Detailed ESA-GODOT Installation and Configuration Guide

Buyanbileg Amarsanaa

Tobias Cornelius Hinse

June 27, 2025

Denmark

Disclaimer

This document is for informational and educational purposes only. While efforts have been made to ensure accuracy, the author and contributors do not guarantee functionality across all system configurations. Users proceed at their own risk, and the author and contributors are not liable for issues, data loss, or damages. ESA-GODOT is developed by the European Space Agency (ESA). This guide is not official ESA documentation; refer to official ESA resources for updates and support. Ensure compatibility, security, and proper licensing when using ESA-GODOT.

Contents

Chapter 1

ESA-GODOT Package Installation on Python Interface

This chapter provides detailed instructions for installing the ESA-GODOT package (`esa-godot`) on a native Linux system, specificially Ubuntu 22.04 LTS, using Conda to manage the Python environment. The process includes setting up a Conda environment, obtaining a personal access token from https://gitlab.space-codev.org/, and installing the `esa-godot` package. Configuration such as (e.g., `universe.yml`) and advanced usage of the `esa-godot` package are covered in further chapters and the workshops.

The following prerequisites are required to further install and then start working on the software packages as follows:

- A Linux system running Ubuntu 22.04 LTS (other distributions may work but are untested).
- Administrative (`sudo`) privileges for installing system packages
- Internet access for downloading Miniconda, accessing GitLab and installing packages.
- Basic familiarity with the Linux terminal

Disclaimer: Refer to the guide's disclaimer for usage risks and the recommendation to consult official ESA-GODOT documentation.

1.1 Where to Start

According to the first hand-out on the installation of Linux through Conda environment, the conda environment is expected to be permanent. Therefore, the user has to ensure that the `godotdev` environment is active in the Ubuntu 22.04 terminal (`Ctrl+Alt+T`) whilst having their GitLab token ready in a secure location.

1.2 Step 1: Verify Conda Environment

What to Do:

1. Activate the conda environment `godotdev`:

```
conda activate godotdev
```

2. Verify Python Version

```
python --version
```

Expected Output:

```
Python 3.10.X
```

1.3 Step 2: Install ESA-GODOT Package

What to Do:

1. Install `esa-godot` using `pip` with your token:

```
pip install esa-godot --index-url https://__token__:<your_personal_token>@gitlab.space-codev.org/api/v4/projects/107/packages/pypi/simple
```

   - Replace `<your_personal_token>` with your token (e.g.,`glpat-123abc...`)
   - Example:

```
pip install esa-godot --index-url https://__token__:glpat-123abc...@gitlab.space-codev.org/api/v4/projects/107/packages/pypi/simple
```

Why: The `pip` command installs esa-godot from the private GitLab repository using the token for authentication.

1.4 Step 3: Verify Installation

What to Do:

1. Check `esa-godot` installation:

```
pip show esa-godot
```

Expected Output:

```
Name: esa-godot
version: 1.11.0
Location: /home/user/miniconda3/envs/godotdev/lib/python3.10/site-packages
```

Why: This step accordingly verifies installation step, ensuring readiness for the further configuration within the package.

1.5 Notes for WSL and MacOS users

- WSL (Windows): Please follow *Conda Guide* for the environment step, then use the same `pip` command in the WSL Ubuntu terminal.
- MacOS: Please follow *Conda Guide* for the dedication chapter for the MacOS users, then use the `pip` command in the MacOS terminal.

Why: The `esa-godot` installation process is identical across platforms once the Conda environment is setup.

Chapter 2

Configuration of the necessary files

2.1 `universe.yml` Configuration Using Linux Commands

The `universe.yml` configuration for ESA GODOT requires two ephemeris files: `de432s.bsp` for planetary positions and `gm_de431.tpc` for planetary constants. These files must be downloaded from the NASA JPL repository and placed in the appropiate project directory. The following instructions assume that you are working in a Linux terminal (e.g., Ubuntu 22.04) with an active Conda environment (`godotdev`) or a virtual environment as we set up in the ESA-GODOT installation guide.

Prerequisties:

- A Linux system (e.g., Ubuntu 22.04 LTS, WSL on Windows, or Ubuntu Server in VirtualBox on MacOS)
- Internet access for downloading files
- Basic familarity with the Linux terminal
- A project directory (e.g., `~/godotpy_project`) where the files will be stored.
- The Conda environment `godotdev` or a virtual environment is active (if following the ESA-GODOT installation guide).

Please note the following:

- Relative Paths: Use relative paths in `universe.yml` (e.g., `data/ephemeris/de432s.bsp`) to ensure compatibility across systems
- File integrity: Verify the downloaded files are not corrupted by checking their size or checksums (if provided by JPL)
- Security: Store files in a private project directory to avoid unauthorised access
- Official Source: Always ensure to download from the official NASA JPL repository to ensure authenticity.

2.1.1 Step-by-Step Instructions

Step 1: Navigate to Your Project Directory

What to Do:

- Open a terminal (e.g.,`Ctrl+Alt+T` on Ubuntu or WSL)

- Navigate to your ESA-GODOT project directory (e.g.,∼/godotpy_project). If the project directory does not exist to store the necessary files, create the following:

```
        mkdir -p ~/godotpy_project && cd $\sim$/godotpy_project
```

- Create a subdirectory for ephemeris files:

```
        mkdir -p data/ephemeris
        cd data/ephemeris
```

Why: The reason why having a folder within the virtual environment, it allows the user to organise the project structure as recommended above, ensuring `universe.yml` can reference with relative paths like `data/ephemeris/de432s.bsp`.

Expected Output: You are now in the `~/godotpy_project/data/ephemeris` directory.

Step 2: Download `de432s.bsp` for the planetary positions

What to Do:

- Use `wget` to download the `de432s.bsp` file from the NASA JPL repository:

```
        wget https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk
           /planets/de432s.bsp
```

- **Alternatively**, if `wget` is not installed, use `curl`:

```
        curl -O https://naif.jpl.nasa.gov/pub/naif/generic_kernels/
           spk/planets/de432s.bsp
```

Why: The `de432s.bsp` file provides planetary position data required for ESA-GODOT simulations. `wget` or `curl` are standard Linux tools for downloading files from URLs.

Expected Output:

- A filed named `de432s.bsp` appears in the data/ephemeris directory.
- Terminal output shows download progress and completion (e.g., 100%)
- Verify the file exists on the Linux terminal:

```
        ls
```

  Expected : `de432.bsp`

Step 3: Download `gm_de431.tpc` for the planetary constants

- Use `wget` to download the `gm_de431.tpc` file from the NASA JPL repository

```
        wget https://naif.jpl.nasa.gov/pub/naif/generic_kernels/pck
           /gm_de431.tpc
```

- Alternatively, use `curl`:

```
        curl -O https://naif.jpl.nasa.gov/pub/naif/generic_kernels/
           pck/gm_de431.tpc
```

Why: The gm_de431.tpc file provides planetary constants (e.g., gravitational parameters) needed for accurate ESA-GODOT simulations.

Expected Output:

- A file named gm_de431.tpc appears in the data/ephemeris directory.
- Linux terminal output shows download progress and completion
- Verify the file exists:

```
ls
```

  Expected: de432s.bsp gm_de431.tpc

Troubleshooting: If the file gm_de431.tpc is not succeded, please run the following URL, instead:

```
wget https://naif.jpl.nasa.gov/pub/naif/generic_kernels/pck/
    pck00010.tpc
```

Step 3: Copy nutation2000A.ipf to the Ephemeris Directory

What to Do:

1. Assumed the provided nutation2000A.ipf file is located in a local directory, e.g., ~/Downloads. Please copy this to the ephemeris directory.

```
cp ~/Downloads/nutation2000A.ipf ~/godot_project/data/
    ephemeris/
```

   Why: Copying the file to data/ephemeris/ ensures it is accessible to ESA-GODOT via the relative path specified in universe.yml

   Expected Output:

   - The nutation2000A.ipf file appears in ~/godotpy_project/data/ephemeris.
   - In order to verify this, follow:

```
ls ~/godotpy_project/data/ephemeris
```

     Expected: de432s.bsp gm_de431.tpc (or) pck00010.tpc nutation2000A.ipf

Step 4: Copy eigen05c _80_sha.tab to the Ephemeris Directory

What to Do:

1. Copy the provided eigen05c_80_sha.tab file from the same local directory (e.g., ~/Downloads):

```
cp ~/Downloads/eigen05c_80_sha.tab ~/godotpy_project/data/
    ephemeris/
```

Why: This places eigen05c_80_sha.tab in the correct directory for ESA-GODOT to access it during simulations.

Expected Output:

- The eigen05c_80_sha.tab file appears in the following ~/godotpy_project/data/ephemeris
- Verify the following:

```
        ls ~/godotpy_project/data/ephemeris
```

2.1.2 Universe.yml file for the workshop mission

The following `Universe.yml` configuration will be automatically handed out to you, however, it is very crucial to understand the structure of the file to properly use it later in the future.

```
version: '3.0'

# Spacetime configuration
spacetime:
  system: BCRS  # Barycentric Celestial Reference System, standard for
      solar system dynamics

# Ephemeris definitions
ephemeris:
  - name: de440
    files:
      - "/home/user/godotpy_project/data/ephemeris/de440.bsp"  # Adjust
          to your path
  - name: gm440
    files:
      - "/home/user/godotpy_project/data/ephemeris/pck00010.tpc"  #
          Gravitational constants

# Constants from ephemeris
constants:
  ephemeris:
    - source: gm440

# Ground stations
stations:
  - name: earthStations
    file: "/home/user/godotpy_project/KourouStation.json"  # Path to
        your ground stations file and this file will be handed out
        automatically during the workshop

# Reference frames
frames:
  - name: ephem1
    type: Ephem
    config:
      source: de440
  - name: Earth
    type: AxesOrient
    config:
      model: IERS2000
      nutation: "/home/user/godotpy_project/data/ephemeris/
          nutation2000A.ipf"
      erp: ''
  - name: ITRF
    type: AxesOrient
    config:
      model: IERS2000
      nutation: "/home/use/godotpy_project/data/ephemeris/nutation2000A
          .ipf"
      erp: ''
```

```yaml
  - name: Mars
    type: AxesOrient
    config:
      model: MarsIAU2009
  - name: stations1
    type: Stations
    config:
      source: earthStations
      points: false
  - name: stations2
    type: Stations
    config:
      source: earthStations
      axes: false
  - name: LocalOrbitalFrame
    type: AxesLocalOrbital
    config:
      center: Mars
      target: Tyr_center
      axes: RAC
  - name: LCROT
    type: AxesLocalOrbital
    config:
      center: Sun
      target: Mars
      axes: Pos
# Celestial bodies
bodies:
  - name: Sun
    point: Sun
  - name: Earth
    point: Earth
  - name: Moon
    point: Moon
  - name: Mars
    point: Mars

# Gravity models
gravity:
  - name: solarSystem
    bodies:
      - Earth
      - Moon
      - Sun

# Dynamics models
dynamics:
  - name: solarSystemGravity
    type: SystemGravity
    config:
      model: solarSystem

  - name: EMS_gravity
    type: SystemGravity
    config:
      model: solarSystem

  - name: NGA
```

```
    type: CalibratedAcceleration
    config:
      input:
          x : NGA_x
          y : NGA_y
          z : NGA_z
      inputAxes: ICRF

  - name: EMS
    type: Combined
    config:
      - EMS_gravity
      - NGA

# Spacecraft definition
spacecraft:
  - name: Tyr
    mass: 1200 kg
    thrusters:
      - name: main
        thrust: 500 mN
        isp: 3500 s
parameters:
  - name: NGA_x
    type: ProcessNoiseSet
    config:
      gridFile: nga.csv # This will be handed out during the workshop
      column: 0
      bias: "0"
  - name: NGA_y
    type: ProcessNoiseSet
    config:
      gridFile: nga.csv
      column: 1
      bias: "0"
  - name: NGA_z
    type: ProcessNoiseSet
    config:
      gridFile: nga.csv
      column: 2
      bias: "0"
```

2.2 `Trajectory.yml` Configuration

For this workshop, we use the `trajectory.yml` file for the use of the orbital determination when a probe circularises the celestial body, in the workshop context, Earth and Mars.

Even though the file will be given to you directly during the workshop, it is also important to understand how they are systematically structured depending on the nature of the mission.

This configuration simulates the 10-day short orbital simulation circularising the Earth, initially placed in a polar orbit of 90° inclination angle at an altitude of 200 km in Low Earth Orbit (LEO).

```
settings: # As for the setting, it is evident that the relative and
    absolute tolerances are strict ensuring precision. This is followed
    by the steps, 1 million steps by RK787 (8th order of the Runge-Kutta
```

```
    Method) ensuring the trajectory file for a further numerical
    calculation.
  relTol: 1e-09
  absTol: 1e-09
  steps: 1000000

setup:
  - name: Tyr
    type: group
    spacecraft: Tyr
    input: # Here defines the spacecraft and its required parameters to
        use later in the timeline setup.
      - name: center
        type: point
        axes: ICRF # International Celestial Reference Frame
      - name: mass
        type: scalar
        unit: kg
      - name: dv
        type: scalar
        unit: m/s # You can either choose km/s or m/s

timeline:

  - type: point
    name: start
    point:
      reference: initial_state
      dt: 0.5 day

  - type: control
    name: initial_state
    epoch: 2026-01-07T16:42:13.815276 TDB
    state:
      - name: Tyr_center
        body: Earth
        axes: Earth
        project: true
        dynamics: EMS
        value:
          sma: 6578 km  # Semi-major axis
          ecc: 0.0
          inc: 90 deg  # Polar angle
          ran: 0 rad
          aop: 0 rad # Argument of perigee indicates whether or not the
              probe has started moving in scale between 0 deg and 360
              deg.
          tan: 0 deg # True anomaly remains 0 deg meaning that it is
              the initial point.
      - name: Tyr_mass
        value: 1200 kg # Assuming that the mass of the probe remains
            the same
      - name: Tyr_dv
        value: 7780 m/s  # Orbital speed of the probe at 200 km in LEO


  - type: point
    name: end
```

```
    point:
      reference: initial_state
      dt: 10 day
```

2.3 problem.yml Configuration

```
    parameters:
  free: [initial_state_Tyr_center_*]
  consider: []
equations:
- type: expr
  name: a_priori
  config:
       expr:
        - initial_state_Tyr_center_sma = 6578. km @ 2000-01-01T00
            :00:00.000 TDB | 10.0 km
        - initial_state_Tyr_center_ecc = 0. @ 2000-01-01T00:00:00.000
            TDB | 0.10
        - initial_state_Tyr_center_inc = 90 deg @ 2000-01-01T00
            :00:00.000 TDB | 0.10 deg
        - initial_state_Tyr_center_ran = -1.19306469722083 rad @
            2000-01-01T00:00:00.000 TDB | 0.10 rad
        - initial_state_Tyr_center_aop = 2.25115009969338 rad @
            2000-01-01T00:00:00.000 TDB | 0.10 rad
        - initial_state_Tyr_center_tan = 0. deg @ 2000-01-01T00
            :00:00.000 TDB | 0.10 deg
```

2.3.1 Introduction

2.3.2 Parameter Declaration

In the configuration file, the following block specifies the parameters involved in the estimation:

```
parameters:
  free: [initial_state_Tyr_center_*]
  consider: []
```

- **free:** All components of the spacecraft's initial state vector are designated as "free". This means that the estimator will adjust these values during orbit determination to fit the observation data.
- **consider:** No parameters are "considered" in this configuration. Consider parameters are modeled with uncertainty but not directly estimated; they contribute to the covariance propagation.

2.3.3 A Priori Information

The a_priori block provides initial guesses and uncertainties for the spacecraft's orbital elements at a reference epoch:

```
equations:
- type: expr
  name: a_priori
  config:
       expr:
```

```
        - initial_state_Tyr_center_sma = 6578. km @ 2000-01-01T00:00:00.000 TDB | 10.0 km
        - initial_state_Tyr_center_ecc = 0. @ 2000-01-01T00:00:00.000 TDB | 0.10
        - initial_state_Tyr_center_inc = 90 deg @ 2000-01-01T00:00:00.000 TDB | 0.10 deg
        - initial_state_Tyr_center_ran = -1.19306469722083 rad @ 2000-01-01T00:00:00.000 TDB | 0.10 rad
        - initial_state_Tyr_center_aop = 2.25115009969338 rad @ 2000-01-01T00:00:00.000 TDB | 0.10 rad
        - initial_state_Tyr_center_tan = 0. deg @ 2000-01-01T00:00:00.000 TDB | 0.10 deg
```

This information defines a prior estimate for each of the six classical orbital elements, as well as their 1-sigma uncertainties. The epoch is in TDB (Barycentric Dynamical Time), and all values are given at `2000-01-01T00:00:00.000 TDB`.

Parameter	**Value @ Epoch (2000-01-01 TDB)**	**Uncertainty**
Semi-major axis (a)	6578	10.0
Eccentricity (e)	0.0	0.10
Inclination (i)	90	0.10
RAAN (Ω)	−1.19306469722083 rad	0.10 rad
Argument of perigee (ω)	2.25115009969338 rad	0.10 rad
True anomaly (θ)	0.0	0.10

Table 2.1: Initial orbital elements and uncertainties used as a priori input for the GODOT estimation filter.

Chapter 3

Orbital Determination

This guide outlines the process of simulating and optimizing the trajectory of a spacecraft named ”Tyr” to achieve a circular orbit around Earth, as detailed in the Tyr Phase 1.pdf document. The process uses the `godot` and `cosmos` libraries for astrodynamics simulations and the `pygmo` library for optimisation. The goal is to place **Tyr** in a low Earth orbit (LEO) at a semi-major axis of 6578 km (approximately 200 km altitude) with zero eccentricity (circular) and a 90-degree inclination (polar orbit). The simulation spans 10 days, starting 12 hours after a reference time, with a required delta-V of 7.78 km/s.

Prerequisites:

- Software: Python with libraries numpy, matplotlib, godot (with cosmos module), and pygmo.
- `nga.csv` is downloaded into the following directory, ~`/godotpy_project/data/ephemeris`
- Configuration Files:
  - `Tyr_universe.yml`: Defines the simulation environment (e.g., Earth's gravitational parameters)
  - `Tyr_trajectory.yml`: Specifies the spacecraft's initial trajectory and parameters.
  - `Tyr_problem.yml`: Defines the optimisation problem (objectives and constraints).
- Hardware: A system capable of running numerical simulations and 3D visualisations.

3.1 Step-by-Step Instructions

3.1.1 Step 1: Set up the Python Environments

```
import numpy as np
import matplotlib.pyplot as plt
from godot.core import tempo, util, constants
from godot.model import common
from godot import cosmos
import pygmo
```

3.1.2 Step 2: Configure the Simulation Universe

1. Suppress Logging

```
util.suppressLogger()
```

Why: This disables verbose logging to keep console output clean, focusing on essential results.

2. Load the Universe configuration

```
uni_config = cosmos.util.load_yaml("Tyr_universe.yml")
uni = cosmos.Universe(uni_config)
```

Why: This code loads `Tyr_universe.yml`, which defines the physical environment (e.g., Earth's mass, radius, gravitational parameter), and it creates a Universe object (uni) that represents the simulation environment.

3. Inspect Universe Parameters

```
print(uni.parameters)
```

```
Parameter name          Type        Physical value Scale          Scaled value   Lower bound
NGA_x_bias              Fixed       0              1              0              -1.79769e+3
NGA_x_106               Fixed       0              1              0              -1.79769e+3
NGA_x_107               Fixed       0              1              0              -1.79769e+3
NGA_x_108               Fixed       0              1              0              -1.79769e+3
NGA_x_109               Fixed       0              1              0              -1.79769e+3
NGA_x_110               Fixed       0              1              0              -1.79769e+3
NGA_x_111               Fixed       0              1              0              -1.79769e+3
NGA_x_112               Fixed       0              1
...
```

Why: This establishes the simulation environment, ensuring no non-gravitational forces affect the orbit.

3.1.3 Step 3: Define the Spacecraft Trajectory

1. Load Trajectory Configuration

```
tra_cfg = cosmos.util.load_yaml("Tyr_trajectory.yml")
tra = cosmos.Trajectory(uni, tra_cfg, True)
```

Why: This loads `Tyr_trajectory.yml`, which specifies the spacecraft's initial state and trajectory parameters and creates a Trajectory object (tra) linked to the universe (uni), with a flag set to True (likely enabling detailed trajectory propagation).

2. Inspect Trajectory Parameters

```
print(uni.parameters)
```

Expected Output:

```
Parameter name          Type        Physical value Scale          Scaled value   Lower bound
NGA_x_bias              Fixed       0              1              0              -1.79769e+3
NGA_x_106               Fixed       0              1              0              -1.79769e+3
NGA_x_107               Fixed       0              1              0              -1.79769e+3
NGA_x_108               Fixed       0              1              0              -1.79769e+3
NGA_x_109               Fixed       0              1              0              -1.79769e+3
NGA_x_110               Fixed       0              1              0              -1.79769e+3
NGA_x_111               Fixed       0              1              0              -1.79769e+3
```

```
NGA_x_112            Fixed      0            1            0            -1.79769e+3
NGA_x_113            Fixed      0            1            0            -1.79769e+3
NGA_x_114            Fixed      0            1            0            -1.79769e+3
NGA_x_115            Fixed      0            1            0            -1.79769e+3
NGA_x_116            Fixed      0            1            0            -1.79769e+3
NGA_x_117            Fixed      0            1            0            -1.79769e+3
NGA_x_118            Fixed      0            1            0            -1.79769e+3
NGA_x_119            Fixed      0            1            0            -1.79769e+3
NGA_x_120            Fixed      0            1            0            -1.79769e+3
NGA_x_121            Fixed      0            1            0            -1.79769e+3
NGA_x_122            Fixed      0            1            0            -1.79769e+3
NGA_x_123            Fixed      0            1            0            -1.79769e+3
NGA_x_124            Fixed      0            1            0            -1.79769e+3
...
```

Why: This defines the spacecraft's initial orbit as a circular polar orbit at 200 km altitude, with a mass of 1200 kg and a delta-V requirement typical for LEO insertion.

3.1.4 Step 4: Set Up the Optimisation Problem

1. Load Problem Configuration

```python
    pro_cfg = cosmos.util.load_yaml("Tyr_problem.yml")
    prob = cosmos.orb.Problem(uni, pro_cfg)
    print(uni.parameters)
```

Expected Output:

```
Parameter name       Type       Physical value Scale      Scaled value   Lower bound
initial_state_Tyr_center_aopFree   0            0.1          0              -1.7976
initial_state_Tyr_center_eccFree   0            0.1          0              -1.7976
initial_state_Tyr_center_incFree   1.5708       0.1          15.708         -1.7976
initial_state_Tyr_center_ranFree   0            0.1          0              -1.7976
initial_state_Tyr_center_smaFree   6578         1e+06        0.006578       -1.7976
initial_state_Tyr_center_tanFree   0            0.1          0              -1.7976
NGA_x_bias           Fixed      0            1            0            -1.79769e+3
NGA_x_106            Fixed      0            1            0            -1.79769e+3
NGA_x_107            Fixed      0            1            0            -1.79769e+3
NGA_x_108            Fixed      0            1            0            -1.79769e+3
NGA_x_109            Fixed      0            1            0            -1.79769e+3
NGA_x_110            Fixed      0            1            0            -1.79769e+3
NGA_x_111            Fixed      0            1            0            -1.79769e+3
NGA_x_112            Fixed      0            1            0            -1.79769e+3
NGA_x_113            Fixed      0            1            0            -1.79769e+3
NGA_x_114            Fixed      0            1            0            -1.79769e+3
NGA_x_115            Fixed      0            1            0            -1.79769e+3
NGA_x_116            Fixed      0            1            0            -1.79769e+3
NGA_x_117            Fixed      0            1            0            -1.79769e+3
NGA_x_118            Fixed      0            1            0            -1.79769e+3
NGA_x_119            Fixed      0            1            0            -1.79769e+3
NGA_x_120            Fixed      0            1            0            -1.79769e+3
NGA_x_121            Fixed      0            1            0            -1.79769e+3
NGA_x_122            Fixed      0            1            0            -1.79769e+3
```

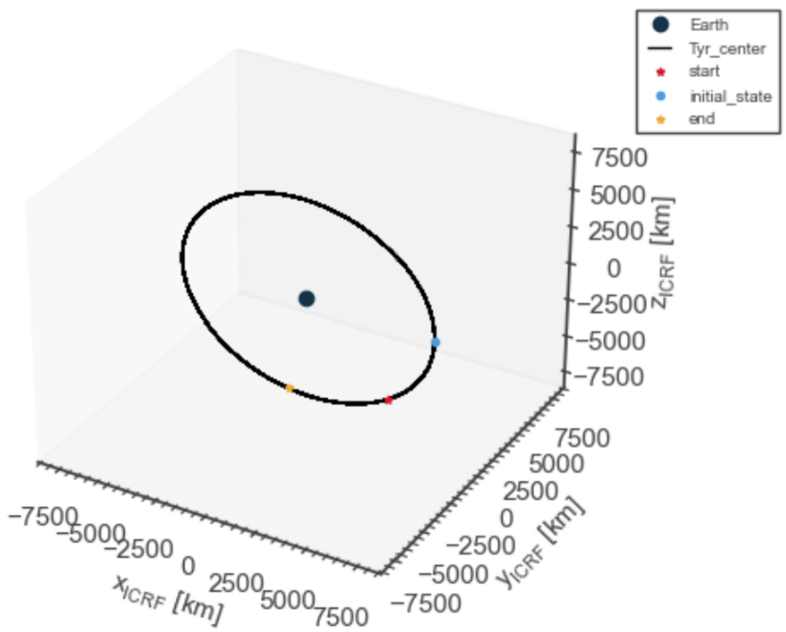
```
NGA_x_123                Fixed        0             1             0              -1.79769e+3
```

Why: This problem configuration prepares the optimisation problem, allowing the solver to adjust orbital elements to achieve the desired circular orbit.

3.1.5 Step 5: Visualise the Initial Trajectory

```python
import godot.cosmos.show

ax = godot.cosmos.show.Axes(
projection=(godot.cosmos.show.Dimension.SPACE, godot.cosmos.show.
    Dimension.SPACE,godot.cosmos.show.Dimension.SPACE),
uni=uni,
origin="Earth",
axes="ICRF",
)
ax.plot(godot.cosmos.show.FramePoint("Earth"), label="Earth")
ax.plot(tra, start="start", end="end", step=100, add_timeline_legend=
    True)
ax.configure_axes(leg_outside=True)
```

Expected Output:



3.1.6 Step 6: Compute and Optimize the Trajectory

1. Compute Trajectory

```python
    tra.compute(True)
```

2. Set Up and Run Solver

```python
solv = cosmos.orb.Solver(prob)
solv.processProblemEquations()
solv.regard()
fs = solv.filterSolution()
```

```python
print(f"The solution {fs.values()}")
print(f"The solution covariance {fs.covariance()}")
```

Expected Output:

```
The solution [ 2.25115010e+00  0.00000000e+00  1.57079633e+00 -1.19306470e+00
  6.57800000e+03  0.00000000e+00]
The solution covariance [[ 1.0000000e-02 -0.0000000e+00 -0.0000000e+00 -0.0000000e+00
  -0.0000000e+00 -0.0000000e+00]
 [-0.0000000e+00  1.0000000e-02 -0.0000000e+00 -0.0000000e+00
  -0.0000000e+00 -0.0000000e+00]
 [-0.0000000e+00  0.0000000e+00  3.0461742e-06 -0.0000000e+00
  -0.0000000e+00 -0.0000000e+00]
 [-0.0000000e+00  0.0000000e+00  0.0000000e+00  1.0000000e-02
  -0.0000000e+00 -0.0000000e+00]
 [-0.0000000e+00  0.0000000e+00  0.0000000e+00  0.0000000e+00
   1.0000000e+02 -0.0000000e+00]
 [-0.0000000e+00  0.0000000e+00  0.0000000e+00  0.0000000e+00
  -0.0000000e+00  3.0461742e-06]]
```

Chapter 4

Lambert Solvers Method

The method leverages Lambert's problem to determine the velocity changes which is often referred to as "Delta-V" required for an interplanetary transfer, comparing long-arc and short-arc solutions to identify the most fuel-efficient trajectory. The implementation relies on the `godot.core.astro` module within the ESA-GODOT package, with results visualised to aid mission planning. This chapter corresponds to Chapter 5 of the ESA-GODOT Installation and Configuration Guide and uses code from the Tyr Mission analysis.

4.1 Objective

The Lambert Solver Method aims to:

- Compute the transfer orbit from Earth's initial position to Mars' target position for various true anomalies and times of flight (TOF).
- Calculate the total Δv (velocity change) for long-arc and short-arc transfers to determine the most fuel-efficient option.
- Verify the trajectory accuracy through position error analysis and visualize the results.

4.2 Lambert's Problem Overview

Lambert's problem is a fundamental astrodynamics technique for finding an orbit that connects two positions ($\mathbf{r}_1$, $\mathbf{r}_2$) in a specified time (TOF) under a central gravitational field (e.g., the Sun, with $\mu = 1.3271244 \times 10^{11}\,\text{km}^3/\text{s}^2$). The solution provides the initial and final velocities ($\mathbf{v}_1$, $\mathbf{v}_2$) of the transfer orbit. For the Tyr Mission, the problem is solved for multiple combinations of Mars' true anomaly (θ_2) and TOF to optimize

$$\Delta v_{\text{total}} = \|\mathbf{v}_1^{\text{transfer}} - \mathbf{v}_1^{\text{Earth}}\| + \|\mathbf{v}_2^{\text{Mars}} - \mathbf{v}_2^{\text{transfer}}\|.$$

4.3 Methodology

4.3.1 Orbital Parameters

The code defines the initial and target orbits using classical orbital elements:

- **Earth's Initial Orbit** (at periapsis):

$$\text{coe1} = [147.692 \times 10^6\,\text{km}, 0.01, 0^\circ, 0^\circ, 0^\circ, 0^\circ]$$

where the elements are semi-major axis (a), eccentricity (e), inclination (i), longitude of the ascending node (Ω), argument of periapsis (ω), and true anomaly (θ).

- **Mars' Target Orbit** (without initial true anomaly):

$$\text{coe2} = [227.9904 \times 10^6\,\text{km}, 0.213, 0^\circ, 0^\circ, 0^\circ]$$

These are converted to Cartesian coordinates ($\mathbf{r}_1$, $\mathbf{v}_1$ for Earth; $\mathbf{r}_2$, $\mathbf{v}_2$ for Mars) using `astro.cartFromKep` with the Sun's gravitational parameter.

4.3.2 Grid Setup

To explore transfer options, a 2D grid is created for:

- **True Anomaly (θ_2)**: From 0 to 2π radians, using 500 points ($\text{dta2} = 2\pi{\cdot}\text{np.linspace}(0, 1, 500)$).
- **Time of Flight (TOF)**: From 30 to 50 days (2,592,000 to 4,320,000 seconds), using 500 points.

The grid is generated with `np.meshgrid`, producing arrays `ta2` (θ_2) and `tof`. Arrays `dv1_la`, `dv2_la`, `dv1_sa`, and `dv2_sa` store Δv values for long-arc and short-arc solutions.

4.3.3 Lambert Solver Implementation

For each grid point (θ_2, TOF):

1. Append θ_2 to `coe2` and convert to Cartesian coordinates ($\mathbf{r}_2$, $\mathbf{v}_2$).
2. Solve Lambert's problem using `astro.lambert` for:
   - **Long Arc**: Clockwise trajectory (`astro.LambertDirection.Clockwise`).
   - **Short Arc**: Counterclockwise trajectory (`astro.LambertDirection.CounterClockwise`).
3. Compute:
$$\Delta v_1 = \|\mathbf{v}_1^{\text{transfer}} - \mathbf{v}_1^{\text{Earth}}\|, \quad \Delta v_2 = \|\mathbf{v}_2^{\text{Mars}} - \mathbf{v}_2^{\text{transfer}}\|$$
   using `np.linalg.norm`.

The total $\Delta v = \Delta v_1 + \Delta v_2$ is calculated, and the minimum is identified. The short-arc solution yields $\Delta v_{\text{total}} = 15.9639\,\text{km/s}$ at $\theta_2 = 0.7681\,\text{rad}$, TOF = 4,320,000 seconds (50 days).

```python
# Example code snippet for Lambert solver
for i in range(nta):
    for j in range(ntof):
        coe2t = np.append(coe2, ta2[i, j])
        xyz2 = astro.cartFromKep(coe2t, GM)
        r2 = xyz2[0:3]
        v2 = xyz2[3:6]
        # Long arc
        sol_la = astro.lambert(r1, r2, tof[i, j], GM, 0, astro.
            LambertBranch.Right, astro.LambertDirection.Clockwise)
        dv1_la[i, j] = np.linalg.norm(sol_la.v1 - v1)
        dv2_la[i, j] = np.linalg.norm(v2 - sol_la.v2)
        # Short arc
        sol_sa = astro.lambert(r1, r2, tof[i, j], GM, 0, astro.
            LambertBranch.Right, astro.LambertDirection.CounterClockwise
            )
        dv1_sa[i, j] = np.linalg.norm(sol_sa.v1 - v1)
        dv2_sa[i, j] = np.linalg.norm(v2 - sol_sa.v2)
```

4.3.4 Verification and Visualisation

The optimal short-arc solution is used to:

- **Propagate Orbits**: Using `astro.keplerPropagate` over 1000 time steps for Earth, Mars, and the transfer orbit.
- **Compute Position Error**: The error at Mars is $\|\mathbf{r}_{\text{transfer}}(t_{\text{final}}) - \mathbf{r}_2\| = 4.2147 \times 10^{-8}$ km.
- **Visualize**: A contour plot of Δv versus θ_2 (in degrees) and TOF (in days) is generated:

```
plt.contourf(ta2*180/np.pi, tof/86400, dvt_sa, levels=100)
plt.colorbar(label='Total $\Delta v$ (km/s)')
plt.xlabel('Arrival True Anomaly (deg)')
plt.ylabel('Time of Flight (days)')
plt.title('Total $\Delta v$ for Short Arc Transfers')
plt.show()
```

4.4 Why the Lambert Solver?

The Lambert Solver Method is chosen because:

- **Efficiency**: It directly computes the transfer orbit for given positions and TOF, critical for interplanetary mission design.
- **Optimisation**: The grid search over θ_2 and TOF identifies the minimum Δv, balancing fuel efficiency and mission duration.
- **Flexibility**: Supports both long-arc and short-arc solutions, allowing comparison to ensure the optimal trajectory.
- **Integration with ESA-GODOT**: The `astro.lambert` function leverages the ESA-GODOT package, configured via `universe.yml`, ensuring accurate gravitational modeling.

4.5 Conclusion

The Lambert Solver Method, as implemented in the Tyr Mission, efficiently computes an optimal Earth-to-Mars transfer trajectory with a total Δv of 15.9639 km/s for a short-arc transfer. By solving Lambert's problem across a parameter space, verifying the solution with minimal position error, and visualizing the results, the method supports robust mission planning. This approach, integrated with the ESA-GODOT package, is ideal for workshop participants learning astrodynamics simulations.

4.6 Tyr Mission Lambert Solver in Heliocentric frame

4.6.1 Step 1: Setting up the Python Environments

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import godot.core.astro as astro
```

4.6.2 Step 2: Setting up the desired parameters

```
# Sun's gravitational parameter
GM = 1.3271244e11

# conversion factor from degrees 2 radians
d2r = np.pi/180

# Initial Orbit at periapsis
coe1 = np.array([147.692e6, 0.01, 0*d2r, 0*d2r, 0.0, 0.0])
xyz1 = astro.cartFromKep(coe1, GM)
r1 = xyz1[0:3]
v1 = xyz1[3:6]

# Target Orbit without true anomaly (NB: You will add the true anomaly
    later on)
coe2 = np.array([227.9904e6, 0.213, 0, 0, 0])
```

4.6.3 Step 3: Setting the grid for the target orbit

Here you have to add the time of flight in days, for this case to ensure that you can get a reasonable delta-V value by optimising it.

```
# Create a 2D grid of equally spaced points in the true anomaly x time-
    of-flight domains
nta, ntof = (500, 500)
dta2 = 2 * np.pi * np.linspace(0, 1, nta)
dtof = np.linspace(30*86400, 50*86400, ntof)  # 30 to 50 days
ta2, tof = np.meshgrid(dta2, dtof)

# Initialising Arrays
long_arc = True
dv1_la = np.zeros(ta2.shape)
dv2_la = np.zeros(ta2.shape)
dv1_sa = np.zeros(ta2.shape)
dv2_sa = np.zeros(ta2.shape)

for i in range(nta):
    for j in range(ntof):
        # Append arrival true anomaly to coe2 and convert to cartesian
            coordinates
        coe2t = np.append(coe2, ta2[i,j])
        xyz2 = astro.cartFromKep(coe2t, GM)
        r2 = xyz2[0:3]
        v2 = xyz2[3:6]
        branch = astro.LambertBranch.Right

        # Lambert solver - long arc
        sol_la = astro.lambert(r1, r2, tof[i,j], GM, 0, branch, astro.
            LambertDirection.Clockwise)
        dv1_la[i, j] = np.linalg.norm(sol_la.v1 - v1)
        dv2_la[i, j] = np.linalg.norm(v2 - sol_la.v2)

        # Lambert solver - short arc
        sol_sa = astro.lambert(r1, r2, tof[i,j], GM, 0, branch, astro.
            LambertDirection.CounterClockwise)
        dv1_sa[i, j] = np.linalg.norm(sol_sa.v1 - v1)
        dv2_sa[i, j] = np.linalg.norm(v2 - sol_sa.v2)
```

4.6.4 Step 5: Calculation of Delta-V for the Short and Long Arcs

```
dvt_la = dv1_la + dv2_la
idx_la = np.where(dvt_la == np.min(dvt_la))
dvt_la_min = dvt_la[idx_la]

dvt_sa = dv1_sa + dv2_sa
idx_sa = np.where(dvt_sa == np.min(dvt_sa))
dvt_sa_min = dvt_sa[idx_sa]
if dvt_la_min < dvt_sa_min:
    print('Long arc solution is more fuel efficient')
    print('delta-v tot: ', dvt_la_min, ' (km/s)')
    long_arc = True
    ta2_min = ta2[idx_la]
    tof_min = tof[idx_la]
else:
    print('Short arc solution is more fuel efficient')
    print('delta-v tot: ', dvt_sa_min, ' (km/s)')
    long_arc = False
    ta2_min = ta2[idx_sa]
    tof_min = tof[idx_sa]

print(ta2_min, tof_min, long_arc)
```

Expected Output:

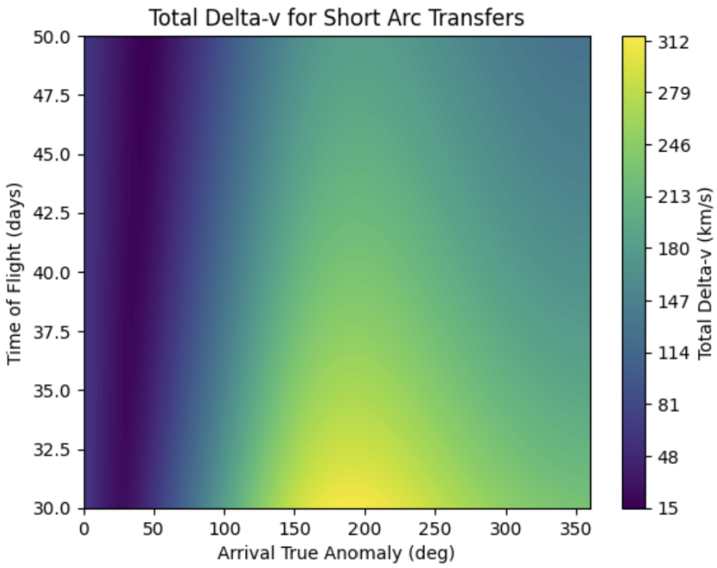
```
Short arc solution is more fuel efficient
delta-v tot:  [15.96392608]  (km/s)
[0.76808478] [4320000.] False
```

4.6.5 Step 6: Contour Map of the Most Efficient Delta-V

```
# Set the true anomaly at arrival
coe2f = np.append(coe2, ta2_min)
xyz2 = astro.cartFromKep(coe2f, GM)
r2 = xyz2[0:3]
v2 = xyz2[3:6]

# Solve the Lambert Problem from r0 to r1 in time TOF=tof_min
sol = astro.lambert(r1, r2, tof_min[0], GM, 0, branch, astro.
    LambertDirection.CounterClockwise)
print(sol.a)
# print(sol.p)
print(sol.v1)
print(sol.v2)
print('dv1: ', sol.v1 - v1)
print('dv2: ', v2 - sol.v2)
print('Total delta-v: ', np.linalg.norm(sol.v1 - v1) + np.linalg.norm(
    v2 - sol.v2), ' (km/s)')

plt.contourf(ta2*180/np.pi, tof/86400, dvt_sa, levels=100)
plt.colorbar(label='Total delta-v (km/s)')
plt.xlabel('Arrival True Anomaly (deg)')
plt.ylabel('Time of Flight (days)')
plt.title('Total delta-v for Short Arc Transfers')
plt.show()
```

Expected Output:

```
208408452.24253184
[ 8.35701823 33.29676494  0.        ]
[-10.58174486  25.64346684   0.        ]
dv1:  [8.35701823 3.0192365  0.        ]
dv2:  [-6.57417283 -2.6232867   0.        ]
Total delta-v:  15.963926079618398  (km/s)
```



4.6.6 Step 7: Propagation of the Spacecraft over time

The following code propagates the orbits of Earth, the transfer trajectory, and Mars over time to generate their trajectories for visualisation in the Tyr Mission. It uses the `astro.keplerPropagate` function from the ESA-GODOT package to evolve the orbits based on Kepler's equation, which describes motion in a two-body gravitational system. The initial point, in our case, Earth, transfer, and target (Mars) orbits are defined by their Keplerian elements, and their positions and velocities are converted to Cartesian coordinates using `astro.cartFromKep` for plotting. The propagation is performed over 1000 time steps, scaled by each orbit's period, to track the spacecraft's path from Earth to Mars[1].

```python
N = 1000
dt = np.linspace(0, 1, N)

# calculate initial orbit's period
n1 = np.sqrt(GM/coe1[0]**3)
P1 = 2*np.pi/n1

# calculate target orbit's period
n2 = np.sqrt(GM/coe2[0]**3)
P2 = 2*np.pi/n2

# convert transfer orbit from Cart to Kep
coet = astro.kepFromCart(np.concatenate((r1, sol.v1)), GM)

# propagate over time to generate trajectories
X1t = np.zeros((N, 6))
Xtt = np.zeros((N, 6))
X2t = np.zeros((N, 6))
for idx, t in enumerate(dt):

```

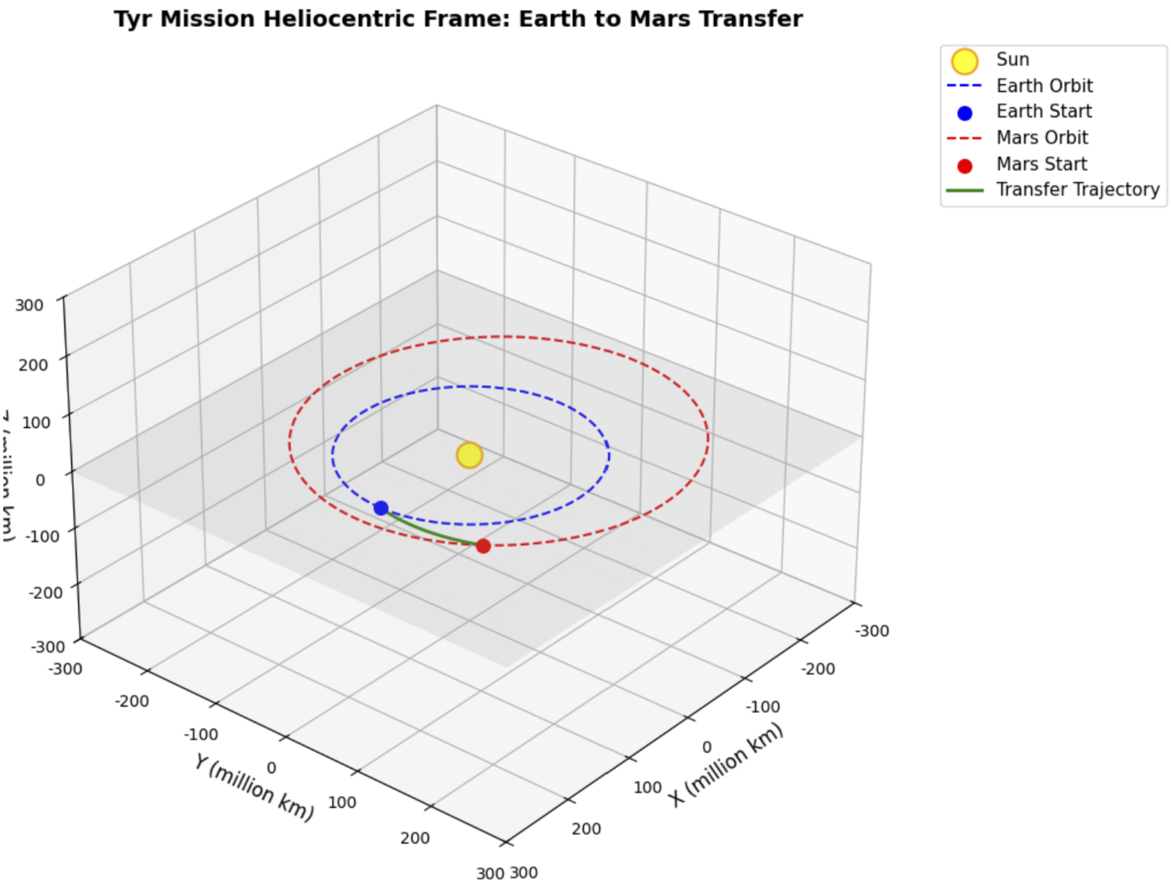
```
    # propagate orbits using Kepler's equation
    coe1t = astro.keplerPropagate(coe1, GM, P1*t)
    coett = astro.keplerPropagate(coet, GM, tof_min[0]*t)
    coe2t = astro.keplerPropagate(coe2f, GM, P2*t)

    # convert and store cartesian coordinates
    X1t[idx] = astro.cartFromKep(coe1t, GM)
    Xtt[idx] = astro.cartFromKep(coett, GM)
    X2t[idx] = astro.cartFromKep(coe2t, GM)
```

4.6.7 Step 8: Visualisation of the Transfer in Heliocentric frame

```
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')


ax.scatter(0, 0, 0, color='yellow', s=250, marker='o', label='Sun',
    edgecolors='orange', linewidth=1.5)


ax.plot3D(X1t[:, 0], X1t[:, 1], X1t[:, 2], linestyle='--', color='blue'
    , label='Earth Orbit')
ax.scatter(X1t[0, 0], X1t[0, 1], X1t[0, 2], color='blue', s=70, marker=
    'o', label='Earth Start')


ax.plot3D(X2t[:, 0], X2t[:, 1], X2t[:, 2], linestyle='--', color='red',
     label='Mars Orbit')
ax.scatter(X2t[0, 0], X2t[0, 1], X2t[0, 2], color='red', s=70, marker='
    o', label='Mars Start')


ax.plot3D(Xtt[:, 0], Xtt[:, 1], Xtt[:, 2], color='green', linewidth=2,
    label='Transfer Trajectory')


lim = 300e6
xx, yy = np.meshgrid(np.linspace(-lim, lim, 10), np.linspace(-lim, lim,
     10))
zz = np.zeros_like(xx)
ax.plot_surface(xx, yy, zz, color='lightgrey', alpha=0.2)

ax.set_xlabel('X (million km)', fontsize=12)
ax.set_ylabel('Y (million km)', fontsize=12)
ax.set_zlabel('Z (million km)', fontsize=12)
ax.set_title('Tyr Mission Heliocentric Frame: Earth to Mars Transfer',
    fontsize=14, fontweight='bold')

# Axis limits
ax.set_xlim(-lim, lim)
ax.set_ylim(-lim, lim)
ax.set_zlim(-lim, lim)

formatter = FuncFormatter(lambda x, _: f'{x*1e-6:.0f}')
ax.xaxis.set_major_formatter(formatter)
```

```python
ax.yaxis.set_major_formatter(formatter)
ax.zaxis.set_major_formatter(formatter)

# Grid and view angle
ax.grid(True, linestyle=':', alpha=0.7)
ax.view_init(elev=30, azim=40)

ax.legend(loc='upper left', bbox_to_anchor=(1.05, 1), fontsize=11)

ax.set_box_aspect([1,1,0.7])

plt.tight_layout()
plt.show()
```

Expected Output:



4.6.8 Step 9: Position Error

We can assess the precision error later in the end as follows:

```python
# Check position and velocity errors
err_pos = Xtt[-1, 0:3] - r2
print('Pos. Error: ', np.linalg.norm(err_pos))
```

```
Pos. Error:  4.2146848510894035e-08
```

Bibliography

[1] ESA Flight Dynamics. *GODOT Python (godotpy): Python interface for the ESA/ESOC GODOT astrodynamics library*. https://godot.io.esa.int/godotpy/. Documentation for version 1.11.0 (latest release as of June 2025). 2025. (Visited on 06/11/2025).