# Mission Analysis & Orbital Mechanics Hands-on Workshop (BE)

Siddesh Sutrave

April 2025

## 1. Solving Kepler's Equation

### a) Newton-Raphson Kepler Solver

Kepler's Equation relates the mean anomaly $M$, eccentric anomaly $E$, and eccentricity $e$ as:

$$E - e \sin E = M = n(t - t_0) \tag{1}$$

The mean motion $n$ is given by:

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

where:

- $n$ is the mean motion,
- $t$ is the time,
- $t_0$ is the time of periapsis passage,
- $\mu$ is the gravitational parameter of the central body,
- $a$ is the semi-major axis,
- $\tau$ is the orbital period.

The following Python function implements the Newton-Raphson method to solve Kepler's Equation numerically:

```
# Newton-Raphson method
def Kepler(M, e, tol):
    E = M
    delta_E = 1
    no_iter = 0
    while abs(delta_E) > tol:
        f = E - e * np.sin(E) - M
        delta_f = 1 - e * np.cos(E)
        delta_E = -(f / delta_f)
        E = E + delta_E
        no_iter += 1

    return E, no_iter
```

### b) Eccentric Anomaly Results

The following are the numerical results obtained using the Newton-Raphson method:

- **Case 1:** $M = 21°$ (converted to radians), $e = 0.25$, tolerance $= 10^{-6}$
  - Eccentric anomaly: $E = 0.48$ rad
  - Number of iterations: 3
- Using $E = 0.48$, semi-major axis $a = 24000$ km:

  - True anomaly: $\theta = 0.62$ rad
  - Radial distance: $r = 18685.04$ km
- With increased accuracy requirement (tol $= 10^{-12}$):
  - Number of iterations: 4
- **Case 2:** $M = 180°$ (i.e., $\pi$ rad)
  - Eccentric anomaly: $E = 3.14$ rad
  - True anomaly: $\theta = 3.14$ rad
  - Radial distance: $r = 30000.00$ km
  - Number of iterations: 1

**Interpretation of Result:** When $M = 180°$, the satellite is exactly at the apoapsis of the orbit. In this configuration:

$$r = a(1 + e)$$

which is the maximum distance from the central body. The satellite is moving at its slowest speed in the orbit due to conservation of angular momentum.

## c) MEO Satellite Orbit Plot

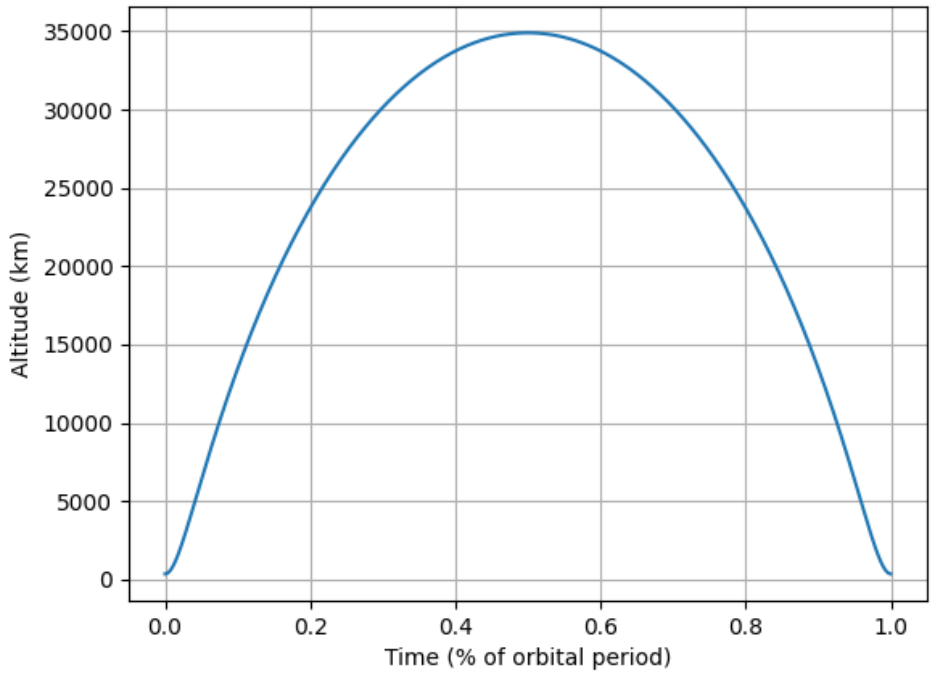


Figure 1: Altitude(km) vs. Time (Hours)

s

## d) Eclipse Analysis

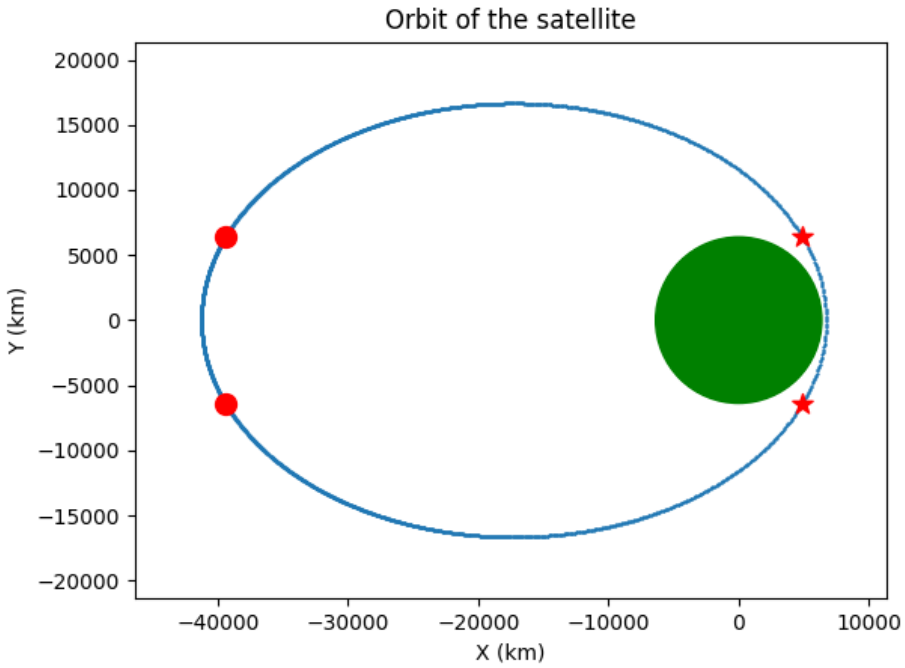


Figure 2: 2D Orbit and Eclipse Points

- Periapsis eclipse duration: 23.02 minutes
- Apoapsis eclipse duration: 131.27 minutes
- Periapsis angle: $\theta_1 = 0.92$ rad
- Apoapsis angle: $\theta_2 = 2.98$ rad

**Discussion:**
The satellite spends significantly more time in eclipse when the Earth is positioned at the apoapsis compared to the periapsis. This difference arises because:

- At **apoapsis**, the satellite's velocity is at its minimum (due to conservation of angular momentum), which means it moves more slowly and remains within the Earth's shadow for a longer duration.
- At **periapsis**, the satellite moves faster, so even though it may enter the shadow, it exits quickly, resulting in a much shorter eclipse duration.

## Reference Code for Section 1

```
import numpy as np
import matplotlib.pyplot as plt

R_E = 6378
u = 398600

#Newton-Raphson method
def Kepler(M,e,tol):

    E = M
    delta_E = 1
    no_iter =  0
    while abs(delta_E) > tol:

        f = E - e*np.sin(E) - M
        delta_f = 1 - e*np.cos(E)
        delta_E = -(f/delta_f)
        E = E + delta_E
        no_iter += 1

    return E,no_iter

def true_anomaly(E):

    theta = 2*np.arctan(np.tan(E/2)*np.sqrt((1+e)/(1-e)))

    return theta

def radial_distance(theta,a):

    r = a * (1-e**2) / (1+e*np.cos(theta))

    return r

def altitude(tp,n,tol,e,a):
    alt_series = []
    time_series = []
    theta_series = []
    for t in range(0,37005,15):
        M = n*t
        E, no_iter = Kepler(M,e,tol)
        theta = true_anomaly(E)
        r = radial_distance(theta,a)
        alt = r - R_E
        alt_series.append(alt)
        time_series.append(t/tp)
        theta_series.append(theta)
    print(no_iter)
    return alt_series,time_series, no_iter, theta_series

def orbit(theta_series,alt_series):
    series_x = []
    series_y = []
    for i in range(len(alt_series)):
        altitude = alt_series[i]
        x = (altitude + R_E) * np.cos(theta_series[i])
        y = (altitude + R_E) * np.sin(theta_series[i])
        series_x.append(x)
        series_y.append(y)

    return series_x, series_y

def theta(e,p,a,R_E):

    alpha = R_E**2 * e**2 + p**2
    beta = 2 * R_E**2 * e
    gamma = R_E**2 - p**2

    discriminant = beta**2 - 4*alpha*gamma
    if discriminant < 0:
        return None

    cos_theta1 = (-beta + np.sqrt(discriminant)) / (2*alpha)
    cos_theta2 = (-beta - np.sqrt(discriminant)) / (2*alpha)
```

```
    theta1 = np.arccos(cos_theta1)
    theta2 = np.arccos(cos_theta2)

    angles = [theta1, -theta1, theta2, -theta2]
    markers = ['*', '*', 'o', 'o']
    colors = ['red', 'red', 'red', 'red']

    for angle, marker, color in zip(angles, markers, colors):
        r = radial_distance(angle, a)
        x = (r) * np.cos(angle)
        y = (r) * np.sin(angle)
        plt.scatter(x, y, color=color, marker=marker,s=100, label=f"0={angle:.2f} rad")

    return angles, theta1, theta2

def eclipse_time(theta1,theta2,e,n,tp):
    E1 = 2*np.arctan(np.tan(theta1/2)*np.sqrt((1-e)/(1+e)))
    E2 = 2*np.arctan(np.tan(-theta1/2)*np.sqrt((1-e)/(1+e)))
    E3 = 2*np.arctan(np.tan(theta2/2)*np.sqrt((1-e)/(1+e)))
    E4 = 2*np.arctan(np.tan(-theta2/2)*np.sqrt((1-e)/(1+e)))
    M1 = E1 - e*np.sin(E1)
    M2 = E2 - e*np.sin(E2)
    M3 = E3 - e*np.sin(E3)
    M4 = E4 - e*np.sin(E4)
    periapsis_eclipse_time = (abs(M2 - M1) / n)/60
    apoapsis_eclipse_time = (tp - M3/n + M4/n)/60
    return periapsis_eclipse_time, apoapsis_eclipse_time

if __name__ == "__main__":

#    Input for 1a, 1b

    # # M = 21*(np.pi/180)
    # M = 180*(np.pi/180)
    # e = 0.25
    # tol = 1e-12

    # E, no_iter = Kepler(M,e,tol)
    # theta = true_anomaly(E)
    # # print(f"Eccentric anomaly: {E:.2f} rad")
    # # print(f"Number of iterations: {no_iter}")
    # a = 24000
    # r = radial_distance(theta,a)
    # print(f"Eccentric anomaly: {E:.2f} rad")
    # print(f"True anomaly: {theta:.2f} rad")
    # print(f"Radial distance: {r:.2f} km")
    # print(f"Number of iterations: {no_iter}")
    # print(E, no_iter,theta,r)


#    Input for 1c

    u = 398600
    a = 24000
    e = 0.72
    tol = 1e-12
    n= np.sqrt(u/a**3)
    tp= (2*np.pi)/n
    p = a*(1-e**2)
    x_center, y_center = 0, 0

    alt_series, time_series,no_iter, theta_series = altitude(tp,n,tol,e,a)

    series_x,series_y = orbit(theta_series,alt_series)

    plt.plot(time_series,alt_series)
    plt.xlabel("Time (% of orbital period)")
    plt.ylabel("Altitude (km)")
    plt.grid()
    plt.show()

    plt.scatter(series_x,series_y,linewidths=0.5, s=1)
    circle = plt.Circle((x_center, y_center), R_E, color='green', fill=True)
    ax = plt.gca()  # Get the current axes
    ax.add_patch(circle)
```

```
    ax.set_xlim(-R_E-40000, R_E+5000)
    ax.set_ylim(-R_E-15000, R_E+15000)
    plt.gca().set_aspect('equal', adjustable='box')
    angles, theta1, theta2 = theta(e, p, a, R_E)
    plt.xlabel("X (km)")
    plt.ylabel("Y (km)")
    plt.title("Orbit of the satellite")
    plt.show()

    eclipse_duration = eclipse_time(theta1, theta2, e, n,tp)
    print(f"Periapsis eclipse duration: {eclipse_duration[0]:.2f} minutes, Apoapsis eclipse
        ↪ duration: {eclipse_duration[1]:.2f} minutes")
    print(f"Periapsis angle: {theta1:.2f}, Apoapsis angle: {theta2:.2f}")
```

# 2. Numerical Integration of the Equations of Motion

## a) & b) Equations of Motion in 3D and Function `twobody(t, X)`

```
# Two body equations of motion in 3D

def two_body(t, X):

    r = np.sqrt(X[0]**2 + X[1]**2 + X[2]**2)
    # v = r_dot = np.array([vx, vy, vz])
    X_dot = np.array([X[3], X[4], X[5], -u*X[0]/r**3, -u*X[1]/r**3, -u*X[2]/r**3])

    return X_dot
```

## c) Orbit Integration

Plot of the magnitude of the position vector versus time:

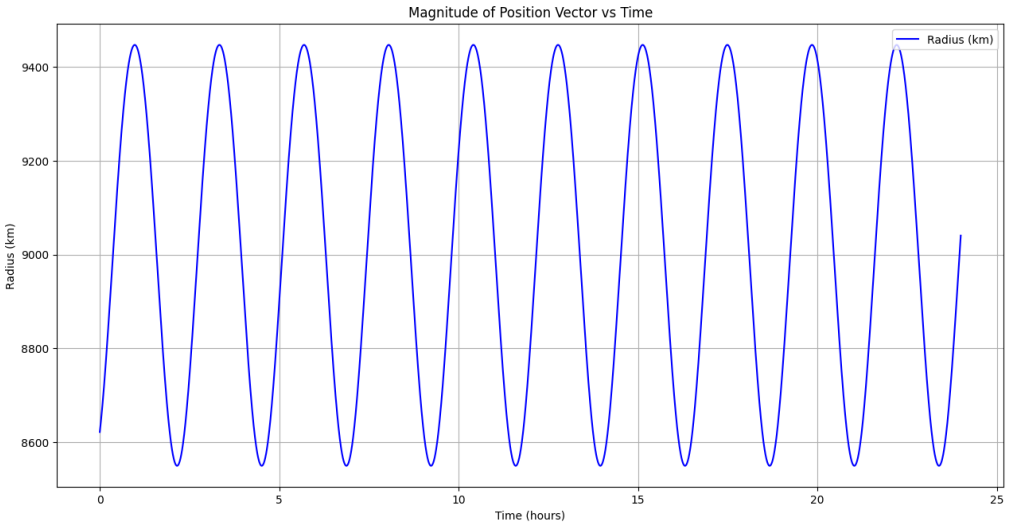


Figure 3: Position Magnitude vs. Time

Plot of the magnitude of the velocity vector versus time:

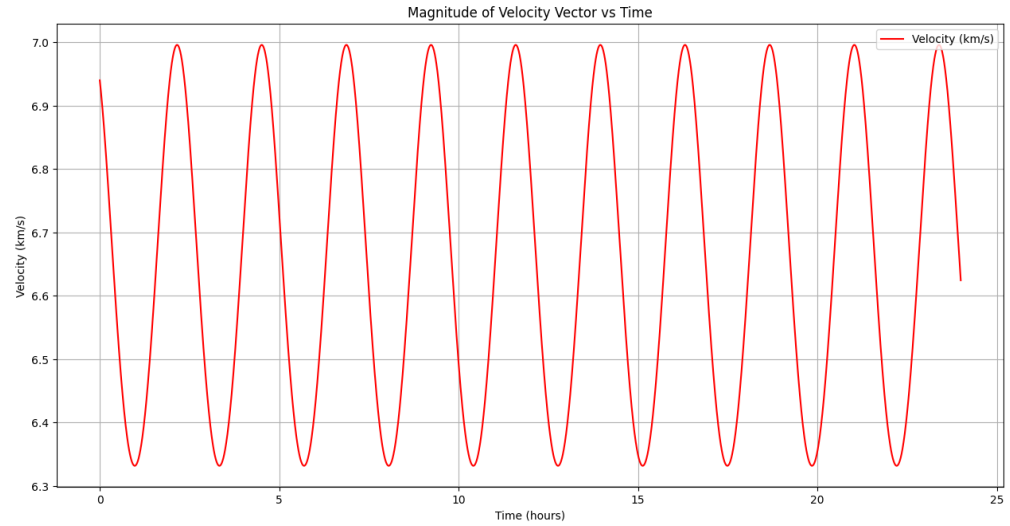


Figure 4: Velocity Magnitude vs. Time

3D plot of the orbit including a sphere representing the Earth:

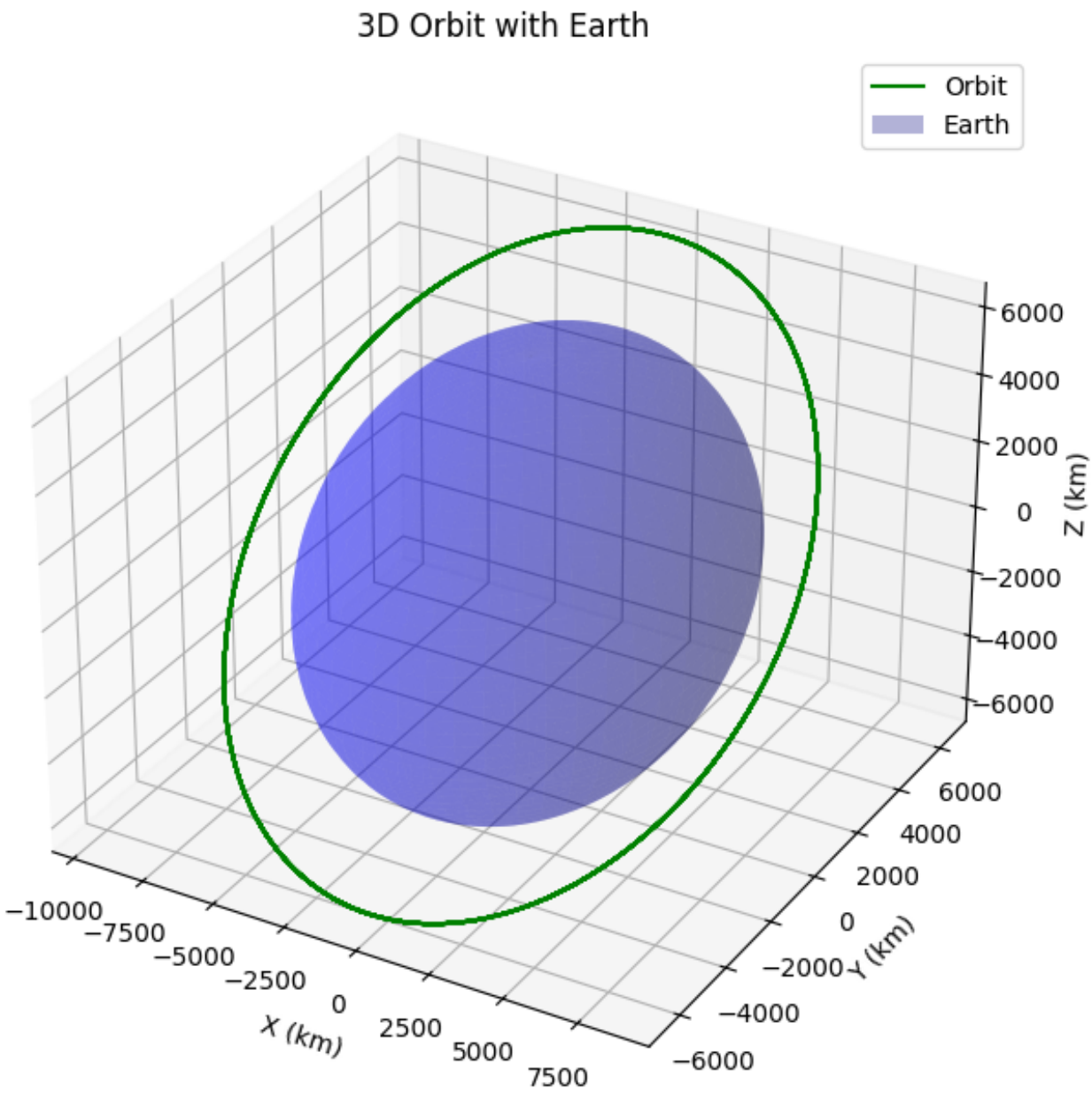


Figure 5: 3D Orbit of Satellite

## d) Specific Energies and Angular Momentum

Specific Energies (Kinetic, Potential, and Total) as a function of time:

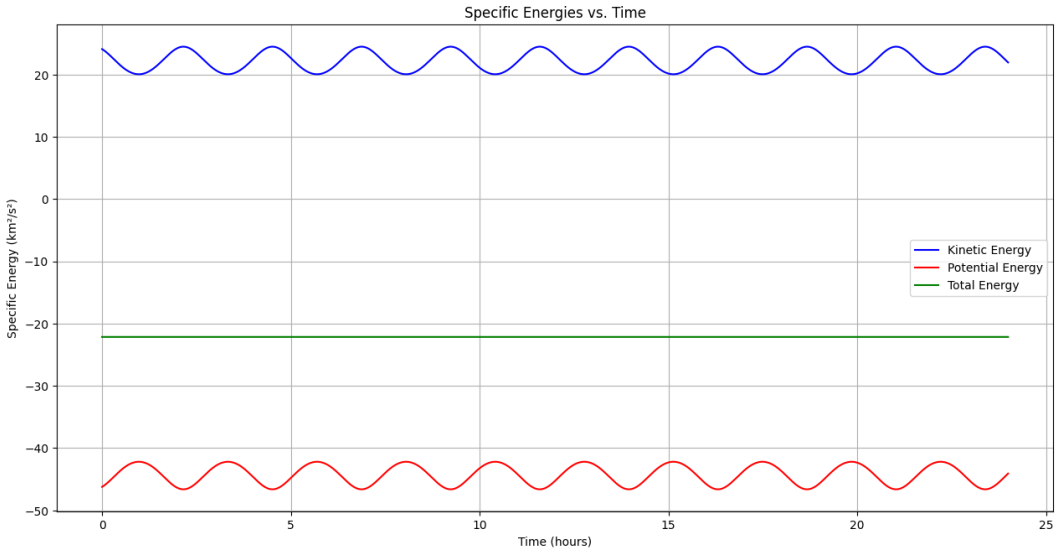


Figure 6: Specific Energies over Time

Specific Angular Momentum over time:

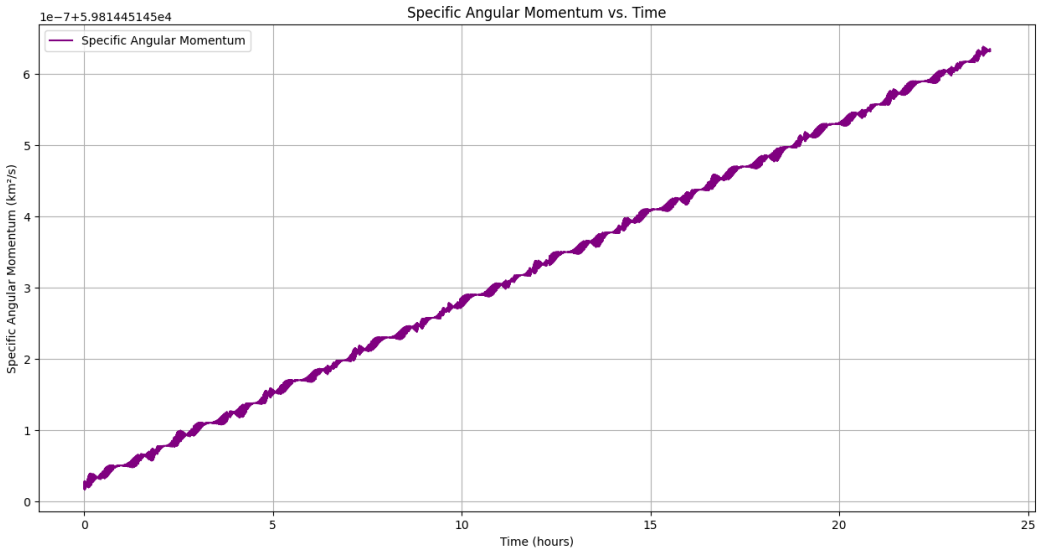


Figure 7: Specific Angular Momentum over Time

**Discussion:**
From the energy plot, we observe the following:

- The **kinetic energy** fluctuates over the orbit, reaching a maximum at *periapsis* (where the satellite is moving fastest) and a minimum at *apoapsis*.
- The **potential energy** follows an inverse pattern — most negative at periapsis (closest to Earth), and least negative at apoapsis.
- The **total specific energy** remains nearly constant throughout the orbit, indicating that the numerical integration conserves orbital energy well, as expected for a two-body system under gravity alone.

The **specific angular momentum is conserved** — a fundamental property in central force motion. This serves as another verification of both the physics and the accuracy of the numerical integration.

## e) New Initial Orbit - Analysis

Initial conditions:

- Position vector: $\mathbf{r} = [0,\ 0,\ 8550]$ km
- Velocity vector: $\mathbf{v} = [0,\ -7.0,\ 0]$ km/s

Plot of the magnitude of the position vector versus time:

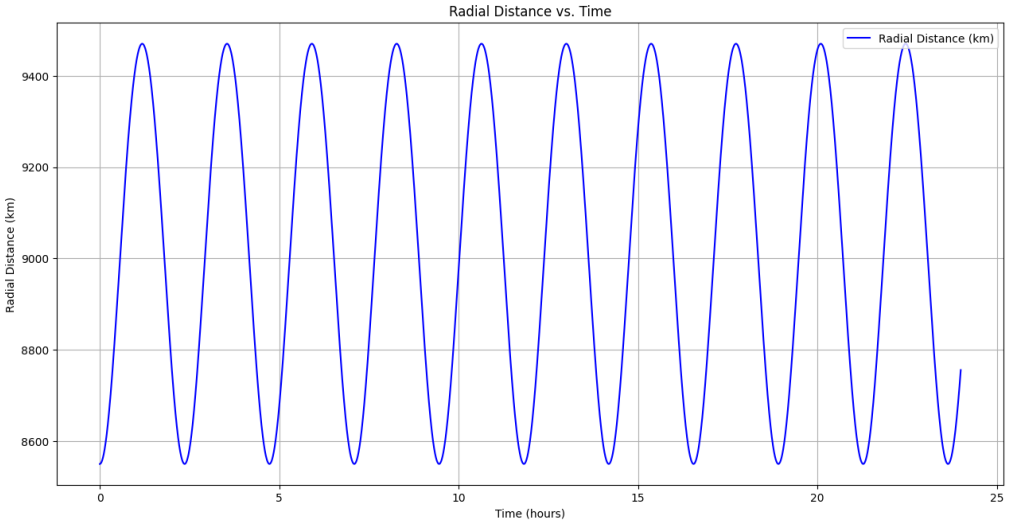


Figure 8: Position Magnitude vs. Time

Plot of the magnitude of the velocity vector versus time:

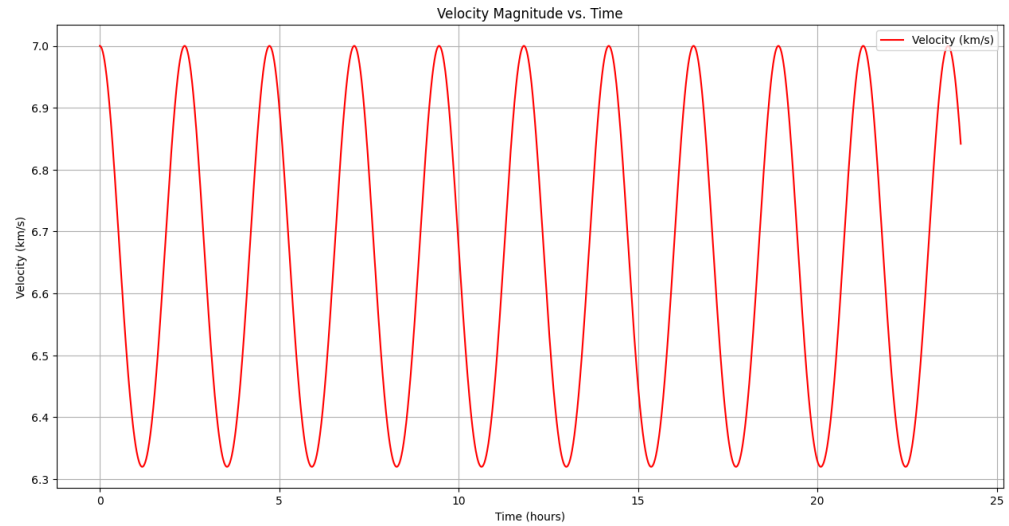


Figure 9: Velocity Magnitude vs. Time

Computed orbital parameters:

- Semi-major axis $a = 9009.99$ km

- Inclination $i = 90.00°$
- Orbital period $T = 141.86$ minutes

**Orbit Type Determination:**
The satellite's position vector lies entirely along the $z$-axis, and its velocity vector lies along the $y$-axis. This configuration yields an inclination of 90°, which indicates a **polar orbit** the satellite passes over the Earth's poles.

## Reference Code for Section 2

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


R_E = 6378
u = 398600

# Two body equations of motion in 3D

def two_body(t, X):

    r = np.sqrt(X[0]**2 + X[1]**2 + X[2]**2)
    # v = r_dot = np.array([vx, vy, vz])
    X_dot = np.array([X[3], X[4], X[5], -u*X[0]/r**3, -u*X[1]/r**3, -u*X[2]/r**3])

    return X_dot

# def specific_energy(X):
#     r = np.sqrt(X[0]**2 + X[1]**2 + X[2]**2)
#     v = np.sqrt(X[3]**2 + X[4]**2 + X[5]**2)
#     K_E = (v**2)/2
#     U_E = -u/r
#     Total_E = K_E + U_E
#     return Total_E,K_E,U_E

if __name__ == "__main__":
    X0 = np.array([7115.804, 3391.696,3492.221,-3.762, 4.063, 4.184])
    t_val = np.arange(0, 86400, 10)
    two_body(t_val, X0)
    sol = solve_ivp(two_body, [0, 86400], X0, t_eval=t_val, method='RK45', rtol=1e-12, atol=1e
        ↪ -12)
    time = sol.t / 3600

    # Plotting Magnitude of position vs time
    radius = np.sqrt(sol.y[0]**2 + sol.y[1]**2 + sol.y[2]**2)
    plt.plot(time, radius, label="Radius (km)", color="blue")
    plt.xlabel("Time (hours)")
    plt.ylabel("Radius (km)")
    plt.title("Magnitude of Position Vector vs Time")
    plt.grid(True)
    plt.legend(loc="upper right")
    plt.show()

    # Plotting Magnitude of velocity vector vs time
    velocity = np.sqrt(sol.y[3]**2 + sol.y[4]**2 + sol.y[5]**2)
    plt.plot(time, velocity, label="Velocity (km/s)", color="red")
    plt.xlabel("Time (hours)")
    plt.ylabel("Velocity (km/s)")
    plt.title("Magnitude of Velocity Vector vs Time")
    plt.grid(True)
    plt.legend(loc="upper right")
    plt.show()

    # Plotting orbit in 3D
    x = sol.y[0]
    y = sol.y[1]
    z = sol.y[2]
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.plot3D(x, y, z, label="Orbit", color="green")
    # Earth sphere for visualization
    phi = np.linspace(0, np.pi, 100)
```

```
    theta = np.linspace(0, 2 * np.pi, 100)
    phi, theta = np.meshgrid(phi, theta)
    x_Earth = R_E * np.sin(phi) * np.cos(theta)
    y_Earth = R_E * np.sin(phi) * np.sin(theta)
    z_Earth = R_E * np.cos(phi)
    ax.plot_surface(x_Earth, y_Earth, z_Earth, color='blue', alpha=0.3, label="Earth")
    ax.set_xlabel("X (km)")
    ax.set_ylabel("Y (km)")
    ax.set_zlabel("Z (km)")
    ax.set_title("3D Orbit with Earth")
    ax.legend()
    plt.show()

    # Plotting specific energies vs time
    vx = sol.y[3]
    vy = sol.y[4]
    vz = sol.y[5]
    K_E = (velocity**2)/2
    U_E = -u/radius
    Total_E = K_E + U_E
    plt.plot(time, K_E, label="Kinetic Energy", color="blue")
    plt.plot(time, U_E, label="Potential Energy", color="red")
    plt.plot(time, Total_E, label="Total Energy", color="green")
    plt.xlabel("Time (hours)")
    plt.ylabel("Specific Energy (km^2/s^2)")
    plt.title("Specific Energies vs. Time")
    plt.grid(True)
    plt.legend()
    plt.show()

    # Specific angular momentum
    hx = y * vz - z * vy
    hy = z * vx - x * vz
    hz = x * vy - y * vx
    h = np.sqrt(hx**2 + hy**2 + hz**2)
    # Plotting specific angular momentum vs time
    plt.plot(time, h, label="Specific Angular Momentum", color="purple",linestyle="-")
    plt.xlabel("Time (hours)")
    plt.ylabel("Specific Angular Momentum (km^2/s)")
    plt.title("Specific Angular Momentum vs. Time")
    plt.grid(True)
    plt.legend()
    plt.show()


    # Input for 2e
    X1 = np.array([0,0,8550,0,-7,0])
    sol1 = solve_ivp(two_body, [0, 86400], X1, t_eval=t_val, method='RK45', rtol=1e-12, atol=1
        ↪ e-12)
    time1 = sol1.t / 3600
    x1, y1, z1 = sol1.y[0], sol1.y[1], sol1.y[2]
    vx1, vy1, vz1 = sol1.y[3], sol1.y[4], sol1.y[5]
    radius1 = np.sqrt(x1**2 + y1**2 + z1**2)
    velocity1 = np.sqrt(vx1**2 + vy1**2 + vz1**2)
    K_E1 = (velocity1**2)/2
    U_E1 = -u/radius1
    Total_E1 = K_E1 + U_E1
    hx1 = y1 * vz1 - z1 * vy1
    hy1 = z1 * vx1 - x1 * vz1
    hz1 = x1 * vy1 - y1 * vx1
    h1 = np.sqrt(hx1**2 + hy1**2 + hz1**2)
    i = np.arccos(hz1.mean()/h1.mean())*180/np.pi
    a = u / (2 * abs(Total_E1.mean()))
    time_period = 2 * np.pi * np.sqrt(a**3 / u)
    print(f"Semi-major axis (a): {a:.2f}")
    print(f"Inclination (i): {i:.2f}")
    print(f"Time period (T): {time_period/60:.2f} minutes")

    # Plot radial distance vs. time
    plt.plot(time1, radius1, label="Radial Distance (km)", color="blue")
    plt.xlabel("Time (hours)")
    plt.ylabel("Radial Distance (km)")
    plt.title("Radial Distance vs. Time")
    plt.grid(True)
    plt.legend()
    plt.legend(loc="upper right")
```

```
    plt.show()

    # Plot velocity magnitude vs. time
    plt.plot(time1, velocity1, label="Velocity (km/s)", color="red")
    plt.xlabel("Time (hours)")
    plt.ylabel("Velocity (km/s)")
    plt.title("Velocity Magnitude vs. Time")
    plt.grid(True)
    plt.legend()
    plt.legend(loc="upper right")
    plt.show()
```

# 3. Orbit Phasing and Rendezvous

## a) ISS Orbit Simulation

The following plot shows the simulated orbit of the International Space Station (ISS) based on the provided orbital parameters:

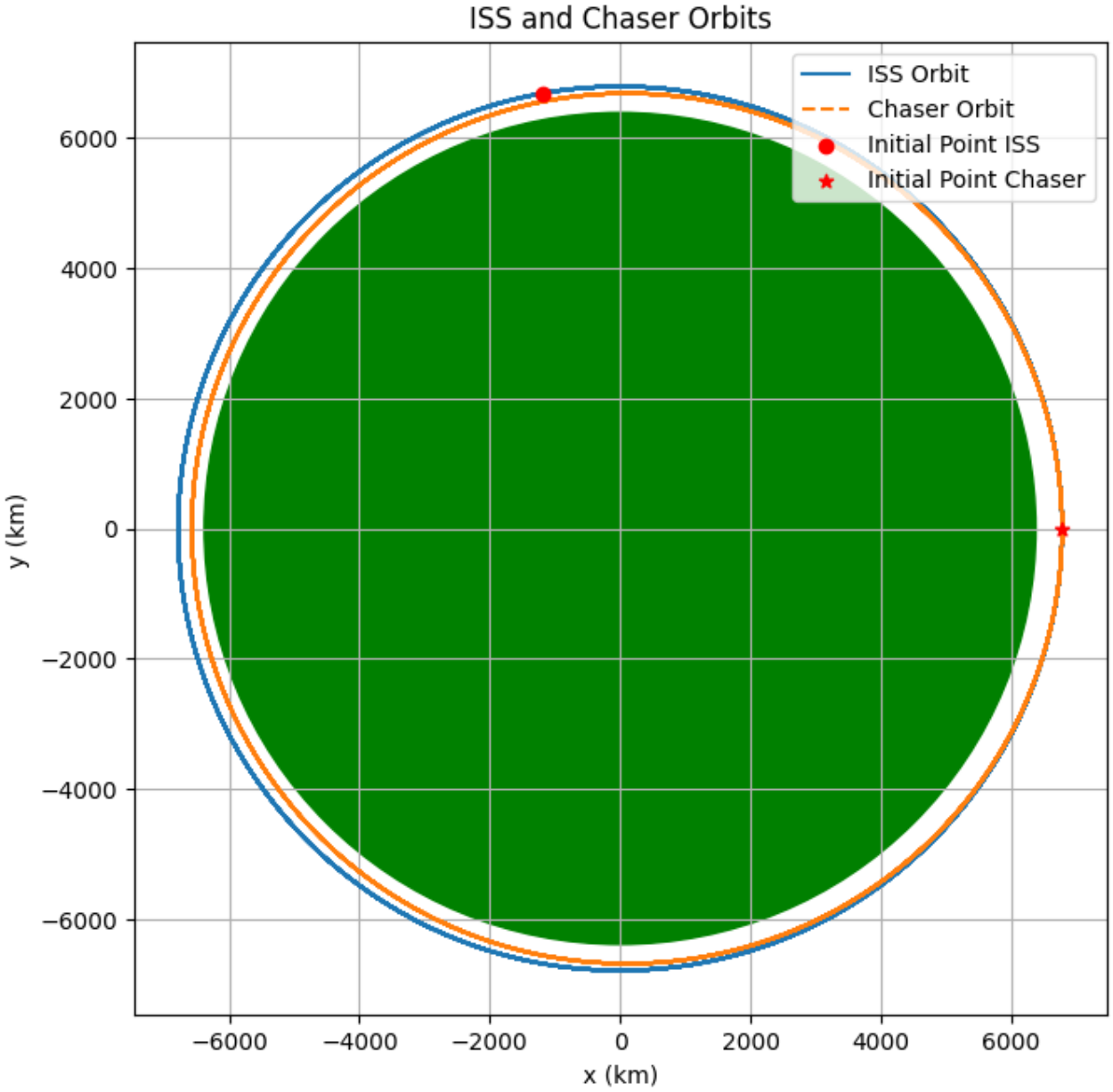


Figure 10: ISS and Chaser Orbits

**Period of ISS: 1.54 hours**

## b) Chaser Satellite Interception

To intercept the ISS after a specified number of revolutions $N_{\text{rev}}$, the semi-major axis of the chaser orbit is computed using the phasing relationship:

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

Given that the chaser's apogee is set equal to the orbital radius of the ISS, the eccentricity of the chaser orbit is calculated as:

$$e_{\text{chaser}} = \frac{a_{\text{ISS}}}{a_{\text{chaser}}} - 1 \tag{4}$$

Using the above equations:

- Semi-major axis of chaser: $a_{\text{chaser}} = 6676.93$ km
- Eccentricity of chaser: $e_{\text{chaser}} = 0.02$

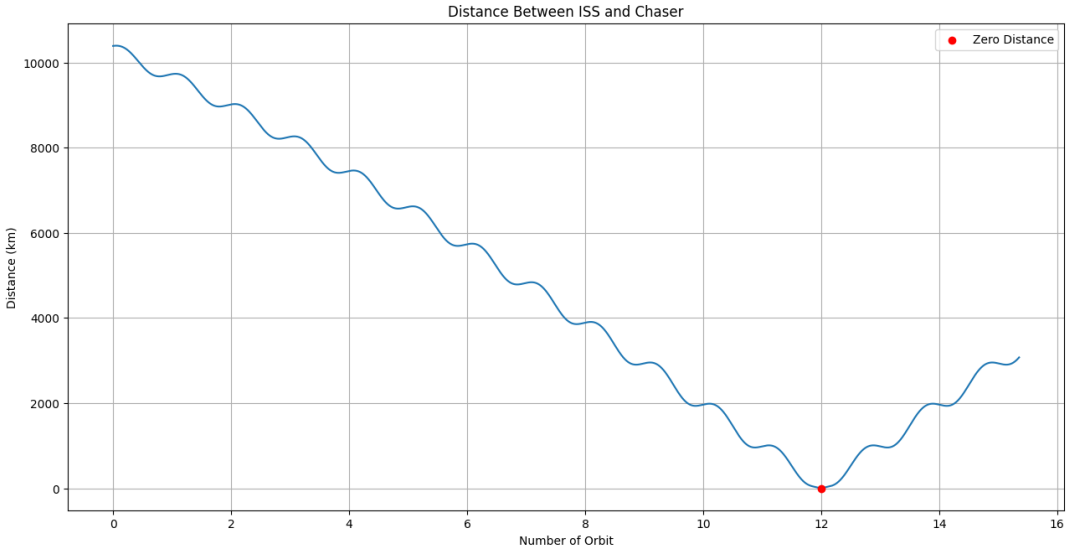


Figure 11: Distance between the ISS and chaser

## c) Delta-V for Various Rendezvous Times

The following plot shows how the required $\Delta V$ at apogee varies as a function of the number of revolutions $N_{\text{rev}}$ over which the chaser spacecraft is allowed to rendezvous with the ISS.

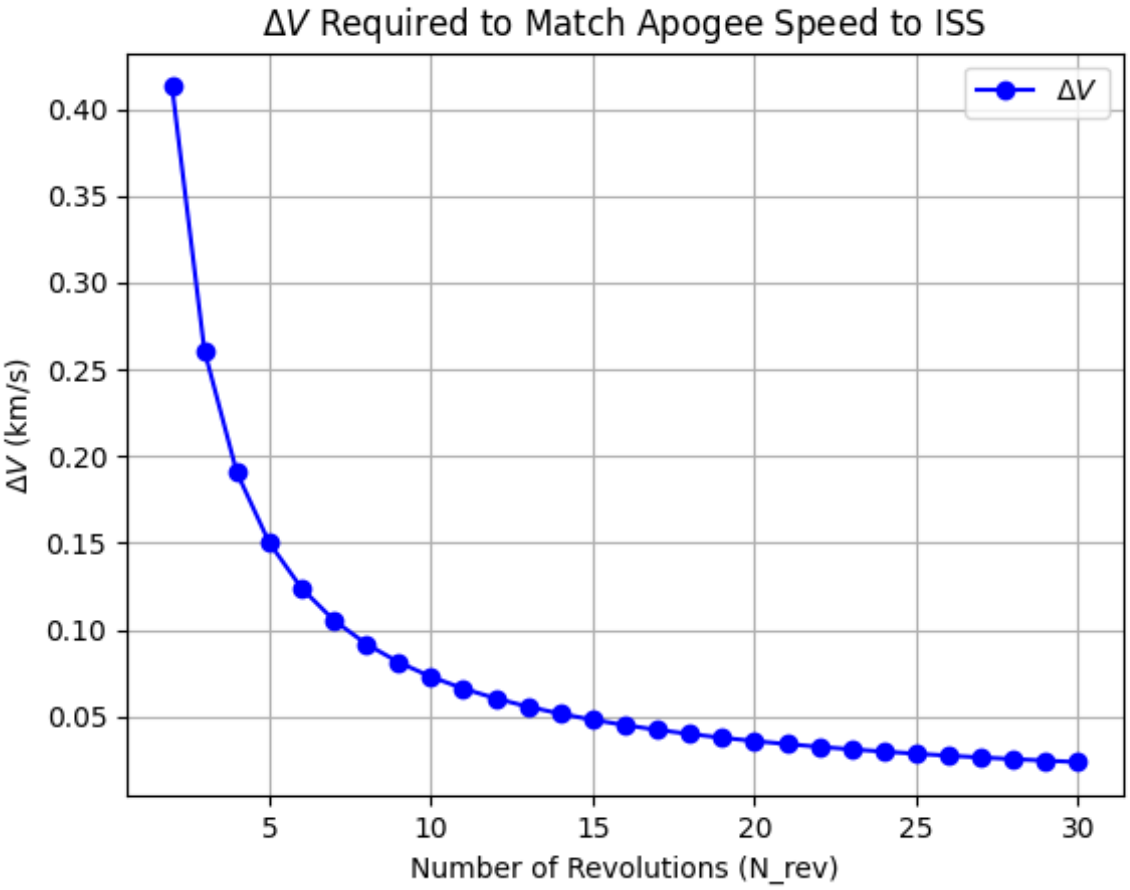


Figure 12: Delta-V Required vs. Number of Revolutions

**Discussion:**

As seen in the plot, the required $\Delta V$ decreases significantly with increasing number of revolutions. This behavior can be explained by orbital mechanics:

- With fewer revolutions, the chaser must catch up more quickly, necessitating a more eccentric (elliptical) orbit and a larger difference in velocity at apogee to match the ISS — resulting in a higher $\Delta V$.
- Allowing more revolutions provides more time for phasing. The chaser can adopt an orbit only slightly different from the ISS, resulting in a smaller velocity mismatch at apogee and thus a smaller $\Delta V$.

## d) De-orbit Maneuver

After docking with the ISS and completing its mission, the Chaser must perform a de-orbit maneuver. This involves applying a retrograde $\Delta V$ at apogee to lower the perigee altitude into Earth's upper atmosphere, ensuring re-entry and burn-up.

The following plot illustrates how the required $\Delta V$ changes for different target perigee altitudes ranging from 60 km to 210 km:

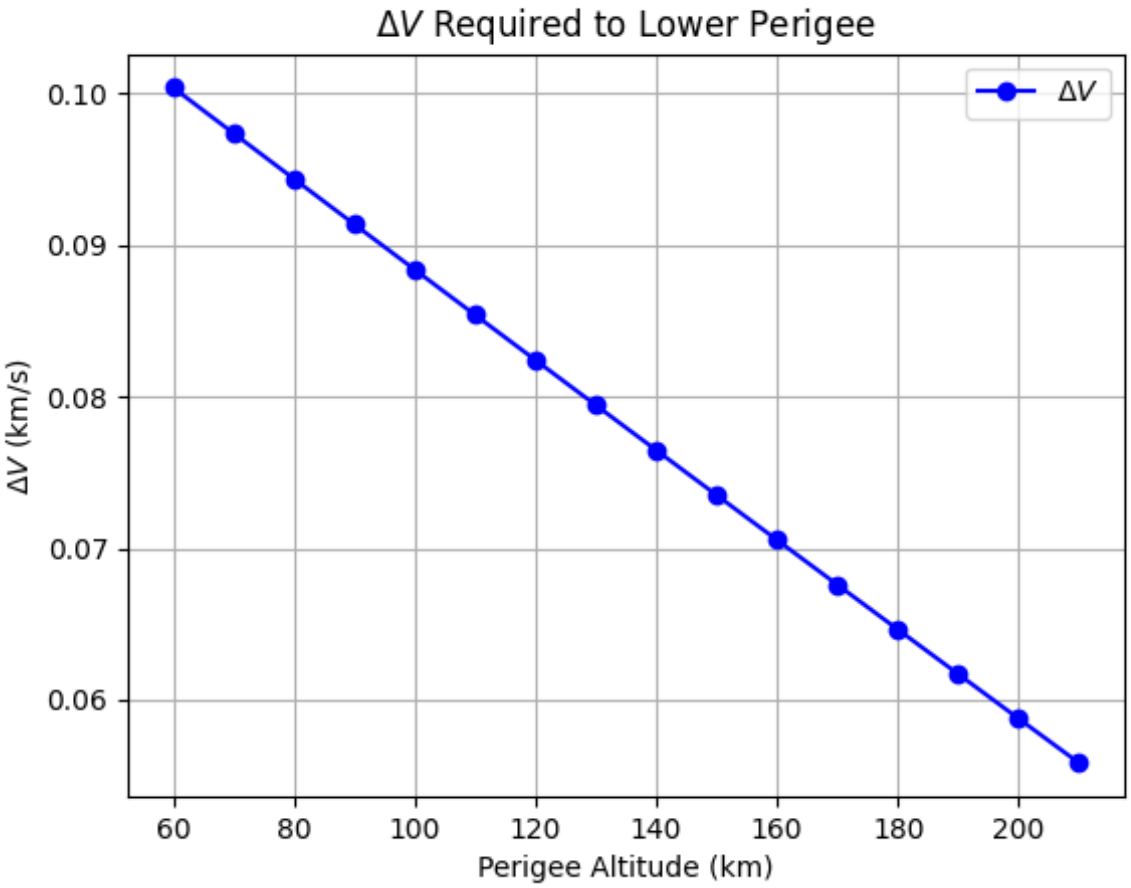


Figure 13: Delta-V Required for De-orbit

**Discussion:**
The relationship between perigee altitude and required $\Delta V$ is nearly linear over the range of 60 km to 210 km. Key observations:

- Lowering the perigee from the ISS altitude ( 404 km) to 200 km requires a relatively small $\Delta V$, due to the minimal energy difference.
- As the target perigee approaches the denser layers of the atmosphere (e.g., 60 km), the required $\Delta V$ increases more substantially.
- The typical de-orbit burn is optimized to balance safe atmospheric entry and fuel efficiency.

## Reference Code for Section 3

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


R_E = 6378
u = 398600

def two_body(t, X):

    r = np.sqrt(X[0]**2 + X[1]**2 + X[2]**2)
    X_dot = np.array([X[3], X[4], X[5], -u*X[0]/r**3, -u*X[1]/r**3, -u*X[2]/r**3])

    return X_dot

if __name__ == "__main__":
    theta = 100 * np.pi / 180
    altitude = 404
    r_iss = R_E + altitude
    x_iss = r_iss * np.cos(theta)
    y_iss = r_iss * np.sin(theta)
    z_iss = 0
    vx_iss = -np.sqrt(u/r_iss) * np.sin(theta)
    vy_iss = np.sqrt(u/r_iss) * np.cos(theta)
    vz_iss = 0
    X_iss = np.array([x_iss, y_iss, z_iss, vx_iss, vy_iss, vz_iss])
    a_iss = R_E + altitude
    tp_iss = (2 * np.pi * np.sqrt(a_iss**3 / u) )
    x_center, y_center = 0, 0

    print(f"Period of ISS: {tp_iss/3600:.2f}  hours")

    t_eval = np.arange(0, 15*tp_iss, 10)
    two_body(t_eval, X_iss)
    sol_iss = solve_ivp(two_body, [0, 15*tp_iss], X_iss, t_eval=t_eval, method='RK45', rtol=1e
        ↪ -12, atol=1e-12)
    x_iss = sol_iss.y[0]
    y_iss = sol_iss.y[1]
    z_iss = sol_iss.y[2]

    N_rev = 12
    a_chaser = a_iss * (1 - (theta / (2 * np.pi * N_rev)))**(2/3)
    e_chaser = a_iss / a_chaser - 1

    v_chaser = np.sqrt(u*((2/r_iss) - (1/a_chaser)))
    print(f"Semi-major axis of chaser: {a_chaser:.2f} km")
    print(f"Eccentricity of chaser: {e_chaser:.2f}")

    x_chaser = r_iss
    y_chaser = 0
    z_chaser = 0
    vx_chaser = 0
    vy_chaser = v_chaser
    vz_chaser = 0
    tp_chaser = (2 * np.pi * np.sqrt(a_chaser**3 / u) )
    X_chaser = np.array([x_chaser, y_chaser, z_chaser, vx_chaser, vy_chaser, vz_chaser])
    print(X_chaser)
    sol_chaser = solve_ivp(two_body, [0, 15*tp_iss], X_chaser, t_eval=t_eval, method='RK45',
        ↪ rtol=1e-12, atol=1e-12)
    x_chaser, y_chaser, z_chaser = sol_chaser.y[0], sol_chaser.y[1], sol_chaser.y[2]

    distance = np.sqrt((x_iss - x_chaser)**2 + (y_iss - y_chaser)**2 + (z_iss - z_chaser)**2)

    plt.plot(x_iss, y_iss, label="ISS Orbit")
    plt.plot(x_chaser, y_chaser, label="Chaser Orbit", linestyle='--')
    plt.scatter(x_iss[0], y_iss[0], color='red', label="Initial Point ISS", zorder=5, marker='
        ↪ o')
    plt.scatter(x_chaser[0], y_chaser[0], color='red', label="Initial Point Chaser", zorder=5,
        ↪  marker='*')
    circle = plt.Circle((x_center, y_center), R_E, color='green', fill=True)
    ax = plt.gca()
    ax.add_patch(circle)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.xlabel('x (km)')
```

```
    plt.ylabel('y (km)')
    plt.title('ISS and Chaser Orbits')
    plt.legend()
    plt.legend(loc="upper right")
    plt.grid()
    plt.show()



    plt.plot(t_eval/tp_chaser, distance)
    plt.xlabel('Number of Orbit')
    plt.ylabel('Distance (km)')
    plt.title('Distance Between ISS and Chaser')
    plt.grid()


    zero_distance_index = np.argmin(distance)
    plt.scatter(t_eval[zero_distance_index]/tp_chaser, distance[zero_distance_index], color='
        ↪ red', label='Zero Distance', zorder=5, marker='o')

    plt.legend()
    plt.show()

    v_iss = np.sqrt(u/r_iss)
    N_rev_range = np.arange(2, 31)
    delta_v_values = []

    for N_rev in N_rev_range:
        a_chaser = a_iss * (1 - (theta / (2 * np.pi * N_rev)))**(2/3)
        e_chaser = a_iss / a_chaser - 1
        r_apoapsis = a_chaser * (1 + e_chaser)
        v_apoapsis_chaser = np.sqrt(u*((2/r_apoapsis) - (1/a_chaser)))
        delta_v = abs(v_apoapsis_chaser - v_iss)
        delta_v_values.append(delta_v)

    plt.plot(N_rev_range, delta_v_values, marker='o', color='blue', label=r'$\Delta V$')
    plt.xlabel('Number of Revolutions (N_rev)')
    plt.ylabel(r'$\Delta V$ (km/s)')
    plt.title(r'$\Delta V$ Required to Match Apogee Speed to ISS')
    plt.grid(True)
    plt.legend()
    plt.show()

    v_circular = np.sqrt(u/r_iss)
    perigee_altitudes = np.arange(60, 211, 10)  # Perigee altitudes from 60 km to 210 km
    delta_v_values = []

    for perigee_altitude in perigee_altitudes:
        r_perigee = R_E + perigee_altitude
        a_new = (r_iss + r_perigee) / 2
        v_apogee_new = np.sqrt(u*((2/r_iss) - (1/a_new)))
        delta_v = abs(v_apogee_new - v_circular)
        delta_v_values.append(delta_v)

    plt.plot(perigee_altitudes, delta_v_values, marker='o', color='blue', label=r'$\Delta V$')
    plt.xlabel('Perigee Altitude (km)')
    plt.ylabel(r'$\Delta V$ (km/s)')
    plt.title(r'$\Delta V$ Required to Lower Perigee')
    plt.grid(True)
    plt.legend()
    plt.show()
```