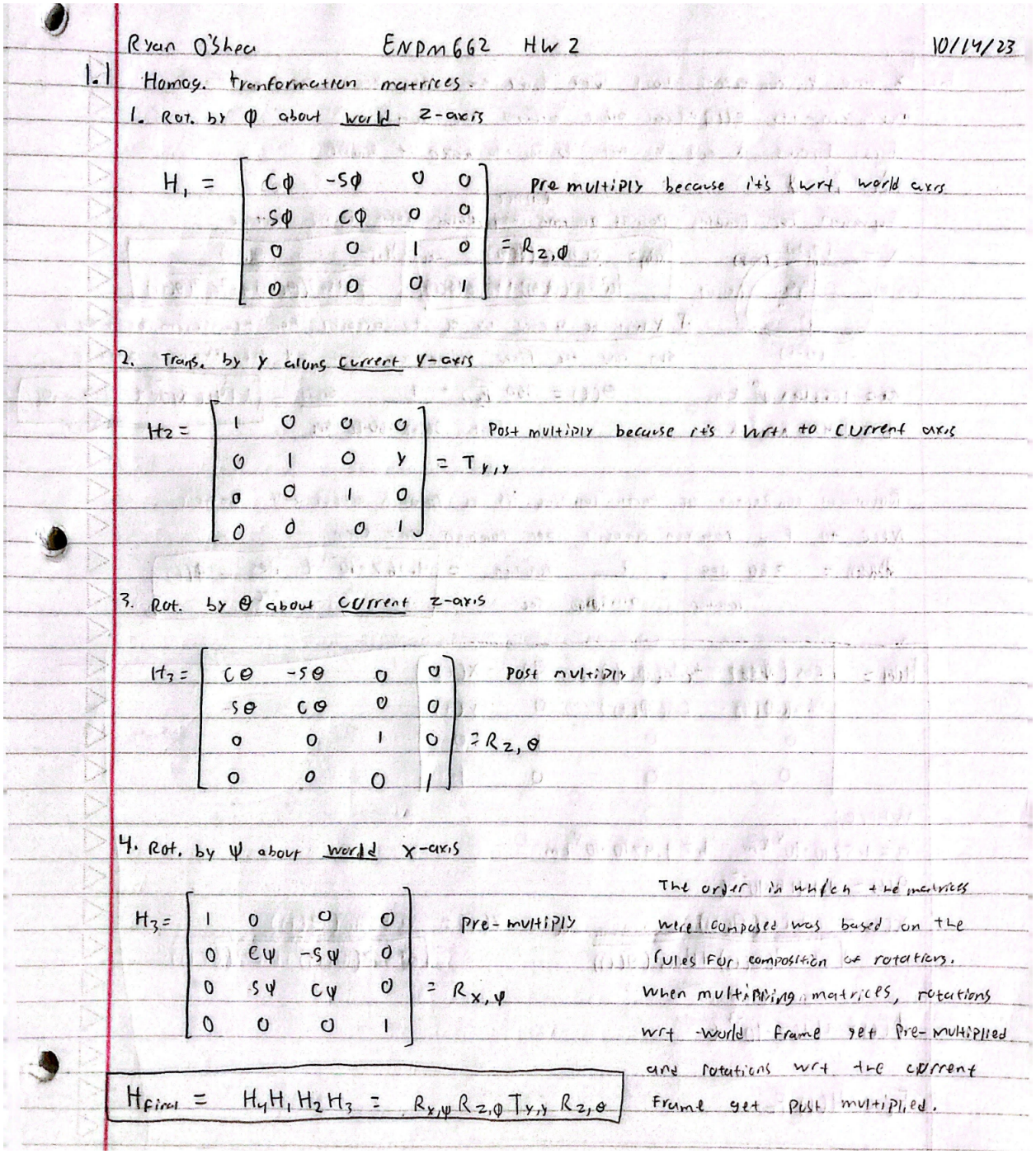
Ryan O'Shea
roshea
ENPM662

Problem 1.1



The homogeneous transformation matrices were formed by following the standard examples for rotation and translation about the standard axis. The order in which the

matrices are multiplied follows the convention described in the textbook of pre-multiplying and post-multiplying matrices depending on whether they are with respect to the world axis or the current axis respectively. H1, a rotation by phi about the world z-axis would be premultiplied if there was an existing non identity matrix but instead we just treat it as the starting point which makes the current order H1. H2, a translation by y along the current y-axis, gets post multiplied which brings the matrix order to H1H2. H3, a rotation of theta about the current z axis also gets post-multiplied which brings the matrix order to H1H2H3. H4, a rotation of psi about the world x axis, gets pre multiplied which brings the final matrix order to H4H1H2H3.

Problem 1.2

The transformation from the pose of earth at the summer solstice to to all other points in time was broken into pure translational and rotational components as there was only translational in the x and y directions and rotation about the z axis.For the translation components the (x,y) position was first calculate with respect to the location of the sun so the standard equations for an ellipse could be used. Polar coordinates were used so that we could calculate the position on an ellipse with respect to time more easily.

$$r(\theta) = \frac{ab}{\sqrt{(b\cos\theta)^2 + (a\sin\theta)^2}}$$

The equation for an ellipse in polar coordinates was found at the following link: https://en.wikipedia.org/wiki/Ellipse#Polar_form_relative_to_center. To convert this back into rectangular coordinates, I multiplied r(theta) by cos(theta) for x and sin(theta) for y. The x value needed to have the major axis value, a, subtracted from it to account for the transform between the sun and the earth at the moment of the summer solstice which is just a translation by a in the x axis. The values for the major and semi major axis of the ellipse that define the orbit of earth were found at the following link (aphelion for a and perihelion for b): https://en.wikipedia.org/wiki/Earth%27s_orbit#Events_in_the_orbit.

For the orbit of the earth around the sun theta was used to define the angle of rotation as it traced out a path along the ellipse. This angle varied with time over the course of 365 days so I divided 360 degrees by the number of seconds in a year to get the angle as a function of seconds since the orbit began.

The rotational component of the transformation matrix is just a rotation about the z axis by phi(t). Similar to the orbit of the earth, I divided 360 degrees by the number of seconds in a day to get the angle of rotation as a function of time. The rotational and translational components were put into a single homogeneous transformation matrix as a function of time which can be seen at the bottom of the page in the image below.

Point on an ellipse in Polar coord.

$$r(\theta) = \frac{ab}{\sqrt{a^2\sin^2(\theta) + b^2\cos^2(\theta)}}$$

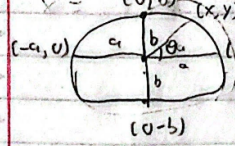
to convert to rectangular just multiply by $\cos(\theta)$ for x and $\sin(\theta)$ for y

1.2 X and Y components vary wrt time over the year as the earth moves in its elliptical orbit around the sun.

First finding X and Y wrt to Sun to make it simple

$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$ ellipse

Equations for finding points on an ellipse in Polar form found online



$$x(t) = \frac{ab\cos(\theta(t))}{\sqrt{a^2\sin^2(\theta(t)) + b^2\cos^2(\theta(t))}} - a \qquad y(t) = \frac{ab\sin(\theta(t))}{\sqrt{a^2\sin^2(\theta(t)) + b^2\cos^2(\theta(t))}}$$

* X will be shifted by a to account for transform between sun and the fixed frame of earth at the summer solstice

$a = 1.5210 \times 10^8$ km

$b = 1.4710 \times 10^8$ km

$$\theta(t) = 360\ \frac{\text{deg}}{\text{orbit}} \cdot \frac{1\ \text{orbit}}{365.24 \cdot 60 \cdot 60\ \text{sec}} = 1.1416 \times 10^{-5}\ t\ \frac{\text{deg}}{\text{sec}} = \theta(t)$$

Rotational component of earth rotating is a rotation about it's z-axis

Need to find rotation amount per common time step

$$\phi(t) = \frac{360\ \text{deg}}{\text{rotation}} \cdot \frac{1\ \text{rotation}}{24 \cdot 60 \cdot 60\ \text{sec}} = 4.167 \cdot 10^{-3}\ t\ \frac{\text{deg}}{\text{sec}} = \phi(t)$$

$$H(t) = \begin{bmatrix} \cos(\phi(t)) & -\sin(\phi(t)) & 0 & X(t) \\ \sin(\phi(t)) & \cos(\phi(t)) & 0 & Y(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where:

$a = 1.5210 \cdot 10^8$ km $\quad b = 1.4710 \cdot 10^8$ km

$\theta(t) = 1.1416 \times 10^{-5} t$

$$X(t) = \frac{ab\cos(\theta(t))}{\sqrt{a^2\sin^2(\theta(t)) + b^2\cos^2(\theta(t))}} - a \qquad Y(t) = \frac{ab\sin(\theta(t))}{\sqrt{a^2\sin^2(\theta(t)) + b^2\cos^2(\theta(t))}}$$

$\phi(t) = 4.167 \cdot 10^{-3} t$

$F_E = H(t)\, F_F$

Problem 1.3

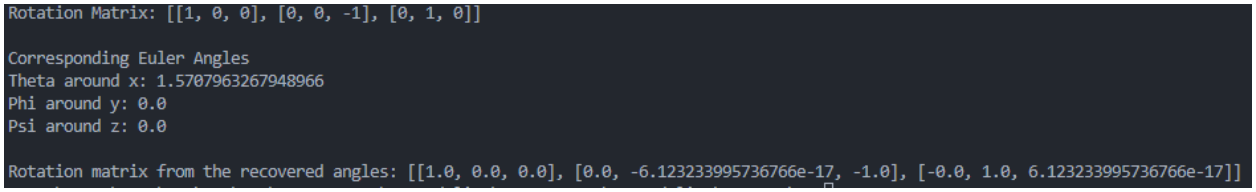
The code and relevant graphs for problem 1.3 and 2.1 can be found in the following google drive link: https://drive.google.com/drive/folders/1Mj5JF8oN183NMsqxufk7-x-qd1Fdd8tU?usp=sharing.

The desired transformation was obtained through 3 separate transformations, H1, H2, and H3. H1 is a 90 rotation about the world x axis. H2 is a translation by 1 meter along the world y axis. H3 is a translation of by -1 along the world Z axis. Because all of these transformations are with respect to the world axis they are all premultiplied and get the resulting matrix order of H3H2H1.

The rotation and translation components are broken out from the resulting 4x4 transformation matrix by taking the upper left 3x3 matrix and upper right 3x1 vector respectively. The translation can stay as is but the rotation matrix needs to be converted back to euler angles. The equations for finding the rotation angles, theta, phi, and psi about the axis x, y, and z were found online as the textbook example used the ZYZ configuration instead of the desired XYZ configuration. The calculations were performed in code in the problem_1_3.py file. A relevant snippet from the code is also posted below.

```python
from math import cos, sin, atan2, sqrt

# Define the rotation matrix we derived
rot_mat = [[1, 0, 0],
           [0, 0, -1],
           [0, 1, 0]]

# Break out the rotation matrix entries to make the theta and k calculations easier
r11 = rot_mat[0][0]
r12 = rot_mat[0][1]
r13 = rot_mat[0][2]
r21 = rot_mat[1][0]
r22 = rot_mat[1][1]
r23 = rot_mat[1][2]
r31 = rot_mat[2][0]
r32 = rot_mat[2][1]
r33 = rot_mat[2][2]

# Calculate the individual components for rotations about each axis
theta = atan2(r32, r33)
phi = atan2(-r31, sqrt(r32**2 + r33**2))
psi = atan2(r21, r11)

print("Rotation Matrix: {} \n".format(rot_mat))

print("Corresponding Euler Angles")
print("Theta around x: {}".format(theta))
print("Phi around y: {}".format(phi))
print("Psi around z: {} \n".format(psi))

# Recreate the rotation matrix from the angles for a sanity checl
rot_mat_from_angles = [[cos(phi)*cos(psi), sin(theta)*sin(phi)*cos(psi) - cos(theta)*sin(psi), cos(theta)*sin(phi)*cos(psi) + sin(theta)*sin(psi)],
                       [cos(phi)*sin(psi), sin(theta)*sin(phi)*sin(psi) - cos(theta)*cos(psi), cos(theta)*sin(phi)*sin(psi) - sin(theta)*cos(psi)],
                       [-sin(phi), sin(theta)*cos(phi), cos(theta)*cos(phi)]]

print("Rotation matrix from the recovered angles: {}".format(rot_mat_from_angles))
```

The print out from the results of the code can be seen below.



In order to check the correctness of the angle calculations, the angles were used to reconstruct the rotation matrix which is stored in the rot_mat_from_angles variable. The matrices match with the exception of the small floating point errors for some of the 0 values in the second and third rows of the rotation matrix.

☆ Write out matrices, use python to multiply them, extract euler angles from 3x3 rot mat in homog. trans. mat. ☆ - Saksham

$\cos(90) = 0$

$\sin(90) = 1$

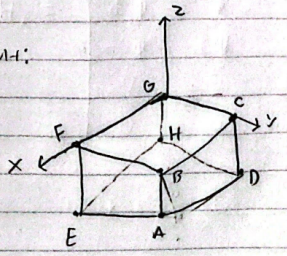
1.3 1. Rotation 90° ccw around the x-axis

2. Translation of 1m along the world y-axis

3. Translation of -1m along the world z-axis

$$H_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Result:



$$H_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$H_E = H_3 H_2 H_1$

$$\text{Temp} = H_2 H_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$H_3 \cdot \text{Temp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Extract rotation aspect as euler angles

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

$\theta_x = \text{atan2}(r_{32}, r_{33}) = 1.5708$

$\psi_y = \text{atan2}(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}) = 0$

$\psi_z = \text{atan2}(r_{21}, r_{11}) = 0$

Translation: $T = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$

Check for angles → R done in code

Problem 2.1

This problem was solved entirely with code and can be found in the problem_2_1.py file. The software uses axis angle representation to find the optimal axis to rotate about to reach the desired orientation in the shortest period of time. The caclculateAxisAngleForm function takes in the desired final orientation in the form of rotation angles that would be applied about the x, y, and z axis in that order. The rotation matrix is calculated using the numpy matmul and then used to find the axis-angle form using the equations in the textbook.The code for converting the rotation matrix to axis angle form can be seen below. The rxx variables are just the entries of the rotation matrix broken out into individual variables to make calculations cleaner. The returned theta is converted from radians to degrees using a helper function because the rest of the calculations are in degrees to better match with the defined angles and angular velocity.

```python
# Converts desired rotations around the x, y, and z axis into axis angle form
def caclculateAxisAngleForm(self, x_rot, y_rot, z_rot):
    # Multiply the rotation matrices together in the order Z(YX)
    final_rot_mat = np.matmul(self.getZRotMat(self.deg2Rad(z_rot)),
                              np.matmul(self.getYRotMat(self.deg2Rad(y_rot)),
                                        self.getXRotMat(self.deg2Rad(x_rot))))

    # Break out the rotation matrix entries to make the theta and k calculations easier
    r11 = final_rot_mat[0][0]
    r12 = final_rot_mat[0][1]
    r13 = final_rot_mat[0][2]
    r21 = final_rot_mat[1][0]
    r22 = final_rot_mat[1][1]
    r23 = final_rot_mat[1][2]
    r31 = final_rot_mat[2][0]
    r32 = final_rot_mat[2][1]
    r33 = final_rot_mat[2][2]

    # Calculate the angle and vector for the new axis of rotation
    theta = math.acos((r11 + r22 + r33 - 1)/2)

    k = (1/(2*math.sin(theta))) * np.array([r32 - r23,
                                            r13 - r31,
                                            r21 - r12])

    print("Axis angle form: theta={} \t k={}".format(theta, k))

    return self.rad2Deg(theta), k
```

The getXRotMat and similar functions for Y and Z just return a rotation matrix for a given passed in angle. An example of one of these functions can be seen below.

```python
# Returns a rotation matrix for rotating around the x axis by the passed in angle
def getXRotMat(self, angle):
    rot_mat = np.array([
            [1, 0, 0],
            [0, math.cos(angle), -math.sin(angle)],
            [0, math.sin(angle), math.cos(angle)]])

    return rot_mat
```

The printout of the axis angle form can be seen below:

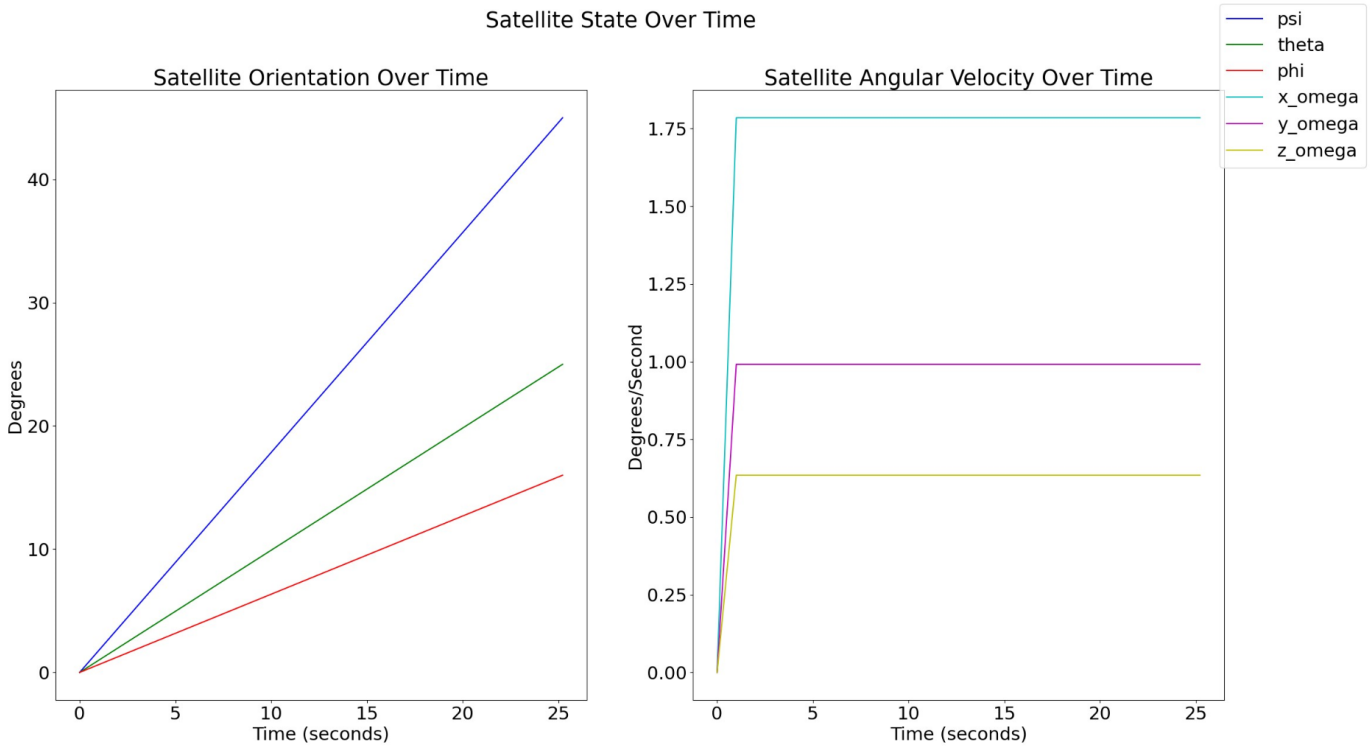
```
Axis angle form: theta=0.8801049931142315       k=[0.80318587 0.58690734 0.10213824]
```

The rotateOtherAxis function is the main function for calculating the orientation over time and can be seen both in the code and below.The time for the rotation is found by dividing the theta found from the axis angle calculation by the given max angular velocity of 2 degrees per second. This time is then used to calculate the angular velocity about each of the axes as they will be proportional to the desired that needs to be rotated for that given axis. Intermediate values for the orientation angles are then calculated at each second and saved using the updateIntermediates function which just uses a dictionary of lists to save the satellite state over time. The timesteps are also incremented so we can plot against time later. The partial variable is used to take a final partial timestep at the max angular velocity but for only a portion of a second.

```python
def rotateOtherAxis(self, axis_angle, world_angles):
    # Find total time it would take to rotate around the new axis at the max rotation speed
    min_time = axis_angle/self.max_ang_vel

    # Calculate the velocity about each axis based on the time it will take to perform the rotation
    scaled_vels = [vel/min_time for vel in world_angles]

    print("Angular velocities (x, y, z): {}".format(scaled_vels))

    # Find the whole and partial seconds required to perform the full rotation
    whole_sec = int(axis_angle/self.max_ang_vel)
    partial = abs(axis_angle) % self.max_ang_vel

    # Get the intermediate values for the state variables for each whole timestep
    for i in range(whole_sec):
        self.current_pose["psi"] += scaled_vels[0]
        self.current_pose["theta"] += scaled_vels[1]
        self.current_pose["phi"] += scaled_vels[2]

        self.current_pose["x_omega"] = scaled_vels[0]
        self.current_pose["y_omega"] = scaled_vels[1]
        self.current_pose["z_omega"] = scaled_vels[2]

        self.timesteps.append(self.timesteps[-1] + 1.0)
        self.updateIntermediates()

    # Get the final values for the state variables based on the remaining partial timestep
    if partial > 0.0:
        self.current_pose["psi"] += scaled_vels[0] * (partial/self.max_ang_vel)
        self.current_pose["theta"] += scaled_vels[1] * (partial/self.max_ang_vel)
        self.current_pose["phi"] += scaled_vels[2] * (partial/self.max_ang_vel)

        self.timesteps.append(self.timesteps[-1] + partial/self.max_ang_vel)

        self.updateIntermediates()
```

The resulting state of the satellite and graph over time can be seen below. A larger easier to read copy is included as part of the submission with the name “Satellite state over time.png”.

```
Axis angle form: theta=0.8801049931142315       k=[0.80318587 0.58690734 0.10213824]
Angular velocities (x, y, z): [1.7847828828202297, 0.9915460460112386, 0.6345894694471927]
psi: 45.00000000000002
theta: 24.99999999999999
phi: 15.999999999999993
x_omega: 1.7847828828202297
y_omega: 0.9915460460112386
z_omega: 0.6345894694471927
time: 25.213150816917924
```



The satellite reaches its desired orientation for each axis at the same time in a **time of 25.21 seconds**. The angular velocity about each of the axes stays constant the entire time after jumping up from 0.

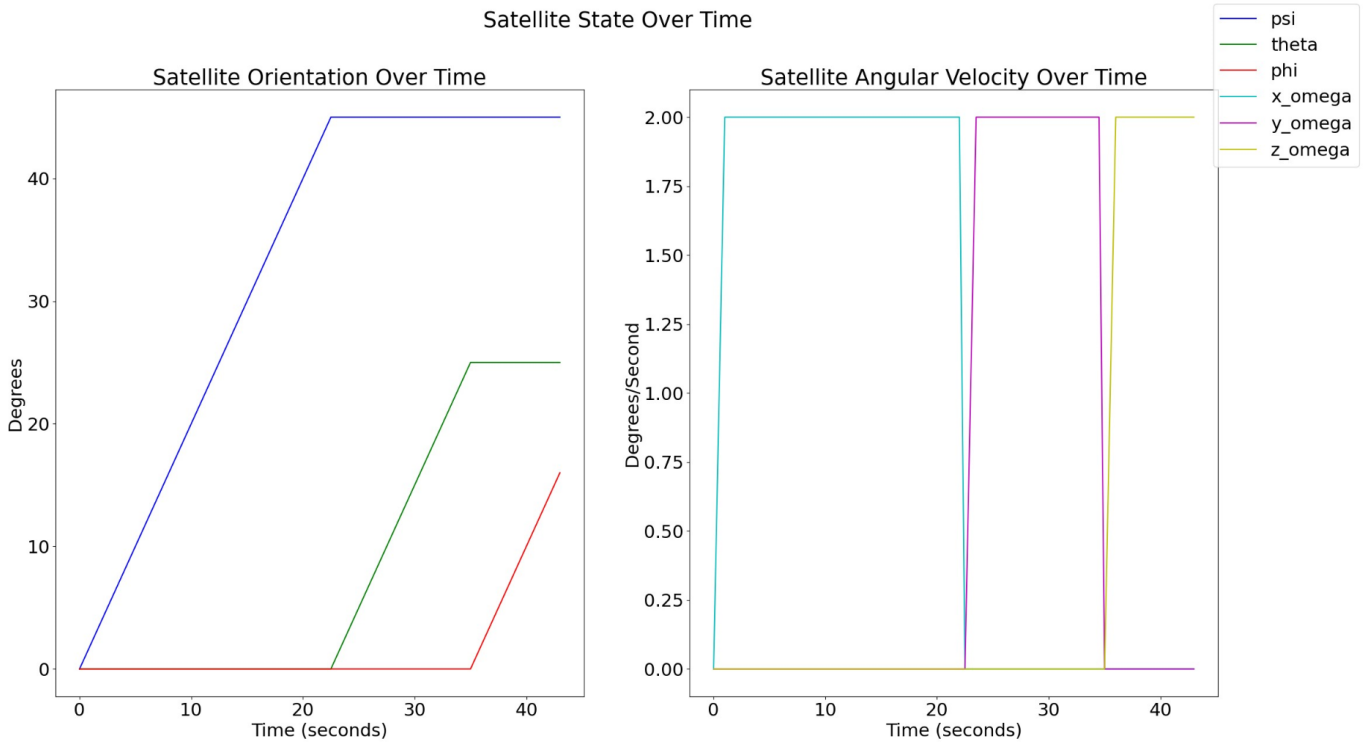
The code used to create the plots can be seen below.

```python
# Plots the 6 state variables over time on a single plot
def displayStatePlot(self):
    # Create two separate plots, one for angles and one for angular velocities
    fig, ax = plt.subplots(nrows=1, ncols=2)
    for key in self.intermediate_vals.keys():
        # Add to angle plot
        if key in self.angles:
            ax[0].plot(self.timesteps, self.intermediate_vals[key], color=next(self.plot_colors), label=key)
        # Add to angular velocity plot
        elif key in self.angle_vels:
            ax[1].plot(self.timesteps, self.intermediate_vals[key], color=next(self.plot_colors), label=key)

    ax[0].set_xlabel("Time (seconds)")
    ax[1].set_xlabel("Time (seconds)")
    ax[0].set_ylabel("Degrees")
    ax[1].set_ylabel("Degrees/Second")

    ax[0].set_title("Satellite Orientation Over Time")
    ax[1].set_title("Satellite Angular Velocity Over Time")

    fig.suptitle("Satellite State Over Time")

    fig.legend()

    plt.show()
```

Initial incorrect attempt:

I initially thought that the rotations had to be applied sequentially instead of around a common axis and implemented it as such using the rotate function in the code. While incorrect I think it provides a nice contrast and highlights how much more efficient rotating about the equivalent axis-angle representation axis is. The resulting state and the graph of the state over time can be found below and in an image named “3 individual rotations graphs.png”. The optimized implementation reached the desired state in 25.21 seconds and the unoptimized 3 rotation method reached the desired state in 43 seconds.

```
psi: 45.0
theta: 25.0
phi: 16.0
x_omega: 0.0
y_omega: 0.0
z_omega: 0.0
time: 43.0
```


Satellite State Over Time
Satellite Orientation Over Time
Degrees
Time (seconds)
Satellite Angular Velocity Over Time
Degrees/Second
Time (seconds)
psi
theta
phi
x_omega
y_omega
z_omega