uc3m | Universidad Carlos III de Madrid

University Degree in Aerospace Engineering

2022-2023

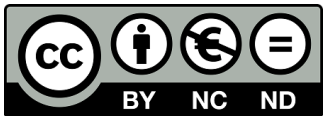
Bachelor Thesis

"Development of an educational astrodynamics library in Julia"

Alicia Sanjurjo Barrio

Mario Merino Martínez

Madrid, Spain, June 2023



ABSTRACT

Julia is a growing open-source programming language. Although there are some Julia libraries available that tackle Orbital Dynamics problems, they do not have an educational purpose. Hence, this project aims to develop a user-friendly Astrodynamics library in Julia, which can be used in the context of education by undergraduate students with a basic knowledge of programming and Orbital Dynamics. The AstrodynamicsEdu.jl package developed in this thesis, will thus allow them to deepen their understanding of the course.

Based on the fundamental concepts from Mechanics and Orbital Dynamics, the structure of the code was designed; so to make sure that the data types and functions developed would fit the real needs of students. This Astrodynamics library includes a broad range of functions that allow the user to: solve problems of the Ideal Two-Body Problem; perform reference frame and vector basis changes; compute the Delta-v needed to perform orbital maneuvers such as the Hohmann Transfer; propagate orbits considering orbital perturbations; to plot and visualize orbits both in the perifocal plane and in ECI frame.

All these functions are backed up thoroughly with rigorous testing . In addition, their application in undergraduate orbital courses have been demonstrated through a set of educational examples, which are available to the user to help them use the library.

The AstrodynamicsEdu.jl package developed can be found in its GitHub repository, which can be accessed through the following link: https://github.com/AliciaSBa/Astrodynamics.jl.git. The installation and usage of the library is explained both in the README document and in this thesis.

Keywords: Astrodynamics library; Astrodynamics package; Astrodynamics; Orbital Dynamics; Orbital Mechanics; Julia; Julia library; Julia package; Educational library.

DEDICATION

First, I would like to thank my parents for their unconditional support; I am who I am today because of them. My mom always says that raising a kid is not easy, because you never know if you are making the right choices. However, I can confidently say that they did a pretty good job.

Thank you also to my brother Sergio, I am so glad to have him. He has always been an inspiration for me, because when he puts his mind into something he always accomplishes it. Thank you for listening to me babbling about orbits for hours, even when you have no idea what I am talking about.

I would also like to thank all my wonderful friends, who are always there for me through thick and thin. These last four years have been a challenge, and I did not spend as much time as I would have liked with all of you. But those moments with you always make for the best memories.

And specially I would like to thank my tutor Mario, for guiding me and helping me get through this journey. As well as all the other professors from Universidad Carlos III de Madrid and UCLA, thank you for making me a bit wiser.

"per aspera ad astra"

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

As an Aerospace student, I have always been widely fascinated by space exploration and the critical role that Astrodynamics plays in this field. The ability to predict and control a spacecraft's trajectory is essential to virtually every aspect of space mission planning, from the initial design phase to the launch and beyond.

Nowadays, computational tools and libraries have become essential in solving astrodynamics problems. For this purpose, many new different programming languages have emerged and been developed in recent years, among them, the Julia programming language. Julia is a dynamic programming language that offers a unique combination of high performance and ease of use that is ideal for computational science applications.

However, there is a significant gap in the availability of educational resources and software libraries that specifically address orbital dynamics problems in Julia. Although there are many resources available for other languages, such as MATLAB and Python, there is a lack of them for Julia.

That is why the main aim of this bachelor thesis is to develop an educational astrodynamics library in Julia that is accessible to aerospace students with a basic background in programming and an interest in learning orbital dynamics. Since this library's focus is to ease their learning and help them understand astrodynamics in a more dynamic way. In addition, this library could also contribute to expand the tools available related to the space sector for the growing community of Julia users.

1.2. Objectives

The main goal of this bachelor thesis is to develop an user-friendly Astrodynamics library in Julia which is accessible to aerospace students with a basic knowledge of programming. For this purpose, this thesis will strive to meet the following four objectives:

- Conduct a comprehensive functional analysis to identify and define the requirements for the Astrodynamics library.
- Establish a modular structure and outline the desired functionalities to be implemented within the library.
- Implement the identified modules and functionalities in Julia, adhering to best practices and coding standards.

- Validate the functionality and performance of the implemented modules through rigorous testing and provide illustrative examples to demonstrate their usage in the context of orbital dynamics.

1.3. Document Structure

This document is divided into five main different parts: Introduction, Mechanics and Orbital Dynamics, Development, Evaluation, and Conclusions and Future Work.

1. **Introduction**. It serves to clarify the project main objectives, the software used, as well as some important concepts to understand how the Julia programming language works and the capabilities that it offers.
2. **Mechanics and Orbital Mechanics**. In this section the basic theory behind the code is explained. All the most relevant formulas and parameters concerning Orbital Dynamics are defined. In addition to some linear algebra concepts definitions and relative kinematics formulas.
3. **Development**. This part focuses on the library itself, how it was made, its progression through time, its different parts and structure. This is where all the available functions and data types are explained.
4. **Evaluation**. Here the developed library is compared with others as well as analyzed through tests to assess that it complies with the requirements and objectives of the project.
5. **Conclusions and Future Work**. The results obtained are further analyzed, and future lines of work are considered. Also, it is briefly explained how the user could access this library on their own.

1.4. Legal Framework

In this thesis, thee legal framework comprises the software tools utilized as well as the intellectual property of the work.

1.4.1. Software

In order to develop this project, the following software tools were used:

- **Julia**
- **GitHub Copilot**
- **Jupyter Notebooks**

1.4.2. Intellectual Property

The intellectual property rights of this work created belong solely to the author. This document and the codes developed have been elaborated without any involvement or support from public or private organizations.

2. STATE OF THE ART

Due to the longstanding demand for tools that can predict spacecraft trajectories in space, numerous Astrodynamics libraries have been developed over time using popular programming languages such as Python, C++, and MATLAB. Despite the existence of these established languages, Julia has emerged as a promising new programming language.

2.1. Why Julia

Julia is a dynamic and high-performance programming language that has gained significant popularity since its introduction in 2012. Combining the strengths of established languages with its unique features, Julia offers a clean syntax and exceptional performance comparable to Fortran and C++. Its interactive nature and support for interactive use make it a preferred choice for various applications. Moreover, as an open-source language, Julia is freely accessible to all, making it particularly attractive for students and researchers. Its versatility extends beyond scientific computing to encompass domains like web development and machine learning. With a growing ecosystem and the ability to handle diverse tasks, Julia has emerged as a powerful and promising language in the field of computational programming. [1]

2.2. Julia basics

2.2.1. Packages and Modules

In Julia, libraries are organized into **packages**. A package is a collection of Julia source code files that provide specific functionality. Packages can be installed and managed using Julia's built-in package manager, which is called Pkg. The package manager allows us to download, update, and manage dependencies for the packages we need in our project. [2]

Packages in Julia are stored in a central package repository called the General registry. The General registry is a collection of packages maintained by the Julia community. However, it is also possible to create our own packages and manage them locally.

Modules, on the other hand, are used for organizing code within a package or a Julia script. A module is a container for related functions, types, and other definitions. It helps to organize code into logical units, improve code reusability, and avoid naming conflicts. By using modules, we can group related functionality together and selectively import specific functionality into our code. [3]

In Julia, we can define a module using the `module` keyword followed by the module

name. We can also control which definitions or functions are accessible outside the module by using the `export` keyword. This allows us to specify which functions or types should be visible to other code that imports the module.

To use a package or a module in Julia, we typically need to import it into our code. The `import` keyword is used to bring a package or module into scope, allowing you to access its functionality. You can also use the `using` keyword, which is similar to `import` but has additional features like automatic name qualification.

Overall, packages and modules are fundamental components of Julia's ecosystem, enabling code organization, code sharing, and modularity. They help manage dependencies, promote code reuse, and allow users to build and distribute their own packages for specific purposes.

2.2.2. Types

In Julia, types play a crucial role in the language's performance, as well as its flexibility and expressiveness. Julia uses a type system that allows you to define and manipulate different types of data. [4]

1. **Primitive Types**: Julia provides a set of primitive types, such as `Int`, `Float64`, `Bool`, `Char`, etc., which represent basic data types like integers, floating-point numbers, boolean values, and characters.

2. **Composite Types**: Julia allows you to define composite types using the `struct` keyword. Composite types are user-defined structures that can hold multiple values together. They are similar to structures or classes in other programming languages.

3. **Abstract Types**: Abstract types in Julia are used as a way to group related types together under a common supertype. Abstract types cannot be instantiated directly, but they can be used as a way to define behavior and constraints for multiple concrete types.

4. **Parametric Types**: Julia supports parametric types, which allow you to define types that can be parameterized by other types. This provides a way to create generic types that can work with different underlying data types.

5. **Type Hierarchy and Multiple Dispatch**: Julia has a rich type hierarchy, where types are organized in a tree-like structure. This hierarchy allows for multiple dispatch, which means that the behavior of a function can be specialized based on the types of its arguments.

The type of a variable or expression can be specified using type annotation, denoted by the `::` notation. For example, `number1::Float64` explicitly declares `number1` as a

`Float64` type. Type annotations are optional but can provide hints to the compiler and improve performance.

To obtain the type of an object or expression, the `typeof()` function can be used. It returns the type of the provided object. For example, `typeof(number1)` will return the type `Float64`.

Using type annotation and `typeof()`, we can define the type of variables, obtain the type of objects, and leverage type information for performance optimization or dynamic behavior based on an object's type.

2.3. Previous Work

In Julia, there are several libraries related to space exploration that are highly effective for performing complex calculations and simulations. For instance, the JuliaAstro ecosystem offers various libraries for astronomical and astrophysical research. One notable example is the AstroLib.jl library, which provides a wide range of functions for performing astrodynamical calculations. Additionally, the SPICE.jl library is another powerful tool that enables users to interact with NASA's SPICE toolkit for handling spacecraft navigation and exploration tasks. Another existing libraries worth mentioning is the SatelliteDynamics.jl, which allows to model orbit and attitude dynamics.

However, despite the existence of these libraries, there is a need for a user-friendly library that can be used by students with a basic programming background to learn about Orbital Dynamics in a dynamic and visual way. Unfortunately, none of the existing libraries available at the time of writing this document are suitable for simple educational purposes. That is why the need to create an easy-to-use, simple, and accessible Astrodynamics library in Julia arises.

3. MECHANICS AND ORBITAL DYNAMICS

This chapter serves as a bridge between the underlying theory of Mechanics and Orbital Dynamics and its practical implementation in the code. It focuses on the necessary elements and their incorporation into the codebase, providing a clear understanding of how these concepts are translated into the library's modules.

The chapter begins by outlining the key structures, definitions, formulas, and notation that are essential for the code implementation. It highlights the specific aspects of Mechanics and Orbital Dynamics that directly influence the design and functionality of the library. However, it does not aim to provide an exhaustive explanation of the entire theory.

By emphasizing the practical application of Mechanics and Orbital Dynamics in the context of the library, this chapter enables readers to grasp the fundamental concepts required to effectively utilize and navigate the codebase.

3.1. Mechanics

In space as on Earth, a ground foundation of Mechanics is necessary for a general understanding of the motion and behavior of objects under the influence of forces. To characterize this behavior, it is first necessary to define some fundamental linear algebra objects and their properties.

3.1.1. Vector Bases

In mechanics, it is often convenient to represent physical quantities using vectors. A vector is a mathematical object that has both magnitude and direction. In order to express any vector in space, vector bases are used.

A **vector basis** is an ordered set of linearly independent vectors that spans the vector space. In three-dimensional space, the most commonly used is the canonical right-handed vector basis B_0, which is composed of mutually orthogonal unit vectors $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$: $B_0 : \{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$. These vectors align with the coordinated axes O_x, O_y, and O_z. B_0 follows the right-hand rule, ensuring a consistent orientation. It is important to note that the vectors in a vector basis are free vectors, meaning their position can be translated without altering their meaning. In this library, the main focus will be in orthonormal right-handed vector bases like B_0.

Any vector can be expressed as the sum of its components along these basis vectors. For example, a vector $\mathbf{v}$ can be written as $\mathbf{v} = v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$, where v_x, v_y, and v_z represent the vector's components along the O_x, O_y, and O_z axes, respectively. [5]

Simple rotations

A rotation from reference frame B_0 to B_1 is considered a simple rotation when the axis of rotation coincides with one of the vectors of B_0. In a simple rotation, the vector of B_0 that serves as the axis of rotation remains fixed, while the rest of the vector space rotates by an angle θ around it, following the right-hand rule.

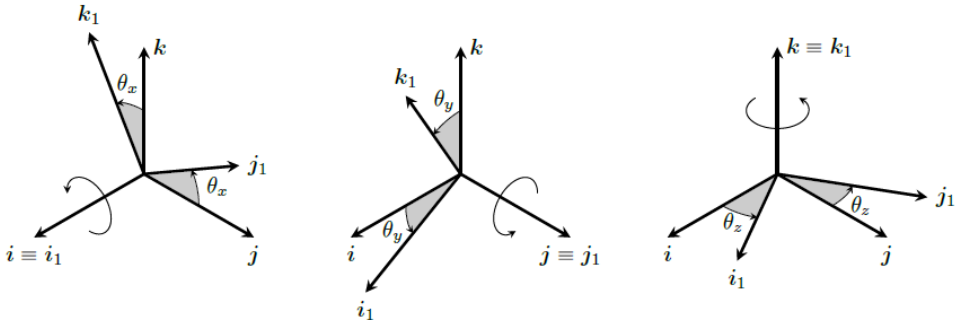


Fig. 3.1. Simple rotations about each of the coordinate axes [5].

Figure 3.1 illustrates the three possibilities of positive rotation about the **i**, **j**, and **k** axes. The rotation matrices for each case are written below:

$$ {}_0R_1(x,\theta_x) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} \tag{3.1} $$

$$ {}_0R_1(y,\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_x y & 0 & \cos\theta_y \end{bmatrix} \tag{3.2} $$

$$ {}_0R_1(z,\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3} $$

It can be noted that to project a vector in basis B_1 to basis B_0, it could just be done by multiplying by the rotation or transformation matrix, as follows

$$ \begin{bmatrix} v_{x_0} \\ v_{y_0} \\ v_{z_0} \end{bmatrix} = [{}_0R_1(y,\theta_y)] \begin{bmatrix} v_{x_1} \\ v_{y_1} \\ v_{z_1} \end{bmatrix} \tag{3.4} $$

These transformations are essential, as in order to do any operation between two vectors (sum, cross product...) both vectors must be expressed in the same vector basis.

3.1.2. Reference Frames

A **reference frame** is a coordinate system used to describe the motion of objects. A reference frame is characterized by its origin point (O) and its vector basis (B_0), and it is written using following notation: $S_0 : \{O; B_0\}$. In mechanics, two common types of reference frames are inertial frames and non-inertial frames. [6]

An **inertial frame** is a reference frame in which Newton's first law of motion holds true. According to this law, an object at rest remains at rest, and an object in motion continues its motion with a constant velocity in a straight line, unless acted upon by an external force. In practical terms, an inertial frame is a reference frame that is not accelerating.

On the other hand, a **non-inertial frame** is a reference frame that is accelerating. In a non-inertial frame, objects appear to experience fictitious forces, such as the centrifugal force or the Coriolis force. These forces arise due to the acceleration of the reference frame itself and do not have a physical origin.

When studying orbital dynamics, it is often convenient to work in an inertial reference frame, such as the Earth-centered inertial (ECI) frame or the heliocentric inertial frame. These frames provide a fixed reference point from which the motion of celestial objects can be described.

3.1.3. Particle

In mechanics, a point mass or a **particle** is an idealized object that is used to simplify the analysis of motion. It is assumed to have mass but no physical size or internal structure. A point mass can be thought of as a mathematical abstraction that represents the concentrated mass of an object.

By considering an object as a point mass, we can neglect the rotational and deformative effects that would occur if the object had a finite size and shape. This simplification allows us to focus solely on the translational motion of the object, making calculations more manageable.

In reality, most objects have a finite size and shape, and their motion involves both translational and rotational components. However, for many practical applications in mechanics, the point mass assumption is sufficiently accurate.

In orbital dynamics, the concept of a point mass is commonly used to model celestial bodies, such as planets, satellites, or spacecraft. These objects are treated as point masses since their sizes and internal structures are negligible compared to the distances involved in orbital motion.

Kinematics of a point particle

The motion of a point mass can be described by its position vector, which specifies the location of the mass in space with respect to a reference frame. For instance, the position vector of point P with respect to the frame S_0 is noted as $\mathbf{r_0^P}$.

In addition to position, the velocity and acceleration of a point mass are essential in understanding its motion. The velocity vector $\mathbf{v_0^P}$ describes the rate of change of position of point P with respect to time in reference frame S_0. Mathematically, it can be defined as the time derivative of the position vector:

$$\mathbf{v_0^P} = \frac{d\mathbf{r_0^P}}{dt} \,|_0 \tag{3.5}$$

Similarly, the acceleration vector $\mathbf{a_0^P}$ represents the rate of change of velocity $\mathbf{v_0^P}$ with respect to time in in reference frame S_0. It can be obtained by taking the time derivative of the velocity vector:

$$\mathbf{a_0^P} = \left.\frac{d\mathbf{v_0^P}}{dt}\right|_0 \tag{3.6}$$

If instead of a fixed reference frame, we are considering a moving reference frame, the velocity and acceleration of a point mass can be expressed relative to that frame. However, additional terms need to be considered due to the relative motion between the point mass and the frame.

Let's consider the case that appears in figure 3.2., in which we have a fixed reference frame $S_0 : \{O; B_0\}$ and a moving reference frame $S_1 : \{A; B_1\}$. Point P is being observed from the moving reference frame S_1, and the following information is known:

1. The linear motion of the origin A, i.e., its position vector $\mathbf{r_0^A}$, velocity vector $\mathbf{v_0^A}$ and acceleration vector $\mathbf{a_0^A}$.

2. The angular motion of the basis B_1, i.e., the rotation matrix $[{}_0R_1]$, the angular velocity vector ω_{10} and acceleration vector α_{10}.

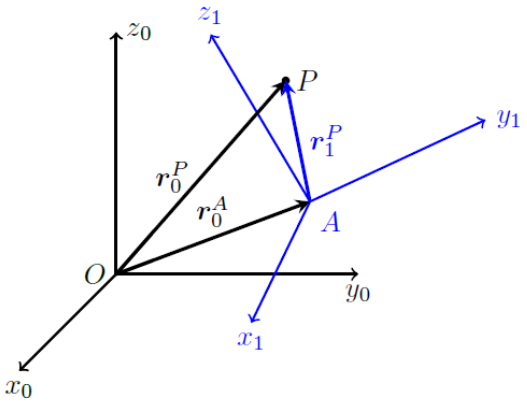


Fig. 3.2. Fixed and moving reference frames for observing the position of a point P. [5]

This case is quite common in orbital dynamics, and it is often convenient to obtain the position, velocity and acceleration vectors with respect to the fixed reference frame (also known as the absolute position, velocity and acceleration vectors). For example, point A could be a ground station on the surface of the Earth, which measures the position of a satellite P in SEZ (South-East-Zenith) frame or in our case S_1, and it is required for some calculations to transform the measured position to an ECI (Equatorial Earth-Centered Inertial) frame, which would be S_0.

Intuitively, the absolute position vector $\mathbf{r_0^P}$ is simply the vector sum of the absolute position vector of A and the relative position vector $\mathbf{r_1^P}$:

$$\mathbf{r_0^P} = \mathbf{OP} = \mathbf{OA} + \mathbf{AP} = \mathbf{r_0^A} + \mathbf{r_1^P} \tag{3.7}$$

To obtain the absolute velocity and absolute acceleration, the derivation is no longer so direct. Now, we need to apply the appropriate relative kinematic relationships that must account for the effect of the angular speed and angular acceleration of basis B_1, and the Coriolis effect. The absolute velocity would then be:

$$\mathbf{v_0^P} = \mathbf{v_0^A} + \mathbf{v_1^P} + \omega_{10} \times \mathbf{r_1^P} \tag{3.8}$$

where $\mathbf{v_0^A}$ is the velocity of the origin A, $\mathbf{v_1^P}$ is the relative velocity , and the term $\omega_{10} \times \mathbf{r_1^P}$ is associated to the rotation of S_1. And the absolute acceleration:

$$\mathbf{a_0^P} = \mathbf{a_0^A} + \mathbf{a_1^P} + \alpha_{10} \times \mathbf{r_1^P} + \omega_{10} \times (\omega_{10} \times \mathbf{r_1^P}) + 2\omega_{10} \times \mathbf{v_1^P} \tag{3.9}$$

where $\mathbf{a_0^A}$ is the acceleration of the origin A, $\mathbf{a_1^P}$ is the relative acceleration, the term $\alpha_{10} \times \mathbf{r_1^P}$ is proportional to the angular acceleration of S_1 (also known as the Euler term), $\omega_{10} \times (\omega_{10} \times \mathbf{r_1^P})$ is the centripetal term, and $2\omega_{10} \times \mathbf{v_1^P}$ is the Coriolis term.

3.2. Orbital Mechanics

Orbital mechanics's main focus is to study the motion of objects in space under the influence of gravitational forces. It provides a framework for understanding and predicting the behavior of celestial bodies, such as satellites, planets, and spacecraft, as they orbit around each other. By employing mathematical models and principles, orbital mechanics allows us to calculate and analyze various aspects of orbital motion, including trajectories, orbital maneuvers, and interplanetary transfers. It plays a crucial role in space exploration, satellite operations, and mission planning, enabling us to navigate and utilize space with precision and efficiency. [7]

3.2.1. Ideal Two-Body Problem

The ideal two-body problem is a simplified model in orbital dynamics that describes the motion of two bodies under their mutual gravitational attraction. These two bodies are modeled as two point masses alone in the universe; this means that other external factors such as atmospheric drag, or the gravitational pull of other celestial bodies are neglected. [8]

In this model, the motion of the two masses can be described by a general equation that considers the gravitational force between them, incorporating variables like mass, distance, and the gravitational constant. Solving this equation allows us to determine essential characteristics of the system, such as the shape, size, and period of the orbit. While the ideal two-body problem overlooks real-world complexities, it provides a valuable starting point for understanding orbital dynamics and serves as a foundation for more advanced analyses and calculations.

The two-body problem equation

The main equation obtained from this model is two-body problem equation, which illustrates the motion of a point particle of mass m^{P2} around another point particle of mass m^{P1}, where $m^{P1} > m^{P2}$.

$$\frac{d^2\mathbf{r}}{dt^2} = -\frac{\mu}{r^3}\mathbf{r} \tag{3.10}$$

where $\mu = Gm$ is the gravitational parameter of the system and $\mathbf{r}$ is the position vector of body P_2 with respect to body P_1. When $m^{P1} >> m^{P2}$, the gravitational parameter is typically approximated as $\mu = Gm^{P1}$.

Conservation of specific mechanical energy

The mass-specific mechanical energy of P_2 in S_1 (the reference frame of P_2) can be defined as:

$$\xi = \frac{v^2}{2} - \frac{\mu}{r} \tag{3.11}$$

Conservation of specific angular momentum

The mass-specific angular momentum vector of P_2 about P_1 in S_1 is:

$$\mathbf{h} = \mathbf{r} \times \mathbf{v} \tag{3.12}$$

Eccentricity vector

The eccentricity vector definition can be seen in equation 3.13. This vector always points towards the orbit's periapsis. Its magnitude e is the eccentricity of the orbit, a key factor in identifying the conic section.

$$\mathbf{e} = \frac{\mathbf{v} \times \mathbf{h}}{\mu} - \frac{\mathbf{r}}{r} = \left(v^2 - \frac{\mu}{r}\right)\frac{\mathbf{r}}{\mu} - \frac{\mathbf{r} \cdot \mathbf{v}}{\mu}\mathbf{v} \tag{3.13}$$

Perifocal reference frame

The perifocal reference frame is a useful coordinate system for analyzing the dynamics of an orbiting object. It simplifies the description of the object's motion, particularly in the orbital plane, by aligning the coordinate axes with important geometric features of the orbit. The perifocal reference frame is defined as $S_1 : \{P_1; B_0\}$, where $B_0 : \{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$.

$$\mathbf{i} = \mathbf{e}/e \tag{3.14}$$

$$\mathbf{i} = \mathbf{k}\mathbf{i} \tag{3.15}$$

$$\mathbf{k} = \mathbf{h}/h \tag{3.16}$$

For circular orbits, as $\mathbf{e} = 0$, any direction in the orbital plane is chosen for vector $\mathbf{i}$. As it can be seen in figure 3.3, the perifocal reference frame is sometimes denoted by the letters PQW, and the unit vectors along the coordinated axes called p, q and w.

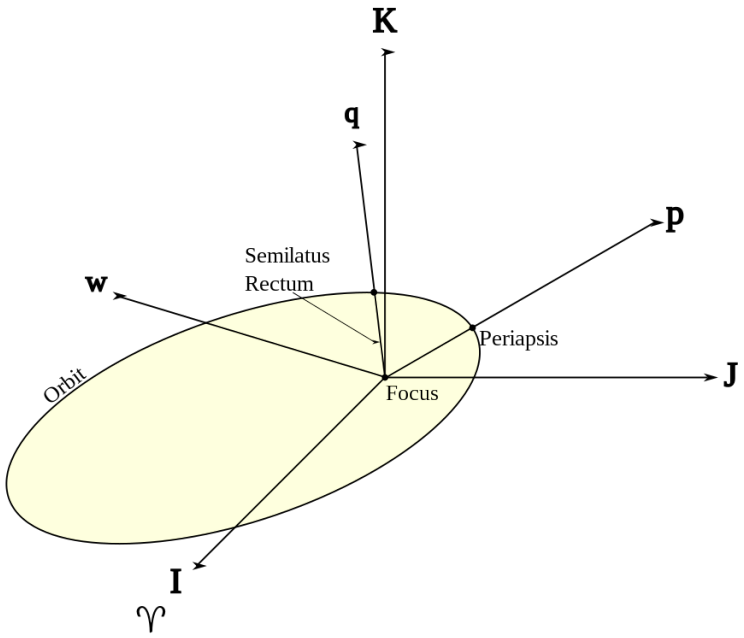


Fig. 3.3. Perifocal Reference Frame [9]

Trajectory equation

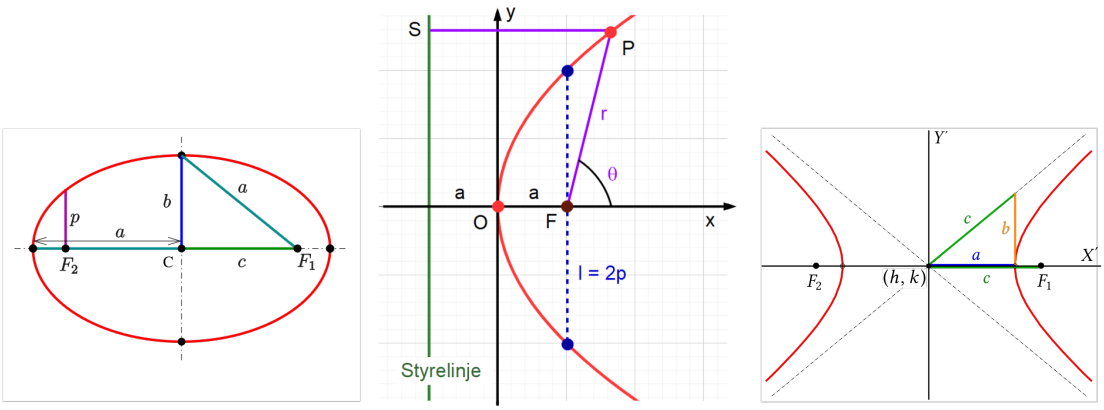
The trajectory equation of of a conic section of orbit parameter $p = h^2/\mu$ and eccentricity **e** is shown in the following expression:

$$r = \frac{h^2/\mu}{1 + e\cos\theta} \tag{3.17}$$

where θ is the true anomaly considered as the polar angle between **e** and **r**.

Conic sections

In the context of the two-body problem, there are three types of non-degenerate conic sections that describe orbital paths: ellipses, parabolas, and hyperbolas. These orbits are commonly referred to as Keplerian orbits.



(a) Ellipse (b) Parabola (c) Hyperbola

Fig. 3.4. Conic sections

The most important parameters characterizing these orbits are the semi-major axis (a), semi-latus rectum (p), and eccentricity (e), the semi-minor axis b and the semi-interfocal distance c.

The following equations provide relationships between these parameters and the periapsis (r_p) and apoapsis (r_a), which are, respectively, the points of the conic section nearest and farthest away from the occupied focus P1.

$$r_p = a(1-e); \qquad r_a = a(1+e); \qquad 2a = r_p + r_a; \qquad e = \frac{r_a - r_p}{r_a + r_p} \tag{3.18}$$

For the parabolic case, $a = \infty$, which makes the equation of the periapsis indeterminate. However, we can use instead the parameter p, which is always defined. The equation for the periapsis would become then: $r_p = p/2$.

Additional relationships between the parameters of a conic section are:

$$a^2 = b^2 + c^2; \qquad b = a\sqrt{1-e^2}; \qquad c = ea; \qquad p = a(1-e^2) = b^2/a \tag{3.19}$$

Furthermore, the semi-latus rectum (p) can be expressed in terms of specific angular momentum (h) and the gravitational parameter (μ). And the semi-major axis (a) can be related to the gravitational parameter (μ) and the mechanical energy (ξ) as:

$$p = \frac{h^2}{\mu}; \qquad a = -\frac{\mu}{2\xi} \tag{3.20}$$

To further understand how these parameters differ in each conic section, table 3.1 can be seen.

Conic section	Semi-major axis	Eccentricity	Specific mechanical energy
Circle	$a > 0$	$e = 0$	$\xi < 0$
Ellipse	$a > 0$	$0 \leq e \leq 1$	$\xi < 0$
Parabola	$a = \infty$	$e = 1$	$\xi = 0$
Hyperbola	$a < 0$	$e > 1$	$\xi > 0$

TABLE 3.1. VALUES OF CONIC SECTIONS PARAMETERS

Orbit types

In the context of the two-body problem, orbital motion can be described by three types of orbits: ellipses, parabolas, and hyperbolas.

1. **Elliptic orbits**. Elliptic trajectories are the only closed orbits in the two-body problem. The eccentric anomaly, denoted as E, is a useful parameter in elliptical orbits. It is defined as the angle between the center of the ellipse (C) and the corresponding point (P') on the auxiliary circle associated with the ellipse. The eccentric anomaly provides a way to uniquely describe the position of a point on the ellipse in terms of an angular measurement. Point P on the ellipse can be defined with its Cartesian coordinates in the perifocal reference frame:

$$x = a(\cos E - e); \qquad y = b \sin E \tag{3.21}$$

The eccentric and true anomalies are related by:

$$\begin{aligned} \sin E &= \tfrac{\sqrt{1-e^2}\sin\theta}{1+e\cos\theta}; \qquad \cos E = \tfrac{e+\cos\theta}{1+e\cos\theta}; \\ \sin\theta &= \tfrac{\sqrt{1-e^2}\sin E}{1-e\cos E}; \qquad \cos\theta = \tfrac{\cos E - e}{1-e\cos E} \end{aligned} \tag{3.22}$$

2. **Parabolic orbits**. Parabolic trajectories occur when the eccentricity of an orbit is exactly equal to 1. In parabolic orbits, the eccentricity is a fundamental parameter that characterizes the shape of the trajectory. Parabolic orbits do not close, meaning that the spacecraft approaches the central body along a specific path without ever returning. The parabolic anomaly, denoted as P, is used to describe the position of a point on the parabolic trajectory in terms of an angular measurement. The parabolic and true anomaly in parabolic orbits are related by: $P = \tan(\theta/2)$. In Cartesian coordinates in the perifocal reference frame, the parabolic trajectory is written as

$$x = \frac{p}{2} - \frac{y^2}{2p} \tag{3.23}$$

3. **Hyperbolic orbits**. In hyperbolic orbits, the spacecraft follows a path that approaches the central body but then escapes to infinity without ever forming a closed loop. Similar to elliptic and parabolic orbits, hyperbolic orbits can also be described

in terms of the hyperbolic anomaly, denoted as H. For a point P on the trajectory, its Cartesian coordinates in the perifocal reference frame can be expressed as:

$$x = |a|(e - \cosh H); \qquad y = |b| \sinh H \tag{3.24}$$

The hyperbolic and true anomalies in hyperbolic orbits are related by:

$$\begin{aligned} \sinh H &= \tfrac{\sqrt{e^2-1}\sin\theta}{1+e\cos\theta}; \qquad \cosh H = \tfrac{e+\cos\theta}{1+e\cos\theta}; \\ \sin\theta &= -\tfrac{\sqrt{e^2-1}\sinh H}{1-e\cosh H}; \qquad \cos\theta = \tfrac{\cosh H - e}{1-e\cosh H} \end{aligned} \tag{3.25}$$

Kepler's laws of orbital motion

These three laws were enunciated by Johannes Kepler (1571–1630) as empirical laws, based on the observations of Tycho Brahe. These laws are fundamental in understanding the behavior of objects in space and have been verified through extensive observations and mathematical analyses. They played a crucial role in the development of celestial mechanics and the understanding of planetary motion.

1. **Kepler's First Law (Law of Ellipses)**: The orbit of a planet is an ellipse, with the Sun at one of the two foci.
2. **Kepler's Second Law (Law of Areas)**: A line segment joining a planet and the Sun sweeps out equal areas during equal intervals of time. This implies that a planet moves faster when it is closer to the Sun (perihelion) and slower when it is farther away (aphelion).
3. **Kepler's Third Law (Law of Harmonies)**: The square of the orbital period of a planet is proportional to the cube of the semi-major axis of its orbit. Mathematically, it can be expressed as $T^2 \propto a^3$, where T is the orbital period and a is the semi-major axis of the orbit. [10]

Vis-viva equation and fundamental velocities

The **vis-viva equation** relates the specific orbital energy of a spacecraft to its distance from the central body and the semi-major axis of its orbit. It is given by:

$$\frac{v^2}{2} - \frac{\mu}{r} = -\frac{\mu}{2a} \tag{3.26}$$

where v is the velocity of the spacecraft, μ is the gravitational parameter of the central body, r is the distance from the central body, and a is the semi-major axis of the orbit.

The **circular velocity** is the velocity required for a spacecraft to maintain a circular orbit at a given distance from the central body. It is calculated using the formula:

$$v_c = \sqrt{\frac{\mu}{r}} \tag{3.27}$$

The **escape velocity** is the minimum velocity required for a spacecraft to escape the gravitational pull of the central body and move away to infinity. It is given by:

$$v_e = \sqrt{\frac{2\mu}{r}} \tag{3.28}$$

The **velocity at periapsis** is the velocity of a spacecraft at its closest point to the central body in an elliptical orbit. It is calculated using the formula:

$$v_p = \sqrt{\frac{\mu}{a}\left(\frac{1+e}{1-e}\right)} \tag{3.29}$$

The **velocity at apoapsis** is the velocity of a spacecraft at its farthest distance from the central body in an elliptical orbit. It is given by:

$$v_a = \sqrt{\frac{\mu}{a}\left(\frac{1-e}{1+e}\right)} \tag{3.30}$$

The **excess hyperbolic velocity** is the velocity of a spacecraft in a hyperbolic orbit above the escape velocity of the central body. It is given by:

$$v_h = \sqrt{-\frac{\mu}{a}} \tag{3.31}$$

Common reference frames

In celestial mechanics and satellite orbit determination, several common reference frames are used to describe the positions and motions of objects. These reference frames provide a standardized coordinate system that simplifies the mathematical representation of orbital dynamics. Here are three commonly used reference frames:

1. **Equatorial Earth-centered inertial reference frame (ECI-equatorial)**: This reference frame is based on an inertial system fixed with respect to the stars and centered at the center of the Earth. It is commonly used for space-based applications and provides a convenient way to describe the orbital motion of satellites. The equatorial Earth-centered inertial reference frame is pseudo-inertial, meaning it approximates an inertial frame for short-term observations. It is typically represented using a Cartesian coordinate system, where the x-axis points toward the vernal equinox, the y-axis lies in the Earth's equatorial plane and completes a right-handed orthogonal system, and the z-axis completes the right-handed system by pointing toward the North Celestial Pole.

2. **Earth-centered Earth-fixed reference frame (ECEF)**: The Earth-centered Earth-fixed reference frame is a non-inertial frame that rotates with the Earth. It provides a

practical way to describe the positions and motions of objects relative to the Earth's surface. In this frame, the origin is fixed at the center of the Earth, and the x, y, and z axes are aligned with the Earth's surface. The x-axis points toward the intersection of the Prime Meridian and the equator (Greenwich), the y-axis lies in the equatorial plane and completes a right-handed orthogonal system, and the z-axis completes the right-handed system by pointing toward the North Pole.

3. **Topocentric reference frame**: The topocentric reference frame is a local reference frame that is centered at a specific observer's location on the Earth's surface. It is particularly useful for ground-based observations, such as tracking satellites or observing celestial objects. In the topocentric frame, the origin is located at the observer's position, and the three axes are defined as follows. The zenith axis (Z) points directly upward along the local vertical, perpendicular to the Earth's surface. The north axis (N) points toward the true north direction. The east axis (E) completes the right-handed system by pointing toward the east direction.

   The topocentric reference frame allows observers to describe the positions and motions of celestial objects or satellites relative to their specific location on the Earth's surface. It is commonly used in applications such as satellite tracking, astronomical observations, and geodetic measurements.

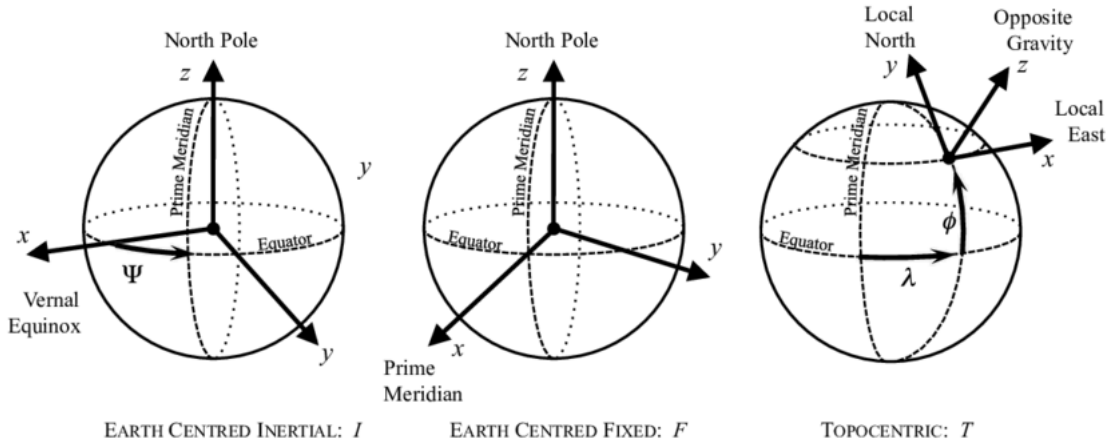


Fig. 3.5. Earth Centered Inertial, Earth Centered Fixed, and Topocentric reference frames [11]

These reference frames, shown in figure 3.5, provide a consistent framework for describing the positions, velocities, and orientations of objects in space and are essential for various applications in celestial mechanics, satellite navigation, and Earth observation.

Classical Orbital Elements

There are two possible ways to define completely the motion of a body in space, this is with either the state vector or with the classical orbital elements. The **state vector** is a

6-dimensional vector that results from the concatenation of its position vector **r** and its velocity vector **v** in a given reference frame:

$$\mathbf{X} = [x, y, z, \dot{x}, \dot{y}, \dot{z}]. \tag{3.32}$$

However, while the state vector contains all the information on the motion of the object, sometimes it is not the preferred representation as the orbit's characteristics are not as intuitive from **X** as they are with the classical orbital elements (COE).

The **classical orbital elements (COE)** provide a concise and convenient way to describe the trajectory of an object in space. Represented for an elliptical orbit in Figure 3.6, the six classical orbital elements are:

1. **Right Ascension of the Ascending Node (RAAN) (Ω):** The angle between a reference direction (e.g., the vernal equinox) and the point where the orbit crosses the reference plane in the ascending direction. It specifies the rotation of the orbital plane around the reference direction. It ranges from 0 to 2π.

2. **Inclination (i):** It is the angle between the orbital plane and a reference plane, such as the Earth's equatorial plane. It defines the tilt of the orbit with respect to the reference plane and determines the orientation of the orbit in space. It ranges from 0 to π.

3. **Argument of Periapsis (ω):** The angle between the ascending node and the periapsis of the orbit. It defines the orientation of the ellipse within the orbital plane. It ranges from 0 to 2π.

4. **Semi-major axis (a):** It is half the length of the major axis of the orbital ellipse. The semi-major axis determines the size of the orbit and is a measure of the average distance between the orbiting object and the central body. The orbital parameter p is sometimes used instead of a, especially for near-parabolic orbits ($e \simeq 1$).

5. **Eccentricity (e):** The eccentricity describes the shape of the orbit. It is a dimensionless parameter that quantifies how much the orbit deviates from a perfect circle. For values of eccentricity between 0 and 1, the orbit is elliptical. A value of 0 represents a circular orbit, while values closer to 1 indicate more elongated elliptical orbits. In the case of parabolic, it is equal to one, and for hyperbolic orbits, the eccentricity is greater than 1.

6. **True Anomaly (θ):** It is the angle between the periapsis and the current position of the orbiting object, measured in the orbital plane. It provides the position of the object along its orbit at a given time. It ranges from 0 to 2π.

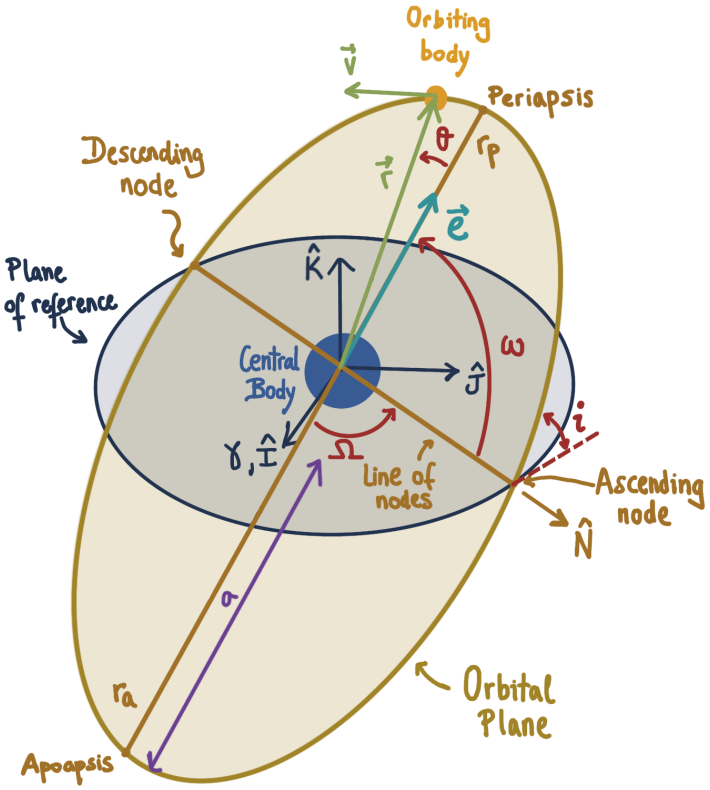


Fig. 3.6. Classical Orbital Elements represented in an Elliptic orbit

In orbital mechanics, it is essential to be able to convert between the classical orbital elements and the state vector (position and velocity) to perform various calculations and simulations.

Kepler's equation

The relation between time t and position (r, θ) can be found through integration,

$$h dt = r^2 d\theta \tag{3.33}$$

$$\int_{t_p}^{t} \frac{\mu^2}{h^3} dt = \int_0^{\theta} \frac{d\theta}{(1 + e\cos\theta)^2} \tag{3.34}$$

This integral is solved separately for $e < 1$, $e = 1$, and $e > 1$ cases. The results can be complicated, so the easiest path is to change the true anomaly θ into the auxiliary anomalies (E, P, H) for each case. Another option is to use the universal formulation for the three conic sections, but it requires more advanced functions.

The equations that relate the true anomaly to the auxiliary anomalies are known as Kepler's equations. When the anomaly is known and we need to find the time, the computation is straightforward. However, these equations are nonlinear, so when the time

is known and we want to find the auxiliary anomaly, it is necessary to use an iterative numerical method, like the Newton-Raphson method.

These are the Kepler's equations for each case:

1. **Elliptic case** ($e < 1$):
$$M_e = E - e \sin E \tag{3.35}$$
where E is the eccentric anomaly and M_e is the mean anomaly in the elliptic case. $M_e = n_e(t - t_0)$, with $n_e = \sqrt{\mu/a^3}$ being the corresponding mean angular rate, and t_0 the time of passage by the periapsis.

2. **Parabolic case** ($e = 1$):
$$M_p = \frac{1}{2}\tan\frac{\theta}{2} + \frac{1}{6}\tan^3\frac{\theta}{2} \tag{3.36}$$
where M_p is the mean anomaly in the parabolic case. $M_p = n_p(t - t_0)$, with $n_p = \sqrt{\mu/p^3}$ being the corresponding mean angular rate, and t_0 the time of passage by the periapsis.

3. **Hyperbolic case** ($e > 1$):
$$M_h = e \sin H - H \tag{3.37}$$
where H is the hyperbolic anomaly and M_h is the mean anomaly in the hyperbolic case. $M_h = n_h(t - t_0)$, with $n_p = \sqrt{-\mu/a^3}$ being the corresponding mean angular rate, and t_0 the time of passage by the periapsis.

As previously mentioned, there is also an **universal formulation** of Kepler's equation done in terms of the universal anomaly χ, which has units of $[L^{1/2}]$.

$$\sqrt{\mu}(t - t_0) = e\chi^3 S\left(\frac{\chi^2}{a}\right) + a(1 - e)\chi \tag{3.38}$$

where $S(z)$ is a Stumpff function:

$$S(z) = \begin{cases} \frac{\sqrt{z} - \sin\sqrt{z}}{\sqrt{z^3}} & (z > 0) \\ \frac{1}{6} & (z = 0) \\ \frac{\sin\sqrt{-z} - \sqrt{-z}}{\sqrt{-z^3}} & (z < 0) \end{cases} \tag{3.39}$$

Depending on the sign of $z = \chi^2/a$, the equation simply reduces to the elliptic, parabolic and hyperbolic cases. Being the relationship between the universal anomaly and the auxiliary anomalies the following:

$$\chi = \begin{cases} \sqrt{a}E & \text{(ellipse)} \\ \sqrt{p}P & \text{(parabola)} \\ \sqrt{-a}H & \text{(hyperbola)} \end{cases} \tag{3.40}$$

Lambert's Problem

The main focus of Lambert's problem is to determine the trajectory, specifically the position and velocity vectors, for a spacecraft to travel between two given points in space within a specified time, assuming only gravitational forces are at play. It is crucial for mission planning and involves solving transcendental equations to find the optimal transfer orbit. Lambert's problem enables precise trajectory calculations for space exploration and satellite missions.

It is important to understand that given two position vectors $\mathbf{r_1}$ and $\mathbf{r_2}$, there is a one-parameter family of conic sections that passes by them. But only one of these conics is the solution, which will be the one for which the time of flight between the positions is exactly Δt we are looking for.

Many different algorithms have been created and continue to be developed in order to solve the Lambert's problem.

3.2.2. Orbital Maneuvering

The two main drivers in any space mission are mass and time. Both of these parameters are strongly dependent on the capabilities and type of rocket chosen for the mission. The Tsiolkolvsky's rocket equation (3.41) relates the propellant mass consumption with the total change in velocity Δv (read as *delta vee*) of the maneuver and specific impulse of the rockets I_{sp}:

$$\Delta v = g_0 I_{sp} \ln \frac{m_0^P}{m_f^P} \tag{3.41}$$

where m_0^P and m_f^P are the initial and final mass of the spacecraft. As it can be deduced from this equation, the higher the specific impulse, the lower the total propellant mass consumption will be for a given space maneuver. Also, it is key to minimize the Δv of the mission by making the maneuver in the appropriate location in order to save mass and money. However, in some case time can be a higher priority than the economic cost, as it is the case in manned missions.

Nowadays, rockets can be divided into two main categories: chemical propulsion and electric propulsion. Chemical propulsion systems provide high thrust but have relatively low specific impulse. On the other hand, electric propulsion systems offer low thrust but have high specific impulse. Performing space maneuvers using these propulsion systems requires different analysis and planning techniques.

High-thrust maneuvers, such as a launcher injection in orbit, are done with chemical rockets because as their name suggests involve applying a high amount of thrust in a short period of time; they are modeled as **impulsive maneuvers**. In contrast, **low-thrust maneuvers** involve continuous, sustained thrust over an extended period of time and are

done using electric propulsion. They are used for tasks like station keeping in geostationary orbit (GEO) or gradual orbit raising.

Impulsive Maneuvers

Impulsive maneuvers are high thrust maneuvers with short burn times much shorter than the orbital period. In these maneuvers, the spacecraft experiences an instantaneous change in velocity without any change in its position. Two commonly used impulsive maneuvers are the Hohmann transfer and plane change maneuvers.

1. The **Hohmann transfer** is a common orbital maneuver used to transfer a spacecraft between two circular orbits. It involves two impulsive burns: the departure burn and the arrival burn. The transfer orbit followed by the spacecraft is an ellipse known as the transfer ellipse.

   The transfer ellipse has a radius at pericenter $r_p = r_A$, a radius at apocenter $r_a = r_C$, and a semi-major axis of $a_t = (r_A + r_C)/2$. The delta-v required for the Hohmann transfer at each firing can be computed using the vis-viva equation.

   The first burn (Δv_1) is performed at the initial circular orbit to transfer the spacecraft to the pericenter of the transfer ellipse. The delta-v required for this burn is given by:

   $$\Delta v_1 = v_{tp} - v_A = \sqrt{\frac{2\mu}{r_A} - \frac{\mu}{r_A + r_C}} - \sqrt{\frac{\mu}{r_A}} \tag{3.42}$$

   where v_{tp} is the velocity at pericenter of the transfer ellipse, v_A is the circular velocity of the initial orbit, r_A is the radius of the initial circular orbit, and r_C is the radius of final circular orbit.

   The second burn Δv_2 is performed at the apocenter of the transfer ellipse to circularize the orbit at the desired altitude. The delta-v required for this burn is given by:

   $$\Delta v_2 = v_C - v_{ta} = \sqrt{\frac{\mu}{r_C}} - \sqrt{\frac{2\mu}{r_C} - \frac{\mu}{r_A + r_C}} \tag{3.43}$$

   where v_C is the circular velocity at the final orbit, v_{ta} is the velocity at apocenter of the transfer ellipse, and r_C is the radius of the transfer ellipse at apocenter.

   The Δv_{total} required for the Hohmann transfer is then simply the sum of Δv_1 and Δv_2.

   The Hohmann transfer is widely used for various space missions, including interplanetary transfers and satellite launches, due to its energy efficiency and relatively low delta-v requirements.

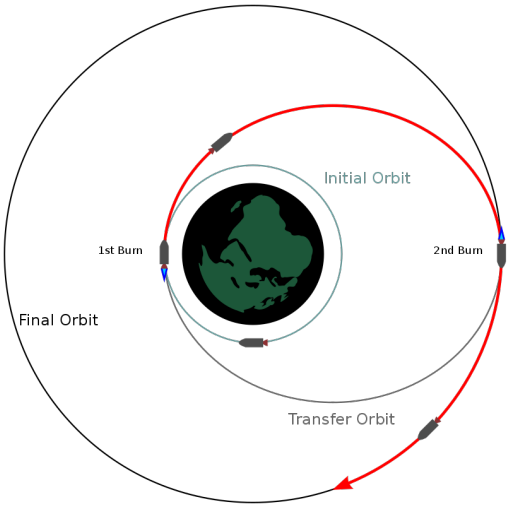


Fig. 3.7. Orbital Hohmann Transfer [12]

2. A **Plane Change Maneuver** is performed to change the inclination of an orbit or to shift the orbital plane to a different one. This maneuver requires a large change in velocity perpendicular to the orbit plane.

   The velocity change required for a plane change maneuver between two circular orbits with equal radius can be calculated using the following equation:

$$\Delta V = 2v \sin\left(\frac{\Delta i}{2}\right) \tag{3.44}$$

   where v is the spacecraft's velocity before the maneuver, and Δi is the desired change in inclination.

   Plane change maneuvers are often less efficient in terms of required delta-v compared to other maneuvers. Therefore, it is usually beneficial to perform them at specific points in the orbit, such as the ascending or descending node, to take advantage of the relative velocity between the spacecraft and the central body.

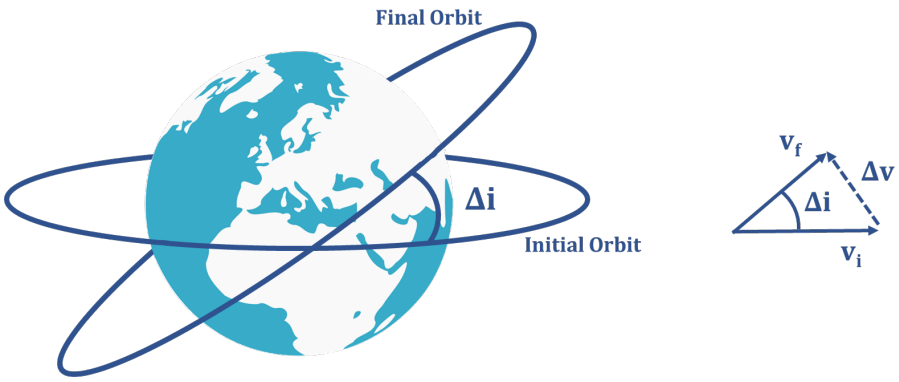


Fig. 3.8. Plane Change Maneuver

Low-thrust Maneuvers

Low-thrust maneuvers involve continuous application of a small thrust force over an extended period, deviating from the impulsive model. They utilize technologies like electric propulsion systems and ion thrusters for efficient propellant usage. Planning and executing these maneuvers require specialized techniques due to factors such as long duration, varying thrust levels, and complex gravitational interactions.

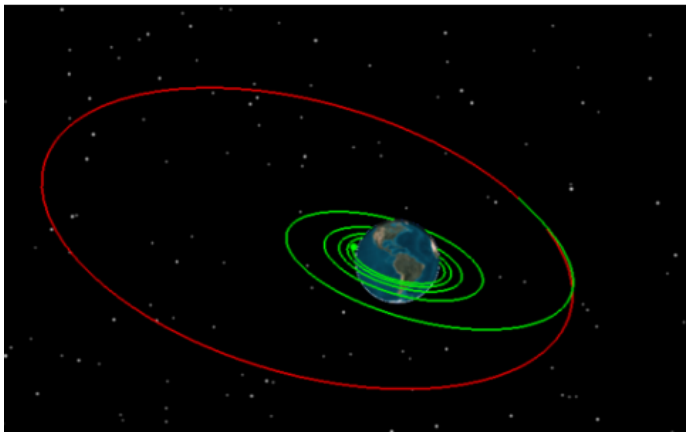


Fig. 3.9. Low Thrust Maneuver [13]

1. **Low-thrust orbit raising** is a maneuver technique that utilizes continuous low-thrust propulsion systems, such as electric propulsion or ion thrusters, to gradually increase the orbit of a spacecraft. Unlike impulsive maneuvers, low-thrust maneuvers have longer burn times and provide a gradual change in velocity. By applying a continuous low-thrust force along the direction of motion, the spacecraft gradually raises its orbit over an extended period.

   For a low-thrust orbit raising between two circular and coplanar orbits where $r_A < r_C$, an analytical approximation to obtain the delta-v can is:

   $$\Delta v = v_A - v_C = \sqrt{\mu/r_A} - \sqrt{\mu/r_C} \tag{3.45}$$

2. **Low-thrust Plane Change Maneuver** is used to change the orbital plane of a spacecraft. By applying a continuous low-thrust force perpendicular to the velocity vector, the spacecraft gradually alters its orbital plane. This maneuver is particularly useful for missions requiring precise orbital alignment or for spacecraft that need to reach specific regions of interest.

   To approximate the inclination change using a low-thrust maneuver from a circular orbit to another circular orbit with the same radius, you can use the following formula:

   $$\Delta v \approx \frac{v_{\text{circ}}}{\Delta i} \tag{3.46}$$

   where Δi is the approximate inclination change, Δv is the delta-v required for the maneuver, and v_{circ} is the circular velocity of the initial orbit.

3.2.3. Orbital Perturbations

To obtain a more realistic model of the motion of two bodies in space, perturbations are added to the ideal two-body problem. These perturbations are the factors that were neglected before in the ideal case, which are the presence of other bodies, the atmospheric drag, the solar radiation pressure, or the fact that the two bodies are not point particles, and that in reality can have a considerable effect on the body's motion. Here, we will focus on the two more commonly used for the case of satellites orbiting around the Earth.

Non-spherical Earth gravity field

The perturbation due to the Earth's oblateness, represented by the J2 term, introduces an acceleration component. This perturbation accounts for the deviation from a perfect sphere and is expressed in terms of the radial unit vector $\mathbf{u_r}$ and the vertical unit vector $\mathbf{k_0}$. It is determined by the parameters J_2, μ_E, and R_E, which represent the Earth's second dynamic form factor, gravitational parameter, and equatorial radius, respectively.

$$\mathbf{a_{p,J2}} = \frac{3}{2} J_2 \frac{\mu_E R_E^2}{r^4} \left[\left(5\frac{z^2}{r^2} - 1 \right) \mathbf{u_r} - 2\frac{z}{r}\mathbf{k_0} \right] \tag{3.47}$$

Atmospheric drag

Another significant perturbation is atmospheric drag. It arises from the interaction between the spacecraft and the Earth's atmosphere.

$$\mathbf{a_{p,Drag}} = -\frac{1}{2B_c}\rho V \mathbf{V} \tag{3.48}$$

with B_c being the ballistic coefficient, and where $\mathbf{V}$ is the velocity of the spacecraft relative to the air. But generally, the velocity of the atmosphere in the ECI reference frame can be neglected, so $\mathbf{V} \simeq \mathbf{v}$. In addition, at high altitudes the atmosphere is nearly-isothermal, and the density ρ can be modeled as such:

$$\rho = \rho exp\left(-\frac{\zeta - \zeta_0}{H} \right), \tag{3.49}$$

where ρ_0 is the density at the reference height ζ_0, and H is the scale height.

4. DEVELOPMENT AND IMPLEMENTATION

4.1. Functional Requirements

The main objective of this Astrodynamics library is to provide a set of functionalities that can assist undergraduate students in their course of orbital dynamics. The code will be designed to offer the following features:

- **Solving Problems of the Ideal Two-Body Problem**: The code will implement numerical algorithms to solve problems related to the Ideal Two-Body Problem. This will involve modeling the motion of a spacecraft under the influence of gravitational forces from a central body. Students will be able to compute and analyze the spacecraft's position and velocity at any given time, facilitating the study of various orbital phenomena and the prediction of future orbits.
- **Change of Reference Frame**: The code will support the transformation of orbital elements and state vectors between different reference frames, such as inertial frames, rotating frames, and non-inertial frames. Students will be able to analyze orbits from different perspectives and understand the impact of reference frame choice on orbital parameters and dynamics.
- **Orbital Maneuvers and Delta-v Computation**: The code will provide functions to compute common orbital maneuvers, such as Hohmann transfers and inclination changes. Students will be able to input the initial and target orbits, and the code will calculate the required delta-v (change in velocity) for executing the maneuvers. This will aid in understanding the principles behind orbital maneuvers and help students gain practical experience in mission planning.
- **Orbital Perturbations**: The code will incorporate models for common orbital perturbations such as atmospheric drag. Students will have the option to include these perturbations in their simulations and observe the effects on spacecraft trajectories. This will enhance their understanding of the challenges and complexities associated with real-world orbital dynamics.
- **Orbit Visualization and Plotting**: The code will include functionality to plot and visualize orbits in two or three dimensions. Students will be able to visualize spacecraft trajectories, observe the effects of different orbital parameters, and analyze orbital elements such as semimajor axis, eccentricity, inclination, and argument of periapsis. These visualizations will enhance the understanding of orbital dynamics concepts and aid in problem-solving exercises.

- **Educational Resources and Examples**: The code will provide educational resources, including sample problems and examples, to help students reinforce their understanding of orbital dynamics concepts. These resources will cover a range of topics such as Kepler's laws, orbital elements, vis-viva equation, and orbital transfers. Students will be able to run code snippets and modify parameters to explore different scenarios and observe the outcomes.

By offering these functionalities, this library aims to serve as a valuable educational tool, supporting undergraduate students in their learning journey of orbital dynamics and providing them with hands-on experience in analyzing and solving problems in this field.

4.2. Code Structure

The Julia package developed can be found in the online repository of GitHub by the name of **AstrodynamicsEdu.jl** which can be accessed clicking on the following link: `https://github.com/AliciaSBa/AstrodynamicsEdu.jl`. It is comprised of multiple files and folders, which include the code, tests, README, and other documents.

The **code** itself can be found inside the *src* folder in the GitHub package, we can find:

- The main module: AstrodynamicsEdu.jl
- Six different files:
    1. linearAlgebraTypes.jl
    2. idealTwoBodyProblem.jl
    3. orbitalPlots.jl
    4. astroConstants.jl
    5. orbitalManeuvers.jl
    6. orbitalPerturbations.jl

Whereas, the **tests** can be found inside the *test* folder and it is subdivided into different files, one for each file that needs to be tested: *test_LinearAlgebraTypes.jl*, *test_IdealTwoBodyProblem.jl*, *test_OrbitalManeuvers.jl* and *test_OrbitalPerturbations.jl*; and an additional one called *runtests.jl* whose function is to run all the test files.

Going back to the structure of the code, the main module *AstrodynamicsEdu.jl* is the Julia module that gathers all the files so that the user only needs to call the package by writing `using AstrodynamicsEdu` and thus will gain access to all the functionalities developed across all the files. Some of these functionalities include plots, solvers, as well as, a list of relevant constants for Orbital Dynamics problems. We will talk thoroughly about the development of each of these files and their capabilities in the next subsections.

Each file follows a general structure. First, all the necessary packages are called by writing the command `using PackageName`. These dependencies are included in the Project.toml and are downloaded automatically when adding the package, such as LinearAlgebra.jl, Plots.jl or DifferentialEquations.jl, among others. Next, there appears the 'export' command followed by all the names of the functions that we want to export outside of the module so that they can be used when calling our library. Then, finally all the functions definitions, types and constructors are defined.

All the functions follow too a consistent structure, so that they the code is legible and easy to an understand for anyone. Above each function its functionality is commented, and inside each function the first thing that can be found are all the inputs and outputs and their meaning. As an example, the definition of the `trajectoryEquation` function from the idealTwoBodyProblem.jl file can be seen in figure 4.1.

```julia
# Calculate the trajectory equation (r = h^2/mu/(1+e*cos(theta))
function trajectoryEquation(stateVector::MyStateVector,mu::Float64,theta::Float64)
    # Input
        # stateVector:: state vector object
        # mu: gravitational parameter
        # theta: true anomaly
    # Output
        # trajectoryEquation: trajectory equation
    e = eccentricityVector(stateVector,mu)
    h = angularMomentumVector(stateVector)
    trajectoryEquation = norm(h)^2/(mu*(1+norm(e)*cos(theta)))
    return trajectoryEquation
end
```

Fig. 4.1. Functions code structure

Code dependencies

As mentioned above, the AstrodynamicsEdu library is dependent on other Julia packages. These packages are included in the Project.toml and are downloaded automatatically with the package. The packages dependencies are the following:

- LinearAlgebra.jl [14]
- Test.jl [15]
- Plots.jl [16]
- Unitful.jl [17]
- Roots.jl [18]
- DifferentialEquations.jl [19]

4.3. Linear Algebra Types

In order to be able to perform operations more at ease and to store information easily, different linear algebra types were created. Their main feature is that through different constructors the user can pass the components they desire and these will always be stored in the Canonical Reference Frame or Basis. Having all vectors, points, vector bases, and reference frame components stored in only one basis/reference frame allows to speed the computation and to reduce the amount of possible errors.

Figure 4.2 shows the five different types defined in the linear algebra module. Below each type, it can be seen the objects they store and the type of each of them (indicated as *fieldName::Type*). To access the field value inside the object it is only necessary to write the name of the variable followed by the fieldName. For example, if 'B1' is MyBasis object and I want to access its angular velocity, I would just have to type 'B1.omega'.

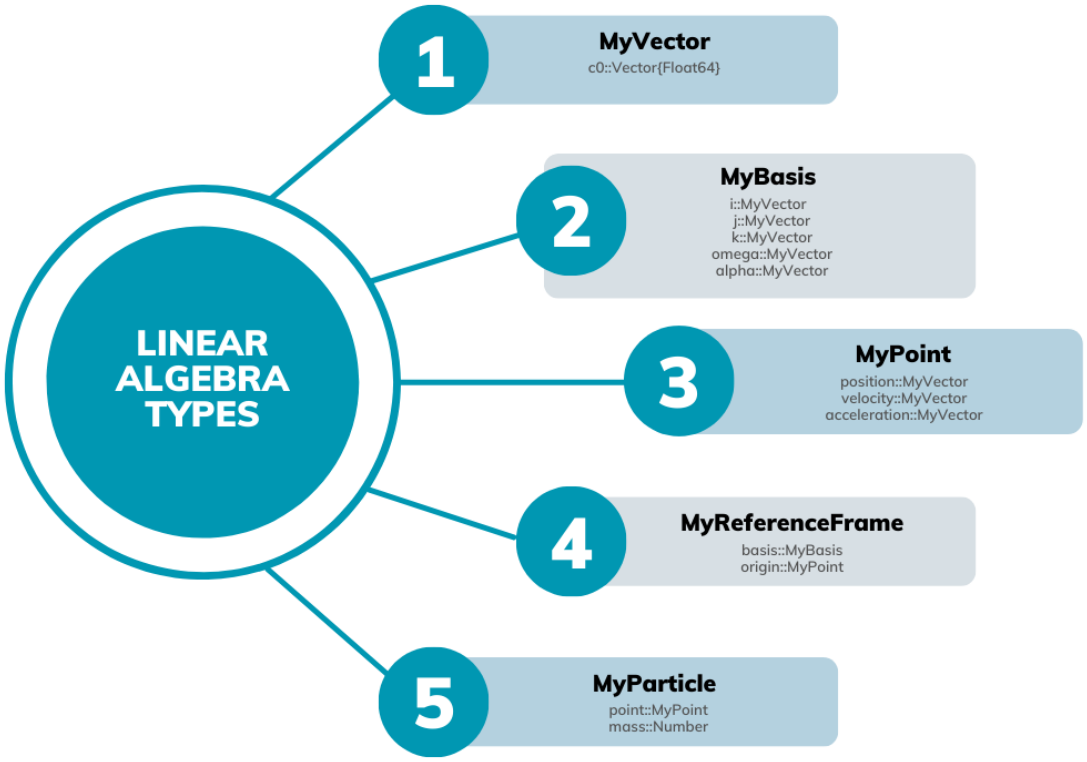


Fig. 4.2. Linear Algebra Types

All these linear algebra types are related among each other, that is why it was so important to get the chain of dependency between them correctly. This is where **constructors** come in. The Julia language allows to have more than one way to build a type/struct. Thanks to this feature, the user can have a vector v_1 expressed in vector basis B_1, and if both v_1 and B_1 (which must be a MyBasis, this means, it must be expressed with respect to the Canonical Basis) are passed, v_1 can be turned into a MyVector. This means simply

that v_1 is stored internally with the coordinates with respect to the Canonical Basis. So, now, we will examine which are the different available constructors for each of the types.

1. **MyVector**: stores a vector of length three in the the canonical right-handed vector basis B_0. These are its possible constructors:

   - **MyVector(c0)**, where c0 is a Vector{Float64} of length = 3 already expressed in the Canonical Basis.
   - **MyVector(c1,B1)**, where c1 is a Vector{Float64} expressed in vector basis B1, which is a MyBasis object.
   - **MyVector(x0,y0,z0)**, where x0, y0, and z0 are the vector components (numbers) already expressed in the Canonical Basis.
   - **MyVector(x1,y1,z1,B1)**, where x1, y1, and z1 are the vector components (numbers) expressed in vector basis B1, which is a MyBasis object.
   - **MyVector()**, which simply builds a MyVector object full of zeros.

2. **MyBasis**: defines a vector basis (i.e. B_1) by storing its unit vectors, angular velocity and angular acceleration with respect to the with respect to the canonical basis B_0. These are the possible constructors.

   - **MyBasis(i1_0, j1_0, k1_0, omega1_0, alpha1_0)**, where all are MyVector objects, i1_0, j1_0, k1_0 are the unit vectors of vector basis B1, and omega1_0, and alpha1_0 are the angular velocity and angular acceleration, respectively.
   - **MyBasis(i2_1, j2_1, k2_1, omega2_1, alpha2_1, B1)**, where all are Vector{Float64} which express the vector basis B2, with respect to the vector basis B1, which is a MyBasis object.
   - **MyBasis()**, which builds the Canonical Basis.

3. **MyPoint**: stores the position, velocity, and acceleration vectors that define a point with respect to the inertial reference frame S_0. These are the possible constructors:

   - **MyPoint(pos, veloc, accel**, where pos, veloc, and accel are the position, velocity and acceleration vector which are already MyVectors objects as they expressed with respect to the Canonical Reference Frame.
   - **MyPoint(pos1, veloc1, accel1, RF1)**, where pos1, veloc1, and accel1 are Vector{Float64} expressed with respect to the reference frame RF1, which is a MyReferenceFrame object.
   - **MyPoint()**, which builds the Canonical Origin Point, this is just an empty point in the Canonical Reference Frame.

4. **MyReferenceFrame**: defined by a vector basis (i.e. B_1) which is MyBasis object and a point (i.e P_1) which is a MyPoint object. These are the possible constructors:

- **MyReferenceFrame(basis, origin)**, where basis is a MyBasis object and origin is a MyPoint object.
- **MyReferenceFrame()**, which returns the inertial reference frame, whose basis is the Canonical Basis and origin point is the Canonical Origin Point.

5. **MyParticle**: object that defines the point particle by storing the point, which is a MyPoint object, and its associated mass. This is the constructor:
   - **MyParticle(point, mass)**, where point is a MyPoint object and mass is just a number.

Some useful objects have been also predefined, so that the user can simply call them every time they are needed, allowing therefore to save time. These **predefined objects** are the canonical vector basis B_0, the three canonical unitary vectors ($\mathbf{i_0}, \mathbf{j_0}, \mathbf{k_0}$), the canonical origin point O_0, and the canonical/inertial reference frame S_0. They can simply be used by typing their correct name, which appear highlighted below:

- **CanonicalBasis** = MyBasis(MyVector([1.0, 0.0, 0.0]), MyVector([0.0, 1.0, 0.0]), MyVector([0.0, 0.0, 1.0]), MyVector([0.0, 0.0, 0.0]), MyVector([0.0, 0.0, 0.0]))
- **i0** = MyVector([1.0, 0.0, 0.0])
- **j0** = MyVector([0.0, 1.0, 0.0])
- **k0** = MyVector([0.0, 0.0, 1.0])
- **CanonicalOriginPoint** = MyPoint(MyVector([0.0, 0.0, 0.0]), MyVector([0.0, 0.0, 0.0]), MyVector([0.0, 0.0, 0.0]))
- **CanonicalReferenceFrame** = MyReferenceFrame(MyBasis(MyVector([1.0, 0.0, 0.0]), MyVector([0.0, 1.0, 0.0]), MyVector([0.0, 0.0, 1.0]), MyVector([0.0, 0.0, 0.0]), MyVector([0.0, 0.0, 0.0])), MyPoint(MyVector([0.0, 0.0, 0.0]), MyVector([0.0, 0.0, 0.0]), MyVector([0.0, 0.0, 0.0])))

Both recipes and functions have been implemented to allow plotting MyBasis and MyReferenceFrame objects in 3D. The only difference in their usage is that for the recipes it is required to include again the Plots.jl package, by writing before:*using Plots*. The proper way to call this functions or recipes is addressed in table 4.1.

Function call	**Recipe call**	**Functionality**
plot_MyBasis(B1)	plot(B1)	Plot the MyBasis object B_1 in 3D using either the function or the recipe
plot_MyReferenceFrame(RF1)	plot(RF1)	Plot the MyReferenceFrame object S_1 in 3D using either the function or the recipe

Table 4.1 continued from previous page

Function call	Recipe call	Functionality

TABLE 4.1. LINEAR ALGEBRA TYPES PLOT FUNCTIONS

The plots of the Canonical Vector Basis in figure 4.3 and the Canonical Reference Frame in figure 4.4 have been plotted using the functions mentioned above.

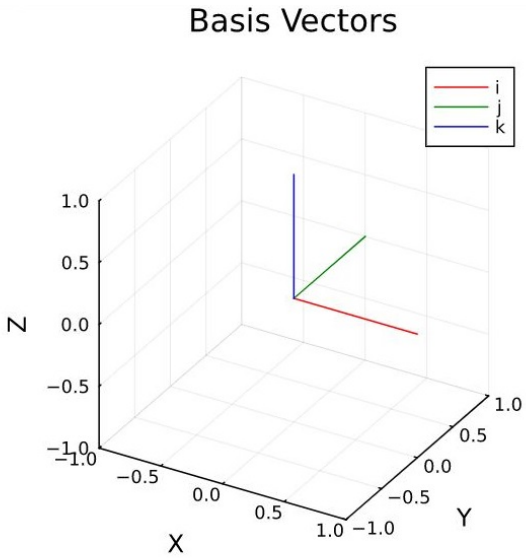


Fig. 4.3. Canonical Vector Basis Plot

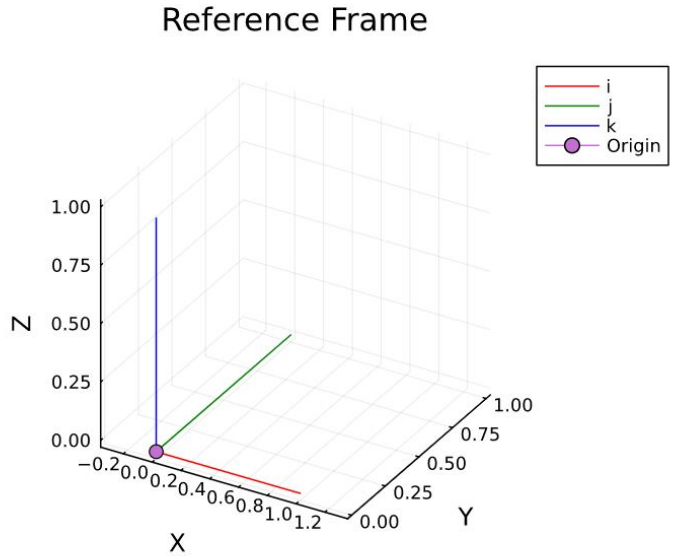


Fig. 4.4. Canonical Reference Frame Plot

Additional functions: rotations, project in basis, change of reference frame

Function Name	Inputs	Outputs	Functionality
componentsInBasis	v::MyVector B1::MyBasis	v1::Vector	Project a MyVector object onto a basis B_1
pos_vel_acc_inRF	point::MyPoint RF1::MyReferenceFrame	pos1::Vector veloc1::Vector accel1::Vector	Obtain the position, velocity, and acceleration vectors of a MyPoint object in a different reference frame S_1
rotation_matrix	basis::MyBasis	R::Matrix	Obtain the Rotation Matrix $[{}_0R_1]$ of a MyBasis object B_1
pure_rotation_MyBasis_x	basis::MyBasis theta::Number	rotB::MyBasis	Obtain the new MyBasis object after a pure rotation about the x-axis
pure_rotation_MyBasis_y	basis::MyBasis theta::Number	rotB::MyBasis	Obtain the new MyBasis object after a pure rotation about the y-axis
pure_rotation_MyBasis_z	basis::MyBasis theta::Number	rotB::MyBasis	Obtain the new MyBasis object after a pure rotation about the z-axis

TABLE 4.2. LINEAR ALGEBRA FUNCTIONS

4.4. Ideal Two-Body Problem

The study of the ideal two-body problem forms the core of our codebase, serving as the foundation for more complex algorithms and calculations in orbital dynamics. This module introduces two key types: the state vector and the coe (classical orbital elements); and provides a comprehensive set of functions that enable precise analysis and prediction of celestial motion.

The ideal two-body problem focuses on the motion of two point masses, assuming no external influences from other celestial bodies. Although this simplification may appear trivial, it serves as a fundamental building block for understanding and modeling more intricate orbital dynamics scenarios.

Within the idealTwoBodyProblem.jl file, a range of functions has been developed to address various aspects of the ideal two-body problem. These functions form the backbone of orbital computations, providing essential tools for the calculation of critical parameters and transformations.

Figure 4.5 shows the two new data types defined in the ideal two-body problem file: MyStateVector and MyCOE. Below each appears their store objects and their type (indicated as *fieldName::Type*). These two data types are crucial for astrodynamics, as they allow to completely describe the motion of a body in space.

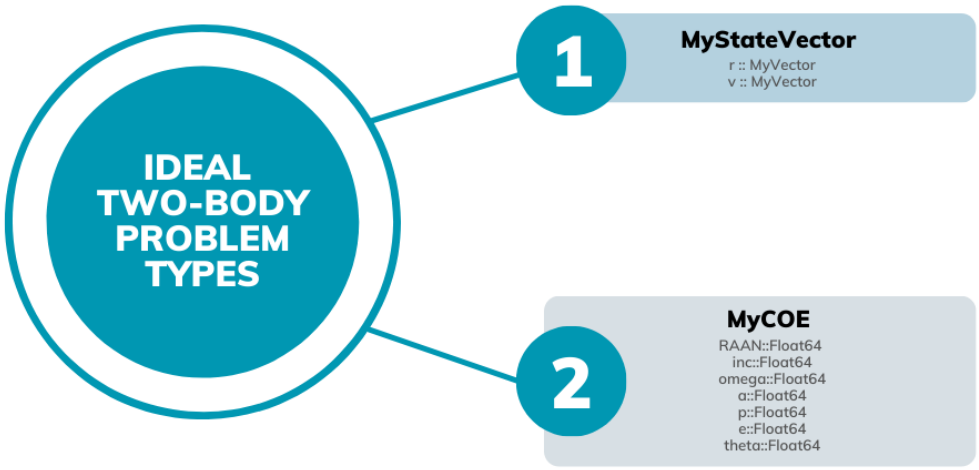


Fig. 4.5. Ideal Two-Body Problem Types

It is important to understand what each of these types entails and how to construct them:

1. **MyStateVector**: stores the position vector (**r**) and the velocity vector (**v**) of a body in space. Although there is not a general imposition on the units, because of the nature of distances in Space, the position of the satellite/spacecraft is usually given in $[km]$, while the velocity is given in $[km/s]$. Providing the coordinates in these units ensures the correct functioning of all the functions in the code. There is only one possible way to construct a MyStateVector object, and it is as follows:

   - **MyStateVector(r,v)**, where both r and v must be MyVector objects.

2. **MyCOE**: stores all the six classical orbital elements($\Omega, i, \omega, a, e, \theta$), and p, so that the orbit can always be well defined. This is because of the peculiarity of the parabolic orbit, where $a = \infty$. However, for the construction the seven elements are not given, only six of them, where the user can choose to either provide the semi-major axis a or the semi-latus rectum p, but never both. However, this seventh element can always be accessed as it is computed internally. Because of this distinction, the MyCOE type is the only one that requires to specify the name of the input followed by its value or stored variable. It is very important to note that all the angles must be given in $[rad]$, this is not optional. In addition, all the parameters must be Float64. An example for each constructor is provided below:

   - **MyCOE(RAAN = 0.18, inc= 0.26, omega= 0.30, a= 12000.0, e= 0.54, theta = pi)**, where a is provided, and p is computed internally.

   - **MyCOE(RAAN = 0.52, inc= 0.0, omega= 0.92, p= 8000.0, e= 1.0, theta = 2.09)**, where p is provided, and a is computed internally.

Table 4.3 presents an overview of the functions available within the ideal two-body problem file. These functions encompass a wide range of functionalities, including the computation of the angular momentum vector, the conversion between state vectors and classical orbital elements, and the propagation of state vectors over time. With a total of 26 functions, this comprehensive set ensures that users have the necessary tools to accurately analyze and simulate the motion of celestial objects.

Function Name	**Inputs**	**Outputs**	**Functionality**
angularMomentumVector	stateVector::MyStateVector	h::MyVector	Obtain **h** vector
mechanicalEnergy	stateVector::MyStateVector mu::Float64	xi::Float64	Obtain ξ
eccentricityVector	stateVector::MyStateVector mu::Float64	e::MyVector	Obtain **e** vector
semilatusRectum	stateVector::MyStateVector mu::Float64	p::Float64	Obtain p
periodOrbit	stateVector::MyStateVector mu::Float64 OrbitType::String	period::Float64	Obtain period τ [s]
perifocalBasis	stateVector::MyStateVector mu::Float64	perifocalBasis ::MyBasis	Perifocal Basis
periapsis	stateVector::MyStateVector mu::Float64	r_p::Float64	Obtain r_p
apoapsis	stateVector::MyStateVector mu::Float64	r_a::Float64	Obtain r_a
trajectoryEquation	stateVector::MyStateVector mu::Float64 theta::Float64	trajectoryEquation ::Float64	Obtain r
inclination	stateVector::MyStateVector	inc::Float64	Obtain i
semiMajorAxis	stateVector::MyStateVector mu::Float64	a::Float64	Obtain a
orbitType	stateVector::MyStateVector mu::Float64	orbitType::String	Obtain orbit type: - Elliptic - Hyperbolic - Parabolic - Circular
trueAnomaly	stateVector::MyStateVector mu::Float64	theta::Float64	Obtain θ [rad]
stateVector_to_COE	stateVector::MyStateVector mu::Float64	coe::MyCOE	$(\mathbf{r}, \mathbf{v}) \rightarrow$ COE
COE_to_stateVector	coe::MyCOE mu::Float64	stateVector ::MyStateVector	COE $\rightarrow (\mathbf{r}, \mathbf{v})$
escapeVelocity	r::MyVector mu::Float64	v_esc::Float64	Obtain v_e
circularVelocity	2 OPTIONS: r::MyVector mu::Float64 ——————— r_norm::Float64 mu::Float64	v_circ::Float64	Obtain v_c

Table 4.3 continued from previous page

Function Name	Inputs	Outputs	Functionality
velocityPeriapsis	stateVector::MyStateVector mu::Float64	v_p::Float64	Obtain v_p
velocityApoapsis	stateVector::MyStateVector mu::Float64	v_a::Float64	Obtain v_a
velocityHyperbolicAsymptote	stateVector::MyStateVector mu::Float64	v_h::Float64	Obtain v_h
speed_visViva	r_norm::Float64 a::Float64 mu::Float64	v::Float64	Obtain v
rightAscensionOfTheAscendingNode	stateVector::MyStateVector	RAAN::Float64	Obtain Ω
argumentOfPeriapsis	stateVector::MyStateVector mu::Float64	omega::Float64	Obtain ω
vector_in_ECI_to_perifocal	ECI_vector::Vector coe::MyCOE	perifocalVector ::MyVector	$\mathbf{r}_{ECI} \rightarrow \mathbf{r}_{PQW}$
vector_in_perifocal_to_ECI	perifocalVector::Vector coe::MyCOE	ECI_vector ::MyVector	$\mathbf{r}_{PQW} \rightarrow \mathbf{r}_{ECI}$
propagate_StateVector	stateVector::MyStateVector mu::Float64 delta_t::Float64	stateVector2 ::MyStateVector	Propagate $(\mathbf{r}, \mathbf{v})$ some Δt

TABLE 4.3. IDEAL TWO BODY PROBLEM FUNCTIONS

Additionally, as displayed on table 4.4, the file incorporates functions specific to Kepler's problem, which focuses on the motion of bodies under the influence of a central gravitational force. The aim of this part is to be able to relate time and position. In order to achieve that, it is necessary to be able to convert between the different anomalies at ease. These conversions differ depending on the orbit type (elliptic, parabolic or hyperbolic), and obtaining them can become quite tricky. Because for the path from the time to the auxiliary anomaly, an iterative numerical method is needed. In our code, the Newton-Raphson method was used, provided by the Roots package.

However, in the code not only the specific cases are developed. We also incorporated the universal formulation of Kepler's equation to solve for the universal anomaly χ which applied an Stumpff function and the numerical solver.

Function Name	Inputs	Outputs	Functionality
timeSincePeriapsis	stateVector::MyStateVector mu::Float64 theta::Float64	time::Float64	$\theta[rad] \rightarrow \Delta t[s]$
timeSinceTrueAnomaly	stateVector::MyStateVector mu::Float64 theta1::Float64 theta2::Float64	time::Float64	Δt between θ_1 & θ_2
stumpffFunction	z::Float64	S::Float64	Stumpff function

Table 4.4 continued from previous page

Function Name	Inputs	Outputs	Functionality
universalAnomaly	stateVector::MyStateVector mu::Float64 time::Float64	X::Float64	$\Delta t \rightarrow \chi$
universalAnomaly_to_trueAnomaly	stateVector::MyStateVector mu::Float64 X::Float64	theta::Float64	$\chi \rightarrow \theta$
trueAnomaly_to_universalAnomaly	stateVector::MyStateVector mu::Float64 theta::Float64	X::Float64	$\theta \rightarrow \chi$
timeSincePeriapsis_to_trueAnomaly	stateVector::MyStateVector mu::Float64 time::Float64	theta::Float64	$\Delta t \rightarrow \theta$
trueAnomaly_to_eccentricAnomaly	stateVector::MyStateVector mu::Float64 theta::Float64	E::Float64	$\theta \rightarrow E$
eccentricAnomaly_to_trueAnomaly	stateVector::MyStateVector mu::Float64 E::Float64	theta::Float64	$E \rightarrow \theta$
eccentricAnomaly_to_meanAnomaly	stateVector::MyStateVector mu::Float64 E::Float64	Me::Float64	$E \rightarrow M_e$
meanAnomaly_to_eccentricAnomaly	stateVector::MyStateVector mu::Float64 Me::Float64	E::Float64	$M_e \rightarrow E$
trueAnomaly_to_meanAnomaly_E	stateVector::MyStateVector mu::Float64 theta::Float64	Me::Float64	$\theta \rightarrow M_e$
meanAnomaly_to_trueAnomaly_E	stateVector::MyStateVector mu::Float64 Me::Float64	theta::Float64	$M_e \rightarrow \theta$
universalAnomaly_to_eccentricAnomaly	stateVector::MyStateVector mu::Float64 X::Float64	E::Float64	$\chi \rightarrow E$
eccentricAnomaly_to_universalAnomaly	stateVector::MyStateVector mu::Float64 E::Float64	X::Float64	$E \rightarrow \chi$
timeSincePeriapsis_to_Me	stateVector::MyStateVector mu::Float64 time::Float64	Me::Float64	$\Delta t \rightarrow M_e$
Me_to_timeSincePeriapsis	stateVector::MyStateVector mu::Float64 Me::Float64	time::Float64	$M_e \rightarrow \Delta t$
trueAnomaly_to_parabolicAnomaly	stateVector::MyStateVector mu::Float64 theta::Float64	P::Float64	$\theta \rightarrow P$
parabolicAnomaly_to_trueAnomaly	stateVector::MyStateVector mu::Float64 P::Float64	theta::Float64	$P \rightarrow \theta$

Table 4.4 continued from previous page

Function Name	Inputs	Outputs	Functionality
parabolicAnomaly_to_meanAnomaly	stateVector::MyStateVector mu::Float64 P::Float64	Mp::Float64	$P \rightarrow M_p$
meanAnomaly_to_parabolicAnomaly	stateVector::MyStateVector mu::Float64 Mp::Float64	P::Float64	$M_p \rightarrow P$
trueAnomaly_to_meanAnomaly_P	stateVector::MyStateVector mu::Float64 theta::Float64	Mp::Float64	$\theta \rightarrow M_p$
meanAnomaly_to_trueAnomaly_P	stateVector::MyStateVector mu::Float64 Mp::Float64	theta::Float64	$M_p \rightarrow \theta$
universalAnomaly_to_parabolicAnomaly	stateVector::MyStateVector mu::Float64 X::Float64	P::Float64	$\chi \rightarrow P$
parabolicAnomaly_to_universalAnomaly	stateVector::MyStateVector mu::Float64 E::Float64	X::Float64	$P \rightarrow \chi$
timeSincePeriapsis_to_Mp	stateVector::MyStateVector mu::Float64 time::Float64	Mp::Float64	$\Delta t \rightarrow M_p$
Mp_to_timeSincePeriapsis	stateVector::MyStateVector mu::Float64 Mp::Float64	time::Float64	$M_p \rightarrow \Delta t$
trueAnomaly_to_hyperbolicAnomaly	stateVector::MyStateVector mu::Float64 theta::Float64	H::Float64	$\theta \rightarrow H$
hyperbolicAnomaly_to_trueAnomaly	stateVector::MyStateVector mu::Float64 H::Float64	theta::Float64	$H \rightarrow \theta$
hyperbolicAnomaly_to_meanAnomaly	stateVector::MyStateVector mu::Float64 H::Float64	Mh::Float64	$H \rightarrow M_h$
meanAnomaly_to_hyperbolicAnomaly	stateVector::MyStateVector mu::Float64 Mh::Float64	H::Float64	$M_h \rightarrow H$
trueAnomaly_to_meanAnomaly_H	stateVector::MyStateVector mu::Float64 theta::Float64	Mh::Float64	$\theta \rightarrow M_h$
meanAnomaly_to_trueAnomaly_H	stateVector::MyStateVector mu::Float64 Mh::Float64	theta::Float64	$M_h \rightarrow \theta$
universalAnomaly_to_hyperbolicAnomaly	stateVector::MyStateVector mu::Float64 X::Float64	H::Float64	$\chi \rightarrow H$
hyperbolicAnomaly_to_universalAnomaly	stateVector::MyStateVector mu::Float64 H::Float64	X::Float64	$H \rightarrow \chi$

Table 4.4 continued from previous page

Function Name	Inputs	Outputs	Functionality
timeSincePeriapsis_to_Mh	stateVector::MyStateVector mu::Float64 time::Float64	Mh::Float64	$\Delta t \rightarrow M_h$
Mh_to_timeSincePeriapsis	stateVector::MyStateVector mu::Float64 Mh::Float64	time::Float64	$M_h \rightarrow \Delta t$

TABLE 4.4. KEPLER'S PROBLEM FUNCTIONS

To better understand all the possible directions of conversion between the different anomalies and time, the diagram in figure 4.6 can be consulted. The meaning of each of the symbols is explained in section 3.2.1.

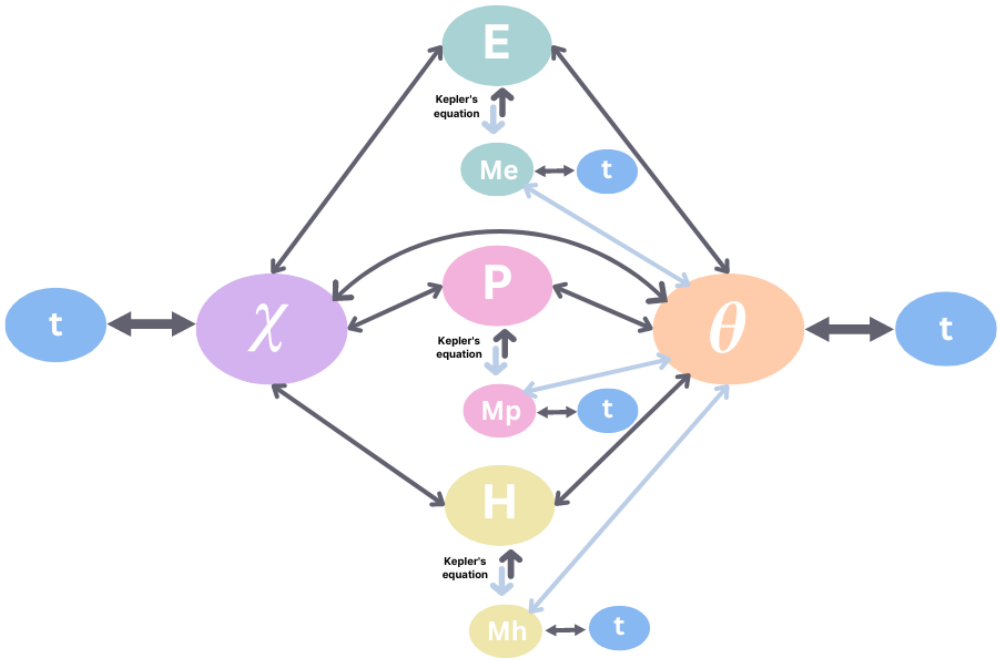


Fig. 4.6. Implemented directions of conversion between anomalies and time.

Inside the file, the initial orbit determination is also tackled through the Lambert's problem. The functions written in table 4.5 facilitate the determination of the time of flight, conic section parameters, and the solution of Lambert's problem itself.

Function Name	Inputs	Outputs	Functionality
Lambert_solve	r1::MyVector r2::MyVector tF::Float64 mu::Float64 k::MyVector	coe1::MyCOE coe2::MyCOE	Given two positions $\mathbf{r_1}$ and $\mathbf{r_2}$ and the time of flight between them, find the orbit that connects them
Lambert_conic	r1::MyVector r2::MyVector, eT::Float64 k::MyVector	coe1::MyCOE coe2::MyCOE	Obtain the conic section for a given transverse eccentricity e_T using the Avanzini algorithm
timeOfFlight	coe1::MyCOE coe2::MyCOE mu::Float64 ———————— coe1::MyCOE theta2::Float64 mu::Float64	delta_t::Float64	Time of flight between two true anomalies (θ_1 and θ_2)

TABLE 4.5. LAMBERT'S PROBLEM FUNCTIONS

4.5. Orbital Maneuvers

The precise control and execution of orbital maneuvers play a critical role in space missions and satellite operations. The orbitalManeuvers.jl file offers a comprehensive set of functions to obtain the delta-v costs of different orbital maneuvers in space; to help students obtain the preliminary estimations performed in the early stages of planning any space mission. These functions, listed in Table 4.6,are designed to address some specific maneuver scenarios and requirements. Therefore, enabling them to determine the optimal strategies for achieving the desired orbital configurations.

One of the fundamental maneuvers supported by the module is the Hohmann transfer, which allows the transfer of a spacecraft or satellite between two coplanar orbits. It also includes functions for performing plane change maneuvers, and some approximations for low-thrust maneuvers, these latter maneuvers are typically executed by utilizing continuous and low-thrust propulsion systems.

Function Name	Inputs	Outputs	Functionality
HohmannTransfer_circular	r1::Float64 r2::Float64 mu::Float64	deltaV::Float64	Hohmann transfer between 2 circular and coplanar orbits
HohmannTransfer_elliptic	rp1::Float64 e1::Float64 rp2::Float64 e2::Float64 mu::Float64	deltaV::Float64	Hohmann transfer between 2 elliptic and coplanar orbits

Table 4.6 continued from previous page

Function Name	Inputs	Outputs	Functionality
planeChange_circular	ra::Float64 delta_i::Float64 mu::Float64	deltaV::Float64	Obtain the Δv required to perform a Plane Change Maneuver between 2 circular orbits
planeChange_apoapsis2apoapsis	stateVector1::MyStateVector stateVector2::MyStateVector delta_i::Float64 mu::Float64 ——————— coe1::MyCOE coe2::MyCOE mu::Float64	deltaV::Float64	Obtain the min Δv required to perform a Plane Change Maneuver between 2 elliptical orbits
LowThrust_orbitRaising_circular	r1::Float64 r2::Float64 mu::Float64	deltaV::Float64	Low Thrust Orbit Raising between 2 circular and coplanar orbits
LowThrust_planeChange_circular	ra::Float64 delta_i::Float64 mu::Float64	deltaV::Float64	Low Thrust Plane Change Maneuver between two circular orbits with the same radius (*ra*)
HohmannTransfer_tof	coe1::MyCOE coe2::MyCOE tof::Float64 mu::Float64	deltaV::MyVector vT1::MyVector vT2::MyVector	Given a desired time of flight and the COE of two orbits, calculate the Δv vector and the velocity vectors at the beginning and at the end of the Hohmann transfer.

TABLE 4.6. ORBITAL MANEUVERS FUNCTIONS

4.6. Orbital Perturbations

Within the field of astrodynamics, the accurate propagation of orbits is of paramount importance. Orbital perturbations, such as those caused by the Earth's oblateness (J2 term), and atmospheric drag, can significantly affect the trajectory of a satellite or spacecraft. To address these perturbations and enable precise orbit prediction, the orbitalPerturbations.jl file has been developed. Table 4.7 shows the functions available inside the module.

Function Name	Inputs	Outputs	Functionality
J2_acceleration	r::Vector	a_p_J2::Vector	Calculate the perturbation acceleration due to the Earth's oblateness.
drag_acceleration	alt::Float64 v::Vector Bc::Float64	a_p_drag::Vector	Calculate the perturbation acceleration due to the drag on Earth

Table 4.7 continued from previous page

Function Name	Inputs	Outputs	Functionality
cowell	r0::Vector{Float64} v0::Vector{Float64} Bc::Float64 t::Vector flag_drag::Bool flag_J2::Bool	r::Vector v::Vector	Propagate the orbit using Cowell's method given the initial conditions and a vector of time instants for which the solution will be provided. It can considered the drag and/or J2 perturbations if desired.

TABLE 4.7. ORBITAL PERTURBATIONS FUNCTIONS

One widely used propagator is the **Cowell's method**, which provides a numerical solution to the equations of motion by accounting for the perturbing forces. As can be seen in figure 4.7, these perturbing forces are all summed together to form the total force acting on the orbiting body, and then this total force is numerically integrated starting from the initial position. [20]

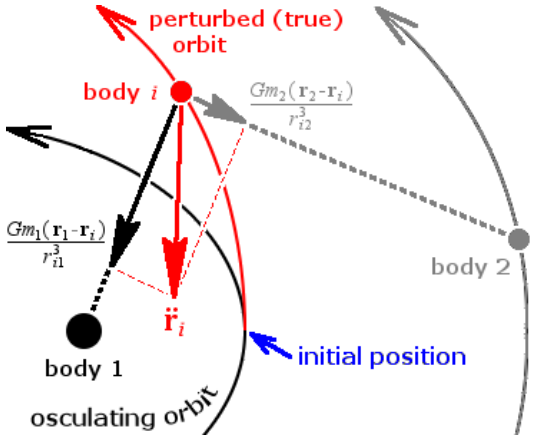


Fig. 4.7. Cowell's Method Representation [20]

This way, the Cowell's method propagates the orbit over time by iteratively advancing the state vector of the object, while considering the effects of gravitational perturbations, atmospheric drag, and other external forces.

4.7. Orbital Visualization

One of the main objectives of this library is not only to allow the computation of orbital dynamics but also the visualization of orbits in an appealing and engaging way.

The orbitalPlots.jl file provides a powerful toolset for generating high-quality orbit representations using the Plots package in Julia. By leveraging the capabilities of Plots, this module enables the creation of visually appealing and informative plots of various orbital elements and parameters

Upon initial use, it is worth noting that the orbitalPlots.jl file, and therefore the package, may take some time to load due to its dependence on the heavy Plots package. However, once loaded, the module functions smoothly and efficiently. In some cases, when writing code outside of the terminal, it may be necessary to include the `display()` function to ensure that the plots are rendered and displayed correctly.

Because of the relevance and regularity of these reference frames, this module allows to plot orbits in both the ECI and the perifocal reference frames. In the perifocal reference frame, the plots are done in the perifocal plane, in 2D. It uses the `gr()` backend to visualize them. Whereas, the plots in ECI reference frame are in 3D, and use the `plotly()`. The reason behind using this backend for the ECI plots instead of `gr()`, is that `plotly()` allows the user not only to visualize but also to interact with it. One is able to move around the plot and observe the orbit from a different perspective. For aesthetic reasons, both use `:juno` theme from PlotThemes.

The following examples showcase the visualization of orbits in the perifocal (PF) coordinate system, providing insights into the shape, size, and orientation of the orbits.

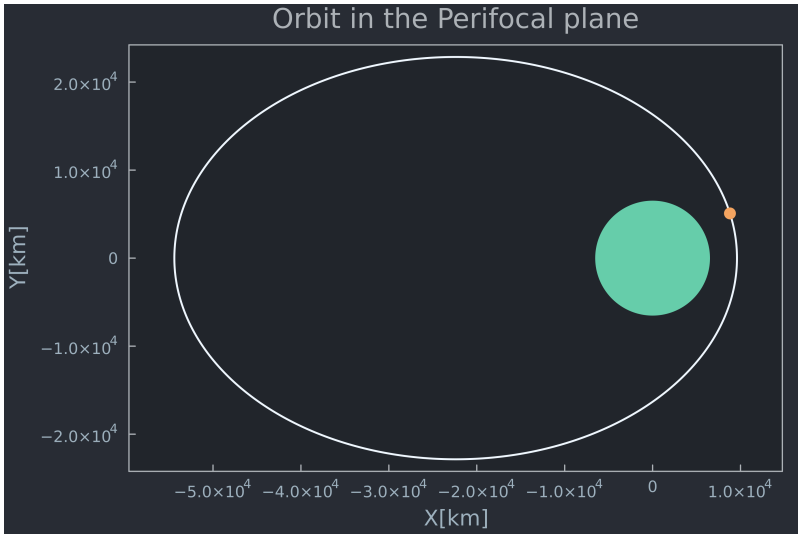


Fig. 4.8. Elliptical orbit around Earth plotted in the perifocal plane

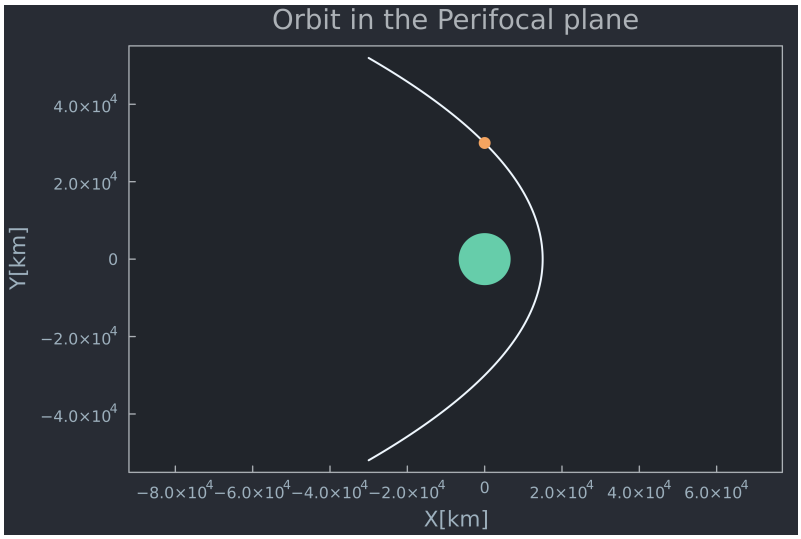


Fig. 4.9. Parabollic orbit around Earth plotted in the perifocal plane

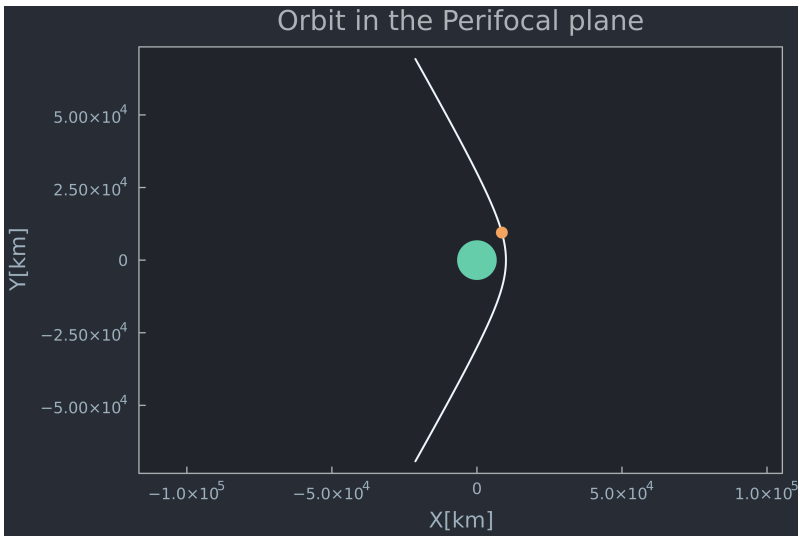


Fig. 4.10. Hyperbolic orbit around Earth plotted in the perifocal plane

Additionally, an example in the Earth-Centered Inertial (ECI) coordinate system is also presented in 4.11. It shows a satellite in a LEO (Low Earth Orbit) around Earth. This coordinate system is commonly used in space missions and satellite tracking.

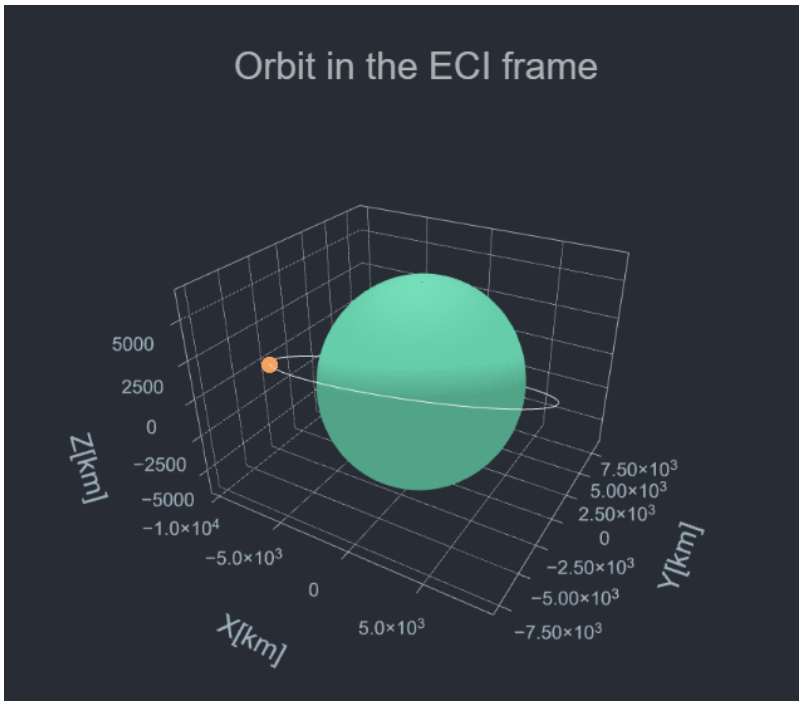


Fig. 4.11. Nearly circular orbit around Earth plotted in ECI reference frame

These examples serve as valuable references for understanding the behavior and characteristics of various types of orbits. Whether it's a circular orbit, elliptical orbit, or more complex trajectories, the orbitalPlots.jl file empowers students to gain deeper insights into orbital dynamics through visually compelling representations.

In table 4.8, it can be seen the functions needed to do each plot, as well as the inputs required. There also appears a function to obtain a characteristic energy (C3) porkchop plot called `plot_porkchop`, an example of its usage and how it looks is shown in section 5.2.2.

Function Name	**Inputs**	**Functionality**
plot_orbit_perifocal	coe::MyCOE planet_radius::Float64	Plot the orbit around a planet and the position of the satellite on it in the perifocal plane (2D)
plot_orbit_ECI	coe::MyCOE planet_radius::Float64 mu::Float64	Plot the orbit around a planet and the position of satellite on it in the ECI frame (3D)
plot_porkchop	t_launch::Vector [days] t_travel::Vector [days] pork_plot::Matrix [km^2/s^2] t_synodic::Number [days]	Plot the C3 porkchop from a matrix of length(t_launch) x length(t_travel) containing the characteristic energy

TABLE 4.8. ORBITAL PLOTS FUNCTIONS

5. VERIFICATION

5.1. Tests

In this section, we present the evaluation of the educational astrodynamics library implemented in Julia. The evaluation consists of a series of tests conducted to verify the functionality and correctness of the library's files. Each test focuses on a specific file and assesses whether the functions within that file produce the expected results. The tests serve as an important step in validating the library's performance and ensuring its reliability for educational purposes.

The evaluation is divided into separate subsections, each corresponding to a file within the library. The tests are designed to cover a wide range of scenarios and input conditions to thoroughly examine the library's capabilities. By systematically testing the library's functions, we aim to demonstrate their correctness and usefulness for educational purposes in the field of Astrodynamics.

It is important to note that the evaluation focuses primarily on the functionality and accuracy of the library's files. Performance benchmarks, such as execution time and memory usage, are not the primary objectives of this evaluation. However, if significant performance differences are observed, they will be briefly discussed to provide a comprehensive overview of the library's overall performance characteristics.

5.1.1. Linear Algebra Types Tests

List of unit tests:

- `test_MyVector()`: Tests various constructors and properties of the `MyVector` type. It also tries invalid inputs for the constructors and ensures that they throw the expected exceptions.
- `test_MyVector_operations()`: Tests various operations on `MyVector` objects, such as addition, subtraction, scalar multiplication, dot product, cross product, and more. These operations were defined so that `MyVector` could use the same operations as a regular Vector object.
- `test_MyBasis()`: Tests constructors and transformations of `MyBasis` objects. It creates `MyBasis` objects using different constructors and checks that the resulting objects have the correct basis vectors, angular velocity, and angular acceleration. It also tests invalid inputs for the constructors and ensures that they throw the expected exceptions.

- `test_MyPoint()`: Tests constructors and properties of `MyPoint` objects.
- `test_MyReferenceFrame()`: Tests constructors and properties of `MyReferenceFrame` objects.
- `test_MyParticle()`: Tests constructors and properties of `MyParticle` objects.
- `test_componentsInBasis()`: Tests the `componentsInBasis` function for converting vectors between different bases.
- `test_pos_vel_acc_inRF()`: Tests the `pos_vel_acc_inRF` function for transforming positions, velocities, and accelerations between reference frames.
- `test_rotation_matrix()`: Tests the computation of a rotation matrix from a `MyBasis` object. It checks if the computed rotation matrix correctly transforms a vector from one coordinate system to another.
- `test_pure_rotation_MyBasis()`: Tests the creation of a new `MyBasis` object that represents a pure rotation from one basis to another. It checks if the basis vectors, angular velocity, and angular acceleration of the new basis are computed correctly based on the given inputs.

In addition to these units test, there is an additional one called `test_LinearAlgebraTypes()`. This test uses the functions from the Linear Algebra Types file to solve an example problem. It checks that the results obtained using the package are the same as the ones obtained with an outside software.

The problem statement is as follows:

"We are located at the inertial reference frame $S_0 : \{O_0; B_0\}$. There exists in space a reference frame 1 $S_1 : \{O_1; B_1\}$ whose basis B_1 is known with respect to the canonical reference basis (B_0) and whose origin point O_1 coordinates are known with respect to the inertial reference frame S_0 at a certain instant time. In addition, there is another non-inertial reference frame $S_2 : \{O_2; B_2\}$ whose basis is defined with respect to B_1, and whose origin coordinates are known in reference frame S_1.

We have two observers: Nick and Jess. Nick is the observer at Reference Frame 1 and calls us to report the position, velocity and acceleration of a point particle 1, with mass = 10 kg. While Jess stationed at Reference Frame 2, reports to us the position, velocity and acceleration of a different point particle 2, with mass = 2 kg.

We would like to determine the vector coordinates (position, velocity and acceleration) of both of those particles with respect to us (S_0), as well as the distance between the two particles."

Here we have how this problem would be computed with our code (the '#' introduces a comment in Julia):

```
# rf0 = CanonicalReferenceFrame
# origin0 = CanonicalOriginPoint

# b1 = Basis of rf1, known wrt b0
i10 = MyVector(0, 0, 1)
j10 = MyVector(0, 1, 0)
k10 = MyVector(1, 0, 0)
omega10 = MyVector(0, 2, 0)
alpha10 = MyVector(0, 0, 1)
b1 = MyBasis(i10, j10, k10, omega10, alpha10)

# Origin Point of rf1 wrt rf0
posO1_10 = MyVector(0,3,1)
velO1_10 = MyVector(0,0,5)
accO1_10 = MyVector(0,0,0)
origin1 = MyPoint(posO1_10,velO1_10,accO1_10) # wrt rf0

# rf1 = ReferenceFrame wrt rf0
rf1 = MyReferenceFrame(b1, origin1)

# p1 = Point wrt rf1
posP1_11 = [1,1,1]
velP1_11 = [10,0,0]
accP1_11 = [-2,0,0]
p1 = MyPoint(posP1_11, velP1_11, accP1_11, rf1)

# particle1 = Particle at point 1 wrt rf1
mass1 = 10 # kg
particle1 = MyParticle(p1, mass1)

# b2 = Basis of rf2, known wrt b1
i21 = [0, 1, 0]
j21 = [1, 0, 0]
k21 = [0, 0, 1]
omega21 = [0, 0, 3]
alpha21 = [0, 0, 1]
b2 = MyBasis(i21, j21, k21, omega21, alpha21, b1)

# Origin Point of rf2 wrt rf1
posO2_21 = [-4,0,0]
velO2_21 = [0,0,10]
accO2_21 = [0,0,1]
```

```
origin2 = MyPoint(posO2_21,velO2_21,accO2_21,rf1) # wrt rf1

# rf2 = ReferenceFrame wrt rf1
rf2 = MyReferenceFrame(b2, origin2)

# p2 = Point wrt rf2
posP2_22 = [1,1,1]
velP2_22 = [20,0,0]
accP2_22 = [-2,0,0]
p2 = MyPoint(posP2_22, velP2_22, accP2_22, rf2)

# particle2 = Particle at point 2 wrt rf2
mass2 = 5 # kg
particle2 = MyParticle(p2, mass2)

# Express Point 1 wrt rf2 using pos_vel_acc_inRF
P1_posRF2, P1_velRF2, P1_accRF2 = pos_vel_acc_inRF(p1, rf2)

# Find distance between particle2 and particle1
dist_P2P1_rf2 = norm(P1_posRF2 - posP2_22)
dist_P2P1 = norm(p1.position.c0 - p2.position.c0)
```

Therefore, with our code the results obtained are:

```
# Point 1 expressed wrt CanonicalReferenceFrame
p1 = MyPoint(MyVector([1.0, 4.0, 2.0]),
    MyVector([2.0, 0.0, 13.0]), MyVector([35.0, 1.0, -6.0]))

# Point 2 expressed wrt CanonicalReferenceFrame
p2 = MyPoint(MyVector([1.0, 4.0, -2.0]),
    MyVector([4.0, 17.0, 6.0]), MyVector([8.0, -11.0, 84.0]))

# Distance between point 1 and point 2 (wrt CanonicalReferenceFrame)
dist_P2P1 = 4.0

# Distance between point 1 and point 2 (wrt rf2)
dist_P2P1_rf2 = 4.0
```

These were the results obtained with the computations made individually with MATLAB:

```
# Point 1 expressed wrt CanonicalReferenceFrame
```

```
P1_pos0 = [1.0,4.0,2.0]
P1_vel0 = [2.0,0.0,13.0]
P1_acc0 = [35.0,1.0,-6.0]

# Point 2 expressed wrt CanonicalReferenceFrame
P2_pos0 = [1.0,4.0,-2.0]
P2_vel0 = [4.0,17.0,6.0]
P2_acc0 = [8.0,-11.0,84.0]

# Distance between point 1 and point 2
distance = 4.0
```

As we can clearly observe, the results obtained are the same. And not only that, to show that the conversion work the distance was compute both with respect to reference frame 2 and with respect to the canonical reference frame, which was the case. Because no matter with respect to which reference frame the distance is computed, it should be the same.

After solving the problem, we have all the objects defined with MyVector, MyBasis, MyReferenceFrame and MyPoint. So now, we can use the recipes to plot the three different reference frames and the two particles, and see this way what they look like in 3D space.

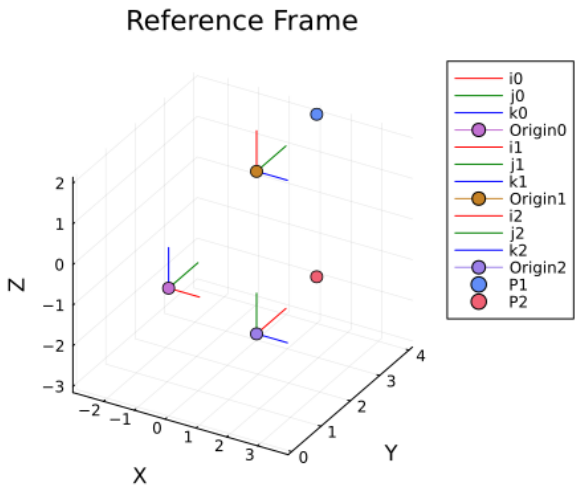


Fig. 5.1. Linear Algebra Example

This proves that this module works properly, but most importantly it also shows its versatility and helpfulness. Especially by how it speeds the calculation process, and allows the user to visualize their results in 3D space, for a better comprehension of the problem.

5.1.2. Ideal Two Body Problem Tests

Using examples and problems, different unit tests are built to assure the proper performance of the Ideal Two Body Problem file. These tests all pass with a relative tolerance of $1e-3$ compared to the results of the solved problems on paper. It is worth noting, that these tests use the constants defined in the astroConstants.jl file, so a higher accuracy is to be expected from our code with respect to the manual calculation, as the latter does not take into account as many decimal positions. The defined unit tests are the following:

- `test_MyStateVector`: Tests the MyStateVector constructor by creating an object with valid input and checks if the attributes r and v are assigned correctly. Also tests the constructor with invalid inputs.
- `test_MyCOE`: Tests the MyCOE constructor by creating an object with valid inputs and checks if the attributes RAAN, inc, omega, a, e, theta, and p are assigned correctly. Also tests the constructor with invalid inputs, and makes the sure the user gets throw the appropriate warning.
- `test_I2BPfunctions`: Tests individual functions of the IdealTwoBody problem, such as perifocalBasis, periodOrbit, velocityPeriapsis, velocityApoapsis, escapeVelocity, stateVector_to_COE, and trajectoryEquation.
- `test_OrbitElements`: Tests the individual computation of orbit elements such as angular momentum vector h, eccentricity vector e, semi-major axis a, inclination i, right ascension of the ascending node Ω, argument of periapsis ω, and true anomaly θ. As well as the conversion from the state vector to the classical orbital elements, and that both obtain the same results.
- `test_COE2StateVector`: Tests the conversion from COE (Classical Orbital Elements) to state vector.
- `test_KeplerElliptic`: Tests Kepler's equation for an elliptic orbit. It calculates and verifies the semi-major axis, eccentricity, distance to apogee, and orbital period. As well as the time that it takes to reach a certain true anomaly, and all the different possible conversions between anomalies.

All of these unit tests pass within the specified tolerance ($rtol = 10^{-3}$), therefore proving that the functions have been properly implemented.

5.1.3. Orbital Maneuvers Tests

The Orbital Maneuvers file includes several unit tests to validate the functionality of the functions within the file. These tests ensure the accuracy and reliability of the implemented orbital maneuver models. Here is a breakdown of the unit tests included in the file:

- `test_HohmannCircular()`: This test verifies the Hohmann transfer between two circular and coplanar orbits. It calculates the delta-V required for the transfer and compares it to the expected value within a specified tolerance.
- `test_HohmannElliptic()`: This test examines the Hohmann transfer between two elliptic orbits around the same central body. It calculates the minimum delta-V required to change the orbit from one set of parameters to another and compares it to the expected value within a specified tolerance.
- `test_PlaneChangeCircular()`: This test focuses on a change of inclination for a circular orbit. It calculates the delta-V required to perform the plane change and compares it to the expected value within a specified tolerance.
- `test_planeChangeElliptic()`: This test demonstrates a plane change between two elliptic orbits. It calculates the delta-V required for the plane change using both classical orbital elements and state vectors and verifies if the results match the expected values within a specified tolerance.
- `test_LowThrustOrbitRaising()`: This test examines low-thrust orbit raising between two circular and coplanar orbits. It calculates the delta-V required for the maneuver and compares it to the expected value within a specified tolerance.
- `test_LowThrustPlaneChange()`: This test focuses on a low-thrust plane change maneuver between two circular orbits with equal radii. It calculates the delta-V required for the plane change and compares it to the expected value within a specified tolerance.
- `test_HohmannTransfer_tof()`: This test calculates the delta-V required to perform a Hohmann transfer given a specific time of flight (tof). It also verifies the resulting velocity vectors at the starting and ending points of the transfer.

All the tests pass within the specified tolerance ($rtol = 10^{-3}$), this provides confidence in the accuracy and reliability of the orbital maneuver approximated models implemented in this module.

5.1.4. Orbital Perturbations Tests

The Orbital Perturbations file incorporates various perturbations, such as J2 gravity effects and atmospheric drag, which significantly impact the orbital motion and stability of objects in space. To ensure the accuracy and reliability of the implemented perturbation models, a series of unit tests have been designed to validate the functionality of key functions within the module. These tests assess the correctness of acceleration calculations, simulate the motion of objects under different perturbation scenarios, and compare the calculated results against expected values. The unit tests included in this section are as follows:

- `test_J2_acceleration`: Tests the J2 acceleration function by providing a position vector and checking if the calculated acceleration matches the expected acceleration within a specified tolerance.
- `test_drag_acceleration`: Tests the drag acceleration function by providing a position vector, velocity vector, and other parameters, and checking if the calculated acceleration matches the expected acceleration within a specified tolerance.
- `test_cowell`: Tests the Cowell method, which integrates the equations of motion, by providing initial position and velocity vectors, along with flags for drag and J2 perturbations. The test verifies if the calculated position and velocity vectors match the expected vectors within a specified tolerance for different perturbation cases.

5.2. Examples

To further evaluate and demonstrate the reliability and correctness of the educational astrodynamics library developed, a series of homework exercises from the Orbital Dynamics course of the Universidad Carlos III of Madrid have been solved as examples using the library's functions. These exercises cover various topics in astrodynamics, including orbital mechanics, trajectory analysis, and spacecraft maneuvers. By applying the library's functions to these real-world homework problems, we aim to assess the library's practical utility and its ability to provide accurate and reliable solutions.

The selected homework exercises represent typical challenges encountered in astrodynamics coursework, requiring the application of fundamental principles and techniques. The exercises involve the determination of orbital elements, the computation of orbital parameters, and the analysis of spacecraft trajectories under the influence of gravitational forces.

By incorporating the resolution of real-world homework exercises, we enhance the evaluation of the educational astrodynamics library. The practical application of the library's functions to authentic problem scenarios further reinforces its value as a learning tool and showcases its potential to facilitate understanding and mastery of astrodynamics concepts.

In the following subsections, we present the solved homework exercises using Jupyter notebooks, outline the implementation details utilizing the library's functions, and provide an overall assessment of the library's performance in resolving these exercises.

5.2.1. Problem 1 - Earth satellite

HOMEWORK: EXERCISE 1 - Earth satellite

We consider the trajectory of an Earth spacecraft of 2000 kg mass. There is ground station located 50 deg North, 800 m above sea level, on which we define a topocentric South-East- Zenith (SEZ) reference frame S1. It is known that at 22:30 (local time at the station), the state of the spacecraft with respect to S1 is:

r1 = [5369.09, 2332.14, 656.480] km, v1 = [-3.37027, 4.90364, 0.106703] km/s.

We also define an equatorial Earth-centered inertial (ECI) reference frame S0. The station crossed the Oxz plane of S0 (with x >= 0) at 09:00 local time.

In [ ]:
```
# Data:
m_sat = 2000.0 # kg
phi = deg2rad(50.0) # rad (North)
#lambda = deg2rad(0.0) # rad (Prime Meridian)
alt = 0.8 # km
# S1: topocentric South-East-Zenith (SEZ) reference frame
# At local time = 22:30
r1 = [5369.09, 2332.14, 656.480] # km
v1 = [-3.37027, 4.90364, 0.106703] # km/s
# S0: equatorial Earth-centered inertial (ECI) reference frame
# At local time = 9:00 crosses Oxz plane (x >=0)
```

```
3-element Vector{Float64}:
-3.37027
 4.90364
 0.106703
```

To use the AstrodynamicsEdu.jl package, first we need to activate the new environment and add the package from its GitHub repository

In [ ]:
```
using Pkg
# Pkg.activate(".")
Pkg.activate("C:\\Users\\alici\\Desktop\\Julia\\HOMEWORK")
Pkg.instantiate()
Pkg.add(url="https://github.com/AliciaSBa/AstrodynamicsEdu.jl.git")

using AstrodynamicsEdu
using LinearAlgebra
```

```
  Activating project at `C:\Users\alici\Desktop\Julia\HOMEWORK`
    Updating git-repo `https://github.com/AliciaSBa/AstrodynamicsEdu.jl.git`
    Updating registry at `C:\Users\alici\.julia\registries\General.toml`
   Resolving package versions...
  No Changes to `C:\Users\alici\Desktop\Julia\HOMEWORK\Project.toml`
  No Changes to `C:\Users\alici\Desktop\Julia\HOMEWORK\Manifest.toml`
```

(a) Implement a function `[r0,v0] = SEZ2ECI(r1,v1,rS,phi,t)` that computes the position and velocity vectors r0, v0 in ECI (expressed in the ECI basis), provided the position and velocity vectors r1, v1 in SEZ (expressed in the SEZ basis), the radius rS of the station (in km) and latitude ϕ (in rad) of the observer from the center of the Earth, and the time t in hours since the observer crossed the Oxz plane of ECI. Use units of km and km/s. Apply it to compute r0, v0 using the initial spacecraft data and report your results.

The r1 and v1 vectors define the position and velocity of the Point of the satellite with respect to reference frame S1 (i.e. SEZ). We need to find its coordinates with respect to the CanonicalReferenceFrame (i.e. ECI). A point expressed with respect to the CanonicalReferenceFrame is simply a MyPoint.

To create a MyPoint, first we need to define the reference frame S1 as a MyReferenceFrame. For that we need to obtain Basis 1, which is the CanonicalBasis rotated by $\theta = \omega_E t + \Lambda$ wrt z-axis, where $\Lambda = 0$ and ω_E is the rotational speed of the Earth; and then by $-\phi$ wrt y-axis. It is important to remember that as the Earth is rotating, there is an angular velocity of $\omega_E k_0$ expressed in the CanonicalBasis. We also need to obtain the origin point of S1, which is the the station. Considering relative kinematics,

$$\mathbf{r_0^{Station}} = [{}_0R_1] * [0, 0, r_{Station}]$$

$$\mathbf{v_0^{Station}} = \omega_{10} \times \mathbf{r_0^{Station}}$$

Knowing this, we would just need to use the constructor of MyPoint for a point defined in a non-canonical reference frame.

In [ ]:
```
function SEZ2ECI(r1::Vector,v1::Vector,rStation::Float64,phi::Float64,t0::Float64)
    # Calculate the siderial time
    omega_Earth = 2*pi/86400 # rad/s
    theta = omega_Earth*t0 # rad

    # Obtain the basis of S1 wrt S0
    R_0_1 = [cos(theta)*sin(phi) -sin(theta) cos(theta)*cos(phi);
                sin(theta)*sin(phi) cos(theta) sin(theta)*cos(phi);
                -cos(phi) 0.0 sin(phi)]
    i1 = MyVector(R_0_1*i0.c0)
    j1 = MyVector(R_0_1*j0.c0)
    k1 = MyVector(R_0_1*k0.c0)
    omega_b1_0 = MyVector([0.0,0.0,omega_Earth])
    alpha_b1 = MyVector([0.0,0.0,0.0])
    basis1 = MyBasis(i1, j1, k1, omega_b1_0, alpha_b1)

    # Obtain the origin point of the station wrt S0
    rStation0 = MyVector([rStation*cos(phi)*cos(theta), rStation*cos(phi)*sin(theta),
                                                        rStation*sin(phi)])
    println("rStation0 = ", rStation0)
    vStation0 = cross(omega_b1_0,rStation0)
    println("vStation0 = ", vStation0)
    aStation0 = MyVector([0.0,0.0,0.0])
    Station = MyPoint(rStation0, vStation0, aStation0)

    # Obtain the referece frame S1, defined by the basis1 and the origin point of Station
    S1 = MyReferenceFrame(basis1, Station)

    # Obtain the position and velocity of the satellite wrt S0
    Sat0 = MyPoint(r1, v1, [0.0,0.0,0.0], S1)
    r0 = Sat0.position
    v0 = Sat0.velocity

    return r0, v0
end

rStation = R_Earth + alt # km
t0 = (22.5 - 9.0)*3600.0 # s
r0,v0 = SEZ2ECI(r1,v1,rStation,phi,t0)
```

```
println("r0 = ", r0, " km")
println("v0 = ", v0, " km/s")
```

```
rStation0 = MyVector([-3783.946420063161, -1567.3619264832814, 4881.081982665504])
vStation0 = MyVector([0.11398177578139404, -0.27517634895481075, 0.0])
r0 = MyVector([-7081.212190082387, -5457.424273163714, 1932.7903113923494]) km
v0 = MyVector([4.595300978204476, -4.083577028931066, 2.2481070375123773]) km/s
```

(b) Compute the classical orbital elements a, e,Ω, i, ω of the spacecraft, and the value of the true anomaly θ0 at the instant of the observation. Plot the orbit.

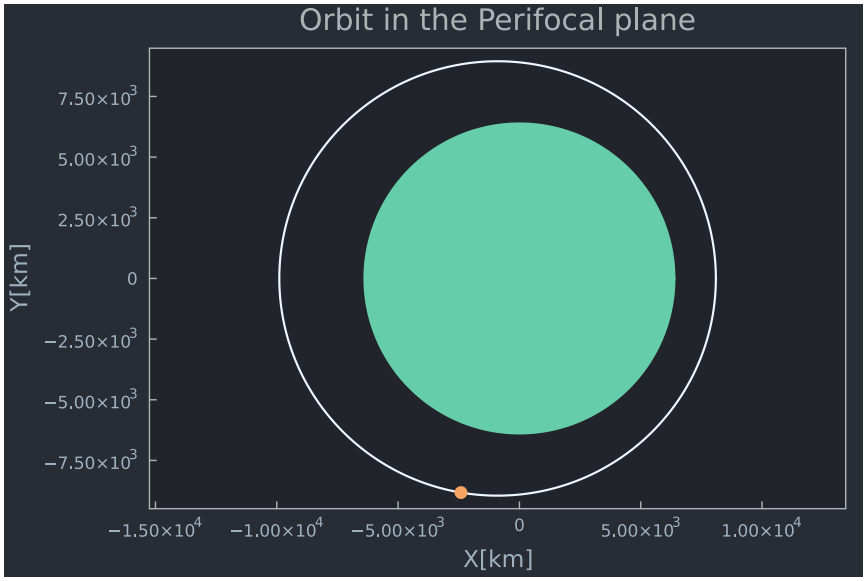
To use the `stateVector_to_COE` defined first we need to create a MyStateVector object with the position and velocity MyVector objects already computed, and then introduced the constant μ of the Earth, defined in the package.

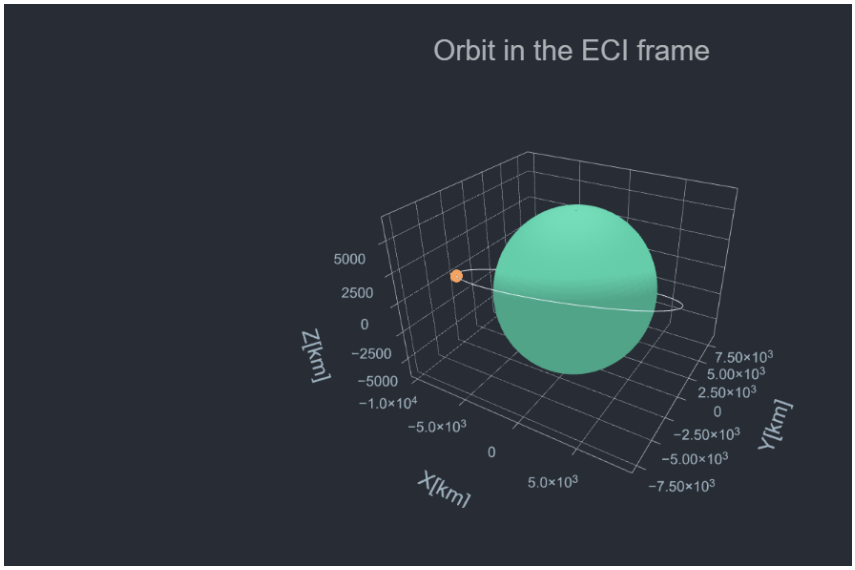
In [ ]:
```
state0 = MyStateVector(r0, v0)
coe = stateVector_to_COE(state0, mu_Earth)
println("a = ", coe.a, " km")
println("e = ", coe.e)
println("inc = ", coe.inc, " rad")
println("RAAN = ", coe.RAAN, " rad")
println("omega = ", coe.omega, " rad")
println("theta0 = ", coe.theta, " rad")
```

```
a = 8995.61613586091 km
e = 0.10010587914623008
inc = 0.4364211510849171 rad
RAAN = 3.316245899179842 rad
omega = 2.36092534167557 rad
theta0 = 4.445749406362666 rad
```

We can plot the orbit both in the perifocal plane or in the ECI reference frame using the built-in functions of `plot_orbit_perifocal` or `plot_orbit_ECI` respectively.

In [ ]:
```
#Pkg.add("PlotlyJS"); Pkg.add("PlotlyBase")
#using PlotlyJS, PlotlyBase
plot1 = plot_orbit_perifocal(coe, R_Earth)
display(plot1)
plot2 = plot_orbit_ECI(coe, R_Earth, mu_Earth)
display(plot2)
```





With the r0 computed in part a, we can find that the orbit has an altitude of around 2,775 km, so it is at the beginning of the Medium Earth Orbit (MEO), which corresponds to an altitude from 2,000 km to 35,786 km above sea level. As it was expected from the Classical Orbital Elements obtained, we have a near circular orbit with an inclination of around 22º.

(c) Determine the local time at the station when the spacecraft would reach θ = θ0 + π/2. Determine also the position and velocity vectors of the spacecraft in ECI 6 h after the initial observation. Finally, implement a function `[r1,v1] = ECI2SEZ(r0,v0,rS,phi,t)` that does the inverse of SEZ2ECI, and compute the position and velocity vectors of the spacecraft in SEZ at that instant.

To find the local time we would need to add the time of flight Δt to the initial time the position was measured (22:30). For that we would just need to use the `timeOfFlight` function from the package.

In [ ]:
```
## Determine the local time at the station when the spacecraft would reach ϑ = ϑ0 + π/2.
theta2 = coe.theta + pi/2
coe2 = MyCOE(RAAN = coe.RAAN, inc = coe.inc, omega = coe.omega, a = coe.a, e = coe.e,
                                                                    theta = theta2)
Dtime = timeOfFlight(coe,coe2,mu_Earth)
# Dtime = timeSinceTrueAnomaly(state0,mu_Earth,coe.theta,theta2)
println("Dtime = ", Dtime/3600, " h")

localTime = 22.5 + Dtime/3600.0 # h
Hour = floor(localTime)
Minutes = floor((localTime - Hour)*60.0)
Seconds = round(((localTime - Hour)*60.0 - Minutes)*60.0,digits=2)
println("Local Time = ", Hour, " h  ", Minutes, " min  ", Seconds, " s  ")
```

```
Dtime = 0.5342585085344397 h
Local Time = 23.0 h  2.0 min  3.33 s
```

To determine the position and velocity vectors of the spacecraft in ECI 6 h after the initial observation we would have to make use of Kepler's equations to go from the time to the trueAnomaly. And then convert from the COE to the state vector. Thankfully for us, these functions already exist.

In [ ]:
```
# First, let's calculate the time since periapsis for the initial observation
time0 = timeSincePeriapsis(state0,mu_Earth,coe.theta) # s
println("time0 = ", time0, " s")
# Now, let's calculate the time since periapsis for the observation 6 h later
time3 = time0 + 6*3600.0 # s
# So we can obtain that for this time, the true anomaly is
theta3 = timeSincePeriapsis_to_trueAnomaly(state0,mu_Earth,time3)
coe3 = MyCOE(RAAN = coe.RAAN, inc = coe.inc, omega = coe.omega, a = coe.a, e = coe.e,
                                                                    theta = theta3)
println("theta3 = ", coe3.theta, " rad")
# Let's convert the COE to state vector in ECI frame at this time
state3 = COE_to_stateVector(coe3,mu_Earth)
r3_0 = state3.r
v3_0 = state3.v
println("r3_0 = ", r3_0, " km")
println("v3_0 = ", v3_0, " km/s")
```

```
time0 = 6273.735313339321 s
theta3 = 1.966523657851537 rad
r3_0 = MyVector([2073.3694816878774, 8266.427689909542, -3628.895472057815]) km
v3_0 = MyVector([-6.112701337049588, 1.6711825727263003, -1.2630271576367642]) km/s
```

To convert from ECI to SEZ, we would have to follow a similar approach to before, now the only difference is that as we want the components in the S1 reference frame, we would need to use the `pos_vel_acc_inRF` function.

In [ ]:
```
# Function to convert from ECI to SEZ
function ECI2SEZ(r0::MyVector,v0::MyVector,rStation::Float64,phi::Float64,t0::Float64)

    # Calculate the siderial time
    omega_Earth = 2*pi/86400 # rad/s
    theta = omega_Earth*t0 # rad

    # Obtain the basis of S1 wrt S0
    R_0_1 = [cos(theta)*sin(phi) -sin(theta) cos(theta)*cos(phi);
             sin(theta)*sin(phi) cos(theta) sin(theta)*cos(phi);
             -cos(phi) 0.0 sin(phi)]
    i1 = MyVector(R_0_1*i0.c0)
    j1 = MyVector(R_0_1*j0.c0)
    k1 = MyVector(R_0_1*k0.c0)
```

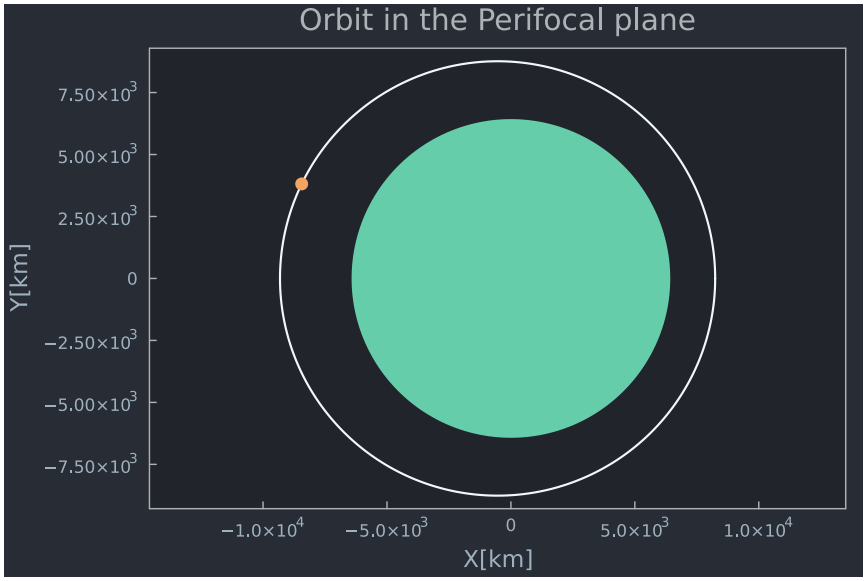
```
    omega_b1_0 = MyVector([0.0,0.0,omega_Earth])
    alpha_b1 = MyVector([0.0,0.0,0.0])
    basis1 = MyBasis(i1, j1, k1, omega_b1_0, alpha_b1)

    # Obtain the origin point of the station wrt S0
    rStation0 = MyVector([rStation*cos(phi)*cos(theta), rStation*cos(phi)*sin(theta),
                                                        rStation*sin(phi)])
    vStation0 = cross(omega_b1_0,rStation0)
    aStation0 = MyVector([0.0,0.0,0.0])
    Station = MyPoint(rStation0, vStation0, aStation0)

    # Obtain the referece frame S1, defined by the basis1 and the origin point of Station
    S1 = MyReferenceFrame(basis1, Station)

    # Obtain the position and velocity of the satellite wrt S1
    Sat0 = MyPoint(r0, v0, MyVector([0.0,0.0,0.0]))
    r1,v1,a1 = pos_vel_acc_inRF(Sat0,S1)

    return r1, v1
end

## Compute the position and velocity vectors of the spacecraft in SEZ at that instant,
        # using the function ECI2SEZ.
t3 = t0 + 6*3600.0 # s
r3_1, v3_1 = ECI2SEZ(r3_0,v3_0,rStation,phi,t3)
println("r3_S1 = ", r3_1, " km")
println("v3_S1 = ", v3_1, " km/s")
```

```
r3_S1 = [-2909.9993379362068, 5078.968549237365, -13550.765972637573] km
v3_S1 = [-1.8799054401191884, -4.510175011507853, -3.226192819655835] km/s
```

(d) At exactly 6 h from the first observation, our spacecraft releases a secondary payload it carried within, of 500 kg, with a velocity v = [-6.0, 3,-1.6] km/s in ECI. Determine the classical orbital elements of the spacecraft after this release operation.

The position at that instant would be the same, however, because of the change in mass, the velocity of the aircraft would change. To obtain it we just need to apply conservation linear momentum: $p = m * v$.

In [ ]:
```
# Set the new position and speed of the spacecraft after the satellite is deployed
r4_0 = r3_0 # km

# Obtain the speed from the conservation of linear momentum (p = m*v),
# consider that the mass of the spacecraft changes from 2000 kg to 1500 kg
m_sc = 2000.0 # kg
m_sat = 500.0 # kg
m_new = m_sc - m_sat # kg
v_sat = MyVector([-6.0, 3.0,-1.6]) # km/s
v4_0 = (m_sc*v3_0 - m_sat*v_sat)/m_new # km/s

# Use the new state vector to obtain the new COE
state4 = MyStateVector(r4_0, v4_0)
coe4 = stateVector_to_COE(state4,mu_Earth)
println("RAAN = ", coe4.RAAN, " rad")
println("inc = ", coe4.inc, " rad")
println("omega = ", coe4.omega, " rad")
println("a = ", coe4.a, " km")
println("e = ", coe4.e)
println("theta = ", coe4.theta, " rad")

# Plot the new orbit
plot3 = plot_orbit_perifocal(coe4,R_Earth)
display(plot3)
```

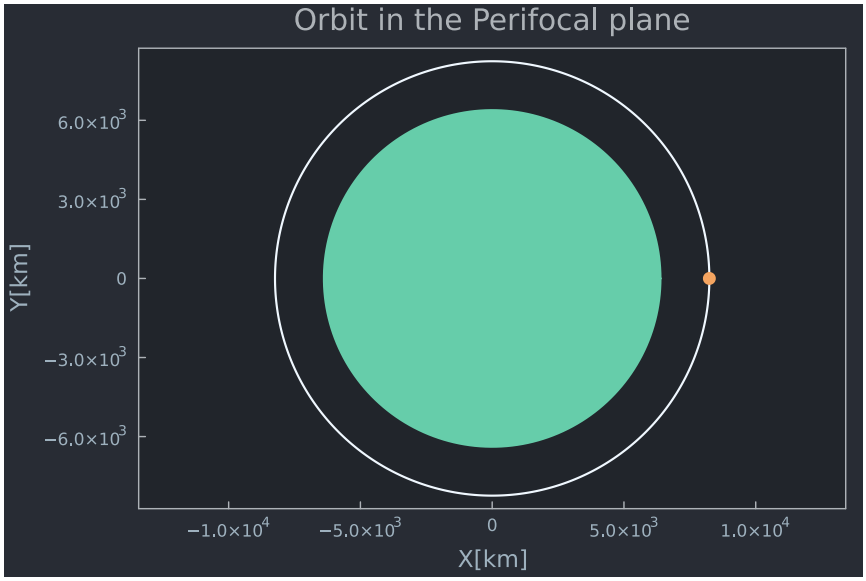



```
RAAN = 3.343426453001852 rad
inc = 0.4412667299383676 rad
omega = 1.5855775112078734 rad
a = 8778.734728431185 km
e = 0.06126011793632399
theta = 2.717266123784746 rad
```

(e) Compute the ΔV required to circularize the orbit of the spacecraft at its new pericenter.

The ΔV needed would just be the difference between the circular velocity of the new orbit and velocity at the periapsis, which can be calculated using the `speed_visViva` function.

In [ ]:
```
# First we need to obtain the rp4 of the last orbit computed
rp4 = coe4.a*(1.0 - coe4.e) # km
# The radius of the circular orbit is the same as the rp4
R_circ = rp4 # km
e_circ = 0.0
# Calculate the circular velocity and the speed of the spacecraft in the last orbit
v_circ = circularVelocity(R_circ,mu_Earth)
vp4 = speed_visViva(rp4,coe4.a,mu_Earth)
# Calculate the DeltaV
DeltaV = abs(v_circ - vp4) # km/s
println("DeltaV = ", DeltaV, " km/s")

# Validate the orbit by plotting it
coeCirc = MyCOE(RAAN=coe4.RAAN, inc=coe4.inc, omega=coe4.omega, a=R_circ, e=e_circ,
                                                                    theta=0.0)
plot4 = plot_orbit_perifocal(coeCirc,R_Earth)
display(plot4)
```

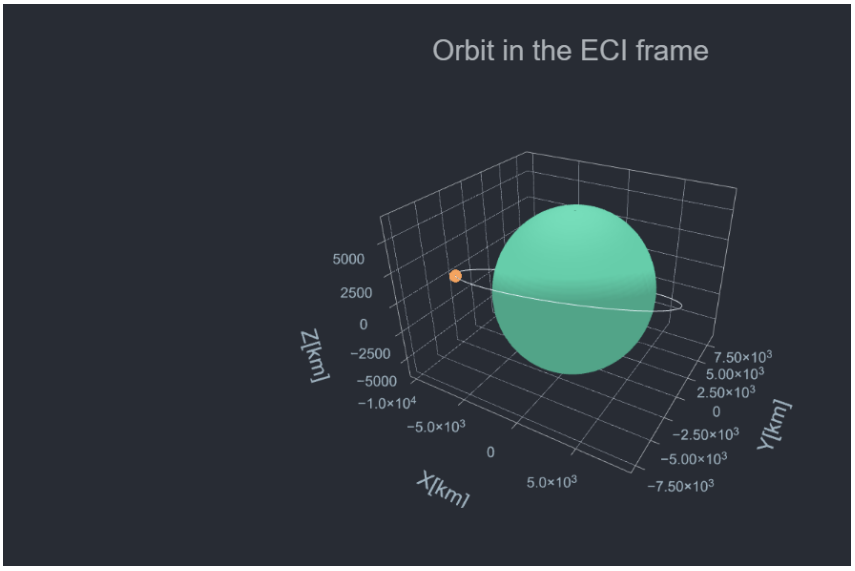


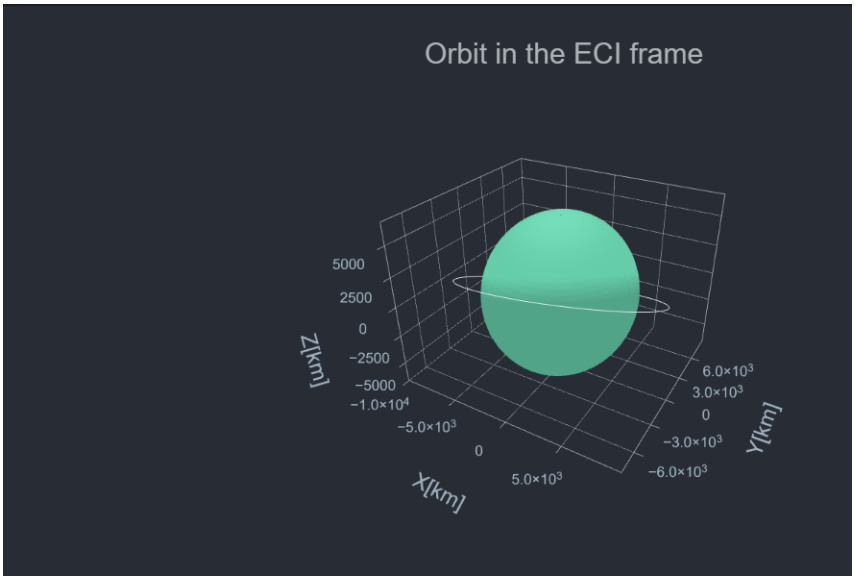

```
DeltaV = 0.20985759248770997 km/s
```

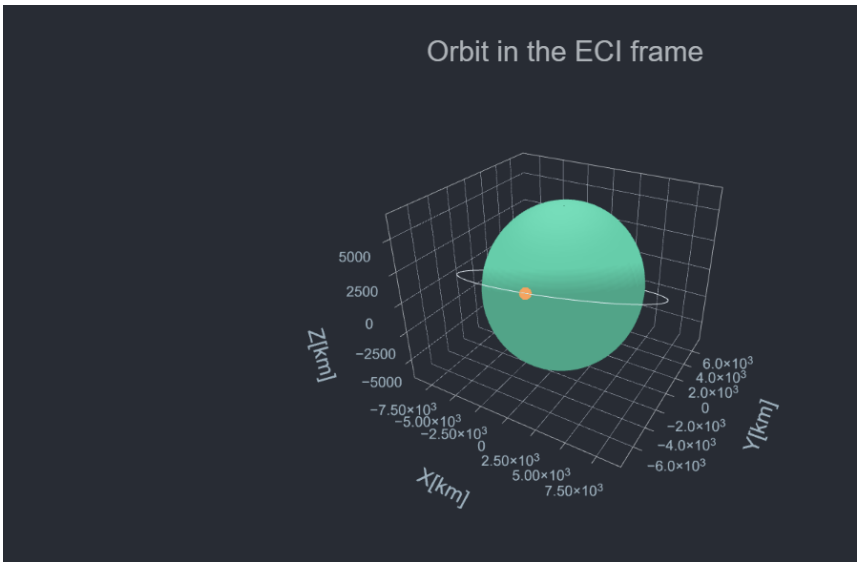
As expected the ΔV needed is quite small as the eccentricity of the previous orbit was very close already to zero.

Now, to further visualize the problem, let's plot in 3D all the different orbits obtained. Using the `plotly()` backend we can rotate and move around the figure to see the orbit from different perspectives.

In [ ]:
```
####### PLOTTING ALL THE ORBITS IN 3D ########
#using PlotlyJS, PlotlyBase
plotInitialOrbit_ECI = plot_orbit_ECI(coe, R_Earth, mu_Earth)
display(plotInitialOrbit_ECI)
plotAfterEjection_ECI = plot_orbit_ECI(coe4, R_Earth, mu_Earth)
display(plotAfterEjection_ECI)
plotCircular_ECI = plot_orbit_ECI(coeCirc, R_Earth, mu_Earth)
display(plotCircular_ECI)
```


Orbit in the ECI frame
5000
2500
0
−2500
−5000
Z[km]
−1.0×10^4
−5.0×10^3
0
5.0×10^3
X[km]
7.50×10^3
5.00×10^3
2.50×10^3
0
−2.50×10^3
−5.00×10^3
−7.50×10^3
Y[km]


Orbit in the ECI frame
5000
2500
0
−2500
−5000
Z[km]
−1.0×10^4
−5.0×10^3
0
5.0×10^3
X[km]
6.0×10^3
3.0×10^3
0
−3.0×10^3
−6.0×10^3
Y[km]


Orbit in the ECI frame
5000
2500
0
−2500
−5000
Z[km]
−7.50×10³
−5.00×10³
−2.50×10³
0
2.50×10³
5.00×10³
7.50×10³
X[km]
6.0×10³
4.0×10³
2.0×10³
0
−2.0×10³
−4.0×10³
−6.0×10³
Y[km]

As it can be seen, most of the functions required to solve this exercise were already implemented in our library, and those that were not could at least use some of the functions allowing for a faster implementation. The exercise was solved successfully, taking to run a total of 3 min 26 s, of which more than 2 minutes were to activate the environment and precompile the package. This is because of having to precompile the heavy package of Plots. However, this time would be significantly reduced in the consequent runs, when the package has already loaded.

This sample problem highlights the importance of the Ideal Two-Body Problem functions implemented, as well as the visualization of the orbital plots, which allows the student to give a tangible meaning to the values obtained.

5.2.2. Problem 2 - Space mining mission

HOMEWORK: EXERCISE 2 - Space mining mission

It is March 20, 2053. A space mining mission has been established in Europa, Jupiter's moon. From there, they intend to visit all other major moons in the search for valuable resources for space colonization. The first target is Ganymede, the next of the four Galilean moons.

As the mission commanders, you must use the patched-conics method and Lambert's algorithm with elliptic trajectories to determine the best departure time to Ganymede, between the present date and for the next two weeks, and best flight duration, to achieve the lowest mission ΔV .

(a) Implement the following functions, using units of km, s, and rad for the inputs and outputs:

i. `[a,e,Omega,i,omega,theta]=rv2coe(r,v,mu)` ,

ii. `[r,v] = coe2rv(a,e,Omega,i,omega,theta,mu)` ,

iii. `[E] = theta2E(theta,e)` ,

iv. `[theta] = E2theta(E,e)` ,

v. `[M] = E2m(E,e)` ,

vi. `[E] = M2E(M,e,E0,maxiter,tol)` ,

for the elliptic two body problem, as we have done in class. Note that in the last one you need to implement a solver. Use Newton-Raphson's method.

These are the equivalent functions we would have with AstrodynamicsEdu:

i. `coe = stateVector_to_COE(stateVector,mu)`

ii. `stateVector = COE_to_stateVector(coe,mu)`

iii. `E = trueAnomaly_to_eccentricAnomaly(stateVector,mu,theta)`

iv. `theta = eccentricAnomaly_to_trueAnomaly(stateVector,mu,E)`

v. `Me = eccentricAnomaly_to_meanAnomaly(stateVector,mu,E)`

vi. `E = meanAnomaly_to_eccentricAnomaly(stateVector,mu,E)`

In [ ]:
```
using Pkg
# Pkg.activate(".")
Pkg.activate("C:\\Users\\alici\\Desktop\\Julia\\HOMEWORK")
Pkg.instantiate()
Pkg.add(url="https://github.com/AliciaSBa/AstrodynamicsEdu.jl.git")

using Roots
using AstrodynamicsEdu
using LinearAlgebra
```

```
 Activating project at `C:\Users\alici\Desktop\Julia\HOMEWORK`
   Updating git-repo `https://github.com/AliciaSBa/AstrodynamicsEdu.jl.git`
   Updating registry at `C:\Users\alici\.julia\registries\General.toml`
  Resolving package versions...
No Changes to `C:\Users\alici\Desktop\Julia\HOMEWORK\Project.toml`
No Changes to `C:\Users\alici\Desktop\Julia\HOMEWORK\Manifest.toml`
```

(b) Use the NASA JPL Horizons ephemeris server to find the position and velocity of Europa and Ganymede with respect to Jupiter, in a jupiter-centered, ecliptic, pseudoinertial reference frame, at 00:00 March 20, 2053. This will be t = 0; the state of the moons at later times shall be computed using the 2BP functions from the previous point. You should also use Horizons to find the gravitational parameter μ of Europa and Ganymede (they are not available in the Course Formulary).

The radius and gravitational parameter of Jupiter are already in the AstrodynamicsEdu library as `mu_Jupiter` and `R_Jupiter`

In [ ]:
```
# Gravitational parameters and radius of the moons of Jupiter
    # (that do not appear in the AstroConstants module)
mu_Europa = 3202.71 # km^3/s^2
mu_Ganymede = 9887.83 # km^3/s^2
R_Europa = 1560.8 # km
R_Ganymede = 2631.2 # km

# Positions of the moons with respect to Jupiter at the specified date
    # (20 March 2053 00:00)
r0_Europa = MyVector([3.660674324064600E+05, -5.574382118933767E+05,
                                -1.130028161238341E+04]) # km
v0_Europa = MyVector([1.160430186118396E+01, 7.496163341137031E+00,
                                5.325632226511305E-01])  # km/s
state0_Europa = MyStateVector(r0_Europa,v0_Europa)
r0_Ganymede = MyVector([-1.574197237263062E+05, 1.056563893373634E+06,
                                3.441445759743074E+04]) # km
v0_Ganymede = MyVector([-1.077978381266393E+01, -1.575638346889765E+00,
                                -2.390760437391640E-01]) # km/s
state0_Ganymede = MyStateVector(r0_Ganymede,v0_Ganymede)
```

```
MyStateVector(MyVector([-157419.7237263062, 1.0565638933736346e6, 34414.45759743074]), MyVector
([-10.77978381266393, -1.575638346889765, -0.239076043739164]))
```

(c) Implement a function `[v1,v2] = Lambert(r1,r2,t,s,mu,maxiter,tol)` that solves the Lambert problem for two position vectors r1, r2 in km and a time of flight t in s. In this function, s is a flag that takes the value +1 for a short arc transfer and -1 for a long arc transfer, mu is the gravitational parameter of the 2BP, maxiter is the maximum number of iterations to consider in the solver, and tol is the solver tolerance. The function shall return the velocity vectors v1 and v2 in km/s at the beginning and the end of the transfer. It shall only solve for elliptic transfers, and raise an error if no elliptic transfer is possible for the given time of flight.

The functions that we already have are:

`coe1,coe2 = Lambert_solve(r1,r2,tF,mu,k)`

`coe1,coe2 = Lambert_conic(r1,r2,eT,k)`

`err = error_function(mu, r1, r2, eT, tf, k)`

The functions that we need:

`v1,v2 = Lambert(r1,r2,tF,s,mu,maxiter,tol)`

note: s = +1(short arc) or -1(long arc)
        v1: velocity vector @ beginning transfer
        v2: velocity vector @ end transfer
Only solves for elliptic transfers.

We would use: `coe1,coe2 = Lambert_conic(r1,r2,eT,k)`
`stateVector1 = COE_to_stateVector(coe1,mu)`
`v1 = stateVector.v`
`stateVector2 = COE_to_stateVector(coe2,mu)`
`v2 = stateVector.v`

In [ ]:
```
function Lambert(r1::MyVector,r2::MyVector,tF::Float64,s,mu::Float64,
                 maxiter::Number,tol::Number)
    # Input:
        # r1: initial position vector in the inertial frame
        # r2: final position vector in the inertial frame
        # tF: time of flight
        # s: Boolean that activates the short way when true (1 or -1)
        # mu: gravitational parameter of the central body
        # maxiter: maximum number of iterations
        # tol: tolerance for the root-finding algorithm
    # Output:
        # v1: velocity vector at the beginning of the transfer arc
        # v2: velocity vector at the end of the transfer arc

    # Would need to modify Lambert_solve to take in the s,maxiter,tol arguments

    function error_function(mu, r1, r2, eT, tf, k)
        coe1, coe2 = Lambert_conic(r1, r2, eT, k)
        err = tf - timeOfFlight(coe1, coe2, mu)
        return err
    end

    function Lambert_solve2(r1::MyVector,r2::MyVector,tF::Float64,mu::Float64,
                    k::MyVector,tol::Number,maxiter::Number)
        # Input:
            # r1: initial position vector
            # r2: final position vector
            # tF: time of flight
            # mu: gravitational parameter
            # K: plane normal (only used when rr1 and rr2 are collinear)
        # Output:
            # coe1: Classical Orbital Elements of the connecting orbit at the initial time
            # coe2: Classical Orbital Elements of the connecting orbit at the final time

        #tol = 1.0e-8
        #maxiter = 1000
        # Calculate the fundamental eccentricity
        r1_norm = norm(r1)
        r2_norm = norm(r2)
        c = norm(r2 - r1)
        eF = -(r2_norm - r1_norm)/c
        eTP = sqrt(1.0 - eF^2)

        # Solve for the transverse eccentricity using the bisection method
        try
            f(eT) = error_function(mu,r1,r2,eT,tF,k)
            eT = find_zero(f, (-eTP*0.999,eTP*0.999), xtol=tol, maxevals=maxiter)
            coe1, coe2 = Lambert_conic(r1,r2,eT,k)
            return coe1, coe2
        catch
            #println("No elliptic transfer was found")
```

```
            coe1 = MyCOE(RAAN=NaN, inc=NaN, e=NaN, omega=NaN, theta=NaN, a=NaN)
            coe2 = MyCOE(RAAN=NaN, inc=NaN, e=NaN, omega=NaN, theta=NaN, a=NaN)
            return coe1, coe2
        end

    end

    coe1,coe2 = Lambert_solve2(r1,r2,tF,mu,k0,tol,maxiter)

    stateVector1 = COE_to_stateVector(coe1,mu)
    stateVector2 = COE_to_stateVector(coe2,mu)
    v1 = stateVector1.v
    v2 = stateVector2.v

    return v1,v2
end
```

```
Lambert (generic function with 1 method)
```

(d) Use the Lambert function to produce a porkchop plot of the characteristic energy v_1,rel^2 + v_2,rel^2 in km^2/s^2 as a function of the departure time and the time of flight in the considered window. Discuss your results and identify a preferred option for the trip.

The objective is to determine the transfer orbit with the lowest characteristic energy between Europa and Ganymede. To achieve this, a sensitivity analysis is performed by varying the departure date and travel time. For each combination, the Lambert function will be used, which solves for the velocity at the beginning and at the end of the transfer orbit. Because of the dependency on time of the positions, to compute the new state vectors in each instance of time, the relationships between the anomalies and time will be used.

In the calculation of the relative energy required for the transfer, the short arc and long arc transfers are compared, and the option with the minimum energy is selected for each combination of launch and travel times.

In [ ]:
```
# Initial true anomaly of the moons
coe0_Europa = stateVector_to_COE(state0_Europa,mu_Jupiter)
coe0_Ganymede = stateVector_to_COE(state0_Ganymede,mu_Jupiter)

# Initial mean anomaly of each moon
M0_Europa = trueAnomaly_to_meanAnomaly_E(state0_Europa, mu_Jupiter,
                coe0_Europa.theta)
M0_Ganymede = trueAnomaly_to_meanAnomaly_E(state0_Ganymede, mu_Jupiter,
                coe0_Ganymede.theta)

# Mean angular rate of each moon
n_Europa = sqrt(mu_Jupiter/coe0_Europa.a^3)
n_Ganymede = sqrt(mu_Jupiter/coe0_Ganymede.a^3)

# Set the loop that will solve Lambert's problem for each travel time
maxiter = 1000
tol = 1e-8
t_launch = range(0,stop=14,length=200)*24*3600 # s
t_travel = range(1,stop=8,length=1000)*24*3600 # s

# Initialize the vectors
pork_plot = zeros(length(t_travel),length(t_launch))
r_Europa = zeros(3,length(t_launch))
v_Europa = zeros(3,length(t_launch))
r_Ganymede = zeros(3,length(t_launch),length(t_travel))
v_Ganymede = zeros(3,length(t_launch),length(t_travel))
```

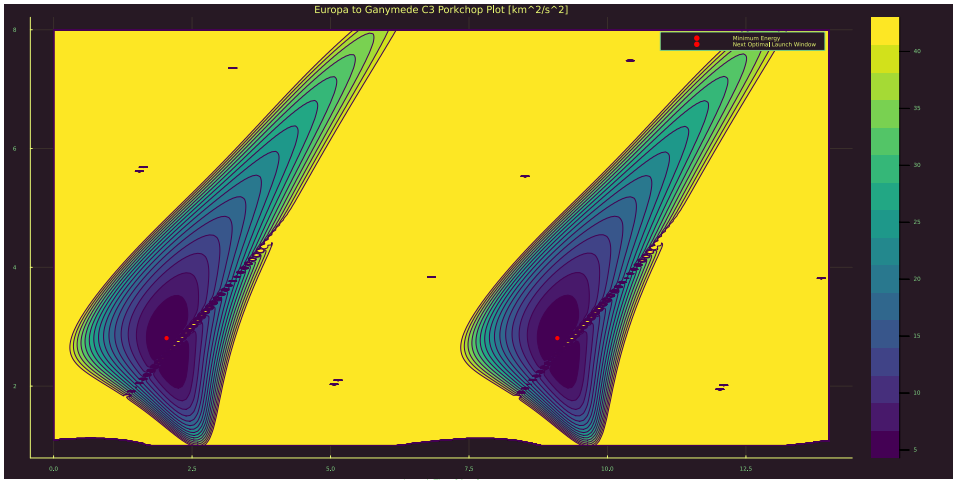
```
for i=1:length(t_launch)
    # Initial positions of Europa
    M1_Europa = M0_Europa + t_launch[i]*n_Europa
    theta1_Europa = meanAnomaly_to_trueAnomaly_E(state0_Europa,mu_Jupiter,
                            M1_Europa)
    coe1_Europa = MyCOE(RAAN=coe0_Europa.RAAN, inc=coe0_Europa.inc,
                        e=coe0_Europa.e, omega=coe0_Europa.omega,
                        theta=theta1_Europa, a=coe0_Europa.a)
    state1_Europa = COE_to_stateVector(coe1_Europa,mu_Jupiter)
    r_Europa[:,i] = state1_Europa.r.c0
    v_Europa[:,i] = state1_Europa.v.c0

    # Iterate for different travel times
    for j=1:length(t_travel)
        # Final positions of Ganymede
        M1_Ganymede = M0_Ganymede + (t_travel[j]+t_launch[i])*n_Ganymede
        theta1_Ganymede = meanAnomaly_to_trueAnomaly_E(state0_Ganymede,mu_Jupiter,
                                M1_Ganymede)
        coe1_Ganymede = MyCOE(RAAN=coe0_Ganymede.RAAN, inc=coe0_Ganymede.inc,
                                e=coe0_Ganymede.e, omega=coe0_Ganymede.omega,
                                theta=theta1_Ganymede, a=coe0_Ganymede.a)
        state1_Ganymede = COE_to_stateVector(coe1_Ganymede,mu_Jupiter)
        r_Ganymede[:,i,j] = state1_Ganymede.r.c0
        v_Ganymede[:,i,j] = state1_Ganymede.v.c0

        # Solve Lambert's problem for the short distance and long distance cases
        vs1_Europa, vs1_Ganymede = Lambert(MyVector(r_Europa[:,i]),
                MyVector(r_Ganymede[:,i,j]),t_travel[j],1,mu_Jupiter,maxiter,tol)
        vns1_Europa, vns1_Ganymede = Lambert(MyVector(r_Europa[:,i]),
                MyVector(r_Ganymede[:,i,j]),t_travel[j],-1,mu_Jupiter,maxiter,tol)

        C3s1 = norm(vs1_Europa.c0 - v_Europa[:,i])^2 +
                norm(vs1_Ganymede.c0 - v_Ganymede[:,i,j])^2
        C3ns1 = norm(vns1_Europa.c0 - v_Europa[:,i])^2 +
                norm(vns1_Ganymede.c0 - v_Ganymede[:,i,j])^2
        try
            if C3s1 < C3ns1
                pork_plot[j,i] = C3s1
            else
                pork_plot[j,i] = C3ns1
            end
        catch
            pork_plot[j,i] = NaN
        end
    end
end

# Find the minimum energy and get the launch time and travel time
minC3 = minimum(skipmissing(isnan(x) ? missing : x for x in pork_plot))
println("Minimum value:", minC3)
positions = findall(x -> x == minC3, pork_plot)
println("Positions: ", positions)

t_launch_min = t_launch[positions[1][2]]/(24*3600)
t_travel_min = t_travel[ positions[1][1]]/(24*3600) # days

# Calculate the synodic period to determine the next optimal launch window
t_synodic = 2*pi/(abs(n_Europa - n_Ganymede))/(24*3600) # days
t_launch_next = t_launch_min + t_synodic
t_travel_next = t_travel_min
```

```
Minimum value:4.3003262737889685
Positions: CartesianIndex{2}[CartesianIndex(259, 30)]
```

```
2.8078078078078077
```

The analysis results are presented graphically in a Porkchop plot, which illustrates the characteristic energy for different combinations of departure date and travel time. We can use the `plot_porkchop` function of AstrodynamicsEdu.

In [ ]:
```
# Plot the contour of the relative energy
figure = plot_porkchop(t_launch/(24*3600),t_travel/(24*3600),pork_plot,
                       t_synodic)
title!("Europa to Ganymede C3 Porkchop Plot [km^2/s^2]")
xlabel!("Launch Time [days]")
ylabel!("Travel Time [days]")
display(figure)
```



For the reminder of the problem, consider your preferred option for the trip:

(e) For the departure from Europa, assume that your launchpad is on the moon's equator at the datum. Determine the ΔV to launch into a circular parking orbit 200 km high, ignoring any atmosphere. Then, compute the semimajor axis and eccentricity of the departure hyperbola, and the ΔV for injection. You can assume that the launch time and the plane of the parking orbit allow a direct tangential burn into the departure hyperbola.

1st Maneuver: From Europa surface to a circular parking orbit we would need 2 burns. The first for going into a Hohmann transfer and the second to circularize the orbit.

$$\Delta v_1 = \sqrt{\frac{2\mu}{R_{Europa}} - \frac{\mu}{a_{tr}}}$$

$$\Delta v_2 = \sqrt{\frac{\mu}{r_{parking}}} - \sqrt{\frac{2\mu}{r_{parking}} - \frac{\mu}{a_{tr}}}$$

$$\Delta v_{parking} = \Delta v_1 + \Delta v_2$$

In [ ]:
```
# 1. Delta-v to launch from Europa surface to circular parking orbit (2 burns)
r_park = R_Europa + 200 # km
v_park = circularVelocity(r_park,mu_Europa)
a_tr = (r_park + R_Europa)/2
```

```
deltaV1_park = speed_visViva(R_Europa,a_tr,mu_Europa)
deltaV2_park = abs(speed_visViva(r_park,a_tr,mu_Europa) - v_park)
deltaV_park = deltaV1_park + deltaV2_park
println("Delta-v to launch from Europa surface to circular parking orbit: ",
            deltaV_park, " km/s")
```

```
Delta-v to launch from Europa surface to circular parking orbit: 1.5161975221417654 km/s
```

2nd Maneuver: Hyperbolic trajectory. We must use the the absolute velocity of the spacecraft at the beginning of Lambert transfer and the velocity of Europa to find the excess hyperbolic velocity. From here we can obtain the semi-major axis of the hyperbolic orbit. And using the relationship with the periapsis, obtain the eccentricity of the hyperbolic orbit.

$$v_h = v_{s/c} - v_{Europa} = \sqrt{-\frac{\mu}{a}}$$

$$r_p = a(1-e)$$

$$\Delta v_{hyper} = \sqrt{\frac{2\mu}{r_{parking}} - \frac{\mu}{a_{hyper}}} - \sqrt{\frac{\mu}{r_{parking}}}$$

Total Departure:

$$\Delta v_{departure} = \Delta v_{parking} + \Delta v_{hyper}$$

In [ ]:
```
# 2. Compute a and e of the departure hyperbola, and Delta-v for injection

# Considering preferred option for the trip:
r_opt_EU = r_Europa[:,positions[1][2]]
v_opt_EU = v_Europa[:,positions[1][2]]
r_opt_GA = r_Ganymede[:,positions[1][2],positions[1][1]]
v_opt_GA = v_Ganymede[:,positions[1][2],positions[1][1]]

vs1_Europa, vs1_Ganymede = Lambert(MyVector(r_opt_EU),MyVector(r_opt_GA),
                            t_travel_min*24*3600,1,mu_Jupiter,maxiter,tol)
vns1_Europa, vns1_Ganymede = Lambert(MyVector(r_opt_EU),MyVector(r_opt_GA),
                            t_travel_min*24*3600,-1,mu_Jupiter,maxiter,tol)
C3s1 = norm(vs1_Europa.c0 - v_Europa[:,end])^2 +
                    norm(vs1_Ganymede.c0 - v_Ganymede[:,end,end])^2
C3ns1 = norm(vns1_Europa.c0 - v_Europa[:,end])^2 +
                    norm(vns1_Ganymede.c0 - v_Ganymede[:,end,end])^2

if C3s1<C3ns1
    vopt_EU = vs1_Europa.c0
    vopt_GA = vs1_Ganymede.c0
else
    vopt_EU = vns1_Europa.c0
    vopt_GA = vns1_Ganymede.c0
end

# Compute a and e of the departure hyperbola
a_hyper = -mu_Europa/norm(vopt_EU - v_opt_EU)^2
println("a_hyper: ", a_hyper)
e_hyper = 1 - r_park/a_hyper
println("e_hyper: ", e_hyper)
v_hyper = speed_visViva(r_park,a_hyper,mu_Europa)
println("v_hyper: ", v_hyper)
deltaV_hyper = abs(v_hyper - v_park)
println("Delta-v for injection: ", deltaV_hyper, " km/s")

println("")
```

```
deltaV_departure = deltaV_park + deltaV_hyper
println("Total Delta-v for departure from Europa: ", deltaV_departure, " km/s")
```

```
a_hyper: -1306.8836643189525
e_hyper: 2.3473272702644072
v_hyper: 2.4674756806346116
Delta-v for injection: 1.1188115932740414 km/s

Total Delta-v for departure from Europa: 2.635009115415807 km/s
```

(f) For the arrival at Ganymede, determine the impact parameter for an arrival hyperbola with pericenter at 300 km from the moon's datum. Determine the ΔV needed to acquire a circular orbit at that altitude.

Final Maneuver: Arrival at Ganymede.

$$v_h = v_{s/c} - v_{Ganymede} = \sqrt{-\frac{\mu}{a}}$$

$$r_p = a(1-e)$$

Impact parameter: $B = -a\sqrt{e^2-1}\Delta v_{arrival} = \Delta v_{parking,arrival} + \Delta v_{hyper,arrival}$$

In [ ]:
```
# 1. Impact parameter for arrival hyperbola
rp_GA = R_Ganymede + 300 # km
a_hyper_arrival = -mu_Ganymede/norm(vopt_GA - v_opt_GA)^2
e_hyper_arrival = 1 - rp_GA/a_hyper_arrival
B_arrival = -a_hyper_arrival*sqrt(e_hyper_arrival^2 - 1)
println("Impact parameter for arrival hyperbola: ", B_arrival, " km")
println("Eccentricity of arrival hyperbola: ", e_hyper_arrival)
println("Semi-major axis of arrival hyperbola: ", a_hyper_arrival, " km")
println("")

# 2. Compute Delta-v to acquire circular orbit at that altitude
v_hyper_arrival = speed_visViva(rp_GA,a_hyper_arrival,mu_Ganymede)
v_park_arrival = circularVelocity(rp_GA,mu_Ganymede)
deltaV_arrival = abs(v_hyper_arrival - v_park_arrival)
println("Delta-v to acquire circular orbit at Ganymede: ", deltaV_arrival,
                                                          " km/s")
println("")
```

```
Impact parameter for arrival hyperbola: 6319.062957954673 km
Eccentricity of arrival hyperbola: 1.5483287110525419
Semi-major axis of arrival hyperbola: -5345.6985580299615 km

Delta-v to acquire circular orbit at Ganymede: 1.0952870206913197 km/s
```

Although we are not explicitly asked to compute, it is interesting to find the total Δv that the mission would require, as this would most important result in the preliminary evaluation of the mission. Depending of this value, we will need to choose one propulsion system or another.

In [ ]:
```
# Total Delta-v of the mission
deltaV_tot = deltaV_departure + deltaV_arrival
println("Total Delta-v of the mission: ", deltaV_tot, " km/s")
```

```
Total Delta-v of the mission: 3.7302961361071265 km/s
```

This result is relatively low, which is to be expected as we are we are performing the mission inside the the time window necessary to obtain the minimum Δv.

From this problem, the main functions used were those relating to the Lambert problem, although they had to be adapted to fit the problem's needs. As the problem required to only use elliptical orbit transfers, so only the elliptical solutions were valid. In addition, the relationships between anomalies were used, as well as the vis-viva equation, among others.

Once again, the visualization capabilities of the package were shown now through the `plot_porkchop` function. With a clear and aesthetic color scale the different energy levels are provided; the plot also includes a clear legend generated in the appropriate range. This helps interpret the plot easily. It must be noted that the quality of the porkchop plot obtained with our function is better than the one shown, however some of this quality is lost when transforming the Jupyter notebook into a PDF.

With respect to the computational time, for this exercise it took 8 min 33 s to run. However, as the package had already been loaded inside the folder, the precompilation only lasted around 1 min. What took longer was the loop to solve Lambert's problem, which lasted over 7 min. This is not unreasonable, as there is a combination of 200 x 1000 cases for which the Lambert solver has to be executed.

5.2.3. Problem 3 - Space debris

HOMEWORK: EXERCISE 3 - Space Debris

One of our retired satellites was not properly passivized at end of life, and has suffered an internal explosion that has generated space debris. The orbit of the satellite at the time of explosion was given by: ζp = 350 km; e = 0.1; Ω = 195 deg; i = 54 deg; ω = 235 deg; θ = 20 deg.

In [ ]:
```
# Data:
zeta_p = 350.0 # km
e_sat = 0.1
Omega_sat = deg2rad(195.0) # rad
inc_sat = deg2rad(54.0) # rad
omega_sat = deg2rad(235.0) # rad
theta_sat = deg2rad(20.0) # rad
```

```
0.3490658503988659
```

After the explosion, 30 pieces of debris were created with the same initial position r0 as the satellite, but with a dispersion2 in initial velocity v0 that follows a normal distribution with mean equal to the velocity of the satellite, and standard deviation 100 m/s in each direction. The ballistic coefficient of each fragment follows lognormal distribution, with μ = ln(20 kg/m2) and σ = 0.5.

In [ ]:
```
using Pkg
# Pkg.activate(".")
Pkg.activate("C:\\Users\\alici\\Desktop\\Julia\\EXERCISE3")
Pkg.instantiate()
Pkg.add(url="https://github.com/AliciaSBa/AstrodynamicsEdu.jl.git")
Pkg.update()

using AstrodynamicsEdu
using LinearAlgebra
using DifferentialEquations
```

In [ ]:
```
N = 30
# Calculate the initial position and velocity vectors
r_p = zeta_p + R_Earth
a_sat = r_p/(1-e_sat)
coe_sat = MyCOE(RAAN = Omega_sat, inc = inc_sat, omega = omega_sat, a = a_sat,
                                                    e = e_sat, theta = theta_sat)
state_sat = COE_to_stateVector(coe_sat,mu_Earth)
# Debris position and velocity
r_0 = state_sat.r
mu_v = state_sat.v
sigma_v = 0.1 # km/s (in each direction)
# Ballistic coefficient - lognormal distribution
mu_bc = log(20.0)
sigma_bc = 0.5
```

```
0.5
```

We want to study how the orbits of the cloud of debris evolve in time, ignoring the possibility that they may subsequently collide among themselves.

(a) Code a function `[v,bc] = explosion(mu_v,sigma_v,mu_bc,sigma_bc)` that generates the (3 x N) array v and the (1 x N) array bc with the initial velocities (in km/s) and the ballistic coefficients (in kg/m2) of N pieces of debris, with distribution parameters `mu_v` and `sigma_v` for the velocity and `mu_bc` , `sigma_bc` for the mass as described above.

In [ ]:
```
using Random
# Set the random number generator's seed to 0, to ensure that subsequent runs of the
    # code produce the same random numbers
Random.seed!(0)
rng = Random.MersenneTwister(1234)

# Function to calculate velocity and Bc of each piece
function explosion(mu_v, sigma_v, mu_bc, sigma_bc)
    N = 30
    v = zeros(3, N)
    bc = zeros(1, N)

    # Generate 3 normal distributions (1 per velocity component)
    for i in 1:3
        for j in 1:N
            u1 = rand()
            u2 = rand()
            z1 = sqrt(-2*log(u1))*cos(2*pi*u2)
            v[i, j] = mu_v[i] + z1*sigma_v
        end
    end

    # Obtain the Ballistic coefficients lognormal distribution
    # First calculate the mean and variance of the underlying normal distribution
    mu = log(mu_bc^2/sqrt(sigma_bc^2 + mu_bc^2))
    sigma = sqrt(log(1 + (sigma_bc/mu_bc)^2))

    # Generate N random samples from the log-normal distribution with the
        #given mean and standard deviation
    bc[1, :] = (exp.(mu .+ sigma .* randn(N,1))).*10^6 # kg/km^2

    return v, bc
end

v0,Bc = explosion(mu_v,sigma_v,mu_bc,sigma_bc)
```

```
println("Velocity of each piece: ",v0, " km/s")
println("Ballistic coefficient of each piece: ",Bc, " kg/km^2")
```

```
Velocity of each piece: [-7.6629393269444925 -7.738283858976623 -7.7173204909554585 -7.8474809
35733069 -7.630070452467509 -7.696254727298384 -7.87221879201388 -7.676887194957834 -7.7420593
119813965 -7.885801796923632 -8.070934180456844 -7.703701705334888 -7.901924028889331 -7.87760
7295483539 -7.838986571520096 -7.736517678641874 -7.916577725083215 -7.650507326615872 -7.8256
71641091026 -7.723903408710842 -7.765523239448662 -7.800280797947573 -7.940712435740529 -7.898
833678652749 -7.711711070725511 -7.859423467230367 -7.764918705014169 -7.823957734116187 -7.78
2450204588986 -8.027895560280653; -0.5734555253366086 -0.6595586156075784 -0.7092307321905793
-0.5822622184572072 -0.7054408539185247 -0.6944872946529371 -0.8006808170375868 -0.69404460114
82864 -0.6406423646571583 -0.7696529728107184 -0.7592531668273151 -0.7473968976855457 -0.84392
77347424528 -0.5720234645409044 -0.6265175158026375 -0.8091882834222964 -0.7070338923337296 -
0.7579143749824253 -0.8648002902894655 -0.5984475371268543 -0.7279945827555802 -0.649923291587
6587 -0.6183411034893357 -0.7638676950220178 -0.6000259826109311 -0.5528931755814932 -0.718297
0222515797 -0.7241943206951555 -0.8415146278523714 -0.6207993915686888; -1.7048073120622609 -
1.8047212975460851 -1.9357039700353258 -1.8576705596746446 -1.9155328087527639 -1.957638280276
7247 -1.838560129117264 -1.848795356509331 -1.940239090013048 -1.7461330133421722 -1.963524314
1912864 -2.046343227780515 -1.8383746478186571 -2.0104188673213814 -1.9467989258383283 -2.1059
601432223443 -1.7671448500154905 -1.7376846674679445 -1.7424697917613199 -1.908404271681895 -
1.8705664377017575 -1.72227449239178 -2.0755429087637385 -1.9088693093331681 -1.92692934683782
7 -1.8576700871589729 -1.9265515390024979 -1.8764807178934508 -1.9452058074777285 -1.808632145
8379022] km/s
Ballistic coefficient of each piece: [2.430723586335961e6 2.9054543372566802e6 3.2582204251140
33e6 2.538867165270325e6 2.071980201351082e6 3.2028641552824564e6 2.7802652379787965e6 2.32590
69977419465e6 2.378518664015541e6 2.1899764563880036e6 2.642568213486311e6 2.489284098402383e6
3.0355147398743 84e6 2.607242997399797e6 3.394098500040479e6 2.7357543505942053e6 2.76407808369
72333e6 2.711509853021118e6 3.8004484831795366e6 2.349247636981357e6 2.4882897989726495e6 2.84
0668652866057e6 2.6552409417419103e6 2.723477406032103e6 3.10251207839299e6 4.353135078049892e
6 3.733188007308912e6 2.1615818428855366e6 3.2582293192650992e6 3.2873022034945018e6] kg/km^2
```

(b) Implement a function `ap = drag_acceleration(zeta, v, bc)` that returns the perturbation acceleration vector (in km/s2) due to drag on an object at altitude zeta, with velocity vector v (in km/s) and ballistic coefficient BC. Assume a simple isothermal atmosphere model with scale height H = 50 km and ρ0 = 10^-8 kg/m3 at ζ0 = 100 km. To avoid issues with the drag acceleration, set it to NaNs when the altitude of the object is below 100 km (this will result in NaNs in the subsequent time integration below, and remove those points from the plots).

The equivalent equation in the AstrodynamicsEdu library would be: `ap_drag = drag_acceleration(alt,v,Bc)`

(c) Implement a function `ap = J2_acceleration(r)` that returns the perturbation acceleration vector (in km/s2) due to the J2 coefficient of the non-sphericity of the Earth on an object with position vector r in ECI (in km).

The equivalent equation in the AstrodynamicsEdu library would be: `ap_J2 = J2_acceleration(r)`

(d) Establish a Cowell propagator `[r,v] = cowell(r0,v0,bc,t,flag_drag,flag_J2)` that integrates the trajectory of multiple objects, with (3 x N) arrays r0 and v0 giving their initial positions and velocities. Use ode45 (with the right settings!) to perform the integration for each object. The vector t, of length M, is the vector of time instants at which the outputs must be provided. The outputs r and v are (3 x N x M) arrays with the integration results for each object. flag drag and flag J2 are Booleans that activate the corresponding perturbations when true.

This function exists in the AstrodynamicsEdu library by the same name: `r,v = cowell(r0, v0, bc, t, flag_drag, flag_J2)`, the only difference is that it does not use the ode45, but instead the

DifferentialEquations and the OrdinaryDiffEq package to integrate the ODE. The flags must be equalt to either `true` or `false`.

(e) Simulate the trajectories of the cloud of debris for 6 months (1) without perturbations (2) with the drag perturbation and (3) with both perturbations. For each case, generate plots of (i) perigee and apogee altitudes ζp, ζa vs orbital period τ (Gabbard plot), (ii) e vs a, and (iii) Ω vs a for the state of the cloud of debris on t = 0. Using different colors, overlay the state of the cloud of debris every 7 days afterward, up to 6 months. Discuss your results.

We have 3 different cases for which we have to simulate the cloud of debris:

Case 1: Without perturbations

Case 2: With the drag pertubation

Case 3: With both the drag and the J2 perturbation

This will be done by using the cowell method to propagate the trajectories of the cloud of debris of each case.

In [ ]:
```
# Time span (6 months)
M = 26
t_final = 6*30*24*60*60 # s
t = range(0.0,t_final,M) # s
t = collect(t)

# Case 1: Without perturbations
flag_drag = false
flag_J2 = false
r_c1 = zeros(3,M,N)
v_c1 = zeros(3,M,N)
for i=1:N
    v_0 = v0[:,i]
    r_c1[:,:,i], v_c1[:,:,i] = cowell(r_0.c0, v_0, Bc[i], t, flag_drag, flag_J2)
end

# Case 2: With Drag perturbation
flag_drag = true
flag_J2 = false
r_c2 = zeros(3,M,N)
v_c2 = zeros(3,M,N)
for i=1:N
    v_0 = v0[:,i]
    r_c2[:,:,i], v_c2[:,:,i] = cowell(r_0.c0, v_0, Bc[i], t, flag_drag, flag_J2)
end

# Case 3: With both perturbations
flag_drag = true
flag_J2 = true
r_c3 = zeros(3,M,N)
v_c3 = zeros(3,M,N)
for i=1:N
    v_0 = v0[:,i]
    r_c3[:,:,i], v_c3[:,:,i] = cowell(r_0.c0, v_0, Bc[i], t, flag_drag, flag_J2)
end
```

In [ ]:

```
#Predifine the arrays
a1 = zeros(M,N)
e1 = zeros(M,N)
RAAN1 = zeros(M,N)
hp1 = zeros(M,N)
ha1 = zeros(M,N)
tau1 = zeros(M,N)
a2 = zeros(M,N)
e2 = zeros(M,N)
RAAN2 = zeros(M,N)
hp2 = zeros(M,N)
ha2 = zeros(M,N)
tau2 = zeros(M,N)
a3 = zeros(M,N)
e3 = zeros(M,N)
RAAN3 = zeros(M,N)
hp3 = zeros(M,N)
ha3 = zeros(M,N)
tau3 = zeros(M,N)

for j=1:N
    for i=1:M
        # Case 1 (no perturbations)
        coe1 = stateVector_to_COE(MyStateVector(MyVector(r_c1[:,i,j]),
                                        MyVector(v_c1[:,i,j])),mu_Earth)
        a1[i,j] = coe1.a
        e1[i,j] = coe1.e
        RAAN1[i,j] = coe1.RAAN
        hp1[i,j] = coe1.a * (1 - coe1.e) - R_Earth
        ha1[i,j] = coe1.a * (1 + coe1.e) - R_Earth
        tau1[i,j] = 2*pi*sqrt(coe1.a^3/mu_Earth)
        # Case 2 (drag perturbation)
        coe2 = stateVector_to_COE(MyStateVector(MyVector(r_c2[:,i,j]),
                                        MyVector(v_c2[:,i,j])),mu_Earth)
        a2[i,j] = coe2.a
        e2[i,j] = coe2.e
        RAAN2[i,j] = coe2.RAAN
        if coe2.a < 0
            tau2[i,j] = NaN
            hp2[i,j] = NaN
            ha2[i,j] = NaN
        else
            tau2[i,j] = 2*pi*sqrt(coe2.a^3/mu_Earth)
            hp2[i,j] = coe2.a * (1 - coe2.e) - R_Earth
            ha2[i,j] = coe2.a * (1 + coe2.e) - R_Earth
        end
        # Case 3 (drag and J2 perturbations)
        coe3 = stateVector_to_COE(MyStateVector(MyVector(r_c3[:,i,j]),
                                        MyVector(v_c3[:,i,j])),mu_Earth)
        a3[i,j] = coe3.a
        e3[i,j] = coe3.e
        RAAN3[i,j] = coe3.RAAN

        if coe3.a < 0
            tau3[i,j] = NaN
            hp3[i,j] = NaN
            ha3[i,j] = NaN
        else
            tau3[i,j] = 2*pi*sqrt(coe3.a^3/mu_Earth)
            hp3[i,j] = a3[i,j] * (1 - e3[i,j]) - R_Earth
            ha3[i,j] = a3[i,j] * (1 + e3[i,j]) - R_Earth
        end
    end
```

```
end

n0 = sqrt(mu_Earth/a_sat^3)
tau = 2*pi/n0
t_period = t/tau
```

```
26-element Vector{Float64}:
    0.0
   96.860828501153
  193.721657002306
  290.582485503459
  387.443314004612
  484.30414250576496
  581.164971006918
  678.025799508071
  774.886628009224
  871.747456510377
    ⋮
 1646.634084519601
 1743.494913020754
 1840.355741521907
 1937.2165700230598
 2034.0773985242129
 2130.938227025366
 2227.7990555265187
 2324.659884027672
 2421.520712528825
```

Now, in order to be able to intepret the results we are going to generate the following plots for each case:

(i) perigee and apogee altitudes ζp, ζa vs orbital period τ (Gabbard plot)

(ii) e vs a

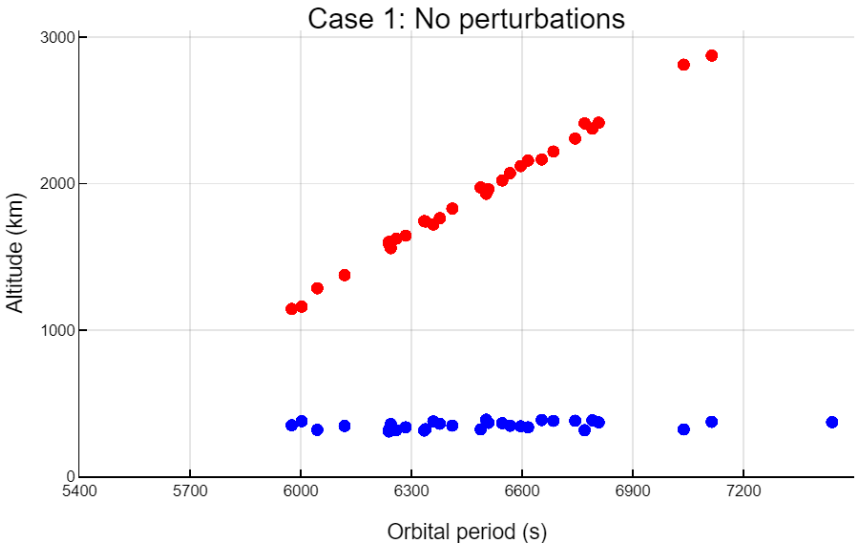
(iii) Ω vs a

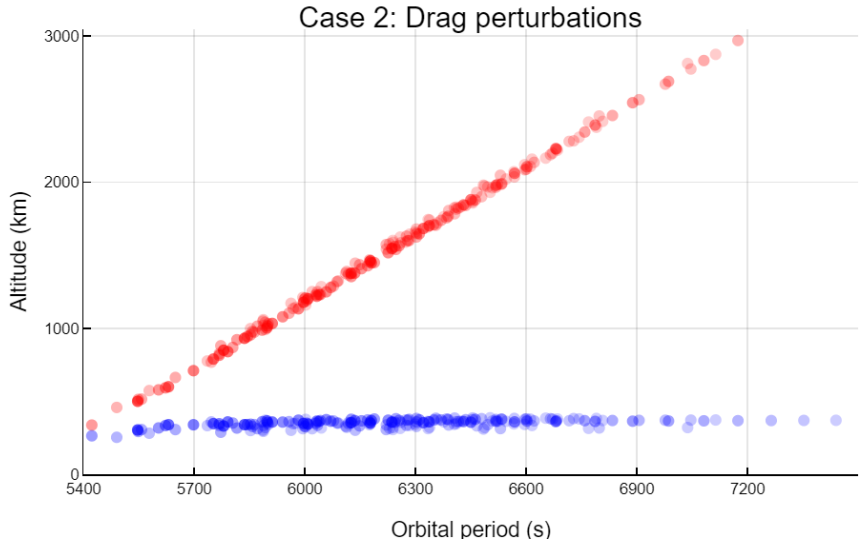
For (i), the apogee is in red and the perigee altitude in blue, and the scatter points become more faded with time. Where for (ii) and (iii), each color represents a different piece of debris, and they are more faded as more time has passed. This way, the later states of space debris are the more faded points, and we can see their evolution.

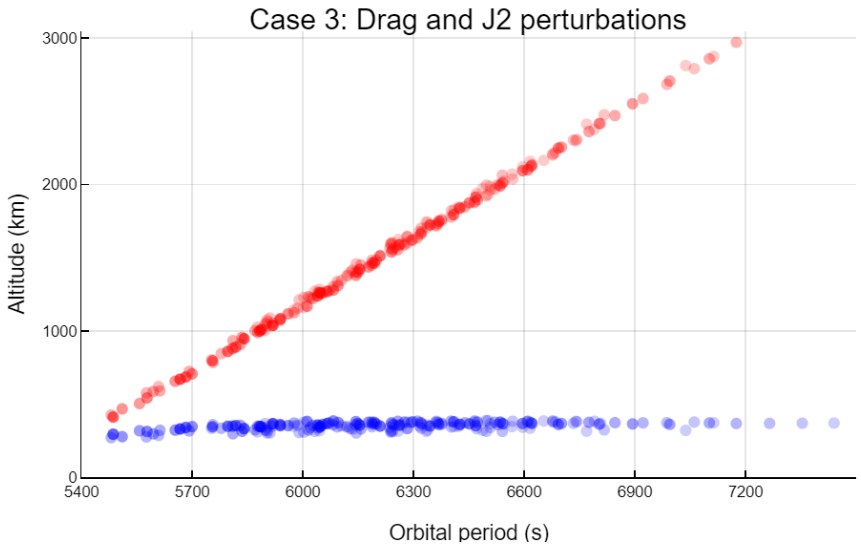
In [ ]:
```
using Plots
plotly()

# 1. Gavvard plots

function plot_Gavvard(tau,ha,hp)
    # Define the color gradient
    function fade_colorBlue(alpha)
        RGBA(0, 0, 1, alpha)
    end

    function fade_colorRed(alpha)
        RGBA(1, 0, 0, alpha)
    end

    # Calculate the alpha values for each point
    alphas = LinRange(0.2, 1, length(tau))
    # Apogee altitude
    scatter!(tau,ha,label="ha",color=fade_colorRed.(alphas), markerstrokewidth=0,
                    legend=false, xlims=(5400,7500), ylims=(0,3050))
```

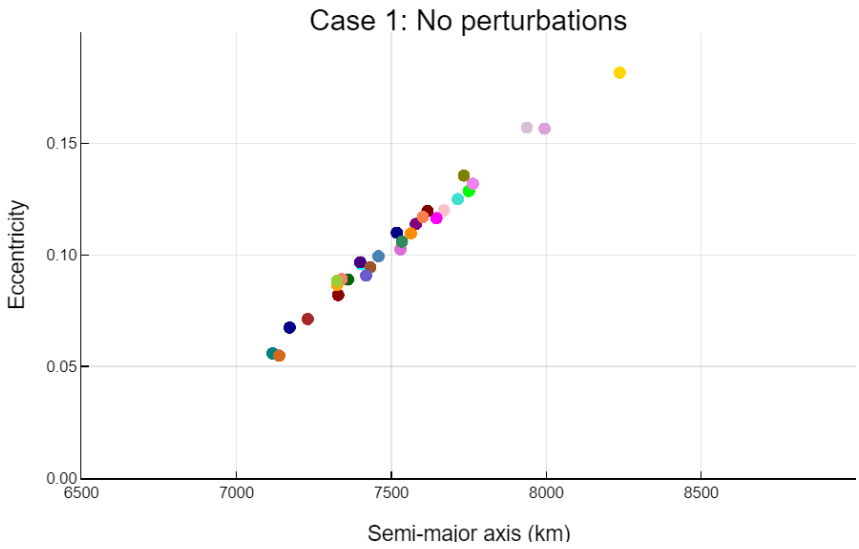
```
    # Perigee altitude
    scatter!(tau,hp,label="hp",color = fade_colorBlue.(alphas), markerstrokewidth=0,
                        legend = false, xlims=(5400,7500), ylims=(0,3050))
    xlabel!("Orbital period (s)")
    ylabel!("Altitude (km)")
end

figure1 = plot()
for i = 1:N
    title!(figure1, "Case 1: No perturbations")
    plot_Gavvard(tau1[:,i],ha1[:,i],hp1[:,i])
end
display(figure1)

figure2 = plot()
for i = 1:N
    title!(figure2, "Case 2: Drag perturbations")
    plot_Gavvard(tau2[:,i],ha2[:,i],hp2[:,i])
end
display(figure2)

figure3 = plot()
for i = 1:N
    title!(figure3, "Case 3: Drag and J2 perturbations")
    plot_Gavvard(tau3[:,i],ha3[:,i],hp3[:,i])
end
display(figure3)

# Define N different colors
colors = [:teal, :darkred, :darkgreen, :purple, :darkblue, :orange, :pink, :brown, :cyan,
          :magenta, :gold, :indigo, :lime, :olive, :maroon, :navy, :turquoise, :chocolate,
          :orchid, :salmon, :sienna, :slateblue, :thistle, :violet, :yellowgreen, :coral,
          :steelblue, :darkorange, :seagreen, :plum]
rgb_colors = [ColorTypes.color(string(name)) for name in colors]

# 2. e vs. a plot
function plot_e_vs_a(a,e,colorVal)

    # Define the color gradient
    function fade_color(colorVal,alpha)
        RGBA(colorVal, alpha)
    end

    # Calculate the alpha values for each point
    alphas = LinRange(0.2, 1, length(a))

    scatter!(a,e, legend=false, xlims=(6500,9000), ylims=(0,0.2),
                    color = fade_color.(colorVal, alphas), markerstrokewidth=0)
    xlabel!("Semi-major axis (km)")
    ylabel!("Eccentricity")
end

figure4 = plot()
for i = 1:N
    title!(figure4, "Case 1: No perturbations")
    plot_e_vs_a(a1[:,i],e1[:,i], rgb_colors[i])
end
display(figure4)

figure5 = plot()
for i = 1:N
    title!(figure5, "Case 2: Drag perturbations")
    plot_e_vs_a(a2[:,i],e2[:,i], rgb_colors[i])
```

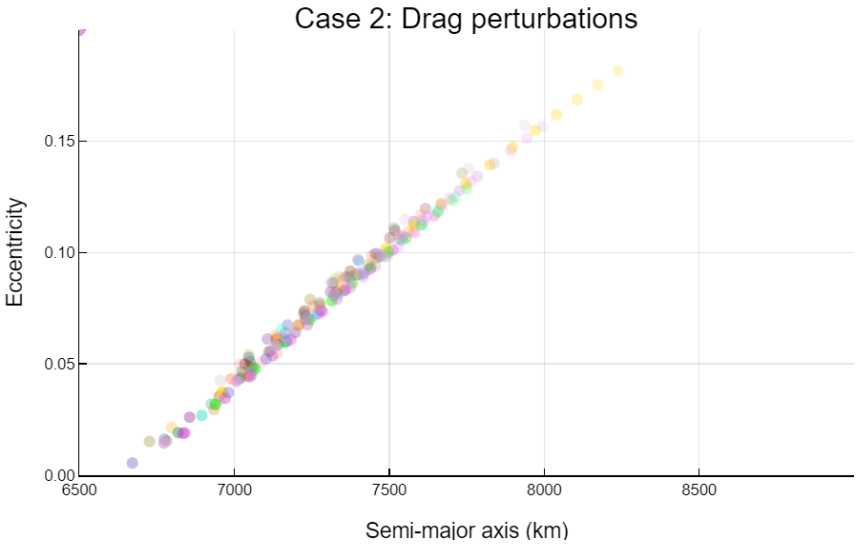
```
end
display(figure5)

figure6 = plot()
for i = 1:N
    title!(figure6, "Case 3: Drag and J2 perturbations")
    plot_e_vs_a(a3[:,i],e3[:,i], rgb_colors[i])
end
display(figure6)

# 3. RAAN vs. a plot
function plot_RAAN_vs_a(a,RAAN,colorVal)
    # Define the color gradient
    function fade_color(colorVal,alpha)
        RGBA(colorVal, alpha)
    end

    # Calculate the alpha values for each point
    alphas = LinRange(0.2, 1, length(a))
    scatter!(a,RAAN, legend=false, xlims = (6500,8000), markerstrokewidth=0,
                                    color = fade_color.(colorVal, alphas))
    xlabel!("Semi-major axis (km)")
    ylabel!("RAAN (rad)")
end

figure7 = plot()
for i = 1:N
    title!(figure7, "Case 1: No perturbations")
    plot_RAAN_vs_a(a1[:,i],RAAN1[:,i],rgb_colors[i])
end
display(figure7)

figure8 = plot()
for i = 1:N
    title!(figure8, "Case 2: Drag perturbations")
    plot_RAAN_vs_a(a2[:,i],RAAN2[:,i],rgb_colors[i])
end
display(figure8)

figure9 = plot()
for i = 1:N
    title!(figure9, "Case 3: Drag and J2 perturbations")
    plot_RAAN_vs_a(a3[:,i],RAAN3[:,i],rgb_colors[i])
end
display(figure9)
```

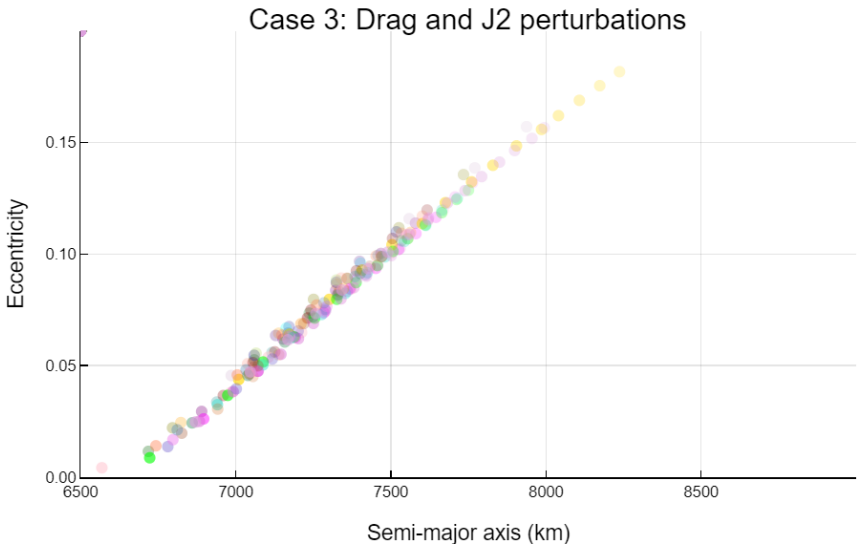

Case 1: No perturbations
Altitude (km)
3000
2000
1000
0
5400
5700
6000
6300
6600
6900
7200
Orbital period (s)

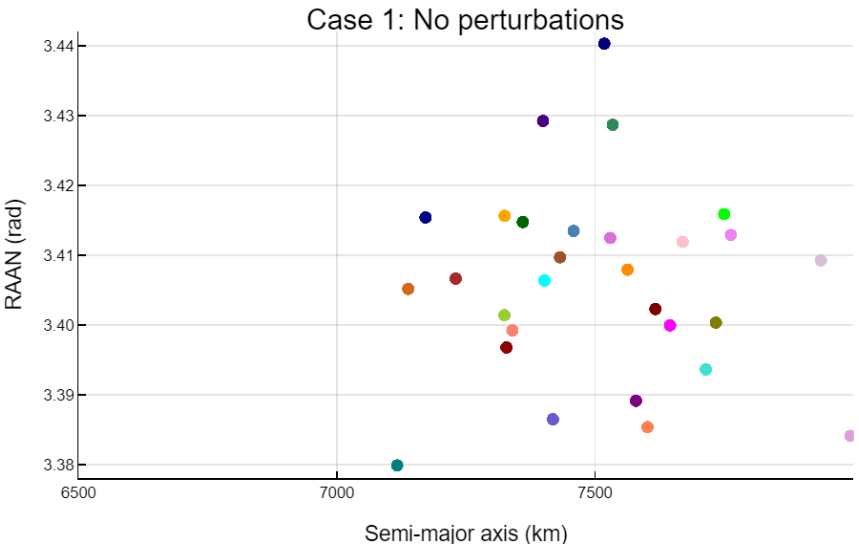

Case 2: Drag perturbations
Altitude (km)
3000
2000
1000
0
5400
5700
6000
6300
6600
6900
7200
Orbital period (s)

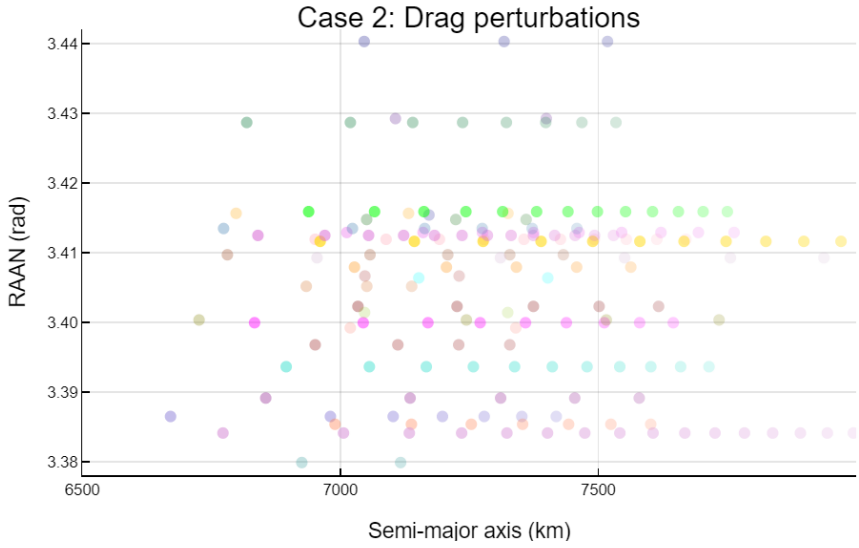

Case 3: Drag and J2 perturbations
Altitude (km)
3000
2000
1000
0
5400
5700
6000
6300
6600
6900
7200
Orbital period (s)

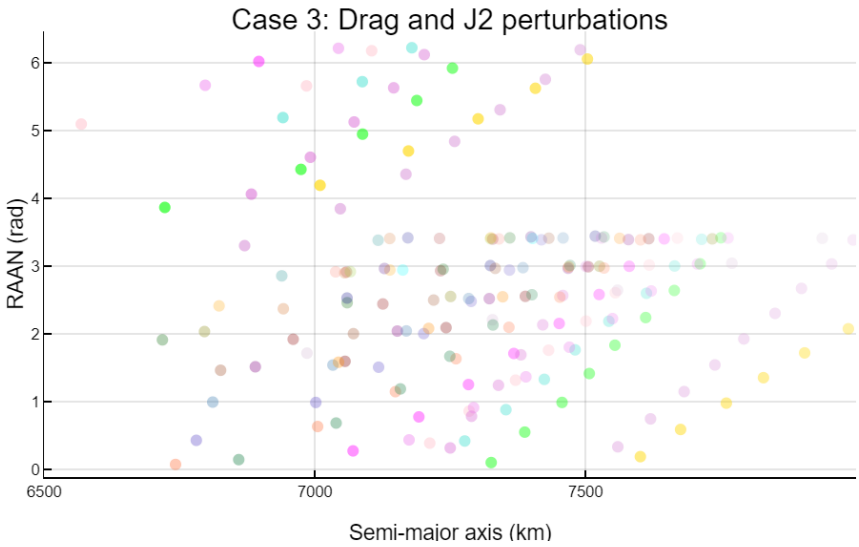

Case 1: No perturbations
Eccentricity
0.15
0.10
0.05
0.00
6500
7000
7500
8000
8500
Semi-major axis (km)


Case 2: Drag perturbations
Eccentricity
0.15
0.10
0.05
0.00
6500
7000
7500
8000
8500
Semi-major axis (km)


Case 3: Drag and J2 perturbations
Eccentricity
0.15
0.10
0.05
0.00
6500
7000
7500
8000
8500
Semi-major axis (km)


Case 1: No perturbations
RAAN (rad)
3.44
3.43
3.42
3.41
3.40
3.39
3.38
6500
7000
7500
Semi-major axis (km)


Case 2: Drag perturbations
RAAN (rad)
3.44
3.43
3.42
3.41
3.40
3.39
3.38
6500
7000
7500
Semi-major axis (km)



As we can see from the plots:

Case 1: No perturbations result in constant orbital altitudes and period for each fragment. The eccentricity and semi-major axis remain constant. It also results in constant values for both RAAN and semi-major axis.

Case 2: Atmospheric drag causes the orbital period to decrease over time. Perigee altitudes remain constant, while apogee altitudes decrease, leading to the reduction in the period.The rate of loss of eccentricity per kilometer of semi-major axis is constant due to atmospheric drag. The relationship between eccentricity and semi-major axis is not linear but inversely proportional. The drag circularizes the orbit. Atmospheric drag causes variations in the semi-major axis while keeping the RAAN constant. Only parameters within the orbital plane are affected by time variations.

Case 3: Drag and Earth's oblateness perturbations cause both apogee and perigee altitudes to decrease over time, resulting in a decrease in the orbital period. The decrease in apogee altitude is more abrupt due to the combined perturbations, while perigee altitude is affected only by Earth's oblateness. The relationship between eccentricity and semi-major axis remains inversely proportional. The effect of perturbations (drag and oblateness) is primarily on the semi-major axis, reducing it faster. Both RAAN and semi-major axis decrease over time due to drag and oblateness perturbations. The oblateness perturbation affects the RAAN, leading to its change over time. The rate of change in RAAN is influenced by J2 and (Re / a)^2, with stronger effects for lower altitudes and higher inclinations.

Therefore, the main effect of the drag perturbation is that it decreases the apocenter and circularizes the orbit. Whereas, the main effect of the J2 perturbation is that the RAAN changes, and the eccentricity therefore decreases.

With this example, the main functionalities used were those relating to the Orbital Perturbations file, and some recurrent ones from the Ideal Two-Body problem (changing from the state vector to COE and back). With these functions, it was managed to propagate a debris cloud along time, and compare the effects that considering perturbations have on the altitude, eccentricity, semi-major axis or the RAAN of the orbit.

The results obtained were in agreement with reality, as what causes the RAAN to change is the oblateness of the Earth, and the drag lowers the apocenter circularizing the orbit.

Due to having to compute the integration of a large number of cases, the computational time of this exercise was 9 min 14s; 5 of those needed for the integration, and 3 to load the Plots package, as the plots needed for this exercise had to be done from scrap.

As this example proves again, the functions available with our AstrodynamicsEdu package can be easily used and incorporated into solving problems that undergraduate students must face in Orbital Dynamics courses; in this case, students from the University Carlos III.

In addition, these sample problems will also be available for future users. Thus, ensuring that the library is user-friendly, with clear documentation, examples, and tutorials to make it easy to use.

6. CONCLUSIONS AND FUTURE WORK

6.1. Conclusions

Based on the solutions presented above, it can be concluded that this project has successfully achieved its main goal of creating an accessible Astrodynamics library in Julia for undergraduate students. The AstrodynamicsEdu.jl package will bring students closer to Orbital Dynamics by bringing them the opportunity to visualize how spacecrafts move through space; how orbital perturbations affect satellites orbiting around Earth; or the ability to perform essential calculations for space mission planning. Therefore, the four objectives established at the beginning of this thesis have been effectively met.

A comprehensive functional analysis of Mechanics and Orbital Dynamics was conducted to identify and define the requirements of the library, ensuring that it caters to the needs of the target audience. By also considering the scope of the problems that students coursing Orbital Dynamics may face, the following requirements were drawn out for our Julia package: solving problems of the ideal two-body problem; change of reference frame and vector basis; delta-v computation of orbital maneuvers; propagate orbits considering orbital perturbations; and orbit visualization and plotting.

The modular structure necessary to tackle each of the requirements was carefully designed, outlining the specific functionalities to be implemented within the library. To ensure a coherent and user-friendly experience, all the functions available were exhaustively documented specifying what they do, their inputs needed, and the obtained outputs.

The identified files and functionalities were then implemented in Julia to develop the AstrodynamicsEdu.jl package, following best practices and coding standards to ensure code efficiency and maintainability.

Finally, rigorous testing and illustrative examples were employed to validate the functionality and performance of the implemented modules, showcasing their practical usage within the realm of Orbital Dynamics in an educational setting. As they were directly taken from the course of Space Vehicles and Orbital Dynamic of University Carlos III de Madrid.

By fulfilling its objectives, the AstrodynamicsEdu.jl package represents a significant contribution to astrodynamics education, equipping undergraduate students with a powerful tool to enhance their understanding and practical skills in orbital dynamics. This libary stands as a testament to the potential of Julia as a programming language for scientific and educational purposes. Its modular design and the successful implementation of its functionalities provide a solid foundation for future expansion and improvement, ensuring its continued relevance and usability in an ever-evolving field.

6.2. Future work

Due to time constraints and the complexity of certain topics, several useful functionalities were not implemented in the current version of our Astrodynamics library. However, there are potential areas of improvement and expansion that could enhance the capabilities of the code in the future.

Here are some suggestions for future work:

- Extend the orbital perturbations module: Incorporate models for Solar Radiation Pressure and Third Body effects. Implement these functions in the propagator to account for these additional perturbations.
- Introduce interplanetary flight capabilities: Develop algorithms and modules to handle trajectory calculations and orbital transfers between celestial bodies in the solar system.
- Address the Three Body Problem: Explore methods and techniques to solve the Three Body Problem, which involves the gravitational interactions between three celestial bodies.
- Enhance the Orbital Visualization module: Provide more options and flexibility in the visualization of orbits. Consider incorporating additional libraries or allowing the user to choose whether to use the Plots package for plotting, by having recipes instead of plot functions, as it has already been done for the Linear Algebra types.

These future developments would expand the functionality and usefulness of the AstrodynamicsEdu package, enabling it to tackle more complex scenarios and providing students with a broader range of tools for their Orbital Dynamics simulations and analyses.

BIBLIOGRAPHY

[1] L. Xiao, G. Mei, N. Xi, and et al., "Julia language in computational mechanics: A new competitor," *Archives of Computational Methods in Engineering*, vol. 29, pp. 1713–1726, 2022. DOI: 10.1007/s11831-021-09636-0.

[2] JuliaLang. "How to develop a julia package." (2019), [Online]. Available: https://julialang.org/contribute/developing_package/.

[3] JuliaLang. "Modules." (2020), [Online]. Available: https://docs.julialang.org/en/v1/manual/modules/.

[4] JuliaLang. "Types." (2020), [Online]. Available: https://docs.julialang.org/en/v1/manual/types/.

[5] M. Merino, *Mechanics Applied to Aerospace Engineering Lecture Notes*. Universidad Carlos III de Madrid, Version 2020.

[6] J. R. Taylor, *Classical Mechanics*. University Science Books, 2005.

[7] H. D. Curtis, *Orbital mechanics for engineering students*. Elsevier Butterworth-Heinemann, 2005.

[8] M. Merino, *Space Vehicles and Orbital Dynamics Lecture Notes*. Universidad Carlos III de Madrid, Version 2020.2.

[9] W. Commons. "The perifocal coordinate system (with unit vectors p, q, w), against the reference coordinate system (with unit vectors i, j, k)." File: `Perifocal coordinates.svg`. (2023), [Online]. Available: https://commons.wikimedia.org/wiki/File:Perifocal_coordinates.svg.

[10] NASA. "Orbits and kepler's laws." (2008), [Online]. Available: https://solarsystem.nasa.gov/resources/310/orbits-and-keplers-laws/#:~:text=Kepler's%20Laws%20of%20Planetary%20Motion&text=They%20describe%20how%20(1)%20planets,its%20semi%2Dmajor%20axis)..

[11] "Sun sensor navigation for planetary rovers: Theory and field testing." Scientific Figure on ResearchGate. (), [Online]. Available: https://www.researchgate.net/figure/Earth-Centred-Inertial-Earth-Centred-Fixed-and-Topocentric-reference-frames-See-Figure_fig1_224244574.

[12] W. C. MenteMagica. "Two-impulse hohmann orbital transfer maneuver." File: `Orbital Hohmann Transfer.svg`. (2011), [Online]. Available: https://commons.wikimedia.org/wiki/File:Orbital_Hohmann_Transfer.svg.

[13] A. Geeks, *Low thrust trajectory optimizationr*, Generated with STK's Astrogator, 2019. [Online]. Available: https://www.agi.com/articles/Low-Thrust-Trajectory-Optimization.

[14] JuliaLang. “Linearalgebra.jl documentation.” (), [Online]. Available: https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/.

[15] JuliaLang. “Test.jl documentation.” (), [Online]. Available: https://docs.julialang.org/en/v1/stdlib/Test/.

[16] JuliaLang. “Plots.jl documentation.” (), [Online]. Available: https://docs.juliaplots.org/stable/.

[17] JuliaLang. “Unitful.jl documentation.” (), [Online]. Available: https://painterqubits.github.io/Unitful.jl/stable/.

[18] JuliaLang. “Roots.jl documentation.” (), [Online]. Available: https://juliamath.github.io/Roots.jl/dev/.

[19] JuliaLang. “Differentialequations.jl documentation.” (), [Online]. Available: https://docs.sciml.ai/DiffEqDocs/stable/.

[20] W. Media. “Cowell’s method. forces from all perturbing bodies (black and gray) are summed to form the total force on body i (red), and this is numerically integrated starting from the initial position (the epoch of osculation).” File: File:Cowells method.png. (2012), [Online]. Available: https://en.wikipedia.org/wiki/Perturbation_(astronomy)#/media/File:Cowells_method.png.

APPENDIX 1 - PACKAGE SET UP

Fulfilling its main purpose, this code can be used by undergraduate students to solve Orbital Dynamics problems, and to gain some further understanding into the subject. To be able to use all the functions described, it is only required to follow these simple steps:

1. Install Julia: If you haven't already, download and install Julia from the official Julia website (https://julialang.org/downloads/) according to your operating system.

2. Clone the GitHub repository: Go to the GitHub page of the package AstrodynamicsEdu (https://github.com/AliciaSBa/AstrodynamicsEdu.jl) and find the repository's URL. Clone the repository to your local machine using a Git client or by downloading the ZIP file and extracting it.

3. Set up the package: Open a Julia REPL (Read-Eval-Print Loop) or start Julia from your terminal. In the Julia REPL, enter the package manager mode by pressing `]` (you will see a prompt change to `pkg>`).

4. Activate the package environment: In the package manager mode, activate the package environment specific to the cloned repository by running the following command, replacing `<package_name>` with the actual name of the package:

   ```
   activate /path/to/package/repository
   ```

5. Add the package from the GitHub repository : Clone the link from the repository, and in the package manager mode, run the following command:

   ```
   add https://github.com/AliciaSBa/AstrodynamicsEdu.jl.git
   ```

6. Build the package: Run the following command to build the package and resolve any dependencies:

   ```
   instantiate
   ```

   This command will read the `Project.toml` file in the package repository and install the required dependencies.

7. Start using the package: Once the package has been successfully built, you can start using it in your Julia code. Import the package by using the `using` keyword followed by the package name.

```
using AstrodynamicsEdu
```

It is important to periodically update the package by pulling the latest changes from the GitHub repository and rebuilding the package using the package manager.

Once these steps have been followed, the AstrodynamicsEdu.jl functions are ready to be used. This package will allow the user to:

- Have multiple Linear Algebra Capabilities at their disposal, such as plotting Vector Basis and Reference Frames in 3D, or converting easily between Vector Basis and Reference Frames.
- Solve exercises of the Ideal Two Body Problem.
- Plot orbits in the Perifocal Plane (2D) and in the ECI Reference Frame (3D).
- Obtain the classical orbital elements from the state vector, and the other way around.
- Convert between anomalies. Find the time of flight between two positions in an orbit.
- Solve the Lambert's problem.
- Compute the Δv needed to perform a Hohmann transfer, a plane change maneuver or a low-thrust orbit raising.
- Propagate the orbit considering the drag and/or the Earth oblateness perturbations.