Spacecraft Mission

Alex Freeman

April 28, 2025

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

1 Orbital Propagation

1.1 Solving Kepler's Equation

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

```python
def Kepler(e, M, tol = 1e-12, max_i = 1000):
    E = M                            # Guess solution
    for i in range(max_i):
        f_E = E - e * np.sin(E) - M  # Define the function in terms of f(E) = 0
         f_prime = 1 - e * np.cos(E) # Derive the function in terms of E
        del_E = f_E / f_prime
        E_new = E - del_E            # Calculate the new eccentric anamoly
        if np.abs(del_E) < tol:      # If the value is within the set tolerance
            theta = 2*np.arctan(np.tan(E_new/2) * ((1+e)/(1-e))**(0.5))
            return theta             # Return true anamoly
        E = E_new
```

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

```python
trueAnamoly_asteroidt_0 = Kepler(A_ae0[2], A_ae0[6])
meanAnamolyt_100 = get_mean_anamoly(100*(3600*24), A_ae0[6], A_ae0[1])
trueAnamoly_asteroidt_100 = Kepler(A_ae0[2], meanAnamolyt_100)
```

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

III) Now I have created 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 defind matricies in the appendix.

```python
def COE2RV(arr, mu):
    a, e, i, Omega, omega, theta_var = arr[0:6]
```

```python
    h = np.sqrt(mu * a * (1 - e**2))
    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])

    # Rotate position and velocity from perifocal to inertial frame using the
    # transfomration matrix
    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 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**.

At t_0:

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

At $t_0 + 100$:

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

IV) Next, I have written a function called "Ephemeris". It returns the position and velocity at some time t.

```python
def Ephemeris(t, OBJdata, mu):
    time, a, e, i, Omega, omega, mean_anamoly = OBJdata[0:7]
    nu_t = (mu / (a**3))**0.5
    t = t - t_0_days*days_convert
    mean_anamoly_t = mean_anamoly + nu_t * (t)

    h = np.sqrt(mu * a * (1 - e**2))
    theta_var = Kepler(e, mean_anamoly_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])
    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.

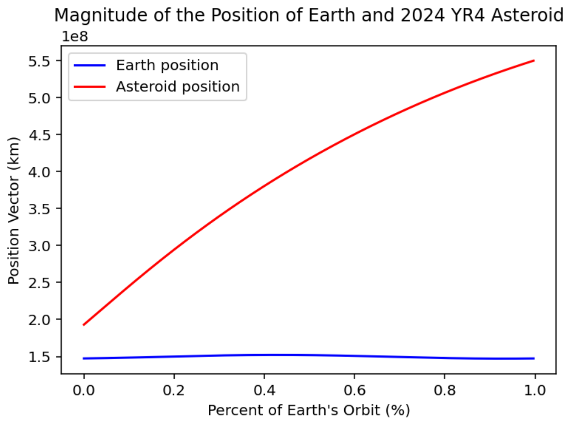


Figure 1: Position vectors of Earth and the Asteroid's for a full Earth orbital period

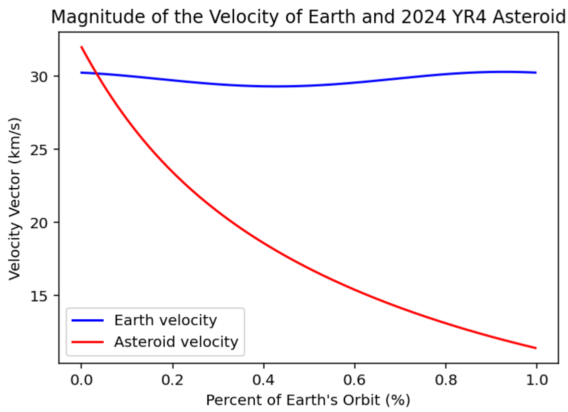


Figure 2: Velocity vectors of Earth and the Asteroid's for a full Earth orbital period

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

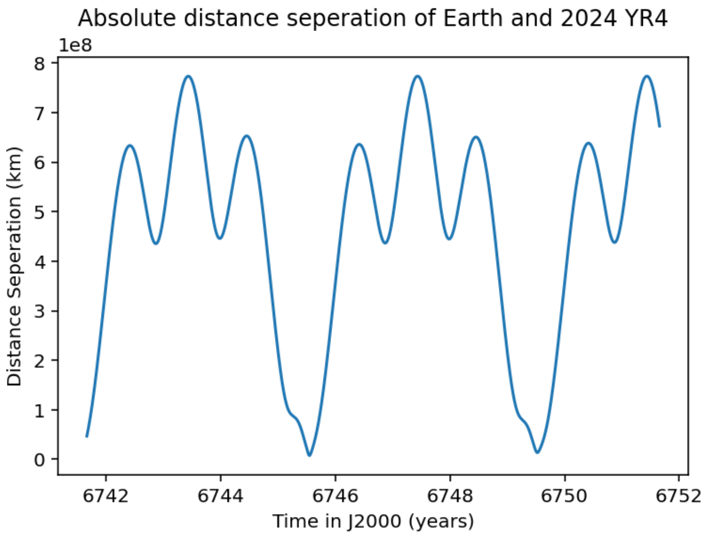


Figure 3: Distance between Earth and 2024 YR4 asteroid in kilometers

We can see how the distance is sinusoidal in nature and repeats in an oscillatory fashion. This graph shows three distinct peaks, where the middle one is the greatest. We can examine the sydonic period of the two bides by examing 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 in their seperation.

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:

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

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

```
def TBP_ECI(t, state_X, mu):
    x, y, z, vx, vy, vz = state_X # Unpack state vector
    r = np.sqrt(x**2 + y**2 + z**2) # Compute radius
    ax, ay, az = -mu * x / r**3, -mu * y / r**3, -mu * z / r**3 # Acceleration
        components
    return [vx, vy, vz, ax, ay, az] # Return derivatives
```

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.

```
r0 = np.linalg.norm(X0[:3]) # Initial distance from Earth's center (km)
v0 = np.linalg.norm(X0[3:]) # Initial speed (km/s)
a = 1/(2/r0 - v0**2/mu_earth) # Semi-major axis (km)
T = 2 * np.pi * np.sqrt(a**3/mu_earth) # Orbital period (s)

# 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 strict tolerances
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]
```

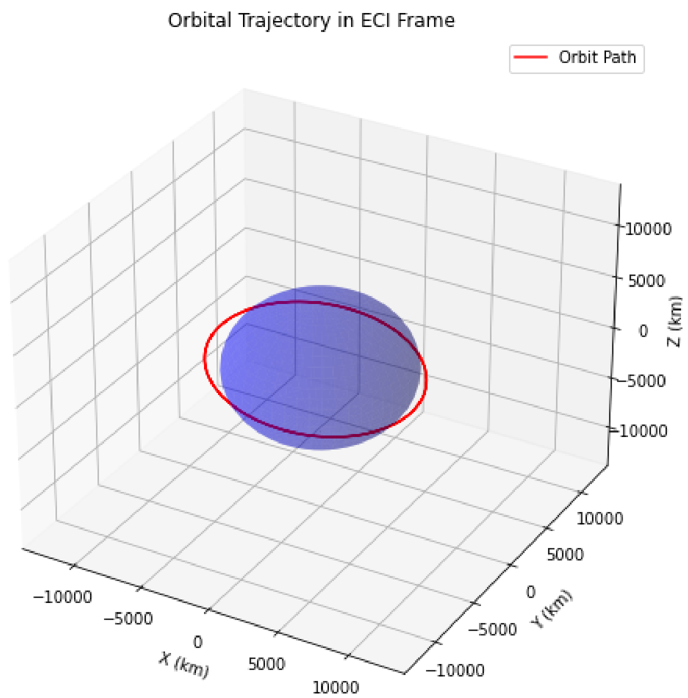


Figure 4: Orbital trajectory in ECI frame with initial conditions given from X_0

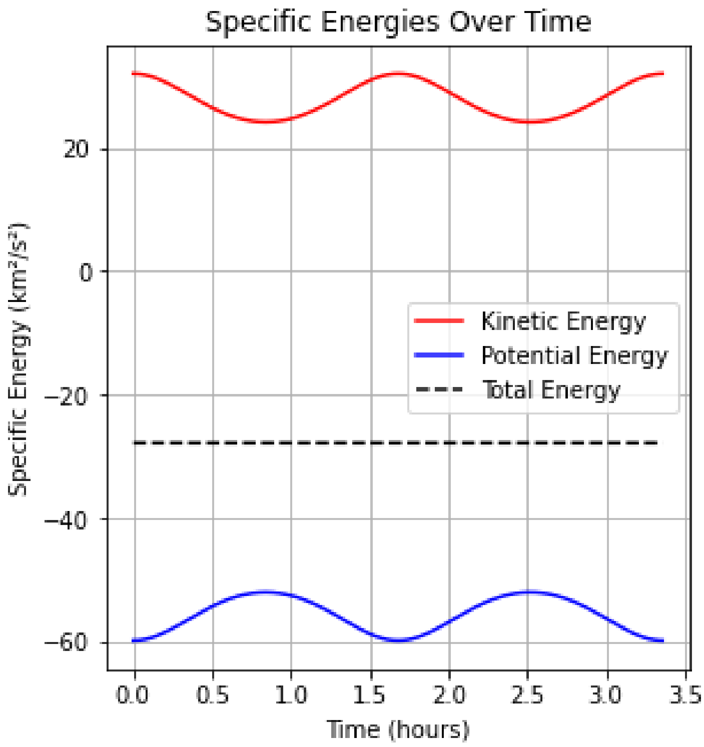


Figure 5: How Kinetic, Potential, and Total energies change throughout the orbit

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.

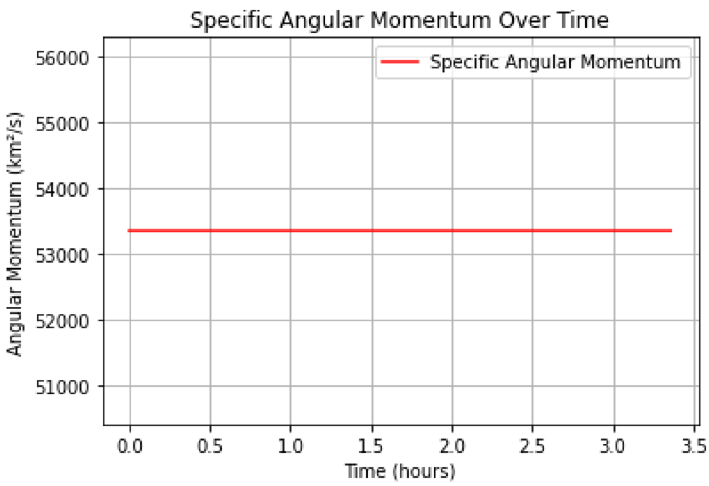


Figure 6: Graph shows the change in the angular momentum over the orbit is zero

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 defined as the cross product between the position and velocity vector.

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

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

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

```
def RV2COE(state_x, mu):
    r_vec = state_x[0:3]
    v_vec = state_x[3:6]
    r_mag = norm(r_vec)
    v_mag = norm(v_vec)

    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 = norm(e_vec)
    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
    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)
    theta = np.arccos(cos_theta)
    if np.dot(r_vec, v_vec) < 0:
        theta = 2*np.pi - theta

    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.

$$\textbf{At} \quad X_0\text{: COE} = \begin{bmatrix} a = 7.17813700\text{e}{+}03 \\ e = 7.00000000\text{e}{-}02 \\ i = 1.67551608\text{e}{+}00 \\ \Omega = 3.49065850\text{e}{-}01 \\ \omega = 7.85398163\text{e}{-}01 \\ \theta = 6.28318529\text{e}{+}00 \end{bmatrix}$$

```python
def rotate_matrix(state_x):
    r_vec = state_x[0:3]
    v_vec = state_x[3:6]

    r_hat = r_vec/(norm(r_vec))


    h_vec = np.cross(r_vec, v_vec)
    h_hat = h_vec / np.linalg.norm(h_vec)
    t_hat = np.cross(h_hat, r_hat)

    rotation = np.column_stack((r_hat, t_hat, h_hat))

    return rotation
```

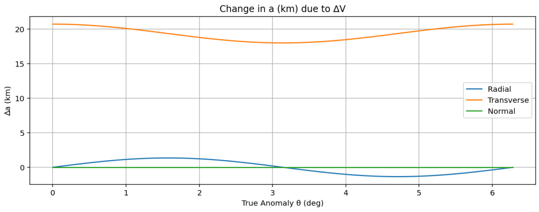
With this rotation matrix now defined at every point along the orbit, I can define a function named "impulse". This function takes a rotation matrix, direciton and an initial state as input parameters. It calculates the impulse in the direction of either, radius, transverse or normal (which are calculated 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.

```python
def impulse(r_matrix, direct, initial_state):
    dv_ = np.dot(delta_v, direct)

    impulse_eci = r_matrix @ dv_

    state_final = initial_state.copy()
    state_final[3:] += impulse_eci
    oElements_initial = RV2COE(initial_state, mu_earth)
    oElements_final = RV2COE(state_final, mu_earth)

    coe_diff = oElements_final - oElements_initial

    return coe_diff, oElements_final
```

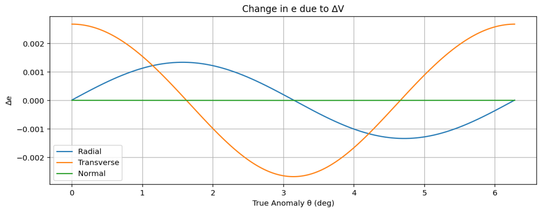
Using these combinations of functions we can apply an impulse in the three different directions and calculate the changes.

Impulse Type	Δa	Δe	Δi	$\Delta\Omega$	$\Delta\omega$	ΔM
Radial	0.012927	1.28e−5	0	0	-0.0191214	-6.26406
Transverse	20.7374	0.00267900	0	0	0	0.0000000149012
Normal	0.012927	1.67e−6	8.85e−4	0.000889606	0.0000933805	0.0000000149012

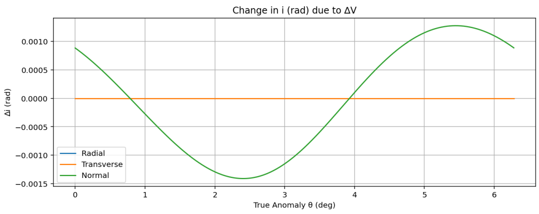
Table 1: Changes in orbital elements after applying impulses in radial, transverse, and normal directions.



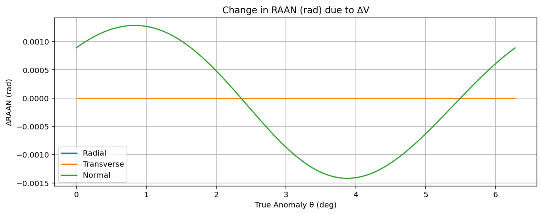
(a) Graph 1



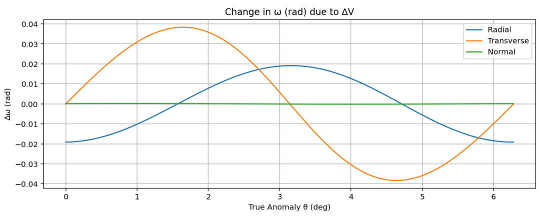
(b) Graph 2



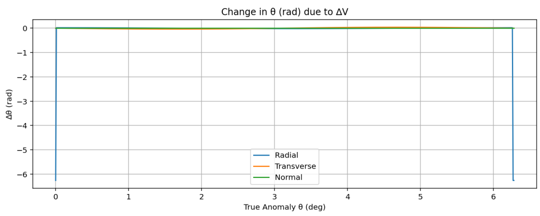
(c) Graph 3



(d) Graph 4



(e) Graph 5



(f) Graph 6

Figure 7: Six graphs arranged in three rows, two per row.

Using these graphs we can examine which θ values maximise Δi. Using the Signal library apart of Python and its corrosponding find_peaks_cwt function, the peaks of Δi are -0.0014 and 0.0013 radians. These corrosponding θ values 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.

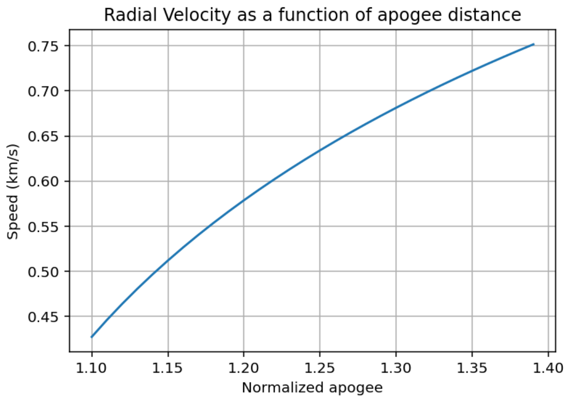


Figure 8: Graph depicting how the radial velocity increases as the apogee distance increases

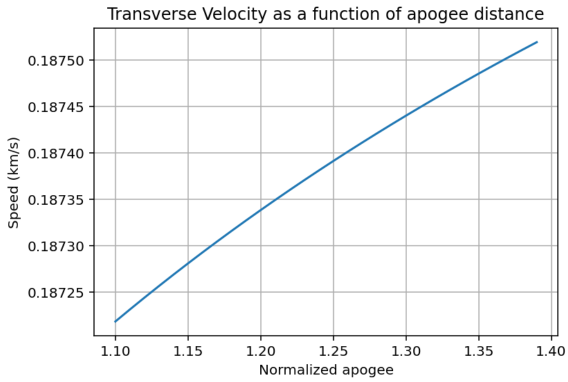


Figure 9: Graph depicting how the transverse velocity increases as the apogee distance increases

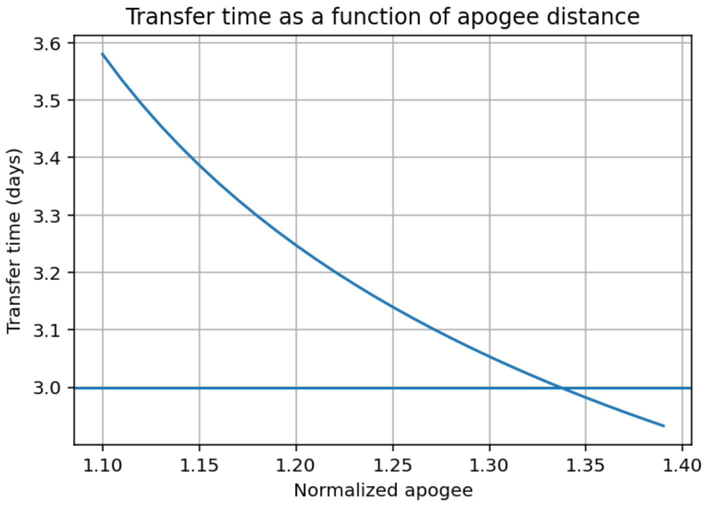


Figure 10: Distance between Earth and 2024 YR4 asteroid in kilometers

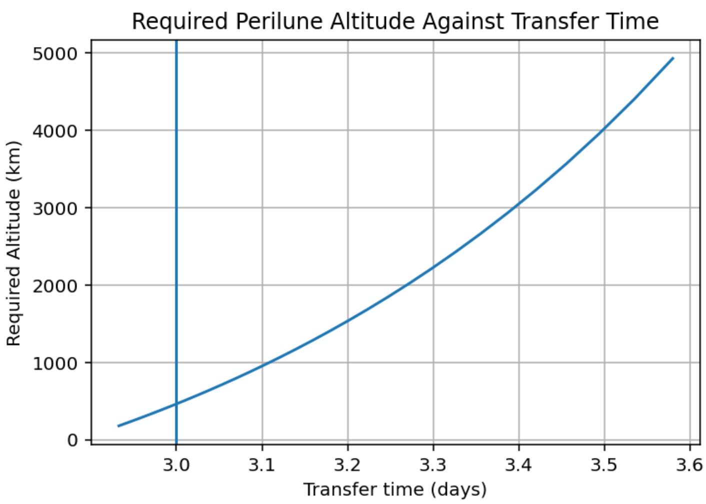


Figure 11: Distance between Earth and 2024 YR4 asteroid in kilometers

3 Appendix

```python
# -*- 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 mpl_toolkits.mplot3d import Axes3D
from scipy.integrate import solve_ivp
from scipy import signal

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
'''

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

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


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

    return mean_anamoly_t
```

```python
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]
    ])

    return R

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

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


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


    for i in range(max_i):

        f_E = E - e * np.sin(E) - M
        f_prime = 1 - e * np.cos(E)

        del_E = f_E / f_prime


        E_new = E - del_E
        if np.abs(del_E) < tol:
            theta = 2*np.arctan(np.tan(E_new/2) * ((1+e)/(1-e))**(0.5))
            return theta
        E = E_new
```

```
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)]


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 ----------------------------------------------------------------------


trueAnamoly_asteroidt_0 = Kepler(A_ae0[2], A_ae0[6])
meanAnamolyt_100 = get_mean_anamoly(100*(3600*24), A_ae0[6], A_ae0[1])
trueAnamoly_asteroidt_100 = Kepler(A_ae0[2], meanAnamolyt_100)

# print(trueAnamoly_asteroidt_0)
# print(trueAnamoly_asteroidt_100)



Obj2_t0 = A_ae0.copy()
Obj2_t0[6] = trueAnamoly_asteroidt_0
Obj2_t0 = Obj2_t0[1:]


Obj2_t100 = A_ae0.copy()
Obj2_t100[6] = trueAnamoly_asteroidt_100
Obj2_t100 = Obj2_t100[1:]

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

t_0 = 2460705.5*(3600*24)

def COE2RV(arr, mu):
    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))

    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 inertial frame using the
       transfomration matrix
    R_matrix = rotation_matrix(i, Omega, omega)

    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))
print(state_vector_0) # t = t0



state_vector_100 = np.array(COE2RV(Obj2_t100, mu_sun))
print(state_vector_100) # t = t0 + 100

# print('\n')
# print(state_vector_0 - state_vector_100)


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

def Ephemeris(t, OBJdata, mu):

    time, a, e, i, Omega, omega, mean_anamoly = OBJdata[0:7]
    nu_t = (mu / (a**3))**0.5
    t = t - t_0_days*days_convert
    mean_anamoly_t = mean_anamoly + nu_t * (t)

    h = np.sqrt(mu * a * (1 - e**2))
    theta_var = Kepler(e, mean_anamoly_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])
    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_0_days = 2460705

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


x_earth = np.zeros((6,len(t_array)))
x_asteroid = np.zeros((6,len(t_array)))
```

```
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))



time_Earth = time_orbit(E_ae0[1], mu_sun)
orbital_percent_E = (t_array / time_Earth)


plt.plot(orbital_percent_E,[norm(x_earth[0:3, t]) for t in
    range(len(t_array))], label="|r| - Earth", color='b')
plt.plot(orbital_percent_E, [norm(x_asteroid[0:3, t]) for t in
    range(len(t_array))], label="|r| - Asteroid ", color='r')
plt.legend()


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

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


years_shown = 10
t_total = days_convert*np.arange(0,years_shown*365, 1)



normed_diff = []
for t in t_total:

    normed_diff.append(norm(Ephemeris(t,E_ae0, mu_sun)[0] - Ephemeris(t,A_ae0,
        mu_sun)[0]))


plt.figure()
plt.plot(t_total*(10/t_total[-1]), normed_diff)
plt.title("Absolute distance seperation of Earth and 2024 YR4")
plt.xlabel("Time (years)")
plt.ylabel("Distance Seperation (km)")
if show_plots:
    plt.show()
else:
    plt.close()
```

```python
#1.2.1 ----------------------------------------------------------------------

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

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

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

def TBP_ECI(t, state_X, mu):
    x, y, z, vx, vy, vz = state_X # Unpack state vector
    r = np.sqrt(x**2 + y**2 + z**2) # Compute radius
    ax, ay, az = -mu * x / r**3, -mu * y / r**3, -mu * z / r**3 # Acceleration
        components
    return [vx, vy, vz, ax, ay, az] # Return derivatives



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

r0 = np.linalg.norm(X0[:3]) # Initial distance from Earth's center (km)
v0 = np.linalg.norm(X0[3:]) # Initial speed (km/s)
a = 1 / (2 / r0 - v0**2 / mu_earth) # Semi-major axis (km)
T = 2 * np.pi * np.sqrt(a**3 / mu_earth) # Orbital period (s)

# 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 strict tolerances
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]

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

# Compute speed


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
ax.plot(x, y, z, label="Orbit Path", color='r')

# 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()

# Set limits to give a clear view
ax.set_xlim([-2*r0, 2*r0])
ax.set_ylim([-2*r0, 2*r0])
ax.set_zlim([-2*r0, 2*r0])

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


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


plt.figure()

KE = 0.5 * v_lin**2
PE = -mu_earth / r
E_total = KE + PE


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 (km/s)
h_array = np.sqrt(((y * vz) - (z * vy))**2 + ((z * vx) - (x * vz))**2 + ((x *
    vy) - (y * vx))**2)

# Convert time to hours for better readability
```

```
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 ----------------------------------------------------------------------

k_hat = np.array([0,0,1])


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)

    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)

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



#print(RV2COE(X0, mu_earth))

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

def rotate_matrix(state_x):
    r_vec = state_x[0:3]
    v_vec = state_x[3:6]

    r_hat = r_vec/(norm(r_vec))


    h_vec = np.cross(r_vec, v_vec)
    h_hat = h_vec / np.linalg.norm(h_vec) # N (Normal) direction
    t_hat = np.cross(h_hat, r_hat)

    rotation = np.column_stack((r_hat, t_hat, h_hat))
```

```python
    return rotation

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

delta_v = 0.01
rotation_transform = rotate_matrix(X0)


def impulse(r_matrix, direct, initial_state):

    dv_ = np.dot(delta_v, direct)

    impulse_eci = r_matrix @ dv_

    state_final = initial_state.copy()
    state_final[3:] += impulse_eci
    oElements_initial = RV2COE(initial_state, mu_earth)
    oElements_final = RV2COE(state_final, mu_earth)

    coe_diff = oElements_final - oElements_initial

    return coe_diff, oElements_final



# '''
# Reporting the different states for the radial, transverse and normal
    impulses at that point
# '''

er = [1,0,0]
et = [0,1,0]
en = [0,0,1]
# print(impulse(rotate_matrix(X0), er, X0)[0])
# print(impulse(rotate_matrix(X0), et, X0)[0])
# print(impulse(rotate_matrix(X0), en, X0)[0])


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

oElements_initial = RV2COE(X0, mu_earth)
theta_array = np.arange(0,2*np.pi, 0.01)

delta_elements_radial = []
delta_elements_transverse = []
delta_elements_normal = []

all_elements_N = np.zeros((6,len(theta_array)))
i = 0
for theta in theta_array:
```

```
    coe = oElements_initial.copy()
    coe[5] = theta
    state_RV = np.concatenate(COE2RV(coe, mu_earth))

    rotation = rotate_matrix(state_RV)


    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

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)
peak_widths = 75

peak_indices = signal.find_peaks_cwt(absdel_i, peak_widths)

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(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)']


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()

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

parking_altitude = 220
parking_radius = parking_altitude + radius_earth



radius_apogee = r_mean*np.arange(1.1, 2, 0.01)

semi_major_axis = 0.5*(parking_radius + radius_apogee)
#print(semi_major_axis)

transfer_e = (radius_apogee-parking_radius)/(radius_apogee+parking_radius)


cos_theta_A = (semi_major_axis * (1 - transfer_e**2) / r_mean - 1) /
    transfer_e


#cos_theta_A = np.clip(cos_theta_A, -1.0, 1.0)


theta_2 = np.arccos(cos_theta_A)


semi_latus_rectum = semi_major_axis * (1 - transfer_e**2)

v_radial = np.sqrt(mu_earth / semi_latus_rectum) * transfer_e *
    np.sin(theta_2)
v_transverse = np.sqrt(mu_earth / semi_latus_rectum) * (1 + transfer_e *
    np.cos(theta_2))

normed_apogee = radius_apogee/r_mean

if show_plots:
    plt.plot(normed_apogee, v_radial, label="radial")
    plt.legend()
    plt.show()
    plt.figure()
    plt.plot(normed_apogee, v_transverse, label="transverse")
    plt.legend()
    plt.show()

#2.2.2 ----------------------------------------------------------------------
```

```
theta_eccentric = 2*np.arctan(np.tan(theta_2/2) *
    ((1-transfer_e)/(1+transfer_e))**(0.5))
theta_mean = theta_eccentric - transfer_e*np.sin(theta_eccentric)
time_total = time_orbit(semi_major_axis, mu_earth)

delta_t = (time_total / (2 * np.pi)) * theta_mean / days_convert
if show_plots:
    plt.plot(normed_apogee, delta_t)
    plt.axhline(3)


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



v_moon = np.sqrt(mu_earth/r_mean)

v_ir = v_radial
v_it = v_transverse-v_moon
v_inf = np.sqrt(v_radial**2 + v_it**2)


delta_angle = 2* np.arctan(np.abs(v_it),v_ir)
#print(delta_angle)

hyperbolic_e = (np.sin(0.5*delta_angle))**-1
#print(hyperbolic_e)

r_perilune = (hyperbolic_e - 1)*mu_moon/(v_inf**2)

#print(r_perilune)
altitude_perilune = r_perilune - radius_moon

plt.plot(delta_t, altitude_perilune)

# If the radial velocity is flipped while the transverse velocity remains
    unchanged
# Then the total turning angle, delta, is twice the angle between v_inf and
    the moon
# in the radial direction
```