UNIVERSIDAD EUROPEA DE MADRID

ESCUELA DE ARQUITECTURA, INGENIERÍA Y DISEÑO

Máster Universitario en Ingeniería Aeronáutica

FINAL PROJECT REPORT

Optimización en el diseño de trayectorias para misiones interplanetarias

Autor:
Jesús Jiménez Granados

Tutor:
Michele Armano

Curso 2023-2024

Madrid - Junio de 2024

Título: Optimización en el diseño de trayectorias para misiones interplanetarias

Autor: Jesús Jiménez Granados

Tutor: Michele Armano

Titulación: Máster Universitario en Ingeniería Aeronáutica

Curso: 2023-2024

Agradecimientos

Quisiera expresar mi profundo agradecimiento a todas las personas que han dedicado su tiempo y esfuerzo para ayudarme, apoyarme y guiarme a lo largo de estos duros años de formación académica. Sin su inestimable contribución, no habría alcanzado este hito.

En particular, quiero agradecer a mis amigos del máster, Guille, Raquel y Eugenia, quienes me han brindado su orientación y apoyo constante. Un agradecimiento especial a Monge y Víctor por sus valiosos consejos en programación y Latex. Mi tutor, Michele, ha sido fundamental en este proceso y su orientación ha sido de un valor incalculable.

También agradezco a mi familia por su inquebrantable apoyo a lo largo de estos años. Han sido mi roca en los momentos difíciles y gracias a ellos, todo el esfuerzo invertido ha valido la pena. No puedo olvidar a mis abuelos, quienes jugaron un papel crucial en mis primeros pasos hacia la ingeniería aeronáutica, llevándome a ver aviones en el aeropuerto de Barajas mientras mis padres trabajaban en días festivos.

A todos mis amigos que han estado a mi lado, brindándome ánimo y compartiendo este viaje conmigo, les estoy profundamente agradecido. Vuestra presencia y vuestros consejos han sido un gran soporte para mí.

En resumen, solo tengo una palabra para todos vosotros: gracias.

Resumen

Este Trabajo de Fin de Máster tiene como objetivo principal el desarrollo e implementación de un marco de optimización para el diseño de trayectorias interplanetarias.

Para ello se buscará **minimizar parámetros críticos de la misión, como el incremento de velocidad**. Además, se persiguen varios objetivos secundarios, que incluyen explorar y comprender los principios fundamentales de la mecánica orbital, investigar la aplicación de algoritmos de optimización y comparar diferentes criterios de optimización.

La investigación se centra en la **optimización de misiones interplanetarias dentro del sistema solar**, considerando misiones entre la Tierra y otros planetas como Marte y Venus entre otros. Con este objetivo será necesario el uso de modelos matemáticos para la mecánica orbital y los parámetros de la misión. Para ello se utilizarán diversas herramientas y librerías de software, como `Pykep` y `Pygmo`, para la implementación práctica de la optimización.

Además, se tendrán en cuenta **restricciones de la misión**, como las limitaciones de las ventanas de lanzamiento, limitaciones de número de flybys, etc. Se explorará **un criterio de optimización, minimizar el ΔV**, es decir, el incremento de velocidad necesario, dentro de las ventanas de lanzamiento especificadas.

El proyecto incluirá una explicación de la teoría detrás del problema físico, las implementaciones de software, los resultados de simulación y optimización. Además, se realizará una validación del marco de optimización y las simulaciones a través de problemas de referencia o comparaciones con datos de misiones conocidos como la misión Galileo.

Palabras clave: Astrodinámica, asistencia gravitatoria, optimización, algoritmo, Lambert, trayectoria.

Índice general

Índice de figuras

Índice de tablas

Nomenclatura

Capítulo 3

α	Semieje mayor
ΔV	Incremento de velocidad
$\hat{r}$	Vector distancia
ν	Anomalía verdadera
Ω	Longitud del nodo ascendente
ω	Argumento de la periapsis
a	Semieje mayor
b	Semieje menor
C_3	Nivel de energía característica
E	Anomalía excéntrica
e	Excentricidad
h	Momento angular específico
i	Inclinación
M	Anomalía media
T	Periodo orbital

Acrónimos

2D Bidimensional

3D Tridimensional

ESA European Space Agency

IPOPT Interior Point OPTimizer

JUICE JUpiter ICy moons Explorer

LEVEEGA Lunar Earth-Venus-Earth-Earth Gravity Assist

LTP Lambert's Targeting Problem

MGA Multiple Gravity Assist

MGA-1DSM Multiple Gravity Assist 1 Deep Space Maneuver

NASA National Aeronautics and Space Administration

NLOPT NonLinear OPTimizer

PyGMO Python Parallel Global Multiobjective Optimizer)

SNOPT Sparse Nonlinear OPTimizer

UDI User Defined Interface

UDA User Defined Algorithm

UDP User Defined Problem

URSS Union of Soviet Socialist Republics

VEEGA Venus-Earth-Earth Gravity Assist

VVEJGA Venus-Venus-Earth-Jupiter Gravity Assist

VIM Voyager Interstellar Mission

Capítulo 1

Introducción

1.1. Perspectiva del proyecto y objetivos

Desde el inicio de la exploración espacial, la humanidad se ha enfrentado a un reto. Este desafío acompaña a todas las misiones espaciales, tanto desde aquellas pioneras con mayores limitaciones hasta las más actuales cargadas de instrumentación y equipos para cualquier investigación científica.

Actualmente, la sociedad se enfrenta a un nuevo auge de la carrera espacial, tras años de parón después de la Guerra Fría. Los objetivos actuales ya no pasan por enviar al hombre a la Luna o misiones científicas para explorar y observar el espacio que nos rodea. Ahora, es el nuevo paso del ser humano para colonizar planetas y llegar lo más lejos posible.

Todo esto requiere de sistemas de propulsión y combustibles capaces de permitir alcanzar destinos lejanos como nunca antes se ha visto. Sin embargo, no es el único medio que conseguiría los objetivos ambiciosos de la humanidad, ya que optimizar las trayectorias espaciales permite, a su vez, mejorar el uso de los sistemas de propulsión y combustibles, siendo capaces de llegar a lugares jamás pisados por el ser humano.

Por ello, este Trabajo Final de Máster se centra en la optimización en el diseño de trayectorias para misiones interplanetarias, enfrentándose a los desafíos actuales y tipos de trayectorias que permitan alcanzar los objetivos del mayor número de misiones posibles.

Los objetivos principales de este proyecto serán:

- Desarrollar un código por software que permita estudiar trayectorias, optimizadas para una misión definida por el usuario.
- Evaluar las condiciones de optimización y tomar decisiones al respecto. El principal objetivo de optimización será el incremento total de velocidad necesario para llevar a cabo la misión.
- Comprobar la efectividad del software con casos reales que puedan servir de ejemplo como la misión Galileo.

Teniendo en cuenta que estos serán los objetivos principales de este proyecto, se procederá en los siguientes capítulos a explicar la teoría detrás de todo ello, el software empleado, cómo se han alcanzado a través de un código software, una comparación con ejemplos prácticos, conclusiones y propuestas de mejora.

1.2. Antecedentes históricos y tecnológicos

Considerando el interés actual por los viajes interplanetarios, ya sea para investigación científica como para futura colonización de otros planetas, muestra la importancia de desarrollar trayectorias óptimas. Gracias a ello se puede ahorrar una gran cantidad de combustible, y optimizar los recursos económicos, que añadido a un contexto de operacional, es bastante necesario.

El avance de la computación ha cambiado la capacidad para obtener trayectorias más optimizadas ya que actualmente existen herramientas muy potentes para poder llevar a cabo cualquier proceso de optimización matemática. Además, con la mejora de la tecnología, los algoritmos de optimización juegan un papel cada vez más importante ya que permiten añadir más complejidad.

Pero ya en la década de 1970 aparecen las primeras misiones que hacen uso de trayectorias optimizadas de manera habitual, con maniobras de asistencia gravitatoria, que permiten aprovechar la gravedad de los astros para impulsar la nave espacial, con la trayectoria y dirección que se desea hacia su siguiente destino.

Cabe destacar que el uso de técnicas como la asistencia gravitatoria había sido teorizada anteriormente. La primera misión que empleó este tipo de trayectorias fue Luna 3, aunque no fue usada para aumentar su velocidad sino para cambiar la trayectoria y seguir tomando diversas fotografías de la Luna.

Tras esas misiones pioneras, la misión Mariner 10, lanzada en 1973, cuyo objetivo era estudiar Mercurio y Venus en una única misión. Otras misiones conocidas han llevado a cabo este tipo de técnicas como las misiones Voyager 1 y Voyager 2, cuyo objetivo era salir fuera del Sistema Solar para su exploración, o Cassini, que se trataba de una misión de exploración científica de Saturno, sus anillos y sus lunas.

Por ello, esta introducción se centrará en dar un contexto histórico a las misiones que han empleado optimizaciones de trayectoria, que permitían un ahorro sustancial de combustible, a través de maniobras como la asistencia gravitatoria, que se explicará con más detalle en el apartado teórico.

1.2.1. Luna 3

Luna 3 fue una sonda espacial lanzada por la URSS, consiguiendo ser la tercera nave espacial lanzada con éxito hacia la Luna y siendo la primera capaz de devolver imágenes de la cara oculta del satélite natural. Sin embargo, las imágenes eran poco nítidas aunque con mejoras informáticas se consiguió generar un atlas provisional de la cara oculta lunar.

Fue la primera misión que hizo uso de una maniobra de asistencia gravitatoria con la Luna con el objetivo de cambiar el plano orbital además de la órbita en sí. El objetivo principal era conseguir que en la órbita de retorno volviera pasar sobre el hemisferio norte, lugar donde se encontraban las estaciones terrestres soviéticas, que recibirían las señales. Para lograr esto hicieron uso de las investigaciones llevadas a cabo por Mstislav Keldysh en el Instituto de Matemáticas Steklov.

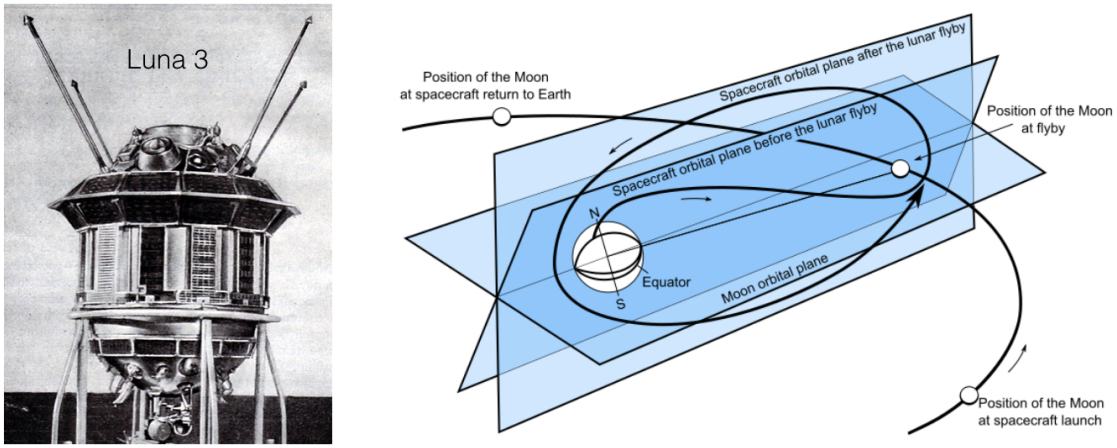


Figura 1.1: Sonda Luna 3 y su órbita

1.2.2. Mariner 10

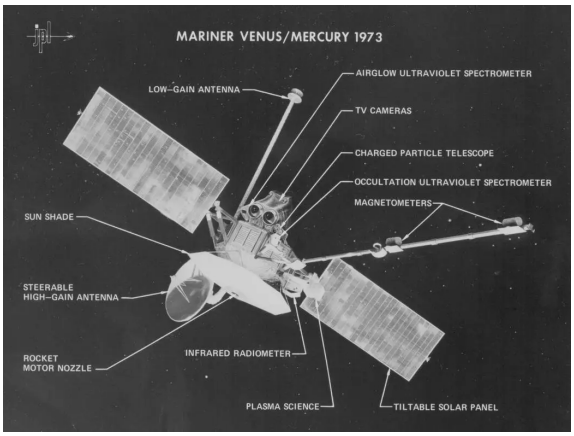


Figura 1.2: Sonda Mariner 10

Esta fue la primera nave espacial hacia Mercurio consiguiendo un gran número de primeros logros únicos en esta misión que comenzó en 1973 y comenzó a enviar datos satisfactorios en 1974:

- Primera nave espacial enviada para estudiar Mercurio.
- Primera nave que hace uso de una asistencia gravitatoria con un planeta (Venus en esta ocasión) para llegar a otro.
- Primera nave que es capaz de enviar datos de un cometa de largo período.
- Primera misión que explora dos planetas (en este caso, Mercurio y Venus) en una sola misión.
- Primera nave espacial que vuelve a su objetivo tras haber tenido ya un primer encuentro.
- Primera sonda que usa el viento solar para orientarse durante la misión.

Su objetivo era hacer un "flyby" por ambos planetas, ya fuera con objetivo de explorar de manera científica o para hacer uso del planeta como maniobra de asistencia gravitatoria (este caso también se conoce como "swingby"). Sin embargo, realizó un flyby en Venus y tres flybys en Mercurio.

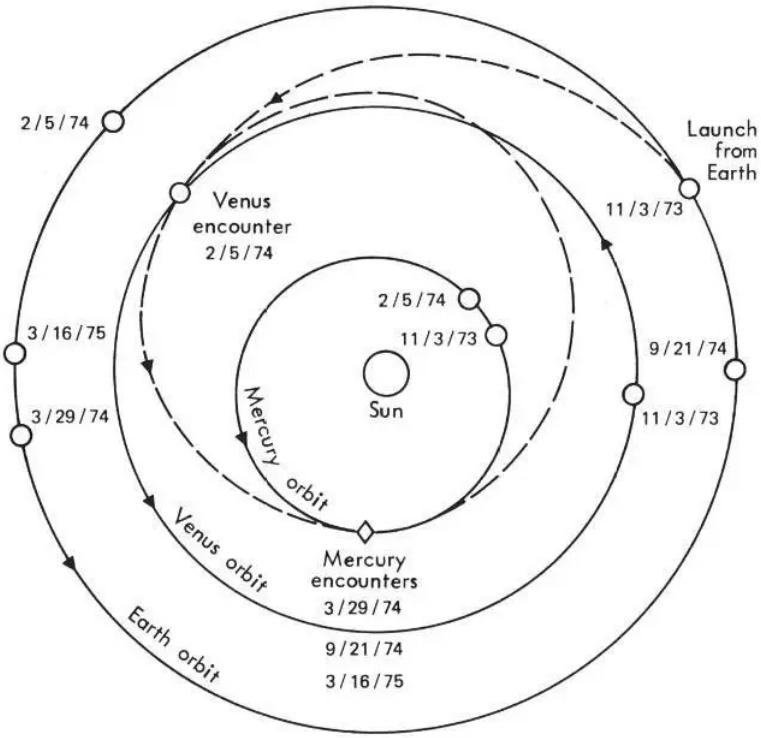


Figura 1.3: Trayectoria del Mariner 10

1.2.3. Voyager 1 y 2

Las sondas gemelas Voyager 1 y Voyager 2 fueron lanzadas en 1977 con el objetivo de explorar Júpiter y Saturno inicialmente. Consiguieron descubrir diversos detalles de ambos planetas y sus astros como los volcanes activos de la luna Io de Júpiter y las complejidades de los anillos de Saturno. Tras ello, la Voyager 2 amplió su exploración a Urano y Neptuno. Tras su paso por Neptuno en 1989, ambas misiones se ampliaron y dieron lugar a la Misión Interestelar Voyager (VIM).

El objetivo de esa misión interestelar era ampliar la exploración más allá del Sistema Solar. El objetivo de la NASA era llegar desde los planetas exteriores hasta los límites exteriores de la esfera de la influencia del sol, e incluso, más allá. Gracias a explorar este entorno, ambas sondas pueden penetrar el límite de la heliopausa entre el viento solar y el medio interestelar, consiguiendo realizar mediciones de campos, partículas y ondas no afectados por el viento solar.

Lo más destacado es el uso de asistencias gravitatorias, como se puede ver con ambas misiones, que hicieron uso de esta técnica para pasar junto a Júpiter y y tomar impulso hacia Saturno. Además, la Voyager 2 continúa haciendo lo mismo para tomar impulso hacia Urano, después hacia Neptuno, y tras ello, más allá. La Voyager 1, hizo uso de una asistencia gravitatoria en Saturno para continuar su viaje hacia fuera del Sistema Solar.

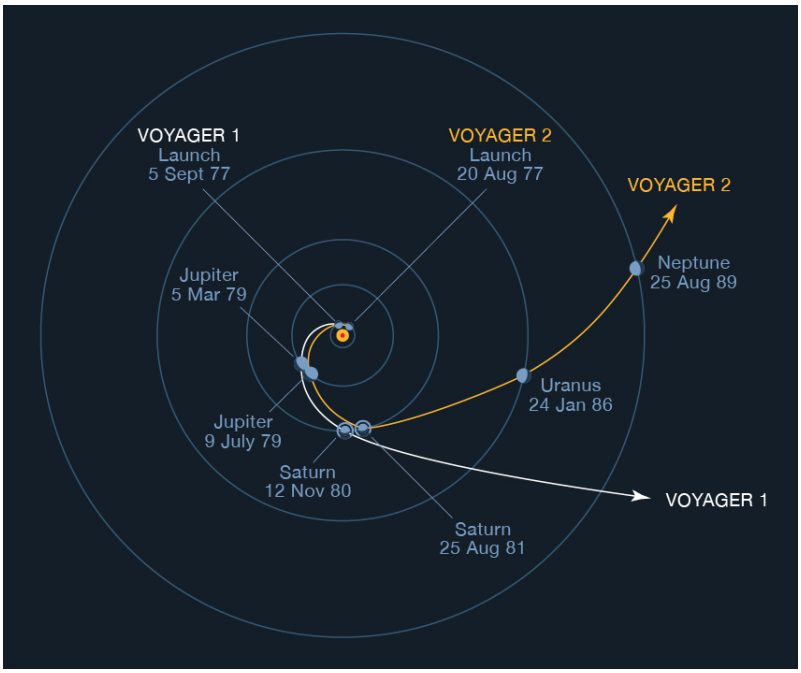


Figura 1.4: Trayectoria de las sondas Voyager 1 y Voyager 2

1.2.4. Galileo

La misión Galileo tenía como objetivo investigar científicamente Júpiter y sus lunas. Uno de los hallazgos más importantes fue encontrar que en la luna helada de Júpiter, Europa, existía una alta probabilidad de encontrar un océano subterráneo, con más agua que la cantidad total que tiene la Tierra. Comenzó en 1989, llegando a Júpiter en 1995 y, finalizando la misión en 2003 al estrellarse dentro del planeta.

Las dos naves que componen esta misión son un orbitador y una sonda atmosférica. Fueron lanzadas en 1989 durante el vuelo STS 34 del orbitador Atlantis. Para alcanzar el objetivo, se siguió una asistencia gravitatoria Venus-Tierra-Tierra (VEEGA). Gracias al uso de esta maniobra se pudo conseguir la velocidad suficiente para llegar a Júpiter.

Además, esta misión pudo también aprovechar cada vez que se sobrevolaba por la Tierra para observar el cinturón de asteroides del Sistema Solar, estudiando de cerca dos de ellos, Gaspra e Ida.

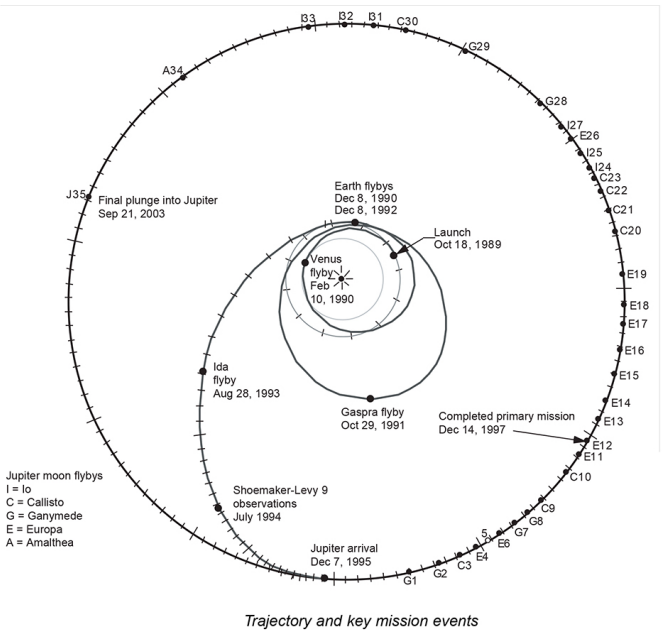


Figura 1.5: Trayectoria de la misión Galileo

Gracias a estos flybys, la misión puede desarrollar incrementos de velocidad de alrededor de 6000-8000 km/s. Sin embargo, la capacidad real de la nave era muy inferior, en torno a cuatro o cinco veces menos que la necesitada para desarrollar la misión (ver D´Amario et al. (1992)).

El motivo de hacer una trayectoria tipo VEEGA se debe a que la Etapa Superior Inercial (IUS) de Galileo, que le permitió a la nave salir de la órbita terrestre no disponía de la potencia suficiente para hacer maniobras más directas, que requerían más ΔV.

1.2.5. Cassini

La misión Cassini tenía como objetivo estudiar de manera específica Saturno y su complejo sistema de anillos y lunas. El nivel de detalle era tan alto que se realizó un gran esfuerzo por alcanzar los estándares que se buscaban. Comenzó en 1997 y finalizó en 2017.

Esta misión también hizo uso de asistencias gravitatorias para alcanzar Saturno. Para ello, realizó una maniobra Venus-Venus-Tierra-Júpiter, VVEJGA, que le permitió llegar en más de seis años a Saturno.

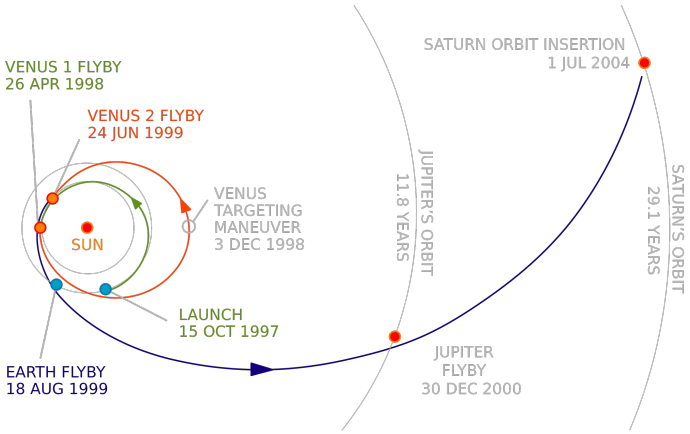


Figura 1.6: Misión interplanetaria de la sonda Cassini, que hace uso de asistencias gravitacionales

Para la misión, se hizo uso del cohete más potente que disponía la NASA, pero no tenía la capacidad para hacer directamente una transferencia a Saturno. Por ello, se planificaron las diversas asistencias gravitatorias que permitieron alcanzar el objetivo.

1.2.6. JUICE

JUICE (JUpiter ICy moons Explorer) es la primera gran misión del programa de la ESA Cosmic Vision 2015-2025. Fue lanzada el 14 de abril de 2023 y tiene prevista su llegada a Júpiter en julio de 2031. Tiene como objetivo explorar Júpiter y sus tres lunas más grandes, Ganímedes, Calisto y Europa.

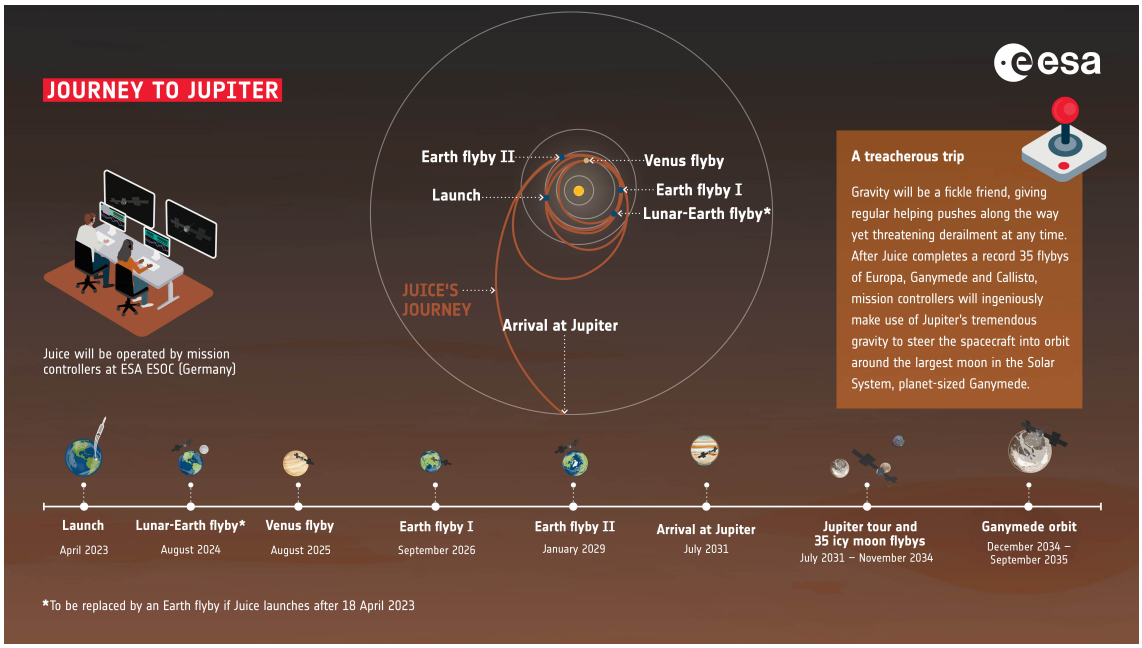


Figura 1.7: Misión interplanetaria de JUICE

Para conseguir alcanzar estos objetivos, la misión hace uso de asistencias gravitatorias con una secuencia Luna Tierra-Venus-Tierra-Tierra, LEVEEGA. Este tipo de maniobras permite completar la trayectoria haciendo un menor uso de combustible y ahorrando tam-

bién peso al ser necesarios sistemas propulsivos menos potentes, a cambio de un mayor tiempo de misión como se puede observar en esta situación al llegar a Júpiter en 2031.

Capítulo 2

Mecánica orbital, análisis de misión y algoritmos

El estudio de la Astrodinámica y la Mecánica Celeste ha sido esencial para comprender el universo y clave para la exploración humana más allá de la Tierra. Ambas disciplinas desempeñan un papel importante en el desarrollo, planificación, elaboración y ejecución de misiones interplanetarias para alcanzar el destino con suficiente precisión y eficacia. El aumento de interés y actividades en el ámbito de la exploración espacial hace que el estudio de trayectorias óptimas esté tomando cada vez más importancia.

Por ese motivo, este capítulo del Trabajo Fin de Máster se centrará en introducir el tema de la Astrodinámica y la Mecánica Celeste, además de una pequeña introducción de algoritmos. el motivo es establecer un marco teórico que permita entender el desarrollo posterior del código y los objetivos finales de este proyecto.

Los siguientes puntos serán desarrollados en esta introducción teórica:

- Explorar y comprender los principios fundamentales de la mecánica orbital, las trayectorias interplanetarias y los factores que influyen en el diseño óptimo de trayectorias.
- Explorar conceptos como las leyes de Kepler, los problemas de los Dos Cuerpos, el problema de Lambert, las perturbaciones orbitales y las interacciones gravitatorias entre cuerpos celestes.
- Introducir brevemente el concepto de algoritmo aplicado a la optimización de trayectorias interplanetarias y posibles librerías de software que permiten una integración por código.

Entonces, este capítulo se centrará en desarrollar las bases para comprender como una trayectoria como la mostrada en *Figura 2.1* se puede llevar a cabo y la física que existe debajo de la misma.

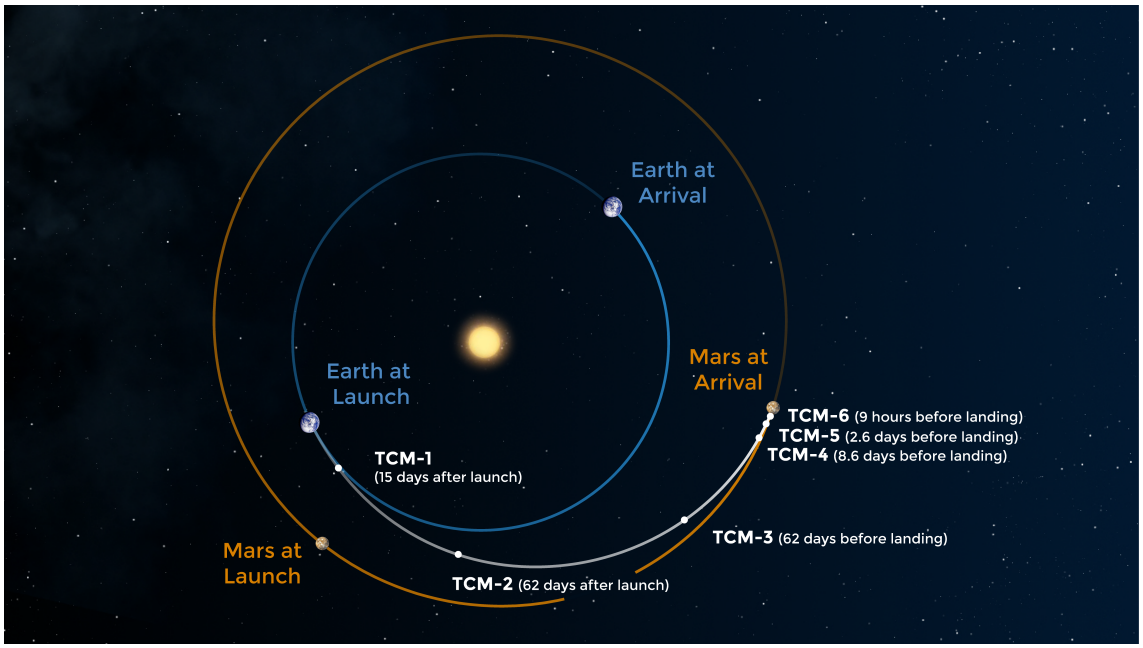


Figura 2.1: Trayectoria del Mars Perseverance

Por tanto, la mecánica clásica de Newton explica desde el inicio la problemática que existía en la interacción gravitatoria entre dos cuerpos. Ese problema de los dos cuerpos serán analizado más adelante en esta introducción teórica junto al problema de los múltiples cuerpos. El problema de los dos cuerpos da lugar al estudio de la mecánica orbital como disciplina, la cual será tratada en el siguiente apartado:

2.1. Mecánica orbital

El primer tema principal a tratar es la Mecánica Orbital, que también se denomina Astrodinámica, y es el estudio de los objetos en movimiento, como cohetes, satélites y otras naves espaciales, en relación con la mecánica celeste.

Las leyes del movimiento de Newton y la ley de la gravitación universal se utilizan para investigar el comportamiento de estos objetos y facilitar el diseño y el control de las misiones espaciales. Además, hay otros conceptos críticos que son esenciales en Mecánica Orbital:

- Leyes de Kepler: En el siglo XVII, Johannes Kepler trabajó para comprender el movimiento celeste y formuló las famosas tres leyes de Kepler. Describen la naturaleza elíptica de las órbitas planetarias, la ley de igual área y la relación entre el periodo orbital de un planeta y su distancia al Sol. Estas leyes son esenciales para determinar las características básicas de la órbita de un objeto.
- Parámetros orbitales: Existen varios parámetros clave para definir la órbita de un objeto, facilitando su comprensión. Algunos de ellos son el semieje mayor, la excentricidad, la inclinación y el argumento de periapsis. Estos parámetros definirán el tamaño, la forma y la orientación de una órbita, permitiendo cálculos precisos para la planificación de trayectorias.

- Problemas de Dos Cuerpos: El problema de dos cuerpos es el que La mecánica orbital implica la resolución del problema de dos cuerpos, que trata del movimiento de un cuerpo en el campo gravitatorio de otro. Para escenarios más complejos en los que intervienen varios cuerpos (problemas de N-cuerpos), se emplean técnicas como la integración numérica y la teoría de perturbaciones para modelizar con precisión el movimiento de los objetos celestes.
- Maniobras orbitales: Para alcanzar otros cuerpos celestes o ajustar las características de una órbita, las naves espaciales deben realizar maniobras orbitales. Esta sección tratará conceptos clave como las transferencias de Hohmann, las maniobras bi-impulsivas y el uso de ΔV (cambio en la velocidad) para alterar las trayectorias.
- Perturbaciones orbitales: Las órbitas del mundo real se ven afectadas por diversas perturbaciones, como la influencia gravitatoria de la Luna, la presión de la radiación solar y la resistencia atmosférica. Entender cómo tener en cuenta estas perturbaciones es crucial para la precisión de la misión.
- Asistencias gravitatorias: Una de las técnicas más innovadoras de la mecánica orbital es el uso de ayudas gravitatorias. Las naves espaciales se aprovechan de la gravedad de otros cuerpos celestes, como pueden ser planetas, consiguiendo ganar energía y cambiar su trayectoria de manera eficiente, permitiendo alcanzar objetivos lejanos usando poco combustible.
- Aproximación por Secciones Cónicas (Patched Conics): La aproximación por secciones cónicas es un método de simplificación que divide una trayectoria interplanetaria en múltiples segmentos, en los que cada uno es aproximado por una sección cónica conocida (elipse, parábola o hipérbola). Se utiliza esta aproximación a menudo para la planificación preliminar de misiones.

Entender los principios fundamentales de la Mecánica Orbital es esencial a la hora de diseñar trayectorias óptimas que permitan alcanzar los destinos que se planifican en misiones interplanetarias. Por ello, equipar de herramientas y conocimientos a los planificadores de dichas misiones se vuelve una tarea extremadamente importante, debido a que es necesario calcular precisos itinerarios para minimizar el consumo de combustible o el tiempo de misión con una especial precisión. Es por ello que los siguientes subapartados son muy relevantes para el diseño y planificación de trayectorias con éxito.

2.1.1. Leyes de Kepler

Las leyes de Kepler fueron formuladas en el siglo XVII por el astrónomo y matemático alemán con mismo apellido, Johannes Kepler, describiendo así el movimiento de las órbitas de los astros alrededor del Sol. Estas leyes se han convertido en la base de la Astrodinámica y Mecánica Celeste. A continuación se exponen las tres leyes de Kepler:

Primera Ley de Kepler

Esta ley estableció que todos los planetas que giraban alrededor de nuestro Sol lo hacían con órbitas elípticas. Posteriormente, esto permitió extrapolar a que la trayectoria de un planeta o de cualquier objeto en órbita alrededor de un cuerpo central es una elipse con el cuerpo central en uno de los dos focos.

Es fruto del descubrimiento de Kepler al descartar las órbitas circulares tras la observación de los diferentes astros e introducir el concepto de órbitas elípticas, que poseen el parámetro de la excentricidad, determinando así la diferencia de forma respecto a una órbita circular.

Segunda Ley de Kepler

La segunda ley de Kepler establece que una línea que une un planeta al Sol barre áreas iguales en intervalos de tiempo iguales. En ella se pone de manifiesto, como la velocidad varía a lo largo de una órbita elíptica. Un astro se mueve más rápido cuando se encuentra más cerca del Sol (perihelio) y más despacio cuando se encuentra más lejos (afelio).

Esta ley se vuelve esencial en las misiones interplanetarias, concretamente en las transferencias, ya que éstas son las que deben de ser optimizadas. Un ejemplo es la transferencia de Hohmann, en la que dar el impulso en el momento adecuado permitirá lograr un encuentro eficaz y preciso con los planetas objetivo.

Tercera Ley de Kepler

La tercera ley permite relacionar el periodo orbital (T) con su semieje mayor (a), estableciendo así una relación matemática en la que T^2 es proporcional a a^3 y tienen una relación constante.

Además, permite determinar los períodos orbitales y las distancias a partir de valores conocidos y se convierte en una gran herramienta para las misiones interplanetarias. También se puede calcular el tiempo necesario para alcanzar un destino y el semieje mayor de una órbita.

Gracias a las leyes de Kepler se revolucionó la comprensión del cosmos, siendo un pilar básico para la Astrodinámica. El dominio de estas leyes es importante para los planificadores de misión, ya que pueden establecer trayectorias con el objetivo de minimizar el consumo de combustible o el tiempo de viaje, permitiendo así llegar aún más lejos dentro de la exploración espacial. A continuación, se hablará de los parámetros que definen las órbitas.

2.1.2. Problema de los dos cuerpos

El problema de los dos cuerpos simplifica la mecánica orbital, considerando que existe una interacción gravitacional entre un astro y el objeto que orbita alrededor de él. Asume, por lo tanto, que no existen otras fuerzas gravitacionales en el entorno que actúen en este sistema. Las características principales de este problema son las siguientes:

- Órbitas Keplerianas: en este tipo de problemas, las órbitas siguen las leyes formuladas por Kepler, por lo tanto, pueden ser elipses, parábolas o hipérbolas.
- Ley de gravitación universal de Newton: para el problema de los dos cuerpos se empleará esta ley universal
- Conservación de la energía y momento angular: la energía total y el momento angular se conservan en este tipo de problemas lo que permite calcular y predecir los parámetros orbitales y las características de la órbita

- Órbitas predecibles: dadas unas condiciones iniciales, este problema permite calcular de manera precisa las futuras posiciones y la velocidad de los objetos, siendo esencial para el cálculo de los tiempos y conseguir alcanzar los objetivos de la misión.

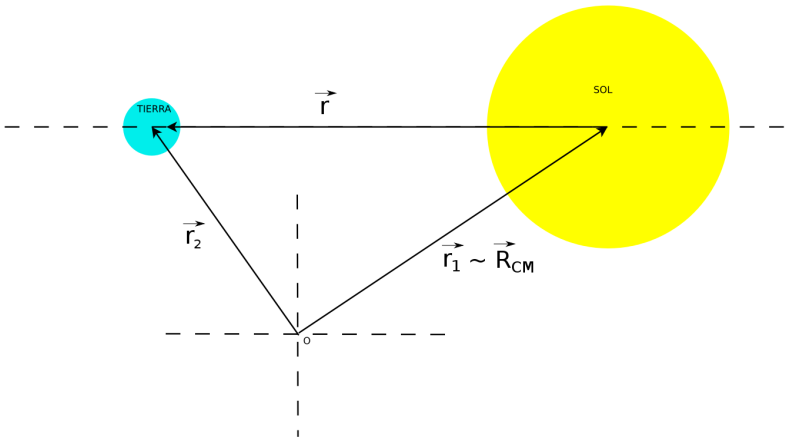


Figura 2.2: Ejemplo del problema de los dos cuerpos con la Tierra y el Sol

Relación entre los dos cuerpos. Fuerzas gravitatorias

Asumiendo entonces que el potencial gravitacional de Newton es aplicable cuando tenemos este problema con dos cuerpos, se usarán la siguiente ecuación particularizadas para dos cuerpos genéricos:

$$m_1 \cdot \ddot{\mathbf{r}}_1 = G\frac{m_1 \cdot m_2}{\mathbf{r}^2} \cdot \hat{\mathbf{r}} \tag{2.1}$$

$$m_2 \cdot \ddot{\mathbf{r}}_2 = G\frac{m_1 \cdot m_2}{\mathbf{r}^2} \cdot \hat{\mathbf{r}} \tag{2.2}$$

Donde $\hat{\mathbf{r}}$ representa al vector distancia, y $\mathbf{r}$ al módulo del vector distancia. Por su parte $\hat{\mathbf{r}}$ puede descomponerse en $\hat{\mathbf{r}} = \mathbf{r}_1 - \mathbf{r}_2$. Cuando una letra se representa con negrita como $\mathbf{r}$, implica que es un vector.

Si el problema se simplifica, teniendo en cuenta y estableciendo un centro de masas, el problema de dos cuerpos puede pasar a uno con un centro de masas y una masa reducida. A continuación, se exponen las expresiones que definen el centro de masas y la masa reducida:

$$M = m_1 + m_2 \tag{2.3}$$

$$\frac{1}{m} = \frac{1}{m_1} + \frac{1}{m_2} \tag{2.4}$$

Gracias a este nuevo sistema se puede llegar a la siguiente ecuación:

$$\ddot{\mathbf{r}} = \dot{\mathbf{v}} = G\frac{M}{\mathbf{r}^2} \cdot \hat{\mathbf{r}} \tag{2.5}$$

Donde $G \cdot M = G \cdot (m_1 + m_2)$ se puede denominar parámetro estándar gravitacional. Para profundizar en las demostraciones, ver Vallado (2001) o Armano (2023).

Con estas ecuaciones haciendo uso de la ley de gravitación de Newton para el problema de los cuerpos en un sistema de coordenadas cartesianas, se pueden deducir los parámetros de las órbitas Keplerianas.

Conservación del momento angular

La ecuación definitiva, gracias a las ecuaciones de Newton, que definirá la conservación de la energía es la siguiente:

$$\mathbf{r} \times \mathbf{v} = \mathbf{h} = vector\ constante \tag{2.6}$$

El término $\mathbf{h}$ se denomina momento angular específico. Esto implica que el plano orbital será una constante en el tiempo, por lo que el movimiento ocurrirá en ese plano, demostrando la primera ley de Kepler. Para profundizar en las demostraciones, ver Vallado (2001) o Armano (2023).

Vector de excentricidad

Como el producto vectorial del vector posición y del vector velocidad, resultaba interesante para obtener la conservación del momento angular, para llegar al vector de excentricidad, se apostará por el producto vectorial del vector velocidad y del momento angular específico, $\mathbf{v} \times \mathbf{h}$.

Teniendo en cuenta lo anterior, se puede concluir que teniendo en cuenta que la cantidad bajo la derivada temporal debe ser un vector constante, se denomina adecuadamente como $GM\mathbf{e}$, para identificar al vector de excentricidad $\mathbf{e}$ por lo que todo se reduce a:

$$\mathbf{e} = \frac{1}{GM}\mathbf{v} \times \mathbf{h} - \frac{\mathbf{r}}{r} \tag{2.7}$$

Ecuación polar

Se puede demostrar que los vectores de excentricidad ($\mathbf{e}$) y el de momento angular específico ($\mathbf{h}$) son perpendiculares, lo que permite relacionar el radio de la órbita y el vector de excentricidad $\mathbf{e}$, apareciendo entonces el término de la anomalía verdadera (ν).

Dicha ecuación junto a relacionar dicha ecuación con el producto escalar de $\mathbf{r}$ y $\mathbf{e}$, se llega a obtener la ecuación polar (para estudiar con más en detenimiento las demostraciones, ver Vallado (2001) o Armano (2023)):

$$er\ \cos\nu = \frac{h^2}{GM} - r \tag{2.8}$$

$$\boxed{p = \frac{h^2}{GM}} \tag{2.9}$$

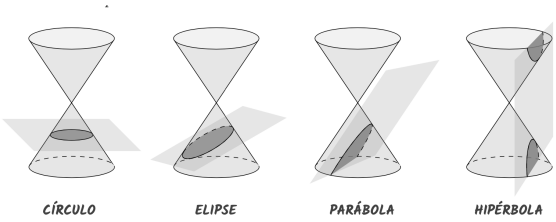
Finalmente, se expone la ecuación polar de la órbita:

$$\boxed{r(\nu) = \frac{p}{1 + e\cos\nu}} \tag{2.10}$$

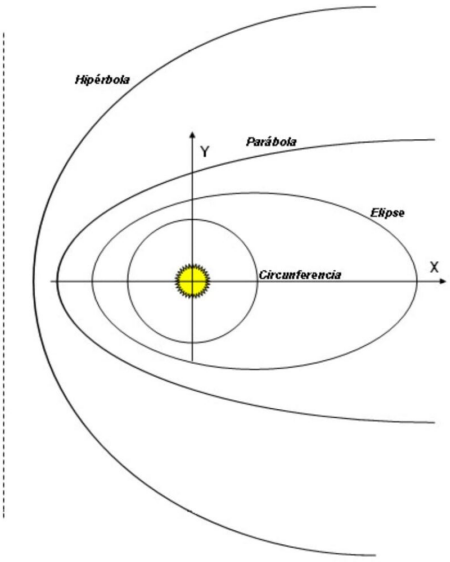
2.1.3. Secciones cónicas

Cuando ν es 0, el objeto se encuentra en el perigeo de la órbita, donde la proyección de la excentricidad sobre el radio vector es máxima. Lo que servirá para definir el tipo de órbita gracias a las secciones cónicas que tendrá en función de la excentricidad (como se explicará en el subapartado 2.1.6 dentro de la sección de parámetros orbitales):

- $e = 0 \longrightarrow$ Órbita circular
- $0 < e < 1 \longrightarrow$ Órbita elíptica
- $e = 1 \longrightarrow$ Órbita parabólica
- $e > 1 \longrightarrow$ Órbita hiperbólica



(a) Secciones cónicas



(b) Órbitas en 2D

Figura 2.3: Secciones cónicas y sus representaciones bidimensionales en plano

Por ello, en los próximos subapartados se han analizarán tipos de órbitas de secciones cónicas como las comentadas anteriormente.

Elipse

La elipse, como anteriormente se ha clasificado, tiene una excentricidad cuyo valor se encuentra entre 0 y 1. Este fue el tipo de órbitas que definió Kepler cuando apreció que los planetas giraban alrededor del sol, describiendo trayectorias no circulares, sino con forma elíptica. Una orbita circular no deja de ser un caso de una órbita elíptica, pero sin excentricidad.

Hipérbola

La hipérbola es la trayectoria orbital que mostrará cuando se supere un valor de excentricidad $e > 1$. Proviene también de una sección cónica y mostrará dos posibles trayectorias. A pesar de ello, realmente solo se usará una de las dos ramas que se muestren, la otra quedará vacante. El caso de la parábola es un caso concreto de hipérbola, ya que la excentricidad valdrá 1, mostrando una única curva, es decir, una única rama.

2.1.4. Elementos orbitales geométricos

Analizando de manera más detallada, la ecuación polar puede mostrar los elementos orbitales geométricos. Comenzando por obtener el resultado final de la relación de la ecuación polar con el periapsis y el apoapsis de la órbita (puntos perifocales de la órbita):

$$\boxed{r = \frac{a(1-e^2)}{1+e\cos\nu}} \tag{2.11}$$

También, puede obtenerse el semieje menor (b) gracias a los radios en la periapsis junto con el teorema de Pitágoras y las propiedades de la elipse:

$$\boxed{b = a\sqrt{1-e^2}} \tag{2.12}$$

Además, se pueden demostrar tanto la segunda ley de Kepler como la tercera ley de Kepler. Para comprobar la tercera ley de Kepler, será necesario hacer uso de la segunda ley de Kepler. Después, se evaluará el recorrido completo del objeto en la órbita elíptica para obtener la relación entre semieje mayor (a) y el periodo (T). Teniendo la segunda ley de Kepler formulada antes, se procede a desarrollar la tercera ley de Kepler:

$$\boxed{\frac{4\pi^2}{GM} = \frac{T^2}{a^3}} \tag{2.13}$$

2.1.5. Energía de la órbita

Volviendo a la ecuación del vector excentricidad (ecuación 2.7), se busca sentar las bases del desarrollo para la ecuación de la energía. Relacionando diferentes parámetros como p y el semieje mayor a están relacionados con la excentricidad e y el momento angular específico h, además se usa la ecuación de la energía (ver Vallado (2001) o Armano (2023)), que siendo dividida entre dos, permitirá alcanzar finalmente a la ecuación llamada Vis-viva:

$$\boxed{\frac{1}{2}v^2 - \frac{GM}{r} = -\frac{GM}{2a}} \tag{2.14}$$

Gracias a la ecuación Vis-viva se puede ver que en función del semieje mayor, el tipo de órbita que tendrá el objeto. Hay dos casos posibles, si el semieje mayor es igual que la órbita o si tiene un valor infinito, obteniéndose entonces la velocidad necesaria para dichas órbitas:

$$\begin{cases} v = \sqrt{\dfrac{GM}{r}} \; si \; a = r \Rightarrow \; Órbita \; circular \\ v = \sqrt{\dfrac{2GM}{r}} \; si \; a = \infty \Rightarrow Velocidad \; de \; escape \end{cases} \tag{2.15}$$

2.1.6. Parámetros orbitales

Los parámetros orbitales son características fundamentales que van a definir el tipo de órbita, su tamaño, su orientación y la posición del objeto que se encuentre en dicha órbita. Entender estos parámetros será esencial para la correcta definición de trayectorias en el momento en el que se busque calcular y diseñar las más óptimas. los siguientes parámetros son los más significativos en la planificación de la misión:

Semieje mayor (a)

El semeje mayor es el parámetro que definirá el tamaño de la órbita. Se caracteriza por ser la distancia media desde el centro de la masa sobre la que orbita y el objeto que realiza la órbita. Para obtener este semieje mayor, lo que se hará será hacer la suma de la distancia al astro desde el punto más alejado (apoapsis) y la distancia al astro desde el punto más cercano (periapsis), dividiéndose entre dos. Este parámetro será importante para el cálculo de incrementos de velocidades y cambios energéticos, cuando se planifiquen cambios orbitales dentro del diseño de misión.

Excentricidad (e)

La excentricidad definirá la forma que tendrá la órbita, siendo 0 cuando sea una órbita circular, entre 0 y 1 cuando sea una órbita elíptica, 1 cuando sea parabólica y mayor que 1 cuando sea hiperbólica. La excentricidad afecta de manera directa al diseño de transferencias y, por lo tanto, a la planificación de la misión espacial, es por ello, por lo que se buscarán órbitas no demasiado excéntricas, ya que a mayor excentricidad, mayor complejidad.

Inclinación (i)

La inclinación es el parámetro que definirá el ángulo entre el plano orbital del objeto frente a un plano de referencia, que suele ser, en el caso de la Tierra, el plano eclíptico. La inclinación debe ser considerada por varios motivos cuando se planifica la misión. Uno de los motivos es que, dependiendo del punto de lanzamiento, la órbita tendrá una inclinación predefinida. Por otro lado, influirá a la hora de realizar transferencias de órbitas y a la llegada al astro objetivo de la misión.

Argumento de la periapsis (ω)

Define la orientación de la órbita dentro del plano orbital, haciendo uso del ángulo, medido entre el nodo ascendente de la órbita y la periapsis. Siendo importante para deter-

minar el punto preciso dentro de una órbita en la que se encontrará el objeto de la misión espacial.

Longitud del nodo ascendente (Ω)

La longitud del nodo ascendente identifica la localización a la que se encuentra la órbita respecto al plano de referencia, como puede ser por ejemplo el plano eclíptico. Gracias a ello da información de referencia respecto a la orientación de la órbita. Por ello, este parámetro jugará un papel a la hora de determinar la ventana de lanzamiento, además de alinear la trayectoria del objeto con la del astro objetivo.

Anomalía verdadera (ν)

La anomalía verdadera mide el ángulo entre el semi eje mayor y la línea que une el cuerpo central con la periapsis. gracias a ello se podrá determinar la posición relativa que tiene el objeto vitando respecto a astro central, ayudando al diseño de trayectorias y la navegación.

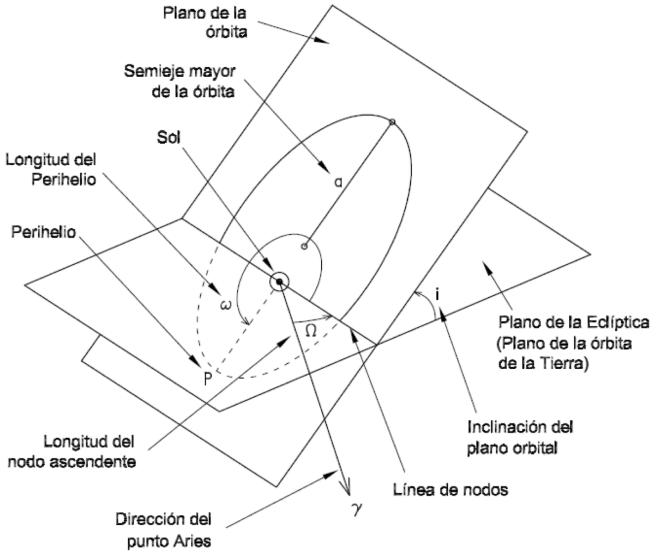


Figura 2.4: Parámetros orbitales en una órbita alrededor del Sol

Anomalía excéntrica (E)

Por otro lado, es importante definir la anomalía excéntrica como el ángulo que forman la proyección de la posición dentro de una órbita elíptica que sigue un objeto en la correspondiente órbita circular, cuyo radio es el semieje mayor de la órbita elíptica, con el eje de la órbita elíptica. Para su identificación suele emplearse la letra E.

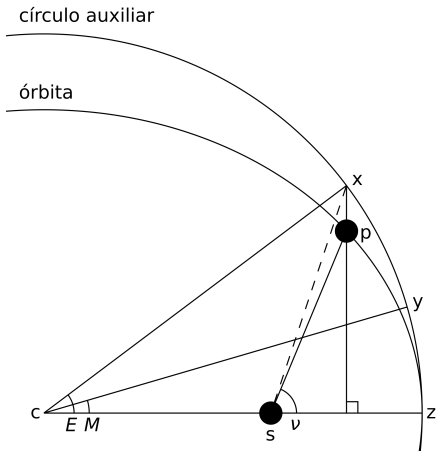


Figura 2.5: Anomalía excéntrica en una órbita

Haciendo uso de las propiedades geométricas se llega a la siguiente ecuación:

$$r(E) = a(1 - e\cos E) \tag{2.16}$$

La anomalía verdadera (ν) se puede relacionar con la anomalía excéntrica (E), quedando la siguiente ecuación final:

$$\tan\frac{\nu}{2} = \sqrt{\frac{1+e}{1-e}}\tan\frac{E}{2} \tag{2.17}$$

Anomalía media (M)

La anomalía media (M) es el ángulo que forma un objeto ficticio en una órbita circular de radio con movimiento uniforme el semieje mayor de la elipse con el eje de la misma, en el mismo tiempo orbital que el objeto verdadero en su propia órbita elíptica.

Esa circunferencia se denomina circunferencia principal. Además, también la anomalía verdadera es la fracción de un periodo orbital que ha transcurrido expresada como ángulo. Se representa con la letra M y responde a la siguiente ecuación:

$$M = n(t - t_0) \tag{2.18}$$

El término n es el movimiento medio, que es la velocidad angular de un objeto medido en grados o radianes en un día y se puede definir como $n = \frac{2\pi}{T}$. Gracias a la anomalía media, junto con la excéntrica se puede obtener el tiempo en órbita desde el periapsis, que se evaluará en el siguiente subapartado donde se explicará la ecuación de Kepler.

Por ello, entender todos estos parámetros será muy importante a la hora de planificar misiones de manera óptima y obtener las trayectorias deseadas con el objetivo de ser lo más eficientes desde el punto de vista energético o menores tiempos, alcanzando así

recorridos precisos a la vez que se consigue ahorrar combustible o menores tiempos de misión.

2.1.7. Tiempo de órbita desde el periapsis

Cuando Kepler se encontraba analizando las órbitas del sistema solar, observó que las áreas de la elipse y un círculo concéntrico de radio el semieje mayor de la elipse, son equivalentes por similitud.

Entonces, haciendo uso la segunda ley de Kepler, se puede relacionar el tiempo de órbita desde el periapsis con parámetros orbitales como el semieje mayor o la anomalía excéntrica (apoyarse en la figura 2.5 para las áreas).

Finalmente, desarrollando estos conceptos se acaba obteniendo la conocida ecuación de Kepler gracias a todo este desarrollo, junto con la tercera ley de Kepler (ver ecuación 2.13:

$$\boxed{2\pi\frac{t}{T} = \sqrt{\frac{GM}{a^3}}t = M = E - e\sin E} \tag{2.19}$$

2.1.8. Problema de Lambert. Tiempo de órbita entre dos puntos

Una vez que se conoce el tiempo desde el periapsis se procede a buscar conocer el tiempo entre dos puntos cualquiera de una órbita. Para ello, Johann Heinrich Lambert descubrió que el tiempo empleado a lo largo de una órbita entre dos puntos solo dependía de las distancias al foco de la órbita.

Este teorema, que se conoce como el teorema de Lambert, permite solucionar un gran número de problemas para cualquier órbita, siendo especialmente notorio en el caso de las órbitas de transferencia.

Suponiendo que un cuerpo se encuentra bajo la influencia de otro cuerpo másico principal y su fuerza gravitacional. Éste viaja siguiendo una trayectoria alrededor de ese astro central. En un instante concreto t_1, se encuentra en una posición P_1 dentro de la órbita inicial. A través de una transferencia, se llegará a otra posición P_2, en otro instante t_2.

Esa transferencia será una sección cónica. Para ello se tendrán las anteriores condiciones en cuenta, obteniéndose las posibles trayectorias que conectan estos dos puntos gracias al teorema de Lambert. Esos caminos podrán ser más largos o más cortos:

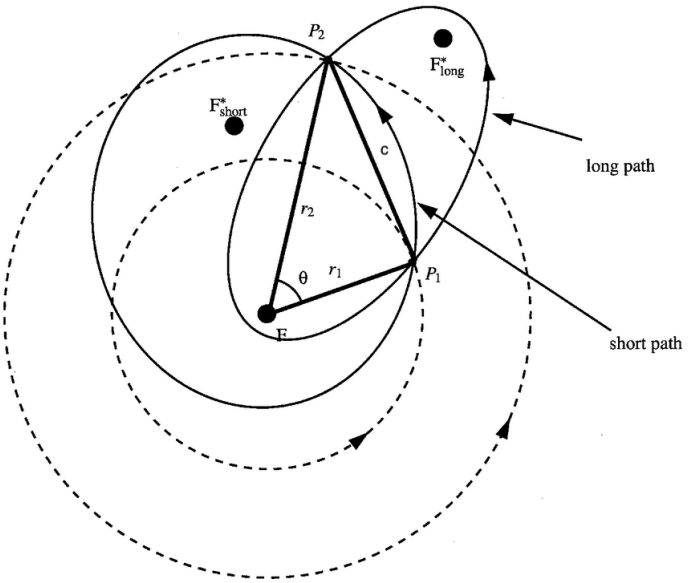


Figura 2.6: Posibles órbitas de transferencia entre dos puntos para el problema de Lambert

En resumen, este problema matemático se centra entonces en determinar la trayectoria óptima para la transferencia de un objeto desde un punto a otro en el espacio dentro de un intervalo de tiempo sometido a una fuerza gravitatoria de un cuerpo central. Además, se asume que se trabaja en el entorno del problema de los dos cuerpos, motivo por el cual solo se encuentra sometido a la fuerza gravitatoria del cuerpo central.

Los problemas que se pueden plantear es obtener ese foco ficticio de la órbita de transferencia. Para ellos hará uso de la geometría de este tipo de órbitas llegando a la conclusión de que ese foco ficticio se encuentra una distancia constante entre los dos puntos y, por tanto, es una hipérbola.

Para resolver el problema de Lambert, es necesario apoyarse en varias ecuaciones anteriormente desarrolladas, como la ecuación de Kepler (ecuación 2.19) o la ecuación de la anomalía excéntrica 2.16, además de las siguientes:

$$r\cos\nu = a(\cos E - e) \tag{2.20}$$

$$r\sin\nu = a\sqrt{1-e^2}\sin E \tag{2.21}$$

Desarrollando las ecuaciones (ver con más profundidad Vallado (2001) o Armano (2023)), se conseguirá definir el parámetro cuerda c:.

Por otro lado, se define el semiperímetro (s), que no será otro parámetro más que la suma de todos los lados del triángulo que forman el foco, P_1 y P_2, quedando las siguientes ecuaciones:

$$\boxed{s = \frac{r_1 + r_2 + c}{2}} \tag{2.22}$$

Quedando finalmente:

$$\boxed{\sqrt{\frac{GM}{a^3}}(t_2 - t_1) = (\alpha - \sin\alpha) - (\beta - \sin\beta)} \tag{2.23}$$

$$\boxed{\sqrt{\frac{s}{2a}} = \sin\frac{\alpha}{2}} \tag{2.24}$$

$$\boxed{\sqrt{\frac{s-c}{2a}} = \sin\frac{\beta}{2}} \tag{2.25}$$

En el caso de una órbita hiperbólica:

$$\boxed{\sqrt{-\frac{GM}{a^3}}(t_2 - t_1) = 2 \cdot [(\sinh\alpha - \alpha) - (\sinh\beta - \beta)]} \tag{2.26}$$

$$\boxed{\sqrt{-\frac{s}{2a}} = \sinh\frac{\alpha}{2}} \tag{2.27}$$

$$\boxed{\sqrt{-\frac{s-c}{2a}} = \sinh\frac{\beta}{2}} \tag{2.28}$$

Una vez definidas las ecuaciones básicas, se particularizarán para diferentes situaciones en los siguientes apartados:

Mínimo arco y mínimo tiempo de vuelo

Para alcanzar el mínimo arco y por lo tanto, el mínimo tiempo de vuelo, el a_{arcmin} es igual a la mitad del semiperímetro, lo que conllevará a que $\alpha_{arcmin} = \pi$. Con todo lo anterior, sustituyendo en la ecuación de Lambert (ecuación 2.26) para conocer el tiempo de vuelo (Δt_{arcmin}) para arco mínimo será:

$$4\sqrt{\frac{GM}{2s^3}}\Delta t_{arcmin} = \pi - (\beta_{arcmin} - \sin\beta_{arcmin}) \tag{2.29}$$

Resolución de la ecuación de Lambert

Resolver esta ecuación no es un problema analítico, por lo que un método numérico es la manera de enfocar este problema. Una posibilidad es el método de la bisección. Para ello se establecerán unas condiciones de iteración como $a_{low} = a_{arcmin} = \frac{s}{2}$ y $a_{high} = 2s$. Las ecuaciones finales con este método sería:

$$\mathbf{v}_1 \equiv \mathbf{v}(t_1) = (B + A)\mathbf{u}_c + (B - A)\mathbf{u}_1 \tag{2.30}$$

$$\mathbf{v}_2 \equiv \mathbf{v}(t_2) = (B + A)\mathbf{u}_c - (B - A)\mathbf{u}_2 \tag{2.31}$$

Donde los parámetros, A, B, $\mathbf{u}_c$, $\mathbf{u}_1$ y $\mathbf{u}_2$:

$$\mathbf{u}_c = \frac{\mathbf{r}_2 - \mathbf{r}_1}{c} \tag{2.32}$$

$$\mathbf{u}_1 = \frac{\mathbf{r}_1}{r_1} \tag{2.33}$$

$$\mathbf{u}_2 = \frac{\mathbf{r}_2}{r_2} \tag{2.34}$$

$$A = \sqrt{\frac{GM}{4a}} \cot\frac{\alpha}{2} \tag{2.35}$$

$$B = \sqrt{\frac{GM}{4a}} \cot\frac{\beta}{2} \tag{2.36}$$

Siendo $\mathbf{v}(t_1)$ una función de la órbita inicial. En el caso de la Tierra, sería la velocidad rotacional a una latitud concreta.

Pork-chop plots

Como parte de la búsqueda de encontrar el momento de lanzamiento óptimo y el momento de llegada al destino, los planificadores de misión tienen en cuenta el Problema de Alcance de Lambert (*LTP*). También, se busca el incremento de velocidad necesario para alcanzar este objetivo (ΔV).

Para responder a esta necesidad surgen los pork-chop plots, que son gráficas con forma de çhuletas de cerdo". El objetivo es representar curvas niveladas con el mismo nivel de energía característica, C_3, ante combinaciones de fechas de salida y de llegada al destino que se requiere.

La energía característica también se denomina energía hiperbólica, pudiendo definirse como $C_3 = 2\epsilon = v_\infty^2$. Se define como la medida del exceso de energía específica que se requiere para poder escapar mínimamente de un cuerpo, cómo puede ser el ΔV en lanzamiento.

Por ello, un pork-chop plot permitirá determinar cual es la ventana de lanzamiento óptima, que sea compatible con las características de la nave en órbita. Un contorno se denominará curva pork-chop, representando un C_3, donde se encontrará el C_3 óptimo en el centro de la curva.

Este tipo de simulaciones fueron empleadas en misiones reales como el programa Voyager, donde se obtuvieron más de 10000 pork-chop plots. A continuación se muestra un ejemplo de pork-chop plot:

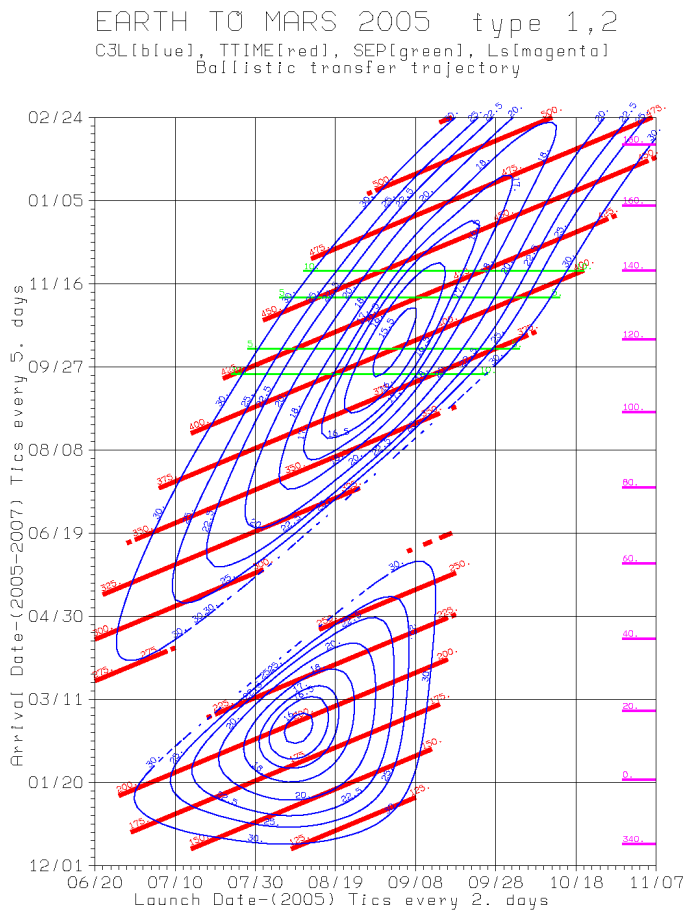


Figura 2.7: Pork-chop plot para la oportunidad de lanzamiento a Marte en 2005

Donde las líneas de color representan:

- Líneas azules: C_3, Δv en lanzamiento
- Líneas rojas: Duración de la misión
- Líneas verdes: Ángulo Sol-Tierra-Objeto (en el caso de que el Sol y la Tierra sean los astros principales). En el caso de que fuera demasiado pequeño, la señal del objeto puede verse afectada.
- Líneas magentas: Ángulo Tierra-Sol-Marte (en el caso de que el Sol, la Tierra y Marte sean los astros principales). Es el ángulo que se consigue al proyectar una línea desde la Tierra hasta el Sol y después hacia Marte.

Como puede observarse, entender el problema de Lambert y sus aplicaciones es esencial en el diseño de misiones interplanetarias. Su resolución es la piedra angular para garantizar misiones en las que se puedan optimizar variables importantes como mínimo consumo de combustible, mínimo incremento de velocidad o la mejor ventana de lanzamiento posible.

2.2. Análisis de misión

2.2.1. Maniobras orbitales

Las maniobras orbitales son esenciales para entender y realizar correctas planificaciones de misión, con el objetivo final de diseñar las trayectorias óptimas. es por ello, por lo que estas maniobras tendrán que englobar aquellas operaciones necesarias para alcanzar el objetivo. Algunas de ellas son transferencias entre órbitas, ajuste de órbitas o rendezvous", y para ello serán necesario aplicar diferentes impulsos de velocidad para lograr dicha operación.

Para poder realizar cualquier tipo de maniobra, se suele necesitar un impulso energético, que se traduce en un incremento de velocidad (ΔV), que se asume como instantáneo. Tiene como objetivo de conseguir cambiar la órbita en la que se encuentra por otra, cambiando alguno de los siguientes parámetros:

- Aumentar o disminuir el apoapsis o el periapsis
- Cambiar la inclinación de la órbita
- Escapar de la órbita
- Cambiar el periodo de la órbita
- Cambiar la longitud del nodo ascendente de la órbita
- Iniciar o finalizar una órbita de transferencia

La elección del momento para aplicar una maniobra es crítico ya que se deben de considerar las ventanas de lanzamiento, las posiciones de los astros o la posición del objeto en su órbita. todo ello ser necesario para la correcta y exitosa planificación de la misión.

Cabe destacar que existen dos tipos de cambio de órbita que afectan al plano orbital, maniobras coplanares y maniobras no coplanares. Las coplanares son aquellas que no cambiarán ni la inclinación (i) ni la longitud del nodo ascendente (Ω).

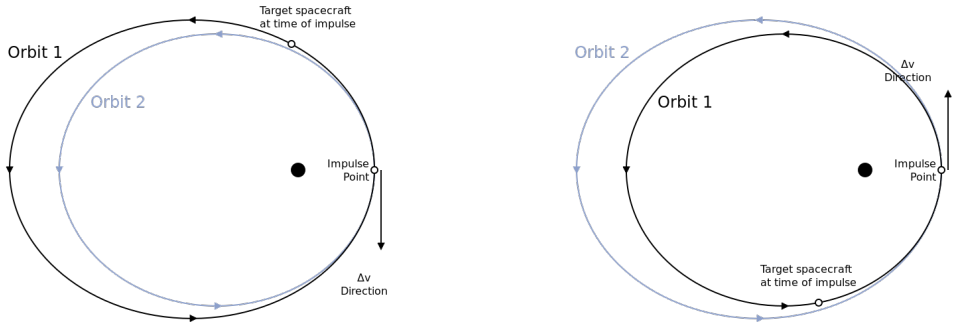
El momento óptimo para dar un impulso será en el apoapsis o el periapsis Para este tipo de maniobras, la ecuación 2.37 que se muestra a continuación es de gran ayuda:

$$\frac{v^2}{2} - \frac{GM}{r} = -\frac{GM}{2a} \tag{2.37}$$

A continuación, se exponen varios tipos de maniobras impulsivas orbitales:

Cambio del apoapsis o periapsis

Consiste en dar un impulso en el apoapsis o en el periapsis, para aumentar o disminuir la distancia correspondiente. En caso de querer aumentar el apoapsis o disminuirse, habrá que dar un impulso en el periapsis de la órbita y el sentido de la ΔV (prógrado si quiere aumentarse y retrógrado si quiere disminuirse). En caso de querer variar el periapsis, se dará un impulso en el apoapsis en el sentido adecuado.



(a) Disminución del periodo **(b)** Aumento del periodo

Figura 2.8: Cambio del periodo de la órbita modificando el apoapsis

Transferencias coplanares orbitales de dos impulsos

Este tipo de maniobras buscan hacer un cambio de órbita entre la inicial y la final. Para ello se calcula el punto de la órbita inicial que será desde el que partirá la órbita de transferencia, obteniéndose el ΔV para moverse a esa órbita de transferencia. Tras ello, se procede a calcular el punto de la órbita final con el que tendrá intersección el de la transferencia, consiguiendo también el ΔV para pasar de la de transferencia a la final.

Muchas maniobras de transferencia pueden obtenerse haciendo uso de esta técnica en función de lo que interese, como el tiempo de llegada y el rendezvous, característico en el problema de Lambert que se trató anteriormente. Otra posibilidad es la optimización de transferencias desde el punto de vista del coste de la energía cinética, ya que cuanto más cerca de la mínima energía, será mejor para la misión.

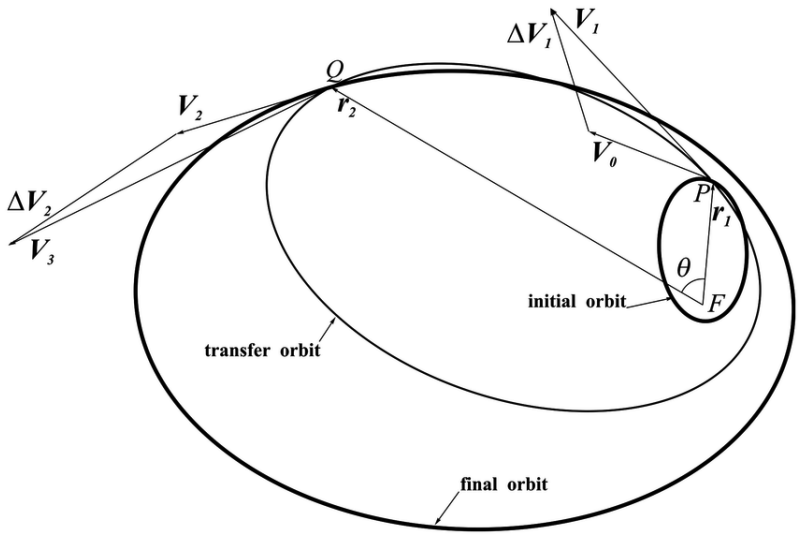


Figura 2.9: Esquema de una transferencia coplanar de dos impulsos

Transferencia de Hohmann

Esta maniobra es una de las más famosas de la mecánica orbital, ya que se caracteriza por ser un método de transferencia entre órbitas bastante eficiente. Consiste en una maniobra que cambiará la órbita de la nave espacial entre dos órbitas coplanares (como por

ejemplo un paso de órbita de la Tierra a otra de Marte) gracias a una semiórbita elíptica. Para ser realizada, serán necesarios, dos impulsos de velocidad, uno en la trayectoria inicial y otro en la trayectoria final.

Esta maniobra ha sido esencial en muchas misiones interplanetarias, ya que ha permitido minimizar la energía requerida para hacer cambios de trayectoria, además de permitir trayectorias precisas con un buen alineamiento con los astros de destino.

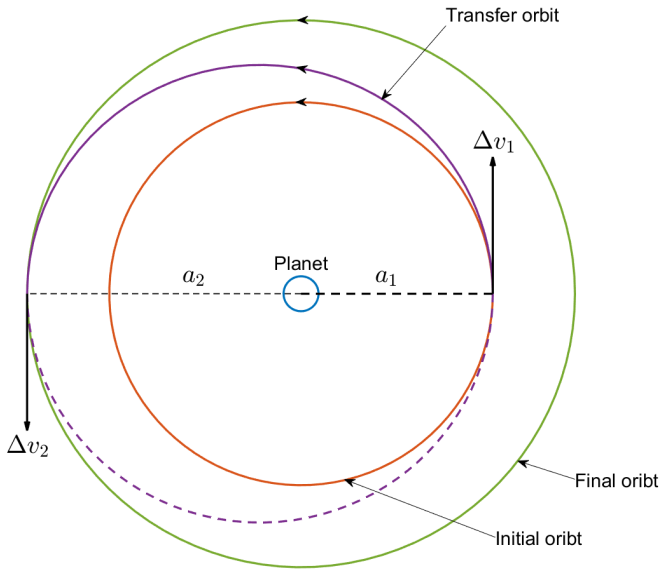


Figura 2.10: Transferencia de Hohmann

Transferencia biélíptica

La transferencia biélíptica es tipo de maniobra orbital, cuyo objetivo es similar al de la órbita Hohmann, pero haciendo uso de dos medias órbitas elípticas para completar la transferencia de órbita. Gracias a este tipo se consiguen menores incrementos de velocidad totales para alcanzar la órbita final, reduciendo así el combustible necesario, en determinadas circunstancias.

El primer incremento de velocidad se dará en la órbita inicial, una vez conseguida la primera órbita elíptica exterior, se dará otro incremento de velocidad en el apoapsis de la órbita elíptica exterior, consiguiendo así, la segunda orbita elíptica, cuyo periapsis coincidirá con el radio de la órbita final del objetivo. Por último, se dará un tercer impulso para conseguir la órbita deseada.

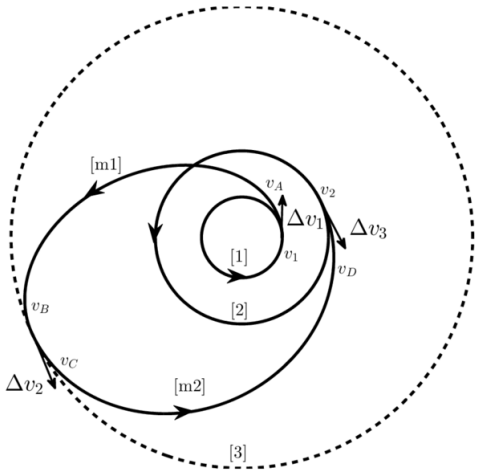


Figura 2.11: Transferencia bi-elíptica

Transferencia circular-elíptica y elíptica-elíptica

Este tipo de transferencias se dan cuando las dos órbitas (inicial y final) son elípticas o una de ellas es circular. Estas trayectorias son similares a las transferencias de Hohmann. Existen dos posibles trayectorias dependiendo de donde se aplique el impulso. La regla óptima será coger aquella donde está el mayor de los apogeo.

Maniobras de asistencia gravitatoria

Este tipo de maniobras se caracterizan por hacer uso de la interacción gravitatoria y el movimiento relativo con respecto a un astro, cambiando su trayectoria y modificando también la velocidad. Gracias a ellas es posible reducir el uso de combustible, suponiendo un gran avance para determinadas misiones espaciales, ya que permiten conseguir las trayectorias deseadas reduciendo el gasto de carburante.

En determinadas ocasiones, se suelen denominar como tirachinas (slingshots) gravitatorios o swing-bys. La deflexión provocada por la interacción gravitatoria con el astro en el que se apoya, provocará una nueva trayectoria que permita aumentar la velocidad para llegar rápidamente al destino de manera eficiente.

En la planificación de misiones, emplear este tipo de transferencias permite aumentar el alcance de las misiones, además de conseguir diversas trayectorias interplanetarias que permiten investigaciones científicas al observar varios astros que sean de interés.

Algunos ejemplos de este tipo de misiones son las famosas Voyager, Cassini, o New Horizons, entre otros como las ya comentadas en el *Capítulo 1*. En ellas se ha hecho uso de asistencias gravitatorias para explorar planetas u otros astros que se encuentran en lugares distantes dentro del sistema solar, siendo importante su uso para lograr estos objetivos.

En conclusión, para optimizar las trayectorias, las maniobras orbitales tienen un rol vital. Siendo aún más importantes, cuanto más larga es la misión, pues, supone un ahorro de combustible, además de reducir tiempos y llegar al destino con gran precisión. Siendo ese el motivo por el cual todos los conceptos comentados anteriormente son de extrema utilidad en cualquier misión espacial, además de otros factores como los que se explicarán en los siguientes subapartados.

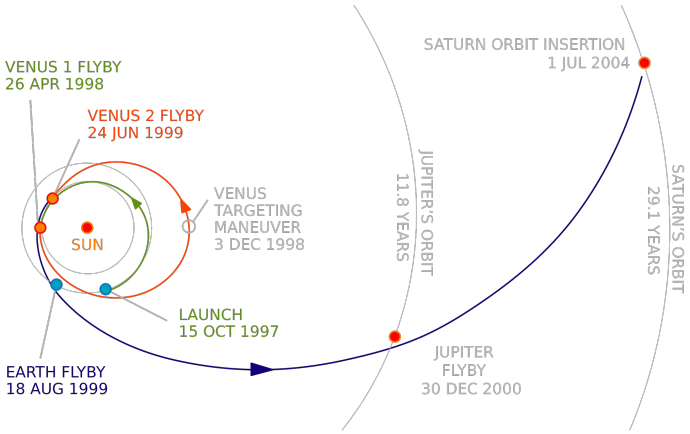


Figura 2.12: Misión interplanetaria de la sonda Cassini, que hace uso de asistencias gravitacionales

2.2.2. Perturbaciones orbitales

No solo los astros suponen una perturbación posible a considerar en la órbita del objeto. Existen muchos tipos de perturbaciones que se vuelven importantes a la hora de analizar su influencia en las órbitas de cualquier objeto. Además de las perturbaciones gravitatorias, destacan entre otras la presión solar o la resistencia aerodinámica. A continuación, se explicarán estos tres tipos de perturbaciones:

Perturbación gravitatoria

Este tipo de perturbación sucede cuando un astro, como puede ser un planeta, un satélite o un cuerpo de masa considerable, ejerce una fuerza gravitacional sobre un objeto que se encuentra orbitando alrededor de otro astro.

Estas perturbaciones provocan desviaciones respecto de la trayectorias perfectamente definidas que se podrían encontrar en un problema de dos cuerpos, afectando a la que será la trayectoria óptima e influyendo en el gasto de combustible o el tiempo de misión.

Presión solar

En nuestro sistema solar, el Sol ejerce una fuerza de presión debido a los fotones de su propia actividad como estrella. Esta presión solar puede alterar la trayectoria de un objeto en su órbita, ya que tiene pequeñas influencias al ser capaz de desviar la trayectoria. Se debe a esas pequeñas aceleraciones, que no se tienen en cuenta cuando se hace una simplificación en las primeras fases de las planificaciones de misión.

Resistencia aerodinámica

Por último, se encuentra la resistencia aerodinámica, ya que en alturas muy bajas en aquellos astros donde existe una atmósfera, el fluido que la compone genera una fuerza contra el avance. En el caso de las órbitas terrestres, este problema es bien conocido, ya que puede provocar que un objeto que se encuentra orbitando alrededor de la Tierra a cotas bajas, entre de nuevo al planeta de manera accidental si no era éste el propósito.

Impacto de las perturbaciones orbitales

El impacto de perturbaciones orbitales anteriormente mencionadas es diverso. Dentro de las posibles consecuencias de estos fenómenos se encuentran las siguientes:

- Precisión de la trayectoria: Las perturbaciones pueden introducir incertidumbre en la trayectoria del objeto. Por lo que entender y modelar estas perturbaciones permitirán trayectorias de mayor precisión.
- Navegación y corrección de la trayectoria: Las perturbaciones provocarán pequeños cambios en el curso de la órbita de la aeronave, siendo necesarios pequeños ajustes para mantener la trayectoria deseada.
- Misiones a largo plazo: Cuando una misión espacial tiene un largo periodo de tiempo de ejecución, corre el riesgo de que la suma de todas las perturbaciones provoque grandes cambios en su trayectoria, afectando a la llegada al destino final. Por ello, es muy importante el uso de modelos precisos para considerar este tipo de desviaciones.
- Observaciones científicas: En el caso de las misiones científicas cualquier perturbación de las anteriormente comentadas puede afectar tanto a la posición como al tiempo de los instrumentos que se usan con esa labor científica, siendo nuevamente importante considerar estos fenómenos para optimizar la recogida de datos.

Todo ello sumado muestra la importancia de este tipo de perturbaciones y cómo afectará a las órbitas. Por ello, en la planificación de misión en las primeras etapas no se considerarán, pero se volverá mucho más importante según se vaya avanzando en la definición de la misión.

2.2.3. Esfera de influencia

En cualquier trayectoria simple (problema de los dos cuerpos), rápidamente se convierte en una trayectoria más compleja, con al menos tres cuerpos entrando en juego. Ese es el caso de una misión como la trayectoria Tierra-Luna, en la que se puede ver pronto que se tienen dos grandes astros que influirán en la trayectoria.

En el caso de una misión interplanetaria, el Sol es el astro dominante siempre. Las trayectorias fuera del planeta en este caso suelen ser ramas de elipses. Mientras que cerca de un planeta es una rama hiperbólica salvo que dicha trayectoria corrija a una elipse.

Por ello el diseño de la misión acabará dividiéndose en dos segmentos. Los problemas de dos cuerpos entre el vector y el cuerpo masivo más relevante, mientras se trata al siguiente cuerpo, el tercero (si hubiera más astros sería el siguiente en orden de preferencia por tamaño), como si resultara en una perturbación.

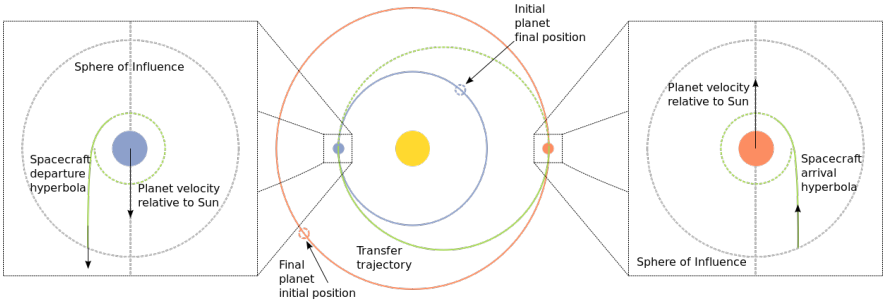


Figura 2.13: Esferas de influencia

2.2.4. Aproximación por secciones cónicas (Patched Conics)

Gracias al uso de esferas de influencia, se pueden subdividir los problemas de múltiples cuerpos en una serie de espacios contiguos donde cada uno de ellos se caracteriza por resolver un problema de 2 cuerpos. Debido a que la solución de este tipo de problemas se caracteriza por ser siempre una sección cónica, este método se denomina aproximación por secciones cónicas o, su traducción literal del inglés, cónicas parcheadas.

Cada sección puede aproximarse por una cónica, una elipse, una parábola o una hipérbola, conocidas gracias a las órbitas de Kepler. Estas secciones son denominadas secciones o parches ("patch").

En cada punto de transición o de intercambio entre secciones, se encuentra un cambio de esfera de influencia, por lo que la trayectoria se representa a través de una cónica diferente. Este tipo de transiciones simplifican el problema, ya que reduce las complejas interacciones gravitatorias, tratando el problema como un problema de dos cuerpos. En cada sección se aplicarán las leyes de Kepler y la conservación de la energía, lo que permite determinar las características de la trayectoria, como pueden ser parámetros como el semieje mayor (a), la excentricidad (e) o la anomalía verdadera (ν).

Este método es especialmente útil cuando la planificación de misión se encuentra en sus primeras fases. Ofrece un enfoque simplificado para alcanzar los sitios de misión y comenzar a realizar estimaciones de duración de misión, además de ventanas de oportunidades e incrementos de velocidades necesarios.

También resulta muy útil para el estudio de maniobras de aproximación o de encuentro, fly-bys o rendezvous. Esto permite los planificadores determinar cuáles son las trayectorias más óptimas de acercamiento para cumplir con los objetivos de la misión, como pueden ser la investigación científica y la acumulación de datos.

También este método permite una representación visual de la trayectoria de la aeronave en su órbita, lo que permite ayudar a la hora de conocer como la nave, interactuará con el resto de cuerpos celestiales de su entorno a lo largo del viaje, facilitando la planificación de la misión.

Por último, gracias a la simplicidad de este método, los planificadores pueden realizar ajustes basados en las aproximaciones de los aumentos de trayectoria, lo que les ayuda a refinar la trayectoria, según va progresando la misión. a pesar de que es un método simple, se convierte en una herramienta esencial para el entendimiento de las misiones. Por ello, los siguientes apartados tratarán de ajustar estas aproximaciones para conseguir las trayectorias más detalladas y precisas posibles.

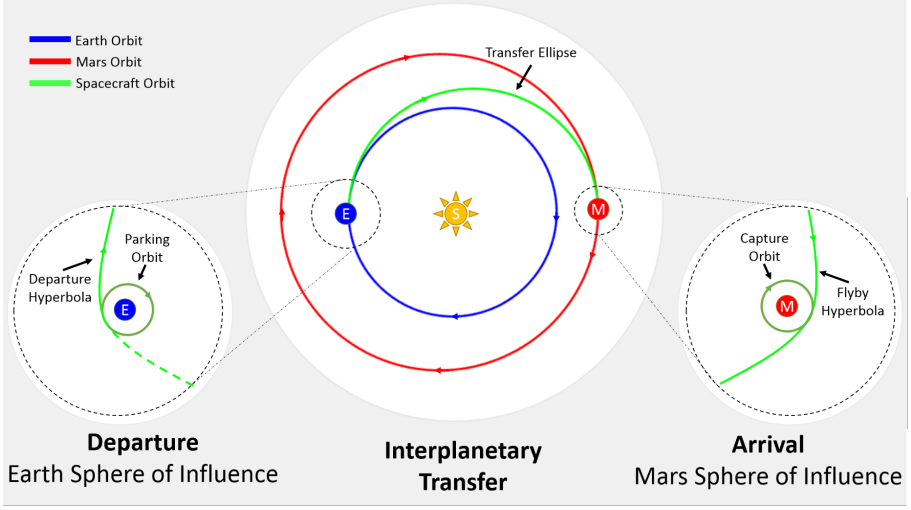


Figura 2.14: Aproximación por secciones cónicas (Patched Conics)

2.3. Algoritmos

Para la optimización de las misiones interplanetarias, es necesario planificar y ejecutar las maniobras de manera adecuada, ya que cualquier cambio en ellas puede afectar a cumplir finalmente con los propósitos que se requieren de estas misiones. Para poder tener en cuenta estas interacciones gravitatorias, además de otras posibilidades, como puede ser las fechas de lanzamiento, es necesario el uso de algoritmos que permitan optimizar las trayectorias interplanetarias.

Gracias a estos algoritmos se pueden minimizar el consumo de combustible o reducir los tiempos de vuelo entre los diferentes planetas. Para poder llegar a ellos serán necesario el uso de poderosas herramientas computacionales, además de técnicas que puedan contemplar la complejidad de la mecánica orbital. por ello, seleccionar o diseñar los algoritmos para este tipo de aplicaciones debe de tener en cuenta algunos factores como son:

- Precisión: Los algoritmos tienen que ser capaces de reflejar con precisión la mecánica orbital en cualquier misión interplanetaria.
- Eficientes: Los algoritmos deben ser capaces de proporcionar soluciones en el tiempo menor posible, ya que puede ser necesario en momentos críticos de la misión o de la planificación.
- Robustez: Los algoritmos tienen que ser robustos en caso de posibles incertidumbres o variaciones en los parámetros de la misión, adaptándose así a la mecánica orbital real de cada momento de la trayectoria.
- Adaptabilidad: Los algoritmos tienen que ser capaces de adaptarse a entornos cambiantes a lo largo de la misión.

Por ello, en este apartado se centrará en describir los algoritmos que pueden ser empleados junto a sus características, como las funciones objetivos que pueden tener o los tipos de algoritmos útiles, a la hora de optimizar nuevas misiones interplanetarias.

2.3.1. Función objetivo

En una optimización siempre hay que definir qué parámetro o parámetros se van a optimizar y, por tanto, la función objetivo asociada para conseguir la trayectoria óptima deseada. Estas funciones se centrarán en definir los objetivos y restricciones de la misión, incluyendo criterios como el consumo de combustible, la duración de la misión, el incremento total de velocidad o la masa de la nave espacial.

1. Consumo de combustible: Uno de los objetivos de estas funciones suele ser minimizar el consumo de combustible ya que reduce el coste de misión y la masa de la nave espacial, suponiendo una mejora en la viabilidad en la misión.
2. Duración de la misión: Otro de los objetivos suele ser la reducción del tiempo de vuelo hasta alcanzar el destino. Por ello, una función objetivo puede tratar de minimizar este aspecto respetando cualquier restricción previamente definida.
3. Incremento total de la velocidad: La velocidad necesaria por la aeronave para realizar cambios de trayectorias dependerá del tipo de maniobras que se empleen y las distancias que sean necesarias recorrer. Por ello, esta característica suele ser importante en las funciones objetivo ya que reduciendo el incremento total de la velocidad suele afectar también a una reducción del consumo de combustible.
4. Masa de la nave: Este aspecto es crítico en cualquier misión ya que a menor peso que tenga el cohete, menor uso de combustible será necesario para conseguir los mismos impulsos y mejorando la viabilidad de la misión.

Todo lo anterior implica la complejidad e importancia de formular adecuadamente la función objetivo para conseguir que los algoritmos de optimización den con la mejor trayectoria posible para una misión interplanetaria teniendo en cuenta las posibles necesidades y restricciones del diseño de la misión.

2.3.2. Técnicas de optimización

Para lograr trayectorias óptimas será necesario encontrar la mejor solución o las mejores soluciones dentro de una optimización. Para ello es necesario definir cuál será la técnica empleada pues dependiendo de la técnica podrán conseguirse mejores resultados o resultados más rápidos.

Existe un amplio rango de técnicas posibles que van desde algoritmos de optimización clásicos hasta algoritmos heurísticos y evolutivos, que se inspiran en procesos naturales, como son las siguientes:

- Optimización matemática: Este tipo de técnica se centra en aprovechar modelos matemáticos, que buscan sistemáticamente soluciones óptimas y pueden ser de diferentes tipos como programaciones lineales, no lineales, dinámicas, etc.
- Optimización heurística: Este tipo de algoritmos se centran en tener enfoques flexibles cables que permiten explorar soluciones complejas de grandes dimensiones y relaciones no lineales, como puede ser algoritmos genéticos, optimización de enjambre de partículas o de colonia de hormigas. tratan de imitar procesos naturales o estrategia de resolución para mejorar interactivamente las soluciones candidatas,

consiguiendo así soluciones, casi óptimas con recursos computacionalmente razonables.

- Optimización multiobjetivo: Esta técnica permite optimizar de manera simultánea múltiples objetivos, como podría ser la duración de la misión y el incremento de velocidad total de la misma. estas técnicas hacen uso de los principios de optimización de Pareto, con las que identificar soluciones de compromiso entre diferentes objetivos contrapuestos.
- Optimización metaheurística: aquellos algoritmos de optimización metaheurística, se caracterizan por ofrecer enfoques robustos y adaptables a problemas de optimización con condiciones inciertas o dinámicas, como son algoritmos evolutivos o de enjambre. Aprovechan estrategias de búsqueda estocástica para alcanzar soluciones eficientemente con entornos y restricciones cambiantes.

Para elegir una técnica u otra, será necesario tener en cuenta cuál es la finalidad del código empleado y de los algoritmos que se pueden considerar óptimos, dependiendo de la fase de diseño de misión en la que se encuentre.

2.3.3. Librerías con algoritmos de optimización

Dependiendo del lenguaje de programación que se vaya a usar, pueden existir diferentes algoritmos de optimización ya programados. Uno de ellos enfocado para Python es el desarrollado por la Agencia Espacial Europea (ESA) denominado `PyGMO` (Python Parallel Global Multiobjective Optimizer).

Esta librería, de la que se hablará más adelante detenidamente, esta desarrollada por un equipo personal, enfocado claramente a las trayectorias espaciales. Gracias a una herramienta tan potente se pueden resolver, problemas complejos de optimización y alcanzar trayectorias eficientes para una misión concreta.

`PyGMO` ofrece diferentes algoritmos de optimización que permiten obtener trayectorias interplanetarias aprovechando características como la capacidad de computación paralela y la heurística de optimización, consiguiendo explorar grandes espacios de soluciones. Dentro de las principales características y ventajas de `PyGMO` se encuentran:

1. Computación paralela: La librería aprovecha la potencia computacional de arquitectura paralelas que permiten explorar escenarios de manera eficiente, haciendo uso de varios núcleos de computación, escalando, así cuando son necesarias altas demandas computacionales.
2. Algoritmos personalizables: `PyGMO` permite personalizar implementar diferentes algoritmos de optimización, que se adapten a los requisitos que se desean para cada problema. esto lo hace muy útil para cualquier tipo de problema, que sea necesario optimizar, no solo problemas de trayectorias óptimas.
3. Optimización multiobjetivo: el hecho de poder soportar una optimización multiobjetivo permite que `PyGMO` tenga una característica muy importante para una optimización de trayectorias, ya que hace posible encontrar soluciones que satisfagan dos posibles características críticas de una misión espacial, como son el tiempo de vuelo y el incremento de velocidad total.

4. Integración con el ecosistema Python: otra de las ventajas que tiene esta librería es que está construida en el lenguaje Python y le permite interaccionar con otras librerías muy importantes como son `Pykep`, `NumPy`, `SciPy` o `Astropy`. Cabe destacar su integración con `Pykep` ya que es una librería también desarrollada por la ESA y centrada en optimización de trayectorias espaciales.

Por tanto, las ventajas de esta librería permiten hacer uso de algoritmos de optimización, claramente pensados para misiones interplanetarias, ya programados y listos para su uso, junto a una librería como `Pykep` desarrolladas ambas conjuntamente por la ESA.

2.3.4. Integración con PyGMO

Cuando se habla de integración dentro de optimización, se debe a la importancia que desempeñan los algoritmos a la hora de planificar misiones interplanetarias dentro del proceso de la simulación de trayectorias. Por ello, librerías como `PyGMO` tienen un papel crucial para conseguir los resultados de la optimización y permitir a los equipos valorar la viabilidad de la misión en determinadas condiciones:

- Integración de `PyGMO`: Gracias a la flexibilidad de la librería, permite a los usuarios que puedan aprovechar los algoritmos de optimización en busca de trayectorias óptimas para misiones espaciales.
- Generación de escenarios de misión: El uso de una librería como `PyGMO` permite a los equipos planificar diversos escenarios de misión como la fecha de lanzamiento, especificando así diferentes parámetros de simulación y restricciones.
- Optimización de trayectorias: `PyGMO` fue construido por la propia ESA al igual que `Pykep` permitiendo a ambas librerías trabajar en la optimización de trayectorias de misiones espaciales, dando la posibilidad de valorar los resultados de la misma.
- Evaluación del rendimiento: Gracias al uso de `PyGMO` se hace posible la evaluación cuantitativa de los resultados obtenidos por la optimización y simulación, gracias a las herramientas de análisis y las capacidades de visualización integradas en `PyGMO` junto con `Pykep`.

Debido a esta sincronización de `PyGMO` junto a `Pykep` se puede observar una perfecta integración para el desarrollo de misiones interplanetarias óptimas.

2.3.5. Algoritmos de PyGMO para Pykep

Dentro de `PyGMO` existe la posibilidad de importar diversos algoritmos para resolver optimizaciones. Los creadores de la librería recomiendan varios algoritmos cuando se usa conjuntamente con `Pykep`, algunos de ellos incluidos internamente en `PyGMO` y otros externos. Los tres más aconsejados son los siguientes:

1. SNOPT: Es un paquete de software pensado para resolución de problemas de optimización a gran escala (lineales y no lineales). SNOPT se caracteriza por ser muy eficaz para problemas no lineales cuyas funciones y gradientes son costosos de evaluar. Es externa a `PyGMO` y desarrollada en C++.

2. NLOPT: Este algoritmo se encuentra disponible de manera "open-source´´y ofrece diferentes rutinas de optimización no lineales. Está implementada en `PyGMO` y admite su uso en varios lenguajes, incluyendo Python.
3. IPOPT: Esta librería de optimización sirve para optimizar problemas a gran escala no lineales. Es externa a `PyGMO` y desarrollada en C y Fortran.

Cabe destacar que a pesar de que SNOPT es la más aconsejada, requiere de una instalación compleja, haciendo que NLOPT sea más fácil para su uso en primeras instancias.

Capítulo 3

Librerías de software empleadas

Como se explica anteriormente para desarrollar este tipo de cálculos es necesario crear un código propio. Debido a la existencia de dos librerías tan importantes como son `Pykep` y `PyGMO`, siendo adaptadas a Python, hace el lenguaje usado para el código haya sido este mismo.

Por ello, se hace uso de ambas librerías, además de otras, que permitirán el almacenaje de datos, exportación de resultados o el diseño de una interfaz gráfica que haga más fácil el uso.

A continuación se explican de manera introductoria las principales librerías empleadas:

1. **Pykep**: `Pykep` es una biblioteca científica que ofrece herramientas básicas para la investigación en astrodinámica. Está escrita en C++ y expuesta a Python, y su propósito es la implementación de un solver" eficiente para resolver el problema de Lambert de múltiples revoluciones, además de órbitas que representan métodos directos (Sims-Flanagan), indirectos (Pontryagin) y híbridos para representar problemas de optimización con empujes bajos (low-thrust"), entre otros.
2. **PyGMO**: `PyGMO` es una biblioteca científica diseñada para facilitar la distribución de tareas masivas de optimización en múltiples CPUs. En el núcleo de `PyGMO` (Parallel Global Multiobjective Optimizer) se encuentra un paradigma innovador llamado "modelo de islas generalizado"para la paralelización a gran escala de algoritmos de optimización.
3. **PyQT5**: La librería `PyQT` permite crear interfaces gráficas de usuario (GUI, Graphic User Interface) en Python, funcionando similar a Tkinter, otra librería de GUI. Su gran ventaja es hacer uso de la herramienta QtDesigner, que permite importar la interfaz gráfica desarrollada de manera externa y relacionar sus variables dentro de un código en Python.
4. **Pandas**: `Pandas` es otra de las librerías importantes de Python ya que fue diseñada específicamente para poder manipular y analizar datos. Permite gestionar estructuras de datos y funciones que manejen tablas y series, de manera parecida a Excel dentro de Python.
5. **itertools**: El módulo de `itertools` implementa bloques de construcción de iteradores que permite a Python aumentar las posibilidades de iteración, consiguiendo estandarizar un conjunto central de herramientas rápidas y eficientes en términos

de memoria. En concreto, la herramienta utilizada es `product` permite realizar combinatoria, lo que permitirá iterar de manera adecuada las posibles secuencias.

6. **os**: Este módulo permite hacer uso de métodos para interactuar con el sistema operativo en Python. Así se consiguen realizar diversas tareas, como la creación y gestión de archivos y directorios, la entrada y salida de datos, la gestión de variables de entorno y la administración de procesos, entre otras funciones.
7. **matplotlib**: Es una librería que permite crear visualizaciones interactivas, animadas o estáticas en Python, consiguiendo personalizar los datos obtenidos y que se quieran mostrar haciendo uso de este lenguaje.

3.1. Pykep

Como se explicó anteriormente, `Pykep` es una biblioteca científica que fue escrita en lenguaje C++ pero que se encuentra aplicada en Python. Gracias a esta librería se pueden resolver problemas de trayectorias interplanetarias como el problema de Lambert u optimizaciones haciendo uso de empujes bajos o asistencias gravitatorias.

Dentro de la librería se disponen de varios módulos que harán los cálculos de las trayectorias. Gracias a ellos se pueden resolver dos tipos de problemas, los problemas genéricos y los específicos de los Temas de Búsqueda Avanzada (a partir de ahora, Advanced Research Topics).

3.1.1. Problemas resueltos

Problemas genéricos

Existen diversos ejemplos de problemas genéricos que pueden resolverse con esta librería. Algunos de los más importantes son los siguientes:

- Transferencia interplanetaria de múltiples impulsos: Gracias al uso de la clase `pl2pl_N_impulses`, que se encuentra contenida en el módulo `trajopt`. Esta clase permite resolver un problema de trayectoria interplanetaria haciendo uso de múltiples impulsos. El número de impulsos óptimo dependerá del planeta de destino y del planeta inicial. Con `PyGMO` se buscará la trayectoria óptima en función del incremento de velocidad, estableciéndose de esa manera la fecha de salida y llegada de la trayectoria.
- Problema de Lambert de revolución múltiple: Dentro de la librería de `Pykep` se implementó el algoritmo para la resolución del problema de Lambert, pudiendo obtenerse para diversas trayectorias teniendo en cuenta que se puede cambiar el número de revoluciones máximas, representando las diferencias entre cada número de revoluciones.
- Estudio de los parámetros orbitales: `Pykep` permite obtener los elementos kepleriannos (el semieje mayor, la excentricidad, la inclinación, la ascensión recta del nodo ascendente, el argumento del periastro y la anomalía verdadera) y los elementos equinocciales (p, f, g, h, k, L).

- Uso de SPICE: SPICE es un acrónimo inglés para Spacecraft, Planet, Instrument, C-matrix y Events, es decir, en castellano sería, Aeronave, Planeta, Instrumento, C-Matriz y eventos. En definitiva, es un sistema de geometría de observación para misiones científicas espaciales desarrollado por el Servicio de Navegación e Información Auxiliar (NAIF) de la NASA. En él, las misiones de vuelo almacenan información sobre la posición y orientación de la nave espacial, indexada por tiempo, en archivos de datos que serán los núcleos (kernels) SPICE. NAIF proporciona kernels multi-misión adicionales con datos de posición, orientación y forma/tamaño para los cuerpos del sistema solar. Se pueden utilizar un conjunto de núcleos gracias al de APIs que conectarán con la biblioteca SPICE.

Advanced Research Topics

Otro tipo de problemas que se pueden resolver, gracias a la librería de `Pykep`, son problemas de optimización de trayectorias, ya sean interplanetarias o entre otro tipo de astros. Para poder llevar a cabo esta optimización, será necesario el uso de otra librería desarrollada por la ESA y mencionada anteriormente, `PyGMO`.

En este apartado se explicarán algunos ejemplos de esos problemas, que permitirán optimizar las trayectorias entre planetas:

- Optimización global de una trayectoria con transferencia gravitatoria múltiple de bajo empuje: Gracias al uso de las dos librerías `Pykep`/`PyGMO` cuyo objetivo es realizar una optimización global de una trayectoria interplanetaria de múltiples tramos a lo largo de grandes ventanas de lanzamiento. Por lo tanto, se realizará una transferencia entre un planeta de origen y un planeta destino haciendo uso de fly-bys intermedios para conseguir la solución optimizada al final.
- Optimización global de una trayectoria de asistencia gravitatoria múltiple con una maniobra en el espacio profundo por tramo: Haciendo uso de la función específica para ello dentro del módulo planetario de `Pykep` junto con la optimización de `PyGMO`. Solo se permite una maniobra de espacio profundo (DSM), qué es el nombre de las maniobras cuando se encuentran lejos de la Tierra. En la optimización se tendrá un objetivo único que será el incremento de velocidad total (ΔV) o multiobjetivo (ΔV y tiempo de vuelo).
- Optimización de masa al realizar un rendezvous en Marte mediante un método directo: haciendo uso de la clase del módulo de trayectorias, que se centra en una trayectoria directa entre dos planetas, `pykep.trajopt.direct_pl2pl`, que utilizando un enfoque Sims-Flanagan, permite obtener la masa óptima con la que llegaría la nave en una maniobra de rendezvous en Marte.

Para resolver este tipo de problemas y poder optimizar trayectorias interplanetarias, cabe destacar cuáles son los principales módulos de `Pykep`, que junto con la librería `PyGMO`, permitirán llegar a soluciones óptimas en función de los input que decida el usuario:

3.1.2. Módulos y funciones importantes

En los siguientes subapartados se explicarán aquellos módulos y aquellas funciones más importantes:

Módulo principal (core)

El módulo central (`core`) contiene aquellas clases y funciones que serán necesarias para poder realizar los cálculos básicos de mecánica orbital.

Existen diversas funciones o clases dentro de este módulo principal, siendo algunas más relevantes en función del tipo de cálculos orbitales que se quieran llevar a cabo. Algunas de ellas se describen brevemente a continuación:

- `epoch`: Esta clase define un momento preciso en el tiempo, siendo compatible con el formato de fecha juliana.
- `lambert_problem`: Esta clase resuelve un problema de Lambert de múltiples revoluciones. Tiene diferentes posibles argumentos de entrada como el radio de la órbita inicial, el radio de la órbita final, el tiempo de vuelo, el número de revoluciones, etc.
- `ic2par`: Esta función haciendo uso de las posiciones y velocidades cartesianas devuelve los elementos keplerianos osculantes, es decir, a (semieje mayor), e (excentricidad), i (inclinación), Ω (ascensión recta del nodo ascendente), ω (argumento del periastro), E (anomalía excéntrica).

Módulo de optimización de trayectorias (trajopt)

Este módulo `trajopt` contiene clases cuyo objetivo es dar apoyo en los cálculos para obtener trayectorias orbitales óptimas entre planetas. Esto se debe a que este módulo es totalmente compatible con la librería `PyGMO`, desarrollada también por la ESA.

Por lo tanto, las funciones que se verán aquí servirán para alcanzar dichas optimizaciones en función de una serie de condiciones y características que definirán el tipo de trayectorias que se lleven a cabo.

Algunas de las clases más útiles de este módulo son las siguientes:

- `trajopt.mga`: Esta clase resuelve trayectorias que contienen asistencias gravitatorias múltiples (MGA). Esta clase en concreto no tendrá maniobras de espacio profundo. Además puede definirse como un problema de optimización UDP de `PyGMO`, que se explicará en el apartado correspondiente más adelante.
- `trajopt.mga_1dsm`: Esta clase resuelve trayectorias que contienen asistencias gravitatorias múltiples (MGA) con una maniobra de espacio profundo por cada tramo de la trayectoria. Además puede definirse como un problema de optimización UDP de `PyGMO`, que se explicará en el apartado correspondiente más adelante.
- `trajopt.pl2pl_N_impulses`: Esta clase es un problema que puede resolverse haciendo uso de `PyGMO` y representa una transferencia de un solo tramo entre dos planetas que permite hasta un número máximo de maniobras impulsivas en el espacio profundo.

Módulo de representación de trayectorias (orbit plotting)

Este módulo permitirá la representación de las trayectorias. Dentro de las funciones más destacables de este módulo se encuentran las siguientes:

- `orbit_plots.plot_planet`: Esta función dibujará la posición del planeta y su órbita en función de los parámetros de entrada.

- `orbit_plots.plot_lambert`: Esta función dibuja de manera particular un problema de Lambert en función de los parámetros de entrada.
- `orbit_plots.plot_kepler`: Esta función dibuja de manera particular una propagación bajo condiciones keplerianas en función de los parámetros de entrada.

3.1.3. Aplicación de Pykep

El problema de Lambert de revoluciones múltiples se encuentra contemplado en **Pykep**. En esta ocasión, se define una trayectoria entre la Tierra y Marte, donde se contemplan todas las posibles soluciones resultantes al problema de Lambert:

```
# Imports
import pykep as pk
from pykep.orbit_plots import plot_planet, plot_lambert
from pykep import AU, DAY2SEC
import pygmo as pg
import numpy as np

# Plotting imports
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# We define the Lambert problem
t1 = pk.epoch(0)
t2 = pk.epoch(640)
dt = (t2.mjd2000 - t1.mjd2000) * DAY2SEC

earth = pk.planet.jpl_lp('earth')
rE, vE = earth.eph(t1)

mars = pk.planet.jpl_lp('mars')
rM, vM = mars.eph(t2)

# We solve the Lambert problem
l = pk.lambert_problem(r1 = rE, r2 = rM, tof = dt, mu = pk.MU_SUN, max_revs=2)

# We plot
mpl.rcParams['legend.fontsize'] = 10

# Create the figure and axis
fig = plt.figure(figsize = (16,5))
ax1 = fig.add_subplot(1, 3, 1, projection='3d')
ax1.scatter([0], [0], [0], color=['y'])

ax2 = fig.add_subplot(1, 3, 2, projection='3d')
ax2.scatter([0], [0], [0], color=['y'])
ax2.view_init(90, 0)

ax3 = fig.add_subplot(1, 3, 3, projection='3d')
ax3.scatter([0], [0], [0], color=['y'])
ax3.view_init(0,0)

for ax in [ax1, ax2, ax3]:
    # Plot the planet orbits
    plot_planet(earth, t0=t1, color=(0.8, 0.8, 1), legend=True, units=AU, axes=ax)
```

```
46      plot_planet(mars, t0=t2, color=(0.8, 0.8, 1), legend=True, units=AU, axes=ax)
47      # Plot the Lambert solutions
48      axis = plot_lambert(l, color='b', legend=True, units=AU, axes=ax)
49      axis = plot_lambert(l, sol=1, color='g', legend=True, units=AU, axes=ax)
50      axis = plot_lambert(l, sol=2, color='g', legend=True, units=AU, axes=ax)
```

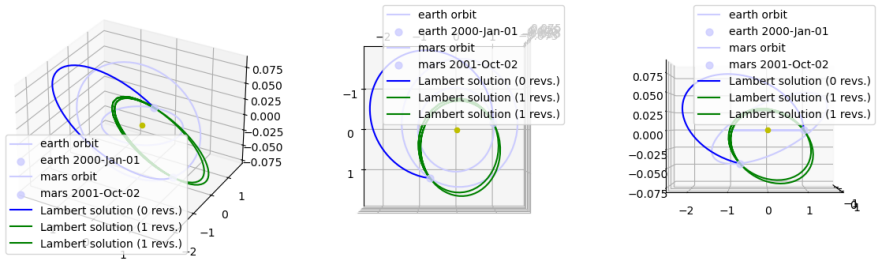


Figura 3.1: Resultado del problema de Lambert de revoluciones múltiples entre Tierra y Marte

Cómo se puede ver gracias este código se hace uso principalmente de un problema de Lambert de múltiples revoluciones `lambert_problem`. Finalmente, se representan las posibles soluciones dentro del intervalo, dado en función de las revoluciones. hay que tener en cuenta que se definirá un tiempo de vuelo, `tof`, que podría cambiar la solución representada si cambiarán los momentos de inicio y final de vuelo.

3.2. PyGMO

Esta librería científica desarrollada para Python se caracteriza en la optimización paralela masiva. La intención de esta librería es unificar a través de una interfaz algoritmo y problemas de optimización para su despliegue en entornos con diversos objetivos en paralelo, usando el concepto del paradigma del modelo de isla generalizado (`generalized island-model`), siendo cada isla realmente un núcleo de la CPU que se asigna para resolver el problema.

La librería se encuentra interconectada con diferentes algoritmos de optimización como aquellos provenientes de otra librería de Python como es SciPy, NLOPT, SNOPT, IPOPT, etc.

Estos últimos algoritmos de optimización serán los empleados para poder obtener aquellas trayectorias más adecuadas que se estudian en las misiones interplanetarias de esta investigación.

3.2.1. Módulos y funciones importantes

Dentro de esta librería, se definen diferentes clases, que serán muy útiles a la hora de optimizar y se explicarán en los siguientes subapartados:

Clase problema (problem)

Esta clase se centrará en representar y caracterizar un problema genérico, que posteriormente se vaya optimizar. Para ello, el problema será definido por el usuario o el programador. Este motivo es el que determina el nombre de UDP.

Dentro de esta clase se establecerán múltiples funciones, cuyo objetivo será establecer por ejemplo la tolerancia que tendrán todas las restricciones y que, por defecto, se encuentran en cero. La función que permite cambiar dicha tolerancia es `c_tol`.

Por ello, las condiciones y funciones más importantes para definir un problema de optimización son las siguientes:

- Propiedad `c_tol`: Esta característica contiene un array de float utilizado para comprobar si las restricciones aplicadas son viables
- `fitness()`: Esta función llamará al método `fitness()` de la UDP con el objetivo de obtener el vector decisión de entrada dv.
- `gradient()`: Esta función permite obtener el gradiente de la función del problema para poder ser optimizada en caso de que no tenga gradiente asociado.
- `get_bounds()`: devuelve los límites establecidos en el problema

Clase población (population)

Esta clase se centrará en almacenar todas aquellas soluciones del problema optimizado que se consideren candidatos, a veces denominados también individuos. Para ello contendrá un problema, un número de vectores de decisión (llamados también cromosomas) y vectores de aptitud. estarán asociados a un objeto único para poder ser seguido de manera adecuada.

Estos candidatos están determinados por las siguientes características:

1. ID único usado para seguir al candidato a través de generaciones y migraciones dentro de la optimización.
2. Cromosoma (vector de decisión).
3. Aptitud del cromosoma (vector “fitness”), es decir, objetivos, restricciones, etc.

Dentro de la población se rastreará cuál es el mejor individuo y ese será el campeón (*champion)*. Ese campeón se actualizará de manera constante, según surjan individuos con mejores resultados aunque no es necesariamente un individuo que pertenezca actualmente en la población.

El campeón sólo se define y es accesible dentro de la interfaz de la población cuando el problema que está contenido en la población es de objetivo único. El campeón será accesible a través de las propiedades de la clase `population`.

Para todo ello serán necesarias las siguientes funciones o propiedades, definiendo previamente el problema de la población (`prob`) y el tamaño de la misma (`size`):

- `best_idx(tol=self.problem.c_tol)`: Esta función permitirá obtener la posición en la que se encontrará el mejor individuo, que si el problema es de objetivo único y sin restricciones, el mejor es simplemente el individuo con el menor “fitness”.

- Propiedad `champion_f`: Esta característica devuelve el vector "fitness" del candidato campeón.
- Propiedad `champion_x`: Esta característica devuelve el vector de decisión del candidato campeón.
- Propiedad `problem`: Esta característica devuelve una referencia al problema interno.

Clase algoritmo (algorithm)

Como su propio nombre indica, esta función dentro de la librería de `PyGMO` se caracteriza por determinar cuál será el algoritmo utilizado. Para ello, se hará uso de un algoritmo definido por el usuario, UDA.

Estos algoritmos son destinados a la optimización y pueden ser de diferentes tipos como estocástico, determinista, basado en poblaciones, libre de derivadas, usando hessianas, usando gradientes, un meta-heurístico, evolutivo, etc.

Gracias a esta clase se puede acceder a muchos algoritmos con la capacidad de encontrar una solución al problema del usuario. Pero para ello es necesario primero definir las propiedades y funciones del algoritmo, destacando dos funciones principales:

- `evolve(pop)`: Esta función llamará al método `evolve()` de la UDA con el objetivo de obtener la "evolución". Es decir, este método toma como entrada una población, y se espera que devuelva una nueva población generada por la evolución (u optimización) de la población original.
- `set_verbosity()`: Esta función establece la "verbosidad" de los registros y la salida en pantalla, es decir, le permite especificar el nivel de registro de su servidor Memcached. Por tanto, los niveles más altos producen salidas más detalladas (verbose).

Clase isla (island)

Esta función se trata de un bloque paralelo par unidades de `PaGMO`, la librería asociada a `PyGMO`. Por ello, una isla se trata de una unidad computacional qué puede ser tanto física como virtual (`evolve`).

También se definirán sus características en UDI, similar a como se hicieron en las UDP y UDA anteriormente comentadas. Gracias a la librería muchas UDI ya vienen previamente implementadas. Cuando se suman varias islas, se tiene un archipiélago, que es la clase que se describirá en el siguiente apartado.

Una isla necesita las siguientes clases y propiedades:

1. Una isla definida por el usuario (UDI).
2. Un algoritmo (`algorithm`).
3. Una población (`population`).
4. Una política de reemplazo (de tipo `r_policy`).
5. Una política de selección (de tipo `s_policy`).

Las políticas de reemplazo y selección se utilizan cuando la isla forma parte de un archipiélago. Gracias a ellas se puede establecer cómo se seleccionan y sustituyen los individuos de la isla cuando se producen migraciones entre islas dentro del archipiélago. En

el caso de que la isla no forme parte de un archipiélago, esas políticas de reemplazo y selección no jugarán ningún papel.

Dentro de las funciones importantes de una isla, se encuentran las siguientes:

- `evolve(n=1)`: Esta función llamará al método `evolve` para evolucionar a la población (`population`) de la isla usando el algoritmo (`algorithm`) de la isla, añadiéndose a una cola para su resolución.
- `get_algorithm()`: Esta función devuelve el algoritmo de la isla.
- `get_population()`: Esta función devuelve la población de la isla.
- `get_r_policy()`: Esta función devuelve la política de reemplazo de la isla.
- `get_s_policy()`: Esta función devuelve la política de selección de la isla.

Clase archipiélago (archipelago)

Gracias a esta clase, que contiene varias islas, se pueden llevar a cabo tareas de optimización. Para ello, un archipiélago es capaz de iniciar la evolución de cada isla, de manera asíncrona al tiempo, realizando un seguimiento de resultados y del intercambio de información entre diferentes tareas. Hay que tener en cuenta que las distintas islas pueden ser heterogéneas, y por lo tanto, referirse a diferentes UDA, UDP y UDI.

Las islas están conectadas por una topología y pueden intercambiar individuos (es decir, posibles soluciones que son candidatas) mediante un proceso denominado migración. Para ello, se tendrá que hacer uso de las rutas descritas por la topología, y las políticas de reemplazo y selección de las islas (véase r_policy y s_policy) donde se establece cómo reemplazar y seleccionar individuos de las poblaciones de las islas.

Funciona de manera similar a las islas con el añadido de la migración y nuevas funciones:

- `evolve(n=1)`: Esta función llamará al método `evolve` para evolucionar a la población (`population`) de las islas usando el algoritmo (`algorithm`) de la islas, añadiéndose a una cola para su resolución.
- `get_champion_f`: Esta función devuelve los vectores “fitness” de los candidatos campeones de las islas.
- `get_champion_x`: Esta característica devuelve los vectores de decisión de los candidatos campeones de las islas.

3.2.2. Aplicación de PyGMO

Habiendo definido las clases más importantes y sus funciones, se procede a explicar cómo se aplica la librería de `PyGMO` en el caso de que se use `Pykep` para la optimización de trayectorias interplanetarias.

Primero, será necesario definir el problema que se busca optimizar a través de la función `problem()`. Posteriormente, se definirá la población de candidatos junto con el problema previamente definido a través de la función `population()`.

Por otro lado, hay que conseguir el algoritmo con el que se optimizará el problema por lo que haciendo uso de la función `algorithm()`, se conseguirá el algoritmo con el que

se "evolucionará". Independientemente del optimizador que se use, la función (`evolve()`) tiene que ser utilizada para poder optimizar el problema deseado.

Finalmente, se obtiene el mejor candidato tras la optimización a través de la función `get_f()`. Un código de ejemplo para un problema a optimizar sería el siguiente:

```
1  import PyGMO as pg
2
3  # The problem
4  prob = pg.problem(pg.rosenbrock(dim = 10))
5
6  # The initial population
7  pop = pg.population(prob, size = 20)
8
9  # The algorithm (a self-adaptive form of Differential Evolution (sade - jDE variant)
10 algo = pg.algorithm(pg.sade(gen = 1000))
11
12 # The actual optimization process
13 pop = algo.evolve(pop)
14
15 # Getting the best individual in the population
16 best_fitness = pop.get_f()[pop.best_idx()]
17 print(best_fitness)
```

Resolver un problema de optimización utilizando un algoritmo de optimización se describe, en **PyGMO**, como evolucionar una población. En la literatura científica se ha desarrollado en las últimas décadas un interesante debate sobre si la evolución es o no una forma de optimización. En **PyGMO** adoptamos el punto de vista opuesto y consideramos la optimización, de todo tipo, como una forma de evolución. Independientemente de si usas un SQP, un optimizador de punto interior o un solucionador de estrategias evolutivas, en **PyGMO** siempre tendrás que llamar a un método llamado evolve() para mejorar tus soluciones iniciales, es decir, tu población.

3.2.3. Aplicación de Pykep y PyGMO

La mayor parte de los problemas de trayectorias desarrollados en **Pykep** están pensados para ser desarrollados y optimizados junto a **PyGMO**. En esta ocasión se mostrará un problema sencillo de este tipo, que se contempla dentro de los ejemplos de la librería **Pykep** y siendo el siguiente código junto con su output gráfico:

```
1  import pykep as pk
2  import pygmo as pg
3  import numpy as np
4
5  # Plotting imports
6  import matplotlib as mpl
7  from mpl_toolkits.mplot3d import Axes3D
8  import matplotlib.pyplot as plt
9
10 # We define the optimization problem
11 udp = pk.trajopt.pl2pl_N_impulses(
12     start=pk.planet.jpl_lp('earth'),
13     target=pk.planet.jpl_lp('venus'),
14     N_max=3,
15     tof=[100., 1000.],
```

```
16      vinf=[0., 4],
17      phase_free=False,
18      multi_objective=False,
19      t0=[pk.epoch(0), pk.epoch(1000)])
20  # All pykep problems in the module trajopt are compatible with pygmo.
21  # So we create a pygmo problem from the pykep udp (User Defined Problem)
22  prob = pg.problem(udp)
23  print(prob)
24
25  # Here we define the solution strategy, which in this simple case is to use
26  # Covariance Matrix adaptation Evolutionary Strategy (CMA-ES)
27  uda = pg.cmaes(gen=1000, force_bounds = True)
28  algo = pg.algorithm(uda)
29  # Here we activate some degree of screen output (will only show in the terminal)
30  algo.set_verbosity(10)
31  # We construct a random population of 20 individuals (the initial guess)
32  pop = pg.population(prob, size = 20, seed = 123)
33  # We solve the problem
34  pop = algo.evolve(pop)
35
36
37  # Plot our trajectory
38  fig = plt.figure(figsize = (16,5))
39  ax1 = fig.add_subplot(1, 3, 1, projection='3d')
40  ax2 = fig.add_subplot(1, 3, 2, projection='3d')
41  ax3 = fig.add_subplot(1, 3, 3, projection='3d')
42  ax1 = udp.plot(pop.champion_x, axes = ax1)
43  ax2 = udp.plot(pop.champion_x, axes = ax2)
44  ax2.view_init(elev=90, azim=0)
45  ax3 = udp.plot(pop.champion_x, axes = ax3)
46  ax3.view_init(elev=0, azim=0)
```

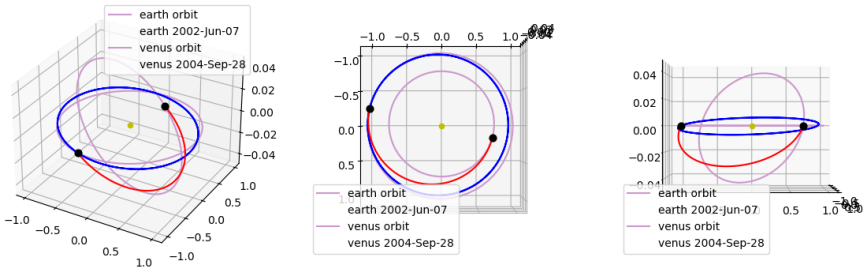


Figura 3.2: Resultado de la transferencia de impulsos múltiples entre la Tierra y Venus optimizada en Pykep y PyGMO

Cómo se puede ver gracias este código se hace uso principalmente de un problema de múltiples impulsos **`pl2pl_N_impulses`** dentro de la librería **`trajopt`**. Primero se define el problema y después se procede ejecutar la optimización. Finalmente, se representa lo que se define como candidato campeón.

Para ello se tendrán en cuenta cuáles son los planetas de inicio y final de la trayectoria, el número máximo de impulsos, el tiempo de vuelo, el tiempo posible de inicio de vuelo, la velocidad de escape máxima o si el problema es multiobjetivo. esto quiere decir que

cualquier cambio dentro de estas variables supondrá un cambio en el problema y por tanto, en la solución optimizada.

También pueden observarse el gran número de funciones empleadas, dependiendo de los algoritmos y problemas creados gracias a `PyGMO` y `Pykep`. Por ello, se puede observar como diferentes variables y diferentes problemas pueden llevar a una solución completamente diferente.

3.3. PyQT5

Esta librería se apoyará en el uso principal de la herramienta QT Designer para el desarrollo de la interfaz gráfica dentro de Python. Por ello, se procederá a explicar brevemente cómo funciona esta herramienta y las posibilidades de widgets que incorpora.

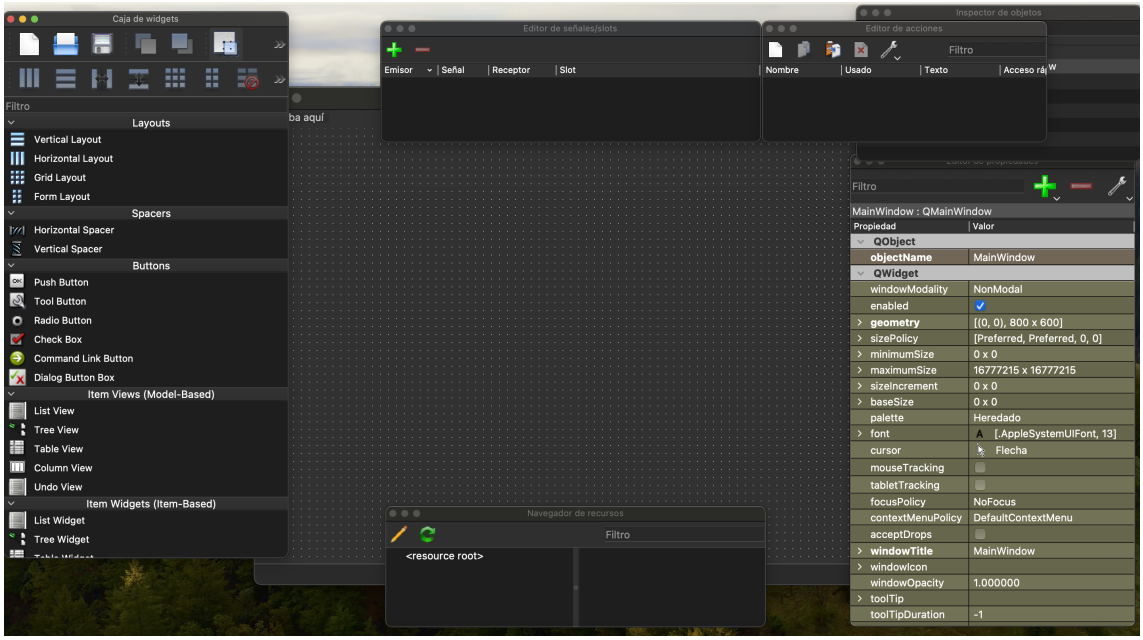


Figura 3.3: Muestra de la interfaz de QT Design

Se pueden observar los principales elementos que permitirán el diseño de interfaces gráficas, destacando los siguientes:

- Caja de widgets: a la izquierda del todo, se pueden ver todos aquellos widgets que se pueden utilizar para el diseño de la interfaz gráfica.
- Área de diseño: en el centro se tendrá el área de diseño de la interfaz gráfica, donde se arrastrarán todos aquellos widgets que se vayan a utilizar.
- Inspector de objetos: Situado en la esquina derecha será una lista de todos los widgets desplegados en el área de diseño.
- Editor de propiedades: En la ventana inferior derecha, se podrán editar las propiedades de cada widget.

3.4. Pandas

Como ya se ha adelantado, `Pandas` es una librería muy adecuada para el manejo de datos, que pueden ser de diferentes tipos. Por ejemplo: datos tabulares con columnas heterogéneas (similar a una tabla en SQL o una hoja de cálculo de Excel), series temporales, datos matriciales en filas y columnas, datos estadísticos, etc.

Las dos estructuras principales que se usan en esta librería y que se usarán en el desarrollo del software son Series (unidimensionales) y DataFrames (bidimensionales). Ambas se usan en conjunto, componiendo Dataframes a partir de Series.

`Pandas` fue desarrollado a partir de NumPy, una librería científica importante de Python, por lo que se integra bien dentro de un entorno de computación científica y con otras librerías. Se destaca por su buen comportamiento en ámbitos como los siguientes:

- Sencillo manejo de datos no rellenados (representados como `NaN`.
- Insertar y eliminar filas y columnas dentro de DataFrames.
- Alinear datos de manera manual o automática en función de etiquetas o valores.
- Agrupar de manera flexible diferentes datos, además de poder convertir datos de Numpy y Python a objetos de DataFrame y viceversa.
- Capacidad para cargar o exportar archivos CSV, delimitados, provenientes de Excel, bases de datos, etc.

El objetivo era solventar deficiencias que se habían encontrado en otros lenguajes y entornos de investigación científica, ya que para estos últimos, cuando trabajan con datos necesitan dividir adecuadamente en varias etapas. Normalmente, los pasos a seguir suele ser: manipulación y limpieza de datos, análisis y modelado, y por último, obtención de resultados y conclusiones. Esos resultados se muestran de diversas maneras, tanto de manera gráfica como en tablas.

Además, existen otros motivos que hacen que sea muy útil esta librería:

- `Pandas` es una librería rápida, ya que muchos algoritmos han sido optimizados en `Cython` (`Cython` es un compilador estático optimizador tanto para Python que facilita la escritura de extensiones en C para Python, siendo tan sencillo como escribir en el propio Python).
- `Pandas` forma parte del ecosistema de computación estadística de Python, siendo una dependencia de `statsmodels`, módulo que proporciona clases y funciones para la estimación de diversos modelos estadísticos.
- `Pandas` es usado de manera habitual en producir aplicaciones financieras

3.5. Itertools

`itertools` es una librería que proporciona varias funciones para operar con iteradores y producir unos más complejos. Es un herramienta que permite de manera rápida y eficiente, en términos de memoria, iterar por sí misma o en combinación, formando un "álgebra de iteradores".

Existen tres tipos de iteradores:

1. Iteradores infinitos: Estos iteradores en Python son aquellos en los que se usen con un bucle `for in`. Elementos como listas, tuplas, diccionarios y conjuntos en Python son ejemplos de iteradores incluidos, en los que no es necesario que el objeto iterador se agote, por eso se denominan infinitos.
2. Iteradores combinatorios: Se caracterizan por ser utilizados para simplificar elementos de combinatoria como permutaciones, combinaciones y productos cartesianos. Destacar en este apartado las funciones `product`, `permutations`, `combinations` y `combinations_with_replacement`.
3. Iteradores terminantes: Estos últimos se utilizan para trabajar con secuencias de entrada cortas y producir una salida que está orientada en la funcionalidad del método utilizado.

3.6. Os

`os` proporciona de manera portátil funcionalidades dependientes del sistema operativo, permitiendo así leer o escribir archivos, además de gestionar directorios entre otras características.

Por ejemplo, para leer o escribir un archivo será necesario el uso de la función `open`, para gestionar rutas `path`, para exportar a un archivo CSV desde un DataFrame se hace uso de la función `to_csv`.

3.7. Matplotlib

La librería `matplotlib` permite exportar datos de forma gráfica, pudiendo mostrar así figuras con diferentes ejes y coordenadas, en dos dimensiones y tres dimensiones, generando así numerosas posibilidades de representación.

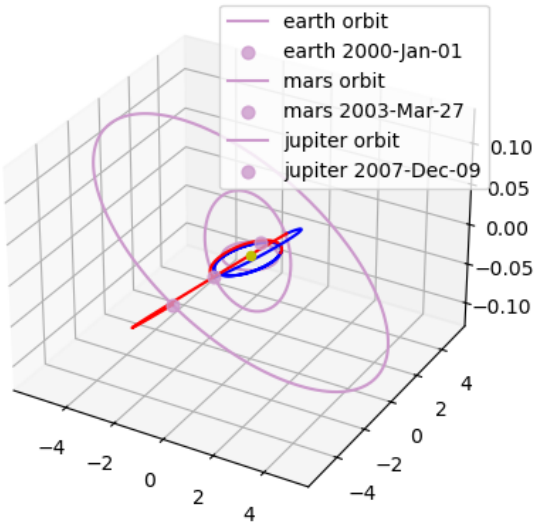


Figura 3.4: Muestra de un gráfico 3D de matplotlib

Los tipos de gráficos que puede mostrar son, entre otros:

- Diagramas de barras
- Histogramas
- Gráficos de sectores
- Gráficos de cajas
- Gráficos de dispersión o puntos
- Gráficos de áreas
- Gráficos de líneas
- Gráficos de contorno
- Mapas de color

Capítulo 4

Software desarrollado

Este capítulo tratará del software desarrollado para poder obtener trayectorias óptimas basadas en las dos librerías previamente comentadas, `Pykep` y `PyGMO`, explicando el código que se ha creado.

Se debe de comenzar explicando que este código se ha desarrollado de manera modular, es decir, se tendrán diferentes scripts para cada funcionalidad que se quiera obtener llamando a las clases correspondientes para hacer uso de sus funciones.

A continuación, se muestra una lista que desglosa los diferentes módulos de los que se compone el código que permite obtener trayectorias óptimas:

1. `TrajectoryProblem`: Este módulo se centrará en clasificar los problemas que se han introducidos por código por parte del usuario, en el caso de que lo hiciera manual, o por un modelo de optimización que llame al módulo con unos parámetros de entrada y un tipo de problemas específico para que los acumule y los resuelva posteriormente.
2. `TrajectoryProblemOptimization`: Esta clase buscará conseguir el problema optimizado y resuelto, haciendo uso de los algoritmos comentados en apartados anteriores y se elegirá el más adecuado en función del problema que se haya identificado en la clase `TrajectoryProblem`.
3. `TrajectoryIterinator`: Este módulo tendrá la función de iterar las posibles secuencias de planetas que permitan alcanzar la trayectoria más óptima entre un planeta de origen y un planeta de destino. Para ello, se ejecutarán bucles en los que se resolverán varios tipos de problemas con las diferentes secuencias calculando el parámetro o los parámetros de dichas optimizaciones. Ese parámetro será el incremento de velocidad necesario, añadiéndose la optimización del tiempo en el caso de que sea multiobjetivo.
4. `Main program`: El programa principal proporcionará la interfaz gráfica con la que el usuario podrá seleccionar los planetas de origen y destino, tanto como si las simulaciones multiobjetivo y otra serie de parámetros que puedan considerarse útiles para el planteamiento de la misión.

El software desarrollado en módulos o clases es el siguiente:

4.1. TrajectoryProblem

Este código define una clase `TrajectoryProblemSolver` que se utiliza para crear y resolver diferentes tipos de problemas de optimización de trayectoria. La clase inicializa listas vacías para almacenar los problemas de cada tipo. Los métodos `mga_problem`, `mga_1dsm_problem`, `pl2pl_N_impulses_problem`, `lt_margo_problem`, `direct_pl2pl_problem`, `mr_lt_nep_problem`, `indirect_pt2pt_problem`, `indirect_or2or_problem`, `indirect_pt2or_problem` se utilizan para crear el tipo específico de problema llamando a la función apropiada del módulo `pykep.trajopt` y añadiendo la instancia del problema a la lista correspondiente.

El método `solve_all_problems` se utiliza para resolver todos los problemas específicos de cada tipo de problema, buscando así resolver cada uno de los problemas ya que llamará a las funciones: `solve_mga_problems`, `solve_mga_1dsm_problems`, `solve_pl2pl_N_impulses_problems`, `solve_lt_margo_problems`, `solve_direct_pl2pl_problems`, `solve_mr_lt_nep_problems`, `solve_indirect_pt2pt_problems`, `solve_indirect_or2or_problems` y `solve_indirect_pt2or_problems`.

Cada vez que se resuelva un problema, se imprimirán primero un mensaje indicando que se está resolviendo y el número de problema específico, es decir, si es el cuarto problema resuelto de *MGA*. Por otro lado, un segundo mensaje aparecerá una vez se haya resuelto, indicando también el número del problema. Finalmente, todos problemas resueltos se añaden a la lista `trajectory_problems`.

Para ver el código desarrollado, ver el apartado del apéndice correspondiente referente al código `TrajectoryProblem.py` (*Apéndice A.1*).

4.2. TrajectoryOptimization

Esta clase busca optimizar los problemas resueltos anteriormente, siguiendo aquellos parámetros en los que se definirán las optimizaciones. Para ello, se definirán dos funciones que contendrán los dos métodos de optimización que se usarán, `optimize_with_nlopt` y `optimize_with_archipelago`. Para ambas funciones se hará uso de la librería `Pygmo`, descrita previamente en el capítulo 3.

Aquella cuyo nombre es `optimize_with_nlopt` se centrará en generar los problemas, poblaciones y algoritmos para resolver la trayectoria óptima de la solución del problema que se analice. También al ser usado el algoritmo `NLOPT`, es necesario generar el gradiente del problema a optimizar.

Por otro lado, `optimize_with_archipelago` hará uso de las clases de `Pygmo` donde se usan algoritmos, problemas y poblaciones para generar islas. Esas islas se juntarán en archipiélagos, donde el usuario podrá iniciar directamente el desarrollo de varias islas a la vez y obtener el resultado más óptimo.

La función que permitirá seleccionar la opción a optimizar en función de la clase de problema que se introduzca es `optimize_choice`. Con esta función se podrá identificar el problema que se analiza, gracias a un conjunto de estructuras de control (`if, else if`) en

la que se clasifican los problemas haciendo uso de `isinstance`, distinguiendo problemas como `mga` y `mga_1dsm`.

Estos dos problemas serán los únicos que se resolverán y optimizarán las secuencias para ellos como se verá más adelante con la explicación del código de `TrajectoryIterinator.py`. El motivo reside en que se han descartado otro tipo de problemas porque tienen objetivos diferentes en cuanto a trayectorias.

Dentro de cada una de las estructuras de problemas se obtendrán las ΔV totales de cada problema optimizado, que posteriormente permitirá ser comparado, y los ΔV_i, que serán los ΔV particulares de cada una de las maniobras que se emplean en una trayectoria para una secuencia definida.

Todos los problemas, poblaciones, UDP, archipiélagos o algoritmos, ΔV y ΔV_i se acumularán en listas respectivas haciendo uso de la librería `Pandas`. Estas a su vez compondrán un `DataFrame` que permitirá disponer los datos en esa variable de `Pandas` como si fuera una tabla. Siendo ese `DataFrame` la variable que devolverá la función `optimize_choice` a través de `return`.

Para ver el código desarrollado, ver el apartado del apéndice correspondiente referente al código `TrajectoryOptimization.py` (*Apéndice A.2*).

4.3. TrajectoryIterinator

La clase `TrajectoryIterinator` tiene como objetivo generar diferentes secuencias de planetas para poder ser analizadas y encontrar la trayectoria óptima entre un planeta de origen y un planeta de destino. Esas secuencias estarán compuestas por el planeta de origen, planetas intermedios en los que hacer un flyby y el planeta de destino, siendo por tanto esos planetas intermedios los que serán iterados.

Por un lado, se tendrá una función que definirán los radios de seguridad (`safe_radius`) de cada planeta, siendo identificados cada uno de los planetas de la secuencia y recibiendo el valor del radio de seguridad para el que se efectuará una asistencia gravitatoria.

Por otro lado, la función `iterinator` buscará y iterar una serie de secuencias con el objetivo de poder usar dichas secuencias en la optimización de problemas para alcanzar la trayectoria óptima. Para ello, se generarán varios bucles `for`.

Esta función, por lo tanto, tendrá una gran importancia, ya que dará todas las combinaciones necesarias para encontrar esa trayectoria óptima. Se definirán entonces varias variables importantes que serán usadas en la iteración.

Primero, se definirá el diccionario `planet_safe_radii`, donde se asociarán los planetas con su radio de seguridad. Seguido a ello, se buscará obtener el planeta de salida (`departure_planet`) y de llegada (`arrival_planet`) dentro de la secuencia de planetas.

Antes de comenzar la iteración, se seleccionarán aquellos planetas que quedarán excluidos de la iteración para la secuencia ya que por motivos energéticos no tiene sentido viajar al planeta de destino o más lejanos respecto al Sol que el planeta de destino para realizar flybys (`excluded_planets`).

Se comienza entonces el bucle principal de la función, que tiene tres niveles de iteración:

- Iteraciones `max_repetitions`: `for _ in range(max_repetitions)`.
- Iteraciones `max_dimension`: `for dimension in range(1, max_dimension + 1)`.
- Producto cartesiano de nombres de planetas con repeticiones de dimensión (utilizando la función `product` del módulo `itertools`):
  `for new_planet_names in product(planet_safe_radii.keys(), repeat=dimension)`

Esos tres niveles de iteración tendrán como objetivo iterar la secuencia y obtener los datos de cada problema optimizado resuelto, para ello, en cada iteración se seguirá el siguiente proceso:

1. Se crea una nueva lista de nombres de planetas `new_planet_names` con elementos de la dimensión actual.
2. Verifica si alguno de los planetas en `new_planet_names` está en la lista de planetas excluidos; si es así, pasa a la siguiente condición.
3. Verifica si `dimension` es mayor que 1; si es así, se procede a evaluar dentro de un bucle `for` si el índice del planeta 1 (`index_planet_1`), que pertenece a la lista `new_planet_names`, es mayor que el del planeta 2 (`index_planet_1`), además de comprobar si (`index_planet_1`) es mayor o igual al (`index_of_departure_planet`). En el caso de que se cumpla, se asignará a la variable `energia_innecesaria` un valor de 1 y se finalizará el bucle. En el caso contrario, se seguirá iterando hasta acabar la secuencia.
4. Se evalúa si `energia_innecesaria` es 1, y si lo es se procede a volver a empezar con la secuencia.
5. Se evalúa de nuevo si alguno de los planetas en `new_planet_names` está en la lista de planetas excluidos.
6. Para el resto de casos, se ejecutará la simulación que permitirá obtener la secuencia y posterior resultado. Para ello, se crea una nueva secuencia `new_seq` compuesta por el planeta de salida, `new_planet_names`, y el planeta de llegada.
7. Se llamará a la función `jpl_lp` para cada planeta de esa secuencia a y convertir cada nombre de planeta en un objeto `jpl_lp` para que `Pykep` pueda reconocer la secuencia y resolver el problema.
8. Se llama a una función `problem_choice` (que se explicará posteriormente) para obtener los resultados de la optimización dentro del `DataFrame data_opt_seq` para la nueva secuencia.
9. La función devuelve (`yield`) la nueva secuencia `new_seq` y los datos de optimización `data_opt_seq`.
10. Si la `dimension` es 1, se ejecutarían los pasos anteriores desde el paso 6 al 9.

Además, se generará una función que será `get_names`, cuyo objetivo será devolver el nombre de cada elemento dentro de la secuencia. Con ese nombre, se podrá identificar el planeta y proceder a las optimizaciones de las posibles trayectorias.

Por último, la función `problem_choice` tiene como objetivo definir los dos problemas que se optimizarán para obtener la mejor trayectoria posible. Para ello, se llamará al módulo `TrajectoryProblemSolver()`, escogiendo los dos problemas que se analizarán, `mga_problem` y `mga_1dsm_problem`.

Ambos problemas tendrán unas condiciones de entrada definidas de manera genérica, mientras otras serán variables de entrada como la secuencia (`seq`), la fecha de inicio de misión (`t0`), siendo el día 1 de enero de 2000 la fecha 0, y la duración de la misión (`tof`).

Posteriormente a llamar a ambos problemas, se resolverán con la función `solve_all_problems` del módulo `TrajectoryProblemSolver`. Entonces, se optimizarán ambos problemas gracias al módulo `TrajectoryProblemOptimization` haciendo uso de la función `optimize_choice` y la nueva secuencia con la función de `get_names`, para obtener los resultados correspondientes que se acumularán en un `DataFrame`.

Para ver el código desarrollado, ver el apartado del apéndice correspondiente referente al código `TrajectoryIterinator.py` (*Apéndice A.3*).

4.4. TrajectoryMin5

Esta clase tiene como objetivo procesar los datos de trayectorias optimizadas recibidas en bruto y seleccionar los cinco mejores casos en los que se minimiza ΔV total de la maniobra.

Para ello lo que se hará es uso de la función `best_5_results`, que toma el `DataFrame data_opt` y selecciona las 5 filas cuyos ΔV_{total} son los menores de todas las posibilidades. En el caso de que existan menos de cinco filas en `data_opt`, se añadirán las filas "dummy" necesarias para llegar a esas cinco filas. Finalmente, se devuelven esas cinco filas ordenadas en la nueva variable con la misma clase que el nombre de la función.

Para ver el código desarrollado, ver el apartado del apéndice correspondiente referente al código `TrajectoryMin5.py` (*Apéndice A.4*).

4.5. TrajectoryVisualizer

Este módulo trata de mostrar por pantalla el mejor caso de la optimización, para así evaluar la consistencia física de la trayectoria y descartar dicho caso si se considera que la maniobra no tiene sentido.

Para ello, se definirá la clase `visualizer` en la que se tendrán que extraer del `DataFrame data_opt` los siguientes elementos: la UDP, la población y el algoritmo o archipiélago.

Entonces, se diferenciará entre la UDP que proviene de una trayectoria de asistencia gravitatoria (`MGA Trajectory`) y una de asistencia gravitatoria con maniobra de espacio profundo (`MGA_1DSM Trajectory`) para representar adecuadamente la trayectoria haciendo uso de las librerías `PyGMO` y `matplotlib`.

Para ver el código desarrollado, ver el apartado del apéndice correspondiente referente al código `TrajectoryVisualizer.py` (*Apéndice A.5*).

4.6. Interfaz TFM

El objetivo del código `Interfaz_TFM.py` no es otro que generar una interfaz gráfica que permita introducir datos por pantalla a cualquier usuario sin experiencia en Python y las librerías que se han usado en estos códigos.

Para ello, se diseña la siguiente interfaz:

Figura 4.1: Diseño de la interfaz gráfica

En ella se pueden ver los siguientes parámetros de entrada:

- Planeta de origen: Se selecciona de la lista el planeta de salida de la misión. Por defecto, aparece la Tierra.
- Planeta de destino: Se selecciona de la lista el planeta de destino de la misión. Por defecto, aparece Mercurio.
- Fecha de salida del planeta de origen: Se selecciona en un calendario la fecha de origen de la misión. Por defecto, la fecha que aparecerá será el 1 de enero de 1970, con hora 00:00 UTC.
- Fecha límite de salida del planeta de origen: Se selecciona en un calendario la fecha límite del planeta de origen de la misión. Por defecto, la fecha que aparecerá será el 1 de enero de 1970, con hora 00:00 UTC.
- Fecha de llegada al planeta de destino: Se selecciona en un calendario la fecha de llegada al destino de la misión. Por defecto, la fecha que aparecerá será el 1 de enero de 1970, con hora 00:00 UTC.
- Número máximo de flybys intermedios: Será el número de flybys intermedios entre el planeta de origen y el planeta de destino. El número por defecto es 1.
- Número máximo de iteraciones para misma secuencia: Será el número de pasadas que se dará dentro del bucle para optimizar nuevamente el problema y evaluar si

se consigue mejor resultado. El número por defecto es 1.

- Iniciar: Este botón servirá para iniciar todo el programa y que se obtengan los diferentes resultados para un problema planteado.

Una vez definidos y relacionados esos parámetros de entrada con unas variables que guarden dichos datos, se procede a llamar al programa principal de este TFM, `mainTFM.py`, que se explicará en el siguiente apartado.

Cabe destacar también el visualizador de trayectorias, que una vez ejecutados todos los problemas, aparecerá una lista de botones en las que se podrá seleccionar todos los resultados, como se observará en la siguiente imagen.

Figura 4.2: Interfaz gráfica del ejemplo de la misión

En dichos botones, podrá verse la opción ordenada en función de la secuencia con menor ΔV, además de mostrarse la secuencia abreviada justo debajo de la opción y su número ordenado.

Una vez pulsado, se abrirán dos ventanas, una que contenga la imagen interactiva con la trayectoria proveniente del visualizador, `TrajectoryVisualizer`, y otra que contenga la secuencia, junto a los ΔV total y ΔV parciales de cada misión en particular.

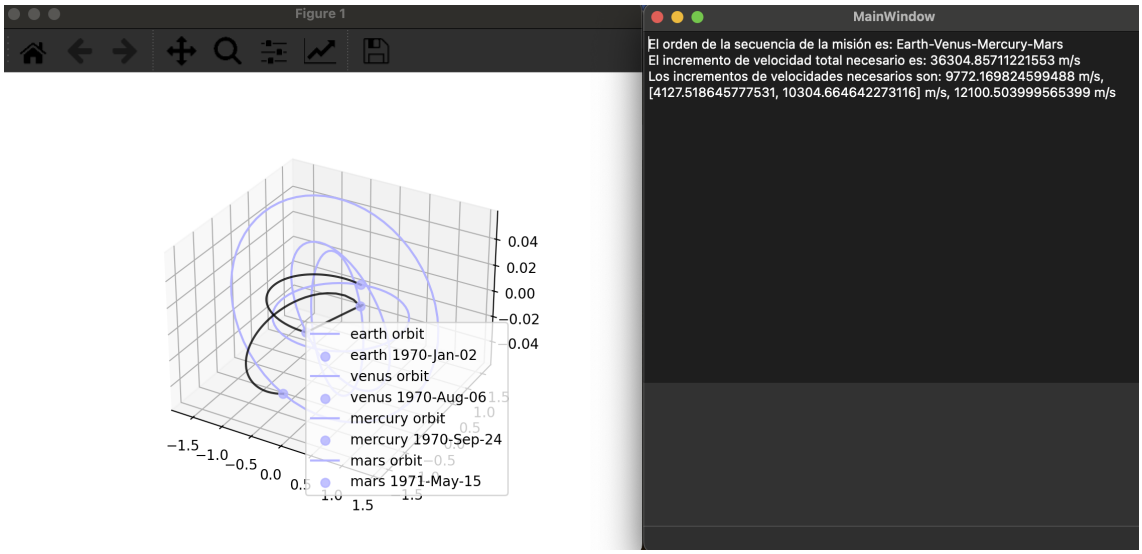


Figura 4.3: Resultados después de la simulación del ejemplo de la misión

Para ver el código desarrollado, ver el apartado del apéndice correspondiente referente al código `Interfaz_TFM.py` (*Apéndice A.6*).

4.7. Main Program - mainTFM

Esta clase `mainTFM` será el código principal que ejecutará la serie de módulos necesarios para llevar a cabo la optimización de trayectorias siguiendo los datos iniciales que se introduzcan por pantalla gracias a la interfaz gráfica desarrollada en QT Designer y haciendo uso de `PyQT5`.

Por ello, primero se definirán las variables de entrada provenientes de la interfaz gráfica: `planeta_origen`, `planeta_destino`, `fecha_salida`, `fecha_salida_limite`, `fecha_llegada`, `n_flybys` y `n_iteraciones`.

Además, se tendrán que definir varias funciones, cada una con un objetivo en la ejecución del código principal. `change_planet_name`, `change_epoch_to_date` y `calculate_tof` dirán para poder realizar aquellas tareas o cálculos que permitan definir las variables necesarias para resolver problemas y generar la optimización en la función principal de este código, `ejecucion`.

La función `change_planet_name` tiene como objetivo cambiar los nombres españoles que se seleccionan en las listas de la entrada a la interfaz, por los nombres de los planetas en inglés, que serán leídos en las secuencias para resolver los problemas.

Como `Pykep` considera como día de referencia (día 0) el 1 de enero del año 2000, se tendrán que convertir las fechas (que ya están en `epoch`) respecto a la fecha de referencia. Esta operación será ejecutada por la función `change_epoch_to_date`, que también convertirá la fecha de segundos (`epoch`) a días (formato requerido por `Pykep`).

Por otra parte, la función `calculate_tof` permite obtener a través de dos fechas, ya convertidas en días a través de `change_epoch_to_date`, el límite superior del tiempo de vuelo (`tof`).

Finalmente, la función principal es **ejecucion**, con la que se conseguirá llegar a los resultados de la optimización en función de las variables de entrada. Para ello, se definirá la secuencia inicial que se usará para comenzar la iteración de secuencias (**seq**), el tiempo de salida límite (**t0**) y el tiempo de vuelo (**tof**).

Posteriormente, se procede a llamar a iniciar la instancia **itera** con la que llamar a la clase **TrajectoryIterinator**, que buscará efectuar la iteración cuando sea llamada. Antes habrá que definir el **data_opt** a través de un **DataFrame** vacío.

A continuación se cargará la nueva lista de secuencia de países, **new_seq_list** y se procederá a iniciar la iteración por medio del bucle **for** con las variables introducidas desde la interfaz gráfica como son **n_iteraciones** y **n_flybys**. Mientras se realiza cada iteración, se generan los resultados de cada nueva secuencia y la siguiente nueva secuencia.

Una vez finalizado se procede a acumular todos los datos en un CSV y a la hora de visualizar se permitirá al usuario ver todas las soluciones a las que se ha llegado en una simulación, dando la posibilidad de poder evaluar las trayectorias, además de los incrementos de velocidad, tanto total como parciales que se mostrará por pantalla.

Para ver el código desarrollado, ver el apartado del apéndice correspondiente referente al código **mainTFM.py** (*Apéndice A.7*).

Capítulo 5

Comparación del software frente a misiones reales

En este capítulo, se buscará emplear el software y códigos creados con el objetivo de poder compararlo con misiones reales. Así se podrá validar sus resultados, e incluso, proporcionar nuevas soluciones a la trayectoria real que se llevó a cabo.

Para ello se seleccionarán misiones con asistencias gravitatorias como las comentadas en la Introducción (*capítulo* 1). Las fechas de inicio y llegada serán las de la propia misión real, además de obviamente los planetas de origen y destino.

También, será importante definir el número de flybys intermedios necesarios en función de la misión principal. Por ejemplo, si la secuencia original ha sido Tierra-Venus-Venus-Tierra-Marte, serán necesarios tres flybys intermedios, por lo que `n_flybys` sería 3, haciendo que la iteración `max_dimension` tenga dicho índice y se generen por tanto múltiples posibilidades en esa secuencia de 3 planetas intermedios.

Las misiones escogidas para la comparación serán:

- Mariner 10
- Voyager 1
- Galileo
- Cassini
- JUICE

5.1. Comparativa con Mariner 10

Esta misión comenzó el 3 de noviembre de 1973, volando hacia Venus para realizar un flyby en este planeta el 5 de febrero de 1974. Posteriormente, se dirigió a su destino, Mercurio, en la que realizó tres flybys con el objetivo de obtener diferentes datos. El primer flyby en Mercurio fue el 29 de marzo de 1974, seguido del segundo el 21 de septiembre de 1974, y finalizando con el tercero, el 6 de marzo de 1975.

Su trayectoria real puede verse representada en la siguiente figura:

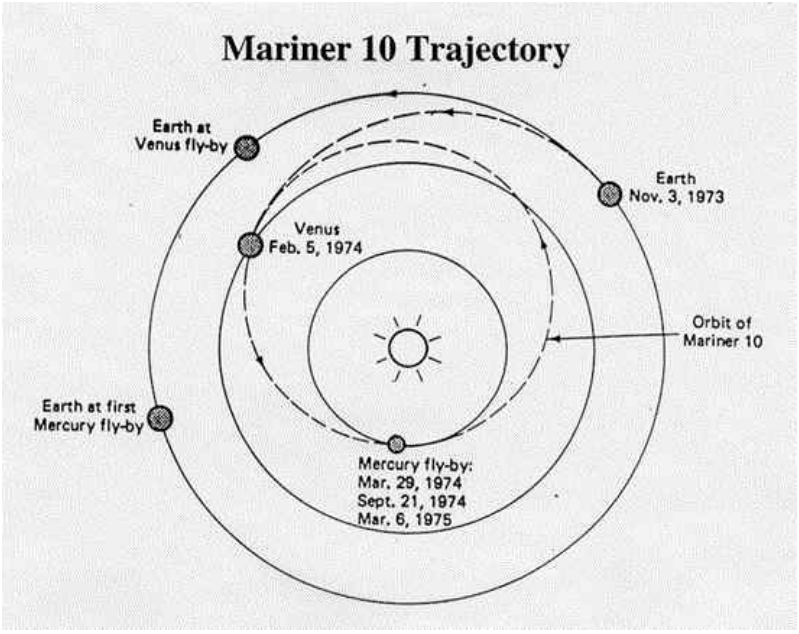


Figura 5.1: Trayectoria de la misión Mariner 10 con flybys indicados

Debido a los tres flybys que se realizan en Mercurio, se ha tomado el último para definir la fecha de llegada dentro del software:

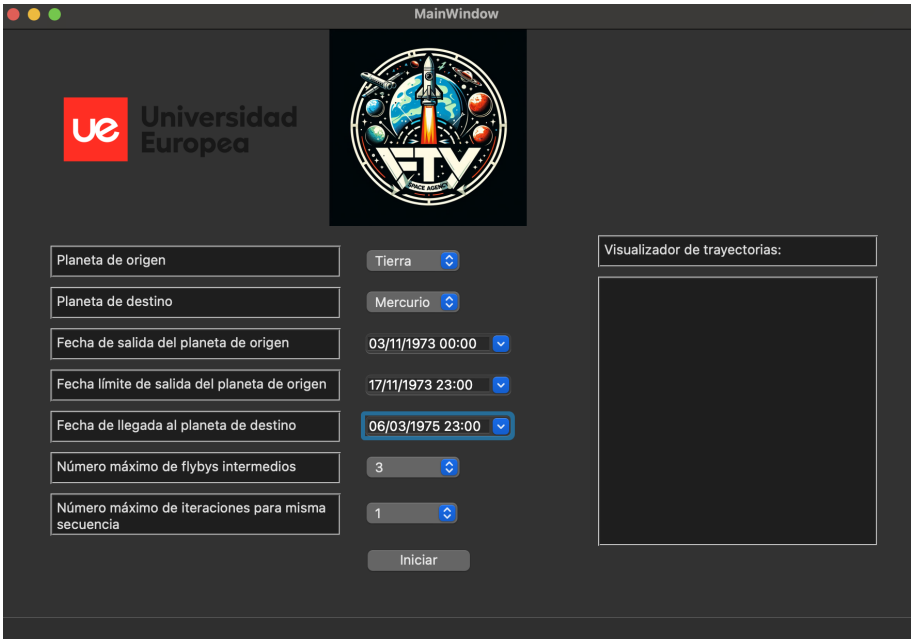


Figura 5.2: Configuración del software para la optimización de la trayectoria del Mariner 10

La mejor solución encontrada tiene un ΔV = 9,745 km/s. Se trata de una trayectoria de asistencia gravitatoria con maniobra de espacio profundo, por lo que sus ΔV parciales son las siguientes: [1,324 km/s, 1,929 km/s, 1,878 km/s, 4,613 km/s].

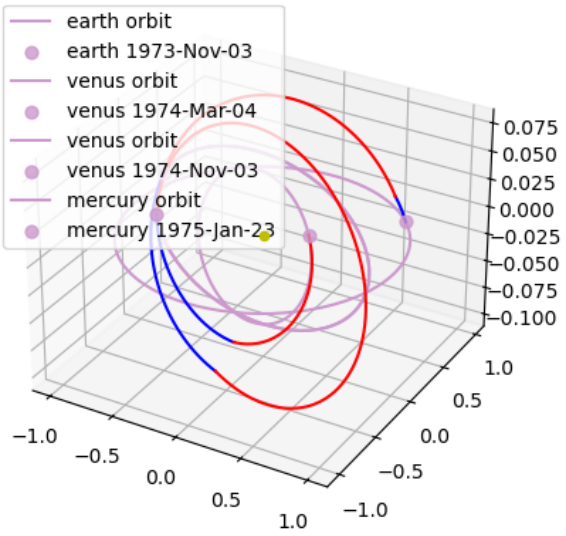


Figura 5.3: Mejor trayectoria de la misión simulada de Mariner 10 con el software de optimización

La llegada de esta trayectoria sería dos meses antes que la original, haciendo un paso extra por Venus el 3 de noviembre de 1974. Sin embargo, la gran ventaja de este programa es que permite visualizar y obtener los datos de otras optimizaciones con peor resultado, pero que pueden cuadrar más con la misión que se busque o respecto a la original con la que se compara.

Este es el caso de la segunda mejor opción cuyo ΔV es algo superior, 10,296 km/s. Sin embargo, la trayectoria se asemeja más a la original, puesto que solo hace un flyby en Venus, con llegada posterior a Mercurio:

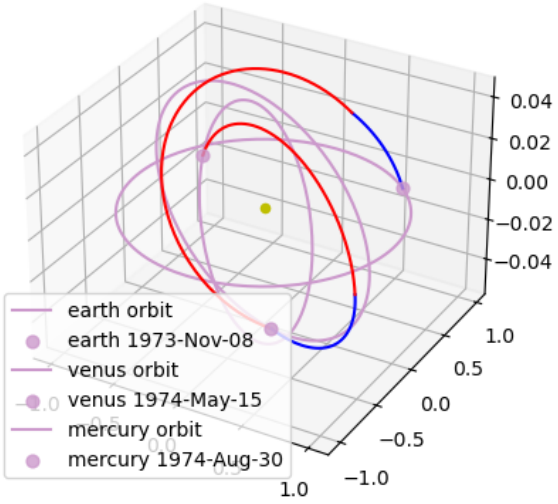


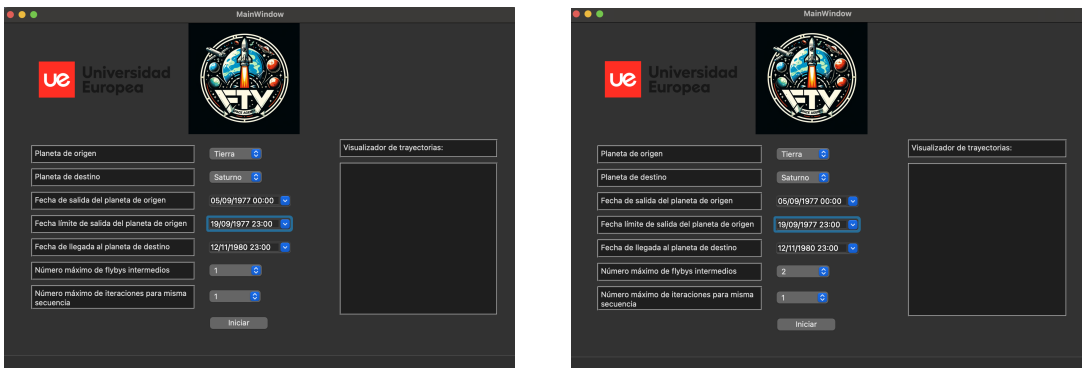
Figura 5.4: Segunda mejor trayectoria de la misión simulada de Mariner 10 con el software de optimización

5.2. Comparativa con Voyager 1 y Voyager 2

Las misiones Voyager 1 y Voyager 2 tenían como objetivo inicial llegar a Júpiter y Saturno para exploración científica. Sin embargo, Voyager 2 después fue ampliada a Urano y Neptuno.

Para esta comparativa se tomará la misión Voyager 1 como referencia desde su salida en la Tierra hasta su llegada a Júpiter y a Saturno. Para ello, se usarán las fechas de la *Figura 1.4*.

Al solo estar especificados los dos flybys de Júpiter y Saturno para después salir fuera del Sistema Solar. Como la misión original consistía en llegar a Saturno, se considerará éste como destino final. Para ello se han planteado dos posibilidades, 1 único flyby intermedio y 2 flybys intermedios.

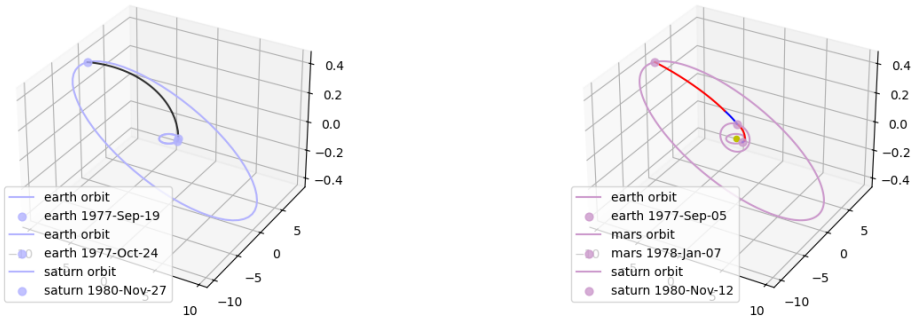


(a) Setup único flyby intermedio Voyager 1 **(b)** Setup dos flybys intermedios Voyager 1

Figura 5.5: Setup de las dos configuraciones simuladas para Voyager 1

Para un solo flyby intermedio se tiene como mejor solución una secuencia Tierra-Tierra-Saturno, con un ΔV = 21,663 km/s. Se trata de una trayectoria de asistencia gravitatoria, por lo que sus ΔV parciales son las siguientes: [0 km/s, 11,660 km/s, 10,003 km/s].

Mientras, la segunda mejor solución de un solo flyby es una trayectoria de asistencia gravitatoria con maniobra de espacio profundo de secuencia Tierra-Marte-Saturno, con un ΔV = 21,824 km/s y sus ΔV parciales son las siguientes: [11,579 km/s, 0,211 km/s, 10,034 km/s].

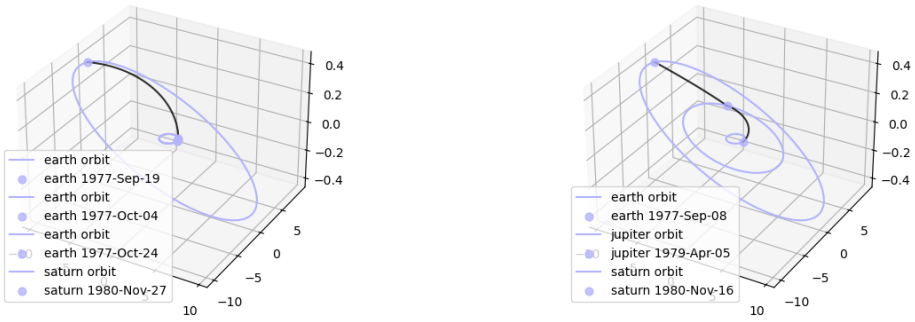


(a) Primera mejor solución **(b)** Segunda mejor solución

Figura 5.6: Simulación con único flyby intermedio de Voyager 1

En el caso de dos flybys intermedios, la mejor solución vuelve a ser la misma que para un único flyby intermedio, secuencia Tierra-Tierra-Saturno y ΔV = 21,663 km/s. Mientras que la segunda mejor solución es muy parecida, trayectoria de asistencia gravitatoria de secuencia Tierra-Tierra-Tierra-Saturno, con un ΔV = 21,663 km/s y ΔV parciales: [0 km/s, $3x10^{-6}$ km/s, 11,660 km/s, 10,003 km/s].

Con ambos casos, se puede observar que los mejores resultados suelen ser para un flyby intermedio único. Para ver la secuencia de Tierra-Júpiter-Saturno, se tiene que esperar hasta la opción novena cuando se simulan dos flybys intermedios. Esta trayectoria será del tipo *MGA*, con un ΔV = 26,622 km/s y ΔV parciales: [7,663 km/s, 3,246 km/s, 15,713 km/s].



(a) Segunda mejor solución **(b)** Novena mejor solución (flyby en Júpiter)

Figura 5.7: Simulación con dos flybys intermedios máximos de Voyager 1

Por otro lado, la simulación de Voyager 2 comienza con una fecha del 20 de agosto de 1977, como la original, siendo la fecha de destino de Neptuno, el 25 de agosto de 1989. Se han seleccionado tres flybys para poder contemplar la posibilidad de la misión original que realizó un flyby en Júpiter, otro en Saturno y otro en Urano.

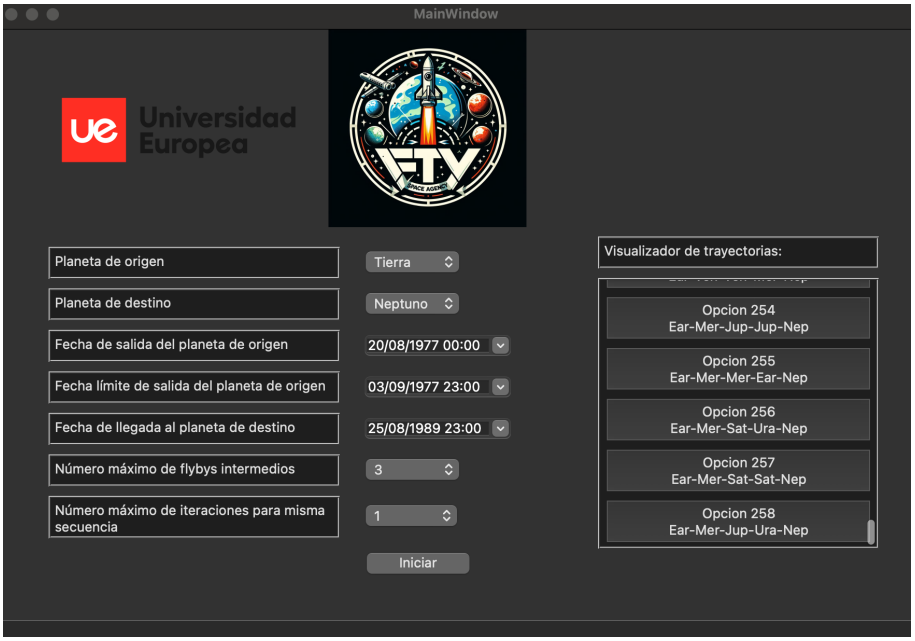
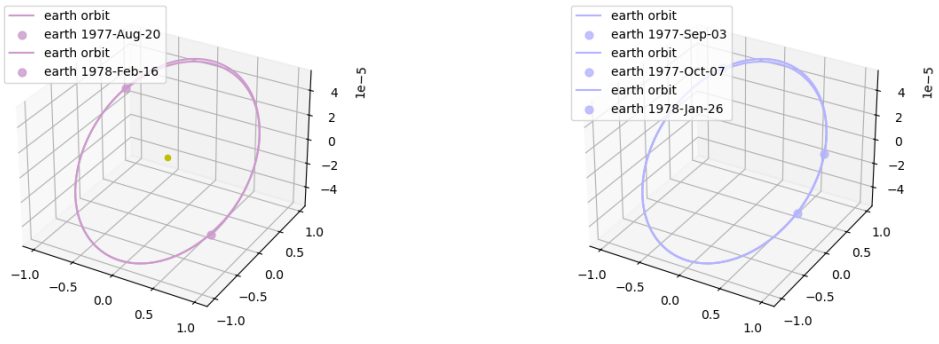


Figura 5.8: Configuración del software para la optimización de la trayectoria del Voyager 2

La problemática principal de esta simulación es la falta de visualización gráfica de la trayectoria ya que tras realizar múltiples simulaciones, se ha descubierto que más allá de Saturno, la sincronización entre `Pykep` y `matplotlib` falla, impidiendo ver dichas maniobras. Pero si se pueden obtener valores numéricos, por lo que se comentarán dichos casos.

El mejor resultado corresponde a una secuencia de misión Tierra-Tierra-Neptuno, con un incremento de velocidad total necesario de 22,289 km/s. Los incrementos de velocidades parciales necesarios son: [0,149 km/s, 10,424 km/s, 11,716 km/s].

El segundo mejor caso es una secuencia Tierra-Tierra-Tierra-Neptuno. El ΔV total de la maniobra es similar al anterior, 22,525 km/s. Los ΔV parciales asociados a la maniobra son [0 km/s, 16,5x10^{-6} km/s, 12,571 km/s, 9,954 km/s].



(a) Mejor solución **(b)** Segunda mejor solución

Figura 5.9: Simulación con flybys intermedios de Voyager 2

La opción 4 tiene la misma secuencia que la misión: Tierra-Júpiter-Saturno-Urano-Neptuno. El incremento de velocidad total necesario es 23,719 m/s. Por otro lado, los incrementos de velocidades necesarios son: [7,029 km/s, 0,002 km/s, 12x10^{-6} km/s, 0,003 km/s, 16,685 km/s].

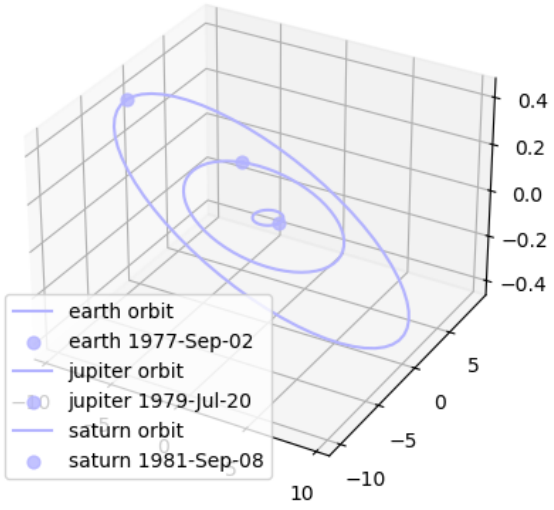


Figura 5.10: Simulación con tres flybys intermedios de Voyager 2

5.3. Comparativa con Galileo

La misión Galileo tenía como objetivo llegar a Júpiter para investigar sobre el planeta y sus lunas. Por ello, la misión salió el 18 de octubre de 1989, llegando al planeta Júpiter el 7 de diciembre de 1995. El 21 de septiembre de 2003 Galileo se vería atrapado por la atmósfera de Júpiter y acabaría estrellándose en el propio planeta (ver *Figura* 1.5).

Su trayectoria fue la siguiente:

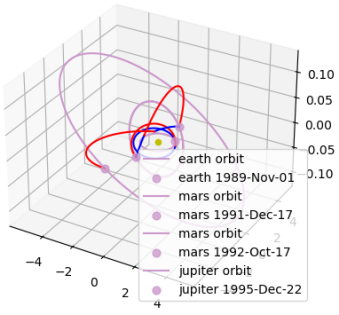
1. Lanzamiento desde la Tierra: 18 de octubre de 1989
2. Primer flyby en Venus: 10 de febrero de 1990
3. Primer flyby en la Tierra: 8 de diciembre de 1990
4. Segundo flyby en la Tierra: 8 de diciembre de 1992
5. Llegada a Júpiter: 7 de diciembre de 1995

Para la misión Galileo se han establecido las mismas fechas de lanzamiento y llegada, eligiendo tres flybys intermedios en la configuración de la simulación como se puede observar en la siguiente imagen:

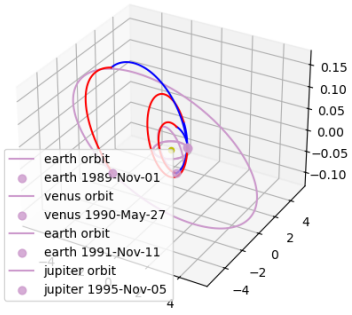
Figura 5.11: Configuración del software para la optimización de la trayectoria de la misión Galileo

Dentro de las simulaciones llevadas a cabo, las dos mejores han sido trayectorias *MGA-1DSM*. La primera opción ha sido una secuencia Tierra-Marte-Marte-Júpiter, con ΔV = 8,857 km/s y ΔV parciales: [0,628 km/s, 0,277 km/s, 3,857 km/s, 4,094 km/s].

Por otro lado, el segundo mejor resultado es una secuencia Tierra-Venus-Tierra-Júpiter, cuya maniobra se realiza gracias a un ΔV = 9,903 km/s y ΔV parciales: [1,891 km/s, 0,654 km/s, 5,148 km/s, 2,210 km/s].



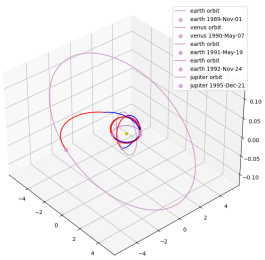
(a) Primera mejor solución



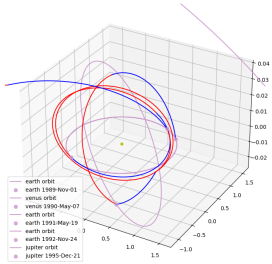
(b) Segunda mejor solución

Figura 5.12: Simulación con tres flybys intermedios de Galileo

Para poder conseguir la misma secuencia de la misión original de Galileo, es decir, la secuencia Tierra-Venus-Tierra-Tierra-Júpiter, hay que esperar a la solución número 17 ya que su ΔV es considerablemente más alta que las mejores opciones pues necesita 14,725 km/s. Por otro lado, los ΔV parciales son [1,256 km/s, 4,621 km/s, 0,418 km/s, 4,231 km/s, 4,199 km/s].



(a) Solución con secuencia original



(b) Detalle de la solución con secuencia original

Figura 5.13: Simulación con tres flybys intermedios máximos de Galileo

Esta solución se puede comparar con los datos obtenidos del estudio NASA (1992), en la que se afirma que "el ΔV total obtenido en estos flybys es de 18,3 km/s", un número superior al que se obtiene en la simulación.

Ese número puede deberse a diferentes factores que no se tengan en cuenta en este software, pero el hecho de acercarse en valor, y en el orden de magnitud, permite evaluar si las soluciones obtenidas aportan valor al estudio de misiones interplanetarias.

5.4. Comparativa con Cassini

Cassini tenía como objetivo estudiar Saturno y sus anillos, por lo que se requería de diversos flybys para llegar al planeta. La trayectoria original consistía en una secuencia Tierra-Venus-Venus-Tierra-Júpiter-Saturno. Los pasos por cada planeta fueron:

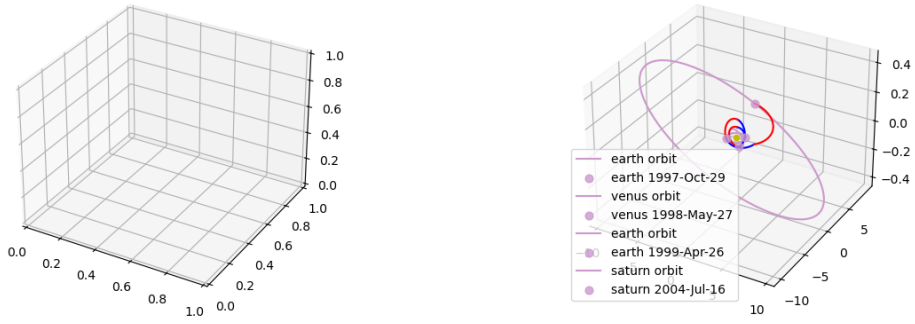
1. Lanzamiento desde la Tierra: 15 de octubre de 1997
2. Primer flyby en Venus: 26 de abril de 1998
3. Segundo flyby en Venus: 24 de junio de 1999
4. Flyby en la Tierra: 18 de agosto de 1999
5. Flyby en Júpiter: 30 de diciembre de 2000
6. Llegada a Saturno: 1 de julio de 2004

La configuración de la fecha de lanzamiento desde la Tierra y la llegada a Saturno han sido las mismas que la misión Cassini original. Además, se han dado 14 días más para el lanzamiento como en el resto de simulaciones que ya se han realizado y las que se harán. Además, se establecerán cuatro flybys intermedios.

Figura 5.14: Configuración del software para la optimización de la trayectoria de la misión Cassini

De esta simulación se consiguen las dos mejores trayectorias que se representan en esta comparación, Tierra-Venus-Júpiter-Saturno (*MGA*) y Tierra-Venus-Tierra-Saturno (*MGA-1DSM*). Cabe destacar que por el error de visualización que en ocasiones ocurre entre la clase de `Pykep` que calcula los problemas de *MGA*, no se mostrará figura en el mejor resultado de la simulación.

La primera consigue desarrollarse con un ΔV = 10,761 km/s y ΔV parciales: [0,839 km/s, 6,247 km/s, $7x10^{-6}$ km/s, 3,675 km/s]. La segunda mejor solución tiene un ΔV = 12,499 km/s y ΔV parciales: [1,887 km/s, 0,009 km/s, 5,246 km/s, 5,357 km/s]. Su representación gráfica no se mostrará debido a un error que en ocasiones sucede con trayectorias *MGA* entre `Pykep` y `matplotlib`.



(a) Primera mejor solución **(b)** Segunda mejor solución

Figura 5.15: Simulación con cuatro flybys intermedios máximos de la misión Cassini

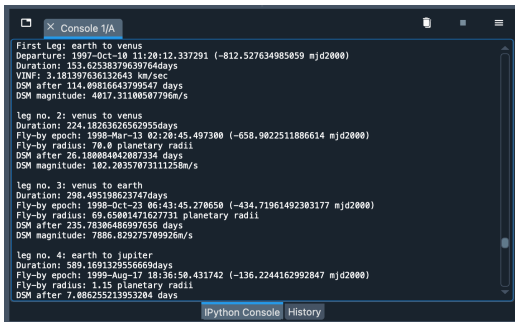
Teniendo todo ello en cuenta, se buscó el caso para el que la misión original cumplía con la secuencia Tierra-Venus-Venus-Tierra-Júpiter-Saturno y trayectoria *MGA*, estando situada en el puesto 41. Esta misión también necesita un ΔV considerablemente más alto, concretamente 18,829 km/s y ΔV parciales: [0,501 km/s, 3,106 km/s, 1,658 km/s, 2,768 km/s, 3,871 km/s, 6,924 km/s].

(a) Solución con secuencia original **(b)** Detalle de la solución con secuencia original

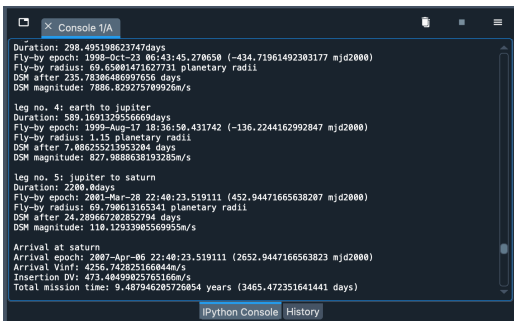
Figura 5.16: Simulación con trayectoria de flybys intermedios originales de la misión Cassini

Gracias a la propia ESA, dentro de `Pykep` se dispone del ejemplo de Cassini como trayectoria optimizada, por lo tanto, se pueden comparar los resultados provenientes de una simulación de esta misión interplanetaria con los resultados que se han obtenido en la prueba del software.

El resultado de la simulación de la misión Cassini con `Pykep` es de un ΔV total de 13,415 km/s, teniendo en cada fase los siguientes ΔV parciales: [4,017 km/s, 0,102 km/s, 7,886 km/s, 0,827 km/s, 0,110 km/s, 0,473 km/s]. La secuencia seguida ha sido Tierra-Tierra-Venus-Tierra-Marte-Tierra-Júpiter, saliendo desde la Tierra el 10 de octubre de 1997 y llegando a Júpiter, el 6 de abril de 2007.



(a) Parte 1 de la solución



(b) Parte 2 de la solución

Figura 5.17: Solución numérica de la misión Cassini con Pykep

Este es el caso en el que más diferencia se ha encontrado, y las causas pueden ser diversas como una mayor claridad en diversos parámetros de entrada para el problema, o simular diferentes tipos de maniobra de asistencia gravitatoria. La misión Cassini de `Pykep` es una trayectoria *MGA-1DSM*, mientras el caso equivalente del software desarrollado es una trayectoria *MGA*.

5.5. Comparativa con JUICE

La misión JUICE tiene como objetivo estudiar las lunas de Júpiter, haciéndose necesaria una adecuada optimización en la trayectoria para reducir los ΔV necesarios para alcanzar el objetivo, ahorrando peso a la nave al disminuir el combustible necesario.

Para ello se seguirá una secuencia Tierra-Luna Tierra-Venus-Tierra-Tierra-Júpiter. Las fechas para llevar a cabo esta misión son las siguientes:

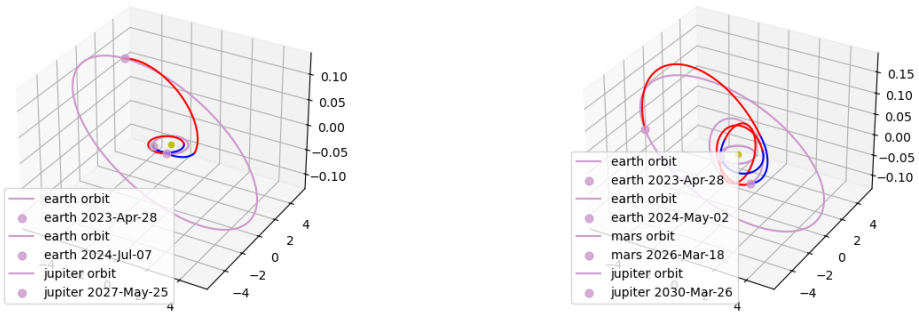
1. Lanzamiento desde la Tierra: 14 de abril de 2023
2. Flyby en la Tierra y la Luna: agosto de 2024
3. Flyby en Venus: agosto de 2025
4. Primer flyby en la Tierra: septiembre de 2026
5. Segundo flyby en la Tierra: enero de 2029
6. Llegada a Júpiter: julio de 2031

En la simulación llevada a cabo en el software se ha tomado como referencia la salida desde la Tierra, el 14 de abril de 2023, mientras que para la llegada a Júpiter, el 1 de agosto, para incluir todo julio como posibilidad de llegada.

Figura 5.18: Configuración del software para la optimización de la trayectoria de la misión JUICE

Tras las simulaciones, las dos mejores opciones han sido trayectorias *MGA-1DSM*. La mejor opción consistía en una secuencia del tipo Tierra-Tierra-Júpiter, que ofrece un ΔV de 10,191 km/s y ΔV parciales: [1,802 km/s, 3,222 km/s, 5,167 km/s].

Por otro lado, la segunda mejor solución propone dos flybys intermedios en vez de uno, siendo la secuencia Tierra-Tierra-Marte-Júpiter. A cambio, el ΔV es algo mayor, de 10,485 km/s, mientras sus ΔV particulares desde cada planeta son: [0,339 km/s, 1,205 km/s, 5,681 km/s, 3,260 km/s].



(a) Primera mejor solución **(b)** Segunda mejor solución

Figura 5.19: Simulación con cuatro flybys intermedios máximos de la misión JUICE

Como el diseño del iterador de secuencias impide que una vez realizado un flyby en la Tierra, se pueda retroceder a un planeta anterior, la mejor secuencia con cuatro flybys que se ha obtenido es Tierra-Tierra-Tierra-Marte-Marte-Júpiter. Para esta secuencia, el ΔV total necesario es de 11,846 km/s, teniendo en cada fase los siguientes ΔV parciales:

[0,562 km/s, 0,056 km/s, 3,799 km/s, 0,002 km/s, 3,418 km/s, 4,009 km/s].

(a) Solución con secuencia de cuatro flybys **(b)** Detalle de la solución de cuatro flybys

Figura 5.20: Simulación de la mejor trayectoria de cuatro flybys intermedios de la misión JUICE

Gracias de nuevo a `Pykep` se dispone de otro ejemplo de misión, en este caso de JUICE como trayectoria optimizada. Por tanto, se simulará dicho código para obtener el resultado y compararlo con la simulación.

El resultado de la simulación con `Pykep` de la misión JUICE es de un ΔV total de 11,568 km/s, teniendo en cada fase los siguientes ΔV parciales: [1,629 km/s, 2,752 km/s, 0,514 km/s, 1,683 km/s, 0,129 km/s, 3,925 km/s, 0,933 km/s]. La secuencia seguida ha sido Tierra-Tierra-Venus-Tierra-Marte-Tierra-Júpiter, saliendo desde la Tierra el 13 de agosto 2022 y llegando a Júpiter, el 3 de agosto de 2030.

(a) Parte 1 de la solución **(b)** Parte 2 de la solución

Figura 5.21: Solución numérica de la misión JUICE con Pykep

Capítulo 6

Conclusiones y propuestas de mejora

6.1. Conclusiones

Este Trabajo Final de Máster ha permitido **explorar sobre la optimización en el diseño de trayectorias interplanetarias**, enfrentando desafíos y problemáticas. A través del proyecto, se han podido obtener diversas conclusiones y alcanzar objetivos que se plantearon inicialmente, mostrando la importancia y viabilidad de la optimización en este ámbito.

Para poder conseguir los hitos planteados, se ha **construido un código en lenguaje** `Python` que apoyado en potentes librerías permite a los usuarios diseñar misiones interplanetarias. Además, se pueden realizar con pocos datos iniciales como planeta de lanzamiento, planeta de destino, fechas de salida y llegada, número de flybys e iteraciones.

Al contar con una interfaz gráfica, permite al usuario modificar diversos parámetros de manera muy sencilla. además, se posee la habilidad de visualizar dichas trayectorias y las fechas en las que suceden cada maniobra, sumado a parámetros importantes, como son los incrementos de velocidad, tanto totales como parciales.

Para una misma simulación se han podido **exportar múltiples casos, todos ellos optimizados en función del ΔV**. Estos resultados permitirán evaluar la viabilidad de una trayectoria en fases iniciales del diseño de una misión interplanetaria.

También se ha podido **comparar misiones simuladas** en el software de optimización **respecto a diversos casos reales** como las misiones Mariner 10, Voyager 1, Galileo, Cassini y JUICE. Con ellas se ha podido observar que el funcionamiento del programa y comparar sus resultados para la validación del software.

En la siguiente tabla se mostrará una comparativa entre diferentes misiones y ΔV. Por un lado, **ΔV mejor solución** representa el mejor resultado ΔV de las simulaciones llevadas a cabo por este software propio.

ΔV secuencia similar representará el mejor resultado ΔV obtenido del software de las simulaciones llevadas con la misma secuencia o parecida a la misión original.

ΔV misión representará el resultado ΔV de la misión original. Para ello se puede obtener esta información de dos formas, un método sería el aportado en la misión Galileo, a través

de un artículo, o el otro caso, una simulación parametrizada por la propia ESA en `Pykep`.

Misión	ΔV mejor solución (km/s)	ΔV secuencia similar (km/s)	ΔV misión (km/s)
Mariner 10	9,745	10,296	-
Voyager 1	21,663	26,622	-
Voyager 2	22,289	23,719	-
Galileo	8,857	14,725	18,7
Cassini	10,761	18,829	13,415
JUICE	10,191	11,846	11,568

Tabla 6.1: Comparativa de las misiones realizadas

Este trabajo demuestra como en cuestión de minutos se pueden obtener soluciones iniciales para un problema complejo como es la optimización de trayectorias interplanetarias, logrando resultados coherentes y del orden respecto a la misión real, siendo valiosos para comenzar a planificar de manera más detallada la trayectoria que se considere.

Estas **diferencias en los ΔV pueden deberse a diversos factores**, como no considerar efectos como perturbaciones en el caso de la comparativa de Galileo, o, simulaciones mucho más optimizadas y con diferentes parámetros seleccionados como es el caso de la comparativa de Cassini y JUICE.

Sin embargo, la **falta de accesibilidad a los resultados reales** de las misiones impiden una mejor comparativa que permita determinar de manera más fiable si el desempeño del software desarrollado podría ser más adecuado, valorando posibles mejoras para implementar y poder alcanzar las soluciones más reales posibles

En resumen, la optimización en el diseño de trayectorias espaciales constituye no solo una necesidad técnica, sino también una oportunidad para mejorar la eficiencia de las misiones espaciales ya en el diseño inicial de la misión. Gracias a la combinación de ahorro de peso, conocimiento de tiempos de vuelo y un mayor número de iteraciones, se puede avanzar hacia misiones más eficientes y exitosas.

6.2. Propuestas de mejora

Tras estas conclusiones, se plantean diferentes propuestas de mejora que podrían hacerse en futuros trabajos. La principal es tener un algoritmo más optimizado que permita contemplar un mayor número de problemas y tipos de trayectorias para alcanzar los objetivos que se quieran entre diferentes planetas u otros astros.

El motivo reside en las limitaciones actuales tanto del código desarrollado como en `Pykep`. Como se ha podido visualizar a través de los ejemplos y del código disponible, la librería desarrollada por la ESA está pensada para desarrollo de misiones con un alcance no mayor de Saturno para su visualización.

Tampoco se ha podido **implementar el uso de maniobras de asistencia gravitatoria alrededor de astros**, como asteroides, como se ha llegado a realizar en la actualidad, ni satélites. Por ejemplo, la misión JUICE contempla un flyby Tierra Luna, que aunque será muy similar a un flyby solo de Tierra, sería interesante poder contar con dicha posibilidad.

También, se han enfrentado problemas a la hora de visualizar algunas trayectorias, debido a la integración entre `Pykep` y `matplotlib`. Por lo tanto, otra propuesta de mejora, podré ir por el camino de realizar la integración propia para dibujar de manera adecuada todos los casos.

Otro aspecto que supondría una mejora para este software, sería una mejor integración de la interfaz gráfica, incluyendo más posibilidades de personalización. Algunas propiedades de la misión se han puesto por defecto como pueden ser la optimización multiobjetivo o la posibilidad de determinar cómo será la inserción orbital que se busca.

Poder llevar a cabo simulaciones multiobjetivo supondría otra mejora importante, brindando de mayor número de posibilidades al contemplar tanto el incremento de velocidad como el tiempo de vuelo. Esta es una posibilidad que se ofrece de manera nativa en `Pykep` pero que no se implementado en esta versión de software desarrollado.

El software no tiene en cuenta **posibles perturbaciones que existen a la hora de diseñar una misión espacial**. No solo existen perturbaciones gravitatorias, sino también presión solar, resistencia aerodinámica, etc. Estas perturbaciones no dejan de tener un impacto tanto en la trayectorias como en la recogida de datos, tanto para uso científico como para control de misión.

Apéndice A

Códigos Python creados para programa principal

A.1. TrajectoryProblem.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 20 21:25:25 2023

@author: jesusjimenezgranados
"""

from pykep.trajopt import mga, mga_1dsm, pl2pl_N_impulses, lt_margo, direct_pl2pl, mr_lt_nep
from pykep.trajopt import indirect_pt2pt, indirect_or2or, indirect_pt2or

class TrajectoryProblemSolver:

    def __init__(self):

        self.mga_problems = []
        self.mga_1dsm_problems = []
        self.pl2pl_N_impulses_problems = []
        self.lt_margo_problems = []
        self.direct_pl2pl_problems = []
        self.mr_lt_nep_problems = []
        self.indirect_pt2pt_problems = []
        self.indirect_or2or_problems = []
        self.indirect_pt2or_problems = []
        self.trajectory_problems = []

    def mga_problem(self, seq, t0, tof, vinf, max_revs, multi_objective=False,
                          tof_encoding='direct', orbit_insertion=False,
                          e_target=None, rp_target=None):

        problem = mga(seq=seq, t0=t0, tof=tof, vinf=vinf,
                      multi_objective=multi_objective,
                      tof_encoding=tof_encoding,
                      orbit_insertion=orbit_insertion,
                      e_target=e_target,
                      rp_target=rp_target,
```

```
37                          max_revs=0)
38
39          self.mga_problems.append(problem)
40
41      def mga_1dsm_problem(self, seq, t0, tof, vinf, max_revs, eta_lb, eta_ub,
42                               rp_ub, add_vinf_dep=False, add_vinf_arr=True,
43                               tof_encoding='direct', multi_objective=False,
44                               orbit_insertion=False, e_target=None, rp_target=None):
45
46          problem = mga_1dsm(seq=seq, t0=t0, tof=tof, vinf=vinf,
47                             add_vinf_dep=add_vinf_dep,
48                             add_vinf_arr=add_vinf_arr,
49                             multi_objective=multi_objective,
50                             orbit_insertion=orbit_insertion,
51                             e_target=e_target,
52                             rp_target=rp_target,
53                             tof_encoding=tof_encoding,
54                             eta_lb = 0.1,
55                             eta_ub = 0.9,
56                             rp_ub = 30,
57                             max_revs = 0)
58
59          self.mga_1dsm_problems.append(problem)
60
61      def pl2pl_N_impulses_problem(self, start, target, N_max, tof, vinf,
62                                   phase_free=False, multi_objective=False,
63                                   t0=None):
64
65          problem = pl2pl_N_impulses(start=start, target=target, N_max=N_max,
66                                     tof=tof, vinf=vinf,
67                                     phase_free=phase_free,
68                                     multi_objective=multi_objective,
69                                     t0=t0)
70
71          self.pl2pl_N_impulses_problems.append(problem)
72
73      def lt_margo_problem(self, target, n_seg, grid_type, t0, tof, m0, Tmax,
74                               Isp, earth_gravity=True, sep=False, start='earth'):
75
76          problem = lt_margo(target=target, n_seg=n_seg, grid_type=grid_type, t0=t0,
77                             tof=tof, m0=m0, Tmax=Tmax, Isp=Isp,
78                             earth_gravity=earth_gravity, sep=sep, start=start)
79
80          self.lt_margo_problems.append(problem)
81
82      def direct_pl2pl_problem(self, p0, pf, mass, thrust, isp, nseg, t0, tof,
83                                   vinf_dep, vinf_arr, hf):
84
85          problem = direct_pl2pl(p0=p0, pf=pf, mass=mass, thrust=thrust, isp=isp,
86                                 nseg=nseg, t0=t0, tof=tof, vinf_dep=vinf_dep,
87                                 vinf_arr=vinf_arr, hf=hf)
88
89          self.direct_pl2pl_problems.append(problem)
90
91      def mr_lt_nep_problem(self, seq, n_seg, t0, leg_tof, rest, mass, Tmax,
92                                Isp, traj_tof, c_tol):
93
94          problem = mr_lt_nep(seq=seq, n_seg=n_seg, t0=t0, leg_tof=leg_tof,
95                              rest=rest, mass=mass, Tmax=Tmax, Isp=Isp,
```

```
96                              traj_tof=traj_tof, c_tol=c_tol)
97
98          self.mr_lt_nep_problems.append(problem)
99
100     def indirect_pt2pt_problem(self, x0, xf, tof, thrust, isp, mu,
101                                freetime=False, alpha=0.0, bound=False,
102                                atol=1e-8, rtol=1e-8):
103
104         problem = indirect_pt2pt(x0=x0, xf=xf, tof=tof, thrust=thrust, isp=isp,
105                                  mu=mu, freetime=freetime, alpha=alpha,
106                                  bound=bound, atol=atol, rtol=rtol)
107
108         self.indirect_pt2pt_problems.append(problem)
109
110     def indirect_or2or_problem(self, elem0, elemf, mass, thrust, isp,
111                                atol=1e-8, rtol=1e-8, tof=[100, 300],
112                                freetime=False, alpha=0.0, bound=False,
113                                mu=398600.0):
114
115         problem = indirect_or2or(elem0=elem0, elemf=elemf, mass=mass,
116                                  thrust=thrust, isp=isp, atol=atol, rtol=rtol,
117                                  tof=tof, freetime=freetime, alpha=alpha,
118                                  bound=bound, mu=mu)
119
120         self.indirect_or2or_problems.append(problem)
121
122     def indirect_pt2or_problem(self, x0, elemf, mass, thrust, isp,
123                                atol=1e-8, rtol=1e-8, tof=[100, 300],
124                                freetime=False, alpha=0.0, bound=False,
125                                mu=398600.0):
126
127         problem = indirect_pt2or(x0=x0, elemf=elemf, mass=mass,
128                                  thrust=thrust, isp=isp, atol=atol, rtol=rtol,
129                                  tof=tof, freetime=freetime, alpha=alpha,
130                                  bound=bound, mu=mu)
131
132         self.indirect_pt2or_problems.append(problem)
133
134     def solve_all_problems(self):
135
136         self.solve_mga_problems()
137         self.solve_mga_1dsm_problems()
138         self.solve_pl2pl_N_impulses_problems()
139         self.solve_lt_margo_problems()
140         self.solve_direct_pl2pl_problems()
141         self.solve_mr_lt_nep_problems()
142         self.solve_indirect_pt2pt_problems()
143         self.solve_indirect_or2or_problems()
144         self.solve_indirect_pt2or_problems()
145
146         for each_problem_solved in self.mga_problems:
147
148             self.trajectory_problems.append(each_problem_solved)
149
150         for each_problem_solved in self.mga_1dsm_problems:
151
152             self.trajectory_problems.append(each_problem_solved)
153
154         for each_problem_solved in self.pl2pl_N_impulses_problems:
```

```
155
156             self.trajectory_problems.append(each_problem_solved)
157
158         for each_problem_solved in self.lt_margo_problems:
159
160             self.trajectory_problems.append(each_problem_solved)
161
162         for each_problem_solved in self.direct_pl2pl_problems:
163
164             self.trajectory_problems.append(each_problem_solved)
165
166         for each_problem_solved in self.mr_lt_nep_problems:
167
168             self.trajectory_problems.append(each_problem_solved)
169
170         for each_problem_solved in self.indirect_pt2pt_problems:
171
172             self.trajectory_problems.append(each_problem_solved)
173
174         for each_problem_solved in self.indirect_or2or_problems:
175
176             self.trajectory_problems.append(each_problem_solved)
177
178         for each_problem_solved in self.indirect_pt2or_problems:
179
180             self.trajectory_problems.append(each_problem_solved)
181
182         return self.trajectory_problems
183
184     def solve_mga_problems(self):
185
186         for idx, problem in enumerate(self.mga_problems, start=1):
187
188             print(f"Solving MGA Problem {idx}")
189             print(f"MGA Problem {idx} Solved!")
190
191     def solve_mga_1dsm_problems(self):
192
193         for idx, problem in enumerate(self.mga_1dsm_problems, start=1):
194
195             print(f"Solving MGA_1DSM Problem {idx}")
196             print(f"MGA_1DSM Problem {idx} Solved!")
197
198     def solve_pl2pl_N_impulses_problems(self):
199
200         for idx, problem in enumerate(self.pl2pl_N_impulses_problems, start=1):
201
202             print(f"Solving PL2PL_N_IMPULSES Problem {idx}")
203             print(f"PL2PL_N_IMPULSES Problem {idx} Solved!")
204
205     def solve_lt_margo_problems(self):
206
207         for idx, problem in enumerate(self.lt_margo_problems, start=1):
208
209             print(f"Solving LT_MARGO Problem {idx}")
210             print(f"LT_MARGO Problem {idx} Solved!")
211
212     def solve_direct_pl2pl_problems(self):
213
```

```
214         for idx, problem in enumerate(self.direct_pl2pl_problems, start=1):
215 
216             print(f"Solving DIRECT_PL2PL Problem {idx}")
217             print(f"DIRECT_PL2PL Problem {idx} Solved!")
218 
219     def solve_mr_lt_nep_problems(self):
220 
221         for idx, problem in enumerate(self.mr_lt_nep_problems, start=1):
222 
223             print(f"Solving MR_LT_NEP Problem {idx}")
224             print(f"MR_LT_NEP Problem {idx} Solved!")
225 
226     def solve_indirect_pt2pt_problems(self):
227 
228         for idx, problem in enumerate(self.indirect_pt2pt_problems, start=1):
229 
230             print(f"Solving INDIRECT_PT2PT Problem {idx}")
231             print(f"INDIRECT_PT2PT Problem {idx} Solved!")
232 
233     def solve_indirect_or2or_problems(self):
234 
235         for idx, problem in enumerate(self.indirect_or2or_problems, start=1):
236 
237             print(f"Solving INDIRECT_OR2OR Problem {idx}")
238             print(f"INDIRECT_OR2OR Problem {idx} Solved!")
239 
240     def solve_indirect_pt2or_problems(self):
241 
242         for idx, problem in enumerate(self.indirect_pt2or_problems, start=1):
243 
244             print(f"Solving INDIRECT_PT2OR Problem {idx}")
245             print(f"INDIRECT_PT2OR Problem {idx} Solved!")
```

A.2. TrajectoryOptimization.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Dec 26 13:05:29 2023

@author: jesusjimenezgranados
"""

import pygmo as pg
import pykep as pk
import numpy as np
import pandas as pd
from pykep.examples import add_gradient, algo_factory
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

class TrajectoryProblemOptimization:

    def __init__(self):

        self.udp_list = []
        self.prob_list = []
        self.pop_list = []
        self.algo_archi_list = []
        self.seq_list = []
        self.DV_list = []
        self.T_list = []
        self.DV_i_list = []

    def optimize_with_nlopt(self, udp, population):

        udp2 = add_gradient(udp, with_grad=True)
        prob = pg.problem(udp2)
        prob.c_tol = 1e-4
        uda = pg.nlopt("slsqp")
        uda.ftol_rel = 1e-8
        uda.xtol_rel = 1e-6
        uda.maxeval = 1000000
        uda2 = pg.mbh(uda, 3, 0.05)
        algo = pg.algorithm(uda2)
        algo.set_verbosity(1)
        pop = pg.population(prob, population)
        pop = algo.evolve(pop)

        return udp, prob, algo, pop

    def optimize_with_archipelago(self, udp, generations=100, islands=8,
                                  island_pop=20, evolve_steps=10):

        prob = pg.problem(udp)
        uda = pg.pso_gen(gen=generations)
        archi = pg.archipelago(algo=uda, prob=prob, n=islands, pop_size=island_pop)
        archi.evolve(evolve_steps)
        archi.wait()

```

```
57          return prob, archi, udp
58
59      def optimize_choice(self, problems, seq):
60
61          for udp in problems:
62
63              if isinstance(udp, pk.trajopt._mga.mga):
64
65                  print("Yes, it's an instance of mga")
66
67                  udp, prob, algo, pop = self.optimize_with_nlopt(udp, population=100)
68
69                  self.seq_list.append(seq)
70                  self.udp_list.append(udp)
71                  self.prob_list.append(prob)
72                  self.pop_list.append(pop)
73                  self.algo_archi_list.append(algo)
74
75                  # Extract the champion solution
76                  champion_solution = pop.champion_x
77
78                  # Calculate total delta-v
79                  total_dv = udp.fitness(champion_solution)
80                  DVlaunch, DVfb, DVarrival, _, _, _, _, _ = udp._compute_dvs(
                        champion_solution)
81                  dv_i = [DVlaunch, DVfb, DVarrival]
82
83                  self.DV_list.append(total_dv)
84                  self.DV_i_list.append(dv_i)
85
86              elif isinstance(udp, pk.trajopt._mga_1dsm.mga_1dsm):
87
88                  prob, archi, udp = self.optimize_with_archipelago(udp, generations=100,
89                                                                     islands=8, island_pop=20,
90                                                                     evolve_steps=10)
91
92                  self.seq_list.append(seq)
93                  self.udp_list.append(udp)
94                  self.prob_list.append(prob)
95                  self.algo_archi_list.append(archi)
96
97                  sols = archi.get_champions_f()
98                  sols_array = np.array(sols)
99                  idx = np.argmin(sols_array)
100
101                 total_dv = udp.fitness(archi.get_champions_x()[idx])
102                 dv_i, _, _, _, _ = udp._compute_dvs(archi.get_champions_x()[idx])
103
104                 self.DV_list.append(total_dv)
105                 self.DV_i_list.append(dv_i)
106
107             else:
108
109                 print("Pasando")
110
111         seq_list_pd = pd.Series(self.seq_list)
112         udp_list_pd = pd.Series(self.udp_list)
113         pop_list_pd = pd.Series(self.pop_list)
114         prob_list_pd = pd.Series(self.prob_list)
```

```
115         algo_archi_list_pd = pd.Series(self.algo_archi_list)
116         DV_list_pd = pd.Series(self.DV_list)
117         DV_i_list_pd = pd.Series(self.DV_i_list)
118 
119         data_opt = pd.DataFrame({"seq": seq_list_pd, "udp": udp_list_pd, "pop":
                pop_list_pd,
120                                  "prob": prob_list_pd, "algo/archi": algo_archi_list_pd,
121                                  "DV_total": DV_list_pd, "DV_i": DV_i_list_pd})
122 
123         return data_opt
```

A.3. TrajectoryIterinator.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jan  3 21:00:57 2024

@author: jesusjimenezgranados
"""

import multiprocessing

import pygmo as pg
import pykep as pk
import numpy as np
import pandas as pd
import os

from pykep.examples import add_gradient, algo_factory
from pykep.planet import jpl_lp, keplerian
from pykep import AU, DEG2RAD, MU_SUN, epoch
from TrajectoryProblem import TrajectoryProblemSolver
from TrajectoryOptimization import TrajectoryProblemOptimization
from itertools import product

class TrajectoryIterinator:

    def __init__(self, seq, t0, tof):

        self.seq = []
        self.t0 = []
        self.tof = []

    def safe_radius_planets(self, seq):

        for i, element in enumerate(seq):

            if isinstance(element, jpl_lp) and element.name == 'mercury':

                mercury = jpl_lp('mercury')
                mercury.safe_radius = 1.05
                seq[i] = mercury

            elif isinstance(element, jpl_lp) and element.name == 'venus':

                venus = jpl_lp('venus')
                venus.safe_radius = 1.05
                seq[i] = venus

            elif isinstance(element, jpl_lp) and element.name == 'earth':

                earth = jpl_lp('earth')
                earth.safe_radius = 1.05
                seq[i] = earth

            elif isinstance(element, jpl_lp) and element.name == 'mars':

                mars = jpl_lp('mars')
```

```
57                mars.safe_radius = 1.05
58                seq[i] = mars
59
60            elif isinstance(element, jpl_lp) and element.name == 'jupiter':
61
62                jupiter = jpl_lp('jupiter')
63                jupiter.safe_radius = 1.1
64                seq[i] = jupiter
65
66            elif isinstance(element, jpl_lp) and element.name == 'saturn':
67
68                saturn = jpl_lp('saturn')
69                saturn.safe_radius = 1.1
70                seq[i] = saturn
71
72            elif isinstance(element, jpl_lp) and element.name == 'uranus':
73
74                uranus = jpl_lp('uranus')
75                uranus.safe_radius = 1.1
76                seq[i] = uranus
77
78            elif isinstance(element, jpl_lp) and element.name == 'neptune':
79
80                neptune = jpl_lp('neptune')
81                neptune.safe_radius = 1.1
82                seq[i] = neptune
83
84    def iterinator(self, seq, t0, tof, max_repetitions=3, max_dimension=3):
85
86        # Define planets and their safe radii
87        planet_safe_radii = {
88            'mercury': 1.05,
89            'venus': 1.05,
90            'earth': 1.05,
91            'mars': 1.05,
92            'jupiter': 1.1,
93            'saturn': 1.1,
94            'uranus': 1.1,
95            'neptune': 1.1
96            }
97
98        # Get the first and last planets (departure and arrival)
99        departure_planet, arrival_planet = seq[0], seq[-1]
100
101       planet_list = list(planet_safe_radii.keys())
102
103       index_of_arrival_planet = planet_list.index(arrival_planet.name)
104       index_of_departure_planet = planet_list.index(departure_planet.name)
105
106       # Define the excluded planets based on the arrival planet
107
108       excluded_planets = planet_list[index_of_arrival_planet:]
109
110       if arrival_planet.name == 'mercury':
111           if 'venus' in excluded_planets:
112               excluded_planets.remove('mercury')
113               excluded_planets.remove('venus')
114       else:
115           pass
```

```
116
117         print(excluded_planets)
118
119         pre_arrival_planet = planet_list[index_of_arrival_planet-1]
120
121         for _ in range(max_repetitions):
122             for dimension in range(1, max_dimension + 1):
123                 for new_planet_names in product(planet_safe_radii.keys(), repeat=
                        dimension):
124
125                     new_planet_names = list(new_planet_names)
126                     max_dimension_list = len(new_planet_names)
127
128                     index_of_last_planet = planet_list.index(new_planet_names[-1])
129
130                     if any(planet in excluded_planets for planet in new_planet_names):
131
132                         continue
133
134                     if dimension > 1:
135
136                         index_of_previous_planet = planet_list.index(new_planet_names
                            [-2])
137                         energia_innecesaria = 0
138
139                         for i in range(0, max_dimension_list-1):
140
141                             index_planet_1 = planet_list.index(new_planet_names[i])
142                             index_planet_2 = planet_list.index(new_planet_names[i+1])
143
144                             if index_planet_1 >= index_of_departure_planet and
                                index_planet_1 > index_planet_2:
145
146                                 energia_innecesaria = 1
147
148                                 break
149
150                             else:
151
152                                 pass
153
154                         if energia_innecesaria == 1:
155
156                             continue
157
158                         elif any(planet in excluded_planets for planet in
                            new_planet_names):
159
160                             continue
161
162                         else:
163
164                             for i in range(max_dimension_list):
165                                 new_planet_names[i] = jpl_lp(new_planet_names[i])
166
167                             # Create a new sequence with the new planets
168                             new_seq = seq[:1] + new_planet_names + seq[-1:]
169
170                             # Ensure the new sequence has the correct departure and
```

```
                            arrival planets
171                         new_seq[0] = jpl_lp(departure_planet)
172                         new_seq[-1] = jpl_lp(arrival_planet)
173
174                         # Calculate data_opt_seq for each new sequence
175                         data_opt_seq = self.problem_choice(new_seq, t0, tof)
176
177                         # Yield the new sequence and data_opt_seq
178                         yield new_seq, data_opt_seq
179
180                     elif dimension == 1:
181
182                         for i in range(max_dimension_list):
183                             new_planet_names[i] = jpl_lp(new_planet_names[i])
184
185                         # Create a new sequence with the new planets
186                         new_seq = seq[:1] + new_planet_names + seq[-1:]
187
188                         # Ensure the new sequence has the correct departure and arrival
                                planets
189                         new_seq[0] = jpl_lp(departure_planet)
190                         new_seq[-1] = jpl_lp(arrival_planet)
191
192                         # Calculate data_opt_seq for each new sequence
193                         data_opt_seq = self.problem_choice(new_seq, t0, tof)
194
195                         # Yield the new sequence and data_opt_seq
196                         yield new_seq, data_opt_seq
197
198     def get_names(self, seq):
199         return [element.name for element in seq]
200
201     def problem_choice(self, seq, t0, tof):
202
203         solver = TrajectoryProblemSolver()
204
205         # Add MGA problem
206         solver.mga_problem(
207             seq=seq,
208             t0=t0,
209             tof=tof,
210             vinf=2.5,
211             multi_objective=False,
212             tof_encoding = 'alpha',
213             orbit_insertion=False,
214             e_target=None,
215             rp_target=None,
216             max_revs = 0
217             )
218
219         # Add MGA_1DSM problem
220         solver.mga_1dsm_problem(
221             seq = seq,
222             t0=t0,
223             tof=tof,
224             vinf=[0.5, 2.5],
225             add_vinf_dep = False,
226             add_vinf_arr = True,
227             tof_encoding = 'alpha',
```

```
228             multi_objective = False,
229             orbit_insertion = False,
230             e_target = None,
231             rp_target = None,
232             eta_lb = 0.1,
233             eta_ub = 0.9,
234             rp_ub = 30,
235             max_revs = 0
236             )
237 
238         problems = solver.solve_all_problems()
239 
240         optimizer = TrajectoryProblemOptimization()
241 
242         # Optimize
243 
244         new_seq_list = TrajectoryIterinator.get_names(self, seq)
245 
246         data_opt = optimizer.optimize_choice(problems, new_seq_list)
247 
248         return data_opt
```

A.4. TrajectoryMin5.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed May 22 20:35:43 2024

@author: jesusjimenezgranados
"""

import pygmo as pg
import pykep as pk
import numpy as np
import pandas as pd
import os

from pykep.examples import add_gradient, algo_factory
from pykep.planet import jpl_lp, keplerian
from pykep import AU, DEG2RAD, MU_SUN, epoch
from TrajectoryProblem import TrajectoryProblemSolver
from TrajectoryOptimization import TrajectoryProblemOptimization
from itertools import product


class TrajectoryMin5:

    def __init__(self, data_opt):

        self.data_opt = pd.DataFrame()

    def best_5_results(self, data_opt):

        # Add some dummy rows to the data_opt DataFrame if it has fewer than 5 rows
        if len(data_opt) < 5:
            diff = 5 - len(data_opt)
            dummy_rows = pd.DataFrame(columns=data_opt.columns)
            data_opt = pd.concat([data_opt, dummy_rows]*diff, ignore_index=True)

        # Sort the data_opt DataFrame by the DV column in ascending order and select the top 5 rows
        best_5_results = data_opt.sort_values(by='DV_total').head(5)

        return best_5_results
```

A.5. TrajectoryVisualizer.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed May 22 23:13:04 2024

@author: jesusjimenezgranados
"""

import pygmo as pg
import pykep as pk
import numpy as np
import pandas as pd
import os

from pykep.examples import add_gradient, algo_factory
from pykep.planet import jpl_lp, keplerian
from pykep import AU, DEG2RAD, MU_SUN, epoch
from TrajectoryProblem import TrajectoryProblemSolver
from TrajectoryOptimization import TrajectoryProblemOptimization
from itertools import product
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt


class TrajectoryVisualizer:

    def __init__(self, data_opt):

        self.data_opt = pd.DataFrame()

    def solution(self, data_opt, option):

        data_opt_DV_total = data_opt.sort_values(by='DV_total')
        data_opt_DV_total.reset_index(drop=True, inplace=True)
        solution = data_opt_DV_total.iloc[option-1]

        return solution

    def visualizer(self, solution):

        udp_visualizar = solution['udp']
        pop_visualizar = solution['pop']
        archi_visualizar = solution['algo/archi']

        if str(udp_visualizar) == "MGA_1DSM Trajectory":

            sols = archi_visualizar.get_champions_f()
            sols_array = np.array(sols)
            idx = np.argmin(sols_array)

            fig = plt.figure()
            ax = fig.add_subplot(projection='3d')
            udp_visualizar.plot(archi_visualizar.get_champions_x()[idx], ax)

        elif str(udp_visualizar) == "MGA Trajectory":
```

```
57
58              # We plot
59              '''mpl.rcParams['legend.fontsize'] = 10
60
61              # Create the figure and axis
62
63
64              fig = plt.figure(figsize = (16,5))
65              ax1 = fig.add_subplot(1, 3, 1, projection='3d')
66              udp_visualizar.plot(pop_visualizar.champion_x, axes = ax1)
67
68              ax2 = fig.add_subplot(1, 3, 2, projection='3d')
69              ax2.view_init(90, 0)
70              udp_visualizar.plot(pop_visualizar.champion_x, axes = ax2)
71
72              ax3 = fig.add_subplot(1, 3, 3, projection='3d')
73              ax3.view_init(0,0)
74              udp_visualizar.plot(pop_visualizar.champion_x, axes = ax3)
75              '''
76
77              fig = plt.figure()
78              ax = fig.add_subplot(projection='3d')
79              udp_visualizar.plot(pop_visualizar.champion_x, ax)
80
81          else:
82
83              pass
```

A.6. Interfaz TFM.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri May 24 20:25:57 2024

@author: jesusjimenezgranados
"""

from mainTFM import mainTFM
from PyQt5.QtWidgets import QFileDialog,QMainWindow, QApplication,QVBoxLayout,QPushButton
    ,QComboBox, QListWidget,QListWidgetItem
from PyQt5 import uic,QtCore,QtWidgets,QtGui
from TrajectoryVisualizer import TrajectoryVisualizer
import numpy as np
import pandas as pd
import sys

class VentanaSecundaria(QtWidgets.QMainWindow):

    def __init__(self, param):

        super().__init__()
        uic.loadUi('Stellar_aux.ui', self)

        seq_visualizar = param['seq']
        DV_visualizar = param['DV_total']
        DVi_visualizar = param['DV_i']

        seq_planets = [planet.capitalize() for planet in seq_visualizar]

        resultado_seq = '-'.join(seq_planets)
        resultado_DVi = ' m/s, '.join(map(str,DVi_visualizar)) + ' m/s'

        texto_seq = 'El orden de la secuencia de la mision es: ' + f'{resultado_seq}' +
            '\n'
        texto_DV = 'El incremento de velocidad total necesario es: ' + f'{DV_visualizar
            [0]} ' + 'm/s' + '\n'
        texto_DVi = 'Los incrementos de velocidades necesarios son: ' + f'{resultado_DVi
            }' + '\n'

        texto = texto_seq + texto_DV + texto_DVi

        self.textEdit_show_DV.setText(texto)

class Interfaz_TFM(QMainWindow):

    def __init__(self):

        super(Interfaz_TFM, self).__init__()
        uic.loadUi("Stellar.ui", self)

        self.show()
        self.pushButton_init.clicked.connect(self.inicio)

    def transformar_datos(self, datos):

```

```
53          acortados = [planeta[:3].capitalize() for planeta in datos]
54
55          # Unir los nombres acortados con guiones
56          resultado = '-'.join(acortados)
57          return resultado
58
59      def pasaOpcion(self, option_name, data_opt):
60
61          def handler():
62
63              print(f'Se selecciono la opcion: {option_name}')
64              visualizer = TrajectoryVisualizer(data_opt)
65              solution = visualizer.solution(data_opt, int(option_name))
66
67              try:
68
69                  visualizer.visualizer(solution)
70
71              except:
72
73                  pass
74
75              self.ventana_secundaria = VentanaSecundaria(solution)
76              self.ventana_secundaria.show()
77
78          return handler
79
80      def inicio(self):
81
82          planeta_origen = self.comboBox_planeta_origen.currentText()
83
84          planeta_destino = self.comboBox_planeta_destino.currentText()
85
86          fecha_salida = self.date_salida.dateTime().toSecsSinceEpoch()
87
88          fecha_salida_limite = self.date_salida_limite.dateTime().toSecsSinceEpoch()
89
90          fecha_llegada = self.date_llegada.dateTime().toSecsSinceEpoch()
91
92          n_flybys = self.comboBox_flybys.currentText()
93
94          n_iteraciones = self.comboBox_iteraciones.currentText()
95
96          main = mainTFM(planeta_origen, planeta_destino, fecha_salida,
                fecha_salida_limite,
97                         fecha_llegada, n_flybys, n_iteraciones)
98
99          main.ejecucion()
100
101         #options = ["1", "2", "3", "4", "5"]
102         options = range(1, len(main.data_opt)+1)
103
104
105
106         self.seleccion_graf_sol.clear()
107
108         for option in options:
109
110             seq_natural = main.data_opt.sort_values(by='DV_total').iloc[option-1]['seq']
```

```
111             seq_button = self.transformar_datos(seq_natural)
112
113             item = QListWidgetItem(self.seleccion_graf_sol)
114             option_button = QPushButton("Opcion " + str(option) + "\n" + f'{seq_button}'
                    )
115             option_button.clicked.connect(self.pasaOpcion(option, main.data_opt))
116
117             current_size_hint = item.sizeHint()
118             new_width = current_size_hint.width() + 100
119             new_height = current_size_hint.height() + 50
120             new_size_hint = QtCore.QSize(new_width, new_height)
121             item.setSizeHint(new_size_hint)
122
123             self.seleccion_graf_sol.setItemWidget(item, option_button)
124
125
126 def main():
127
128     app = 0
129     app = QApplication([])
130     #app = QtWidgets.QApplication(sys.argv)
131     window = Interfaz_TFM()
132     window.show()
133
134     sys.exit(app.exec())
135
136 if __name__ == '__main__':
137     main()
```

A.7. mainTFM.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat May 25 11:44:47 2024

@author: jesusjimenezgranados
"""

import pygmo as pg
import pykep as pk
import numpy as np
import pandas as pd
import os

from pykep.examples import add_gradient, algo_factory
from pykep.planet import jpl_lp, keplerian
from pykep import AU, DEG2RAD, MU_SUN, epoch
from TrajectoryProblem import TrajectoryProblemSolver
from TrajectoryOptimization import TrajectoryProblemOptimization
from TrajectoryIterinator import TrajectoryIterinator
from TrajectoryMin5 import TrajectoryMin5
from TrajectoryVisualizer import TrajectoryVisualizer
from itertools import product

class mainTFM:

    def __init__(self, planeta_origen, planeta_destino, fecha_salida,
                 fecha_salida_limite, fecha_llegada, n_flybys, n_iteraciones):

        self.planeta_origen = planeta_origen
        self.planeta_destino = planeta_destino
        self.fecha_salida = fecha_salida
        self.fecha_salida_limite = fecha_salida_limite
        self.fecha_llegada = fecha_llegada
        self.n_flybys = n_flybys
        self.n_iteraciones = n_iteraciones

    def change_planet_name(self, planet):

        if planet == "Venus":

            planet_new_name = "venus"

            return planet_new_name

        elif planet == "Mercurio":

            planet_new_name = "mercury"

            return planet_new_name

        elif planet == "Tierra":

            planet_new_name = "earth"

            return planet_new_name
```

```
57
58          elif planet == "Marte":
59
60              planet_new_name = "mars"
61
62              return planet_new_name
63
64          elif planet == "Jupiter":
65
66              planet_new_name = "jupiter"
67
68              return planet_new_name
69
70          elif planet == "Saturno":
71
72              planet_new_name = "saturn"
73
74              return planet_new_name
75
76          elif planet == "Urano":
77
78              planet_new_name = "uranus"
79
80              return planet_new_name
81
82          elif planet == "Neptuno":
83
84              planet_new_name = "neptune"
85
86              return planet_new_name
87
88          else:
89
90              pass
91
92      def change_epoch_to_date(self, date):
93
94          date_20000101 = 946684800
95          day_in_seconds = 86400
96
97          new_date = (date - date_20000101)/day_in_seconds
98
99          return new_date
100
101     def calculate_tof(self, date1, date2):
102
103         tof = date2 - date1
104
105         return tof
106
107     def ejecucion(self):
108
109         seq = [jpl_lp(self.change_planet_name(self.planeta_origen)), jpl_lp('mercury'),
110                jpl_lp(self.change_planet_name(self.planeta_destino))]
111         t0 = [epoch(self.change_epoch_to_date(self.fecha_salida)),
112               epoch(self.change_epoch_to_date(self.fecha_salida_limite))]
113         tof = [abs(self.calculate_tof(self.change_epoch_to_date(self.fecha_salida),
114                                       self.change_epoch_to_date(self.fecha_salida_limite))),
115                abs(self.calculate_tof(self.change_epoch_to_date(self.fecha_salida),
```

```
116                                 self.change_epoch_to_date(self.fecha_llegada))))]
117
118         print(tof)
119
120         itera = TrajectoryIterinator(seq, t0, tof)
121         data_opt = pd.DataFrame(columns=["seq", "udp", "pop", "prob", "algo/archi", "
                DV_total", "DV_i"])
122
123         itera.safe_radius_planets(seq)
124
125         print(tof)
126
127         new_seq_list = itera.get_names(seq)
128         print('\n')
129         print('The following is a list of the planets involved in the maneuver before
                the loop starts')
130         print(new_seq_list, '\n')
131
132         for new_seq, data_opt_seq in itera.iterinator(seq, t0, tof,
133                                                       max_repetitions=int(self.
                                                            n_iteraciones),
134                                                       max_dimension=int(self.n_flybys)):
135
136             new_seq_list = itera.get_names(new_seq)
137             data_opt = pd.concat([data_opt, data_opt_seq], axis=0)
138
139             print('\n')
140             print('The following is a list of the planets involved in the maneuver')
141
142             print(new_seq_list, '\n')
143
144             print('Now, the problems are resolved for previous planet sequence')
145
146             print(data_opt_seq)
147
148             # Save the DataFrame to a CSV file
149             data_opt_seq.to_csv('output.csv', mode='a', header=not os.path.exists('
                    output.csv'))
150
151         data_opt.reset_index(inplace=False)
152
153         self.data_opt = data_opt
154
155         min5 = TrajectoryMin5(data_opt)
156
157         best_5_results = min5.best_5_results(data_opt)
158
159         print(best_5_results['udp'])
```

Apéndice B

Códigos Python para comprobación

B.1. cassini2.py

```
from pykep.trajopt import mga_1dsm, launchers
from pykep.planet import jpl_lp

import pygmo as pg
from pykep import epoch
from pykep.planet import jpl_lp
from pykep.trajopt import mga_1dsm, mga
import matplotlib.pyplot as plt
from pykep.examples import add_gradient, algo_factory

import numpy as np
from numpy.linalg import norm
from math import log, acos, cos, sin, asin, exp
from copy import deepcopy

# CASSINI2 (we need to modify the safe radius of the planets to match the wanted problem)
_earth_cassini2 = jpl_lp('earth')
_earth_cassini2.safe_radius = 1.15
_venus_cassini2 = jpl_lp('venus')
_venus_cassini2.safe_radius = 1.05
_jupiter_cassini2 = jpl_lp('jupiter')
_jupiter_cassini2.safe_radius = 1.7
_seq_cassini2 = [_earth_cassini2,
                 _venus_cassini2,
                 _venus_cassini2,
                 _earth_cassini2,
                 _jupiter_cassini2,
                 jpl_lp('saturn')]


class _cassini2_udp(mga_1dsm):
    """
    Write Me
    """
    def __init__(self):
        """
```

```
37          Write Me
38          """
39          super().__init__(
40              seq =_seq_cassini2,
41              t0 = [-1000, 0],
42              tof = [[100, 400], [100, 500], [30, 300], [400, 1600], [800, 2200]],
43              vinf = [3., 5.],
44              add_vinf_dep = False,
45              add_vinf_arr = True,
46              tof_encoding = "direct",
47              multi_objective = False,
48              orbit_insertion=True,
49              e_target=0.98,
50              rp_target=108950000,
51              eta_lb = 0.01,
52              eta_ub = 0.9,
53              rp_ub = 70.
54          )
55
56      def get_name(self):
57          return "Cassini 2 (Trajectory Optimisation Gym P11)"
58
59      def __repr__(self):
60          return self.get_name()
61
62  # Problem P11: Cassini mission MGA1DSM, single objective, direct encoding
63  cassini2 = _cassini2_udp()
64
65  if __name__ == "__main__":
66
67
68      udp = _cassini2_udp()
69
70      population=100
71      generations=100
72      islands=8
73      island_pop=20
74      evolve_steps=10
75
76      prob = pg.problem(udp)
77      uda = pg.pso_gen(gen=generations)
78      archi = pg.archipelago(algo=uda, prob=prob, n=islands, pop_size=island_pop)
79      archi.evolve(evolve_steps)
80      archi.wait()
81
82      sols = archi.get_champions_f()
83      sols_array = np.array(sols)
84      idx = np.argmin(sols_array)
85
86      dvi, _, _, _, _ = udp._compute_dvs(archi.get_champions_x()[idx])
87
88      total_dv = sum(dvi)
89
90      print("DV: ", total_dv)
91
92      print(dvi)
93
94      udp.fitness(archi.get_champions_x()[idx])
95      udp.pretty(archi.get_champions_x()[idx])
```

B.2. juice.py

```
from pykep.trajopt import mga_1dsm, launchers
from pykep.planet import jpl_lp

import pygmo as pg
from pykep import epoch
from pykep.planet import jpl_lp
from pykep.trajopt import mga_1dsm, mga
import matplotlib.pyplot as plt
from pykep.examples import add_gradient, algo_factory

import numpy as np
from numpy.linalg import norm
from math import log, acos, cos, sin, asin, exp
from copy import deepcopy


class _juice_udp(mga_1dsm):
    """

    """

    def __init__(self, multi_objective, tof_encoding, tof):
        """
        Args:

        """
        # Redefining the planets as to change their safe radius
        earth = jpl_lp('earth')
        earth.safe_radius = 1.05
        # We need the Earth eph in the fitness
        venus = jpl_lp('venus')
        venus.safe_radius = 1.05
        mars = jpl_lp('mars')
        mars.safe_radius = 1.05
        jupiter = jpl_lp('jupiter')

        super().__init__(
            seq=[earth, earth, venus, earth, mars, earth, jupiter],
            t0=[8000, 8400],
            tof=tof,
            vinf=[1., 4.],
            add_vinf_dep=False,
            add_vinf_arr=True,
            tof_encoding=tof_encoding,
            multi_objective=multi_objective,
            orbit_insertion=True,
            e_target=0.98531407996358,
            rp_target=1070400000,
            eta_lb=0.01,
            eta_ub=0.99,
            rp_ub=10)

    def fitness(self, x):
        T, Vinfx, Vinfy, Vinfz = self._decode_times_and_vinf(x)
        # We transform it (only the needed component) to an equatorial system rotating
            along x
```

```
56          # (this is an approximation, assuming vernal equinox is roughly x and the
                ecliptic plane is roughly xy)
57          earth_axis_inclination = 0.409072975
58          # This is different from the GTOP tanmEM problem, I think it was bugged there as
                the rotation was in the wrong direction.
59          Vinfz = - Vinfy * sin(earth_axis_inclination) + Vinfz * cos(
                earth_axis_inclination)
60          # And we find the vinf declination (in degrees)
61          sindelta = Vinfz / x[3]
62          declination = asin(sindelta) / np.pi * 180.
63          # We now have the initial mass of the spacecraft
64          m_initial = launchers.ariane5(x[3] / 1000., declination)
65          # And we can evaluate the final mass via Tsiolkowsky
66          Isp = 312.
67          g0 = 9.80665
68
69          if self._multi_objective:
70              DV, T = super().fitness(x)
71          else:
72              DV, = super().fitness(x)
73
74          DV = DV + 275.  # losses for 5 swingbys + insertion
75          m_final = m_initial * exp(-DV / Isp / g0)
76          # Numerical guard for the exponential
77          if m_final == 0:
78              m_final = 1e-320
79
80          encoded_m_final = -log(m_final)
81
82          if self._multi_objective:
83              return (encoded_m_final, T)
84
85          return (encoded_m_final,)
86
87      def get_name(self):
88          return "Juice (Trajectory Optimization Gym P13-14)"
89
90      def __repr__(self):
91          return self.get_name()
92
93      def get_extra_info(self):
94          retval = "\t Sequence: " + \
95              [pl.name for pl in self._seq].__repr__()
96          return retval
97
98      def pretty(self, x):
99          """
100         prob.plot(x)
101
102         - x: encoded trajectory
103
104         Prints human readable information on the trajectory represented by the decision
                vector x
105
106         Example::
107
108           print(prob.pretty(x))
109         """
110         super().pretty(x)
```

```
111         T, Vinfx, Vinfy, Vinfz = self._decode_times_and_vinf(x)
112         # We transform it (only the needed component) to an equatorial system rotating
                along x
113         # (this is an approximation, assuming vernal equinox is roughly x and the
                ecliptic plane is roughly xy)
114         earth_axis_inclination = 0.409072975
115         # This is different from the GTOP tanmEM problem, I think it was bugged there as
                the rotation was in the wrong direction.
116         Vinfz = - Vinfy * sin(earth_axis_inclination) + Vinfz * cos(
                earth_axis_inclination)
117         # And we find the vinf declination (in degrees)
118         sindelta = Vinfz / x[3]
119         declination = asin(sindelta) / np.pi * 180.
120         m_initial = launchers.ariane5(x[3] / 1000., declination)
121         # And we can evaluate the final mass via Tsiolkowsky
122         Isp = 312.
123         g0 = 9.80665
124         DV = super().fitness(x)[0]
125         DV = DV + 275.  # losses for 5 swgbys + insertion
126         m_final = m_initial * exp(-DV / Isp / g0)
127         print("\nInitial mass:", m_initial)
128         print("Final mass:", m_final)
129         print("Declination:", declination)
130
131
132 # Problem P13: JUICE mission MGA1DSM, single objective, direct encoding
133 juice = _juice_udp(
134     multi_objective=False,
135     tof_encoding='direct',
136     tof=[[200, 500], [30, 300], [200, 500], [30, 300], [500, 800], [900, 1200]])
137
138 # Problem P14: JUICE mission MGA1DSM, multiple objective, alpha encoding
139 juice_mo = _juice_udp(
140     multi_objective=True,
141     tof_encoding='alpha',
142     tof=[2000, 3000])
143
144 if __name__ == "__main__":
145
146
147     juice = _juice_udp(
148         multi_objective=False,
149         tof_encoding='alpha',
150         tof=[2000, 3000])
151
152     udp = juice
153     population=100
154     generations=100
155     islands=8
156     island_pop=20
157     evolve_steps=10
158
159     prob = pg.problem(udp)
160     uda = pg.pso_gen(gen=generations)
161     archi = pg.archipelago(algo=uda, prob=prob, n=islands, pop_size=island_pop)
162     archi.evolve(evolve_steps)
163     archi.wait()
164
165     sols = archi.get_champions_f()
```

```
166     sols_array = np.array(sols)
167     idx = np.argmin(sols_array)
168 
169     dvi, _, _, _, _ = udp._compute_dvs(archi.get_champions_x()[idx])
170 
171     total_dv = sum(dvi)
172 
173     print("DV: ", total_dv)
174 
175     print(dvi)
176 
177     juice.fitness(archi.get_champions_x()[idx])
178     juice.pretty(archi.get_champions_x()[idx])
```

Bibliografía

Armano, M. (2023). *Apuntes de la asignatura. Diseño Avanzado de Vehículos Espaciales (DAVE).*

Biscani, F. and Izzo, D. (2020). A parallel global multiobjective framework for optimization: pagmo. *Journal of Open Source Software*, 5(53):2338.

Confederación Española de Sociedades de Física (2023). Naturaleza de las Órbitas elípticas.

Cornish, N. J. (July 2012). Observatory at l2. Part of WMAP's education and outreach program.

Daglis, I. (1999-2000). Lecture notes 11-3: Space physics.

D´Amario, L. A., Bright, L. E., and Wolf, A. A. (1992). Galileo trajectory design. *ssr*, 60(1-4):23–78.

European Space Agency (2022). Juice's journey to jupiter.

European Space Agency (Year of publication if available). Juice mission.

Falcó, J. (2021). Construcción de cónicas.

Galperin, A. and Gurfil, P. (2015). Closed-form solutions for optimal orbital transfers around oblate planets. *The Journal of the Astronautical Sciences*, 61.

GeeksforGeeks (Year of publication if available). Python itertools.

Hosseini, S. (2011). An exploration of fuel optimal two-impulse transfers to cyclers in the earth-moon system.

Izzo, D. (2012). Pygmo and PyKEP: Open Source Tools for Massively Parallel Optimization in Astrodynamics (the case of interplanetary trajectory optimization). International Conference on Astrodynamics Tools.

Izzo, D. (2015). Revisiting Lambert's problem. *Celestial Mechanics and Dynamical Astronomy*, 121(1):1–15.

Izzo, D. (2023). Pykep: scientific library developed at the european space agency (ESA) to provide basic tools for astrodynamics research.

MathWorks (Year of publication if available). Hohmann transfer with the spacecraft dynamics block.

NASA (1992). The galileo spacecraft: Heat redistribution, mass spectrometry, and doppler effects. Technical report, NASA Technical Reports Server (NTRS).

NASA (2019). 45 years ago: Mariner 10 first to explore mercury.

NASA (2022). Perseverance's Route to Mars.

NASA Jet Propulsion Laboratory (Año no definido). Voyager mission.

NASA National Space Science Data Center (NSSDC) (1996). Galileo.

NASA Science (Año no definido). Galileo mission.

NASA Science (Not availableb). Cassini trajectory.

NASA Science (Year of publication if availablea). Cassini gravity assists. Accessed on Date accessed.

Orbital Mechanics (Year of publication if available). Phasing maneuvers in orbital mechanics.

P, J. S. (2022). Python pyqt tutorial.

Pelegrina, J. M. A. (2015). La danza de los dos cuerpos.

Python Software Foundation (Year of publication if availablea). itertools — functions creating iterators for efficient looping.

Python Software Foundation (Year of publication if availableb). os — miscellaneous operating system interfaces.

Romeu, J. L. (2004). Understanding series and parallel systems reliability. *Selected Topics in Assurance Related Technologies (START), Department of Defense Reliability Analysis Center (DoD RAC)*, 11(5):1–8.

Shenand, H. and Tsiotras, P. (2003). Optimal two-impulse rendezvous using multiple-revolution lambert solutions. *Journal of Guidance Control and Dynamics - J GUID CONTROL DYNAM*, 26:50–61.

Sánchez, D. M. (2022). Diseño preliminar de una misión interplanetaria a saturno y sus lunas. Trabajo de Fin de Máster, Máster Universitario en Ingeniería Aeronáutica, Curso 2021-2022.

Tatum, J. B. and Montenegro, C. E. (2012). *Determinación de Órbitas. Método de Gauss para Órbitas Elípticas*. Universidad de Victoria, Observatorio Astronómico Prof. Victorio Capolongo, Columbia Británica, Canadá and Rosario, Santa Fe, Argentina.

The Cython Project (Year of publication if available). Cython, c-extensions for python.

The Matplotlib Development Team (Year of publication if available). Matplotlib: Visualization with python.

The Statsmodels Development Team (Year of publication if available). Statsmodels, statistical models in python.

Valhondo, V. (2021). Designing interplanetary orbits.

Vallado, D. A. (2001). *Fundamentals of astrodynamics and applications*, volume 12. Springer Science & Business Media.

Vallat, C. (2018). Presentation at opag meeting, february 2018.

Wikipedia (Año no definido). Luna 3.

Zaborsky, S. (2015). Geometrical characteristics of two-impulse transfer between coplanar elliptical orbits. *Journal of Guidance, Control, and Dynamics*, 38:1–5.