Trabajo Fin de Grado
Grado en Ingeniería Aeroespacial

Development of a Python-Based Orbital Mechanics Laboratory Class on Lambert's Problem

Autor: Lucas Reino Diez
Tutor: Rafael Vázquez Valenzuela

Dpto. Ingeniería Aeroespacial y Mecánica de Fluidos
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado

Grado en Ingeniería Aeroespacial

Development of a Python-Based Orbital Mechanics Laboratory Class on Lambert's Problem

Autor:

Lucas Reino Diez

Tutor:

Rafael Vázquez Valenzuela

Profesor Titular

Dpto. Ingeniería Aeroespacial y Mecánica de Fluidos

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado:	Development of a Python-Based Orbital Mechanics Laboratory Class on Lambert's Problem

Autor:	Lucas Reino Diez
Tutor:	Rafael Vázquez Valenzuela

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Resumen

En la asignatura de *Mecánica Orbital y Vehículos Espaciales* impartida en el cuarto curso del Grado en Ingeniería Aeroespacial se proporcionan las bases necesarias para afrontar el cálculo de trayectorias espaciales. Sin embargo, para realizar el análisis y optimización de una misión interplanetaria (o en general, de cualquier transferencia entre dos órbitas genéricas) surge la necesidad de resolver el *Problema de Lambert*. El objetivo de este material es proporcionar al alumno en formato de práctica de la asignatura unas pautas para programar en Python un algoritmo que permita resolver de forma numérica dicho problema. Con esta nueva herramienta, se procederá a obtener ciertos resultados que permiten ahondar un poco más en un campo tan interesante como es el de misiones espaciales.

Abstract

At the *Orbital Mechanics and Space Vehicles* course imparted in the fourth year of the Bachelor in Aerospace Engineering, the basis necessary for the calculation of space trajectories are given. However, to perform the analysis and optimization of an interplanetary mission (or generally, of any transfer between two generic orbits) the need to solve *Lambert's Problem* arises. The goal of this work is to provide the student with the guidelines for coding in Python an algorithm that allows said problem to be solved numerically, in a laboratory class format. With this new tool, certain results will be obtained which allow to delve a bit more into such a fascinating field as interplanetary missions are.

Contents

1 Introduction

1.1 Lambert's Problem

Lambert's problem, sometimes referred to as the orbital boundary problem, is concerned with the determination of an orbit from two position vectors and the time of flight. Since the beginning of the space age with the launch of the satellite Sputnik 1 in 1957, continuing to the present day, Lambert's problem has been fundamental in the areas of rendezvous, targeting and preliminary orbit determination. In the past, from the time of Euler and Lambert, its solution has been essential for obtaining the elements of the orbits of planets and comets from observations. Over the years a variety of techniques and procedures have been developed for solving this problem. Each is characterized by a particular form of the transfer-time equation and a particular choice of the independent variable to be used in an iterative algorithm to determine the orbital elements.

Following the fundamental work laid down, among others, by Euler, Lambert, Lagrange and Gauss, the need of a robust algorithm to function for a wide range of conditions led to revisit the problem during the space age. In this context, the work of Lancaster *et al.* in 1969 [1] needs to be highlighted, as it provided an universal solution to the problem (that is, valid for elliptic, hyperbolic and parabolic orbits) and paved the way for further research. In the following years, a good number of studies were built upon these results. In 1990, Gooding [2] published a procedure which, based on good heuristics for the initial estimate of the independent variable, yielded an accurate value in only three iterations. Further developments have also been made, looking to reduce computational cost and increase accuracy.

Even though multiple other alternative procedures and algorithms exist, in this project the approach of Lancaster *et al.* is followed. Furthermore, as far as the algorithm is concerned, clarity and simplicity will prevail over robustness and efficiency given that students will have to comprehend the procedure and code it themselves in a matter of hours.

In addition to solving the orbital boundary problem, some properties of conic sections and their relevance within this problem are discussed. Perhaps one of the most remarkable theorems in this connection is the one discovered by Lambert, having to do with the dependency of the orbital transfer time upon various geometrical parameters. Lambert's theorem has had a great relevance in orbital mechanics and is intimately linked with the solution to Lambert's problem.

1.2 Learning Python

Python is a high-level programming language whose design emphasizes productivity and code readability. Its clear, easy to understand syntax and object-oriented approach aim to help programmers, data scientists and engineers write comprehensible and logical code for small and big-scale projects. It was created in the late eighties by Guido Van Rossum. However, it has only been in the past decade when its popularity has sky-rocketed due to the development of artificial intelligence and the non-stopping growth of data science, consolidating as the fastest-growing major programming language in these past years. Nowadays it stands as one of the top programming languages, along with JavaScript, Java or C++.

One of the aims of this project is to introduce the students to Python. As of now, in the Bachelor of Aerospace Engineering at the University of Seville only MATLAB is taught as part of the academic program. It is of great interest for the students to learn multiple languages, and Python constitutes a great start due to its simplicity compared to other alternatives. For this reason, it has been chosen for coding the algorithm which solves Lambert's problem.

Another great advantage of Python is that its releases are open source, which means everybody can access the source code and use the latest version of the language at zero cost. Other popular languages for computational science, such as MATLAB, are included in proprietary software and do not share these characteristics. Additionally, Python offers an extensive number of libraries such as numpy, scipy and the plotting package matplotlib, which are introduced in subsequent chapters.

There are two versions of the Python language being used nowadays: Python 2 and Python 3. The changes in Python 3 were introduced to address shortcomings in the design of the language that were identified since Python's inception. These changes are however very slight, and the differences are minimal. However, in this project Python 3.7.4 is used, as it is the latest stable version. To ease the students approach to this new language, an Integrated Development Environment (IDE) similar to that of MATLAB is also used.

1.3 Scope of this project

"The discussion of the relations of two or more places of a heavenly body in its orbit as well as in space, furnishes an abundance of elegant propositions, such as might easily fill an entire volume. But our plan does not extend so far as to exhaust this fruitful subject, ..."

CARL FRIEDRICH GAUSS, 1809

Quoting Gauss from his book *Theoria Motus* [3], it is clear that the two-body problem hides a great amount of remarkable and elegant properties. However, given the short span of time in which this project is bound to to be finished and in line with Gauss' remark, the concepts developed along the document are those strictly necessary to approach the various subjects treated and fulfill our final purpose, which is to design a two-hour laboratory class for the students enrolled in the course *Orbital Mechanics and Space Vehicles.*

The main objectives pursued by this laboratory class are therefore, to:

- Teach basic Python
- Introduce Lambert's problem
- Be understandable given the knowledge of the students
- Have both theoretical content and a practical application
- Have a duration of approximately two hours

1.4 Document structure

This document is structured in clearly differentiated chapters. On Chapter 2, everything relative to Lambert's problem is discussed. Firstly, we derive some fundamental equations and establish the basic knowledge necessary to address the problem. After that, a brief recap on some geometrical properties of conics is done. With all this background information, a solution to Lambert's problem is obtained following the approach taken by Lancaster *et al.*.

On Chapter 3 we introduce a set of tools required for the analysis of interplanetary missions, which allow us to deal with real world examples and obtain reasonable results. Among them, planetary ephemerides coupled to a trajectory propagator and the concept of porkchop plots.

Chapter 4 is dedicated exclusively to Python. Here, a more detailed introduction to the language is going be made. All the necessary packages and modules are discussed, and the basic commands are reviewed. Furthermore, a few notions on the different data types and structures is given, and an in-depth comparison between MATLAB and Python is made.

After learning about Lambert's problem, its solution and the Python language, on Chapter 5 we devise an algorithm to solve the problem in a numerical manner. This algorithm is then implemented in Python and a demo of the laboratory class is made, showing all the results which are expected to be obtained later by the student.

On a final chapter, the conclusions obtained from this project are laid out. We also analyze possible future work to be made on the subject.

1.4.1 Structure of the laboratory class

Said class will have a duration of two hours, along which Lambert's Problem will be introduced. The laboratory report will consist of a series of examples, which include code that the students may reproduce on their own, interspersed with the required theoretical contents. Additionally, the document will be split into three main parts:

- Programming with Python
- Lambert's problem
- Analysis and optimization of interplanetary missions

In the first part, a brief introduction to Python will be made so that the students get acquainted with the language and the IDE (Integrated Development Environment) of choice. A few simple examples with some relation to orbital mechanics will be given at this stage to aid the student in the understanding of Python programming.

After this introduction, a detailed step-by-step solution to Lambert's Problem will be given, following the approach taken in Sect. 2.4, and its solution thoroughly analyzed. Before diving into the analysis and optimization of interplanetary missions, the algorithm which solves Lambert's problem will need to be programmed in Python.

The last part will consist mainly of two examples regarding spacecraft missions from Earth to Mars, including a brief theoretical section introducing pork-chop plots and the optimization of interplanetary missions through the characteristic energy (C_3) (reviewed in Sect. 3.5). Finally, some further insight on the Mars launch opportunity windows will be given, which allow to plan future missions ahead.

At this stage, the teacher can propose an (optional) additional problem for the students to finish at home. All the tools required to solve this problem would have been reviewed during the laboratory class —this provides endless possibilities for the teacher, as these tools are not only prepared for an Earth-Mars mission, but include generic algorithms and data from every Major Planet within the Solar System.

2 Lambert's Problem

Since the times of the ancient Greeks and throughout the years, orbital mechanics have caught the attention of many of the greatest minds. Both physicists and mathematicians have been intrigued by the fascinating properties of celestial bodies, and it has only been in the last few centuries where great progress has been made. From Kepler to Gauss, including Newton, Euler, Lagrange and Lambert among others, many have contributed with their efforts to this subject. In the past century, with the emergence of space exploration, a great amount of research has also been made. Nowadays, the technological advances have led to an increase in the number of interplanetary missions carried out every year.

Just like Kepler's equation, the solution to Lambert's problem lays at the very heart of the most fundamental orbital mechanics and space engineering questions. At the beginning of this chapter, the basic equations required to face this subject are introduced. Later on, we address the two-body orbital boundary problem an the procedure necessary to obtain a solution.

A great number of books have been written in the past years on these topics. The contents presented in the following section have been inspired by the class notes from the *Orbital Mechanics and Space Vehicles* course imparted by Rafael Vázquez Valenzuela at the University of Seville [4], and the work laid out by Battin [5] and Curtis [6].

2.1 Basic orbital mechanics

In 1687, Newton showed in his book *Principia* [7] that the attraction of a homogeneous (by layers) sphere on a particle is the same as if the mass of the sphere were concentrated at its center. Based on the fundamental laws found on the same book, the force of attraction between two isolated bodies can be expressed as

$$\mathbf{F}_1 = \frac{G m_1 m_2}{r^2}\frac{\mathbf{r}}{r} = m_1 \ddot{\mathbf{R}}_1, \qquad \mathbf{F}_2 = -\mathbf{F}_1 = m_2 \ddot{\mathbf{R}}_2, \tag{2.1}$$

where G is the gravitational constant, m_1 and m_2 are the masses of the bodies, and $\mathbf{R}_1$ and $\mathbf{R}_2$ are the position vectors in relation to an arbitrary reference origin, as shown in Fig. 2.1a. Additionally, $\mathbf{r}$ is the vector going from the first body to the second one, and $r = \|\mathbf{r}\|$.

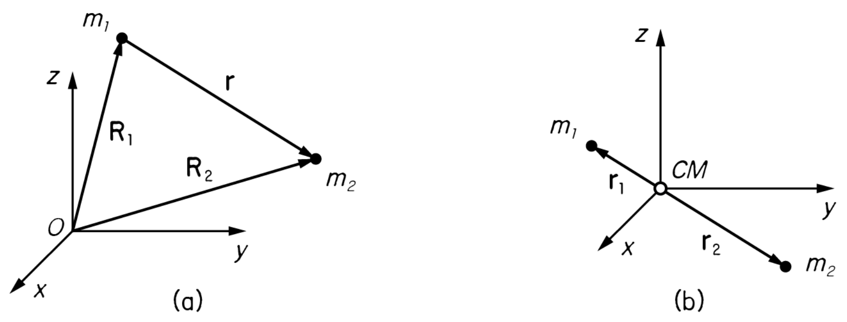


Figure 2.1 Diagram showing the frames of reference used.

If we compute the position of the center of mass of the system and find its acceleration:

$$\mathbf{R}_{CM} = \frac{m_1\mathbf{R}_1 + m_2\mathbf{R}_2}{m_1 + m_2}, \qquad \ddot{\mathbf{R}}_{CM} = \frac{m_1\ddot{\mathbf{R}}_1 + m_2\ddot{\mathbf{R}}_2}{m_1 + m_2} = \mathbf{0} \tag{2.2}$$

it is obtained that it either stays still or drifts through space at a constant speed, due to its acceleration being zero. Therefore, the frame of reference can be displaced so that its origin lays at the center of mass while still being an inertial frame of reference (Fig. 2.1b). This property simplifies greatly the formulation of the equations that follow.

2.1.1 The equation of motion and conserved quantities

A classical approach to the two-body problem is to consider one mass much larger than the other, that is, $m_1 \gg m_2$. In most cases this is a reasonable assumption to make; the mass of the Sun is a few million times larger than Earth's, and the mass of the International Space Station (or any regular satellite) is over a billion times smaller. Additionally, throughout the entirety of this document all possible disturbances to the system are neglected. That includes those generated by the gravity of external bodies, the non-homogeneity and non-spherical shape of the masses studied, solar radiation pressure and atmospheric drag. The non-perturbed two-body problem is also known in classical mechanics as *Kepler's problem.*

Based on these assumptions, on an isolated system where $m_1 \gg m_2$

$$\mathbf{r}_1 = -\frac{m_2}{m_1 + m_2}\mathbf{r} \approx \mathbf{0}, \qquad \mathbf{r}_2 = \frac{m_1}{m_1 + m_2}\mathbf{r} \approx \mathbf{r}, \tag{2.3}$$

and therefore $\ddot{\mathbf{R}}_{CM} = \ddot{\mathbf{R}}_1$. The equation of motion of the system is then

$$\ddot{\mathbf{r}} = \ddot{\mathbf{R}}_2 - \ddot{\mathbf{R}}_1 = -G(m_1 + m_2)\frac{\mathbf{r}}{r^3} = -\mu\frac{\mathbf{r}}{r^3}, \tag{2.4}$$

where μ is the standard gravitational parameter of the system. When $m_1 \gg m_2$, then $\mu \approx \mu_1 = Gm_1$, which is that of the first body alone. For several objects in the Solar System, the value of μ is known to greater accuracy than either G or m as it can be measured by observational astronomy alone.

As a natural consequence of the equation of motion there are a few quantities that are conserved throughout the movement, which appear in what physicists refer to as *first integrals.* In our case, the state of an orbiting body at any given time is defined by the orbiting body's position and velocity with respect to the central body. This can be represented by the three-dimensional Cartesian coordinates (represented by x, y, and z) and the similar Cartesian components of the orbiting body's velocity (v_x, v_y and v_z). Therefore the configuration of two gravitationally interacting bodies constitutes a mechanical system with six degrees of freedom, and a maximum of six functionally independent first integrals can be obtained.

Before proceeding let us first define $\mathbf{v} = \dot{\mathbf{r}}$, keeping in mind that $v = \|\mathbf{v}\| \neq \dot{r}$. Additionally,

$$\mathbf{r} \cdot \mathbf{r} = r^2,$$

so that

$$\frac{d}{dt}(\mathbf{r} \cdot \mathbf{r}) = 2r\frac{dr}{dt}.$$

But

$$\frac{d}{dt}(\mathbf{r} \cdot \mathbf{r}) = \mathbf{r} \cdot \frac{d\mathbf{r}}{dt} + \frac{d\mathbf{r}}{dt} \cdot \mathbf{r} = 2\mathbf{r} \cdot \frac{d\mathbf{r}}{dt}.$$

Thus, we obtain the important identity

$$\mathbf{r} \cdot \mathbf{v} = r\dot{r}. \tag{2.5}$$

Recall as well the vector identity known as the ***bac-cab*** rule:

$$\mathbf{A} \times (\mathbf{B} \times \mathbf{C}) = \mathbf{B}(\mathbf{A} \cdot \mathbf{C}) - \mathbf{C}(\mathbf{A} \cdot \mathbf{B}). \tag{2.6}$$

Carrying out the dot product of (2.4) with $\mathbf{v}$ yields

$$\ddot{\mathbf{r}} \cdot \mathbf{v} = -\mu\frac{\mathbf{r}}{r^3} \cdot \mathbf{v} \quad \Longrightarrow \quad \frac{1}{2}\frac{d\mathbf{v}^2}{dt} = -\frac{\mu}{r^3}\frac{1}{2}\frac{d\mathbf{r}^2}{dt}$$

$$\Longrightarrow \quad \frac{dv^2}{dt} = -\mu\frac{2\dot{r}}{r^2} \quad \Longrightarrow \quad \frac{d}{dt}\left(\frac{v^2}{2}\right) = 2\mu\frac{d}{dt}\left(\frac{1}{r}\right) = \frac{d}{dt}\left(\frac{\mu}{r}\right),$$

which leads to

$$\frac{v^2}{2} - \frac{\mu}{r} = \text{const.} = \epsilon, \tag{2.7}$$

a scalar first integral which defines the ***specific energy*** ϵ (a name given by the units of the equation, $m^2/s^2 = J/kg$). (2.7) is a statement of conservation of energy, as $v^2/2$ is the relative kinetic energy per unit of mass and $(-\mu/r)$ is the potential energy per unit mass of the body m_2 in the gravitational field of m_1. This equation is also known as the ***vis-viva*** ("living force") equation.

Taking now the cross product of $\mathbf{r}$ with (2.4):

$$\mathbf{r} \times \ddot{\mathbf{r}} = -\mu\frac{\mathbf{r} \times \mathbf{r}}{r^3} = \mathbf{0} \quad \Longrightarrow \quad \frac{d}{dt}(\mathbf{r} \times \mathbf{v}) = \mathbf{0},$$

therefore

$$\mathbf{r} \times \mathbf{v} = \mathbf{const.} = \mathbf{h}. \tag{2.8}$$

The cross product of $\mathbf{r}$ with $\mathbf{v}$, $\mathbf{h}$, corresponds with the definition of ***specific angular momentum.***

It follows that $\mathbf{r}\cdot\mathbf{h}=\mathbf{0}$ and $\mathbf{v}\cdot\mathbf{h}=\mathbf{0}$, which implies that the motion is confined to the plane perpendicular to $\mathbf{h}$ and containing the origin.

By carrying out the cross product of (2.4) with h, we obtain

$$\begin{aligned}\ddot{\mathbf{r}}\times\mathbf{h}=-\mu\frac{\mathbf{r}\times\mathbf{h}}{r^3} \quad\Longrightarrow\quad \frac{d}{dt}(\mathbf{v}\times\mathbf{h}) &= -\frac{\mu}{r^3}(\mathbf{r}\times\mathbf{h}) = -\frac{\mu}{r^3}[\mathbf{r}\times(\mathbf{r}\times\mathbf{v})]\\ &= -\frac{\mu}{r^3}[\mathbf{r}(\mathbf{r}\cdot\mathbf{v})-\mathbf{v}(\mathbf{r}\cdot\mathbf{r})]\\ &= -\frac{\mu}{r^3}[\mathbf{r}(r\dot{r})-\mathbf{v}r^2]\\ &= -\mu\frac{\mathbf{r}\dot{r}-\mathbf{v}r}{r^2}\\ &= \mu\frac{d}{dt}\left(\frac{\mathbf{r}}{r}\right),\end{aligned}$$

that is,

$$\mathbf{v}\times\mathbf{h}-\mu\frac{\mathbf{r}}{r}=\mathbf{const.}=\mu\mathbf{e}. \tag{2.9}$$

The dimensionless vector $\mu\mathbf{e}$ is sometimes called the *Laplace vector.* We shall, instead, use the terminology *eccentricity vector* for $\mathbf{e}$ since its magnitude e is the *eccentricity* of the orbit. The line defined by $\mathbf{e}$ is commonly called the *line of apsides*, and an it lays on the orbital plane. In order to obtain a scalar equation, let us take the dot product of (2.9) with $\mathbf{r}$:

$$\mathbf{r}\cdot(\mathbf{v}\times\mathbf{h})-\mu\frac{\mathbf{r}\cdot\mathbf{r}}{r}=\mu\mathbf{e}\cdot\mathbf{r},$$

where

$$\mathbf{r}\cdot(\mathbf{v}\times\mathbf{h})=\mathbf{h}\cdot(\mathbf{r}\times\mathbf{v})=\mathbf{h}\cdot\mathbf{h}=h^2.$$

Therefore

$$h^2-\mu r=\mu e r\cos\theta, \tag{2.10}$$

in which θ is the *true anomaly*, defined as the angle between the fixed vector $\mathbf{e}$ and the variable position vector $\mathbf{r}$. We may write (2.10) as

$$r=\frac{h^2/\mu}{1+e\cos\theta}=\frac{p}{1+e\cos\theta}, \tag{2.11}$$

where p is simply known as the *parameter.* This is the orbit equation in polar coordinates, and it defines the path of the body m_2 around m_1, relative to m_1. Clearly the orbit is symmetrical about the line of apsides. Furthermore, the orbit is bounded if $e<1$ an unbounded if $e\geq 1$. Interestingly, (2.11) represents a conic section or simply a *conic.* Namely, it represents a *circle* when $e=0$, an *ellipse* when $e<1$, a *parabola* when $e=1$ or an *hyperbola* when $e>1$.

(2.7)-(2.9) constitute seven scalar first integrals. The specific energy ϵ and the vectors $\mathbf{h}$ and $\mathbf{e}$ together determine the size, shape and orientation of the orbit with respect to the frame of reference. Their components provide seven scalar constants of integration of the two-body equation of motion. However, this constants are not independent as it would render the problem over-determined. An important relationship is revealed by calculating the magnitude of the eccentricity vector from (2.9):

$$e^2=\mathbf{e}\cdot\mathbf{e}=\frac{1}{\mu^2}(\mathbf{v}\times\mathbf{h})\cdot(\mathbf{v}\times\mathbf{h})-\frac{2}{\mu r}\mathbf{r}\cdot(\mathbf{v}\times\mathbf{h})+1,$$

but

$$(\mathbf{v}\times\mathbf{h})\cdot(\mathbf{v}\times\mathbf{h}) = \mathbf{v}\cdot\mathbf{h}\times(\mathbf{v}\times\mathbf{h}) = h^2v^2,$$

since **h** and **v** are orthogonal and

$$\mathbf{r}\cdot(\mathbf{v}\times\mathbf{h}) = \mathbf{h}\cdot(\mathbf{r}\times\mathbf{v}) = h^2,$$

which leads to

$$1-e^2 = \frac{h^2}{\mu}\left(\frac{2}{r}-\frac{v^2}{\mu}\right) = p\left(\frac{2}{r}-\frac{v^2}{\mu}\right).$$

The second factor has the dimensions of length^{-1}. It is usual to define

$$a = \left(\frac{2}{r}-\frac{v^2}{\mu}\right)^{-1}, \tag{2.12}$$

which resembles (2.7). In fact, the specific energy equation can be written in the form

$$\epsilon = \frac{v^2}{2}-\frac{\mu}{r} = -\frac{\mu}{2a}. \tag{2.13}$$

Clearly, the quantities p, a and e are then related by

$$p = a(1-e^2). \tag{2.14}$$

It is also shown that $\mathbf{h}\cdot\mathbf{e} = \mathbf{0}$. In summary, for Kepler's problem (formulated by a sixth-order differential system), (2.7)-(2.9) only provide five independent scalar constants of integration. To complete the solution, an additional first integral will be required. More precisely, the relation between the location of the body in orbit and some particular instant of time is missing. This is the subject of Sect. 2.3

2.1.2 Kepler's laws

Tycho Brahe's observations of the planet Mars, whose orbital eccentricity, fortunately, was pronounced, provided Kepler with the means of testing his theories of planetary motion. Between 1609 and 1619 he published the famous *Kepler's laws of planetary motion*, which state that

1. *The orbit of a planet is an ellipse with the Sun at one of the two foci.*

2. *A line segment joining a planet and the Sun sweeps out equal areas during equal intervals of time.*

3. *The square of the orbital period of a planet is directly proportional to the cube of the semi-major axis of its orbit.*

With the results obtained in Sect. 2.1 we can tackle such assertions. Since the orbit equation describes conics, including ellipses, it is a mathematical statement of Kepler's first law (since the ellipse is the only bounded conic and the orbits of the planets are indeed bounded). Recall that this is not true for all orbits, but since the Sun's mass is far greater than that of any other body in the Solar System, the solution obtained is indeed valid. Two-body orbits are often referred to as Keplerian orbits.

Kepler's second law is a direct consequence of (2.8). Let us first express $\mathbf{r}$ and $\mathbf{v}$ in polar coordinates:

$$\mathbf{r} = r\ \mathbf{e}_r, \qquad \mathbf{v} = \dot{\mathbf{r}} = \dot{r}\ \mathbf{e}_r + r\dot{\theta}\ \mathbf{e}_\theta. \tag{2.15}$$

We then find that

$$\mathbf{h} = \mathbf{r} \times \mathbf{v} = r^2\dot{\theta}\ \mathbf{e}_z,$$

while

$$A(t) = \int_0^{\theta(t)} \int_0^{r(t)} r' dr' d\theta' = \int_0^{\theta(t)} \frac{r^2(\theta')}{2} d\theta' \quad \Longrightarrow \quad \frac{dA}{dt} = \frac{r^2\dot{\theta}}{2} = \frac{h}{2} = \text{const.}$$

Since dA/dt is the rate at which the radius vector sweeps out area, we have a verification of Kepler's second law. The module of the specific angular momentum vector is found to be twice the areal velocity.

The *period* of elliptic motion may be obtained from a simple application of Kepler's second law since it is the time required for the radius vector to sweep over the entire closed area. Denoting the period by T and recalling that the area of an ellipse is πab (where a is the semi-major axis [1] and b the semi-minor axis), we have

$$T = \frac{A}{\dot{A}} = \frac{\pi ab}{h/2} = \frac{2\pi ab}{h}. \tag{2.16}$$

Additionally, as it is seen in Sect. 2.2, the semi-minor axis can be expressed in terms of a and the eccentricity of the orbit e:

$$b = a\sqrt{1-e^2}.$$

Substituting this identity and (2.14) into (2.16), we obtain

$$T = \frac{2\pi a^2\sqrt{1-e^2}}{\sqrt{\mu a(1-e^2)}} \quad \Longrightarrow \quad T = 2\pi\sqrt{\frac{a^3}{\mu}}, \tag{2.17}$$

which leads to Kepler's third law:

$$T^2 = 4\pi^2\frac{a^3}{\mu}.$$

2.1.3 Orbital elements and coordinate systems

The solution to Kepler's problem depends on six independent arbitrary constants. These are referred to as the *elements* of the orbit. For practical applications, a set of parameters which provide easy to interpret information about the properties and characteristics of the orbit are preferred. Three possible orbital elements are a, e and θ. They define the conic and the position of the body irrespective of its relation to the frame of reference. Three other quantities are required for the spatial orientation of the orbit. The classical choices for the remaining three elements are the Euler angles. In Fig. 2.2, these angles are illustrated.

[1] The use of a for both the semi-major axis and an energetic constant in (2.12) is not coincidental. In Sect. 2.2 a more detailed explanation can be found.

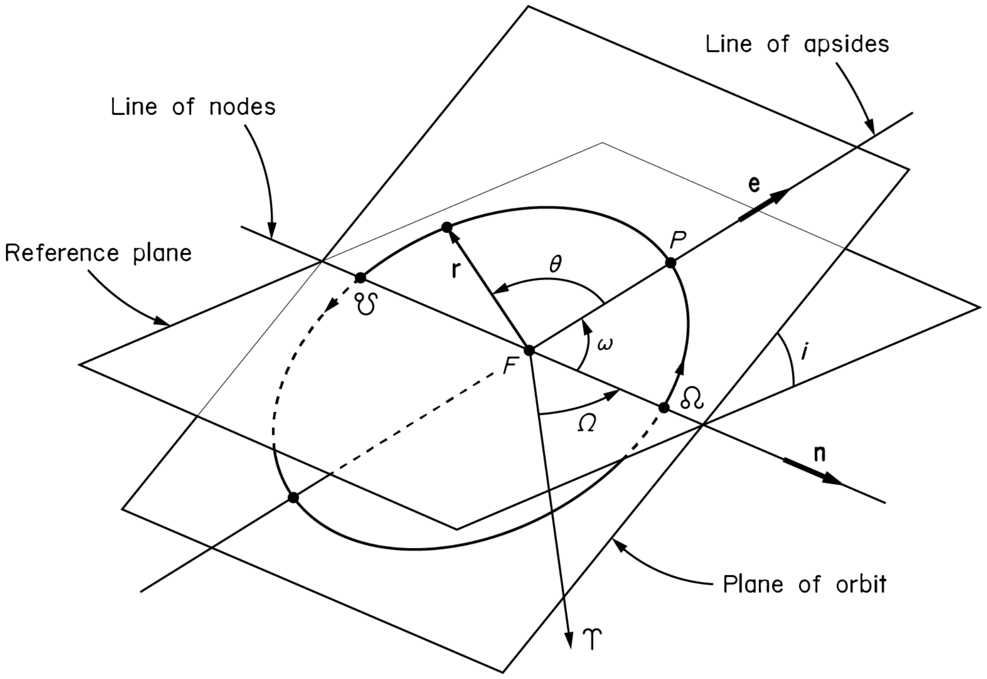


Figure 2.2 Orbital elements.

On one hand, the *angle of inclination* of the orbital plane with respect to a reference plane is symbolized by i. The line of intersection of the reference plane with the plane of orbit is called the *line of nodes*. It passes through the *focus* F of the conic section with a direction given by the nodal vector $\mathbf{n}$. The points where the orbit intersects this line receive the name of *ascending node* ☊ and *descending node* ☋, the ascending node being the point at which the body crosses the reference plane going upwards.

The point P in the figure at which r is minimum receives the name of *periapsis* —an *apsis* being the point in an orbit where the motion is perpendicular to the radius vector. The ellipse has a second apsis called *apoapsis*, where r reaches its maximum value. Both lay on the line of apsides, hence its name. Recall that the line of apsides is defined by the eccentricity vector $\mathbf{e}$ and also contains the focus of the ellipse. The true anomaly θ is the angle between the line of apsides and the position vector $\mathbf{r}$. The periapsis therefore corresponds to $\theta = 0$. Moreover, ω is the angle which the line of apsides makes with the line of nodes, computed from ☊.

On the other hand, Ω is the angle between the line of nodes and a reference direction. The three angles Ω, i and ω are the Euler angles of the orbit. When dealing with Earth's orbit around the Sun, it is usual to take the reference direction towards the *First Point of Aries*, which is where $\mathbf{n}$ points when the Sun crosses the equator during the *vernal equinox*. In an heliocentric reference system, the reference plane is commonly the one in which the Earth orbits the Sun, known as the *ecliptic*.

Typically, the coordinates for the bodies in the Solar System can be either *heliocentric* (Sun-centered) or *geocentric* (Earth-centered). The two fundamental coordinate systems are known as the *ecliptic system* and the *equatorial system*. In this project we will work with an heliocentric ecliptic coordinate system; its origin being the Sun's center, its reference plane the ecliptic, and its primary direction the First Point of Aries as shown on Fig. 2.3.

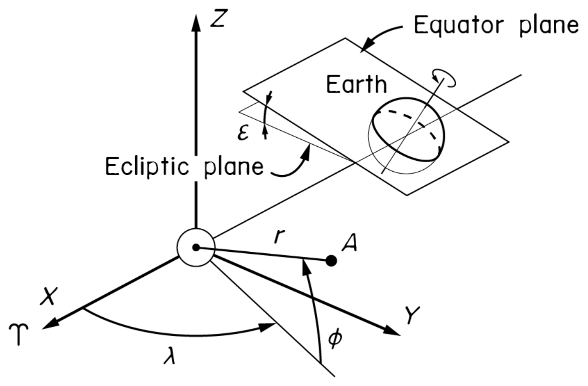


Figure 2.3 Heliocentric ecliptic coordinate system.

The position of a point A in space is determined by the ***heliocentric latitude*** λ, the ***heliocentric longitude*** ϕ and the distance to the Sun r. In the figure, the Sun is represented by the astronomical symbol $\odot$. Usually both the heliocentric latitude and longitude carry the subscript $\odot$ to differentiate them from those used in the geocentric system, centered on the Earth. However, we don't use this coordinate system so the subscript will be dropped. The X and Y axis lay at the ecliptic plane, which in turn forms an angle ε with the equator plane. This angle is referred to as the ***obliquity of the ecliptic*** and takes a value of about $23.4°$.

2.1.4 The perifocal frame

The ***perifocal frame*** is the "natural frame" for an orbit. It is a reference system whose origin lays at the focus of the orbit. Its $\bar{x}\bar{y}$ plane contains the orbit, and the $\bar{x}$ axis goes from the focus through the periapsis. The $\bar{y}$ axis is placed at $\theta = 90°$ following the motion, and the $\bar{z}$ axis is chosen as to obtain a right-handed coordinate system. In other words, $\bar{x}$ and $\bar{z}$ take the direction of **e** and **h** respectively.

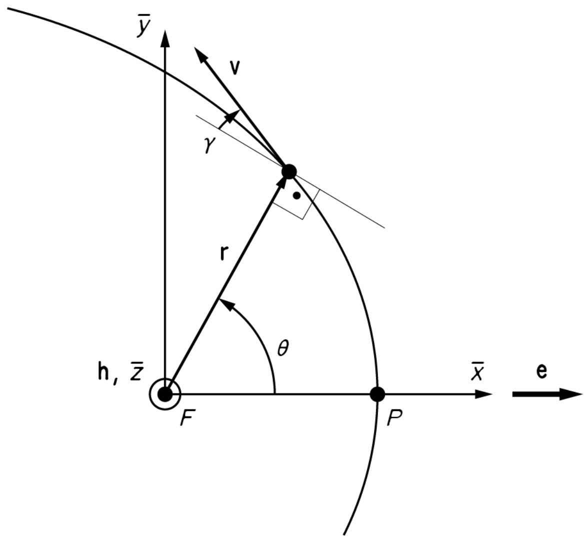


Figure 2.4 Perifocal frame $\bar{x}\bar{y}\bar{z}$.

An alternative reference frame can be obtained if the origin is displaced to the center of the conic section. We will refer to this frame as the xyz frame. In both coordinate systems, the unit vectors will be $\mathbf{e}_x$, $\mathbf{e}_y$ and $\mathbf{e}_z$.

In Fig. 2.4 a diagram showing the position and velocity vectors is showed. Additionally, we define the *flight-path angle* γ as the one that $\mathbf{v}$ makes with the direction perpendicular to $\mathbf{r}$. The velocity vector can be therefore expressed in polar coordinates as

$$\mathbf{v} = v \sin\gamma \ \mathbf{e}_r + v \cos\gamma \ \mathbf{e}_\theta. \tag{2.18}$$

When dealing with Lambert's problem, a two dimensional approach is taken by using the perifocal frame of reference, usually combined with polar coordinates. However, depending on the context, the data for celestial bodies is generally given in either heliocentric or geocentric coordinate systems. Therefore, a means to translate information from one system to the other is required. This is discussed in Chapter 3.

2.2 Geometric properties of conic trajectories

In this section we review some basic geometric properties that become useful in the analysis of Lambert's Problem. Our focus is on conics, namely circles, ellipses, parabolas and hyperbolas.

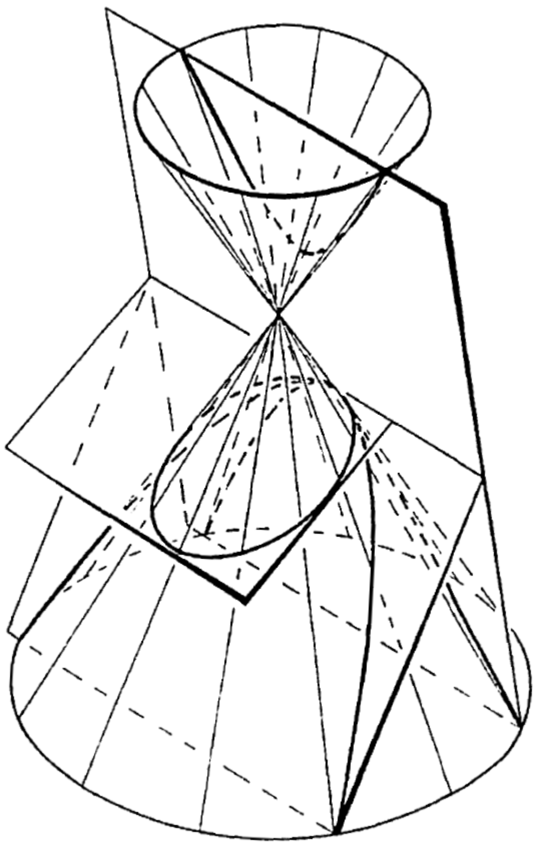


Figure 2.5 An ellipse, parabola or hyperbola can be obtained by intersecting a cone with a plane. [5]

2.2.1 Elliptical and circular trajectories ($0 \leq e < 1$)

We may begin by studying elliptical trajectories. Let F, F' be the foci and C the center of the ellipse shown in Fig. 2.6. Assume that the massive body is located at F, therefore we will refer to this point as the *focus* and to F' as the *vacant focus.*

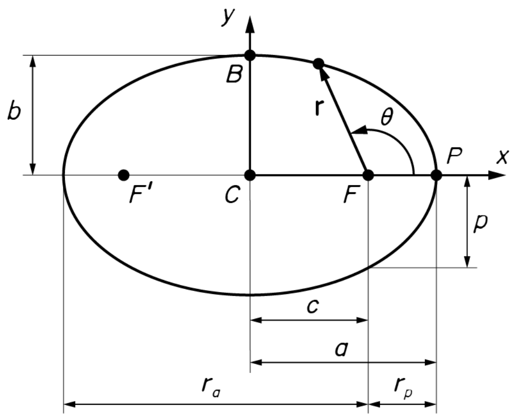


Figure 2.6 Geometric parameters of an ellipse.

An ellipse whose axes of symmetry lay on the x and y Cartesian axes can be fully determined by two parameters. Typically its semi-major axis a and the eccentricity e are chosen. However, there are some other parameters which can become useful under certain circumstances. The distance c between the center and its focus receives the name of *focal distance*, and by definition $c = ae$. Moreover, the *parameter* p gains a geometric interpretation as it becomes the length of the segment which starts vertically from the focus C and ends at the ellipse itself. This length is sometimes referred to as the *semilatus rectum* and can be verified by introducing $\theta = 90°$ in (2.11). The semi-minor axis is denoted by the letter b, and the distances from the periapsis and apoapsis to the focus are r_p and r_a respectively, being

$$r_p = a - c = a(1-e), \qquad r_a = a + c = a(1+e). \tag{2.19}$$

There are multiple ways to geometrically define an ellipse. One of them is to use the orbit equation in polar coordinates (2.11), centered at the focus F. However, a perhaps more *geometric* definition is the following. Given two fixed points F, F' (the foci) and a distance $2a$ which is greater than the distance between the foci ($2c$), the ellipse is the set of points P such that the sum of the distances $|PF|$ and $|PF'|$ is equal to $2a$. Therefore, looking at Fig. 2.6, the triangle $\triangle BCF$ provides the following relation between a, b, and c:

$$a^2 = b^2 + c^2. \tag{2.20}$$

Substituting $c = ae$ in (2.20) reveals an expression of the semi-minor axis b in terms of the semi-major axis and the eccentricity,

$$b = \sqrt{a^2 - a^2e^2} = a\sqrt{1-e^2}. \tag{2.21}$$

The case $F = F'$ yields a circle, which is really an ellipse whose eccentricity is zero. When this occurs, both axis have the same length ($a = b$).

2.2.2 A comment on the energetic properties of conic trajectories

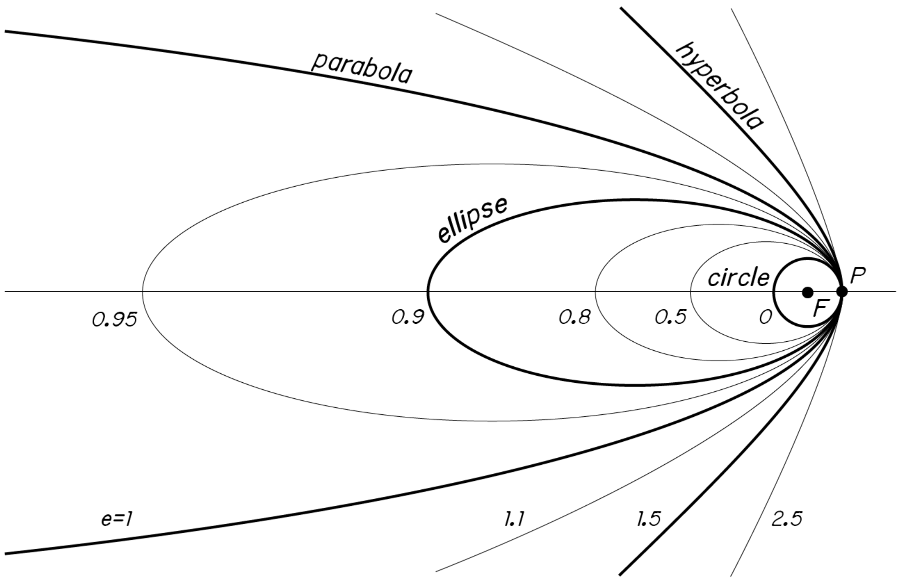


Figure 2.7 Orbits of various eccentricities, having a common focus F and periapsis P.

As seen in Sect. 2.1.1, the specific energy of a two-body orbit is $-\mu/2a$. Different orbits can be classified according to the value of this parameter. Recall that when deriving this formula, a was an arbitrarily defined parameter. From the conservation of the eccentricity vector, the relation (2.14) was also found. It involved a, the ***parameter*** p, and the eccentricity:

$$p = a(1-e^2).$$

However, in Sect. 2.2.1 we consider a to be the semi-major axis of an ellipse. The use of a for both the semi-major axis and an energetic constant is not coincidental. Consider the formula which describes an ellipse centered at the origin of a Cartesian coordinate system

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1. \tag{2.22}$$

To switch between the Cartesian system and a set of polar coordinates centered at the focus F, the following change of variables must be made:

$$x = ae + r\cos\theta, \quad y = r\sin\theta.$$

Substituting this change into (2.22) leads to the familiar formula

$$r = \frac{b\sqrt{1-e^2}}{1+e\cos\theta},$$

which upon being compared with the orbit equation (2.11) reveals an expression of the semi-minor axis b in terms of the semi-major axis and the eccentricity,

$$p = b\sqrt{1-e^2}. \tag{2.23}$$

By substituting (2.21), we arrive to

$$p = a(1-e^2),$$

an identical expression to that involving the energetic constant. Therefore, these two apparently unrelated parameters are indeed equivalent. This applies not only to ellipses, but to any given conical section.

Consequently, a relationship between the energy of an orbit and its *size* exist. In Fig. 2.7, various orbits of different size (proportional to their eccentricity, being r_p fixed) are represented. In Sect. 2.2.4, it will be shown that a takes a negative value for the hyperbola. The parabola then acts as the demarcation between closed, negative energy orbits (ellipses) and open, positive energy orbits (hyperbolas). The *minimum-energy* orbit is an ellipse ($a > 0$) with the smallest possible value of the semi-major axis a.

From the specific energy equation (2.13), we also obtain the speed v of an orbiting body at a distance r from its focus:

$$v = \sqrt{\frac{2\mu}{r} - \frac{\mu}{a}}. \tag{2.24}$$

2.2.3 Parabolic trajectories ($e = 1$)

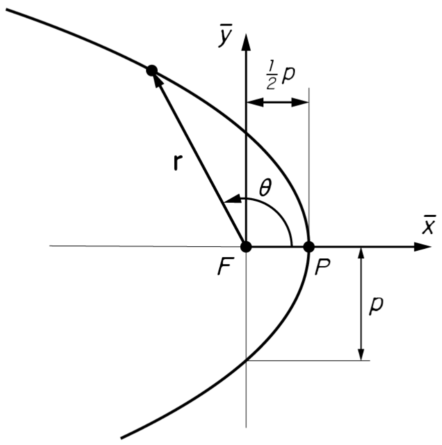


Figure 2.8 Geometric parameters of a parabola.

When the eccentricity equals 1, then the orbit equation becomes

$$r = \frac{p}{1+\cos\theta}. \tag{2.25}$$

As the true anomaly θ approaches 180°, the denominator approaches zero, so that r tends towards infinity. Therefore, $-180° < \theta < 180°$. The *parameter* p still holds its geometric significance, but not the same can be applied to the rest of parameters. For instance, recall that $p = h^2/\mu$ and thus it carries a physical meaning. At the same time, $p = a(1-e^2)$, which due to e being zero can only imply that a tends towards infinity (so an indeterminate form of the type $\infty \times 0$ can be obtained, which results in p taking a finite value). Therefore, a parabola is not a closed trajectory and only the periapsis P can be defined. From (2.25) its coordinates turn out to be $(p/2,\, 0)$.

As a tends towards infinity, the specific energy of a parabolic orbit is zero. In other words, the speed anywhere on a parabolic path is

$$v = \sqrt{\frac{2\mu}{r}}.$$

An orbiting body will coast to infinity, arriving there with zero speed relative to the massive body. It will not return. Parabolic paths are therefore called escape trajectories. The *escape velocity* at a given distance r from the center of attraction is then

$$v_e = \sqrt{\frac{2\mu}{r}}. \tag{2.26}$$

2.2.4 Hyperbolic trajectories ($e > 1$)

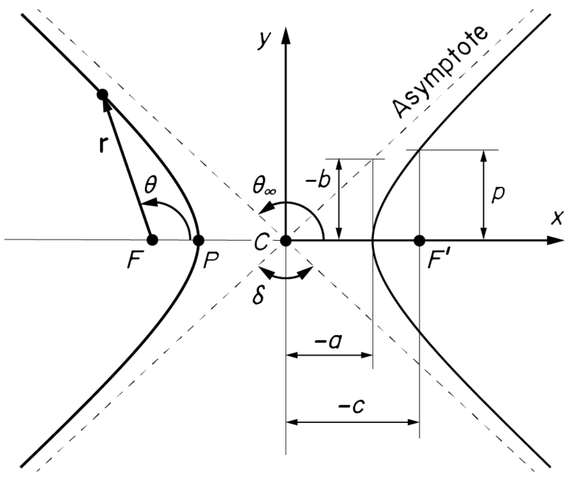


Figure 2.9 Geometric parameters of a hyperbola.

Lastly, if $e > 1$, the orbit equation

$$r = \frac{p}{1 + e\cos\theta} \tag{2.27}$$

describes the geometry of the hyperbola shown in Fig. 2.9. The system consists of two symmetric curves. One of them is occupied by the orbiting body, and the other one is its empty, mathematical image. The vacant trajectory, shown on the right, is physically impossible because it would require a repulsive gravitational force (as the massive body is located at the focus, F).

A hyperbolic orbit has a few particular characteristics that differentiate it from an elliptical or parabolic one. For instance, the denominator of (2.27) will go to zero when $\cos\theta = -1/e$. We denote this value of the true anomaly

$$\theta_\infty = \cos^{-1}(-1/e), \tag{2.28}$$

since the radial distance approaches infinity as the true anomaly approaches θ_∞. From trigonometry it follows that

$$\sin\theta_\infty = \frac{\sqrt{e^2 - 1}}{e}. \tag{2.29}$$

Therefore, the occupied orbit is bounded by two asymptotes (as illustrated in Fig. 2.9) and $-\theta_\infty < \theta < \theta_\infty$. The angle between these asymptotes is called the turn angle δ. As its name implies, it is the angle through which the velocity vector is rotated as the orbiting body comes closer to the attracting body and heads back towards infinity. From the figure we see that

$$\delta = 2\theta_\infty - 180° = 2(\theta_\infty - 90°),$$

which by introducing 2.28 leads to

$$\delta = 2\sin^{-1}(1/e). \tag{2.30}$$

As to the obtaining of the parameters a, b and c, a few discrepancies exist between the approaches taken by different authors, mostly having to do with the sign of each parameter. When dealing with a hyperbola, these quantities are not as intuitively measured as they were in the previous cases. In this work, the approach taken by Vázquez in his class notes [4] is followed and all three parameters turn out to be negative. First, from (2.14) the length $-a$ of the semi-major axis can be obtained. Given that p is strictly positive, a must then take a negative value. Secondly, the vertical distance $-b$ from the periapsis P to an asymptote becomes the semi-minor axis of the hyperbola. From Fig. 2.9, we see that its length is

$$-b = -a\tan(180° - \theta_\infty) = -a\frac{\sin(180° - \theta_\infty)}{\cos(180° - \theta_\infty)} = -a\frac{\sin\theta_\infty}{-\cos\theta_\infty} = -a\frac{\sqrt{e^2-1}/e}{-(-1/e)},$$

so therefore

$$b = a\sqrt{e^2-1}. \tag{2.31}$$

When a and b are equal, the resulting hyperbola is called an *equilateral* or *rectangular* hyperbola, and $e = \sqrt{2}$. To obtain c we simply apply its definition for the elliptic case, $c = ae$, and similarly it represents the distance between the center C and the focus F. Finally, we remark that the geometric significance of the *parameter* p, being the *semilatus rectum*, still remains valid.

Let v_∞ denote the speed at which a body on a hyperbolic path arrives at infinity. According to (2.24)

$$v_\infty = \sqrt{-\frac{\mu}{a}}, \tag{2.32}$$

and it receives the name of *hyperbolic excess velocity*. In terms of v_∞ we may write (2.13) as

$$\frac{v^2}{2} - \frac{\mu}{r} = \frac{v_\infty^2}{2}.$$

Substituting the expression for escape velocity (2.26), we obtain for a hyperbolic trajectory

$$v^2 = v_e^2 + v_\infty^2. \tag{2.33}$$

This equation shows that hyperbolic excess velocity represents the excess specific kinetic energy over that which is required to simply escape from the center of attraction. The square of v_∞ is denoted C_3 ($C_3 = v_\infty^2$), and is known as the *characteristic energy*. C_3 is a measure of the energy required for an interplanetary mission, and it allows for the comparison of different trajectories in energetic terms. To match a launcher with a mission, $C_3|_{\text{launcher}} > C_3|_{\text{mission}}$.

2.2.5 Eccentric and hyperbolic anomalies

A new angle to replace the true anomaly θ is customary for elliptic motion, and helps relating time and position. Let C be the center and F the focus of an ellipse as shown in Fig. 2.10. Construct a circle of center C and radius a. For any given value of θ, a vertical line can be drawn from P (belonging to the ellipse) to Q (belonging to the circle). The angle $\angle QCF$ was called the *eccentric anomaly* by Kepler and is denoted by E.

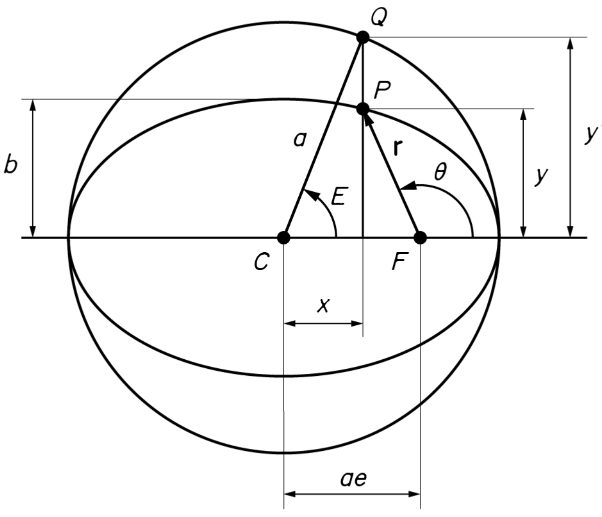


Figure 2.10 Eccentric anomaly.

On a Cartesian coordinate system centered at C, the equation of the ellipse can be expressed in parametric form as

$$x = ae + r\cos\theta, \quad y = r\sin\theta, \tag{2.34a}$$

and in terms of E,

$$x = a\cos E, \quad y = b\sin E = a\sqrt{1-e^2}\sin E. \tag{2.34b}$$

The radial position of the point P is easily expressed in terms of E using (2.34a) and (2.34b):

$$\begin{aligned} r^2 = r^2\sin^2\theta + r^2\cos^2\theta &= a^2[(1-e^2)\sin^2 E + \cos^2 E - 2e\cos E + e^2] \\ &= a^2[1 - 2e\cos E + e^2\cos^2 E] \\ &= [a(1-e\cos E)]^2. \end{aligned}$$

Thus

$$r = a(1-e\cos E), \tag{2.35}$$

and if this is compared with the polar equation of the ellipse

$$r = \frac{a(1-e^2)}{1+e\cos\theta},$$

we obtain the identities

$$\cos\theta = \frac{\cos E - e}{1-e\cos E}, \qquad \sin\theta = \frac{\sqrt{1-e^2}\sin E}{1-e\cos E}. \tag{2.36}$$

From (2.36) an alternative identity can be derived:

$$\tan\frac{\theta}{2} = \frac{\sin\theta}{1+\cos\theta} = \sqrt{\frac{1+e}{1-e}}\frac{\sin E}{1+\cos E} \quad\Longrightarrow\quad \tan\frac{\theta}{2} = \sqrt{\frac{1+e}{1-e}}\tan\frac{E}{2}. \tag{2.37}$$

This is a most useful relation between θ and E, since $\theta/2$ and $E/2$ are always in the same quadrant.

An analogous procedure for hyperbolic motion can be formulated. However, the analysis is more easily accomplished in terms of hyperbolic, rather than trigonometric, functions. Consider the formula which describes an hyperbola centered at the origin of a Cartesian coordinate system,

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1,$$

and the familiar identity

$$\cosh^2 H - \sinh^2 H = 1.$$

The parametric equations of the hyperbola can then be written as

$$x = a\cosh H, \qquad y = b\sinh H, \tag{2.38}$$

and the radial position becomes

$$r = a(1 - e\cosh H). \tag{2.39}$$

Similarly to the elliptic case, it can be shown that the relation between H and the true anomaly is

$$\tan\frac{\theta}{2} = \sqrt{\frac{e+1}{e-1}}\tanh\frac{H}{2}. \tag{2.40}$$

An alternative formulation using trigonometric functions and the *Gudermannian transformation* can be made, but is not as elegant and requires a greater number of steps. However, it does provide a clearer geometric interpretation. For further information, refer to [5].

2.2.6 Geometric representation of E and H

From Sect. 2.1.2, the rate at which the radius vector sweeps out area is

$$\frac{dA}{dt} = \frac{r^2\dot{\theta}}{2},$$

therefore

$$dA = \frac{1}{2}r^2 d\theta.$$

Expressing this equation in Cartesian coordinates we obtain

$$dA = \frac{1}{2}(x dy - y dx).$$

Hence, for the unit circle

$$x^2 + y^2 = 1 \qquad \text{or} \qquad x = \cos E, \quad y = \sin E,$$

and for the unit equilateral hyperbola

$$x^2 - y^2 = 1 \qquad \text{or} \qquad x = \cosh H, \quad y = \sinh H,$$

we have

$$dA = dE/2 \qquad \text{(unit circle)},$$
$$dA = dH/2 \qquad \text{(unit equilateral hyperbola)}.$$

The shaded areas shown in Fig. 2.11 are then equal to $E/2$ and $H/2$ respectively. For a circle, E also corresponds to the physical angle $\angle RCP$. However, H does not have such an interpretation, and this geometric construction is perhaps the only way to visualize it. Trigonometric functions are frequently called *circular functions* and this analogy between circular and and hyperbolic functions is the reason for the designation of the latter as hyperbolic.

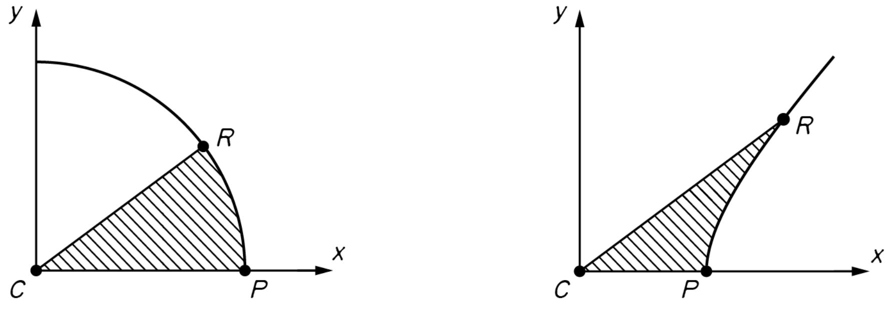


Figure 2.11 Geometric significance of E and H.

From this discussion, it is clear that just as an auxiliary circle is used in the analysis of elliptical orbits, when treating hyperbolic orbits an equilateral hyperbola with identical major axis to the one under consideration should be used. This is shown in Fig. 2.12. When the equilateral parabola is not unitary, the shaded area corresponds to $a^2H/2$.

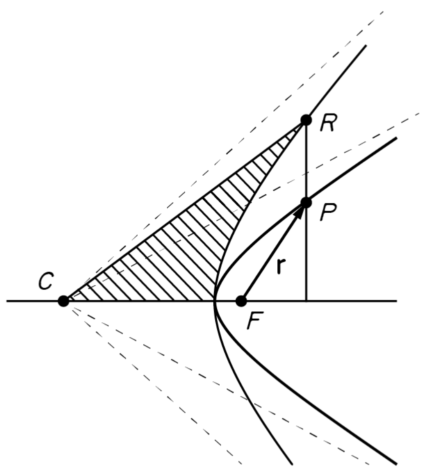


Figure 2.12 Hyperbolic anomaly.

2.3 Orbital position as a function of time

Being able to determine a relation between the position of an orbiting body and time is essential to the resolution of Lambert's problem. However, it's form depends on the type of orbit (circular, elliptic, parabolic or hyperbolic) and in most cases it is not an easily obtained relation.

For circular orbits the solution is yet quite simple. As the distance a from any point of a circle to its center does not vary, from (2.24) we obtain that the velocity of an orbiting body in a circular motion must be constant and equal to

$$v_c = \sqrt{\frac{\mu}{a}}. \tag{2.41}$$

The angular velocity is the linear velocity divided by the radius a,

$$\omega = \frac{v_c}{a} = \sqrt{\frac{\mu}{a^3}}, \tag{2.42}$$

and therefore the argument of latitude u ($u = \omega + \theta$) is proportional to the time t,

$$u = \omega t = \sqrt{\frac{\mu}{a^3}}t. \tag{2.43}$$

For elliptic orbits, it is usual to define the *mean angular motion* or simply *mean motion*, denoted by n. This quantity represents the angular velocity of a body moving along the auxiliary circle, with a radius equal to the semi-major axis of the ellipse. The term used implies an *average* angular velocity, and indeed from (2.17) it follows that

$$n = \sqrt{\frac{\mu}{a^3}} = \frac{2\pi}{T}, \tag{2.44}$$

being T the period of the orbit. Thus, *Kepler's third law of motion* may be stated simply as

$$\mu = n^2 a^3. \tag{2.45}$$

2.3.1 Barker's equation

The relation between the true anomaly and the time for a parabola is called *Barker's equation.* Substituting the orbit equation of a parabola,

$$r = \frac{p}{1+\cos\theta},$$

into Kepler's law of areas,

$$r^2 \frac{d\theta}{dt} = h = \sqrt{p\mu}, \tag{2.46}$$

yields

$$\frac{p^2}{(1+\cos\theta)^2}\frac{d\theta}{dt} = \sqrt{p\mu} \quad \Longrightarrow \quad \frac{1}{(1+\cos\theta)^2}\,d\theta = \sqrt{\frac{h}{p^3}}\,dt.$$

Performing the integration we obtain Barker's equation,

$$\tan^3\frac{\theta}{2}+3\tan\frac{\theta}{2}=2B \qquad \text{where} \qquad B=3\Delta t\sqrt{\frac{h}{p^3}} \tag{2.47}$$

and Δt is the time relative to periapsis passage[1]. The solution for θ when Δt is given requires the root of a cubic equation. To obtain it, we substitute

$$\tan\frac{\theta}{2}=z-\frac{1}{z}$$

and derive

$$z^6-2Bz^3-1=0,$$

for which

$$z=(B\pm\sqrt{B^2+1})^{\frac{1}{3}}.$$

The two solutions verify $z_1z_2=-1$, and therefore

$$\tan\frac{\theta}{2}=z_1-\frac{1}{z_1}=z_2-\frac{1}{z_2}=z_1+z_2=(B+\sqrt{B^2+1})^{\frac{1}{3}}+(B-\sqrt{B^2+1})^{\frac{1}{3}}.$$

2.3.2 Kepler's equation

A direct integration of (2.46) does not result in a useful expression except for a circle or parabola. Kepler's original derivation of the equation which bears his name was geometric, but for this project an analytic derivation is preferred because of its simplicity. This approach involves the eccentric anomaly defined in Sect. 2.2.5. We use the identities (2.36) to obtain

$$d\theta=\frac{\sqrt{1-e^2}}{1-e\cos E}\,dE. \tag{2.48}$$

Hence, from the law of areas and (2.35), we have

$$r^2d\theta=a\sqrt{1-e^2}(1-e\cos E)\ dE=h\ dt.$$

The integration is now trivial and the result is traditionally expressed as

$$M=E-e\sin E \quad \text{where} \quad M=\sqrt{\frac{\mu}{a^3}}\Delta t=n\Delta t, \tag{2.49}$$

n being the mean motion and Δt, once again, the time relative to periapsis passage. M was called the ***mean anomaly*** by Kepler, and one may interpret it as the angular position of a body with constant angular velocity along the auxiliary circle. The relation between the mean anomaly and the eccentric anomaly, as expressed by (2.49), is called ***Kepler's equation.*** The hyperbolic form of Kepler's equation can be obtained following a similar procedure, but using the hyperbolic anomaly instead of E. We then arrive to

$$N=e\sin H-H \quad \text{where} \quad N=\sqrt{\frac{\mu}{-a^3}}\Delta t. \tag{2.50}$$

[1] The time of periapsis passage is the time at which an orbiting body moves through the periapsis of the orbit. Many authors denote this instant by τ, and therefore $\Delta t=t-\tau$.

2.4 Solving Lambert's problem

Lambert's problem, sometimes referred to as the orbital boundary problem, is concerned with the determination of an orbit from two position vectors and the time of flight. Over the years a variety of techniques and procedures have been developed for solving this problem. In this project the approach taken by Lancaster *et al.* [1] will be followed, as it is relatively simple compared to other alternative procedures and algorithms. Furthermore, it provides an universal solution to the problem (that is, valid for elliptic, hyperbolic and parabolic orbits).

Consider an orbiting body located at distances r_1 and r_2 from the center of attraction at times t_1 and t_2 respectively. Let c be the distance and $\Delta\theta$ the transfer angle between the positions of the orbiting body at the two times, where $0 \le \Delta\theta \le 2\pi$.

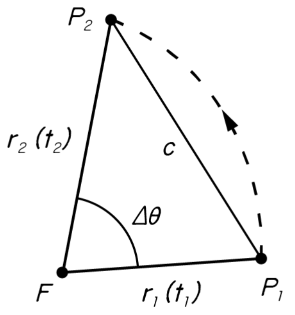


Figure 2.13 A diagram of the two-body orbital boundary problem.

Lambert's problem is that of finding the semi-major axis (or some related quantity) of the orbit, given t_1, r_1, t_2, r_2 and $\Delta\theta$. When Lambert's problem has been solved, other quantities associated with the orbit are easily found, as will be later discussed.

2.4.1 The classical form of Lambert's equations

With origin at the center of attraction, we have, for elliptic motion,

$$r_1 = a(1 - e\cos E_1), \tag{2.51}$$

$$r_2 = a(1 - e\cos E_2), \tag{2.52}$$

$$n(t_1 - \tau) = E_1 - e\sin E_1, \tag{2.53}$$

$$n(t_2 - \tau) = E_2 - e\sin E_2, \tag{2.54}$$

where n is the mean motion and τ the time of periapsis passage, as defined in the previous section. In the perifocal frame of reference, $\mathbf{r}_1$ and $\mathbf{r}_2$ are expressed as

$$\begin{aligned} \mathbf{r}_1 &= a(\cos E_1 - e)\ \mathbf{e}_x + a\sqrt{1-e^2}\sin E_1\ \mathbf{e}_y, \\ \mathbf{r}_2 &= a(\cos E_2 - e)\ \mathbf{e}_x + a\sqrt{1-e^2}\sin E_2\ \mathbf{e}_y. \end{aligned} \tag{2.55}$$

Using the law of cosines, we can express c in terms of r_1, r_2 and $\Delta\theta$:

$$c^2 = r_1^2 + r_2^2 - 2r_1 r_2 \cos\Delta\theta, \tag{2.56}$$

which is to say

$$c^2 = r_1^2 + r_2^2 - 2\mathbf{r}_1 \cdot \mathbf{r}_2. \tag{2.57}$$

Substituting (2.55) in (2.57), we have

$$\begin{aligned} c^2 &= a^2(\cos E_2 - \cos E_1)^2 + a^2(1-e^2)(\sin E_2 - \sin E_1)^2, \\ &= 4a^2\left(1 - e^2\cos^2\frac{E_1+E_2}{2}\right)\sin^2\frac{E_2-E_1}{2}. \end{aligned} \tag{2.58}$$

Adding (2.51) and (2.52),

$$r_1 + r_2 = 2a\left(1 - e\cos\frac{E_1+E_2}{2}\cos\frac{E_2-E_1}{2}\right). \tag{2.59}$$

Subtracting (2.53) from (2.54),

$$n(t_2 - t_1) = (E_2 - E_1) - 2e\cos\frac{E_1+E_2}{2}\sin\frac{E_2-E_1}{2}. \tag{2.60}$$

(2.58), (2.59) and (2.60) determine the three unknowns a, $(E_2 - E_1)$ and $e\cos[(E_1+E_2)/2]$. To simplify these equations it is customary to define two new parameters, α and β. Let

$$\cos\frac{\alpha+\beta}{2} = e\cos\frac{E_1+E_2}{2}, \qquad 0 \le \alpha+\beta < 2\pi, \tag{2.61}$$

and

$$\alpha - \beta = E_2 - E_1 - 2m\pi, \qquad 0 \le \alpha - \beta < 2\pi, \tag{2.62}$$

where m is the number of complete revolutions made by the orbiting body between times t_1 and t_2. (2.58), (2.59) and (2.60) then become

$$\frac{c}{2a} = \sin\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}, \tag{2.63}$$

$$\frac{r_1+r_2}{2a} = 1 - \cos\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2}, \tag{2.64}$$

$$n(t_1 - t_2) = 2m\pi + \alpha - \beta - \cos\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}. \tag{2.65}$$

With appropriate trigonometric identities, (2.63) and (2.64) can be expressed as

$$\cos\beta - \cos\alpha = \frac{c}{a},$$

$$\cos\beta + \cos\alpha = 2 - \frac{r_1+r_2}{a}.$$

Solving these two equations, yields

$$\cos\alpha = 1 - \frac{s}{a} = 1 + 2U, \tag{2.66}$$

$$\cos\beta = 1 + 2KU, \tag{2.67}$$

where we have defined

$$s = \frac{r_1 + r_2 + c}{2}, \qquad U = -\frac{s}{2a}, \qquad \text{and} \quad K = 1 - \frac{c}{s}. \tag{2.68}$$

Since $\cos\alpha = 1 - 2\sin^2(\alpha/2)$, (2.66) can be changed to

$$U = -\sin^2\frac{\alpha}{2}, \qquad 0 \le \alpha < 2\pi, \tag{2.69}$$

and similarly, (2.67) becomes

$$KU = -\sin^2\frac{\beta}{2}, \qquad -\pi \le \beta < \pi. \tag{2.70}$$

Note that the limits for α and β come from the two inequalities for (2.61) and (2.62), as detailed below in Sect. 2.4.2.

The parameter K can be alternatively expressed as

$$K = \frac{s - c}{s} = \frac{r_1 + r_2 - c}{2s} = \frac{(r_1 + r_2)^2 - c^2}{4s^2},$$

and introducing (2.56), we have

$$K = \frac{r_1 r_2 (1 + \cos\Delta\theta)}{2s^2} = \frac{r_1 r_2}{s^2}\cos^2\frac{\Delta\theta}{2}.$$

Now, substituting (2.69) in (2.70),

$$\sin\frac{\beta}{2} = q\sin\frac{\alpha}{2}, \qquad -\pi \le \beta < \pi, \tag{2.71}$$

where

$$q = \pm\sqrt{K} = \frac{\sqrt{r_1 r_2}}{s}\cos\frac{\Delta\theta}{2}. \tag{2.72}$$

Note that the sign of q is taken care of by the angle $\Delta\theta$:

$$\begin{aligned} 1 \ge q \ge 0 \quad &\text{for} \quad \Delta\theta \le \pi, \\ 0 \ge q \ge -1 \quad &\text{for} \quad \Delta\theta \ge \pi. \end{aligned} \tag{2.73}$$

We now have an equation relating α and β as a function of q, which is known. Therefore, we need another equation relating these two parameters to find their values. We can introduce U into (2.65), since

$$n(t_2 - t_1) = \sqrt{\frac{\mu}{a^3}}(t_2 - t_1) = J(-U)^{\frac{3}{2}},$$

where

$$J = \sqrt{\frac{8\mu}{s^3}}(t_2 - t_1). \tag{2.74}$$

Consequently,

$$J = (-U)^{-\frac{3}{2}} \left[2m\pi + \alpha - \beta - \cos\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}\right], \tag{2.75}$$

which can also be written as

$$J = (-U)^{-\frac{3}{2}} [2m\pi + \alpha - \beta - (\sin\alpha - \sin\beta)]. \tag{2.76}$$

Substituting (2.69) into (2.76), we obtain

$$J \sin^3\frac{\alpha}{2} = 2m\pi + \alpha - \beta - \sin\alpha + \sin\beta. \tag{2.77}$$

(2.71) and (2.77) with $0 \le \alpha < 2\pi$ are Lambert's equations for elliptic motion. Given J and q, they are to be solved for α and β. Then, a can be obtained from (2.69) and it is a simple matter to find all other quantities associated with the orbit (more information in Sect. 3.4).

2.4.2 The α and β parameters

As it has been shown, in the classical formulation of Lambert's Problem, two angles α and β appear. Historically, they serve as a means to simplify the equations involved in Lambert's problem. In the following lines, some further insight into these parameters is provided, and a geometric interpretation given by John E. Prussing [8] is briefly summarized.

First, consider their definition,

$$\cos\frac{\alpha+\beta}{2} = e\cos\frac{E_1+E_2}{2}, \qquad 0 \le \alpha+\beta < 2\pi,$$

$$\alpha - \beta = E_2 - E_1 - 2m\pi, \qquad 0 \le \alpha - \beta < 2\pi.$$

These two inequalities are geometrically equivalent to the shaded region showed in Fig. 2.14.

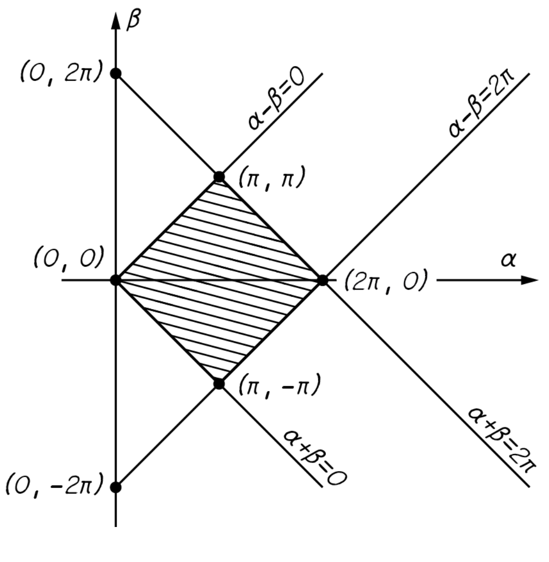


Figure 2.14 The $\alpha\beta$ plane.

From this representation it is evident that $0 \leq \alpha < 2\pi$ and $-\pi \leq \beta < \pi$. We can also obtain $0 \leq \alpha < 2\pi$ by adding both inequalities, and if we add $\beta - \alpha$ to each part of the first inequality and divide the result by two, we obtain $-\pi \leq \beta < \pi$.

These limits are defined due to the nature of trigonometric functions. If α and β were unbounded, an infinite number of solutions would satisfy Lambert's equations; given a pair of values α_0, β_0 that were a solution to the problem, any pair $\alpha_0 + 4k\pi$, $\beta_0 + 4k\pi$ (k being an integer) would also satisfy the equations. Therefore, limits are required to obtain an unique solution.

When the orbiting body makes a number of complete revolutions between times t_1 and t_2, $E_2 - E_1$ will be greater than 2π. The parameter m, which accounts for the number of revolutions, is introduced to maintain α and β within their limits.

To approach the geometric interpretation of the parameters in question it is necessary to introduce perhaps one of the most remarkable theorems in the connection between orbital quantities and the many surprising properties of conics. Discovered by Lambert, ***Lambert's theorem*** has to do with the time to traverse an elliptic arc (although it is true for a general conic). Making reference to the parameters shown in Fig. 2.13, Lambert conjectured that the orbital flight time ***depends only upon the semi-major axis, the sum of*** r_1 ***and*** r_2***, and*** c. If $t_2 - t_1$ is the time to describe the arc from P_1 to P_2, then Lambert's theorem states that

$$\sqrt{\mu}(t_2 - t_1) = F(a,\, r_1 + r_2,\, c), \tag{2.78}$$

which can indeed be proven using the equations obtained in Sect. 2.4.1.

Lambert's theorem permits interesting and important geometric transformations of the boundary value problem while maintaining some key properties. For example, consider an elliptic arc from P_1 to P_2. By moving the focus and vacant focus the ellipse can become very flat. Ultimately, the limiting case is obtained with the foci at F_r and F_r', as illustrated in Fig. 2.15, and the entire curve flattens out to coincide with the major axis. The orbit is then a rectilinear ellipse ($e = 1$, $p = 0$, $a \neq 0$), which has the same values of $r_1 + r_2$ and a and hence the same flight time $t_2 - t_1$, and α and β as the original orbit.

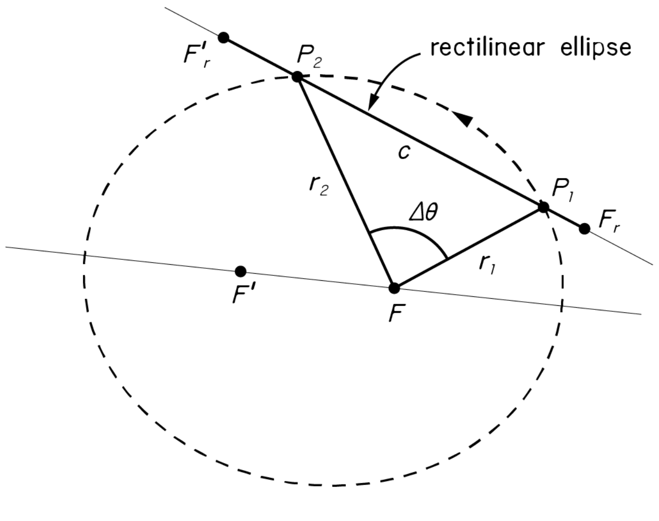


Figure 2.15 Transformation of an ellipse.

For simplicity, consider the case with no complete revolutions between times t_1 and t_2, that is, $m = 0$. Kepler's equation for the flight time between two points in an elliptical orbit, whose locations are specified by the values of eccentric anomaly E_1 and E_2, is

$$\sqrt{\mu}(t_2 - t_1) = a^{\frac{3}{2}} [E_2 - E_1 - e(\sin E_2 - \sin E_1)].$$

By comparing this equation with the one obtained in terms of α and β,

$$\sqrt{\mu}(t_2 - t_1) = a^{\frac{3}{2}} [\alpha - \beta - (\sin\alpha - \sin\beta)],$$

one can interpret α and β as the values of eccentric anomaly on the rectilinear ellipse ($e = 1$) between P_1 and P_2, having the same values of a and $r_1 + r_2$ as the original orbit.

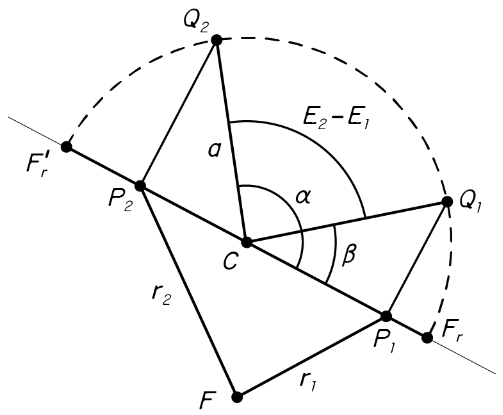


Figure 2.16 Geometric interpretation of the angles α and β for an elliptic orbit.

The geometric interpretation of α and β then follows the usual interpretation of eccentric anomaly, as shown in Fig. 2.16. Of course, this interpretation corresponds to the orbit sketched in Fig. 2.15. When the transfer angle $\Delta\theta$ exceeds 180 degrees, or when F' is contained within the area A_{t_f} swept[1] by the radius vector from t_1 to t_2, α and β may belong in different quadrants. Based on Battin's transformations of the four basic ellipses (Fig. 2.17), a simple rule can be derived:

$$\begin{array}{llllll}
\alpha \le \pi, & \beta \ge 0 & \text{if} & \Delta\theta \le \pi & \text{and} & F' \notin A_{t_f} \\
\alpha \ge \pi, & \beta \ge 0 & \text{if} & \Delta\theta \le \pi & \text{and} & F' \in A_{t_f} \\
\alpha \ge \pi, & \beta \le 0 & \text{if} & \Delta\theta \ge \pi & \text{and} & F' \in A_{t_f} \\
\alpha \le \pi, & \beta \le 0 & \text{if} & \Delta\theta \ge \pi & \text{and} & F' \notin A_{t_f}
\end{array} \tag{2.79}$$

For the particular case sketched in Fig. 2.15, $\Delta\theta \le \pi$ and $F' \notin A_{t_f}$, so the result shown in Fig. 2.16 is obtained. When solving Lambert's problem, this correspondence between the quadrants of α and β and whether F' is contained within A_{t_f}, gives a good intuition on how the complete orbit may look like. Recall from (2.73) that the condition for $\Delta\theta$ can be substituted by one involving the sign of q.

[1] In other words, A_{t_f} is the area enclosed by the straight segments r_1, r_2 and the elliptic arc traveled from P_1 to P_2.

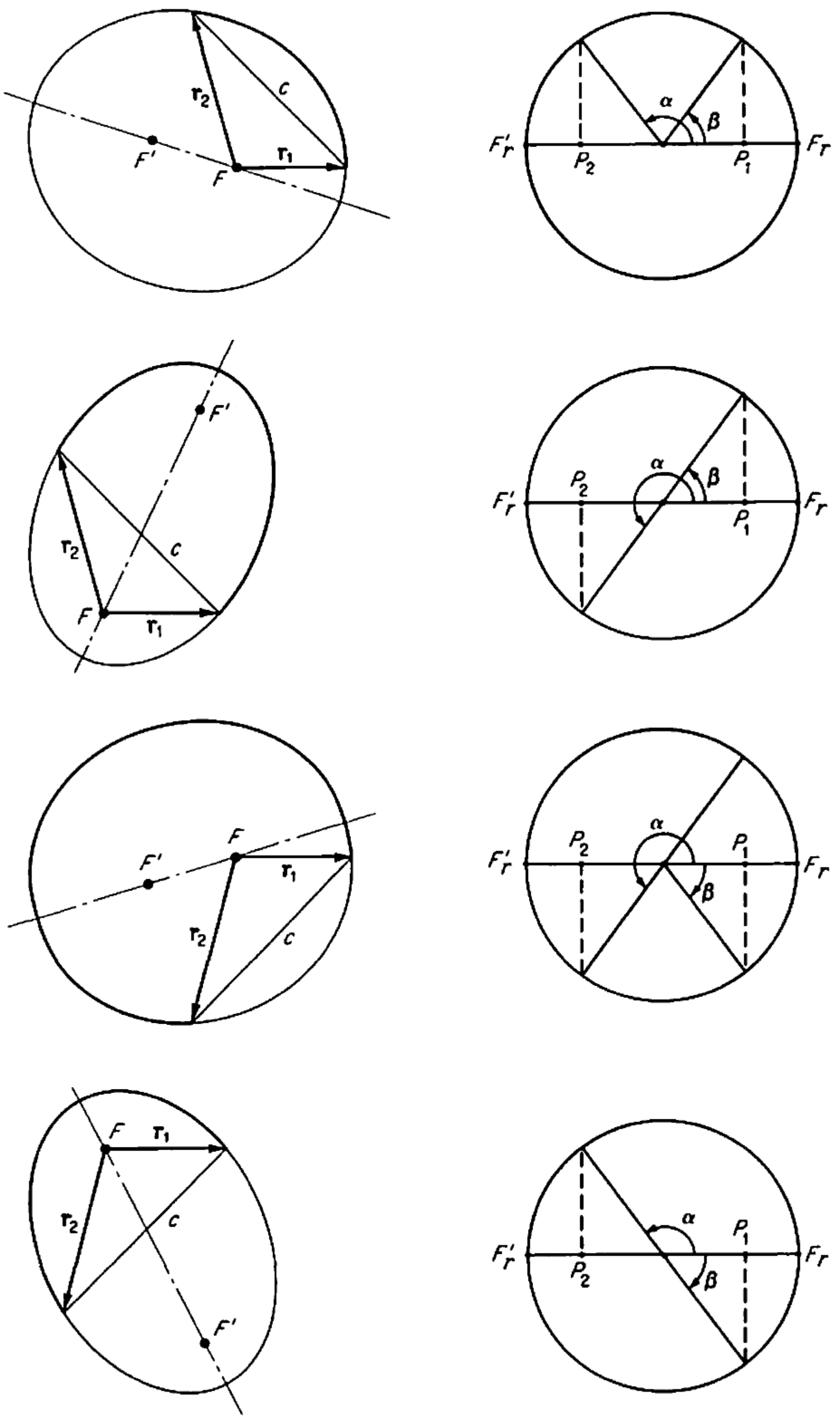


Figure 2.17 Transformations of the four basic ellipses. Adapted from [5].

The cases illustrated in Fig. 2.17 follow the same order (from top to bottom) as those presented in (2.79).

2.4.3 A unified form of Lambert's equations

Up until now the focus has been on elliptic orbits. However, following the approach taken by Lancaster *et al.* [1], in this section we show how to compute an universal solution valid for elliptic, hyperbolic and parabolic transfers.

By a derivation very similar to that for the elliptic case, one finds for the hyperbolic case

$$J = -U^{-\frac{3}{2}}[\gamma - \delta - (\sinh\gamma - \sinh\delta)], \tag{2.80}$$

$$U = \sinh^2\frac{\gamma}{2}, \tag{2.81}$$

$$\sinh\frac{\delta}{2} = q\sinh\frac{\gamma}{2}. \tag{2.82}$$

Unfortunately, just as the hyperbolic anomaly H does not have a geometric interpretation as an angle, neither do γ and δ.

It is customary in the literature to consider J as a function of U. However, when $m = 0$, (2.76) and (2.80) break down for $U = 0$ and suffer from a critical loss of significant digits in the neighborhood of $U = 0$. To remedy this, (2.76) is written in the form

$$J = \sigma(-U) - qK\sigma(-KU), \tag{2.83}$$

where

$$\sigma(u) = 2\,\frac{\arcsin\sqrt{u} - \sqrt{u(1-u)}}{u^{\frac{3}{2}}}.$$

Replacing $\arcsin\sqrt{u}$ and $\sqrt{u(1-u)}$ by a series [9] with $0 \le \alpha < \pi$, we have

$$\sigma(u) = \frac{4}{3} + \sum_{n=1}^{\infty} a_n u^n, \qquad a_n = \frac{2n-1}{n^{n-2}(2n+3)n!} \qquad \text{for} \qquad |u| < 1.$$

A similar procedure produces the same series for the hyperbolic case. For the parabolic case, we have $U = 0$, and the series gives

$$J = \frac{4}{3}(1 - q^3). \tag{2.84}$$

Thus with $m = 0$, we have a series which is valid for elliptic, hyperbolic and parabolic transfers provided $|U| < 1$ (with $0 \le \alpha < \pi$ for the elliptic case).

The choice of U as the independent variable makes J a double-valued function (two values of α or γ can be obtained from (2.69) and (2.81) respectively, which provide two different solutions for J). This problem can be avoided by choosing α or γ as the independent variable. However, an even better choice was proposed by Lancaster *et al.* If we choose

$$\begin{aligned} x &= \cos\frac{\alpha}{2}, \qquad -1 \le x \le 1, \\ &= \cosh\frac{\gamma}{2}, \qquad x > 1, \end{aligned}$$

as the independent variable, a better-behaved function is obtained.

We then have, for elliptic, hyperbolic and parabolic transfers,

$$U = x^2 - 1.$$

For the parabolic case, we let $x = 1$.

For the elliptic case, let

$$y = \sin\frac{\alpha}{2} = \sqrt{-U},$$

$$z = \cos\frac{\beta}{2} = \sqrt{1+KU},$$

$$f = \sin\frac{\alpha-\beta}{2} = y(z-qx),$$

$$g = \cos\frac{\alpha-\beta}{2} = xz - qU,$$

$$h = \frac{1}{2}(\sin\alpha - \sin\beta) = y(x-qz),$$

$$\lambda = \arctan\frac{f}{g}, \qquad 0 \le \lambda \le \pi.$$

It then follows from (2.75) that

$$J = 2\,\frac{m\pi + \lambda - h}{y^3}.$$

For the hyperbolic case, let

$$y = \sinh\frac{\gamma}{2} = \sqrt{U},$$

$$z = \cosh\frac{\delta}{2} = \sqrt{1+KU},$$

$$f = \sinh\frac{\gamma-\delta}{2} = y(z-qx),$$

$$g = \cosh\frac{\gamma-\delta}{2} = xz - qU,$$

$$h = \frac{1}{2}(\sinh\gamma - \sinh\delta) = y(x-qz).$$

Note that $0 \le \gamma - \delta < \infty$, since $0 \le f < \infty$. Additionally,

$$\frac{\gamma-\delta}{2} = \operatorname{arctanh}\frac{f}{g} = \ln(f+g).$$

Thus, for the hyperbolic case,

$$J = 2\,\frac{h - \ln(f+g)}{y^3}.$$

A self-contained universal algorithm

Given two position vectors[1] $\mathbf{r}_1$, $\mathbf{r}_2$ and a time of flight $t_f = t_2 - t_1$, the parameters required to formulate Lambert's equations (q and J) can be obtained.

Usually, an iterative process is employed to solve Lambert's problem. Therefore, an initial estimate for the iteration parameter (in this case x, but alternative variables can be chosen) is made, through which a value of J is obtained. Then, depending on whether this value is greater or less than the actual one, a new estimation of the iteration parameter is proposed. This choice is based on the derivative of J with respect to the iteration parameter. The following formula for the derivative holds for all cases except for $x = 0$ with $q = \pm 1$, and for $x = 1$:

$$\frac{dJ}{dx} = \frac{4 - 4qKx/z - 3xT}{E}.$$

If x is near 1, we differentiate (2.83) to obtain

$$\frac{dJ}{dx} = 2x[qK^2\sigma'(-KE) - \sigma'(-E)], \qquad \sigma'(u) = \frac{d\sigma}{du} = \sum_{n=1}^{\infty} n a_n u^{n-1}.$$

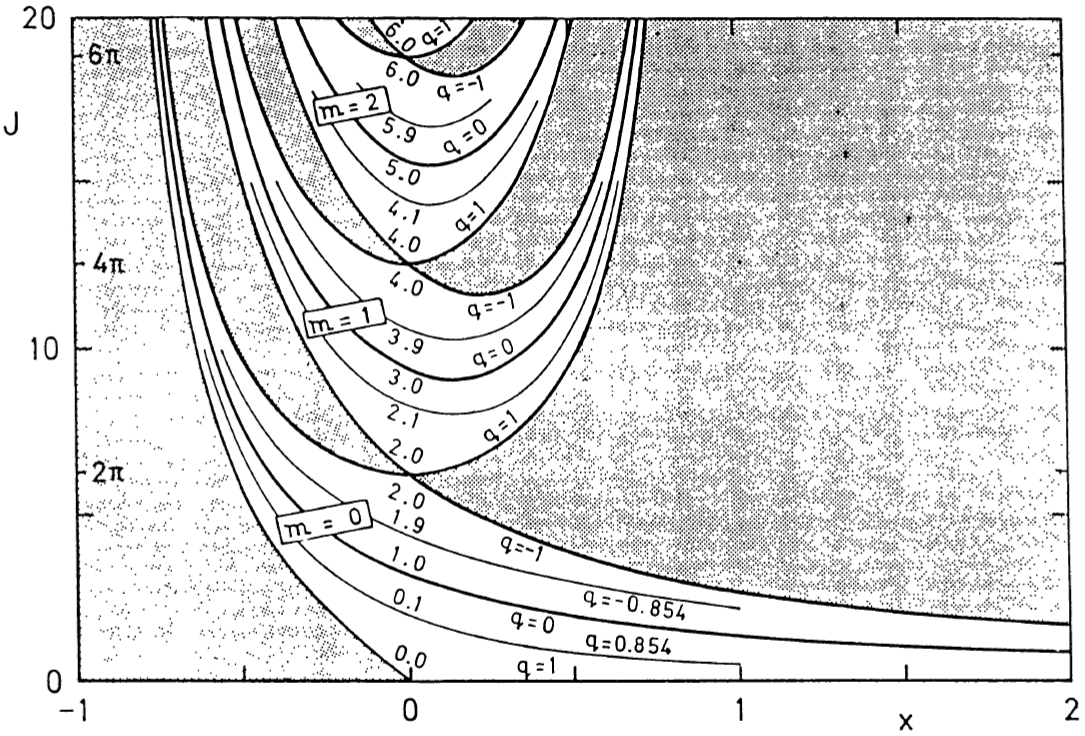


Figure 2.18 Plot of J against x for selected values of q and m. Adapted from [2].

Fig. 2.18 plots J against x for particular values of q and m. Its most striking feature consists in the gaps (unrealizable regions) that occur in the part of the figure associated with elliptic orbits ($x < 1$). Another interesting observation is that in the case $m \neq 0$ for given values of J and q, two different elliptic orbits are possible solutions to the problem. In Chapter 5 an in-depth analysis of this figure is made.

[1] These can be obtained through the *planetary ephemerides*, an important tool which is reviewed in Sect. 3.1.

As for now, note the discontinuity in the slope for $x = 0$ with $q = \pm 1$. Thus we are led to consider four cases: $q = \pm 1$ with $x \geq 0$ and $q = \pm 1$ with $x \leq 0$. Examination of the formulas for dJ/dx in these cases reveals that if $q = 1$, we have a left-hand derivative of -8 and a right-hand derivative of 0 at $x = 0$. If $q = -1$, we have a left-hand derivative of 0 and a right-hand derivative of -8 at $x = 0$.

According to Lancaster *et al.* [1], if the Newton-Raphson method is being used to find x, for $-0.05 < x < 0.05$ a switch should be made to the secant (*regula falsi*) method to avoid computing the derivative.

Below, a simple algorithm which covers all key aspects and procedures addressed so far is presented.

```
Algorithm 1: Universal algorithm for Lambert's Problem
Data: $\mathbf{r}_1$: radius vector of the starting position at $t_1$
      $\mathbf{r}_2$: radius vector of the arrival position at $t_2$
      $t_f$: flight time ($t_f = t_2 - t_1$)
Result: Find $x$
Obtain $r_1 = \|\mathbf{r}_1\|$ and $r_2 = \|\mathbf{r}_2\|$;
Compute $\Delta\theta$, the angle between $\mathbf{r}_1$ and $\mathbf{r}_2$;
Compute $c$ and $s$ from (2.56) and (2.68) respectively;
Compute $q$ from (2.72);
$K = q^2$;
Compute $J_{\text{real}}$ from (2.74), where $\mu$ is that of the central body;
$x = x_0$ (initial estimate);
$J = 0$ to enter the while condition;
while $J$ does not come close enough to $J_{real}$ ($x$ to the correct solution) do
    $U = x^2 - 1$;
    if $x$ is near 1 then
        Compute $J$ from (2.83);
    else
        $y = \sqrt{|U|}$;
        $z = \sqrt{1 + KU}$;
        $f = y(z - qx)$;
        $g = xz - qU$;
        if $U < 0$ then
            $\lambda = \arctan(f/g)$;
            $d = m\pi + \lambda$, where $0 \leq \lambda \leq \pi$;
        else
            $d = \ln(f + g)$;
        end
        $J = 2(x - qz - d/y)/U$;
    end
    if $|x| < 0.05$ then
        Estimate new value of $x$ using the Newton-Raphson method;
    else
        Estimate new value of $x$ using the secant method;
    end
end
```

Multiple improvements have been made to this algorithm, including those by Gooding [2] regarding well-thought heuristics for the initial estimate of x. Moreover, many variants have been developed to optimize speed, precision and robustness (although in this form it is already quite robust).

2.5 Conclusions

Along this first chapter, all the necessary background in orbital mechanics needed to approach Lambert's problem has been addressed. Starting from Newton's basic law of universal gravitation, we have worked up to obtaining some of the most fundamental equations for the analysis of the two-body orbital boundary problem, culminating in Lambert's equations.

When developing a laboratory class for a group of students with a limited duration, an equilibrium must be reached between the quantity and the difficulty of the contents introduced. Therefore, for this project, a decision has been made to consider solely the classical form of Lambert's equations, valid for ellipses.

In this form, (2.61) and (2.62) can be solved using refined root finding algorithms such as `fsolve`, readily accessible in Python. This approach is appropriate for the laboratory class, as the mathematical details of the different methods used to solve this pair of non-linear equations are beyond its scope.

The universal solution presented in Sect. 2.4.3 adds quite some difficulty to the problem, as decision trees appear within the algorithm. These imply a more complex structure of the code, and `fsolve` could no longer be applied to the equations directly. However, its existence is mentioned in the laboratory report and some references are given for the students who may have an interest in extending their knowledge.

The proposed idea is then to introduce the classical form of Lambert's equations,

$$J\sin^3\frac{\alpha}{2} = 2m\pi + \alpha - \beta - \sin\alpha + \sin\beta,$$

$$\sin\frac{\beta}{2} = q\sin\frac{\alpha}{2},$$

and solve them through `fsolve` or a similar function. However, for the cases in which the resulting orbit approaches a parabola (that is, for α near 0), an additional piece of code (based on the series expression for J) may be provided to increase the precision and ensure convergence.

In Chapter 5, a straight-forward algorithm using Python (which is introduced in Chapter 4) is developed, together with a comprehensive analysis of its convergence, robustness and precision. Additionally, some examples and interesting results are reviewed.

3 Basic orbital mechanics' tools used in Lambert's problem

To perform the analysis and optimization of an interplanetary mission (or generally, of any transfer between two generic orbits), a great amount of information related to the different orbits involved is necessary. For instance, the positions of the starting and arrival planets within the set of dates studied are required. Based on planetary ephemerides and using orbit propagators (important tools reviewed in Sect. 3.1) a procedure to obtain these position vectors is described in Sect. 3.2. In Sect. 3.3 we introduce the concept of *sphere of influence* and the method of *patched conic approximation.* The orbital parameters of the trajectory between each pair of points can then be computed through solving Lambert's equations. Given α and β, a procedure to obtain all the other quantities associated with the orbit is addressed in Sect. 3.4. This information, together with the values of the initial and final velocity vectors, provide the characteristic energy C_3 required for each studied trajectory. The results are traditionally expressed in the so-called *pork-chop plots*, described in Sect. 3.5.

3.1 Planetary ephemerides and orbit propagators

In astronomy and celestial navigation, an *ephemeris* (or *ephemerides* in plural) gives the trajectory of a given astronomical object. In most cases, this includes its position and velocity over time. Historically, positions were given as printed tables of values for regular intervals of date and time. Modern ephemerides, however, are often computed electronically, although in some cases printed tables are still produced. Usually, ephemerides are used in conjunction with *orbit propagators.* A propagator is an algorithm whose objective is to obtain future ephemerides from the known orbital parameters at a certain *epoch.* An epoch is an instant in time that serves as a reference point from which time is measured. For astronomical purposes, epoch J2000.0 is commonly used and most data has been updated to this standard. It corresponds to January 1, 2000 at 12:00 TT (Terrestrial Time)[1], and the prefix "J" indicates that it is a Julian epoch (based on Julian years, which we discuss later).

[1] Terrestrial Time (TT) is a modern astronomical time standard defined by the International Astronomical Union, primarily for time measurements of astronomical observations made from the surface of Earth. TT is distinct from the time scale often used as a basis for civil purposes, Universal Time (UT), but the difference is minimal (~ 32 seconds). When using propagators for large time intervals, J2000.0 can be approximated to January 1, 2000 at 12:00 UT.

For this project, a simple linear propagator is chosen, as it provides enough accuracy for our purposes and is easy to implement. For simplicity, we will only consider the major planets of the Solar System. The initial values of each set of orbital elements and their rates, with respect to the plane of the ecliptic and J2000.0, can be obtained from Standish *et al.* [10]. These are valid for any date between 1800 AD and 2050 AD. The orbital elements for one of the planets at a given Julian date *JD* are then obtained from:

$$(a,\, e,\, i,\, \Omega,\, \varpi,\, L) = (a_0,\, e_0,\, i_0,\, \Omega_0,\, \varpi_0,\, L_0) + \frac{d}{dt}(a,\, e,\, i,\, \Omega,\, \varpi,\, L)\cdot T_c, \tag{3.1}$$

where $\varpi = \Omega + \omega$ is the ***longitude of periapsis*** and $L = \varpi + M$ is the ***mean longitude*** (M being the mean anomaly and the rest of variables those defined in Sect. 2.1.3). T_c, the number of centuries past J2000.0, is

$$T_c = \frac{JD - 2451545}{36525}.$$

A Julian date represents the continuous count of days since the beginning of the Julian Period (January 1, 4713 BC at 12:00 UT). For instance, J2000.0 corresponds to *JD* 2452545. A Julian year is defined as exactly 365.25 days, and therefore a Julian century consists of 36525 days. The following formula provides the Julian day corresponding to a given date at 00:00 UT:

$$JD = 367Y - \left\lfloor \frac{7A + 7\lfloor (M+9)/12 \rfloor}{4} \right\rfloor + \left\lfloor \frac{275M}{9} \right\rfloor + D + 1721013.5, \tag{3.2}$$

where Y is the year, M the month and D the day of the month. The symbol $\lfloor x \rfloor$ represents the floor function, which omits the fractional part of x. This formula is valid for all Julian calendar years $\geq$ 4713 BC, that is, $JD \geq 0$.

For the analysis of interplanetary missions, the positions of the starting and arrival planets within the set of dates studied are required. To compute these, a simple function can be created in Python which takes each date expressed in Julian days as an input, and outputs the orbital elements of a selected major planet. In Appendix B such function can be found, along another that computes the Julian day of a date given in DD/MM/YYYY format. From those orbital elements, the radius vector and the velocity can be obtained following the procedure in Sect. 3.2.

3.2 Determining r and v through the orbital elements

The orbital elements of a celestial body fully describe its location and motion through space. However, in many cases it is more convenient to express this information in terms of the body's position vector $\mathbf{r}$ and its velocity $\mathbf{v}$.

For this project, students will be required to plot the orbits using Python. To accomplish this, multiple values of the position vector must be obtained within a fixed interval of time (for example, to plot a complete orbit its period must be considered). Therefore, a new set of orbital elements slightly different to those in (3.1) (which are the ones found in [10]) is proposed:

$$(a,\, e,\, i,\, \Omega,\, \omega,\, \Delta t), \tag{3.3}$$

where $\Delta t = t - \tau$ and τ is the time of periapsis passage. Δt is preferred because then a sweep in days can be made to compute each position vector, instead of having to figure out the range in terms of L, M or any other angular variable. To obtain ω and Δt from ϖ and L, we use the following identities:

$$\omega = \varpi - \Omega,$$

$$M = L - \varpi \quad \Longrightarrow \quad \Delta t = M\sqrt{\frac{a^3}{\mu}}.$$

From (3.3), the position vector and the velocity can be computed in a simple and straightforward manner. First, by using an iterative method (`fsolve` in practice) E is obtained from Kepler's equation,

$$E - e\sin E = \Delta t\sqrt{\frac{\mu}{a^3}}.$$

Then, $\mathbf{r}$ is expressed in perifocal coordinates as

$$\mathbf{r}_{\text{perif}} = a(\cos E - e)\ \mathbf{e}_x + a\sqrt{1-e^2}\sin E\ \mathbf{e}_y,$$

and $\mathbf{v}$ can be derived from the conservation of the eccentricity vector,

$$\mathbf{v}\times\mathbf{h} - \mu\frac{\mathbf{r}}{r} = \mu\mathbf{e} \quad \Longrightarrow \quad \mathbf{v}\times\mathbf{h} = -\frac{\mu}{h^2}\left[\left(\mathbf{e}+\frac{\mathbf{r}}{r}\right)\times\mathbf{h}\right]\times\mathbf{h} \quad \Longrightarrow \quad \mathbf{v} = \frac{\mathbf{h}}{a(1-e^2)}\times\left(\mathbf{e}+\frac{\mathbf{r}}{r}\right),$$

$$\Longrightarrow \quad \mathbf{v}_{\text{perif}} = -\frac{\sqrt{\mu a}}{r}\sin E\ \mathbf{e}_x + \frac{\sqrt{\mu a(1-e^2)}}{r}\cos E\ \mathbf{e}_y. \tag{3.4}$$

To convert $\mathbf{r}$ and $\mathbf{v}$ to heliocentric ecliptic coordinates, we simply multiply them by the following rotation matrix $\mathcal{R}$:

$$\mathcal{R} = \begin{pmatrix} \cos\omega\cos\Omega - \sin\omega\sin\Omega\cos i & -\sin\omega\cos\Omega - \cos\omega\sin\Omega\cos i \\ \cos\omega\sin\Omega + \sin\omega\cos\Omega\cos i & -\sin\omega\sin\Omega + \cos\omega\cos\Omega\cos i \\ \sin\omega\sin i & \cos\omega\sin i \end{pmatrix},$$

which is a 3×2 matrix and therefore transforms two-dimensional vectors ($\mathbf{r}_{\text{perif}}$ and $\mathbf{v}_{\text{perif}}$) into three-dimensional ones ($\mathbf{r}_{\text{ecl}}$ and $\mathbf{v}_{\text{ecl}}$):

$$\mathbf{r}_{\text{ecl}} = \mathcal{R}\mathbf{r}_{\text{perif}}, \qquad \mathbf{v}_{\text{ecl}} = \mathcal{R}\mathbf{v}_{\text{perif}}.$$

In this new coordinate system we use theJ2000.0 ecliptic plane as the reference plane, with the X axis pointing towards the vernal equinox.

3.3 Spheres of influence and patched conic approximation

In this section we briefly introduce the concept of *sphere of influence* and the method of *patched conic approximation*. They both simplify trajectory calculations for spacecraft in a multiple-body environment.

A sphere of influence (SOI) is the approximately spherical region around a celestial body where the primary gravitational influence on an orbiting object is said body. This is typically used to describe the areas in the Solar System where planets dominate the orbits of surrounding objects such as moons, despite the much more massive but distant Sun.

The patched conic approximation divides space into various parts by assigning each of the n bodies its own SOI. Therefore, an orbiting object is considered to be under the gravitational influence of only one massive body at a time. This reduces a complicated n-body problem to multiple two-body problems, for which the solutions are the well-known conic sections. For example, once an orbiting object leaves a planet's SOI, the only gravitational influence considered would be the Sun (until the object enters another body's sphere of influence).

For our purposes, only the transfer orbit within the Sun's SOI is considered. To compute the energetic cost of a mission in terms of C_3, we need to obtain its value for both the launch and arrival maneuvers. By way of illustration, let us consider an interplanetary mission from Earth to Mars. Initially, the spacecraft would be orbiting the Earth in a parking orbit. A parking orbit is a temporary orbit, usually at low altitudes. When a spacecraft embarks on a space mission, it first boosts into the parking orbit, then coasts for a while, then fires again to enter the final desired trajectory.

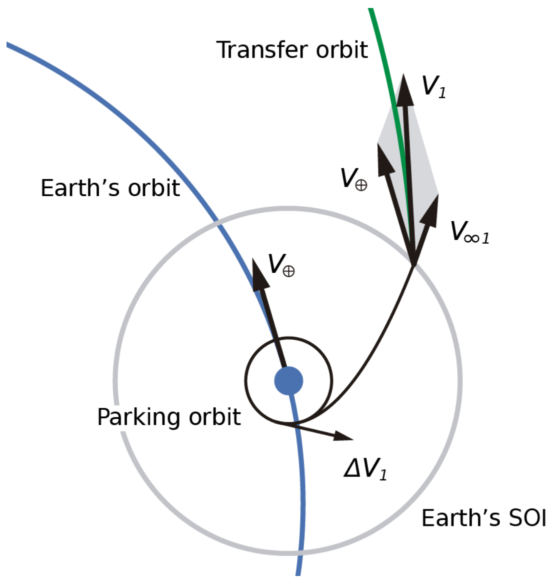


Figure 3.1 Geocentric phase in the patched conic approximation.

In Fig. 3.1, the Earth is represented by a blue dot. The impulse $\Delta\mathbf{V}_1$ is given to the spacecraft at the parking orbit to set it onto an hyperbolic trajectory. This allows it to escape Earth's SOI with a velocity $\mathbf{V}_{\infty 1}$ along one of the hyperbola's asymptotes, which is a good approximation considering that the Earth's SOI is close to 2 million km in diameter (while parking orbits range within $100 \sim 300$ km of altitude).

When the spacecraft reaches the Earth's SOI, we switch from a geocentric frame of reference to a heliocentric one. Considering these as inertial reference frames[1], $\mathbf{V}_{\infty 1}$ and the Earth's velocity $\mathbf{V}_\oplus$ can be added to obtain $\mathbf{V}_1$, the velocity in the heliocentric frame of reference.

[1] Even though this is not true, the error introduced is small and within the order of magnitude of those introduced with the rest of approximations.

Computing C_3 in this scenario is simple. On one hand, $\mathbf{V}_\oplus$ can be obtained from the Earth's ephemerides as described in Sect. 3.2. Furthermore, $\mathbf{V}_1$ is the velocity at the starting point of the transfer orbit and therefore it can be obtained through Lambert's problem. Then,

$$C_3|_{\text{launch}} = V_{\infty 1}^2 = \left\|\mathbf{V}_1 - \mathbf{V}_\oplus\right\|^2$$

After reaching the Earth's SOI, the spacecraft begins its journey through the Solar System under the Sun's gravitational influence, as illustrated in Fig. 3.2. In this figure, the planets' spheres of influence have been exaggerated to distinguish the different phases (on real scale, these would be barely noticeable). It is common practice to simply consider the starting and arrival points of the transference orbit as the planets' positions, which greatly reduces the complexity of the analysis and is a good approximation. Additionally, the time spent cruising through outer space (on a transfer orbit) greatly exceeds the time spent escaping and entering the planets' spheres of influence. Therefore, it is usual to neglect the latter in order to further simplify the analysis.

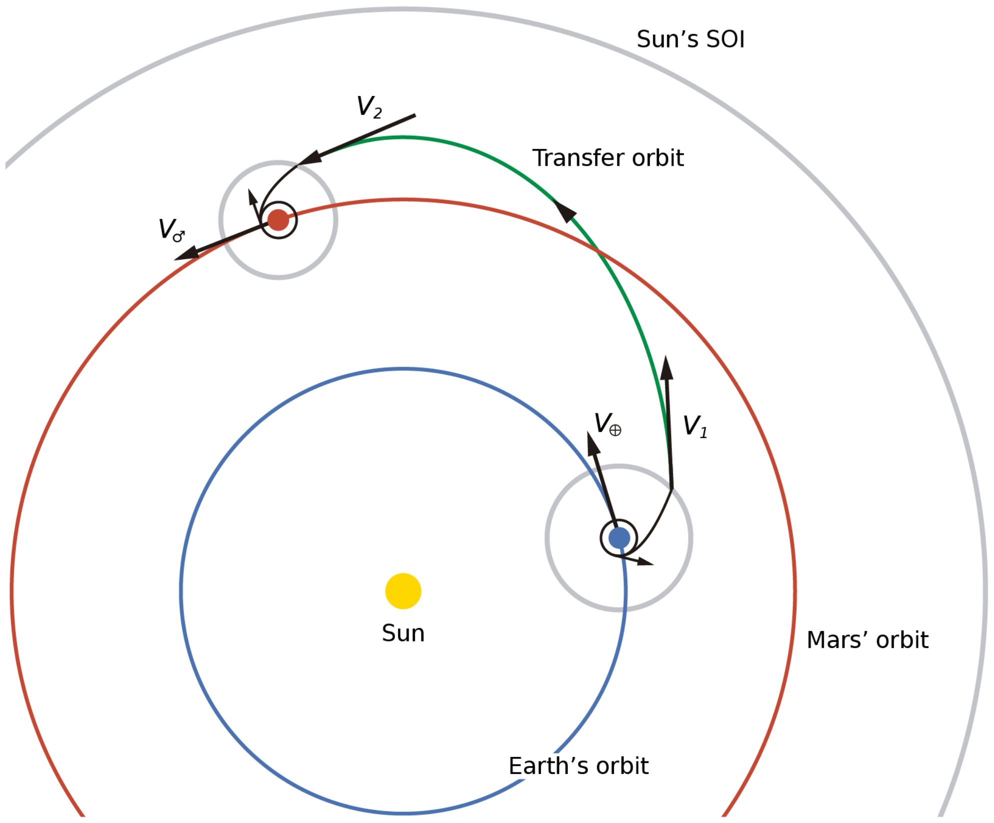


Figure 3.2 Heliocentric phase in the patched conic approximation. Note that the Earth and Mars' spheres of influence are exaggerated, and the drawing is not to scale.

Lastly, the spacecraft enters a planetocentric phase (in this example, based on Mars) when it approaches the arrival planet. Once inside Mars' SOI, it describes a hyperbolic trajectory (arriving once again along an asymptote), at whose periapsis an impulse $\Delta\mathbf{V}_1$ is applied to the spacecraft to return it into a parking orbit.

Note that the radius of periapsis of the arrival hyperbola cannot be deduced under the considered approximations, and has to be arbitrarily imposed. However, this is not unrealistic since small impulse corrections of the orbit during the heliocentric phase would easily allow to fix this value in a real flight. In Fig. 3.3 a diagram detailing this final phase is shown.

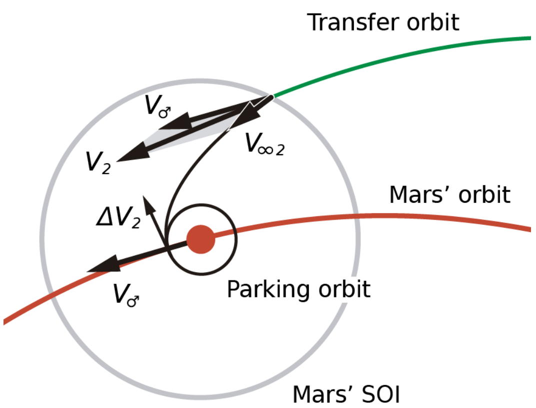


Figure 3.3 Planetocentric phase in the patched conic approximation.

To compute $C_3|_{\text{arrival}}$, a procedure analogous to the one described for the geocentric phase is followed. In this case,

$$C_3|_{\text{arrival}} = V_{\infty 2}^2 = \left\|\mathbf{V}_2 - \mathbf{V}_♂\right\|^2,$$

and the total energetic cost of the mission is then

$$C_3|_{\text{total}} = C_3|_{\text{launch}} + C_3|_{\text{arrival}}.$$

3.4 Determining an orbit through α, β and the initial data of Lambert's problem

Once Lambert's equations have been solved for α and β, all other quantities associated with the transfer orbit that is solution to Lambert's problem can be obtained. In this section a procedure is described which allows to fully determine its orbital elements. Our aim is to compute the same set of orbital elements that were used previously to obtain $\mathbf{r}$ and $\mathbf{v}$, as this reduces the number of functions that will need to be introduced in the laboratory class.

Furthermore, as discussed in Sect. 2.5, only the elliptic form of Lambert's equations will be considered. Therefore, our focus is on ellipses, although a similar procedure can be applied for the hyperbolic and parabolic cases.

In addition to α and β, we require the initial problem data (t_1, r_1, t_2, r_2 and $\Delta\theta$, as stated in Sect. 2.4). Usually, the starting and arrival position vectors ($\mathbf{r}_1$ and $\mathbf{r}_2$) are obtained from the planetary ephemerides at a pair of given dates (t_1 and t_2). However, $\Delta\theta$ cannot be computed unambiguously from these position vectors. Two possible angles can be measured between two vectors; if we denote one of them as δ, the other will be $2\pi - \delta$. This translates into two possible transfer orbits between $\mathbf{r}_1$ and $\mathbf{r}_2$. One of them will follow a clockwise motion, while the other an anticlockwise motion.

In our Solar System, the orbits about the Sun of all planets and most other objects are *direct*, that is, in the same direction as the Sun rotates. An orbit in the direction opposite to the rotation of its central object receives the name of *retrograde* orbit. To avoid possible confusion, we will refer to direct and retrograde motion instead of clockwise or anticlockwise motion (always with respect to the Sun's rotation, in the context of an interplanetary mission).

Therefore, to correctly choose $\Delta\theta$, we need to specify whether the considered transfer orbit is direct or retrograde. In the heliocentric ecliptic coordinate system, a direct orbit has its specific angular momentum $\mathbf{h}$ pointing towards the positive Z axis (in other words, its Z component $\mathbf{h}_z$ is positive). If the vector resulting from the operation $\mathbf{r}_1 \times \mathbf{r}_2$ points towards the positive Z axis, the smallest angle between $\mathbf{r}_1$ and $\mathbf{r}_2$ would provide a direct transfer orbit:

$$\Delta\theta = \Delta\theta_{\text{small}} = \arccos\frac{\mathbf{r}_1 \cdot \mathbf{r}_2}{r_1 r_2}$$

However, if $\mathbf{r}_1 \times \mathbf{r}_2$ points towards the negative Z axis and a direct transfer orbit is still desired, $\Delta\theta = 2\pi - \Delta\theta_{\text{small}}$. On the other hand, when a retrograde orbit is desired the opposite is true.

Usually, direct orbits are preferred as they imply much lower energetic costs since $\mathbf{V}_\oplus$ is added to the escape velocity (whereas to achieve a retrograde orbit it should be cancelled). Therefore, in this project we will ignore retrograde orbits and only consider direct motion. Once $\Delta\theta$ has been determined, we proceed to the calculation of the orbital elements. First, let us compute c and s from $\mathbf{r}_1$ and $\mathbf{r}_2$,

$$c^2 = r_1^2 + r_2^2 - 2\mathbf{r}_1 \cdot \mathbf{r}_2, \qquad s = \frac{r_1 + r_2 + c}{2}.$$

Then, the semi-major axis a can be obtained from

$$\sin^2\frac{\alpha}{2} = \frac{s}{2a} \quad \Longrightarrow \quad a = \frac{s}{2\sin^2(\alpha/2)},$$

and additionally,

$$E_2 - E_1 = 2\pi m + \alpha - \beta.$$

Carrying out the dot product of (3.2) with (3.4) yields

$$\mathbf{r} \cdot \mathbf{v} = \sqrt{\frac{\mu a}{r^2}\sin^2 E + \frac{\mu a(1-e^2)}{r^2}\cos^2 E} = \frac{\sqrt{\mu a}}{r} e \sin E,$$

which together with (2.5) and (2.35) provide two important identities,

$$e\cos E = 1 - \frac{r}{a}, \qquad e\sin E = \frac{r\dot{r}}{\sqrt{\mu a}}.$$

Substituting these into

$$n(t_2 - t_1) = E_1 - e\sin E_1 - E_2 + e\sin E_2,$$

leads to

$$\dot{r}_1 = v_{r1} = \frac{n(t_2 - t_1) - (E_2 - E_1) + (1 - r_1/a)\sin(E_2 - E_1)}{r_1[1 - \cos(E_2 - E_1)]/\sqrt{\mu a}},$$

and a similar equation for $\dot{r}_2$. Furthermore,

$$v = \sqrt{\frac{2\mu}{r} - \frac{\mu}{a}},$$

and

$$v_\theta = \sqrt{v^2 - v_r^2}.$$

For the velocity $\mathbf{v}_1$ we have

$$\mathbf{v}_1 = \mathbf{v}_{r1} + \mathbf{v}_{\theta 1}$$

where $\mathbf{v}_{r1}$ is along $\mathbf{r}_1$ and $\mathbf{v}_{\theta 1}$ is in the plane of motion perpendicular to $\mathbf{r}_1$ and in the direction of motion (in the direction of increasing true anomaly). We have

$$\mathbf{v}_{r1} = v_{r1}\,\frac{\mathbf{r}_1}{r_1}, \qquad \mathbf{v}_{\theta 1} = c_1\mathbf{r}_1 + c_2\mathbf{r}_2,$$

where c_1 and c_2 are to be determined. Since

$$\mathbf{r}_1 \cdot \mathbf{v}_{\theta 1} = \mathbf{0} = c_1 r_1^2 + c_2 \mathbf{r}_1 \cdot \mathbf{r}_2,$$

$$\mathbf{r}_2 \cdot \mathbf{v}_{\theta 1} = r_2 v_{\theta 1} \sin\Delta\theta = c_1 \mathbf{r}_1 \cdot \mathbf{r}_2 + c_2 r_2^2,$$

and $\mathbf{r}_1 \cdot \mathbf{r}_2 = r_1 r_2 \cos\Delta\theta$, we have

$$c_1 r_1 + c_2(r_2 \cos\Delta\theta) = 0,$$

$$c_1(r_1 \cos\Delta\theta) + c_2 r_2 = v_{\theta 1} \sin\Delta\theta.$$

Solving for c_1 and c_2, we obtain

$$\mathbf{v}_1 = \left(v_{r1} - \frac{v_{\theta 1}}{\tan\Delta\theta}\right)\frac{\mathbf{r}_1}{r_1} + \frac{v_{\theta 1}}{\sin\Delta\theta}\frac{\mathbf{r}_2}{r_2}.$$

In a similar way, we find

$$\mathbf{v}_2 = -\frac{v_{\theta 2}}{\sin\Delta\theta}\frac{\mathbf{r}_1}{r_1} + \left(v_{r2} - \frac{v_{\theta 2}}{\tan\Delta\theta}\right)\frac{\mathbf{r}_2}{r_2}.$$

We must keep in mind that these procedures later become functions in Python. Even though $\mathbf{r}_1$, $\mathbf{r}_2$, $\mathbf{v}_1$ and $\mathbf{v}_2$ can all be computed directly, without determining the complete set of orbital parameters, it is good practice to keep the functions simple and modular. Therefore, we obtain the remaining parameters to later make use of the previous function which outputs $\mathbf{r}$ and $\mathbf{v}$. Moreover, this allows us to easily compute any pair $\mathbf{r}$, $\mathbf{v}$ from the transfer orbit, and not just those at times t_1 and t_2.

Through $\mathbf{r}_1$ and $\mathbf{v}_1$, the specific angular momentum $\mathbf{h}$ and the eccentricity vector $\mathbf{e}$ can be computed. From their definitions in Sect. 2.1, we have

$$\mathbf{h} = \mathbf{r}_1 \times \mathbf{v}_1, \qquad h = \|\mathbf{h}\|,$$

$$\mathbf{e} = \frac{\mathbf{v}_1 \times \mathbf{h}}{\mu} - \frac{\mathbf{r}_1}{r_1}, \qquad e = \|\mathbf{e}\|.$$

Additionally, we obtain the nodal vector **n**,

$$\mathbf{n} = \frac{\mathbf{e}_z \times \mathbf{h}}{\|\mathbf{e}_z \times \mathbf{h}\|},$$

where $\mathbf{e}_z$ represents the unit vector in the Z direction, and not the Z axis component of the eccentricity vector.

The angle of inclination i can be easily computed from **h**,

$$i = \arccos \frac{\mathbf{h}_z}{h},$$

where $0 \le i \le \pi$.

Recall from Fig. 2.2 that the longitude of the ascending node Ω becomes the angle between **n** and the First Point of Aries (the x direction in our coordinate system). Therefore,

$$\Omega = \arctan \frac{n_y}{n_x},$$

and to choose the quadrant correctly, the Python function `arctan2()` can be used. This function returns an angle between $-\pi$ and π.

The argument of periapsis ω can be interpreted as the angle between **n** and **e**,

$$\omega = \arccos \frac{\mathbf{n} \cdot \mathbf{e}}{e}.$$

If the Z component of **e** is positive, **e** will lay above the reference plane and $0 \le \omega \le \pi$. On the other hand, if the Z component of **e** is negative, $-\pi \le \omega \le 0$.

Finally, to obtain Δt we must first compute the true anomaly θ. From the orbit equation in polar coordinates,

$$\theta_1 = \arccos \frac{a(1-e^2) - r_1}{e r_1},$$

where $0 \le \theta \le \pi$ if the orbit is being traveled from periapsis to apoapsis (if $\dot{r}_1 > 0$), and $-\pi \le \theta \le 0$ in the opposite scenario ($\dot{r}_1 < 0$). Then,

$$E_1 = 2\arctan\left(\sqrt{\frac{1-e}{1+e}}\, \tan \frac{\theta_1}{2}\right) \quad \Longrightarrow \quad M_1 = E_1 - e \sin E_1,$$

$$\Delta t_1 = M_1 \sqrt{\frac{a^3}{\mu}}.$$

Note that $\Delta t_1 = t_1 - \tau$ has been arbitrarily chosen, instead of Δt_2. In the practical class, students will be asked to plot the transfer orbit. This is achieved through a simple for-loop, indicating a selected number of dates (ranging between t_1 and t_2). By choosing Δt_1 it becomes simpler to perform a sweep in the usual direction of time, which is perhaps more intuitive.

A Python function which implements the contents presented in this section can be found in Appendix B.

3.5 Pork-chop plots

The tool traditionally used to choose the starting and arrival dates for a one-way impulsive mission is the pork-chop[1] plot. The classical pork-chop plot uses as a cost function the characteristic energy $C_3 = v_\infty^2$, and expresses it as a function of possible starting and arrival dates. A given contour, called a *pork-chop curve*, represents constant C_3, and at the center of these pork-chop curves we find the optimal transfer orbit (the one with minimum C_3).

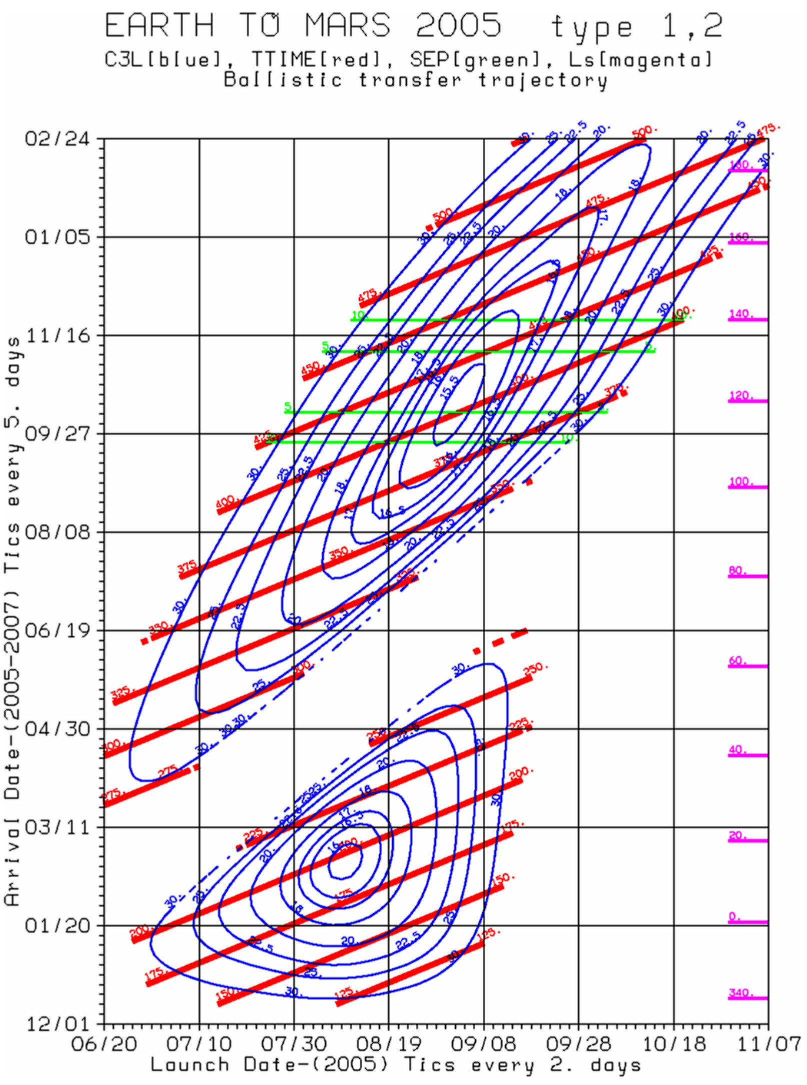


Figure 3.4 Representative pork-chop plot for the 2005 Mars launch opportunity. A given blue contour represents a solution with a constant C_3, and the red lines represent trips with the same time of flight. [11]

More than one optimal solution may exist for a given range of dates. In Fig. 3.4 a pork-chop plot for the 2005 Mars launch opportunity is represented ($C_3|_{\text{launch}}$). On the horizontal axis, different starting dates are considered. On the vertical axis, we find possible arrival dates.

[1] This name comes from the distinctive shape of the plot, resembling grilled pork meat.

Here, the pork-chop curves are represented up to $C_3 = 30\ \mathrm{km}^2/\mathrm{s}^2$, which is an upper limit given by the maximum C_3 that can be provided by the launcher. We can clearly differentiate two separate regions on this pork-chop plot. In the lower part of the graph we find transfer orbits of *Type 1*, which imply short times of flight of about seven months. In the upper part, we find transfer orbits of *Type 2*, which can take up to two years. The decision whether to choose Type 1 or Type 2 transfer orbits depends on multiple factors; the C_3 required, the suitability of both the starting and arrival dates, the time of flight, etc.

That a transfer orbit is Type 1 or Type 2 has further geometrical implications. A transfer orbit of Type 1 implies $\Delta\theta < 180°$, while one of Type 2 $\Delta\theta > 180°$. To appreciate this more clearly, let us plot the transfer orbits corresponding to both optimal points present in Fig. 3.4. To do this, the Python functions developed through Sects. 3.1-3.4, and an additional piece of code to produce the plots (addressed in Chapter 5) are to be used.

In Figs. 3.5 and 3.6 a three dimensional plot of each transfer orbit is shown, together with its two-dimensional projection in the XY plane. Clearly, the rule indicated above is met. It can also be appreciated that both solutions appear to be very similar to Hohmann transfer orbits. This is no coincidence; when considering the two-dimensional problem, the Hohmann transfer orbit can be proven to be the optimal solution. In three dimensions, the starting and arrival orbits are no longer co-planar and therefore slight variations occur. Furthermore, neither the Earth or Mars' orbits are circular, nor their lines of apsides are aligned. Even if this was the case, $\Delta\theta = 180°$ implies, in general, a polar heliocentric orbit, which requires a rather large C_3. This is due to $\mathbf{r}_1$ and $\mathbf{r}_2$ lying in different planes.

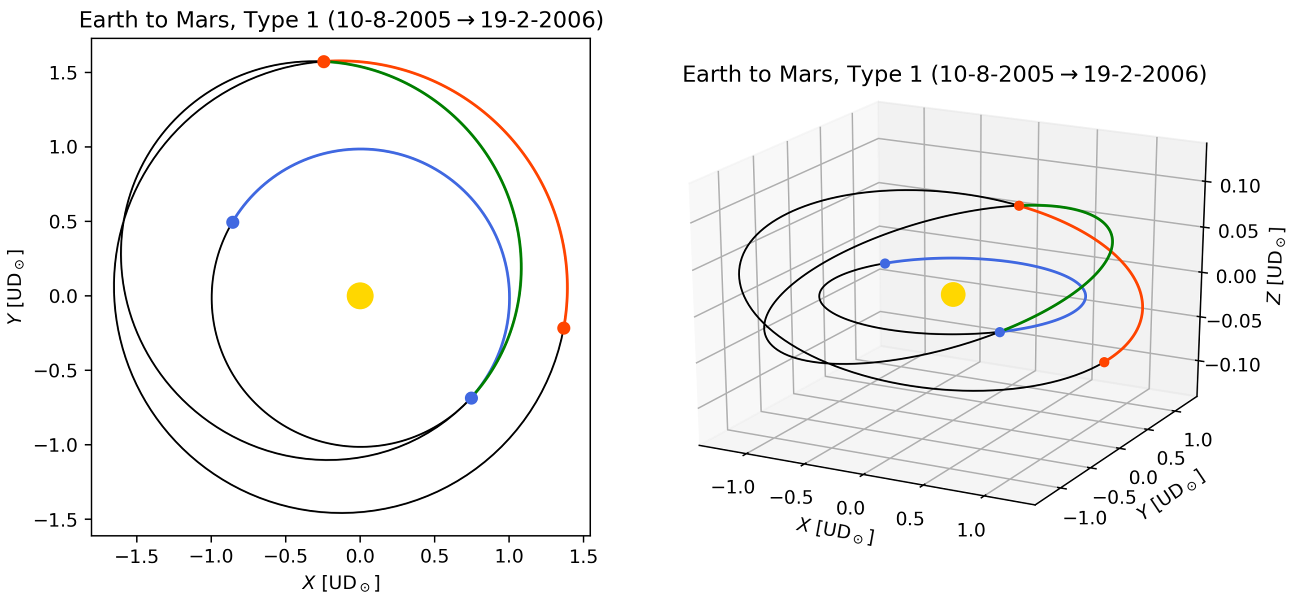


Figure 3.5 Transfer orbit of Type 1 from Earth to Mars, and its two-dimensional projection.

Note that on these figures, the axes are expressed in *canonical units* with respect to the Sun. In orbital mechanics, canonical units are defined in terms of an object's reference orbit (in our case, the Earth's orbit around the Sun). For instance, the unit of distance ($\mathrm{UD}_\odot$) corresponds to the Earth's mean distance to the Sun (1 astronomical unit). By using these, many calculations can be simplified (as it can be appreciated in Appendix B).

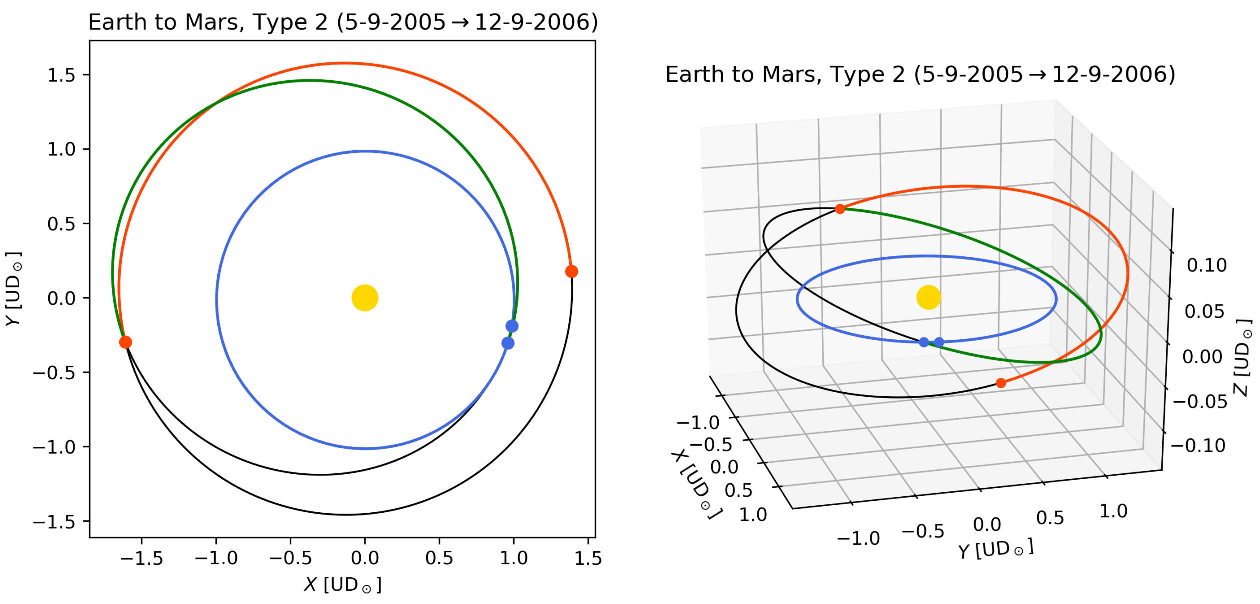


Figure 3.6 Transfer orbit of Type 2 from Earth to Mars, and its two-dimensional projection.

4 Programming with Python

By no means is the purpose of this chapter to be an exhaustive beginners guide to Python programming. Only the basics and the strictly necessary knowledge to code the algorithm for the solution of Lambert's problem are reviewed.

As all the students enrolled in the course *Orbital Mechanics and Aerospace Vehicles* are familiar with MATLAB syntax, the approach followed will focus on comparing both languages, to take advantage of the students' existing knowledge.

4.1 Introduction

Python is a powerful programming language which has many different applications. Ranging from the creation of web applications to handling big data or performing complex mathematics, the options are almost limitless. In this project, we focus on computational science and engineering. Perhaps one of the most important advantages of programming with Python is the great number of existing libraries available —for scientific computation, it is crucial to make use of numerical libraries such as NumPy, SciPy and the plotting package Matplotlib.

All major editors that are used for programming (such as Atom, Vim, Sublime Text, etc.) provide Python modes. For beginners, however, working within an Integrated Development Environment (IDE) may result more intuitive. IDEs provide an useful range of tools, such as a display showing the variables created by the user or allowing to execute the code in debug mode (step by step). Furthermore, the students are already familiarized with MATLAB's IDE. For this project, Spyder seems a sensible choice as it provides a similar graphic user interface and is easy to pick up. Additionally, Spyder is included by default in the Anaconda distribution. Anaconda is a free and open-source Python distribution for scientific computing, that aims to simplify package management and deployment. It also incorporates some of the most popular Python libraries (or packages), including those which will be used throughout the laboratory class.

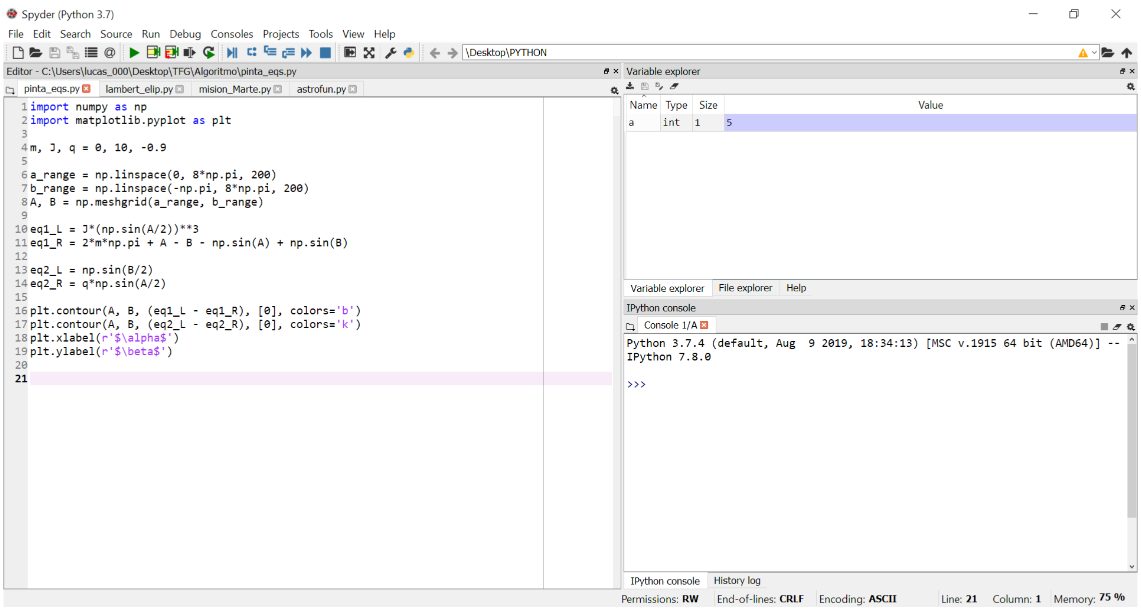
Our aim is to keep the process of learning Python simple. Therefore, during the laboratory class our focus will be on the actual programming part, and we will simplify the tedious process of installing a Python distribution, an IDE and all the different packages by using Anaconda, which to this day is installed in most computers of the University of Seville's ETSI (Superior Technical School of Engineering).

4.2 Basics

Python was designed for readability, and has some similarities to the English language with influence from mathematics. It uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses. Additionally, Python relies on indentation, using whitespace, to define the scope of loops or functions. Other programming languages often use curly-brackets for this purpose.

4.2.1 Working with Spyder

In Fig. 4.1, a typical configuration of Spyder's main window is shown. It can be split into three main modules: editor, variable explorer and console. In the editor, selected parts of the code can be executed and the debug mode is available. The variable explorer, as its name implies, shows the variables created by the user, indicating their type, size and value. On the other hand, the interactive console provides programmers with a quick way to execute commands and try out or test code without creating a file.



```python
import numpy as np
import matplotlib.pyplot as plt

m, J, q = 0, 10, -0.9

a_range = np.linspace(0, 8*np.pi, 200)
b_range = np.linspace(-np.pi, 8*np.pi, 200)
A, B = np.meshgrid(a_range, b_range)

eq1_L = J*(np.sin(A/2))**3
eq1_R = 2*m*np.pi + A - B - np.sin(A) + np.sin(B)

eq2_L = np.sin(B/2)
eq2_R = q*np.sin(A/2)

plt.contour(A, B, (eq1_L - eq1_R), [0], colors='b')
plt.contour(A, B, (eq2_L - eq2_R), [0], colors='k')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\beta$')

```

Name	Type	Size	Value
a	int	1	5

```
Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)] --
IPython 7.8.0

>>>
```

Figure 4.1 Typical configuration of the main window in Spyder 3.3.6.

Additionally, it implements a file explorer and a "help" functionality. Overall, Spyder is very similar in many aspects to MATLAB's IDE, and will flatten Python's learning curve for the students by providing a familiar working environment.

4.2.2 Importing modules and/or packages

Python is an interpreted language —it executes instructions directly, without previously compiling the code into machine-language instructions. The interpreter executes every line in sequential order, from top to bottom. If several commands are given in the same line (separated by a semicolon), then these are processed from left to right. Therefore, the first lines of a Python script (a file with the extension “.py” containing code written in Python) must contain the modules and/or packages (a collection of modules) that are intended to be used throughout the file. Importing a module or package is straightforward:

```
import numpy
```

The names within the numpy package must be accessed through the name numpy. For example: numpy.sin. However, the name by which the module or package is known locally can be different from its “official” name. For instance, writing

```
import numpy as np
```

changes the package name to something more manageable. Furthermore, writing

```
from numpy import sin
```

will only import the sin function from the numpy package. It is possible to import more than one function from a given module or package in one go:

```
from numpy import sin, cos
```

4.2.3 Basic operations

Basic operations such as addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (** and not ^ as in MATLAB) work as expected. Parentheses can be used for grouping.

We can enter individual commands at the interactive console which are immediately evaluated and carried out by the Python interpreter. The *primary prompt* (usually three greater-than signs >>>) signals that Python is waiting for input from us:

```
>>>
```

We can now enter commands, for example 2**3, followed by the Enter key ⏎ :

```
>>> 2**3
ans = 8
```

Once we press the Enter key, Python will evaluate the given command and display the result in the next line.

4.2.4 Data types and indexing

In python, data can be expressed in different ways. Numbers can be defined as integers, floating point numbers, etc. and sequences can be expressed in different forms. Strings, lists and *tuples* are the three basic types of sequences.

A string can declared using simple quotes:

```
>>> a = 'Hello world'
```

A list is a sequence of objects of any type, for example integers, declared using square brackets:

```
>>> a = [1, 2, 3]
```

Different types of objects can be combined within a single list:

```
>>> a = [1, 2.2, 'three']
```

Tuples are very similar in behaviour to lists, and are declared using simple parentheses:

```
>>> a = (1, 2, 'three')
```

Tuples and strings are "immutable" (which means we cannot change individual elements within the tuple or individual character within a string once created) whereas lists are "mutable".

Table 4.1 Some useful sequence operations.

`a[i]`	returns *i*-th element of a
`a[i:j]`	return elements *i* up to $j-1$
`len(a)`	returns number of elements in sequence
`min(a)`	returns smallest value in sequence
`max(a)`	returns largest value in sequence
`a + b`	concatenates a and b

Sequences share the operations indicated in Table 4.1, among some others which are not reviewed here. It is important to note that Python uses zero-based indexing, that is, the first element has an index 0, the second has index 1, and so on. This is in contrast to MATLAB's one-based indexing, where the first element has index 1.

The use of indices in slicing also differs from MATLAB's approach. In MATLAB, stating `a(1:2)` returns the first and second elements from the vector `a`. However, stating `a[1:2]` in Python only returns the second element. The best way to remember how slices work in Python is to think of the indices as pointing between elements:

```
+-----+-----+-----+-----+-----+
| 'H' | 'e' | 'l' | 'l' | 'o' |
+-----+-----+-----+-----+-----+
0     1     2     3     4     5
```

Additionally, `a[:i]` would return all the elements up to the *i*-th one, while `a[i:]` would return all the elements between the *i*-th and the last one. In other words, if we declare `a` as

```
>>> a = [1, 2, 3, 4, 5]
```

writing

```
>>> b = a[:3] + a[3:]
>>> print(b)
b = [1, 2, 3, 4, 5]
```

returns a list b identical to a (while in MATLAB this wouldn't be the case). The `print()` function prints the given object to the console.

Keep in mind that only the essentials are being reviewed here. The underlying details of each data type could fill an entire volume. For more information, refer to the approachable beginner's guide to Python by Hans Fangohr [12] or to the Python documentation [13].

4.2.5 Defining and using functions

Functions allow us to group a number of statements into a logical block. We communicate with a function through a clearly defined interface, providing certain parameters to the function, and receiving some information back.

We can group functions together into a Python module (a module is simply a file containing Python functions and statements), and in this way create our own libraries of functionality.

To introduce the generic format of functions lets address a simple example. Suppose we need a function that computes the multiplication of two variables. The way of defining this function in Python would be:

```
def multiply(a,b):
    '''This function multiplies a times b'''
    result = a*b   # this is a comment
    return result

# this is not part of the function
```

and we would call it by simply writing

```
>>> multiply(2,3)
ans = 6
```

which returns the expected value a=6.

A function cannot be called before its declaration (the Python interpreter has to first come across the `def` line in order to recognize the function and allow it to be called). Functions can also take and/or return an arbitrary number of arguments. Moreover, it is good practice to provide a brief description of the function just below its declaration. This receives the name of *docstring* (documentation string), and can be declared using triple quotes.

The indentation after the declaration of the function is required and defines its scope. The same applies to for-loops and if-else statements. Additionally, there is no need to specify an end command to these blocks (while it is necessary in MATLAB); returning to the previous indentation level is enough.

Anonymous functions

Anonymous functions, also known as lambda functions, are functions that are defined without a name. Consider a lambda function which computes the square of a given variable. In Python it has the following syntax:

```
>>> (lambda x: x*x)(10)
ans = 100
```

Although lambda functions are intended to be defined without a name, they provide a quick way to declare a function without using the `def` keyword:

```
>>> square = lambda x: x*x
>>> square(10)
ans = 100
```

Lambda functions can also take multiple arguments (although they can only return one). For example, the function `multiply` defined previously using `def` could be alternatively declared as:

```
>>> multiply = lambda a,b: a*b
>>> multiply(2,3)
ans = 6
```

MATLAB implements similar functionality with its function handles, defined through the @ symbol.

4.2.6 For-loops and if-else statements

For-loops

The `for`-loop allows to iterate over a sequence, as in the following example:

```
>>> for i in [1, 2, 3, 4]:  # the colon is always required
...     print(i)
...
1
2
3
4
```

The *secondary prompt*, by default three dots (...), prompts for the next command in continuation lines. Because this is a multi-line statement, the Enter key ⏎ must be pressed a second time to tell the interpreter that we are finished with it.

We must be careful when assigning values to growing variables (variables increasing in size with each iteration) inside `for`-loops. In contrast to MATLAB approach, Python does not allow to address elements which have not been created previously. For example, in MATLAB we could state the following:

```
>> a = 1   # Matlab's primary prompt consists of two greater-than signs
>> a(4) = 2
>> print(a)
a = [1, 0, 0, 2]
```

and automatically, the vector a would increase its size to accommodate a fourth element equal to 2 (while filling the other newly created spaces with zeros). To accomplish this in Python, we would first have to define a as a list (a mutable sequence) of size 4 to pre-allocate "space" for new re-assignations:

```
>>> a = [0]*4   # manually creating a list of zeros
>>> print(a)
a = [0, 0, 0, 0]
>>> a[0] = 1
>>> a[3] = 2
>>> print(a)
a = [1, 0, 0, 2]
```

Lists of zeros will be the chosen method to pre-allocate space within this project, as to keep certain similarity with MATLAB's approach.

If-else statements

On the other hand, the `if` statement allows conditional execution of code, for example:

```
>>> a = 2
>>> if a > 0:
...     print('hello')
...
hello
```

The `if` statement can also have an `else` branch which is executed if the condition is wrong:

```
a = 2
if a > 0:
    print('a is positive')
else:
    print('a is non-positive')
```

Finally, the `elif` key word (read as “else-if”) allows checking for several (exclusive) possibilities:

```
a = 2
if a > 0:
    print('a is positive')
elif a < 0:
    print('a is negative')
else
    print('a is zero')
```

Table 4.2 Typical conditional expressions and logical operators.

`>`	is greater than	`>=`	is greater than or equal to	`and`	logical and
`<`	is lesser than	`<=`	is lesser than or equal to	`or`	logical or
`==`	is equal to	`!=`	is not equal to	`not`	logical not

4.3 Numerical Python (NumPy)

The NumPy package provides access to a new data structure called *array* (similar to MATLAB’s) which allow efficient vector and matrix operations. It also provides fundamental mathematical functions (`sin()`, `cos()`, `log()`, etc.), physical constants (`pi`, `e`, etc.) and a number of linear algebra operations.

An array appears to be very similar to a list, but an array can keep only elements of the same type (whereas a list can mix different kinds of objects).

Vectors (1D arrays)

The data structure we will need most often is a vector (a one-dimensional array). We can convert a list into an array using `numpy.array`:

```
>>> import numpy as np  # np is the standard abbreviation for numpy
>>> a = np.array([1, 2, 3, 4])
>>> print(a)
[1 2 3 4]
```

Alternatively, arrays can be generated through other NumPy functions:

- `numpy.arange(start, stop, step)` returns evenly spaced values within the half-open interval `[start, stop)`:

```
>>> a = np.arange(1,6,2) # if not indicated, the default step size is 1
>>> print(a)
[1 3 5]
```

```
>>> b = np.arange(3) # if only one argument is given, start=0 by default
>>> print(b)
[0 1 2]
```

- numpy.linspace(start, stop, num) is similar to numpy.arange but uses a given number of samples instead of the step size. Additionally, the [start, stop] interval is considered to be closed by default (this can be changed —these functions have additional optional input parameters which are not showed here):

```
>>> a = np.linspace(1,2,5) # if not indicated, num=50 by default
>>> print(a)
[1 1.25 1.5 1.75 2]
```

- numpy.zeros(size) creates an array of zeros:

```
>>> a = np.zeros(4)
>>> print(a)
[0. 0. 0. 0.]  # the elements are floating point numbers by default
```

Further information on these functions (and many more) can be found in the NumPy documentation [14].

Once the array is established, we can set and retrieve individual values in the same way we did with lists:

```
>>> a = np.zeros(4)
>>> a[1] = 3
>>> print(a)
[0. 3. 0. 0.]
>>> a[0]
ans = 3.0
```

and slicing also works similarly:

```
>>> print(a[0:3])
[0. 3. 0.]
```

We can perform calculations on every element in the array with a single statement:

```
>>> a = np.arange(3)
>>> print(a + 2)
[2 3 4]
>>> print(a**2)
[0 1 4]
```

To perform an element-wise operation of two vectors in MATLAB, we would need to place a dot before the operator (which results in the so-called *dot operators*).

Matrices (2D arrays)

The nature of two-dimensional arrays is very similar to that of 1D ones. Here are various ways to create a 2D array:

- By converting a list of lists into an array:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(a)
[[1 2 3]
 [4 5 6]]
```

- Using the `numpy.zeros(size)` method:

```
>>> a = np.zeros([2,3])
>>> print(a)
[[0. 0. 0.]
 [0. 0. 0.]]
```

- Using the function `numpy.meshgrid(x_range, y_range)`, which generates a pair of 2D coordinate arrays for the evaluation of 2D scalar/vector fields over 2D grids, given the one-dimensional coordinate arrays `x_range`, `y_range`:

```
>>> x_range = np.array([1, 2, 3, 4])
>>> y_range = np.array([0, 4, 8])
>>> X, Y = np.meshgrid(x_range, y_range)
>>> print(X)
[[1 2 3 4]
 [1 2 3 4]              -|--------->  (x axis)
 [1 2 3 4]]              |
>>> print(Y)             |
[[0 0 0 0]               |
 [4 4 4 4]               v  (y axis)
 [8 8 8 8]]
```

  Note how multiple simultaneous assignations (to `X` and `Y`) have been addressed through a comma separator.

The size of a matrix can be obtained like this:

```
>>> a = np.array([[1, 2], [ 3, 4], [5, 6]])
>>> a.shape
ans = (3, 2)  # a has 3 rows and 2 columns
```

and individual elements can be accessed and set using the following syntax:

```
>>> a = np.array([[1, 2], [ 3, 4], [5, 6]])
>>> a[2,0]  # element in the third row and first column
ans = 5
>>> print(a[:,2])
[1 2]  # 1D array
```

In this example we also appreciate two important characteristics of matrix slicing. First, a single colon (:) without starting or ending indices indicates that the whole row/column is being selected. Secondly, when the sliced array is either vertical (part of a column) or horizontal (part of a row), it is automatically transformed into a 1D vector. These do not take into account whether the orientation is vertical or horizontal (while MATLAB does).

4.4 Visualizing Data

In this section we succinctly review Matplotlib, which is a plotting library for Python and its numerical mathematics extension NumPy. Matplotlib allows us to generate high quality line plots, histograms, bar chats, scatter plots, 3D plots, etc., with just a few lines of code.

Plotting nearly always needs arrays of numerical data, and it is the reason that the `numpy` package is used a lot —it provides fast and memory efficient array handling for Python.

4.4.1 Plotting y=f(x)

For this project our focus will be on the module `matplotlib.pyplot`, which provides a MATLAB-like plotting framework. It is a collection of command style functions which make changes to a figure. This state-driven interface is more convenient to use for easy plots, although it is generally less flexible than the object-oriented Matplotlib interface.

Many of the examples in the Matplotlib documentation follow the convention of importing `matplotlib.pyplot` as `plt`. Generating simple line plots with `pyplot` is very quick:

```
# example 1
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-np.pi,np.pi,100)
y = np.sin(x)

plt.plot(x,y)
```

which produces the output shown in Fig. 4.2. For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are taken from MATLAB, and the color string can be concatenated with a line style string. The default format string is `'b-'`, which generates a solid blue line.

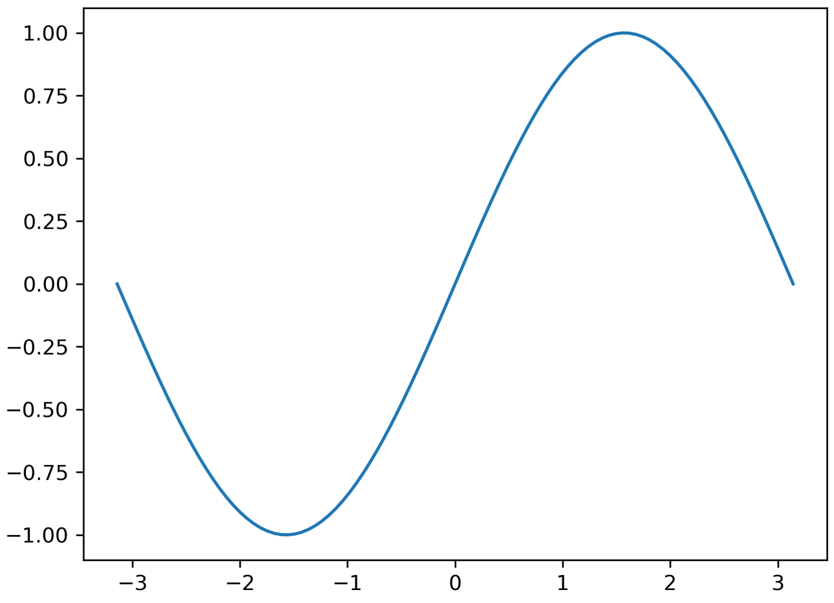
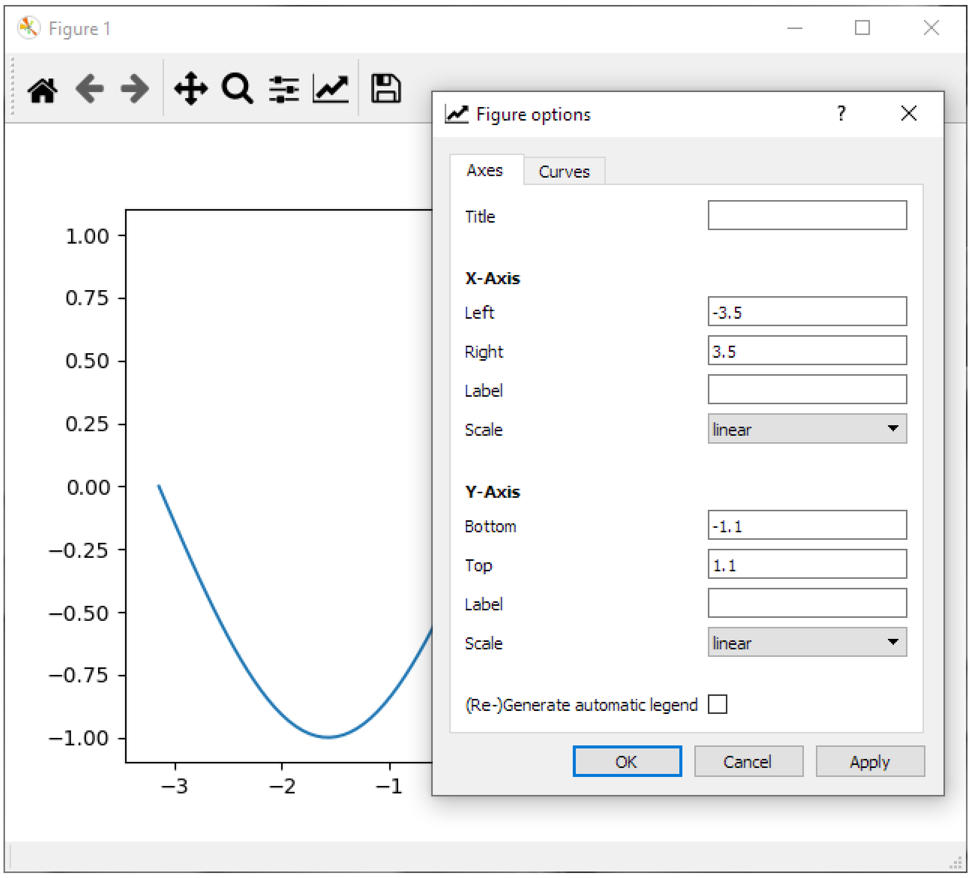


Figure 4.2 The output of *example 1*

Mimicking MATLAB's interface, Matplotlib also provides an interactive window where we can zoom, pan or edit the plot's basic attributes.



Title:
X-Axis
Left: -3.5
Right: 3.5
Label:
Scale: linear
Y-Axis
Bottom: -1.1
Top: 1.1
Label:
Scale: linear
(Re-)Generate automatic legend ☐

Figure 4.3 Matplotlib's interactive window.

4.4.2 Plotting more than one curve

By calling the plt.plot() command repeatedly, more than one curve can be drawn in the same graph. Alternatively, multiple x, y pairs can be introduced in one single command (just as in MATLAB):

```
plt.plot(x1,y1,x2,y2)
```

On the other hand, to display plots in separate figures we must create them by using the plt.figure() command. In the previous examples this had not been necessary because plt.plot() internally calls gca (get current axes), which then calls gcf (get current figure), which returns the current (last used) figure. If there is no current figure it automatically calls plt.figure() and returns the newly created figure.

For instance, a way to display two simple plots in two separate figures is:

```
plt.figure()
plt.plot(x1,y1)
plt.figure()
plt.plot(x2,y2)
```

To avoid confusing the students, a new figure will always be created before using the plt.plot() command.

4.4.3 Implicit functions and contour plots

Recall from Chapter 2 the two equations from Lambert's problem. To plot Lambert's equations we first need to express them as implicit functions, defined by equations of the type $F(x,y) = 0$. To plot implicit functions, a contour plot is the most suitable tool. These can be generated in Python using the command plt.contour():

```
# example 2
x_range = np.linspace(-10,10,200)
y_range = np.linspace(-10,10,200)
X, Y = np.meshgrid(x_range, y_range)

F = X**2 + Y**2

plt.figure()
plt.contour(X,Y,F,[1, 4, 9]) # plots contours at F=1, F=4 and F=9
```

This code plots three circles of radius 1, 2 and 3, although an arbitrary number of contour lines of the function F(X,Y) could be generated.

4.4.4 3D plots

Unfortunately, 3D plots are not supported within `pyplot`. Therefore, some object-oriented programming (although quite simple) is required. For our purposes, the simplest way of plotting 3D data is to import the sub-module `mplot3D` from the module `mpl_toolkits`:

```
from mpl_toolkits import mplot3d
```

Then, 3D axes can be created through the command `plt.axes(projection='3d')`, which generates an Axes3D object. Then, plots can be added to the Axes3D. This approach is object-oriented because instead of writing `plt.*` and letting `matplotlib.pyplot` guess to what figure we are referring to, we are responsible to refer the Axes3D object (which in turn is associated with a Figure object).

During the laboratory class we will use 3D plots to visualize the orbits of the different planets and the transfer orbit between them (the solution to Lambert's problem). Plotting a simple 3D curve is straightforward:

```
# example 3
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

x = np.linspace(0,30,300)
y = np.sin(x)
z = np.cos(x)

plt.figure()
ax = plt.axes(projection='3d')  # an Axes3D object is created
ax.plot3D(x,y,z)  # a plot is added to the Axes3D
```

which produces the output shown in Fig. 4.4. To plot single points (which can represent planets), the command `plt.scatter3D(x,y,z)` is used.

It is worth mentioning that the 3D plot window interactive capabilities are limited with respect to the 2D version. For instance, panning is not available and zooming can only be achieved by holding the mouse's right button, although this only zooms the central portion of the graph. Additionally, accessing the figure options window can result in unwanted shifting of the plot within the figure window (the interface is somewhat buggy). Therefore, it is reasonable to avoid using the interactive interface in 3D plots.

4.4.5 Setting and/or editing a plot's attributes

A plot's attributes can be set and/or edited using commands instead of the interactive window. The syntax used differs between 2D plots created with `matplotlib.pyplot` and 3D plots created with regular (object-oriented) Matplotlib commands.

2D plots

- Set the title: `plt.title()` with a string as an input[1].
- Label the axes: `plt.xlabel()`, `plt.ylabel()` with a string as an input.
- Set custom limits of the axes: `plt.xlim()`, `plt.ylim()` with input `[min, max]`.
- Set the scaling of both axis to be equal (useful to correctly visualize the eccentricity and general shape of an orbit): `plt.gca().set_aspect('equal')`.

Note that the command `plt.gca().set_*` (object-oriented, as it acts on the axes —in this case the current axes) can also be used to set the title, axes and limits of a 2D plot.

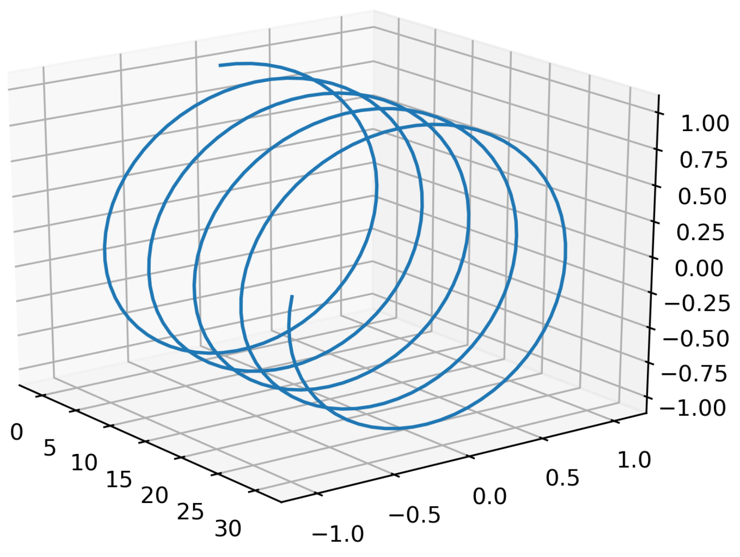


Figure 4.4 The output of *example 3*.

3D plots

- Set the title: `Axes.set_title()` with a string as an input.
- Label the axes: `Axes.set_xlabel()`, `Axes.set_ylabel()`, `Axes.set_zlabel()` with a string as an input.
- Set custom limits of the axes: `Axes.set_xlim()`, `Axes.set_ylim()`, `Axes.set_zlim()` with input `[min, max]`.

These commands need to be referred to an Axes or Axes3D object.

[1] It is possible to have LaTeX formatting in Matplotlib graphs (for titles, labels, etc.). This is accomplished by adding an `r` before the actual Python string (`r'$ \alpha $ is a greek letter'`). This makes the string a *raw string literal*, which accepts a backslash as a valid string literal instead of considering it as the start of an escape sequence.

4.4.6 Saving the figure to a file

By clicking the save button of the interactive window, a medium resolution (100 dpi) image in PNG format is saved to the selected directory. However, it is possible to save images in higher resolution (higher dpi) PNG files, or even as vector graphics. The command `plt.savefig()` saves the current figure (the last one created) to the working directory (shown at the top bar in Spyder):

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure()
plt.plot([1, 2, 3], [1, 4, 9])

plt.savefig("name.png", dpi=300, bbox_inches='tight')  # raster image
plt.savefig("name.pdf", bbox_inches='tight')  # vector image
```

Usually a 300 dpi PNG image is good enough for most printed or web publications, and the PDF format is commonly used for vector graphics (although other popular formats such as SVG or EPS are also supported). By default, `plt.savefig()` leaves wide white margins around the actual graph. This can be addressed by adding `bbox_inches='tight'`, which tightly adjusts the margins.

4.5 Python, NumPy and Matplotlib for MATLAB users

In this section a brief summary of the key differences between Python and MATLAB syntax is provided. The purpose of this comparison is to create a concise *cheat sheet* with the most frequent commands and expressions, to aid the student in the otherwise tedious task of searching and remembering this information. Part of this content is based on the helpful table of NumPy-MATLAB equivalents by The SciPy community [15].

4.5.1 General purpose equivalents

Table 4.3 Slicing and indexing.

MATLAB	Python	Notes
`a(2,5)`	`a[1,4]`	Access element in second row, fifth column
`a(2,:)`	`a[1,:]`	Entire second row of a
`a(1:5,:)`	`a[0:5,:]`	First five rows of a

```
                  1     2     3     4     5    <-- indexing
                +-----+-----+-----+-----+-----+
       Matlab   | 'H' | 'e' | 'l' | 'l' | 'o' |
                +-----+-----+-----+-----+-----+
                  1     2     3     4     5    <-- slicing
```

```
                  0     1     2     3     4    <-- indexing
                +-----+-----+-----+-----+-----+
       Python   | 'H' | 'e' | 'l' | 'l' | 'o' |
                +-----+-----+-----+-----+-----+
                0     1     2     3     4     5  <-- slicing
```

Table 4.4 Basic commands.

MATLAB	Python	Notes
`help func`	`help(func)`	Get help on the function `func`

`a && b`
`a || b`
	`a and b`
	`a or b`
		Logical operators

`~=`	`!=`	Conditional expressions
`^`	`**`	Basic operators
`[x, y] = 2output_fun()`	`x, y = 2output_fun()`	Multiple assignments
`disp(a)`	`print(a)`	Display a result
`;` (semicolon)	(nothing)	Terminate a statement
`%`	`#`	Comment

`for i=[1 2 3]`
`  disp(i)`
`end`
	`for i in [1, 2, 3]:`
	`  print(i)`
		For-loops

`if a==0`
`   disp('a is zero')`
`elseif a<0`
`   disp('a is negative')`
`else`
`   disp('a is positive')`
`end`
	`if a==0:`
	`   print('a is zero')`
	`elif a<0:`
	`   print('a is negative')`
	`else:`
	`   print('a is positive')`
		If-else statements

`function [x, y] = pm(a,b)`
`   x = a + b;`
`   y = a - b;`
`end`
	`def = pm(a,b):`
	`   x = a + b`
	`   y = a - b`
	`   return [x, y]`
		Regular functions

`mult = @(a,b) a*b`	`mult = lambda a,b: a*b`	Anonymous functions

4.5.2 NumPy equivalents

Before using the NumPy package, remember to execute the following command in Python:

```
import numpy as np
```

The table below gives some rough equivalents for some common MATLAB expressions. For more detail read the built-in documentation on the NumPy functions.

Table 4.5 Linear algebra equivalents.

MATLAB	Python	Notes
`size(a)`	`shape(a)` *or* `a.shape`	Get the size of an array
`size(a,n)`	`shape(a)[n-1]` *or* `a.shape[n-1]`	Get the number of elements of the *n*-th dimension of array a
`[1 2 3]`	`np.array([1, 2, 3])`	Create vectors
`[1 2 3; 4 5 6]`	`np.array([[1, 2, 3],[4, 5, 6]])`	Create matrices

`a.*b`
`a.\b`
`a.^b`
	`a*b`
	`a\b`
	`a**b`
		Element-wise operations

`2:10`
`0:9`
	`np.arange(2,11)`
	`np.arange(9)`
		Create an increasing vector

`zeros(3,4)`
`ones(3,4)`
	`np.zeros([3,4])`
	`np.ones([3,4])`
		Array full of zeros or ones

`linspace(1,3,4)`	`np.linspace(1,3,4)`	Four equally spaced samples between 1 and 3, inclusive
`max(a)`	`a.max()`	Maximum element of a
`max(a)`	`a.max(0)`	Maximum element of each column of matrix a
`max(a,[],2)`	`a.max(1)`	Maximum element of each row of matrix a

4.5.3 Matplotlib equivalents

Before using the Matplotlib package, remember to execute the following command in Python:

```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
```

2D Plots

Plotting a simple 2D line plot:

Matlab

```
theta = linspace(-pi,pi,100);
f1 = sin(theta);
f2 = sin(theta);

figure
plot(theta,f1,theta,f2)
title('Trigonometric functions')
xlabel('$\theta$', 'interpreter',...
'latex')
ylim([-1,1])
```

Python

```
theta = np.linspace(-np.pi,np.pi,100)
f1 = np.sin(theta)
f2 = np.sin(theta)

plt.figure()
plt.plot(theta,f1,theta,f2)
plt.title('Trigonometric functions')
plt.xlabel(r'$\theta$')

plt.ylim([-1,1])
```

Plotting dots:

Matlab

```
x = [0 2 4]
y = [5 4 3]

figure
scatter(x,y)
```

Python

```
x = np.array([0, 2, 4])
y = np.array([5, 4, 3])

plt.figure()
plt.scatter(x,y)
```

3D Plots

Plotting a 3D curve:

Matlab

```
x = linspace(0,2*pi,100);
y = sin(x);
z = sin(x);

figure

plot3(x,y,z)
title('Helix')
xlabel('Longitudinal axis')
ylim([-1,1])
zlim([-1,1])
```

Python

```
x = linspace(0,2*np.pi,100)
y = np.sin(x)
z = np.sin(x)

plt.figure()
ax = plt.axes(projection='3d')
ax.plot3D(x,y,z)
ax.set_title('Helix')
ax.set_xlabel('Longitudinal axis')
ax.set_ylim([-1,1])
ax.set_zlim([-1,1])
```

Plotting dots:

Matlab

```
scatter3(x,y,z)
```

Python

```
ax.scatter3d(x,y,z)
```

Plotting the contour line at $Z(X,Y) = 0$:

Matlab

```
x = linspace(-pi,pi,100);
y = linspace(0,2*pi,100);
[X, Y] = meshgrid(x,y);

Z = sin(X) + cos(Y)

figure
contour(X,Y,Z,[0 0])
```

Python

```
x = linspace(-np.pi,np.pi,100);
y = linspace(0,2*np.pi,100);
X, Y = meshgrid(x,y);

Z = sin(X) + cos(Y)

plt.figure()
plt.contour(X,Y,Z,0)
```

Saving a figure to a file in Python

To obtain a raster image (.png):

```
plt.savefig("name.png", bbox_inches='tight', dpi=300)
```

and to obtain a vector image (.pdf, .eps, .svg, etc.):

```
plt.savefig("name.pdf", bbox_inches='tight')
```

5 Development of a laboratory class on Lambert's problem

Throughout Chapter 2, Lambert's equations have been thoroughly reviewed for all possible cases (elliptic, parabolic and hyperbolic transfer orbits). Additionally, a few essential auxiliary algorithms have been introduced in Chapter 3, and a comprehensive introduction to Python has been made in Chapter 4. Once the theoretical framework has been laid out, we can proceed to tackle the development of a Python-based orbital mechanics laboratory class.

During this chapter, both the structure and contents of the laboratory class are addressed. Within this context, a straight-forward Lambert algorithm for the elliptic case is developed and duly justified. Moreover, some insight on Lambert's equations is given, and we carefully analyze the graph representing their solution for a wide range of input data. Finally, a practical application regarding interplanetary missions is presented.

The main deliverable of this project consists of a laboratory report which will serve as a guideline during the laboratory class. It consists of a series of examples, which include code that the students may reproduce on their own, interspersed with the required theoretical contents. The finished document can be found in Appendix A, and throughout this chapter a detailed review of its main aspects is carried out, showing the multiple results and conclusions which are to be later obtained by the student.

5.1 Structure of the laboratory report

Besides teaching basic Python and introducing Lambert's problem, another main objective pursued by this laboratory class is to include a practical application of the contents presented, in the context of interplanetary missions. Consequently, the laboratory report is reasonably split into three main parts:

- Programming with Python
- Lambert's problem
- Analysis and optimization of interplanetary missions

First, a brief introduction to Python is made so that the students get familiarized with the language and Spyder, the IDE of choice for this project. Then, two basic examples with relation to orbital mechanics are carried out to aid the student in the understanding of Python programming. In the first example, a simple algorithm which computes the orbital position as a function of time (relative to periapsis passage) is programmed. At this stage, NumPy is also introduced, along with some basic Python syntax. The goal of the second example is to plot Mercury's orbit and introduce Matplotlib.

At the beginning of the second part, Lambert's equations are derived following the steps introduced in Sect. 2.4. Then, a third example is included to visualize the equations for elliptic motion and analyze their solution. Before diving into the analysis and optimization of interplanetary missions, the algorithm which solves Lambert's problem is programmed.

The last part consists mainly of two examples regarding spacecraft missions from Earth to Mars. The first one analyzes the path of the Mars Science Laboratory mission (2011), and its transfer orbit is plotted. Then, a brief theoretical section introducing pork-chop plots and the optimization of interplanetary missions through the characteristic energy (C_3) is included, after which a second example on the Mars Reconnaissance Orbiter (2005) is carried out (proving that its launch date is indeed the optimal solution and minimizes the total C_3 required). Finally, some further insight on the Mars launch opportunity windows are given, which allow to plan future missions ahead.

At this stage, an (optional) additional problem can be given by the teacher for the students to finish at home. All the tools required to solve this problem would have been reviewed during the laboratory class —this provides endless possibilities for the teacher, as these tools are not only prepared for an Earth-Mars mission, but include generic algorithms and data from every major planet within the Solar System.

5.2 Part 1: Programming with Python

At the beginning of each part, a brief comment is made to motivate the students and to describe the tasks the will be completing:

In this section, you will learn the basics of Python programming, which is widely used in the space industry. The language is similar to MATLAB in many aspects, so you will be doing complex tasks in just a few minutes!

After reviewing some basic notions on Python, Anaconda and Spyder, the first example is introduced. Here, boxes with code are placed in between paragraphs to aid the student in programming the script (just as it is done in this document throughout Chapter 4).

In the second example, some of the auxiliary functions provided through a custom module, `astrofun.py` (whose contents can be found in Appendix B), are used to plot Mercury's orbit. This specific orbit has been chosen because it has a relatively large eccentricity ($e = 0.21$) compared to the other Major Planets, and therefore a more *elliptic* shape can be appreciated.

More details on both *Example 1* and *Example 2* an be found in the laboratory report, together with the required code. The results obtained when the script from *Example 2* is run are presented in the following section.

5.2.1 Example 2: Plotting Mercury's orbit - Results

First, the students are asked to plot the complete orbit by computing Mercury's orbital period and choosing two dates spaced accordingly:

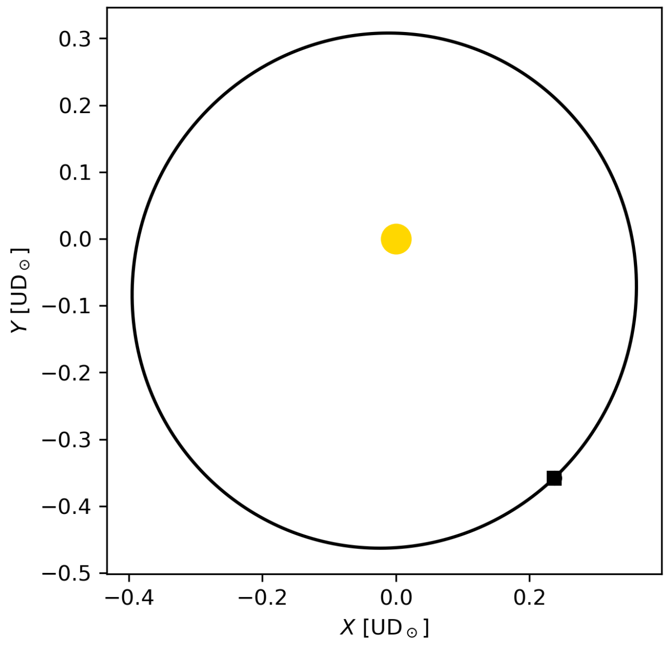


Figure 5.1 Mercury's complete orbit around the Sun.

In Fig. 5.1, a square marker has been plotted to represent the planet's starting position, and a circle to represent the ending position. Just as we would expect, these overlap when a full orbit has been completed. However, the student is prompted to zoom in at this region:

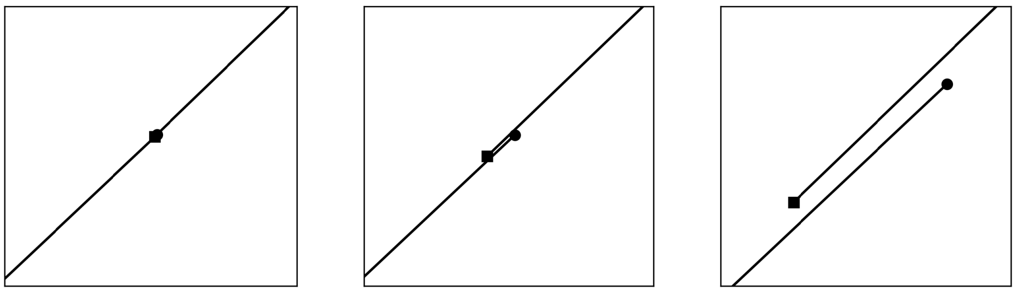


Figure 5.2 Consecutive zooms on the starting/ending region.

which reveals that in reality this is not the case. Recall that the propagator introduced in Chapter 3 (which is the one implemented in `astrofun.py`) is linear, and includes the effect of perturbations. These can be of various natures, and their overall impact on a planet's orbit translates into rates of change of the different orbital elements. Their effect can be clearly appreciated through this example.

Next, the students are asked to plot the arc traveled by Mercury in a 30-day month. Here, the 3D plotting capabilities of Matplotlib are introduced, and two figures are generated:

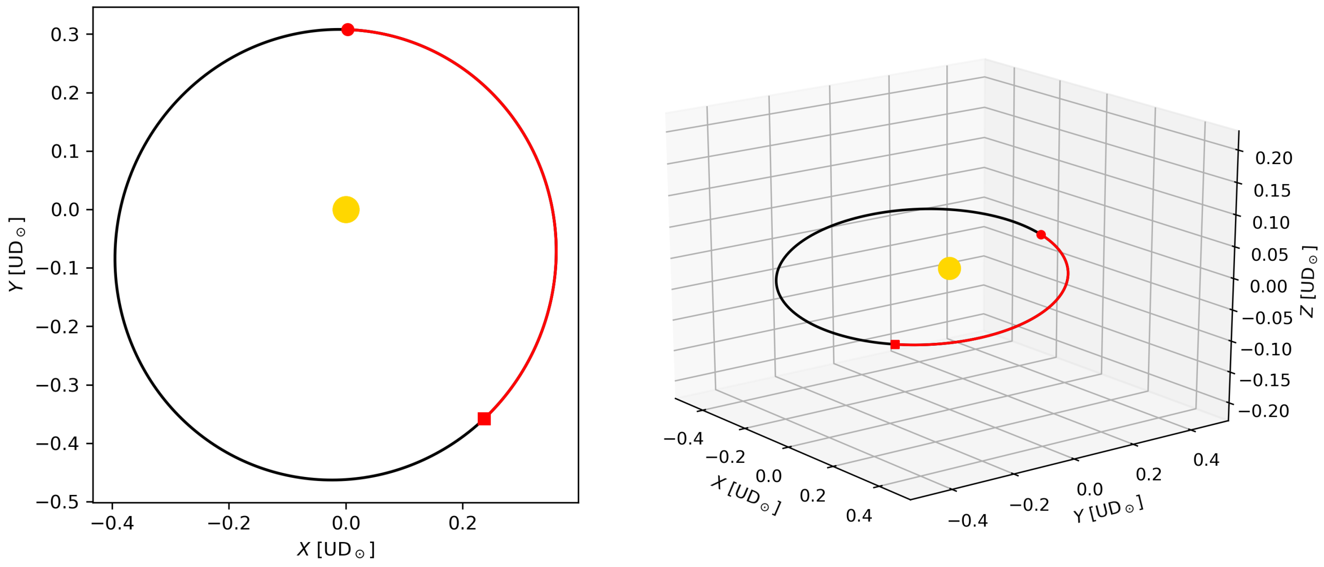


Figure 5.3 Arc traveled by Mercury in a 30-day month.

It must be noted that the 2D representation does not reflect the true measure of the real orbit, but is a projection in the XY axis instead. However, its angle of inclination is approximately $7°$, and therefore the differences are small.

5.3 Part 2: Lambert's problem

Solving Lambert's problem is crucial to design realistic interplanetary transfers and maneuvers. We don't do it by hand because it requires solving two nonlinear equations with two unknowns. Here we learn how to make our computer do it for us; then you can relax and start designing your first missions!

At the beginning of this part, Lambert's equations for elliptic motion are obtained following an identical procedure to that in Sect. 2.4. These equations constitute a system of two equations with two unknowns:

$$J \sin^3 \frac{\alpha}{2} = 2m\pi + \alpha - \beta - \sin\alpha + \sin\beta,$$

$$\sin\frac{\beta}{2} = q \sin\frac{\alpha}{2},$$

which can be visualized as two curves whose intersection becomes the solution to Lambert's problem. To analyze their behaviour, the two curves can be plotted for various values of q, J and m. A new example is introduced to accomplish this through Python, using Matplotlib. Below, the results which are to be obtained by the student are presented, and further conclusions are drawn.

5.3.1 Example 3: Visualizing Lambert's equations - Results

To plot Lambert's equations, both need to be expressed as implicit functions:

$$F_1(\alpha,\beta) = 2m\pi + \alpha - \beta - \sin\alpha + \sin\beta - J\sin^3\frac{\alpha}{2} = 0, \tag{5.1}$$

$$F_2(\alpha,\beta) = \sin\frac{\beta}{2} - q\sin\frac{\alpha}{2} = 0, \tag{5.2}$$

which can then be plotted through the `plt.contour()` function:

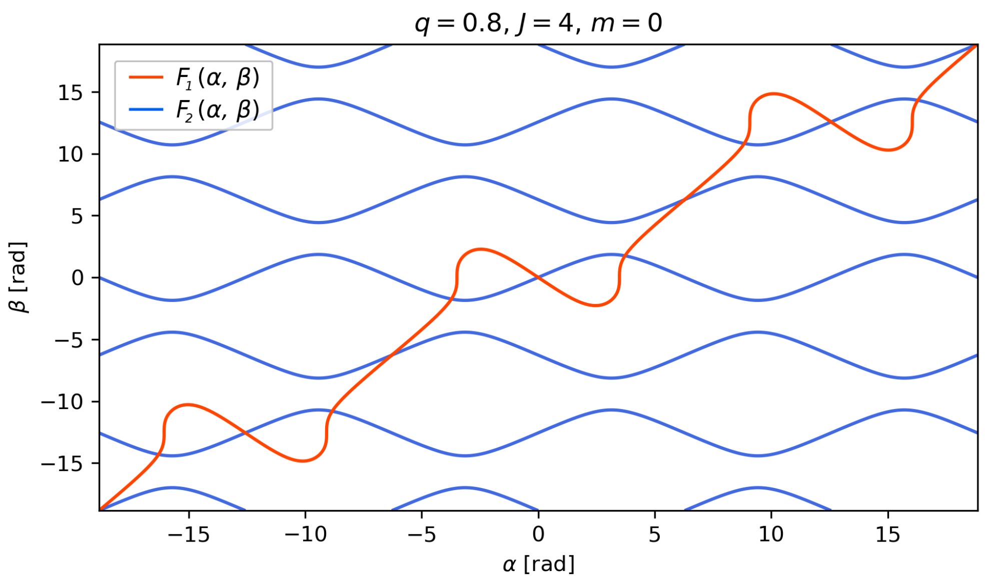


Figure 5.4 Global view of (5.1) and (5.2).

Notice how $F_2(\alpha,\beta) = 0$ is in reality an infinite grid-like pattern, and multiple intersections occur between both curves. The bounds $0 < \alpha < 2\pi$ and $-\pi < \beta < \pi$ introduced in Sect. 2.4.2 are required to choose the correct solution. Zooming into the region delimited by these bounds:

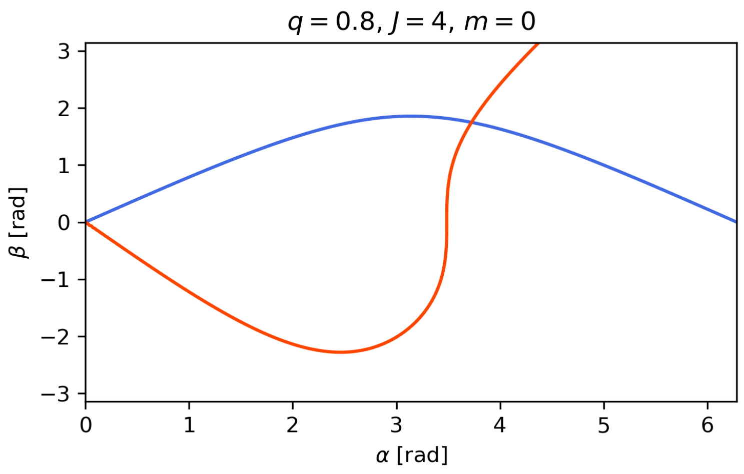


Figure 5.5 (5.1) and (5.2) in the valid region.

Two intersections can be appreciated. However, the one at $\alpha = \beta = 0$ always occurs, and must be disregarded. Recall that these equations are only valid for elliptic motion. Values of α near 0 imply $a \longrightarrow \infty$ according to (2.69), approaching a parabolic solution. When $m = 0$, (5.1) breaks down for $\alpha = 0$ and suffers from a critical loss of significant digits in the neighborhood of $\alpha = 0$.

To observe why this happens, students are prompted to plot the equations for $q = -0.5$, $m = 0$ and a range of J between 10 and 1.6. When J becomes smaller (a shorter time of flight), a more direct and quick path is required between the starting and arrival positions. The transfer orbit therefore begins to approach a parabola. In Fig. 5.6, (5.1) has been plotted at uniform intervals of J. Note how the solution drastically changes for small variations in J at lower values.

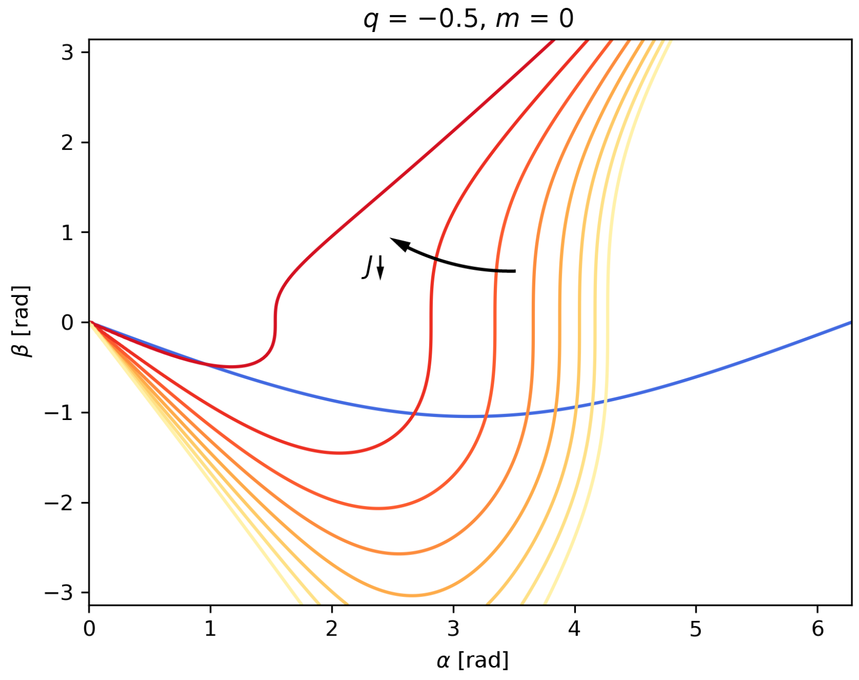


Figure 5.6 Behaviour of (5.1) with equally spaced changes in J.

When a clear intersection between (5.1) and (5.2) occurs, the transfer orbit is elliptic and the solution is valid. When both curves become tangent at $\alpha = \beta = 0$ ($q = -0.5$, $m = 0$ and $J = 1.5$), a parabolic transfer orbit is reached, which can't be solved using these equations. Hyperbolic transfer orbits appear when none of the previous conditions are met ($0 < J < 1.5$).

To conclude this example, a graph plotting J against α for selected values of q and m (Fig. 5.7) is presented to the students. Here, the conclusions obtained earlier can be clearly appreciated. When $m = 0$, for a fixed value of q there exists a lower bound for J, below which no solution can be obtained. This graph is very similar to that reviewed in Sect. 2.4. Even though J has been plotted against α instead of the universal parameter x, the two share identical characteristics. In Fig. 5.7, only the part associated with elliptic orbits is shown. Its most striking feature consists in the gaps (unrealizable regions) that have been shaded in the figure. Another interesting observation is that in the case $m \neq 0$ for given values of J and q, two different elliptic orbits are possible solutions to the problem.

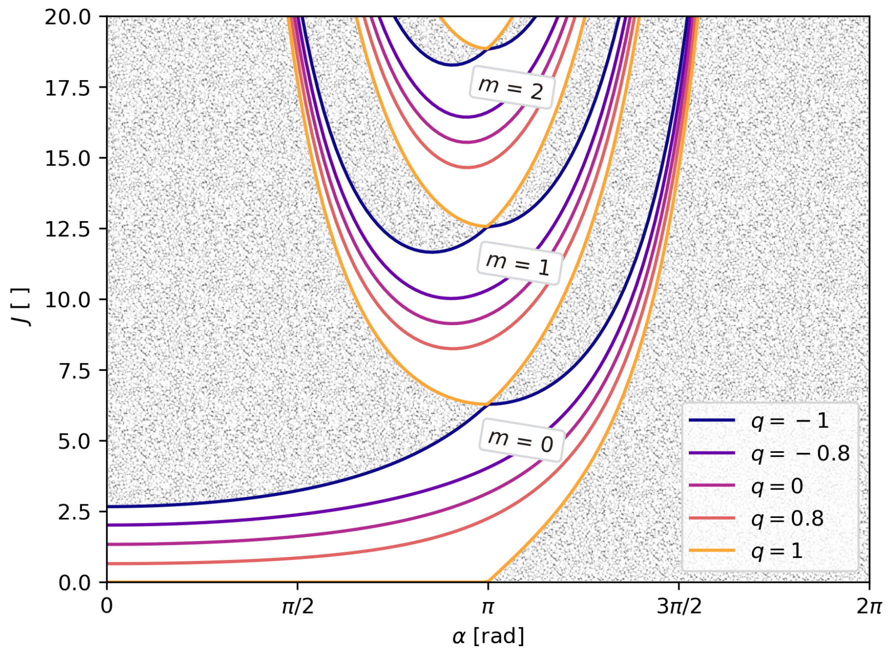


Figure 5.7 Plot of J against α for selected values of q and m.

To visualize the multi-revolution case, the transfer orbits of a generic Earth-Mars mission with $m = 1$ can be plotted:

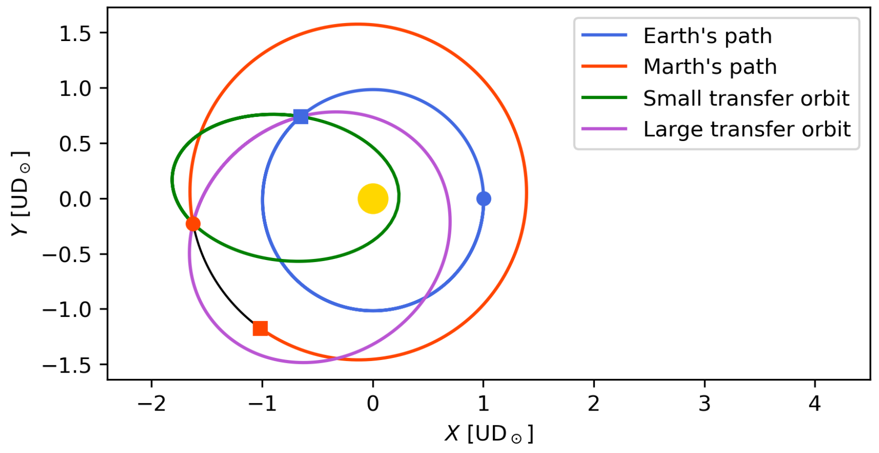


Figure 5.8 Duplicity of solutions for the multi-revolution case.

Observing Fig. 5.7, this is true for $m \neq 0$. In view of this duplicity of solutions for the multi-revolution case, the task of developing a simple algorithm using refined root algorithms such as `fsolve()` is more complex, and lays beyond the scope of this project[1]. However, in the context of direct transfer orbits with no DSM (Deep Space Maneuvers), single-revolution orbits tend to be the optimal solution. Therefore, it seems reasonable to limit ourselves to the case with $m = 0$, which is considerably easier to solve.

[1] Additional remarks regarding the multi-revolution case and its increased difficulty can be found in the paper by Gooding [2].

Theoretically, one could find the solution to Lambert's problem by simply plotting the equations and finding the intersection, or locating it in the graph from Fig. 5.7. However, to optimize interplanetary missions it needs to be solved a large number of times, and therefore a numeric approach is the only feasible option. In the following section, an algorithm to accurately solve Lambert's equations for elliptic motion and the single-revolution case is developed.

5.3.2 Developing a simple algorithm to solve Lambert's problem

As it has been stated previously, Lambert's equations can be solved using refined root finding algorithms such as `fsolve()`, readily accessible in Python. However, bounds can't be set to the variables with `fsolve()`. Therefore, it might not be able to find the correct solution among the many others which can be observed in Fig. 5.4.

To find a solution within $0 < \alpha < 2\pi$ and $-\pi < \beta < \pi$, constrained optimization is perhaps the smartest way to proceed. The `least_squares()` function is convenient here; equations can be directly passed to it, and it will minimize the sum of squares of its components. In other words, it will find the values of α and β which at the same time bring (5.1) and (5.2) closer to 0 (which is what we are looking for). However, optimization functions such as `least_squares()` are not built to solve non-linear systems. They can be tricky to set up and in some cases become quite unstable.

In this section, we analyze both functions to choose the one with better performance at this particular set of equations.

Choosing an appropriate initial estimation

Both `fsolve()` and `least_squares()` require an initial estimation of the independent variables (α and β) to begin their iteration processes. A complete discussion on this topic can be found in Gooding's papers (including [2]), and involves rather complex heuristics. For this project, a simpler approach is developed. Three different methods are proposed, with increasing complexity. Through comprehensive testing, the one which provides better overall results is chosen.

The first method, which we will call **Method A**, is based on a single initial estimation, valid for all cases. Choosing the central values $\alpha = \pi$ and $\beta = 0$ seems a reasonable guess, as they become average values for both variables.

Method B goes one step beyond. Because a graph representing the complete solution exists (Fig. 5.7), a value of α can be approximated according to J. Then, a corresponding initial estimate of β can be computed from (5.2). To keep it simple, the complete solution graph is split into three regions, which can be visualized in Fig. 5.9.

Along the same lines, **Method C** utilizes the graph in Fig. 5.7 to estimate α and then obtains β from (5.2). However, instead of only using J, multiple points of the graph are stored in an array (visualize a matrix where the rows correspond to values of J, and columns to values of q. The actual numbers stored in the matrix correspond to the values of α solution to each pair of q and J). Then, given a pair of values of q and J, the closest corresponding value of α found in the array serves as the initial estimate.

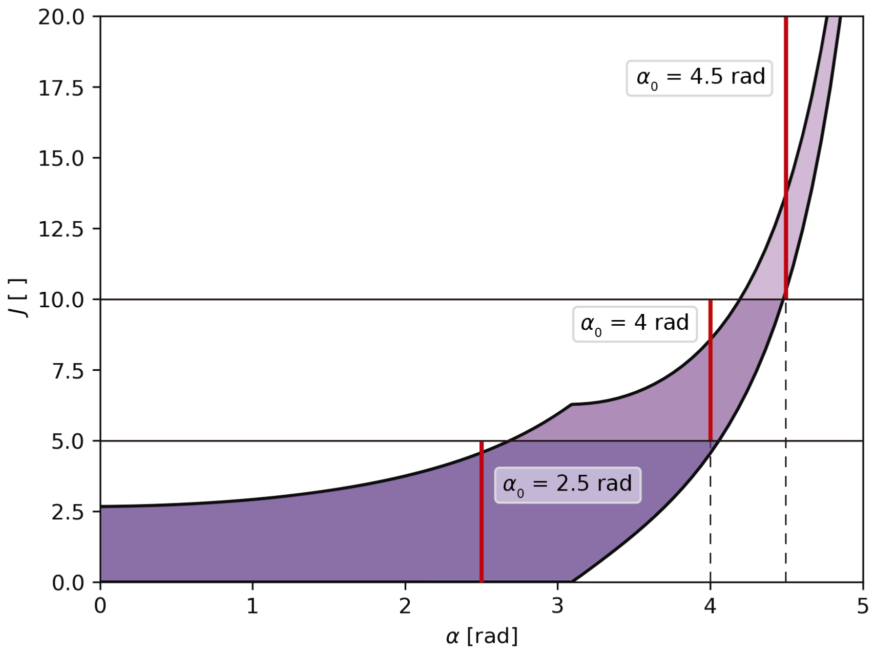


Figure 5.9 Choice of an initial estimate of α according to J.

Global convergence and accuracy

Now, each function can be tested for a wide range of initial data ($q \in [-1, 1]$, $J \in [0.1, 100]$ and of course $m = 0$). To evaluate their performance, they can be compared against a graphic method (which is much slower and less efficient, but provides very accurate results).

The validity of this approach has been tested by comparing a (visually verified) solution obtained through `fsolve()` to that provided by the graphic method. The error term for α (which will be denoted by the Greek letter ϵ, and is not to be confused with the specific energy reviewed in Chapter 2) turns out to be $\epsilon \sim 10^{-8}$. This value coincides with the default tolerance provided by `fsolve()`, and therefore it is reasonable to consider the graphic solution to be close enough to the real solution for comparison purposes.

First, the global convergence and accuracy are checked. A solution obtained through either `fsolve()` or `least_squares()` will be considered valid if the error term for α (compared to the graphic solution) is less than 10^{-2}. The purpose of this analysis is to visually identify whether the correct solution, or a different intersection point from those shown in Fig. 5.4, has been chosen.

🟨 hyperbola 🟥 ϵ > 1e−2 🟩 ϵ < 1e−2

Figure 5.10 Color code employed to test the convergence.

In addition to the color code illustrated in Fig. 5.10, the notation $v_0 \longrightarrow A$ is used to indicate which initial estimation method has been used. v_0 represents a vector containing the initial values of α and β, and in the previous example they would have been obtained following Method A.

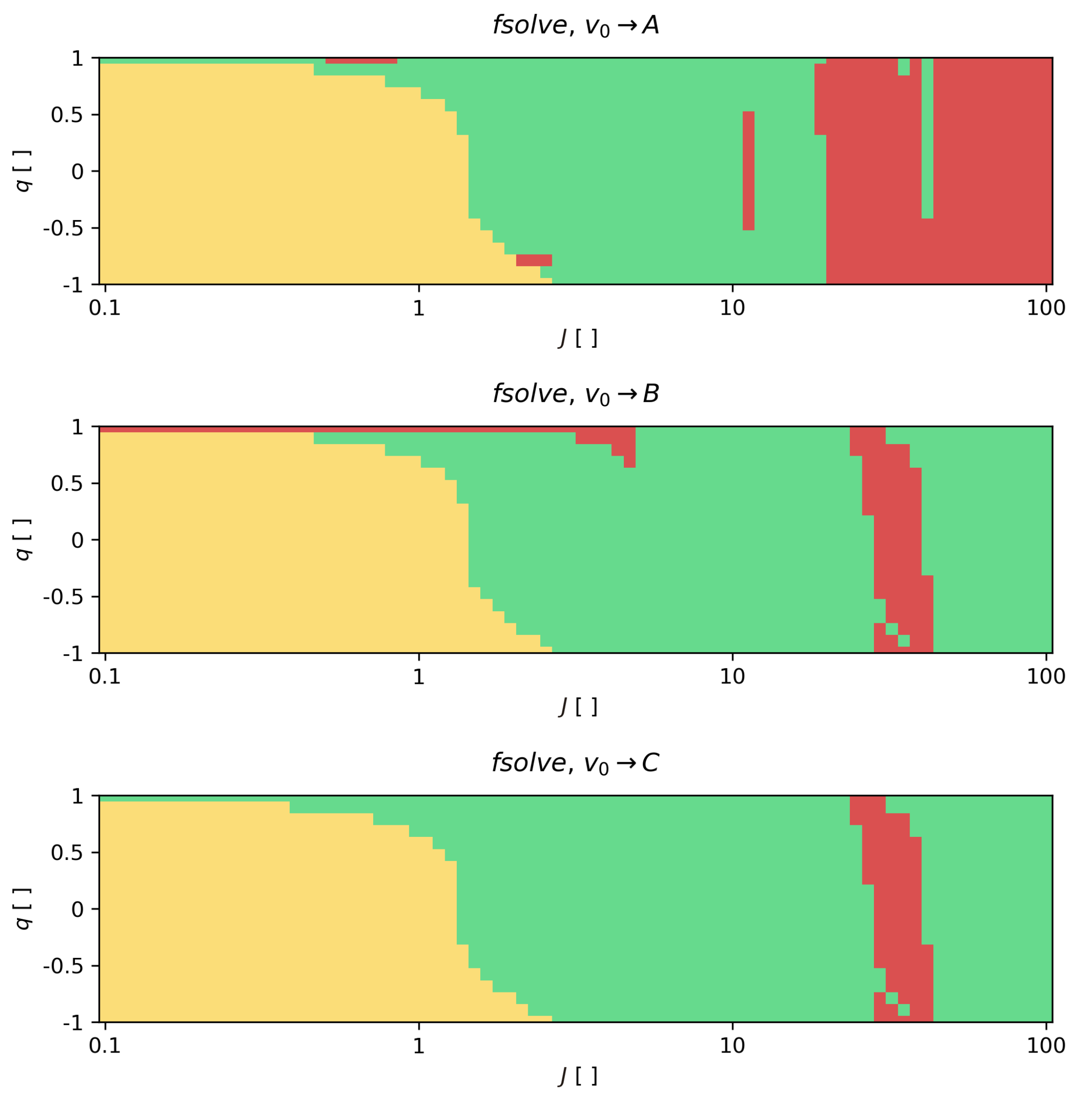
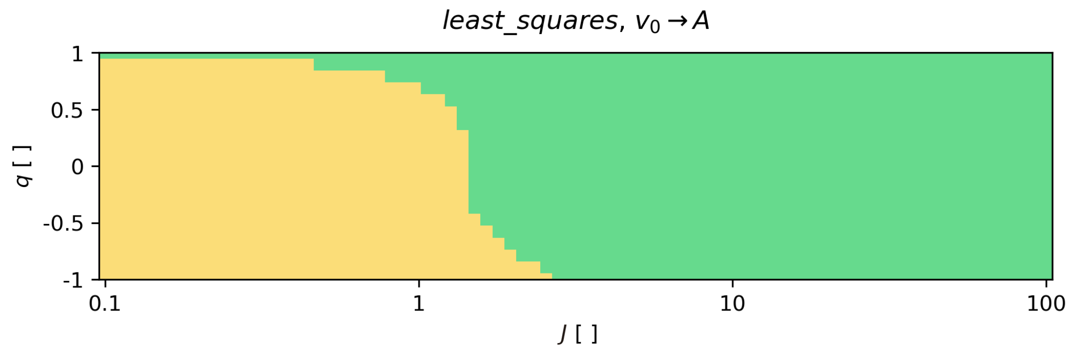


Figure 5.11 Convergence of `fsolve()` on a logarithmic scale.

A logarithmic scale has been employed to focus on the region where the solution approaches an parabola, as it can become quite unstable (a behaviour shown in Fig. 5.6). However, `fsolve()` seems to be performing reasonably well here. Not so much around $J \sim 35$, where red strips appear for all cases (and imply inaccurate solutions). On the other hand:



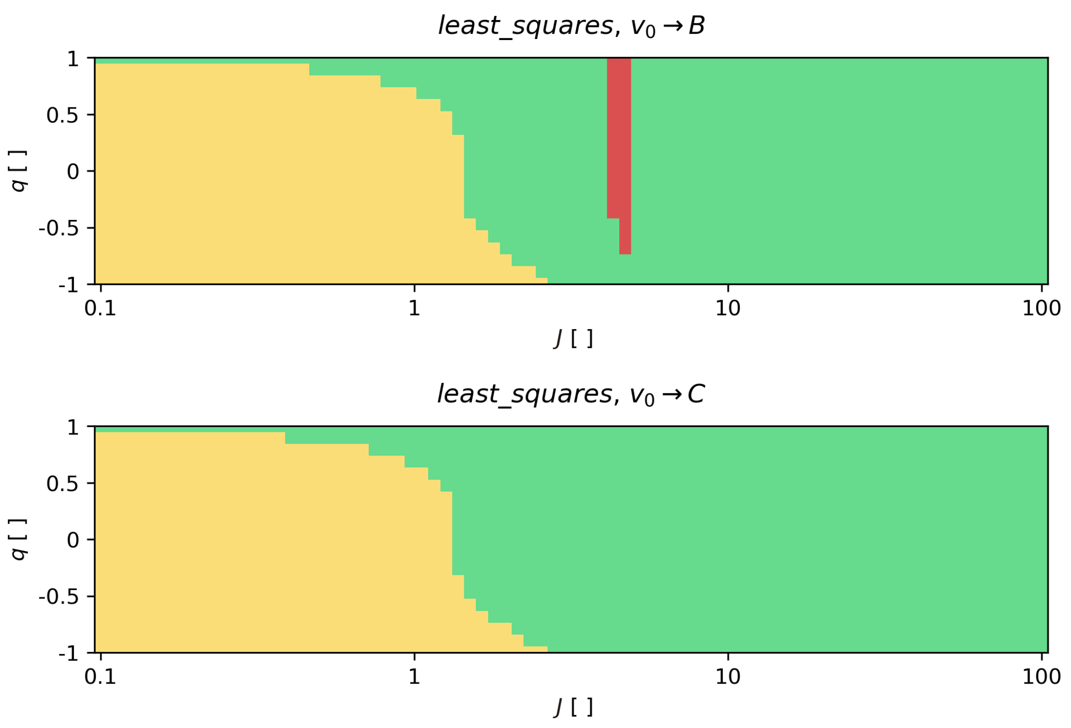


Figure 5.12 Convergence of `least_squares()` on a logarithmic scale.

The function `least_squares()` appears to be doing a much better job on locating the correct solution. This should be expected, as the independent variables have been bounded and therefore other solutions are automatically discarded. However, when using Method B to make the initial estimation, a thin red stripe can be seen around $J \sim 5$. Recall that this value of J has been used to split the region in Fig. 5.9, and therefore it appears that `least_squares()` somehow struggles when α needs to be decreased from its initial guess at this particular section.

Additionally, the series expression for J introduced in Sect. 2.4 can be analyzed:

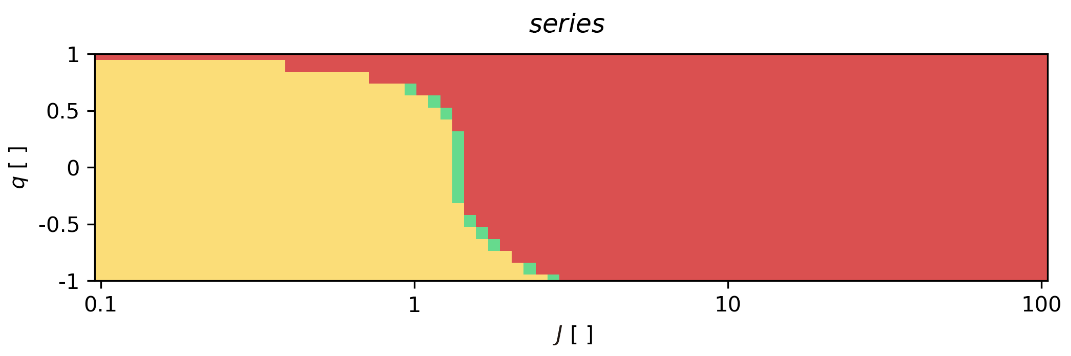


Figure 5.13 Convergence of the series expression for J on a logarithmic scale.

and just as would be expected, the solution converges only when it closely approaches a parabola. The main advantage of this expression is that it actually provides a solution for the parabolic case, while this is not possible using Lambert's equations for elliptic motion.

From the results shown in Figs. 5.11, 5.12 and 5.13, a few conclusions can be drawn. First, Method B, although simpler, has proven to perform worse than Method C for both `fsolve()` and `least_squares()` and therefore will be discarded. Moreover, Method A and C provide similar results when paired to the `least_squares()` function, and for simplicity Method A is preferred.

On the other hand, even though `least_squares()` (using Method A) seems to be providing good results, only convergence is being analyzed. To determine whether this approach is valid, additional insight into the precision of these results is required.

Testing the precision

To study the precision of the solutions provided by `fsolve()` and `least_squares()` (and the series expression), they can be compared to those obtained through the graphic method. The results are then expressed in terms of the error term for α, ϵ, but this time much more detailed graphs will be generated. A new color code to differentiate between different values of ϵ is introduced in Fig. 5.14.

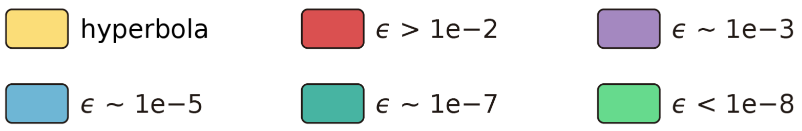


Figure 5.14 Color code employed to test the precision.

First, the solutions provided by `least_squares()` (and Method A) are studied, as they have shown a better overall performance:

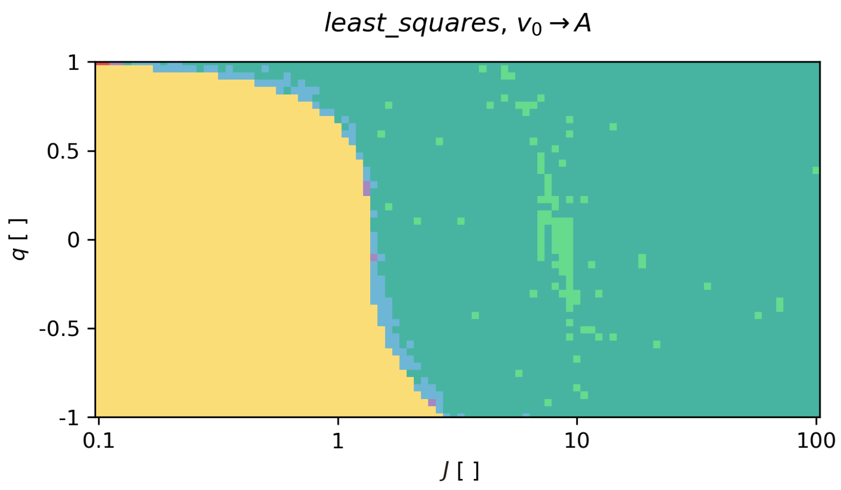


Figure 5.15 Precision provided by `least_squares()` on a logarithmic scale.

Looking at Fig. 5.15, this approach appears to provide reasonable precision ($\epsilon \sim 10^{-7}$) for a very wide range of q and J. However, when moving closer to the parabolic solution, a considerable loss of significant digits can be appreciated. Zooming into this region:

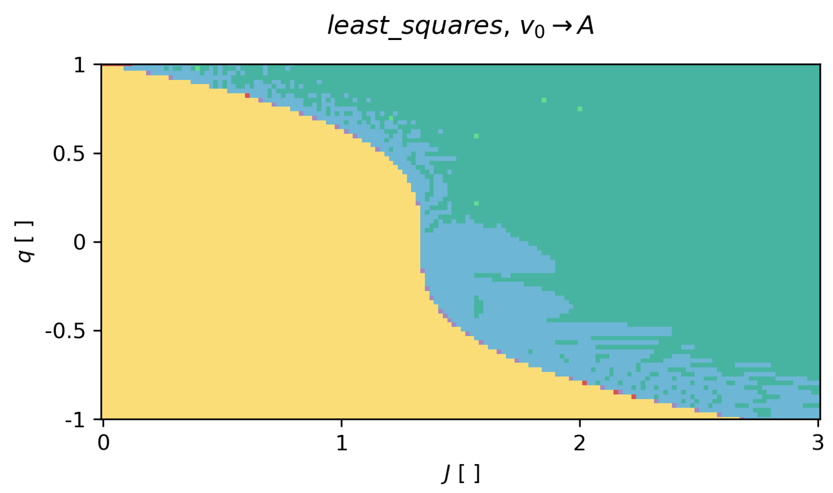


Figure 5.16 Precision provided by `least_squares()` on a linear scale.

We find that only for solutions very close to the parabolic case, `least_squares()` begins to fail (a few red dots can be observed), as (5.1) breaks down for $\alpha = 0$. Comparing this result with those generated by `fsolve()`:

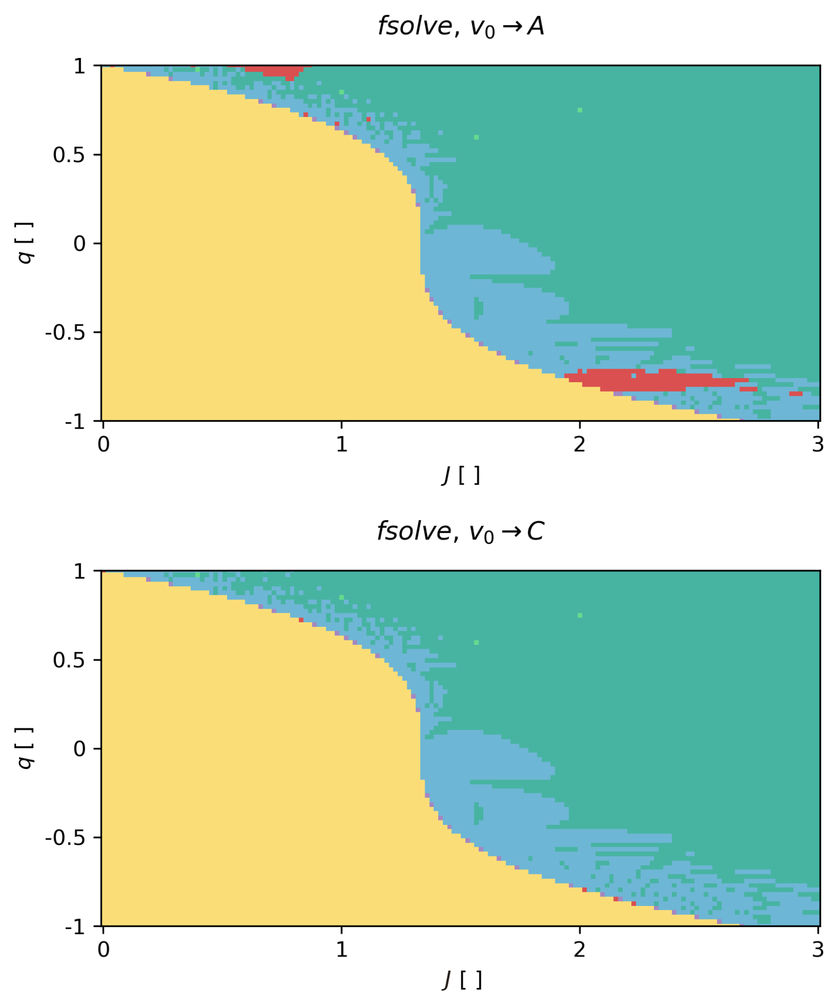


Figure 5.17 Precision provided by `fsolve()` on a linear scale.

Which proves that `least_squares()` is capable of solving Lambert's equations for elliptic motion with appropriate precision, similar to that provided by a dedicated root finding algorithm such as `fsolve()`. Moreover, `least_squares()` can be paired with Method A, which is very simple to code. In contrast, `fsolve()` must be used in conjunction with Method C to provide accurate results.

In Sect. 2.5, it was stated that for the cases in which the resulting orbit approached a parabola (that is, for α near 0), an additional piece of code (based on the series expression for J) may be provided to increase the precision and ensure convergence. However, the results shown in Fig. 5.17 seem to be accurate enough, except for really small values of α. To evaluate whether an additional piece of code is reasonable, let us analyze the precision provided by the series expression:

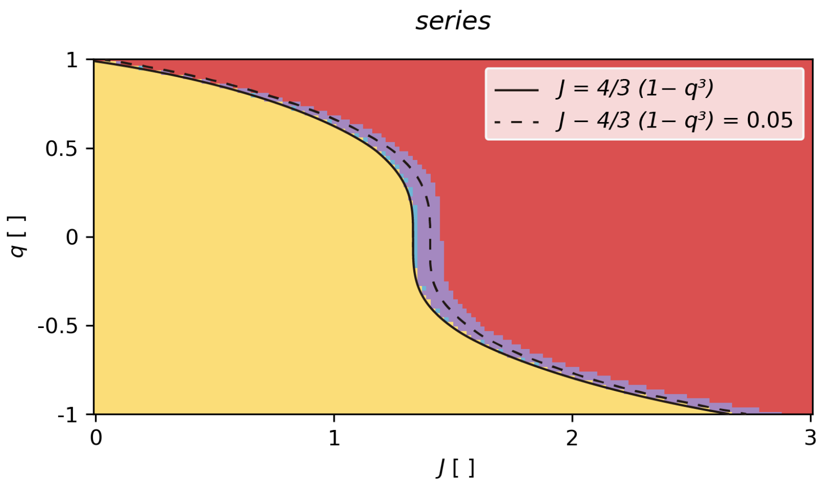


Figure 5.18 Precision provided by the series expression for J on a linear scale.

Recall that for the parabolic case, the following relationship holds:

$$J = \frac{4}{3}(1-q^3),$$

and for hyperbolic orbits, $J < 4/3(1-q^3)$. Thus, a quick and simple procedure to identify whether a transfer orbit is elliptic, parabolic or hyperbolic is to calculate $J - 4/3(1-q^3)$ and check if it's a positive value (ellipse), equal to zero (parabola) or a negative value (hyperbola).

In Fig. 5.18, the series expression is apparently providing results with not as much accuracy as we would expect (only reaching $\epsilon \sim 10^{-5}$, represented by the color blue, when almost parabolic). However, this can very well be due to the graphic method also breaking down and/or loosing precision for values of α near 0. To ensure an acceptable precision of `least_squares()`, a value lesser than $J - 4/3(1-q^3) = 0.05$ will be considered to imply a transfer orbit too close to a parabola and therefore a result with questionable validity (which would be discarded).

Given that the final objective of this laboratory class is the analysis and **optimization** of interplanetary missions, hyperbolic and parabolic solutions will usually be neglected. This is because they constitute higher energy transfer orbits, which are generally not the optimal solution. Moreover, that $J - 4/3(1-q^3) < 0.05$ is quite uncommon, as very short times of flight are required.

Therefore, a decision not to include the series expression has been made. Besides not really being necessary for the optimization of interplanetary missions, it would add some further complexity to the already quite complex laboratory class.

5.4 Part 3: Analysis and optimization of interplanetary missions

Finally, here we use Python to solve Lambert's problem and go to Mars. We see how our numbers match those produced by NASA's engineers!

In this third and final part, all the contents presented previously are put into practice. First, an example regarding the visualization of the Mars Science Laboratory mission is introduced. Launched by NASA on November 26, 2011, it successfully landed Curiosity, a Mars rover, on August 6, 2012. To plot its transfer orbit, the algorithm discussed in the previous section, which solves Lambert's problem for elliptic motion and the single-revolution case, needs to be programmed:

```python
'''Lambert solver (only valid for m = 0 and elliptic motion)'''
def Lambert(q, J):
    def F(v): # where v is an array containing alpha (a) and beta (b)
        a, b = v # a = v[0] and b = v[1]
        F1 = J*(np.sin(a/2))**3 - (a - b - np.sin(a) + np.sin(b))
        F2 = np.sin(b/2) - q*np.sin(a/2)
        return [F1, F2]
    alpha_beta = least_squares(F, [np.pi, 0], bounds=([1e-5,-np.pi],
    ↪  [2*np.pi, np.pi])).x
    return alpha_beta
```

The function `least_squares(fun, x0, bounds)` takes in the following parameters:

- `fun`: Function(s) to be solved.
- `x0`: Initial guess on independent variables.
- `bounds`: In the format `([a_start, b_start], [a_end, b_end])`

and outputs a *result* structure with multiple fields. To access the actual solution, `.x` is appended to the call.

Then, the starting and arrival positions are obtained at t_1 and t_2 respectively, after which the parameters q and J are obtained through the function `params()`. Calling `Lambert()`, a solution α, β to this particular problem is found. This solution is then used to obtain the orbital elements of the transfer orbit at t_1 through `elements()`. Further details on the auxiliary functions `params()` and `elements()` can be found in Appendix B.

5.4.1 Example 4: Analyzing the MSL mission - Results

In this section, the results and conclusions which are to be obtained by the students through *Example 4* are presented.

To plot the planet's orbits, a pair of functions created in *Example 2*, `PlotOrbit2D()` and `PlotOrbit3D()`, can been used. To plot the transfer orbit, `PlotSol2D()` and `PlotSol3D()` are to be called instead. These are included within `astrofun.py`, and are very similar to the previous pair but with slight variations (now the ephemerides are not used, and instead the orbital elements are directly passed as an input argument).

Both a three-dimensional graph and its two-dimensional projection are to be generated by the student:

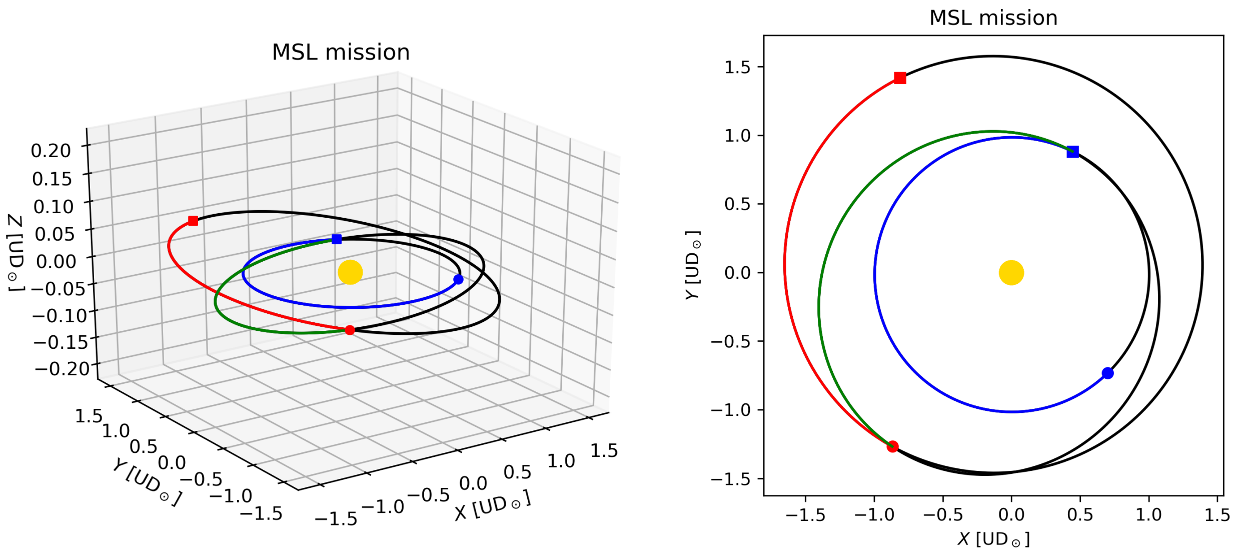


Figure 5.19 Visualization of the Mars Science Laboratory mission.

The solution appears to be very similar to a Hohmann transfer orbit. This is no coincidence; as stated in Sect. 3.5, when considering the two-dimensional problem, the Hohmann transfer orbit can be proven to be the optimal solution. In three dimensions, the starting and arrival orbits are no longer co-planar and therefore slight variations occur. Furthermore, neither the Earth or Mars' orbits are circular, nor their lines of apsides are aligned.

The engineers at NASA try to choose launch and arrival dates which provide the optimal solution, and real world transfer orbits may traverse slightly more, or slightly less, than 180° around the Sun. Sure enough, the MSL mission describes what's called a *Type 1* ($\Delta\theta < 180°$) Hohmann transfer orbit.

At this stage, a brief review of the theoretical background on the optimization of interplanetary missions is made. Here, the concept of characteristic energy (denoted C_3) and pork-chop plots are introduced. Then, a final example is presented to the students, regarding the optimization of NASA's Mars Reconnaissance Orbiter mission.

5.4.2 Example 5: Optimizing NASA's MRO mission - Results

The Mars Reconnaissance Orbiter (MRO) is a spacecraft designed to study the geology and climate of Mars, provide reconnaissance of future landing sites, and relay data from surface missions back to Earth. It was launched in the year 2005 and will continue to operate through the late 2020s, far beyond its intended design life.

Through this example, students obtain by themselves a pork-chop plot similar to that represented in Fig. 5.20 (already reviewed in Sect. 3.5), which shows the 2005 Mars launch opportunity ($C_3|_{\text{launch}}$). Then, an optimal launch date is calculated and some additional insight offered.

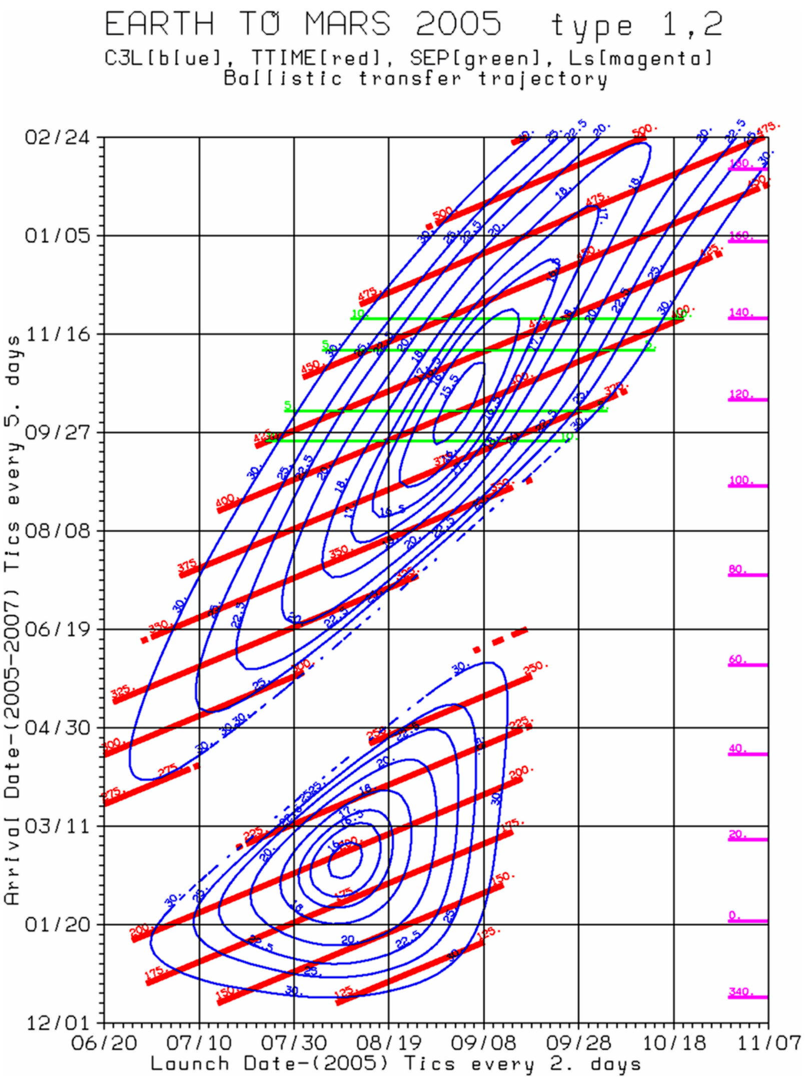


Figure 5.20 Representative pork-chop plot for the 2005 Mars launch opportunity. A given blue contour represents a solution with a constant C_3. [11]

Although this pork-chop plot does not provide the true optimal dates, which would be obtained by plotting $C_3|_{\text{total}}$, it is still of great interest. To match a launcher with a mission, $C_3|_{\text{launcher}}$ must be greater than $C_3|_{\text{launch}}$. In this particular case, $C_3 = 30\ \text{km}^2/\text{s}^2$ has been selected as an upper limit, given by the maximum C_3 that can be provided by the desired launcher.

To generate a contour plot, a grid in Julian days must be created. Computing the launch and arrival C_3 at every every pair of dates involves calling a few functions every time, and is a very time-consuming process. For instance, creating a 100 by 100 grid implies a total of 10000 discretization points, which roughly translates into a full minute of computing time. For the purposes of the laboratory class, however, this proves to be enough resolution.

Once the scalar fields of $C_3|_{\text{launch}}$ and $C_3|_{\text{arrival}}$ have been calculated, they can be plotted separately or added together to generate the pork-chop plot of $C_3|_{\text{total}}$. First, $C_3|_{\text{launch}}$ is plotted to compare the result with Fig. 5.21:

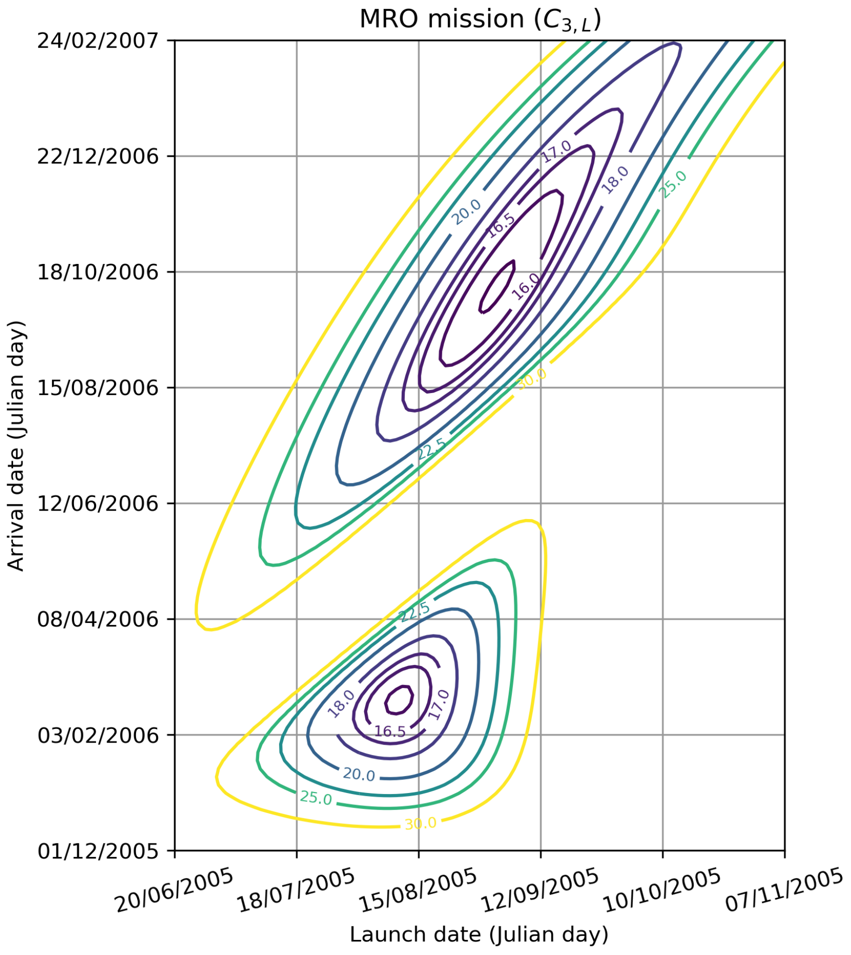


Figure 5.21 Self-made representative pork-chop plot for the 2005 Mars launch opportunity.

The resulting pork-chop plot is clearly identical to the one generated by NASA. In a way, this proves the validity of the algorithms and procedures which have been created for this project and are being applied here.

A three-dimensional representation of the scalar field of $C_3|_{\text{launch}}$ (or $C_3|_{\text{total}}$) can also be generated through a surface plot. In Fig. 5.22, for clarity, $C_3|_{\text{launch}}$ has been clipped to a maximum value of 300 km^2/s^2 (originally, at some points it can reach values of around 3000 km^2/s^2, before turning into parabolic or hyperbolic solutions). Additionally, the surface plot has been displayed along its top view (with the same orientation as the pork-chop plot in Fig. 5.20).

It is interesting to observe the overall shape of the surface, and perhaps its most striking feature consists on the narrow region where the characteristic energy takes extremely high values. This is due to $\Delta\theta$ becoming $\sim 180°$, which implies, in general, a polar heliocentric orbit (that requires a rather large C_3) as $\mathbf{r}_1$ and $\mathbf{r}_2$ lie on different planes.

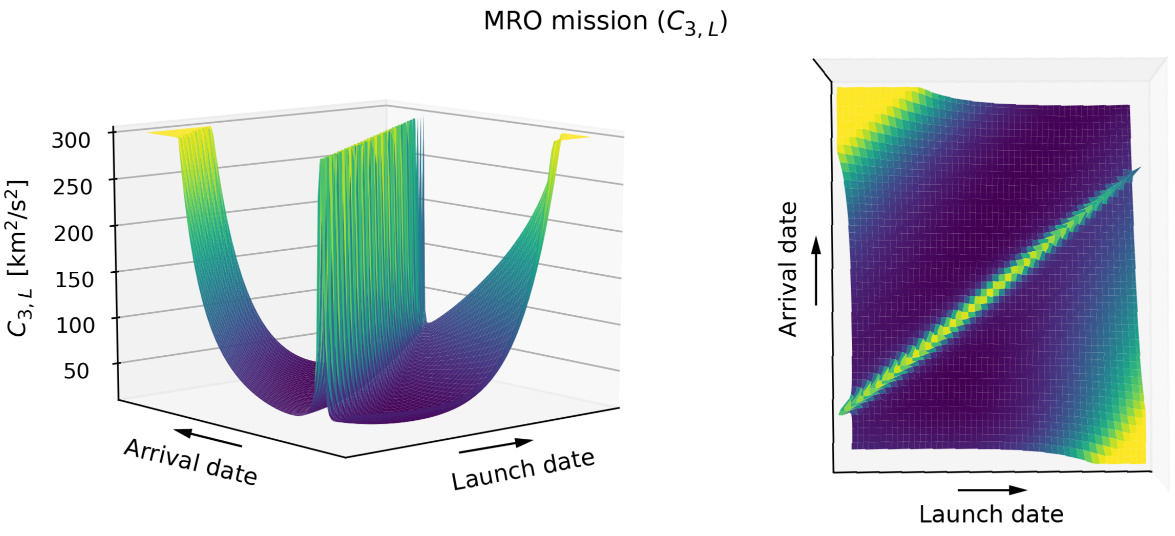


Figure 5.22 Three-dimensional representation of $C_3|_{\text{launch}}$.

To calculate the true optimal date, students are prompted to generate a pork-chop plot of $C_3|_{\text{total}}$:

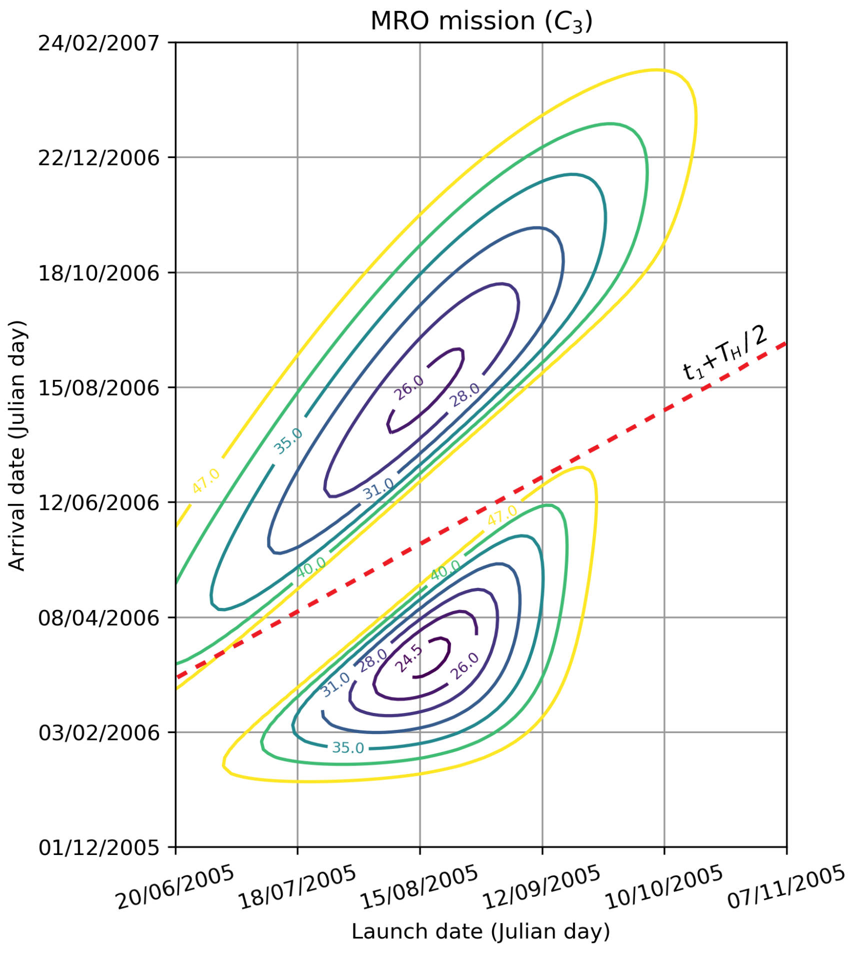


Figure 5.23 Pork-chop plot for the 2005 Earth-to-Mars total C_3 required.

This new contour plot turns out to be very similar to the previous one. The optimal solution can be found applying `np.min()` to the total characteristic energy's scalar field (which in the Laboratory class will be stored within a matrix named `C3`). However, this approach does not provide a way to differentiate between Type 1 and Type 2 solutions, or compute both minimums separately. Therefore, a perhaps more *rudimentary* procedure needs to be implemented. On one hand, a pair of for-loops can be created to access every element from the matrix `C3`, which contains the values of the total characteristic energy.

On the other hand, recall that Type 1 and Type 2 orbits resemble a Hohmann transfer orbit ($\Delta\theta = 180°$), the former being slightly shorter ($\Delta\theta < 180°$) and the latter slightly longer ($\Delta\theta > 180°$). To differentiate between both solutions, a simple rule can be developed. Consider the hypothetical period of a Hohmann transfer orbit solution to the 2D problem:

$$a_H = \frac{a_{\text{Earth}} + a_{\text{Mars}}}{2}, \qquad T_H = 2\pi\sqrt{\frac{a_H^3}{\mu_\odot}}.$$

Then, a transfer orbit will be of Type 1 if its time of flight is less than $T_H/2$, or of Type 2 otherwise. The line $t_2 = t_1 + T_H/2$ has been plotted in Fig. 5.23. Even though it does not overlap with the $\Delta\theta \sim 180°$ region mentioned earlier (whose *expression* cannot be easily obtained for a generic mission), this method proves to be quite reliable, and separates both minimums at any given range of dates.

Through the pair of for-loops (which can be found in the laboratory report), besides obtaining the C_3 of each solution, the launch and arrival dates can also be conveniently stored. The optimal launch and arrival dates turn out to be August 17, 2005 and March 15, 2006. That is, a short time of flight of about seven months, which implies a Type 1 orbit.

In reality, the MRO was launched on August 12, 2005 and reached Mars on March 10, 2006. Although not identical, the optimal dates which have been obtained are very close to the actual launch and arrival dates (only 5 days off). The discrepancies can very well be due to other factors such as bad weather at the optimal launch date or restraints of other nature, and surely due to the multiple simplifications which have been made to tackle the problem (including the concept of spheres of influence[1] and the patched conics approximation).

However, this outcome proves that most of these simplifications are indeed appropriate, and provide surprisingly accurate results. In a real-life scenario, more complex multiple-body simulations would be carried out to validate and refine the final solution.

5.4.3 Additional insight on the Mars launch opportunity windows

In the previous example, only the 2005 Mars launch opportunity has been analyzed. However, in the context of future interplanetary missions, it is interesting to consider whether spacecraft can be launched at any desired year.

As it has been studied, the feasibility of either a Type 1 or Type 2 transfer orbit depends primarily on the relative position between Earth and Mars —how long does it take for these planets to be in a similar relative position? The answer to this question corresponds with the *synodic period*.

[1] It must be noted that the time spent by the spacecraft in escaping Earth's million-kilometer SOI, which can be substantial, has not been taken into account.

If the orbital periods of two bodies around a third one (e.g. the Sun) are called T_1 and T_2, so that $T_1 < T_2$, their synodic period T_s is given by:

$$\frac{1}{T_s} = \frac{1}{T_1} - \frac{1}{T_2}$$

The synodic period of Mars, relative to Earth, turns out to be approximately of 779.9 days or 2.135 years[1]. Therefore, a launch opportunity should be expected roughly every two years. Since 2005, the number of missions to Mars carried out every year have been represented in Fig. 5.24.

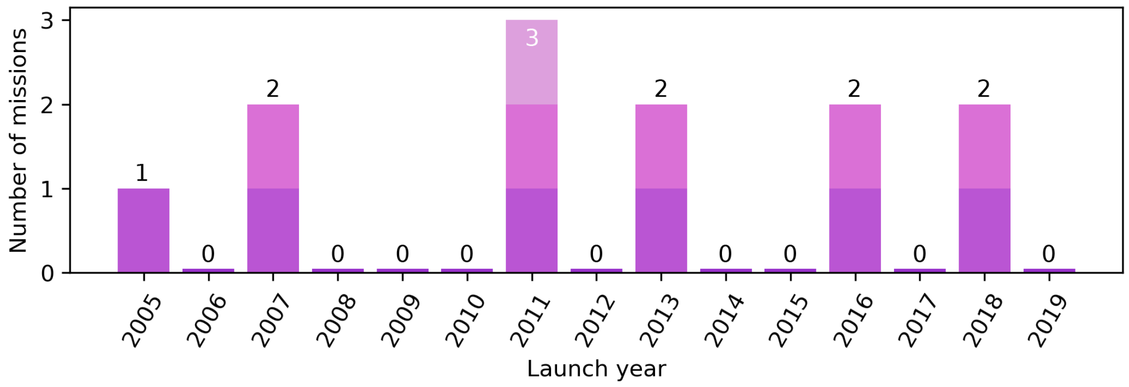


Figure 5.24 Number of missions to Mars carried out every year since 2005.

Just as it has been predicted, a trend can be clearly appreciated —spacecraft are launched approximately every two years.

Furthermore, the characteristic energies required for future missions can be calculated, in order to choose an optimal launch year. For instance, during the decade of the 2020s, the optimal (minimum) $C_3|_{\text{total}}$ can be computed for both Type 1 and Type 2 solutions:

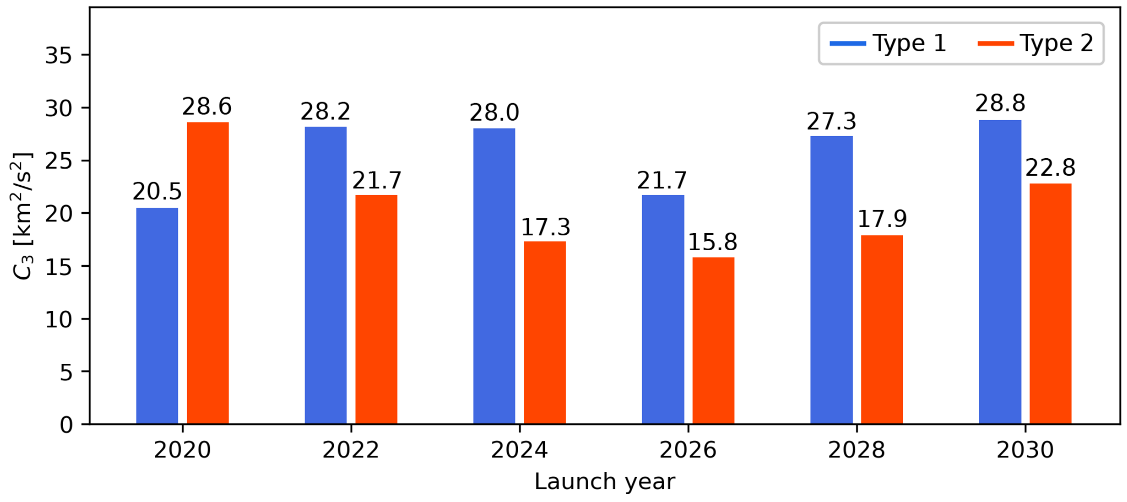


Figure 5.25 Optimal $C_3|_{\text{total}}$ of Type 1 and Type 2 transfer orbits during the 2020s.

[1] In practice, a more accurate synodic period (obtained empirically) turns out to be of 775.5 days.

Observing Fig. 5.25, the best year at first sight seems to be 2026, although many more factors need to be taken into account when designing a real mission. Additionally, notice how the overall optimal $C_3|_{\text{total}}$ can be achieved through transfer orbits of either type.

6 Conclusions and future work

6.1 Conclusions

The concepts developed along this project have been those strictly necessary to approach the various subjects treated and fulfill our final purpose, which is to design a two-hour laboratory class for the students enrolled in the *Orbital Mechanics and Space Vehicles* course, imparted by Rafael Vázquez Valenzuela at the University of Seville. In this regard, a comprehensive analysis of Lambert's problem and the basics of Python programming have been covered. Additionally, an appropriate laboratory class, along its accompanying laboratory report (which can be found in Appendix A), have been developed. In the following lines, some further thoughts are presented.

Clearly, the main limitation in the design of the laboratory class has been its two-hour time restriction. This intended duration has significantly shaped the amount and complexity of the contents introduced, and has led to a substantial simplification of Lambert's problem (which has only be solved for elliptic motion). However, this approach has proven to be well suited for basic analysis and design of interplanetary missions.

Regarding the introduction to Python, an attempt has been made to cover the basic knowledge required to code the various algorithms present throughout the laboratory class. However, students who are interested in learning more about the language are advised to read Chapter 4 from this document, and the fantastic guide by H. Fangohr [12]. Intermediate knowledge of MATLAB has been assumed, as it is taught during the first years of the Bachelor in Aerospace Engineering. If students had a previous background in Python, the laboratory class could have been approached in a different manner. Perhaps a greater focus would have been allowed on the practical application of Lambert's problem, in the context of interplanetary missions. More on this idea is addressed in the next section, dealing with future work.

Overall, the results has been very satisfactory. The first part of the laboratory class serves as a great introductory chapter for students lacking any previous knowledge in Python programming. Moreover, regarding the Earth to Mars missions analyzed within the laboratory class, the obtained solutions closely resemble those obtained by the NASA. In a way, this proves the validity of the algorithms and procedures which have been developed for this project. To conclude this project, the laboratory report has been provided to a group of students, obtaining very positive feedback.

6.2 Future work

The main subject of this project, which is Lambert's problem, has been approached in a rigorous manner. However, even though an universal algorithm valid for elliptic, hyperbolic and parabolic transfers is addressed in Sect. 2.4, it has not been implemented in Python. Only a simpler, more limited version for the elliptic case has been developed. For future work, programming the universal algorithm (and perhaps exploring more recent and efficient approaches than the one addressed in this project, adapted from Lancaster *et al.* [1]) could be interesting, as hyperbolic and parabolic trajectories tend to show up in the analysis of more complex interplanetary missions.

Furthermore, the precision of the resulting algorithm, although appropriate for the purposes of the laboratory class, is low compared to other existing algorithms. An effort to increase it should be made, and possibly an extension of the analysis presented in Sect. 5.3.2, with more robust techniques for error detection and increased testing, would be advisable.

Throughout the laboratory class, only direct transfer orbits have been reviewed. However, in reality more elaborate techniques are used to design and optimize interplanetary missions. For instance, multiple-revolution orbits are frequently utilized in combination with gravity assisted and deep space maneuvers (GAM and DSM respectively). On one hand, a more robust Lambert solver to deal with the multi-revolution case should be developed. On the other hand, an additional laboratory class (*a Part 2*, so to say) could be introduced to deal with these concepts, emphasizing their practical application and providing the students with additional tools to program more elaborate code.

Moreover, every year multiple students are involved in projects (*TFGs*) regarding orbital mechanics, which focus on the design and optimization of interplanetary missions. The work developed throughout this document (particularly the numerous auxiliary algorithms included in Appendix B), could be made accessible to them so they can invest their efforts in the actual designing and optimizing of the mission, instead of trying to code them themselves (which takes a surprisingly large amount of time).

Lastly, it should be noted that the laboratory report will surely be subject to numerous changes in the future. An effort to update the latter examples to more recent events, and the commands and tools used in the Python tutorial to the latest version of the language should be made. Additionally, in the event of Python being included within the Bachelor in Aerospace Engineering's study plan, the laboratory report would most definitely need to be restructured.

Bibliography

[1] E. R. Lancaster and R. C. Blanchard. *A Unified Form of Lambert's Theorem.* National Aeronautics and Space Administration, Washington, 1969.

[2] R. H. Gooding. *A Procedure for the Solution of Lambert's Orbital Boundary-Value Problem.* Celestial Mechanics and Dynamical Astronomy 48, 145–165, Hampshire, 1990.

[3] C. F. Gauss. *Theory of the Motion of the Heavenly Bodies Moving about the Sun in Conic Sections.* Dover Publications, New York, 1963.

[4] Rafael Vázquez Valenzuela. *Orbital Mechanics and Space Vehicles, Class Notes (ETSI Seville).* Available at: http://aero.us.es/astro/desc.html (Accessed: March 2020).

[5] R. H. Battin. *An Introduction to the Mathematics and Methods of Astrodynamics.* AAIA, New York, 1999.

[6] H. D. Curtis. *Orbital Mechanics for Engineering Students.* Elsevier Butterworth-Heinemann, Oxford, 2005.

[7] I. Newton (translated by A. Motte). *The Mathematical Principles of Natural Philosophy.* University of Oxford, Oxford, 1729.

[8] J. E. Prussing. *Geometrical Interpretation of the Angles α and β in Lamberts Problem.* Journal of Guidance, Control and Dynamics, Vol.2, No. 5, 442–443, Illinois, 1979.

[9] G. S. Gedeon. *Lambertian Mechanics, Proceedings of the XIIth International Astronautical Congress.* Academic Press Inc., New York, 1963.

[10] E. M. Standish. *Keplerian Elements for Approximate Positions of the Major Planets.* NASA Jet Propulsion Laboratory, Caltech, 2006.

[11] *"Porkchop" is the First Menu Item on a Trip to Mars.* NASA Jet Propulsion Laboratory, Available at: http://mars.jpl.nasa.gov/spotlight/porkchopAll.html (Accessed: March 2020).

[12] H. Fangohr. *Python for Computational Science and Engineering.* University of Southampton, Southampton, 2015.

[13] *Python 3.7 Documentation.* Python Software Foundation, Available at: https://docs.python.org/3.7/ (Accessed: May 2020).

[14] *NumPy 1.18 Documentation.* The SciPy community, Available at: https://numpy.org/doc/stable/ (Accessed: May 2020).

[15] *NumPy for Matlab users.* The SciPy community, Available at: https://numpy.org/doc/stable/user/numpy-for-matlab-users.html (Accessed: June 2020).

Appendix A
Laboratory report

ORBITAL MECHANICS AND SPACECRAFT

Laboratory Class

Programming with Python. Lambert's Problem.
Analysis and Optimization of Interplanetary Missions.

To perform the analysis and optimization of an interplanetary mission (or generally, of any transfer between two generic orbits) the need to solve *Lambert's Problem* arises. Sometimes referred to as the orbital boundary problem, it is concerned with the determination of an orbit from two position vectors and the time of flight. The goal of this laboratory class is to provide guidelines for coding in Python an algorithm that allows Lambert's problem to be solved numerically.

1 Programming with Python

In this section, you will learn the basics of Python programming, which is widely used in the space industry. The language is similar to MATLAB in many aspects, so you will be doing complex tasks in just a few minutes!

Python is a powerful programming language which has many different and interesting applications. Perhaps one of its most important advantages is the great number of existing libraries available —for scientific computation, it is crucial to make use of numerical libraries such as NumPy, SciPy and the plotting package Matplotlib. It was designed for readability, and has some similarities to the English language with influence from mathematics. It uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses. Additionally, **Python relies on indentation to define the scope of loops or functions**.

All major editors that are used for programming (such as Atom, Vim, Sublime Text, etc.) provide Python modes. For beginners, however, working within an Integrated Development Environment (IDE) may result more intuitive. IDEs provide an useful range of tools, such as a display showing the variables created by the user or allowing to execute the code in debug mode (step by step). Furthermore, this provides a smoother learning curve for those familiarized with MATLAB's IDE. For this laboratory class, Spyder seems a sensible choice as it provides a similar graphic user interface and is easy to pick up. Additionally, Spyder is included by default in the Anaconda distribution (which is installed in most computers of the University of Seville's ETSI).

Anaconda is a free and open-source Python distribution for scientific computing, that aims to simplify package management and deployment. It also incorporates some of the most popular Python libraries (or packages), including those which will be used throughout this laboratory class. Anaconda can be downloaded from: https://www.anaconda.com/products/individual.

There are two versions of the Python language being used nowadays: Python 2 and Python 3. The changes in Python 3 were introduced to address shortcomings in the design of the language that were identified since Python's inception. Therefore, the contents presented throughout this laboratory class are programmed using the latter version.

1.1 Getting started

By default, Spyder's main window consists of three main modules: editor, variable explorer and (IPython) console. In the editor, selected parts of the code can be executed and the debug mode is available. The variable explorer, as its name implies, shows the variables created by the user, indicating their type, size and value. On the other hand, the interactive console provides programmers with a quick way to execute commands and try out or test code without creating a file.

First, create a new file by clicking the *New file* icon or pressing `ctrl+N`. Python is an interpreted language —it executes instructions directly, without previously compiling the code into machine-language instructions. The interpreter executes every line in sequential order, from top to bottom. Therefore, the first lines of a Python script (a file with the extension "`.py`" containing code written in Python) must contain the modules and/or packages (a collection of modules) that are intended to be used throughout the file. Importing a module or package is straightforward:

```python
import numpy
```

The names within the `numpy` package must be accessed through the name `numpy`. For example: `numpy.sin`. However, the name by which the module or package is known locally can be different from its "official" name. For instance, writing

```python
import numpy as np
```

changes the package name to something more manageable. Furthermore, writing

```python
from numpy import sin
```

will only import the `sin` function from the `numpy` package (which can then be called directly, without writing `numpy.`). It is possible to import more than one function:

```python
from numpy import sin, cos
```

A newly created file contains a *docstring* indicating the author and current date, in addition to the command `# -*- coding: utf-8 -*-`. This default template is only optional, and therefore everything can be deleted.

1.2 Example 1 - Orbital position as a function of time

Programming a simple algorithm which computes the orbital position as a function of the time relative to periapsis passage (the time at which an orbiting body moves through the periapsis of the orbit) provides a good starting point for getting familiarized with Python.

To develop a numerical algorithm, the NumPy package (https://numpy.org/) is required. It provides access to a data structure called *array* (similar to MATLAB's) which allow efficient vector and matrix operations. It also provides fundamental mathematical functions (`sin()`, `cos()`, `log()`, etc.), physical constants (`pi`, `e`, etc.) and a number of linear algebra operations.

In addition to the NumPy package, a root finder is required to solve Kepler's equation:

```python
import numpy as np # np is the standard abbreviation for numpy
from scipy.optimize import fsolve
```

Then, the initial data can be introduced, including those orbital elements which define the orbit's shape (the eccentricity, e) and size (the parameter p in this example):

```python
''' Data (in the Sun's canonical units)'''
Delta_t_days = 100.0 # [days] Time relative to periapsis passage
Delta_t = Delta_t_days/58.1324 # [UT] (1 UT = 58.1324 days)
e = 0.0167 # Earth's orbit eccentricity
p = 0.999 # [UD] Earth's orbit p parameter (a = 1 UD)
```

Throughout this laboratory class, the Sun's canonical units will be used:

$$1\ \mathrm{UD}_\odot = 1\ \mathrm{AU}, \qquad 1\ \mathrm{UV}_\odot = 29.7847\ \mathrm{km/s}, \qquad 1\ \mathrm{UT}_\odot = 58.1324\ \mathrm{days}$$

Basic operations such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`) and exponentiation (`**` and not `^` as in MATLAB) work as expected. Parentheses can be used for grouping. To perform an element-wise operation of two arrays, there is no need to place a dot before the operator (as it would be done using MATLAB).

Single-line comments start with the hash `#` character, and they are automatically terminated by the end of line. However, when a string (enclosed by single `'` or double `"` quotation marks) is just written down, with no additional operations applied, and not assigned to a variable, it is basically ignored by the interpreter. Therefore, they can be employed as a sort of comment. This isn't that common, or semantically correct, but it is allowed. Triple quotes `'''` allow a string to span multiple lines, but throughout this practical class they will be used primarily to emphasize headings (as they allow regular single quotes `'` within the string).

Next, a function which computes the orbital position as a function of the time relative to periapsis passage (Δt) is defined:

```python
'''Function which computes the orbital position'''
def OrbitalPosition(Delta_t, p, e):
    mu = 1 # [UD*UV^2]

    if e<1: # Elliptic orbit
        a = p/(1-e**2) # [UD]
        M = Delta_t*np.sqrt(mu/a**3)
        E = fsolve(lambda E: E-e*np.sin(E)-M, M)
        theta = 2*np.arctan(np.sqrt((1+e)/(1-e))*np.tan(E/2))
        # np.arctan() outputs a value between -pi and pi

    elif e==1: # Parabolic orbit
        B = 3*Delta_t*np.sqrt(mu/p**3)
        z = (B+np.sqrt(B**2+1))**(1/3)
        theta = 2*np.arctan(z-1/z)

    else: # Hyperbolic orbit
        a = p/(1-e**2)
        N = Delta_t*np.sqrt(mu/(-a)**3)
        H = fsolve(lambda H: e*np.sinh(H)-H-N, N)
        theta = 2*np.arctan(np.sqrt((1+e)/(e-1))*np.tanh(H/2))

    return theta
```

The indentation after the declaration of the function is required and defines its scope. The same applies to for-loops and if-else statements. Additionally, there is no need to specify an `end` command to these blocks (while it is necessary in MATLAB); returning to the previous indentation level is enough.

For the elliptic case, `fsolve` returns the roots of Kepler's equation ($E - e \sin E - M = 0$) using M as a starting estimate. This equation has been defined through an anonymous function, also known as lambda function (e.g. `f = lambda x: x**x`). Anonymous functions provide similar functionality as MATLAB's function handles, defined through the @ symbol.

A function cannot be called before its declaration (the Python interpreter has to first come across the `def` line in order to recognize the function and allow it to be called). Functions can also take and/or return an arbitrary number of arguments. To conclude this example, the previous function is called to obtain the orbital position at the time Δt (relative to periapsis passage):

```python
'''Calling the function'''
theta_rad = OrbitalPosition(Delta_t, p, e) # [rad]
theta_deg = theta_rad*180/np.pi # [º]
print('theta =', theta_deg)
```

Once the code is completed, click the *Run file* icon (or press F5) to execute the script. The variables assigned outside the function appear at the variable explorer module (just at they would in MATLAB). These variables can be erased by clicking the *Remove all variables* (rubber-like) icon or writing `%reset -f` in the IPython console. Additionally, the `clear` command clears the console window, while `plt.close('all')` closes all open figures. These are analogous to MATLAB's `clear`, `clc` and `close all` respectively.

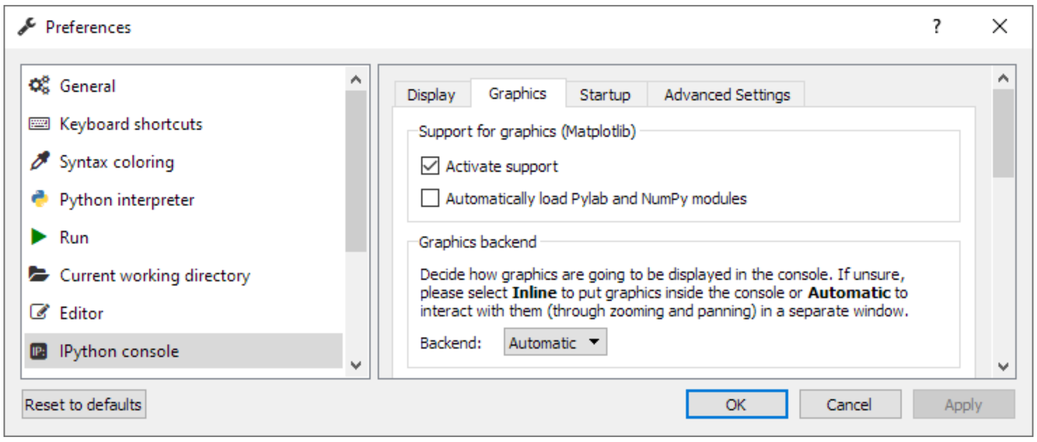
Note: By default, variables with all-uppercase names are not shown at the variable explorer. To fix this, click the top right *Options* (gear-shaped) icon and make sure the *Exclude all-uppercase references* option is not selected.

1.3 Example 2 - Plotting Mercury's orbit

In this second example another essential package is reviewed, Matplotlib (https://matplotlib.org/), which is a plotting library for Python and its numerical mathematics extension NumPy. It enables the generation of high quality line plots, scatter plots, 3D plots, etc., with just a few lines of code, and its module `matplotlib.pyplot` provides a MATLAB-like plotting framework.

To plot Mercury's orbit, its planetary ephemeris and an orbit propagator are required. The function `eph(planet_number, JD)`, included in the `astrofun.py` module, computes the orbital elements of a Major Planet's orbit at a given Julian day. To calculate the Julian day of a date expressed in DD/MM/YYYY format, the function `julian(D, M, Y)` can be used. Additionally, a function that transforms the orbital elements into position and velocity vectors (`posvel(elements)`) is provided.

Before getting started, go to `Tools > Preferences` (or press `Ctrl+Alt+Shift+P`) and then `IPython console > Graphics > Graphics backend`. Once there, the default option (`Inline`) needs to be changed to `Automatic`. This will generate figures in separate windows, allowing the user to interact with them. To apply this change, Spyder will need to be restarted (even though it doesn't ask to). Before doing so, the previous script must be saved so that no information is lost.



Support for graphics (Matplotlib)

☑ Activate support

☐ Automatically load Pylab and NumPy modules

Graphics backend

Decide how graphics are going to be displayed in the console. If unsure, please select **Inline** to put graphics inside the console or **Automatic** to interact with them (through zooming and panning) in a separate window.

Backend: Automatic

Figure 1: Change of settings.

Now lets proceed with the second example. First, a new file must be created. Then, delete the default template and include the following lines of code:

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from astrofun import julian, eph, posvel
```

Note: when importing functions from a file `a.py` to another file `b.py`, both need to be in the same directory.

It is smart to define a function which plots the orbit of a generic planet, given launch and arrival dates. This way, if later during the laboratory class other orbits need to be plotted, it will just be a matter of importing said function (just as it has been done with those from `astrofun.py` here). To begin, a function which plots a 2D projection of the orbit can be created:

```
def PlotOrbit2D(planet_num, JD1, JD2, color, ends=False):
    pts = 500 # Number of discretization points
    r_vec = np.zeros([pts, 3]) # Initializing the array
    i = 0
    for JD in np.linspace(JD1, JD2, pts):
        elements = eph(planet_num, JD)
        r, v = posvel(elements) # A multiple assignment
        r_vec[i,:] = r
        i+=1 # Increases i by 1 in every iteration
    plt.plot(r_vec[:,0], r_vec[:,1], color)
    plt.scatter(0, 0, s=180, c='gold') # s=size, c=color
    if ends: # if ends==True
        plt.plot(r_vec[0,0], r_vec[0,1], 's', c=color)
        plt.plot(r_vec[-1,0], r_vec[-1,1], 'o', c=color)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.xlabel(r'$X$ [UD$_\odot$]') # \odot is the Sun's symbol
    plt.ylabel(r'$Y$ [UD$_\odot$]')
```

A color string and whether to plot the planet at the launch (square marker) and arrival (circle marker) dates have been considered as input arguments. By default, the variable `ends` is set to `False` (doesn't plot markers).

Many commands included in NumPy and Matplotlib are very similar to MATLAB's. For instance, the functions `np.zeros(shape)` and `np.linspace(start, stop, num)` are identical to its MATLAB equivalents. However, there are some key differences between the two languages. It is important to note that Python uses zero-based indexing, that is, the first element has an index 0, the second has index 1, and so on. This is in contrast to MATLAB's one-based indexing, where the first element has index 1.

The use of indices in slicing also differs from MATLAB's approach. In MATLAB, stating `a(1:2)` returns the first and second elements from the vector `a`. However, stating `a[1:2]` in Python only returns the second element. The best way to remember how slices work in Python is to think of the indices as pointing between elements:

```
     indexing -->     0     1     2     3     4
                   +-----+-----+-----+-----+-----+
                   | 'H' | 'e' | 'l' | 'l' | 'o' |
                   +-----+-----+-----+-----+-----+
                   0     1     2     3     4     5    <-- slicing
```

In the previous function, a for-loop is required to obtain the planet's position at various instants of time between the dates `JD1` and `JD2`. Through the command `r_vec[i,:] = r`, the three coordinates of the radius vector are assigned to the corresponding row of `r_vec`. Note that Python (in contrast to Matlab) doesn't allow to address elements which have not been created previously. Therefore, `r_vec` needs to be initialized before entering the for-loop (in this case through the function `np.zeros()`).

The actual plotting is straight-forward. For 2D line plots and scatter plots, the functions `plt.plot()` and `plt.scatter()` are identical to its Matlab equivalents. The line style, marker symbol, and color can also be set the same way. For a 2D representation of the orbit, only the X (`r_vec[:,0]`) and Y (`r_vec[:,1]`) components of the radius vector are being plotted.

The command `plt.gca().set_aspect('equal', adjustable='box')` sets the scale of both axis to be equal to show the true shape of the orbit. Adding an `r` before a string allows LaTeX formatting in Matplotlib graphs (for titles, labels, etc.).

To plot 3D data, the sub-module `mplot3D` from the module `mpl_toolkits` needs to be imported. 3D plots are not supported within `pyplot`, and therefore some object-oriented programming (although quite simple) is required. 3D axes can be created through the command `plt.axes(projection='3d')`, which generates an *Axes3D* object. Then, plots can be added to the *Axes3D*. This slightly differs from Matlab's and `matplotlib.pyplot`'s state-driven interface.

A function which plots a 3D orbit can be similarly defined:

```
22 def PlotOrbit3D(planet_num, JD1, JD2, color, ends=False):
23     r_vec = np.zeros([pts, 3])
24     i, pts = 0, 500 # How multiple simultaneous assignments are done
25     for JD in np.linspace(JD1, JD2, pts):
26         r_vec[i,:] = posvel(eph(planet_num, JD))[0]
27         i+=1
28     ax = plt.gca()
29     ax.plot3D(r_vec[:,0], r_vec[:,1], r_vec[:,2], color)
30     ax.scatter3D(0, 0, 0, s=150, c='gold')
31     if ends:
32         ax.scatter3D(r_vec[0,0], r_vec[0,1], r_vec[0,2], c=color,
           ↪  marker='s')
33         ax.scatter3D(r_vec[-1,0], r_vec[-1,1], r_vec[-1,2], c=color)
34     lim = 1.2*eph(planet_num, JD)[0]
35     ax.auto_scale_xyz([-lim,lim], [-lim,lim], [-0.2,0.2])
36     ax.set_xlabel(r'$X$ [UD$_\odot$]')
37     ax.set_ylabel(r'$Y$ [UD$_\odot$]')
38     ax.set_zlabel(r'$Z$ [UD$_\odot$]')
```

The arrow symbol in line 32 only indicates that the line has been continued below so it can fit in the box. In your script, just write the whole command in one single line.

In this case, the plots are being added to an *Axes3D* object called `ax`. Through the command `plt.gca()` (*get current axes*), `ax` becomes the current axes (which will have to be created before calling the function). The command `ax.auto_scale_xyz()` scales the axes according to the limits introduced. To encompass the complete orbit, the limits for the horizontal plane are set to ±1.2 times its semi-major axis (`a = eph(planet_num, JD)[0]`).

Lastly, a pair of dates are selected and the orbit can be plotted in 2D and 3D. Lets plot a 2D complete orbit first:

```python
if __name__ == '__main__': # only executed if this file is run

    JD1 = julian(1,1,2021)
    T = 2*np.pi*np.sqrt(eph(1, JD1)[0]**3/1) # [UT] Mercury's period
    T_days = T*58.1324 # [days]
    JD2 = JD1 + T_days

    '''Complete orbit'''
    plt.figure() # This command creates a new figure
    PlotOrbit2D(1, JD1, JD2, 'k', True) # plots the complete orbit
```

The special variable `__name__` is only equal to `'__main__'` when the actual file is being run directly. The functions `PlotOrbit2D()` and `PlotOrbit3D()` will be required later, and to import them, Python internally runs the entire file where they are located. To avoid executing these lines of code (which are only relevant to this example), we include them inside an if-statement which is only entered if `__name__ == '__main__'`.

Before clicking *Run file*, save the file at the same directory were `astrofun.py` can be found. Then, run the script and a new window should appear. Mimicking MATLAB 's interface, Matplotlib also provides an interactive interface which allows to zoom, pan or edit the plot's basic attributes.

Now, analyze the orbit. **What can be observed if the ends (the markers) are zoomed in enough?** *Hint:* A linear propagator is being used, and the rates of change of the orbital elements due to perturbations are being taken into account. Zoom in a few times using the magnifying glass icon to appreciate this effect.

Next, lets plot the arc traveled by Mercury in a 30-day month. To show multiple plots on different windows, new figures must be created (by default, the equivalent to MATLAB's `hold` option is set to `on`). Additionally, for the 3D plot an *Axes3D* object named `ax` must be created:

```python
    JD3 = JD1 + 30 # a 30-day month from JD1

    '''2D Figure'''
    plt.figure()
    PlotOrbit2D(1, JD1, JD2, 'k')
    PlotOrbit2D(1, JD1, JD3, 'r', True)
    # The starting and ending points of the traveled arc are being
    ↪  plotted
    '''3D Figure'''
    plt.figure()
    ax = plt.axes(projection='3d')
    PlotOrbit3D(1, JD1, JD2, 'k')
    PlotOrbit3D(1, JD1, JD3, 'r', True)
```

To generate the plots, click *Run file* once again.

Further information about any function can be obtained by typing `help(func)` into the console, searching the packages' documentation:

- NumPy documentation: https://numpy.org/doc/stable/
- Matplotlib documentation: https://matplotlib.org/3.2.1/contents.html

or else in Python's documentation (https://docs.python.org/3.7/). Additionally, **in the last pages of this laboratory report, a comparison between MATLAB and Python's syntax of the most frequent commands and expressions can be found**.

2 Lambert's problem

Solving Lambert's problem is crucial to design realistic interplanetary transfers and maneuvers. We don't do it by hand because it requires solving two nonlinear equations with two unknowns. Here we learn how to make our computer do it for us; then you can relax and start designing your first missions!

Consider an orbiting body located at distances r_1 and r_2 from the center of attraction at times t_1 and t_2 respectively. Let c be the distance and $\Delta\theta$ the transfer angle between the positions of the orbiting body at the two times, where $0 \leq \Delta\theta \leq 2\pi$.

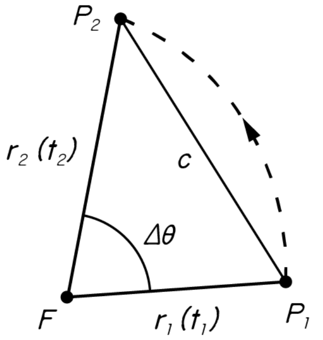


Figure 2: A diagram of the two-body orbital boundary problem.

Lambert's problem is that of finding the semi-major axis (or some related quantity) of the orbit, given t_1, r_1, t_2, r_2 and $\Delta\theta$. When Lambert's problem has been solved, other quantities associated with the orbit are easily found.

2.1 The classical form of Lambert's equations

With origin at the center of attraction, we have, for elliptic motion,

$$r_1 = a(1 - e\cos E_1), \tag{1}$$

$$r_2 = a(1 - e\cos E_2), \tag{2}$$

$$n\Delta t_1 = E_1 - e\sin E_1, \tag{3}$$

$$n\Delta t_2 = E_2 - e\sin E_2, \tag{4}$$

where n is the mean motion and E the eccentric anomaly. In the perifocal frame of reference, $\mathbf{r}_1$ and $\mathbf{r}_2$ are expressed as

$$\begin{aligned} \mathbf{r}_1 &= a(\cos E_1 - e)\ \mathbf{e}_x + \ a\sqrt{1-e^2}\sin E_1\ \mathbf{e}_y, \\ \mathbf{r}_2 &= a(\cos E_2 - e)\ \mathbf{e}_x + \ a\sqrt{1-e^2}\sin E_2\ \mathbf{e}_y. \end{aligned} \tag{5}$$

Using the law of cosines, we can express c in terms of r_1, r_2 and $\Delta\theta$:

$$c^2 = r_1^2 + r_2^2 - 2r_1r_2\cos\Delta\theta, \tag{6}$$

which is to say

$$c^2 = r_1^2 + r_2^2 - 2\mathbf{r}_1 \cdot \mathbf{r}_2. \tag{7}$$

Substituting (5) in (7), we have

$$c^2 = a^2(\cos E_2 - \cos E_1)^2 + a^2(1-e^2)(\sin E_2 - \sin E_1)^2,$$

$$= 4a^2\left(1 - e^2\cos^2\frac{E_1+E_2}{2}\right)\sin^2\frac{E_2-E_1}{2}. \tag{8}$$

Adding (1) and (2),

$$r_1 + r_2 = 2a\left(1 - e\cos\frac{E_1+E_2}{2}\cos\frac{E_2-E_1}{2}\right). \tag{9}$$

Subtracting (3) from (4),

$$n(t_2 - t_1) = (E_2 - E_1) - 2e\cos\frac{E_1+E_2}{2}\sin\frac{E_2-E_1}{2}. \tag{10}$$

(8), (9) and (10) determine the three unknowns a, $(E_2 - E_1)$ and $e\cos[(E_1+E_2)/2]$. To simplify these equations it is customary to define two new parameters, α and β. Let

$$\cos\frac{\alpha+\beta}{2} = e\cos\frac{E_1+E_2}{2}, \qquad 0 \le \alpha+\beta < 2\pi, \tag{11}$$

and

$$\alpha - \beta = E_2 - E_1 - 2m\pi, \qquad 0 \le \alpha - \beta < 2\pi, \tag{12}$$

where m is the number of complete revolutions made by the orbiting body between times t_1 and t_2. (8), (9) and (10) then become

$$\frac{c}{2a} = \sin\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}, \tag{13}$$

$$\frac{r_1+r_2}{2a} = 1 - \cos\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2}, \tag{14}$$

$$n(t_1 - t_2) = 2m\pi + \alpha - \beta - \cos\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}. \tag{15}$$

With appropriate trigonometric identities, (13) and (14) can be expressed as

$$\cos\beta - \cos\alpha = \frac{c}{a},$$

$$\cos\beta + \cos\alpha = 2 - \frac{r_1+r_2}{a}.$$

Solving these two equations, yields

$$\cos\alpha = 1 - \frac{s}{a} = 1 + 2U, \tag{16}$$

$$\cos\beta = 1 + 2KU, \tag{17}$$

where we have defined

$$s = \frac{r_1+r_2+c}{2}, \qquad U = -\frac{s}{2a}, \qquad \text{and} \quad K = 1 - \frac{c}{s}. \tag{18}$$

Since $\cos\alpha = 1 - 2\sin^2(\alpha/2)$, (16) can be changed to

$$U = -\sin^2\frac{\alpha}{2}, \qquad 0 \le \alpha < 2\pi, \tag{19}$$

and similarly, (17) becomes

$$KU = -\sin^2\frac{\beta}{2}, \qquad -\pi \le \beta < \pi. \tag{20}$$

Note that the limits for α and β come from the two inequalities for (11) and (12).

The parameter K can be alternatively expressed as

$$K = \frac{s-c}{s} = \frac{r_1 + r_2 - c}{2s} = \frac{(r_1+r_2)^2 - c^2}{4s^2},$$

and introducing (6), we have

$$K = \frac{r_1 r_2 (1 + \cos\Delta\theta)}{2s^2} = \frac{r_1 r_2}{s^2}\cos^2\frac{\Delta\theta}{2}.$$

Now, substituting (19) in (20),

$$\sin\frac{\beta}{2} = q\sin\frac{\alpha}{2}, \qquad -\pi \le \beta < \pi, \tag{21}$$

where

$$q = \pm\sqrt{K} = \frac{\sqrt{r_1 r_2}}{s}\cos\frac{\Delta\theta}{2}. \tag{22}$$

Note that the sign of q is taken care of by the angle $\Delta\theta$:

$$\begin{aligned} 1 \ge q \ge 0 \quad &\text{for} \quad \Delta\theta \le \pi, \\ 0 \ge q \ge -1 \quad &\text{for} \quad \Delta\theta \ge \pi. \end{aligned} \tag{23}$$

We now have an equation relating α and β as a function of q, which is known. Therefore, we need another equation relating these two parameters to find their values. We can introduce U into (15), since

$$n(t_2 - t_1) = \sqrt{\frac{\mu}{a^3}}(t_2 - t_1) = J(-U)^{\frac{3}{2}},$$

where

$$J = \sqrt{\frac{8\mu}{s^3}}(t_2 - t_1). \tag{24}$$

Consequently,

$$J = (-U)^{-\frac{3}{2}}\left[2m\pi + \alpha - \beta - \cos\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}\right], \tag{25}$$

which can also be written as

$$J = (-U)^{-\frac{3}{2}}\left[2m\pi + \alpha - \beta - (\sin\alpha - \sin\beta)\right]. \tag{26}$$

Substituting (19) into (26), we obtain

$$J\sin^3\frac{\alpha}{2} = 2m\pi + \alpha - \beta - \sin\alpha + \sin\beta. \tag{27}$$

(21) and (27) with $0 \leq \alpha < 2\pi$ are Lambert's equations for elliptic motion. Given q, J and m, they are to be solved for α and β. Then, a can be obtained from (19) and it is a simple matter to find all other quantities associated with the orbit.

This solution is only valid for the elliptic case. However, an universal solution valid for elliptic, hyperbolic and parabolic transfers (although more elaborated) can also be obtained. For curious readers, an interesting and relatively simple approach can be found in Lancaster *et al.*[1].

The `astrofun.py` module includes two relevant functions. The function `params(data)` computes q and J from Lambert's problem initial data, and `elements(alpha_beta, data)` calculates the orbital elements of the transfer orbit (given the problem's solution α, β and initial data). Both functions assume direct motion (that is, in the same direction as the Sun rotates).

2.2 Example 3: Visualizing Lambert's equations for elliptic motion

$$J \sin^3 \frac{\alpha}{2} = 2m\pi + \alpha - \beta - \sin\alpha + \sin\beta,$$
$$\sin\frac{\beta}{2} = q \sin\frac{\alpha}{2}.$$

These constitute a system of two equations with two unknowns, which can be visualized as two curves whose intersection becomes the solution to Lambert's problem. To analyze their behaviour, the two curves can be plotted for different values of q, J and m.

To accomplish this, both equations need to be expressed as implicit functions:

$$F_1(\alpha,\beta) = 2m\pi + \alpha - \beta - \sin\alpha + \sin\beta - J\sin^3\frac{\alpha}{2} = 0, \tag{28}$$
$$F_2(\alpha,\beta) = \sin\frac{\beta}{2} - q\sin\frac{\alpha}{2} = 0, \tag{29}$$

which can then be plotted through the `plt.contour()` function, just as it would be done using MATLAB:

```python
import numpy as np
import matplotlib.pyplot as plt

q, J, m = 0.8, 4, 0

a_range = np.linspace(-6*np.pi, 6*np.pi, 1000)
b_range = np.linspace(-6*np.pi, 6*np.pi, 1000)
A, B = np.meshgrid(a_range, b_range)

F1 = 2*m*np.pi + A - B - np.sin(A) + np.sin(B) - J*(np.sin(A/2))**3
F2 = np.sin(B/2) - q*np.sin(A/2)
```

[1] E. R. Lancaster and R. C. Blanchard, *A Unified Form of Lambert's Theorem*, National Aeronautics and Space Administration, Washington, 1969.

```
plt.figure()
plt.contour(A, B, F1, 0, colors='r') # Plots the contour line at F1 = 0
plt.contour(A, B, F2, 0, colors='k')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'$\beta$')
```

By clicking *Run file*, the figure containing both curves shows up. Observe how $F_2(\alpha, \beta) = 0$ is in reality an infinite grid-like pattern, and multiple intersections occur between both curves. Modify the values of $q \in [-1, 1]$, $J \in (0, \infty]$ and $m \in \mathbb{N}$ and observe how the curves change. The bounds $0 < \alpha < 2\pi$ and $-\pi < \beta < \pi$ are required to choose the correct solution. By adding the following lines:

```
plt.xlim([0, 2*np.pi])
plt.ylim([-np.pi, np.pi])
```

this can be clearly appreciated. The intersection at $\alpha = \beta = 0$ always occurs, and must be disregarded.

Recall that these equations are only valid for elliptic motion. Values of α near 0 imply $a \to \infty$ according to (19), approaching a parabolic solution. When $m = 0$, (28) breaks down for $\alpha = 0$ and suffers from a critical loss of significant digits in the neighborhood of $\alpha = 0$.

To observe why this happens, plot the equations for $q = -0.5$, $m = 0$ and a range of J between 10 and 1.6. When J becomes smaller (a shorter time of flight), a more direct and quick path is required between the starting and arrival positions. The transfer orbit therefore begins to approach a parabola. In Figure 3, (28) has been plotted at uniform intervals of J. Note how the solution drastically changes for small variations in J at lower values.

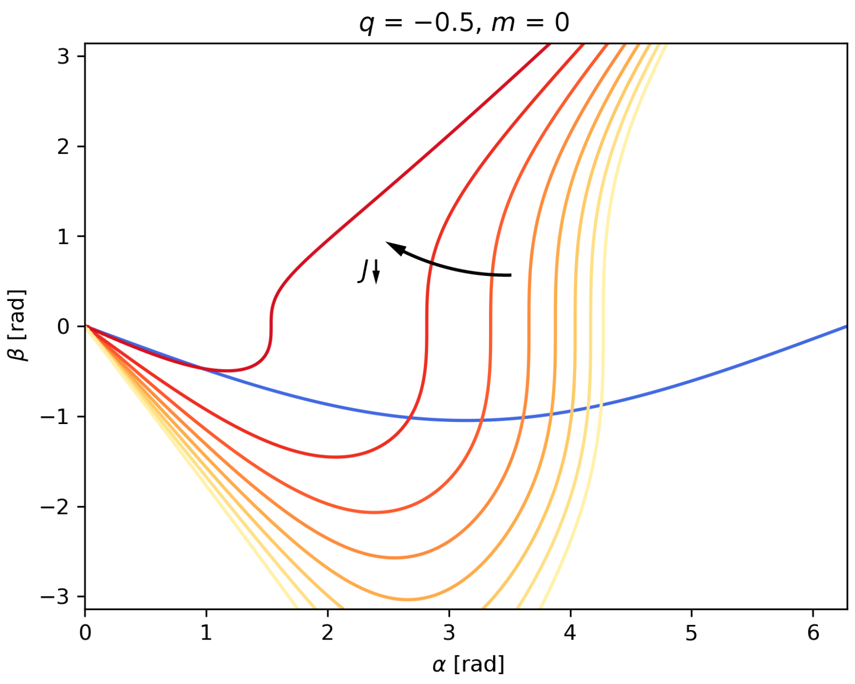


Figure 3: Behaviour of (28) with equally spaced changes in J.

When a clear intersection between (28) and (29) occurs, the transfer orbit is elliptic and the solution is valid. When both curves become tangent at $\alpha = \beta = 0$ (execute the previous code with $q = -0.5$, $m = 0$ and $J = 1.5$), a parabolic transfer orbit is reached, which can't be solved using these equations. Hyperbolic transfer orbits appear when none of the previous conditions are met (try $0 < J < 1.5$).

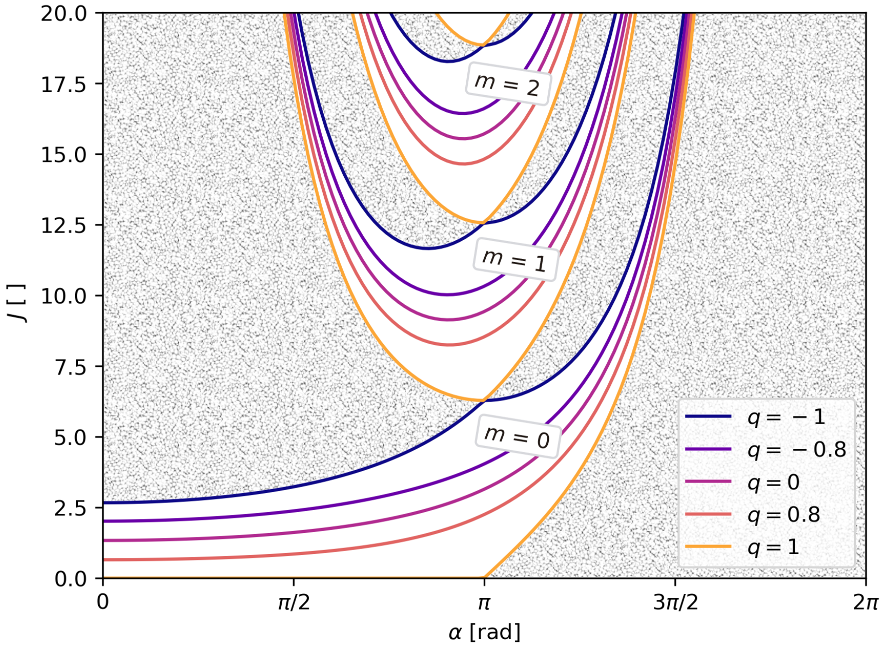


Figure 4: Plot of J against α for selected values of q and m.

In Figure 4, the complete set of solutions (values of α) for a range of J and selected values of q and m has been plotted. The conclusions obtained earlier can be clearly appreciated in this graph. When $m = 0$, for a fixed value of q there exists a lower bound for J, below which no solution can be obtained. Note that for the cases with multiple revolutions ($m \neq 0$) two different solutions exist. Additionally, unrealizable regions (no existing solution) have been shaded in the figure.

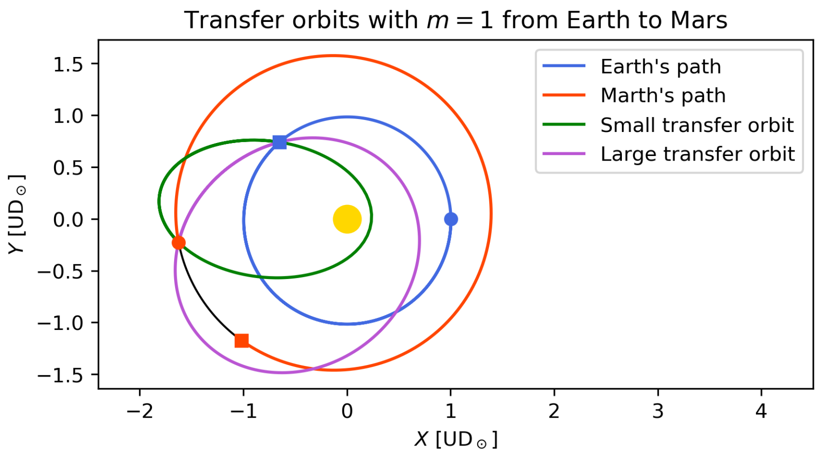


Figure 5: Duplicity of solutions for the multi-revolution case.

Theoretically, one could solve Lambert's problem by simply plotting the equations and finding the intersection, or locating it in the graph from Figure 4. However, to optimize interplanetary missions it needs to be solved an elevated number of times, and therefore a numeric approach is the only viable option. In the following section, an algorithm to accurately solve Lambert's equations is developed.

2.3 Programming a simple algorithm for the elliptic case

Lambert's equations can be solved using refined root finding algorithms such as `fsolve()`, readily accessible in Python. However, bounds can't be set to the variables with `fsolve()`. To find a solution within $0 < \alpha < 2\pi$ and $-\pi < \beta < \pi$, constrained optimization is the easiest way to proceed. The `least_squares()` function is convenient here; equations can be directly passed to it, and it will minimize the sum of squares of its components. In other words, it will find the values of α and β which at the same time bring (28) and (29) closer to 0 (which is what we are looking for).

Create a new file, delete the template and then write the following preamble:

```python
import numpy as np
from scipy.optimize import least_squares
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from astrofun import julian, eph, posvel, params, elements, PlotSol2D,
↪  PlotSol3D
from example2 import PlotOrbit2D, PlotOrbit3D
# Instead of example2, write the name of the file you have created
```

This will be the base file for later work. Besides importing the functions included in the module `astrofun.py`, import as well the functions `PlotOrbit2D()` and `PlotOrbit3D()` which were created earlier in Section 1.3. Remember to save this file at the same directory were the others are.

Next, a function which solves Lambert's equations when $m = 0$ (allowing $m \neq 0$ complicates matters and for our purposes is not required) can be defined:

```python
'''Lambert solver (only valid for m = 0 and elliptic motion)'''
def Lambert(q, J):
    def F(z): # where z is an array containing alpha (a) and beta (b)
        a, b = z # a = z[0] and b = z[1]
        F1 = J*(np.sin(a/2))**3 - (a - b - np.sin(a) + np.sin(b))
        F2 = np.sin(b/2) - q*np.sin(a/2)
        return [F1, F2]
    alpha_beta = least_squares(F, [np.pi, 0], bounds=([1e-5,-np.pi],
    ↪  [2*np.pi, np.pi])).x
    return alpha_beta
```

The function `least_squares(fun, x0, bounds)` takes in the following parameters:

- `fun`: Function(s) to be solved.

- `x0` : Initial guess on independent variables.
- `bounds` : In the format `([a_start, b_start], [a_end, b_end])`

and outputs a *result* structure with multiple fields. To access the actual solution, `.x` is appended to the call.

3 Analysis and optimization of interplanetary missions

Finally, here we use Python to solve Lambert's problem and go to Mars. We see how our numbers match those produced by NASA's engineers!

3.1 Example 4: Analyzing the MSL mission

Now, a real solution to Lambert's problem can be plotted. Consider the Mars Science Laboratory mission, launched by NASA on November 26, 2011, which successfully landed Curiosity, a Mars rover, on August 6, 2012. Continuing the previous script:

```
17  '''Solving the problem'''
18  JD1 = julian(26,11,2011) # launch date
19  JD2 = julian(6,8,2012) # arrival date
20  
21  elem_Earth1 = eph(3, JD1)
22  elem_Mars2 = eph(4, JD2)
23  r_Earth1 = posvel(elem_Earth1)[0] # starting position
24  r_Mars2 = posvel(elem_Mars2)[0] # arrival position
25  
26  dataL = [r_Earth1, r_Mars2, JD2-JD1]
27  q, J = params(dataL)
28  alpha_beta = Lambert(q, J)
29  elem_trans1 = elements(alpha_beta, dataL) # At JD1
30  
31  T_Earth = 2*np.pi*np.sqrt(elem_Earth1[0]**3/1)*58.1324 # [days]
32  T_Mars = 2*np.pi*np.sqrt(elem_Mars2[0]**3/1)*58.1324 # [days]
33  T_trans = 2*np.pi*np.sqrt(elem_trans1[0]**3/1)*58.1324 # [days]
34  # The period doesnt't perceptibly vary between JD1 and JD2
```

The procedure required to solve Lambert's problem is straight-forward. First, the starting and arrival positions are obtained at t_1 and t_2 respectively (lines 18-25). Next, the parameters q and J are obtained through the function `params()`. Calling `Lambert()`, a solution α, β to this particular problem is found. This solution is then used to obtain the orbital elements of the transfer orbit at t_1 through `elements()`.

Plotting the various orbits is simple:

```
35  '''2D Figure'''
36  plt.figure()
37  plt.title('MSL mission')
38  # COMPLETE ORBITS
39  PlotOrbit2D(3, JD1, JD1+T_Earth, 'k') # Earth
40  PlotOrbit2D(4, JD1, JD1+T_Mars, 'k') # Mars
41  PlotSol2D(elem_trans1, JD1, JD1+T_trans, 'k') # Transfer orbit
42  # PATH'S TRAVELED
43  PlotOrbit2D(3, JD1, JD2, 'b', True)
44  PlotOrbit2D(4, JD1, JD2, 'r', True)
45  PlotSol2D(elem_trans1, JD1, JD2, 'g')
```

```python
'''3D Figure'''
plt.figure()
ax = plt.axes(projection='3d')
ax.set_title('MSL mission')
# COMPLETE ORBITS
PlotOrbit3D(3, JD1, JD1+T_Earth, 'k') # Earth
PlotOrbit3D(4, JD1, JD1+T_Mars, 'k') # Mars
PlotSol3D(elem_trans1, JD1, JD1+T_trans, 'k') # Transfer orbit
# PATH'S TRAVELED
PlotOrbit3D(3, JD1, JD2, 'b', True)
PlotOrbit3D(4, JD1, JD2, 'r', True)
PlotSol3D(elem_trans1, JD1, JD2, 'g')
```

For plotting the transfer orbit, functions `PlotSol2D()` and `PlotSol3D()` have been used. These are defined in a very similar way to `PlotOrbit2D()` and `PlotOrbit3D()`, with slight variations (now the ephemerides are not used, and instead the orbital elements are directly passed as an input argument).

By clicking *Run file*, two new windows appear containing both figures. Note that the solution appears to be very similar to a Hohmann transfer orbit. This is no coincidence; when considering the two-dimensional problem, the Hohmann transfer orbit can be proven to be the optimal solution. In three dimensions, the starting and arrival orbits are no longer co-planar and therefore slight variations occur. Furthermore, neither the Earth or Mars' orbits are circular, nor their lines of apsides are aligned.

The engineers at NASA try to choose launch and arrival dates which provide the optimal solution, and real world transfer orbits may traverse slightly more, or slightly less, than 180° around the Sun. Sure enough, the MSL mission describes what's called a *Type 1* ($\Delta\theta < 180°$) Hohmann transfer orbit.

3.2 Characteristic energy and pork-chop plots

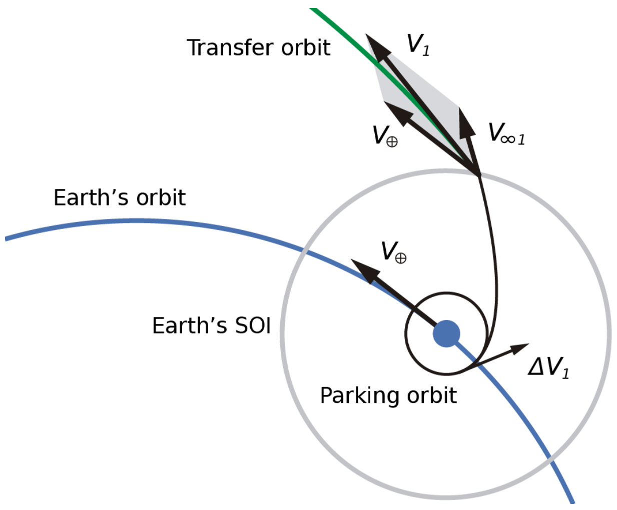


Figure 6: Geocentric phase in the patched conic approximation.

As it has been reviewed in class, through the concept of sphere of influence (SOI) and the method of patched conic approximation, trajectory calculations are greatly simplified. In Figure 6, the Earth is represented by a blue dot. The impulse $\Delta\mathbf{V}_1$ is given to the spacecraft at the parking orbit to set it onto an hyperbolic trajectory. This allows it to escape Earth's SOI with a velocity $\mathbf{V}_{\infty 1}$ along one of the hyperbola's asymptotes, which is a good approximation considering that the Earth's SOI is close to 2 million km in diameter (while parking orbits range within $100 \sim 300$ km of altitude).

When the spacecraft reaches the Earth's SOI, we switch from a geocentric frame of reference to a heliocentric one. $\mathbf{V}_{\infty 1}$ and the Earth's velocity $\mathbf{V}_\oplus$ can be added to obtain $\mathbf{V}_1$, the velocity which the spacecraft has at the starting point of the transfer orbit.

Hyperbolic excess velocity, $\mathbf{V}_\infty$, represents the excess specific kinetic energy over that which is required to simply escape from the center of attraction. Its square is denoted C_3 ($C_3 = V_\infty^2$), and is known as the *characteristic energy*. C_3 is a measure of the energy required for an interplanetary mission, and it allows for the comparison of different trajectories in energetic terms.

The function `v1v2()` included in `astrofun.py` takes the same input arguments as `elements()` and directly computes the velocity vectors $\mathbf{V}_1$ and $\mathbf{V}_2$. Additionally, through the functions `eph()` and `posvel()`, the starting and arrival planets' velocity vectors can be obtained. Considering an Earth to Mars mission, it follows that

$$C_3|_{\text{launch}} = V_{\infty 1}^2 = \|\mathbf{V}_1 - \mathbf{V}_\oplus\|^2$$

$$C_3|_{\text{arrival}} = V_{\infty 2}^2 = \left\|\mathbf{V}_2 - \mathbf{V}_♂\right\|^2$$

And the total characteristic energy, $C_3|_{\text{total}} = C_3|_{\text{launch}} + C_3|_{\text{arrival}}$, can be easily calculated.

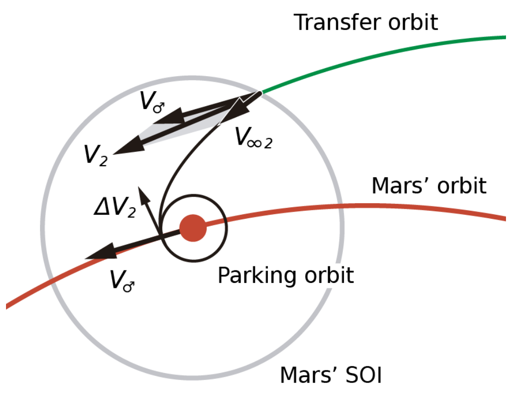


Figure 7: Planetocentric phase in the patched conic approximation.

The tool traditionally used to choose the launch and arrival dates for a one-way impulsive mission is the pork-chop[1] plot. The classical pork-chop plot uses as a cost function the characteristic energy C_3, and expresses it as a function of possible launch and arrival dates. A given contour, called a *pork-chop curve*, represents constant C_3, and at the center of these pork-chop curves we find the optimal transfer orbit (the one with minimum C_3).

[1]This name comes from the distinctive shape of the plot, resembling grilled pork meat.

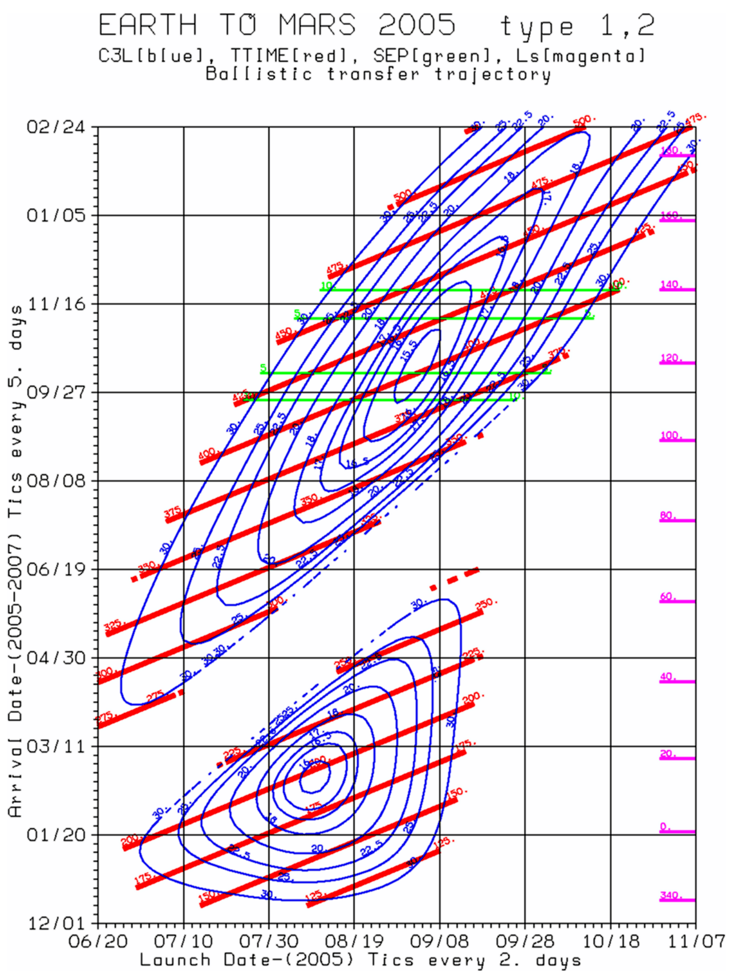


Figure 8: Representative pork-chop plot for the 2005 Mars launch opportunity. A given blue contour represents a solution with a constant C_3, and the red lines represent trips with the same time of flight[1].

More than one optimal solution may exist for a given range of dates. In Figure 8 a pork-chop plot for the 2005 Mars launch opportunity is represented $(C3|_{\text{launch}})$. On the horizontal axis, different launch dates are considered. On the vertical axis, we find possible arrival dates.

Although this pork-chop plot doesn't provide the true optimal dates, which would be obtained by plotting $C3|_{\text{total}}$, it is still of great interest. To match a launcher with a mission, $C3|_{\text{launcher}}$ must be greater than $C3|_{\text{launch}}$.

Here, the pork-chop curves are represented up to $C_3 = 30\ \text{km}^2/\text{s}^2$, which is an upper limit given by the maximum C_3 that can be provided by the launcher. We can clearly differentiate two separate regions on this pork-chop plot. In the lower part of the graph we find transfer orbits of *Type 1*, which imply short times of flight of about seven months. In the upper part, we find transfer orbits of *Type 2*, which can take up to two years. The decision whether to choose Type 1 or Type 2 transfer orbits depends on multiple factors; the C_3 required, the suitability of both the launch and arrival dates, the time of flight, etc.

[1] *"Porkchop" is the First Menu Item on a Trip to Mars*, NASA Jet Propulsion Laboratory, Available at: `http://mars.jpl.nasa.gov/spotlight/porkchopAll.html`

3.3 Example 5: Optimizing NASA's MRO mission

The Mars Reconnaissance Orbiter (MRO) is a spacecraft designed to study the geology and climate of Mars, provide reconnaissance of future landing sites, and relay data from surface missions back to Earth. It was launched in the year 2005 and will continue to operate through the late 2020s, far beyond its intended design life.

In this example, a similar plot to that in Figure 8 will be obtained, and an optimal launch date calculated. To do this, create yet another new file, delete the template, and write the following preamble:

```python
import numpy as np
import matplotlib.pyplot as plt
from astrofun import julian, gregorian, eph, posvel, params, L, v1v2
```

The function `L()` found inside `astrofun.py` is identical to `Lambert()`, the Lambert solver created previously —recall that when importing functions from another file, Python runs the entire script (and unless the lines outside the function(s) are included within an `if __name__ == '__main__':` statement, these are executed). Therefore, this approach is more convenient.

To generate a contour plot, a grid in Julian days must be created first:

```python
'''Selecting the plot's limits (those in Figure 8)'''
JD1_start = julian(20,6,2005)
JD1_stop = julian(7,11,2005)

JD2_start = julian(1,12,2005)
JD2_stop = julian(24,2,2007)

JD1_num, JD2_num = 100, 100 # Discretization points of each axis

JD1_range = np.linspace(JD1_start, JD1_stop, JD1_num)
JD2_range = np.linspace(JD2_start, JD2_stop, JD2_num)
X, Y = np.meshgrid(JD1_range, JD2_range)
```

Computing the launch and arrival C_3 at every every pair of dates involves calling a few functions every time, and is a very time-consuming process. For instance, creating a 100 by 100 grid implies a total of 10000 discretization points, which roughly translates into a full minute of computing time.

After initializing two arrays for $C_3|_{\text{launch}}$ and $C_3|_{\text{arrival}}$ and setting the iteration parameters to zero,

```python
'''Calculating the characteristic energies'''
C3_launch = np.zeros([JD2_num, JD1_num])
C3_arrival = np.zeros([JD2_num, JD1_num])
i, j = 0, 0 # Indices
```

two for-loops must be created to perform a complete sweep through the grid:

```python
for JD1 in JD1_range:
    for JD2 in JD2_range:
        elem_Earth1 = eph(3, JD1)
        elem_Mars2 = eph(4, JD2)

        r_Earth1, V_Earth1 = posvel(elem_Earth1)
        r_Mars2, V_Mars2 = posvel(elem_Mars2)

        dataL = [r_Earth1, r_Mars2, JD2-JD1]
        q, J = params(dataL)
        if J-4/3*(1-q**3)<0.05:
            # Too close to a parabola or no longer elliptic
            C3_launch[i, j] = None
            C3_arrival[i, j] = None # Translates into NaN
        else:
            alpha_beta = L(q, J)
            V1_vec, V2_vec = v1v2(alpha_beta, dataL)

            C3_launch[i, j] = np.linalg.norm(V1_vec-V_Earth1)**2
            C3_arrival[i, j] = np.linalg.norm(V2_vec-V_Mars2)**2
            # In canonical units [UV^2]
        i+=1
    i=0
    j+=1

C3_L = C3_launch*887.128 # [km^2/s^2] SI units
C3_A = C3_arrival*887.128 # [km^2/s^2]
C3 = C3_L + C3_A # [km^2/s^2]
```

It must be noted that not every point in the pork-chop plot may correspond to an elliptic transfer orbit (although parabolic and hyperbolic orbits require higher amounts of energy, and therefore will never be the optimal solution). For the parabolic case, the following relationship holds:

$$J = \frac{4}{3}(1 - q^3),$$

and for hyperbolic orbits, $J < 4/3(1 - q^3)$. Thus, a quick and simple procedure to identify whether a transfer orbit is elliptic, parabolic or hyperbolic is to calculate $J - 4/3(1 - q^3)$ and check if it's a positive value (ellipse), equal to zero (parabola) or a negative value (hyperbola).

For our purposes, a value lesser than 0.05 will imply a transfer orbit too close to a parabola and therefore a result with poor precision (as the equations used are only valid for elliptic motion). In the previous code, an if-statement regarding this behaviour is implemented. If $J - 4/3(1 - q^3) < 0.05$, through `np.inf` a floating point representation of (positive) infinity will be assigned to the characteristic energies of that particular transfer orbit. `np.inf` is not equivalent to a `NaN` (Not A Number) value, and can be used inside conditional expressions (as it simply represents a very big number).

Once the scalar fields of $C3|_{\text{launch}}$ and $C3|_{\text{arrival}}$ have been calculated, they can be plotted separately or added together to generate the pork-chop plot of $C3|_{\text{total}}$:

```python
'''Generating the pork-chop plots'''
# LAUNCH C3
plt.figure()
plt.title(r'MRO mission ($C_{3,L}$)')
P1 = plt.contour(X, Y, C3_L, [15.5, 16, 16.5, 17, 18, 20, 22.5, 25, 30])
# Labeling the contour lines and axes
plt.clabel(P1, fontsize=7, inline=True, fmt='%1.1f')  # Labels
plt.xlabel('Launch date (Julian day)')
plt.ylabel('Arrival date (Julian day)')

# TOTAL C3
plt.figure()
plt.title(r'MRO mission ($C_3$)')
P2 = plt.contour(X, Y, C3, [24.5, 26, 28, 31, 35, 40, 47])
plt.clabel(P2, fontsize=7, inline=True, fmt='%1.1f')
plt.xlabel('Launch date (Julian day)')
plt.ylabel('Arrival date (Julian day)')
```

Now run the file and observe both figures. The first contour plot shows the pork-chop curves for $C3|_{\text{launch}}$, which are indeed identical to those displayed in Figure 8. The second one represents $C3|_{\text{total}}$ and clearly reveals two differentiated relative minimums (Type 1 and Type 2 transfer orbits).

The optimal solution can be found using `np.min(C3)`. However, this approach doesn't provide a way to differentiate between Type 1 and Type 2 solutions, or compute both minimums separately. Therefore, a perhaps more *rudimentary* procedure needs to be implemented. On one hand, a pair of for-loops can be created to access every element from the matrix `C3`, which contains the values of the total characteristic energy.

On the other hand, recall that Type 1 and Type 2 orbits resemble a Hohmann transfer orbit ($\Delta\theta = 180°$), the former being slightly shorter ($\Delta\theta < 180°$) and the latter slightly longer ($\Delta\theta > 180°$). To differentiate between both solutions, a simple rule can be developed. Consider the hypothetical period of a Hohmann transfer orbit solution to the 2D problem:

$$a_H = \frac{a_{\text{Earth}} + a_{\text{Mars}}}{2}, \qquad T_H = 2\pi\sqrt{\frac{a_H^3}{\mu_\odot}}.$$

Then, a transfer orbit will be of Type 1 if its time of flight is less than $T_H/2$, or of Type 2 otherwise:

```python
'''Computing the optimal dates and C3'''
a_H = (eph(3, JD1)[0] + eph(4, JD1)[0])/2 # [UD]
T_H = 2*np.pi*np.sqrt(a_H**3/1)*58.1324 # [days]

C3_T1, C3_T2 = np.inf, np.inf # To enter the if-statement
for j in range(JD1_num):
    for i in range(JD2_num):
        tf = JD2_range[i] - JD1_range[j]
        if tf < T_H/2:
            if C3[i,j] < C3_T1:
                C3_T1 = C3[i,j]
                JD1_T1 = JD1_range[j]
                JD2_T1 = JD2_range[i]
        else:
            if C3[i,j] < C3_T2:
                C3_T2 = C3[i,j]
                JD1_T2 = JD1_range[j]
                JD2_T2 = JD2_range[i]
```

Besides obtaining the C_3 of each solution, the launch and arrival dates can also be conveniently stored. To print the resulting values, add:

```
print('Type 1 transfer orbit:', gregorian(JD1_T1), '-->',
↪ gregorian(JD2_T1), 'with C3 =', C3_T1, 'km^2/s^2')
print('Type 2 transfer orbit:', gregorian(JD1_T2), '-->',
↪ gregorian(JD2_T2), 'with C3 =', C3_T2, 'km^2/s^2')
```

where the function `gregorian()` (imported from `astrofun.py`) has been used to convert the Julian days to dates in the Gregorian calendar (using DD/MM/YYYY format).

The MRO was launched on August 12, 2005 and reached Mars on March 10, 2006. Does this correspond to a Type 1 or a Type 2 transfer orbit?

Although not identical, the optimal dates which have been obtained are very close to the actual launch and arrival dates (only 5 days off). The discrepancies can very well be due to other factors such as bad weather at the optimal launch date, and surely due to the multiple simplifications which have been made to tackle the problem. However, this outcome proves that most of these simplifications are indeed appropriate, and provide surprisingly accurate results.

3.4 Planning ahead

In the previous example, only the 2005 Mars launch opportunity has been analyzed. However, in the context of future interplanetary missions, it is interesting to consider whether spacecraft can be launched at any desired year.

As it has been studied, the feasibility of either a Type 1 or Type 2 transfer orbit depends primarily on the relative position between Earth and Mars —how long does it take for these planets to be in a similar relative position? The answer to this question corresponds with the *synodic period*. If the orbital periods of two bodies around a third one (e.g. the Sun) are called T_1 and T_2, so that $T_1 < T_2$, their synodic period T_s is given by:

$$\frac{1}{T_s} = \frac{1}{T_1} - \frac{1}{T_2}$$

The synodic period of Mars, relative to Earth, turns out to be approximately of 779.9 days or 2.135 years (prove this to yourself). Therefore, a launch opportunity should be expected roughly every two years. Since 2005, the number of missions to Mars carried out every year have been represented in Figure 9.

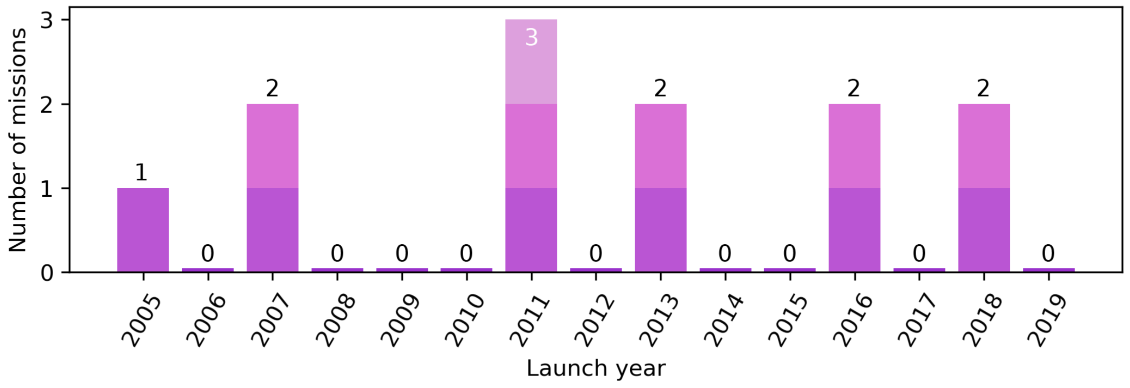


Figure 9: Number of missions to Mars carried out every year since 2005.

Just as it has been predicted, a trend can be clearly appreciated —spacecraft are launched approximately every two years.

Furthermore, the characteristic energies required for future missions can be calculated, in order to choose an optimal launch year. For instance, during the decade of the 2020s, the optimal (minimum) $C_3|_{\text{total}}$ can be computed for both Type 1 and Type 2 solutions:

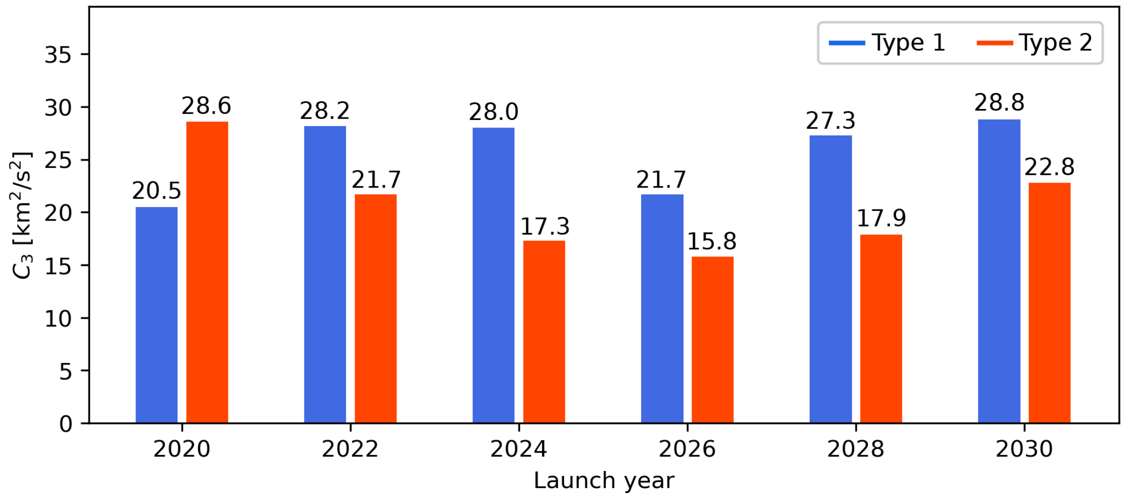


Figure 10: Optimal $C_3|_{\text{total}}$ of Type 1 and Type 2 transfer orbits during the 2020s.

Observing Figure 10, the best year at first sight seems to be 2026, although many more factors need to be taken into account when designing a real mission. Additionally, notice how the overall optimal $C_3|_{\text{total}}$ can be achieved through transfer orbits of either type.

4 Python, NumPy and Matplotlib for MATLAB users

In this section a brief summary of the key differences between Python and MATLAB syntax is provided. The purpose of this comparison is to create a concise *cheat sheet* with the most frequent commands and expressions, to aid the student in the otherwise tedious task of searching and remembering this information.

4.1 General purpose equivalents

MATLAB	Python	Notes
`help func`	`help(func)`	Get help on the function `func`

`a && b`
`a || b`
	`a and b`
	`a or b`
		Logical operators

`~=`	`!=`	Conditional expressions
`^`	`**`	Basic operators
`[x, y] = 2output_fun()`	`x, y = 2output_fun()`	Multiple assignments
`disp(a)`	`print(a)`	Display a result
`;` (semicolon)	(nothing)	Terminate a statement
`%`	`#`	Comment

`for i=[1 2 3]`
`  disp(i)`
`end`
	`for i in [1, 2, 3]:`
	`  print(i)`
		For-loops

`if a==0`
`   disp('a is zero')`
`elseif a<0`
`   disp('a is negative')`
`else`
`   disp('a is positive')`
`end`
	`if a==0:`
	`   print('a is zero')`
	`elif a<0:`
	`   print('a is negative')`
	`else:`
	`   print('a is positive')`
		If-else statements

`function [x, y] = pm(a,b)`
`   x = a + b;`
`   y = a - b;`
`end`
	`def = pm(a,b):`
	`   x = a + b`
	`   y = a - b`
	`   return [x, y]`
		Regular functions

`mult = @(a,b) a*b`	`mult = lambda a,b: a*b`	Anonymous functions
`a(2,5)`	`a[1,4]`	Access element in second row, fifth column
`a(2,:)`	`a[1,:]`	Entire second row of `a`
`a(1:5,:)`	`a[0:5,:]`	First five rows of `a`

4.2 NumPy equivalents

Before using the NumPy package, remember to execute the following command in Python:

```
import numpy as np
```

The table below gives some rough equivalents for some common MATLAB expressions. For more detail read the built-in documentation on the NumPy functions.

MATLAB	Python	Notes
`size(a)`	`shape(a)` *or* `a.shape`	Get the size of an array
`size(a,n)`	`shape(a)[n-1]` *or* `a.shape[n-1]`	Get the number of elements of the *n*-th dimension of array `a`
`[1 2 3]`	`np.array([1, 2, 3])`	Create vectors
`[1 2 3; 4 5 6]`	`np.array([[1, 2, 3],[4, 5, 6]])`	Create matrices

`a.*b`
`a.\b`
`a.^b`
	`a*b`
	`a\b`
	`a**b`
		Element-wise operations

`2:10`
`0:9`
	`np.arange(2,11)`
	`np.arange(9)`
		Create an increasing vector

`zeros(3,4)`
`ones(3,4)`
	`np.zeros([3,4])`
	`np.ones([3,4])`
		Array full of zeros or ones

`linspace(1,3,4)`	`np.linspace(1,3,4)`	Four equally spaced samples between 1 and 3, inclusive
`max(a)`	`a.max()`	Maximum element of `a`
`max(a)`	`a.max(0)`	Maximum element of each column of matrix `a`
`max(a,[],2)`	`a.max(1)`	Maximum element of each row of matrix `a`

Table 1: Linear algebra equivalents.

4.3 Matplotlib equivalents

Before using the Matplotlib package, remember to execute the following command in Python:

```
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
```

2D Plots

Plotting a simple 2D line plot:

Matlab

```
theta = linspace(-pi,pi,100);
f1 = sin(theta);
f2 = sin(theta);

figure
plot(theta,f1,theta,f2)
title('Trigonometric functions')
xlabel('$\theta$', 'interpreter',...
'latex')
ylim([-1,1])
```

Python

```
theta = np.linspace(-np.pi,np.pi,100)
f1 = np.sin(theta)
f2 = np.sin(theta)

plt.figure()
plt.plot(theta,f1,theta,f2)
plt.title('Trigonometric functions')
plt.xlabel(r'$\theta$')

plt.ylim([-1,1])
```

Plotting dots:

Matlab

```
x = [0 2 4]
y = [5 4 3]

figure
scatter(x,y)
```

Python

```
x = np.array([0, 2, 4])
y = np.array([5, 4, 3])

plt.figure()
plt.scatter(x,y)
```

3D Plots

Plotting a 3D curve:

Matlab

```
x = linspace(0,2*pi,100);
y = sin(x);
z = sin(x);

figure

plot3(x,y,z)
title('Helix')
xlabel('Longitudinal axis')
ylim([-1,1])
zlim([-1,1])
```

Python

```
x = linspace(0,2*np.pi,100)
y = np.sin(x)
z = np.sin(x)

plt.figure()
ax = plt.axes(projection='3d')
ax.plot3D(x,y,z)
ax.set_title('Helix')
ax.set_xlabel('Longitudinal axis')
ax.set_ylim([-1,1])
ax.set_zlim([-1,1])
```

Plotting dots:

Matlab

```
scatter3(x,y,z)
```

Python

```
ax.scatter3d(x,y,z)
```

Plotting the contour line at $Z(X, Y) = 0$:

Matlab

```
x = linspace(-pi,pi,100);
y = linspace(0,2*pi,100);
[X, Y] = meshgrid(x,y);

Z = sin(X) + cos(Y)

figure
contour(X,Y,Z,[0 0])
```

Python

```
x = linspace(-np.pi,np.pi,100);
y = linspace(0,2*np.pi,100);
X, Y = meshgrid(x,y);

Z = sin(X) + cos(Y)

plt.figure()
plt.contour(X,Y,Z,0)
```

Saving a figure to a file in Python

To obtain a raster image (.png):

```
plt.savefig("name.png", bbox_inches='tight', dpi=300)
```

and to obtain a vector image (.pdf, .eps, .svg, etc.):

```
plt.savefig("name.pdf", bbox_inches='tight')
```

Appendix B
Auxiliary Python algorithms

B.1 Julian day calculation

The function `julian(D, M, Y)` computes the Julian day of a date given in DD/MM/YYYY format. The calculation is made at 00:00 UT.

Input(s)	`D`: Day. `M`: Month. `M`: Year.
Output(s)	`JD`: Julian day.

```python
def julian(D, M, Y):
    JD = 367*Y - np.floor(7/4*(Y+np.floor((M+9)/12))) +
    ↪  np.floor(275*M/9) + D + 1721013.5
    return JD  # scalar
```

B.2 Julian day to Gregorian calendar date

The function `gregorian(JD)` computes a date in DD/MM/YYYY format (gregorian calendar) from its Julian day (the inverse of what `julian(D, M, Y)` does).

Input(s)	`JD`: Julian day.
Output(s)	`D`: Day. `M`: Month. `M`: Year.

```python
def gregorian(JD):
    f = int(JD + 1401 + 3*(4*JD+274277)/4/146097 - 38)
    e = int(4*f + 3)
    g = int((e % 1461)/4)
    h = int(5*g + 2)
    D = int((h % 153)/5 + 1)
    M = int((h/153+2) % 12 + 1)
    Y = int(e/1461 - 4716 + (14-M)/12)
    return [D, M, Y]
```

B.3 Planetary ephemerides

The function `eph(planet_number, JD)` computes the orbital elements of a major planet's orbit at a given Julian day.

Input(s)	`planet_number`: following the convention indicated in the code below. `JD`: Julian day.
Output(s)	`[a, e, i, RAAN, w, t_tau_days]`: A list containing the orbital elements.

```
def eph(planet_number, JD):
    mu = 1 # [UD*UV^2] in the Sun's canonical units

    # rows: Mercury (1), Venus (2), EM Bary(3), Mars(4), Jupiter(5),
    ↪  Saturn(6), Uranus(7), Neptune(8), Pluto(9)
    # columns: a[AU], e[], i[deg], RAAN[deg], wbar[deg], L[deg]

    elements0 = np.array([
        [ 3.87099270e-01,  2.05635930e-01,  7.00497902e+00,
        ↪  4.83307659e+01,  7.74577963e+01,  2.52250324e+02],
        [ 7.23335660e-01,  6.77672000e-03,  3.39467605e+00,
        ↪  7.66798426e+01,  1.31602467e+02,  1.81979099e+02],
        [ 1.00000261e+00,  1.67112300e-02, -1.53100000e-05,
        ↪  0.00000000e+00,  1.02937682e+02,  1.00464572e+02],
        [ 1.52371034e+00,  9.33941000e-02,  1.84969142e+00,
        ↪  4.95595389e+01, -2.39436296e+01, -4.55343205e+00],
        [ 5.20288700e+00,  4.83862400e-02,  1.30439695e+00,
        ↪  1.00473909e+02,  1.47284798e+01,  3.43964405e+01],
        [ 9.53667594e+00,  5.38617900e-02,  2.48599187e+00,
        ↪  1.13662424e+02,  9.25988783e+01,  4.99542442e+01],
        [ 1.91891646e+01,  4.72574400e-02,  7.72637830e-01,
        ↪  7.40169250e+01,  1.70954276e+02,  3.13238105e+02],
        [ 3.00699228e+01,  8.59048000e-03,  1.77004347e+00,
        ↪  1.31784226e+02,  4.49647623e+01,  5.51200297e+01],
        [ 3.94821168e+01,  2.48827300e-01,  1.71400121e+01,
        ↪  1.10303937e+02,  2.24068916e+02,  2.38929038e+02]])

    rates = np.array([
        [ 3.70000000e-07,  1.90600000e-05, -5.94749000e-03,
        ↪  -1.25340810e-01,  1.60476890e-01,  1.49472674e+05],
        [ 3.90000000e-06, -4.10700000e-05, -7.88900000e-04,
        ↪  -2.77694180e-01,  2.68329000e-03,  5.85178154e+04],
        [ 5.62000000e-06, -4.39200000e-05, -1.29466800e-02,
        ↪  0.00000000e+00,  3.23273640e-01,  3.59993724e+04],
        [ 1.84700000e-05,  7.88200000e-05, -8.13131000e-03,
        ↪  -2.92573430e-01,  4.44410880e-01,  1.91403027e+04],
        [-1.16070000e-04, -1.32530000e-04, -1.83714000e-03,
        ↪  2.04691060e-01,  2.12526680e-01,  3.03474613e+03],
              [-1.25060000e-03, -5.09910000e-04,  1.93609000e-03,
              ↪  -2.88677940e-01, -4.18972160e-01,  1.22249362e+03],
              [-1.96176000e-03, -4.39700000e-05, -2.42939000e-03,
              ↪  4.24058900e-02,  4.08052810e-01,  4.28482028e+02],
              [ 2.62910000e-04,  5.10500000e-05,  3.53720000e-04,
              ↪  -5.08664000e-03, -3.22414640e-01,  2.18459453e+02],
              [-3.15960000e-04,  5.17000000e-05,  4.81800000e-05,
              ↪  -1.18348200e-02, -4.06294200e-02,  1.45207805e+02]])

           # Computation of the orbital elements
           a, e, i_deg, RAAN_deg, wbar_deg, L_deg =
           ↪  elements0[planet_number-1,:] +
           ↪  rates[planet_number-1,:]*(JD-2451545.0)/36525
           i, RAAN, wbar, L = np.pi/180*np.array([i_deg, RAAN_deg, wbar_deg,
           ↪  L_deg]) # convert the angles to radians

           w = (wbar - RAAN)
           M = (L - wbar) % (2*np.pi)
           # % (2*np.pi) yields M between 0 and 2pi

           if M>np.pi:
              M = M-2*np.pi  # to obtain M between -pi and pi

           # Time since periapsis passage (t-tau)
           t_tau = M/np.sqrt(mu/a**3)  # [UT] in the Sun's canonical units
           t_tau_days = t_tau*58.1324  # [days]

           return [a, e, i, RAAN, w, t_tau_days]  # list of scalars
```

B.4 Determining r and v through the orbital elements

The function posvel(elements) computes the position and velocity vectors from the orbital elements. The results are given in the Sun's canonical units:

$$1\ \mathrm{UD}_\odot = 1\ \mathrm{AU}, \qquad 1\ \mathrm{UV}_\odot = 29.7847\ \mathrm{km/s}, \qquad 1\ \mathrm{UT}_\odot = 58.1324\ \mathrm{days}$$

Input(s)	elements: A list containing the following orbital elements: [a, e, i, RAAN, w, t_tau_days].
Output(s)	r_vec_ecl: Radius vector of the body in heliocentric ecliptic coordinates. v_vec_ecl: Velocity vector of the body in heliocentric ecliptic coordinates.

```python
def posvel(elements):
    a, e, i, RAAN, w, t_tau_days = elements

    mu = 1 # [UD*UV^2]
    t_tau = t_tau_days/58.1324   # [UT]

    M = t_tau*np.sqrt(mu/a**3)
    E = fsolve(lambda E: E-e*np.sin(E)-M, np.pi)   # between -pi and pi

    # Position and velocity vectors in perifocal coordinates
    r_vec_perif = np.array([a*(np.cos(E)-e),
     ↪  a*np.sqrt(1-e**2)*np.sin(E)])
    r = np.linalg.norm(r_vec_perif)   # modulus of the position vector

    v_vec_perif = np.array([-np.sqrt(mu*a)/r*np.sin(E),
     ↪  np.sqrt(mu*a*(1-e**2))/r*np.cos(E)])

    # Conversion to heliocentric ecliptic coordinates (J2000)
    R =
    np.array([[np.cos(w)*np.cos(RAAN)-np.sin(w)*np.sin(RAAN)*np.cos(i),
     ↪  -np.sin(w)*np.cos(RAAN)-np.cos(w)*np.sin(RAAN)*np.cos(i)],
    [np.cos(w)*np.sin(RAAN)+np.sin(w)*np.cos(RAAN)*np.cos(i),
     ↪  -np.sin(w)*np.sin(RAAN)+np.cos(w)*np.cos(RAAN)*np.cos(i)],
    [np.sin(w)*np.sin(i), np.cos(w)*np.sin(i)]])

    r_vec_ecl = np.matmul(R, r_vec_perif).flatten()
    v_vec_ecl = np.matmul(R, v_vec_perif).flatten()
    # .flatten() converts vectors to 1D - e.g. (3,)

    return [r_vec_ecl, v_vec_ecl]  # list of 1D vectors
```

B.5 Computing the q and J parameters

The function `params(data)` computes q and J from Lambert's problem data, assuming direct motion.

Input(s)	`data: [r1_vec, r2_vec, tf_days]`, a list where: → `r1_vec`: Radius vector of the starting planet at t_1 in h.e.c.[a] → `r1_vec`: Radius vector of the arrival planet at t_2 in h.e.c. → `tf_days`: Flight time in days ($t_f = t_2 - t_1$)
Output(s)	`[q, J]`: A list containing the parameters q and J.

[a] Heliocentric ecliptic coordinates.

```python
def params(data):
    r1_vec, r2_vec, tf_days = data

    mu = 1 # [UD*UV^2]
    tf = tf_days/58.1324 # [UT] (1 UT = 58.1324 days)

    r1, r2 = np.linalg.norm(r1_vec), np.linalg.norm(r2_vec)
    Dtheta_small =
     ↪  np.arccos(np.clip(np.dot(r1_vec,r2_vec)/r1/r2,-1.0,1.0))
    # np.clip to avoid rounding errors
    if np.cross(r1_vec,r2_vec)[2]>0:
    # if r1xr2 is pointing in the positive Z direction
        Dtheta = Dtheta_small
    else:
        Dtheta = 2*np.pi - Dtheta_small
    c = np.sqrt(r1**2 + r2**2 - 2*np.dot(r1_vec, r2_vec))   # [UD]
    s = (r1 + r2 + c)/2   # [UD]

    q = np.sqrt(r1*r2)/s*np.cos(Dtheta/2)
    J = np.sqrt(8*mu/s**3)*tf

    return [q, J]   # list of scalars
```

B.6 Determining an orbit through α and β

The function `elements(alpha_beta, data)` computes the orbital elements of the transfer orbit solution to Lambert's problem, assuming direct motion.

Input(s)	`alpha_beta: [alpha, beta]`, the solution to Lambert's equations. `data: [r1_vec, r2_vec, tf_days]`, a list where: → `r1_vec`: Radius vector of the starting planet at t_1 in h.e.c.[a] → `r1_vec`: Radius vector of the arrival planet at t_2 in h.e.c. → `tf_days`: Flight time in days ($t_f = t_2 - t_1$). `m`: Indicates the number of complete revolutions ($m = 0$ by default).
Output(s)	`elements`: A list containing the following orbital elements: `[a, e, i, RAAN, w, t_tau_days]`.

[a] Heliocentric ecliptic coordinates.

```python
def elements(alpha_beta, data, m=0):
    alpha, beta = alpha_beta
    r1_vec, r2_vec, tf_days = data
    mu = 1 # [UD*UV^2]
    tf = tf_days/58.1324 # [UT] (1 UT = 58.1324 days)

    r1, r2 = np.linalg.norm(r1_vec), np.linalg.norm(r2_vec)
    Dtheta_small =
    ↪  np.arccos(np.clip(np.dot(r1_vec,r2_vec)/r1/r2,-1.0,1.0))
    # np.clip to avoid rounding errors
    if np.cross(r1_vec,r2_vec)[2]>0:
    # if r1xr2 is pointing in the positive Z direction
        Dtheta = Dtheta_small
    else:
        Dtheta = 2*np.pi - Dtheta_small
    c = np.sqrt(r1**2 + r2**2 - 2*np.dot(r1_vec, r2_vec))  #  [UD]
    s = (r1 + r2 + c)/2  # [UD]

    a = s/(2*np.sin(alpha/2)**2) # semi-major axis [UD]
    n = np.sqrt(mu/a**3) # [UT^-1]

    E2_E1 = 2*np.pi*m + alpha - beta  # (E1-E2)

    r1dot = (n*tf - E2_E1 +
    ↪  (1 - r1/a)*np.sin(E2_E1))/(r1/np.sqrt(mu*a)*(1 - np.cos(E2_E1)))
    # r2dot = (-n*tf + E2_E1 +
    ↪  (1 - r2/a)*np.sin(-E2_E1))/(r2/np.sqrt(mu*a)*(1 -
    ↪  np.cos(-E2_E1)))

    v1 = np.sqrt(2*mu/r1 - mu/a)
    # v2 = np.sqrt(2*mu/r2 - mu/a)
    v1_theta = np.sqrt(v1**2 - r1dot**2)
    # v2_theta = np.sqrt(v2**2 - r2dot**2)
    # or v_theta = np.sqrt(mu*a*(1-e**2))/r alternatively

    v1_vec = (r1dot - v1_theta/np.tan(Dtheta))/r1*r1_vec + v1_theta/np.sin(Dtheta)/r2*r2_vec
    # v2_vec = -v2_theta/np.sin(Dtheta)/r1*r1_vec + (r2dot + v2_theta/np.tan(Dtheta))/r2*r2_vec

    h_vec = np.cross(r1_vec, v1_vec)
    h = np.linalg.norm(h_vec)

    e_vec = np.cross(v1_vec, h_vec)/mu - r1_vec/r1
    e = np.linalg.norm(e_vec)

    n_vec = np.cross(np.array([0, 0, 1]), h_vec)/np.linalg.norm(np.cross(np.array([0, 0, 1]), h_vec))

    i = np.arccos(h_vec[2]/h) % np.pi  # between 0 and pi

    RAAN = np.arctan2(n_vec[1], n_vec[0])
    # arctan2 returns a value between -pi and pi

    if e_vec[2]>0:
    # e_vec above the reference plane, and therefore 0 < w < pi
        w = np.arccos(np.clip(np.dot(n_vec, e_vec)/e, -1.0, 1.0))
    else:  # -pi < w < 0
        w = - np.arccos(np.clip(np.dot(n_vec, e_vec)/e, -1.0, 1.0))

    if r1dot>0:  # 0 < theta1 < pi
        theta1 = np.arccos((a*(1 - e**2) - r1)/e/r1)
    else:  # -pi < theta1 < 0
        theta1 = - np.arccos((a*(1 - e**2) - r1)/e/r1)
    E1 = 2*np.arctan(np.sqrt((1-e)/(1+e))*np.tan(theta1/2))
    # between -pi and pi
    M1 = E1 - e*np.sin(E1)  # between -pi and pi
    t1_tau = M1/n  # [UT] time since periapsis passage (if negative, |t1_tau| is time until periapsis passage)
    t1_tau_days = t1_tau*58.1324  # [days]

    # if r2dot>0:
    #     theta2 = np.arccos((a*(1 - e**2) - r2)/e/r2)
    # else:
    #     theta2 = - np.arccos((a*(1 - e**2) - r2)/e/r2)
    # E2 = 2*np.arctan(np.sqrt((1-e)/(1+e))*np.tan(theta2/2))
    # M2 = E2 - e*np.sin(E2)
    # t2_tau = M2/n

    return [a, e, i, RAAN, w, t1_tau_days]  # list of scalars
```

B.7 Computing the starting and arrival velocity vectors

The function `v1v2(alpha_beta, data)` computes the starting and arrival velocity vectors within the transfer orbit ($\mathbf{V}_1$ and $\mathbf{V}_2$, defined in Sect. 3.3), required to calculate the total C_3. This function is very similar to `elements(alpha_beta, data)`, and also assumes direct motion.

Input(s)	`alpha_beta`: `[alpha, beta]`, the solution to Lambert's equations. `data`: `[r1_vec, r2_vec, tf_days]`, a list where: → `r1_vec`: Radius vector of the starting planet at t_1 in h.e.c.[a] → `r1_vec`: Radius vector of the arrival planet at t_2 in h.e.c. → `tf_days`: Flight time in days ($t_f = t_2 - t_1$). `m`: Indicates the number of complete revolutions ($m = 0$ by default).
Output(s)	`v1_vec`: Starting velocity vector (within the transfer orbit) in h.e.c. `v2_vec`: Arrival velocity vector (within the transfer orbit) in h.e.c.

[a] Heliocentric ecliptic coordinates.

```python
def elements(alpha_beta, data, m=0):
    alpha, beta = alpha_beta
    r1_vec, r2_vec, tf_days = data
    mu = 1 # [UD*UV^2]
    tf = tf_days/58.1324 # [UT] (1 UT = 58.1324 days)

    r1, r2 = np.linalg.norm(r1_vec), np.linalg.norm(r2_vec)
    Dtheta_small =
     ↪  np.arccos(np.clip(np.dot(r1_vec,r2_vec)/r1/r2,-1.0,1.0))
    # np.clip to avoid rounding errors
    if np.cross(r1_vec,r2_vec)[2]>0:
    # if r1xr2 is pointing in the positive Z direction
        Dtheta = Dtheta_small
    else:
        Dtheta = 2*np.pi - Dtheta_small
    c = np.sqrt(r1**2 + r2**2 - 2*np.dot(r1_vec, r2_vec))  #  [UD]
    s = (r1 + r2 + c)/2  # [UD]
    a = s/(2*np.sin(alpha/2)**2) # semi-major axis [UD]
    n = np.sqrt(mu/a**3)  # [UT^-1]

    E2_E1 = 2*np.pi*m + alpha - beta  # (E1-E2)

    r1dot = (n*tf - E2_E1 +
     ↪  (1 - r1/a)*np.sin(E2_E1))/(r1/np.sqrt(mu*a)*(1 - np.cos(E2_E1)))
    r2dot = (-n*tf + E2_E1 +
     ↪  (1 - r2/a)*np.sin(-E2_E1))/(r2/np.sqrt(mu*a)*(1-np.cos(-E2_E1)))

    v1 = np.sqrt(2*mu/r1 - mu/a)
    v2 = np.sqrt(2*mu/r2 - mu/a)
    v1_theta = np.sqrt(v1**2 - r1dot**2)
    v2_theta = np.sqrt(v2**2 - r2dot**2)

    v1_vec = (r1dot - v1_theta/np.tan(Dtheta))/r1*r1_vec +
    ↪  v1_theta/np.sin(Dtheta)/r2*r2_vec
    v2_vec = -v2_theta/np.sin(Dtheta)/r1*r1_vec + (r2dot +
    ↪  v2_theta/np.tan(Dtheta))/r2*r2_vec

    return [v1_vec, v2_vec]  # list of vectors
```

B.8 Solving Lambert's equations for elliptic motion

The function `L(q, J, m=0, LongPath=False)` solves Lambert's equations for elliptic motion. For the multi-revolution case ($m \neq 0$) this algorithm is not very reliable, and the results must be taken with extreme caution.

Input(s)	`q`: Lambert's problem adimensional parameter q. `J`: Lambert's problem adimensional parameter J. `m`: Indicates the number of complete revolutions ($m = 0$ by default). `LongPath`: If `False` (option by default), indicates that the longer path (larger transfer orbit) has been chosen in the multi-revolution case.
Output(s)	`alpha_beta`: `[alpha, beta]`, the solution to Lambert's equations.

```python
def L(q, J, m=0, LongPath=False):
    def f(z):
        a, b = z
        f1 = J*(np.sin(a/2))**3 - (2*m*np.pi+a-b-np.sin(a)+np.sin(b))
        f2 = np.sin(b/2) - q*np.sin(a/2)
        return [f1,f2]

    # For m!=0 this algorithm is not very robust, and should be avoided
    if m==0:
        alpha_beta = least_squares(f, [np.pi, 0], bounds =
         ↪ ([1e-5,-np.pi], [2*np.pi, np.pi])).x
    else:
        if LongPath:
            alpha_beta = least_squares(f, [0.1, 0], bounds =
             ↪ ([1e-5,-np.pi], [np.pi, np.pi])).x
        else:
            alpha_beta = least_squares(f, [6, 0], bounds =
             ↪ ([2.6,-np.pi], [2*np.pi, np.pi])).x

    return alpha_beta  # numpy array
```

B.9 Plotting a 2D representation of the transfer orbit

The function `PlotSol2D(elem_trans1, JD1, JD2, color, ends=False)` plots a 2D figure of the transfer orbit containing the arc traveled from t_1 to t_2.

Input(s)	`elem_trans1`: A list containing the following orbital elements from the transfer orbit at t_1: `[a, e, i, RAAN, w, t1_tau_days]` `JD1`: t_1 in Julian Days. `JD2`: Arbitrary date in Julian days, `JD2>JD1`. `color`: String indicating the desired plotting color. `ends`: If `True`, two markers at the starting and ending points of the curve are plotted.
Output(s)	Two dimensional figure containing the arc traveled from t_1 to t_2.

```python
def PlotSol2D(elem_trans1, JD1, JD2, color, ends=False):
    pts = 500 # Number of discretization points
    r_vec = np.zeros([pts, 3]) # Initializing the array
    i = 0
    for JD in np.linspace(JD1, JD2, pts):
        t_tau_trans = elem_trans1[5] + JD - JD1
        elem_trans = elem_trans1[:5] + [t_tau_trans]
        r_vec[i,:] = posvel(elem_trans)[0]
        i+=1

    plt.plot(r_vec[:,0], r_vec[:,1], color)
    plt.scatter(0, 0, s=180, c='gold') # s=thickness, c=color
    if ends: # if ends==True
        plt.plot(r_vec[0,0], r_vec[0,1], 's', c=color)
        plt.plot(r_vec[-1,0], r_vec[-1,1], 'o', c=color)

    plt.gca().set_aspect('equal', adjustable='box')
    plt.xlabel(r'$X$ [UD$_\odot$]') # \odot is the Sun's symbol
    plt.ylabel(r'$Y$ [UD$_\odot$]')
```

B.10 Plotting a 3D representation of the transfer orbit

The function `PlotSol3D(elem_trans1, JD1, JD2, color, ends=False)` plots a 3D figure of the transfer orbit containing the arc traveled from t_1 to t_2.

Input(s)	`elem_trans1`: A list containing the following orbital elements from the transfer orbit at t_1: `[a, e, i, RAAN, w, t1_tau_days]` `JD1`: t_1 in Julian Days. `JD2`: Arbitrary date in Julian days, `JD2>JD1`. `color`: String indicating the desired plotting color. `ends`: If `True`, two markers at the starting and ending points of the curve are plotted.
Output(s)	Three dimensional figure containing the arc traveled from t_1 to t_2.

```python
def PlotSol2D(elem_trans1, JD1, JD2, color, ends=False):
    pts = 500
    r_vec = np.zeros([pts, 3])
    i = 0
    for JD in np.linspace(JD1, JD2, pts):
        t_tau_trans = elem_trans1[5] + JD - JD1
        elem_trans = elem_trans1[:5] + [t_tau_trans]
        r_vec[i,:] = posvel(elem_trans)[0]
        i+=1

    ax = plt.gca()
    ax.plot3D(r_vec[:,0], r_vec[:,1], r_vec[:,2], color)
    ax.scatter3D(0, 0, 0, s=150, c='gold')
    if ends:
        ax.scatter3D(r_vec[0,0], r_vec[0,1], r_vec[0,2], c=color,
         ↪  marker='s')
        ax.scatter3D(r_vec[-1,0], r_vec[-1,1], r_vec[-1,2], c=color)

    lim = 1.2*elem_trans1[0]
    ax.auto_scale_xyz([-lim,lim], [-lim,lim], [-0.2,0.2])
    ax.set_xlabel(r'$X$ [UD$_\odot$]')
    ax.set_ylabel(r'$Y$ [UD$_\odot$]')
    ax.set_zlabel(r'$Z$ [UD$_\odot$]')
```