Spacecraft Mission

Alex Freeman

May 2, 2025

This report will cover all parts of assignment one in the Aerospace 720 course.

In Two-body Orbital Dynamics, Newton's Second Law of Motion applies. Equating forces

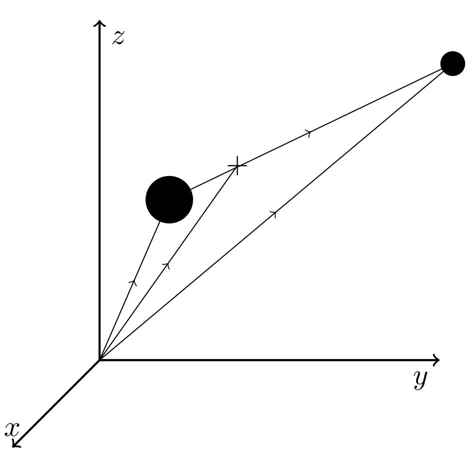
$$\bar{F} = m\bar{a} \tag{1}$$

$$\bar{F} = \frac{Gmm}{r^2}\hat{r} \tag{2}$$

Gives the equations of motion

$$\ddot{\bar{r}} = \frac{-\mu \cdot \bar{r}}{r^3} \tag{3}$$

This can be represented by the simple diagram showing two bodies with different masses where the arrows represent the position vectors involved.



1 Orbital Propagation

1.1 Solving Kepler's Equation

We need to solve Kepler's Equation using numerical methods. Using the Newton-Rasphon Method, we can take the eccentricity and mean anomaly as inputs and numerically solve for the eccentric anomaly.

```
def Kepler(e, M, tol = 1e-12, max_i = 1000):

    # Guess the solution is similar to theta_M
    E = M

    for i in range(max_i):

        # Define function in terms of f(E) = 0
        f_E = E - e * np.sin(E) - M

        # Define the derivative
        f_prime = 1 - e * np.cos(E)

        del_E = f_E / f_prime

        # Calculate the new eccentric anomaly
        E_new = E - del_E

        # If the error is within some tolerance
            return theta
        if np.abs(del_E) < tol:
            theta = 2*np.arctan(np.tan(E_new/2) *
                ((1+e)/(1-e))**(0.5))
            return theta

        # Else restart the for loop with a new value
        E = E_new
```

If we set the tolerance to $1e - 12$, we can compute the true anomaly of the asteroid at t_0 and $t_0 + 100$ days. A_ae0 is the OBJ data of the asteroid, it is an array.

```
# Using the Kepler function to find the true anomaly
    given an eccentricity and mean anomaly
```

```
true_a_t_0 = Kepler(A_ae0[2], A_ae0[6])

# Finding the true anomaly for t_0 + 100 days
mean_a_t_100 = get_mean_anomaly(100*(3600*24),
     A_ae0[6], A_ae0[1])
true_a_t_100 = Kepler(A_ae0[2], mean_a_t_100)
```

Printing these values gives that the true anomaly $\theta_{t_0} = 1.4246$ and $\theta_{t_0+100} = 2.1369$. Where these answers are in radians.

Now, we can create a function that takes in a state of orbital elements and returns the position and velocity vectors at that point. It uses a rotation matrix to convert from the perifocal frame to the ECI frame. This is defined via i, ω, and Ω terms and is calculated using the defined matrices in the appendix.

```
def COE2RV(arr, mu):

    # Orbital calculations for a, h, and r
    a, e, i, Omega, omega, theta_var = arr[0:6]
    h = np.sqrt(mu * a * (1 - e**2))
    r = a*(1-(e**2))/(1 + e*np.cos(theta_var))

    # Using orbital equations to find the position
         and velocity arrays
    arr_r = np.array([r*np.cos(theta_var),
         r*np.sin(theta_var), 0])
    arr_v = (mu/h)* np.array([-np.sin(theta_var), e +
         np.cos(theta_var), 0])

    #Rotate position and velocity from perifocal to
         the inertial frame using the transfomration
         matrix
    R_matrix = rotation_matrix(i, Omega, omega)

    # Perform matrix operations
    r_ijk = R_matrix @ arr_r
    v_ijk = R_matrix @ arr_v

    return r_ijk, v_ijk
```

Using this code, we can output the state vector at some time t. The first three values are the x, y, z positions in **km**. The last three are the velocity values in the x, y, z direction in **km/s**.

$$\textbf{At } t_0 :$$
$$\bar{\mathbf{X}} = \begin{bmatrix} x = -1.1694365\text{e+}08 \\ y = 1.53462780\text{e+}08 \\ z = -6.7446087\text{e+}06 \\ v_x = -3.1710203\text{e+}01 \\ v_y = -3.6285380\text{e+}00 \\ v_z = -1.8931546\text{e+}00 \end{bmatrix}$$

$$\textbf{At } t_0 + 100 :$$
$$\bar{\mathbf{X}} = \begin{bmatrix} x = -3.2057997\text{e+}08 \\ y = 6.72659396\text{e+}07 \\ z = -1.8991445\text{e+}07 \\ v_x = -1.6964807\text{e+}01 \\ v_y = -1.2943780\text{e+}01 \\ v_z = -1.0284663\text{e+}00 \end{bmatrix}$$

Next, there is a function called "Ephemeris". It returns the position and velocity at some time t.

```
def Ephemeris(t, OBJdata, mu):

    time, a, e, i, Omega, omega, mean_anomaly =
         OBJdata[0:7]

    # Find the mean motion
    nu_t = (mu / (a**3))**0.5

    # Find the mean anomaly at some time t
    t = t - t_0_days*days_convert
    mean_anomaly_t = mean_anomaly + nu_t * (t)

    h = np.sqrt(mu * a * (1 - e**2))
    theta_var = Kepler(e, mean_anomaly_t)
    r = a*(1-(e**2))/(1 + e*np.cos(theta_var))

    arr_r = np.array([r*np.cos(theta_var),
         r*np.sin(theta_var), 0])
    arr_v = (mu/h)* np.array([-np.sin(theta_var), e +
         np.cos(theta_var), 0])

    # Perform matrix operations on position and
         velocity arrays
    R_matrix = rotation_matrix(i, Omega, omega)
    r_ijk = R_matrix @ arr_r
    v_ijk = R_matrix @ arr_v

    return r_ijk, v_ijk
```

Using this code we can calculate the position and velocity vectors in the Sun's frame of reference.

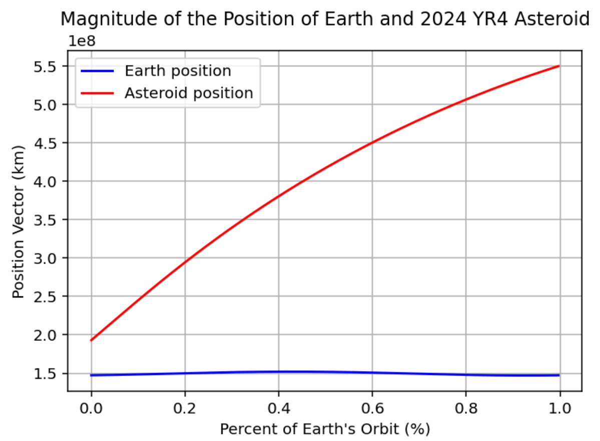


Figure I: Position vectors of Earth and the Asteroid for a full Earth orbital period

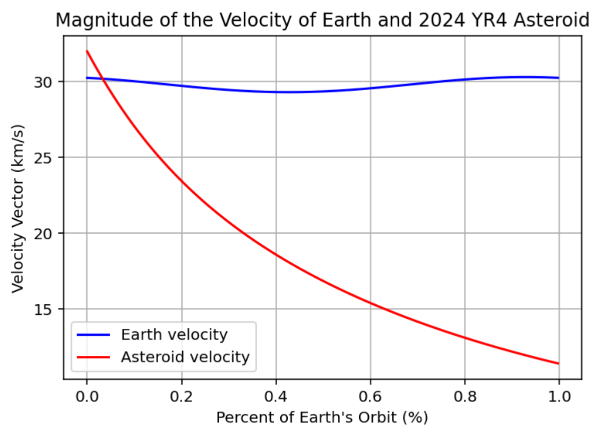


Figure II: Velocity vectors of Earth and the Asteroid for a full Earth orbital period

Next, we can plot the separation of the two bodies over ten years. Doing this we get the following graph

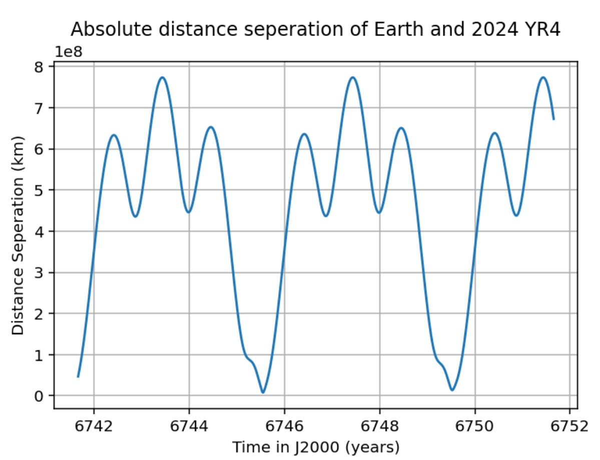


Figure III: Distance between Earth and 2024 YR4 asteroid in kilometers

We can see how the distance is sinusoidal in nature and repeats in an oscillatory fashion. This graph demonstrates three distinct peaks, where the middle one is the greatest. We can examine the sydonic period of the two bides by examining the equation of the two periods $\frac{1}{T_{syd}} = |\frac{1}{T_{Earth}} - \frac{1}{T_{Asteroid}}|$. And using the orbital periods, we can see that roughly every 1.33 years, the two bodies are at their closest approach. This lines up well with the graph produced, showing the magnitude of their separation.

1.2 Numerical Integration

To derive the necessary state function we have:

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3}\mathbf{r} \tag{1}$$

From here we know that $\frac{dv}{dt} = \dot{r}$ giving:

$$\frac{d\mathbf{v}}{dt} = -\frac{\mu}{r^3}\mathbf{r} \tag{2}$$

Expanding each vector as three-dimensional components in x, y, z:

$$\frac{dx}{dt} = v_x, \frac{dy}{dt} = v_y, \frac{dz}{dt} = v_z \tag{3}$$

$$\frac{dv_x}{dt} = -\frac{\mu}{r^3}x, \frac{dv_y}{dt} = -\frac{\mu}{r^3}y, \frac{dv_z}{dt} = -\frac{\mu}{r^3}z \tag{4}$$

Where $r = \sqrt{x^2 + y^2 + z^2}$
We can now define a state vector $\bar{\mathbf{X}}$

$$\bar{\mathbf{X}} = \begin{bmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{bmatrix} \tag{5}$$

Finally, deriving this state vector gives the following:

$$\dot{\mathbf{X}} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ -\frac{\mu}{r^3}x \\ -\frac{\mu}{r^3}y \\ -\frac{\mu}{r^3}z \end{bmatrix} \tag{6}$$

Now we can use a function to define the right-hand side of this equation called "TBP_ECI"

```
def TBP_ECI(t, state_X, mu):
    # Unpack state vector
    x, y, z, vx, vy, vz = state_X

    # Compute radius
    r = np.sqrt(x**2 + y**2 + z**2)

    # Acceleration components
    ax, ay, az = -mu * x / r**3, -mu * y / r**3, -mu
        * z / r**3
    return [vx, vy, vz, ax, ay, az]
```

With this function, we can use `SciPy's` integration feature with `solve_ivp`. We can pass through a set of initial conditions: a position and velocity vector. It passes through the gravitational parameter, μ, as an argument when solving the differential system. Furthermore, it uses the Runge-Kutta 45 method to integrate.

```
# Initial distance from Earth's centre (km)
r0 = np.linalg.norm(X0[:3])

# Initial speed (km/s)
v0 = np.linalg.norm(X0[3:])

# Semi-major axis (km)
a = 1 / (2 / r0 - v0**2 / mu_earth)

# Orbital period (s)
T = 2 * np.pi * np.sqrt(a**3 / mu_earth)

# Set integration time span for two orbital periods
t_start = 0
t_end = 2 * T # Two orbital periods
time_step = 10 # Output every 10 seconds
t_eval = np.arange(t_start, t_end, time_step)

# Solve the system using solve_ivp with specified
    tolerances using RK45
solution = solve_ivp(
    TBP_ECI, (t_start, t_end), X0, t_eval=t_eval,
        method='RK45',
    args=(mu_earth,), rtol=1e-12, atol=1e-12)


# Extract components
x, y, z = solution.y[0], solution.y[1], solution.y[2]
vx, vy, vz = solution.y[3], solution.y[4],
    solution.y[5]
```

Using an intital condition, $\bar{X}_0$, we can extrapolate the orbital path. Here, at time zero let

$$\bar{\mathbf{X}} = \begin{bmatrix} 4604.49276873138 \\ 1150.81472538679 \\ 4694.55079634563 \\ -5.10903235110107 \\ -2.48824074138143 \\ 5.62098648967432 \end{bmatrix} \tag{7}$$

And the following graph portrays the orbital path. It is plotted where the orbit goes behind the Earth with a dashed line to express the 3-D nature of this plot and to also show how this orbit has a very high inclination passing near the North Pole.

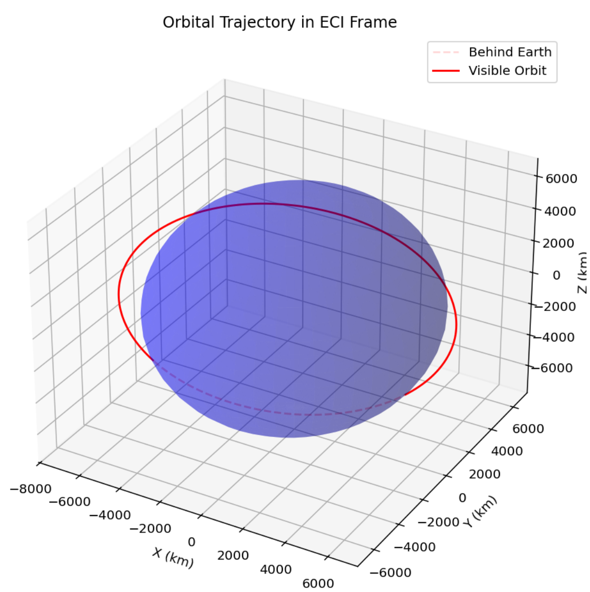


Figure IV: Orbital trajectory in ECI frame with initial conditions given from X_0

Next, we can calculate how the energies and angular momentum change over time. For each timestep, we can find these values using the following simple orbital equations

$$E_k = \frac{1}{2} \cdot |\bar{v}|^2 \tag{1}$$

$$E_p = \frac{-\mu_{earth}}{|\bar{r}|} \tag{2}$$

$$\bar{h} = \bar{r} \times \bar{v} \tag{3}$$

By graphing the magnitude of these quantities over time, we can see how the dynamics of this system

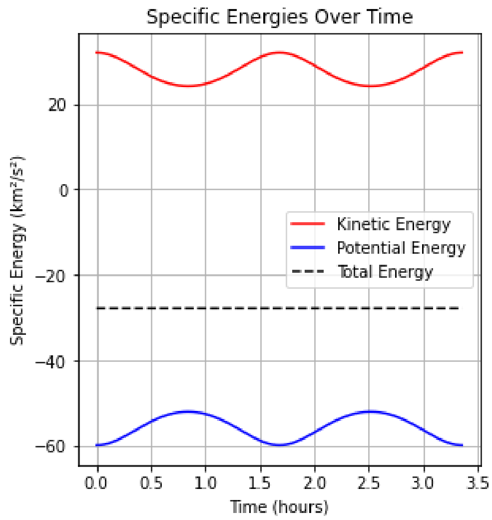


Figure V: How Kinetic, Potential, and Total energies change throughout the orbit

This graph shows the sinusoidal nature of the different specific energies. Importantly, we can see how the kinetic and potential energies always sum to the same value. This ensures that the total energy remains constant, which we would expect as this is a closed system with no external forces.

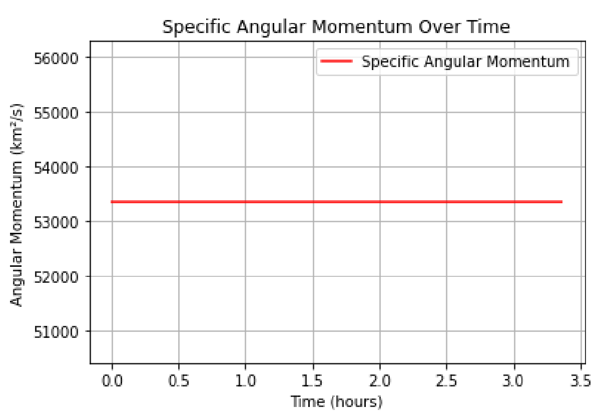


Figure VI: Graph shows the change in the angular momentum over the orbit is zero

Here, we see this graph depicting how the angular momentum changes over time. It shows how the angular momentum stays at a constant value and remains constant. This makes sense because the angular momentum is the cross-product between the position and velocity vector. With the following equations, we can prove this statement.

$$\frac{dh}{dt} = \frac{d}{dt}(\bar{r} \times \dot{\bar{r}}) \implies \frac{dh}{dt} = \bar{r} \times \frac{d\dot{\bar{r}}}{dt} \tag{1}$$

$$\frac{dh}{dt} = \bar{r} \times -\frac{\mu}{r^3}\bar{r} \implies \frac{dh}{dt} = 0 \tag{2}$$

So yes, these diagrams make physical sense, based upon the conservation laws of both Energy and Angular Momentum.

2 Orbital maneuvers and mission design

2.1 Reachability Analysis

Below defines a function that converts from the state vector containing the position and velocity vector in a 6×1 column to an array containing the six classical orbital elements at that point.

It is important to be in the right quadrant for specific values like Ω, ω, and θ. To ensure this, the script uses if statements to correctly identify when and how these values should be dealt with in order to coincide with convention.

```
def RV2COE(state_x, mu):

    #x,y,z,vx,vy,vz = state_x
    r_vec = state_x[0:3]
    v_vec = state_x[3:6]
    r_mag = norm(r_vec)
    v_mag = norm(v_vec)

    # Calculations for semi-major axis, specific
        angular momentum, and eccentricity
    a = r_mag / (2 - (r_mag*v_mag**2/mu))
```

```
    h = np.cross(r_vec, v_vec)
    h_mag = norm(h)
    e_vec = np.cross(v_vec, h) / mu - r_vec / r_mag
    e_mag= np.linalg.norm(e_vec)

    # Calulations for the inclination, node vector,
        and right ascension of ascending node
    i = np.arccos(h[2]/h_mag)
    n_vec = np.cross(k_hat, h)
    n_mag = norm(n_vec)
    n_hat = n_vec / n_mag

    # Using if statements to ensure values are in the
        right quadrant
    Omega_raan = np.arccos(n_hat[0])
    if n_hat[1] < 0:
        Omega_raan = 2*np.pi - Omega_raan

    omega = np.arccos(np.dot(n_vec, e_vec)/(n_mag *
        e_mag))
    if e_vec[2] < 0:
        omega = 2*np.pi - omega

    cos_theta = np.dot(r_vec, e_vec) / (r_mag * e_mag)
    cos_theta = np.clip(cos_theta, -1.0, 1.0) #
        ensures it's in valid domain
    theta = np.arccos(cos_theta)
    if np.dot(r_vec, v_vec) < 0:
        theta = 2*np.pi - theta

    # Return array
    return np.array([a, e_mag, i, Omega_raan, omega,
        theta])
```

Using this function, we can report the COE state for the initial condition. Inputting the vector $\bar{X}$, returns the following elements.

$$\textbf{At} \quad X_0\text{: COE} = \begin{bmatrix} a = 7.17813700\text{e}{+}03 & \text{km} \\ e = 7.00000000\text{e}{-}02 & \\ i = 1.67551608\text{e}{+}00 & \text{rad} \\ \Omega = 3.49065850\text{e}{-}01 & \text{rad} \\ \omega = 7.85398163\text{e}{-}01 & \text{rad} \\ \theta = 0\text{e}{+}00 & \text{rad} \end{bmatrix}$$

Next, we can define a rotation matrix that converts from the radius-transverse-normal frame to the ECI frame. As an input, it takes the state vector and outputs a rotation matrix that can be used as an operator.

```
def rotate_matrix(state_x):

    # Unpack position and velocity vectors
    r_vec = state_x[0:3]
    v_vec = state_x[3:6]

    # Perform calculations to find relevant vectors
    r_hat = r_vec/norm(r_vec)
    h_vec = np.cross(r_vec, v_vec)
    h_hat = h_vec / norm(h_vec)
    t_hat = np.cross(h_hat, r_hat)

    # Stack values to form a 3 x 3 matrix
    rotation = np.column_stack((r_hat, t_hat, h_hat))

    return rotation
```

With this rotation matrix now defined at every point along the orbit, it is now possible to define a function named "impulse". This function takes a rotation matrix, direction and an initial state as input parameters. It calculates the impulse in the direction of either radius, transverse or normal (which are calculated via 3-D basis vectors). Then, the rotation matrix is applied to the impulse to convert it from the RTN frame to the ECI frame. It adds this vector to the velocity components of the state and then calculates the orbital elements of this final state and the inputted initial state to find the difference between components.

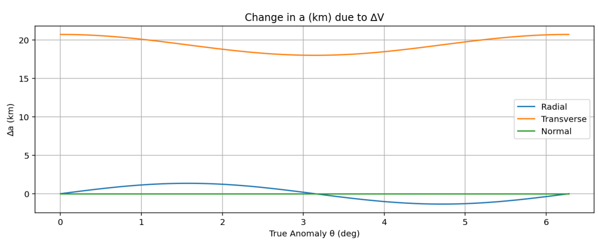
```
def impulse(r_matrix, direct, initial_state):

    # Get the impulse in the specified basis direction
    dv_ = np.dot(delta_v, direct)

    # Multiplye the impulse by rotation matrix at
        that point
    impulse_eci = r_matrix @ dv_

    state_final = initial_state.copy()

    # Change the velocity vector with the impulse
    state_final[3:] += impulse_eci

    # Find the before and after orbital elements
    oElements_initial = RV2COE(initial_state,
        mu_earth)
    oElements_final = RV2COE(state_final, mu_earth)

    # Find the difference in orbital elements at some
        theta value
    coe_diff = oElements_final - oElements_initial

    # Normalize angles into [-pi, pi]
    coe_diff[2:] = (coe_diff[2:] + np.pi) % (2 *
        np.pi) - np.pi

    return coe_diff, oElements_final
```

Using these combinations of functions we can apply an impulse in the three different directions and calculate the changes.

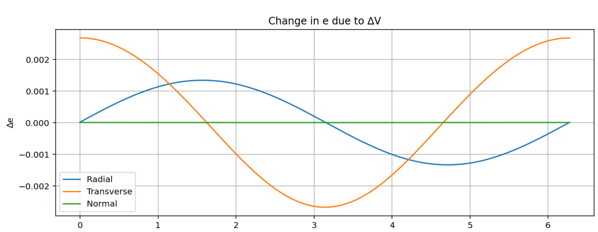
Element	$\Delta\hat{e}_r$	$\Delta\hat{e}_t$	$\Delta\hat{e}_n$
a (km)	0.012927	20.737377	0.012927
e	0.000013	0.002679	0.000002
i (rad)	0	0	0.000885
Ω (rad)	0	0	0.000890
ω (rad)	-0.019121	0	0.000093
θ (rad)	0.019121	0	0

Table 1: Change in Orbital Elements due to ΔV Impulses in RTN Directions on the initial state X_0

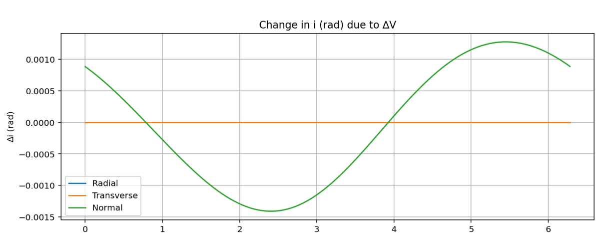
With these defined functions, it is also possible to examine the how the orbital elements change due to the impulses as a function of the true anomaly. The following graphs demonstrate how each orbital element changes depeneding on where on the orbit the impulse was applied.



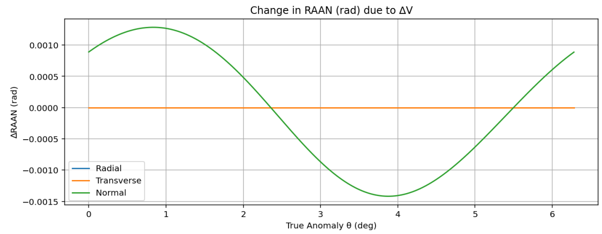
(a) Changes in the semi-major axis, a



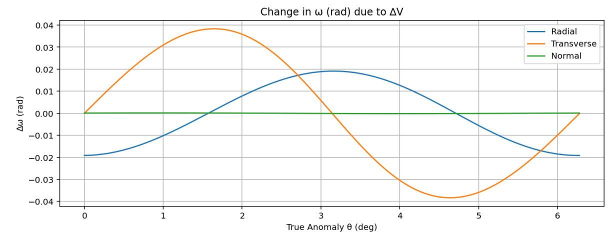
(b) Changes in the eccentricity, e



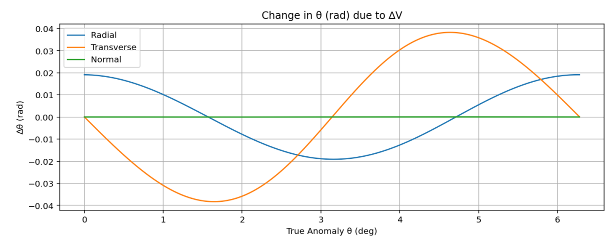
(c) Changes in the inclination, i



(d) Changes in the right accession of ascending nodes, Ω



(e) Changes in the argument of periapsis, ω



(f) Changes in the true anomaly, θ

Figure VII: Six graphs showing how the classical orbital elements change with impulses

Using the `find_peaks_cwt` function from the `scipy.signal` library in Python, the peaks of Δi are found to be -0.0014 and 0.0013 radians. These θ values correspond to the maximum values of Δi. are 2.36 and 5.46 radians. Understanding that the argument of latitude, u, is defined as $\theta + \omega$, and observing that at these theta values, $\omega = 0.79$ and 0.79 radians. Hence, $u = 3.145$ and 6.245 radians. These numbers represent the apogee and perigee respectively (which make sense because the perigee and apogee occur at π and 2π). Therefore, for maximum change to the inclination, one should provide a normal impulse at the perigee and apogee.

2.2 Lunar Mission Design

Now, we can analyse what happens when the apogee of the transfer orbit is increased past the moon's orbital radius. By increasing this distance, we are completing a non-tangential transfer, where the mission will have a lower transfer time but will arrive with some non-zero radial velocity. Using NumPy's matrix-oriented math in Python, calculations can be made swiftly by looking at the range of apogee values.

The following code uses the range of apogee values to calculate the semi-major axis, eccentricity, the true anomaly, and the semi-latus rectum, to ultimately return the radial and transverse components of the arrival velocity vector.

```
# Define the parking altitude and hence the radius of
    the parking orbit
parking_altitude = 220
r_p = parking_altitude + radius_earth

# Initialise array of apogee values from 1.1 to 1.4
r_a = r_mean*np.arange(1.1, 1.41, 0.01)

# Calculations for a, e, true_anomaly, p
a = 0.5*(r_p + r_a)
ecc = (r_a-r_p)/(r_a+r_p)
cos_theta_A = (a * (1 - ecc**2) / r_mean - 1) / ecc
theta_2 = np.arccos(cos_theta_A)
p =  a * (1 - ecc**2)

# Find the radial and transverse velocities using
    orbital equations
v_radial = np.sqrt(mu_earth / p) * ecc *
    np.sin(theta_2)
v_transverse = np.sqrt(mu_earth / p) * (1 + ecc *
    np.cos(theta_2))

# Find the apogee values normalized to the moon's
    orbital radius
normed_apogee = r_a/r_mean
```

Now we have the true anomaly, radial, and transverse velocities inside a matrix array for every apogee value normalized to the Moon's orbital radius. Plotting these components, we can see how these change as we increase the distance.

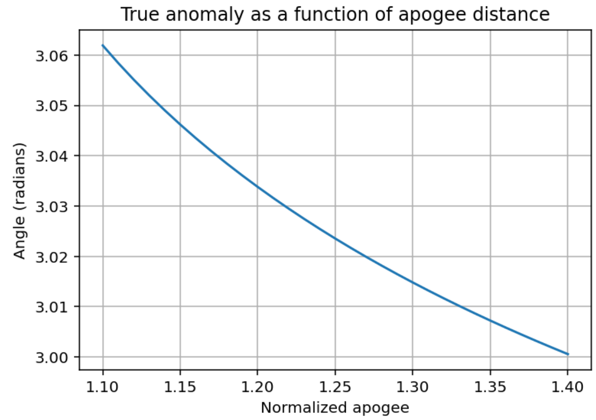


Figure VIII: Graph depicting how the true anomaly decreases as the apogee distance increases

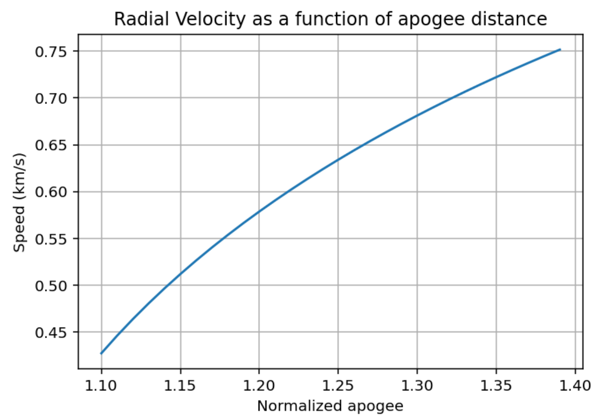


Figure IX: Graph depicting how the radial velocity increases as the apogee distance increases

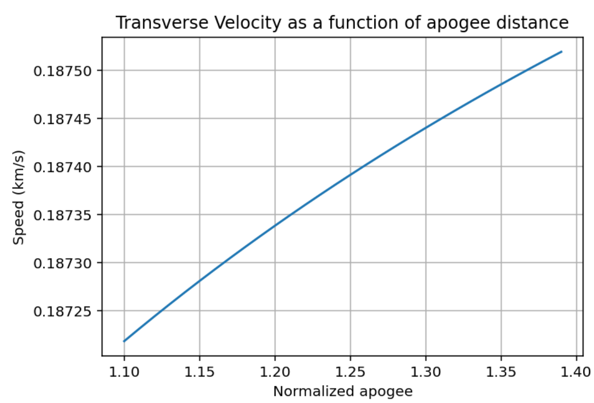


Figure X: Graph depicting how the transverse velocity increases as the apogee distance increases

The first plot shows how the true anomaly decreases as the apogee of orbit increases.

This means that the spacecraft is intercepting the Moon at an earlier point along its orbit. At a normalized apogee of 1, this represents a Hohman transfer so its true anomaly would be π; as we increase the apogee distance, this angle decreases.

These figures also illustrate that the radial component of velocity increases much faster than the transverse component. This is because the radial velocity is directly tied to the orbital energy. When the semi-major axis is significantly increased — to extend the trajectory beyond the Moon — most of the added energy contributes to raising the orbital energy and apogee height, which primarily affects the radial velocity. In contrast, any increase to the transverse velocity depends on how much angular momentum increases. Yet, even as the orbit becomes more energetic — due to a larger semi-major axis — the shape becomes more eccentric, meaning more of the motion is directed radially, and less is needed in the transverse direction to conserve angular momentum.

Next, we can observe how the apogee and transfer time are related. Writing some more code, we can examine this, again using NumPy's matrix operations.

```
# Find the transfer times
theta_e = 2*np.arctan(np.tan(theta_2/2) *
     ((1-ecc)/(1+ecc))**(0.5))

# Kepler's eqation
theta_m = theta_e - ecc*np.sin(theta_e)

# Using the time_orbit function
time_total = time_orbit(a, mu_earth)
delta_t = (time_total / (2 * np.pi)) * theta_m /
     days_convert
```

This Python script returns the transfer time for every apogee value using the time orbit equation, which we can plot as the following.

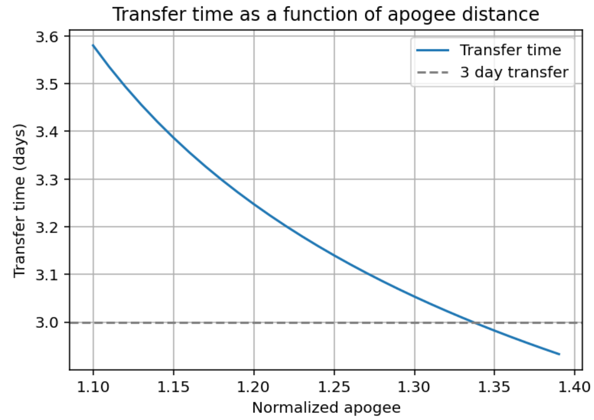


Figure XI: Graph showing how the transfer time decreases as the apogee value increases

This plot shows how the transfer time decreases as the apogee value increases. This makes sense because increasing the apogee point means the spacecraft is intercepting the Moon at an earlier stage of its orbit. There is a horizontal line corresponding to $t = 3$ days. We can look at the intercept of this line with the plot to see that for a transfer time of three days, an apogee value of $1.34 \cdot r_m$ is required.

Finally, we can compare the transfer time with the minimum required perilune altitude for a free return back to Earth. Assuming the Moon is moving in a circular orbit, this implies that it only has velocity components in the transverse direction. The following script uses this idea to calculate the hyperbolic quantities for a free return trajectory, ultimately returning the minimum required perilune altitude.

```
# Define the Moon's velocity, noting it is entirely
    in the transverse direction
v_moon = np.sqrt(mu_earth/r_mean)

# Vector equations
v_ir = v_radial
v_it = v_transverse-v_moon

# Define the incoming velocity in the moons frame
v_infminus = np.sqrt(v_radial**2 + v_it**2)

# Calculate the hyperbolic orbital values
quotient = -v_ir / v_it
delta_angle = np.arctan(quotient)
```

```
hyperbolic_e = (np.sin(delta_angle))**-1
r_perilune = (hyperbolic_e -
    1)*mu_moon/(v_infminus**2)

# Find the perilune altitude for a given transfer time
altitude_perilune = r_perilune - radius_moon
```

Using this code, the following plot can be produced identifying how the transfer time relates to the required perilune altitude:

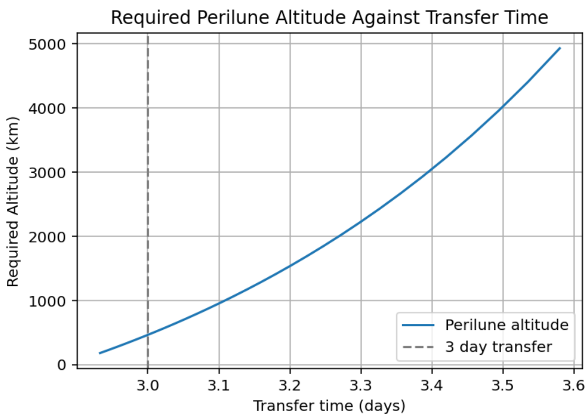
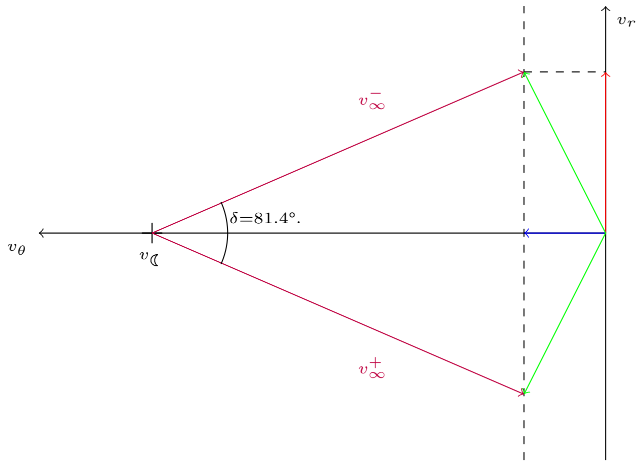


Figure XII: Required perilune altitude for some given transfer time to ensure correct trajectory for a return to Earth

This plot illustrates how for a three-day transfer, the required perilune altitude is 440.88 km. It further shows that the required perilune altitude decreases as transfer time increases. This make sense because as the transfer time decreases, the spacecraft arrives with a greater velocity. This means it needs to be closer to the Moon to be influenced more by the Moon's gravitational field in order for the greater energy change to be on the correct free return trajectory. This calculation lines up closely with the Apollo 11 mission, As noted in the discussion of Apollo free-return trajectories, the craft passes on the near side of the Moon at a radius of 2150 km (410 km above the surface) [1]. The discrepancy would be due to how we are not factoring real-world assumptions, such as that, in reality, the Moon would orbit in an elliptical fashion with some non-zero radial velocity. We can view the changes in velocities and angles using a Michielsen view.



This diagram represents the free return symmetric trajectory representative of a required perilune altitude of 440.88 km. Notably, we can see the angle being rotated by two.

Finally, it can be ideal to visualise this transfer and the free return to Earth. The following script calculates the orbits and transfers.

```
r_altitude = 220
r_p = radius_earth + r_altitude

# Setting the apogee value to the maxiumum
r_ap = 1.41 * r_mean
a = (r_p + r_ap) / 2
e = (r_ap - r_p) / (r_ap + r_p)

# Generate angles for plotting
theta = np.linspace(0, 2*np.pi, 1000)

# Parking orbit
x_parking = r_p * np.cos(theta)
y_parking = r_p * np.sin(theta)

# Moon orbit (circular)
x_moon = r_mean * np.cos(theta)
y_moon = r_mean * np.sin(theta)

# Find the true anomaly angle
nu_ = np.arccos((a * (1 - e**2) / r_mean - 1) / e)

# Find the angle
del_angle = nu_ - np.pi

# Arange an array of theta values
theta_t = np.linspace(0, -nu_, 1000)

# Define the transfer orbit
r_transfer = (a * (1 - e**2)) / (1 + e *
    np.cos(theta_t))
x_transfer = r_transfer * np.cos(theta_t)
y_transfer = r_transfer * np.sin(theta_t)
```

```
# Free-return trajectory noting a symmetric transfer
    by two delta
theta_rot = 2* del_angle
x_return = x_transfer * np.cos(theta_rot) -
    y_transfer * np.sin(theta_rot)
y_return = -x_transfer * np.sin(theta_rot) -
    y_transfer * np.cos(theta_rot)

# Define the full transfer
r_full_transfer = a * (1 - e**2) / (1 + e *
    np.cos(theta))
x_full_transfer = r_full_transfer * np.cos(theta)
y_full_transfer = r_full_transfer * np.sin(theta)
r_p = radius_earth + r_altitude

# Setting the apogee value to the maxiumum
r_ap = 1.41 * r_mean
a = (r_p + r_ap) / 2
e = (r_ap - r_p) / (r_ap + r_p)

# Generate angles for plotting
theta = np.linspace(0, 2*np.pi, 1000)

# Parking orbit
x_parking = r_p * np.cos(theta)
y_parking = r_p * np.sin(theta)

# Moon orbit (circular)
x_moon = r_mean * np.cos(theta)
y_moon = r_mean * np.sin(theta)

# Find the true anomaly angle
nu_  = np.arccos((a * (1 - e**2) / r_mean - 1) / e)

# Find the angle
del_angle = nu_ - np.pi

# Arange an array of theta values
theta_t = np.linspace(0, -nu_, 1000)

# Define the transfer orbit
r_transfer = (a * (1 - e**2)) / (1 + e *
    np.cos(theta_t))
x_transfer = r_transfer * np.cos(theta_t)
y_transfer = r_transfer * np.sin(theta_t)

# Free-return trajectory noting a symmetric transfer
    by two delta
theta_rot = 2* del_angle
x_return = x_transfer * np.cos(theta_rot) -
    y_transfer * np.sin(theta_rot)
y_return = -x_transfer * np.sin(theta_rot) -
    y_transfer * np.cos(theta_rot)

# Define the full transfer
r_full_transfer = a * (1 - e**2) / (1 + e *
    np.cos(theta))
x_full_transfer = r_full_transfer * np.cos(theta)
y_full_transfer = r_full_transfer * np.sin(theta)
```

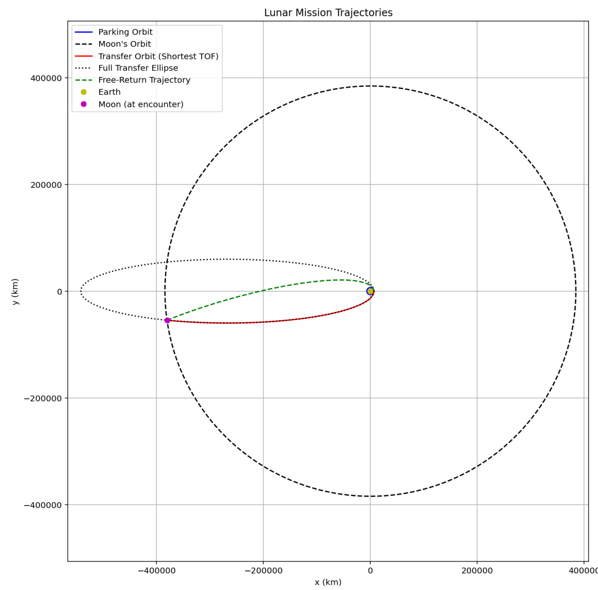


Figure XIII: Graph showing the transfer orbit as well as the free return

With this graph, we can see the cusp at the Moon and how this exact perilune altitude gives a free return directly back to Earth.

References

[1] Wikipedia contributors. Free-return trajectory — Wikipedia, the free encyclopedia, 2024. Accessed: 2025-05-02. See section: *Apollo free-return trajectories*, paragraph discussing a 2150 km radius (410 km above lunar surface).

3 Appendix

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 25 22:08:35 2025

@author: alexa
"""



#Import Python files
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp
from scipy import signal

# Toggle plots
show_plots = True

#Assignment Data
mu_sun = 1.3271244*10**11
mu_earth = 3.986*10**5
radius_earth = 6378.14
J_2_earth = 1082.63*10**-6
mu_moon = 4902.8
radius_moon = 1737.4
r_mean = 384400


'''
Functions
'''

pi = np.pi


def norm(vec):
    return np.linalg.norm(vec)


def radians(deg):
    rads = deg * (pi/180)
    return rads


def get_mean_anomaly(del_t, M_old, a):
    nu_t = (mu_sun / (a**3))**0.5
    mean_anomaly_t = M_old + nu_t * (del_t)

    return mean_anomaly_t

def time_orbit(a, ):
    T = 2 * np.pi * np.sqrt(a**3 / )
    time = T
    return time

def rotation_matrix(i, Omega, omega):
    cos_O, sin_O = np.cos(Omega), np.sin(Omega)
    cos_i, sin_i = np.cos(i), np.sin(i)
    cos_w, sin_w = np.cos(omega), np.sin(omega)

    R = np.array([
        [cos_O * cos_w - sin_O * sin_w * cos_i, -cos_O * sin_w - sin_O * cos_w * cos_i, sin_O * sin_i],
        [sin_O * cos_w + cos_O * sin_w * cos_i, -sin_O * sin_w + cos_O * cos_w * cos_i, -cos_O * sin_i],
        [sin_w * sin_i, cos_w * sin_i, cos_i]
```

```python
    ])

    return R

'''
Newton-Rasphon Method to return the eccentric anomaly to a certain tolerance
'''

#1.1.1 ----------------------------------------------------------------------


def Kepler(e, M, tol = 1e-12, max_i = 1000):

    # Guess the solution is similar to theta_M
    E = M

    for i in range(max_i):

        # Define function in terms of f(E) = 0
        f_E = E - e * np.sin(E) - M

        # Define the derivative
        f_prime = 1 - e * np.cos(E)

        del_E = f_E / f_prime

        # Calculate the new eccentric anomaly
        E_new = E - del_E

        # If the error is within some tolerance return theta
        if np.abs(del_E) < tol:
            theta = 2*np.arctan(np.tan(E_new/2) * ((1+e)/(1-e))**(0.5))
            return theta

        # Else restart the for loop with a new value
        E = E_new





# ObjData 1
E_ae0 = [2460705.5, 1.495988209443421E+08, 1.669829008180246E-02, radians(3.248050135173038E-03),
         radians(1.744712892867145E+02), radians(2.884490093009512E+02), radians(2.621190445180298E+01)]

# ObjData 2
A_ae0 = [2460705.5, 3.764419202360106E+08, 6.616071771587672E-01, radians(3.408286057191753),
         radians(2.713674649188756E+02), radians(1.343644678984687E+02), radians(1.693237490356061E+01)]



#1.1.2 ----------------------------------------------------------------------

# Using the Kepler function to find the true anomaly given an eccentricity and mean anomaly
true_a_t_0 = Kepler(A_ae0[2], A_ae0[6])

# Finding the true anomaly for t_0 + 100 days
mean_a_t_100 = get_mean_anomaly(100*(3600*24), A_ae0[6], A_ae0[1])
true_a_t_100 = Kepler(A_ae0[2], mean_a_t_100)

print(f"The true anomaly at t = t is \n{true_a_t_0:.6f} radians")
print(f"The true anomaly at t = t + 100 is \n{true_a_t_100:.6f} radians")


# Creating two Obj arrays to represent the asteroid at the two diffent times
Obj2_t0 = A_ae0.copy()
Obj2_t0[6] = true_a_t_0 # Substituding updated true anomaly
Obj2_t0 = Obj2_t0[1:]
```

```
Obj2_t100 = A_ae0.copy()
Obj2_t100[6] = true_a_t_100 # Substituding updated true anomaly
Obj2_t100 = Obj2_t100[1:]

#1.1.3 ----------------------------------------------------------------------

t_0 = 2460705.5*(3600*24)

# Function to convert from COE to RV
def COE2RV(arr, mu):

    # Orbital calculations for a, h, and r
    a, e, i, Omega, omega, theta_var = arr[0:6]
    h = np.sqrt(mu * a * (1 - e**2))
    r = a*(1-(e**2))/(1 + e*np.cos(theta_var))

    # Using orbital equations to find the position and velocity arrays
    arr_r = np.array([r*np.cos(theta_var), r*np.sin(theta_var), 0])
    arr_v = (mu/h)* np.array([-np.sin(theta_var), e + np.cos(theta_var), 0])

    #Rotate position and velocity from perifocal to the inertial frame using the transfomration matrix
    R_matrix = rotation_matrix(i, Omega, omega)

    # Perform matrix operations
    r_ijk = R_matrix @ arr_r
    v_ijk = R_matrix @ arr_v

    return r_ijk, v_ijk




state_vector_0 = np.array(COE2RV(Obj2_t0, mu_sun))
labels = ['x (km)', 'y (km)', 'z (km)', 'vx (km/s)', 'vy (km/s)', 'vz (km/s)']

print("State Vector at t0:")
for label, val in zip(labels, state_vector_0.flatten()):
    print(f"{label:<8}: {float(val): .6e}")

state_vector_100 = np.array(COE2RV(Obj2_t100, mu_sun))
print("\nState Vector at t0 + 100:")
for label, val in zip(labels, state_vector_100.flatten()):
    print(f"{label:<8}: {float(val): .6e}")


t_0_days = 2460705.5
days_convert = 3600*24
#1.1.4 ----------------------------------------------------------------------

# Function to return the position and velocity arrays at some time t
def Ephemeris(t, OBJdata, mu):

    time, a, e, i, Omega, omega, mean_anomaly = OBJdata[0:7]

    # Find the mean motion
    nu_t = (mu / (a**3))**0.5

    # Find the mean anomaly at some time t
    t = t - t_0_days*days_convert
    mean_anomaly_t = mean_anomaly + nu_t * (t)

    h = np.sqrt(mu * a * (1 - e**2))
    theta_var = Kepler(e, mean_anomaly_t)
    r = a*(1-(e**2))/(1 + e*np.cos(theta_var))

    arr_r = np.array([r*np.cos(theta_var), r*np.sin(theta_var), 0])
```

```
    arr_v = (mu/h)* np.array([-np.sin(theta_var), e + np.cos(theta_var), 0])

    # Perform matrix operations on position and velocity arrays
    R_matrix = rotation_matrix(i, Omega, omega)
    r_ijk = R_matrix @ arr_r
    v_ijk = R_matrix @ arr_v

    return r_ijk, v_ijk



years_shown_i = 1
t_array = days_convert*np.arange(t_0_days,t_0_days+years_shown_i*365, 1)

# Define arrays in the proper size to hold position and velocity data
x_earth = np.zeros((6,len(t_array)))
x_asteroid = np.zeros((6,len(t_array)))

# Add in six elements of data for every theta value in the orbit
for r in range(len(t_array)):
    x_earth[0:6, r] = np.hstack(Ephemeris(t_array[r], E_ae0, mu_sun))
    x_asteroid[0:6, r] = np.hstack(Ephemeris(t_array[r], A_ae0, mu_sun))


t_day_array = np.arange(0, years_shown_i*365, 1)

# Calculate the time of one Earth orbit converting from seconds to days
time_Earth = time_orbit(E_ae0[1], mu_sun)/days_convert

# Find the percent of one Earth orbit
orbital_percent_E = (t_day_array / time_Earth)

# Plot the position magnitude of Earth and the asteroid for one Earth year
plt.plot(orbital_percent_E,[norm(x_earth[0:3, t]) for t in range(len(t_array))], label="Earth position",
     color='b')
plt.plot(orbital_percent_E, [norm(x_asteroid[0:3, t]) for t in range(len(t_array))], label="Asteroid
     position", color='r')
plt.legend()
plt.grid(True)
plt.title("Magnitude of the Position of Earth and 2024 YR4 Asteroid")
plt.xlabel("Percent of Earth's Orbit (%)")
plt.ylabel("Position Vector (km)")
if show_plots:
    plt.show()
else:
    plt.close()

# Plot the velocity magnitude of Earth and the asteroid for one Earth year
plt.figure()
plt.plot(orbital_percent_E,[norm(x_earth[3:, t]) for t in range(len(t_array))], label="Earth velocity",
     color='b')
plt.plot(orbital_percent_E, [norm(x_asteroid[3:, t]) for t in range(len(t_array))], label="Asteroid
     velocity", color='r')
plt.legend()
plt.grid(True)
plt.xlabel("Percent of Earth's Orbit (%)")
plt.ylabel("Velocity Vector (km/s)")
plt.title("Magnitude of the Velocity of Earth and 2024 YR4 Asteroid")
if show_plots:
    plt.show()
else:
    plt.close()

#1.1.5 ----------------------------------------------------------------------

years_shown = 10
t_total = days_convert*np.arange(t_0_days, t_0_days + years_shown*365, 1)
```

```
normed_diff = []
# Find the normalized difference between the two bodies for every time step
for t in t_total:
    normed_diff.append(norm(Ephemeris(t,E_ae0, mu_sun)[0] - Ephemeris(t,A_ae0, mu_sun)[0]))

# Plotting the separation
plt.figure()
plt.plot(t_total/(days_convert*365), normed_diff)
plt.grid(True)
plt.title("Absolute distance separation of Earth and 2024 YR4")
plt.xlabel("Time in J2000 (years)")
plt.ylabel("Distance separation (km)")
if show_plots:
    plt.show()
else:
    plt.close()

#1.2.1 ----------------------------------------------------------------------

# Initialise data
x0, y0, z0 = 4604.49276873138, 1150.81472538679, 4694.55079634563 # km
vx0, vy0, vz0 = -5.10903235110107 , -2.48824074138143 ,5.62098648967432 # km/s

# Initial state vector
X0 = [x0, y0, z0, vx0, vy0, vz0]

#1.2.2 ----------------------------------------------------------------------

# Function to define the differential equation returning the derivatives
def TBP_ECI(t, state_X, mu):
    # Unpack state vector
    x, y, z, vx, vy, vz = state_X

    # Compute radius
    r = np.sqrt(x**2 + y**2 + z**2)

    # Acceleration components
    ax, ay, az = -mu * x / r**3, -mu * y / r**3, -mu * z / r**3
    return [vx, vy, vz, ax, ay, az]



#1.2.3 ----------------------------------------------------------------------

# Now begin integrating the function

# Initial distance from Earth's center (km)
r0 = np.linalg.norm(X0[:3])

# Initial speed (km/s)
v0 = np.linalg.norm(X0[3:])

# Semi-major axis (km)
a = 1 / (2 / r0 - v0**2 / mu_earth)

# Orbital period (s)
T = 2 * np.pi * np.sqrt(a**3 / mu_earth)

# Set integration time span for two orbital periods
t_start = 0
t_end = 2 * T # Two orbital periods
time_step = 10 # Output every 10 seconds
t_eval = np.arange(t_start, t_end, time_step)

# Solve the system using solve_ivp with specified tolerances using RK45
solution = solve_ivp(
    TBP_ECI, (t_start, t_end), X0, t_eval=t_eval, method='RK45',
```

```python
    args=(mu_earth,), rtol=1e-12, atol=1e-12)


# Extract components
x, y, z = solution.y[0], solution.y[1], solution.y[2]
vx, vy, vz = solution.y[3], solution.y[4], solution.y[5]

# Compute position and velocity magnitude
r = np.sqrt(x**2 + y**2 + z**2)
v_lin = np.sqrt(vx**2 + vy**2 + vz**2)

# 3D Trajectory Plot
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')


#Plot Earth as a sphere
earth_radius = 6378 # km (mean Earth radius)
u, v = np.mgrid[0:2*np.pi:50j, 0:np.pi:25j]
X_earth = earth_radius * np.cos(u) * np.sin(v)
Y_earth = earth_radius * np.sin(u) * np.sin(v)
Z_earth = earth_radius * np.cos(v)
ax.plot_surface(X_earth, Y_earth, Z_earth, color='b', alpha=0.3)


# Plot the orbit
# Dashed section
ax.plot(x[:460], y[:460], z[:460], color='r', label='Behind Earth',linestyle='dashed', alpha = 0.15)

# Dashed segment (between 460 and 890)
ax.plot(x[460:891], y[460:891], z[460:891], color='r', linewidth=1.5, label='Visible Orbit')
ax.plot(x[891:], y[891:], z[891:], color='r',linestyle='dashed', alpha = 0.15)


# Labels and title
ax.set_xlabel("X (km)")
ax.set_ylabel("Y (km)")
ax.set_zlabel("Z (km)")
ax.set_title("Orbital Trajectory in ECI Frame")
ax.legend()

if show_plots:
    plt.show()
else:
    plt.close()


#1.2.4 ----------------------------------------------------------------------

# Plot the energies and angular momentum
plt.figure()

# Find the kinetic, potential, and total energies using orbital equations
KE = 0.5 * v_lin**2
PE = -mu_earth / r
E_total = KE + PE

# Extract solutions
x, y, z = solution.y[0], solution.y[1], solution.y[2]
vx, vy, vz = solution.y[3], solution.y[4], solution.y[5]

# Compute Specific Angular Momentum
h_array = np.sqrt(((y * vz) - (z * vy))**2 + ((z * vx) - (x * vz))**2 + ((x * vy) - (y * vx))**2)

# Convert time to hours
time_hours = solution.t / 3600

# Plot Specific Energies
```

```
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.plot(time_hours, KE, label="Kinetic Energy", color='r')
plt.plot(time_hours, PE, label="Potential Energy", color='b')
plt.plot(time_hours, E_total, label="Total Energy", color='k', linestyle='dashed')
plt.xlabel("Time (hours)")
plt.ylabel("Specific Energy (km/s)")
plt.title("Specific Energies Over Time")
plt.legend()
plt.grid()
if show_plots:
    plt.show()
else:
    plt.close()

# Plot Specific Angular Momentum
plt.figure()
plt.plot(time_hours, h_array.round(2),'r', label="Specific Angular Momentum")
plt.xlabel("Time (hours)")
plt.ylabel("Angular Momentum (km/s)")
plt.title("Specific Angular Momentum Over Time")
plt.legend()
plt.grid()

if show_plots:
    plt.show()
else:
    plt.close()


#2.1.1 ----------------------------------------------------------------------

# Define the perpindicular to the equitorial plane
k_hat = np.array([0,0,1])

# Define function to onvert the state of a spacecraft from ECI position and velocity to COE
def RV2COE(state_x, mu):

    #x,y,z,vx,vy,vz = state_x
    r_vec = state_x[0:3]
    v_vec = state_x[3:6]
    r_mag = norm(r_vec)
    v_mag = norm(v_vec)

    # Calculations for semi-major axis, specific angular momentum, and eccentricity
    a = r_mag / (2 - (r_mag*v_mag**2/mu))
    h = np.cross(r_vec, v_vec)
    h_mag = norm(h)
    e_vec = np.cross(v_vec, h) / mu - r_vec / r_mag
    e_mag= np.linalg.norm(e_vec)

    # Calulations for the inclination, node vector, and right ascension of ascending node
    i = np.arccos(h[2]/h_mag)
    n_vec = np.cross(k_hat, h)
    n_mag = norm(n_vec)
    n_hat = n_vec / n_mag

    # Using if statements to ensure values are in the right quadrant
    Omega_raan = np.arccos(n_hat[0])
    if n_hat[1] < 0:
        Omega_raan = 2*np.pi - Omega_raan

    omega = np.arccos(np.dot(n_vec, e_vec)/(n_mag * e_mag))
    if e_vec[2] < 0:
        omega = 2*np.pi - omega

    cos_theta = np.dot(r_vec, e_vec) / (r_mag * e_mag)
```

```
    cos_theta = np.clip(cos_theta, -1.0, 1.0) # ensures it's in valid domain
    theta = np.arccos(cos_theta)
    if np.dot(r_vec, v_vec) < 0:
        theta = 2*np.pi - theta

    theta = (theta + np.pi) % (2 * np.pi) - np.pi

    # Return array
    return np.array([a, e_mag, i, Omega_raan, omega, theta])


state_zero = RV2COE(X0, mu_earth)
# Labels for COEs: a, e, i, RAAN, ,
coe_labels = ['a (km)', 'e', 'i (rad)', 'RAAN (rad)', ' (rad)', ' (rad)']
print("The COE of  X  are")
for i in range(len(state_zero)):
    print(f"{coe_labels[i]}: {state_zero[i]: .6f}")


#2.1.2 ----------------------------------------------------------------------

# Define function convert from the RTN frame to the ECI frame
def rotate_matrix(state_x):

    # Unpack position and velocity vectors
    r_vec = state_x[0:3]
    v_vec = state_x[3:6]

    # Perform calculations to find relevant vectors
    r_hat = r_vec/norm(r_vec)
    h_vec = np.cross(r_vec, v_vec)
    h_hat = h_vec / norm(h_vec)
    t_hat = np.cross(h_hat, r_hat)

    # Stack values to form a 3 x 3 matrix
    rotation = np.column_stack((r_hat, t_hat, h_hat))

    return rotation

#2.1.3 ----------------------------------------------------------------------

# Define impulse magnitude
delta_v = 0.01


# Define function to take a rotation matrix, basis direction, and initial state
# To find the final coe values and report the difference
def impulse(r_matrix, direct, initial_state):

    # Get the impulse in the right direction
    dv_ = np.dot(delta_v, direct)

    # Multiplye the impulse by rotation matrix at that point
    impulse_eci = r_matrix @ dv_

    state_final = initial_state.copy()

    # Change the velocity vector with the impulse
    state_final[3:] += impulse_eci

    # Find the before and after orbital elements
    oElements_initial = RV2COE(initial_state, mu_earth)
    oElements_final = RV2COE(state_final, mu_earth)

    # Find the difference in orbital elements at some theta value
    coe_diff = oElements_final - oElements_initial

    # Normalize angles into [-pi, pi]
```

```python
    coe_diff[2:] = (coe_diff[2:] + np.pi) % (2 * np.pi) - np.pi


    return coe_diff, oElements_final




# Reporting the different states for the radial, transverse and normal impulses at that point
# Unit vectors
er = [1, 0, 0] # radial
et = [0, 1, 0] # transverse
en = [0, 0, 1] # normal

# Apply impulses and collect results
delta_er = impulse(rotate_matrix(X0), er, X0)[0]
delta_et = impulse(rotate_matrix(X0), et, X0)[0]
delta_en = impulse(rotate_matrix(X0), en, X0)[0]

print("\nCOE due to unit impulses (in appropriate units):\n")
print("{:<12} {:>12} {:>12} {:>12}".format("Element", "er", "et", "en"))
print("-" * 50)
for label, d1, d2, d3 in zip(coe_labels, delta_er, delta_et, delta_en):
    print("{:<12} {:>12.6f} {:>12.6f} {:>12.6f}".format(label, d1, d2, d3))


#2.1.4 ----------------------------------------------------------------------

# Define the inital coe elements for the given RV values
oElements_initial = RV2COE(X0, mu_earth)

# Arange an array of theta values around the orbit
theta_array = np.arange(0,2*np.pi, 0.01)

# Initialise empty lists for each of the elements in each direction
delta_elements_radial = []
delta_elements_transverse = []
delta_elements_normal = []
all_elements_N = np.zeros((6,len(theta_array)))

i = 0

# Foe each value of theta append the differences of coe vales to the correct list
for theta in theta_array:

    coe = oElements_initial.copy()

    # Set the true anomaly to each theta value
    coe[5] = theta
    state_RV = np.concatenate(COE2RV(coe, mu_earth))

    # Find the required rotation matrix for the state
    rotation = rotate_matrix(state_RV)

    # Append the required values in the right way
    delta_elements_radial.append(impulse(rotation, er, state_RV)[0])
    delta_elements_transverse.append(impulse(rotation, et, state_RV)[0])
    delta_elements_normal.append(impulse(rotation, en, state_RV)[0])
    all_elements_N[0:6, i] = impulse(rotation, en, state_RV)[1]

    i = i + 1



# Convert to NumPy arrays
deltaR_array = np.array(delta_elements_radial)
deltaT_array = np.array(delta_elements_transverse)
deltaN_array = np.array(delta_elements_normal)
```

```
delta_i = deltaN_array[:, 2]

absdel_i = np.abs(delta_i)

# Define an arbitary peak width tolerance
peak_widths = 75

# Find the indicies of the peaks
peak_indices = signal.find_peaks_cwt(absdel_i, peak_widths)

# Using these indicies find which theta values these occur at and hence
# Find the argument of periapsis
max_delta_i = [delta_i[peak_indices[0]], delta_i[peak_indices[1]]]
theta_imax = [theta_array[peak_indices[0]], theta_array[peak_indices[1]]]
w_imax = [all_elements_N[4,peak_indices[0]], all_elements_N[4,peak_indices[1]]]
u_val = np.array(theta_imax) + np.array(w_imax)

print("\n")
print(f"Maximum  i : {max_delta_i[0]:.4f} and {max_delta_i[1]:.4f} radians")
print(f"Occurs at true anomaly  = {theta_imax[0]:.2f} and {theta_imax[1]:.2f} radians")
print(f"Here,  = {w_imax[0]:.2f} and {w_imax[0]:.2f} radians")
print(f"These represent a u value of {u_val[0]:.3f} and {u_val[1]:.3f}")
print("Hence, maximum impact from impulse occurs at the preigee and apogee in the normal direction")
labels = ['a (km)', 'e', 'i (rad)', 'RAAN (rad)', ' (rad)', ' (rad)']

# Pring each plot
if show_plots:
    for i in range(6):
        plt.figure(figsize=(10, 4))
        plt.plot(theta_array, deltaR_array[:, i], label='Radial')
        plt.plot(theta_array, deltaT_array[:, i], label='Transverse')
        plt.plot(theta_array, deltaN_array[:, i], label='Normal')
        plt.title(f'Change in {labels[i]} due to V')
        plt.xlabel('True Anomaly  (deg)')
        plt.ylabel(f' {labels[i]}')
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()
else:
    plt.close()

 #2.2.1 ---------------------------------------------------------------------

# Define the parking altitude and hence the radius of the parking orbit
parking_altitude = 220
r_p = parking_altitude + radius_earth

# Initialise array of apogee values from 1.1 to 1.4
r_a = r_mean*np.arange(1.1, 1.41, 0.01)

# Calculations for a, e, true_anomaly, p
a = 0.5*(r_p + r_a)
ecc = (r_a-r_p)/(r_a+r_p)
cos_theta_A = (a * (1 - ecc**2) / r_mean - 1) / ecc
theta_2 = np.arccos(cos_theta_A)
p =  a * (1 - ecc**2)

# Find the radial and transverse velocities using orbital equations
v_radial = np.sqrt(mu_earth / p) * ecc * np.sin(theta_2)
v_transverse = np.sqrt(mu_earth / p) * (1 + ecc * np.cos(theta_2))

# Find the apogee values normalized to the moon's orbital radius
normed_apogee = r_a/r_mean

# Plot figures
plt.figure()
```

```python
plt.plot(normed_apogee, theta_2, label="true anomaly")
plt.grid(True)
plt.xlabel("Normalized apogee")
plt.ylabel("Angle (radians)")
plt.title("True anomaly as a function of apogee distance")
if show_plots:
    plt.show()
else:
    plt.close()

plt.figure()
plt.plot(normed_apogee, v_radial, label="radial")
plt.grid(True)
plt.xlabel("Normalized apogee")
plt.ylabel("Speed (km/s)")
plt.title("Radial Velocity as a function of apogee distance")
if show_plots:
    plt.show()
else:
    plt.close()

plt.figure()
plt.plot(normed_apogee, v_transverse, label="transverse")
plt.grid(True)
plt.xlabel("Normalized apogee")
plt.ylabel("Speed (km/s)")
plt.title("Transverse Velocity as a function of apogee distance")
if show_plots:
    plt.show()
else:
    plt.close()

#2.2.2 ----------------------------------------------------------------------

# Find the transfer times using the time_orbit function
theta_e = 2*np.arctan(np.tan(theta_2/2) * ((1-ecc)/(1+ecc))**(0.5))
theta_m = theta_e - ecc*np.sin(theta_e)
time_total = time_orbit(a, mu_earth)
delta_t = (time_total / (2 * np.pi)) * theta_m / days_convert
print("The index is at 24 for a {} day transfer".format(delta_t[24]))

plt.figure()
plt.plot(normed_apogee, delta_t, label = "Transfer time")
plt.grid()
plt.axhline(3, linestyle='--', color='grey', label = "3 day transfer")
plt.legend()
plt.title("Transfer time as a function of apogee distance")
plt.xlabel("Normalized apogee")
plt.ylabel("Transfer time (days)")
if show_plots:
    plt.show()
else:
    plt.close()


#2.2.3 ----------------------------------------------------------------------


# Define the moon's velocity noting it is entirely in the transverse direction
v_moon = np.sqrt(mu_earth/r_mean)

# Vector equations
v_ir = v_radial
v_it = v_transverse-v_moon

# Define the incoming velocity in the moons frame
v_infminus = np.sqrt(v_radial**2 + v_it**2)
```

```python
# Calculate the hyperbolic orbital values
quotient = -v_ir / v_it
delta_angle = np.arctan(quotient)
hyperbolic_e = (np.sin(delta_angle))**-1
r_perilune = (hyperbolic_e - 1)*mu_moon/(v_infminus**2)

# Find the perilune altitude for a given transfer time
altitude_perilune = r_perilune - radius_moon

print("The required perilune altitude for a three-day transfer is {}".format(altitude_perilune[24]))

plt.plot(delta_t, altitude_perilune, label = "Perilune altitude")
plt.axvline(3, linestyle='--', color='grey', label = "3 day transfer")
plt.grid(True)
plt.legend()
plt.xlabel("Transfer time (days)")
plt.ylabel("Required Altitude (km)")
plt.title("Required Perilune Altitude Against Transfer Time")
if show_plots:
    plt.show()
else:
    plt.close()


# Bonus ------------------------------------------------------------------

r_altitude = 220
r_p = radius_earth + r_altitude

# Setting the apogee value to the maxiumum
r_ap = 1.41 * r_mean
a = (r_p + r_ap) / 2
e = (r_ap - r_p) / (r_ap + r_p)

# Generate angles for plotting
theta = np.linspace(0, 2*np.pi, 1000)

# Parking orbit
x_parking = r_p * np.cos(theta)
y_parking = r_p * np.sin(theta)

# Moon orbit (circular)
x_moon = r_mean * np.cos(theta)
y_moon = r_mean * np.sin(theta)

# Find the true anomaly angle
nu_  = np.arccos((a * (1 - e**2) / r_mean - 1) / e)

# Find the angle
del_angle = nu_ - np.pi

# Arange an array of theta values
theta_t = np.linspace(0, -nu_, 1000)

# Define the transfer orbit
r_transfer = (a * (1 - e**2)) / (1 + e * np.cos(theta_t))
x_transfer = r_transfer * np.cos(theta_t)
y_transfer = r_transfer * np.sin(theta_t)

# Free-return trajectory noting a symmetric transfer by two delta
theta_rot = 2* del_angle
x_return = x_transfer * np.cos(theta_rot) - y_transfer * np.sin(theta_rot)
y_return = -x_transfer * np.sin(theta_rot) - y_transfer * np.cos(theta_rot)

# Define the full transfer
r_full_transfer = a * (1 - e**2) / (1 + e * np.cos(theta))
x_full_transfer = r_full_transfer * np.cos(theta)
y_full_transfer = r_full_transfer * np.sin(theta)
```

```python
# Plotting
plt.figure(figsize=(10, 10))
plt.plot(x_parking, y_parking, 'b', label="Parking Orbit")
plt.plot(x_moon, y_moon, 'k--', label="Moon's Orbit")
plt.plot(x_transfer, y_transfer, 'r', label="Transfer Orbit (Shortest TOF)")
plt.plot(x_full_transfer, y_full_transfer, 'k:', label='Full Transfer Ellipse')
plt.plot(x_return, y_return, 'g--', label="Free-Return Trajectory")

# Earth and Moon positions
plt.plot(0, 0, 'yo', label="Earth")
plt.plot(-r_mean*np.cos(del_angle), r_mean*np.sin(del_angle), 'mo', label="Moon (at encounter)")

plt.axis("equal")
plt.title("Lunar Mission Trajectories")
plt.xlabel("x (km)")
plt.ylabel("y (km)")
plt.legend()
plt.grid(True)
plt.tight_layout()
if show_plots:
    plt.show()
else:
    plt.close()
```