Numerical Orbit Propagation Using Initial State Vectors in a Two-Body Orbit Equation

[1] Seyed Mani Seyed Salehi Zadeh
Faculty of Aerospace Engineering
K. N. Toosi University of Technology
Tehran, Iran
manisalehi2004@gmail.com

[2] Sarah Barati
Faculty of Aerospace Engineering
K. N. Toosi University of Technology
Tehran, Iran
sarahbarati.livia@gmail.com

This project presents a numerical solution to the two-body orbital motion problem using the initial position and velocity vectors of a spacecraft in the Earth-centered inertial coordinate system. The objective was to propagate the spacecraft's trajectory over a 24-hour period and visualize its path around the Earth. The final results accurately demonstrate the orbit's shape and motion under the ideal two-body assumption and verify the correctness of the method against theoretical expectations.

Keywords—Orbit propagation, Two-body problem, Numerical integration, Python, Inertial coordinate system, Spacecraft trajectory

I. Introduction

Orbit determination allows engineers to predict the future position and velocity of a spacecraft based on its initial state. This project focuses on solving the two-body problem numerically and visualizing the trajectory of a spacecraft in Earth orbit.

The problem statement defines initial position and velocity vectors of the spacecraft in the inertial frame with respect to the Earth's center. These vectors serve as the input for solving the two-body differential equation, which governs orbital motion under the assumption of a point-mass Earth. The primary objective is to simulate and visualize the orbit over a 24-hour period, using a custom numerical solver and 3D graphics tools.

II. Problem Statement

To simulate the spacecraft's motion, we adopted the two-body orbital model, which assumes the Earth and the spacecraft as point masses and includes only mutual gravitational interaction. The 2-body equation of motion is given by:

$$\ddot{\vec{r}} = -\mu \frac{\vec{r}}{|\vec{r}|^3} \qquad (1)$$

- where $\vec{r}$ is the position vector of the spacecraft, $\mu = 398600.4418\ \frac{km^3}{s^2}$ is Earth's gravitational parameter, and $\ddot{\vec{r}}$ is the acceleration due to gravity.

A. Initial Conditions

The spacecraft's initial state is defined in the Earth-centered inertial (ECI) frame by the following vectors:

- Position $\vec{r}_0 = 840.5\hat{I} + 485.3\hat{J} + 6905.8\hat{K}\ km$
- Velocity $\vec{v}_0 = 3.7821\hat{I} - 6.5491\hat{J} + 0.0057\hat{K}\ \frac{km}{s}$

These values were combined into a state vector $[\vec{r}_0, \vec{v}_0]$ and serve as the input for the numerical integration of the motion over a time span of 24 hours.

B. Numerical Integration

To solve the two-body differential equations, we used Python. The "odeint" function from Python's "scipy.integrate" module is used. The state vector is propagated using a fixed time step of 10 seconds. The output is a time series of position vectors that describe the spacecraft's trajectory in the inertial reference frame.

C. Visualizations

To gain insight into the resulting orbital motion, several types of visualizations are produced:

- A **static 3D plot** of the complete orbit around Earth.
- A **dynamic animation** showing the time-evolving motion of the spacecraft.
- **Earth-centered view** enhanced with country borders, using real-world GeoJSON data.

All visualizations are created using the **Plotly** library, which provides high-quality, interactive 3D graphics suitable for orbit analysis. The results effectively demonstrate the behavior of the spacecraft in a classical orbital environment and validate the numerical approach.

III. Structure and Implementation Overview

The implementation of this project was modularly designed to improve readability, maintainability, and clarity of purpose. The entire solution was developed in Python and structured into separate files.

The development process began in (orbit_util.py) which contains the core functionality and visualization tools, and then a Jupyter Notebook file (test.ipynb) provides a convenient environment for experimentation, code testing, and iterative visualization. For ease of use, there is a file named (main.py) which has the exact code of (test.ipynb). This file is used for straight forward execution of the project without requiring the Jupyter environment in cases that you cannot run the Jupyter notebook file.

So the code's structure is separated into two Python scripts:

- **orbit_util.py**: Contains the main classes responsible for physics-based orbit propagation (Orbit_2body) and dynamic/static 3D visualization (OrbitVisualizer).
- **test.ipynb or main.py**: Imports the utility classes, initializes the simulation, and calls the visual output routines.

This modular architecture ensures a clear separation between numerical computation and graphical representation, improving maintainability and allowing either component to be reused in other orbit mechanics projects. The final simulation uses trusted scientific libraries such as "NumPy", "SciPy", and "Plotly" to ensure accuracy and high-quality outputs.

IV. Code Explanation

As explained in the previous section, the core implementation of the orbit propagation system is divided into two main components:

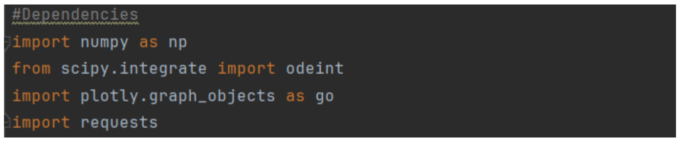
(1) a utility module named **orbit_util.py**, which contains the physics engine and visualization tools.

(2) a Jupyter Notebook file originally named **test.ipynb**, later refactored into a script file (main.py) for simpler and more portable execution.

The code solves the two-body orbital motion problem using Newtonian mechanics and numerically integrates the equations of motion using Python libraries. The resulting spacecraft trajectory is visualized in multiple 3D formats for both analysis and presentation.

A. Dependencies

At the top of the **orbit_util.py** file, several essential Python libraries are imported to support numerical computation, differential equation solving, 3D visualization, and data retrieval:



```python
#Dependencies
import numpy as np
from scipy.integrate import odeint
import plotly.graph_objects as go
import requests
```

Fig. 1. Dependencies

- **numpy**: Used for efficient array operations and numerical computations such as vector concatenation and element-wise math for position and velocity calculations.
- **scipy.integrate.odeint**: An ODE solver used to numerically integrate the system of differential equations describing the two-body orbital motion.
- **plotly.graph_objects**: Enables high-quality 3D static and animated visualizations of the spacecraft's trajectory and Earth.
- **requests**: Fetches live GeoJSON data containing country borders from a public repository to enhance the Earth visualization.

These dependencies provide the foundation for both the scientific calculations and the visual output of the project.

B. Utility Module

This file contains two main classes:

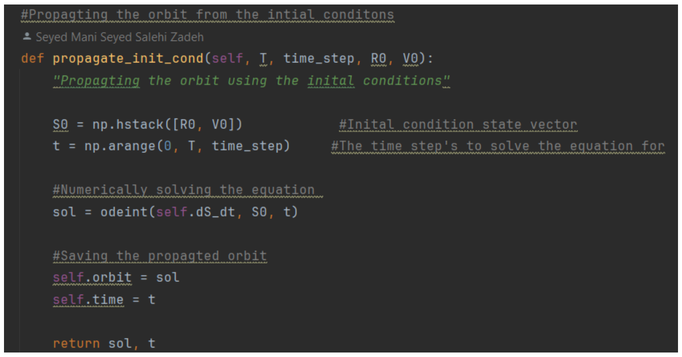
- **Orbit_2body:** Responsible for modeling and solving the two-body differential equations.
- **OrbitVisualizer:** Provides static and dynamic 3D visualizations of the spacecraft's orbit.

1) "Orbit_2body" Class – Two-Body Propagation Model:

Here is the step by step explanation of the class

*a) **Constructor**:* Initializes the class with Earth's gravitational parameter $\mu \ in \ (\frac{km^3}{s^2})$. This value is used throughout the orbital calculations.

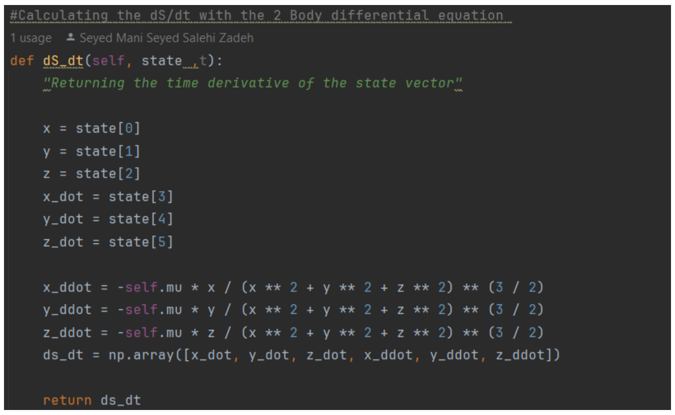
*b) **"propagate_init_cond()" Method**:* First, accepts simulation time T, step size, and initial position R_0 and velocity V_0 vectors. Constructs the initial state vector $\overrightarrow{S_0}$ and then generates the time array using "np.arange". Finally, uses "odeint" to solve the differential equation and finally, returns position and velocity data over time.



```python
#Propagting the orbit from the intial conditons
Seyed Mani Seyed Salehi Zadeh
def propagate_init_cond(self, T, time_step, R0, V0):
    "Propagting the orbit using the inital conditions"

    S0 = np.hstack([R0, V0])            #Inital condition state vector
    t = np.arange(0, T, time_step)     #The time step's to solve the equation for

    #Numerically solving the equation
    sol = odeint(self.dS_dt, S0, t)

    #Saving the propagted orbit
    self.orbit = sol
    self.time = t

    return sol, t
```

Fig. 2."propagate_init_cond()" Method

*c) **"dS_dt()" Method**:* First, decomposes the state vector into position and velocity. Then, computes acceleration using the two-body equation. Finally, returns the full time derivative vector $\frac{d\vec{s}}{dt}$ for integration.



```
#Calculating the dS/dt with the 2 Body differential equation
1 usage   Seyed Mani Seyed Salehi Zadeh
def dS_dt(self, state ,t):
    "Returning the time derivative of the state vector"

    x = state[0]
    y = state[1]
    z = state[2]
    x_dot = state[3]
    y_dot = state[4]
    z_dot = state[5]

    x_ddot = -self.mu * x / (x ** 2 + y ** 2 + z ** 2) ** (3 / 2)
    y_ddot = -self.mu * y / (x ** 2 + y ** 2 + z ** 2) ** (3 / 2)
    z_ddot = -self.mu * z / (x ** 2 + y ** 2 + z ** 2) ** (3 / 2)
    ds_dt = np.array([x_dot, y_dot, z_dot, x_ddot, y_ddot, z_ddot])

    return ds_dt
```

Fig. 3. "dS_dt()" Method

2) "OrbitVisualizer" Class – 3D Visualization Tools: This class creates four different visualizations of the orbit.

*a) **"SimpleStatic()" Method**:*

- Draws a static 3D orbit around a spherical Earth.
- Earth is rendered as a smooth blue sphere.
- The spacecraft's orbit is plotted as a white line.

*b) **"EarthStatic()" Method**:*

- Adds country borders to the static Earth.
- Downloads GeoJSON border data using requests.
- Plots the Earth's land outlines in black and the orbit in red for contrast.

*c) **"SimpleDynamic()" Method**:*

- Animates the orbit over time in a basic environment.
- Displays the current time (in hours) during the animation.
- Ideal for visualizing the dynamic nature of the spacecraft's motion.

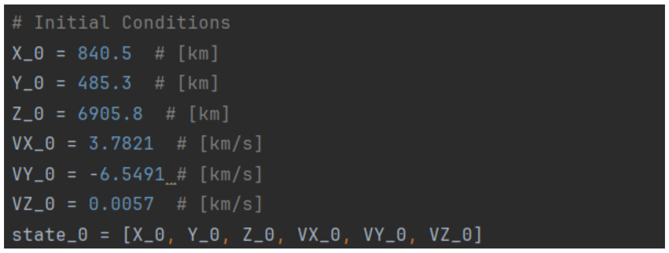
*d) **"EarthDynamic()" Method**:*

- The most detailed animation.
- Combines country borders, realistic Earth rendering, and the orbit line.
- Displays time-stamped frames for a clear view of how the orbit evolves.

C. Simulation Execution Script

This part was originally developed as test.ipynb for iterative testing and visualization, and also converted into main.py for standalone execution.

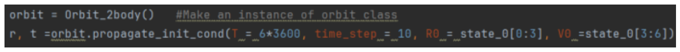
1) Initial Conditions: Defines the spacecraft's initial position and velocity vectors based on the problem statement.



```
# Initial Conditions
X_0 = 840.5   # [km]
Y_0 = 485.3   # [km]
Z_0 = 6905.8   # [km]
VX_0 = 3.7821   # [km/s]
VY_0 = -6.5491 # [km/s]
VZ_0 = 0.0057   # [km/s]
state_0 = [X_0, Y_0, Z_0, VX_0, VY_0, VZ_0]
```

Fig. 4. Initial Condition in Code

2) Propagation: Imports the "Orbit_2body" class. Initializes the orbit engine and runs the numerical propagation over 24 hours, with a 10-second time step. For testing purposes, a shorter duration (e.g., 6 hours) was initially used. However, the final simulation is set to the full 24-hour period.

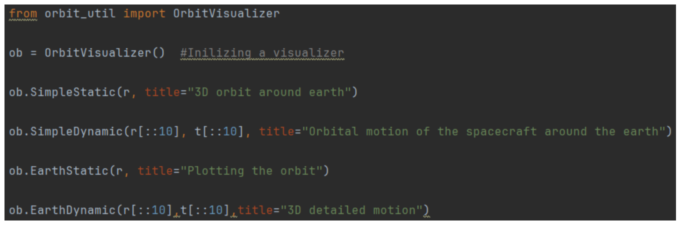


```
orbit = Orbit_2body()   #Make an instance of orbit class
r, t =orbit.propagate_init_cond(T = 6*3600, time_step = 10, R0 = state_0[0:3], V0 =state_0[3:6])
```

Fig. 5. Propagation in Code

3) Visualization: Imports and initializes the visualization class. Each plotting function is called to produce different visual outputs of the orbit. Data is downsampled (e.g., r[::10]) for animations to ensure smooth rendering and reduce computational load.

A note on time step: A smaller time step like 10 seconds allows for higher resolution and more accurate integration results but increases computation time. Increasing the time step reduces simulation time but may compromise accuracy. Therefore, a balance must be maintained between precision and performance.



```
from orbit_util import OrbitVisualizer

ob = OrbitVisualizer()  #Inilizing a visualizer

ob.SimpleStatic(r, title="3D orbit around earth")

ob.SimpleDynamic(r[::10], t[::10], title="Orbital motion of the spacecraft around the earth")

ob.EarthStatic(r, title="Plotting the orbit")

ob.EarthDynamic(r[::10],t[::10],title="3D detailed motion")
```

Fig. 6. Visualization in Code

V. Visualization Improvements

In earlier stages of the project, we used **matplotlib** for visualizing the orbit. Consequently, we transitioned to using **Plotly**. This change significantly enhanced the responsiveness of visualizations. Plotly's pan and zoom tools, as well as its rendering performance, proved to be smoother and faster. The orbit visualization became more dynamic and user-friendly, especially for longer simulations such as the 24-hour case.

Regarding the surface visualization of Earth, we initially used a texture-mapped sphere. Later, we opted for a method using a GeoJSON file that converts geographic coordinates (latitude and longitude) into 3D Cartesian coordinates (x, y, z). These points are plotted as a scatter layer on a blue spherical Earth model. This approach provides more accurate geographic representations and enhances the clarity of the animation.

VI. Results

The simulation successfully propagated the spacecraft's motion using the two-body equation and visualized its trajectory in various formats. The outputs demonstrate the expected behavior of a satellite in Earth orbit, given the initial state vectors defined in the problem statement.

As explained before, the results were generated by integrating the spacecraft's position and velocity over a simulation period of 24 hours. The orbit was computed with a 10-second time step, providing high-resolution data for visualization.

A. 3D Static Visualization

The **SimpleStatic()** method generated a clean 3D view of the orbit as a white trajectory around a blue spherical representation of Earth. This static plot gives a general overview of the orbit's shape and altitude without geographic context.

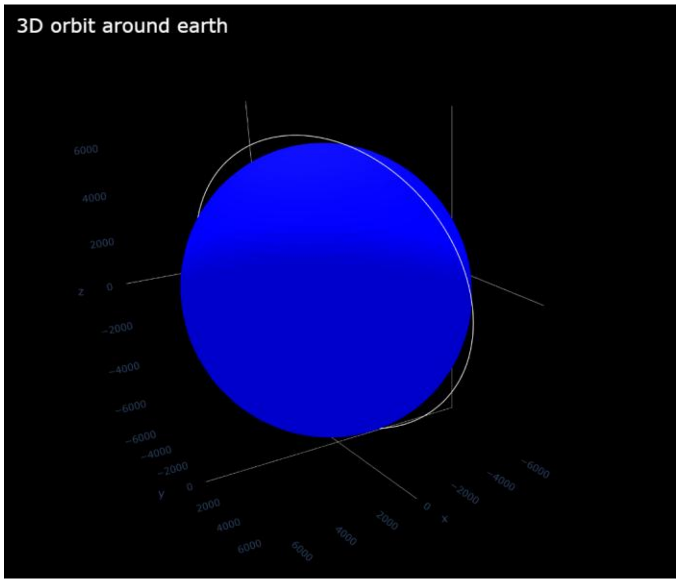


Fig. 7. Static Orbit Visualization

- Shows the full orbit traced from the initial state.
- The Earth is rendered as a uniform blue sphere with radius equal to the average Earth radius (6371 km).
- The orbit appears nearly elliptical, with a slightly inclined trajectory due to the non-zero components of the initial position and velocity.

B. Earth-Referenced Static Visualization

Using the **EarthStatic()** method, a more detailed static view was produced. This plot overlays the spacecraft's orbit on a realistic Earth surface with country borders rendered using GeoJSON data.

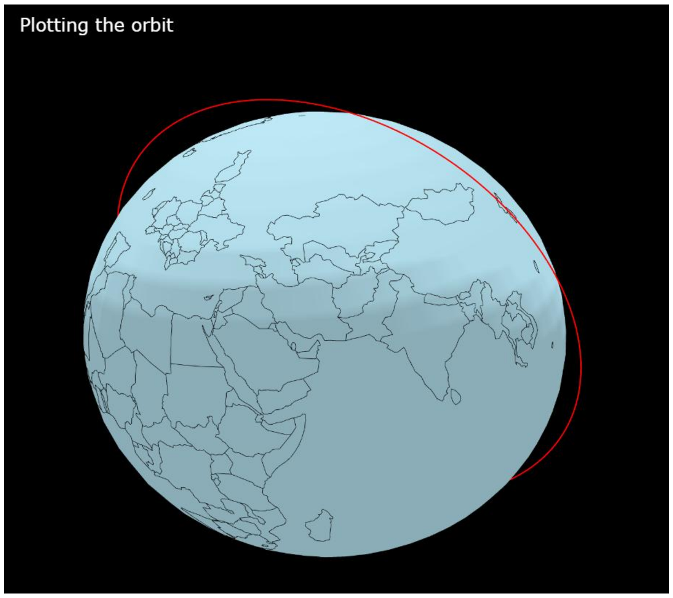


Fig. 8. Static Orbit with Country Borders

- Helps contextualize the orbit over real Earth geography.
- Borders were fetched in real-time from a public dataset, converted into 3D coordinates, and overlaid on the Earth's surface.
- Useful for interpreting which parts of the Earth the spacecraft is passing over during its motion.

C. 3D Animated Visualization

To better understand the motion of the spacecraft over time, the **SimpleDynamic()** function was used to animate the orbit in 3D. The animation traces the orbit incrementally while displaying the elapsed time.

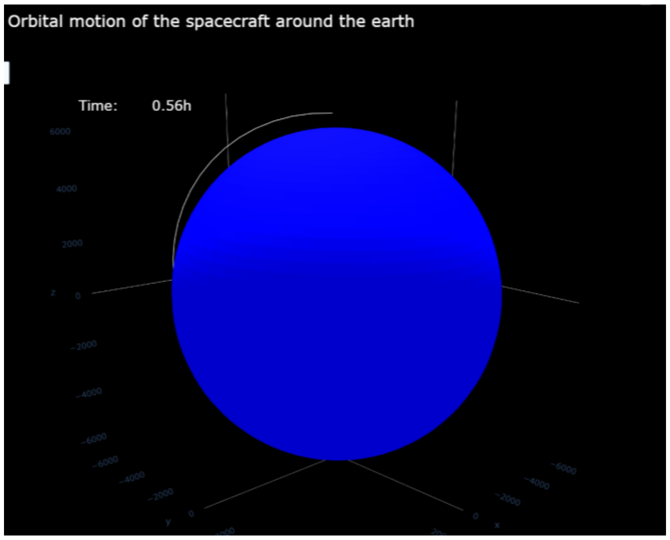


Fig. 9. Orbit Animation (No Borders) after 0.56h

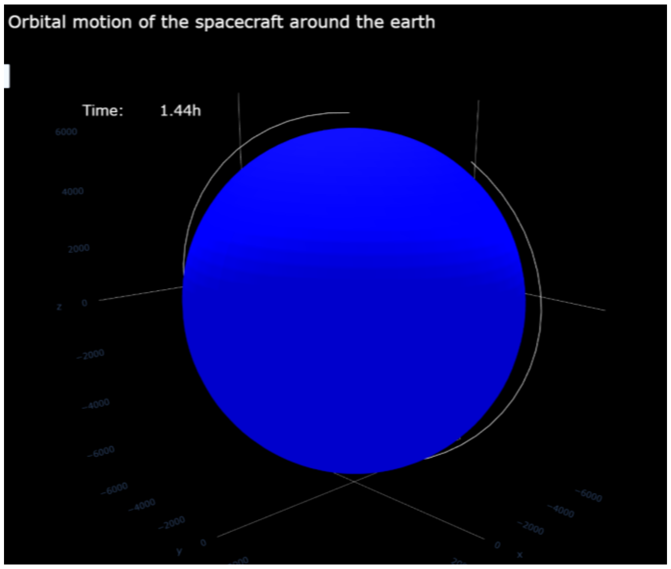


Fig. 10. Orbit Animation (No Borders) after 1.44h

- White orbit trail grows frame-by-frame to show movement.
- Time is displayed in hours to correlate animation frames with the 24-hour simulation.
- Allows viewers to visualize the velocity and progression of the spacecraft's motion in real time.

D. Earth-Referenced Animated Visualization

Finally, the **EarthDynamic()** method provided the most comprehensive visualization. It combined dynamic orbital tracing with an Earth model showing geographic borders.

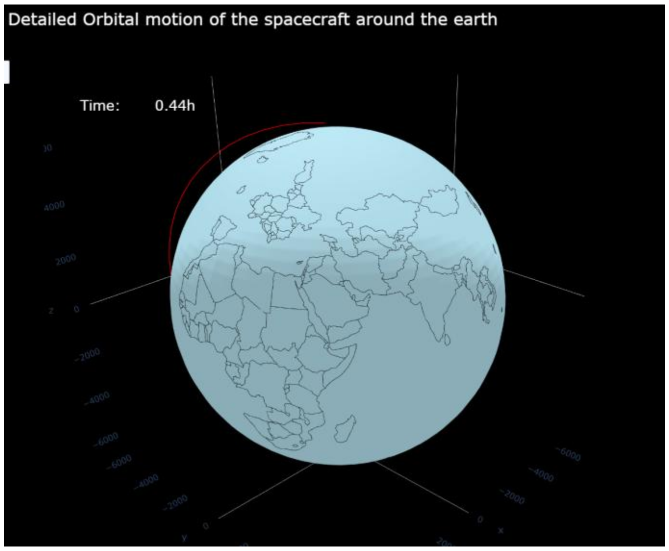


Fig. 11. Orbit Animation with Country Borders after 0.44h

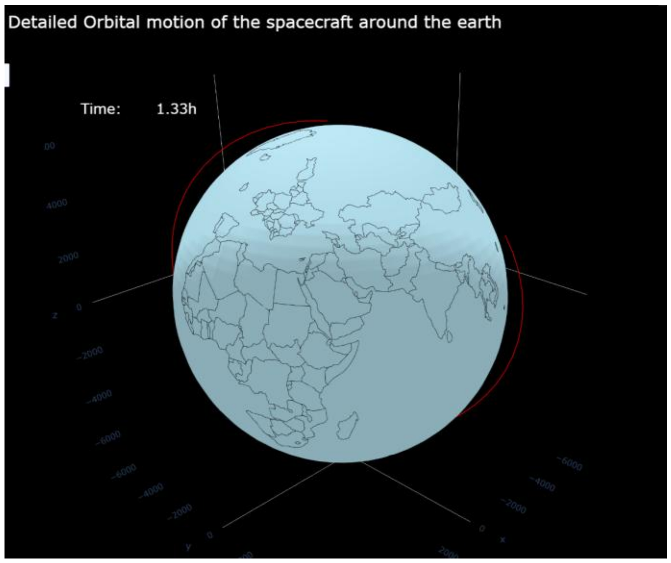


Fig. 12. Orbit Animation with Country Borders after 1.33h

- Highlights the spacecraft's changing position relative to real-world locations.
- Red orbit trail is animated, providing spatial and temporal awareness.
- Particularly effective for presentations and educational demonstrations of orbital motion.

E. Summary of Outputs

TABLE I. OUTPUTS

Visualization Data		
Output Type	***Description***	***Tool Used***
Static Orbit Plot	Basic 3D trajectory over a spherical Earth	SimpleStatic()
Static with Borders	Orbit over realistic Earth with country outlines	EarthStatic()
Animated Orbit	Time-annotated 3D animation (no map)	SimpleDynamic()
Animated with Borders	Orbit animation with Earth surface and borders	EarthDynamic()

All visualizations confirm the correctness of the orbit propagation logic and match the expected results for a spacecraft in Earth orbit governed by two-body gravitational physics.

VII. CONCLUSION

This project successfully implemented a numerical orbit propagation system using the classical two-body model, solving the spacecraft's motion based on given initial conditions in the Earth-centered inertial frame. The simulation demonstrated the theoretical accuracy and predictive capability of the two-body formulation under idealized conditions, where only Earth's gravity is considered.

Python was an effective and flexible platform for this type of engineering simulation. With scientific libraries such as **NumPy**, **SciPy**, and **Plotly**, the development process was efficient, and the output was both accurate and visually compelling. The modular structure of the code allowed for a clean separation between the physical model and visualization components, facilitating reusability and future expansion.

References

[1] D. A. Curtis, *Orbital Mechanics for Engineering Students*, 4th ed., Elsevier, 2020.

[2] SciPy Documentation. Accessed: Apr. 2025. [Online]. Available: https://docs.scipy.org

[3] Plotly Python Graphing Library. Accessed: Apr. 2025. [Online]. Available: https://plotly.com/python/

[4] NumPy Documentation. Accessed: Apr. 2025. [Online]. Available: https://numpy.org/doc/

[5] Natural Earth. “World GeoJSON Data for Country Borders.” [Online]. Available: https://github.com/johan/world.geo.json

[6] Python Requests Documentation. Accessed: Apr. 2025. [Online]. Available: https://docs.python-requests.org

[7] Source Code on GitHub: https://github.com/manisalehi/2-Body-problem-numerical