Assignment 9

Pierce Jackson pmj0008@auburn.edu
903713347

1. Problem 1

Consider an Earth orbit with the following values for its classical orbital elements: perigee altitude, $hp = 5000km$, apogee altitude, $ha = 10000km$, inclination, $i = 45°$, right-ascension of the ascending node, $\Omega = 45°$, argument of perigee, $\omega = 45°$, true anomaly, $\theta = 10°$. Develop a code in MATLAB (I did this in Python because Dr. Taheri said working in Python is acceptable) for propagating the two-body differential equations of motion in the presence of perturbations due to the second zonal harmonic of the Earth.

To do this, we must break down the two-body equation of motion

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

into two differential equations with only one derivative in each:

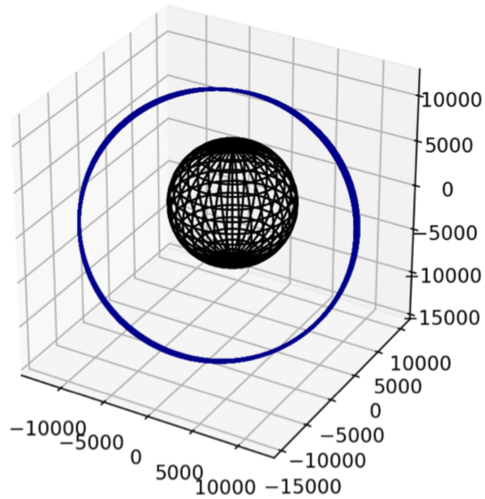
$$\frac{d\mathbf{r_i}}{dt} = \mathbf{v_i} \tag{2}$$

$$\frac{d\mathbf{v_i}}{dt} = \frac{-\mu}{r^3}\mathbf{r_i} \tag{3}$$

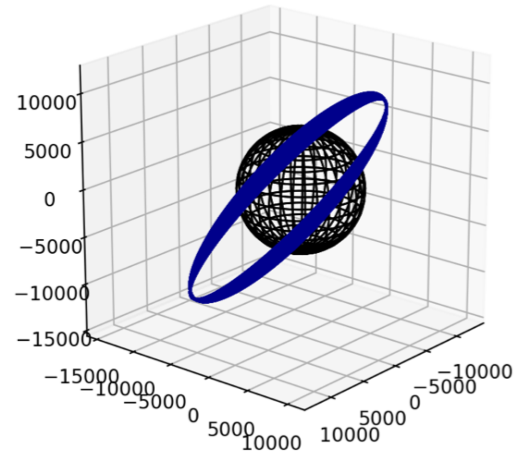
where $\mathbf{r_i}$ is the position vector of body i (i = 1, 2), $\mathbf{v_i}$ is the velocity vector of body i, and r is the norm of the vectors $(\mathbf{r_2} - \mathbf{r_1})$. These two differential equations can then be solved via scipy's *odeint* function when given the 12 initial conditions (3 for body 1's x, y, and z position, 3 for body 2's position, 3 for body 1's x, y, and z velocities, and body 2's velocities) and a time to integrate over; in our case 100 times the orbital period T.

Before these equations can be integrated, however, we must obtain the state vectors $\mathbf{r}$ and $\mathbf{v}$ for the satellite (the Earth is assumed to be stationary, so it's state vectors = 0). They can be obtained via algorithm 4.5 on page 218 of the textbook. Once the state vectors in the geocentric equatorial frame have been calculated, we can must add J2 acceleration onto the satellite's motion. Finally, we can solve eq. 2 and 3 with Numpy's *odeint* function and plot the results.

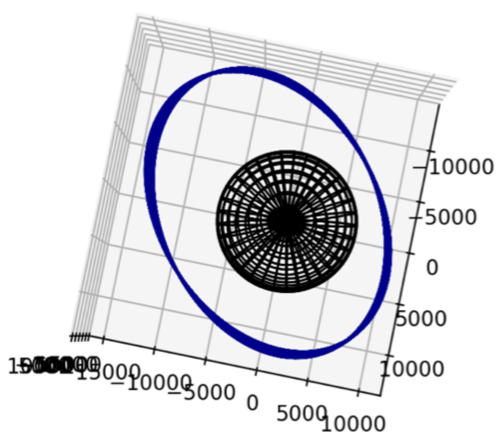
Presented below are figures of the orbit produced by my code. The propagation time is $100 * T$ and the initial classical orbital elements are those given in the problem:



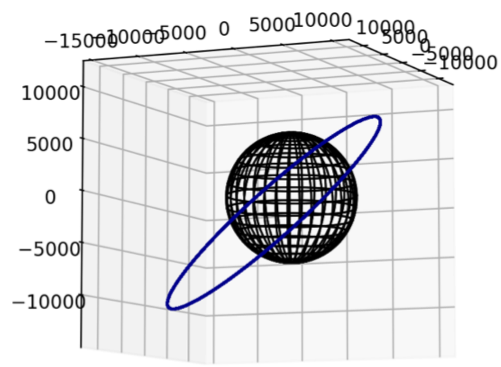
(i) View from above the orbit; J2 included



(ii) Edge on view of orbit. J2 Perturbation is seen as the thickening of the orbit line as it crosses the face of the sphere/Earth



(iii) Polar view of orbit; J2 included



(iv) The same orbit ***without*** J2 perturbations - still propagated by 100 * T. Notice no deviation in the orbit with time, unlike the other three figures which incorporate J2

1.1 Part a

Next, I provide the time evolution of the right-ascension of the ascending node, Ω, argument of perigee, ω, and true anomaly, θ, inclination, i, eccentricity, e, and semimajor axis a.

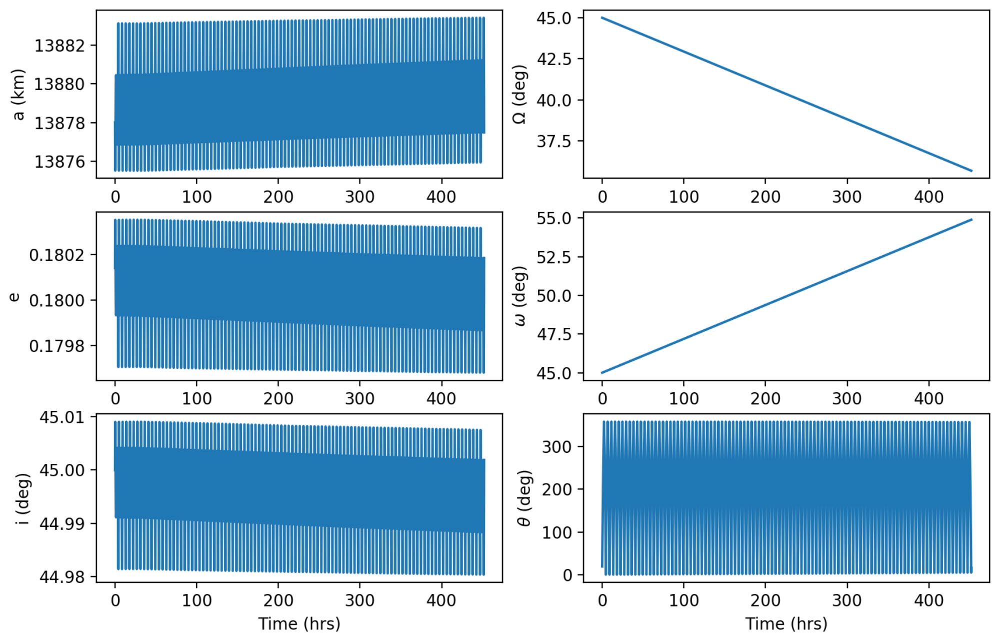


Figure 2: Time evolution of right-ascension of the ascending node, argument of perigee, and true anomaly, inclination, eccentricity, and semimajor axis.

These classical orbital elements can be calculated fairly using the time history of the state vectors (as calculated with *odeint*) and various orbital equations we have learned.

1.2 Part B

The average time rate of change of the right ascension of the ascending node is

$$\dot{\Omega} = -0.495°/day = 0.495° \text{ per day to the west} \tag{4}$$

The average time rate of change of the argument of perigee is

$$\dot{\omega} = 0.5253°/day \text{ to the east} \tag{5}$$

My code will be attached at the end of the document.

2. Problem 2

2.1 Part a

2.1.1 Question 1

The ISS is on an orbit with an inclination of 51.64°because this is the minimum inclination which Russians can launch Soyuz (safely) into orbit. Russia's Launch site is at Baikonur Cosmodrome which has a latitude of 45.6°thus, launching (most efficiently) from here puts spacecrafts in a 45.6°inclined orbit. However, dropping boosters at this inclination would result in them impacting mainland China, so to avoid this, the minimum inclination is pushed up to 51.6°. This inclination also allows the ISS to pass over much of Earth's surface over multiple orbits - about 75% of Earth's surface is traversed by the ISS.

2.1.2 Question 2

According to the video, the Soyuz's targeted parking orbit is about 220 km above the surface of the earth.

2.2 Part b

2.2.1 Question 1

Two docking options exist: Automatic docking where the docking process is controlled by Soyuz's on board computer - this is typically what is done; the second option is a manual docking where the commander of the Soyuz controls translation and rotation of the craft until it mates with the ISS.

2.2.2 Question 2

The purpose of the phasing orbit is to decrease the phasing angle between the Soyuz and the ISS. Because the phasing orbit is lower than the ISS orbit, the Soyuz will have a greater velocity than the ISS and can catch up to the ISS or until the required phasing angle is met.

2.2.3 Question 3

A bi-elliptic transfer is used instead of a Hohmann transfer because a bi-elliptic transfer will allow the Soyuz to reach the correct orbital altitude near the ISS along with exactly the required speed to meet with the ISS.

2.2.4 Question 4

The side-burn is used to change the Soyuz's orbital plane slightly, making a collision with the ISS impossible.

3. PYTHON CODE WRITTEN FOR PROBLEM 1

```
import numpy as np
import scipy as sci
import scipy.integrate  # ode solver solve_ivp(function, t_span, y0) : tspan is interval of integration
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# from mpl_toolkits.basemap import Basemap

# Define Classical Orbital Elements and other variables
Re = 6378  # radius of Earth (km)
u0 = 398600  # Standard Gravitational Parameter in km^3/s^2
hp = 5000  # Altitude of Perigee (km)
ha = 10000  # Altitude of Apogee (km)
```

```
perigee = Re + hp  # Perigee radius
apogee = Re + ha  # Apogee radius
a = (apogee + perigee) / 2  # Semi-major Axis
# print(a)
i = np.deg2rad(45)  # Inclination in radians (value in function in degrees)
raan = np.deg2rad(45)  # Right-Ascension of the Ascending Node in radians (value in function in degrees)
w = np.deg2rad(45)  # Argument of Perigee in radians (value in function in degrees)
ta = np.deg2rad(10)  # True Anomaly in radians (value in function in degrees)
e = (apogee - perigee) / (apogee + perigee)  # Eccentricity of orbit in degrees
v_perigee = np.sqrt(2 * ((u0 / perigee) - (u0 / (2 * a))))  # Velcoity of spacecraft at perigee in km/s; from conservation of energy equation
h = perigee * v_perigee  # specific angular momentum of spacecraft in km^2/s
T = ((2 * np.pi) / np.sqrt(u0)) * a ** (3.0 / 2.0)  # period in seconds
T_hrs = T / 3600.0
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Gravitational Parameters
G = 6.67408e-20  # Gravitational Constant in km^3/(kg*s^2)
Me = 5.9722e+24  # Mass of Earth in kg
theta_E = np.deg2rad(15.04 / 3600)  # rotation rate of earth in rads/s
#--------------------------------------------------------------------------------------------------

# Calculate State Vectors r & v via Algorithm 4.5 from 'Orbital Mechanics for Engineering Students' pg.218

r_xbar_const = ((h ** 2) / u0) * (1 / (1 + e * np.cos(ta)))
r_xbar = (r_xbar_const * np.cos(ta), r_xbar_const * np.sin(ta), 0)

v_xbar_const = u0 / h
v_xbar = (v_xbar_const * -np.sin(ta), v_xbar_const * (e + np.cos(ta)), 0)

q1 = np.matrix([[np.cos(w), np.sin(w), 0], [-np.sin(w), np.cos(w), 0], [0, 0, 1]])
q2 = np.matrix([[1, 0, 0], [0, np.cos(i), np.sin(i)], [0, -np.sin(i), np.cos(i)]])
q3 = np.matrix([[np.cos(raan), np.sin(raan), 0], [-np.sin(raan), np.cos(raan), 0], [0, 0, 1]])

Q_X_xbar = np.linalg.multi_dot([q1, q2, q3])  # Direction Cosine Matrix
Q_xbar_X = np.matrix.transpose(Q_X_xbar)

r_X = np.dot(Q_xbar_X, r_xbar)  # Position state vector of satellite in the geocentric equitorial frame (km)
v_X = np.dot(Q_xbar_X, v_xbar)  # Velocity state vector of satellite in the geocentric equitorial frame (km/s)

r_E = np.zeros((3))  # Position state vector of Earth
v_E = np.zeros((3))  # Velocity state vector of Earth

l = np.array([r_E[0], r_E[1], r_E[2], r_X[0, 0], r_X[0, 1], r_X[0, 2], v_E[0], v_E[1], v_E[2], v_X[0, 0], v_X[0, 1], v_X[0, 2]])
#------------------------------------------------------------------
# Calculate components of acceleration of satellite from equation of motion r'' = -(u0/magnitude(r_X^3))*vector(r_X)


def TwoBodyEoM(l, t):
    r1 = l[:3]
```

```
    r2 = l[3:6]
    v1 = l[6:9]
    v2 = l[9:12]
    r = np.linalg.norm(r2 - r1)
    aj2_coefficient = (-3.0 / 2.0) * J2 * (u0 / r ** 2) * (Re / r) ** 2
    aj2x = aj2_coefficient * (1 - 5 * ((r2[2] / r) ** 2)) * (r2[0] / r)
    aj2y = aj2_coefficient * (1 - 5 * ((r2[2] / r) ** 2)) * (r2[1] / r)
    aj2z = aj2_coefficient * (3 - 5 * ((r2[2] / r) ** 2)) * (r2[2] / r)
    aj2 = np.array([aj2x, aj2y, aj2z])

    dv1dt = (-u0 / r ** 3) * r1
    dv2dt = (-u0 / r ** 3) * r2 + aj2
    dr1dt = v1
    dr2dt = v2

    r_derivs = np.concatenate((dr1dt, dr2dt))
    derivs = np.concatenate((r_derivs, dv1dt, dv2dt))
    return derivs


initial_parameters = np.array([r_E[0], r_E[1], r_E[2], r_X[0, 0], r_X[0, 1], r_X[0, 2], v_E[0], v_E[1],
    v_E[2], v_X[0, 0], v_X[0, 1], v_X[0, 2]])
t_span = np.linspace(0, 100 * T, 10001)

two_body_sol = sci.integrate.odeint(TwoBodyEoM, initial_parameters, t_span)

rE_sol = two_body_sol[:, :3]
rS_sol = two_body_sol[:, 3:6]
rE_sol_velo = two_body_sol[:, 6:9]
rS_sol_velo = two_body_sol[:, 9:12]

#-------------------------------------------------------------------
# Create Sphere at origin to represent Earth
u, v = np.mgrid[0:2 * np.pi:100j, 0:np.pi:50j]
x_sphere = Re * np.cos(u) * np.sin(v)
y_sphere = Re * np.sin(u) * np.sin(v)
z_sphere = Re * np.cos(v)

# This function below taken from Karlo's Solution on Stack Overflow: https://stackoverflow.com/
    questions/13685386/matplotlib-equal-unit-length-with-equal-aspect-ratio-z-axis-is-not-
    equal-to


def set_axes_equal(ax):
    ''' Make axes of 3D plot have equal scale so that spheres appear as spheres,
    cubes as cubes, etc.. This is one possible solution to Matplotlib's
    ax.set_aspect('equal') and ax.axis('equal') not working for 3D.

    Input
      ax: a matplotlib axis, e.g., as output from plt.gca().
    '''

    x_limits = ax.get_xlim3d()
    y_limits = ax.get_ylim3d()
    z_limits = ax.get_zlim3d()
```

```python

    x_range = abs(x_limits[1] - x_limits[0])
    x_middle = np.mean(x_limits)
    y_range = abs(y_limits[1] - y_limits[0])
    y_middle = np.mean(y_limits)
    z_range = abs(z_limits[1] - z_limits[0])
    z_middle = np.mean(z_limits)

    # The plot bounding box is a sphere in the sense of the infinity
    # norm, hence I call half the max range the plot radius.
    plot_radius = 0.5 * max([x_range, y_range, z_range])

    ax.set_xlim3d([x_middle - plot_radius, x_middle + plot_radius])
    ax.set_ylim3d([y_middle - plot_radius, y_middle + plot_radius])
    ax.set_zlim3d([z_middle - plot_radius, z_middle + plot_radius])


fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_aspect("equal")
ax.plot(rE_sol[:, 0], rE_sol[:, 1], rE_sol[:, 2], color="red")
ax.plot(rS_sol[:, 0], rS_sol[:, 1], rS_sol[:, 2], color="darkblue")
ax.plot_surface(x_sphere, y_sphere, z_sphere, rstride=3, cstride=3, color='none', edgecolor='k', shade=0)
set_axes_equal(ax)
# plt.show()
#---------------------------------------------------------
# Get time histories of Orbital elements

r_hist = np.zeros((len(two_body_sol)))
velo_hist = np.zeros((len(two_body_sol)))
ta_hist = np.zeros((len(two_body_sol)))
raan_dot = -((3.0 / 2.0) * ((np.sqrt(u0) * J2 * Re ** 2) / (((1 - e ** 2) ** 2) * a ** (7.0 / 2.0)))) * (np.cos(i))  # rate of node line regression in rad/s
raan_hist = np.zeros((len(two_body_sol)))
w_dot = raan_dot * ((5.0 / 2.0) * np.sin(i) ** 2 - 2) / np.cos(i)
w_hist = np.zeros((len(two_body_sol)))
a_hist = np.zeros((len(two_body_sol)))
e_hist = np.zeros((len(two_body_sol)))
i_hist = np.zeros((len(two_body_sol)))

print(w_dot * (180 / np.pi) * 60 * 60 * 24)
for j in range(len(two_body_sol)):
    r_hist[j] = np.sqrt(rS_sol[j, 0] ** 2 + rS_sol[j, 1] ** 2 + rS_sol[j, 2] ** 2)
    velo_hist[j] = np.sqrt(rS_sol_velo[j, 0] ** 2 + rS_sol_velo[j, 1] ** 2 + rS_sol_velo[j, 2] ** 2)
    ta_hist[j] = np.arccos((h ** 2 - (u0 * r_hist[j])) / (u0 * r_hist[j] * e)) * (180.0 / np.pi)
    raan_hist[j] = (raan + raan_dot * t_span[j]) * (180.0 / np.pi)
    w_hist[j] = (w + w_dot * t_span[j]) * (180.0 / np.pi)
    a_hist[j] = (1.0 / 2.0) * ((2 * r_hist[j] * u0) / ((2 * u0) - (velo_hist[j] ** 2 * r_hist[j])))
    e_hist[j] = (-2 * a_hist[j] + 2 * apogee) / (2 * a_hist[j])
    i_hist[j] = np.arccos((-2.0 / 3.0) * (raan_dot * (1 - e_hist[j] ** 2) ** 2 * a ** (7.0 / 2.0)) / (np.sqrt(u0) * J2 * Re ** 2))

ax1 = plt.subplot(3, 2, 2)
ax1.plot(t_span / 3600, raan_hist)
```

```
plt.ylabel('$\Omega$ (deg)')

ax2 = plt.subplot(3, 2, 4)
ax2.plot(t_span / 3600, w_hist)
plt.ylabel('$\omega$ (deg)')

ax3 = plt.subplot(3, 2, 6)
ax3.plot(t_span / 3600, ta_hist * 2)
plt.ylabel('$\\theta$ (deg)')
plt.xlabel('Time (hrs)')

ax4 = plt.subplot(3, 2, 1)
ax4.plot(t_span / 3600, a_hist)
plt.ylabel('a (km)')

ax5 = plt.subplot(3, 2, 3)
ax5.plot(t_span / 3600, e_hist)
plt.ylabel('e')

ax6 = plt.subplot(3, 2, 5)
ax6.plot(t_span / 3600, i_hist * 180.0 / np.pi)
plt.ylabel('i (deg)')
plt.xlabel('Time (hrs)')

# plt.show()
#-----------------------------------
```