MSc in Aerospace Engineering

2MAE004: Mission Analysis and Orbital Mechanics

Assignment Report

Guilherme Penedo

October 20, 2021

Contents

Introduction

This assignment was completed using the Python programming language. Additionally, the following libraries were used: NumPy, SciPy and Matplotlib. Apart from these 3 libraries, the remaining code is all original and was developed specifically for this assignment.

This report is split into three main sections, correspondig to three parts of the assignment.

For future reference in this report, below are the main equations used throughout the code, where the usual assumptions (inertial coordinate system, negligible mass of the smaller body, sphericity and uniformity of the bodies and no other forces) are considered: equation 1 is the two body equation of motion while equations 2 to 10 establish relationships between the following physical quantities:

- radius (r)
- velocity (v)
- gravitational parameter of the central body (μ)
- true anomaly (θ)
- semilatus rectum (p)
- semi-major axis (a)
- eccentricity (e)
- magnitude of the specific angular momentum (h)
- periapsis radius (r_p)
- apoapsis radius (r_a)
- orbital period (T)
- mean anomaly (M)
- eccentric anomaly (E)
- time (t)
- time of periapsis passage (t_0)

$$\ddot{\mathbf{r}} + \frac{\mu_E}{r^3}\mathbf{r} = 0 \tag{1}$$

$$r = \frac{p}{1 + e\cos\theta} \tag{2}$$

$$\frac{h^2}{\mu} = a(1 - e^2) = p \tag{3}$$

$$v = \sqrt{(\frac{2}{r} - \frac{1}{a})} \tag{4}$$

$$r(\theta = 0) = r_p = \frac{p}{1 + e} = a(1 - e) \tag{5}$$

$$r(\theta = \pi) = r_a = \frac{p}{1 - e} = a(1 + e) \tag{6}$$

$$T = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{7}$$

$$E - e\sin E = M = n(t - t_0) \tag{8}$$

$$n = \sqrt{\frac{\mu}{a^3}} = \frac{2\pi}{\tau} \tag{9}$$

$$\tan\frac{E}{2} = \sqrt{\frac{1 - e}{1 + e}}\tan\frac{\theta}{2} \tag{10}$$

The following imports were used, and the following constants were defined[1]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from mpl_toolkits.mplot3d import axes3d

R_EARTH = 6378  # km (Radius of the Earth)
G_EARTH = 3.986e5  # km^3/s^2 (gravitational parameter of the Earth)
```

Part I - Solving Kepler's Equation Numerically

1a)

The following function was created to solve Kepler's Equation numerically using the Newton-Raphson method:

```
def kepler(M, e, tol=1e-12):
    """
        Solver for Kepler's equation:
            E - e sinE = M
        Using the Newton-Raphson method:
            f(E) = E - e sinE - M = 0
            f'(E) = 1 - e cosE
            x_(n+1) = x_n - f(x_n)/f'(x_n)
    :param M: mean anomaly, in radians
    :param e: eccentricity
    :param tol: tolerance. Default=1e-12
    :return: tuple(E, iters): E, the eccentric anomaly (in radians); iters, number
                                                        of iterations
    """
    # we use M for the first guess of E
    E = M
    iters = 1
    while True:
        diff = (E - e * np.sin(E) - M) / (1 - e * np.cos(E))
        E -= diff
        if np.abs(diff) < tol:
            return E, iters
        iters += 1
```

Where the Newton Raphson method ($x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$) is used with the following equations, deduced from equation 8:

$$f(E) = E - e \sin E - M = 0 \tag{11}$$

$$f'(E) = 1 - e \cos E \tag{12}$$

1b)

Using this function for $M = 21\,\text{deg}$ and $e = 0.25$:

```
# exercise 1b)
M = np.radians(21)
e = 0.25

# after one iteration (very high tolerance)
E, _ = kepler(M, e, 9e99)
```

[1]The full code can be found in the following github repository: `https://github.com/guipenedo/mission-analysis-assignment`

```
print(f"E after 1 iteration: {E:.2f} radians = {np.degrees(E):.2f} degrees")

# number of iterations for error < 10^(-6)
E, iters = kepler(M, e, 1e-6)
print(f"E and number of iterations for error < 10^(-6): E = {E:.2f} radians = {np.
                                        degrees(E):.2f} degrees in {iters}
                                        iterations")
```

```
E after 1 iteration: 0.48 radians = 27.70 degrees

E and number of iterations for error < 10^(-6):
E = 0.48 radians = 27.65 degrees in 3 iterations
```

For a tolerance of 10^{-12} and semi-major axis of $24000km$, the eccentric anomaly, number of iterations, true anomaly and radial distance were computed, using two auxiliary functions, whose code may be found in the appendix:

```
def peinr_eccentric_anomaly_theta_r():
    a = 24000  # km

    # for tol = 10^(-12)
    E, iters = kepler(M, e)
    print(f"Eccentric anomaly={np.degrees(E):.5f} degrees, computed in {iters}
                                        iterations.")

    theta = eccentric_anomaly_to_theta(E, e)
    r = theta_to_r_distance(theta, e, a)
    print(f"True anomaly={np.degrees(theta):.3f} degrees; r = {r:.0f} km")


peinr_eccentric_anomaly_theta_r()
```

```
Eccentric anomaly=27.64653 degrees, computed in 4 iterations.
True anomaly=35.245 degrees; r = 18685 km
```

For $M = 180\,\text{deg}$, running the same code:

```
# change M to 180 degrees
M = np.radians(180)
peinr_eccentric_anomaly_theta_r()
```

```
Eccentric anomaly=180.00000 degrees, computed in 1 iterations.
True anomaly=180.000 degrees; r = 30000 km
```

In this last case of $M = 180\,\text{deg}$, the eccentric anomaly and the true anomaly are both equal to the mean anomaly: $180\,\text{deg}$. This means that we are at the apogee and, therefore, the apogee radius (r_a) is the obtained radius (r): $30000km$.

1c)

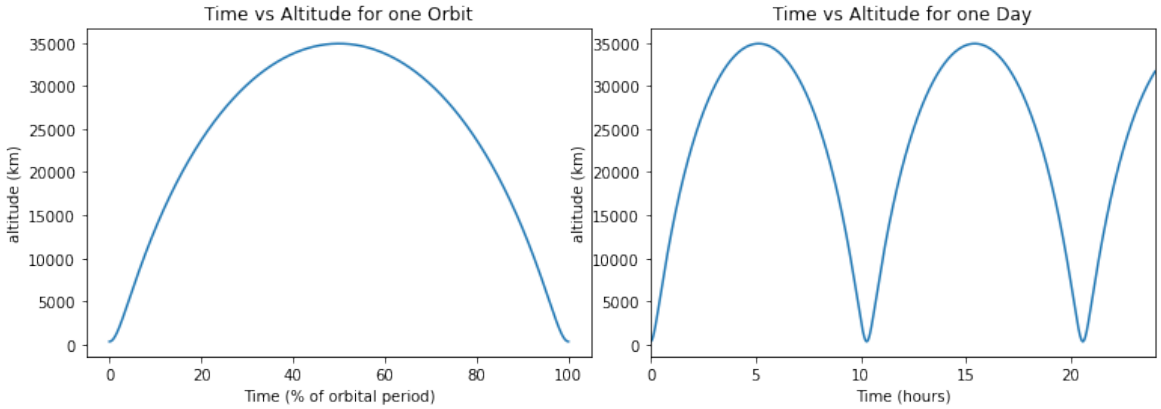
The *kepler* function was used to compute the orbit of a MEO satellite. This orbit was computed every 15 seconds, considering that at t_0 the spacecraft was at perigee, $a = 24000km$ and $e = 0.72$. The auxiliary function *ts_to_alts* is defined in the appendix.

The obtained plots can be seen in figure 1. We can see the altitude increasing until 50% of the orbital period (apogee) and then decreasing in a symetric fashion. On the second plot we can see that there are almost two and a half periods in a full day.

If, instead of being at the perigee ($t_0 = 0$), the satellite were at the apogee at t_0, then $t_0 = \frac{T}{2}$ and therefore the computation of M (which is inside the *ts_to_alts* function) would become $M = n(t - \frac{T}{2})$ instead of $M = nt$.

```python
# orbit parameters
a = 24000
e = 0.72
n = np.sqrt(G_EARTH / np.power(a, 3))  # mean motion
T = 2 * np.pi / n  # period
p = a * (1 - e * e)
print(f"{n=} {T=:.3f}s")

# for one orbit, we compute from 0 to the period:
percts_ts = np.arange(0, T, 15.0)
percts_hs, percts_thetas = ts_to_alts(percts_ts, e, a)

plt.plot(np.divide(percts_ts, T / 100.0), percts_hs)  # divide times by T and * 100 to get percentage
plt.title("Time vs Altitude for one Orbit")
plt.xlabel("Time (% of orbital period)")
plt.ylabel("altitude (km)")
plt.show()

# for one day:
day_ts = np.arange(0, 24 * 3600, 15.0)
day_hs, _ = ts_to_alts(day_ts, e, a)

plt.plot(np.divide(day_ts, 3600.0), day_hs)  # divide times by 3600 to get hours
plt.title("Time vs Altitude for one Day")
plt.xlim([0, 24])
plt.xlabel("Time (hours)")
plt.ylabel("altitude (km)")
plt.show()
```

```
n=0.00016980551238707486 T=37002.246s
```



(a) Time vs Altitude for one Orbit (b) Time vs Altitude for one Day

Figure 1: Time vs Altitude

1d)

For this section two eclipse cases were considered: the case where the Earth eclipses at the periapsis and the case where it eclipses at the apoapsis.

Using the data from the previous question, a 2D plot of the orbit around Earth was obtained. The helper function *"theta_to_x_y"*, which converts angular position to (x,y) pairs, can be found in the appendix.

The eclipse angular positions are represented on the plot with a blue "*" (eclipse at periapsis) and with an orange "o" (eclipse at apoapsis). The angular positions were calculated using the following equations:

$$\alpha \cos^2 \theta + \beta \cos \theta + \gamma = 0 \tag{13}$$

$$\alpha = R_E^2 e^2 + p^2 \tag{14}$$

$$\beta = 2R_E^2 e \tag{15}$$

$$\gamma = R_E^2 - p^2 \tag{16}$$

Solving equation 13:

$$\theta = \arccos \frac{-\beta \pm \sqrt{\beta^2 - 4\alpha\gamma}}{2\alpha} \tag{17}$$

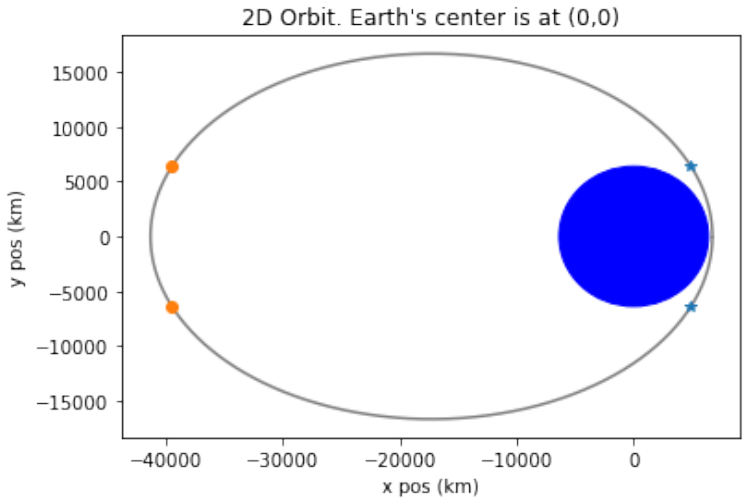


Figure 2: 2D Orbit with eclipse positions

```
alpha = R_EARTH * R_EARTH * e * e + p * p
beta = 2 * R_EARTH * R_EARTH * e
gamma = R_EARTH * R_EARTH - p * p
b = np.sqrt(beta * beta - 4 * alpha * gamma)
X1 = (-beta + b)/(2 * alpha)
X2 = (-beta - b)/(2 * alpha)
theta1 = np.arccos(X1)
theta2 = np.arccos(X2)

# convert (height, theta) pairs into (x, y)
xs, ys = theta_h_to_x_y(percts_thetas, percts_hs)

# eclipse points
# eclipse at periapsis
p_e_xs, p_e_ys = theta_to_x_y(np.array([-theta1, theta1]), e, a)
# eclipse at apoapsis
a_e_xs, a_e_ys = theta_to_x_y(np.array([-theta2, theta2]), e, a)

plt.plot(xs, ys, color='grey')
```

```
plt.plot(p_e_xs, p_e_ys, '*')
plt.plot(a_e_xs, a_e_ys, 'o')
plt.xlabel("x pos (km)")
plt.ylabel("y pos (km)")
plt.title("2D Orbit. Earth's center is at (0,0)")
earth = plt.Circle((0, 0), R_EARTH, color='b')
plt.gca().add_patch(earth)
plt.show()
```

The eclipse time spent by the satellite over one orbit can be computed for each of these two cases, using equations 8, 9 and 10. The helper function that performs this calculation is called *"delta_ theta_ to_ delta_ t"* and can also be found in the appendix.

Running these functions yields the desired times:

```
# 2 * delta t (periapsis -> theta1)
delta_t_1 = (2 * delta_theta_to_delta_t(theta1, e, a)) / 60.0
# T - 2 * delta t (periapsis -> theta2)
delta_t_2 = (T - 2 * delta_theta_to_delta_t(theta2, e, a)) / 60.0

print(f"Time in eclipse at periapsis: {delta_t_1:.3f} minutes")
print(f"Time in eclipse at apoapsis: {delta_t_2:.3f} minutes")
```

```
Time in eclipse at periapsis: 23.018 minutes
Time in eclipse at apoapsis: 131.266 minutes
```

Part II - Solving Equation of Motion using ODE solvers

2a)

We start by defining a state vector $\mathbf{X}$ and its derivative $\dot{\mathbf{X}}$ which may be obtained directly from the two body equation of motion (equation 1):

$$\mathbf{X} = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} ; \dot{\mathbf{X}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \frac{-\mu x}{\sqrt{x^2+y^2+z^2}^3} \\ \frac{-\mu y}{\sqrt{x^2+y^2+z^2}^3} \\ \frac{-\mu z}{\sqrt{x^2+y^2+z^2}^3} \end{bmatrix} \tag{18}$$

This defines a system of differential equations $\dot{\mathbf{X}} = f(t, \mathbf{X}) = f(\mathbf{X})$ (the time isn't actually used).

2b)

This system of differential equations was implemented in python in vectorized (and very compact) form:

```
def twobody(_, X):
    return np.concatenate((X[3:], - G_EARTH * X[:3] / np.power(np.linalg.norm(X[:3]
                                                    ), 3)))
```

2c)

The previously defined *twobody* function can be used together with one of *SciPy*'s ODE solvers to integrate the equations of motion of the two-body problem.

Doing this for the initial position and velocity state vector $\mathbf{X_0} = (\mathbf{r}, \mathbf{v})^T$ where:

$$\mathbf{r} = [7115.804; 3391.696; 3492.221]km$$

$$\mathbf{v} = [-3.762; 4.063; 4.184]km/s$$

And using a relative and absolute tolerance of $1e-12$, we can obtain the position and velocity vectors for an entire day, evaluated every 10 seconds:

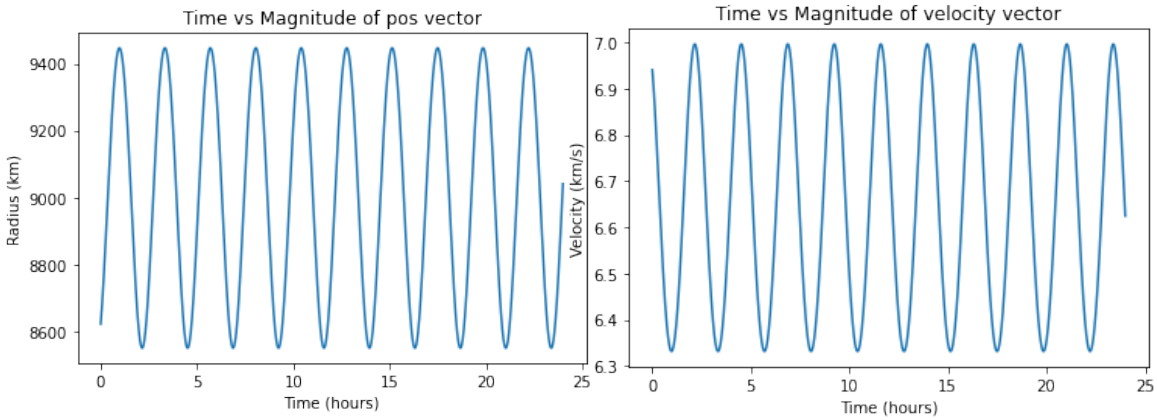
```
def twobody(_, X):
    return np.concatenate((X[3:], - G_EARTH * X[:3] / np.power(np.linalg.norm(X[:3]
                                                    ), 3)))

r0 = np.array([7115.804, 3391.696, 3492.221])
v0 = np.array([-3.762, 4.063, 4.184])

X0 = np.concatenate((r0, v0))

ts = np.arange(0, 3600 * 24, 10)

sol = solve_ivp(twobody, [0, np.max(ts)], X0, t_eval=ts, rtol=1e-12, atol=1e-12,
                                              vectorized=True)
pos = sol.y[:3,:]
vels = sol.y[3:,:]
```

Now that we have obtained the position and velocity vectors we can plot their evolution over time (figure 3).

```
# time vs magnitude of the position vector
plt.plot(ts / 3600.0, np.linalg.norm(pos, axis=0))
plt.xlabel("Time (hours)")
plt.ylabel("Radius (km)")
plt.title("Time vs Magnitude of pos vector")
plt.show()

# time vs magnitude of the velocity vector
plt.plot(ts / 3600.0, np.linalg.norm(vels, axis=0))
plt.xlabel("Time (hours)")
plt.ylabel("Velocity (km/s)")
plt.title("Time vs Magnitude of velocity vector")
plt.show()
```

Using the position data we can also plot the orbit (in green) around the Earth (in blue), in figure 4.

```
# plot
fig = plt.figure()
ax = axes3d.Axes3D(fig)
ax.scatter(*pos, color='green')

# draw earth
u, v = np.mgrid[0:2*np.pi:20j, 0:np.pi:20j]
x = R_EARTH * np.cos(u) * np.sin(v)
y = R_EARTH * np.sin(u) * np.sin(v)
z = R_EARTH * np.cos(v)
ax.plot_surface(x, y, z, color="b")

plt.show()
```



(a) Time vs Magnitude of position vector

(b) Time vs Magnitude of velocity vector

Figure 3: Time vs Magnitude during 1 day

2d)

From the position and velocity vector data obtained in the previous section, we can also calculate the specific energy (kinetic, potential and total) and the specific angular momentum. According to the theory, the total specific energy as well as the specific angular momentum should remain constant throughout the orbit.

Plotting these quantities in figure 5, we can see that this does indeed happen: a variation in kinetic energy always comes with a symetric variation in the potential energy, so that the total energy indeed remains constant. The specific angular momentum is also constant.

```
# time vs specific energies
# kinetic energy: 0.5 * v^2
kinetic = 0.5 * np.power(np.linalg.norm(vels, axis=0), 2)
# potential energy: - g_param/r
potential = - G_EARTH / np.linalg.norm(pos, axis=0)

plt.plot(ts / 3600.0, kinetic)
plt.plot(ts / 3600.0, potential)
plt.plot(ts / 3600.0, kinetic + potential)
plt.legend(labels=('Kinetic', 'Potential', 'Total'))
plt.xlabel("Time (hours)")
plt.ylabel("Specific Energy (km^2/s^2)")
plt.title("Time vs Specific Energy")
plt.show()

# time vs specific angular momentum
angular_momentum = np.linalg.norm(np.cross(pos, vels, axis=0), axis=0)

plt.plot(ts / 3600.0, angular_momentum)
plt.xlabel("Time (hours)")
plt.ylabel("Angular Momentum (km^2/s)")
plt.title("Time vs Specific Angular Momentum")
plt.ylim((50000, 70000))
plt.show()
```

2e)

For a satellite with initial position and velocity vectors as follows:

$$\mathbf{r} = [0; 0; 8550] km$$

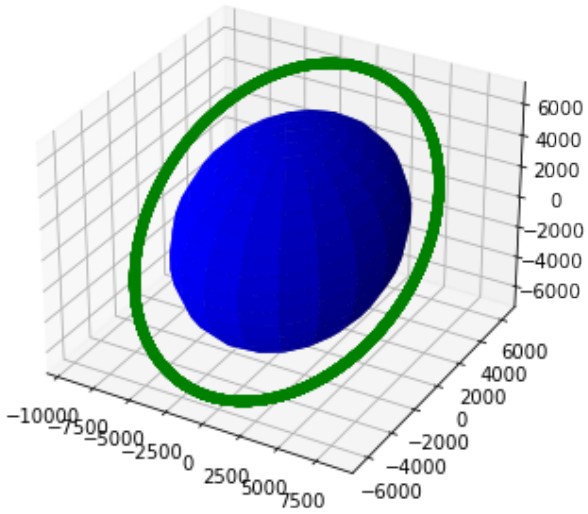


Figure 4: 3D Orbit around the Earth

$$\mathbf{v} = [0; -7.0; 0] km/s$$

We can determine the positions and velocities of this satellite's orbit using the two-body ODE solver:

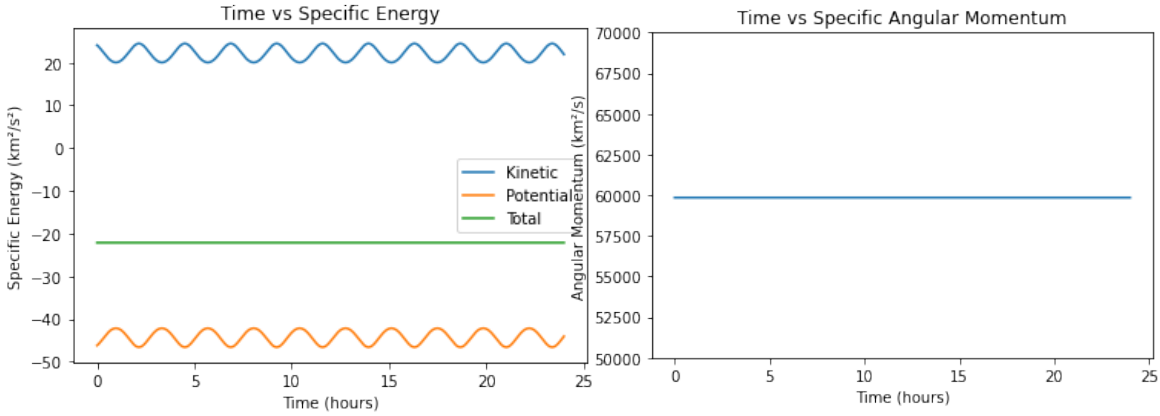
```
r0 = np.array([0, 0, 8550])
v0 = np.array([0, -7.0, 0])

X0 = np.concatenate((r0, v0))

ts = np.arange(0, 3600 * 24, 10)

sol = solve_ivp(twobody, [0, np.max(ts)], X0, t_eval=ts, rtol=1e-12, atol=1e-12,
                                              vectorized=True)
pos = sol.y[:3,:]
vels = sol.y[3:,:]
```

We can now plot the magnitudes of the position and velocity vectors in figure 6.

```
# time vs magnitude of the position vector
plt.plot(ts / 3600.0, np.linalg.norm(pos, axis=0))
plt.xlabel("Time (hours)")
plt.ylabel("Radius (km)")
plt.title("Time vs Magnitude of pos vector")
plt.show()

# time vs magnitude of the velocity vector
plt.plot(ts / 3600.0, np.linalg.norm(vels, axis=0))
plt.xlabel("Time (hours)")
plt.ylabel("Velocity (km/s)")
plt.title("Time vs Magnitude of velocity vector")
plt.show()
```

We can also calculate some additional orbital parameters, using the position and velocity data, such as the semi-major axis (a), inclination (i), eccentricity (e) and period (T):

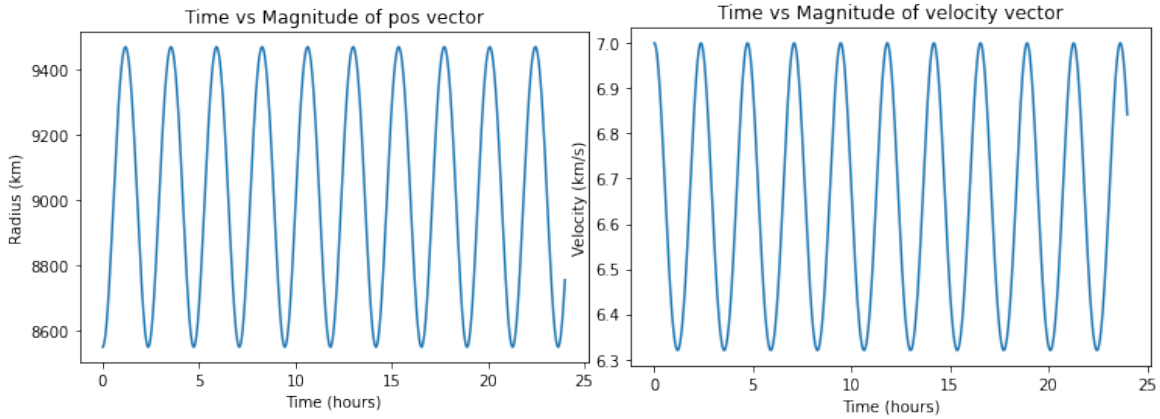
```
## calculate a
# specific energy = kinetic energy + potential energy: 0.5 * v^2
spec_energy = 0.5 * np.power(np.linalg.norm(vels, axis=0), 2) - G_EARTH / np.linalg
                                              .norm(pos, axis=0)
```



(a) Time vs Specific Energy

(b) Time vs Specific Angular Momentum

Figure 5: Energy and Angular Momentum evolution



(a) Time vs Radial Distance

(b) Time vs Magnitude of Velocity Vector

Figure 6: Magnitudes of Position and Velocity vectors

```python
# energy = - mu/(2a) => a = -mu/(2energy)
a = np.average(- G_EARTH / (2 * spec_energy))
print(f"{a=:.3f} km")

## calculate inclination
angular_momentum = np.average(np.cross(pos, vels, axis=0), axis=1)
z = np.array((0, 0, 1))
i = np.degrees(np.arccos(np.dot(angular_momentum, z) / np.linalg.norm(
                                                    angular_momentum)))
print(f"{i=} degrees")

## eccentricity
e = np.sqrt(1 - np.power(np.linalg.norm(angular_momentum), 2) / (G_EARTH * a))
print(f"{e=:.3f}")

## calculate period
period = 2 * np.pi * np.sqrt(np.power(a, 3) / G_EARTH) / 3600.0
print(f"{period=:.5f} hours")
```

```
a=9009.993 km
i=90.0 degrees
```

```
e=0.051
period=2.36426 hours
```

From the obtained parameters, we can conclude that this satellite is in a polar orbit (inclination of 90 degrees) and that this orbit is almost circular (eccentricity of 0.05).

Part III - Orbit Phasing and Rendezvous

In this section, we consider a "chaser" spacecraft that catches up and docks with the ISS, applies a certain change in velocity and then de-orbits and burns-up upon reentry to the Earth's atmosphere.

As the chaser is meant to intercept the ISS, it is placed in an equatorial orbit with an eccentricity such that its apogee coincides with the orbital radius of the ISS (we consider the ISS's orbit to be circular and equatorial).

Denoting the initial angular separation between the chaser and the ISS by $\Delta\Theta$, the period of the chaser by T_{chaser}, the chaser and the ISS's mean motion by n_{chaser} and n_{ISS}, respectively, and by defining N_{rev} to be the number of orbits after which the chaser will catch up to the ISS, we can define the following relations:

$$N_{rev}T_{chaser}(n_{chaser} - n_{ISS}) = \Delta\Theta \tag{19}$$

$$a_{chaser} = (1 - \frac{\Delta\Theta}{2\pi N_{rev}})^{\frac{2}{3}} a_{ISS} \tag{20}$$

$$e_{chaser} = \frac{a_{ISS}}{a_{chaser}} - 1 \tag{21}$$

For the following sections we will consider a 2D (the ISS and the chaser are on the same orbital plane) referential frame where the chaser's apogee is on the x axis (y=0). We therefore measure the initial angular separation from this axis, counter clock wise.

3a)

Considering the ISS's orbit to have 404 km of altitude, for a $\Delta\Theta = 100\,\text{deg}$, we can plot the orbit using our ODE solver, as can be seen in figure 7.

```
1  R_ISS = R_EARTH + 404  # km
2  D_THETA = np.radians(100)  # initial angular separation in radians
3
4  V_ISS = np.sqrt(G_EARTH / R_ISS)
5
6  r0 = np.array([R_ISS * np.cos(D_THETA), R_ISS * np.sin(D_THETA), 0])
7  v0 = np.array([-V_ISS * np.sin(D_THETA), V_ISS * np.cos(D_THETA), 0])
8
9  X0 = np.concatenate((r0, v0))
10
11 # 2.5 hours (a bit more than the period)
12 ts = np.arange(0, 3600 * 2.5, 10)
13
14 def solve_for_ts(X0, ts):
15     def twobody(_, X):
16         return np.concatenate((X[3:], - G_EARTH * X[:3] / np.power(np.linalg.norm(X
                                                                [:3]), 3)))
17
18     sol = solve_ivp(twobody, [0, np.max(ts)], X0, t_eval=ts, rtol=1e-12, atol=1e-12
                                                    , vectorized=True)
19     return sol.y[:2, :], sol.y[3:5, :]
20
21 pos2d, vels2d = solve_for_ts(X0, ts)
22
```

```
# time vs magnitude of the velocity vector
plt.scatter(*pos2d)
plt.xlabel("X (km)")
plt.ylabel("Y (km)")
plt.title("ISS orbit")
plt.gca().set_aspect('equal')
plt.axhline(0, color='black')
plt.axvline(0, color='black')
plt.quiver(0, 0, *r0[:2], color=['r'], angles='xy', scale_units='xy', scale=1)
plt.show()
```

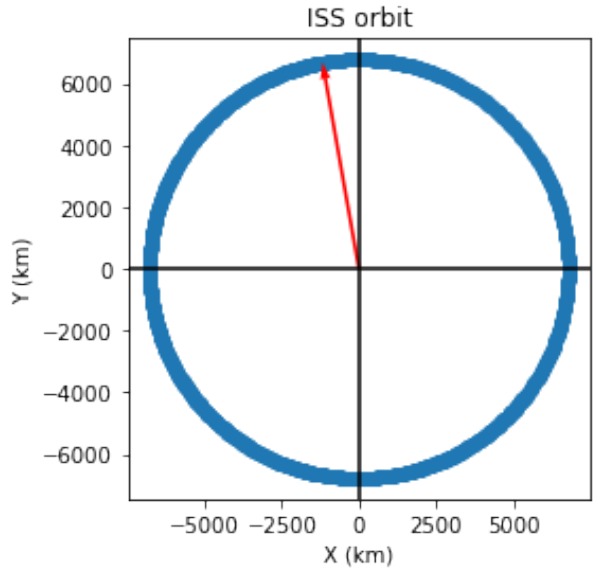


Figure 7: 2D Orbit of the ISS with initial position in red

We can obtain the period directly from the existing data, by calculating the first time instant where the ISS returns to the initial position ($\Delta\Theta = 100\,\text{deg}$). This yields a value of 5560 seconds, which is very close to the value obtained using equation 7: 5558.4 seconds. The difference stems from the fact that our ODE solver only evaluates the position once every 10 seconds.

```
# theta, measured from x axis CCW
thetas = np.arctan2(pos2d[1, :], pos2d[0, :])

# we can see in the plot that we have less than 2 periods
plt.plot(ts, np.degrees(thetas))
plt.xlabel("Time (seconds)")
plt.ylabel("Theta (degrees)")
plt.show()

# period computation from obtained thetas: time until theta is 100 again
print(ts[np.where(np.abs(thetas - D_THETA) < 1e-2)][-1])

# exact period computation
T_ISS = 2*np.pi*np.sqrt(np.power(R_ISS, 3) / G_EARTH)
print(T_ISS)  # the difference between the two comes from the fact we are
                                        considering time in increments of 10
                                        seconds
```

```
5560.0
5558.375715461923
```

The time interval currently being considered (2.5 hours, which was tweaked to give the desired behaviour), is, as can be seen from figure 8, less than 2 periods. Therefore, by simply taking the latest value equal (or very close to) 100 degrees, we obtain an approximate value for the period (the already mentioned value of 5560 seconds).

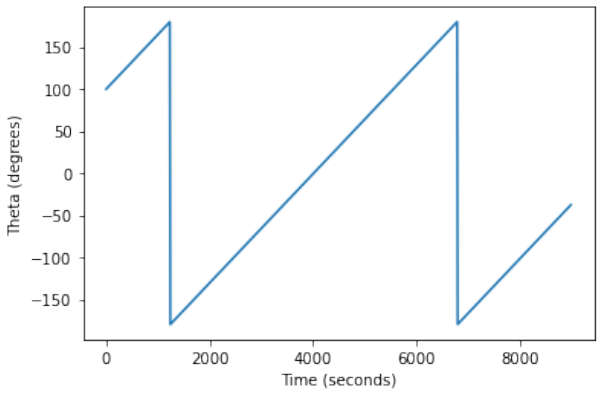


Figure 8: ISS theta over time

3b)

Assuming we want the chaser to intercept the ISS after $N_{rev} = 12$ orbits, we can use equations 20 and 21 to calculate the chaser orbit's semi-major axis and eccentricity:

```
N_REV = 12
a_chaser = np.power(1 - D_THETA / (2 * np.pi * N_REV), 2/3) * R_ISS
print(f"{a_chaser=:.3f} km")

e_chaser = R_ISS / a_chaser - 1
print(f"{e_chaser=:.3f}")
```

```
a_chaser=6676.932 km
e_chaser=0.016
```

As the chaser is at apogee at $t = 0$, with $y = 0$, its velocity only has a non zero component on the y direction. It's initial state vector is therefore:

$$\mathbf{X_0} = \begin{bmatrix} R_E + 404 \\ 0 \\ 0 \\ 0 \\ \sqrt{(\frac{2}{R_E+404} - \frac{1}{a_{chaser}})} \\ 0 \end{bmatrix} \tag{22}$$

Propagating its orbit using the ODE solver, we can now plot the distance between the ISS and chaser over time, as was done in figure 9. The minimum is, as was expected, at $12T_{period}$ seconds.

3c)

```
# velocity at apogee
v_abs_c = np.sqrt(G_EARTH * (2/R_ISS - 1/a_chaser))

```

```
r0_c = np.array([R_ISS, 0, 0])
v0_c = np.array([0, v_abs_c, 0])
X0_c = np.concatenate((r0_c, v0_c))

T_chaser = 2 * np.pi * np.sqrt(np.power(a_chaser, 3) / G_EARTH)

# solve for N_REV revolutions
ts = np.arange(0, T_chaser * (N_REV + 0.5), 10)

pos2d_iss, vels2d_iss = solve_for_ts(X0, ts)
pos2d_c, vels2d_c = solve_for_ts(X0_c, ts)

dist = np.linalg.norm(pos2d_c - pos2d_iss, axis=0)
print(f"Minimum distance={min(dist)*1000:.0f}m")
plt.plot(ts, dist)
plt.title("Distance between the ISS and chaser")
plt.xlabel("Time (seconds)")
plt.ylabel("Distance (km)")
plt.show()
```

```
Minimum distance=211m
```

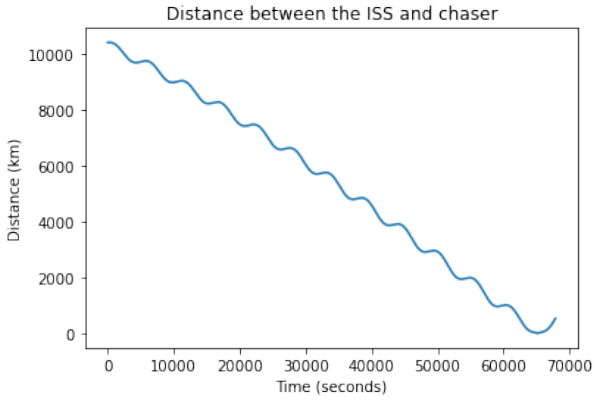


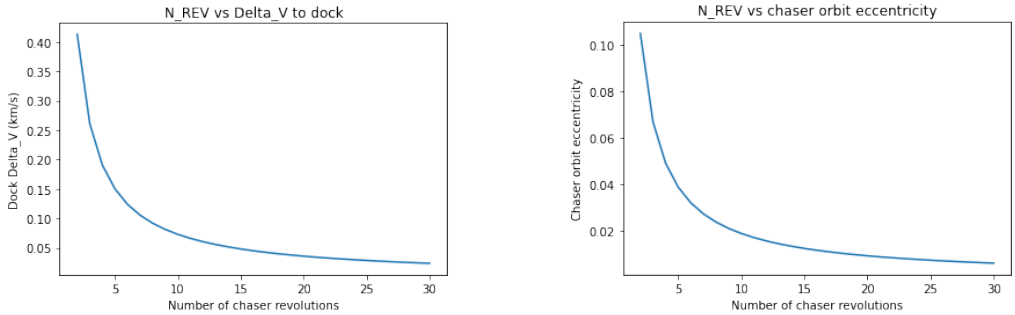
Figure 9: ISS-chaser distance over time

3c)

For the ΔV needed for the chaser to match the ISS's speed at the apogee, we consider an interval of possible N_{rev} from 2 to 30, and determine the required ΔV for each of them, which can be seen in figure 10a. The required ΔV decreases as the number of revolutions increase, and starts to approach 0. If we look at the plot on figure 10b, we can see that the orbit eccentricity is also approaching 0: what is happening is that as the number of N_{rev} increase, the orbit is slowly becoming circular, which, as N_{rev} tends to infinity would mean that the chaser orbit would become the ISS's orbit (and therefore not require a ΔV at all.

```
N_REVS = list(range(2, 30 + 1))

delta_vs = []
es_chaser = []
for N_REV in N_REVS:
    # calculate a and e
    a_chaser = np.power(1 - D_THETA / (2 * np.pi * N_REV), 2/3) * R_ISS
    es_chaser.append(R_ISS / a_chaser - 1)

    # calculate velocity at apogee
```

```
    v_abs_c = np.sqrt(G_EARTH * (2/R_ISS - 1/a_chaser))

    delta_vs.append(np.abs(V_ISS - v_abs_c))

plt.plot(N_REVS, delta_vs)
plt.title("N_REV vs Delta_V to dock")
plt.xlabel("Number of chaser revolutions")
plt.ylabel("Dock Delta_V (km/s)")
plt.show()

plt.plot(N_REVS, es_chaser)
plt.title("N_REV vs chaser orbit eccentricity")
plt.xlabel("Number of chaser revolutions")
plt.ylabel("Chaser orbit eccentricity")
plt.show()
```



(a) Docking ΔV for each N_{rev}

(b) Orbit eccentricity for each N_{rev}

Figure 10: N_{rev} effect on ΔV

3d)

To undock and re-enter, another ΔV will be required. The code below calculates and plots the absolute value of the ΔV needed to lower the perigee of the chaser to the range of 60 to 210 km altitude (without changing the apogee). The resulting plot can be seen in figure 11.

```
# apogee unchanged: r_apogee = R_ISS
# for perigee = 60 km altitude
h_pers = np.arange(60, 210 + 1)
r_per = R_EARTH + h_pers
a_lower_per = (r_per + R_ISS) / 2
v_apogee = np.sqrt(G_EARTH * (2/R_ISS - 1/a_lower_per))
delta_vs_lower_per = np.abs(V_ISS - v_apogee)

plt.plot(h_pers, delta_vs_lower_per)
plt.title("Perigee altitude vs delta_v")
plt.xlabel("Perigee altitude (km)")
plt.ylabel("Undock Delta_V (km/s)")
plt.show()
```

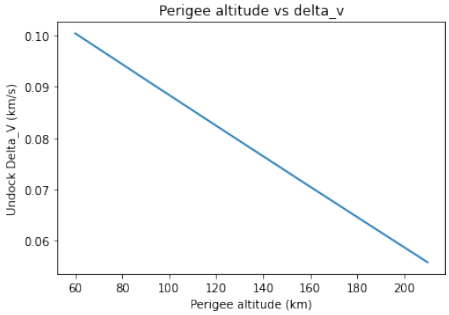


Figure 11: ΔV to lower perigee to a certain value

Appendix

Below are some helper functions developed to calculate a variety of parameters:

```
1
2  def eccentric_anomaly_to_theta(E, e):
3      """
4          Calculates the true anomaly, given an eccentric anomaly and the orbit's
                                                    eccentricity,
5          from the formula:
6              tan(E/2) = sqrt((1-e)/(1+e))*tan(theta/2)
7              => theta = 2*arctan(tan(E/2)*sqrt((1+e)/(1-e)))
8      :param E: eccentric anomaly, in radians
9      :param e: eccentricity
10     :return: the true anomaly, in radians
11     """
12     return 2 * np.arctan(np.tan(E / 2.0) * np.sqrt((1.0 + e) / (1.0 - e)))
13
14
15 def theta_to_r_distance(theta, e, a):
16     """
17         Calculates radial distance for a certain theta, e and a
18         from the formula:
19             r = a * (1-e^2) / (1 + e cos(theta))
20     :param theta: true anomaly, in radians
21     :param e: eccentricity
22     :param a: semi-major axis, in km
23     :return:
24     """
25     return a * (1.0 - e*e) / (1.0 + e * np.cos(theta))
26
27
28 def thetas_to_alts(theta, e, a):
29     """
30         Calculates altitude from theta, for a particular orbit (e and a)
31     :param theta: true anomaly, in radians
32     :param e: eccentricity
33     :param a: semi-major axis, in km
34     :return: altitude, in km
35     """
36     return theta_to_r_distance(theta, e, a) - R_EARTH
37
38
39 def ts_to_alts(ts, e, a):
40     """
41         Calculates altitudes for a list of timestamps for a given orbit (e and a),
                                                    assuming t0 is at perigee
42         M = n * t
43         E calculates from M using Newton-Raphson
44         E -> theta
45         theta -> altitude
46     :param ts: array of timestamps, in seconds
47     :param e: eccentricity
48     :param a: semi-major axis, in km
49     :return: tuple of two arrays: altitudes and thetas, both with the same size as
                                                ts parameter
50     """
51     Ms = np.sqrt(G_EARTH / np.power(a, 3)) * ts  # n * t
52     Es = np.empty_like(Ms)
53     # this one can't be vectorized, as we're running Newton-Raphson on each
                                                individual entry
54     for i in range(len(Es)):
55         Es[i], _ = kepler(Ms[i], e)
56     thetas = eccentric_anomaly_to_theta(Es, e)
57     hs = thetas_to_alts(thetas, e, a)
58     return hs, thetas
59
```

```
60
61  def theta_h_to_x_y(thetas, heights):
62      """
63          Convert (theta, height) pairs to (x, y) pairs
64      :param thetas: array of thetas (true anomalies) in radians
65      :param heights: array of altitudes, in km
66      :return: tuple of two arrays: xs and ys, in kms, both with same size as thetas
67      """
68      return np.multiply(np.cos(thetas), R_EARTH + heights), np.multiply(np.sin(
                                                    thetas), R_EARTH + heights)
69
70
71  def theta_to_x_y(theta, e, a):
72      """
73          Calculate x,y position for a given theta in a given orbit (e and a)
74      :param theta: true anomaly, in radians
75      :param e: eccentricity
76      :param a: semi-major axis, in km
77      :return: tuple of x,y, in km
78      """
79      return theta_h_to_x_y(theta, thetas_to_alts(theta, e, a))
80
81
82  def theta_to_eccentric_anomaly(theta, e):
83      """
84          Calculate the eccentric anomaly from true anomaly
85              Uses the formula:
86              tan(E/2) = sqrt((1-e)/(1+e))*tan(theta/2)
87      :param theta: true anomaly, in radians
88      :param e: eccentricity
89      :return: eccentric anomaly, in radians
90      """
91      return 2 * np.arctan(np.sqrt((1 - e) / (1 + e)) * np.tan(theta / 2.0))
92
93
94  def eccentric_anomaly_to_delta_t(E, e, a):
95      """
96          Calculate (t-t_0) for a given eccentric anomaly, for a given orbit (e and a
                                                        )
97              Uses formula:
98              E - e.sin(E) = n.(t-t_0)
99      :param E: eccentric anomaly, in radians
100     :param e: eccentricity
101     :param a: semi-major axis, in km
102     :return: t-t0, in seconds
103     """
104     return (E - e * np.sin(E)) / np.sqrt(G_EARTH / np.power(a, 3))
105
106
107 def delta_theta_to_delta_t(theta, e, a):
108     """
109         Calculates time difference between satellite position theta and apogee, for
                                                    a given orbit (e and a)
110             = (t-t0) of theta
111     :param theta: true anomaly in radians
112     :param e: eccentricity
113     :param a: semi-major axis, in km
114     :return: time difference between passage at theta1 and theta2, in seconds
115     """
116     return eccentric_anomaly_to_delta_t(theta_to_eccentric_anomaly(theta, e), e, a)
```