MSc in Aerospace Engineering

2MAE004: Mission Analysis & Orbital Mechanics

Mission Analysis Assignment 2021

Author: Yannick Roelvink

Professors: Stéphanie Lizy-Destrez & Joan Pau Sánchez

October 20, 2021

Introduction: The Two-Body Problem

Recall the definition of the equations of motion in a two-body problem between the Earth and a spacecraft:

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

Where $\boldsymbol{r}$ the position-vector of the satellite, r the magnitude of $\boldsymbol{r}$ and $\mu_E = 3.986 \cdot 10^5$ kg^3 s^{-2} the gravitational parameter of the Earth is.

In the set of equations defined in 1, the following assumptions are made:

- The mass of the smaller body is negligible compared to the central body
- The coordinate system is inertial
- The two bodies, Earth and satellite, are spherically symmetric with uniform density, enabling us to treat the bodies as point masses
- No other forces act on the system, apart from the gravitational forces along the line joining the centres of mass

Furthermore, we recall the following definition of (some of) the orbital parameters:

$$\boldsymbol{h} = \boldsymbol{r} \wedge \boldsymbol{v} \Longrightarrow h = vr = r^2\dot{\theta} \tag{2}$$

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

$$r = \frac{p}{1 + e\cos\theta} \Longrightarrow \begin{cases} r_p = r(\theta = 0) & = \frac{p}{1+e} \\ r_a = r(\theta = \pi) & = \frac{p}{1-e} \end{cases} \tag{4}$$

$$v = \sqrt{\mu\left(\frac{2}{r} - \frac{1}{a}\right)} \tag{5}$$

$$\tau = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{6}$$

Where h the magnitude of the specific angular momentum, θ the true anomaly, p the semilatus rectum, a the semi-major axis, e the eccentricity, r_p & r_a (respectively) the periapsis and apoapsis radius, v the orbital velocity and τ the orbital period is.

1 Solving Kepler's Equation

Defining Kepler's Equation

As can be found in the course material, Kepler's equation is defined as:

$$E - e\sin E = M - n\,(t - t_0) \tag{7}$$

Where E the eccentric anomaly, t the time, t_0 the time of periapsis passage and M the mean anomaly is.

In addition, equation 7 features the mean motion n, which is defined as:

$$n = \sqrt{\frac{\mu}{a^3}} = \frac{2\pi}{\tau} \tag{8}$$

From the value of the eccentric anomaly, The Cartesian coordinates (x, y) of the satellite can be obtained using the following set of equations:

$$\begin{cases} x &= a(\cos E - e) \\ y &= b\sin E \end{cases} \tag{9}$$

Where E the eccentric anomaly, a the semi-major axis, e the eccentricity and b the semi-minor axis is. The semi-minor axis can be computed using the following formula:

$$b = a\sqrt{1 - e^2} \tag{10}$$

Where a the semi-major axis and e the eccentricity of the orbit is.

Lastly, the true anomaly (θ) can be linked with the eccentric anomaly as follows:

$$\tan\frac{E}{2} = \sqrt{\frac{1-e}{1+e}}\tan\frac{\theta}{2} \tag{11}$$

The Newton-Raphson Method

Equations in the form of $f(x) = 0$ can be solved using an iterative approximation method named 'The Newton-Raphson Method'. By using an initial guess x_1, this method generates a sequence of converging solutions as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{12}$$

This sequence will continue to iterate until the value of the function $f(x)$ reaches a predefined tolerance around zero.

Question 1

1A

First, a function that can solve Kepler's equation (Equation 7) using the Newton-Raphson Method has to be created. The Python code used for these computations is provided below:

Algorithm 1: Definition of the Kepler Equation and its numerical solver

```
import numpy as np

def Kepler(E, M, e):
    # Function used to define the Kepler Equation
    return E-e*np.sin(E) - M

def KeplerPrime(E, e):
    # Function used to define the first derivative of the Kepler Equation
    return 1-e*np.cos(E)

def SolveKepler(M, e, a, tol, printing = True):
    # Use M as first guess for E
    Ei = M

    # Iterate over Newton-Raphson Method & update the Eccentric anomaly
    # until the pre-defined tolerance is met
    i = 0
    while np.abs(Kepler(Ei, M, e)) > tol:
        Er = Ei - Kepler(Ei, M, e)/KeplerPrime(Ei,e)
        if printing:
            print(f'After {i+1} iteration E = {Er:.5f} rad and the error = {Kepler(Er, M, e)}')
        Ei = Er
        i+=1

    if printing:
        print(f'The number of iterations needed to converge = {i} and the final value of E = {E:.5f} rad')

    # Compute true anomaly and the radius from the acquired Eccentric Anomaly
    theta = np.arccos((np.cos(Ei) - e) / (1-e*np.cos(Ei)))
    r = a*(1-e*np.cos(Ei))
    return [Ei, theta, r]

#define constants used in the entire report
Re = 6378               # Earth's radius [km]
mu_e = 3.986e5          # Earth's gravitational parameter [km^3/s^2]
```

1B

Using an initial value for the mean anomaly of $M = 21$ °, a tolerance of 10^{-6} and an eccentricity of $e = 0.25$, the eccentric anomaly after one iteration is determined to be $E = 0.48339$ rad. In addition, the tolerance error value is reached within 2 computation iterations.

Changing the tolerance to 10^{-12} and implementing a semi-major axis of $a = 240000$ km, re-running the solver gives rise to an eccentric anomaly of $E = 0.48252$ rad. Using Equations 3, 4 and 11, the true anomaly (θ) and the radius (r) of the spacecraft can be deduced to be 0.62 rad and 18685.04 km, respectively.

Lastly, changing the initial value for the mean anomaly to $M = 180$ °, the computation converges immediately and gives values for the true anomaly and the radius equal to 3.14 rad and

30000.00 km, respectively. Note that the true anomaly in this case is equal to π, and as the true anomaly is defined as the angle between a spacecraft's position and its perigee, this implies that the spacecraft is in its apogee for $M = 180\ °$. Therefore, the value for r is in this case equal to the apogee radius of the spacecraft's orbit.

The code used for this question (together with the code listed in Algorithm 1) is listed below:

Algorithm 2: Implementation of the solver for question 1B

```
def Deg2Rad(x):
    # Function used to transform degrees into radians
    return (x/180) * np.pi

# Define Variables
e = 0.25                  # Eccentricity
a = 24000                 # Semi-major axis [km]
M_deg = 180               # Mean anomaly [degrees]
tol = 10e-12              # Solver's error tolerance

# Solve the Kepler equation
E, theta, r = SolveKepler(Deg2Rad(M_deg), e, a, tol)

print(f"For M = {M_deg} degree and e = {e}, theta & r are: {theta:.2f} rad and {r:.2f} km")
```

1C

Repeating the code defined in Algorithms 1 and 2 results in the following plots of the spacecraft's altitude over time:

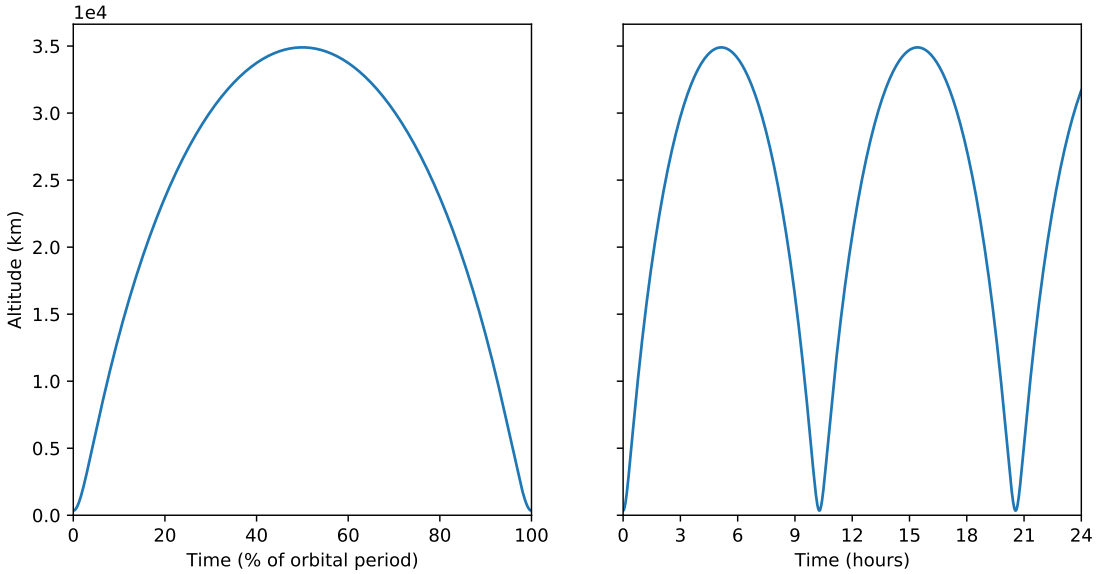

Figure 1: The altitude of the spacecraft, plotted during: (left) a single orbit and (right) a single day (starting in the perigee)

The code used to compute the plots in Figure 1 is provided below:

Algorithm 3: Implementation of the solver for question 1B

```
import matplotlib.pyplot as plt

# Define Variables
e = 0.72                  # Eccentricity
a = 24000                 # Semi-major axis [km]
tol = 1e-12               # Solver's error tolerance
stepsize = 15             # Seconds
max_time = 86400          # Seconds (86400 s = 1 day)

# Compute Parameters
n = np.sqrt(mu_e/a**3)    # Mean Motion
tau = 2*np.pi / n         # Period of Elliptical Curve

# Define time array with a stepsize of 15 seconds for a single day
t = np.linspace(0, max_time, int(max_time / stepsize)+1)

# Compute the mean anomaly from the time array
M_MEO = n*t

# Compute the index after which the orbit will be repeated
max_orbit_index = int(tau/stepsize)

# Initialize solution arrays (radius, eccentric anomaly and altitude)
r_MEO = np.empty(np.size(M_MEO))
E_MEO = np.copy(r_MEO)
h_MEO = np.copy(r_MEO)

# Compute altitudes from the eccentric anomalies
for i, M in enumerate(M_MEO):
    E_MEO[i], _ ,r_MEO[i] = SolveKepler(M, e, a, tol, False)
    h_MEO[i] = r_MEO[i] - Re

# Plotting the altitude for a single orbit
fig, (ax1, ax2) = plt.subplots(1,2,sharey = True, figsize=(10,5))
ax1.plot(t[:max_orbit_index]/tau*100, h_MEO[:max_orbit_index])
ax1.set_xlabel("Time (% of orbital period)")
ax1.set_xlim(0,100)
ax1.set_ylabel("Altitude (km)")
ax1.set_ylim(0)
ax1.ticklabel_format(axis = 'y', style = 'sci', scilimits = (0,0))

# Plotting the altitude for a single day
ax2.plot(t/3600, h_MEO)
ax2.set_xlabel("Time (hours)")
ax2.set_xticks(np.arange(0,27,step = 3))
ax2.set_xlim(0,24)

plt.show()
```

In order to force the plots in Figure 1 to start at the apogee, only the definition of the mean eccentricity has to be altered. Rather than defining it as `M_MEO = n*t`, the mean anomaly is shifted by half a period: `M_MEO = n*(t-tau/2)`. The plots obtained using this shift are plotted below

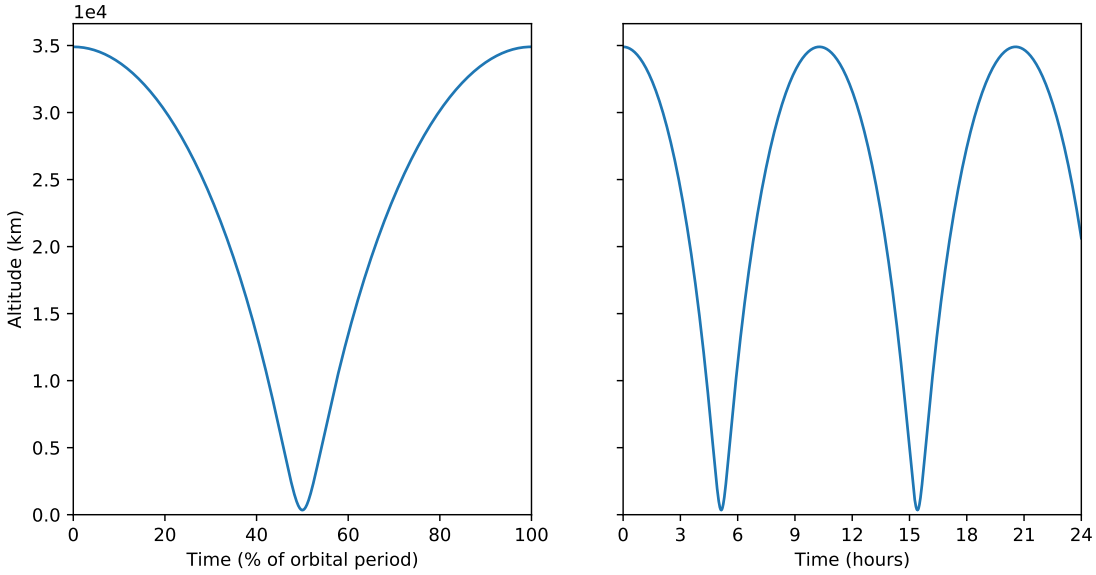

Figure 2: The altitude of the spacecraft, plotted during: (left) a single orbit and (right) a single day (starting in the apogee)

1D

Estimating the time spent within the eclipse caused by the Earth's shadow is crucial for mission design. Therefore, this question will provide a way to compute the time spent in these eclipses. Two cases of eclipses will be considered (around periapsis or apoapsis), as visualised in Figure 3:

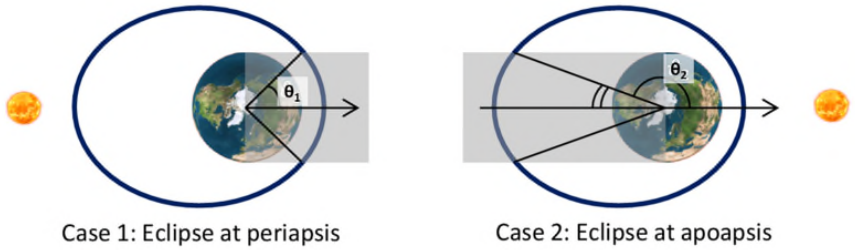

Figure 3: The two eclipses considered in this question: Either the eclipse occurs around the periapsis (case 1) or around the apoapsis (case 2).

Therefore, the true anomalies at which the satellite enters and exits these eclipses have to be computed, which can be done using the following equation:

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

Where θ the true anomaly of entry or exit of the eclipsed region is. The values of θ within Equation 13 can be found using the quadratic formula, as defined as:

$$ax^2 + bx + c = 0 \Longrightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{14}$$

In equation 13, several substitution variables have been implemented (α, β and γ). These variables are defined as follows:

$$\alpha = R_e^2 e^2 + p^2, \quad \beta = 2R_e^2 e, \quad \gamma = R_e^2 - p^2 \tag{15}$$

Where R_e the Earth's radius, e the satellite's eccentricity and p the semilatus rectum (as defined in Equation 3) is. Figure 4 visualises the entry and exit points (obtained from solving Equation 13) of both the eclipses of the orbit from the previous questions:

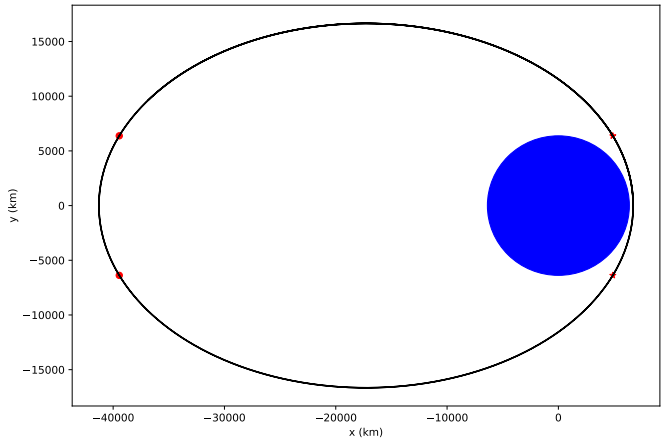

Figure 4: Visualisation of the eclipses of the orbit computed in the previous questions. The Earth has been depicted as a blue circle, whereas the points of the eclipse entry and exit are depicted with the * (case 1) and ° (case 2) markers.

The time spent in an orbital section can be computed using the true anomalies of two points defining said section. These true anomalies are first converted into their representative eccentric anomalies, using equation 11. Afterwards, the eccentric anomalies are converted into a time difference using the Kepler equation (Equation 7), by varying the mean anomaly M (as seen in the function `Calc_Delta_T` at the top of Algorithm 4). The resulting eclipse times are 23.02 and 485.44 minutes for case 1 and case 2 of Figure 4, respectively.

The code used for the Eclipse computations of 4 is provided below:

Algorithm 4: Eclipse location and duration implementation

```
def comp_cartesian(E, a, b):
    # Function that computes the Cartesian coordinates from the orbital parameters
    x = a*(np.cos(E)-e)
    y = b*np.sin(E)
    return (x, y)

def Calc_Delta_T(E1, E2, e, n):
    # Function to compute the time difference between two orbital points
    M1 = E1 - e*np.sin(E1)
    M2 = E2 - e*np.sin(E2)
    dT = np.abs(M2-M1)/n
    return dT

# Define variables
b = a*np.sqrt(1-e**2)       # Semi-minor axis [km]
```

```
16
17 # Initialize arrays
18 x_MEO = np.copy(r_MEO)
19 y_MEO = np.copy(r_MEO)
20
21 # Compute Cartesian coordinates of the satellite
22 for i, E in enumerate(E_MEO):
23     x_MEO[i], y_MEO[i] = comp_cartesian(E, a, b)
24
25 # Compute variables used in Eclipse equations
26 p = a*(1-e**2)
27 alpha = (Re*e)**2 + p**2
28 beta = 2*(Re**2)*e
29 gamma = (Re**2)-(p**2)
30
31 # Solve Eclipse equations for true anomalies & associated eccentric anomalies
32 Delta = np.sqrt(beta**2 - 4 * alpha * gamma)
33 theta_eclipse_1 = np.arccos((-beta + Delta) / (2*alpha))
34 theta_eclipse_2 = np.arccos((-beta - Delta) / (2*alpha))
35 E_eclipse_1 = 2*np.arctan( np.sqrt((1-e)/(1+e)) * np.tan(theta_eclipse_1/2))
36 E_eclipse_2 = 2*np.arctan( np.sqrt((1-e)/(1+e)) * np.tan(theta_eclipse_2/2))
37
38 # Compute cartesian coordinates of the eclipse points
39 x_eclipse_1, y_eclipse_1 = comp_cartesian(E_eclipse_1, a, b)
40 x_eclipse_2 , y_eclipse_2  = comp_cartesian(E_eclipse_2, a, b)
41
42 # Plot Earth as a blue circle and radius Re
43 Earth = plt.Circle((0, 0), Re, color='blue')
44 fig = plt.figure(figsize = (10,7))
45 ax = fig.gca()
46 ax.add_patch(Earth)
47 ax.plot(x_MEO, y_MEO, color = 'k')
48 ax.set_xlabel("x (km)")
49 ax.set_ylabel("y (km)")
50
51 # Plot Eclipse points
52 ax.scatter(x_eclipse_1, y_eclipse_1, marker = "*", color = "r")
53 ax.scatter(x_eclipse_1, -y_eclipse_1, marker = "*", color = "r")
54 ax.scatter(x_eclipse_2, y_eclipse_2, marker = "o", color = "r")
55 ax.scatter(x_eclipse_2, -y_eclipse_2, marker = "o", color = "r")
56
57 # Compute Eclipse times and plot output
58 Eclipse_time_1 = Calc_Delta_T(-E_eclipse_1, E_eclipse_1, e, n)
59 print(f"Duration stayed in Eclipse 1 = {(Eclipse_time_1/60):.2f} min")
60 Eclipse_time_2 = Calc_Delta_T(E_eclipse_2, -E_eclipse_2, e, n)
61 print(f"Duration stayed in Eclipse 2 = {(Eclipse_time_2/60):.2f} min")
```

2 Numerical Integration of the Equations of Motion Using Differential Equation Solvers

In this section, the two body problem (as defined as in Equation 1) will be numerically solved using Python's `scipy.integrate.solve_ivp` function, which can be used to solve initial value ODE problems [1].

[1] See documentation for this function at: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`

Question 2

2A

First, the two body problem of Equation 1 has to be converted into vector form, which results in the following set of differential equations:

$$\boldsymbol{X} = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \quad \text{and} \quad \dot{\boldsymbol{X}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ -x\mu_e/r^3 \\ -y\mu_e/r^3 \\ -z\mu_e/r^3 \end{bmatrix} \tag{16}$$

Where $r = \sqrt{x^2 + y^2 + z^2}$ the radius of the satellite and μ_e the Earth's gravitational parameter is.

2B

Now that the two body problem vector has been defined, a function to solve said vector has to be created. In the code below, the function `twobody` defines the differential equations of Equation 16, whereas the function `solve_twobody` solves the two body problem

Algorithm 5: Two body solver function definitions

```
from scipy.integrate import solve_ivp

def twobody(t,X):
    x = X[0]
    y = X[1]
    z = X[2]
    r = np.sqrt(x**2 + y**2 + z**2)

    xdot = X[3]
    ydot = X[4]
    zdot = X[5]

    return [xdot, ydot, zdot, -mu_e/(r**3)*x, -mu_e/(r**3)*y, -mu_e/(r**3)*z]

def solve_twobody(X0, t, Reltol = 1e-12, Abstol = 1e-12):

    # Solve ODE
    sol = solve_ivp(twobody, (0,max(t)), X0, t_eval = t, rtol = Reltol, atol = Abstol)

    # Compute Positional solutions
    sol_x = sol.y[0,:]
    sol_y = sol.y[1,:]
    sol_z = sol.y[2,:]
    sol_r = np.sqrt(sol_x**2 + sol_y**2 + sol_z**2)

    # Compute Velocity solutions
    sol_vx = sol.y[3,:]
    sol_vy = sol.y[4,:]
    sol_vz = sol.y[5,:]
    sol_v = np.sqrt(sol_vx**2 + sol_vy**2 + sol_vz**2)

    return [sol_x, sol_y, sol_z, sol_r, sol_vx, sol_vy, sol_vz, sol_v]
```

2C

Implementing an initial position and velocity vector of:

$$\boldsymbol{X_0} = [\boldsymbol{r}\ \boldsymbol{v}]^T \text{ with } \begin{cases} \boldsymbol{r} &= [7115.804; 3391.696; 3492.221]\ km \\ \boldsymbol{v} &= [-3.762; 4.063; 4.184]\ km/s \end{cases}$$

The following plots of the magnitude of the position and velocity vector were obtained:

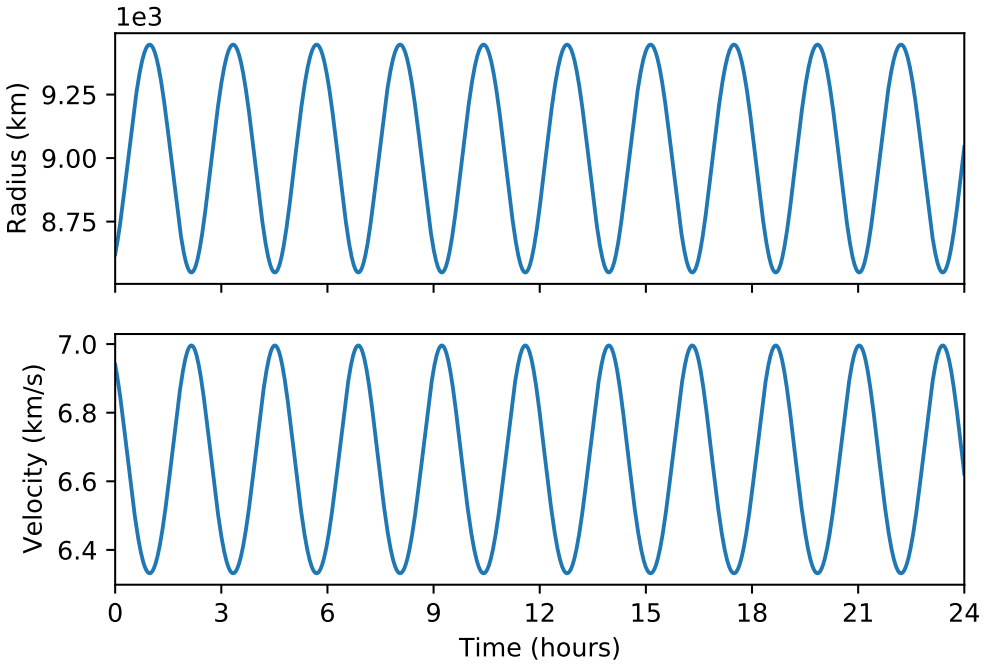

Figure 5: Plot of the magnitude of the position vector $\boldsymbol{r}$ (top) and velocity vector $\boldsymbol{v}$ (bottom) during a full day.

In addition, the following 3D plot of the satellite's orbit has been obtained from the ODE solver:

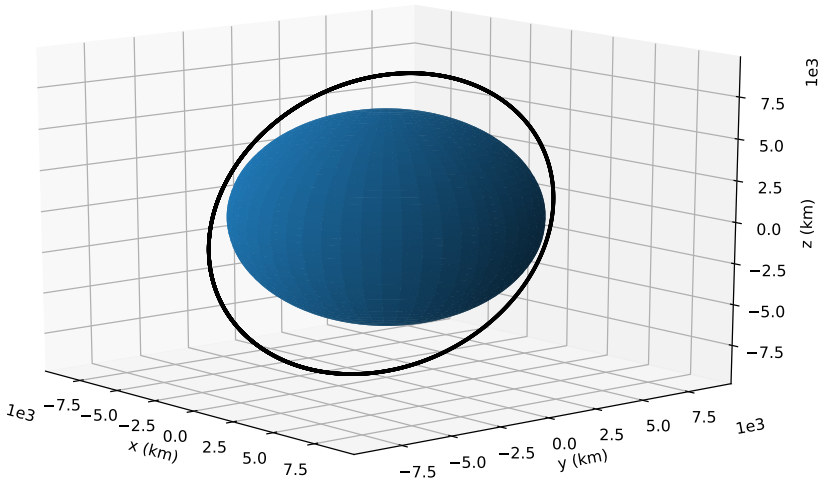

Figure 6: Orbit obtained from the ODE solver, using the data used in Figure 5. The Earth has been added to the plot (as the blue sphere) for a sense of scale.

The code used to generate Figures 5 and 6 can be found below:

Algorithm 6: Two body solver implementation

```
# define variables
r0 = [7115.804, 3391.696, 3492.221]          # km
v0 = [−3.762, 4.063, 4.184]                  # km/s
Reltol = 1e−12                               # Solver's Relative error tolerance
Abstol = 1e−12                               # Solver's Absolute error tolerance
stepsize = 10                                # Seconds
max_time = 86400                             # Seconds (86400 = 1 day)
plt_resolution = 200                         # 3D plotting resolution

# Initialize time and initial position vector
t = np.linspace(0, max_time, int(max_time / stepsize)+1)
X0 = np.concatenate((r0, v0), axis = None)

# Solve ODE
sol_x, sol_y, sol_z, sol_r, _, _, _, sol_v = solve_twobody(X0, t, Reltol, Abstol)

# Plot position magnitude over time
fig, (ax1, ax2) = plt.subplots(2, 1, sharex = True)
ax1.plot(t/3600, sol_r)
ax1.set_ylabel("Radius (km)")
ax1.set_xlim(0,24)
ax1.ticklabel_format(axis = 'y', style = 'sci', scilimits = (0,0))
ax1.set_xticks(np.arange(0,27,step = 3))

# Plot the velocity magnitude over time
ax2.plot(t/3600, sol_v)
ax2.set_ylabel("Velocity (km/s)")
ax2.set_xlabel("Time (hours)")
plt.show()

# Plot the orbit in a 3D plane
fig2 = plt.figure(figsize = (10,7))
ax3 = fig2.add_subplot(111, projection='3d')
ax3.ticklabel_format(axis = 'both', style = 'sci', scilimits = (0,0))
ax3.set_xlabel("x (km)")
ax3.set_ylabel("y (km)")
ax3.set_zlabel("z (km)")
ax3.scatter(sol_x, sol_y, zs = sol_z, color = 'k', s=2)

# Add the Earth to the plot
theta_Earth = np.linspace(0,2*np.pi,plt_resolution)
phi_Earth = np.linspace(0, np.pi, plt_resolution)
x_Earth = Re*np.outer(np.cos(theta_Earth), np.sin(phi_Earth))
y_Earth = Re*np.outer(np.sin(theta_Earth), np.sin(phi_Earth))
z_Earth = Re*np.outer(np.ones(np.size(theta_Earth)), np.cos(phi_Earth))
ax3.plot_surface(x_Earth, y_Earth, z_Earth)
plt.show()
```

2D

Besides the position and velocity of the orbit, the specific energies and angular momentum can be computed and plotted. For the case of the specific energies, the following definition is used:

$$E = E_k + E_p = \frac{v^2}{2} - \frac{\mu_e}{r} \tag{17}$$

Where E_k the specific kinetic energy, E_p the specific potential energy, v the orbital speed and r the orbital radius is. In addition, the specific angular momentum can be computed using Equation 2.Computing the values of the specific energy and angular momentum using Equations 17 and 2 using the data of the ODE solver of question 2C gives rise to the following plots:

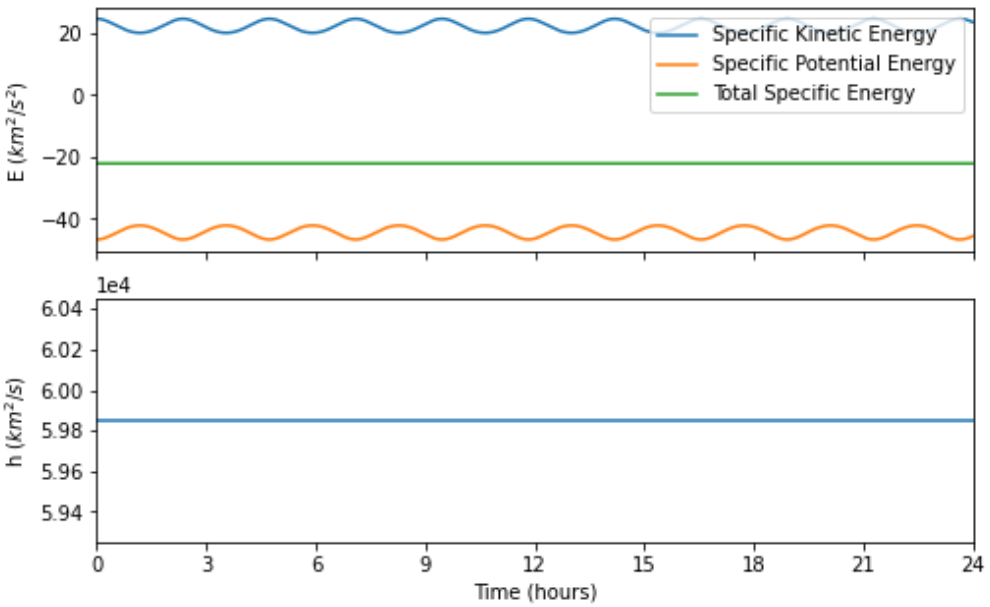

Figure 7: Plot of the specific energy E (top), together with its kinetic and potential components, and specific angular momentum (bottom) of the data used for Figure 5.

Note that, in the upper plot of Figure 7, the specific kinetic energy is strictly positive, while the specific potential and total specific energy are strictly negative. This is to be expected, as a negative total specific energy entails that the satellite is in a bounded state, i.e. it will not escape the Earth's gravitational pull. In addition, the total specific energy should be constant, as per the law of conservation of energy, and can be computed (for an ellipse) to be equal to $E = -\frac{\mu_e}{2a} \approx -22$ km^2/s^2, which perfectly corresponds to the value depicted in Figure 7. In the case of the angular momentum (lower plot in Figure 7), one can observe the conservation of angular momentum, as the value of h remains constant over time.

The code used for the specific energy and angular momentum computations is provided below:

Algorithm 7: Specific energy and angular momentum computations

```
# Compute Specific Energies
Ek = sol_v**2 / 2          # Specific Kinetic Energy [km^2 / s^2]
Ep = -mu_e/sol_r           # Specific Potential Energy [km^2 / s^2]
Etot = Ek + Ep             # Total Specific Energy [km^2 / s^2]

# Compute Specific Angular Momentum
r_vec = [sol_x, sol_y, sol_z]
v_vec = [sol_vx, sol_vy, sol_vz]
h = np.linalg.norm(np.cross(r_vec, v_vec, axis = 0), axis = 0)

# Plot Specific Energies
fig, (ax1, ax2) = plt.subplots(2,1,sharex = True, figsize = (8,5))
ax1.plot(t/3600, Ek, label = 'Specific Kinetic Energy')
ax1.plot(t/3600, Ep, label = 'Specific Potential Energy')
ax1.plot(t/3600, Etot, label = 'Total Specific Energy')
ax1.legend()
ax1.set_ylabel("E ($km^2/s^2$)")

ax2.plot(t/3600, h)
ax2.ticklabel_format(axis = 'y', style = 'sci', scilimits = (0,0))
ax2.set_xlabel("Time (hours)")
ax2.set_ylabel("h ($km^2/s$)")
ax2.set_xlim(0,24)
ax2.set_xticks(np.arange(0,27,step = 3))
plt.show()
```

2E

Changing the initial position and velocity vectors into:

$$\boldsymbol{X_0} = [\boldsymbol{r}\ \ \boldsymbol{v}]^T \text{ with } \begin{cases} \boldsymbol{r} & = [0; 0; 8550]\ \ km \\ \boldsymbol{v} & = [0; -7.0; 0]\ \ km/s \end{cases}$$

In order to determine the type of orbit of the satellite, the orbital parameters of said orbit have to be computed. Using the ODE solver from question 2A and implementing its results into Equations 2 through 6, the following orbital parameters can be computed:

$$\begin{cases} a & = 9009.99\ \ km \\ e & = 0.05 \\ \tau & = 8511.34\ \ s \\ i & = 90.0^\circ \end{cases}$$

In addition, the following plots regarding the orbits position and velocity are found:

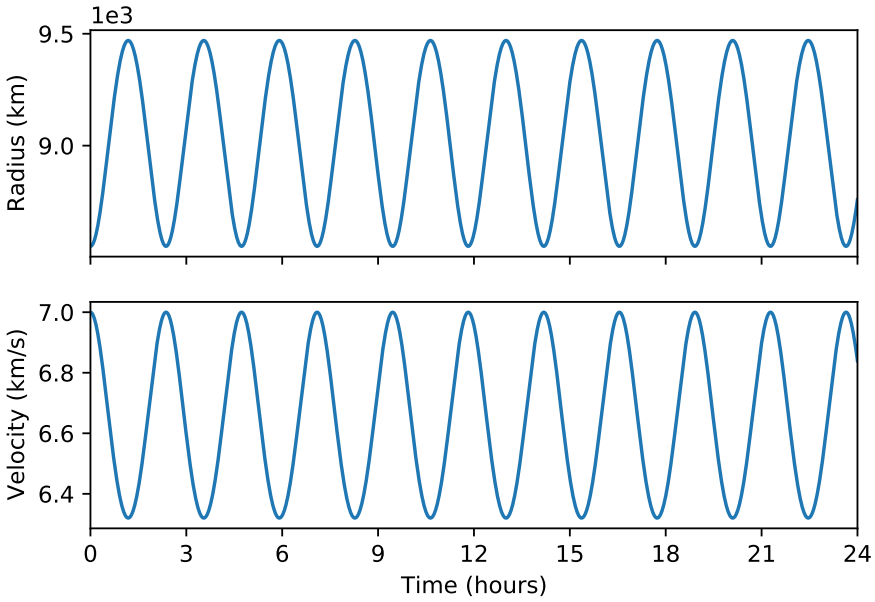

Figure 8: Plot of the magnitude of the position vector $\boldsymbol{r}$ (top) and velocity vector $\boldsymbol{v}$ (bottom) during a full day.

Lastly, the following orbital plot can be computed as well:

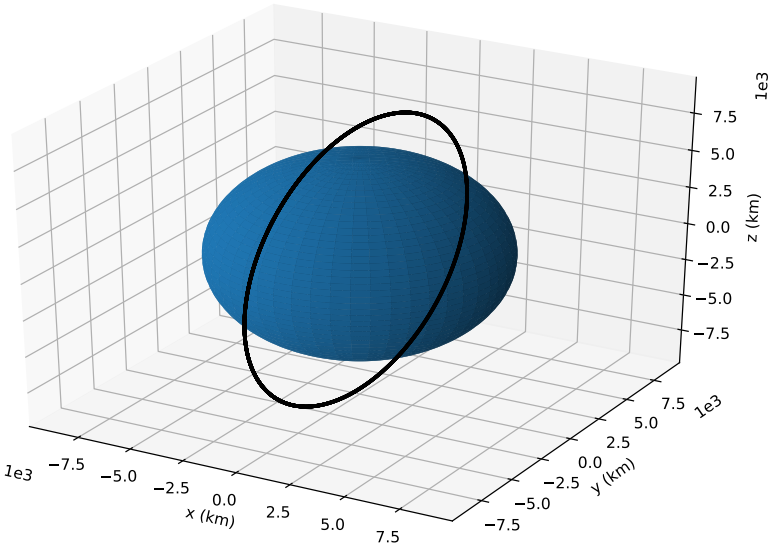

Figure 9: Orbit obtained from the ODE solver, using the data used in Figure 8. The Earth has been added to the plot (as the blue sphere) for a sense of scale.

Following from the computed orbital parameters and the orbital plot in Figure 9, the orbit of the satellite has been determined to be a circular ($e \approx 0$) and polar ($i = 90°$) orbit. The code used for the orbital parameter computations is displayed below:

Algorithm 8: Code used for orbit determination

```
def Rad2Deg(x):
    # Function used to transform radians into degrees
    return x*(180/np.pi)

# define variables
r0 = [0, 0, 8550]                    # km
v0 = [0, -7.0, 0]                    # km/s
Reltol = 1e-12                       # Solver's Relative error tolerance
Abstol = 1e-12                       # Solver's Absolute error tolerance
stepsize = 10                        # Seconds
max_time = 86400                     # Seconds (86400 = 1 day)
plt_resolution = 200                 # 3D plotting resolution

# Compute parameters
t = np.linspace(0, max_time, int(max_time / stepsize)+1)
X0 = np.concatenate((r0, v0), axis = None)

sol_x, sol_y, sol_z, sol_r, sol_vx, sol_vy, sol_vz, sol_v = solve_twobody(X0, t, Reltol, Abstol)

# Plot position magnitude over time
fig, (ax1, ax2) = plt.subplots(2, 1, sharex = True)
ax1.plot(t/3600, sol_r)
ax1.set_ylabel("Radius (km)")
```

```
ax1.set_xlim(0,24)
ax1.ticklabel_format(axis = 'y', style = 'sci', scilimits = (0,0))
ax1.set_xticks(np.arange(0,27,step = 3))

# Plot the velocity magnitude over time
ax2.plot(t/3600, sol_v)
ax2.set_ylabel("Velocity (km/s)")
ax2.set_xlabel("Time (hours)")
plt.show()

# Plot the orbit in a 3D plane
fig2 = plt.figure(figsize = (10,7))
ax3 = fig2.add_subplot(111, projection='3d')
ax3.ticklabel_format(axis = 'both', style = 'sci', scilimits = (0,0))
ax3.set_xlabel("x (km)")
ax3.set_ylabel("y (km)")
ax3.set_zlabel("z (km)")
ax3.set_xlim(-max(sol_r), max(sol_r))
ax3.set_ylim(-max(sol_r), max(sol_r))
ax3.set_zlim(-max(sol_r), max(sol_r))
ax3.scatter(sol_x, sol_y, zs = sol_z, color = 'k', s=2)

# Add the Earth to the plot (using the definition from before)
ax3.plot_surface(x_Earth, y_Earth, z_Earth)
plt.show()

# Compute Orbital Parameters
r_a = max(sol_r)            # Periapsis radius [km]
r_p = min(sol_r)            # Apoapsis radius [km]
a = (r_p + r_a)/2           # semi-major axis [km]
print(f"The semi-major axis of the satellite = {a:.2f} km")

e = r_a/a - 1               # Eccentricity
print(f"The eccentricity of the satellite = {e:.2f}")

n = np.sqrt(mu_e/a**3)      # Mean Motion
tau = 2*np.pi / n           # Period of Elliptical Curve [s]
print(f"The period of the satellite = {tau:.2f} s")

h_vec = np.cross(r0, v0)
normalized_h_vec = h_vec / np.sqrt(np.sum(h_vec**2))
z_axis = [0,0,1]
i = np.arccos(np.dot(normalized_h_vec, z_axis))
print(f"The inclination of the satellite = {Rad2Deg(i)} degrees")
```

3 Orbit Phasing and Rendezvous

In the last section of this report, the orbital parameters needed for a supplier spacecraft (from now on called the *chaser*) to catch up and dock with the ISS, before de-orbiting back into the Earth's atmosphere.

For simplicity, the orbit of the ISS will be assumed to be an equatorial & circular orbit at an altitude of 404 km, whereas the orbit of the chaser will be assumed to be an equatorial & elliptical orbit, with an eccentricity in such a way that it will intercept the ISS at its apogee. In addition, at $t = 0$, the ISS is $\Delta\Theta$ radians ahead of the chaser. A sketch of the situation is provided in Figure 10:

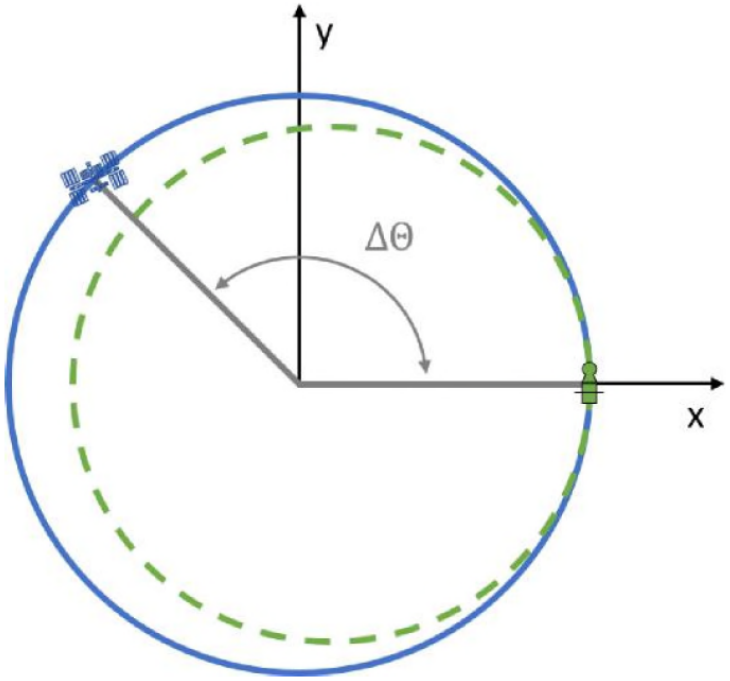

Figure 10: Sketch positioning of the ISS (blue) and chaser (green) at t = 0. The ISS starts off $\Delta\Theta$ radians ahead of the chaser.

The semi-major axis of the catcher is selected in such a way that it catches up with the ISS in N_{rev} orbits. In mathematical terms, this means that the following equation holds:

$$N_{rev} T_{chaser} \left(n_{chaser} - n_{ISS}\right) = \Delta\Theta \tag{18}$$

Where T_{chaser} the orbital period of the chaser, n the mean motion of either the chaser or the ISS and $\Delta\Theta$ the initial angular separation between the chaser and the ISS is.

By substituting the expressions for the mean motion (Equation 8) and orbital period (Equation 6) into Equation 18, it can be found that:

$$a_{chaser} = a_{ISS} \left(1 - \frac{\Delta\Theta}{2\pi N_{rev}}\right)^{2/3} \tag{19}$$

Where a the semi-major axis of either the chaser or the ISS is.

Lastly, by introducing the fact that the apogee radius of the chaser is equal to the radius of the orbit of the ISS, the following equation can be used to compute the required eccentricity of the chaser's orbit (e_{chaser}):

$$e_{chaser} = \frac{a_{ISS}}{a_{chaser}} - 1 \tag{20}$$

Question 3

3A & **3B**

In order to compute the orbit of both the chaser and the ISS, the initial positions of both have to be implemented into the `twobody_solve` function, as defined in Algorithm 5. Therefore, the first step is to compute the initial positions of both of the spacecrafts.

Starting off with the circular orbit of the ISS, its semi-major axis can be computed by simply adding its altitude of 404 km to the radius of the Earth ($R_e = 6378$ km). Therefore, the initial position vector of the ISS is simply a position on a circle with a radius equal to $404 + 6378 = 6782$ km, rotated counterclockwise with an angle of $\Delta\Theta$ (as seen in Figure 10). This rotation (around the z-axis) is performed using a special formula called `Zrotation` in Algorithm 9. For the case of the velocity vector, the following formula for circular orbital velocities can be used:

$$v_{circular} = \sqrt{\frac{\mu_e}{a}} \tag{21}$$

Where μ_e the Earth's gravitational constant and a the satellite's semi-major axis is.

Similar to the position vector, the velocity vector of the ISS is simply the magnitude obtained using Equation 21 propagated in the positive y direction (upwards in Figure 10), rotated counterclockwise with an angle of $\Delta\Theta$. In addition, the period of the ISS can be computed to be 5558.37 s, using Equation 6.

For the chaser, the initial position is simply computed using Equation 19, using the fact that the chaser catches up with the ISS in 12 orbits (i.e. $N_{rev} = 12$). In addition, the notion that the apogee radius of the chaser equals the radius of the orbit of the ISS can be used to determine the chaser's apogee position (i.e. $r_a^{chaser} = r_{ISS} = 6782$ km). Afterwards, the apogee velocity of the chaser can be computed using equation 5, which can then be used to produce the initial velocity vector of the chaser.

Lastly, the initial vectors of both the ISS and the chaser are put in the `twobody_solve` function, as defined in Algorithm 5, which results in the following orbital plots:

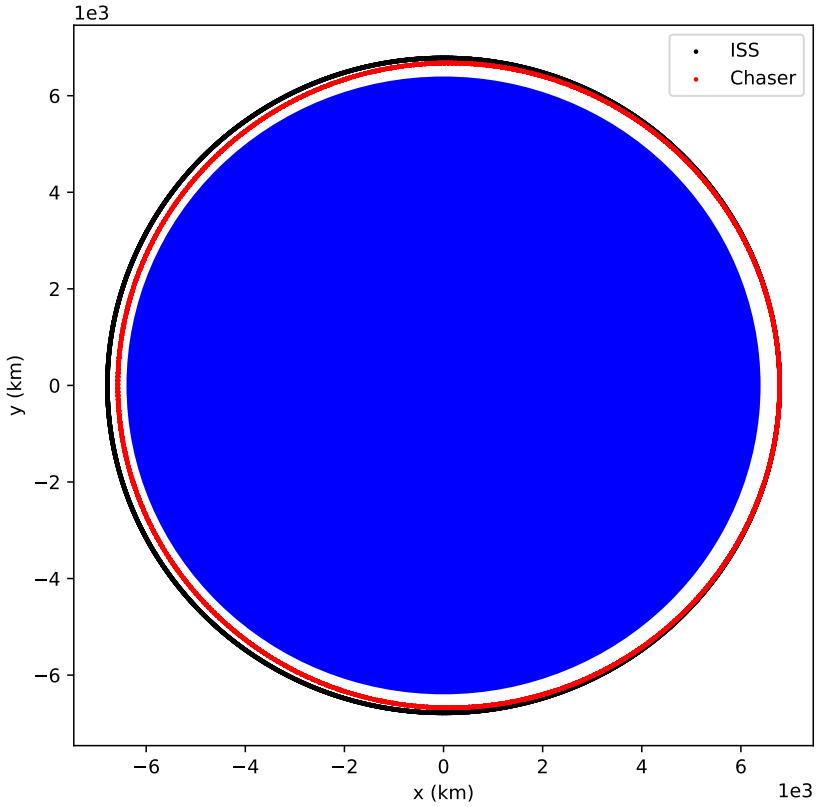

Figure 11: The computed orbits of the ISS (black) and the chaser (red), depicted together with the Earth (blue).

The code used for the orbit computations of both the ISS and the chaser is provided below:

Algorithm 9: Definition of the Kepler Equation and its numerical solver

```
from scipy.spatial.transform import Rotation

def Zrotation(x, phi):
    # Function to rotate a vector x around the z-axis with angle phi
    rotation_rad = Deg2Rad(phi)
    rotation_axis = np.array([0,0,1])
    rotation_vector = rotation_rad*rotation_axis
    rotation = Rotation.from_rotvec(rotation_vector)
    return rotation.apply(x)

def solve_chaser(N, Theta, a_target):
    # Function used to compute the orbital parameters of the chaser's orbit
    a = a_target*(1-Deg2Rad(Theta)/(2*np.pi*N))**(2/3)
    e = a_target/a - 1
    return [a, e]

# Set variables
Delta_Theta = 100                # Initial Angular Separation [degree]
alt_ISS = 404                    # altitude of the ISS [km]
Nrev = 12                        # number of orbits needed for the chaser to intercept in the ISS
stepsize = 10                    # Seconds

```

```
# Compute Parameters ISS
max_time = (Nrev/12)*86400          # Seconds (86400 = 1 day)
a_ISS = Re + alt_ISS                # semi-major axis of the ISS orbit [km]
n_ISS = np.sqrt(mu_e/a_ISS**3)      # Mean Motion
tau_ISS = 2*np.pi / n_ISS           # Period of Elliptical Curve [s]
v_ISS = np.sqrt(mu_e/a_ISS)         # (Circular) Orbital velocity of the ISS [km/s]

print(f"The orbital period of the ISS = {tau_ISS} s")

# Compute Initial vector of the ISS
r0_ISS = Zrotation(np.array([a_ISS, 0, 0]), Delta_Theta)
v0_ISS = Zrotation(np.array([0, v_ISS, 0]), Delta_Theta)
X0_ISS = np.concatenate((r0_ISS, v0_ISS), axis = None)

# Solve ISS two body problem
t_ISS = np.linspace(0, max_time, int(max_time / stepsize)+1)
sol_x_ISS, sol_y_ISS, sol_z_ISS, sol_r_ISS, sol_vx_ISS, sol_vy_ISS, sol_vz_ISS, sol_v_ISS =
    solve_twobody(X0_ISS, t_ISS)

# Plot ISS Orbit
fig = plt.figure(figsize = (7,7))
ax = fig.gca()
ax.ticklabel_format(axis = 'both', style = 'sci', scilimits = (0,0))
ax.set_xlabel("x (km)")
ax.set_ylabel("y (km)")
ax.scatter(sol_x_ISS, sol_y_ISS, color = 'k', s=2, label = "ISS")

# Plot Earth as a blue circle and radius Re
Earth = plt.Circle((0, 0), Re, color='blue')
ax.add_patch(Earth)

# Compute chaser's orbital parameters
a_chaser, e_chaser = solve_chaser(Nrev, Delta_Theta, a_ISS)
print(f"The semi-major axis and the eccentricity of the chaser are {a_chaser:.2f} km and {e_chaser:.2f}")

# Set parameters for the Chaser
ra_chaser = a_ISS                   # Apoapsis radius of the chaser is equal to that of the ISS
r0_chaser = np.array([ra_chaser, 0, 0])
n_chaser = np.sqrt(mu_e/a_chaser**3)    # Mean Motion
tau_chaser = 2*np.pi / n_chaser         # Period of Elliptical Curve [s]
va_chaser = np.sqrt(mu_e*((2/ra_chaser) - 1/a_chaser))  # Orbital Apoapsis velocity of the chaser
v0_chaser = np.array([0,va_chaser,0])

# Set initial position and velocity vector of the chaser
X0_chaser = np.concatenate((r0_chaser, v0_chaser), axis = None)

# Solve Chaser two body problem
sol_x_chaser, sol_y_chaser, sol_z_chaser, sol_r_chaser, sol_vx_chaser, sol_vy_chaser, sol_vz_chaser,
    sol_v_chaser= solve_twobody(X0_chaser, t_ISS)

# Add the Chaser's orbit to the plot
ax.scatter(sol_x_chaser, sol_y_chaser, color = 'r', s=2, label = "Chaser")
plt.legend()
plt.show()
```

From the data of the orbits obtained above, the minimum distance between the chaser and the ISS over time can be computed. Figure 12 visualises the variation of this distance over a single day:

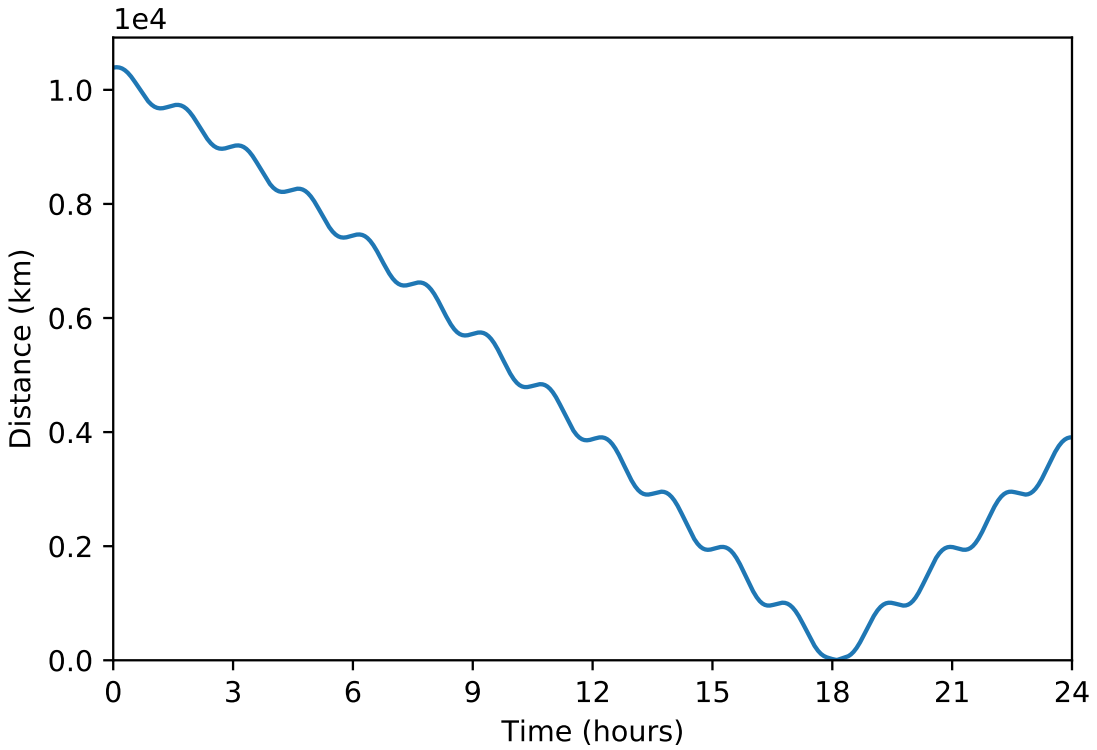

Figure 12: Plot depicting the distance between the chaser and the ISS over time.

Figure 12 clearly shows that the chaser closes in on the ISS with every single orbit, but that it needs 12 orbits before it can minimize the distance. This minimum distance is found to be 211.03 m (for the case of $N_{rev} = 12$).

The code used for the distance computations is provided below:

Algorithm 10: Definition of the Kepler Equation and its numerical solver

```
# Compute the distance between the ISS and the chaser
distance = np.sqrt((sol_x_ISS - sol_x_chaser)**2 + (sol_y_ISS - sol_y_chaser)**2 + (sol_z_ISS -
    sol_z_chaser)**2)

# Plot the distance between the ISS and the chaser
fig2 = plt.figure()
plt.ticklabel_format(axis = 'y', style = 'sci', scilimits = (0,0))
plt.plot(t_ISS/3600, distance)
plt.xlabel("Time (hours)")
plt.ylabel("Distance (km)")
plt.xlim(0,max_time/3600)
plt.xticks(np.arange(0,max_time/3600 + 3,step = 3))
plt.ylim(0)
plt.show()

# Compute the minimum distance obtained
min_distance = min(distance)
print(f'The minimum distance between the chaser and the ISS = {min_distance*1000:.2f} m')
```

3C

Introducing a variation in the number of catch-up orbits from 2 to 30, the effect of the number of these revolutions on the manoeuvre required to circularise the orbit of the chaser can be computed.

First, the computation of question 3B is repeated for all of the number of catch-up orbits between 2 and 30. Then, the minimum distance for each of these values is determined. At this point of minimal distance, the circulation manoeuvre will be performed, so the orbital velocity of the chaser at that point is determined as well. In addition, the velocity of the ISS in that point is determined. Lastly, the difference between the two velocities is computed. The results of these computations are visualised in Figure 13:

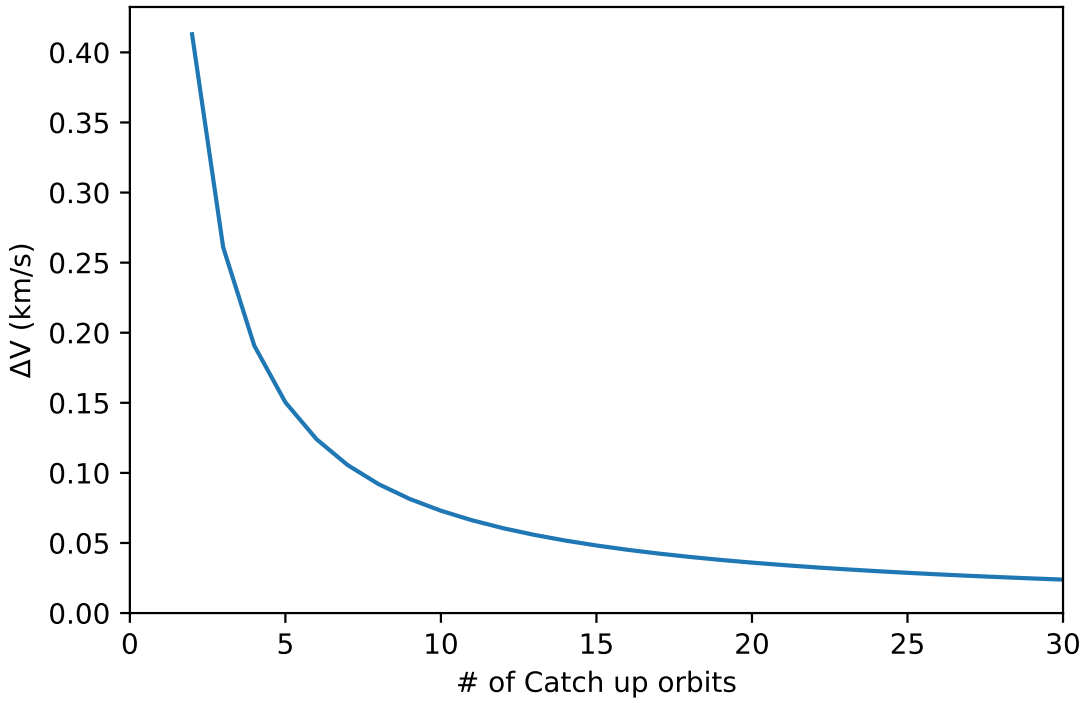

Figure 13: Plot depicting the ΔV required to circularise the chaser's orbit for a range of different catch-up orbits (N_{rev}).

From Figure 13, a distinct inverse relation between the value of N_{rev} and the required Δv can be observed. This is due to the fact that, for a larger number of catch-up orbits, the shape of the chaser much more resembles that of the ISS than for a lower number of catch-up orbits, thus resulting in a lower difference in orbital velocity. An additional effect is that, for larger values of N_{rev}, the minimum distance between the ISS and the chaser reduces as well.

The code required for the ΔV computations is provided below:

Algorithm 11: Delta V computation for circularising the chaser's orbit

```
def ComputeDeltaV(N, Delta_Theta, pos_target, v_target, a_target, t_int, Reltol = 1e-12, Abstol = 1e-12):
    # Function that computes the Delta V needed to circularise the chaser's orbit at the point of minimum
      distance

    # Compute chaser parameters
    a, e = solve_chaser(N, Delta_Theta, a_target)

    # Compute apogee position and velocity
    ra = a_target
    va = np.sqrt(mu_e*((2/ra) - 1/a))

    # Define initial position and velocity vectors
    r0 = np.array([ra, 0, 0])
    v0 = np.array([0,va,0])

    # Define initial value problem & solve the two body problem
    X0 = np.concatenate((r0, v0), axis = None)
    x, y, z, r, vx, vy, vz, v = solve_twobody(X0, t_int, Reltol, Abstol)

    # Compute the minimal distance between the ISS and the chaser
    distance = np.sqrt((pos_target[0] - x)**2 + (pos_target[1] - y)**2 + (pos_target[2] - z)**2)
    min_distance = min(distance)

    # Compute delta V at position of minimum distance
    min_index = np.where(distance == min_distance)
    DeltaV = v_target[min_index] - v[min_index]
    return DeltaV


# Set limits of revolutions needed for the chaser to catch up
Nrev_min = 2
Nrev_max = 30
stepsize = 10     # Seconds

# Define parameter arrays
max_time = (Nrev_max/12)*86400   # Seconds (86400 = 1 day)
t_deltaV = np.linspace(0, max_time, int(max_time / stepsize)+1)
Nrev = np.linspace(Nrev_min, Nrev_max, Nrev_max - Nrev_min + 1)

# Initialise solution array
DeltaV = np.empty(np.size(Nrev))

# Solve two body problem for the ISS velocity & position array
sol_x_ISS, sol_y_ISS, sol_z_ISS, _, _, _, _, sol_v_ISS = solve_twobody(X0_ISS, t_deltaV)
pos_vec_SS = [sol_x_ISS, sol_y_ISS, sol_z_ISS]
v_vec_ISS = sol_v_ISS

# Compute delta V required for circularising the orbit for a given number of catch-up orbits
for i, N in enumerate(Nrev):
    DeltaV[i] = ComputeDeltaV(N, Delta_Theta, pos_vec_SS, v_vec_ISS, a_ISS, t_deltaV)

# Plots solution
fig = plt.figure()
plt.plot(Nrev, DeltaV)
plt.xlabel("# of Catch up orbits")
plt.ylabel("$\Delta$V (km/s)")
plt.xlim(0,Nrev_max)
plt.ylim(0)
plt.show()
```

3D

After docking with the ISS, the chaser spacecraft will have to de-dock and deorbit back into an elliptical orbit and re-enter the atmosphere. In order to compute how much fuel is necessary for this de-orbiting manoeuvre, the Δv of such de-orbiting manoeuvres have to be computed. In the case of the chaser, the Δv for a range of perigee altitudes (ranging from 60 to 210 km above the Earth) have been computed and plotted in Figure 14:

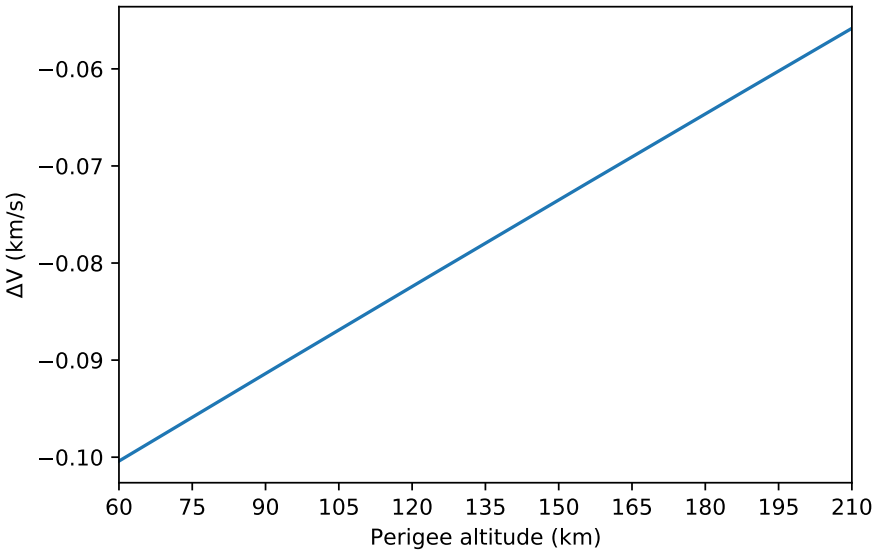

Figure 14: Plot depicting the Δv required to de-orbit the chaser for a given range of perigee altitudes.

Algorithm 12: Delta V computation for de-orbiting the chaser

```
# Define variables
min_perigee = 60 + Re      # km
max_perigee = 210 + Re     # km
stepsize = 1               # km

# Define initial parameters of the chaser
v_init = v_ISS
ra_deorbit = a_ISS

# Define range of perigee radii
perigee = np.linspace(min_perigee, max_perigee, int((max_perigee - min_perigee)/stepsize)+1)

# Initialise solution array
DeltaV_deorbit = np.empty(np.size(perigee))

# Compute Delta V for a range of perigee radii
for i,p in enumerate(perigee):
    a_deorbit = (p + ra_deorbit)/2
    v_deorbit = np.sqrt(mu_e*(2/ra_deorbit - 1/a_deorbit))
    DeltaV_deorbit[i] = v_deorbit - v_init

# Plot the delta V for against the corresponding perigee altitude
fig = plt.figure()
plt.plot(perigee - Re, DeltaV_deorbit)
plt.xlabel("Perigee altitude (km)")
plt.ylabel("$\Delta$V (km/s)")
plt.xlim(min_perigee-Re,max_perigee-Re)
plt.xticks(np.arange(min_perigee-Re,max_perigee+15-Re,step = 15))
plt.show()
```