TITLE OF THE PROJECT:

OPTIMAL TRAJECTORIES TO URANUS

Aerospace's engineering bachelor's final thesis

Author: Germán Sánchez Abellón

The University of Oklahoma tutor: Dr. Diogo Merguizo Sanchez

Universitat Poltècnica de Valéncia tutor: Dr. Pedro Manuel Quintero Igeño

Academic year: 2023-2024

2024/08/07

Acknowledgement

First and foremost, I would like to dedicate this project to my parents, Germán and María, and my sister Carla. Their endless love, support, and effort to provide me with the best opportunities to grow as a professional and as a person are the greatest gifts anyone could ever give me.

Of course, I would like to thank to Dr. Diogo Merguizo Sanchez who has taught me everything I know about astrodynamics from the beginning and has always been willing to answer all my questions and help me.

I would also like to thank Dr. Pedro Manuel Quintero Igeño for his help with the administrative part of the project and his availability.

I am also thankful for having the family I have. Their help and unconditional love are invaluable. A special mention goes to my grandparents Gino, Pedro, Paquita and especially to Ramón. Without those summer mornings learning math with him before going to the beach, I wouldn't be writing this.

I also want to thank all the friends I met during my exchange year in Oklahoma because they have become my family and made me feel at home. Thanks to my American friends Trevor, Cameron, Shelbe and Kate and my international friends Manu, Valentina, Valeria, Sabine, Camilo, Abraham, Laura, Arcil, José, Diego, Anibal, Alex, Daicia, Santi, Juan Pablo, Juan, Nía etc.

Finally, I would like to thank to my friends from Spain. I met them when I was just a child and they have been always there, at the best and the worst moments, even if I am not in the same city or the same country. Thank you, Álvaro, Isabel María, Bárbara, Laura, David, Eva, Jesús, Marín, and Gloria. Without your support, this wouldn’t be possible.

Abstract

Exploring space has become one of the next steps in human civilization. In this respect, Uranus, which has been visited just once by the Voyager 2 mission, has gained importance as it can help to answer elemental questions such as how the origin and potential end of the Solar System will be. This is one of the reasons why it has been elevated to top target planet for solar system exploration by the American Astronomical Society.

Therefore, the aim of this project is to develop an algorithm to calculate optimal trajectories from Earth to Uranus with a launch window between 2030 and 2040. To achieve this, the planetary data from JPL (Jet Propulsion Laboratory) Horizons and the Lambert's problem will be necessary. Python has been selected as the programming language to develop the code for generating a set of trajectories to capture a spacecraft around Uranus with minimum fuel consumption in terms of total delta-v. Additionally, the necessary delta-V to escape Earth's sphere of influence from a 200 km circular parking orbit to the selected transfer orbit will be calculated.

Table of Contents

1. List of Figures

2. List of Tables

3. List of equations

4. List of Symbols

Symbol	Definition
h	Specific angular momentum
i	Inclination
e	Eccentricity
θ	True anomaly
ω	Argument of perigee
Ω	Right ascension of the ascending node
ε	Obliquity of the ecliptic
SOI	Sphere of influence
m_p	Mass of the planet
m_s	Mass of the Sun
a	Semi-major axis
T	Period of the orbit
t	Time
R	Radius
E	Energy
μ	Standard gravitational parameter
V	Velocity
ϕ	Angular position
n	Angular orbital velocity
r	Position vector
v	Velocity vector
$g, \dot{g}, f, \dot{f}$	Lagrange coefficients
χ	Universal anomaly
A	Auxiliar parameter for Lambert's problem
S, C	Stumpp functions
TOF	Time of flight
v_∞	Hyperbolic excess speed
C_3	Characteristic energy
γ	Flight path angle
v_r	Radial velocity
N	Nodal vector
Δv	Delta-V

5. Introduction

5.1. Motivation

Since the beginning of history, humans have been expanding their borders seeking an improvement of their quality of life and a better understanding of the world around them. Technological developments, such as the wheel or the boat, have allowed us to migrate and discover new territories, cultures and extend our horizon to that place where ancient humans believed that gods live, the sky.[1] [2] [3]

Nowadays, what satisfies humans' necessity of exploring is space. Therefore, for the last decades a lot of people around the world have worked together to expand our vision beyond Earth. Since 1957, when the first artificial satellite, Sputnik 1, was launched a countless number of spacecrafts have escaped Earth's atmosphere. Nevertheless, humans' exploration didn't stop when they launched an artificial satellite, walking on the Moon, constantly living in the Space Station around Earth or landing on Mars are also milestones which have been achieved. Therefore, the question that arises now is, what's next? [4]

Even when discovering new places, human curiosity doesn't stop. This is the main reason why we still have some unanswered questions that we daily think about such as the existence of gods or the origin of our civilization. Nevertheless, this last question may be answered, but how? By exploring space, specifically Uranus.[5]

Uranus, the seventh planet in the Solar System, is one of the two ice giants, which means that is mainly composed of hydrogen and helium, and its system includes 27 known moons. Furthermore, it has an interesting characteristic, its odd spin orientation with an obliquity of 98°, which makes its spin axis to be almost parallel to its orbital plane. Despite all these unique characteristics, Uranus has been visited by a spacecraft just once.

The spacecraft which visited Uranus in 1986 was part of the Voyager 2 mission. This mission was mainly designed to explore Jupiter and Saturn, and it made many discoveries such as active volcanoes on Io, one of Jupiter's moons. But it was extended to flyby Neptune and Uranus too, finding 10 new moons around Uranus and two new rings. This is the main reason why this configuration, which occurs every 175 years, through all the giants' planets of the Solar System was called "The Great Tour". Thus, planning a mission to Uranus is one of the next goals in space exploration and some agencies are working on future missions to Uranus, such as the Uranus Orbiter and Probe (UOP). (Anderson, 2021) [6] [7]

Finally, The Planetary Society has elevated Uranus to the category of top target for solar system exploration. This is because having a better understanding of how Uranus was formed will allow us to understand the migration of the gas giants and the evolution of our solar system and early Earth. Furthermore, some of the technology that we daily use has its origin in space such as internet or systems to predict the weather. Therefore, the aim of this project is to contribute to the development of humanity through space exploration, which will be made by optimizing trajectories from Earth to Uranus. [2]

5.2. Scope and project outline

Throughout the "Optimal Trajectories to Uranus" project, theoretical concepts will be used to find optimal trajectories from Earth to Uranus during the launch window of 2030-2040. To accomplish this, both the time of flight and the starting date for the trajectory will vary. This variation combined with real data of the planets obtained from JPL (Jet Propulsion Laboratory) Horizons System will permit to select the best trajectory depending on the requirements of the mission, minimizing fuel consumption. Finally, Python will be used to obtain the data of the planets and develop the necessary code to perform all the required calculations. Additionally, it will be used to plot the trajectory of the planets and the selected transfer.

---

[1] GAIA Mission (ESA) - last access on June 20, 2024
[2] Benefits of space exploration - last access on June 20, 2024
[3] Early astronomy - last access on June 20, 2024
[4] Sputnik 1 - last access on June 20, 2024
[5] Planetary Society - last access on June 20, 2024
[6] Voyager 2 - last access on June 20, 2024
[7] Voyager - last access on June 20, 2024

Therefore, this project will follow the project outline showed above:

1. Some theoretical concepts about astrodynamics, orbital maneuvers and the software used will be summarized because they are necessary to fully understand the project.
2. Simplified Hohmann transfer is calculated to have a reference of the expected order of magnitude of the results.
3. The process followed to determine the trajectories and the programmed code is explained.
4. Numerical results and plots are analyzed to select the optimal trajectory.
5. The regulation that affects the project and how it was met during the project completion will be discussed.
6. The time management during this year is shown using a Gantt chart.
7. The cost of the project will be calculated, and the budget presented.
8. Conclusions and future work will be discussed.

6. Basic concepts about astrodynamics

Once the motivations and the project scope have been exposed, it is necessary to briefly explain some basic concepts about astrodynamics that will be used throughout the project.

6.1. Orbital parameters

There are several ways to describe the orbit of a celestial body in space, but through this project 6 parameters also known as Keplerian orbital elements will be used. These parameters are $\{h, i, \Omega, e, \omega, \theta\}$ which have been described in Figure 1and Table 1. (Curtis, 2020) [8]

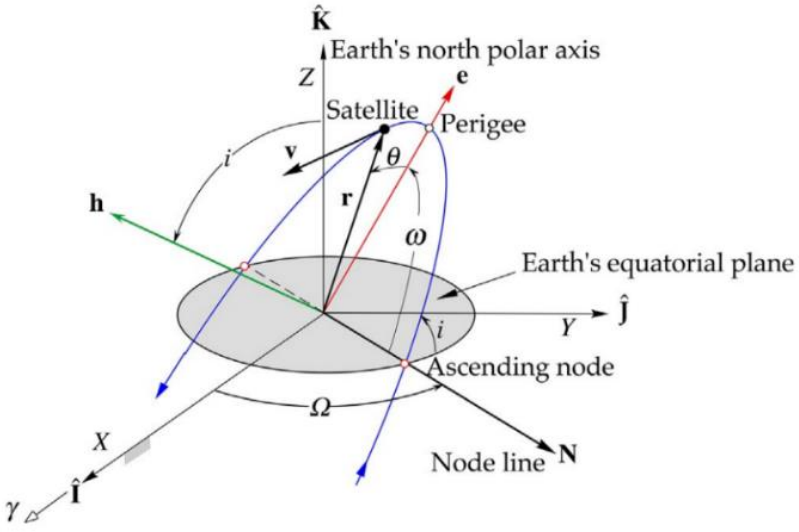

Figure 1: Orbital elements.

[8] Orbital mechanics concepts - last access on December 12, 2023

Element	Symbol	Definition	Range
Specific angular momentum	h	Vector perpendicular to the position vector and the velocity vector of the object	-
Inclination	i	Angle between the positive Z axis and $\vec{h}$	Positive number between 0° and 180°
Right ascension of the ascending node	Ω	Angle between the positive X axis and the node line	Positive number between 0°and 360°
Eccentricity	e	Describes how different is an orbit from being a circle. It defines the type of orbit	Non-negative real number
Argument of perigee	ω	Angle between the node line vector and the eccentricity vector measured in the plane of the orbit	Positive number between 0° and 360°
True anomaly	θ	Angle between the periapsis direction and the current position of the body. Define the position of the body along the orbit.	Positive number between 0° and 360

Table 1: Orbital parameters.

6.2. Epoch

The epoch in astronomy is a reference moment for consistency in calculation of positions and orbits of a celestial body because it can suffer perturbations during time. A common astronomical epoch is J2000 which means noon on January 1, 2000. (Curtis, 2020) [9]

6.3. Ephemeris

Another term that will be important along the project is ephemeris. The ephemeris is basically a table which has the trajectory of astronomical objects over time. It depends on the location of the vernal equinox at a given time or epoch, and it will be useful to obtain where the planets are and where they will be at a certain moment. (Curtis, 2020) [10]

6.4. Coordinate systems

6.4.1. Geocentric equatorial reference frame

The Equatorial reference frame is a celestial coordinate system with its origin at the center of the planet, the primary direction towards the vernal equinox and a right-handed reference frame, Figure 2 .It is mainly used when the spacecraft is into the sphere of influence of the planet. (Curtis, 2020) [11]

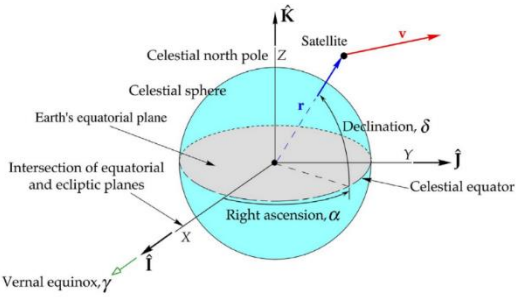

Figure 2: Geocentric equatorial reference frame.

[9] Epoch - last access on December 7, 2023
[10] Ephemeris - last access on December 7, 2023
[11] Geocentric equatorial reference frame - last access on December 7, 2023

6.4.2. Ecliptic reference frame

The Ecliptic reference frame is the plane of Earth's orbit around the Sun, Figure 3. The center of this reference frame is the Sun; therefore, it is useful when the spacecraft is out of the sphere of influence of other celestial bodies. As it can be seen in Figure 3**,** this plane is tilted away the equatorial plane by 23.4º which is known as the obliquity of the ecliptic, ε, and it, obviously, changes between the different celestial bodies. (Curtis, 2020) [12]

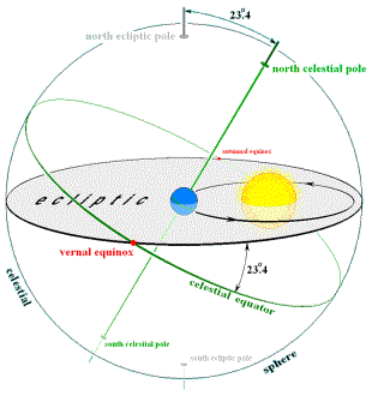

Figure 3: Ecliptic reference frame.

6.4.3. Perifocal reference frame

The perifocal reference frame, Figure 4, also called the natural frame of an orbit, is a Cartesian coordinate system fixed in space and centered at the focus of the orbit. Its x axis is directed from focus through the periapsis, the y axis lies at 90º true anomaly, the x axis and the z axis are normal to the plane of the orbit. (Curtis, 2020)

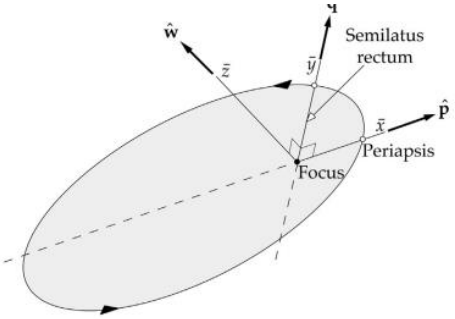

Figure 4: Perifocal reference frame.

6.5. Sphere of influence (SOI)

The sphere of influence (SOI) of a celestial body is the region around it where that body has the greatest gravitational effect. It is important to know the sphere of influence of the planets because that is the moment when the focus of the trajectory changes from one celestial body to another. Obviously, as our mission takes place inside the solar system, the focus will be the Sun. According to the references, the sphere of influence can be obtained with Equation 1. (Curtis, 2020) [8]

$$r_{SOI} = a_p \left(\frac{m_p}{m_s}\right)^{2/5}$$

Equation 1: Sphere of influence.

6.6. Lambert's problem

At the 18th century, the astronomer J.H.Lambert proposed that the transfer time from two different points is independent of the eccentricity of the orbit and only depends on the sum of the magnitude of the position vectors, the semi-major axis and the length of the chord joining both points. This aspect can be used to calculate the trajectory when both points and the time of flight between those points is known . (Curtis, 2020)

Therefore, considering the position vectors $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$, at points P_1 and P_2, the change in the true anomaly can be determined from. Figure 5 (Curtis, 2020)

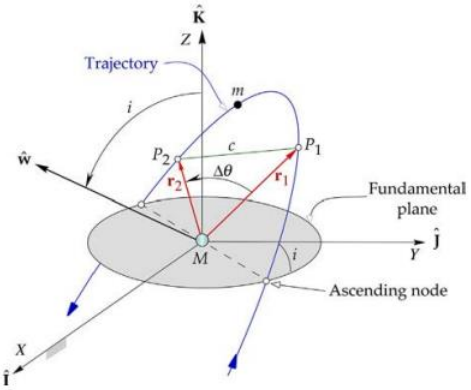

Figure 5: Geometry of Lambert's problem.

$$\cos\Delta\theta = \frac{\boldsymbol{r_1} \cdot \boldsymbol{r_2}}{r_1 r_2}$$

Equation 2: Delta of true anomaly at Lambert's problem.

[12] International celestian reference frame - last access on December 7, 2023

Being $r_1 = \sqrt{\boldsymbol{r_1}.\boldsymbol{r_1}}$ and $r_2 = \sqrt{\boldsymbol{r_2}.\boldsymbol{r_2}}$.

However, to be able to fully determine the quadrant without uncertainties, the orbit's inclination will be necessary. Therefore, using Equation 3, the change in the true anomaly can be defined. (Curtis, 2020)

$$\Delta\theta = \left\{ \begin{array}{l} \cos^{-1}\frac{\boldsymbol{r_1}.\boldsymbol{r_2}}{r_1 r_2} \quad if\ (\boldsymbol{r_1}\times\boldsymbol{r_2})_z \geq 0 \quad prograde\ trajectory \\ \hline 360° - \frac{\boldsymbol{r_1}.\boldsymbol{r_2}}{r_1 r_2} \quad if\ (\boldsymbol{r_1}\times\boldsymbol{r_2})_z < 0 \quad prograde\ trajectory \\ \hline \cos^{-1}\frac{\boldsymbol{r_1}.\boldsymbol{r_2}}{r_1 r_2} \quad if\ (\boldsymbol{r_1}\times\boldsymbol{r_2})_z < 0 \quad retrograde\ trajectory \\ \hline 360° - \frac{\boldsymbol{r_1}.\boldsymbol{r_2}}{r_1 r_2} \quad if\ (\boldsymbol{r_1}\times\boldsymbol{r_2})_z \geq 0 \quad retrograde\ trajectory \end{array} \right\}$$

Equation 3: Delta of the true anomaly at Lambert's problem without uncertainty

As it has been said before, Lambert proposed that transfer time from two different points is independent of the eccentricity of the orbit and depends only on the sum of the magnitude of the position vectors, the semi-major axis and the length of the chord joining both points. Thus, the trajectory can be determined using the Lagrange coefficients when the position vectors at point 1 and 2, $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$, respectively are known. Equation 4 and Equation 5. (Curtis, 2020)

$$\boldsymbol{v_1} = \frac{1}{g}(\boldsymbol{r_2} - f\,\boldsymbol{r_1})$$

Equation 4: Velocity at departure at Lambert's problem.

$$\boldsymbol{v_2} = \frac{1}{g}(\dot{g}\boldsymbol{r_2} - \boldsymbol{r_1})$$

Equation 5: Velocity at arrival at Lambert's problem.

Now, it can be observed that the next step should be obtaining the Lagrange coefficients. However, it will be necessary to introduce the universal anomaly, χ, to avoid the unknown angular momentum. Thus, the coefficients using universal anomaly are, Equation 6 (Curtis, 2020)

$$f = 1 - \frac{y(z)}{r_1}\ ;\quad g = A\sqrt{\frac{y(z)}{\mu}};\quad \dot{f} = \frac{\sqrt{\mu}}{r_1 r_2}\sqrt{\frac{y(z)}{C(z)}}[zS(z) - 1];\quad \dot{g} = 1 - \frac{y(z)}{r_2}$$

Equation 6: Coefficients at Lambert's problem.

Where the other variables are, Equation 7 (Curtis, 2020)

$$A = \sin\Delta\theta\sqrt{\frac{r_1 r_2}{1-\cos\Delta\theta}}\ ;\quad y(z) = r_1 + r_2\,A\frac{zS(z)-1}{\sqrt{C(z)}};\quad z = \alpha\chi^2\ ;\quad \alpha = \frac{1}{a}$$

Equation 7: Other variables at Lambert's problem.

And *S(z)* and *C(z)* are the Stumpff functions. Equation 8 (Curtis, 2020)

$$S(z) = \sum_{k=0}^{\infty}(-1)^k\frac{z^k}{(2k+3)!}$$

$$C(z) = \sum_{k=0}^{\infty}(-1)^k\frac{z^k}{(2k+2)!}$$

Equation 8: Stumpff functions.

7. Orbital maneuvers

The aim of this section is to explain some important theoretical concepts about orbital maneuvers and address the most energy-efficient transfer, the Hohmann transfer.

7.1. Impulsive maneuvers

When trying to change the orbit of a spacecraft is necessary to perform a maneuver. This orbital change requires the application of a force on the spacecraft which changes the magnitude or direction of the velocity vector. Most of the time this force is produced by an on-board system which can be an impulsive or a non-impulsive maneuver.

An impulsive maneuver is an idealized maneuver with the force applied during a brief firing of the onboard rockets which change the vector velocity instantaneously. Furthermore, it is assumed that the position vector of the spacecraft doesn't change during the maneuver. It is essential to mention that this assumption can be made only when the burn time of the rocket is short compared to the coasting period of the maneuver. (Curtis, 2020) [8]

Regarding this project, all the maneuvers will be considered impulsive.

7.2. Hohmann transfer

One of the most relevant orbital maneuvers is the Hohmann transfer. It is the most energy-efficient two-impulsive elliptical orbit between two coplanar circular orbits sharing the same focus. This elliptical orbit is tangent to two elliptical orbits which are the apogee and the perigee of the transfer, Figure 6. Therefore, the time necessary to go from one orbit to the other is half of the period of the elliptical transfer orbit. It is also relevant to remember that the energy of an orbit is given by Equation 9, therefore when the spacecraft goes from an inner orbit to an outer orbit it increases its energy because the energy increases with a greater semi-major axis. Equation 9. (Curtis, 2020) [8] [13] [14]

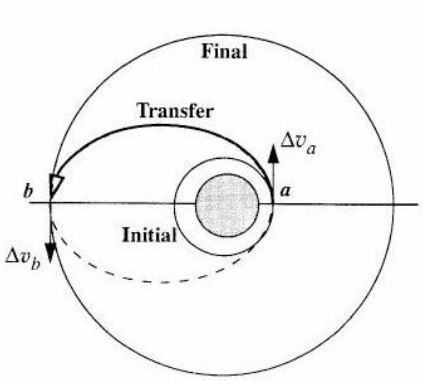

Figure 6: Hohmann transfer.

$$E = -\frac{\mu}{2a}$$

Equation 9: Energy of an orbit.

7.3. Interplanetary Hohmann transfer

As it has been mentioned in 7.2 Hohmann transfer, the most energy-efficient way for a spacecraft to transfer between planets is the Hohmann transfer. Nevertheless, when travelling between planets some aspects such as the eccentricity and the inclination of their orbits should be considered. Talking about eccentricity, all planets' orbits are almost circles, which means that the eccentricity of their orbits is almost zero, although this is not the case with Mercury. Furthermore, once again most of the planets have an inclination smaller than 3.5° except Mercury which inclination is 17°. (Curtis, 2020)

On the other hand, according to the references, delta-V for interplanetary Hohmann transfers can be obtained using Equation 10 and Equation 11, where R_1 is the radius of the outer orbit and $R_2$the radius of the inner orbit. (Curtis, 2020)

$$\Delta V_D = V_D - V_1 = \sqrt{\frac{\mu_{Sun}}{R_1}}\left(\sqrt{\frac{2R_2}{R_2 + R_1}} - 1\right)$$

Equation 10: Delta-V at the departure point.

[13] Ratonels - last Access on December 7, 2023
[14] Orbit transfer and interplanetary trajectories - last access on December 7, 2023

$$\Delta V_A = V_2 - V_A = \sqrt{\frac{\mu_{Sun}}{R_2}}\left(1 - \sqrt{\frac{2R_1}{R_2 + R_1}}\right)$$

Equation 11: Delta-V at the arrival point.

Nevertheless, the information that these equations provide is just the value of the Delta-V. Therefore, instead of using them, the specific angular momentum and the specific mechanical energy can be used. When using mechanical energy, the "Vis-viva" equation, Equation 12, is required. These alternative processes to obtain the Delta-V are explained at *B.4 deltaV_real_orbit_equation* and *B.5 deltaV_real_orbit_equation_energy*. (Curtis, 2020)

$$v = \sqrt{2\left(\frac{\mu}{r} + E\right)}$$

Equation 12: Vis-viva equation.

When performing an interplanetary Hohmann transfer orbit the spacecraft can departure from an inner orbit and arrive at an outer orbit or vice versa. This aspect will determine if the delta-V applied should be positive or negative. In the case of this project, the spacecraft is travelling from an inner orbit near to the Earth to an outer orbit near to Uranus which causes both delta-V to be positive. Nevertheless, when the spacecraft travels from an outer planet to an inner planet such as the case from Earth to Venus, both delta-V are negative. Figure 7[15]

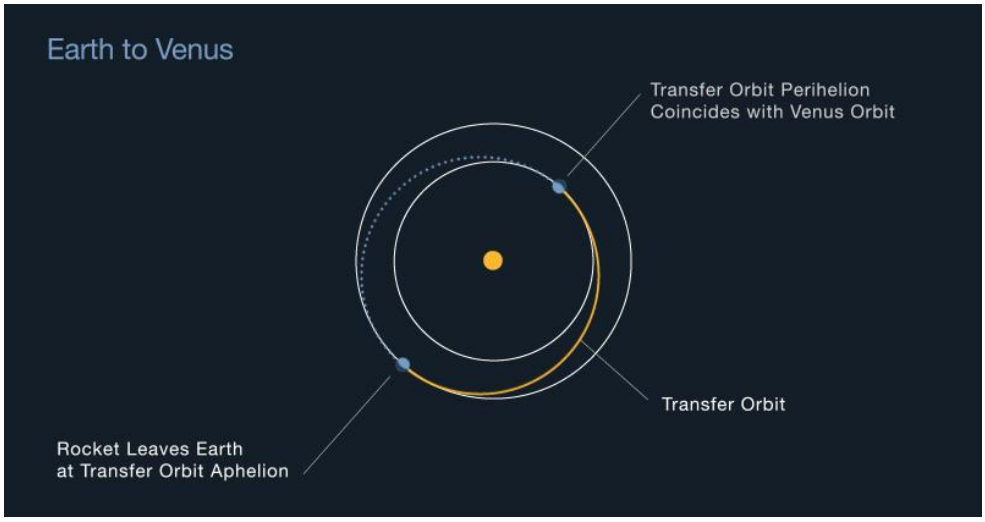

Figure 7: Interplanetary Hohmann transfer from Earth to Venus.

7.4. Rendezvous

Another important aspect which should be considered for every space mission is the rendezvous to the final orbit. When talking about Hohmann transfers, it is not always possible to successfully achieve the final orbit, but there exists only one moment when the transfer should start to achieve the final orbit. This instant of time should be the one which allows the spacecraft to arrive at the apse line of the transfer ellipse at the same time as

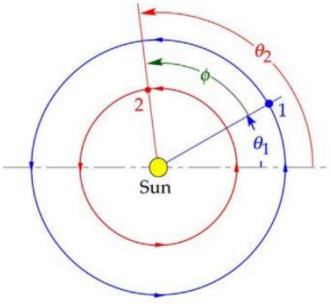

Figure 8: Hohmann transfer around the Sun.

[15] Chapter 4: Trajectories (NASA) - last access on December 7, 2023

the planet does, Figure 8. Therefore, it seems relevant to try to obtain the frequency of this event. This time received the name of synodic period and according to the references can be obtained as: Equation 13 (Curtis, 2020)

$$T_{syn} = \frac{T_1 T_2}{|T_1 - T_2|}$$

Equation 13: Synodic period.

On the other hand, it will be necessary to obtain the time the spacecraft takes to perform the maneuver. As has been mentioned at 7.2 Hohmann transfer, this is half of the period of the ellipse. Equation 14 (Curtis, 2020)

$$TOF = \frac{\pi}{\sqrt{\mu_{Sun}}}\left(\frac{R_1 + R_2}{2}\right)^{3/2}$$

Equation 14: Hohmann transfer time of flight.

Using this data, it is possible to calculate which should be the exact angular position of both celestial bodies when the maneuver starts. According to the references, it can be obtained as: Equation 15.

$$\phi = \phi_0 + (n_2 - n_1)t_{12}$$

Equation 15: Angular position for a Hohmann transfer.

Where *n* is the orbital angular velocity of the planets and ϕ_0 can be obtained as Equation 16.

$$\phi_0 = \pi - n_1 TOF$$

Equation 16: Initial angular position for Hohmann transfer.

7.5. Patched conics

Patched conics is a method used to study interplanetary trajectories. Using this method, the trajectory can be divided into three sections of study:

- **Heliocentric section:** when the Sun is the central body of the elliptical trajectory.
- **Launch section:** when the home planet is the central body of the hyperbolic departure trajectory.
- **Arrival section:** when the target planet is the central body of the hyperbolic arrival trajectory.

This approach can be done with interplanetary trajectories due to the vast distance between the main bodies. Therefore, for heliocentric trajectories, the planet's influence is neglected, and they are considered as points in space. However, when the spacecraft enters the sphere of influence of a planet, the reference frame changes, and the focus shifts to that planet.

Furthermore, for future calculations, it is essential to understand that from the Sun's point of view, the sphere of influence of a planet is small, while from the planet's point of view, that distance is huge. Thus, when studying the launch and arrival sections, the limit of the sphere of influence can be considered infinite. (Curtis, 2020)

7.6. Characteristic energy, *C3*

The characteristic energy, also known as *C3*, is defined as the square of the hyperbolic excess velocity, v_∞, which is the velocity at infinity with respect to the Sun. Equation 17.

$$C_3 = {v_\infty}^2$$

Equation 17: Characteristic energy.

Moreover, it will be positive, zero or negative depending on the trajectory. If *C3* is negative, the orbit's energy won't be sufficient to escape, and it will be in a closed orbit. Nevertheless, if *C3* is zero the energy will be exactly what is needed to escape, resulting in a parabolic orbit. Finally, if *C3* is greater than zero,

the orbit will be a hyperbola and will have enough energy to escape from the planet's sphere of influence. (Andreas, 2017)[16] [17]

This concept will be useful when choosing the optimal solution using a porkchop plot.

8. Software and data base

Throughout this section, the different sources of information to obtain real data about the planets will be shown. Furthermore, some reasons why the selected software is used will be provided.

8.1. Astroquery

Obviously, to be able to perform all the calculations of this project, it will be essential to obtain the data related to the planets. For this purpose, a planetary database called Astroquery will be used. Furthermore, all the data can be requested and used through Python. [18]

8.2. Horizons System

This is a website provided by JPL NASA which is useful to obtain the ephemeris of the celestial bodies. It also has a detailed manual where all the abbreviations used by the website are described and it also explains how to use the website correctly. Furthermore, on that website there is also a time converter from Julian time to calendar date. [19] [20]

8.3. Python

Over the last years Python has become one of the most used programming languages around the world because of its use in creating websites and software as well as automatizing tasks and doing data analysis. Furthermore, it is free and one of the easiest languages for beginners which propitiates its huge and active community. It is also a general-purpose programming language which means that it can be used by a wide variety of programs. Considering all these reasons, it is the perfect program to do some calculations for this project. [21]

9. Development of the project

The purpose of this section is to fully explain how this project has been developed. It is essential to mention that the entire process has been conducted using a transfer between Earth and Venus, instead of Earth and Uranus. The reason for this is to be able to perform all calculations in a faster and more understandable way. For example, the period of Uranus around the Sun is approximately 84 years, which means that from 2030 to 2040, Uranus hasn't even completed a quarter of its path around the Sun. In contrast, the period of Venus is 224.7 days which means that it goes around the Sun more than 10 times during the 10 years of study. Thus, as shown at Figure 9 and

Figure 10 and comparing them, working with Venus instead of Uranus can significantly reduce calculation time which accelerates the process. Furthermore, changing it doesn't take too much time and it demonstrates that the code works for any transfer between planets and not just between Earth and Uranus. However, since the project is about transfers between Earth and Uranus, and the code is the same, everything will be explained using these planets and the results with Venus will only be shown if necessary to explain something more clearly. Moreover, the full code is shown in Appendix G: Python code without interruptions.

[16] Trajectory browser user guide (NASA) - last access on June 20, 2024
[17] Characteristic energy - last access on June 20, 2024
[18] Astroquery - last access on July 2, 2024
[19] Horizons - last access on June 20, 2024
[20] Julian date/time converter - last access on June 20, 2024
[21] Python - last access on January 20, 2024

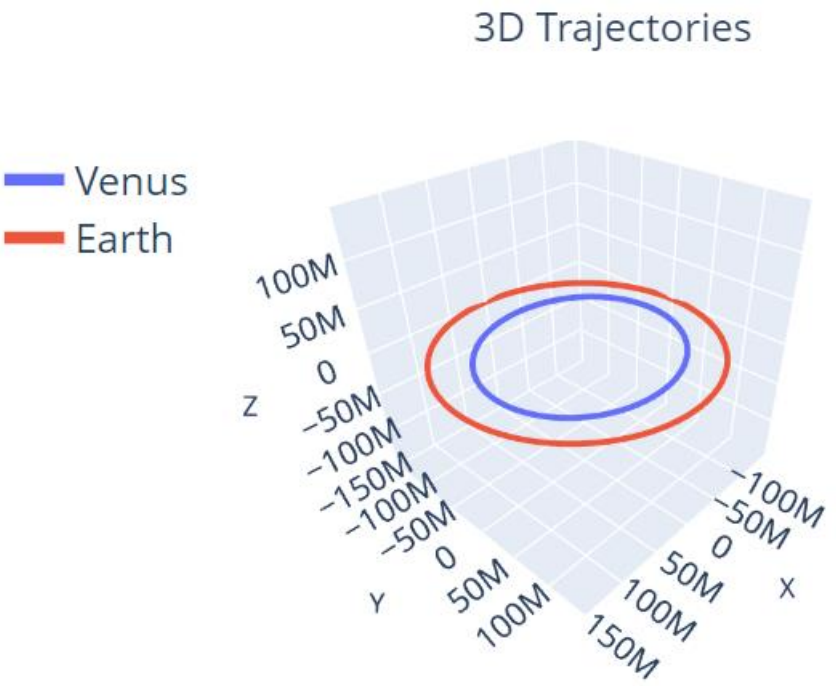

Figure 9: Venus and Earth orbits after 1 year.

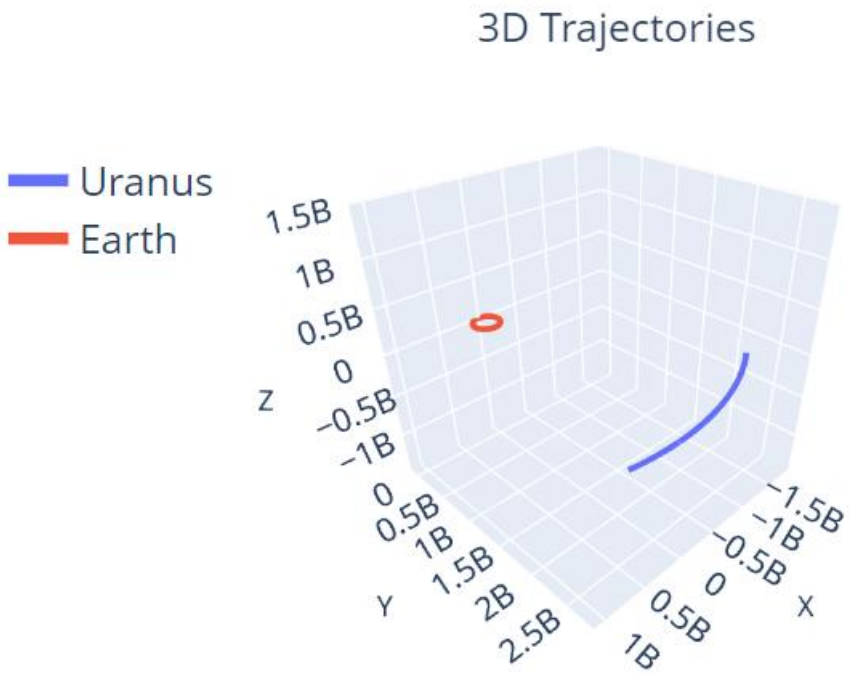

Figure 10: Uranus and Earth orbits after 1 year.

9.1. Generic calculations

First, and before starting with the algorithm to optimize the trajectories, some generic and simplified calculations have been done using some of the equations explained in previous sections such as 6.Basic concepts about astrodynamics and 7. Orbital maneuvers. This allows us to know about the expected results and their order of magnitude. However, it can't be forgotten that these results are just a reference and not the final result. Among all of them, the most relevant one is the difference between

calculating the delta-V using the low eccentricity of the planet's orbits or simplifying the orbits into circular ones (e = 0). The code explained at *B.4 deltaV_real_orbit_equation* and *B.5 deltaV_real_orbit_equation_energy* shows two methods to calculate the delta-V for a Hohmann transfer, the first one using the angular momentum and the second one using the Vis-Viva equation respectively. Obviously, as the transfer is around the Sun, μ will be μ_{Sun}.

For transfers between circular orbits, being the radius constant, semi-major axis is used basically because it represents the average radius of the real orbit. Therefore, delta-V for a Hohmann transfer is calculated in Figure 11. Furthermore, intending to clarify it, in this case, some of the simplifications made were: consider circular orbits of the planets, impulsive maneuver, starting and arrival points of the trajectory on the apse line and both planets are in the same orbital plane, which means that both have the same inclination.

```
98  ##############################################################################################
99  #                                    Delta-V. Circular orbits. Angular momentum
100 ##############################################################################################
101 #Import the function to use
102 from deltaV_real_orbit_equation import deltaV_real_orbit_equation
103 #Use the function to obtain the delta-V considering a circular orbit using the angular momentum
104 DeltaV_perigee_cir,DeltaV_apogee_cir,total_cir,e_departure_cir,e_arrival_cir = deltaV_real_orbit_equation
105 (mu_Sun,a_Earth,a_Earth,a_Uranus,a_Uranus)
106 print("cicurlar orbits, angular momentum","The delta-V at the perigee is: ",
107       DeltaV_perigee_cir,"km/s. At the apogee is: ",DeltaV_apogee_cir,"km/s.",
108       "\n The total delta-V is: ",total_cir,"km/s. The eccentricity of the departure"
109        "and arrival plantes are respectively: ",e_departure_cir," and ",e_arrival_cir)
110
111 ##############################################################################################
112 #                                    Delta-V. Circular orbits. Energy
113 ##############################################################################################
114 #Import the function to use
115 from deltaV_real_orbit_equation_energy import deltaV_real_orbit_equation_energy
116 #Use the function to obtain the delta-V considering a circular orbit using the angular momentum
117 DeltaV_perigee_cir,DeltaV_apogee_cir,total_cir,e_departure_cir,e_arrival_cir = deltaV_real_orbit_equation_energy
118 (mu_Sun,a_Earth,a_Earth,a_Uranus,a_Uranus)
119 print("Cicrular orbits, energy","The delta-V at the perigee is: ",DeltaV_perigee_cir,"km/s. At the apogee is: ",
120       DeltaV_apogee_cir,"km/s.","\n The total delta-V is: ",total_cir,"km/s. The eccentricity of the departure"
121        " and arrival plantes are respectively: ",e_departure_cir," and ",e_arrival_cir)
```

Figure 11: Simplified delta-V for circular orbits code.

On the other side, for real orbits there exists eccentricity which makes the radius vary between different values. Thus, at Figure 12, the radius of the apogee and perigee of the orbits of both planets is used. Moreover, in this case, the simplifications are the same as in the previous one apart from having ellipses as orbits of the planets.

```
118  ##############################################################################################
119  #                                    Delta-V. Real orbits. Angular momentum
120  ##############################################################################################
121  #Import the function to use
122  from deltaV_real_orbit_equation import deltaV_real_orbit_equation
123  #Use the function to obtain the delta-V considering a real orbit using the angular momentum
124  DeltaV_perigee,DeltaV_apogee,total,e_departure,e_arrival = deltaV_real_orbit_equation
125  (mu_Sun,ra_Earth,rp_Earth,ra_Uranus,rp_Uranus)
126  print("Real orbits, angular momentum","The delta-V at the perigee is: ",DeltaV_perigee,
127        "km/s. At the apogee is: ",DeltaV_apogee,"km/s.","\n The total delta-V is: ",total,
128        "km/s. The eccentricity of the departure and arrival plantes are respectively: ",
129        e_departure," and ",e_arrival)
130
131  ##############################################################################################
132  #                                    Delta-V. Real orbits. Energy
133  ##############################################################################################
134  #Import the function to use
135  from deltaV_real_orbit_equation_energy import deltaV_real_orbit_equation_energy
136  #Use the function to obtain the delta-V considering a real orbit using the angular momentum
137  DeltaV_perigee,DeltaV_apogee,total,e_departure,e_arrival = deltaV_real_orbit_equation_energy
138  (mu_Sun,ra_Earth,rp_Earth,ra_Uranus,rp_Uranus)
139  print("Real orbits, energy","The delta-V at the perigee is: ",DeltaV_perigee,
140        "km/s. At the apogee is: ",DeltaV_apogee,"km/s.",
141        "\n The total delta-V is: ",total,
142        "km/s. The eccentricity of the departure and arrival plantes are respectively: ",
143        e_departure," and ",e_arrival)
144
```

Figure 12: Simplified delta-V for real orbits code.

In addition to this, getting the time of flight for a Hohmann transfer is essential too, mainly because it will allow us to, in such way, adjust the maximum and minimum time of flight when doing the optimization. Therefore, the importance of this simplified time of flight lies in reducing the computational effort and the time it takes to get the results. The time of flight for a Hohmann transfer can be obtained using the equation from 7.2 Hohmann transfer. Figure 13.

```
71  ##############################################################################################
72  #                                    Time of flight
73  ##############################################################################################
74  #Import the function to use
75  from ellipse_period import ellipse_period
76  from sec2days import sec2days
77  from days2years import days2years
78  #Use the function to obtain the time of flight
79  TOF = ellipse_period(a_Earth,a_Uranus,mu_Sun)/2 #The TOF of a Hohmann transfer is half of the ellipse TOF
80  #Use the function to convert the TOF from seconds to days
81  TOF_days = sec2days(TOF)
82  #Use the function to convert the TOF from days to years
83  TOF_years = days2years(TOF_days)
84  print("The time of flight from Earth to Uranus is: ",TOF_days,"days = ",TOF_years,"years")
```

Figure 13: Simplified time of flight code.

Other parameters which could be useful to know are the sphere of influence of the planets and the synodic period. They can be obtained using Figure 63 and Figure 64 respectively as shown in Figure 14 and Figure 15.

```
52  ##############################################################################################
53  #                                    Sphere of influence
54  ##############################################################################################
55  #Import the function to use
56  from sphere_influence import sphere_influence
57  #Use the function to obtain the spehere of influence
58  r_SOI_Uranus = sphere_influence(a_Uranus,Mass_Uranus,Mass_Sun)
59  r_SOI_Earth = sphere_influence(a_Earth,Mass_Earth,Mass_Sun)
60  print("The SOI of Earth is: ",r_SOI_Earth,"km and the SOI of Uranus is: ",r_SOI_Uranus,"km" )
```

Figure 14: Sphere of influence code.

```
62  ##########################################################################################
63  #                                  Synodic period
64  ##########################################################################################
65  #Import the function to use
66  from synodic_period import synodic_period
67  #Use the function to obtain the synodic period
68  Tsyn = synodic_period(T_Earth,T_Uranus)
69  print("The synodic period is: ",Tsyn,"days")
```

Figure 15: Synodic period code.

Finally, once all these quick calculations have been made, it will be easier to identify possible mistakes when developing the optimization process.

9.2. Optimization process

When discussing the optimization process, it is necessary to mention that it has been developed following an additive methodology. This means that its development started with the simplest calculations. After completing one step, if the results matched the expected ones and were correct, more complexity was added in the next step. Nevertheless, this makes the code not exactly match the ordered sequence of work, as all the steps are mixed and are not completely independent. Therefore, the procedure will first be briefly shown step by step as its complexity arises, and then a more detailed explanation of the code will be provided. The sequence is as follows:

1. Get all the general data, such as the mass of the planets or the parameter μ.
2. Get the planetary parameters from Astroquery, particularly, the position vector of the planets.
3. Program a function to calculate the delta-V for the heliocentric portion when varying the time of flight. Here, it will be necessary to use Lambert's problem.
4. Plot delta-V vs time of flight and check that it matches the expected results.
5. Use the previous script, varying the time of flight, to program a more complex script which varies not only the time of flight but the starting date of the maneuver too.
6. Plot all the results using a porkchop plot. It may be necessary to slightly modify the previous scripts or write additional code to get all the variables in the necessary format or to organize them.
7. Focus on plotting the trajectories of the planets and the maneuver, as it is a more visual way to understand what is going on. Since only the delta-V has been obtained for the transfer maneuver, it will be necessary to obtain more information about it, such as the orbital elements, to be able to plot it.
8. Finally, once the heliocentric portion of the optimization process is complete, the launch portion of the project can be calculated. This involves the portion inside the sphere of influence of the launch planet, which in this case is Earth.

Throughout all these steps, there is a continuous process of checking the quality of the results. If the results are incorrect, adjustments are made to fix the process.

Once all these steps have been completed, the Python code is finished. By changing a few options at the beginning, the code allows for an optimization process to select the optimal trajectory between two planets.

Initially, the necessary modules and functions are imported to use throughout the script. For instance, the module "numpy" is used for simple math calculations such as finding the norm of a vector or computing a square root, among other operations, Figure 16.

```
###############################################################################################
#                                    Import general functions
###############################################################################################
import numpy as np
import matplotlib.pyplot as plt
from astroquery_vector_ecliptic import vector
from Lambert import Lambert
from oe_from_v_and_r import oe_from_v_and_r
import plotly.graph_objects as go
from datetime import datetime
import datetime
import os
from multiprocessing import Pool, cpu_count
from gregorian_date import gregorian_date
import pickle
import matplotlib.ticker as ticker
```

Figure 16: General functions code.

After importing the functions, some global parameters from *Appendix A: Physical data* that will be used multiple times throughout the code are defined. This approach avoids possible mistakes when typing them and makes it easy to change these parameters if the orbital bodies are changed, Figure 17.

```
###############################################################################################
#                                    Define global parameters
###############################################################################################
pi  = np.pi
pi2 = 2*pi

G = 6.6743*10**-20 #  (km^3/(kg*s^2))
AU = 149597870.69 #km

Mass_Sun = 1.989e30 # Mass of Sun (kg)
mu_sun = G*Mass_Sun # (km^3/s^2)

Mass_Earth = 5.9722e24 #Mass of Earth (kg)
mu_earth = G*Mass_Earth # (km^3/s^2)
R_Earth = 6378.136 #(km)
```

Figure 17: Global parameters code.

After this, the planetary data is imported from Astroquery. This can be done by defining a function called "vector", Figure 19, which provides information such as the position vector of the planet in the ecliptic plane or the datetime in Julian and Gregorian dates. It is essential to set the ecliptic plane as our reference plane in "vector", as we are working from the heliocentric point of view.

Once the "vector" function is properly defined, it is necessary to adequately define the required parameters, Figure 18.

- **Body:** In Astroquery, all the bodies have a number which identifies them, and this is the number we must introduce. For example, Earth is 399, Venus is 299 and Uranus es 799.
- **Start_date:** This is the first date we are going to import as data. It is crucial to write it in the correct format, which is YYYY-MM-DD.
- **End_date:** This is the last date we are going to import as data. It has the same format as "Start_date".
- **Step:** This is the time frame between two values. It also has a format, such as "7h" for 7 hours and "7d" for seven days.

On the other hand, the disadvantage is that the data provided by Astroquery isn't an array but a table with all the values, Figure 20. Thus, it will be necessary to define an array with the data to work with it properly.

```
###############################################################################################
#                                    Define Earth's and Uranus's initial parameters
###############################################################################################

# Earth
body = '399' # Earth has the number 399 assigned
start_date = '2025-01-01' # First date of the analysis
end_date = '2025-07-01' # Last date of the analysis
step = '1d' # Step used between start and end date

vec_earth = vector(body, start_date, end_date, step) #Use the function "vector" to get the data

#-------------------------------------------------------------------------

# Uranus
body = '799' # Uranus has the number 799 assigned
start_date = '2025-01-01' # First date of the analysis
end_date = '2025-07-01' # Last date of the analysis
step = '1d' # Step used between start and end date

vec_uranus = vector(body, start_date, end_date, step) #Use the function "vector" to get the data
```

Figure 18: Earth's and Uranus' initial parameters.

```
#This function is used to obtain from Astroquery the position and velocity vector of an orbit in the ecliptic plane
from astroquery.jplhorizons import Horizons
# import numpy as np
#
# Input:
#     body - the body's name e.g. '301' (The Moon)
#     start_date - first date interval 'yyy-mm-dd'
#     start_date - first date interval 'yyy-mm-dd'
def vector(body, start_date, end_date, step): #Define the function and the data to use

  obj = Horizons(id=body, location='500@10',
                 epochs={'start':start_date, 'stop':end_date,
                         'step':step})
  # oe = obj.elements_async(refplane='earth')
  vec = obj.vectors(refplane='ecliptic') #Selection of the ecliptic plane
  return vec
```

Figure 19: "vector" function.

targetname	datetime_jd	datetime_str	x	y	z
	d		AU	AU	AU
Earth (399)	2463323.5	A.D. 2032-Apr-01 00:00:00.0000	-0.9796063668981648	-0.1970926538110646	1.932829012271692e-05

Figure 20: Table from Astroquery.

As this is a brute force method of optimization, sometimes the time it takes to complete all the calculations is slightly high, particularly when the step is small and the period between the start and end dates spans several years. Therefore, one way to reduce the computational effort is to skip some unnecessary lines of the code and run them only with the optimal dates at the end of the process. Some methods to reduce this effort include avoiding the calculation of trajectories or displaying the plots on the screen. Thus, when the variables shown in Figure 21 are set to "1", those lines of the code are executed, otherwise they are skipped. After conducting some quick comparisons, it has been confirmed that processing time can be reduced by about half if neither of these elements is shown. (Bombardelli)

```
###############################################################################################
#                                  Set if the plots are shown or not. If not, the computational effort is reduced
###############################################################################################

plot_trajectory = 1  # If plot_trajectory=1, The plot is shown
plot_deltaV_launch = 0  # If plot_deltaV_launch=1, The plot is shown
plot_deltaV_heliocentric = 0  # If plot_deltaV_heliocentric=1, The plot is shown
print_results = 1  # If print_results=1, The results are shown
```

Figure 21: Variables to skip parts of the code.

The next step is to initialize some of the variables and set some options for the next calculations, Figure 22

- **Number_of_starting_dates_to_calculate:** This variable sets how many starting dates are going to be calculated. For instance, in this case this variable is set to 2, which means that from Figure 22, only the dates 2025-01-01 and 2025-01-02 will be calculated. This allows us to avoid calculating all the imported dates from Astroquery, reducing the computational effort by not calculating unnecessary starting dates. This is useful because when importing data from Astroquery, not only the data of the launch window will be imported, but also the data until the arrival is necessary. Continuing with the previous example, if the spacecraft departs on January 1st and the time of flight is 20 days, it will be necessary to import data from at least January 1st until January 21st, even though your launch window is just the first two days of the month. Thus, by setting this parameter to two, only the first two days of the month are considered as possible departure dates.
- **TOF_selection:** This parameter is used to choose the specific trajectory to be shown. In this case, the trajectory with a time of flight of 130 days will be shown.
- **Parking_orbit_selection:** This is the radius of the circular parking orbit around Earth before the launch. It is calculated as the radius of Earth plus the distance to the orbit from Earth's surface. In this project, we are considering that distance as 200km.
- **Launch_date:** This variable, using the function "Gregorian_date," converts the departure dates from Astroquery to Gregorian dates in a format called datetime.

```
###############################################################################################
#                                  Initialization of the variables
###############################################################################################
porkchop_plot = []
TOF = []
delta_v = []
delta_v1 = []
number_of_starting_days_to_calculate = 2
TOF_selection = 5800 #Select the specific TOF of the trajectory shown
Parking_orbit_selection = 200 + R_Earth #select in km the radius of the parking orbit
launch_date = gregorian_date(vec_earth, number_of_starting_days_to_calculate) #Set a variable with all lanunch dates
```

Figure 22: Initialization of some variables.

The function "Gregorian_date" is shown in Figure 23. It converts the date from Astroquery, given in string format, to a datetime format, which is easier to work with. This is done by obtaining all the dates from the variable "vec_earth," stacking them, and then changing their format.

```
#This function is used to convert the dates from Astroquery which are strings to dates in datetime format
from datetime import datetime
import numpy as np

def gregorian_date(vec_earth,j):

    dates_array = vec_earth[0][2] #Initilization of the variable
    for i in range(1,j):
        dates_array = np.vstack((dates_array,vec_earth[i][2])) #Stack the dates into a matrix
    # Convert the array of dates into a list of strings
    dates_str = [date[0] for date in dates_array]

    # Convert the date strings into datetime objects
    dates = [datetime.strptime(date.replace("A.D. ", ""), "%Y-%b-%d %H:%M:%S.%f") for date in dates_str]
    return dates
```

Figure 23: "gregorian_date" function.

Once this has been done, the part of the code which performs the calculations and saves the results can be run, Figure 24. At the beginning of this section, the name of the .pkl file and the directory where the results are saved can be defined.

```
###############################################################################
#                              Calculate and save the results
###############################################################################
if __name__ == '__main__':
    # Directory where the file is saved
    output_directory = r'C:\Users\gersa\OneDrive - UPV\Nueva carpeta\Universidad\Cuarto-USA\Space Systems & Mission Design\Nueva
    output_path = os.path.join(output_directory, '2020_01_01_to_2030_01_01_31d__starting_100.pkl') #Define the name of the file
```

Figure 24: Code to define the directory where the results are saved.

After that, the number of processors of the CPU is obtained and the tool "pool" from the module "multiprocessing" is used to use all the processors of the CPU at the same time. This allows us to reduce the calculation time because all the computational power is employed concurrently. This is necessary because before using this approach, the laptop was utilizing only 20% of the CPU; with multiprocessing, more than 90% of the CPU is utilized, resulting in a nearly 5-fold reduction in calculation time, which is essential when the program can run for hours. Furthermore, the function "process_date", which varies the launch date, is used, Figure 25

```
    num_processes = cpu_count()  #Get the number of processors of the CPU
    with Pool(processes=num_processes) as pool:
        results = pool.map(process_date, range(number_of_starting_days_to_calculate)) #Use all the processors ...
        # of the CPU at the same time to accelerate the calculations
```

Figure 25: Code to perform the calculations.

Finally, the results are stacked and saved. Furthermore, the program notifies if everything has been completed successfully or if any error occurred while saving, Figure 26.

```
    if results:
        print("Results generated successfully")
        porkchop_plot_array = np.vstack(results)
        print("Results stacked into an array")

        # Save the results in a file using pickle
        try:
            with open(output_path, 'wb') as file:
                pickle.dump(porkchop_plot_array, file)
                print(f"Results saved to {output_path}")
        except Exception as e:
            print(f"Failed to save results: {e}")
    else:
        print("No results to save")

    print("Done")
```

Figure 26: Code to save the results.

Now, going a step back and talking about "process_date" function, as it has been said before, it is the function which varies the stating date, working as a "for" loop, Figure 27. What it does is obtain from Astroquery the first Julian date, the position and velocity vector of Earth, and convert them to international system units. After that, all the data and variables are introduced into the function called "vary_TOF" which calculates the trajectory from that first starting date to all the departure dates at Uranus and provides all the results. After completing the calculations for the first starting date, it repeats the process for the second, third, fourth, etc., up to the number of starting dates specified previously in

'Number_of_starting_dates_to_calculate".

```
################################################################################
#                              Define the function to vary the Launch date and do the calculations
################################################################################
def process_date(i):
    JD0 = vec_earth['datetime_jd'][i]
    r_vec_earth = np.array([vec_earth['x'][i] * AU, vec_earth['y'][i] * AU, vec_earth['z'][i] * AU]) #Convert to international system
    r_earth = np.linalg.norm(r_vec_earth) #Calculate the norm
    v_vec_earth = np.array([vec_earth['vx'][i] * AU / 24 / 3600, vec_earth['vy'][i] * AU / 24 / 3600,
                            vec_earth['vz'][i] * AU / 24 / 3600]) #Comvert to international system
    v_earth = np.linalg.norm(v_vec_earth) #Calculate the norm
    solu, porkchop_plot = vary_TOF(vec_uranus, JD0, r_vec_earth, v_vec_earth, TOF_selection, vec_earth, plot_trajectory,
                                   Parking_orbit_selection, plot_deltaV_launch, launch_date[i], plot_deltaV_heliocentric, print_results)
    return porkchop_plot
```

Figure 27: Process_date function.

About the function “vary_TOF”, it is also divided into some subsections. The first and second subsections are the same as Figure 16 and Figure 17, as they are about importing general functions and defining general data. Nevertheless, the third and fourth subsections are about calculating the heliocentric and launch portion of the trajectory, respectively. Both sections are included in a “for” loop, which helps to change the arrival date of the trajectory.

Inside the loop, in the first lines of code, the Julian date at arrival is obtained and it will vary with the loop, covering all the possible arrival dates. Furthermore, the time of flight can be calculated as the difference of time between the launch and the arrival date, Figure 28.

```
    #Loop to vary the arrival date and, therefore, the time of flight
    for i in range(0, len(vec_venus)):

        ##############################################################################################
        #                                      Heliocentric portion
        ##############################################################################################
        if print_results == 1:
            print(f"Julian Date at Earth: {JD0}")
        JD = vec_venus['datetime_jd'][i] #Julian date at arrival
        if print_results == 1:
            print(f"Julian Date at Venus: {JD}")

        #Calculate the time of flight using the launch and arrival dates
        delta_t_days = JD - JD0
        delta_t = delta_t_days*24*3600
        if print_results == 1:
            print(f"delta_t = {delta_t_days} days = {delta_t} seconds")
```

Figure 28: Julian dates and time of flight.

The following lines continue with the loop if the time of flight is negative or break it if a specific time of flight is exceeded. A negative time of flight can occur, for example, when the script is calculating all possible maneuvers within a week. For instance, if the starting date is on Monday, all arrival dates from Monday to Sunday are possible. However, if the starting date is on Tuesday and the script tries to calculate with an arrival date on Monday, the time of flight is negative, rendering the maneuver impossible. In such cases, the script stops further calculations and skips to the next arrival date. Moreover, breaking the loop when the time of flight is greater than a specific number also reduces the computational effort and the calculation time. Consider a transfer from Earth to Venus, for example. We may be considering a launch window of 10 years for this transfer, but a Hohmann transfer between Earth and Venus is about half a year. Thus, calculations for a specific launch date will stop when the time of flight exceeds the set limit of 3 years, and the loop will continue with the next launch date. Figure 29.

```
        # Skip negative time of flight
        if delta_t_days < 0:
            continue

        #Breaks the for loop if the TOF is greater than a specific the number of days.
        if delta_t_days > 61320:
            break
```

Figure 29: Save computational effort.

Now, as done previously with Earth, the position and velocity vectors are obtained and converted to international system. Furthermore, the position vector of Uranus is saved in a variable for later use in plots. Figure 30

```
        # Loading Venus's state vector into a numpy array (and converting them to km and km/s)
        r_vec_venus = np.array([vec_venus['x'][i]*AU, vec_venus['y'][i]*AU,
                                vec_venus['z'][i]*AU]) #Convert to international system units
        r_venus = np.linalg.norm(r_vec_venus) #Calculate the norm of the vector
        # print(f"r_vec_venus = {r_vec_venus} (km), r_venus = {r_venus} km")
        r_vec_venus_plot.append(r_vec_venus) #Append the r vector into "r_vec_venus_plot" to be able to plot it later

        v_vec_venus = np.array([vec_venus['vx'][i]*AU/24/3600, vec_venus['vy'][i]*AU/24/3600,
                                vec_venus['vz'][i]*AU/24/3600]) #Convert to international system units
        v_venus = np.linalg.norm(v_vec_venus) #Calculate the norm of the vector
```

Figure 30: Import position and velocity vectors.

At this point of the optimization process and, after setting a prograde orbit, the necessary velocities at launch and arrival can be calculated using Lambert's problem.

To solve Lambert's problem, the function "Lambert" should be used. At the beginning of the function the orbit's orientation is selected and the magnitude of the initial values calculated, Figure 31. In our case, the orbit is a prograde orbit as it will be set at Figure 37.

```
# Create numerical key based on the orbit's orientation
if (orientation.lower() == 'prograde'):
  prograde = 1 # prograde orbit
else:
  prograde = 0 # retrograde orbit

#-----------------------------------------------------------------------
# Calculate the magnitude of the initial vectors
r1 = np.linalg.norm(r1_vec)
r2 = np.linalg.norm(r2_vec)
if print_results == 1:
  print(f"r1 = {r1:0.5f} km")
  print(f"r2 = {r2:0.5f} km")
```

Figure 31: Type of orbit at Lambert's problem.

The next step is to calculate the delta of true anomaly using Equation 3 as explained in *6.6Lambert's problem*. The auxiliary variable *A*, Equation 7, is obtained here too. Figure 32

```
#-------------------------------------------------------------------------
# Calculate deltatheta
# Checking the sign of the z-component of r1_vec cross product r2_vec

cross_product = np.cross(r1_vec, r2_vec)
sign = np.sign(cross_product[2])

argument = np.dot(r1_vec, r2_vec)/(r1*r2)

# Calculate deltatheta according to the orbit's orientation

if prograde == 1:
    if sign >= 0:
        deltatheta = np.arccos(argument)
    else:
        deltatheta = pi2 - np.arccos(argument)
else:
    if sign < 0:
        deltatheta = np.arccos(argument)
    else:
        deltatheta = pi2 - np.arccos(argument)

if print_results == 1:
  print(f"deltatheta = {np.rad2deg(deltatheta):.2f} deg")

#-------------------------------------------------------------------------
# Calculate the auxiliary variable A
A = np.sin(deltatheta)*np.sqrt((r1*r2)/(1 - np.cos(deltatheta)))
# print(f"A = {A} km")
```

Figure 32: Delta of true anomaly at Lambert's problem.

Now, the Stumpp functions and other necessary functions are defined as it is an easier way to work with them. These equations are Equation 7 and Equation 8. Particularly, as the Stumpp functions have infinite terms, in this case just the first four terms will be used as it is enough to precisely work with our data. Figure 33.

```
#-------------------------------------------------------------------------
# By iteration, find z
z0 = 0 # initial value for z

def S(z):
    S_value = 1/6 - z/120 + z**2/5040 - z**3/362880
    return S_value

def C(z):
    C_value = 1/2 - z/24 + z**2/720 - z**3/40320
    return C_value

def y(z):
    y_value = r1 + r2 + A*(z*S(z) - 1)/(np.sqrt(C(z)))
    return y_value

def F(z):
    F_value = (y(z)/C(z))**1.5*S(z) + A*np.sqrt(y(z)) - np.sqrt(mu)*delta_t
    return F_value

def F_prime(z):
    if z == 0:
        term_1 = np.sqrt(2)/40*y(0)**1.5
        term_2 = np.sqrt(y(0)) + A*np.sqrt(1/(2*y(0)))
    else:
        term_1 = (y(z)/C(z))**1.5*(1/(2*z)*(C(z) - 1.5*S(z)/C(z)) + 0.75*S(z)**2/C(z))
        term_2 = 3*S(z)/C(z)*np.sqrt(y(z)) + A*np.sqrt(C(z)/y(z))

    F_prime_value = term_1 + A/8*term_2
    return F_prime_value
```

Figure 33: Functions to solve Lambert's problem.

Once all the functions have been defined, Newton's method is used to calculate z value, Figure 34. Furthermore, at this step if there is not convergence when using the Newton's method, the velocity vectors are set as zero, so it will be able to skip to the next arrival date as it will be shown at Figure 37.

```
# Newton Method
def Newton(x_0):
    nmax = 10000
    convergence = 1

    for j in range(0, nmax):

        f = F(x_0)
        f_prime = F_prime(x_0)
        ratio = f/f_prime
        # stop when achieve the desired precision
        if np.abs(ratio) <= 1.0e-6:
            break

        x =  x_0 - ratio
        x_0 = x #update the value of x_0

        if j == nmax - 1:
            if print_results == 1:
                print(f"Newton: no convergence.")
            convergence = 0
    return x, convergence
z, convergence = Newton(z0)

if convergence == 0:
    v1_vec = np.zeros(3)
    v2_vec = np.zeros(3)
    return v1_vec, v2_vec
```

Figure 34: Newton's method to solve Lambert's problem.

As shown in Figure 35, depending on the value of z, the type of orbit can be stated. If z is a negative number the orbit is a hyperbola, if it is zero the orbit will be a parabola while if it is greater than zero and, therefore, positive, the orbit will be an ellipse. Therefore, as we are working with transfer orbits, all our orbits will be an ellipse and z greater than zero.

```
if z < 0:
    if print_results == 1:
        print(f"z = {z}; the orbit is hyperbolic.")
elif z == 0:
    if print_results == 1:
        print(f"z = {z}; the orbit is parabolic.")
else:
    if print_results == 1:
        print(f"z = {z}; the orbit is elliptic.")
```

Figure 35: Type of orbit at Lambert's problem.

Finally, using the Lagrange coefficients Equation 6, the velocities at both points of the trajectory can be calculated as shown in Figure 36.

```
#-----------------------------------------------------------------------
# Calculate the Lagrange Functions
# Defining the Lagrange coefficients using the iterated value of z:
f = 1 - y(z)/r1
g = A*np.sqrt(y(z)/mu)
gdot = 1 - y(z)/r2

if print_results == 1:
  print(f"f = {f}, g = {g} s, gdot = {gdot}")

#-----------------------------------------------------------------------
# Finally, calculate the velocity vectors at r1 and r2
# Calculating the velocities using the Lagrange coefficients:
if print_results == 1:
  print(r1_vec, f)
  print(f*r1_vec)

v1_vec = 1/g*(r2_vec - f*r1_vec)
v2_vec = 1/g*(gdot*r2_vec - r1_vec)

if print_results == 1:
  print(v1_vec)
  print(v2_vec)

return v1_vec, v2_vec
```

Figure 36: Lagrange functions to calculate velocities at Lambert's problem.

Once Lambert's problem has been solved, the code proceeds to the next starting date if both velocities are zero because that indicates there wasn't a solution to the problem. Obviously, this is another method to reduce the computational method and running time. After solving Lambert's problem and confirming a solution exists, the magnitudes of both velocities are then obtained.

```
    # Calculating the spacecraft's velocity vectors at Earth and Venus at the corresponding dates
    orientation = 'prograde' #Choose the orientation of the orbit

    v1_vec, v2_vec = Lambert(mu_sun, r_vec_earth, r_vec_venus, orientation, delta_t,print_results)
    #Use the Lambert's problem to get the velocities
    if v1_vec.all() == 0 and v2_vec.all() == 0: #Continue with next dates if both velocities
        #are zero because there is no solution. This reduces the computational effort.
        if print_results == 1:
            print("No solution for this date.\n")
        continue

    v1 = np.linalg.norm(v1_vec) # magnitude of the vector v1
    if print_results == 1:
        print(f"v1_vec = {v1_vec} (km/s), v1 = {v1} km/s")
    v2 = np.linalg.norm(v2_vec) # magnitude of the vector v2
    if print_results == 1:
        print(f"v2_vec = {v2_vec} (km/s), v2 = {v2} km/s.")
```

Figure 37: Lambert's problem to calculate velocities at departure and arrival.

Now, the delta-V required at departure can be obtained as the difference between the velocity of the transfer orbit at departure and Earth's velocity at that moment. Similarly, the delta-V at arrival can be obtained as the difference between Uranus' velocity at the arrival and the velocity of the transfer at arrival. There are two methods to perform these calculations as explained in *Appendix C: Comparison between calculating the delta-V using vectors or the flight path angle*, however, in this project, the velocity vectors will be directly subtracted for simplicity, speed of computation, and because this method is not limited to coplanar orbits. Finally, the total delta-V of the transfer maneuver is obtained as the sum of both delta-Vs.

Furthermore, the characteristic speed, *C3*, is obtained, as it will be necessary for the porkchop plot. Figure 38

```
    # Calculate delta-v1, delta-v2, and delta-v (total)
    dv1_vec = np.zeros(3) #Initilization of the variable

    dv1_vec = v1_vec - v_vec_earth #Calculate the delta-V at launch
    dv1 = np.linalg.norm(dv1_vec) #Calculate the norm of the vector
    if print_results == 1:
        print(f"dv1_vec = {dv1_vec} (km/s), dv1 = {dv1} km/s.")
    dv1_vec_plot.append([delta_t_days,v1_vec[0],v1_vec[1],v1_vec[2]])
    #Append the velocity and the TOF to plot it later

    dv2_vec = np.zeros(3) #Initilization of the variable
    dv2_vec = v_vec_venus - v2_vec #Calculate the delta-V at arrival
    dv2 = np.linalg.norm(dv2_vec) #Calculate the norm of the vector
    if print_results == 1:
        print(f"dv2_vec = {dv2_vec} (km/s), dv2 = {dv2} km/s.")

    dvtotal = dv1 + dv2 #Caluclate total delta-V
    if print_results == 1:
        print(f"Total delta-v = {dvtotal}")

    solution.append([delta_t_days, dvtotal]) #Save the results

    C3 = dv1**2 #Calculate the value of C3
```

Figure 38: Calculation of delta-V.

To conclude with the heliocentric portion, the Gregorian date of the arrival is calculated by adding the time of flight converted to a datetime format and the Gregorian date at launch. Furthermore, using the results from the Lambert's problem, the orbital elements of the transfer trajectory are calculated using the function called "oe_from_v_and_r". Figure 39

```
    #Calculate the gregorian date at the arrival from the launch date and the TOF
    gregorian_arrival_date = gregorian_launch_date + datetime.timedelta(days=delta_t_days)

    porkchop_plot_solution.append([JD0,delta_t_days, dvtotal,gregorian_launch_date,gregorian_arrival_date,C3])
    #Append the results to use in the porkchop plot
    if print_results == 1:
        print("\n")

    #Calculate and append the orbital elements of the transfer orbit
    r1,v1,vr1,h_vec1,h1,incl1,k_dir1,Nodal_vec1,Nodal1,Omega1,e_vec1,e1,w1,theta1,a1 = oe_from_v_and_r(r_vec_earth,v1_vec,mu_sun)
    r2,v2,vr2,h_vec2,h2,incl2,k_dir2,Nodal_vec2,Nodal2,Omega2,e_vec2,e2,w2,theta2,a2 = oe_from_v_and_r(r_vec_venus,v2_vec,mu_sun)

    oe1_solution.append([delta_t_days,r1,v1,vr1,h1,incl1,Nodal1,Omega1,e1,w1,theta1,a1])
    oe2_solution.append([delta_t_days,r2,v2,vr2,h2,incl2,Nodal2,Omega2,e2,w2,theta2,a2])
```

Figure 39: Calculate orbital elements of the transfer trajectory.

The function "oe_from_v_and_r" consists of the equations explained in *Appendix E: Orbital elements from velocity and position*, which can be used to obtain the orbital elements of a trajectory when the position and velocity vectors are known, Figure 40 and Figure 41. The equations used are Equation 30, Equation 31, Equation 32, Equation 33, Equation 34, Equation 35, Equation 36 and Equation 37.

```
    # Step 1 ------------------------------------------------------
    r = np.linalg.norm(r_vec) # magnitude of the position vector
    # Step 2 ------------------------------------------------------
    v = np.linalg.norm(v_vec) # magnitude of the velocity vector
    # Step 3 ------------------------------------------------------
    vr = np.dot(r_vec, v_vec)/r
    # Step 4 ------------------------------------------------------
    h_vec = np.cross(r_vec, v_vec) # vector angular momentum
    # Step 5 ------------------------------------------------------
    h = np.linalg.norm(h_vec) # magnitude of the angular momentum
    # Step 6 ------------------------------------------------------
    i = np.arccos(h_vec[2]/h) # inclination
    # Step 7 ------------------------------------------------------
    # Defining the direction k and the Nodal vector
    k_dir = np.array([0.0, 0.0, 1.0])
    #-------------------------------------------------------------
    Nodal_vec = np.cross(k_dir, h_vec)
    # Step 8 ------------------------------------------------------
    Nodal = np.linalg.norm(Nodal_vec)
    # Step 9 ------------------------------------------------------
    # Longitude of the ascending node
    if Nodal_vec[1] >= 0.0:
        Omega = np.arccos(Nodal_vec[0]/Nodal)
    else:
        Omega = pi2 - np.arccos(Nodal_vec[0]/Nodal)
```

Figure 40: Orbital elements calculation.

```
    # Step 10 -----------------------------------------------------
    # Eccentricity vector
    e_vec = np.cross(v_vec, h_vec)/muSun - r_vec/r
    # Step 11 -----------------------------------------------------
    e = np.sqrt(e_vec[0]**2 + e_vec[1]**2 + e_vec[2]**2)
    # Step 12 -----------------------------------------------------
    # Argument of the perigee
    if e_vec[2] >= 0.0:
        w = np.arccos(np.dot(Nodal_vec,e_vec)/(Nodal*e))
    else:
        w = pi2 - np.arccos(np.dot(Nodal_vec,e_vec)/(Nodal*e))
    # Step 13 -----------------------------------------------------
    if np.dot(r_vec,v_vec) >= 0.0:
        theta = np.arccos(np.dot(e_vec,r_vec)/(e*r))
    else:
        theta = pi2 - np.arccos(np.dot(e_vec,r_vec)/(e*r))
    # Semi-major axis
    a = h**2/muSun/(1 - e**2)
    return r,v,vr,h_vec,h,i,k_dir,Nodal_vec,Nodal,Omega,e_vec,e,w,theta,a
```

Figure 41: Orbital elements calculation.

After completing the heliocentric calculations, the patched conics method can be used to calculate the necessary delta-V to transition from the circular parking orbit to the transfer orbit.

As mentioned previously at 7.5 Patched conics, this trajectory is hyperbolic relative to the planet, with the velocity at the sphere of influence being greater than zero. The delta-V at launch calculated earlier serves as the hyperbolic excess speed. Therefore, at this point, both conics—the transfer ellipse and the departure hyperbola—are connected, with the hyperbolic excess speed simply renamed when viewed from the planet's perspective.

Using the equations from *Appendix F: Planetary departure equations*, the speed at the periapsis of the circular 200-km altitude parking orbit and the other parameters can be obtained, Figure 42. This completely defines the launch orbit, which is illustrated similarly in Figure 43.

```
###############################################################################
#                                 Launch portion
###############################################################################
vp = np.sqrt((dv1**2) + (2*mu_earth/rp)) #Calculate the periapsis speed at launch
vc = np.sqrt(mu_earth/rp) #Calculate the speed at the circular parking orbit
delta_V_launch = vp-vc #Calculate the delta-V to put the vehicle onto the hyperbolic departure trajectory
e = 1+rp*dv1**2/mu_earth #Calculate the eccentricity of that departure trajectory
beta = np.arccos(1/e) #Calculate the orientation of the apse line of the hyperbola to the planet's heliocentric velocity vector

delta_v1_sol.append([delta_V_launch,e,beta,JD0,delta_t_days,gregorian_launch_date]) #Append the results
```

Figure 42: Launch calculations inside the sphere of influence.

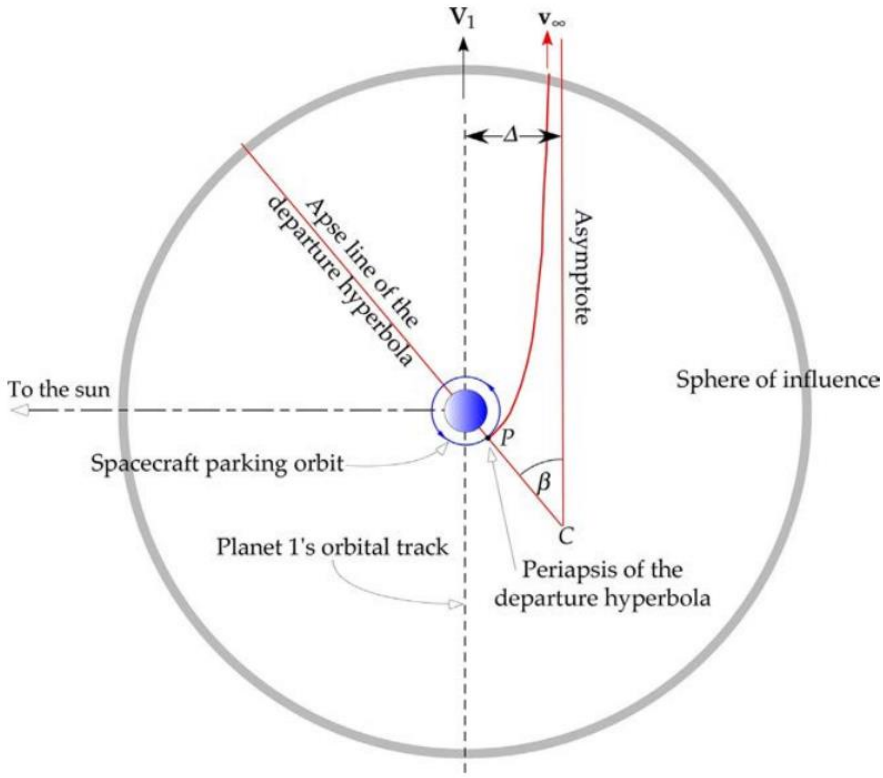

Figure 43: Departure orbit for a mission from a inner to an outer planet. (Curtis, 2020)

Once the loop has finished, all the results are converted to arrays. Figure 44

```
###############################################################################################
#                                        Results
###############################################################################################
#Convert all the results into arrays
delta_v1_sol_array = np.array(delta_v1_sol)
porkchop_plot_solution_array = np.array(porkchop_plot_solution)
dv1_vec_plot_array = np.array(dv1_vec_plot)
r_vec_venus_plot_array = np.array(r_vec_venus_plot)
sol = np.array(solution)
sol_oe1 = np.array(oe1_solution)
sol_oe2 = np.array(oe2_solution)
```

Figure 44: Code results.

The next step is to create the plots to easily analyze the results. The first one is delta-V vs time of flight from a heliocentric point of view. It is important to note that this plot is two-dimensional and represents data for a single departure date. Figure 45. Figure 46 provides an example of what this plot looks like. In addition to this, it can be easily observed on the right of Figure 46 how the density of points is lower than on the left. The reason for this variation in data density is the lack of possible maneuvers; if the Lambert problem doesn't converge, there are no points to plot.

```
##########################################################################################
#                                  Plot the total delta-v results. Heliocentric
##########################################################################################
if plot_deltaV_heliocentric ==1:
    #Plot total Delta-V vs TOF
    plt.scatter(sol[:, 0], sol[:, 1], marker='.')
    plt.xlabel('Time of Flight, days', fontsize=16)
    plt.xticks(fontsize=12)
    plt.ylabel('Total Delta-v, km/s', fontsize=16)
    plt.yticks(fontsize=12)
    plt.title('Total Delta-V vs Time of Flight', fontsize=18)
    # Annotation for departure date
    plt.annotate(f'Departure date: {gregorian_launch_date.strftime("%Y-%m-%d")}',
                 xy=(0.5, 0.9), xycoords='axes fraction', fontsize=12, ha='center')
    plt.show()
```

Figure 45: Code of the total delta-V vs TOF plot.

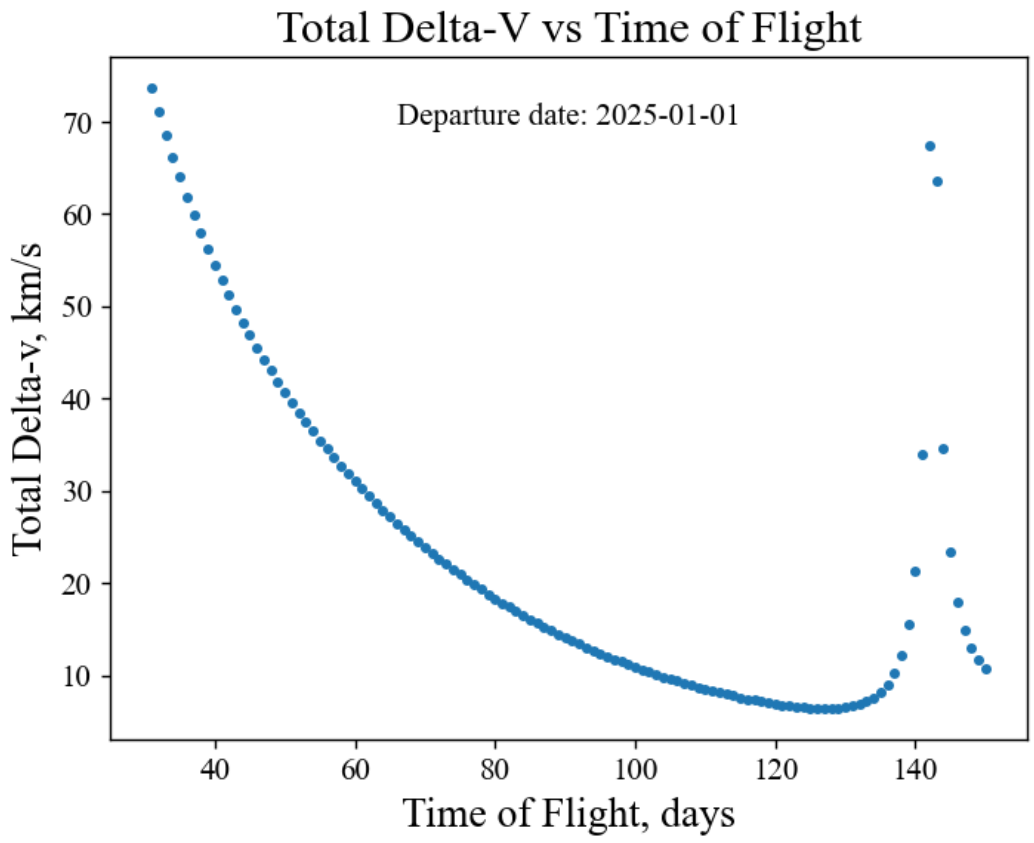

Figure 46: Example of the total delta-V vs TOF plot.

Now, the delta-V at launch, from the planet's point of view, is plotted against time of flight and date Figure 47. An example of such plot is shown in Figure 48.

```
###############################################################################################
#                                    Plot the delta-v results. Launch
###############################################################################################
if plot_deltaV_launch == 1:
    #Plot delta-V at launch vs TOF
    plt.scatter(delta_v1_sol_array[:, 4], delta_v1_sol_array[:, 0], marker='.')
    plt.xlabel('Time of Flight, days', fontsize=16)
    plt.xticks(fontsize=12)
    plt.ylabel('Delta-v at launch, km/s', fontsize=16)
    plt.yticks(fontsize=12)
    plt.title('Total Delta-V at launch vs Time of Flight', fontsize=18)
    plt.annotate(f'Departure date: {gregorian_launch_date.strftime("%Y-%m-%d")}',
                 xy=(0.5, 0.9), xycoords='axes fraction', fontsize=12, ha='center')

    plt.show()

    #Plot delta-V at launch vs Date
    plt.scatter(delta_v1_sol_array[:, 5], delta_v1_sol_array[:, 0], marker='.')
    plt.xlabel('Date', fontsize=16)
    plt.xticks(rotation=45, fontsize=12)
    plt.ylabel('Delta-v at launch, km/s', fontsize=16)
    plt.yticks(fontsize=12)
    plt.title('Total Delta-V vs date', fontsize=18)
    plt.show()
```

Figure 47: Code of the total delta-V at launch vs TOF. Earth's perspective.

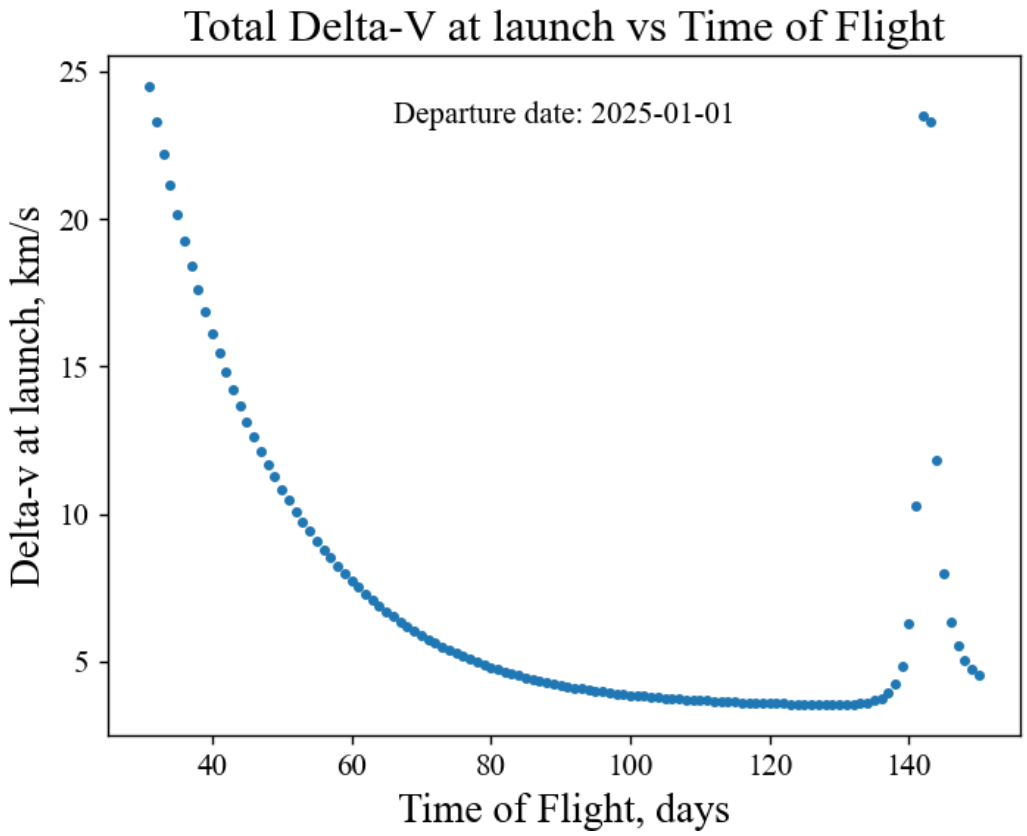

Figure 48: Example plot of the total delta-V at launch vs TOF. Earth's point of view.

In addition to those delta-V plots, it is interesting to see the trajectory of the spacecraft from Earth to Uranus. However, this requires the calculation of some points of the transfer trajectory as now we only have the orbital elements of the transfer trajectory. As these calculations are not completely related to the optimizing method, the process to calculate the trajectory and plot it will be shown at *Appendix D:*

Calculate trajectory. Furthermore, it can be interesting to see how the spacecraft and the planets orbit around the solar system, thus an animation can be created when knowing the trajectory. This animation is explained at *Appendix D: Calculate trajectory* too. An example of the trajectory plot is shown in Figure 49.

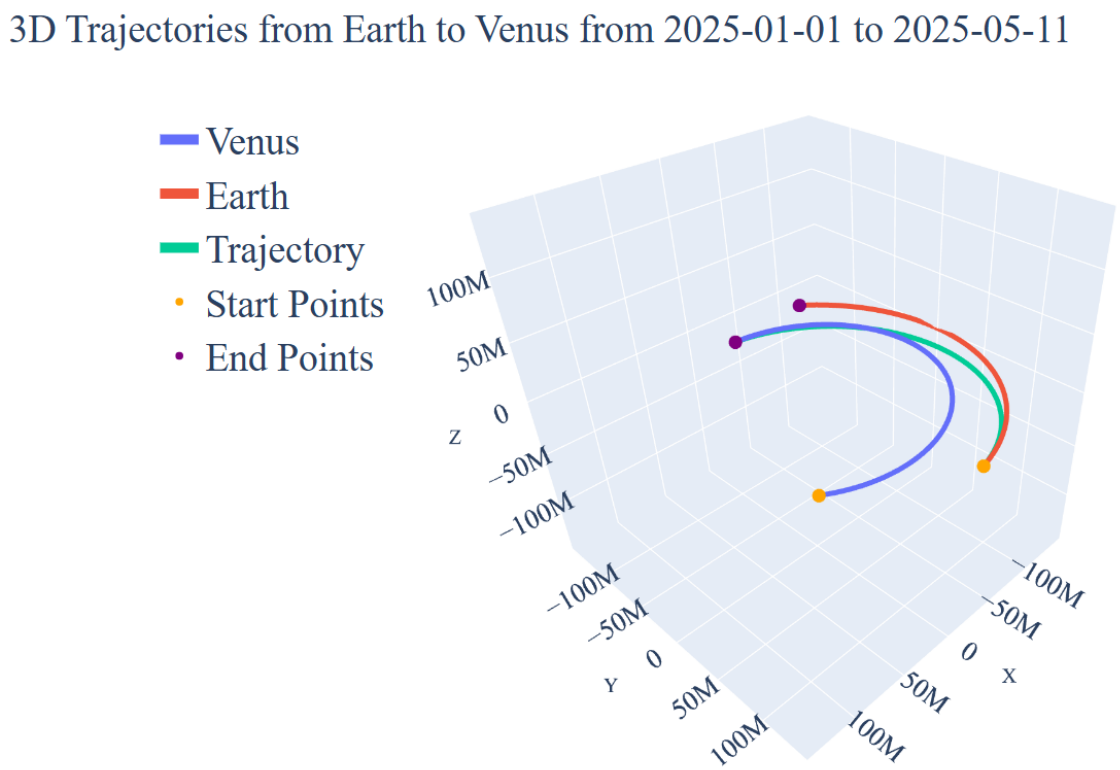

Figure 49: Example of the trajectory.

Once all the calculations have been completed and the results have been saved, the next step is to organize them effectively for proper analysis. Since the results are saved in a file, and analysis may not be immediate after running the script, the first step in the analysis process is to import these results. This can be achieved using a Python module called “pickle,” as demonstrated in Figure 50, where replacing "name_of_the_file" with the actual chosen filename will load the results into the variable "loaded_results".

```
# Read the results from the file using pickle
with open('name_of_the_file.pkl', 'rb') as file:
    loaded_results = pickle.load(file)
```

Figure 50: Code to read the results from a .pkl file.

Nevertheless, the best way to organize the results not only to plot them, but also to be easily understood if read, is to define a matrix where the rows are the departure dates and the columns the arrival dates. Thus, the first step to organize the data is to know the unique launch and arrival dates as most of them are repeated. Once that information is known, the matrices can be initialized, the data sorted and the dates converted from strings, as they are imported from Astroquery, to datetime format. Figure 51

```python
# Extract unique launch and arrival dates. "Set" removes duplicates and "list" converts them into a list.
unique_launch_dates = list(set(row[3] for row in data))
unique_arrival_dates = list(set(row[4] for row in data))

# Sort the dates to ensure the labels are in chronological order
unique_launch_dates.sort()
unique_arrival_dates.sort()

# Initialize the results matrices. The lenght is the number of unique dates.
results_C3 = np.full((len(unique_launch_dates), len(unique_arrival_dates)), np.nan)
results_delta_v = np.full((len(unique_launch_dates), len(unique_arrival_dates)), np.nan)
results_time_of_flight = np.full((len(unique_launch_dates), len(unique_arrival_dates)), np.nan)

# Fill the matrices with the corresponding values
for row in data:
    launch_date = row[3]
    arrival_date = row[4]
    C3 = row[5]
    delta_v = row[2]
    time_of_flight = row[1]
    i = unique_launch_dates.index(launch_date)
    j = unique_arrival_dates.index(arrival_date)
    results_C3[i, j] = C3
    results_delta_v[i, j] = delta_v
    results_time_of_flight[i, j] = time_of_flight

# Convert dates to strings to use as labels
row_labels = [date.strftime('%Y-%m-%d') for date in unique_launch_dates]
column_labels = [date.strftime('%Y-%m-%d') for date in unique_arrival_dates]
```

Figure 51: Unique dates and data to sort the results.

The last step is to print the matrices in the best way to be visually understood and print the minimum delta-V values. Figure 52.

```python
# Create a function to print the results matrix with labels
def print_matrix(matrix, rows, columns, name):
    print(f"\nResults matrix for {name}:")
    print(" " * 10, end="")
    for column in columns:
        print(f"{column:>10}", end=" ")
    print()

    for i, row in enumerate(matrix):
        print(f"{rows[i]:<10}", end="")
        for value in row:
            if np.isnan(value):
                print(f"{'NaN':>10}", end=" ")
            else:
                print(f"{value:>10.2f}", end=" ")
        print()

# Display the results matrices
#print_matrix(results_C3, row_labels, column_labels, "C3")
#print_matrix(results_delta_v, row_labels, column_labels, "Delta V")
print_matrix(results_time_of_flight, row_labels, column_labels, "Time of Flight")

# Find the minimum delta-v value and its indices. "nanmin" is used to not consider "nan" values
min_delta_v = np.nanmin(results_delta_v)
min_indices = np.where(results_delta_v == min_delta_v)

# Print the minimum delta-v and corresponding dates
print(f"\nMinimum Delta V: {min_delta_v:.2f}")
print("Corresponding Launch and Arrival Dates:")
for idx in zip(min_indices[0], min_indices[1]):
    launch_date = unique_launch_dates[idx[0]]
    arrival_date = unique_arrival_dates[idx[1]]
    print(f"Launch Date: {launch_date.strftime('%Y-%m-%d')}, Arrival Date: {arrival_date.strftime('%Y-%m-%d')}")
```

Figure 52: Sort the results and print them.

When plotting a porkchop plot, it must be correctly dimensioned, as it is one of the main requirements to be understood and useful. Thus, throughout the function “plot_porkchop” some characteristics such as the tick spacing of the axis and the font size are edited. Figure 53.

```python
#Function to customize the labels and number of ticks
def plot_porkchop(title, X, Y, Z, clevels, Z_time_of_flight):

    def set_ticks(ax):
        # Adjust the tick spacing to generate fewer ticks
        #x_tick_spacing, y_tick_spacing = max(1, int(len(X[0]) / 300)), max(1, int(len(Y) / 300))
        x_tick_spacing, y_tick_spacing = max(1, int(len(X[0]) / 8)), max(1, int(len(Y) / 8))
        ax.xaxis.set_major_locator(ticker.MultipleLocator(x_tick_spacing))
        ax.yaxis.set_major_locator(ticker.MultipleLocator(y_tick_spacing))
        ax.xaxis.set_tick_params(labelsize=10, rotation=90)
        ax.yaxis.set_tick_params(labelsize=10)
        ax.set_xlabel("Arrival Date (yyyy-mm-dd)", fontsize=12)
        ax.set_ylabel("Departure Date (yyyy-mm-dd)", fontsize=12)
        ax.grid(which='major', linestyle='dashdot', linewidth='0.5', color='gray')
        ax.grid(which='minor', linestyle='dotted', linewidth='0.5', color='gray')
        ax.minorticks_on()
        ax.tick_params(which='both', top=False, left=False, right=False, bottom=False)
        return
    #Add C3 contour
    plt.figure(figsize=(15, 15))  # Adjust the figure size
    cp1 = plt.contour(X, Y, Z, clevels, cmap="rainbow")
    plt.clabel(cp1, inline=True, fontsize=10)

    # Add time of flight contour with thicker lines and a distinct color
    cp2 = plt.contour(X, Y, Z_time_of_flight, colors='black', linewidths=1.5)
    plt.clabel(cp2, inline=True, fontsize=10, fmt='%1.0f')

    #Customize the figure adding title and x and y limits
    plt.title(title, fontdict={'fontsize': 16})
    ax = plt.gca()
    set_ticks(ax)
    ax.set_xlim(np.min(X), np.max(X))
    ax.set_ylim(np.min(Y), np.max(Y))
    ax.set_aspect('auto', adjustable='box')  # Maintain an appropriate aspect ratio
    plt.tight_layout()
    plt.show()
    return
```

Figure 53: Plot_porkchop function.

At Figure 54, the data and the contour levels are defined. This step is essential as if the levels aren’t correctly defined, the porkchop plot will be blank.

```python
#Function to assign the values, create the mesh, define the contour levels and select which information is shown
def make_porkchop_plot_from_solutions(plot_type='delv_plot'):
    y = unique_launch_dates
    x = unique_arrival_dates
    X, Y = np.meshgrid(x, y)

    # Variables for Z
    Z_c3 = results_C3
    Z_delv = results_delta_v
    Z_time_of_flight = results_time_of_flight

    # Define more contour levels
    min_c3 = np.nanmin(Z_c3)
    max_c3 = np.nanmax(Z_c3)
    min_delv = np.nanmin(Z_delv)
    max_delv = np.nanmax(Z_delv)


    #c3_levels = [1,2,3,4,5,6,7, 8, 10, 12, 14, 16, 20, 30, 50, 70, 90, 110, 130, 150]

    delv_levels = [4, 6, 8, 10, 12, 14, 16, 20, 30, 50, 70, 90, 110, 130, 150, 250, 300, 350, 500, 700, 900, 1000]

    if plot_type == 'delv_plot':
        title = 'Porkchop plot (ΔV Total)'
        plot_porkchop(title, X, Y, Z_delv, delv_levels, Z_time_of_flight)
    elif plot_type == 'c3_plot':
        title = 'Porkchop plot (C3-characteristic energy)'
        plot_porkchop(title, X, Y, Z_c3, c3_levels, Z_time_of_flight)
    else:
        raise Exception("Error: plot types should be 'c3_plot' or 'delv_plot'")
    return
```

Figure 54: make_porkchop_plot function.

Finally, all the data is plotted using the porkchop plot. Figure 55. One example of how the porkchop plot is, is shown in Figure 56.

```python
#Use the functions to create the plot
if __name__ == "__main__":

    # Prompt the user to choose the type of plot to display
    plot_type = input("Enter the type of plot to display ('c3_plot' or 'delv_plot'): ").strip()

    if plot_type not in ['c3_plot', 'delv_plot']:
        print("Invalid plot type. Please enter 'c3_plot' or 'delv_plot'.")
    else:
        make_porkchop_plot_from_solutions(plot_type)
```

Figure 55: Code to create the porkchop plot.

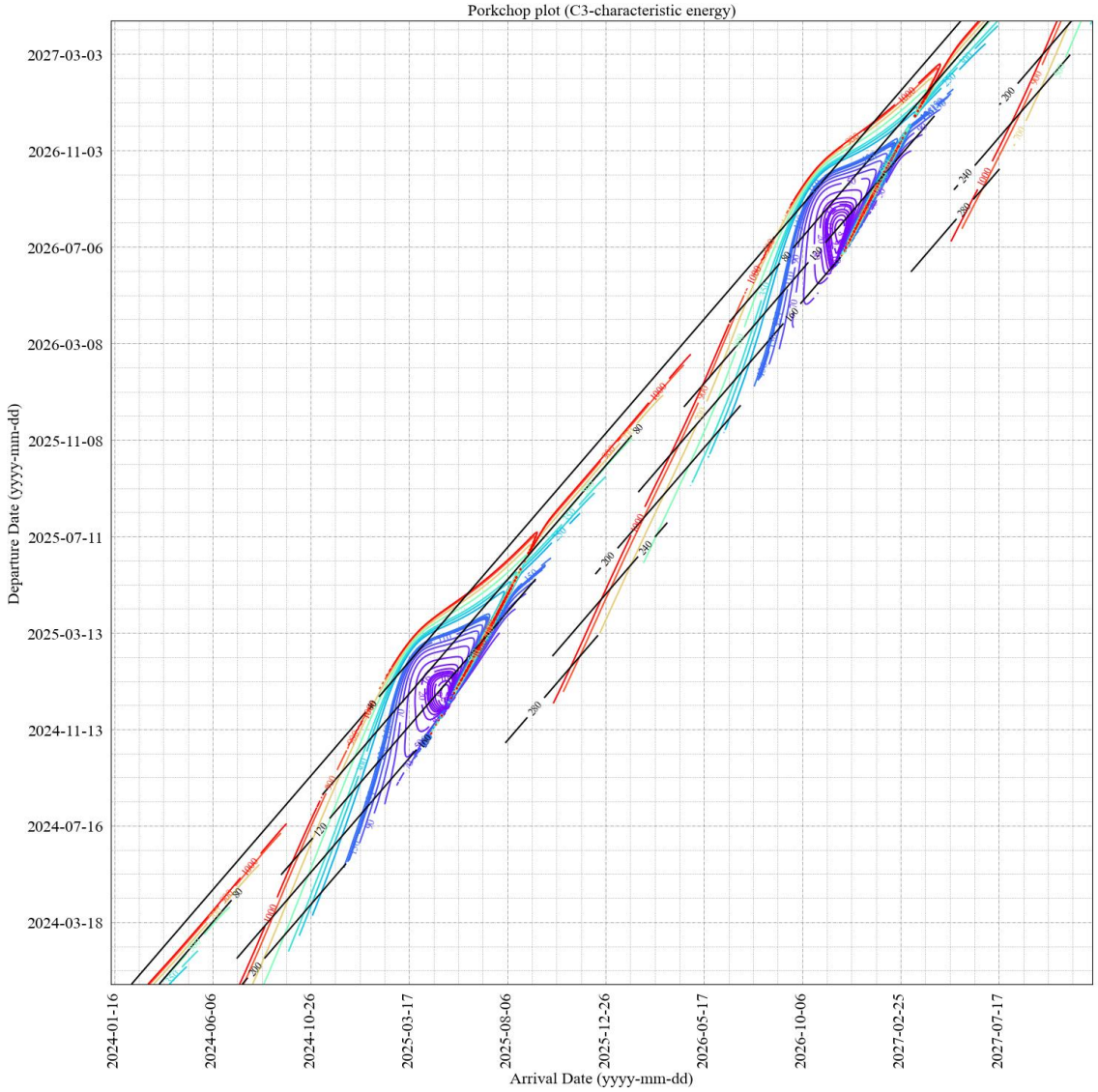

Figure 56: Example of a porkchop plot.

10. Analysis and Results

10.1. Generic calculations analysis and results

Following the process explained at 9.1 Generic calculations and using the respective Python code, the results are shown at Table 2 and Table 3.

Planet	Radius of the sphere of influence (km)	Synodic period (days)	Time of flight of a Hohmann transfer (years)	Orbit eccentricity
Earth	924725.796	369.656	16.009	0.0167
Uranus	51700059.692			0.0469

Table 2: General calculations about the transfer.

	Angular momentum method		Energy method		Error (%)
	Circular (km/s)	Ellipse (km/s)	Circular (km/s)	Ellipse (km/s)	
Delta-V at perigee	11.278	11.186	11.278	11.186	-0.823
Delta-V at apogee	4.660	4.459	4.660	4.459	-4.522
Total delta-V	15.939	15.645	15.939	15.645	-1.877

Table 3: Simplified results of the delta-V. Hohmann transfer.

Thus, analyzing the results, it has been proved it is true that both the energy approximation and the angular momentum one provides the same results. In addition to this, it is interesting to mention that just a slight difference on the eccentricity of the orbits can change the necessary delta-V to perform the maneuvers.

On the other side, now a better starting point can be achieved when the optimization process is being developed. Knowing the order of magnitude of the theoretical Hohmann transfer, which is the most optimal solution, can help to realize any mistakes while programming and, in fact, that was a scenario which happened while working on the project. Besides, the time of flight of about 16 years can help to accurately select the starting and arrival dates when optimizing.

Moreover, it is interesting to analyze the plot of the relative distance between Earth and Uranus vs time, Figure 57. Obviously, the distance between both planets is decreasing each year, however there is a moment every year when the distance tends to increase, and this event seems to be periodic. The main reason for the periodicity of this event is the Earth moving away from Uranus when its orbiting around the Sun, as it moves faster than Uranus. Furthermore, the time between each one of those peaks is the synodic period, which is 369.656 from Table 2, approximately one year as shown in the plot.

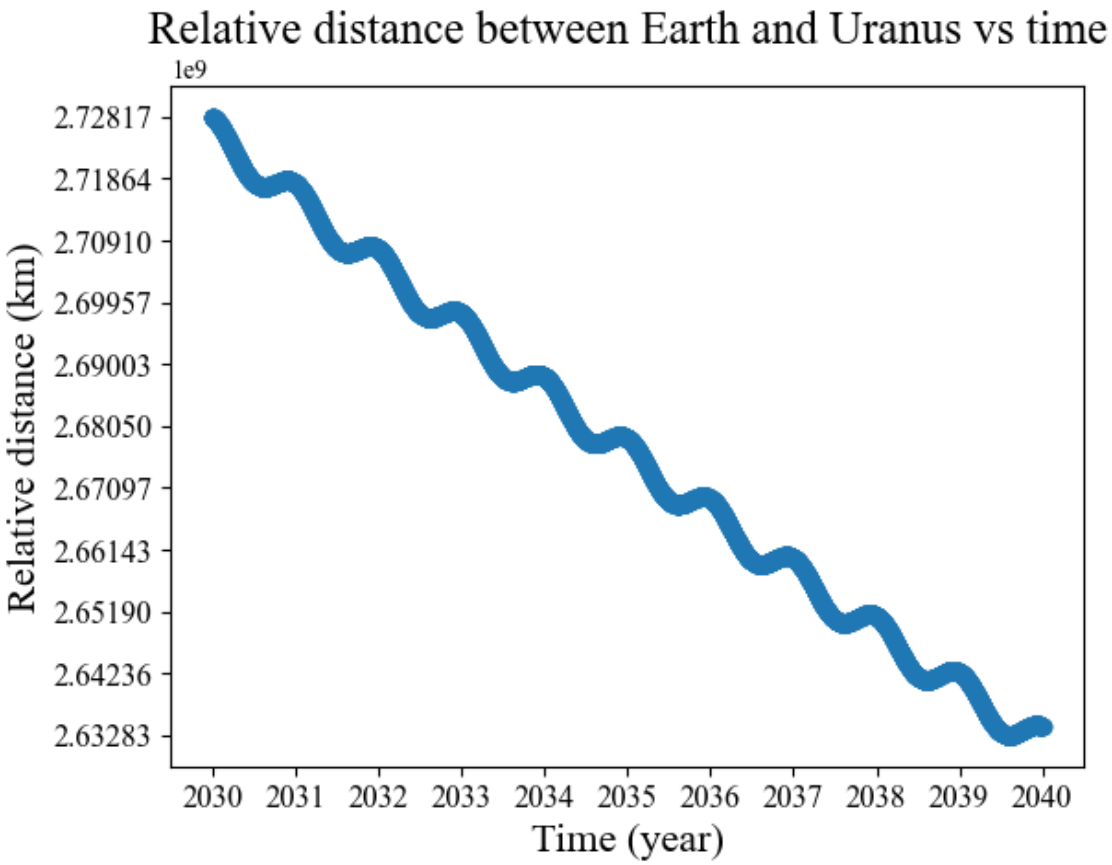

Figure 57: Relative distance vs Time.

10.2. Optimization process analysis and results

When working on the optimization process, it is important not only to understand the plots and deduce the results from them, but also to select which information is necessary to calculate and which plot is the best to show.

As the launch window for this study is between 2030 and 2040, maybe the first thought is to import the data between the first of January of those years. However, this approach is flawed, as the data for Uranus at the arrival is also needed. Therefore, being the time of flight of the simplified Hohmann transfer obtained at 10.1 Generic calculations analysis and results around 16 years, we are going to import from Astroquery data from 2030 to 2070. Selecting 2070 as the final year is because if the transfer starts at the end of the launch window, 2040, and takes around the double of the estimated time of flight of 16 years, it will finish the maneuver around 2070. Thus, we can be sure that the optimal maneuver will be included within this time frame.

Once the initial study period is selected, it is necessary to set a "step". For this first approximation, with a span of 40 years between the start and end dates, a step of 31 days is chosen. This interval is sufficient to have several measurements each year and to represent the data in the porkchop plot. The parameters are:

- Start_date = 2030-01-01
- End_date = 2070-01-01
- Step = 31d

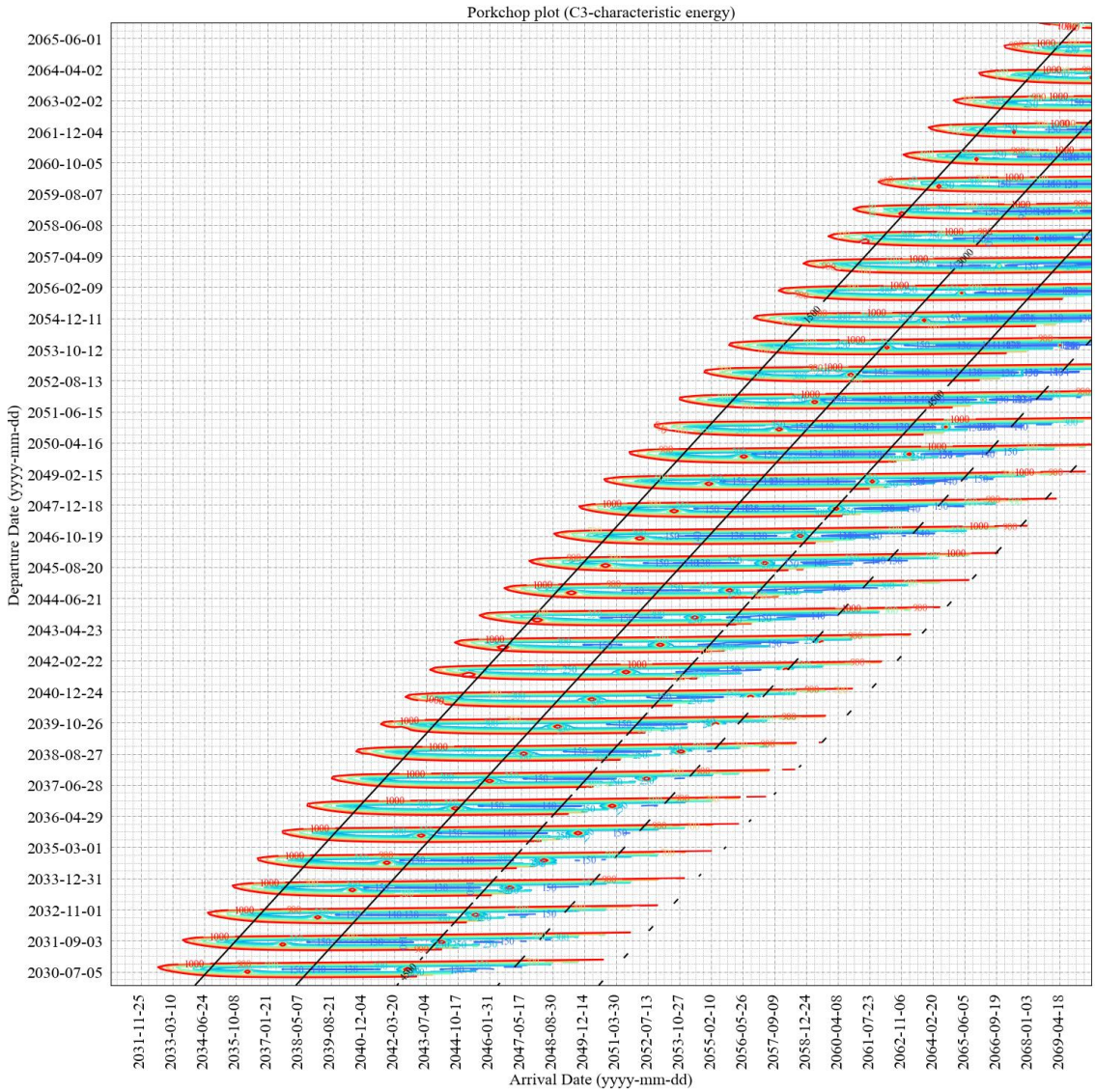

Figure 58: Porkchop plot. 2030 to 2070. Step = 31 d

Nevertheless, as shown in Figure 58, it is difficult to analyze the results because the figures are too small. Furthermore, since we haven't set a limit on the number of starting dates to calculate, dates outside the launch window have also been considered as launch dates. This can be resolved by setting a value for the variable "starting_dates." Therefore, in Figure 59, the launch dates are limited to the launch window, while the arrival date range remains between 2030 and 2070.

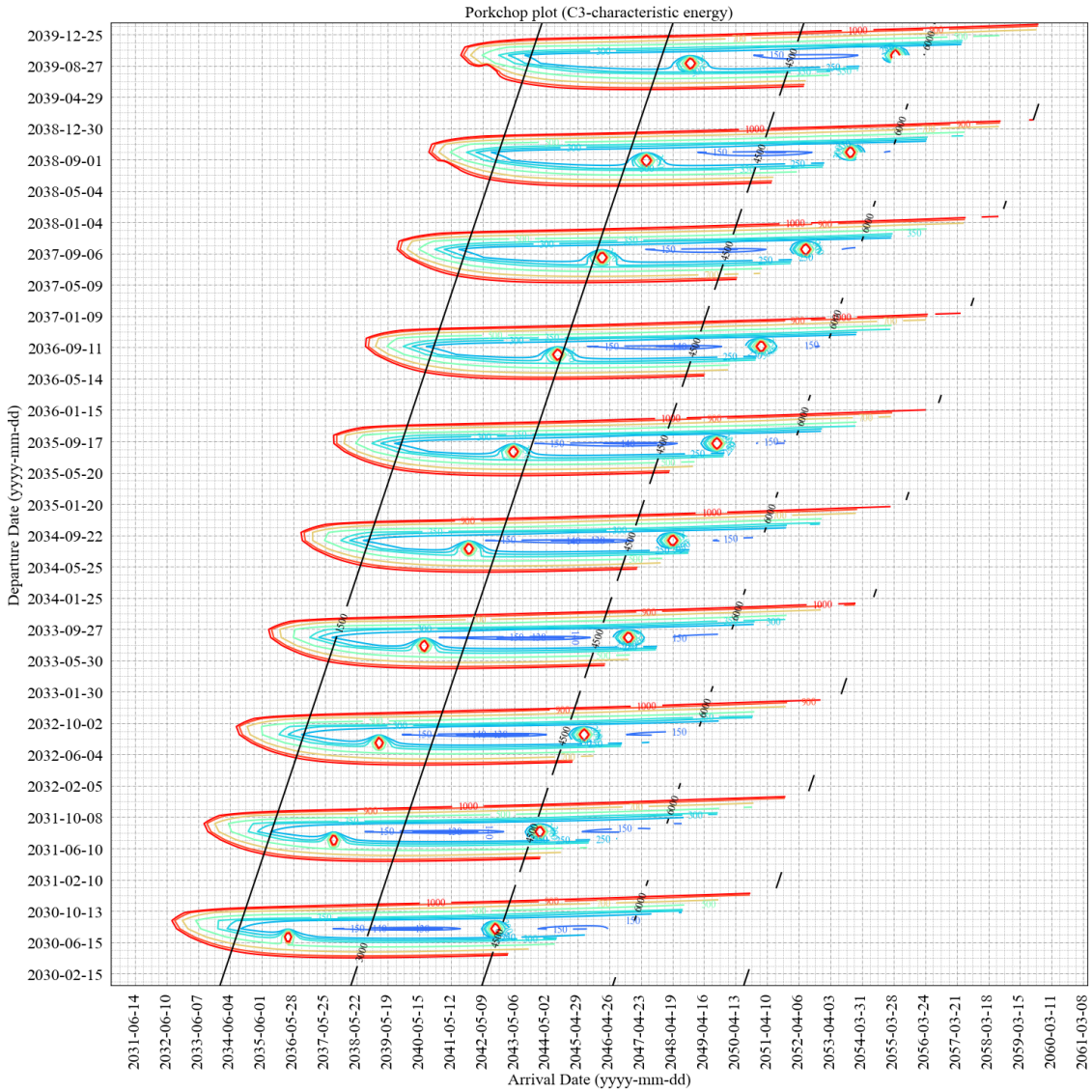

Figure 59: Porkchop plot. 2030-2070. Step = 31. Launch from 2030 to 2040.

Analyzing Figure 59, a pattern that repeats approximately every year when the maneuver starts can be observed. The main reason for this periodicity is likely due to the movement of Earth and Uranus around the Sun. According to Kepler's second law, planets closer to the Sun move faster, causing Earth to orbit the Sun more rapidly than Uranus. Therefore, while Earth completes multiple orbits, Uranus moves relatively little. This difference also explains the slight rightward displacement observed in the arrival dates; the longer time of flight results from Uranus being progressively farther from Earth each year. In addition to this, it is interesting to compare these results with the one obtained in Figure 57, where the same pattern occurred and the time between the peaks was the synodic period. [22]

On the other hand, as it has been mentioned at *7.6* Characteristic energy, *C3*, the optimal solutions can be easily identified using a porkchop plot, as they are the smallest "islands" located within larger ones. In this case, the optimal solutions are represented by the blue lines, indicating delta-V values around 150 km^2/s^2.

Furthermore, although some analysis can be conducted from the previous plot, its precision is limited by the 31-day step between delta-V calculations. To improve accuracy, reducing the step size to 5 days, for

[22] Kepler's law (NASA) - last access on July 6, 2024

example, could be considered. However, this adjustment would increase computational effort, as it would involve six times the number of dates compared to the previous plot. To manage this issue, focus can be directed towards the initial large "island" in the plot. This approach ensures that the optimal solution is identified promptly, which is crucial for securing funding for the mission. Companies are more likely to invest in missions that can commence earlier, making early optimal solutions critical.

The parameters selected for the new porkchop plot, Figure 60, are:

- Start_date = 2030-01-01
- End_date = 2050-01-01
- Step = 5d

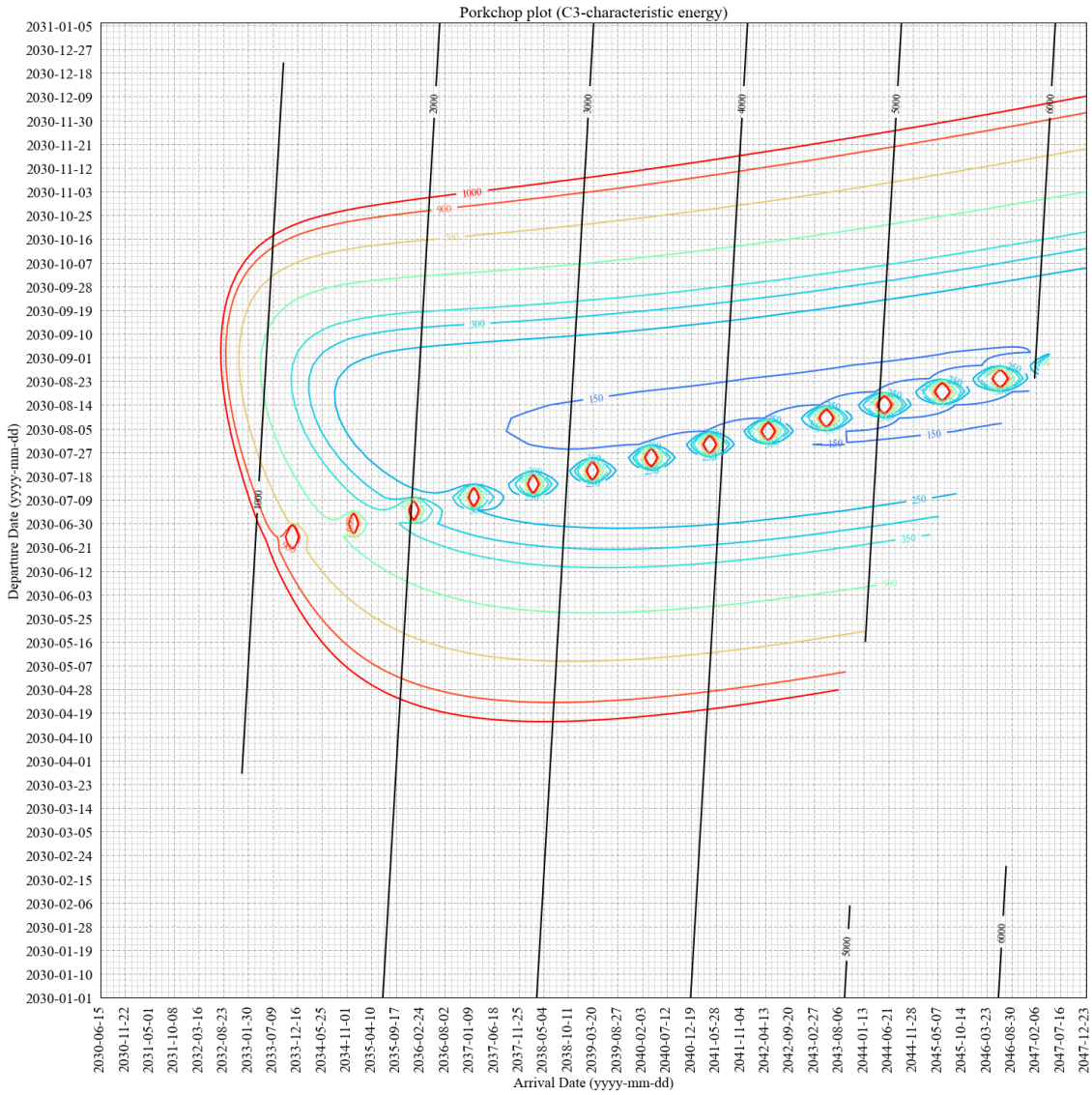

Figure 60: Porkchop plot. 2030-2050. Step = 5d. Launch from 2030 to 2031.

Analyzing Figure 60, an optimal solution "island" for the transfer can be observed, starting approximately between October and September 2032 and concluding between April 2039 and July 2048. However, there are smaller "islands" intersecting this solution. Regarding these islands, it appears they are inverse in nature, as the highest *C3* value is found within them. Based on the data from Figure 60, the exact reason for this phenomenon is uncertain, but it could potentially be attributed to a mathematical singularity, such

as a value divided by a number close to zero. Further investigation would be necessary to explore this aspect, although it falls outside the scope of the current project.

Returning to the optimal solution, with the significant reduction in the number of dates considered, the next step involves pinpointing the exact dates with the minimum delta-V using numerical results. Therefore, by running the code and printing the delta-V values, the minimum delta-V values, launch dates, and arrival dates are presented in Table 4.

Departure Date (YYYY-MM-DD)	Arrival date (YYYY-MM-DD)	Delta-V (km/s)	Time of flight (days)	Time of flight (years)
2030-08-14	2046-07-01	16.750	5800	15.880
	2046-07-06		5805	15.893
	2046-07-11		5810	15.907
	2046-07-16		5815	15.921
	2046-07-21		5820	15.934
	2046-07-26		5825	15.948
	2046-07-31		5830	15.962
	2046-08-05		5835	15.975
	2046-08-10		5840	15.989
	2046-08-15		5845	16.003
	2046-08-20		5850	16.016
	2046-08-25		5855	16.030
	2046-08-30		5860	16.044
	2046-09-04		5865	16.057
	2046-09-09		5870	16.071
	2046-09-14		5875	16.085
	2046-09-15		5880	16.099
	2046-09-20		5885	16.112
	2046-09-25		5890	16.126
	2046-09-30		5895	16.140
	2046-10-04		5900	16.153

Table 4: Optimal delta-V solutions.

Once the optimal solutions have been obtained and comparing them with the fast solutions calculated with the simplified method from Table 3, it can be said that the results are similar. While the optimal solution with real data from Astroquery calculates an optimal transfer with a time of flight between 15.880 and 16.153 years, the time of flight for the simplified Hohmann transfer was 16.009 years, which is a close and intermediate value. Furthermore, the delta-V is also at the same order of magnitude, being 15.645 km/s for the simplified solution and 16.750 km/s for the optimal one, although this last one is slightly higher.

Comparing the obtained delta-V with that of the New Horizons mission, it may not be feasible nowadays to design a satellite with sufficient delta-V to meet these requirements. Hence, alternative methods to power the spacecraft and achieve the necessary delta-V, such as utilizing gravitational assists from other planets via fly-bys, should be considered. Nevertheless, this aspect also lies beyond the scope of the current project and should be explored in future research. [23]

Therefore, selecting the transfer starting the 14th of August 2030 and arrival the 1st of July 2046 from Table 4 as it is one of the optimal solutions, it is possible to represent the trajectory of the satellite from Earth to Uranus at Figure 61.

[23] New Horizons - last access on July 6, 2024

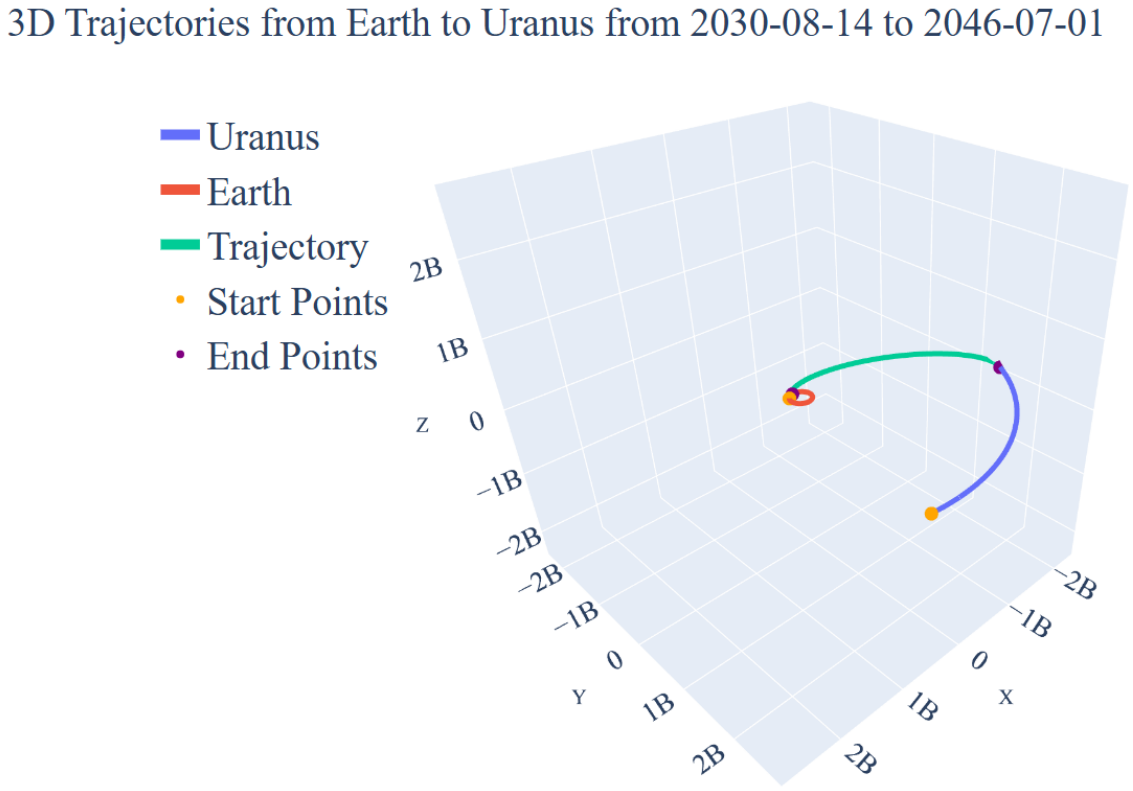

Figure 61: Optimal trajectory from Earth to Uranus.

Lastly, when the transfer maneuver has been selected, the portion from the circular parking orbit around Earth to the heliocentric maneuver is known. Therefore, some of the characteristics of this launch segment is e = 3.321, β = 1.266 rad and Δv= 8.425 km/s.

11. Conclusions and future work

Finally, throughout this project, an optimization process implemented in Python to find the optimal trajectory between two planets has been carried out. Specifically, optimizing a transfer from Earth to Uranus with a lunch window from 2030 to 2040 has been achieved.

Regarding the obtained results, it is essential to mention that there exists a periodic sequence of delta-V when talking about transfers between Earth and Uranus. However, the earliest optimal solution has been chosen due to the intention to attract interest and, therefore, funds from companies. Furthermore, two future works related to the topic have been proposed. The first involves a possible fly-by maneuver during the transfer between Earth and Uranus with the objective of reducing the necessary delta-V for the transfer maneuver. The second involves researching the presence of some inverse "islands" in the porkchop plot, with one hypothesis being a mathematical issue, such as a number which tends to zero

In conclusion, the selected departure and arrival date for the transfer are 14th of August 2030 and arrival the 1st of July 2046 respectively, with a time of flight of 5800 days and a necessary delta-V of the heliocentric portion of 16.750 km/s. Moreover, the necessary velocity to departure from a 200 km circular orbit around Earth to the optimal transfer trajectory is Δv= 8.425 km/s.

12. Scope statement

12.1. Objective

Output is not the only aspect to consider while working, the safety and health of the workers are also essential sides to think about. Therefore, the objective of this section is to address current regulations to guarantee the best conditions for the workers, at the same time as productivity is raised. Moreover, if these requirements aren't fulfilled, not only the health and privacy of workers are at risk, but as one of the links of the chain would be broken, a bigger project, which needs this one, will be stopped and maybe cancelled.

12.2. Regulation

About regulation presented through this section, it is necessary to mention that, with the intention to preserve their full meaning, it will be presented in Spanish, the original language in which they published in "*El Boletín Oficial del Estado*" which is the official paper where the Spanish government publishes new laws and regulations.

On the other side, there are two laws that, because of the nature of the project, have a high influence on the workplace and workers:

- Real Decreto 488/1997, de 14 de abril, sobre disposiciones mínimas de seguridad y salud relativas al trabajo con equipos que incluyen pantallas de visualización.
- Ley 10/2021, de 9 de julio, de trabajo a distancia.

The first law includes regulations about security and health when using screens and the second one is about remote work. [24]

" Real Decreto 488/1997, de 14 de abril, sobre disposiciones mínimas de seguridad y salud relativas al trabajo con equipos que incluyen pantallas de visualización.

Disposiciones mínimas

Observación preliminar: las obligaciones que se establecen en el presente anexo se aplicarán para alcanzar los objetivos del presente Real Decreto en la medida en que, por una parte, los elementos considerados existan en el puesto de trabajo y, por otra, las exigencias o características intrínsecas de la tarea no se opongan a ello.

En la aplicación de lo dispuesto en el presente anexo se tendrán en cuenta, en su caso, los métodos o criterios a que se refiere el apartado 3 del artículo 5 del Real Decreto de los Servicios de Prevención.

1. Equipo

a) Observación general. La utilización en sí misma del equipo no debe ser una fuente de riesgo para los trabajadores.

b) Pantalla.

Los caracteres de la pantalla deberán estar bien definidos y configurados de forma clara, y tener una dimensión suficiente, disponiendo de un espacio adecuado entre los caracteres y los renglones.

La imagen de la pantalla deberá ser estable, sin fenómenos de destellos, centelleos u otras formas de inestabilidad.

[24] Real Decreto 488/1997 - last access on June 10, 2024

El usuario de terminales con pantalla deberá poder ajustar fácilmente la luminosidad y el contraste entre los caracteres y el fondo de la pantalla, y adaptarlos fácilmente a las condiciones del entorno.

La pantalla deberá ser orientable e inclinable a voluntad, con facilidad para adaptarse a las necesidades del usuario.

Podrá utilizarse un pedestal independiente o una mesa regulable para la pantalla.

La pantalla no deberá tener reflejos ni reverberaciones que puedan molestar al usuario.

c) Teclado.

El teclado deberá ser inclinable e independiente de la pantalla para permitir que el trabajador adopte una postura cómoda que no provoque cansancio en los brazos o las manos.

Tendrá que haber espacio suficiente delante del teclado para que el usuario pueda apoyar los brazos y las manos.

La superficie del teclado deberá ser mate para evitar los reflejos.

La disposición del teclado y las características de las teclas deberán tender a facilitar su utilización.

Los símbolos de las teclas deberán resaltar suficientemente y ser legibles desde la posición normal de trabajo.

d) Mesa o superficie de trabajo.

La mesa o superficie de trabajo deberán ser poco reflectantes, tener dimensiones suficientes y permitir una colocación flexible de la pantalla, del teclado, de los documentos y del material accesorio.

El soporte de los documentos deberá ser estable y regulable y estará colocado de tal modo que se reduzcan al mínimo los movimientos incómodos de la cabeza y los ojos.

El espacio deberá ser suficiente para permitir a los trabajadores una posición cómoda.

e) Asiento de trabajo.

El asiento de trabajo deberá ser estable, proporcionando al usuario libertad de movimiento y procurándole una postura confortable.

La altura del mismo deberá ser regulable.

El respaldo deberá ser reclinable y su altura ajustable.

Se pondrá un reposapiés a disposición de quienes lo deseen.

2. Entorno

a) Espacio.

El puesto de trabajo deberá tener una dimensión suficiente y estar acondicionado de tal manera que haya espacio suficiente para permitir los cambios de postura y movimientos de trabajo.

b) Iluminación.

La iluminación general y la iluminación especial (lámparas de trabajo), cuando sea necesaria, deberán garantizar unos niveles adecuados de iluminación y unas relaciones adecuadas de luminancias entre la pantalla y su entorno, habida cuenta del carácter del trabajo, de las necesidades visuales del usuario y del tipo de pantalla utilizado.

El acondicionamiento del lugar de trabajo y del puesto de trabajo, así como la situación y las características técnicas de las fuentes de luz artificial, deberán coordinarse de tal manera que se eviten los deslumbramientos y los reflejos molestos en la pantalla u otras partes del equipo.

c) Reflejos y deslumbramientos.

Los puestos de trabajo deberán instalarse de tal forma que las fuentes de luz, tales como ventanas y otras aberturas, los tabiques transparentes o translúcidos y los equipos o tabiques de color claro no provoquen deslumbramiento directo ni produzcan reflejos molestos en la pantalla.

Las ventanas deberán ir equipadas con un dispositivo de cobertura adecuado y regulable para atenuar la luz del día que ilumine el puesto de trabajo.

d) Ruido.

El ruido producido por los equipos instalados en el puesto de trabajo deberá tenerse en cuenta al diseñar el mismo, en especial para que no se perturbe la atención ni la palabra.

e) Calor.

Los equipos instalados en el puesto de trabajo no deberán producir un calor adicional que pueda ocasionar molestias a los trabajadores.

f) Emisiones.

Toda radiación, excepción hecha de la parte visible del espectro electromagnético, deberá reducirse a niveles insignificantes desde el punto de vista de la protección de la seguridad y de la salud de los trabajadores.

g) Humedad.

Deberá crearse y mantenerse una humedad aceptable.

3. Interconexión ordenador/persona

Para la elaboración, la elección, la compra y la modificación de programas, así como para la definición de las tareas que requieran pantallas de visualización, el empresario tendrá en cuenta los siguientes factores:

a) El programa habrá de estar adaptado a la tarea que deba realizarse.

b) El programa habrá de ser fácil de utilizar y deberá, en su caso, poder adaptarse al nivel de conocimientos y de experiencia del usuario; no deberá utilizarse ningún dispositivo cuantitativo o cualitativo de control sin que los trabajadores hayan sido informados y previa consulta con sus representantes.

c) Los sistemas deberán proporcionar a los trabajadores indicaciones sobre su desarrollo.

d) Los sistemas deberán mostrar la información en un formato y a un ritmo adaptados a los operadores.

e) Los principios de ergonomía deberán aplicarse en particular al tratamiento de la información por parte de la persona. "

[25]

"Ley 10/2021, de 9 de julio, de trabajo a distancia.

Artículo 1. Ámbito de aplicación.

Las relaciones de trabajo a las que resultará de aplicación la presente Ley serán aquellas en las que concurran las condiciones descritas en el artículo 1.1 del texto refundido de la Ley del Estatuto de los Trabajadores aprobado por el Real Decreto Legislativo 2/2015, de 23 de octubre, que se desarrollen a distancia con carácter regular.

Se entenderá que es regular el trabajo a distancia que se preste, en un periodo de referencia de tres meses, un mínimo del treinta por ciento de la jornada, o el porcentaje proporcional equivalente en función de la duración del contrato de trabajo.

Artículo 2. Definiciones.

A los efectos de lo establecido en esta Ley, se entenderá por:

a) «Trabajo a distancia»: forma de organización del trabajo o de realización de la actividad laboral conforme a la cual esta se presta en el domicilio de la persona trabajadora o en el lugar elegido por esta, durante toda su jornada o parte de ella, con carácter regular.

b) «Teletrabajo»: aquel trabajo a distancia que se lleva a cabo mediante el uso exclusivo o prevalente de medios y sistemas informáticos, telemáticos y de telecomunicación.

c) «Trabajo presencial»: aquel trabajo que se presta en el centro de trabajo o en el lugar determinado por la empresa.

Artículo 6. Obligaciones formales del acuerdo de trabajo a distancia.

1. El acuerdo de trabajo a distancia deberá realizarse por escrito. Este acuerdo podrá estar incorporado al contrato de trabajo inicial o realizarse en un momento posterior, pero en todo caso deberá formalizarse antes de que se inicie el trabajo a distancia.

2. La empresa deberá entregar a la representación legal de las personas trabajadoras una copia de todos los acuerdos de trabajo a distancia que se realicen y de sus actualizaciones,

[25] Ley 10/2021 - last access on June 10, 2024

excluyendo aquellos datos que, de acuerdo con la Ley Orgánica 1/1982, de 5 de mayo, de protección civil del derecho al honor, a la intimidad personal y familiar y a la propia imagen, pudieran afectar a la intimidad personal, de conformidad con lo previsto en el artículo 8.4 del Estatuto de los Trabajadores. El tratamiento de la información facilitada estará sometido a los principios y garantías previstos en la normativa aplicable en materia de protección de datos.

Esta copia se entregará por la empresa, en un plazo no superior a diez días desde su formalización, a la representación legal de las personas trabajadoras, que la firmarán a efectos de acreditar que se ha producido la entrega.

Posteriormente, dicha copia se enviará a la oficina de empleo. Cuando no exista representación legal de las personas trabajadoras también deberá formalizarse copia básica y remitirse a la oficina de empleo.

Artículo 7. Contenido del acuerdo de trabajo a distancia.

Será contenido mínimo obligatorio del acuerdo de trabajo a distancia, sin perjuicio de la regulación recogida al respecto en los convenios o acuerdos colectivos, el siguiente:

a) Inventario de los medios, equipos y herramientas que exige el desarrollo del trabajo a distancia concertado, incluidos los consumibles y los elementos muebles, así como de la vida útil o periodo máximo para la renovación de estos.

b) Enumeración de los gastos que pudiera tener la persona trabajadora por el hecho de prestar servicios a distancia, así como forma de cuantificación de la compensación que obligatoriamente debe abonar la empresa y momento y forma para realizar la misma, que se corresponderá, de existir, con la previsión recogida en el convenio o acuerdo colectivo de aplicación.

c) Horario de trabajo de la persona trabajadora y dentro de él, en su caso, reglas de disponibilidad.

d) Porcentaje y distribución entre trabajo presencial y trabajo a distancia, en su caso.

e) Centro de trabajo de la empresa al que queda adscrita la persona trabajadora a distancia y donde, en su caso, desarrollará la parte de la jornada de trabajo presencial.

f) Lugar de trabajo a distancia elegido por la persona trabajadora para el desarrollo del trabajo a distancia.

g) Duración de plazos de preaviso para el ejercicio de las situaciones de reversibilidad, en su caso.

h) Medios de control empresarial de la actividad.

i) Procedimiento a seguir en el caso de producirse dificultades técnicas que impidan el normal desarrollo del trabajo a distancia.

j) Instrucciones dictadas por la empresa, con la participación de la representación legal de las personas trabajadoras, en materia de protección de datos, específicamente aplicables en el trabajo a distancia.

k) Instrucciones dictadas por la empresa, previa información a la representación legal de las personas trabajadoras, sobre seguridad de la información, específicamente aplicables en el trabajo a distancia.

l) Duración del acuerdo de trabajo a distancia.

Artículo 9. Derecho a la formación.

1. Las empresas deberán adoptar las medidas necesarias para garantizar la participación efectiva en las acciones formativas de las personas que trabajan a distancia, en términos equivalentes a las de las personas que prestan servicios en el centro de trabajo de la empresa, debiendo atender el desarrollo de estas acciones, en lo posible, a las características de su prestación de servicios a distancia.

2. La empresa deberá garantizar a las personas que trabajan a distancia la formación necesaria para el adecuado desarrollo de su actividad tanto al momento de formalizar el acuerdo de trabajo a distancia como cuando se produzcan cambios en los medios o tecnologías utilizadas.

Artículo 11. Derecho a la dotación suficiente y mantenimiento de medios, equipos y herramientas.

1. Las personas que trabajan a distancia tendrán derecho a la dotación y mantenimiento adecuado por parte de la empresa de todos los medios, equipos y herramientas necesarios para el desarrollo de la actividad, de conformidad con el inventario incorporado en el acuerdo referido en el artículo 7 y con los términos establecidos, en su caso, en el convenio o acuerdo colectivo de aplicación. En el caso de personas con discapacidad trabajadoras, la empresa asegurará que esos medios, equipos y herramientas, incluidos los digitales, sean universalmente accesibles, para evitar cualquier exclusión por esta causa.

2. Asimismo, se garantizará la atención precisa en el caso de dificultades técnicas, especialmente en el caso de teletrabajo.

Artículo 13. Derecho al horario flexible en los términos del acuerdo.

De conformidad con los términos establecidos en el acuerdo de trabajo a distancia y la negociación colectiva, respetando los tiempos de disponibilidad obligatoria y la normativa sobre tiempo de trabajo y descanso, la persona que desarrolla trabajo a distancia podrá flexibilizar el horario de prestación de servicios establecido.

Artículo 14. Derecho al registro horario adecuado.

El sistema de registro horario que se regula en el artículo 34.9 del Estatuto de los Trabajadores, de conformidad con lo establecido en la negociación colectiva, deberá reflejar fielmente el tiempo que la persona trabajadora que realiza trabajo a distancia dedica a la actividad laboral, sin perjuicio de la flexibilidad horaria, y deberá incluir, entre otros, el momento de inicio y finalización de la jornada.

Artículo 17. Derecho a la intimidad y a la protección de datos.

1. La utilización de los medios telemáticos y el control de la prestación laboral mediante dispositivos automáticos garantizará adecuadamente el derecho a la intimidad y a la protección de datos, en los términos previstos en la Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales, de acuerdo con los principios de idoneidad, necesidad y proporcionalidad de los medios utilizados.

2. La empresa no podrá exigir la instalación de programas o aplicaciones en dispositivos propiedad de la persona trabajadora, ni la utilización de estos dispositivos en el desarrollo del trabajo a distancia.

3. Las empresas deberán establecer criterios de utilización de los dispositivos digitales respetando en todo caso los estándares mínimos de protección de su intimidad de acuerdo con los usos sociales y los derechos reconocidos legal y constitucionalmente. En su elaboración deberá participar la representación legal de las personas trabajadoras.

Los convenios o acuerdos colectivos podrán especificar los términos dentro de los cuales las personas trabajadoras pueden hacer uso por motivos personales de los equipos informáticos puestos a su disposición por parte de la empresa para el desarrollo del trabajo a distancia, teniendo en cuenta los usos sociales de dichos medios y las particularidades del trabajo a distancia.

Artículo 18. Derecho a la desconexión digital.

1. Las personas que trabajan a distancia, particularmente en teletrabajo, tienen derecho a la desconexión digital fuera de su horario de trabajo en los términos establecidos en el artículo 88 de la Ley Orgánica 3/2018, de 5 de diciembre.

El deber empresarial de garantizar la desconexión conlleva una limitación del uso de los medios tecnológicos de comunicación empresarial y de trabajo durante los periodos de descanso, así como el respeto a la duración máxima de la jornada y a cualesquiera límites y precauciones en materia de jornada que dispongan la normativa legal o convencional aplicables.

2. La empresa, previa audiencia de la representación legal de las personas trabajadoras, elaborará una política interna dirigida a personas trabajadoras, incluidas los que ocupen puestos directivos, en la que definirán las modalidades de ejercicio del derecho a la desconexión y las acciones de formación y de sensibilización del personal sobre un uso razonable de las herramientas tecnológicas que evite el riesgo de fatiga informática. En particular, se preservará el derecho a la desconexión digital en los supuestos de realización total o parcial del trabajo a distancia, así como en el domicilio de la persona empleada vinculado al uso con fines laborales de herramientas tecnológicas.

Los convenios o acuerdos colectivos de trabajo podrán establecer los medios y medidas adecuadas para garantizar el ejercicio efectivo del derecho a la desconexión en el trabajo a distancia y la organización adecuada de la jornada de forma que sea compatible con la garantía de tiempos de descanso.

Artículo 20. Protección de datos y seguridad de la información.

1. Las personas trabajadoras, en el desarrollo del trabajo a distancia, deberán cumplir las instrucciones que haya establecido la empresa en el marco de la legislación sobre protección de datos, previa participación de la representación legal de las personas trabajadoras.

2. Las personas trabajadoras deberán cumplir las instrucciones sobre seguridad de la información específicamente fijadas por la empresa, previa información a su representación legal, en el ámbito del trabajo a distancia.

Artículo 22. Facultades de control empresarial.

La empresa podrá adoptar las medidas que estime más oportunas de vigilancia y control para verificar el cumplimiento por la persona trabajadora de sus obligaciones y deberes laborales, incluida la utilización de medios telemáticos, guardando en su adopción y aplicación la consideración debida a su dignidad y teniendo en cuenta, en su caso, sus circunstancias personales, como la concurrencia de una discapacidad."

12.3. Execution conditions

The conditions under which this project has been carried out meet all the requirements set out above. The requirements from “Real Decreto 488/1997” have been satisfied by working in a quiet and cozy room where temperature can be regulated by using air conditioning too. Moreover, the orientation of the room is appropriate to avoid glare caused by the sunlight and there is a comfortable armchair which height and back can be adjusted to be suitable for the desk. The software and the hardware used also meet the requirements and it are presented in Table 5 and Table 6.

Hardware	
Laptop	HP Pavilion Gaming Laptop 16-a0xxx
Operative system	Windows 11
Processor	Intel Core i7-10750H
RAM Memory	16GB
Graphic card	Nvidia GeForce GTX 1650 Ti

Table 5: Hardware used.

Software
Python
Microsoft Office 365

Table 6: Software used.

On the other hand, the law known as “Ley 10/2021” also has a significant impact on this project because nearly all of it has been accomplished as remote work. Some of the most relevant articles which have been met are about data protection which has been guaranteed as access to the laptop is limited by the worker. In addition to this, a proper schedule has been warranted by weekly meetings with the professor and assignments. Finally, the right of education has been fulfilled by having a professor to guide the project.

13. Time management

Managing the time wisely while working on a project is, at least, as crucial as the knowledge in the field of study since it is related with the ability of finishing the project on time. One of the multiples ways to organize this type of research is doing a Gantt Chart which divides the time span for the project into different sections which have to be completed on a specific time gap. This will also allow us to know if the project is being completed smoothly or if it is necessary to change any aspect to successfully finish it.

The Gantt Chart from Figure 62 shows how the academic year 2023-2024 has been divided into different tasks and the time it takes to complete each task.

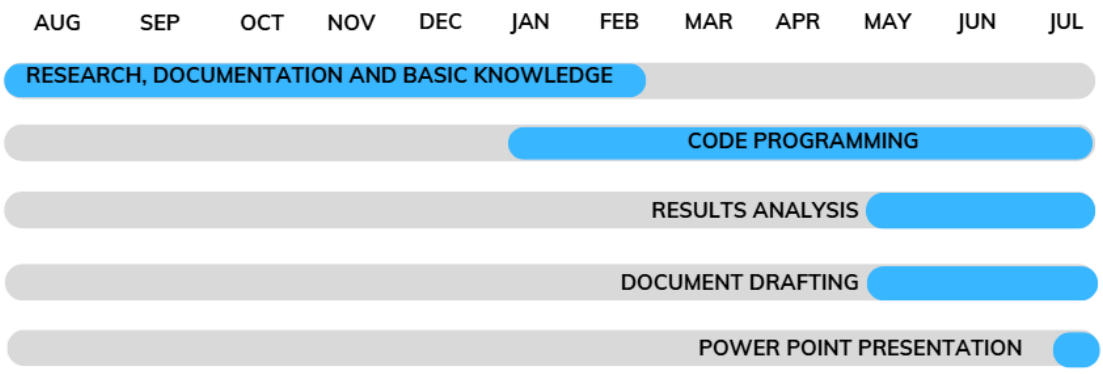

Figure 62: Gantt chart.

14. Budget

The budget is one of the main points when the viability of a project is being studied because it determines some important aspects such as the complexity of the mission or even if the goals can be achieved. Furthermore, not only does it affect the current project but also it can limit the future ones, which puts the budget in the spotlight. Thus, the purpose of this section is to address the necessary budget for the development of this project. This has been done using the method of costs by nature which means that the costs have been classified into categories. [26]

14.1. Labor costs

This section gathers the costs related to the time spent by different people on the project such as the student or the tutors of the project, Table 7. Obviously, the price per hour is different because it depends on the experience and the position, being 9.010 €/h for an undergraduate aerospace engineer and 46.153 €/h for a doctor. The cost per hour at these positions was given in dollars but it has been converted to euros with the conversion rates of the 8th of July 2024.

Labor costs			
	Cost per hour (€/h)	Time (h)	Subtotal (€)
Undergraduate aerospace engineer	9.010	300.000	2702.965
Doctor	46.153	48.000	2215.344
Total			4918.309

Table 7: Labor costs.

14.2. Hardware costs

This section gathers the costs of the hardware necessary to complete the project. The cost per hour has been obtained considering the maintenance and energy costs and the depreciation, *Appendix G: Hardware cost per hour*. Finally, hardware costs are shown at Table 8 [27] [28]

Hardware costs			
	Time (h)	Cost per hour (€/h)	Subtotal (€)
HP Pavilion laptop	350.000	0.117	41.051
Total			41.051

Table 8: Hardware costs.

It is important to comment that the number of hours of the hardware doesn't match the number of hours of the student because sometimes the program was running at night as it took around a day to finish all the calculations.

14.3. Software costs

This section gathers the costs of the software used for the project, such as licenses. It is relevant to mention that the price of the licenses is the price annually. Table 9.

Software costs				
	License cost (€)	Time (h)	Cost per hour (€/h)	Subtotal (€)
Windows 11	145.000	350.000	0.414	145.000
Microsoft Office 365	99.000	350.000	0.283	99.000
Python	0.000	350.000	0.000	0.000
Total				244.000

Table 9: Software costs.

[26] Costs by nature - last access on July 6, 2024
[27] Depreciation - last Access on June 20, 2024
[28] HP laptops - last Access on June 20, 2024

14.4. Total costs

Finally, adding all the calculated costs and considering 21% of taxes, the final budget of the project is 6296.066 € which is shown at Table 10.

Category	Subtotal (€)
Personnel	4918.309
Hardware	41.051
Software	244.000
Total before taxes	5203.360
Taxes (21%)	1092.706
Total after taxes	6296.066

Table 10: Total costs.

References

[32] Curtis, H. D., *Orbital Mechanics for Engineering Students*, Elsevier, 2020

[33] Richard C.Anderson, *Uranus orbiter and probe*, NASA, 2021

[34] Claudio Bombardelli, Juan Luis Gonzalo, and Javier Roa, *MULTIPLE REVOLUTION LAMBERT´S TARGETING PROBLEM: AN ANALYTICAL APPROXIMATION ,* Technical University of Madrid

[35] Andreas M. Hein1, Nikolaos Perakis1, Kelvin F. Long1, Adam Crowl1, Marshall Eubanks2, Robert G. Kennedy III1, Richard Osborne, *Project Lyra: Sending a Spacecraft to 1I/'Oumuamua (former A/2017 U1), the Interstellar Asteroid* ,Initiative for Interstellar Studies, 2017

Appendix A: Physical data

Table 11 is a summary of all the values used along the document with their units. This data can be obtained from the Curtis book and from the fact sheets from NASA. (Curtis, 2020) [29] [30]

	Earth	Uranus	Sun
Equatorial radius (km)	6378.136	25559.0	695700
Mass (kg)	5.9722e24	86.811e24	1.988e30
Semi-major axis (km)	149.598e6	2.867e9	-
Orbital period (days)	365.256	30685.157	-
Perigee radius (km)	147.095e6	2732.696e6	-
Apogee radius (km)	152.100e6	3001.390e6	-
Eccentricity	0.0469	0.0167	-
Mean orbital velocity (km/s)	29.780	6.790	-

Table 11: Physical data

[29] Earth fact sheet - last access on July 4, 2024
[30] Uranus fact sheet - last access on July 4, 2024

Appendix B: Python functions

B.1. *Sphere_influence*

The estimated sphere of influence of the planets can be easily calculated using Equation 1 from 6.5 Sphere of influence (SOI) . Thus, a Python function can be programmed, as shown in Figure 63

```
#This function is used to calculate the sphere of influence of a planet
def sphere_influence (a_planet,mass_planet,mass_central_body): #Define the function and the data to use
    #a_planet: semi-major axis of the planet
    #mass_planet: mass of the planet
    #mass_central_body: mass of the central body
    r_SOI = a_planet*(mass_planet/mass_central_body)**(2/5) #Equation of the SOI
    return r_SOI
```

Figure 63: Sphere of influence code.

B.2. *Synodic_period*

The aim of this function is to get the synodic period using Equation 13 from 7.3 Interplanetary Hohmann transfer. Therefore, the code used to get it is shown in Figure 64

```
#This function is used to calculate the synodic period
import numpy as np
def synodic_period (T_planet_1,T_planet_2): #Define the function and the data to use
    #T_planet_1: Period of the orbit (first planet)
    #T_planet_2: Period of the orbit (second planet)
    Tsyn = T_planet_1*T_planet_2/np.abs(T_planet_1-T_planet_2)  #Equation of the synodic period
    return Tsyn
```

Figure 64: Synodic period code.

B.3. *Ellipse_period*

This function calculates the period of an elliptical orbit, using Equation 18, as shown in Figure 65. (Curtis, 2020)

$$T_{ellipse} = 2\frac{\pi}{\sqrt{\mu_{Sun}}}\left(\frac{R_1 + R_2}{2}\right)^{3/2}$$

Equation 18: Period of the ellipse.

```
#This function is used to calculate the period of an elliptical orbit
import numpy as np
def ellipse_period (a_planet_1,a_planet_2,mu_center_body): #Define the function and the data to use
    #a_planet_1:semi-major axis of the departure planet
    #a_planet_2: semi-major axis of the arrival planet
    #mu_center_body: mu of the central body
    T_orbit = 2*np.pi*((a_planet_1+a_planet_2)/2)**(3/2)/np.sqrt(mu_center_body) #calculate the period of the orbit
    return T_orbit
```

Figure 65: Ellipse period code.

This is useful because the time of flight of a Homann transfer is just half of the period of the ellipse. Therefore, dividing Equation 18 by 2, the time of flight of the Hohmann transfer can be obtained as shown in Equation 14 from 7.3 Interplanetary Hohmann transfer.

B.4. *deltaV_real_orbit_equation*

There are several ways to calculate the delta-V for a transfer between planets. One method involves using the fact that angular momentum is conserved at all instants of a specific orbit. Therefore, considering a transfer from the perigee of the departure planet to the apogee of the arrival planet, it is necessary to calculate the delta-V at those points: the perigee and the apogee of the transfer orbit.

To do this, Equation 19 and Equation 20 can be used. It is important to mention that the velocities can be obtained using Equation 21 and Equation 22 because at the perigee and at the apogee the flight path angle is zero, otherwise, Equation 23 should be used. (Curtis, 2020)

$$e = \frac{r_a - r_p}{r_a + r_p}$$

Equation 19: Eccentricity from perigee and apogee radius.

$$h = \sqrt{2\mu}\sqrt{\frac{r_a r_p}{r_a + r_p}}$$

Equation 20: Angular momentum.

$$v_a = \frac{h}{r_a}$$

Equation 21: Velocity at apogee.

$$v_p = \frac{h}{r_p}$$

Equation 22: Velocity at perigee.

$$v = \frac{h}{r\cos(\gamma)}$$

Equation 23: Velocity at any point of the orbit.

Finally, the total delta-V can be obtained by adding the absolute delta-V at perigee and apogee. Equation 24, Equation 25 and Equation 26.

$$\Delta v_p = v_p - v_{departure}$$

Equation 24: Delta-V at departure.

$$\Delta v_a = v_{arrival} - v_a$$

Equation 25: Delta-V at arrival.

$$\Delta v_{total} = |\Delta v_p| + |\Delta v_a|$$

Equation 26: Total delta-V.

This code following this process is shown at Figure 66.

```python
#This function is used to calculate the delta-V which should be applied to complete a transfer between two planets with real orbits.
#The angular momentum approach has been used to solve it. It can be solved with the Vis-viva equation too
import numpy as np
def deltaV_real_orbit_equation(mu,ra_departure,rp_departure,ra_arrival,rp_arrival): #Define the function and the data to use
    #mu: mu of the central body
    #ra_departure: distance from the central body to the apogee of the departure body
    #rp_departure:distance from the central body to the perigee of the departure body
    #ra_arrival:distance from the central body to the apogee of the arrival body
    #rp_arrival: distance from the central body to the perigee of the arrival body

    #Departure
    e_departure = (ra_departure-rp_departure)/(ra_departure+rp_departure) #eccentricity of the orbit (departure body)
    h_departure = np.sqrt(2*mu)*np.sqrt(ra_departure*rp_departure/(ra_departure+rp_departure)) #angular momentum of the orbit (departure body)
    v_departure = h_departure/rp_departure #velocity of the orbit of the orbit (departure body) #same as the fact sheet maximum velocity

    #Arrival
    e_arrival = (ra_arrival-rp_arrival)/(ra_arrival+rp_arrival) #eccentricity of the orbit (arrival body)
    h_arrival = np.sqrt(2*mu)*np.sqrt(ra_arrival*rp_arrival/(ra_arrival+rp_arrival)) #angular momentum of the orbit (arrival body)
    v_arrival = h_arrival/ra_arrival #velocity of the orbit of the orbit (arrival body)#same as the fact sheet minimum velocity

    #Transfer orbit
    h_transfer = np.sqrt(2*mu)*np.sqrt(ra_arrival*rp_departure/(ra_arrival+rp_departure)) #angular momentum of the orbit (transfer orbit)
    vp = h_transfer/rp_departure #perigee velocity of the orbit (transfer orbit)
    va = h_transfer/ra_arrival #apogee velocity of the orbit (transfer orbit)

    #Total Delta-V
    DeltaV_perigee = vp-v_departure #delta-V applied at the perigee (transfer orbit)
    DeltaV_apogee = v_arrival-va #delta-V applied at the apogee (transfer orbit)
    total = np.abs(DeltaV_perigee)+np.abs(DeltaV_apogee) #total delta-V (transfer orbit)
    return DeltaV_perigee,DeltaV_apogee,total,e_departure,e_arrival
```

Figure 66: deltaV_real_orbit_equation function.

B.5. *deltaV_real_orbit_equation_energy*

Another way to calculate the delta-V is using the Vis-Viva equation or the energy. The process is like the one from *B.4 deltaV_real_orbit_equation*, although the energy of the orbits can provide more information than the previous one. Equation 9 Equation 12 and Equation 19 will be necessary, and the total delta-V can be calculated in the same way as in *B.4 deltaV_real_orbit_equation*, using Equation 24, Equation 25 and Equation 26.

The code is shown in Figure 67.

```python
#This function is used to calculate the delta-V which should be applied to complete a transfer between two planets with real orbits.
#The energy has been used to solve it.
import numpy as np
def deltaV_real_orbit_equation_energy(mu,ra_departure,rp_departure,ra_arrival,rp_arrival): #Define the function and the data to use
    #mu: mu of the central body
    #ra_departure: distance from the central body to the apogee of the departure body
    #rp_departure:distance from the central body to the perigee of the departure body
    #ra_arrival:distance from the central body to the apogee of the arrival body
    #rp_arrival: distance from the central body to the perigee of the arrival body

    #Departure
    e_departure = (ra_departure-rp_departure)/(ra_departure+rp_departure) #eccentricity of the orbit (departure body)
    E_departure = -mu/(rp_departure+ra_departure)
    v_departure = np.sqrt(2*(E_departure+mu/rp_departure)) #velocity of the orbit of the orbit (departure body) #same as the fact sheet maximum velocity

    #Arrival
    e_arrival = (ra_arrival-rp_arrival)/(ra_arrival+rp_arrival) #eccentricity of the orbit (arrival body)
    E_arrival = -mu/(rp_arrival+ra_arrival)
    v_arrival = np.sqrt(2*(E_arrival+mu/ra_arrival))

    #Transfer orbit
    E = -mu/(rp_departure+ra_arrival) #energy of the orbit (transfer orbit)
    va = np.sqrt(2*(E+mu/ra_arrival)) #apogee velocity of the orbit (transfer orbit)
    vp = np.sqrt(2*(E+mu/rp_departure)) #perigee velocity of the orbit (transfer orbit)

    #Total Delta-V
    DeltaV_perigee = vp-v_departure #delta-V applied at the perigee (transfer orbit)
    DeltaV_apogee = v_arrival-va #delta-V applied at the apogee (transfer orbit)
    total = np.abs(DeltaV_perigee)+np.abs(DeltaV_apogee) #total delta-V (transfer orbit)
    return DeltaV_perigee,DeltaV_apogee,total,e_departure,e_arrival
```

Figure 67: deltaV_real_orbit_equation_energy function.

B.6. *error_X*

The aim of this function is to calculate the error between two values. Therefore, the code shown at Figure 68 calculates that error using Equation 27.

$$error\ (\%) = \left(\frac{|value1| - |value2|}{|value1|}\right)100$$

Equation 27: Error equation.

```python
#This function is used to calculate the error bewteen two values
import numpy as np
def error_X (X,Y): #Define the function and the data to use
    #X: value 1
    #Y: value 2
    error_X = ((np.abs(X)-np.abs(Y))/np.abs(X))*100  # Calculate the error in %
    return error_X
```

Figure 68: error_X function.

Appendix C: Comparison between calculating the delta-V using vectors or the flight path angle

The aim of this appendix is to compare the results between calculating the delta-V of a maneuver using the velocity vectors or using the flight path angle.

Obviously using vectors directly means that the delta-V is just the difference between both velocity vectors. Nevertheless, when talking about the flight path angle, the cosine law to obtain the relations between the parameters will be necessary. Finally, the resulting equations for impulsive maneuvers in coplanar orbits are Equation 28 and Equation 29. (Curtis, 2020)

$$\cos(\Delta\gamma) = \frac{\boldsymbol{v_1} \cdot \boldsymbol{v_2}}{v_1 v_2}$$

Equation 28: Flight path angle.

$$\Delta v = \sqrt{{v_1}^2 + {v_2}^2 - -2v_1 v_2 \cos{(\Delta\gamma)}}$$

Equation 29: Delta-V using flight path angle.

When using these equations at Figure 69, and the vectors as in *9.2* Optimization process, results can be compared. Finally, after comparing them, the difference is extremely small, as it is at the power of minus fourteen. Therefore, as it is easier to use and not limited to coplanar orbits, the vectors option is used through the optimization process.

```
cos_fpa1 = np.dot(v1_vec,v_vec_earth)/(v1*v_earth)
dv1 = np.sqrt(v1**2+v_earth**2-2*v1*v_earth*cos_fpa1)

print(f" dv1 = {dv1} km/s.")


cos_fpa2 = np.dot(v2_vec,v_vec_venus)/(v2*v_venus)
dv2 = np.sqrt(v2**2+v_venus**2-2*v2*v_venus*cos_fpa2)
print(f" dv2 = {dv2} km/s.")

dvtotal = dv1 + dv2
print(f"Total delta-v = {dvtotal}")
solution.append([delta_t_days, dvtotal])
print("\n")
```

Figure 69: Code to calculate the delta-V using the flight path angle.

Appendix D: Calculate trajectory

The aim of this appendix is to fully explain how the transfer trajectory can be calculated with the objective of plotting it.

The first step is to define the Stumpp functions as they will be necessary in further calculations. Figure 70

```
#############################################################################################
#                                    Calculate the trajectory
#############################################################################################

if plot_trajectory==1:
#.....................................
    #Define the functions to use
    def stumpC(z):
        if z > 0:
            return (1 - np.cos(np.sqrt(z))) / z
        elif z < 0:
            return (1 - np.cosh(np.sqrt(-z))) / z
        else:
            return 1 / 2

    def stumpS(z):
        if z > 0:
            return (np.sqrt(z) - np.sin(np.sqrt(z))) / (np.sqrt(z) ** 3)
        elif z < 0:
            return (np.sinh(np.sqrt(-z)) - np.sqrt(-z)) / (np.sqrt(-z) ** 3)
        else:
            return 1 / 6
```

Figure 70: Stumpp functions to calculate trajectory.

Stumpp functions can be used to define Kepler's equation. Figure 71.

```
    def kepler_U(dt, ro, vro, a, mu):
        error = 1.e-8
        nMax = 1000
        x = np.sqrt(mu) * abs(a) * dt

        n = 0
        ratio = 1
        while abs(ratio) > error and n <= nMax:
            n += 1
            z = a * x ** 2
            C = stumpC(z)
            S = stumpS(z)
            F = ro * vro / np.sqrt(mu) * x ** 2 * C + (1 - a * ro) * x ** 3 * S + ro * x - np.sqrt(mu) * dt
            dFdx = ro * vro / np.sqrt(mu) * x * (1 - a * x ** 2 * S) + (1 - a * ro) * x ** 2 * C + ro
            ratio = F / dFdx
            x -= ratio

        if n > nMax:
            print(f'\n **No. iterations of Kepler's equation = {n}')
            print(f'\n F/dFdx = {F / dFdx}\n')

        return x
```

Figure 71: Solve Kepler equation.

Once all the functions are defined, the input data can be obtained from the Lambert's problem solved previously and the function "oe_from_r_and_v". Furthermore, Kepler's equation can be solved. Figure 72.

```
# Input data

for i in range(0, len(sol_oe1)):
    if sol_oe1[i][0] == TOF_days:
        e = sol_oe1[i][8]
        theta0 = sol_oe1[i][10]
        theta = sol_oe2[i][10]
for i in range(0,len(dv1_vec_plot_array)):
    if dv1_vec_plot_array[i][0] == TOF_days:
        v0_vec = np.array([dv1_vec_plot_array [i][1],dv1_vec_plot_array [i][2],dv1_vec_plot_array [i][3]])

r0_vec = r_vec_earth #
ro = np.linalg.norm(r0_vec) #r0
vo = np.linalg.norm(v0_vec)
vro = np.dot(r0_vec, v0_vec)/ro #km/s
dt = TOF_days*24*3600 #delta_t
alpha = 2/ro-vo**2/mu_sun
a = 1/alpha


# Calculation of x using the kepler_U function
x = kepler_U(dt, ro, vro, alpha, mu_sun)
```

Figure 72: Input data to plot the trajectory.

Now, the position and velocity vectors are obtained using the Lagrange coefficients as it was done in9.2 Optimization process. Furthermore, it is necessary to calculate the eccentric anomaly at the first and last point of the transfer trajectory. Figure 73.

```
#Calculate the position and velocity vectors using f,g,fdot and gdot
chi0 = np.sqrt(mu_sun)*np.abs(alpha)*dt

z = alpha*x**2
C = stumpC(z)
S = stumpS(z)
#C = 1/2 - z/24 + z**2/720 - z**3/40320 + z**4/3628800 - z**5/479001600
#S = 1/6 - z/120 + z**2/5040 - z**3/362880  + z**4/39916800 - z**5/6227020800

f = 1-x**2*C/ro
g = dt-(x**3)*S/np.sqrt(mu_sun)
r_vec = f*r0_vec+g*v0_vec

r = np.linalg.norm(r_vec)
fdot = np.sqrt(mu_sun)*(alpha*x**3*S-x)/(r*ro)
gdot = 1-x**2*C/r
v_vec = fdot*r0_vec+gdot*v0_vec

#Calculate E at the first and last point the the transfer trayectory
E = 2*np.arctan(np.sqrt((1-e)/(1+e))*np.tan(theta/2))
E0 = 2*np.arctan(np.sqrt((1-e)/(1+e))*np.tan(theta0/2))
x = np.sqrt(1/alpha)*(E-E0)
```

Figure 73: Position and velocity vectors using Lagrange coefficients.

Finally, once all these data are known, the necessary variables can be defined, and the position vector of the transfer maneuver obtained. Figure 74.

```
#Create the variables
x_vec =  np.linspace(0,x,num=len(r_vec_venus_plot_array))
r_plot = np.array(np.zeros((len(x_vec),3)))
dt = np.array(np.zeros((len(x_vec),1)))
z = np.array(np.zeros((len(x_vec),1)))
C = np.array(np.zeros((len(x_vec),1)))
S = np.array(np.zeros((len(x_vec),1)))

#Calculate the position vectors of the transfer trayectory
for i in range(0,len(x_vec)):
    z[i] = alpha*x_vec[i]**2
    C[i] = stumpC(z[i])
    S[i] = stumpS(z[i])
    #C[i] = 1/2 - z[i]/24 + z[i]**2/720 - z[i]**3/40320 + z[i]**4/3628800 - z[i]**5/479001600
    #S[i] = 1/6 - z[i]/120 + z[i]**2/5040 - z[i]**3/362880  + z[i]**4/39916800 - z[i]**5/6227020800

for i in range(0,len(x_vec)):
    dt[i] = (ro*vro*x_vec[i]**2*C[i]/np.sqrt(mu_sun)+(1-alpha*ro)*x_vec[i]**3*S[i]+ro*x_vec[i])/np.sqrt(mu_sun)

for i in range(0,len(x_vec)):
    r_plot[i] = (1-x_vec[i]**2*C[i]/ro)*r0_vec+(dt[i]-x_vec[i]**3*S[i]/np.sqrt(mu_sun))*v0_vec
```

Figure 74: Calculate the position vectors of the trajectory.

When the trajectory has been calculated and knowing the orbits of Earth and Uranus around the Sun, they can be appended to a variable with the JD of each position vector. Furthermore, the departure and arrival date can be defined in the appropriate format to work with them. Figure 75.

```
##############################################################################################
#                                  Plot the trajectory
##############################################################################################
from datetime import datetime, timedelta
vec_plot = [] #Initialization of the variables
for i in range(0,len(vec_earth)):
    r_vec_earth_x = vec_earth['x'][i]*AU #Convert to interntional system units
    r_vec_earth_y = vec_earth['y'][i]*AU #Convert to interntional system units
    r_vec_earth_z = vec_earth['z'][i]*AU #Convert to interntional system units
    r_vec_venus_x = vec_venus['x'][i]*AU #Convert to international system units
    r_vec_venus_y = vec_venus['y'][i]*AU #Convert to international system units
    r_vec_venus_z = vec_venus['z'][i]*AU #Convert to international system units
    JD = vec_venus['datetime_str'][i] #Julian date
    vec_plot.append([JD,r_vec_earth_x,r_vec_earth_y,r_vec_earth_z,r_vec_venus_x,r_vec_venus_y,r_vec_venus_z]) #Append the results
vec_plot_array = np.array(vec_plot) #Convert the variable to array

# Data
data = vec_plot_array

# Set the launch date in datetime format and time of flight in timedelta format
launch_date = gregorian_launch_date
time_of_flight = timedelta(days=TOF_days)

# Calculate the arrival date
arrival_date = launch_date + time_of_flight

# Format of dates in data
date_format = "A.D. %Y-%b-%d %H:%M:%S.%f"
```

Figure 75: Obtain the data to plot the trajectory.

As the position vectors from the planets are not only defined during the duration of the transfer, it is necessary to extract those vectors that we need. Therefore, new variables with the position vectors during the departure and arrival date are defined. Figure 76.

```
# Variables to store the position vectors
departure_values = []
arrival_values = []

# Flag to start collecting data
collect_data = False

for row in data:
    # Convert date from string to datetime
    date = datetime.strptime(row[0], date_format)
    if date == launch_date:
        collect_data = True
    if collect_data:
        departure_values.append(row[1:4])
        arrival_values.append(row[4:7])
    if date == arrival_date:
        collect_data = False

# Convert lists to numpy arrays and then to float arrays
arrival_values_string = np.array(arrival_values)
departure_values_string = np.array(departure_values)
arrival_values_array = arrival_values_string.astype(float)
departure_values_array = departure_values_string.astype(float)
```

Figure 76: Convert the data to the correct format.

Finally, the trajectory can be shown if some parameters such as the axis limits are correctly defined. Figure 77, Figure 78 and Figure 79.

```
# Separate x, y, z components for each trajectory
x1, y1, z1 = arrival_values_array[:, 0], arrival_values_array[:, 1], arrival_values_array[:, 2]
x2, y2, z2 = departure_values_array[:, 0], departure_values_array[:, 1], departure_values_array[:, 2]
x3, y3, z3 = r_plot[:, 0], r_plot[:, 1], r_plot[:, 2]

# Find global minimum and maximum for axes and ensure the same scaling
min_val = min(np.min(x1), np.min(y1), np.min(z1),
              np.min(x2), np.min(y2), np.min(z2),
              np.min(x3), np.min(y3), np.min(z3))
max_val = max(np.max(x1), np.max(y1), np.max(z1),
              np.max(x2), np.max(y2), np.max(z2),
              np.max(x3), np.max(y3), np.max(z3))

# Create a 3D plot for the trajectories
fig = go.Figure()
```

Figure 77: Set the limits of the axis.

```python
# Add the first trajectory
fig.add_trace(go.Scatter3d(
    x=x1, y=y1, z=z1,
    mode='lines',
    name='Uranus',
    line=dict(width=10)
))

# Add the second trajectory
fig.add_trace(go.Scatter3d(
    x=x2, y=y2, z=z2,
    mode='lines',
    name='Earth',
    line=dict(width=10)
))

# Add the third trajectory
fig.add_trace(go.Scatter3d(
    x=x3, y=y3, z=z3,
    mode='lines',
    name='Trajectory',
    line=dict(width=10)
))
```

Figure 78: Define the trajectories to plot.

```python
# Add large markers for start points (same color for all starts)
fig.add_trace(go.Scatter3d(
    x=[x1[0], x2[0], x3[0]], y=[y1[0], y2[0], y3[0]], z=[z1[0], z2[0], z3[0]],
    mode='markers',
    name='Start Points',
    marker=dict(size=10, color='orange')
))

# Add large markers for end points (same color for all ends)
fig.add_trace(go.Scatter3d(
    x=[x1[-1], x2[-1], x3[-1]], y=[y1[-1], y2[-1], y3[-1]], z=[z1[-1], z2[-1], z3[-1]],
    mode='markers',
    name='End Points',
    marker=dict(size=10, color='purple')
))

# Update layout to add axis labels, title, and set equal axis ranges
fig.update_layout(
    title={'text': f'3D Trajectories from Earth to Uranus from {launch_date.strftime("%Y-%m-%d")} to {arrival_date.strftime("%Y-%m-%d")}',
           'font': {'size': 30 }
        },
    scene=dict(
        xaxis=dict(title='X', range=[min_val, max_val]),
        yaxis=dict(title='Y', range=[min_val, max_val]),
        zaxis=dict(title='Z', range=[min_val, max_val])
    ),
    legend=dict(x=0.05, y=1.0,font=dict(size=30)
)
)
# Show the plot
fig.show()
```

Figure 79: Edit visual aspects of the trajectories and plot them.

As it was done with the trajectory, an animation can be shown to clarify how the transfer between both planets will be. Therefore, in Figure 80, Figure 81 and Figure 82, it is shown how to program an animation using the data obtained before.

```
###############################################################################################
#                                    Trjectory animation
###############################################################################################
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation

x1, y1, z1 = arrival_values_array[:, 0], arrival_values_array[:, 1], arrival_values_array[:, 2]
x2, y2, z2 = departure_values_array[:, 0], departure_values_array[:, 1], departure_values_array[:, 2]
x3, y3, z3 = r_plot[:, 0], r_plot[:, 1], r_plot[:, 2]

# Find the maximum length
max_len = max(len(x1), len(x2), len(x3))

# Create a common time vector
common_time = np.linspace(0, 1, max_len)

# Interpolate the shorter trajectories
x1_interp = np.interp(common_time, np.linspace(0, 1, len(x1)), x1)
y1_interp = np.interp(common_time, np.linspace(0, 1, len(y1)), y1)
z1_interp = np.interp(common_time, np.linspace(0, 1, len(z1)), z1)

x2_interp = np.interp(common_time, np.linspace(0, 1, len(x2)), x2)
y2_interp = np.interp(common_time, np.linspace(0, 1, len(y2)), y2)
z2_interp = np.interp(common_time, np.linspace(0, 1, len(z2)), z2)

x3_interp = np.interp(common_time, np.linspace(0, 1, len(x3)), x3)
y3_interp = np.interp(common_time, np.linspace(0, 1, len(y3)), y3)
z3_interp = np.interp(common_time, np.linspace(0, 1, len(z3)), z3)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

Figure 80: Adjust the trajectories to have the same number of points.

```
# Concatenate all coordinates to calculate limits
all_x = np.concatenate((x1_interp, x2_interp, x3_interp))
all_y = np.concatenate((y1_interp, y2_interp, y3_interp))
all_z = np.concatenate((z1_interp, z2_interp, z3_interp))

# Set axis limits
ax.set_xlim(np.min(all_x), np.max(all_x))
ax.set_ylim(np.min(all_y), np.max(all_y))
ax.set_zlim(np.min(all_z), np.max(all_z))

# Initialize lines for trajectories
line1, = ax.plot([], [], [], 'r-', label='Uranus')
line2, = ax.plot([], [], [], 'g-', label='Earth')
line3, = ax.plot([], [], [], 'b-', label='R_plot')

# Add a title
ax.set_title(f'3D Trajectories from Earth to Uranus from {launch_date} to {arrival_date}')

# Initialization function for the animation
def init():
    line1.set_data([], [])
    line1.set_3d_properties([])
    line2.set_data([], [])
    line2.set_3d_properties([])
    line3.set_data([], [])
    line3.set_3d_properties([])
    return line1, line2, line3
```

Figure 81: Set axis limits and initialize some variables.

```python
    # Update function for the animation
    def update(num):
        line1.set_data(x1_interp[:num], y1_interp[:num])
        line1.set_3d_properties(z1_interp[:num])
        line2.set_data(x2_interp[:num], y2_interp[:num])
        line2.set_3d_properties(z2_interp[:num])
        line3.set_data(x3_interp[:num], y3_interp[:num])
        line3.set_3d_properties(z3_interp[:num])
        return line1, line2, line3

    # Animation parameters
    interval = 2  # Interval in milliseconds between each frame
    frames = max_len // 1  # Number of frames

    # Create animation
    ani = FuncAnimation(fig, update, frames=frames, init_func=init, blit=True, interval=interval)

    # Show the animation
    plt.legend()
    plt.show()
```

Figure 82: Plot the animation.

Appendix E: Orbital elements from velocity and position

The aim of this appendix is to define all the equations to convert from position and velocity vectors to orbital elements. Equation 30, Equation 31, Equation 32, Equation 33, Equation 34, Equation 35, Equation 36 and Equation 37. (Curtis, 2020)

$$v_r = \boldsymbol{r} \cdot \boldsymbol{v}$$

Equation 30: Radial velocity.

$$\boldsymbol{h} = \boldsymbol{r} \times \boldsymbol{v}$$

Equation 31: Angular momentum.

$$i = \cos^{-1}\frac{h_z}{h}$$

Equation 32: Inclination.

$$\boldsymbol{N} = \boldsymbol{K} \times \boldsymbol{h}$$

Equation 33: Nodal vector.

$$\Omega = \begin{cases} \cos^{-1}\left(\frac{N_x}{N}\right) & (N_y \geq 0) \\ 360° - \cos^{-1}\left(\frac{N_x}{N}\right) & (N_y < 0) \end{cases}$$

Equation 34: Longitude of the ascending node.

$$\boldsymbol{e} = \frac{1}{\mu}(\boldsymbol{v} \times \boldsymbol{h}) - \frac{\boldsymbol{r}}{r}$$

Equation 35: Eccentricity vector.

$$\omega = \begin{cases} \cos^{-1}\left(\frac{\boldsymbol{N} \cdot \boldsymbol{e}}{Ne}\right) & (e_z \geq 0) \\ 360° - \cos^{-1}\left(\frac{\boldsymbol{N} \cdot \boldsymbol{e}}{Ne}\right) & (e_z < 0) \end{cases}$$

Equation 36: Argument of the perigee.

$$\theta = \cos^{-1}\left(\frac{\boldsymbol{e}}{e} \cdot \frac{\boldsymbol{r}}{r}\right)$$

Equation 37: True anomaly.

Appendix F: Planetary departure equations

The aim of this appendix is to define equations of a planetary departure. These equations are Equation 38, Equation 39, Equation 40, Equation 41 and Equation 42. (Curtis, 2020)

$$v_p = \sqrt{{v_\infty}^2 + \frac{2\mu_1}{r_p}}$$

Equation 38: Periapsis speed at departure.

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

Equation 39: Speed at the circular parking orbit.

$$\Delta v_{departure} = v_p - v_c$$

Equation 40: Delta-V for departure orbit.

$$e = 1 + r_p \frac{{v_\infty}^2}{\mu_1}$$

Equation 41: Eccentricity.

$$\beta = \cos^{-1}\left(\frac{1}{e}\right)$$

Equation 42: Orientation of the apse line.

Appendix G: Hardware cost per hour

This appendix shows how the cost per hour has been calculated at 14 Budget. The equations used are: Equation 43, Equation 44, Equation 45 and Equation 46.

$$Residual\ value = \frac{Acquisition\ value}{Lifetime}$$

Equation 43: Residual value.

$$Depreciation = \frac{(Acquisition\ value - Residual\ value)}{Lifetime}$$

Equation 44: Depreciation.

$$Maintenance = Acquisition\ value \cdot \frac{Maintenance\ coefficient}{100}$$

Equation 45: Maintenance.

$$Energy = Nominal\ power \cdot Price$$

Equation 46: Energy.

Table 12 shows the results where the lifetime has been calculated considering some aspects:

- Lifetime of 5 years as the manufacturer affirms.
- Workday of 8 hours.
- 335 workable days annually. Weekends and holidays aren't workable days.

The reason is because the hardware is not being used at every moment for 5 years, it is used only when a worker is using it during the workday. In addition to this, maintenance coefficient has been considered as 0% because the hardware didn't need any maintenance during the duration of the project which means that the maintenance cost was zero. Regarding nominal power and acquisition cost, they are given by the company which sells the hardware, and the cost of the energy is the average cost in Spain during 2023. Table 12

Acquisition cost (€)	Residual value (€)	Lifetime (hours)	Maintenance coefficient	Nominal power (kW)	Energy cost (€/kWh)	Depreciation	Maintenance	Energy	Cost per hour (€/h)
1100.000	0.082	13400.000	0.000	0.150	0.235	0.082	0.000	0.035	0.117

Table 12: Hardware cost per hour.

The cost per hour (€/h) is 0.117 €/h.

Appendix G: Python code

The aim of this appendix is to show the code of the optimization process completely as it is in Python.

G.1. Main script

```
def
vary_TOF(vec_venus,JD0,r_vec_earth,v_vec_earth,TOF_days,vec_earth,plot_trajectory,rp,plot_deltaV_l
aunch,gregorian_launch_date,plot_deltaV_heliocentric,print_results):

    # Find the transfer's total delta-v using Lambert's problem and data from JPL/Horizons

##############################################################################################
#####
    #                           Import general functions

##############################################################################################
#####
    import numpy as np
    import matplotlib.pyplot as plt
    from astroquery_vector_ecliptic import vector
    from Lambert import Lambert
    from oe_from_v_and_r import oe_from_v_and_r
    from datetime import datetime, timedelta
    from datetime import datetime
    import datetime
    import plotly.graph_objs as go
    import datetime
    import os
    from multiprocessing import Pool, cpu_count
    from gregorian_date import gregorian_date
    import pickle
    import matplotlib.ticker as ticker


##############################################################################################
#####
    #                           General data

##############################################################################################
#####
    pi  = np.pi
    pi2 = 2*pi

    # Define global parameters
    G = 6.6743*10**-20 #  (km^3/(kg*s^2))
    AU = 149597870.69 #(km)

    Mass_Sun = 1.989e30 # Mass of Sun (kg)
    mu_sun = G*Mass_Sun # (km^3/s^2)


    Mass_Earth = 5.9722e24 #Mass of Earth (kg)
    mu_earth = G*Mass_Earth # (km^3/s^2)
```

```
###############################################################################################
######
   #                              Initialization of the variables

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

   solution = []
   oe1_solution = []
   oe2_solution = []
   r_vec_venus_plot = []
   dv1_vec_plot = []
   porkchop_plot_solution = []
   delta_v1_sol = []
   print(gregorian_launch_date)

   #Loop to vary the arrival date and, therefore, the time of flight
   for i in range(0, len(vec_venus)):


###############################################################################################
######
      #                              Heliocentric portion

###############################################################################################
######
      if print_results == 1:
         print(f"Julian Date at Earth: {JD0}")
      JD = vec_venus['datetime_jd'][i] #Julian date at arrival
      if print_results == 1:
         print(f"Julian Date at Venus: {JD}")

      #Calculate the time of flight using the launch and arrival dates
      delta_t_days = JD - JD0
      delta_t = delta_t_days*24*3600
      if print_results == 1:
         print(f"delta_t = {delta_t_days} days = {delta_t} seconds")


      # Skip negative time of flight
      if delta_t_days < 0:
         continue

      #Breaks the for loop if the TOF is greater than a specific the number of days.
      #if delta_t_days > 61320:
       #   break

      # Loading Venus's state vector into a numpy array (and converting them to km and km/s)
      r_vec_venus = np.array([vec_venus['x'][i]*AU, vec_venus['y'][i]*AU,
                              vec_venus['z'][i]*AU]) #Convert to international system units
      r_venus = np.linalg.norm(r_vec_venus) #Calculate the norm of the vector
      # print(f"r_vec_venus = {r_vec_venus} (km), r_venus = {r_venus} km")
      r_vec_venus_plot.append(r_vec_venus) #Append the r vector into "r_vec_venus_plot" to be able to
plot it later

      v_vec_venus = np.array([vec_venus['vx'][i]*AU/24/3600, vec_venus['vy'][i]*AU/24/3600,
                              vec_venus['vz'][i]*AU/24/3600]) #Convert to international system units
```

```
        v_venus = np.linalg.norm(v_vec_venus) #Calculate the norm of the vector
        # print(f"v_vec_venus = {v_vec_venus} (km/s), v_venus = {v_venus} km/s")

        # Calculating the spacecraft's velocity vectors at Earth and Venus at the corresponding dates
        orientation = 'prograde' #Choose the orientation of the orbit

        v1_vec, v2_vec = Lambert(mu_sun, r_vec_earth, r_vec_venus, orientation, delta_t,print_results)
#Use the Lambert's problem to get the velocities
        if v1_vec.all() == 0 and v2_vec.all() == 0: #Continue with next dates if both velocities are zero
because there is no solution. This reduces the computational effort.
            if print_results == 1:
                print("No solution for this date.\n")
            continue

        v1 = np.linalg.norm(v1_vec) # magnitude of the vector v1
        if print_results == 1:
            print(f"v1_vec = {v1_vec} (km/s), v1 = {v1} km/s")
        v2 = np.linalg.norm(v2_vec) # magnitude of the vector v2
        if print_results == 1:
            print(f"v2_vec = {v2_vec} (km/s), v2 = {v2} km/s.")

        # Calculate delta-v1, delta-v2, and delta-v (total)
        dv1_vec = np.zeros(3) #Initilization of the variable

        dv1_vec = v1_vec - v_vec_earth #Calculate the delta-V at launch
        dv1 = np.linalg.norm(dv1_vec) #Calculate the norm of the vector
        if print_results == 1:
            print(f"dv1_vec = {dv1_vec} (km/s), dv1 = {dv1} km/s.")
        dv1_vec_plot.append([delta_t_days,v1_vec[0],v1_vec[1],v1_vec[2]])
        #Append the velocity and the TOF to plot it later

        dv2_vec = np.zeros(3) #Initilization of the variable
        dv2_vec = v_vec_venus - v2_vec #Calculate the delta-V at arrival
        dv2 = np.linalg.norm(dv2_vec) #Calculate the norm of the vector
        if print_results == 1:
            print(f"dv2_vec = {dv2_vec} (km/s), dv2 = {dv2} km/s.")

        dvtotal = dv1 + dv2 #Caluclate total delta-V
        if print_results == 1:
            print(f"Total delta-v = {dvtotal}")

        solution.append([delta_t_days, dvtotal]) #Save the results

        C3 = dv1**2 #Calculate the value of C3

        #Calculate the gregorian date at the arrival from the launch date and the TOF
        gregorian_arrival_date = gregorian_launch_date + datetime.timedelta(days=delta_t_days)

        porkchop_plot_solution.append([JD0,delta_t_days,
dvtotal,gregorian_launch_date,gregorian_arrival_date,C3])
        #Append the results to use in the porkchop plot
        if print_results == 1:
            print("\n")

        #Calculate and append the orbital elements of the transfer orbit
        r1,v1,vr1,h_vec1,h1,incl1,k_dir1,Nodal_vec1,Nodal1,Omega1,e_vec1,e1,w1,theta1,a1 =
oe_from_v_and_r(r_vec_earth,v1_vec,mu_sun)
        r2,v2,vr2,h_vec2,h2,incl2,k_dir2,Nodal_vec2,Nodal2,Omega2,e_vec2,e2,w2,theta2,a2 =
oe_from_v_and_r(r_vec_venus,v2_vec,mu_sun)
```

```
        oe1_solution.append([delta_t_days,r1,v1,vr1,h1,incl1,Nodal1,Omega1,e1,w1,theta1,a1])
        oe2_solution.append([delta_t_days,r2,v2,vr2,h2,incl2,Nodal2,Omega2,e2,w2,theta2,a2])


#############################################################################################
######
        #                           Launch portion

#############################################################################################
######
        vp = np.sqrt((dv1**2) + (2*mu_earth/rp)) #Calculate the periapsis speed at launch
        vc = np.sqrt(mu_earth/rp) #Calculate the speed at the circular parking orbit
        delta_V_launch = vp-vc #Calculate the delta-V to put the vehicle onto the hyperbolic departure
trajectory
        e = 1+rp*dv1**2/mu_earth #Calculate the eccentricity of that departure trajectory
        beta = np.arccos(1/e) #Calculate the orientation of the apse line of the hyperbola to the planet's
heliocentric velocity vector

        delta_v1_sol.append([delta_V_launch,e,beta,JD0,delta_t_days,gregorian_launch_date])


#############################################################################################
######
    #                           Results

#############################################################################################
######
    #Convert all the results into arrays
    delta_v1_sol_array = np.array(delta_v1_sol)
    porkchop_plot_solution_array = np.array(porkchop_plot_solution)
    dv1_vec_plot_array = np.array(dv1_vec_plot)
    r_vec_venus_plot_array = np.array(r_vec_venus_plot)
    sol = np.array(solution)
    sol_oe1 = np.array(oe1_solution)
    sol_oe2 = np.array(oe2_solution)


#############################################################################################
######
    #                           Plot the total delta-v results. Heliocentric

#############################################################################################
######
    if plot_deltaV_heliocentric ==1:
        #Plot total Delta-V vs TOF
        plt.scatter(sol[:, 0], sol[:, 1], marker='.')
        plt.xlabel('Time of Flight, days', fontsize=16)
        plt.xticks(fontsize=12)
        plt.ylabel('Total Delta-v, km/s', fontsize=16)
        plt.yticks(fontsize=12)
        plt.title('Total Delta-V vs Time of Flight', fontsize=18)
        # Annotation for departure date
        plt.annotate(f'Departure date: {gregorian_launch_date.strftime("%Y-%m-%d")}',
                     xy=(0.5, 0.9), xycoords='axes fraction', fontsize=12, ha='center')
        plt.show()

#############################################################################################
######
    #                           Plot the delta-v results. Launch
```

```
##########################################################################################
#####
    if plot_deltaV_launch == 1:
        #Plot delta-V at launch vs TOF
        plt.scatter(delta_v1_sol_array[:, 4], delta_v1_sol_array[:, 0], marker='.')
        plt.xlabel('Time of Flight, days', fontsize=16)
        plt.xticks(fontsize=12)
        plt.ylabel('Delta-v at launch, km/s', fontsize=16)
        plt.yticks(fontsize=12)
        plt.title('Total Delta-V at launch vs Time of Flight', fontsize=18)
        plt.annotate(f'Departure date: {gregorian_launch_date.strftime("%Y-%m-%d")}', xy=(0.5, 0.9),
xycoords='axes fraction', fontsize=12, ha='center')

        plt.show()

        #Plot delta-V at launch vs Date
        plt.scatter(delta_v1_sol_array[:, 5], delta_v1_sol_array[:, 0], marker='.')
        plt.xlabel('Date', fontsize=16)
        plt.xticks(rotation=45, fontsize=12)
        plt.ylabel('Delta-v at launch, km/s', fontsize=16)
        plt.yticks(fontsize=12)
        plt.title('Total Delta-V vs date', fontsize=18)
        plt.show()


##########################################################################################
#####
    #                           Calculate the trajectory

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

    if plot_trajectory==1:
    #.....................................
        #Define the functions to use
        def stumpC(z):
            if z > 0:
                return (1 - np.cos(np.sqrt(z))) / z
            elif z < 0:
                return (1 - np.cosh(np.sqrt(-z))) / z
            else:
                return 1 / 2

        def stumpS(z):
            if z > 0:
                return (np.sqrt(z) - np.sin(np.sqrt(z))) / (np.sqrt(z) ** 3)
            elif z < 0:
                return (np.sinh(np.sqrt(-z)) - np.sqrt(-z)) / (np.sqrt(-z) ** 3)
            else:
                return 1 / 6

        def kepler_U(dt, ro, vro, a, mu):
            error = 1.e-8
            nMax = 1000
            x = np.sqrt(mu) * abs(a) * dt

            n = 0
            ratio = 1
            while abs(ratio) > error and n <= nMax:
```

```
            n += 1
            z = a * x ** 2
            C = stumpC(z)
            S = stumpS(z)
            F = ro * vro / np.sqrt(mu) * x ** 2 * C + (1 - a * ro) * x ** 3 * S + ro * x - np.sqrt(mu) * dt
            dFdx = ro * vro / np.sqrt(mu) * x * (1 - a * x ** 2 * S) + (1 - a * ro) * x ** 2 * C + ro
            ratio = F / dFdx
            x -= ratio

        if n > nMax:
            print(f'\n **No. iterations of Kepler's equation = {n}')
            print(f'\n F/dFdx = {F / dFdx}\n')

        return x

    # Input data

    for i in range(0, len(sol_oe1)):
        if sol_oe1[i][0] == TOF_days:
            e = sol_oe1[i][8]
            theta0 = sol_oe1[i][10]
            theta = sol_oe2[i][10]
    for i in range(0,len(dv1_vec_plot_array)):
        if dv1_vec_plot_array[i][0] == TOF_days:
            v0_vec = np.array([dv1_vec_plot_array [i][1],dv1_vec_plot_array [i][2],dv1_vec_plot_array
[i][3]])

    r0_vec = r_vec_earth #
    ro = np.linalg.norm(r0_vec) #r0
    vo = np.linalg.norm(v0_vec)
    vro = np.dot(r0_vec, v0_vec)/ro #km/s
    dt = TOF_days*24*3600 #delta_t
    alpha = 2/ro-vo**2/mu_sun
    a = 1/alpha


    # Calculation of x using the kepler_U function
    x = kepler_U(dt, ro, vro, alpha, mu_sun)

    # Display results
    #print('————————————————————————————————————————————————————————')
    #print(f'Initial radial coordinate (km) = {ro}')
    #print(f'Initial radial velocity (km/s) = {vro}')
    #print(f'Elapsed time (seconds) = {dt}')
    #print(f'Semimajor axis (km) = {a}')
    #print(f'Universal anomaly (km^0.5) = {x}')
    #print('————————————————————————————————————————————————————————')

    #Calculate the position and velocity vectors using f,g,fdot and gdot
    chi0 = np.sqrt(mu_sun)*np.abs(alpha)*dt

    z = alpha*x**2
    C = stumpC(z)
    S = stumpS(z)
    #C = 1/2 - z/24 + z**2/720 - z**3/40320 + z**4/3628800 - z**5/479001600
    #S = 1/6 - z/120 + z**2/5040 - z**3/362880  + z**4/39916800 - z**5/6227020800

    f = 1-x**2*C/ro
    g = dt-(x**3)*S/np.sqrt(mu_sun)
    r_vec = f*r0_vec+g*v0_vec
```

```
    r = np.linalg.norm(r_vec)
    fdot = np.sqrt(mu_sun)*(alpha*x**3*S-x)/(r*ro)
    gdot = 1-x**2*C/r
    v_vec = fdot*r0_vec+gdot*v0_vec

    #Calculate E at the first and last point the the transfer trayectory
    E = 2*np.arctan(np.sqrt((1-e)/(1+e))*np.tan(theta/2))
    E0 = 2*np.arctan(np.sqrt((1-e)/(1+e))*np.tan(theta0/2))
    x = np.sqrt(1/alpha)*(E-E0)

    #Create the variables
    x_vec =  np.linspace(0,x,num=len(r_vec_venus_plot_array))
    r_plot = np.array(np.zeros((len(x_vec),3)))
    dt = np.array(np.zeros((len(x_vec),1)))
    z = np.array(np.zeros((len(x_vec),1)))
    C = np.array(np.zeros((len(x_vec),1)))
    S = np.array(np.zeros((len(x_vec),1)))

    #Calculate the position vectors of the transfer trayectory
    for i in range(0,len(x_vec)):
        z[i] = alpha*x_vec[i]**2
        C[i] = stumpC(z[i])
        S[i] = stumpS(z[i])
        #C[i] = 1/2 - z[i]/24 + z[i]**2/720 - z[i]**3/40320 + z[i]**4/3628800 - z[i]**5/479001600
        #S[i] = 1/6 - z[i]/120 + z[i]**2/5040 - z[i]**3/362880  + z[i]**4/39916800 -
z[i]**5/6227020800

    for i in range(0,len(x_vec)):
        dt[i] = (ro*vro*x_vec[i]**2*C[i]/np.sqrt(mu_sun)+(1-
alpha*ro)*x_vec[i]**3*S[i]+ro*x_vec[i])/np.sqrt(mu_sun)

    for i in range(0,len(x_vec)):
        r_plot[i] = (1-x_vec[i]**2*C[i]/ro)*r0_vec+(dt[i]-x_vec[i]**3*S[i]/np.sqrt(mu_sun))*v0_vec



###########################################################################################
######
    #                           Plot the trajectory

###########################################################################################
######
    from datetime import datetime, timedelta
    vec_plot = [] #Initialization of the variables
    for i in range(0,len(vec_earth)):
        r_vec_earth_x = vec_earth['x'][i]*AU #Convert to interntional system units
        r_vec_earth_y = vec_earth['y'][i]*AU #Convert to interntional system units
        r_vec_earth_z = vec_earth['z'][i]*AU #Convert to interntional system units
        r_vec_venus_x = vec_venus['x'][i]*AU #Convert to international system units
        r_vec_venus_y = vec_venus['y'][i]*AU #Convert to international system units
        r_vec_venus_z = vec_venus['z'][i]*AU #Convert to international system units
        JD = vec_venus['datetime_str'][i] #Julian date

vec_plot.append([JD,r_vec_earth_x,r_vec_earth_y,r_vec_earth_z,r_vec_venus_x,r_vec_venus_y,r_vec_v
enus_z]) #Append the results
    vec_plot_array = np.array(vec_plot) #Convert the variable to array

    # Data
    data = vec_plot_array
```

```
# Set the launch date in datetime format and time of flight in timedelta format
launch_date = gregorian_launch_date
time_of_flight = timedelta(days=TOF_days)

# Calculate the arrival date
arrival_date = launch_date + time_of_flight

# Format of dates in data
date_format = "A.D. %Y-%b-%d %H:%M:%S.%f"

# Variables to store the position vectors
departure_values = []
arrival_values = []

# Flag to start collecting data
collect_data = False

for row in data:
    # Convert date from string to datetime
    date = datetime.strptime(row[0], date_format)
    if date == launch_date:
        collect_data = True
    if collect_data:
        departure_values.append(row[1:4])
        arrival_values.append(row[4:7])
    if date == arrival_date:
        collect_data = False

# Convert lists to numpy arrays and then to float arrays
arrival_values_string = np.array(arrival_values)
departure_values_string = np.array(departure_values)
arrival_values_array = arrival_values_string.astype(float)
departure_values_array = departure_values_string.astype(float)

# Separate x, y, z components for each trajectory
x1, y1, z1 = arrival_values_array[:, 0], arrival_values_array[:, 1], arrival_values_array[:, 2]
x2, y2, z2 = departure_values_array[:, 0], departure_values_array[:, 1], departure_values_array[:, 2]
x3, y3, z3 = r_plot[:, 0], r_plot[:, 1], r_plot[:, 2]

# Find global minimum and maximum for axes and ensure the same scaling
min_val = min(np.min(x1), np.min(y1), np.min(z1),
              np.min(x2), np.min(y2), np.min(z2),
              np.min(x3), np.min(y3), np.min(z3))
max_val = max(np.max(x1), np.max(y1), np.max(z1),
              np.max(x2), np.max(y2), np.max(z2),
              np.max(x3), np.max(y3), np.max(z3))

# Create a 3D plot for the trajectories
fig = go.Figure()

# Add the first trajectory
fig.add_trace(go.Scatter3d(
    x=x1, y=y1, z=z1,
    mode='lines',
    name='Uranus',
    line=dict(width=8)
))

# Add the second trajectory
```

```
    fig.add_trace(go.Scatter3d(
        x=x2, y=y2, z=z2,
        mode='lines',
        name='Earth',
        line=dict(width=8)
    ))

    # Add the third trajectory
    fig.add_trace(go.Scatter3d(
        x=x3, y=y3, z=z3,
        mode='lines',
        name='Trajectory',
        line=dict(width=8)
    ))

    # Add large markers for start points (same color for all starts)
    fig.add_trace(go.Scatter3d(
        x=[x1[0], x2[0], x3[0]], y=[y1[0], y2[0], y3[0]], z=[z1[0], z2[0], z3[0]],
        mode='markers',
        name='Start Points',
        marker=dict(size=6, color='orange')
    ))

    # Add large markers for end points (same color for all ends)
    fig.add_trace(go.Scatter3d(
        x=[x1[-1], x2[-1], x3[-1]], y=[y1[-1], y2[-1], y3[-1]], z=[z1[-1], z2[-1], z3[-1]],
        mode='markers',
        name='End Points',
        marker=dict(size=6, color='purple')
    ))

    # Update layout to add axis labels, title, and set equal axis ranges
    fig.update_layout(
        title={'text': f'3D Trajectories from Earth to Uranus from {launch_date.strftime("%Y-%m-%d")}
to {arrival_date.strftime("%Y-%m-%d")}',
               'font': {'size': 30 }
           },
        scene=dict(
            xaxis=dict(title='X', range=[min_val, max_val]),
            yaxis=dict(title='Y', range=[min_val, max_val]),
            zaxis=dict(title='Z', range=[min_val, max_val])
        ),
        legend=dict(x=0.05, y=1.0,font=dict(size=30)
    )
    )
    # Show the plot
    fig.show()

##########################################################################################
#####
    #                           Trjectory animation

##########################################################################################
#####
    from mpl_toolkits.mplot3d import Axes3D
    from matplotlib.animation import FuncAnimation

    x1, y1, z1 = arrival_values_array[:, 0], arrival_values_array[:, 1], arrival_values_array[:, 2]
    x2, y2, z2 = departure_values_array[:, 0], departure_values_array[:, 1], departure_values_array[:, 2]
    x3, y3, z3 = r_plot[:, 0], r_plot[:, 1], r_plot[:, 2]
```

```
# Find the maximum length
max_len = max(len(x1), len(x2), len(x3))

# Create a common time vector
common_time = np.linspace(0, 1, max_len)

# Interpolate the shorter trajectories
x1_interp = np.interp(common_time, np.linspace(0, 1, len(x1)), x1)
y1_interp = np.interp(common_time, np.linspace(0, 1, len(y1)), y1)
z1_interp = np.interp(common_time, np.linspace(0, 1, len(z1)), z1)

x2_interp = np.interp(common_time, np.linspace(0, 1, len(x2)), x2)
y2_interp = np.interp(common_time, np.linspace(0, 1, len(y2)), y2)
z2_interp = np.interp(common_time, np.linspace(0, 1, len(z2)), z2)

x3_interp = np.interp(common_time, np.linspace(0, 1, len(x3)), x3)
y3_interp = np.interp(common_time, np.linspace(0, 1, len(y3)), y3)
z3_interp = np.interp(common_time, np.linspace(0, 1, len(z3)), z3)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Concatenate all coordinates to calculate limits
all_x = np.concatenate((x1_interp, x2_interp, x3_interp))
all_y = np.concatenate((y1_interp, y2_interp, y3_interp))
all_z = np.concatenate((z1_interp, z2_interp, z3_interp))

# Set axis limits
ax.set_xlim(np.min(all_x), np.max(all_x))
ax.set_ylim(np.min(all_y), np.max(all_y))
ax.set_zlim(np.min(all_z), np.max(all_z))

# Initialize lines for trajectories
line1, = ax.plot([], [], [], 'r-', label='Arrival')
line2, = ax.plot([], [], [], 'g-', label='Departure')
line3, = ax.plot([], [], [], 'b-', label='R_plot')

# Add a title
ax.set_title(f'3D Trajectories from Earth to Uranus from {launch_date} to {arrival_date}')

# Initialization function for the animation
def init():
    line1.set_data([], [])
    line1.set_3d_properties([])
    line2.set_data([], [])
    line2.set_3d_properties([])
    line3.set_data([], [])
    line3.set_3d_properties([])
    return line1, line2, line3

# Update function for the animation
def update(num):
    line1.set_data(x1_interp[:num], y1_interp[:num])
    line1.set_3d_properties(z1_interp[:num])
    line2.set_data(x2_interp[:num], y2_interp[:num])
    line2.set_3d_properties(z2_interp[:num])
    line3.set_data(x3_interp[:num], y3_interp[:num])
    line3.set_3d_properties(z3_interp[:num])
    return line1, line2, line3
```

```
    # Animation parameters
    interval = 2  # Interval in milliseconds between each frame
    frames = max_len // 1  # Number of frames

    # Create animation
    ani = FuncAnimation(fig, update, frames=frames, init_func=init, blit=True, interval=interval)

    # Show the animation
    plt.legend()
    plt.show()

  return sol,porkchop_plot_solution_array

# Find the transfer's total delta-v using Lambert's problem and data from JPL/Horizons
################################################################################################
#                               Import general functions
################################################################################################
import numpy as np
import matplotlib.pyplot as plt
from astroquery_vector_ecliptic import vector
from Lambert import Lambert
from oe_from_v_and_r import oe_from_v_and_r
import plotly.graph_objects as go
from datetime import datetime
import datetime
import os
from multiprocessing import Pool, cpu_count
from gregorian_date import gregorian_date
import pickle
import matplotlib.ticker as ticker

################################################################################################
#                               Define global parameters
################################################################################################
pi  = np.pi
pi2 = 2*pi

G = 6.6743*10**-20 #  (km^3/(kg*s^2))
AU = 149597870.69 #km

Mass_Sun = 1.989e30 # Mass of Sun (kg)
mu_sun = G*Mass_Sun # (km^3/s^2)

Mass_Earth = 5.9722e24 #Mass of Earth (kg)
mu_earth = G*Mass_Earth # (km^3/s^2)
R_Earth = 6378.136 #(km)

################################################################################################
#                               Define Earth's and Uranus's initial parameters
################################################################################################

# Earth
body = '399' # Earth has the number 399 assigned
```

```
start_date = '2030-08-13' # First date of the analysis
end_date = '2046-07-02' # Last date of the analysis
step = '1d' # Step used between start and end date
vec_earth = vector(body, start_date, end_date, step) #Use the function "vector" to get the data

#-------------------------------------------------------------------

# Uranus
body = '799' # Uranus has the number 799 assigned
start_date = '2030-08-13' # First date of the analysis
end_date = '2046-07-02' # Last date of the analysis
step = '1d' # Step used between start and end date

vec_uranus = vector(body, start_date, end_date, step) #Use the function "vector" to get the data


################################################################################################
#####
#                               Set if the plots are shown or not. If not, the computational effort is reduced
################################################################################################
#####

plot_trajectory = 0  # If plot_trajectory=1, The plot is shown
plot_deltaV_launch = 1 # If plot_deltaV_launch=1, The plot is shown
plot_deltaV_heliocentric = 0  # If plot_deltaV_heliocentric=1, The plot is shown
print_results = 0  # If print_results=1, The results are shown


################################################################################################
#####
#                               Initialization of the variables
################################################################################################
#####
porkchop_plot = []
TOF = []
delta_v = []
delta_v1 = []
number_of_starting_days_to_calculate = 2
TOF_selection = 5800 #Select the specific TOF of the trajectory shown
Parking_orbit_selection = 200 + R_Earth #select in km the radius of the parking orbit
launch_date = gregorian_date(vec_earth, number_of_starting_days_to_calculate) #Set a variable with all
lanunch dates

################################################################################################
#####
#                               Define the function to vary the launch date and do the calculations
################################################################################################
#####
def process_date(i):
    JD0 = vec_earth['datetime_jd'][i]
    r_vec_earth = np.array([vec_earth['x'][i] * AU, vec_earth['y'][i] * AU, vec_earth['z'][i] * AU])
#Convert to international system
    r_earth = np.linalg.norm(r_vec_earth) #Calculate the norm
    v_vec_earth = np.array([vec_earth['vx'][i] * AU / 24 / 3600, vec_earth['vy'][i] * AU / 24 / 3600,
vec_earth['vz'][i] * AU / 24 / 3600]) #Comvert to international system
    v_earth = np.linalg.norm(v_vec_earth) #Calculate the norm
    solu, porkchop_plot = vary_TOF(vec_uranus, JD0, r_vec_earth, v_vec_earth, TOF_selection,
vec_earth, plot_trajectory,
                                   Parking_orbit_selection, plot_deltaV_launch, launch_date[i],
plot_deltaV_heliocentric, print_results)
```

```
    return porkchop_plot


#############################################################################################
#######
#                                Calculate and save the results
#############################################################################################
#######
if __name__ == '__main__':
    # Directory where the file is saved
    output_directory = ''
    output_path = os.path.join(output_directory, 'prueba.pkl') #Define the name of the file

    num_processes = cpu_count()  #Get the number of processors of the CPU
    with Pool(processes=num_processes) as pool:
        results = pool.map(process_date, range(number_of_starting_days_to_calculate)) #Use all the
processors ...
        # of the CPU at the same time to accelerate the calculations

    if results:
        print("Results generated successfully")
        porkchop_plot_array = np.vstack(results)
        print("Results stacked into an array")

        # Save the results in a file using pickle
        try:
            with open(output_path, 'wb') as file:
                pickle.dump(porkchop_plot_array, file)
                print(f"Results saved to {output_path}")
        except Exception as e:
            print(f"Failed to save results: {e}")
    else:
        print("No results to save")

    print("Done")
```

G.2. Lagrange function

```
# Function to solve the Lambert's problem.
#
def Lambert (mu, r1_vec, r2_vec, orientation, delta_t,print_results):
  import numpy as np

  pi = np.pi
  pi2 = 2*pi

  if print_results == 1:
        # Print the initial data to verify if they are correct
        print ("Check the initial data.")
        print (f"mu = {mu} km^3/s^2")
        print(f"r1_vec = {r1_vec} (km)")
        print(f"r2_vec = {r2_vec} (km)")
        print (f"Orbit's orientation: {orientation}.")
        print(f"delta_t = {delta_t} sec")

  # Create numerical key based on the orbit's orientation
  if (orientation.lower() == 'prograde'):
    prograde = 1 # prograde orbit
  else:
    prograde = 0 # retrograde orbit
```

```
#---------------------------------------------------------------------
# Calculate the magnitude of the initial vectors
r1 = np.linalg.norm(r1_vec)
r2 = np.linalg.norm(r2_vec)
if print_results == 1:
  print(f"r1 = {r1:0.5f} km")
  print(f"r2 = {r2:0.5f} km")

#---------------------------------------------------------------------
# Calculate deltatheta
# Checking the sign of the z-component of r1_vec cross product r2_vec

cross_product = np.cross(r1_vec, r2_vec)
sign = np.sign(cross_product[2])

argument = np.dot(r1_vec, r2_vec)/(r1*r2)

# Calculate deltatheta according to the orbit's orientation

if prograde == 1:
    if sign >= 0:
        deltatheta = np.arccos(argument)
    else:
        deltatheta = pi2 - np.arccos(argument)
else:
    if sign < 0:
        deltatheta = np.arccos(argument)
    else:
        deltatheta = pi2 - np.arccos(argument)

if print_results == 1:
  print(f"deltatheta = {np.rad2deg(deltatheta):.2f} deg")

#---------------------------------------------------------------------
# Calculate the auxiliary variable A
A = np.sin(deltatheta)*np.sqrt((r1*r2)/(1 - np.cos(deltatheta)))
# print(f"A = {A} km")

#---------------------------------------------------------------------
# By iteration, find z
z0 = 0 # initial value for z

def S(z):
    S_value = 1/6 - z/120 + z**2/5040 - z**3/362880
    return S_value

def C(z):
    C_value = 1/2 - z/24 + z**2/720 - z**3/40320
    return C_value

def y(z):
    y_value = r1 + r2 + A*(z*S(z) - 1)/(np.sqrt(C(z)))
    return y_value

def F(z):
    F_value = (y(z)/C(z))**1.5*S(z) + A*np.sqrt(y(z)) - np.sqrt(mu)*delta_t
    return F_value

def F_prime(z):
```

```
    if z == 0:
        term_1 = np.sqrt(2)/40*y(0)**1.5
        term_2 = np.sqrt(y(0)) + A*np.sqrt(1/(2*y(0)))
    else:
        term_1 = (y(z)/C(z))**1.5*(1/(2*z)*(C(z) - 1.5*S(z)/C(z)) + 0.75*S(z)**2/C(z))
        term_2 = 3*S(z)/C(z)*np.sqrt(y(z)) + A*np.sqrt(C(z)/y(z))

    F_prime_value = term_1 + A/8*term_2
    return F_prime_value

# print(f"Initial values:\nS(z) = {S(z0)}, C(z) = {C(z0)}, y(z) = {y(z0)}\n")

# Newton Method
def Newton(x_0):
    nmax = 10000
    convergence = 1

    for j in range(0, nmax):

        f = F(x_0)
        f_prime = F_prime(x_0)
        ratio = f/f_prime
        # stop when achieve the desired precision
        if np.abs(ratio) <= 1.0e-6:
            break

        x =  x_0 - ratio
        x_0 = x #update the value of x_0

        if j == nmax - 1:
            if print_results == 1:
              print(f"Newton: no convergence.")
            convergence = 0
    return x, convergence
z, convergence = Newton(z0)

if convergence == 0:
    v1_vec = np.zeros(3)
    v2_vec = np.zeros(3)
    return v1_vec, v2_vec

if z < 0:
    if print_results == 1:
      print(f"z = {z}; the orbit is hyperbolic.")
elif z == 0:
    if print_results == 1:
      print(f"z = {z}; the orbit is parabolic.")
else:
    if print_results == 1:
      print(f"z = {z}; the orbit is elliptic.")

#---------------------------------------------------------------------
# Calculate the Lagrange Functions
# Defining the Lagrange coefficients using the iterated value of z:
f = 1 - y(z)/r1
g = A*np.sqrt(y(z)/mu)
gdot = 1 - y(z)/r2

if print_results == 1:
  print(f"f = {f}, g = {g} s, gdot = {gdot}")
```

```
 #---------------------------------------------------------------------
 # Finally, calculate the velocity vectors at r1 and r2
 # Calculating the velocities using the Lagrange coefficients:
 if print_results == 1:
   print(r1_vec, f)
   print(f*r1_vec)

 v1_vec = 1/g*(r2_vec - f*r1_vec)
 v2_vec = 1/g*(gdot*r2_vec - r1_vec)

 if print_results == 1:
   print(v1_vec)
   print(v2_vec)

 return v1_vec, v2_vec

# Test function
if __name__ == "__main__":
 import numpy as np

 # Earth data
 mu = 398600.4418 # km^3/s^2  - Gravitational parameter

 # Loading the vectors:
 r1_vec = np.array([5000, 10000, 2100]) # km
 r2_vec = np.array([-14600, 2500, 7000]) # km

 # Defining the orbit's orientation
 orientation = 'Prograde'

 # Defining the delta_t
 delta_t = 3600 # seconds

 v1_vec, v2_vec = Lambert(mu, r1_vec, r2_vec, orientation, delta_t)

 v1 = np.linalg.norm(v1_vec) # magnitude of the vector v1
 if print_results == 1:
   print(f"v1_vec = {v1_vec} (km/s), v1 = {v1} km/s")
 v2 = np.linalg.norm(v2_vec) # magnitude of the vector v2
 if print_results == 1:
   print(f"v2_vec = {v2_vec} (km/s), v2 = {v2} km/s")
```

G.3. Oe_from_v_and_r function

```
#This function is used to calculate the orbital elements using the position and the velocity vectors of the
orbit
import numpy as np
def oe_from_v_and_r (r1_vec,v1_vec,muSun): #Define the function and the data to use
   #r1_vec: position vector
   #v1_vec: velocity vector
   #muSun: mu of the central body
   r_vec = r1_vec
   v_vec = v1_vec
   pi2 = 2*np.pi

   # Step 1 ------------------------------------------------------
   r = np.linalg.norm(r_vec) # magnitude of the position vector
```

```
#print(f"r = {r} km")
#------------------------------------------------------------

# Step 2 ------------------------------------------------------
v = np.linalg.norm(v_vec) # magnitude of the velocity vector
#print(f"v = {v} km/s")
#------------------------------------------------------------

# Step 3 ------------------------------------------------------
vr = np.dot(r_vec, v_vec)/r
#print(f"vr = {vr} km/s")
#------------------------------------------------------------

# Step 4 ------------------------------------------------------
h_vec = np.cross(r_vec, v_vec) # vector angular momentum
#print(f"h_vec = {h_vec} (km^2/s)")
#------------------------------------------------------------

# Step 5 ------------------------------------------------------
h = np.linalg.norm(h_vec) # magnitude of the angular momentum
#print(f"h = {h} km^2/s")
#------------------------------------------------------------

# Step 6 ------------------------------------------------------
i = np.arccos(h_vec[2]/h) # inclination
#print(f"i = {np.rad2deg(i):.2f} deg")
#------------------------------------------------------------
# Step 7 ------------------------------------------------------
# Defining the direction k and the Nodal vector
k_dir = np.array([0.0, 0.0, 1.0])
#------------------------------------------------------------

Nodal_vec = np.cross(k_dir, h_vec)
#print(f"Nodal_vec = {Nodal_vec} (km^2/s)")
#------------------------------------------------------------

# Step 8 ------------------------------------------------------
Nodal = np.linalg.norm(Nodal_vec)
#print(f"Nodal = {Nodal} km^2/s")
#------------------------------------------------------------

# Step 9 ------------------------------------------------------
# Longitude of the ascending node
if Nodal_vec[1] >= 0.0:
    Omega = np.arccos(Nodal_vec[0]/Nodal)
else:
    Omega = pi2 - np.arccos(Nodal_vec[0]/Nodal)

#print(f"\u03A9 = {np.rad2deg(Omega):.2f} deg")
#------------------------------------------------------------

# Step 10 -----------------------------------------------------
# Eccentricity vector
e_vec = np.cross(v_vec, h_vec)/muSun - r_vec/r
#print(f"e_vec = {e_vec}")
#------------------------------------------------------------

# Step 11 -----------------------------------------------------
e = np.sqrt(e_vec[0]**2 + e_vec[1]**2 + e_vec[2]**2)
#print(f"e = {e}")
```

```
    #-------------------------------------------------------------

    # Step 12 -----------------------------------------------------
    # Argument of the perigee
    if e_vec[2] >= 0.0:
        w = np.arccos(np.dot(Nodal_vec,e_vec)/(Nodal*e))
    else:
        w = pi2 - np.arccos(np.dot(Nodal_vec,e_vec)/(Nodal*e))
    #print(f"\u03C9 = {np.rad2deg(w):.2f} deg")
    #-------------------------------------------------------------

    # Step 13 -----------------------------------------------------
    if np.dot(r_vec,v_vec) >= 0.0:
        theta = np.arccos(np.dot(e_vec,r_vec)/(e*r))
    else:
        theta = pi2 - np.arccos(np.dot(e_vec,r_vec)/(e*r))
    #print(f"\u03B8 = {np.rad2deg(theta):.2f} deg")
    #-------------------------------------------------------------

    # Semi-major axis
    a = h**2/muSun/(1 - e**2)

    #print(f"a = {a} km")
    return r,v,vr,h_vec,h,i,k_dir,Nodal_vec,Nodal,Omega,e_vec,e,w,theta,a
```

G.4. Plots

```
#########################################################################################
#####
#                           Import general functions
#########################################################################################
#####
import numpy as np
import matplotlib.pyplot as plt
from astroquery_vector_ecliptic import vector
from Lambert import Lambert
import pickle
import matplotlib.ticker as ticker

#########################################################################################
#####
#                           Load the results
#########################################################################################
#####
# Read the results from the file using pickle
with open('2030_01_01_to_2050_01_01_5d_no_TOF_starting_75.pkl', 'rb') as file:
    loaded_results = pickle.load(file)

#########################################################################################
#####
#                           Organize the data using a matrix
#########################################################################################
#####
import numpy as np
from datetime import datetime

#The data to organize is in the variable "loaded_results"
data = loaded_results

# Extract unique launch and arrival dates. "Set" removes duplicates and "list" converts them into a list.
```

```
unique_launch_dates = list(set(row[3] for row in data))
unique_arrival_dates = list(set(row[4] for row in data))

# Sort the dates to ensure the labels are in chronological order
unique_launch_dates.sort()
unique_arrival_dates.sort()

# Initialize the results matrices. The lenght is the number of unique dates.
results_C3 = np.full((len(unique_launch_dates), len(unique_arrival_dates)), np.nan)
results_delta_v = np.full((len(unique_launch_dates), len(unique_arrival_dates)), np.nan)
results_time_of_flight = np.full((len(unique_launch_dates), len(unique_arrival_dates)), np.nan)

# Fill the matrices with the corresponding values
for row in data:
    launch_date = row[3]
    arrival_date = row[4]
    C3 = row[5]
    delta_v = row[2]
    time_of_flight = row[1]
    i = unique_launch_dates.index(launch_date)
    j = unique_arrival_dates.index(arrival_date)
    results_C3[i, j] = C3
    results_delta_v[i, j] = delta_v
    results_time_of_flight[i, j] = time_of_flight

# Convert dates to strings to use as labels
row_labels = [date.strftime('%Y-%m-%d') for date in unique_launch_dates]
column_labels = [date.strftime('%Y-%m-%d') for date in unique_arrival_dates]

# Create a function to print the results matrix with labels
def print_matrix(matrix, rows, columns, name):
    print(f"\nResults matrix for {name}:")
    print(" " * 10, end="")
    for column in columns:
        print(f"{column:>10}", end=" ")
    print()

    for i, row in enumerate(matrix):
        print(f"{rows[i]:<10}", end="")
        for value in row:
            if np.isnan(value):
                print(f"{'NaN':>10}", end=" ")
            else:
                print(f"{value:>10.2f}", end=" ")
        print()

# Display the results matrices
#print_matrix(results_C3, row_labels, column_labels, "C3")
#print_matrix(results_delta_v, row_labels, column_labels, "Delta V")
print_matrix(results_time_of_flight, row_labels, column_labels, "Time of Flight")

# Find the minimum delta-v value and its indices. "nanmin" is used to not consider "nan" values
min_delta_v = np.nanmin(results_delta_v)
min_indices = np.where(results_delta_v == min_delta_v)

# Print the minimum delta-v and corresponding dates
print(f"\nMinimum Delta V: {min_delta_v:.2f}")
print("Corresponding Launch and Arrival Dates:")
for idx in zip(min_indices[0], min_indices[1]):
    launch_date = unique_launch_dates[idx[0]]
```

```
    arrival_date = unique_arrival_dates[idx[1]]
    print(f"Launch Date: {launch_date.strftime('%Y-%m-%d')}, Arrival Date: {arrival_date.strftime('%Y-%m-%d')}")

##########################################################################################
#                                Porkchop plot
##########################################################################################
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

#Function to customize the labels and number of ticks
def plot_porkchop(title, X, Y, Z, clevels, Z_time_of_flight):

    def set_ticks(ax):
        # Adjust the tick spacing to generate fewer ticks
        #x_tick_spacing, y_tick_spacing = max(1, int(len(X[0]) / 300)), max(1, int(len(Y) / 300))
        x_tick_spacing, y_tick_spacing = max(1, int(len(X[0]) / 8)), max(1, int(len(Y) / 8))
        ax.xaxis.set_major_locator(ticker.MultipleLocator(x_tick_spacing))
        ax.yaxis.set_major_locator(ticker.MultipleLocator(y_tick_spacing))
        ax.xaxis.set_tick_params(labelsize=10, rotation=90)
        ax.yaxis.set_tick_params(labelsize=10)
        ax.set_xlabel("Arrival Date (yyyy-mm-dd)", fontsize=12)
        ax.set_ylabel("Departure Date (yyyy-mm-dd)", fontsize=12)
        ax.grid(which='major', linestyle='dashdot', linewidth='0.5', color='gray')
        ax.grid(which='minor', linestyle='dotted', linewidth='0.5', color='gray')
        ax.minorticks_on()
        ax.tick_params(which='both', top=False, left=False, right=False, bottom=False)
        return
    #Add C3 contour
    plt.figure(figsize=(15, 15))  # Adjust the figure size
    cp1 = plt.contour(X, Y, Z, clevels, cmap="rainbow")
    plt.clabel(cp1, inline=True, fontsize=10)

    # Add time of flight contour with thicker lines and a distinct color
    cp2 = plt.contour(X, Y, Z_time_of_flight, colors='black', linewidths=1.5)
    plt.clabel(cp2, inline=True, fontsize=10, fmt='%1.0f')

    #Customize the figure adding title and x and y limits
    plt.title(title, fontdict={'fontsize': 16})
    ax = plt.gca()
    set_ticks(ax)
    ax.set_xlim(np.min(X), np.max(X))
    ax.set_ylim(np.min(Y), np.max(Y))
    ax.set_aspect('auto', adjustable='box')  # Maintain an appropriate aspect ratio
    plt.tight_layout()
    plt.show()
    return

#Function to assign the values, create the mesh, define the contour levels and select which information is shown
def make_porkchop_plot_from_solutions(plot_type='delv_plot'):
    y = unique_launch_dates
    x = unique_arrival_dates
    X, Y = np.meshgrid(x, y)

    # Variables for Z
    Z_c3 = results_C3
```

```
    Z_delv = results_delta_v
    Z_time_of_flight = results_time_of_flight

    # Define more contour levels
    min_c3 = np.nanmin(Z_c3)
    max_c3 = np.nanmax(Z_c3)
    min_delv = np.nanmin(Z_delv)
    max_delv = np.nanmax(Z_delv)

    c3_levels = [1,2,3,4,5,6,7,8,9, 10, 11,12,13, 14, 16, 20, 30, 50, 70, 90, 110, 130,134,136,138,140, 150, 250, 300, 350, 500, 700, 900, 1000]
    #c3_levels = [1,2,3,4,5,6,7, 8, 10, 12, 14, 16, 20, 30, 50, 70, 90, 110, 130, 150]

    delv_levels = [4, 6, 8, 10, 12, 14, 16, 20, 30, 50, 70, 90, 110, 130, 150, 250, 300, 350, 500, 700, 900, 1000]

    if plot_type == 'delv_plot':
        title = 'Porkchop plot (ΔV Total)'
        plot_porkchop(title, X, Y, Z_delv, delv_levels, Z_time_of_flight)
    elif plot_type == 'c3_plot':
        title = 'Porkchop plot (C3-characteristic energy)'
        plot_porkchop(title, X, Y, Z_c3, c3_levels, Z_time_of_flight)
    else:
        raise Exception("Error: plot types should be 'c3_plot' or 'delv_plot'")
    return

#Use the functions to create the plot
if __name__ == "__main__":

    # Prompt the user to choose the type of plot to display
    plot_type = input("Enter the type of plot to display ('c3_plot' or 'delv_plot'): ").strip()

    if plot_type not in ['c3_plot', 'delv_plot']:
        print("Invalid plot type. Please enter 'c3_plot' or 'delv_plot'.")
    else:
        make_porkchop_plot_from_solutions(plot_type)
```