major_axis_length(self):#a
        return (self.periapsis+self.apoapsis)/2

    def semi_major_axis_unit_vector(self):
        answer = vector(self.semi_major_axis_length(), 0, 0)
        answer = rotate_point(answer, vector(0,0,1), self.arg_of_periapsis)
        inclination_rotation_vector = rotate_point(vector(1,0,0), vector(0,0,1), self.ascending_node)
        answer = rotate_point(answer, inclination_rotation_vector, self.inclination
```

```
)
        return answer.vector_div_scalar(self.semi_major_axis_length())

    def semi_minor_axis_length(self):#b or c (elipsoid)
        return (self.periapsis+self.apoapsis)/2*math.sqrt(1-self.eccentricity*self.eccentricity)

    def semi_minor_axis_unit_vector(self):
        answer = vector(0, self.semi_minor_axis_length(), 0)
        answer = rotate_point(answer, vector(0,0,1), self.arg_of_periapsis)
        inclination_rotation_vector = rotate_point(vector(1,0,0), vector(0,0,1), self.ascending_node)
        answer = rotate_point(answer, inclination_rotation_vector, self.inclination)
        return  answer.vector_div_scalar(self.semi_minor_axis_length())

    #cos and sin make shure angle isn't too close to pi/2 and pi
    def cos(self, angle_rad):
        if abs(angle_rad) > np.pi/2-0.00000001 and abs(angle_rad) < np.pi/2+0.00000001:
            return 0
        else:
            return np.cos(angle_rad)

    def sin(self, angle_rad):
        if abs(angle_rad) > np.pi-0.00000001 and abs(angle_rad) < np.pi+0.00000001:
            return 0
        else:
            return np.sin(angle_rad)

    def set_time(self, isot_time):
        self.MJD = isot_time.mjd

    def change_time_MJD(self, MJD_time):
        self.MJD = MJD_time

    def period(self):
        return 2*math.pi/math.sqrt(self.orbited_body.mu())*self.semi_major_axis_length()**(3/2)#seconds

    def orbital_plane_unit_vector(self):
        k = self.specific_angular_momentum()
        if k.z >= 0:
            return k.get_unit_vector()
        else:
            return k.vector_mul_scalar(-1).get_unit_vector()

    def ellipse_function(self,X,A,B):
        return B*math.sqrt(1-((X-A)**2)/(A**2))

    def area_under_ellipse(self,x):
        A = self.semi_major_axis_length()
        B = self.semi_minor_axis_length()
        return quad(self.ellipse_function,0, x, args=(A,B))[0]

    def find_true_anomaly(self):#returns angle in rad (so true anomaly)
        time_current = days_to_seconds(self.MJD)
        half_ellipse_area = math.pi*self.semi_major_axis_length()*self.semi_minor_axis_length()/2
```

```
        P = self.period()
        time_at_periapsis = self.mean_anomaly/(2*np.pi/P)+days_to_seconds(self.
    epoch_MJD)
        unit_time = time_current - time_at_periapsis
        time0 = P/2#time of half a orbit
        unit_time = unit_time % P
        if unit_time == P/2:#check extreems (apoapsis)
            return math.pi
        if unit_time == 0:#priapsis
            return 0

        which_half_of_orbit = 1
        if unit_time > P/2:
            which_half_of_orbit = -1
            unit_time = unit_time - P/2

        swept_area = half_ellipse_area*unit_time/time0
        X_max = self.semi_major_axis_length()*2
        rotation_point = self.semi_major_axis_length()-math.sqrt(self.
    semi_major_axis_length()**2-self.semi_minor_axis_length()**2)

        error = np.zeros(3)
        error[0] = self.semi_major_axis_length()#guessed x
        error[1] = 2#scale of guess
        error[2] = 0 #save if right (1) or left (2) of rotation point
        accuracy = 1 #km along x axis
        x_previous = 0

        while 1==1:
            area = self.area_under_ellipse(error[0])

            if error[1] > 1e+300:
                break

            triangle_area = np.zeros(2)
            if error[0] > rotation_point:
                triangle_area[0] = 0.5*(error[0]-rotation_point)*(self.
    ellipse_function(error[0],self.semi_major_axis_length(),self.
    semi_minor_axis_length()))
                triangle_area[1] = -1
                error[2] = 1
            else:
                triangle_area[0] = 0.5*(rotation_point-error[0])*(self.
    ellipse_function(error[0],self.semi_major_axis_length(),self.
    semi_minor_axis_length()))
                triangle_area[1] = 1
                error[2] = 2

            if (area+(triangle_area[0]*triangle_area[1]))-swept_area >= 0:
                error[1] = error[1]*2
                error[0] = error[0] - X_max/error[1]
            elif (area+(triangle_area[0]*triangle_area[1]))-swept_area < 0:
                error[1] = error[1]*2
                error[0] = error[0] + X_max/error[1]
            else:
                sys.exit("find_true_anomaly loop failed to meet continue or exit
    criteria")

            if abs(x_previous - error[0]) < accuracy:#check chanege of x, exit if
```

```
change less than accuracy
            break
        x_previous = error[0]

    #0 - true_anomaly [rad], 1 - radius [km], 2 - x [km], 3 - y [km]
    #print(seconds_to_days(unit_time))
    angle = np.zeros(5)

    if error[2] == 1:
        if which_half_of_orbit == 1:#right of rotation_point, first half
            angle[0] = np.pi-np.arctan(self.ellipse_function(error[0],self.semi_major_axis_length(),self.semi_minor_axis_length())/(error[0]-rotation_point))
            angle[1] = abs((error[0]-rotation_point)/np.cos(angle[0]))
            angle[2] = error[0]-rotation_point
            angle[3] = self.ellipse_function(error[0],self.semi_major_axis_length(),self.semi_minor_axis_length())
            angle[4] = unit_time
            return angle
        elif which_half_of_orbit == -1:#right of rotation_point, second half
            angle[0] = 2*np.pi-np.arctan(self.ellipse_function(error[0],self.semi_major_axis_length(),self.semi_minor_axis_length())/(error[0]-rotation_point))
            angle[1] = abs((error[0]-rotation_point)/np.cos(angle[0]))
            angle[2] = error[0]-rotation_point
            angle[3] = -self.ellipse_function(error[0],self.semi_major_axis_length(),self.semi_minor_axis_length())
            angle[4] = unit_time
            return angle
        else:
            sys.exit("find_true_anomaly failed to determin whether angle is + ro - on the y axis")
    elif error[2] == 2:
        if which_half_of_orbit == 1:#left of rotation_point, first half of orbit
            angle[0] = np.arctan(self.ellipse_function(error[0],self.semi_major_axis_length(),self.semi_minor_axis_length())/(rotation_point-error[0]))
            angle[1] = abs((rotation_point-error[0])/np.cos(angle[0]))
            angle[2] = error[0]-rotation_point
            angle[3] = self.ellipse_function(error[0],self.semi_major_axis_length(),self.semi_minor_axis_length())
            angle[4] = unit_time
            return angle
        elif which_half_of_orbit == -1:#left of rotation_point, second half of orbit
            angle[0] = np.pi+np.arctan(self.ellipse_function(error[0],self.semi_major_axis_length(),self.semi_minor_axis_length())/(rotation_point-error[0]))
            angle[1] = abs((rotation_point-error[0])/np.cos(angle[0]))
            angle[2] = error[0]-rotation_point
            angle[3] = -self.ellipse_function(error[0],self.semi_major_axis_length(),self.semi_minor_axis_length())
            angle[4] = unit_time
            return angle
        else:
            sys.exit("find_true_anomaly failed to determine whether angle is + or - on the y axis")
    else:
```

```
            sys.exit("find_true_anomaly failed to determine whether angle is left or right of rotation_point")


    def frame_of_reference_correction(self):#move x axis from center of ellipse to foci
        return math.sqrt(self.semi_major_axis_length()*self.semi_major_axis_length()-self.semi_minor_axis_length()*self.semi_minor_axis_length())

    def position_vector(self):
        position_data = self.find_true_anomaly()
        position = vector(position_data[2], position_data[3], 0)
        position = rotate_point(position, vector(0,0,1), self.arg_of_periapsis)
        inclination_rotation_vector = rotate_point(vector(1,0,0), vector(0,0,1), self.ascending_node)
        position = rotate_point(position, inclination_rotation_vector, self.inclination)
        return position

    def distance_from_orbited_body(self):
        r = self.position_vector()
        return math.sqrt(r*r)

    def speed(self):
        return math.sqrt(self.orbited_body.mu()*(2/self.distance_from_orbited_body()-1/self.semi_major_axis_length()))

    def velocity_vector(self):
        p1 = self.position_vector()
        time = self.MJD
        self.change_time_MJD(self.MJD+0.1)
        p2 = self.position_vector()
        self.change_time_MJD(time)
        return p2.vector_subtract(p1).vector_div_scalar(days_to_seconds(0.1))

    def specific_angular_momentum(self):
        return self.position_vector().vector_mul(self.velocity_vector())

    def position_of_ascending_node(self, reference_plane_unit_vector):
        return reference_plane_unit_vector.vector_mul(self.specific_angular_momentum())

    def angle_to_ascending_node(self, reference_plane_unit_vector):
        #https://adndevblog.typepad.com/manufacturing/2012/06/find-angle-between-two-vectors-along-a-given-direction-anti-clockwise-or-clockwise.html
        n = self.position_of_ascending_node(reference_plane_unit_vector)
        p = self.position_vector()
        v = self.velocity_vector()
        n2 = n.get_unit_vector().vector_mul_scalar(-1)
        if p.vector_mul(n).get_unit_vector().vector_add(p.vector_mul(v).get_unit_vector()).vector_length() == 0:
            return p.angle_between_vectors(n)
        else:
            return p.angle_between_vectors(n2)

    def time_after_travel_by_angle(self, angle, radius_of_target_point):
        P = self.period()/2#seconds!
        true_anmoaly = self.find_true_anomaly()
        current_angle = true_anmoaly[0]
```

```
        new_angle = current_angle+angle
        rotation_point =  self.semi_major_axis_length()-math.sqrt(self.semi_major_axis_length()**2-self.semi_minor_axis_length()**2)
        half_orbit_area = np.pi*self.semi_major_axis_length()*self.semi_minor_axis_length()/2
        time_at_current = true_anmoaly[4]

        time_at_target_point = 0
        time_at_target_point = self.time_after_travel_by_angle_find_angle(rotation_point, radius_of_target_point, new_angle, P, half_orbit_area, time_at_target_point)

        return self.MJD+seconds_to_days(time_at_target_point-time_at_current)

    def time_after_travel_by_angle_find_angle(self, rotation_point, radius_of_target_point, new_angle, P, half_orbit_area, time_at_target_point):
        time = 0
        if new_angle <= np.pi/2:
            x = rotation_point - radius_of_target_point*np.cos(new_angle)
            area = self.area_under_ellipse(x)+1/2*self.ellipse_function(x,self.semi_major_axis_length(),self.semi_minor_axis_length())*(rotation_point-x)
            time = area*P/half_orbit_area#time to travel from periapsis to point, seconds!
        elif new_angle > np.pi/2 and new_angle <= np.pi:
            x = rotation_point + radius_of_target_point*np.cos(np.pi-new_angle)
            area = self.area_under_ellipse(x)-1/2*self.ellipse_function(x,self.semi_major_axis_length(),self.semi_minor_axis_length())*(x-rotation_point)
            time = area*P/half_orbit_area
        elif new_angle > np.pi and new_angle <= np.pi + np.pi/2:
            x = rotation_point + radius_of_target_point*np.cos(np.pi-(2*np.pi-new_angle))
            area = self.area_under_ellipse(x)+half_orbit_area-1/2*self.ellipse_function(x,self.semi_major_axis_length(),self.semi_minor_axis_length())*(x-rotation_point)
            time = area*P/half_orbit_area
        elif new_angle > np.pi + np.pi/2 and new_angle <= 2*np.pi :
            x = rotation_point - radius_of_target_point*np.cos(2*np.pi-new_angle)
            area = self.area_under_ellipse(x)+half_orbit_area+1/2*self.ellipse_function(x,self.semi_major_axis_length(),self.semi_minor_axis_length())*(rotation_point-x)
            time = area*P/half_orbit_area
        elif new_angle > 2*np.pi:
            time_at_target_point = time_at_target_point+P*2
            self.time_after_travel_by_angle_find_angle(rotation_point, radius_of_target_point, new_angle-2*np.pi, P, half_orbit_area, time_at_target_point)
        else:
            sys.exit("time_after_travel_by_angle angle out of bounds")
        return time+time_at_target_point



def find_times_of_ascending_node(start_of_search_time, end_of_search_time, origin, destination):#future launch windows \\ (MJD, MJD, param, param)
    origin.change_time_MJD(start_of_search_time)
    destination.change_time_MJD(start_of_search_time)
    destination_plane_unit_vector = destination.orbital_plane_unit_vector()
    time_current = start_of_search_time
    transfer_times = np.array([])
```

```python
    while time_current < end_of_search_time:
        position_of_next_ascending_node = origin.position_of_ascending_node(destination_plane_unit_vector)
        answer = origin.time_after_travel_by_angle(origin.angle_to_ascending_node(destination_plane_unit_vector), position_of_next_ascending_node.vector_length())
        transfer_times = np.append(transfer_times, answer)
        time_current = answer+10
        origin.change_time_MJD(time_current)

    return transfer_times


def time_of_transfer(origin, destination, max_time, min_time, time0, time_step, central_body):
    dv = np.array([])
    time_at_dv = np.array([])
    t = min_time
    while t < max_time:
        destination.change_time_MJD(time0+t)
        orb1 = Orbit.from_vectors(central_body, origin.position_vector().to_np_1d_vector()*units.km, origin.velocity_vector().to_np_1d_vector()*units.km/units.s, epoch=Time(origin.MJD, format = 'mjd'))
        orb2 = Orbit.from_vectors(central_body, destination.position_vector().to_np_1d_vector()*units.km, destination.velocity_vector().to_np_1d_vector()*units.km/units.s, epoch=Time(destination.MJD, format = 'mjd'))
        m_lambert = Maneuver.lambert(orb1,orb2)
        dva = vector(m_lambert[0][1][0].value, m_lambert[0][1][1].value, m_lambert[0][1][2].value).vector_length()*1e-3
        dvb = vector(m_lambert[1][1][0].value, m_lambert[1][1][1].value, m_lambert[1][1][2].value).vector_length()*1e-3
        dv = np.append(dv, dva+dvb)
        time_at_dv = np.append(time_at_dv, t+time0)
        t = t + time_step
    mini = np.argmin(dv)
    mini_time = time_at_dv[mini]
    return mini_time

def transfer(origin, destination, time0, central_body):
    origin.change_time_MJD(time0)
    max_time = seconds_to_days(destination.period())

    time_step = 10
    time = time_of_transfer(origin, destination, max_time, 10, time0, time_step, central_body)



    while time_step > 1e-5:
        min_time = time-time0-time_step
        if min_time <= 0:
            min_time = 1
        max_time = time-time0+time_step
        if max_time <= min_time:
            max_time = min_time + 1
        time = time_of_transfer(origin, destination, max_time, min_time, time0, time_step/2, central_body)
        time_step = time_step*0.5

```

```
    destination.change_time_MJD(time)
    orb1 = Orbit.from_vectors(central_body, origin.position_vector().to_np_1d_vector()*units.km, origin.velocity_vector().to_np_1d_vector()*units.km/units.s, epoch=Time(origin.MJD, format = 'mjd'))
    orb2 = Orbit.from_vectors(central_body, destination.position_vector().to_np_1d_vector()*units.km, destination.velocity_vector().to_np_1d_vector()*units.km/units.s, epoch=Time(destination.MJD, format = 'mjd'))

    m_lambert = Maneuver.lambert(orb1,orb2)

    dv1 = vector(m_lambert[0][1][0].value/1000, m_lambert[0][1][1].value/1000, m_lambert[0][1][2].value/1000)#departure velocity vector
    dv2 = vector(m_lambert[1][1][0].value/1000, m_lambert[1][1][1].value/1000, m_lambert[1][1][2].value/1000)#arrival velocity vecroe


    orb1, _ = orb1.apply_maneuver(m_lambert, intermediate=True)#get transfer orbit

    transfer_data = data(' '.join(["transfer z ",origin.name, " do " , destination.name]))

    transfer_data.time_of_departure = time0
    transfer_data.time_of_arrival = time

    if origin.has_mass == True:
        dv = planetary_departure(origin, dv1)
        transfer_data.delta_v_hyperbolic_departure = dv[0]
        transfer_data.beta_departure = dv[1]
        transfer_data.dv_departure = change_plane(rotate_point(dv1, origin.position_vector().get_unit_vector(), orb1.inc.value), dv1)#dv_rot =
    else:
        transfer_data.dv_departure = dv1
    if destination.has_mass == True:
        dv = planetary_rendezvous(destination, dv2)
        transfer_data.delta_v_hyperbolic_arrival = dv[0]
        transfer_data.beta_arrival = dv[1]
    else:
        transfer_data.dv_arrival = dv2

    return transfer_data

def change_plane(velocity_vector_initial, velocity_vector_final):
    return velocity_vector_final.vector_subtract(velocity_vector_initial)

def planetary_departure(origin, departure_velocity_vector):
    V_1 = origin.velocity_vector()
    excess_velocity = departure_velocity_vector.vector_subtract(V_1)
    excess_speed = excess_velocity.vector_length()
    speed_parking = np.sqrt(origin.mu()/origin.parking_orbit_periapse_radius)
    delta_v_hyperbolic_departure = np.sqrt(excess_speed**2+2*origin.mu()/origin.parking_orbit_periapse_radius) - speed_parking
    orientation_apse_line = np.arccos(1/(1+(origin.parking_orbit_periapse_radius*excess_speed**2/origin.mu()))) #beta [rad]

    answer = np.zeros(3)
    answer[0] = delta_v_hyperbolic_departure
    answer[1] = orientation_apse_line
    answer[2] = excess_speed
    return answer
```

```
def planetary_rendezvous(destination, arrival_velocity_vector):
    V_2 = destination.velocity_vector()
    excess_velocity = arrival_velocity_vector.vector_subtract(V_2)
    excess_speed = excess_velocity.vector_length()
    speed_parking = np.sqrt(destination.mu()/destination.
    parking_orbit_periapse_radius)
    delta_v_hyperbolic_arrival = np.sqrt(excess_speed**2+2*destination.mu()/
    destination.parking_orbit_periapse_radius) - speed_parking
    orientation_apse_line = np.arccos(1/(1+(destination.
    parking_orbit_periapse_radius*excess_speed**2/destination.mu())))

    answer = np.zeros(3)
    answer[0] = delta_v_hyperbolic_arrival
    answer[1] = orientation_apse_line
    answer[2] = excess_speed
    return answer

#####################################################
#                                                   #
# change values beneath to set transfer parameters  #
#                                                   #
#####################################################
#second, minute, hour, day, month, year
start_of_analysis = Time('2027-01-01T00:00:00', format = 'isot')
end_of_analysis = Time('2033-01-01T00:00:00', format = 'isot')

Central_body = param("Sun", True)
Central_body.body(M_sun.value, R_sun.value/1000)

Origin = param("Ziemia", True)
Origin.body(M_earth.value, R_earth.value/1000)
epoch_earth = Time(2000.0, format = 'jyear')
#  apoapsis[km], periapsis[km], eccentricity[-], inclination[deg], arg_of_periapsis
     [deg], orbited_body, mean_anomaly[deg], epoch_MJD [MJD], ascending_node [deg]
Origin.orbit(152100000, 147095000, 0.0167086, 0, 0, Central_body, 358.617, Time('
    J2000.0', format = 'jyear_str').mjd, 0)#365.256 #, epoch_earth.mjd
Origin.parking_orbit(300)
Origin.set_time(Time('2004-05-12T14:45:30', format = 'isot'))

Destination = param("1996FG3", False)
#  apoapsis[km], periapsis[km], eccentricity[-], inclination[deg], arg_of_periapsis
     [deg], orbited_body, mean_anomaly[deg], epoch_MJD [MJD], ascending_node [deg]
Destination.orbit(212728172.12260202, 102474541.42333502, 0.35, 2, 24.08,
    Central_body, 202.32, 59600.0, 299.6764) #, 59600.0
Destination.set_time(Time('2004-10-12T14:45:30', format = 'isot'))



#####################################################
#                                                   #
#####################################################

start_of_analysis.format = 'mjd'
end_of_analysis.format = 'mjd'

print(start_of_analysis.value, end_of_analysis.value)

array = find_times_of_ascending_node(start_of_analysis.value, end_of_analysis.value
```

```
    , Origin, Destination)

print("found transfer times ", np.size(array))

all_transfer_data = np.array([])

i = np.size(array)-1
print(i)
while i >= 0:
    all_transfer_data = np.append(all_transfer_data, transfer( Origin, Destination,
     array[i], Sun))
    print(i)
    i = i - 1

i = 0

#print to latex table format
print(all_transfer_data[0].name)
#\begin{tabular}{|B|A|B|L|B|A|B|L|C|C|}
#\usepackage{array}
#\newcolumntype{L}{>{\centering\arraybackslash}m{2.9cm}}
#\newcolumntype{C}{>{\centering\arraybackslash}m{1.7cm}}
#\newcolumntype{B}{>{\centering\arraybackslash}m{2cm}}
#\newcolumntype{A}{>{\centering\arraybackslash}m{1.5cm}}
#copy next line as first line of latex table
# \textbf{czas odlotu} & \textbf{$\beta$ odlotu [$\degree$] \ref{eqn:beta}} & \
    textbf{$\Delta v$ hiperboli odlotu [km/s]} & \textbf{$\Delta V$ odlotu [km/s]}
    & \textbf{czas przylotu} & \textbf{$\beta$ przylotu [$\degree$] \ref{eqn:beta}}
     & \textbf{$\Delta v$ hiperboli przylotu [km/s]} & \textbf{$\Delta V$ przylotu
    [km/s]} & \textbf{$\Delta v$ transferu [km/s]} & \textbf{czas transferu [dni
    ]}\\ \hline

while i < np.size(all_transfer_data):
    tdep = Time(all_transfer_data[i].time_of_departure, format = 'mjd')
    tdep.format = 'iso'
    beta_dep = all_transfer_data[i].beta_departure
    if beta_dep != None:
        beta_dep = round(np.rad2deg(beta_dep), 2)
    dv_hip_dep = all_transfer_data[i].delta_v_hyperbolic_departure
    if dv_hip_dep != None:
        dv_hip_dep = round(dv_hip_dep, 2)
    dv_dep = all_transfer_data[i].dv_departure
    if dv_dep != None:
        dv_dep = dv_dep.vector_round(2)
    tarr = Time(all_transfer_data[i].time_of_arrival, format = 'mjd')
    tarr.format = 'iso'
    beta_arr = all_transfer_data[i].beta_arrival
    if beta_arr != None:
        beta_arr = round(np.rad2deg(beta_arr), 2)
    dv_hip_arr = all_transfer_data[i].delta_v_hyperbolic_arrival
    if dv_hip_arr != None:
        dv_hip_arr = round(dv_hip_arr, 2)
    dv_arr = all_transfer_data[i].dv_arrival
    if dv_arr != None:
        dv_arr = dv_arr.vector_round(2)
    tavel_t = round(all_transfer_data[i].travel_time(), 2)
    dv_tot = round(all_transfer_data[i].dv_total(), 2)
    print(tdep.value ," & ", beta_dep ," & ", dv_hip_dep ," & ", dv_dep ," & ",
    tarr.value ," & ", beta_arr  ," & ", dv_hip_arr , " & " , dv_arr, " & " ,
```

```
    dv_tot,  " & " , tavel_t," \\\ \hline")
    i = i + 1
```

Dodatek B

Notacja

e - ekscentryczność
l - długości odcinka między środkiem elipsy
a - długości wielkiej półosi
ΔA - zakreślone pole
Δt - odstęp czasu
T_1, T_2 – okresy obiegów planet
a_1, a_2 – wielkie półosie planet
v - prędkość na orbicie
G - stała grawitacji
r - promień orbity, odległość ciała orbitującego od centralnego
$M(t)$ = średnia anomalia w czasie t
M_0 = średnia anomalia w chwili t=0
n = średnia ruchliwość orbity satelitarnej
t = wybrana godzina
t_0 = czas ostatniej znanej średniej anomalii
m – masa mniejszego ciała orbitującego
M – masa centralnego ciała
V_{esc} - prędkość ucieczki
v_∞ - nadmiarowa prędkość hiperboliczna
r_1 - promień orbity planety odlotu
r_2 - promień orbity planety docelowej
μ_{sun} - parametr grawitacyjny słońca
h - właściwy moment pędu
μ_1 - parametr grawitacyjny planety odlotu
r_p - promień perycentrum orbity parkującej
v_c - prędkość statku na orbicie parkowania
Δv, ΔV - delta-v
β - orientacja linii apsyd hiperboli względem heliocentrycznego wektora prędkości planety
$\boldsymbol{v_\infty}$ - wektor hiperbolicznej prędkości
$\boldsymbol{V_D}$ - wektor prędkości statku w odniesieniu do Słońca
$\boldsymbol{V_1}$ - wektor prędkości planety odlotu
v_{odlotu} - prędkość statku na perycentrum hiperboli odlotu
$\boldsymbol{V_2}$ - wektor prędkości planety przylotu
$\boldsymbol{V_A}$ - wektor prędkości statku w odniesieniu do Słońca

$v_{przylotu}$ - prędkość statku na perycentrum hiperboli przylotu
v_{el} - prędkość statku na eliptycznej orbicie parkingowej
Δv_A - delta-v wymagana do manewru w punkcie A
Δv_B - delta-v wymagana do manewru w punkcie B
Δv_{total} - całkowity zapotrzebowanie delta-v
C - ciało centralne
ϕ - kąt zenitalny
δ - kąt asymptot
b - parametr zderzenia
r_{min} - długość perycentrum
r_{max} - długość apocentrum
p - parametr elipsy
F - parametr zwany hiperboliczną anomalią mimośrodową
F_O - parametr zwany hiperboliczną anomalią mimośrodową dla czasu odniesienia
θ - inklinacja
v_i - prędkość orbity początkowej
v_f - prędkość orbity końcowej
$\boldsymbol{h}$ - wektor właściwego momentu pędu
$\boldsymbol{r}$ - wektor zględnego położenia
$\boldsymbol{v}$ - wektor względnej prędkości
$\boldsymbol{L}$ - wektor momentu pędu
Ω - długość węzła wstępującego
n - wektor skierowany w stronę węzła wstępującego
k - wektor normalny płaszczyzny odniesienia
T - czas
c - odległość między punktami
ν - anomalia prawdziwa
x - współrzędna x
y - współrzędna y

Instytut Energetyki Cieplnej,
WYDZIAŁ INŻYNIERII LĄDOWEJ I TRANSPORTU
Politechnika Poznańska

Skład przy użyciu systemu LaTeX na platformie Overleaf.