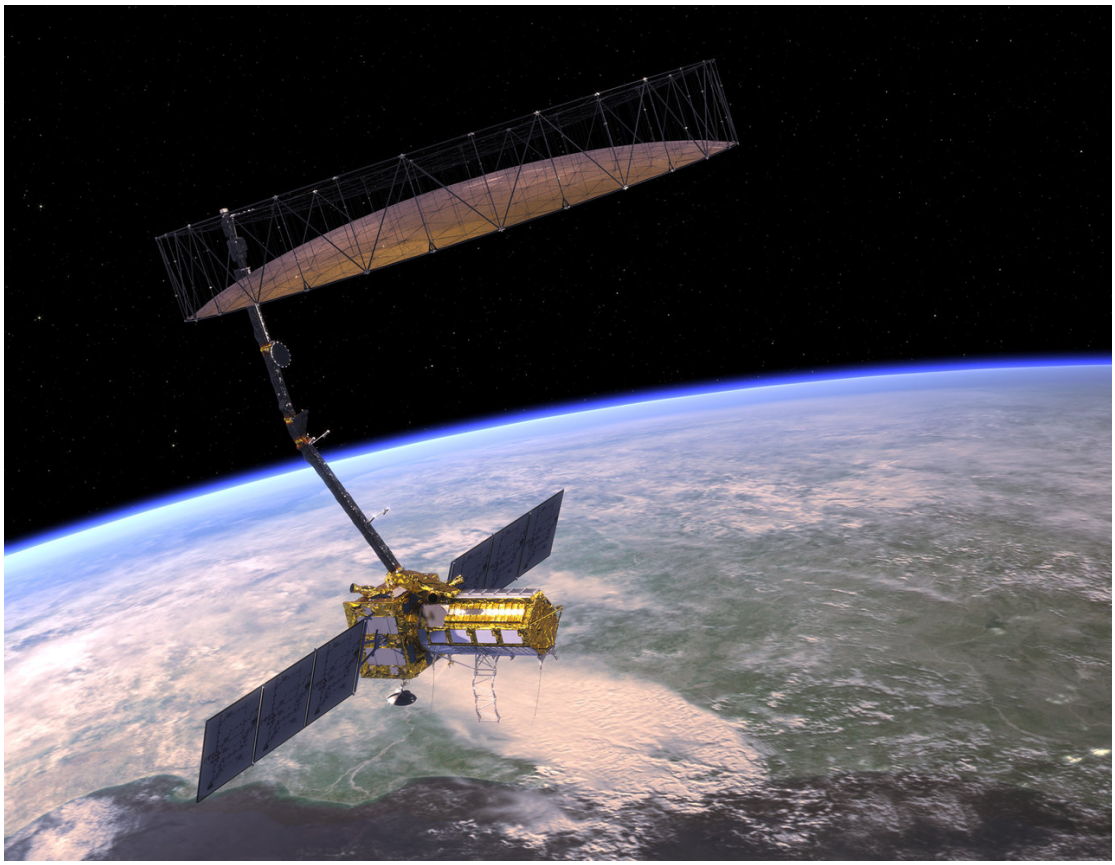
SATELLITE DYNAMICS AND ATTITUDE CONTROL

Kusal Uprety, Harry Zhao

12 June 2024



REVISION HISTORY

VERSION	REVISION NOTES
PS1	- Created document - Added PS1 material
PS2	- Added PS2 material - Updated title page - Updated moment of inertia for center of mass
PS3	- Added PS3 material - Updated title page
PS4	- Added PS4 material - Updated title page - Switched to 312 Euler angles
PS5	- Added PS5 material - Updated title page - Fix PS2 figure caption typos - Fix PS4 figure numbering
PS6	- Introduce Simulink model - Added PS6 material - Updated title page - Fixed PS5 stability plot
PS7	- Added PS7 material - Updated title page - Added PS5 RTN attitude and net torque plots
PS8	- Added PS8 material - Updated title page
PS9	- Added PS9 material - Updated title page - Fixed advanced magnetic field model
PS10	- Reformatted for final project submission - Added magnetorquer actuator model - Added desaturation mode - Extended magnetic field model for accurate modeling of magnetorquer - Addressed PS8 comments about covariance errors

Table 1: Summary of project revisions.

TABLE OF CONTENTS

1 INTRODUCTION

Our mission will utilize a satellite with synthetic aperture radar (SAR), designed to gather key remote sensing and environmental data for the Earth. The satellite will be in low Earth orbit (LEO) and use quaternions to describe its orientation, avoiding gimbal lock effects of other conventions. For state estimation, the spacecraft will require gyroscopes, star trackers, and a potentially a sun sensor. For actuation, the spacecraft will likely utilize thrusters, reaction wheels, and magnetorquers.

1.1 Literature Review

Space agencies such as NASA have been constructing SAR satellites to gather satellite images and data of Earth for over a decade. Additionally, there exist commercial entities also utilizing SAR in their spacecraft.

For example, Soil Moisture Active Passive (SMAP) is a NASA satellite launched in 2015 that utilizes L-band synthetic aperture radar (SAR) technology to measure soil moisture from LEO. This data has applications in climate change research climate change research applications (such as updating climate models) and some day-to-day activities (such as improving weather forecasts). SMAP is unique in that it had a large deployable reflector, held above the spacecraft body by a deployable boom [1].

Companies EOS and Capella Space are also developing satellites that use SAR technology in the X-band and S-band frequencies for commercial applications ranging from agriculture to mining [2], [3]. The commercial applicability of SAR is substantial, especially as SAR can penetrate cloud cover while generating high-resolution data, making it superior to many other forms of remote sensing technology. EOS claims to obtain resolution of up to 0.25 m, while Capella Space claims a capability of up to 0.5 m. These satellites all operate in LEO, which enables high-frequency monitoring of the Earth's surface.

NASA and ISRO have partnered to create a SAR satellite as well. The joint project between NASA JPL and ISRO has resulted in the NASA-ISRO Synthetic Aperture Radar (NISAR), a satellite that captures data in the L-band and S-band SAR frequencies [4]. NISAR's high resolution will permit the detailed measurement of the Earth's surface, enabling better observation of changes in Earth's crust for disaster prevention and mitigation. NISAR will also support science goals such as monitoring ice sheets and the oceans, and its orbit is designed to cover the entire Earth every 12 days.

1.2 Mission Selection

NISAR is a joint Earth-observation satellite mission between NASA and ISRO. It is the first satellite to operate in two different Synthetic Aperture Radar (SAR) bands, incorportating both L- and S-band SAR instruments. Both frequencies can penetrate clouds for reliable data collection, but the L-band can also penetrate thicker vegetation that the S-band cannot. Uniquely, NISAR is intended to be used for a wide range of science objectives, including disaster response and agriculture [5].

2 SPECIFICATIONS

2.1 Requirements

NISAR is a joint Earth-observation satellite mission between NASA and ISRO. It is the first satellite to operate in two different Synthetic Aperture Radar (SAR) bands, incorportating both L- and S-band SAR instruments. Both frequencies can penetrate clouds for reliable data collection, but the L-band can also penetrate thicker vegetation that the S-band cannot. Uniquely, NISAR is intended to be used for a wide range of science objectives, including disaster response and agriculture [5].

NISAR ADCS requirements are $<$273 arcseconds for pointing and $<$500 m for orbit control [6]. The satellite duty cycle is specified as $>$30%. NISAR will operate in LEO with nominal altitude of 747 km and 6 AM/6 PM orbit. NISAR's L- and S-band instruments operate at 24 cm and 12 cm wavelengths, respectively. NISAR collects terrestial SAR imagery with an image swatch of 240 km using a sweep approach. The science payload can also perform polarimetry, with the SAR incorporating multiple polarization modes.

2.2 Layout

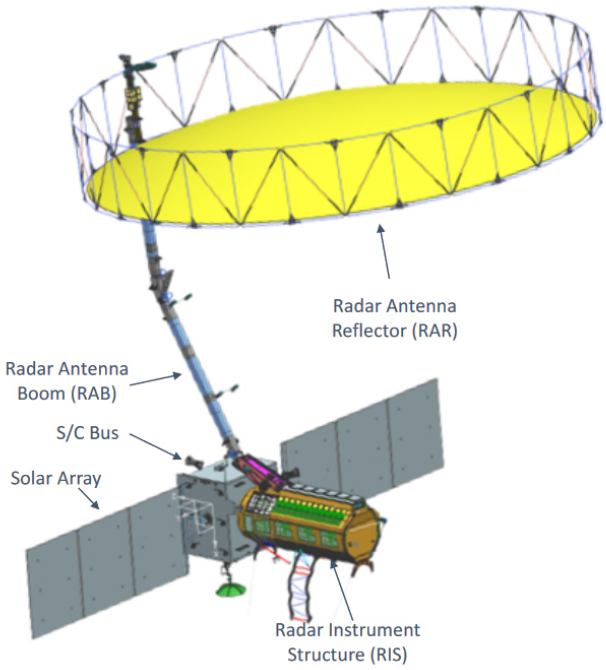


Figure 1: The basic components of the NISAR satellite

As shown in Figure 1, NISAR's satellite consists of a 1.2 m x 1.8 m x 1.9 m spacecraft bus cuboid with a 1.2 m wide octagonal Radar Instrument Structure (RIS). The spacecraft bus includes ADCS hardware, power subsystem, and engineering payload, while the RIS houses hardware for the L- and S-band SAR. The satellite is powered by 23 m^2 of solar panels, consisting of an array of two panels, one on each side of the satellite. Additionally, a 12 m diameter radar antenna is positioned above the body of the spacecraft, attached by a 9 m long boom. This boom consists of beams with 7 in x 7 in cross-section area [4].

Table 2 contains mass properties of the satellite. Unfortunately, detailed mass distribution and inertia properties of NISAR are not openly available, so we provide estimates of mass distribution based on known overall component-level masses. We are given total masses for the bus structure and RIS, and we also know the masses of the payloads located within, allowing us to compute an accurate mass for these components [4]. We estimate that solar panels have a mass of 23 kg each based on knowledge that NISAR's solar panels are 23 m^2 and a typical solar panel mass per area is 2.06 kg/m^2 [7]. We know that the entire radar antenna assembly has a mass of 292 kg, and we estimate that the reflector has a mass of approximately 100 kg based on a similar deployable SAR S- and L-band mesh antenna reflector [8]. We will use these masses to compute center of mass and moments of inertia in the following section. For our model, we neglect the effects of the truss structure supporting the antenna reflector, instead modeling the entire RAR as just the disk-shaped reflector mesh.

Table 2: Mass of NISAR components

Components	**Mass [kg]**
Bus	964.1
Radar Instrument Structure (RIS)	1375.9
Solar Panel +y	23
Solar Panel -y	23
Radar Antenna Boom (RAB)	192
Radar Antenna Reflector (RAR)	100

Since NISAR is a remote sensing satellite requiring high attitude control performance, it has an ADCS system with an array of sensors and actuators. Sensors include star sensors, sun sensors, GPS, and a 3-axis gyroscope for roll, pitch, and yaw. For actuators, NISAR has four 50 N·m reaction wheels mounted in tetrahedral configuration, three 565 and 350 A·m^2 magnetorquers, and fourteen thrusters (ten canted 11 N thrusters, one central 11 N thruster, and four 1 N thrusters for roll) [4].

2.3 Mass Properties

We simplify the spacecraft geometry into six components, each individually assumed to have uniform mass distribution. These components of the simplified geometry are: bus structure (including ADCS hardware and engineering payload), RIS (Radar Instrument Structure), RAB (radar antenna boom), RAR (radar antenna reflector), and two solar panels (identified as the +y solar panel and -y solar panel). The bus structure is modeled as a rectangular prism, while the RIS is modeled as an octagonal prism. The RAB is also modeled as a rectangular prism, while the RAR is modeled as a thin disk and the solar panels are modeled as thin rectangular plates. Within each geometry, our model assumes mass is distributed uniformly. From analyzing diagrams found in technical reports, we estimate that the RAR is tilted -3.87° about the y-axis (relative to the x-axis), while the RAB is modeled as a single beam with an angle approximately -18° from vertical (from the z-axis, about the y-axis in the x-z plane). Note that we have simplified the shape of the RAB from a beam of two angled segments to a single, straight beam.

We choose the body axes to have an origin at the center of the rectangular bus. This configuration is chosen because the bus houses the ADCS hardware, including actuators and sensors. The x-axis points in the direction of the RIS, and the z-axis points up vertically, normal to the

upper surface of the bus. See Figure 4 for a visual depiction of the body axes relative to the spacecraft.

We compute the center of mass after extracting the centroid of each component. The mass of each component is previously found in Table 2. The centroid of each component is listed in Table 3.

Table 3: Component centroids [m]

Part	**x**	**y**	**z**
Bus	0	0	0
RIS	1.85	0	0
Panel +y	0	3.9	0
Panel -y	0	-3.9	0
RAB	-0.899	0	5.194
RAR	4.283	0	8.308

The center of mass can be found by taking the weighted average of each component centroid, weighted by the mass of each component. The center of mass formula is:

$$\vec{r}_{cm} = \frac{\sum m_i \vec{r}_i}{\sum m_i}$$

This yields a result for center of mass at $[1.046, 0, 0.683]\,\mathrm{m}$ relative to the origin we defined. We use a MATLAB script to compute the center of mass from a CSV file containing centroid and mass data.

We compute the moment of inertia of the satellite, finding an inertia tensor in our body axes. To do this, we break the satellite into individual components, first finding the moment of inertia about the center of mass of each component. We then compute the moment of inertia of the entire satellite about the body axes by using parallel axis theorem and combining all the components.

To compute the moment of inertia, we need the following geometric properties of each component:

Table 4: Dimensions of modeled components [m]

Part	**L (x-dim)**	**W (y-dim)**	**H (z-dim)**	**S (oct. side length)**	**R (radius)**
Bus		1.8	1.9	-	-
RIS	2.5	-	-	0.459	-
Panel +y	-	6	1.9	-	-
Panel -y	-	6	1.9	-	-
RAB	0.1778	0.1778	9	-	-
RAR	-	-	-	-	6

For the bus, we choose to model the geometry as a rectangular prism. Since the bus is aligned

with the body axes, we obtain a diagonal inertia tensor:

$$
\begin{aligned}
I_{bus} &= \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \\
&= \begin{bmatrix} m\frac{W^2+H^2}{12} & 0 & 0 \\ 0 & m\frac{L^2+H^2}{12} & 0 \\ 0 & 0 & m\frac{L^2+W^2}{12} \end{bmatrix} \\
&= \begin{bmatrix} 550.340 & 0 & 0 \\ 0 & 405.725 & 0 \\ 0 & 0 & 375.9993 \end{bmatrix} \text{ kg m}^2
\end{aligned}
$$

For the solar panels, we approximate their geometry as a flat plate. These axes are also aligned, so the tensor can be diagonal.

$$
\begin{aligned}
I_{panel} &= \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \\
&= \begin{bmatrix} m\frac{W^2+H^2}{12} & 0 & 0 \\ 0 & m\frac{H^2}{12} & 0 \\ 0 & 0 & m\frac{W^2}{12} \end{bmatrix} \\
&= \begin{bmatrix} 75.919 & 0 & 0 \\ 0 & 6.919 & 0 \\ 0 & 0 & 69 \end{bmatrix} \text{ kg m}^2
\end{aligned}
$$

For the RIS, we model the geometry as an octagonal prism. For the moment of inertia about the axisymmetric axis of the octagon, we use the formula $m\left(\frac{S^2}{24}+\frac{a^2}{2}\right)$, where S is the side length and a is the apothem length, where the apothem is the perpendicular length from a side of the octagon to the center [9]. When calculating the moment of inertia about the non-axisymmetric axes, we approximate the geometry as a cylinder with radius equal to the average of the octagonal radius (distance from center to vertex) and the apothem. As will be shown later, this approximation yields a very close result to the inertia tensor generated from the CAD model.

$$
\begin{aligned}
I_{RIS} &= \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \\
&= \begin{bmatrix} m\left(\frac{S^2}{24}+\frac{a^2}{2}\right) & 0 & 0 \\ 0 & m\left(\frac{L^2}{12}+\frac{R_{avg}^2}{4}\right) & 0 \\ 0 & 0 & m\left(\frac{L^2}{12}+\frac{R_{avg}^2}{4}\right) \end{bmatrix} \\
&= \begin{bmatrix} 223.268 & 0 & 0 \\ 0 & 831.089 & 0 \\ 0 & 0 & 831.089 \end{bmatrix} \text{ kg m}^2
\end{aligned}
$$

For the RAB, we first model the geometry as a rectangular prism. We also must rotate the inertia tensor to match the orientation of the body axes, as the RAB itself is rotated relative to

the bus about the y-axis by -18° from vertical (z-axis).

$$
\begin{aligned}
I_{RAB} &= \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \\
&= \begin{bmatrix} m\frac{W^2+H^2}{12} & 0 & 0 \\ 0 & m\frac{L^2+H^2}{12} & 0 \\ 0 & 0 & m\frac{L^2+W^2}{12} \end{bmatrix} \\
&= \begin{bmatrix} 1296.506 & 0 & 0 \\ 0 & 1296.506 & 0 \\ 0 & 0 & 1.012 \end{bmatrix} \text{ kg m}^2
\end{aligned}
$$

We apply the rotation to the inertia tensor using a rotation matrix about the y-axis. Note that doing so results in non-zero products of inertia, meaning our principal axes will not be aligned with our body axes.

$$
\begin{aligned}
I_{RAB,rotated} &= R_y(-18^\circ) I_{RAB} R_y^\intercal(-18^\circ) \\
&= \begin{bmatrix} 1172.797 & 0 & 380.736 \\ 0 & 1296.506 & 0 \\ 380.736 & 0 & 124.720 \end{bmatrix} \text{ kg m}^2
\end{aligned}
$$

Finally, we model the RAR as a flat disk. Similar to the RAB, we must rotate the inertia tensor to match the orientation of the body axes.

$$
\begin{aligned}
I_{RAR} &= \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \\
&= \begin{bmatrix} m\frac{R^2}{4} & 0 & 0 \\ 0 & m\frac{R^2}{4} & 0 \\ 0 & 0 & m\frac{R^2}{2} \end{bmatrix} \\
&= \begin{bmatrix} 900 & 0 & 0 \\ 0 & 900 & 0 \\ 0 & 0 & 1800 \end{bmatrix} \text{ kg m}^2
\end{aligned}
$$

Applying a rotation:

$$
\begin{aligned}
I_{RAR,rotated} &= R_y(-3.87^\circ) I_{RAR} R_y^\intercal(-3.87^\circ) \\
&= \begin{bmatrix} 904.100 & 0 & -60.605 \\ 0 & 900 & 0 \\ -60.605 & 0 & 1795.900 \end{bmatrix} \text{ kg m}^2
\end{aligned}
$$

Now, we use parallel axis theorem to compute the moment of inertia of each component about the body axes at the specified origin. We can use the following displacement tensor,

$$
D = \begin{bmatrix} y^2+z^2 & -xy & -xz \\ -yx & x^2+z^2 & -yz \\ -zx & -yz & x^2+y^2 \end{bmatrix},
$$

where x, y, z are the coordinates of the center of mass of the component, giving the moment of inertia about a new point:

$$
I' = I_c + mD
$$

Performing the parallel axis theorem on each component and summing the inertia tensors, we obtain the following inertia tensor for the entire spacecraft:

$$I_{NISAR,body} = \begin{bmatrix} 15783.996 & 0 & -2341.659 \\ 0 & 22227.752 & 0 \\ -2341.659 & 0 & 10663.970 \end{bmatrix} \text{kg m}^2$$

Compare this with the inertia tensor computed by SolidWorks CAD software:

$$I_{NISAR,body} = \begin{bmatrix} 15780.361 & 0 & -2336.285 \\ 0 & 22225.721 & 0 \\ -2336.285 & 0 & 10665.796 \end{bmatrix} \text{kg m}^2$$

The errors are 0.0230%, 0.00914%, 0.0171%, and 0.230% for I_{xx}, I_{yy}, I_{zz}, and I_{xz}, respectively. We created the following MATLAB script to compute the inertia tensor.

2.4 Surface Properties

For the purpose of discretizing the spacecraft into surfaces, we consider the outer faces of the bus, RIS, and RAB. We also consider the faces of the solar panels and RAR, which are modeled as thin plates. The centroid (barycenter) coordinates and area for each surface are obtained using the surface properties tool in SolidWorks, and a unit normal vector is manually computed based on the orientation of the surface. We then enter this data into a CSV file, which can be read into MATLAB.

This function stores the data into arrays of barycenter coordinates, unit normal vector components, and area. Each row of an array corresponds to a particular surface. The data is shown in Table 5, annotated with the identity of each surface.

Table 5: Surface parameters

	Barycenter [m]			**Normal**			
Surface	**x**	**y**	**z**	**x**	**y**	**z**	**Area [m^2]**
Bus front, minus RIS (+x)	0.6	0	0	1	0	0	2.4
Bus rear (-x)	-0.6	0	0	-1	0	0	3.42
Bus side (+y)	0	0.9	0	0	1	0	2.28
Bus side (-y)	0	-0.9	0	0	-1	0	2.28
Bus top (+z)	0	0	0.95	0	0	1	2.16
Bus bottom (-z)	0	0	-0.95	0	0	-1	2.16
RIS front (+x)	3.1	0	0	1	0	0	1.02
RIS top (+z)	1.85	0	0.55	0	0	1	1.15
RIS bottom (-z)	1.85	0	-0.55	0	0	-1	1.15
RIS side (+y)	1.85	0.55	0	0	1	0	1.15
RIS side (-y)	1.85	-0.55	0	0	-1	0	1.15
RIS angle face (y-z I)	1.85	0.39	0.39	0	0.707	0.707	1.15
RIS angle face (y-z II)	1.85	-0.39	0.39	0	-0.707	0.707	1.15
RIS angle face (y-z III)	1.85	-0.39	-0.39	0	-0.707	-0.707	1.15
RIS angle face (y-z IV)	1.85	0.39	-0.39	0	0.707	-0.707	1.15
Panel +y front (+x)	0	3.9	0	1	0	0	11.4
Panel +y rear (-x)	0	3.9	0	-1	0	0	11.4

Panel -y front (+x)	0	-3.9	0	1	0	0	11.4
Panel -y rear (-x)	0	-3.9	0	-1	0	0	11.4
RAB front (x-z, +x)	-0.81	0	5.21	0.951	0	0.309	1.6
RAB rear (x-z, -x)	-0.99	0	5.18	-0.951	0	-0.309	1.6
RAB side (+y)	-0.9	-0.09	5.19	0	1	0	1.6
RAB side (-y)	-0.9	0.09	5.19	0	-1	0	1.6
RAB top (+z)	-2.31	0	9.47	0	0	1	0.03
RAR top (+z)	4.28	0	8.31	-0.067	0	0.998	113.1
RAR bottom (-z)	4.28	0	8.31	0.067	0	-0.998	113.1

2.5 Body Axes

A 3D model of the spacecraft is shown in Figure 2. The model shown is a screen capture from the SolidWorks CAD software.

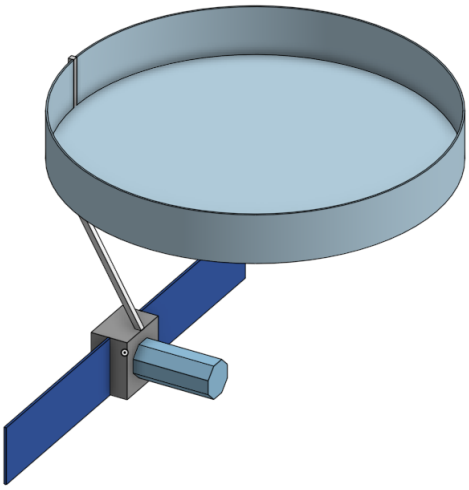


Figure 2: A 3D model of the satellite

We also show a simplified model of the spacecraft in Figure 3. This is the model we use to compute our mass and surface properties.

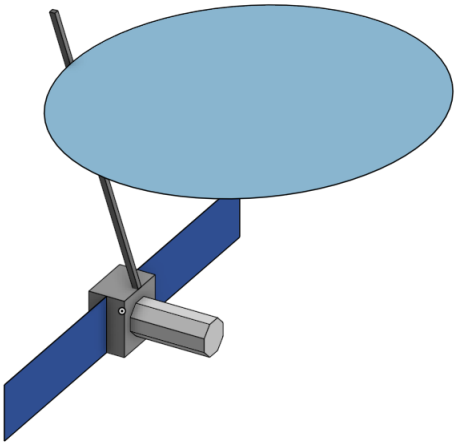


Figure 3: A 3D model of the simplified satellite geometry

We also plot the model in MATLAB by importing an STL version of the CAD model. We show the body axes in Figure 4, with the origin chosen as the center of the spacecraft bus.

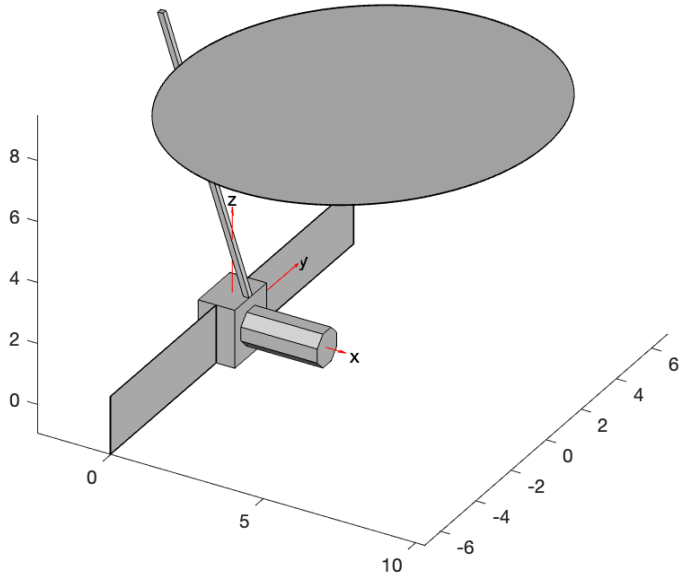


Figure 4: Satellite model in MATLAB with body axes shown

2.6 Principal Axes

The unit vectors of the principal axes with respect to the body axes ($\vec{e_i}$) and the inertia tensor in the principal axes (I_i) can be found by taking the eigenvalue decomposition of the inertia tensor in the body axis. This can be seen in the two equations below.

$$I_i \cdot \vec{e_i} = I_i \cdot \vec{I}_{body} \qquad i = x, y, z$$

$$\vec{I}_{principal} = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} = \begin{bmatrix} 7707.07 & 0 & 0 \\ 0 & 14563.2 & 0 \\ 0 & 0 & 18050.4 \end{bmatrix} \text{kg m}^2$$

We follow convention $I_z > I_y > I_x$ for defining principal axes.

The unit vectors of the principal axes ($\vec{e_i}$) can then be used to find the rotation matrix ($\vec{R}$), as shown below.

$$\vec{R} = \begin{bmatrix} \vec{e_x} & \vec{e_y} & \vec{e_z} \end{bmatrix} = \begin{bmatrix} -0.06278 & -0.99803 & 0 \\ 0 & 0 & 1 \\ -0.99803 & 0.06278 & 0 \end{bmatrix}$$

$$\vec{I}_{body} = \vec{R}\vec{I}_{principal}\vec{R}^{\mathsf{T}}$$

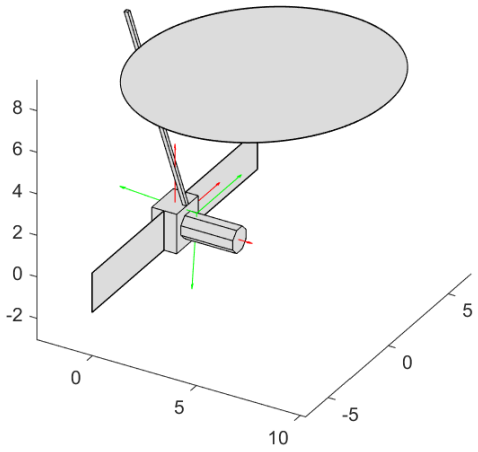


Figure 5: Principal axes at center of mass (green) and body axes at origin (red)

3 DYNAMICS

3.1 Orbit

From the science users' handbook, we obtain the following orbital elements [10].

OE	a	e	i	Ω	ω	ν
Value	7125.48662 km	0.0011650	98.40508°	-19.61601°	89.99764°	-89.99818°

We convert these using a MATLAB function into ECI coordinates that can be fed into a numerical orbital propagator. Notice that we first convert the orbital elements a, e, and ν into perifocal (PQW) coordinates, using a and e to find the semi-latus rectum and a, e, and ν to find the distance to the central body (Earth). Then, we perform a series of rotations on these coordinates parameterized by ω, i, and Ω to obtain new coordinates in the ECI frame.

Then, we can numerically propagate in MATLAB using `ode113` using a function that computes the time derivative of the ECI state. This is accomplished simply by setting the time derivative of position equal to the velocity portion of the state and setting the time derivative of velocity equal to an acceleration computed using the law of universal gravitation. Note that while our propagator does not include disturbance forces, it will be easy to incorporate these later. See the appendix corresponding to Problem Set 2 for application of `ode113`.

Now, we plot the trajectory for one orbit in Figure 6. Plotting multiple orbits (for example, over 12 days) yields the same plot, as `ode113` is very stable for this application.

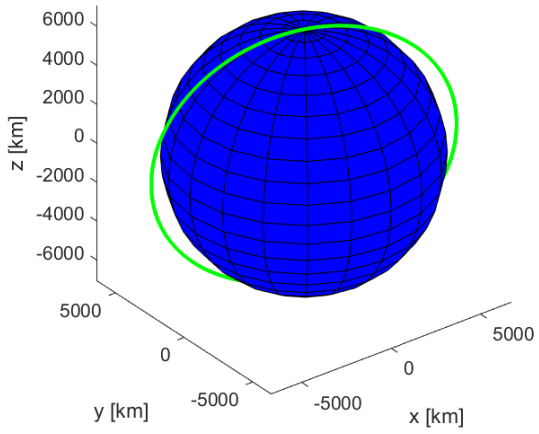


Figure 6: A single orbit for NISAR in ECI coordinates (no perturbations)

3.2 Euler Equations

Initially, we represent our dynamics using simplified Euler equations. We use the following equations with zero external moments ($M_x, M_y, M_z = 0$).

$$I_x\dot{\omega}_x + (I_z - I_y)\omega_y\omega_z = M_x$$
$$I_y\dot{\omega}_y + (I_x - I_z)\omega_z\omega_x = M_y$$
$$I_z\dot{\omega}_z + (I_y - I_x)\omega_x\omega_y = M_z$$

We choose arbitrary initial conditions $\omega_x = 8\,^\circ\,\mathrm{s}^{-1}$, $\omega_y = 4\,^\circ\,\mathrm{s}^{-1}$, and $\omega_z = 6\,^\circ\,\mathrm{s}^{-1}$. The results of numerical integration using `ode113` are shown in Figure 7.

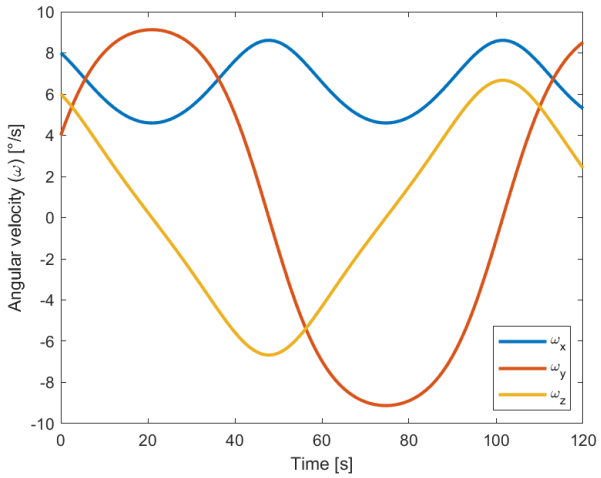


Figure 7: Results from numerical integration of Euler equations

3.3 Energy Ellipsoid

We use the energy ellipsoid to visualize our dynamics. We compute our surface using rotational kinetic energy based on initial conditions and principal axes inertia tensor.

$$2T = \omega_x^2 I_x + \omega_y^2 I_y + \omega_z^2 I_z$$
$$\frac{\omega_x^2}{2T/I_x} + \frac{\omega_y^2}{2T/I_y} + \frac{\omega_z^2}{2T/I_z} = 1$$

For the given initial conditions, we get semi-major axes of the following lengths: $\omega_x = 0.2332\,\mathrm{rad\,s}^{-1}$, $\omega_y = 0.1697\,\mathrm{rad\,s}^{-1}$, and $\omega_z = 0.1524\,\mathrm{rad\,s}^{-1}$. These values make sense given the equation for the energy ellipsoid.

Similarly, we compute our surface for the momentum ellipsoid with angular momentum based on our initial conditions and the principal axes inertia tensor.

$$L = \omega_x^2 I_x^2 + \omega_y^2 I_y^2 + \omega_z^2 I_z^2$$
$$\frac{\omega_x^2}{(L/I_x)^2} + \frac{\omega_y^2}{(L/I_y)^2} + \frac{\omega_z^2}{(L/I_z)^2} = 1$$

For the given initial conditions, we get semi-major axes of the following lengths: $\omega_x = 0.3115\,\mathrm{rad\,s}^{-1}$, $\omega_y = 0.1649\,\mathrm{rad\,s}^{-1}$, and $\omega_z = 0.1330\,\mathrm{rad\,s}^{-1}$. These values make sense given the equation for the momentum ellipsoid and are shown in the plots below.

We plot the energy ellipsoid in Figure 8 and the momentum ellipsoid in Figure 9.

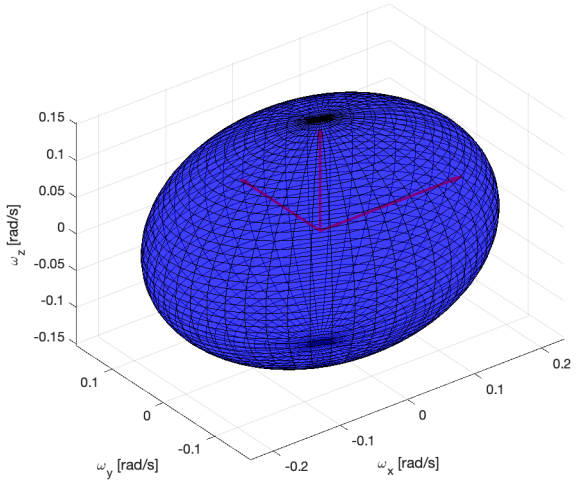


Figure 8: Energy ellipsoid with axes in red

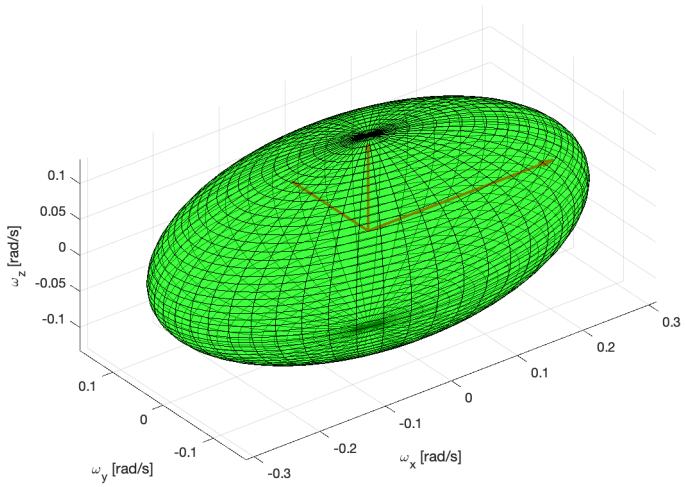


Figure 9: Momentum ellipsoid with axes in red

3.4 Polhode

We also show the polhode for our system. For a polhode plot to be real, the condition below must be verified.

$$I_x < \frac{L^2}{2T} < I_z$$

Based on previously calculated values ($I_x = 7707.1$, $\frac{L^2}{2T} = 13752.1$, $I_z = 18050.4$) we can verify that the polhode here will be real.

Figure 11 shows that the polhode is indeed the intersection between the ellipsoids.

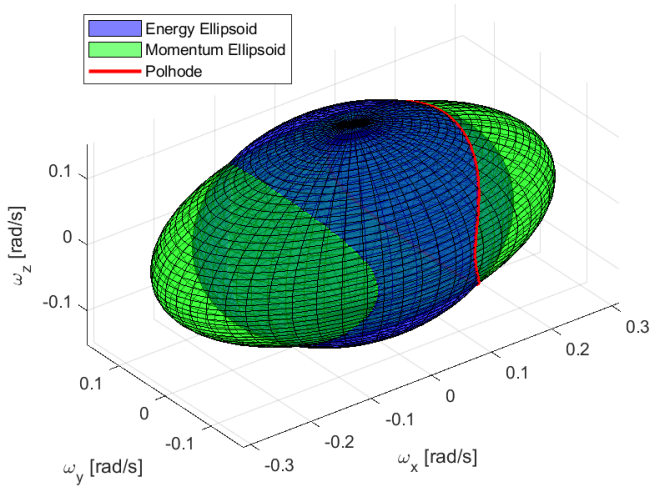


Figure 10: Energy and momentum ellipsoids with polhode

We also show polhode conic sections in Figure 11, which we find to match expected theory. The polhode as seen along the x-axis is an ellipse, while the polhode along the y-axis is a hyperbola. We also see that when seen along the z-axis, the polhode also forms an ellipse, shown as a half-ellipse in our plot.

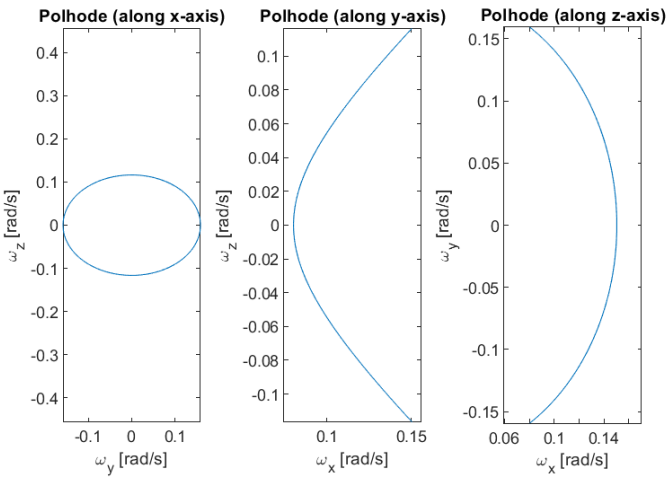


Figure 11: 2D views of polhode

Now, changing the initial conditions, we can observe changes in stability for different axes. We show the angular velocity evolution with the initial conditions shown in Table 6. Case 1 involves rotation about the principal x-axis, Case 2 involves rotation about the principal y-axis with a slight disturbance, and Case 3 involved rotation about the z-axis with a slight disturbance.

Table 6: Various initial conditions

Case	ω_x **(deg/s)**	ω_y **(deg/s)**	ω_z **(deg/s)**
1	8	0	0
2	0.08	8	0.08
3	0.08	0	8

The specifics of Case 1 are shown in the angular velocity plot in Figure 12, the polhode and ellipsoids in Figure 15, and the 2D views of the polhode in Figure 18. The behavior shown is as expected–when the angular velocity is parallel to the principal axis, we do not have coupling with the other components of angular velocity, and the polhode views in 2D become points rather than conic sections.

For Case 2, Figure 13 shows that the satellite's rotational behavior will oscillate as expected, owing to the properties of the intermediate axis. Additionally, Figure 16, and the 2D views in Figure 19 show a larger polhode, with the slight disturbances leading to ellipsoids with a substantial intersection. Interestingly, there seems to be a very sharp hyperbola in the xz-plane of the polhode.

Figure 14 illustrates a slight oscillation of angular velocities about the x- and y-axes in Case 3. Meanwhile, the actual region of intersection in the polhode as shown in Figures 17 and 20 is much smaller than in other cases, but not a single point like in the Case 1.

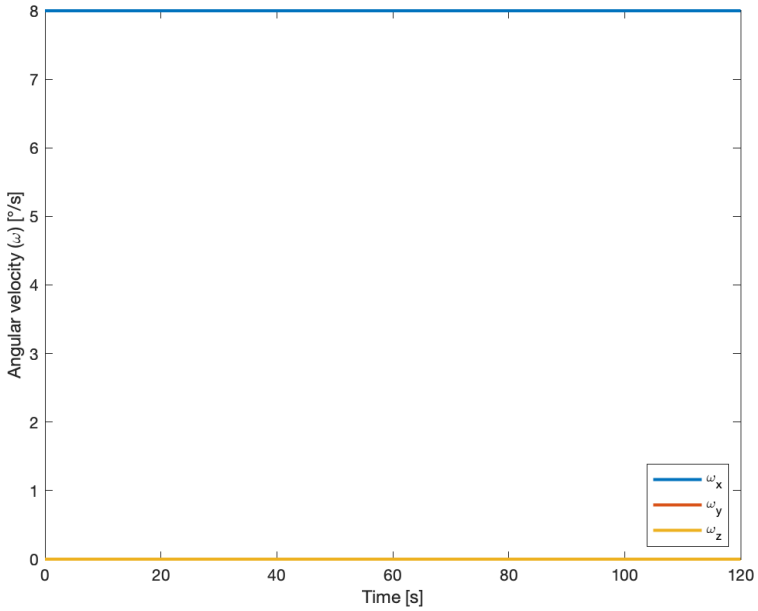


Figure 12: Angular velocity evolution for angular velocity vector for Case 1

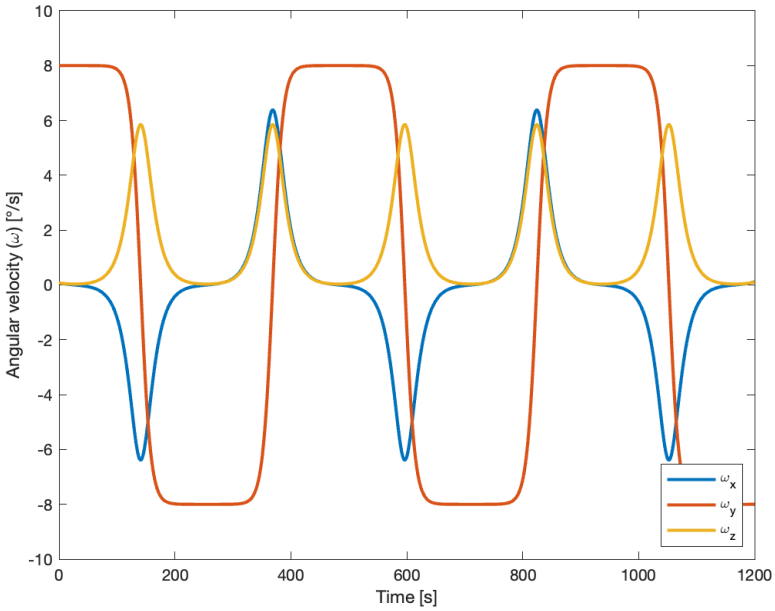


Figure 13: Angular velocity evolution for angular velocity vector for Case 2

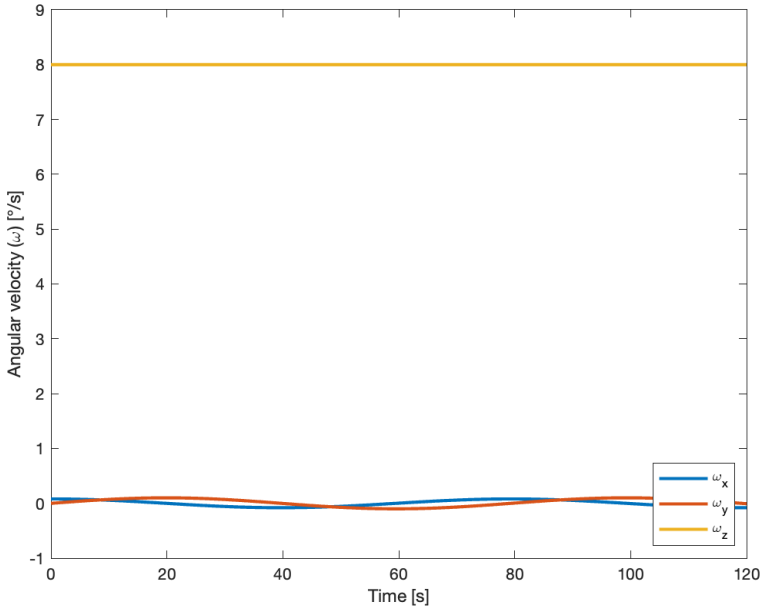


Figure 14: Angular velocity evolution for angular velocity vector for Case 3

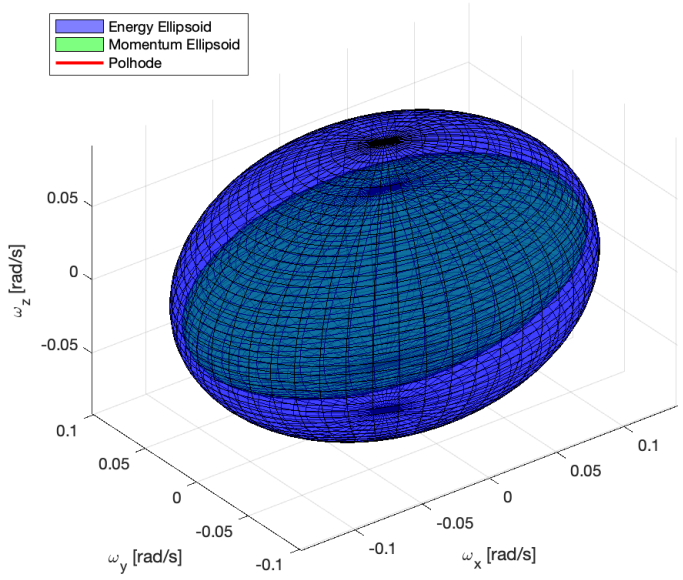


Figure 15: Polhode and ellipsoids for angular velocity vector for Case 1

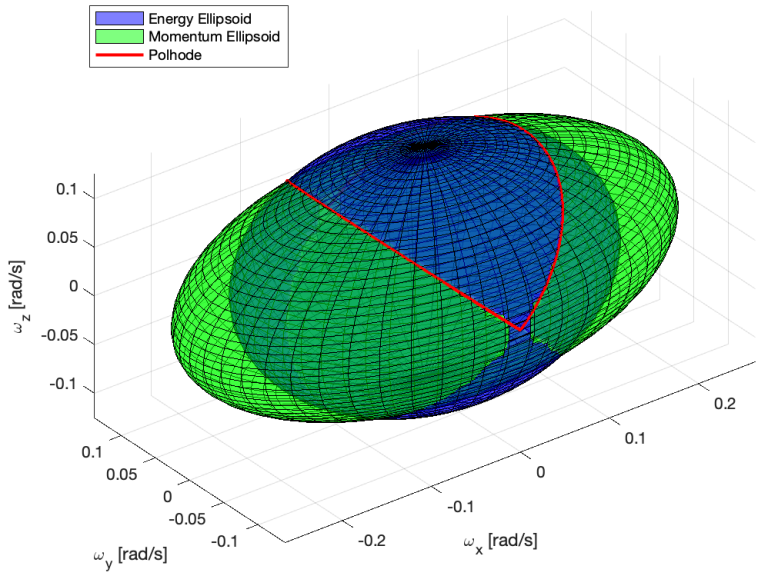


Figure 16: Polhode and ellipsoids for angular velocity vector for Case 2

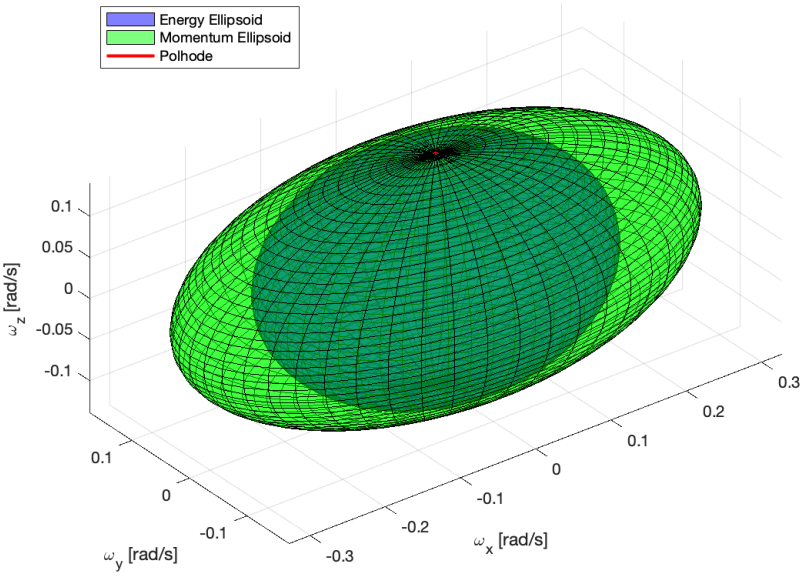


Figure 17: Polhode and ellipsoids for angular velocity vector for Case 3

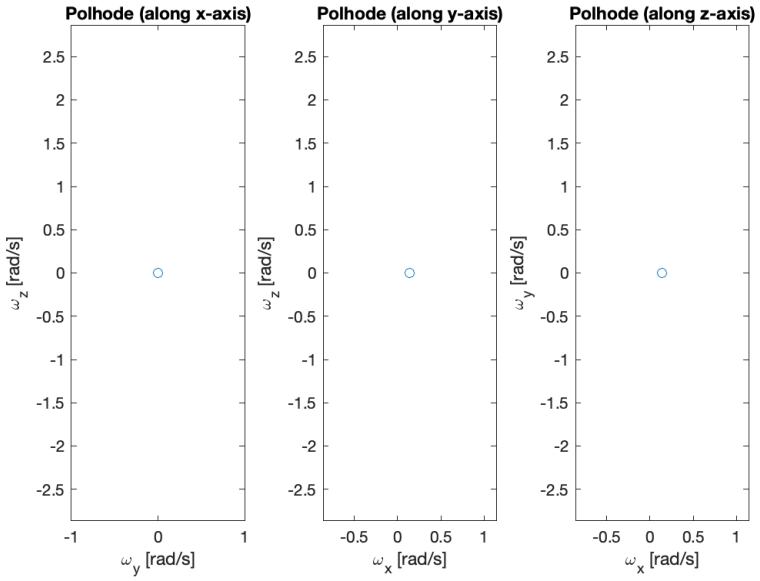


Figure 18: 2D views of polhode for angular velocity vector for Case 1

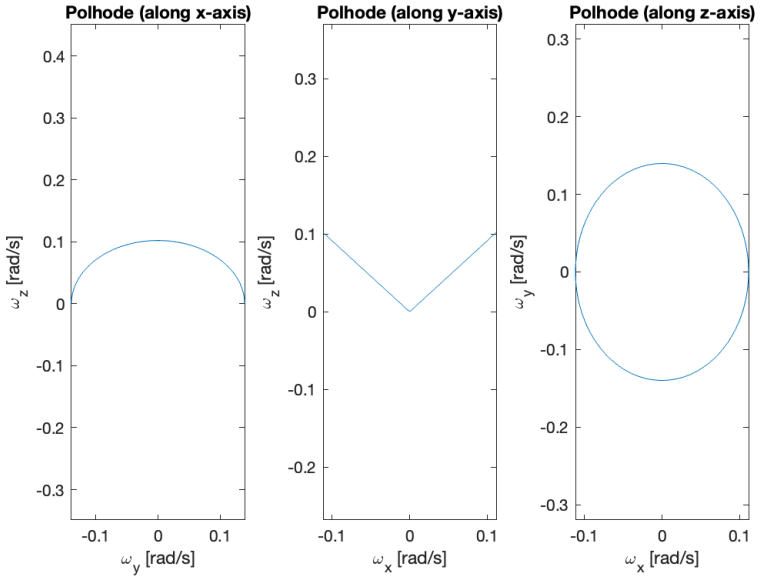


Figure 19: 2D views of polhode for angular velocity vector for Case 2

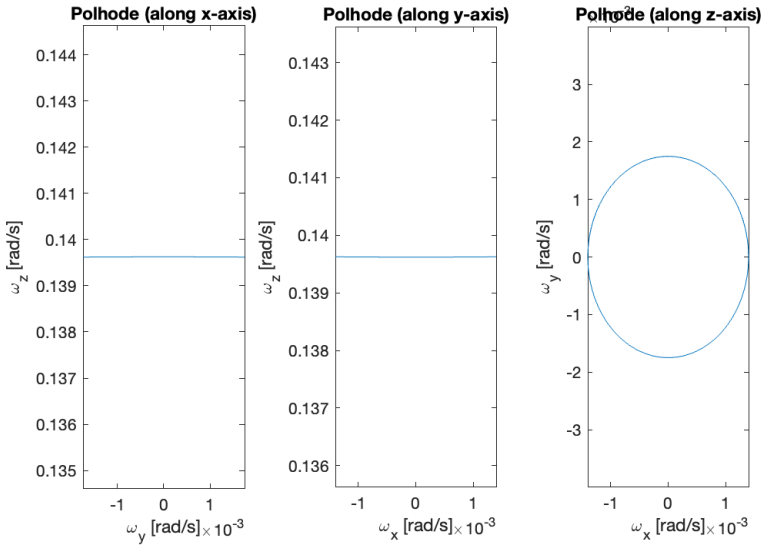


Figure 20: 2D views of polhode for angular velocity vector for Case 3

3.5 Axisymmetric Satellite

For an axisymmetric satellite, we set $I_x = I_y = 7707.07\,\mathrm{kg}\cdot\mathrm{m}^2$ and use the same Euler equation solver from before with the same initial conditions ($\omega_x = 8\,^\circ\,\mathrm{s}^{-1}$, $\omega_y = 4\,^\circ\,\mathrm{s}^{-1}$).

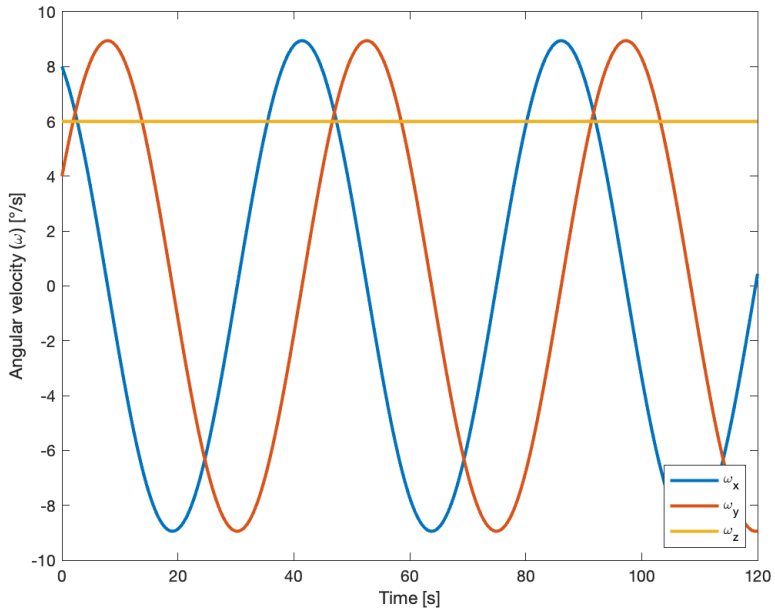


Figure 21: Numerical solution results

The analytical solution to the Euler equations for an axial-symmetric satellite is based on variables λ and ω_{xy}, as defined below.

$$\lambda = \frac{I_z - I_x}{I_x}\omega_{z_0}$$
$$\omega_{xy} = (\omega_{x_0} + i\omega_{y_0})e^{i\lambda t}$$

We take the real and imaginary parts of this result to obtain an analytical solution.

$$\omega_x = \mathrm{Re}(\omega_{xy})$$
$$\omega_y = \mathrm{Im}(\omega_{xy})$$
$$\omega_z = \omega_{z_0}$$

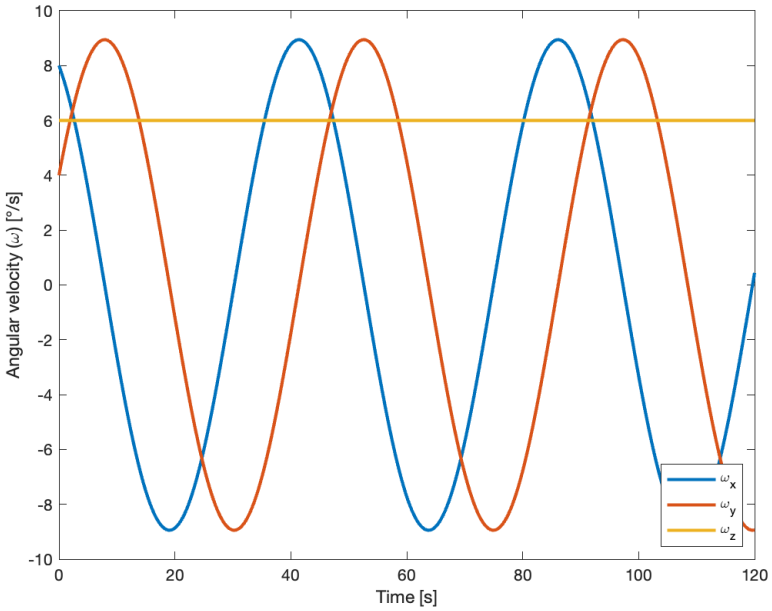


Figure 22: Analytical solution results

Figure 23 is the error between the numerical and analytical solutions. We observe that the error is very small, thus our numerical solution is a good candidate.

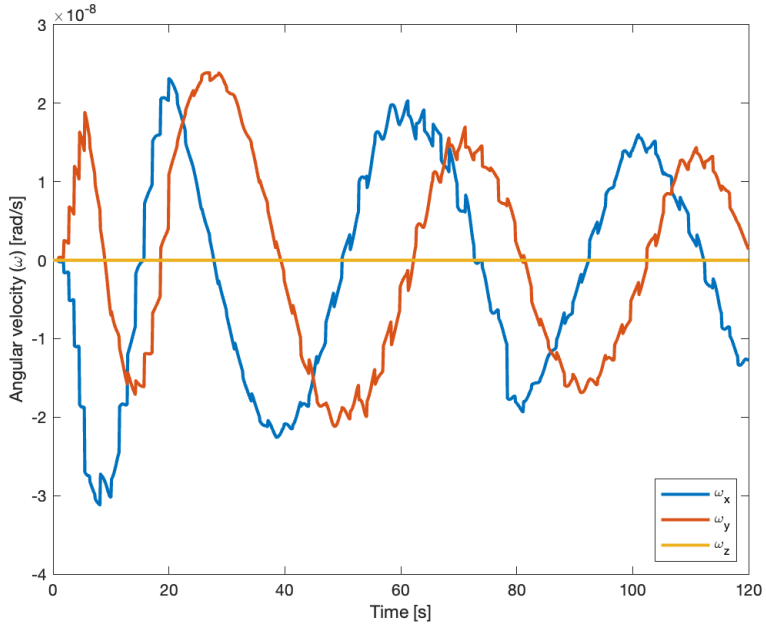


Figure 23: Error between numerical and analytical solutions

The angular velocity vector and angular momentum vectors rotate in a plane, offset at a constant angle from the z-axis, as observed in Figure 24. This matches the expected theoretical behavior.

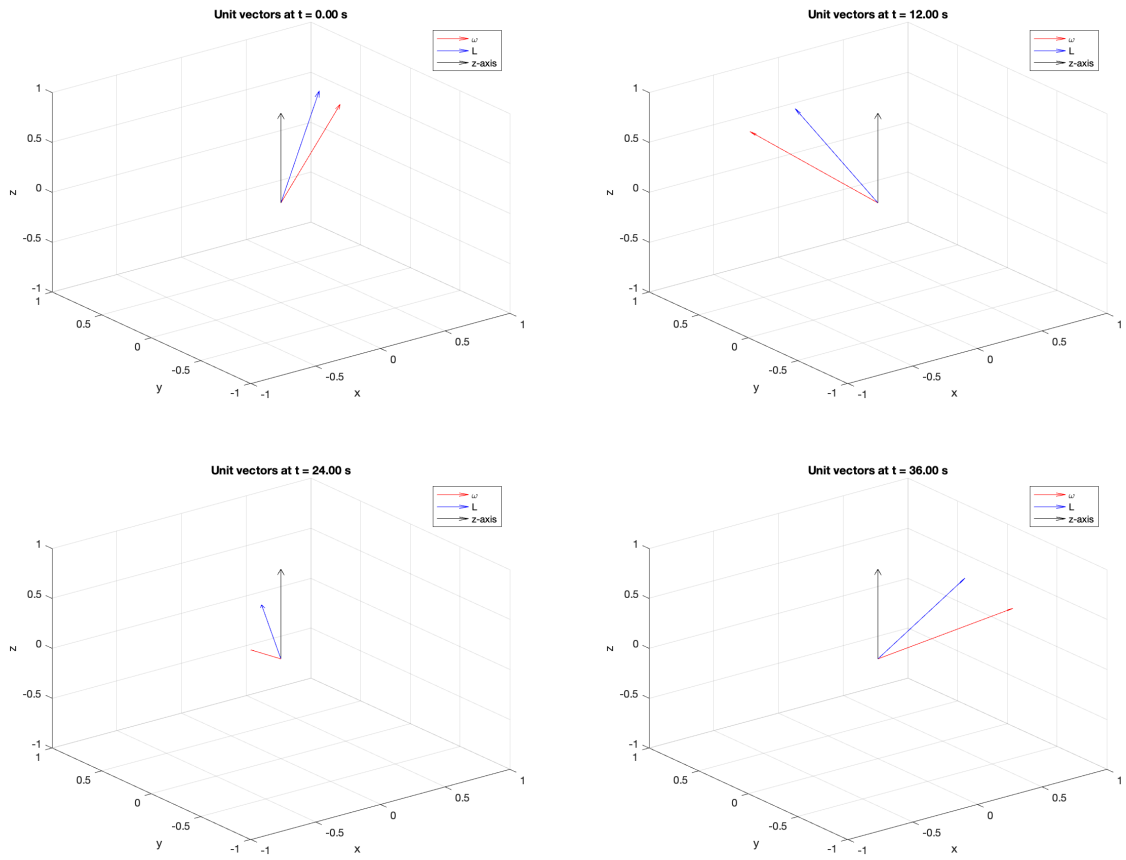


Figure 24: Angular velocity (red) and angular momentum (blue) unit vectors over time.

4 KINEMATICS

4.1 Quaternions

We choose a nominal attitude parameterization of quaternions, our choice being based on the absence of singularities. The following function computes the time derivative for a state consisting of quaternions (4 parameters) and angular velocity (3 parameters).

The equations below describe the propagation of kinematics using quaternions.

$$\vec{\Omega} = \begin{bmatrix} 0 & \omega_z & -\omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix}$$

$$\frac{d\vec{q}}{dt} = \frac{1}{2}\vec{\Omega}\vec{q}(t)$$

The following script shows the computation of the time derivative for quaternions

```
function stateDot = kinQuaternion(t,state,Ix,Iy,Iz)
    % Computes state derivatives for quaternions, angular velocity
    % Assign variables
    q = state(1:4);
    wx = state(5);
    wy = state(6);
    wz = state(7);

    stateDot = zeros(7,1);
    % Angular velocity time derivatives
    stateDot(5) = (Iy - Iz) / Ix * wy * wz;
    stateDot(6) = (Iz - Ix) / Iy * wz * wx;
    stateDot(7) = (Ix - Iy) / Iz * wx * wy;
    sigma = [0, wz, -wy, wx; ...
        -wz, 0, wx, wy; ...
        wy, -wx, 0, wz; ...
        -wx, -wy, -wz, 0];
    % Quaternion time derivative
    qDot = 0.5 * sigma * q;
    stateDot(1:4) = qDot;
end
```

We can use the previous function to perform a forward Euler numerical integration. We call the previous function over a fixed time step to compute the evolution of the state.

```
function [q,w] = kinQuaternionForwardEuler(q0,w0,Ix,Iy,Iz,tFinal,tStep)
    % Forward Euler integration for quaternions, angular velocity
    nStep = ceil(tFinal/tStep);
    q = nan(nStep+1,4);
    w = nan(nStep+1,3);
    q(1,:) = q0';
    w(1,:) = w0';
    for i = 1:nStep
        t = i * tStep;
        qi = q(i,:)';
```

```
        wi = w(i,:)';
        state = [qi;wi];
        stateDot = kinQuaternion(t,state,Ix,Iy,Iz);
        nextState = state + tStep * stateDot;
        q(i+1,:) = nextState(1:4) / norm(nextState(1:4));
        w(i+1,:) = nextState(5:7);
    end
end
```

For improved precision, we implement a 4th order Runge-Kutta method, which uses a weighted sum of slopes to obtain a better result. This also calls the time derivative function, but does so with different values of the state, which are weighted to obtain the next state for each step.

4.2 Euler Angles

Similarly, we create a function that computes the time derivative of a state consisting of Euler angles and angular velocity using the 3-1-3 symmetric Euler angle sequence.

The equations for the propagation of kinematics for Euler angles are below.

$$\frac{d\phi}{dt} = \frac{\omega_x \sin(\psi) + \omega_y \cos(\psi)}{\sin(\theta)}$$
$$\frac{d\theta}{dt} = \omega_x \cos(\psi) - \omega_y \sin(\psi)$$
$$\frac{d\psi}{dt} = \omega_z - (\omega_x \sin(\psi) + \omega_y \cos(\psi))cot(\theta)$$

```
function stateDot = kinEulerAngle(t,state,Ix,Iy,Iz)
    % Computes state derivative for Euler angles, angular velocity
    % Assign variables
    phi = state(1);
    theta = state(2);
    w = state(4:6);

    stateDot = zeros(6,1);
    % Angular velocity time derivatives
    stateDot(4) = (Iy - Iz) / Ix * w(2) * w(3);
    stateDot(5) = (Iz - Ix) / Iy * w(3) * w(1);
    stateDot(6) = (Ix - Iy) / Iz * w(1) * w(2);
    % Euler angle time derivatives
    % 312
    EPrimeInv = [sin(phi)*sin(theta) cos(phi)*sin(theta) cos(theta); ...
        cos(theta)*cos(phi) -sin(phi)*cos(theta) 0; ...
        sin(phi) cos(phi) 0] * (1 / cos(theta));
    % 313
    % EPrimeInv = [-sin(phi)*cos(theta) -cos(phi)*cos(theta) ...
       sin(theta); ...
    %     cos(phi)*sin(theta) -sin(phi)*sin(theta) 0; ...
    %     sin(phi) cos(phi) 0] * (1 / sin(theta));
    stateDot(1:3) = EPrimeInv * w;
end
```

We can propagate this with forward Euler, as in the previous section.

```
function [state] = ...
    kinEulerAngleForwardEuler(state0,Ix,Iy,Iz,tFinal,tStep)
    % Forward Euler integration for state Euler angles, angular velocity
    nStep = ceil(tFinal/tStep);
    state = nan(nStep+1,6);
    state(1,:) = state0;
    for i = 1:nStep
        t = i * tStep;
        statei = state(i,:)';
        stateDot = kinEulerAngle(t,statei,Ix,Iy,Iz);
        state(i+1,:) = statei + tStep * stateDot;
    end
end
```

For our actual implementation, we choose to use the time derivative function with `ode113` for improved accuracy. Note that this cannot be done as simply for quaternions, as they require normalization at each step, hence our decision to implement RK4.

4.3 Integration of Attitude Parameterizations

We integrate our attitude parameterizations (including angular velocity using Euler equations).

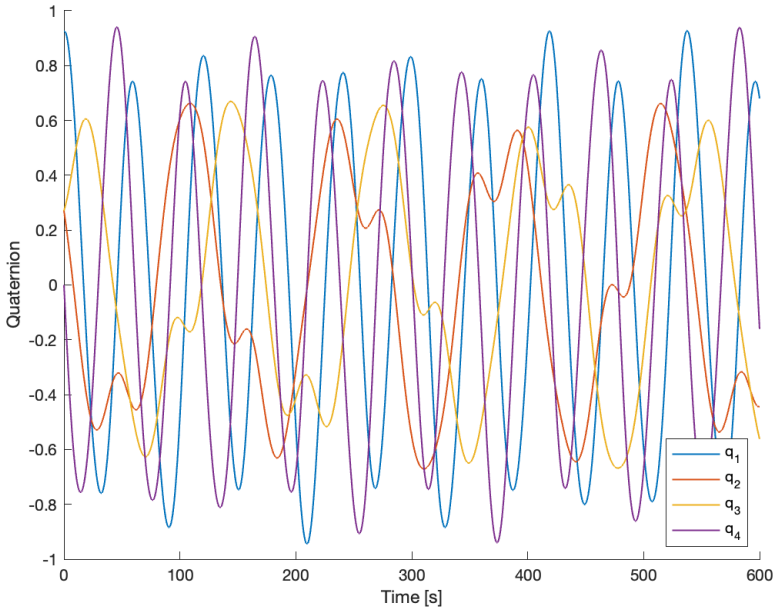


Figure 25: Evolution of quaternions

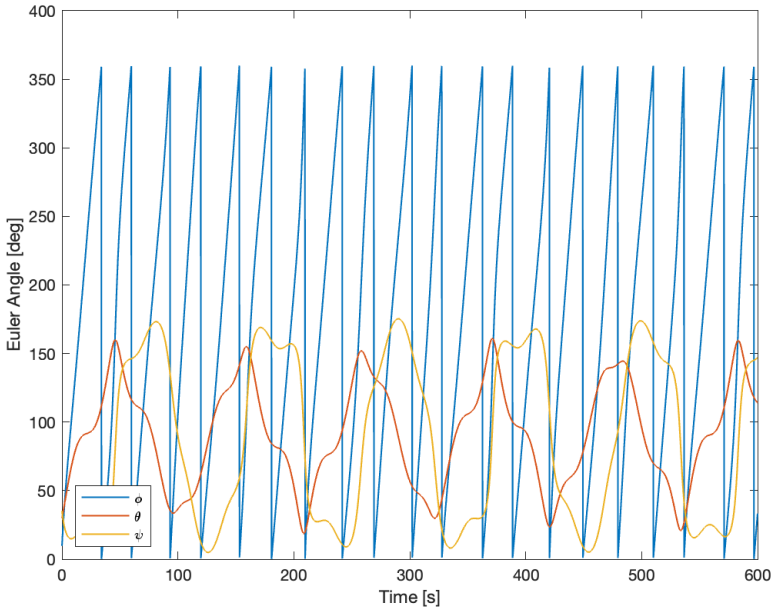


Figure 26: Evolution of Euler angles

4.4 Angular Momentum Vector

Figure 27 shows the components of the angular momentum vector over time, as computed from our primary attitude representation of quaternions. The angular momentum vector (and its individual components) remains constant in the absence of external torques.

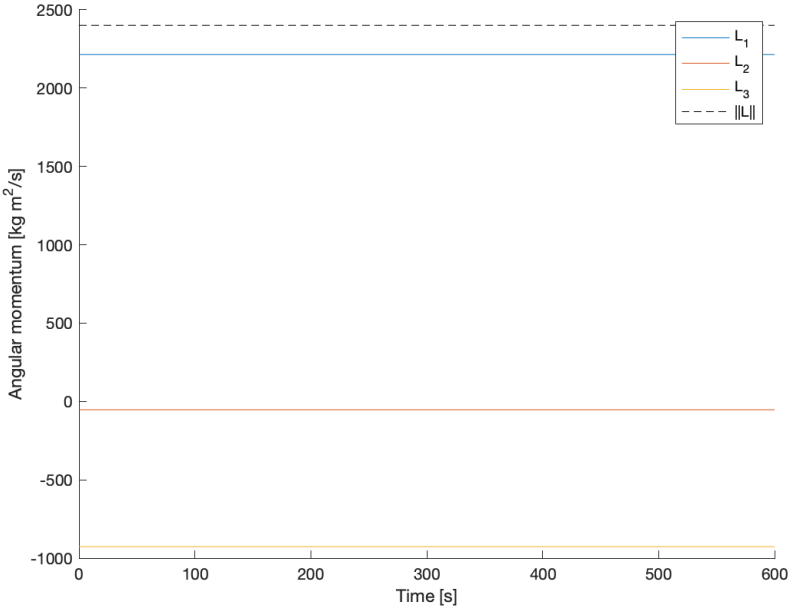


Figure 27: Angular momentum in inertial coordinates is constant

Figure 28 shows the angular momentum and angular velocity vectors overlaid with the herpolhode. The animation (see caption) shows the evolution of the herpolhode and provides a better

visualization of the herpolhode's geometry.

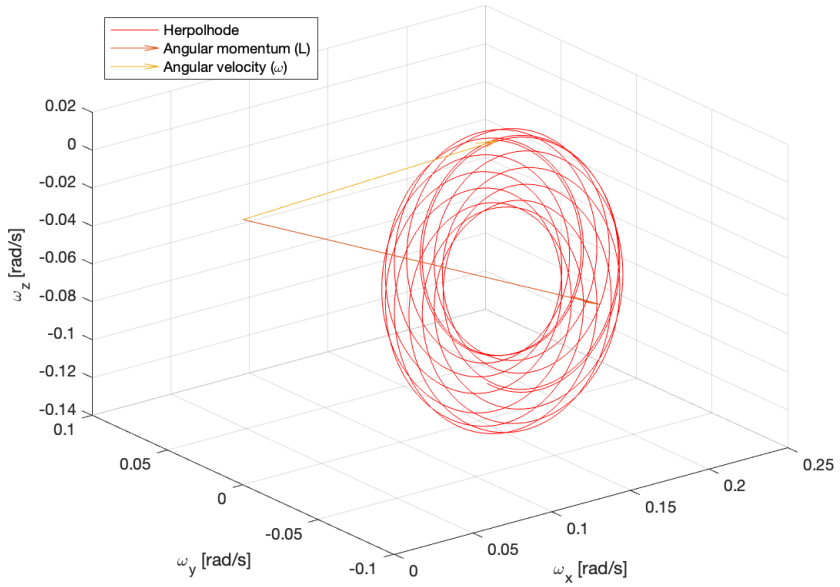


Figure 28: Herpolhode (Animated: `https://tinyurl.com/herpolhode`)

Figures 29, 30, and 31 include the plots of the orbital (RTN), body, and principal axes over the course of a single orbit. The RTN frame varies with rotation about the orbit, while rotation can be seen in the body and principal axis plots based on the rotation in our initial condition.

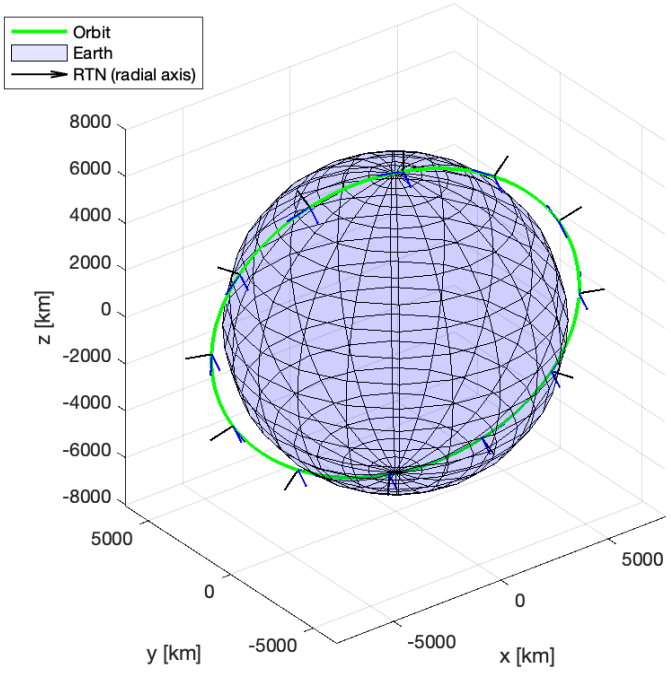


Figure 29: Propagation of RTN frame

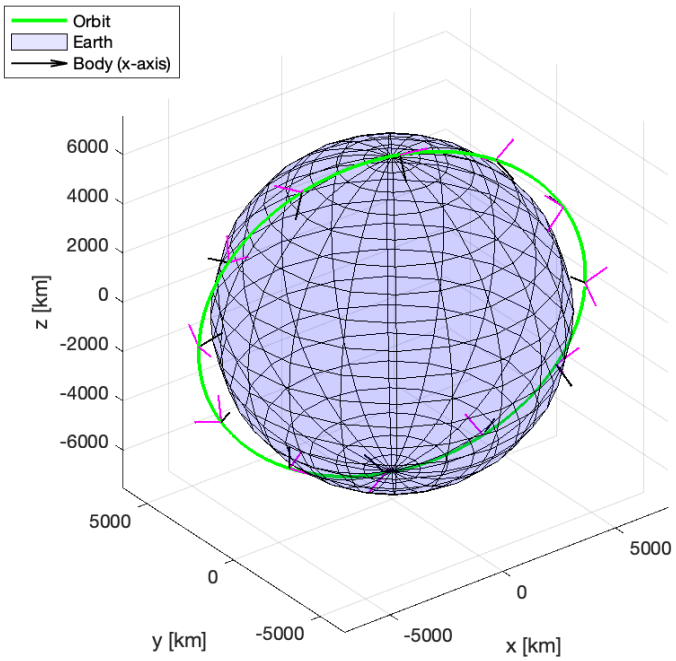


Figure 30: Propagation of body axes

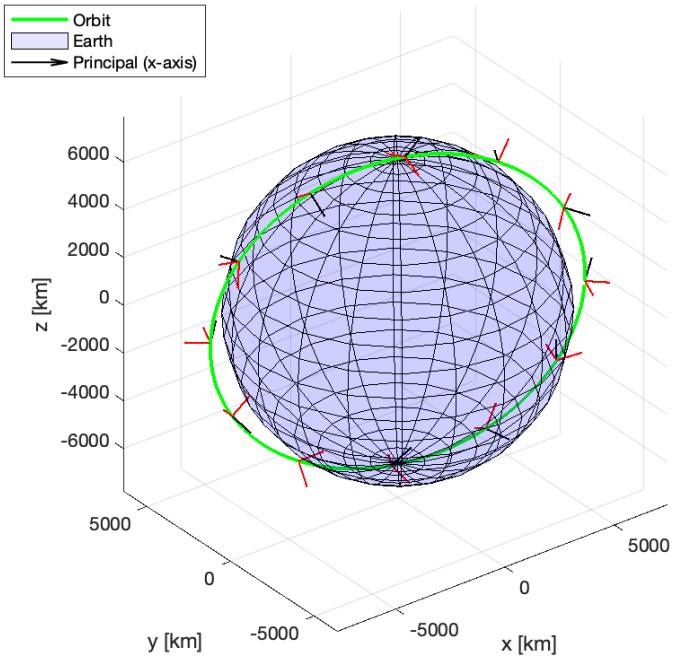


Figure 31: Propagation of principal axes

4.5 Stability, Inertial

We assume that two components of the initial angular velocity are zero and that the principal axes are aligned with the inertial frame. This is an equilibrium state. To test the equilibrium state, we choose the following initial angular velocity,

$$\vec{\omega} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{ rad s}^{-1},$$

and we set all Euler angles to zero. We use a 312 convention for Euler angles, which avoids singularities for this configuration.

Figure 32 shows results of the simulation, where ω_x and ω_y remain zero and ω_z maintains a constant value.

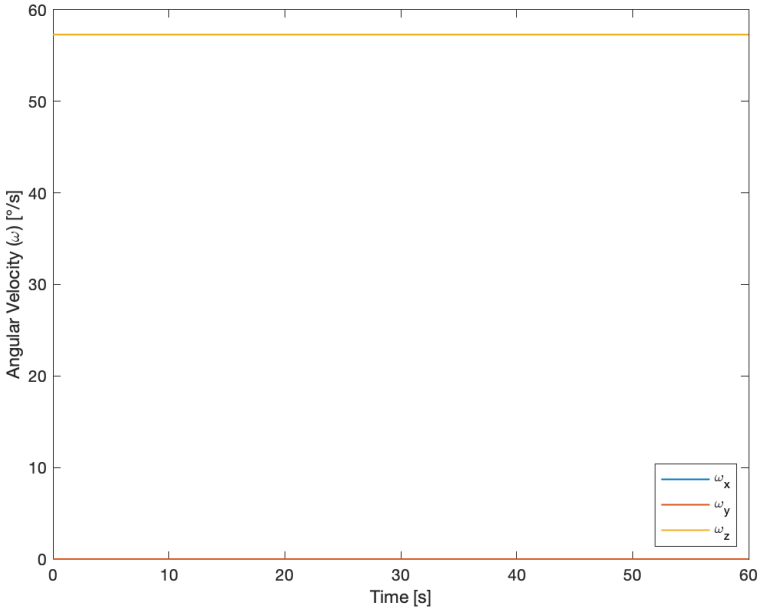


Figure 32: Evolution of angular velocity

Similarly, in Figure 33, ϕ and θ Euler angles remain at zero while the ψ Euler angle increases linearly.

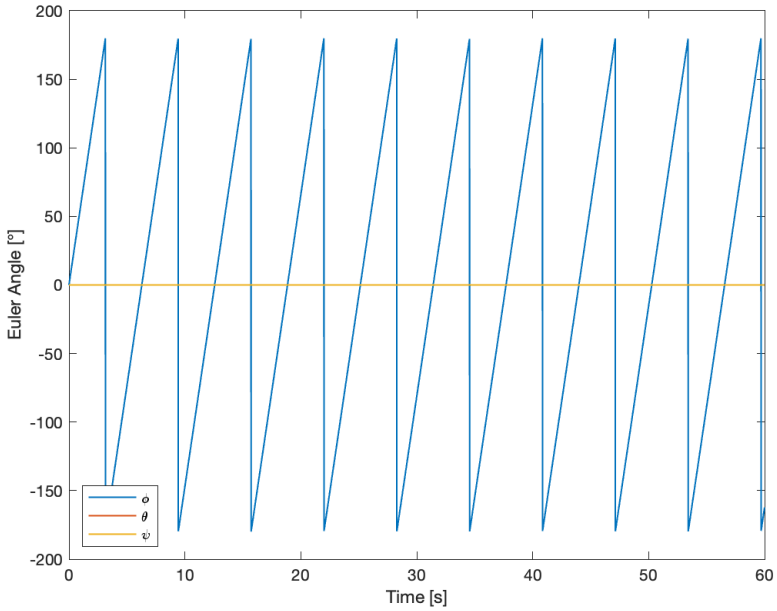


Figure 33: Evolution of Euler angles

4.6 Stability, RTN

Now, we choose to align our principal axes with the RTN frame. For selected initial orbital conditions taken from the NISAR science users' handbook, we obtain the initial position and compute the RTN frame, which we then use to find initial aligned Euler angles.

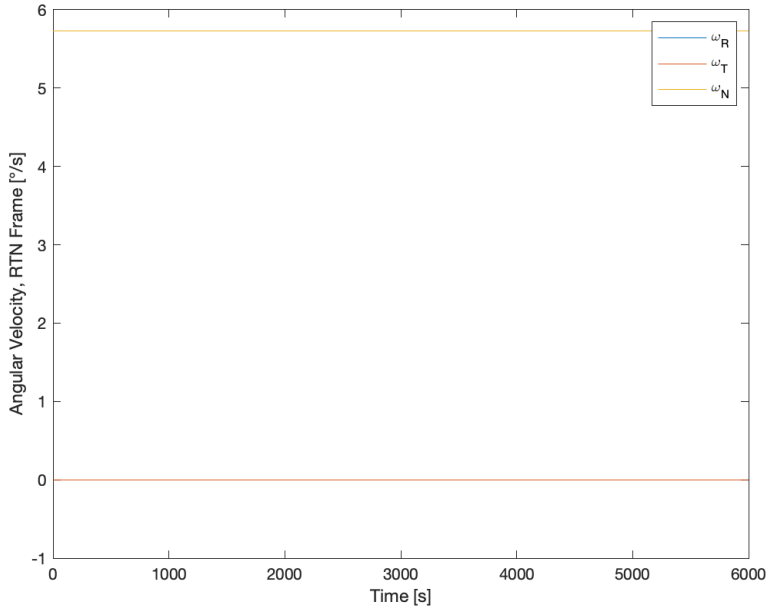


Figure 34: Evolution of angular velocity

We set a nonzero ω_z angular velocity, which is aligned with the normal direction of the RTN frame, and we choose all other angular velocities to be zero. Propagating the orbit and attitude,

we find that the angular velocity remains constant throughout the orbit, even relative to the RTN frame, as seen in Figure 34. From Figure 35, we see that the θ and ψ Euler angles related to the RTN frame are constant while ϕ varies.

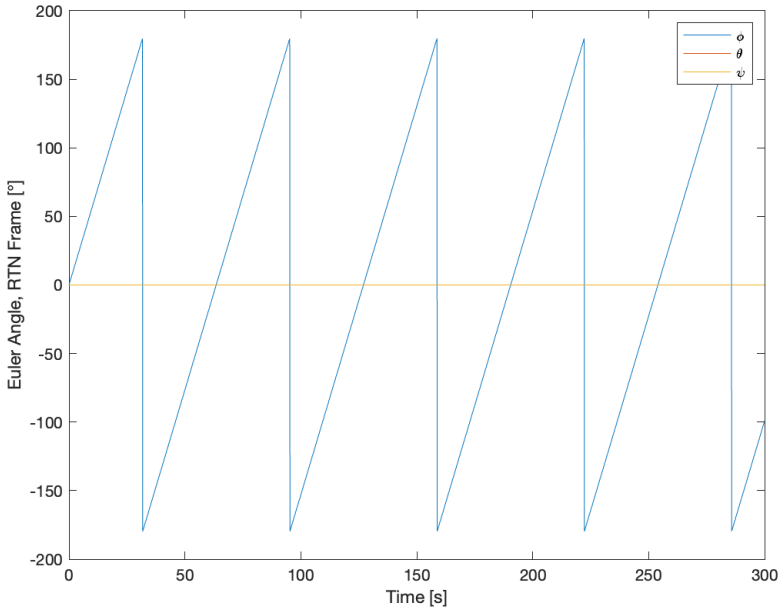


Figure 35: Evolution of Euler angles

This result shows our satellite's maximum inertia principal axis remains aligned with the normal direction of the orbit such that the maximum inertia principal axis remains normal to the plane of the orbit, given the initial condition that axes are aligned with the RTN frame and angular velocity along other axes is nonzero.

4.7 Stability, Single-Spin

For a single spin satellite, the three possible equilibrium configurations are rotation about the minimum inertia principal axis, rotation about the intermediate axis, and rotation about the maximum inertia principal axis. Figures 36, 37, 38 show that the Euler angles are stable about the minimum and maximum axes, but it is unstable about the intermediate axes. This is as expected for our system, with the minimum and maximum axes exhibiting periodic stability with small oscillations in angular velocity.

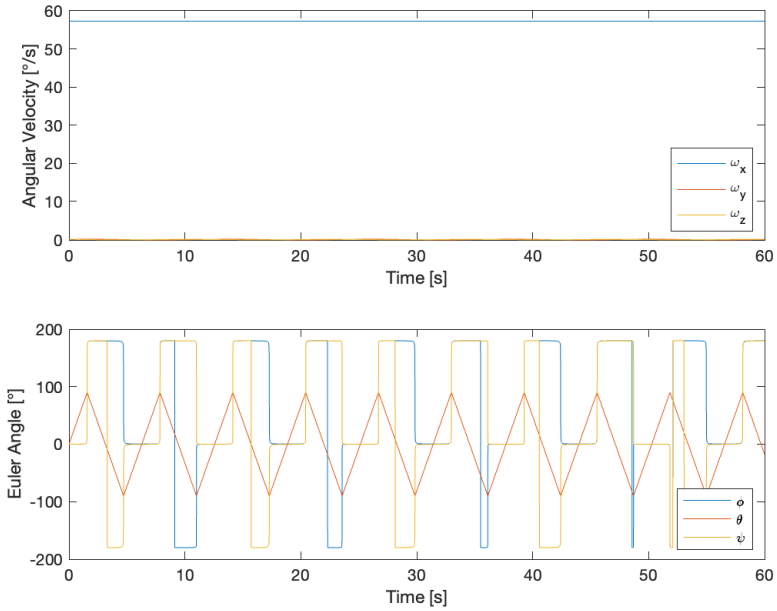


Figure 36: Simulation of satellite spinning on its minimum principal axis

Note that while in Figure 36 the angular velocities are periodically stable, the Euler angles oscillate. This is likely a consequence of the sequence of rotations used for our choice of Euler angle convention.

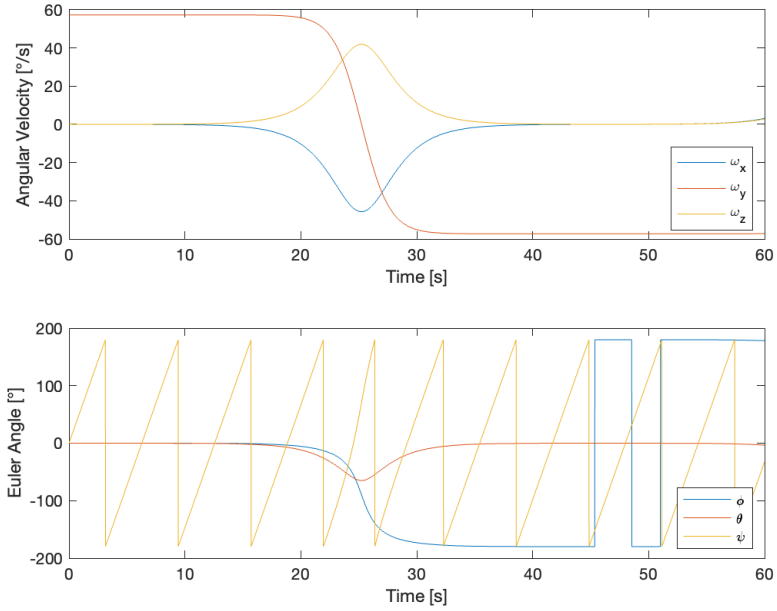


Figure 37: Simulation of satellite spinning on its intermediate principal axis

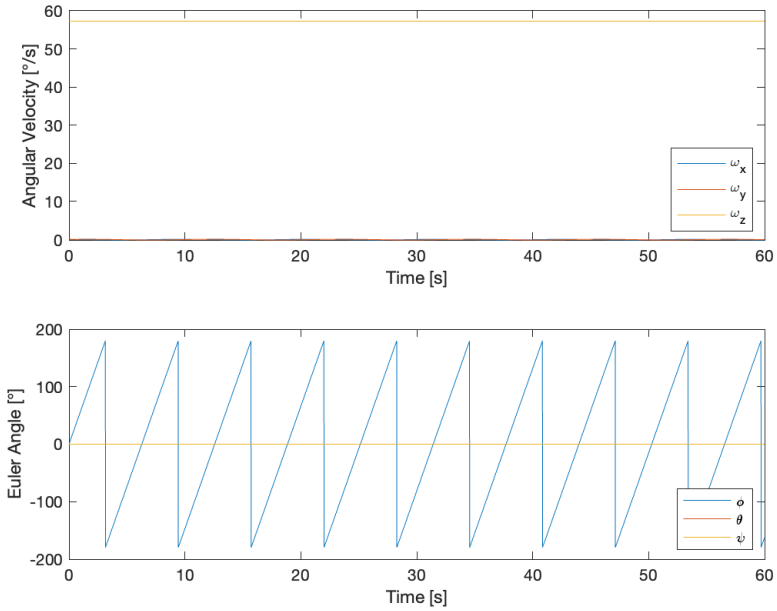


Figure 38: Simulation of satellite spinning on its maximum principal axis

4.8 Stability, Dual-Spin

We select a momentum wheel similar to our mission specifications to model a dual-spin satellite.

We choose to use specifications from the RSI 68 momentum wheel, for which datasheets are readily available online [11]. This particular momentum wheel is intended for spacecraft in the 1,500 to 5,000 kg range, which matches our mission. We use a diameter of 347 mm and mass of 8.9 kg and model our reaction wheel as a hoop with mass concentrated about the outer diameter. We also use 2,500 RPM, which yields approximately the nominal angular momentum from the datasheet–the maximum angular velocity is 6,000 RPM. The following Euler equations are used to model a momentum wheel as a rotor. For this problem, we set the torques on the right side of each equation to zero, as we are not considering external torques.

$$
\begin{aligned}
I_x\dot{\omega_x} + I_r\dot{\omega_r}r_x + (I_z - I_y)\omega_y\omega_z + I_r\omega_r(\omega_y r_z - \omega_z r_y) &= M_x \\
I_y\dot{\omega_y} + I_r\dot{\omega_r}r_y + (I_x - I_z)\omega_z\omega_x + I_r\omega_r(\omega_z r_x - \omega_x r_z) &= M_y \\
I_z\dot{\omega_z} + I_r\dot{\omega_r}r_z + (I_y - I_x)\omega_x\omega_y + I_r\omega_r(\omega_x r_y - \omega_y r_x) &= M_z \\
I_r\dot{\omega_r} &= M_r
\end{aligned}
$$

The function `kinEulerAngleWheel`, shown below, is used in addition to `ode113` to simulate the angular velocities over time.

```
function stateDot = kinEulerAngleWheel(t,state,M,r,Ix,Iy,Iz,Ir)
    % Computes state derivative for Euler angles, angular velocity
    % Adds momentum wheel
    % Assign variables
    phi = state(1);
    theta = state(2);
    w = state(4:7);

```

```
    stateDot = zeros(7,1);
    % Angular velocity time derivatives
    wDot = eulerEquationWheel(t,w,M,r,Ix,Iy,Iz,Ir);
    stateDot = zeros(7,1);
    stateDot(4) = wDot(1);
    stateDot(5) = wDot(2);
    stateDot(6) = wDot(3);
    stateDot(7) = wDot(4);
    % Euler angle time derivatives
    % 312
    EPrimeInv = [sin(phi)*sin(theta) cos(phi)*sin(theta) cos(theta); ...
        cos(theta)*cos(phi) -sin(phi)*cos(theta) 0; ...
        sin(phi) cos(phi) 0] * (1 / cos(theta));
    % 313
    % EPrimeInv = [-sin(phi)*cos(theta) -cos(phi)*cos(theta) ...
       sin(theta); ...
    %     cos(phi)*sin(theta) -sin(phi)*sin(theta) 0; ...
    %     sin(phi) cos(phi) 0] * (1 / sin(theta));
    stateDot(1:3) = EPrimeInv * w(1:3);
end
```

Simulating the Euler and kinematic equations, Figure 39 shows the angular momentum vector remain constant in the inertial frame, as expected.

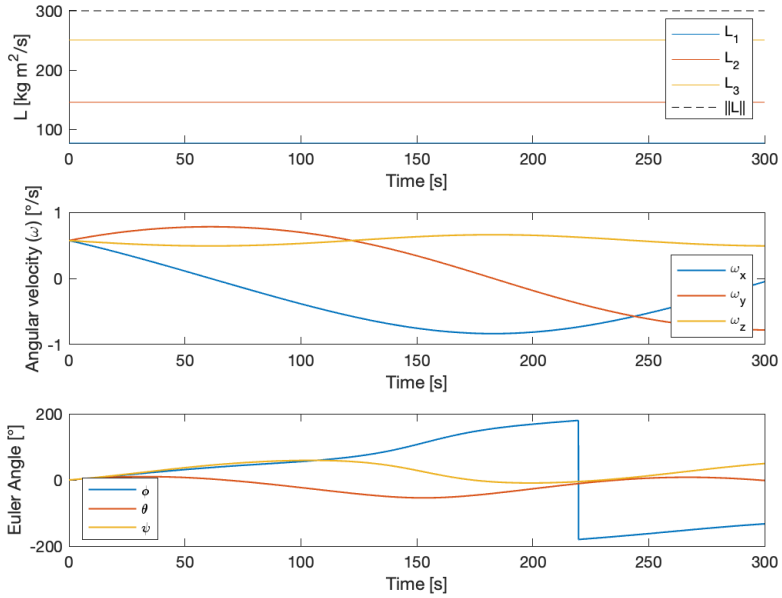


Figure 39: Angular momentum conserved with angular momentum components constant

By linearizing the Euler equations about equilibrium, the following result shows that periodic stability (in this case, about the z-axis) can be met with a reaction wheel if one of the following conditions are met:

$$1)\ I_r\omega_r > (I_y - I_z)\omega_z \quad AND \quad I_r\omega_r > (I_x - I_z)\omega_z$$
$$2)\ I_r\omega_r < (I_y - I_z)\omega_z \quad AND \quad I_r\omega_r < (I_x - I_z)\omega_z$$

We demonstrate equilibrium and stability for this new system with the reaction wheel. Figures 40, 41, 42 show the analysis for each of the principal axes. As before, the minimum and maximum inertia principal axes are periodically stable, but the intermediate axis is unstable.

This behavior resembles that found previously. In this case, we perturb each angular velocity as well as the rotor angular velocity. Our rotor velocity in this case is not enough to stabilize spin about the intermediate axis.

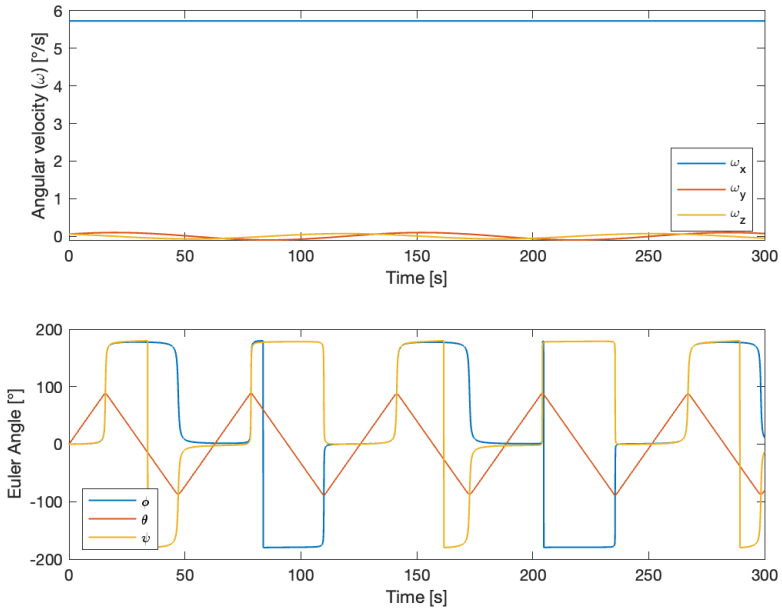


Figure 40: Stability analysis about minimum inertia principal axis

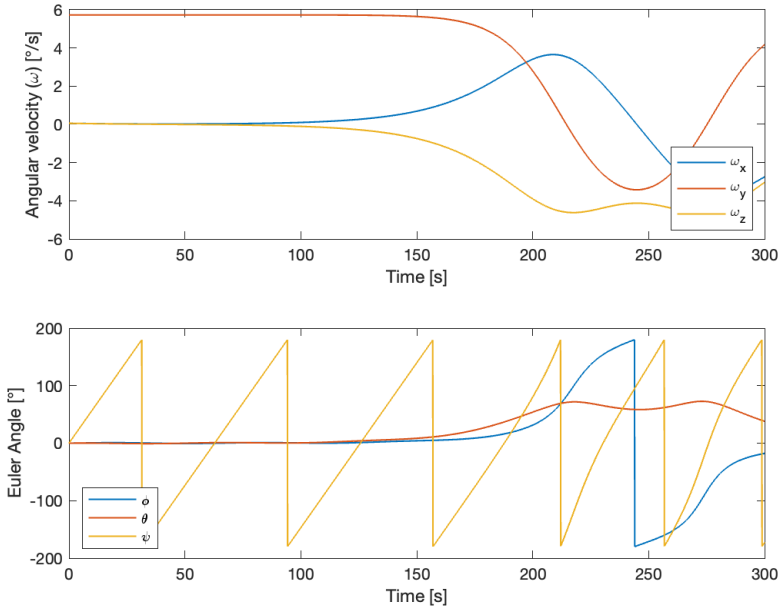


Figure 41: Stability analysis about intermediate axis

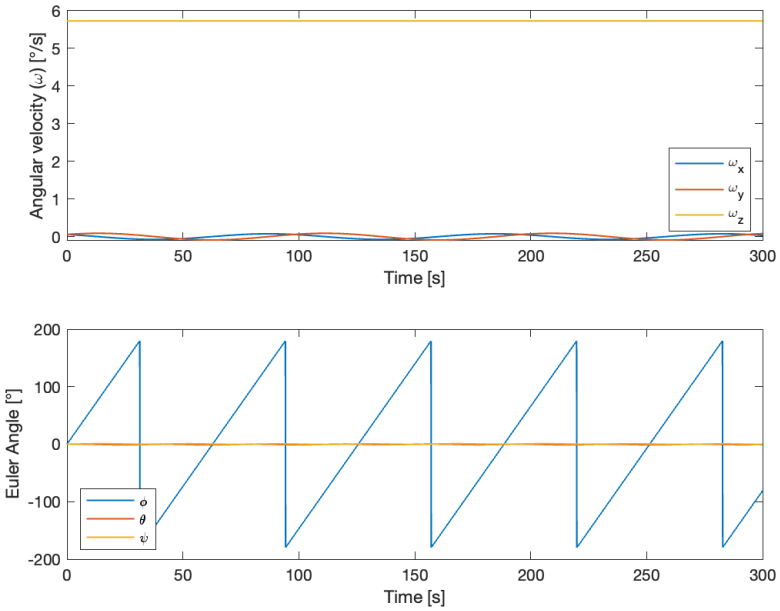


Figure 42: Stability analysis about maximum inertia principal axis

Using the stability condition, we can make the attitude motion stable. In our case, increasing the angular velocity of the rotor by a factor of 10 obtains a stable system for spin about the intermediate axes.

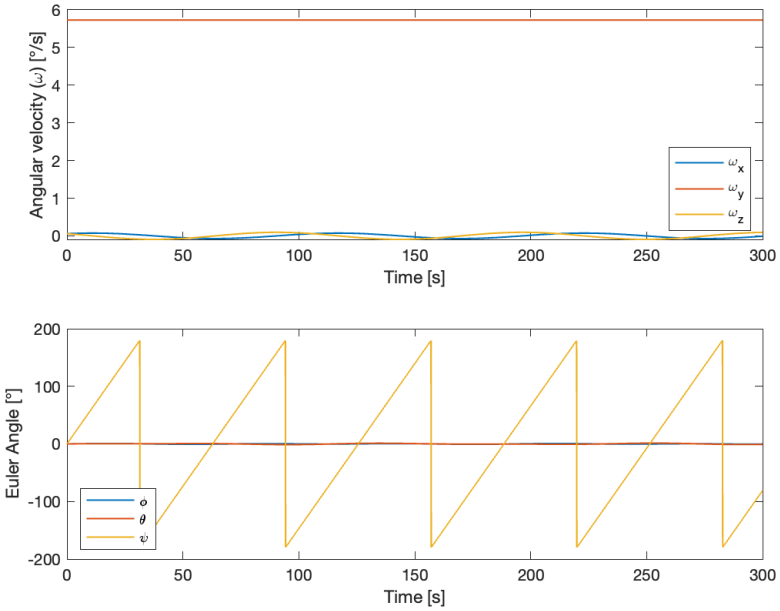


Figure 43

For the purpose of our mission, we choose to stabilize spin about the body x-axis, which is important for pointing our satellite and appropriately sweeping the target with our SAR. We use the rotation previously found between the principal and body axes to achieve this, applying the

rotation to the angular velocity and the angular momentum vector direction of the momentum wheel.

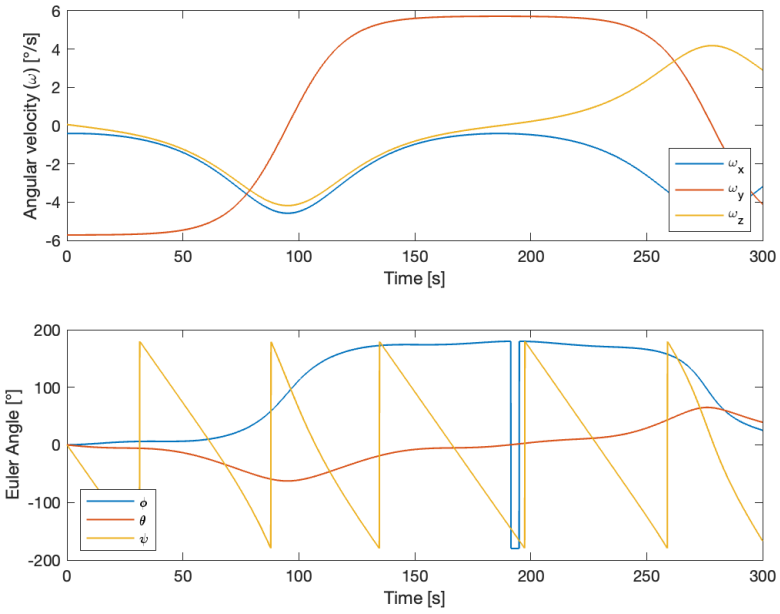


Figure 44: Initially unstable attitude with low momentum wheel angular velocity

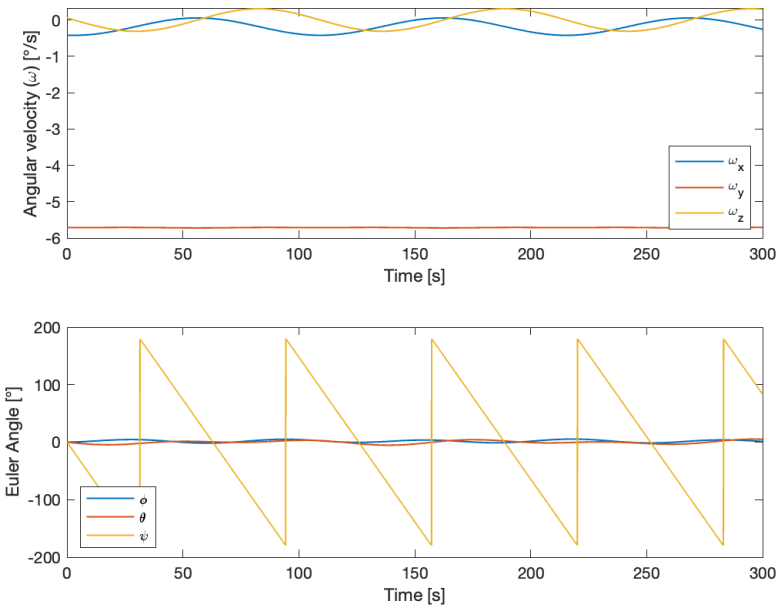


Figure 45: Periodically stable attitude after increasing momentum wheel angular velocity 10x

5 DISTURBANCES

5.1 Gravity Gradient

The gravity gradient can generate a torque on our spacecraft while in orbit. The equations for the gravity gradient torque are below.

$$
\begin{aligned}
I_x\dot{\omega_x} + (I_z - I_y)\omega_y\omega_z &= 3n^2(I_z - I_y)c_yc_z \\
I_y\dot{\omega_y} + (I_x - I_z)\omega_z\omega_x &= 3n^2(I_x - I_z)c_zc_x \\
I_z\dot{\omega_z} + (I_y - I_x)\omega_x\omega_y &= 3n^2(I_y - I_x)c_xc_y
\end{aligned}
$$

In the above equation, $\vec{c} = [c_x, c_y, c_z]^\intercal$ is the normalized direction of $\vec{R}$.

The function `gravGrad` was developed to be used with `ode113` to propagate the Euler equations and kinematics with gravity gradient torques.

```
function [stateDot] = gravGrad(t,state,Ix,Iy,Iz,n)
    % Orbit position and velocity
    r = state(1:3);
    v = state(4:6);

    % Angular velocity
    w = state(7:9);

    % Euler angles
    phi = state(10);
    theta = state(11);

    stateDot = zeros(12,1);
    stateDot(1:3) = v;
    stateDot(4:6) = (-3.986 * 10^5 / norm(r)^2) * r / norm(r); % km/s^2

    radial = r / norm(r);
    A_ECI2P = e2A(state(10:12));
    c = A_ECI2P * radial;
    M = gravGradTorque(Ix,Iy,Iz,n,c);
    stateDot(7) = (M(1) - (Iz - Iy) * w(2) * w(3)) / Ix;
    stateDot(8) = (M(2) - (Ix - Iz) * w(3) * w(1)) / Iy;
    stateDot(9) = (M(3) - (Iy - Ix) * w(1) * w(2)) / Iz;

    % Euler angle time derivatives
    % 312
    EPrimeInv = [sin(phi)*sin(theta) cos(phi)*sin(theta) cos(theta); ...
        cos(theta)*cos(phi) -sin(phi)*cos(theta) 0; ...
        sin(phi) cos(phi) 0] * (1 / cos(theta));
    % 313
    % EPrimeInv = [-sin(phi)*cos(theta) -cos(phi)*cos(theta) ...
       sin(theta); ...
    %     cos(phi)*sin(theta) -sin(phi)*sin(theta) 0; ...
    %     sin(phi) cos(phi) 0] * (1 / sin(theta));
    stateDot(10:12) = EPrimeInv * w;
end
```

```
function M = gravGradTorque(Ix,Iy,Iz,n,c)
    cx = c(1);
    cy = c(2);
    cz = c(3);
    M(1) = 3 * n^2 * (Iz - Iy) * cy * cz;
    M(2) = 3 * n^2 * (Ix - Iz) * cz * cx;
    M(3) = 3 * n^2 * (Iy - Ix) * cx * cy;
end
```

We can estimate the order of magnitude for gravity gradient torques using the following equation.

$$\vec{M} = \frac{3\mu}{a^3} \begin{bmatrix} (I_z - I_y)c_y c_z \\ (I_x - I_z)c_z c_x \\ (I_y - I_x)c_x c_y \end{bmatrix}$$

The parameters used were the known moments of inertia ($I_x = 7707$, $I_y = 14\,563$, $I_z = 18\,050\,\mathrm{kg} \cdot \mathrm{m}^2$), known values for Earth ($a = 7125.49\,\mathrm{km}$, $\mu = 398\,600\,\mathrm{km}^3/\mathrm{s}^2$), and an arbitrary $\vec{c} = [\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}]$. With these values, we arrive at:

$$\vec{M} = \begin{bmatrix} 0.384174 \\ 1.13956 \\ -0.755387 \end{bmatrix} \cdot 10^{-2}\,\mathrm{N\,m}$$

These values are in line with what we see in Figure 50. We also expect gravity gradient torque to be zero, i.e., in equilibrium, when the principal axes are aligned with RTN and the angular velocity is the same as mean motion. Figures 46, 47 demonstrate zero torque when aligned with RTN frame and constant angular velocity. Simulating for longer periods reveals the torque error is periodically stable.

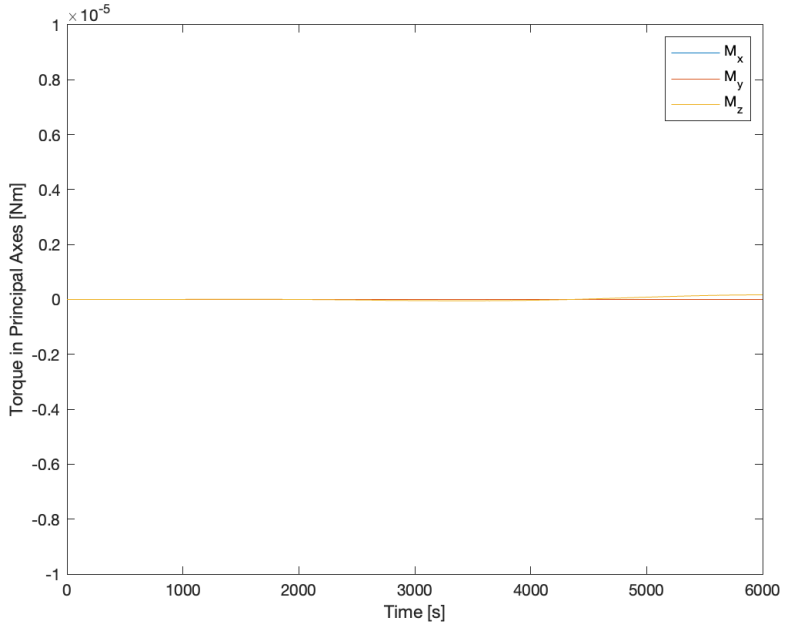


Figure 46: Zero gravity gradient torque for satellite aligned with RTN frame

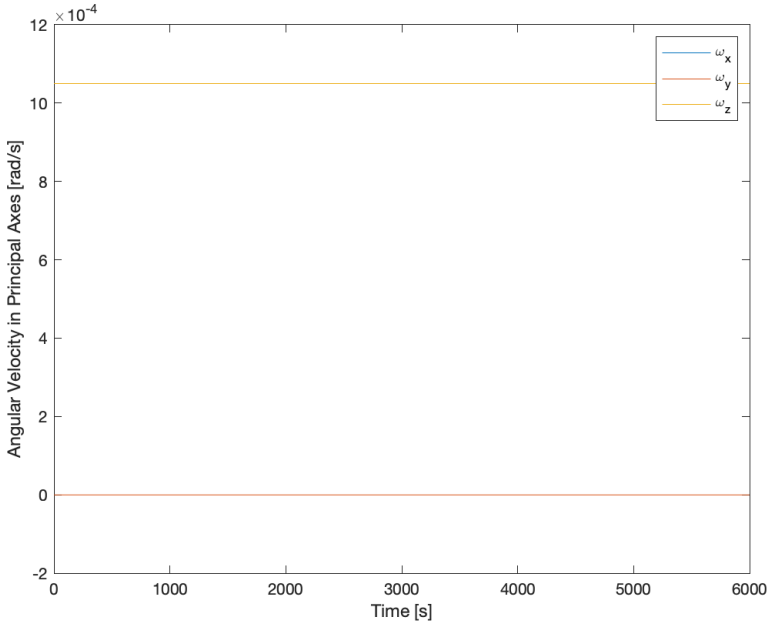


Figure 47: Angular velocity parallel to RTN normal equal to mean motion, all others zero

We align a permuted set of body axes for a different set of initial conditions, which will more closely resemble the orientation of the satellite when collecting science data.

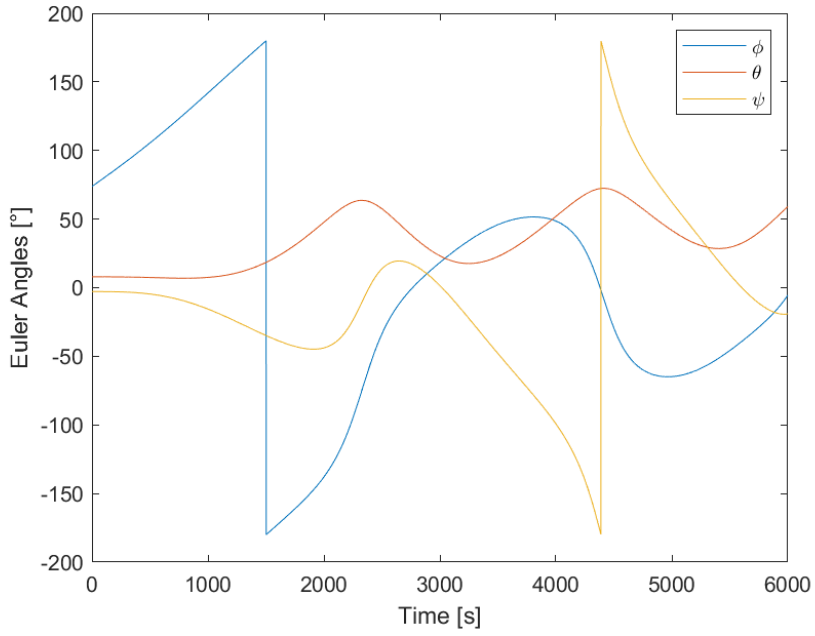


Figure 48: Euler angles with gravity gradient for satellite with arbitrary initial conditions

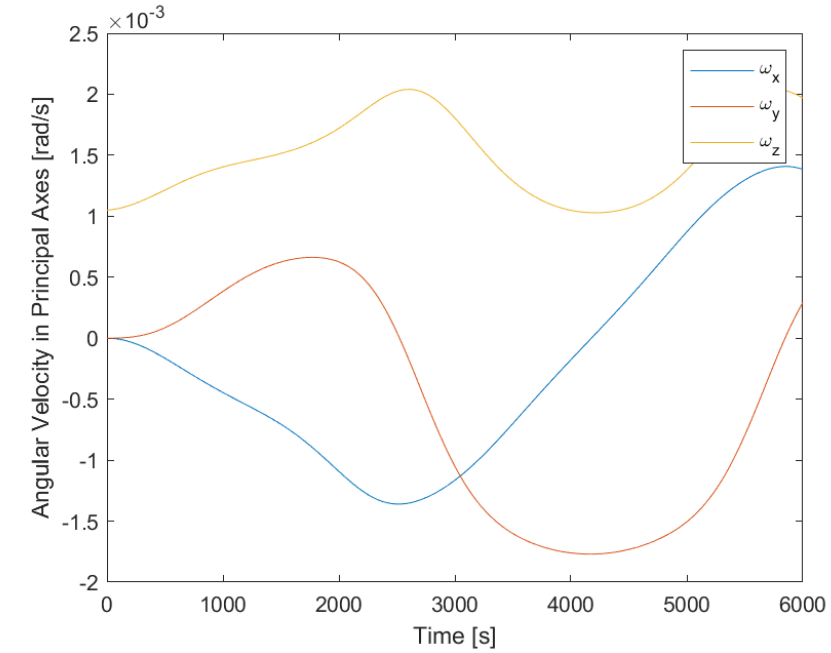


Figure 49: Angular velocity with gravity gradient for satellite with arbitrary initial conditions

The figures show that there is a noticeable effect of gravity gradient torque on the satellite, although the magnitude of the torque is low. This causes changes to our Euler angles throughout the orbit, meaning we will need a control system to stabilize and point our satellite in order to meet mission requirements.

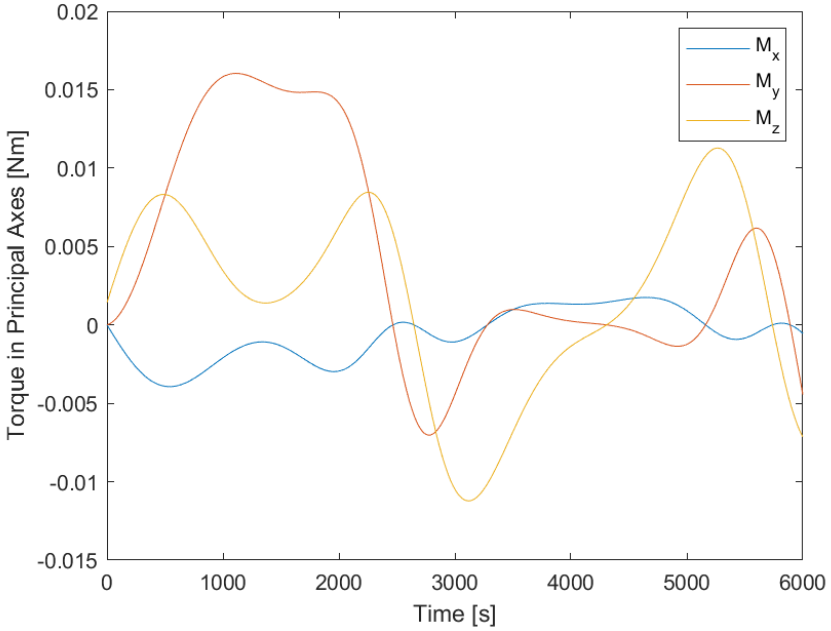


Figure 50: Gravity gradient torques for satellite with arbitrary initial conditions

5.2 Stability, Gravity Gradient

The following equations show the relationships for gravity gradient stability in terms of moments of inertia. The first inequality hold for when the pitch is stable, and the last two inequalites hold for when the roll and yaw are stable.

$$k_N = \frac{I_T - I_R}{I_N}, \qquad k_T = \frac{I_N - I_R}{I_T}, \qquad k_R = \frac{I_N - I_T}{I_R}$$
$$k_T > k_R, \qquad k_R k_T > 0, \qquad 1 + 3k_T + k_R k_T > 4\sqrt{k_R k_T}$$

We show the plot of stable and unstable motion under gravity gradient. When computing the coefficients using moments of inertia about the principal axes, we obtain the following plot. In this case, we investigate the stability for the satellite when it is rotated such that the spacecraft is oriented appropriately for SAR operations, that is, principal x-axis is anti-radial, principal y-axis is opposite of cross-track, and principal z-axis is opposite of along-track. However, this orientation is unstable, as shown below.

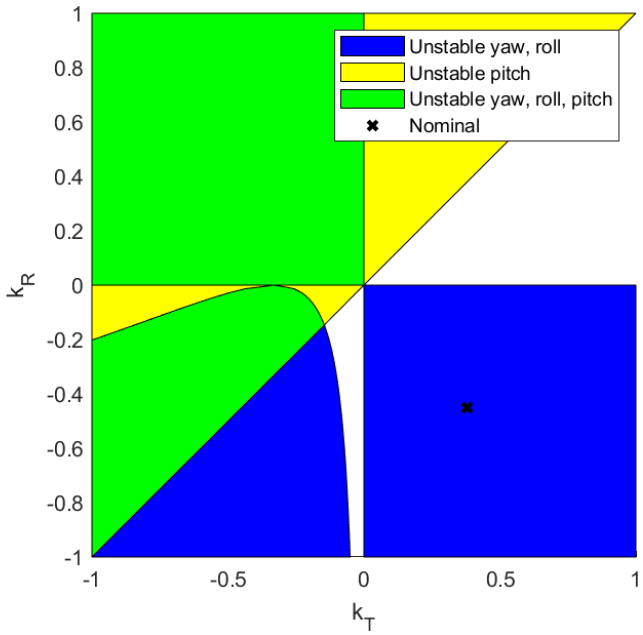


Figure 51: Stability for nominal axes

First, without perturbations, the satellite can maintain steady attitude, demonstrating that this chosen orientation is indeed an equilibrium.

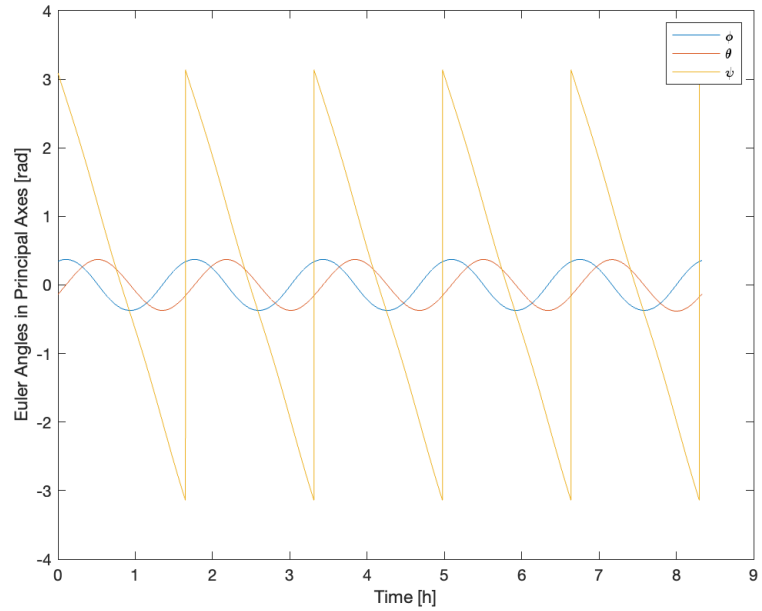


Figure 52: Attitude evolution for unstable orientation without perturbations

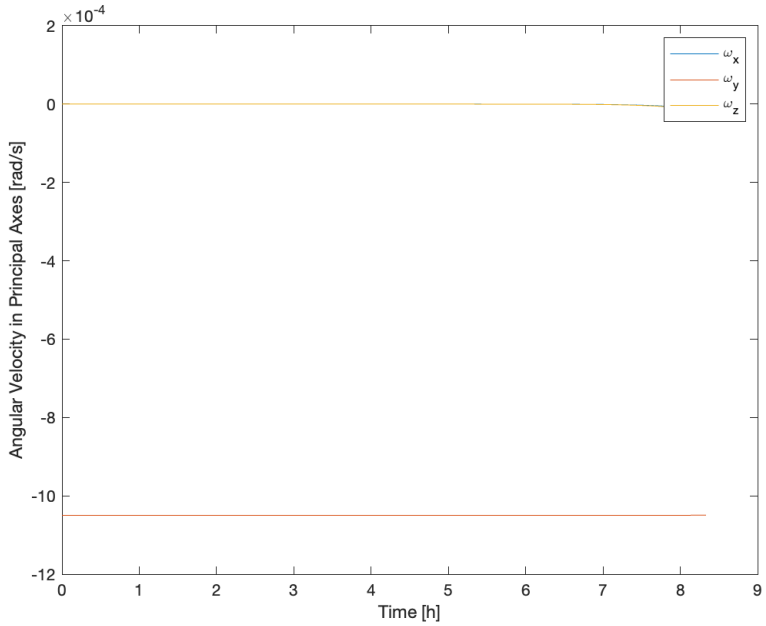


Figure 53: Angular velocity evolution for unstable orientation without perturbations

We expect unstable behavior for small perturbations. Previously, we have already shown that aligning principal axes with RTN produces stable behavior when there are no perturbations. Now, we will introduce small perturbations, causing the system to leave equilibrium and demonstrating that it is unstable.

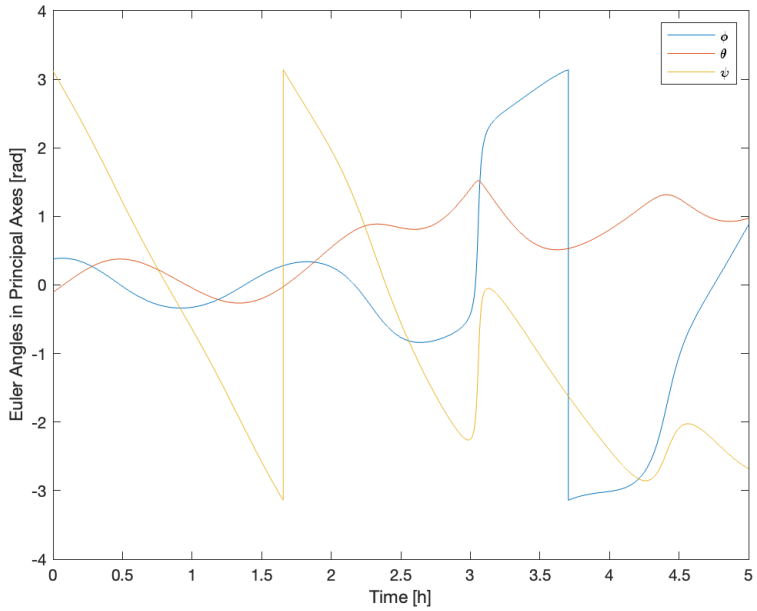


Figure 54: Attitude evolution for unstable orientation with 1% perturbations

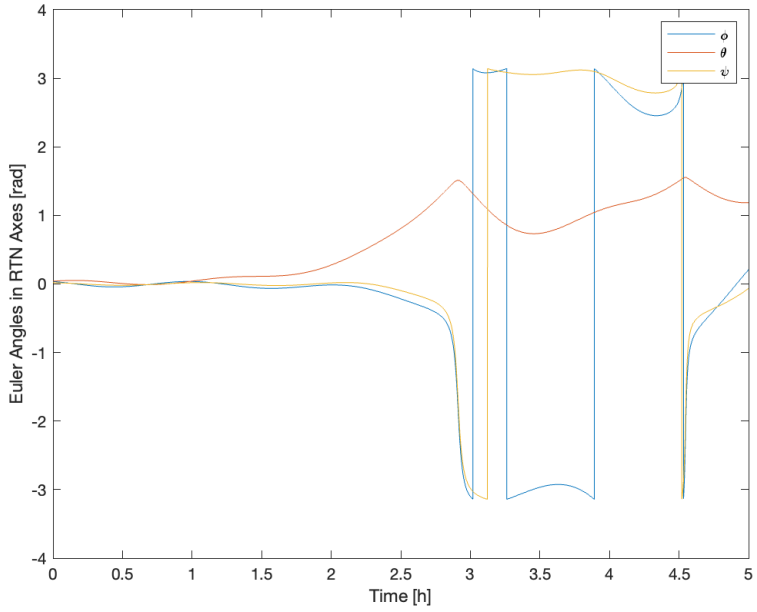


Figure 55: Attitude evolution (relative to RTN) for unstable orientation with 1% perturbations

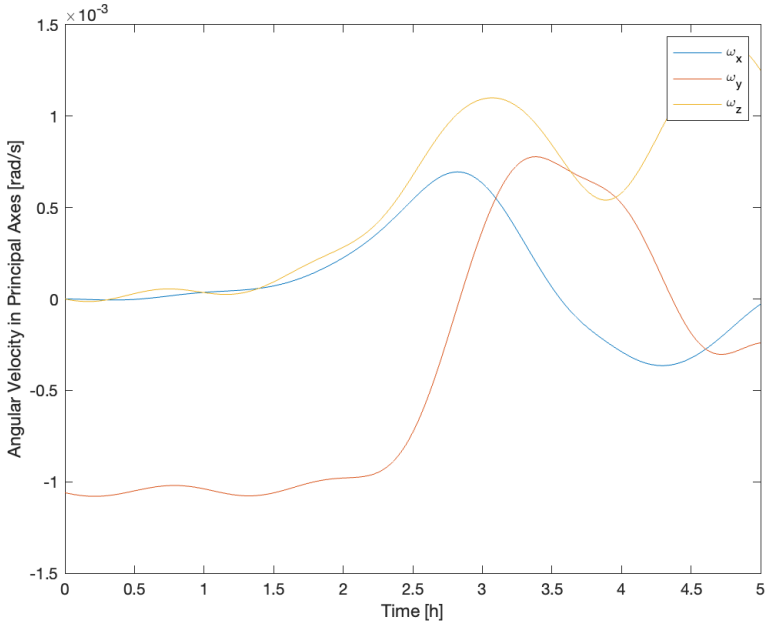


Figure 56: Angular velocity evolution for unstable orientation with 1% perturbations

Now, we change the nominal attitude to generate stable motion due to gravity gradient torque. We will have to align the principal axes XYZ with RTN (respectively) to obtain a stable attitude motion.

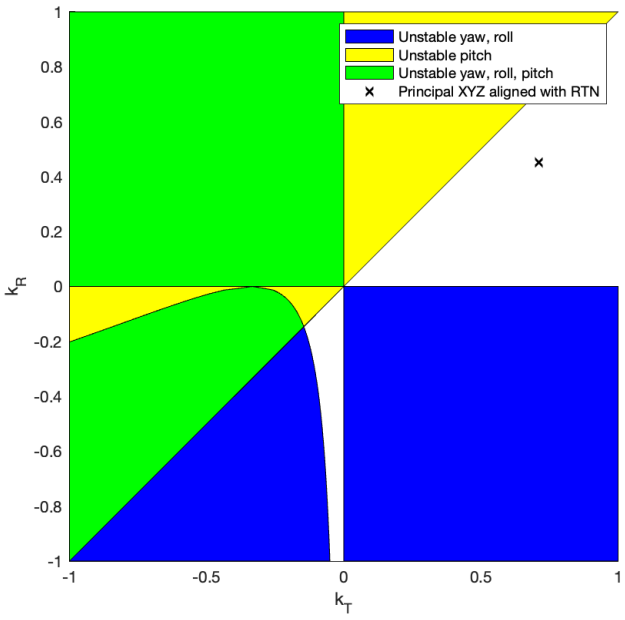


Figure 57: Stability for aligned principal axes

We expect stable behavior for small perturbations. Previously, we have already shown that aligning principal axes with RTN produces stable behavior when there are no perturbations. Now, we will introduce small perturbations.

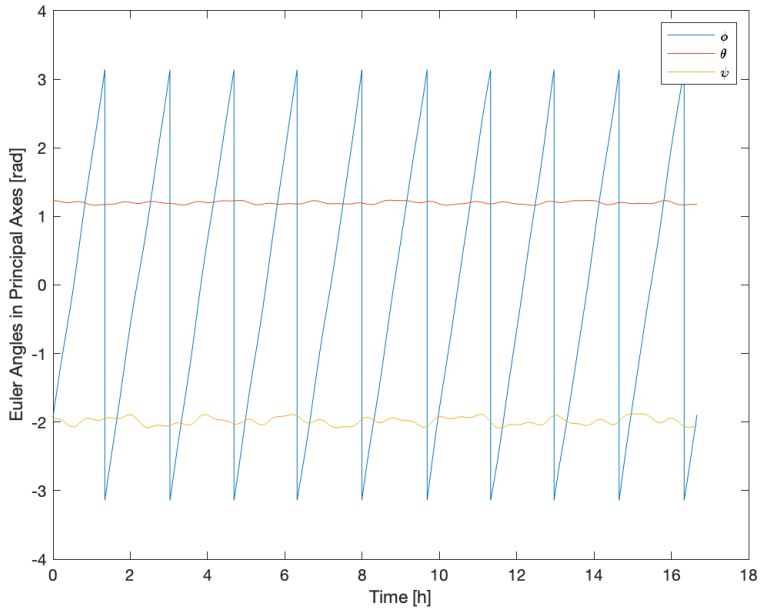


Figure 58: Attitude evolution for stable orientation with 1% perturbations

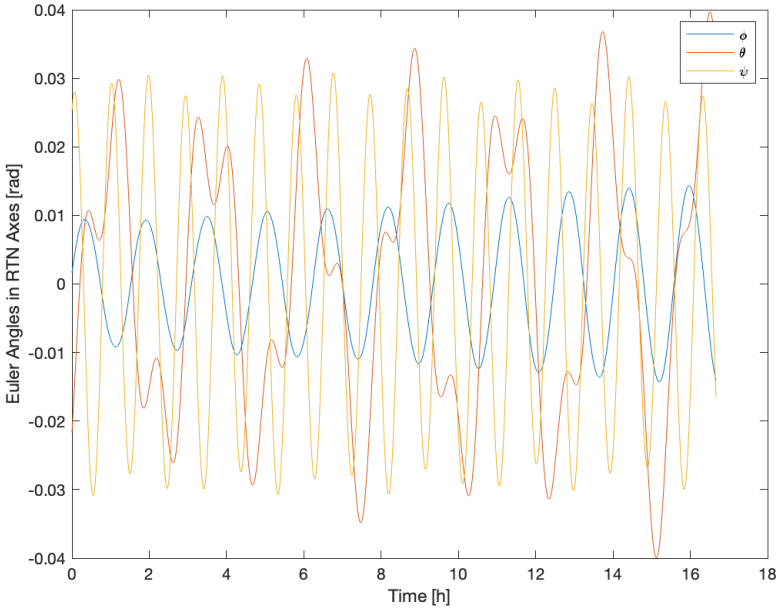


Figure 59: Attitude evolution (relative to RTN) for stable orientation with 1% perturbations

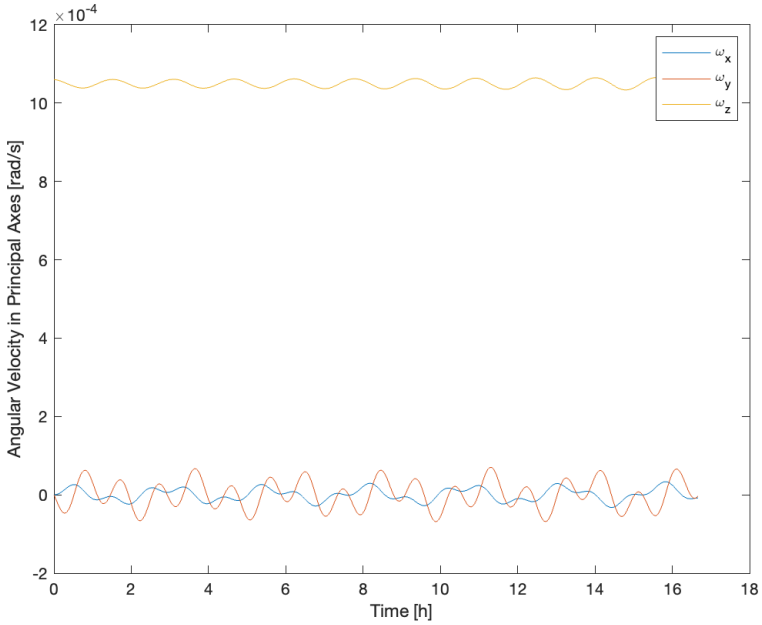


Figure 60: Angular velocity evolution for stable orientation with 1% perturbations

As expected, our attitude motion (angular velocities and Euler angles) are periodically stable with a small perturbation in initial condition.

We cannot maintain this orientation if we want to properly point the radar antenna located at the top of the spacecraft, so it does not make sense to orient the satellite along principal axes for gravity gradient stability. Instead, we will likely require magnetorquers to offset angular momentum changes from environmental torques, including gravity gradient torques due to our unstable equilibrium.

5.3 Magnetic Field

In addition to gravity gradient torques, we modeled torque from magnetic field interactions, solar radiation pressure, and atmospheric drag.

The equation for the torque from the magnetic field interactions is shown below.

$$\vec{M}_m = \vec{m}_{sat} \times \vec{B}_{Earth}$$

Since the satellite is operates in LEO, the spherical harmonic model up to n = 4 was used for maximum accuracy. This model is based on the geocentric distance (R), colatitude (θ), and longitude (ϕ). The model requires Gaussian coefficients ($g^{n,m}$, $h^{n,m}$) and Legendre functions ($P^{n,m}$) and their derivatives ($\frac{\delta P^{n,m}(\theta)}{\delta\theta}$), which are explained in more detail in Wertz [12].

$$B_R = \sum_{n=1}^{4}(\frac{R_{Earth}}{R})^{n+2}(n+1)\sum_{m=0}^{n}(g^{n,m}\cos m\phi + h^{n,m}\sin m\phi)P^{n,m}(\theta)$$

$$B_\theta = -\sum_{n=1}^{4}(\frac{R_{Earth}}{R})^{n+2}\sum_{m=0}^{n}(g^{n,m}\cos m\phi + h^{n,m}\sin m\phi)\frac{\delta P^{n,m}(\theta)}{\delta\theta}$$

$$B_\phi = -\frac{1}{\sin\theta}\sum_{n=1}^{4}(\frac{R_{Earth}}{R})^{n+2}\sum_{m=0}^{n}m(-g^{n,m}\sin m\phi + h^{n,m}\cos m\phi)P^{n,m}(\theta)$$

5.4 Solar Radiation Pressure

The equations below define the solar radiation torque, where C_S is the specular reflection coefficient, C_d is the diffuse reflection coefficient, $\vec{S}$ is the vector facing the Sun, and e_i is 1 if the surface is illuminated and 0 otherwise. In implementation, we represent e_i as a boolean tensor in tensor operations for fast computation.

$$\vec{M}_S = \sum_{i=1}^{n} \vec{r}_i \times e_i \int_{S_i} d\vec{f}_{total_i}$$

$$d\vec{f}_{total} = -P((1 - C_S)\hat{\vec{S}} + 2(C_S \cos\theta + \frac{1}{3}C_d)\cos\theta dA$$

5.5 Drag

Finally, the equation below define the aerodynamic torque. Velocity is relative velocity of the spacecraft to the atmosphere, and we can compute the atmosphere's relative motion using a cross product of the position vector and Earth's rotational rate in ECI.

$$d\vec{f}_{aero} = -\frac{1}{2}C_D\rho V^2(\hat{\vec{V}} \cdot \hat{\vec{N}})\hat{\vec{V}}dA$$

Implementation of these disturbances in MATLAB code can be found in the appendix.

5.6 Analysis

For most torques, the worst-case estimated magnitudes were found by using existing properties of the spacecraft model. We use simplified equations (compared to those in the previous section) to estimate these, making assumptions for worst-case torques. However, the magnetic torque calculation assumes a model of the spacecraft's magnetic field as a single coil wrapped around the surface of the RIS and satellite bus. The estimated maximum values of each torque are listed in the table below.

Table 7: Table of disturbance torques

Disturbance Torque	Estimated Maximum Value (N-m)
M_{gg}	1.7093e-02
M_{srp}	1.8294e-02
M_{drag}	1.3682e-03
M_{mag}	1.3751e-10

In Figures 61, 62, 63, 64, we show the numerical simulation results for each type of disturbance based on the equations and code in the previous section. The numerical simulations indicate that the simulated results and estimated values are roughly within an order of magnitude for each type of disturbance torque shown.

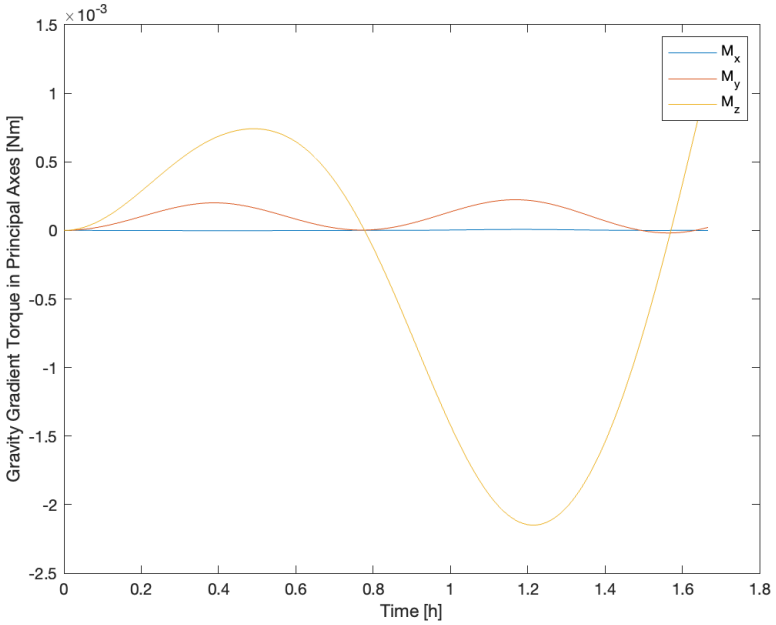


Figure 61: Numerical simulation of gravity gradient torques

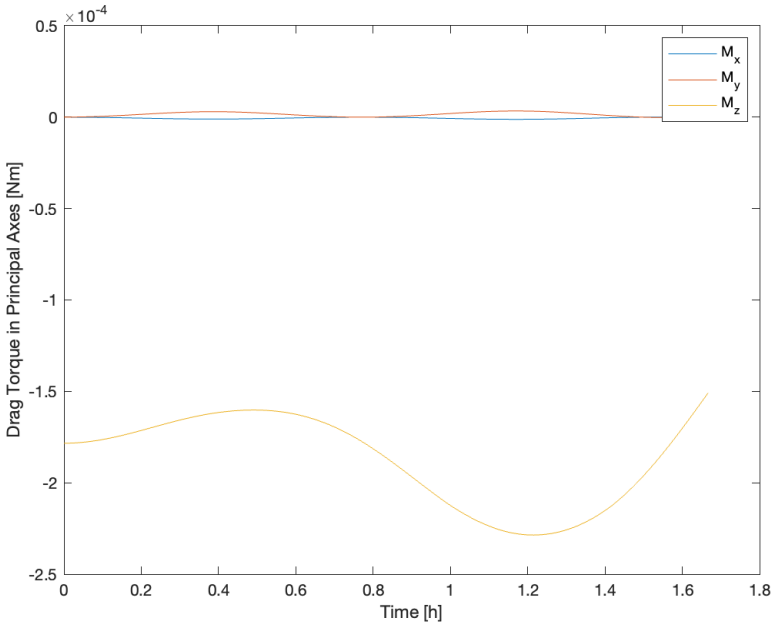


Figure 62: Numerical simulation of drag torques

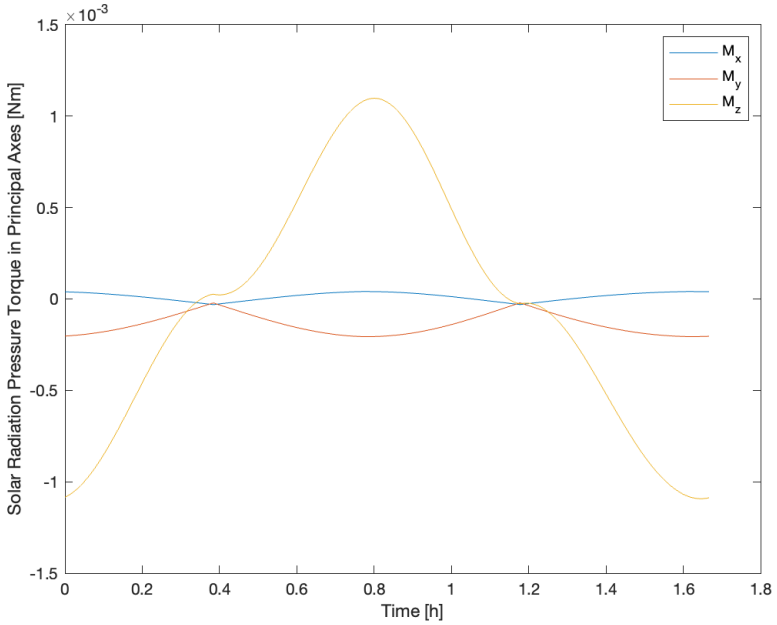


Figure 63: Numerical simulation of solar radiation pressure torques

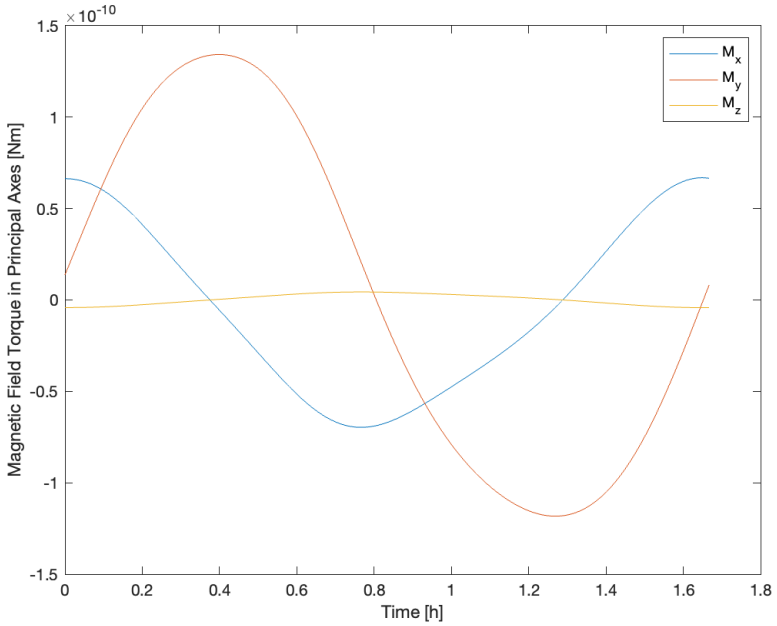


Figure 64: Numerical simulation of magnetic field torques

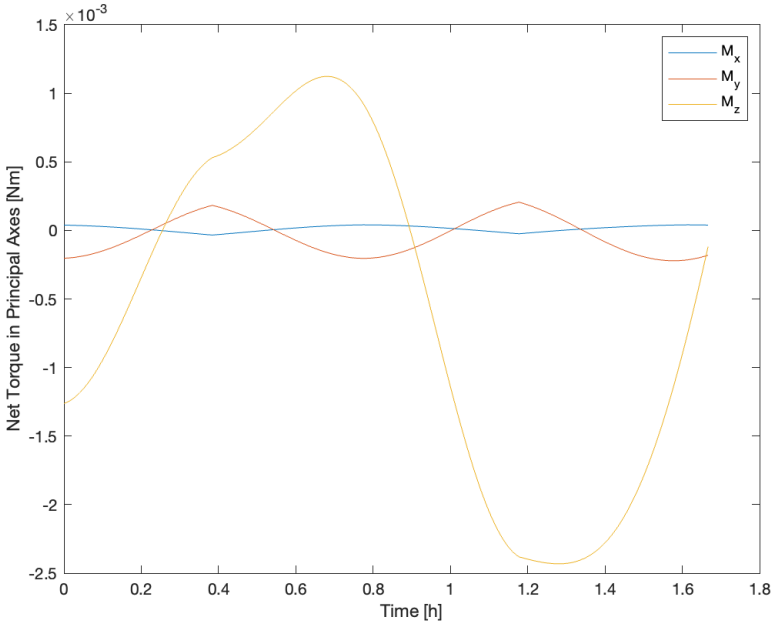


Figure 65: Numerical simulation of net torques

6 ATTITUDE DETERMINATION

6.1 Attitude Control Error

In operation, NISAR will point its antenna towards Earth such that the principal x-axis facing the negative radial direction and the principal y-axis is pointing in the negative normal direction. The angular velocity about the principal y-axis is the negative of mean motion.

$$\vec{P}_{desired} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix} \vec{P}_{RTN}$$

We first simulate attitude without disturbance torques, calculating error as the rotation between target and actual attitude. Figures 66, 67, 68 demonstrate that the error is approximately zero.

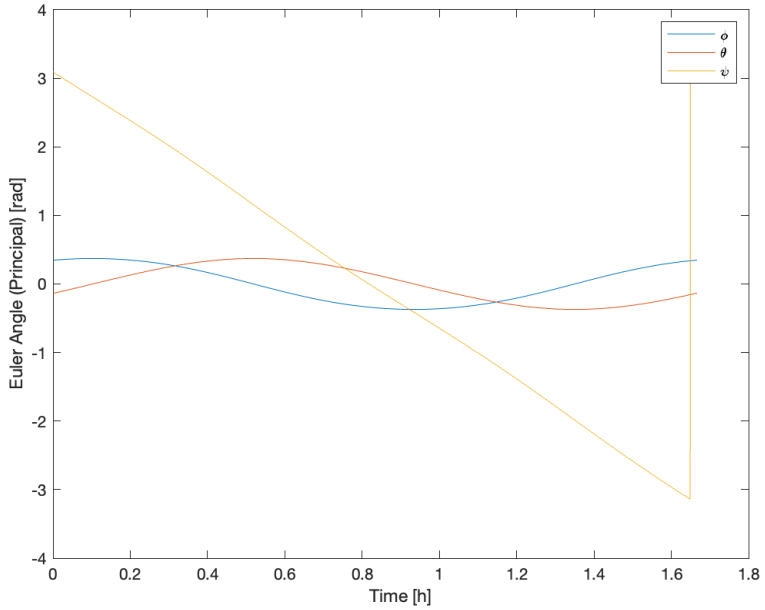


Figure 66: Actual attitude relative to inertial frame, without perturbations

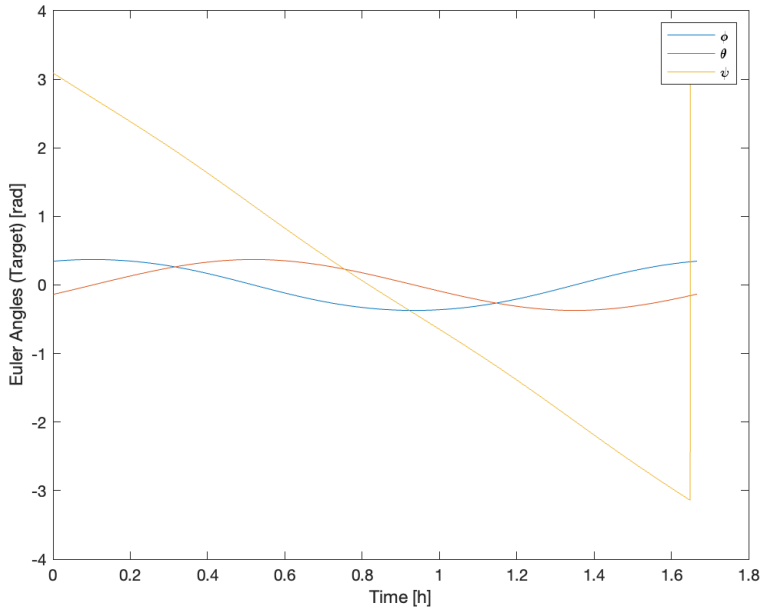


Figure 67: Target attitude relative to inertial frame, without perturbations

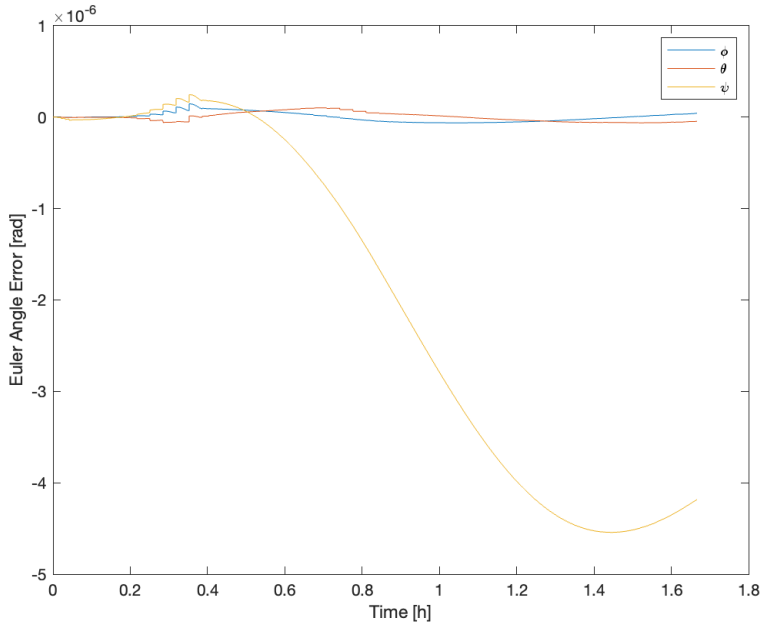


Figure 68: Error between target and actual attitude, without perturbations

For computing errors, we multiply the DCM representing the rotation from the inertial frame to our actual attitude with the inverse of the DCM representing the rotation from the inertial frame to the target attitude. We observe that there is a very small error (approximately 10^{-7} to 10^{-6}) in our result in Figure 68. This is most likely arising from our use of numerical integration.

6.2 Attitude Control Error, with Perturbation

With perturbations, we observe a small but noticeable change to actual attitude in Figure 69.

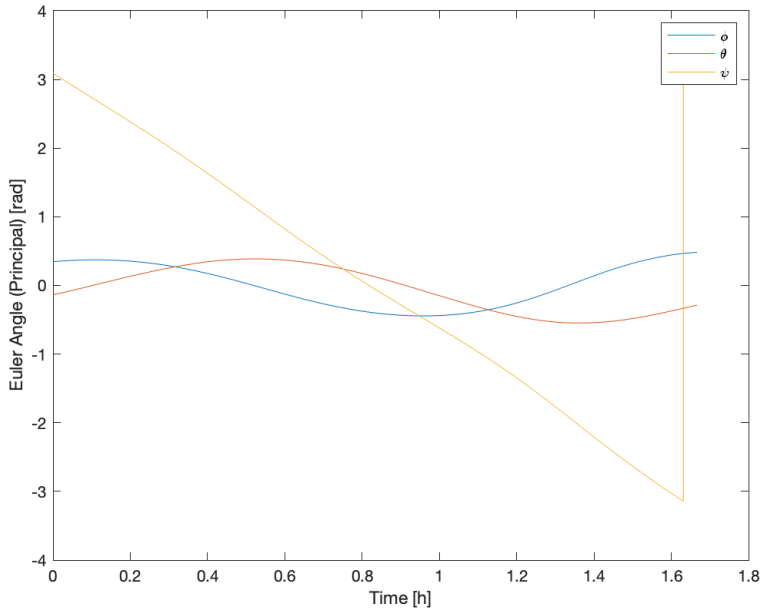


Figure 69: Actual attitude relative to inertial frame, with perturbations

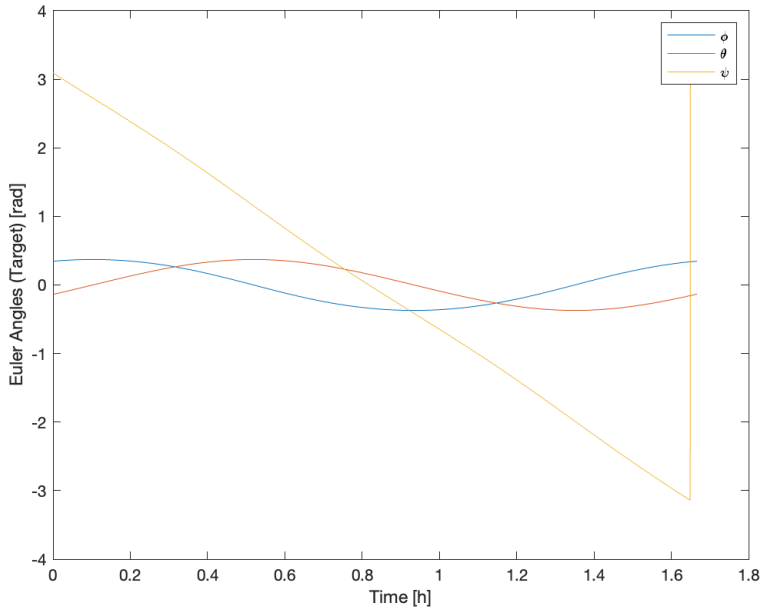


Figure 70: Target attitude relative to inertial frame, with perturbations

A much larger error appears in Figure 71. When compared with the disturbance-free error in Figure 68 (which is approximately zero), we can interpret the attitude control error as the

amount which the disturbance torques rotate our satellite during the simulation period, which in this case is one orbital period.

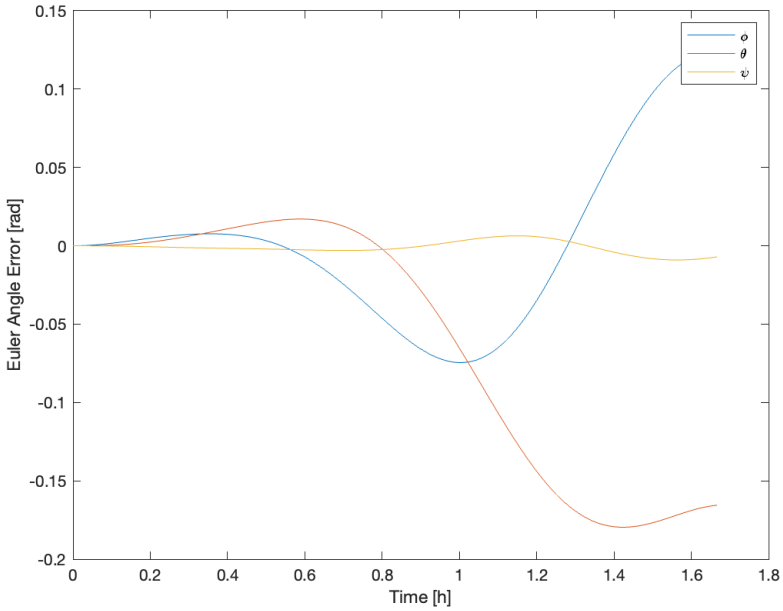


Figure 71: Error between target and actual attitude, with perturbations

6.3 Attitude Determination Subsystem

We create a Simulink model using the same functions and variables previously modeled in MATLAB and shown in this paper, shown in Figure 72. Notably, we have a simulation subsystem which is responsible for simulating the dynamics, kinematics, and disturbances of the system, which we have modeled in previous sections. We now add an attitude determination subsystem for this step. Since we choose to model noise-free measurements for now, we directly generate measurements from the true state without injecting noise.

We initially model our attitude determination subsystem using "dummy" sensors and measurements. Each of the algorithms is fed randomly generated unit vectors as a noise-free measurement. These unit vectors represent fixed directions in the celestial sphere such that an orientation can be computed when a sufficient number of vectors is available. This represents the functionality of a star tracker sensor, which is able to determine the orientation of the satellite's axes relative to an inertial reference using unit vector directions of known stars. We do not yet model the inner workings of a star tracker, such as the identification and computation of star positions.

In our test implementation, we generate five such random unit vectors. These unit vectors are both our ground truth (analogous to known directions from a star catalog) and our measurement signal, since we do not inject any noise at this step. Thus, we expect to obtain an exact solution from each attitude determination algorithm in the absence of such noise.

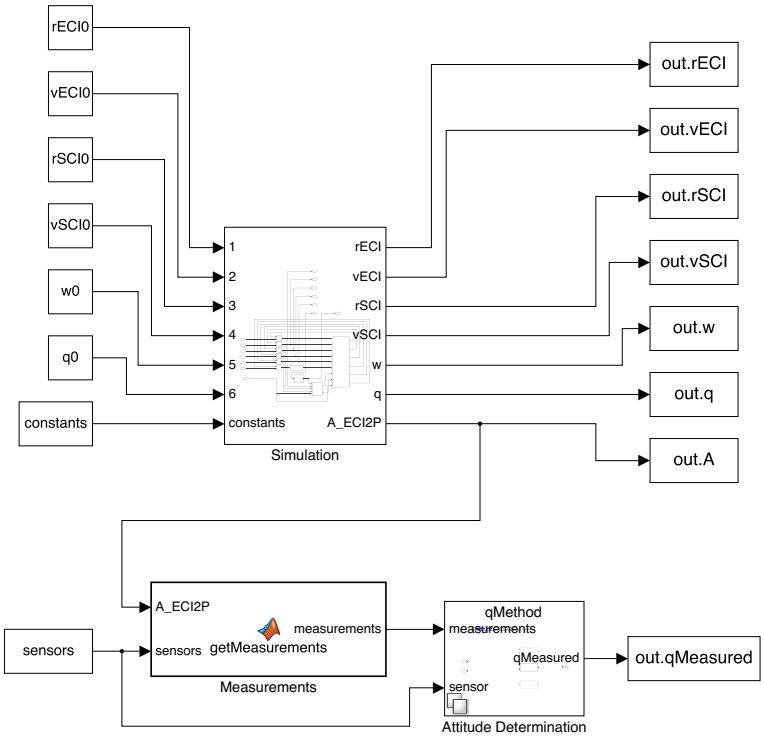


Figure 72: Simulink model for NISAR spacecraft

Additionally, we can compute the attitude by integration of our angular velocity, equivalent to using a rate gyro sensor that measures angular rate. We can simply use the same kinematics equations used to propagate our attitude to model the evolution of our attitude for the rate gyro. In simulation and without noise, we also expect an exact solution from this technique.

6.4 Deterministic Attitude Determination Algorithm

For measurements $\vec{M}$ and ground truth $\vec{V}$, we can use deterministic attitude determination to find the attitude by taking the pseudoinverse.

$$\vec{M} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \vec{m}_1 & \vec{m}_2 & \vec{m}_3 \\ \vdots & \vdots & \vdots \end{bmatrix} = \vec{A} \begin{bmatrix} \vdots & \vdots & \vdots \\ \vec{v}_1 & \vec{v}_2 & \vec{v}_3 \\ \vdots & \vdots & \vdots \end{bmatrix} = \vec{A}\vec{V}$$

$$\vec{A} = \vec{M}\vec{V}^{-1}$$

The implementation of the deterministic attitude determination method above assumes that we have three or more measurements. However, if there are only two measurements, a third measurement can be "created" using the cross product. This technique creates measurements

and corresponding ground truth vectors p, q, and r as shown below.

$$\tilde{\vec{m}}_1 = \frac{\vec{m}_1 + \vec{m}_2}{2} \quad \tilde{\vec{m}}_2 = \frac{\vec{m}_1 - \vec{m}_2}{2} \quad \tilde{\vec{v}}_1 = \frac{\vec{v}_1 + \vec{v}_2}{2} \quad \tilde{\vec{v}}_2 = \frac{\vec{v}_1 - \vec{v}_2}{2}$$

$$\vec{p}_m = \tilde{\vec{m}}_1; \quad \vec{q}_m = \frac{\tilde{\vec{m}}_1 \times \tilde{\vec{m}}_2}{||\tilde{\vec{m}}_1 \times \tilde{\vec{m}}_2||} \quad \vec{r}_m = \vec{p}_m \times \vec{v}_m$$

$$\vec{p}_v = \tilde{\vec{v}}_1; \quad \vec{q}_v = \frac{\tilde{\vec{v}}_1 \times \tilde{\vec{v}}_2}{||\tilde{\vec{v}}_1 \times \tilde{\vec{v}}_2||} \quad \vec{r}_v = \vec{p}_v \times \vec{v}_v$$

6.5 Statistical Attitude Determination Algorithm (q-Method)

Given measurements $\vec{M}$, ground truth $\vec{V}$, and sensor weights $\vec{W}$, we can use statistical attitude determination (aka the q-method). This method ultimately finds the quaternions using the eigenvalue/eigenvector below, where the eigenvector associated with the largest eigenvalue is the quaternion vector.

$$\vec{K}\vec{q} = \lambda\vec{q}$$

In the eigenvalue decomposition above, $\vec{K}$ can be found with the following relation.

$$\vec{w}_i = \sqrt{w_1}\vec{m}_1 \quad \vec{u}_i = \sqrt{w_1}\vec{v}_1$$

$$\vec{W} = \begin{bmatrix}\vec{w}_1 & \vec{w}_2 & ... & \vec{w}_n\end{bmatrix} \quad \vec{U} = \begin{bmatrix}\vec{u}_1 & \vec{u}_2 & ... & \vec{u}_n\end{bmatrix}$$

$$\vec{B} = \vec{W}\vec{U}^T \quad \vec{S} = \vec{B} + \vec{B}^T \quad \sigma = trace(\vec{B})$$

$$Z = \begin{bmatrix}B_{23} - B_{32} & B_{31} - B_{13} & B_{12} - B_{21}\end{bmatrix}^T$$

$$K = \begin{bmatrix}\vec{S} - \sigma\vec{I} & \vec{Z} \\ \vec{Z}^T & \sigma\end{bmatrix}$$

6.6 Kinematic Attitude Determination Algorithm

We can reconstruct our attitude using the kinematic equations used to model our attitude motion in the ground truth simulation, instead simulated as part of the spacecraft's onboard computer. We simply duplicate our previously-implemented kinematic equations and pass in the angular velocity to reconstruct attitude evolution for this technique.

The methods for attitude determination are implemented in MATLAB code included in the appendix. We include implementations for deterministic attitude determination both for two measurements and for three or more measurements.

6.7 Attitude Estimation, without Sensor Errors

Figures 73, 74, 75, and 76 depict the attitude determination using the different methods discussed earlier. All are identical, which is expected, since each algorithm is given perfect measurements free of noise. Error for each is on the order of 10^{-16}, which is purely numerical.

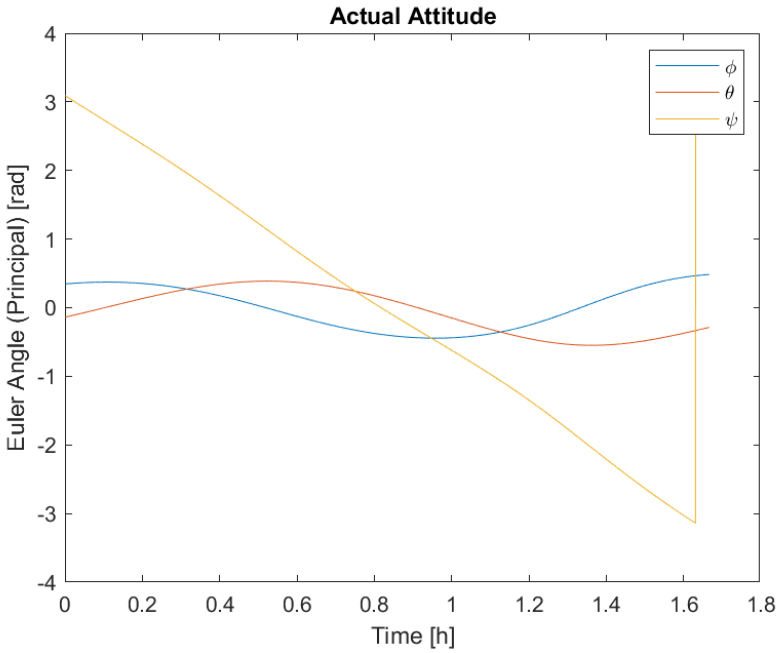


Figure 73: True satellite attitude relative to inertial frame

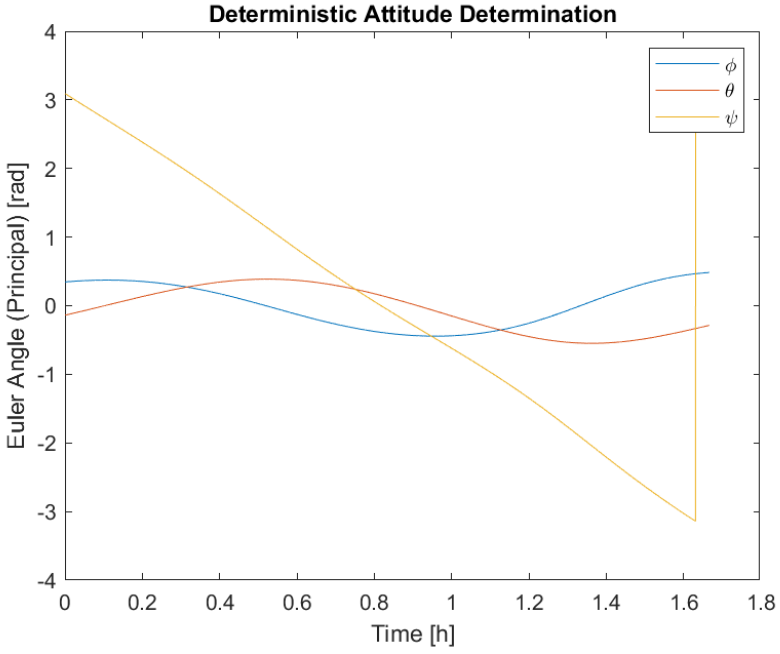


Figure 74: Deterministic attitude determination

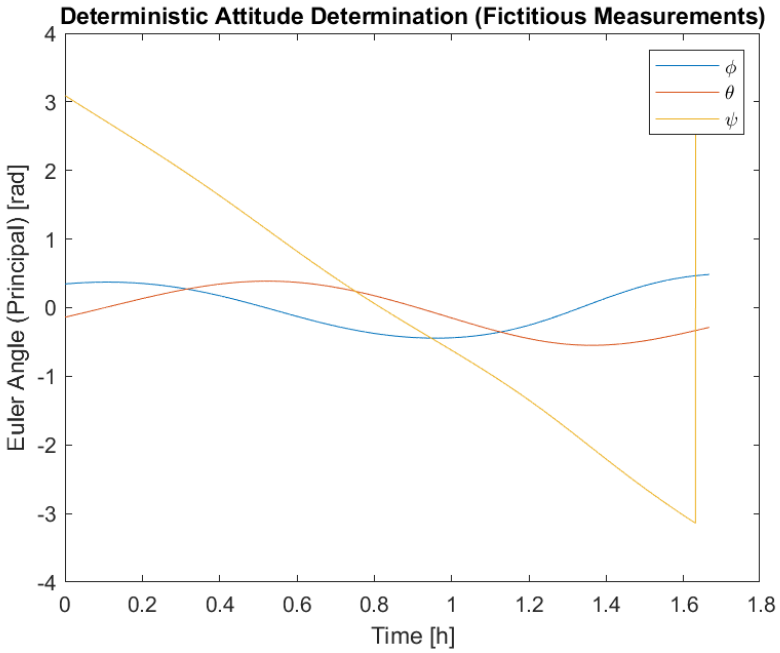


Figure 75: Deterministic attitude determination with fictitious measurements

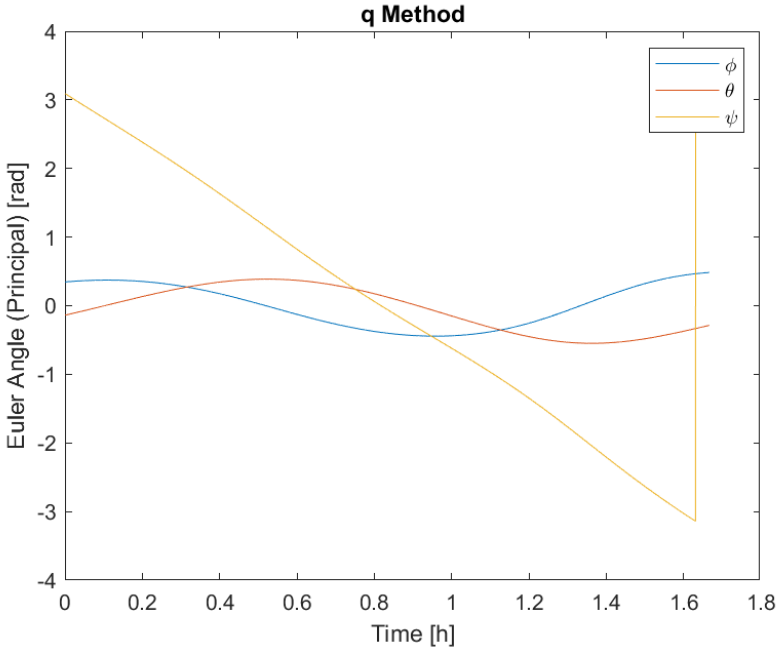


Figure 76: q-method

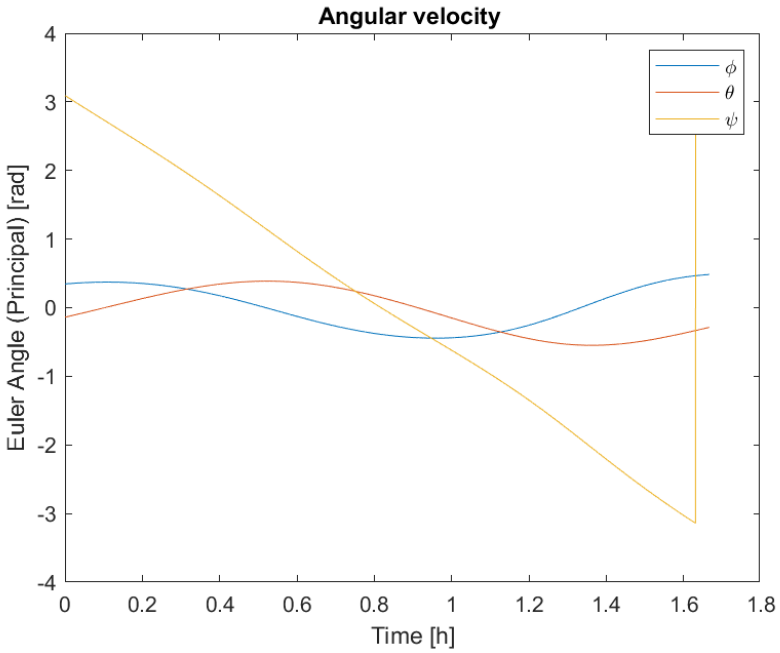


Figure 77: Attitude reconstruction using kinematic equations

6.8 Sensor Specifications

As the specific data regarding for NISAR ADCS sensors are not widely available, much of the sensor error and bias information was extrapolated from data from Wertz and lecture materials, as well as manufacturers of sensors for similar satellites. The table below includes the general sensor information used in this analysis.

Table 8: Sensor performance specifications

Sensor	**Sensor Error**	**Sensor Bias**
Sun Sensor [12]	0.5°	0°
Star Tracker [12]	0.01°	0°
Gyroscope [13]	0.001°/ s	5E-5°/s

6.9 Attitude Estimation, with Sensor Errors

The estimation errors for the deterministic method, q-method, and kinematics-based method are plotted below. As expected, the q-method has a significantly lower error than the deterministic method. Additionally, since the gyroscopes were modeled to have a slight bias, the Euler angles slowly drift away from expected values.

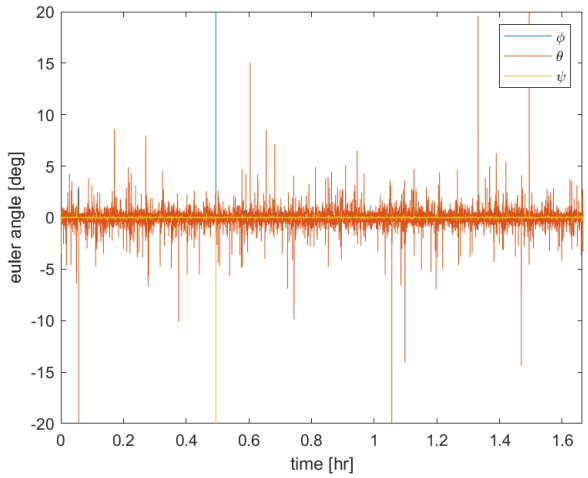


Figure 78: Attitude error using deterministic attitude determination

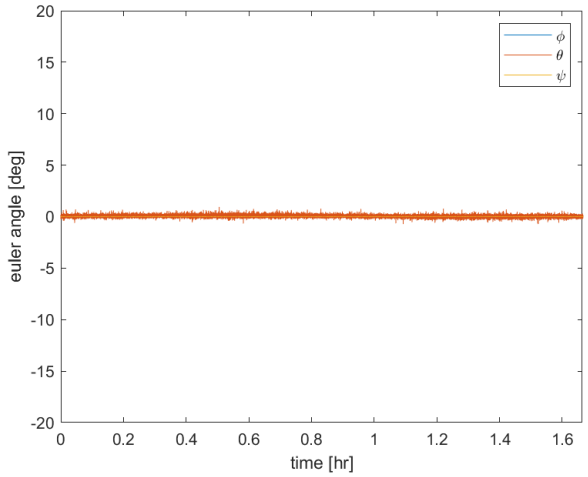


Figure 79: Attitude error using q-method

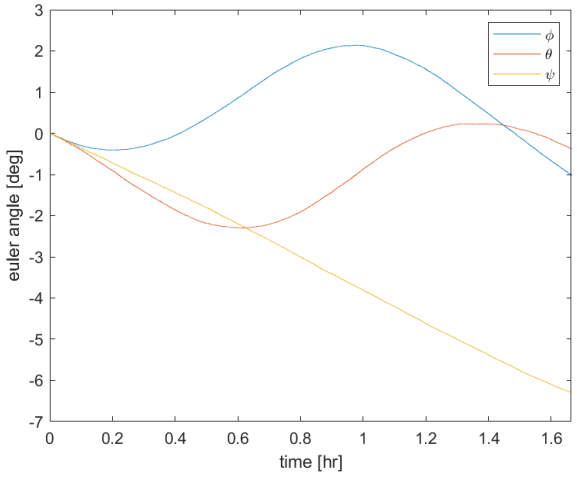


Figure 80: Attitude error using kinematic attitude determination

6.10 Small Angle Errors

For this analysis, we choose to use q-method, as it produced the smallest errors in the previous section. In order to estimate the size of the errors, we compare the exact error DCM to the small angle approximation DCM based on the corresponding Euler angles, shown below.

$$A_{\text{small angle}} = \begin{bmatrix} 1 & \phi & -\psi \\ -\phi & 1 & \theta \\ \psi & -\theta & 1 \end{bmatrix}$$

The Frobenius norm is then calculated from the difference in the error DCM and small angle DCM to see how applicable small angle assumptions are for the error. Based on Figure 81, the magnitudes of the Frobenius norm reflect that the overall errors were relatively small for the q-method, implying only a very small angular difference.

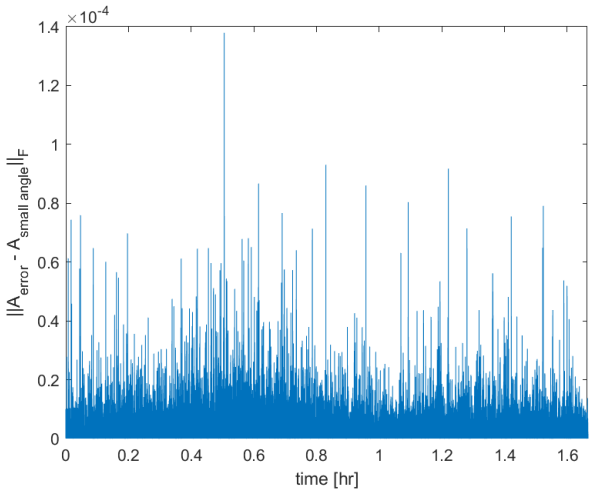


Figure 81: Frobenius norm of error estimate versus small angle approximation

6.11 Sensor Models

There are three major types of sensors used in this analysis: sun sensors, star sensors, and gyroscopes. The following section discusses the basics of each sensor's function and modeling.

Sun sensors utilize a photoelectric effect to produce an electric current based on the Sun's emitted electromagnetic radiation. For a single sensor, the current is based on sensitive surface S, optical properties α, and angle of incidence θ.

$$I = \alpha S \cos(\theta)$$

Due to the nonlinearity of this relationship, sun sensors operate within a limited field of view where the relationship between the angle of incidence and current draw is approximately linear.

However, because NISAR operates in a sun-synchronous orbit, the sun sensor model we used did not factor in the sensor's field of view, as a well-placed sensor with a wide enough FOV will always have the sun in frame. Our sun sensor model outputs the direction of the Sun from the simulation and adds Gaussian noise.

Star trackers are more complicated. These sensors are similar to cameras that capture star positions and compare them to an existing star catalog, which can be used to determine the

satellite's attitude. These trackers are often designed to have a much narrower field of view to maximize the pixel pitch. Star trackers use algorithms such as angular separation matching to match each star to a star in the star catalog.

The star tracker model used in our simulations introduces simplifying assumptions. The model we use generates 10 random unit vectors within the FOV of the star tracker on board (approximately $20°$) and adds Gaussian noise. These random vectors represent the direction of 10 arbitrary stars in the celestial sphere.

Gyroscopes measure the angular velocity of the satellite. There are two main types of gyroscopes: laser gyroscopes and mechanical gyroscopes. Laser gyroscopes emit light along a fiber optic cable ring in opposite directions. The time or phase difference of the light along the ring can be used to compute the angular velocity. Mechanical gyroscopes, such as rate gyros, use gimbals and measure the angle induced in one direction by angular velocity in another. This measurement can come from a device such as a potentiometer, which provides an electrical voltage signal.

To simplify the model of the sensor, the gyroscope model takes angular velocity from the simulation and adds Gaussian noise and expected bias.

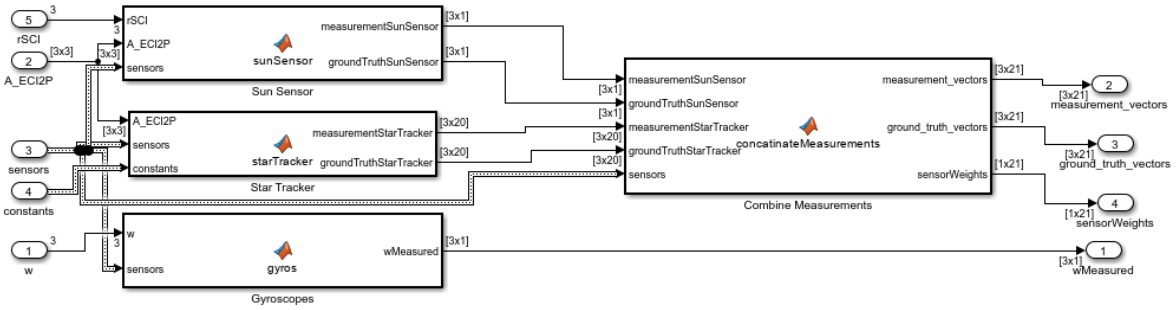


Figure 82: Simulink model of sensors

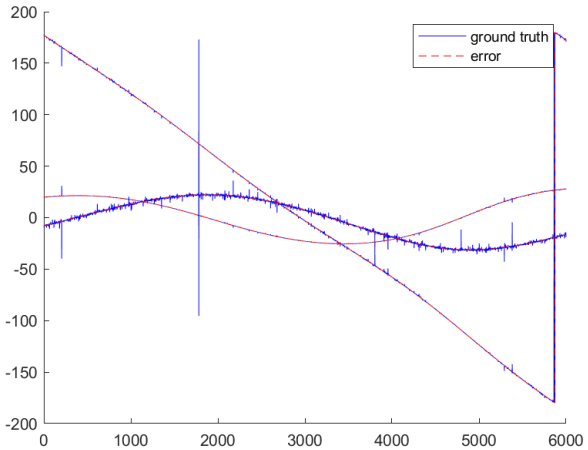


Figure 83: Attitude error using deterministic attitude determination

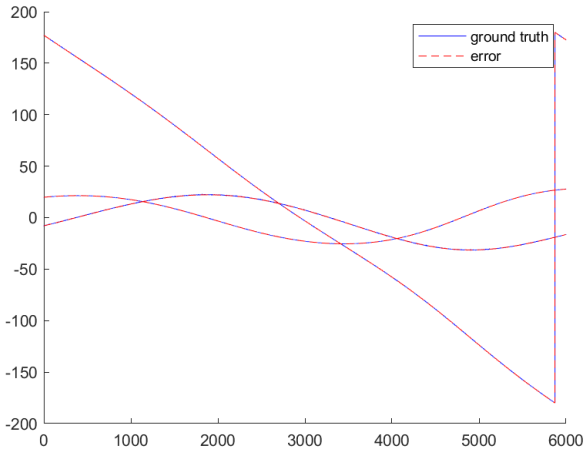


Figure 84: Attitude error using q-method

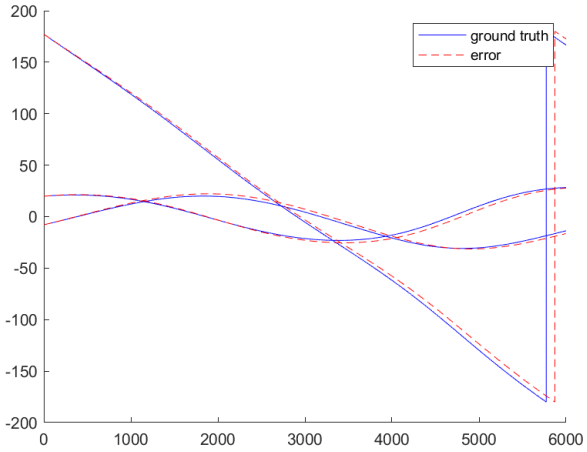


Figure 85: Attitude error using kinematic attitude determination

6.12 Multiplicative Extended Kalman Filter, Time Update

The following MATLAB function is our time update-only implementation of a multiplicative extended Kalman filter (MEKF). We describe the formulation of the MEKF immediately following this function.

```
function [qkminus,wkminus,Pkminus] = ...
    timeUpdate(dt,qkminus1,wkminus1,Pkminus1,Q,Ix,Iy,Iz,u)
    % MEKF, only time update implemented
    xkminus1 = [zeros([3 1]); wkminus1];

    w = norm(wkminus1);
    wcross = [0, -wkminus1(3), wkminus1(2); ...
        wkminus1(3), 0, -wkminus1(1); ...
        -wkminus1(2), wkminus1(1), 0];
    cw = cos(w * dt / 2);
    sw = sin(w * dt / 2);

```

```
12      A = eye(3) + sw * wcross / w + (1 - cw) * wcross^2 / w^2;
13      phi = eye(3) + 0.5 * dt * ...
14          [0, (Iy - Iz) / Ix * wkminus1(3), (Iy - Iz) / Ix * ...
                wkminus1(2); ...
15          (Iz - Ix) / Iy * wkminus1(3), 0, (Iz - Ix) / Iy * ...
                wkminus1(1); ...
16          (Ix - Iy) / Iz * wkminus1(2), (Ix - Iy) / Iz * wkminus1(1), 0];
17      B = dt * [1 / Ix, 0, 0; ...
18          0, 1 / Iy, 0; ...
19          0, 0, 1 / Iz];
20      STM = [A, zeros(3); zeros(3), phi];
21      O = eye(4) + 0.5 * dt * ...
22          [0, wkminus1(3), -wkminus1(2), wkminus1(1); ...
23          -wkminus1(3), 0, wkminus1(1), wkminus1(2); ...
24          wkminus1(2), -wkminus1(1), 0, wkminus1(3); ...
25          -wkminus1(1), -wkminus1(2), -wkminus1(3), 0];
26
27      xkminus = STM * xkminus1 + [zeros([3 1]); B * u];
28      wkminus = xkminus(4:6);
29      qkminus = O * qkminus1;
30      qkminus = qkminus / norm(qkminus);
31      Pkminus = STM * Pkminus1 * STM' + Q;
32  end
```

Our state consists of 3 small angles (zero mean) and angular velocities. Our state does not track the attitude directly, but we can compute quaternions using known kinematics.

$$\vec{x} = \begin{bmatrix} \alpha_x & \alpha_y & \alpha_z & \omega_x & \omega_y & \omega z \end{bmatrix}^\intercal$$

The small angle component of our state is reset every iteration and not updated in the time update step [14].

$$A = I_3 + \frac{s_\omega}{\omega}[\vec{\omega}_t\times] + \frac{1 - c_\omega}{\omega^2}[\vec{\omega}_t\times][\vec{\omega}_t\times]$$

$$\Phi_\omega = I_3 + \Delta t \begin{bmatrix} 0 & \frac{I_y - I_z}{I_x}\omega_{z_t} & \frac{I_y - I_z}{I_x}\omega_{y_t} \\ \frac{I_z - I_x}{I_y}\omega_{z_t} & 0 & \frac{I_z - I_x}{I_y}\omega_{x_t} \\ \frac{I_x - I_y}{I_z}\omega_{y_t} & \frac{I_x - I_y}{I_z}\omega_{x_t} & 0 \end{bmatrix}$$

$$\Phi = \begin{bmatrix} A & 0 \\ 0 & \Phi_\omega \end{bmatrix}$$

Since we are using a MEKF, we propagate the quaternion attitude representation outside of the state vector. The quaternion attitude is propagated by $\Omega(\vec{\omega})$, identical to the quaternion kinematics used previously.

We plot the attitude from our MEKF and our numerical integration simulation, respectively. Upon inspection, these plots appear identical. Small errors in the simulation are analyzed later.

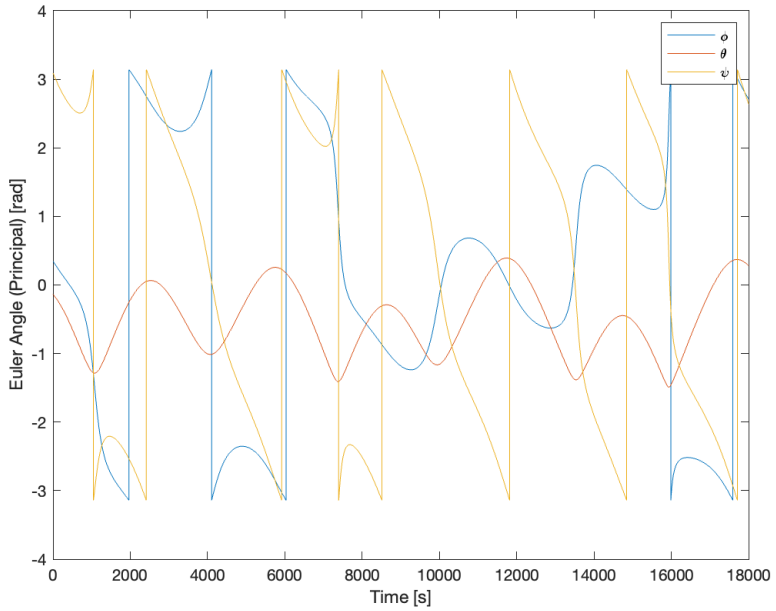


Figure 86: Attitude evolution based on MEKF time update

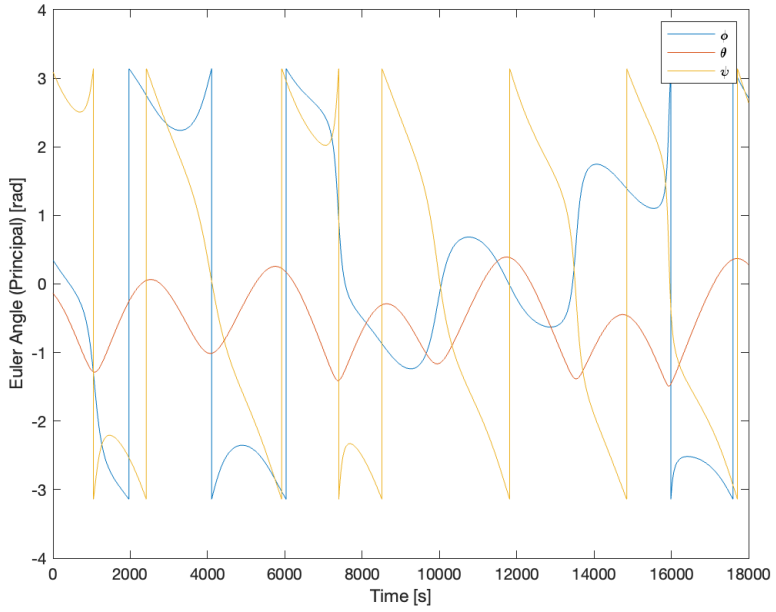


Figure 87: Attitude evolution from ”ground truth” simulation

We also plot the angular velocity from both the MEKF time update step and the numerical simulation. As with the attitude evolution, both plots appear to be the same. Note that we do not include perturbations in our simulation in our comparison, as the state would quickly diverge otherwise because the measurement update is not yet implemented in the MEKF.

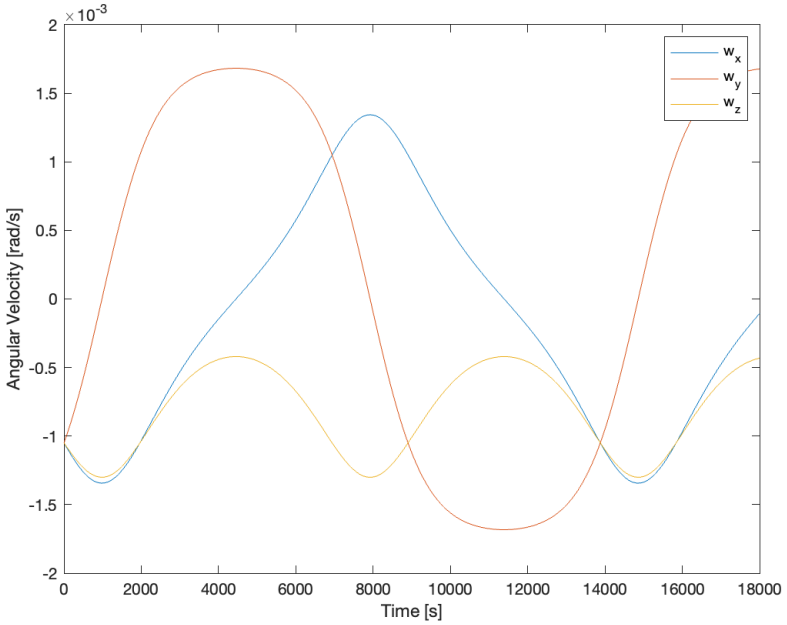


Figure 88: Angular velocity based on MEKF time update

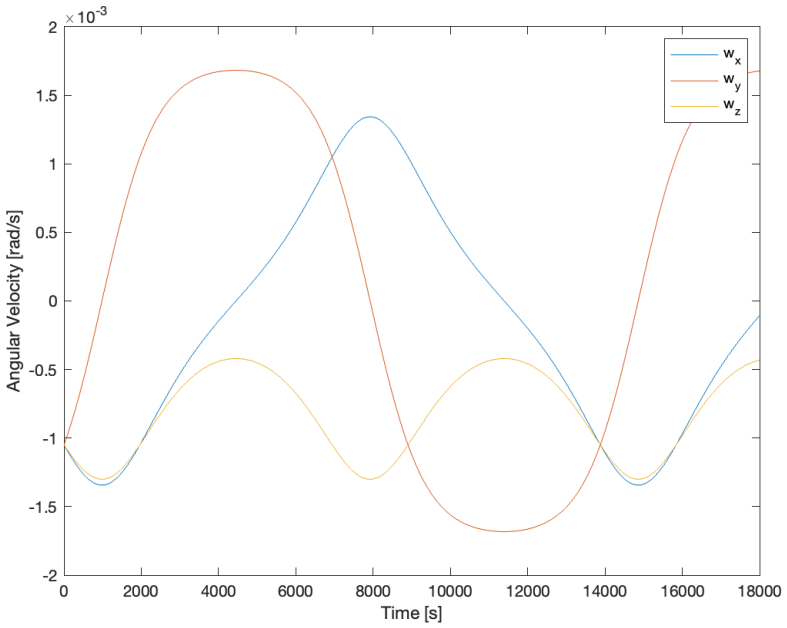


Figure 89: Angular velocity from ”ground truth” simulation

For now, we choose to model our control inputs as torque inputs (moments) about the principal axes. Thus, we can choose a B matrix as follows:

$$u_t = \begin{bmatrix} M_{x_t} & M_{y_t} & M_{z_t} \end{bmatrix}$$

$$B = \Delta t \begin{bmatrix} \frac{1}{I_x} & 0 & 0 \\ 0 & \frac{1}{I_y} & 0 \\ 0 & 0 & \frac{1}{I_z} \end{bmatrix}$$

We have not yet implemented a controller at this step. As such, we will run the MEKF with $u_t = 0$ for now.

We can choose our initial state error covariance matrix P as a diagonal matrix, with the diagonal entries corresponding to α set to 1 and the entries corresponding to ω to the variance computed from angular velocity history. This can be collected as statistics from the propagation shown previously. Once the entire MEKF is implemented, this matrix should converge over time. For now, the state error covariance matrix P has no impact on the output of just our time update.

For the purpose of this section, we arbitrarily define $Q = P_{t=0} \times 0.01$.

For just the time update, we show plots of errors for the attitude and angular velocity. We can see that the angular velocity error is very low, reaching a maximum of 10^{-7} over three orbits. Note that our raw attitude representation is in quaternions, but we convert to Euler angles for visualization. The Euler angle error is somewhat larger, beginning initially around 10^{-7} and ending around 10^{-3} over the course of the three orbits. The Euler angle error is possibly larger because there is a compounding effect of errors in computing angular velocity.

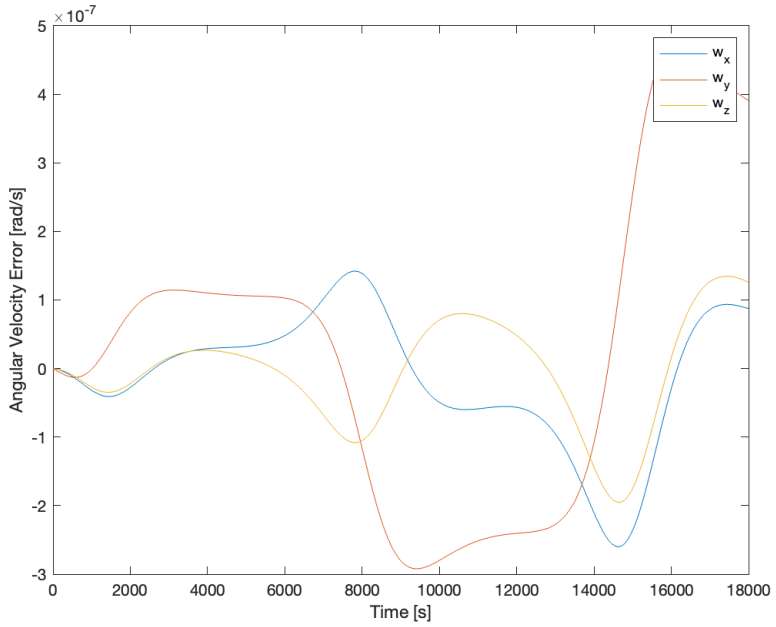


Figure 90: Attitude error between estimate and simulation

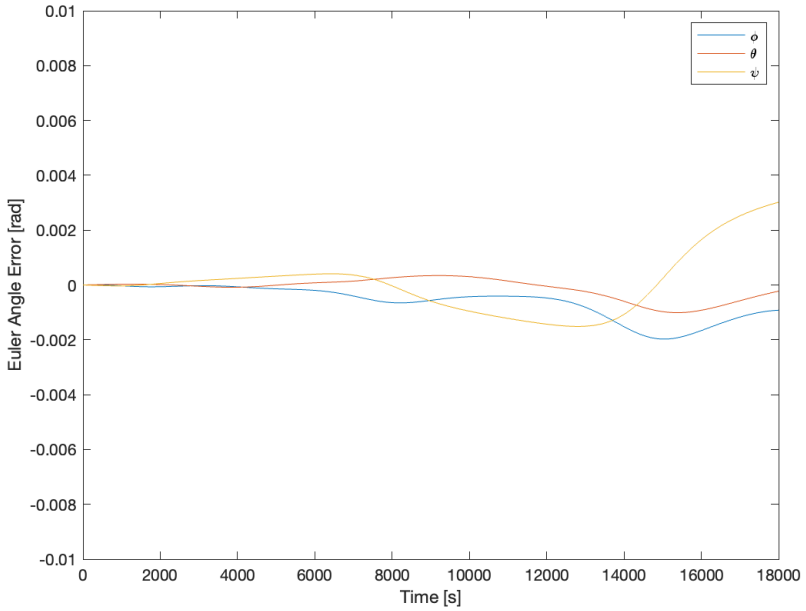


Figure 91: Angular velocity error between estimate and simulation

These results are within expectations, as we do not have a modeled measurement update step that could help reduce our errors. Also, our injected process noise Q is not beneficial to the time update alone, as Q is intended to help accommodate measurement updates.

6.13 Multiplicative Extended Kalman Filter, Measurement Update

For our measurement, we require a sensitivity matrix. The following equations define the sensitivity matrix $\mathbf{H_k}$ for the case where there are n unit direction vector measurements (sun sensors, star trackers, etc.), and a 3-axis rate gyroscope measurement.

$$\mathbf{H} = \frac{\delta \mathbf{h}}{\delta \mathbf{x}} = \begin{bmatrix} [\mathbf{h_1}\times] & \mathbf{0}_{3\times3} \\ [\mathbf{h_2}\times] & \mathbf{0}_{3\times3} \\ \dots & \dots \\ [\mathbf{h_n}\times] & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} \end{bmatrix}$$

In the equation above, $[\mathbf{h_i}\times]$ refers to the cross-product matrix for the measurement vector $\mathbf{h_i}$.

For the sensors used in our measurements, $\mathbf{R}$ is defined as a diagonal matrix of the sensor variances discussed in Problem Set 7, Problem 1 (where the sensor errors are based on the variance of the white noise in each case).

$$\mathbf{R} = \begin{bmatrix} diag(\sigma^2_{startracker}) & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & diag(\sigma^2_{sunsensor}) & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & diag(\sigma^2_{gyroscopes}) \end{bmatrix}$$

The modeled measurement vector at step k+1 was found by plugging the estimated state at step k+1 into the previously developed measurement model, without measurement noise.

Our modeled measurement uses simulated unit vectors describing directions for a star tracker and sun sensor. We also provide angular velocity data from simulation with noise to simulate

a rate gyro. Our functions for the star tracker and sun sensor in the modeled measurements are shown in the appendix.

The pre-fit residuals were calculated as the difference of the output (raw measurements with noise) and the modeled measurements based on our the measurement model and state prior to the measurement update.

The following equations are used in the measurement step calculation.

$$\mathbf{x_{k+1|k+1}} = \mathbf{x_{k+1|k}} + \mathbf{K_k}(\mathbf{y_k} - \mathbf{z_k})$$
$$\mathbf{P_{k+1|k+1}} = \mathbf{P_{k+1|k}} - \mathbf{K_k}\mathbf{H_k}\mathbf{P_{k+1|k}}$$
$$\mathbf{K_k} = \mathbf{P_{k+1|k}}\mathbf{H_k}^T[\mathbf{H_k}\mathbf{P_{k+1|k}}\mathbf{H_k}^T + \mathbf{R_k}]^{-1}$$

We implement the measurement update in code with the following function. This MATLAB function is used as part of our Simulink model.

```
function [qkplus,wkplus,Pkplus,xkplus,z] = ...
    measurementUpdate(qkminus, Pkminus, xkminus, yk, hk, u, sensors)
    % Get R
    numVectMeasurements = size(hk,2)-1;
    R = diag([repelem(sensors.trackerError, ...
        3*(numVectMeasurements-1)), ...
         repelem(sensors.sunError, 3), repelem(sensors.gyroError, ...
            3)]).^2;

    % Compute residual
    z = yk - hk;

    % Adjust yk and hk vectors
    yk = yk(:);
    hk = hk(:);

    % Get Hk
    Hk = zeros([3*(numVectMeasurements+1), 6]);
    for n = 1:3:(3*numVectMeasurements)
        Hk_vect = crossMatrix(hk(n:n+2));
        Hk(n:n+2,:) = [Hk_vect, zeros(3)];
    end
    Hk(end-2:end,:) = [zeros(3), eye(3)];
    Kk = (Pkminus * Hk') / (Hk * Pkminus * Hk' + R);
    xkplus = xkminus + Kk * (yk - hk);
    Pkplus = Pkminus - Kk * Hk * Pkminus;
    wkplus = xkplus(4:6);
    ax = xkplus(1); ay = xkplus(2); az = xkplus(3);

    % Attitude update
    qkplus = [1, az/2, -ay/2, ax/2; ...
        -az/2, 1, ax/2, ay/2; ...
        ay/2, -ax/2, 1, az/2; ...
        -ax/2, -ay/2, -az/2, 1] * qkminus;
end
```

Our entire MEKF is modeled in Simulink, as shown in the diagram below.

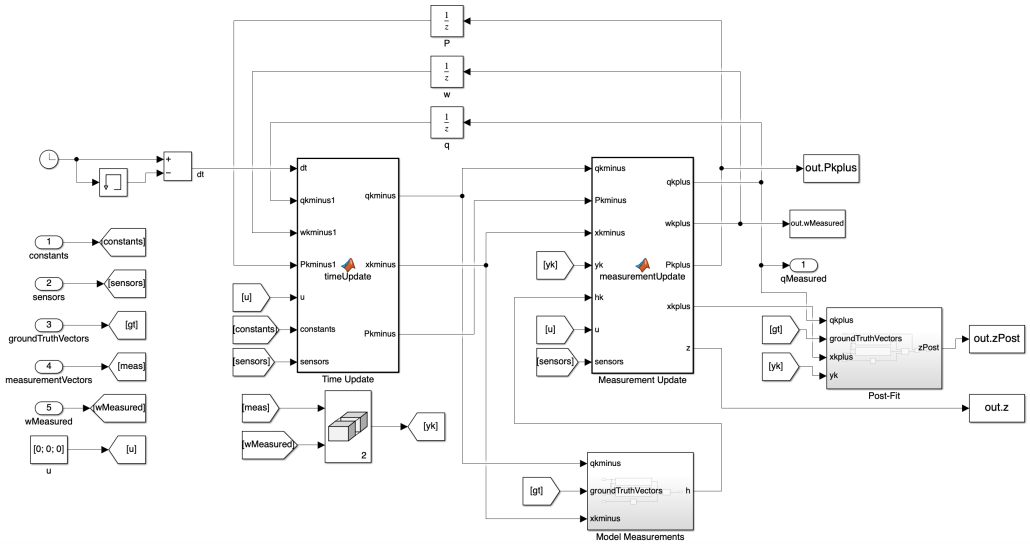


Figure 92: Simulink model of MEKF

6.14 Multiplicative Extended Kalman Filter, Errors

Figure 93 shows the attitude estimate errors with the MEKF method. As expected, the errors are very small compared to the attitude determination methods used in the previous section.

Figure 94 shows the estimated attitude determination errors based on the covariance of the filter. Comparing this with Figure 93, it seems that this method may under-report the error and is not necessarily the most accurate representation of error. Figure 95, makes the difference between these two values even clearer. In short, there is are multiple orders of magnitude difference between the actual error and what the covariance bounds would predict. This could be caused by a couple of factors. After further inspection, it seems that the small covariance values come from the measurement step, where they get shrunken from roughly $1 - 2\,\mathrm{deg}$ to the tiny values plotted. It seems possible that the values in the **R** matrix might be too low, suggesting that the measurement are being trusted too much, leading to covaraince values that are too low. Tuning the **R** matrix more could lower this value; however, the current estimate has proven effective in state estimation.

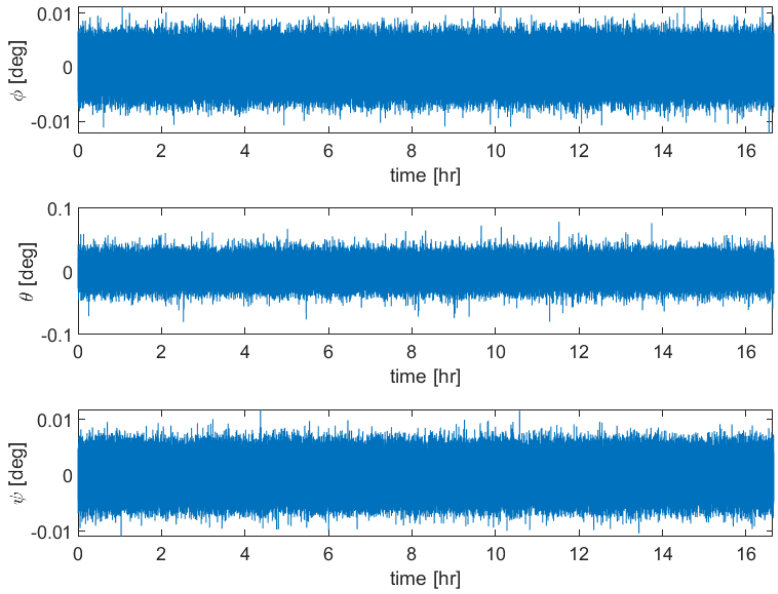


Figure 93: Errors between MEKF measurements and ground truth

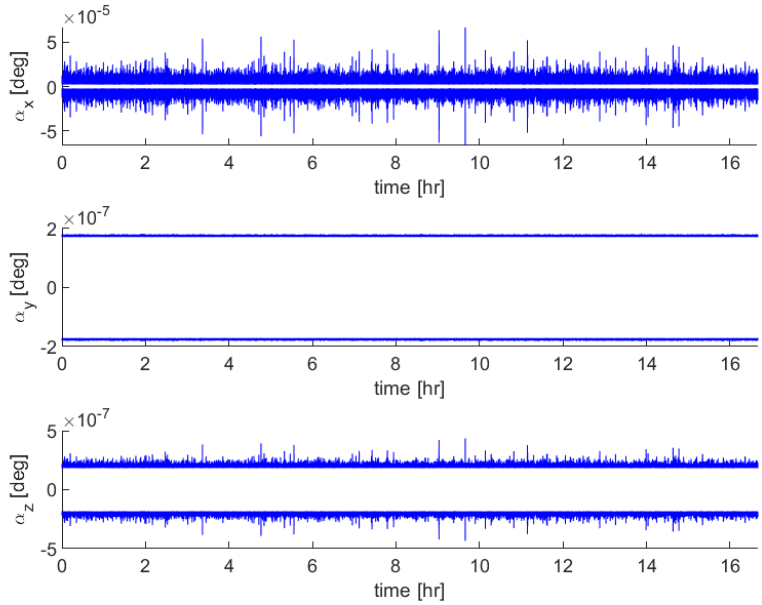


Figure 94: Estimated attitude estimation errors from covariance (small angles)

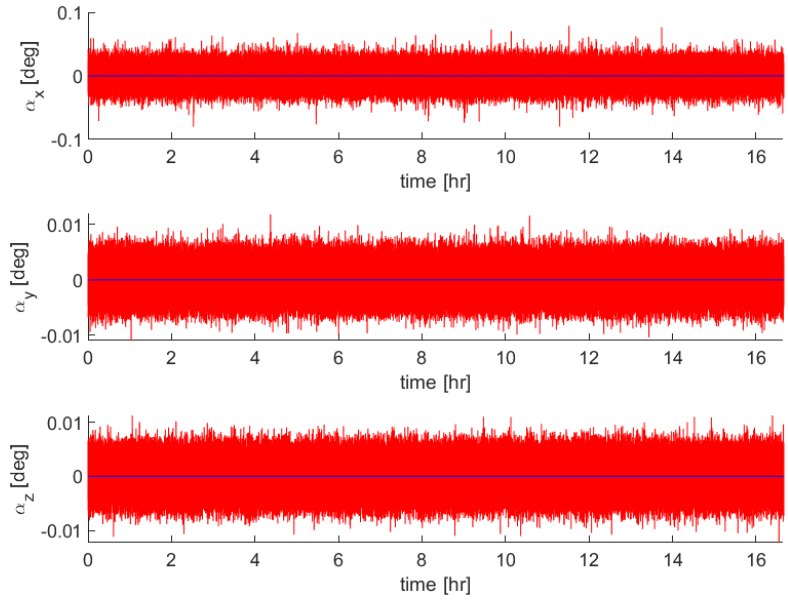


Figure 95: Estimated Error (95% confidence interval) vs Actual Errors

The post-fit residuals z are found by taking the updated state from the measurement update step and computing a modeled measurement using our measurement model again. Figures 96 and 97 below show the difference in the norms of the pre-fit and post-fit residuals. As expected, the post-fit residuals are slightly smaller than the pre-fit residuals. However, the difference between pre-fit and post-fit sensor noise is much better for the gyroscope measurements than the unit direction vector measurements.

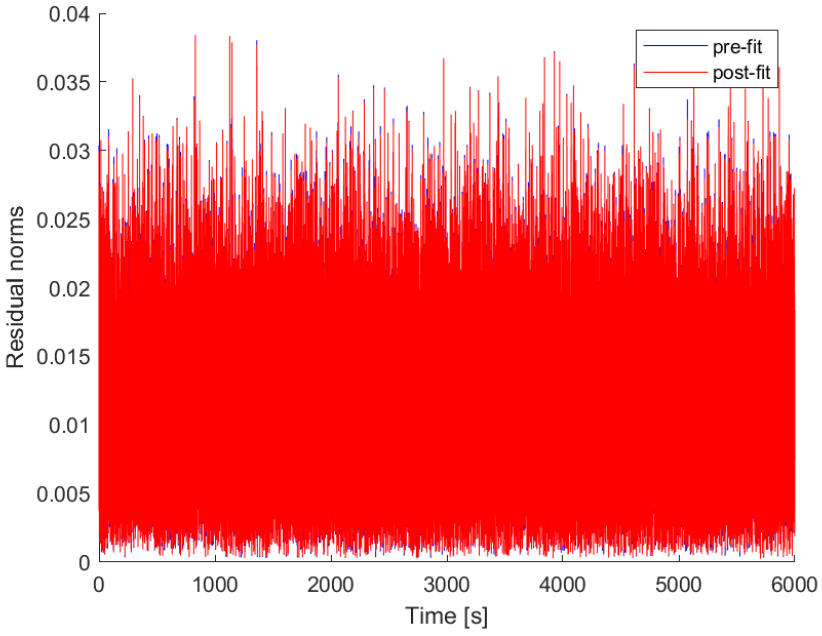


Figure 96: Norm of pre-fit and post-fit direction-vector residuals

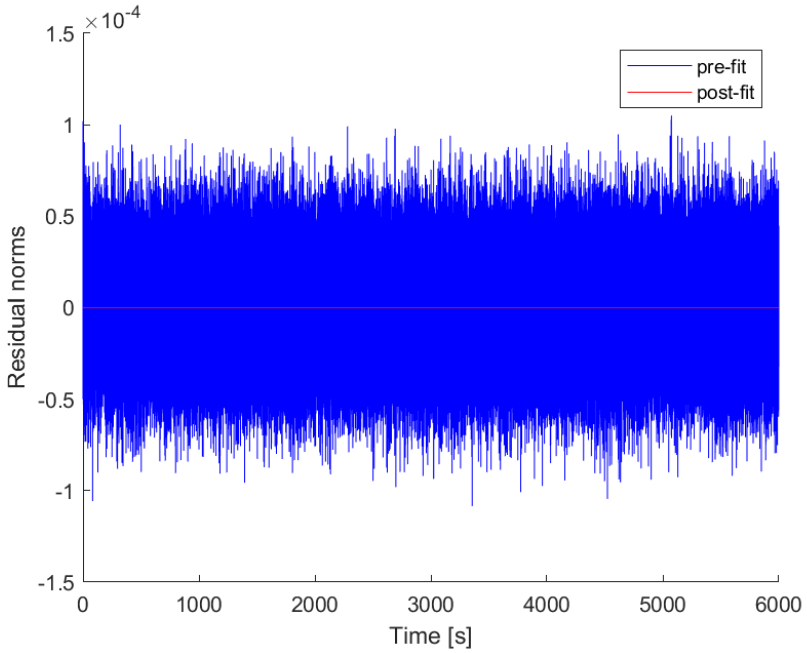


Figure 97: Pre-fit and post-fit gyroscope residuals

Investigating the measurement model, noise model, and sensitivity matrix for the unit vector-based measurements, we may be able to improve the performance shown in Figure 96. This is an area for potential future work, as well as the topics discussed in the following section.

7 CONTROL

7.1 Actuator Specifications

We size reaction wheels, magnetorquers, and thrusters in this section. Initially, we only simulate the reaction wheels, but we include magnetorquers later. At this point, we will couple state estimation with the controller by providing the MEKF with the torque control input.

The calculations for actuator sizing are based on the worst case disturbance torque (M_d), orbit period (P), largest expected magnetic field (B), and estimated moment arm of a thruster (b). This is used to find the required momentum storage of a reaction wheel (L), required dipole of a magnetorquer (D), and required thrust of a thruster (T), based on the equations below. These values are presented in Table 9.

$$L = \frac{M_d P}{4\sqrt{2}}$$
$$D = \frac{M_d}{B}$$
$$T = \frac{M_d}{b}$$

Table 9: Actuator Parameters for Disturbance Rejection

L [Nms]	D [Am^2]	T [N]
2.5392	55.0737	0.004

In addition to disturbance rejection, the actuators should also be sized in order to perform necessary maneuvers, meaning that the values in Table 9 may need to be greater when considering all the operational modes of the satellite ADCS system. The NISAR satellite uses four $50\,\mathrm{Nms}$ reaction wheels, a combination of three $565\,\mathrm{Am}^2$ and $350\,\mathrm{Am}^2$ magnetorquers, and four $1\,\mathrm{N}$ thrusters for attitude control [4]. Using these metrics, we found equivalent commercially available actuators to estimate other key actuator parameters. For reaction wheels, we referenced the ASTROFEIN RW3000, which has properties listed in Table 10. For thrusters, we referenced the Ariane Group 1N Hydrazine Thruster, which has properties listed in Table 11. For the magnetorquer, we only require the dipole moment to perform our sizing.

Table 10: RW 3000 Reaction Wheel Data [15]

Model Name	Momentum	Moment of Inertia	Maximum RPM
RW 3000	$50\,\mathrm{Nms}$	$0.119\,\mathrm{kgm}^2$	4000

Table 11: Ariane Group Thruster Data

Model Name	Thrust (T)	Specific Impulse (I_{sp})
1N Hydrazine Thruster	1 N	220 s

7.2 Actuator Models

Our actuator model accepts M_c as a torque command from the control law and tracks the angular momentum state of our reaction wheels. We can propagate the angular momentum by computing the rate of change of the angular momentum. This is also equivalent to commanding a change in angular velocity of the reaction wheels.

$$\dot{\vec{L}}_w = \vec{A}^*(-\vec{M}_c - \vec{\omega} \times \vec{A}\vec{L}_w)$$

We can incorporate estimation error by passing in ω_{est} when computing $\dot{\vec{L}}_w$, and then propagating the dynamics using ω_{true} in the same formula for $\vec{M}_c$.

For the layout of our system. We position four reaction wheels in a tetrahedral configuration–this system is overactuated but allows for redundancy in case of reaction wheel failure. Our mounting matrix is as follows:

$$\vec{A} = \frac{1}{\sqrt{3}} \begin{bmatrix} -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Prior to implementing the control law, we use an arbitrary sinusoidal signal as our torque command and plot the resulting angular velocities of our reaction wheels. This output is coupled with our simulation of the spacecraft dynamics.

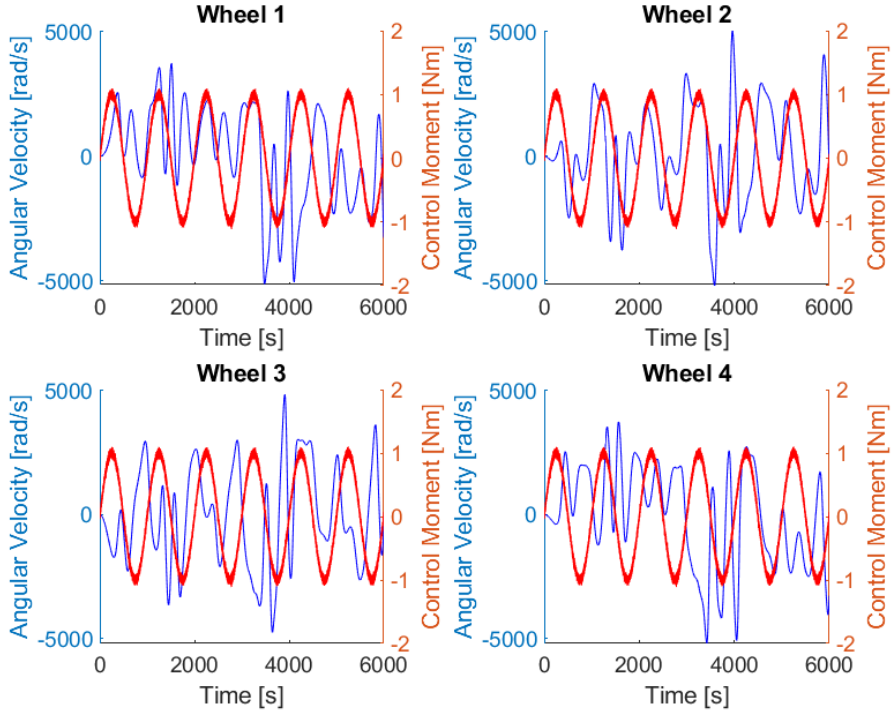


Figure 98: Reaction wheel angular velocities for arbitary control moment inputs

7.3 Control Law

We implement the control law using the following equations, where f is frequency of the actuator response and I_i is the moment of inertia about principal axis i. The control law differs

slightly for each axis based on the moment of inertia about that axis.

$$M_{c,i} = -K_{p,i}\alpha_i - K_{d,i}\dot{\alpha}_i$$
$$K_{p,i} = \frac{f^2}{I_i}$$
$$K_{d,i} = 2\sqrt{I_i\left[3n^2(I_k - I_j) + K_{p,i}\right]}$$

To obtain small angle α_i, we compute the rotation between the current attitude and the target attitude by using the direction cosine matrices of each relative to the inertial frame. From this new rotation A_E from target attitude to current attitude, we can directly extract α_x, α_y, and α_z from the corresponding indices in the matrix to create our linear control law. Similarly, for a nonlinear control law, we can used the following formula.

$$\vec{M}_{c,i} = -K_{p,i}\frac{A_{E,jk} - A_{E,kj}}{2} - K_{d,i}\omega_i$$

This formula effectively takes an average of the relevant indices for each α_i in A_E. Before adding sensor errors, we obtain the following control result for our desired (originally unstable) Earth-pointing attitude, which we have detailed previously in this report.

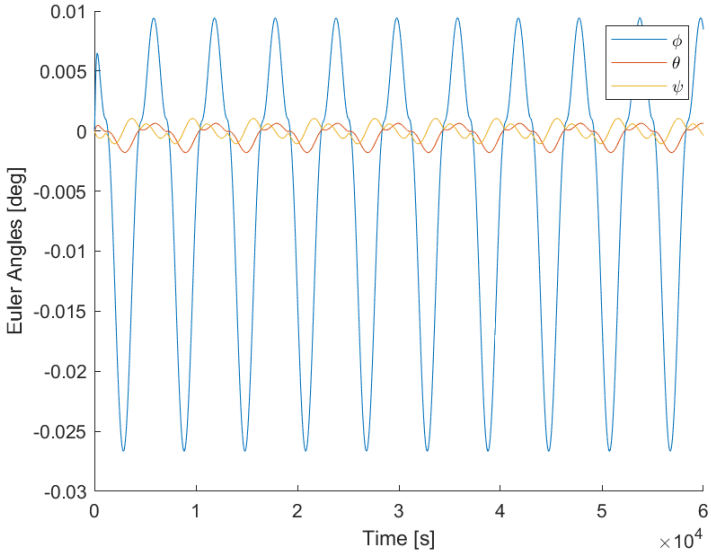


Figure 99: Control error over 10 orbits without sensor errors

7.4 Control and Estimation Errors

Upon adding state estimation error, our control error changes, as shown in Figure 100 Note that we are implementing the nonlinear control law here, but we also have implemented the linear control law and obtained nearly identical results–this is as expected, since we are attempting to reject small disturbances, and the difference between the linear and nonlinear control laws should be approximately zero in this regime.

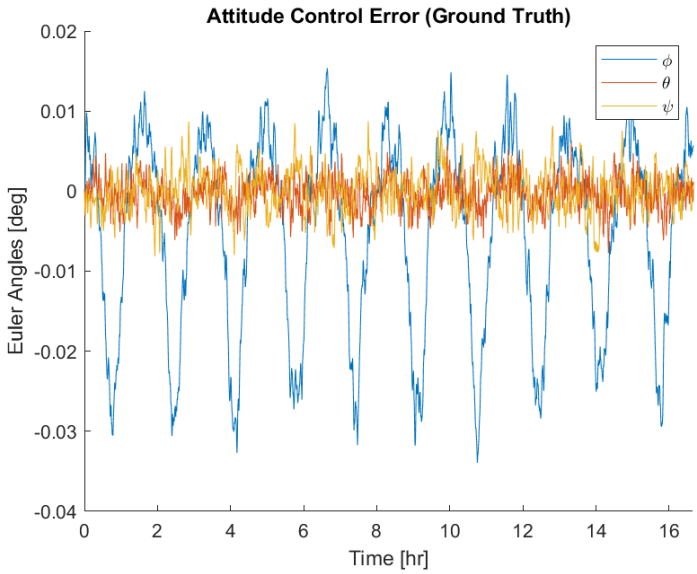


Figure 100: Angular error relative to target attitude over 10 orbits

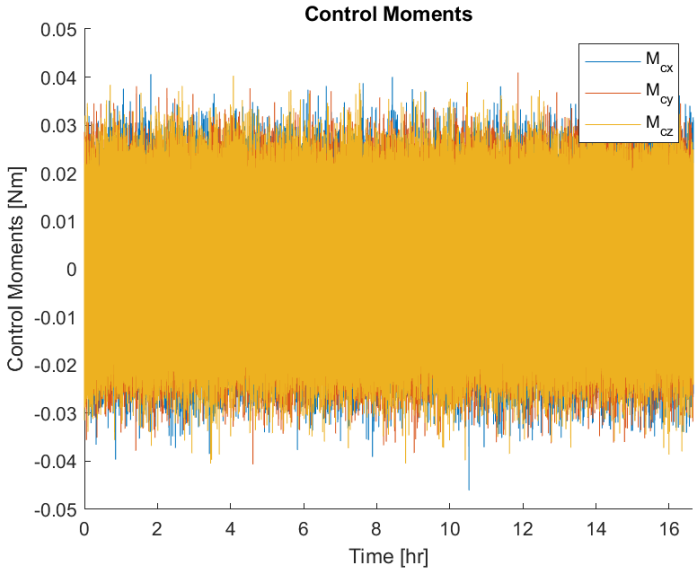


Figure 101: Control torque commands over 10 orbits

For simply pointing the radar instrument directly at the ground below, we are able to meet the NISAR pointing requirement of less than 273 arcsec (approximately 0.075 degrees). In Figure 101, we show the torque commands issued by our control law. We also show the attitude determination error based on our filter in Figure 102.

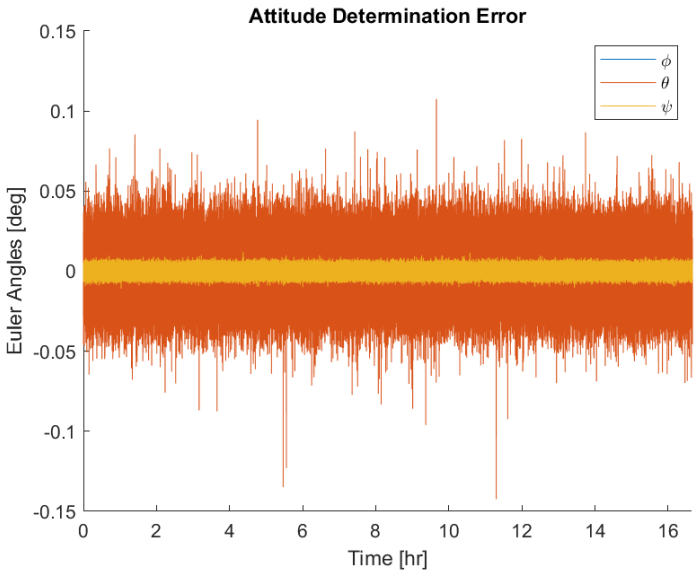


Figure 102: Attitude determination error over 10 orbits

In Figure 103, we can see that the angular velocities of our four reaction wheels grow over time. This will eventually lead to saturation and require desaturation over longer periods of time.

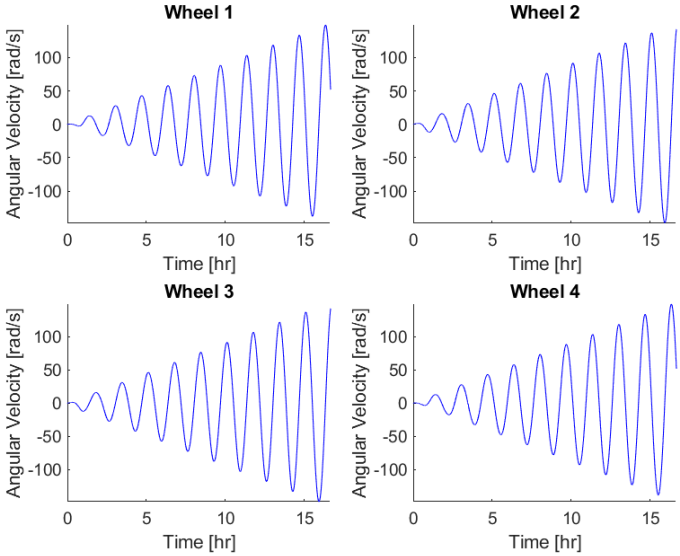


Figure 103: Reaction wheel angular velocity over 10 orbits

8 ADVANCED TOPICS

8.1 Magnetorquers

We now add magnetorquers to our set of modeled actuators. The magnetorquer works by creating a dipole which interacts with the Earth's magnetic field and generates a torque on the spacecraft. A magnetorquer is a very suitable choice of actuator for NISAR, as the magnetic field is stronger in LEO than in higher orbits, allowing for increased effectiveness.

The following equation shows the expected moment generated by a magnetorquer, where $\vec{m}$ is the magnetoruqer's dipole moment and $\vec{B}$ is the magnetic field of the Earth.

$$\begin{aligned}\vec{M}_{magnetorquer} &= \vec{m} \times \vec{B} \\ &= \begin{bmatrix} 0 & B_z & -B_y \\ -B_z & 0 & B_x \\ B_y & -B_x & 0 \end{bmatrix} \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}\end{aligned}$$

The amount of torque generated can vary at any given moment depending on the attitude of the spacecraft (the real NISAR spacecraft has a different torque rod configuration in one axis) and the location in the magnetic field, which we estimate using a fourth-order model described previously in this paper. Note that the cross product matrix of the magnetic field is singular, making it difficult to solve for the actuator commands. This system, assuming magnetorquer-only operation, is not controllable in all axes without a changing magnetic field. Additional techniques using trajectory analysis may be necessary to find a generally applicable controller for full 3-axis control using magnetorquers. Alternatively, combining the two magnetorquers with a reaction wheel can also make the system controllable.

However, we are able to find a control law that does not require these techniques for desaturation. We use the following equation from Hogan and Schaub to model our magnetorquers for reaction wheel desaturation, where $\vec{m}$ is the dipole, k_M is a gain parameter, $\vec{B}$ is the magnetic field, and $\vec{H}_D$ is the difference between system and desired bias angular momentum–in our case, it is simply the angular momentum of our reaction wheel system [16].

$$\vec{m} = -\frac{k_M}{B^2} \vec{H}_D \times \vec{B}$$

The torque is then simply found as before from the dipole and magnetic field.

$$\vec{M}_{magnetorquer} = \vec{m} \times \vec{B}$$

For the purpose of desaturation, we assume that the net dipole can be arbitrarily generated based on a 3-axis magnetorquer system. We assume the commands to individual magnetorquer rods, however they may be configured, are computed automatically, although we verify that the magnitude of the dipole and resulting torque are realistic in selecting our gain coefficient.

For our modeled spacecraft, we elect to use the magnetorquers solely for desaturation of the reaction wheels, retaining the reaction wheels as our primary attitude control actuator. In real-world operation, continuous momentum management may be performed by a combination of magnetorquers and reaction wheels.

Additionally, it is important to note that magnetorquer operation on a real spacecraft would have to operate in pulses such that magnetometer readings can be taken without being affected

by the noise generated by the magnetorquers. As our system does not include the magnetorquer contribution to the magnetic field, we do not model the magnetometer/magnetorquer duty cycle in our simulation.

8.2 Reaction Wheel Desaturation

In this section, we investigate the desaturation of the reaction wheels in our system using the magnetorquers. Desaturation of reaction wheels is an important part of the attitude control system, enabling persistent operation on orbit for satellites which would otherwise lose attitude control due to momentum saturation. Using magnetorquers is a favorable technique for momentum management because it does not expend propellant as with firing thrusters, instead using electrical power (which can be acquired via solar panels) to operate the magnetorquers. This is especially important for NISAR given the frequency of desaturation required, as our spacecraft experiences substantial disturbance torques from gravity gradient and atmospheric drag in LEO. For other satellites positioned in significantly higher orbits, using thrusters for desaturation may be preferable owing to the diminished magnetic field and smaller disturbance torques.

The control law is modified to incorporate a desaturation mode which is activated when any of the four reaction wheels surpasses a maximum angular momentum threshold. This desaturation mode persists until all reaction wheels have reached a sufficiently low angular momentum, also defined by a threshold. During this desaturation mode, we operate our magnetorquers at a nominal dipole moment, which is parameterized by a gain coefficient. We elect to operate the magnetorquers in bursts during desaturation periods rather than continuously in order to conserve power, although a continuous operation approach may also be used–this choice is likely dependent on power budgeting onboard the real spacecraft, where less power may be available during activities such operating the SAR instruments.

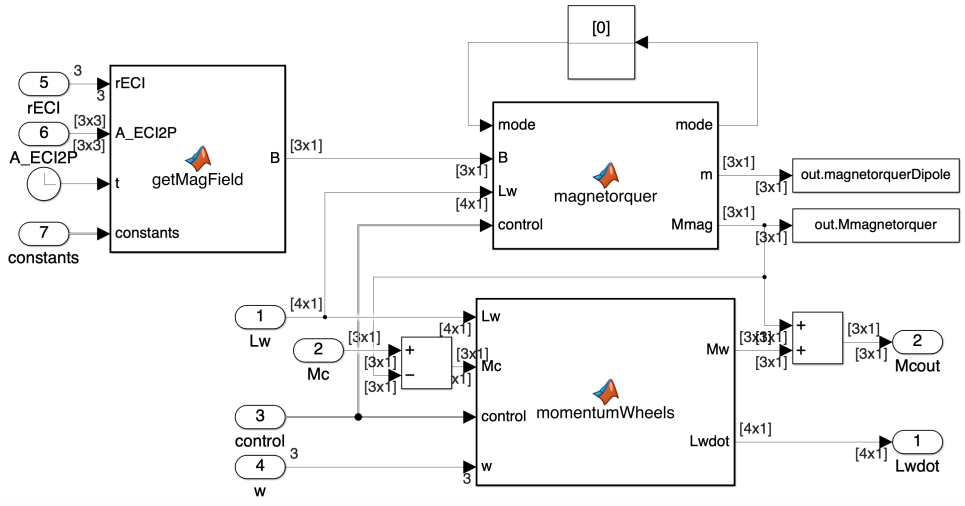


Figure 104: Simulink diagram of actuator subsystem with magnetorquers for desaturation

As discussed in the previous section, to generate the desired torque for desaturation, we must determine the direction of the torque which will permit us to slow down the reaction wheels while maintaining the desired attitude. We can compute the appropriate dipole moment for our magnetorquer system based on the net angular momentum vector of our reaction wheel system.

To fully achieve desaturation, we must also include the reaction wheels in the loop–a corresponding command must be sent to the reaction wheels as well. We can achieve this by taking the difference of the original torque command (based on our original attitude error control law) and the torque from the magnetorquer (an estimate which may be based on measurements, although our simulation uses the exact value for simplicity) and sending this as the new torque command to our reaction wheels. We can also neglect to include the magnetorquer torque in the torque command and simply rely on the controller to account for the attitude error, essentially treating the magnetorquer torque as another disturbance–albeit a beneficial one–but this leads to increased attitude control error during desaturation.

Implementing this controller in simulation, we can observe that the desaturation control law using magnetorquers does indeed work to desaturate the reaction wheels. Figure 105 shows that the reaction wheel never saturates, and we require desaturation every 12-14 orbits, or approximately daily.

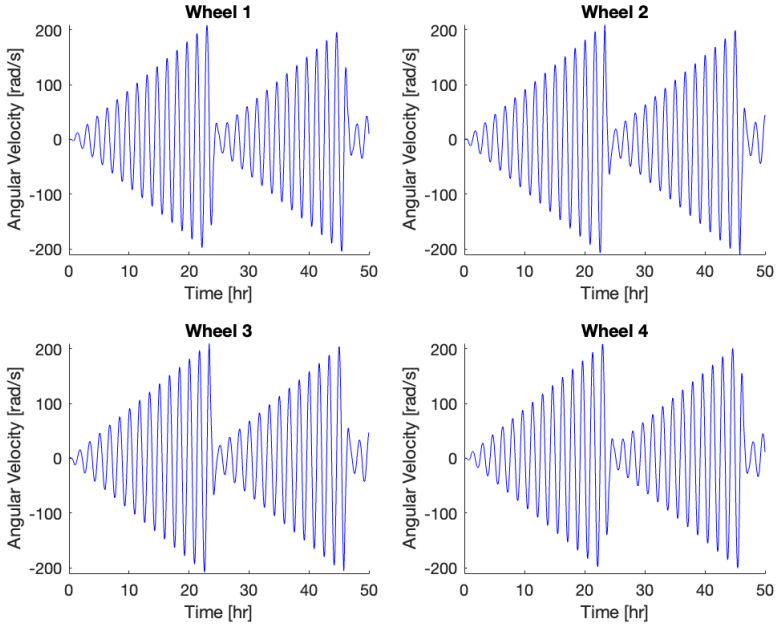


Figure 105: Angular velocity of reaction wheels over 30 orbits

In this implementation, our desaturation mode is activated whenever any reaction wheel reaches half of the maximum angular momentum capability and ceases whenever all reaction wheels are desaturated to a tenth of this threshold. Our gain is chosen as $k_M = 0.001$, on the same order of magnitude as the value ($k_M = 0.003$) used in Hogan and Schaub [16]. Desaturation occurs relatively quickly, taking approximately an hour, and over the course of 30 orbits, only two desaturation periods are required.

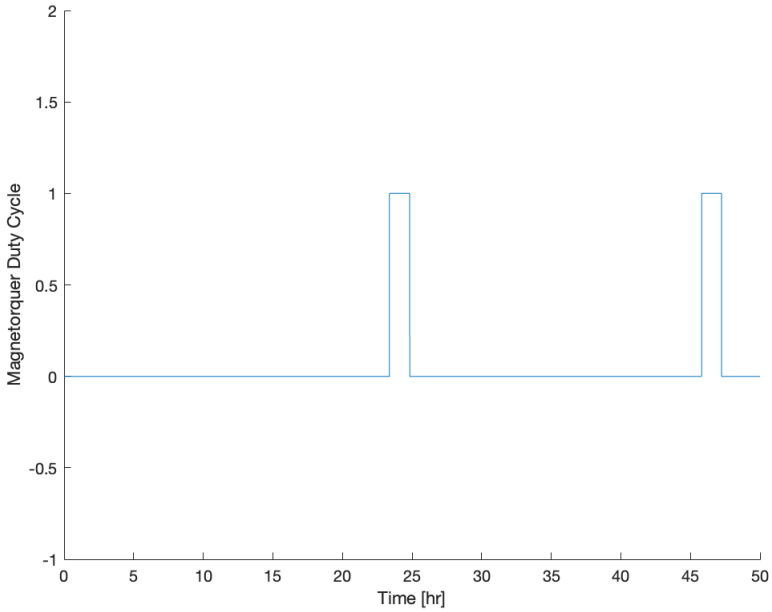


Figure 106: Desaturation mode over 30 orbits (0 is off, 1 is on)

We also verify that the attitude control error remains within our bounds for the duration of the simulation. Figure 107 shows attitude control error for the case where our controller accounts for the magnetorquer torque, while Figure 108 shows attitude control error when the magnetorquer is not accounted for.

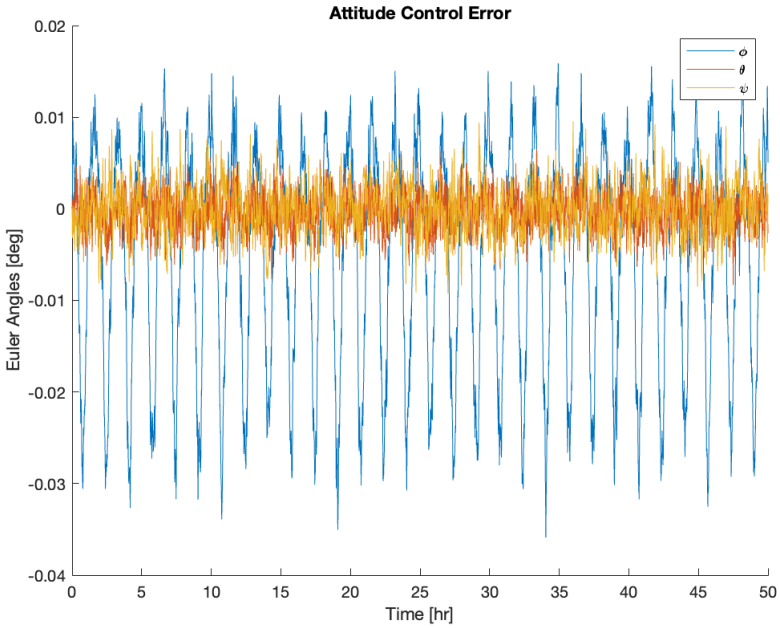


Figure 107: Attitude control error, considering magnetorquer in reaction wheel command

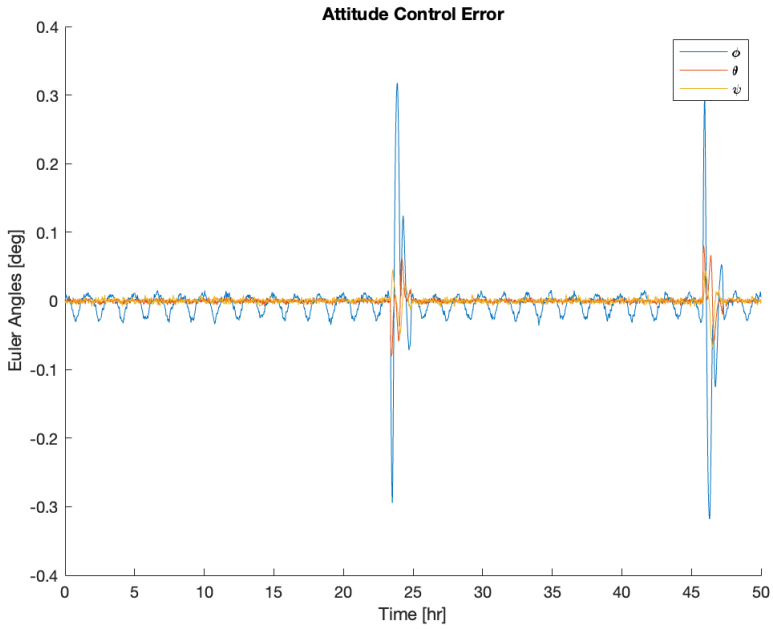


Figure 108: Attitude control error, not modeling for magnetorquer torques

The attitude control error remains consistently very low–identical to nominal controller operation in the previous section–as expected for the case where we account for the additional torque. However, if we do not include this additional torque, while attitude control error remains the same as before when magnetorquers are not operating, we observe periods of relatively large attitude control error during desaturation. Notably, these errors exceed our science pointing requirement.

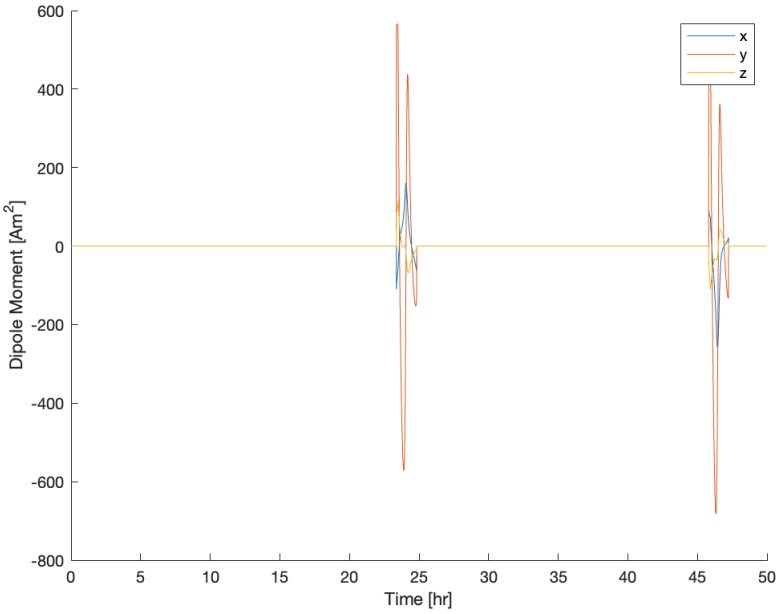


Figure 109: Magnetorquer dipole moments over 30 orbit period

We can infer from this data that a noisy or inaccurate estimate of the resultant torque from the magnetorquers has an impact on pointing accuracy during magnetorquer operation, but it not enough to destabilize the system, which is able to resume normal pointing accuracy immediately after desaturation is complete. This may be acceptable if no activities requiring precision control are occurring in parallel with desaturation and if the desaturation period is relatively short. Otherwise, feeding an accurate prediction of the magnetorquer torque into the actuator system control input allows us to maintain extremely precise pointing accuracy regardless of operating mode.

Additionally, we show the magnetorquer dipole moment and torque over the duration of the simulation in Figures 109 and 110.

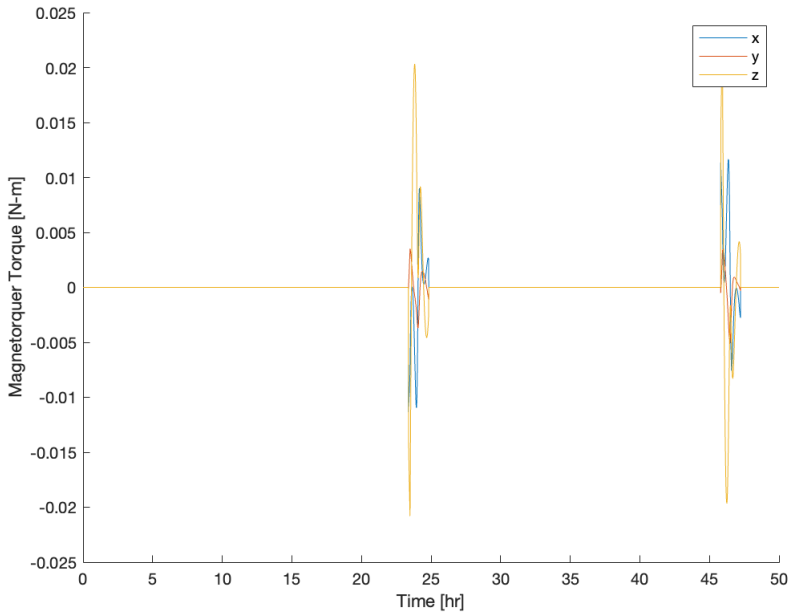


Figure 110: Magnetorquer torques over 30 orbit period

The magnitude of the dipole moment in any axis never exceeds 565 A-m^2, as we have modeled maximum limits for our magnetorquers based on specifications. Our limiting model normalizes the net dipole such that the maximum dipole moment in any axis does not exceed the limit while maintaining the correct direction of the dipole–that which directly aligns with the angular momentum of the reaction wheels for desaturation.

Even so, we only experience such large magnitudes for brief periods. Most operation does not exceed 400 A-m^2. This is a reasonable value for the dipole moment of a satellite of such size [17]. Note that this value is much larger than the dipole required for disturbance rejection, as we are effectively canceling out accumulated momentum from disturbances over a much longer period. The torques generated by the magnetorquers during desaturation are approximately 0.02 N-m or less, which is also appropriate for a satellite of NISAR's size. These results were considered when tuning gain parameter k_M and resulted in our selected value of 0.001.

9 CONCLUSION

9.1 Summary

The objective of this paper was to develop an ADCS system for a satellite based on NASA and ISRO's joint NISAR mission. NISAR is a satellite with a synthetic aperture radar (SAR) in a sun-synchronous low Earth orbit (LEO) intended to gather data about Earth's surface. Specifically, science operations require the radar to be pointed at Earth precisely (273 arcseconds or 0.075° of pointing accuracy). To investigate the ADCS system for such a mission, we performed the following tasks:

- Identified specifications and physical characteristics
- Discretized the satellite to determine mass and surface properties
- Modeled the dynamics of the satellite using mass properties
- Propagated dynamics and kinematics using models and analyzed stability
- Introduced perturbations based on satellite properties and environmental factors
- Investigated attitude determination and Kalman filtering techniques
- Simulated controller-in-the-loop operation
- Modeled reaction wheel desaturation using magnetorquers

We used numerical simulation in MATLAB and Simulink extensively in this paper. Importantly, we demonstrated that the nominal operating attitude for SAR operation was an unstable equilibrium, and we designed an attitude determination and control system that was able to stabilize the satellite to within the pointing accuracy constraints in the presence of realistic modeled disturbance torques. We also showed that the magnetorquer system on board can successfully desaturate the reaction wheels, allowing extended on-orbit operation.

Much additional work can be done in examining additional modes of operation such as slew and detumble. This work includes attitude determination and control design, especially for cases of nonlinearity. Electrical power and thruster analyses will also be important for planning real-world operations

9.2 Lessons Learned

Developing a simulated model of NISAR was an incredible learning experience for the authors. We learned a full-stack approach to ADCS design: orbits, dynamics, kinematics, disturbance modeling, state estimation, actuator modeling, and control. It became clear to us from the AA 279C lectures that there are a lot of tools in ADCS design, and using them in the project was a great way to learn through application.

We also learned much about the software tools used in generating the simulations and report. MATLAB and Simulink were used extensively for every task, and adapting our initially MATLAB-based model to Simulink was a great way to build our first major model in Simulink. Git version control has been extremely important for managing large volumes of code and LaTeX files–integration of code, image generation, and LaTeX contributed significantly to the authors' ability to write a paper of this quality.

Perhaps the most important lesson learned was about the kind of decision-making that lies at the heart of engineering. Throughout this project, many key decisions were made about what exactly to model and the fidelity of the models required. In these decisions, we would need to weigh the benefits of increasing model fidelity (often the accuracy of our simulation) versus the costs (often computational resources and engineering time).

10 REFERENCES

[1] K. H. Kellogg, S. Thurman, W. Edelstein, *et al.*, "NASA's SMAP Observatory," in *2013 IEEE Aerospace Conference*, Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2013. DOI: 2014/44370. [Online]. Available: https://hdl.handle.net/2014/44370.

[2] *EOS SAR Satellites*, Sep. 2021. [Online]. Available: https://eossar.com/technology/.

[3] *Capella Space*, Apr. 2024. [Online]. Available: https://www.capellaspace.com/.

[4] K. H. Kellogg, P. Barela, R. Sagi, *et al.*, "NASA-ISRO Synthetic Aperture Radar (NISAR) Mission," in *2020 IEEE Aerospace Conference*, Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2020. DOI: 2014/51150. [Online]. Available: https://hdl.handle.net/2014/51150.

[5] N. N. Stavros, S. Owen, C. Jones, and B. Osmanoglu, *NISAR Applications*, version V2, 2018. DOI: 2014/49181. [Online]. Available: https://hdl.handle.net/2014/49181.

[6] P. Siqueira, *The NISAR Mission*, 2018. [Online]. Available: https://climate.esa.int/sites/default/files/D1_S1_T7_Siqueira.pdf.

[7] Spectrolab, *Space solar panels datasheet*. [Online]. Available: https://www.spectrolab.com/DataSheets/Panel/panels.pdf.

[8] L3Harris, *Prebuilt 12-meter s- or l-band reflector*, Mar. 2021. [Online]. Available: https://www.l3harris.com/sites/default/files/2021-03/l3harris-prebuilt-12m-unfurlable-mesh-reflector-spec-sheet-sas.pdf.

[9] S. G. McCarron, "The effect of a change in orientation of a rectangular four-paddle solar array on the spin rate of a satellite," Goddard Space Flight Center, National Aeronautics and Space Administration, Tech. Rep., Jan. 1966.

[10] "NASA-ISRO SAR (NISAR) Mission Science Users' Handbook," Jet Propulsion Laboratory, National Aeronautics and Space Administration, Tech. Rep., 2019. [Online]. Available: https://nisar.jpl.nasa.gov/system/documents/files/26_NISAR_FINAL_9-6-19.pdf.

[11] Rockwell Collins, *Ht-rsi high motor torque momentum and reaction wheels 14 – 68 nms with integrated wheel drive electronics*, 2007.

[12] J. R. Wertz, *Spacecraft Attitude Determination and Control*. Springer Dordrecht, 1978. DOI: https://doi.org/10.1007/978-94-009-9907-7.

[13] *Cvg new space gyroscopes*. [Online]. Available: https://www.innalabs.com/cvg-new-space.

[14] C. Beierle, A. Norton, B. Macintosh, and S. D'Amico, "Two-stage attitude control for direct imaging of exoplanets with a CubeSat telescope," in *Space Telescopes and Instrumentation 2018: Optical, Infrared, and Millimeter Wave*, M. Lystrup, H. A. MacEwen, G. G. Fazio, N. Batalha, N. Siegler, and E. C. Tong, Eds., International Society for Optics and Photonics, vol. 10698, SPIE, 2018, 106981Z. DOI: 10.1117/12.2314233. [Online]. Available: https://doi.org/10.1117/12.2314233.

[15] ASTROFEIN, *Astrofein reaction wheel family*. [Online]. Available: https://www.astrofein.com/wp-content/uploads/2023/04/ASTROFEIN-Reaction-Wheel-Family_20230425.pdf.

[16] E. A. Hogan and H. Schaub, "Three-axis attitude control using redundant reaction wheels with continuous momentum dumping," *Journal of Guidance, Control, and Dynamics*, vol. 38, no. 10, pp. 1865–1871, Oct. 2015, ISSN: 1533-3884. DOI: 10.2514/1.g000812. [Online]. Available: http://dx.doi.org/10.2514/1.G000812.

[17] C. Gibson and M. Polites, “A low-power magnetic torquer for satellite attitude control,” in *Technology Today and Tomorrow, Space Congress*, Canaveral Council of Technical Societies, 1971.

A APPENDIX A: CODE

The following MATLAB code are used in the original AA 279C problem sets. The full code (including Simulink models) and resource files can be found in the GitHub repository: `https://github.com/zhao-harry/aa-279c-project`

A.1 Problem Set 1 – Mass and Surface Properties

```
%% Center of mass
cm = computeCM('res/mass.csv');

%% Moment of inertia
origin = [0;0;0];
I = computeMOI('res/mass.csv',origin);

%% Surface properties
[barycenter,normal,area] = surfaces('res/area.csv');

%% Plot spacecraft with body axes
figure
gm = importGeometry('res/NISAR.stl');
pdegplot(gm);
quiver = findobj(gca,'type','Quiver');
textx = findobj(gca,'type','Text','String','x');
texty = findobj(gca,'type','Text','String','y');
textz = findobj(gca,'type','Text','String','z');
set(quiver,'XData',[0;0;0])
set(quiver,'YData',[0;0;0])
set(quiver,'ZData',[0;0;0])
set(textx,'Position',[4 0 0])
set(texty,'Position',[0 4 0])
set(textz,'Position',[0 0 4])
saveas(gcf,'Images/ps1_model.png');
```

A.2 Problem Set 2 – Dynamics

```
%% Problem Set 2
clear; close all; clc;

%% Problem 1
a = 7125.48662; % km
e = 0.0011650;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree

yECI = oe2eci(a,e,i,O,w,nu);

days = 0.5;
tspan = 0:days*86400;
options = odeset('RelTol',1e-6,'AbsTol',1e-9);
[t,y] = ode113(@orbitSimple,tspan,yECI,options);

plot3(y(:,1),y(:,2),y(:,3),'LineWidth',2,'Color','green')
xlabel('x [km]')
ylabel('y [km]')
zlabel('z [km]')
axis equal
hold on
[xE,yE,zE] = ellipsoid(0,0,0,6378.1,6378.1,6378.1,20);
surface(xE,yE,zE,'FaceColor','blue','EdgeColor','black');
hold off
saveas(gcf,'Images/ps2_problem1.png');

%% Problem 2
cm = computeCM('res/mass.csv');
I = computeMOI('res/mass.csv',cm);

[rot,IPrincipal] = eig(I);
Ix = IPrincipal(1,1);
Iy = IPrincipal(2,2);
Iz = IPrincipal(3,3);
xPrincipal = rot(:,1);
yPrincipal = rot(:,2);
zPrincipal = rot(:,3);

%% Problem 3
figure
gm = importGeometry('res/NISAR.stl');
pdegplot(gm);

quiver = findobj(gca,'type','Quiver');
textx = findobj(gca,'type','Text','String','x');
texty = findobj(gca,'type','Text','String','y');
textz = findobj(gca,'type','Text','String','z');
set(quiver,"XData",[0;0;0])
set(quiver,"YData",[0;0;0])
set(quiver,"ZData",[0;0;0])
set(textx,"Position",[4 0 0])
set(texty,"Position",[0 4 0])
```

```
set(textz,"Position",[0 0 4])

quiverPrincipal = copyobj(quiver,gca);
textxPrincipal = copyobj(textx,gca);
textyPrincipal = copyobj(texty,gca);
textzPrincipal = copyobj(textz,gca);
set(quiver,"Color",[0 1 0])
set(quiver,"UData",4.14 * rot(1,:)')
set(quiver,"VData",4.14 * rot(2,:)')
set(quiver,"WData",4.14 * rot(3,:)')
set(quiver,"XData",repmat(cm(1),3,1))
set(quiver,"YData",repmat(cm(2),3,1))
set(quiver,"ZData",repmat(cm(3),3,1))
set(textx,"String",'x''')
set(texty,"String",'y''')
set(textz,"String",'z''')
set(textx,"Position",4 * xPrincipal + cm)
set(texty,"Position",4 * yPrincipal + cm)
set(textz,"Position",4 * zPrincipal + cm)
saveas(gcf,'Images/ps2_model.png');

%% Problem 5
w0Deg = [8;4;6];
w0 = deg2rad(w0Deg);
tspan = 0:120;
w = eulerPropagator(w0,Ix,Iy,Iz,tspan,'Images/ps2_euler_equations.png');

%% Problem 6
[XE,YE,ZE] = ellipsoidEnergy(IPrincipal, ...
    w0, ...
    'Images/ps2_problem6_energy.png');
[XM,YM,ZM] = ellipsoidMomentum(IPrincipal, ...
    w0, ...
    'Images/ps2_problem6_momentum.png');

%% Problem 7
w = polhode(XE,YE,ZE,XM,YM,ZM,w,'Images/ps2_problem7.png');

%% Problem 8
w = polhode2D(w,'none','Images/ps2_problem8.png');

%% Problem 9, x-axis
axis = 'x';
w0Deg = 8*[1;0;0];
w0 = deg2rad(w0Deg);
tspan = 0:120;
marker = 'o';

%% Problem 9, y-axis
axis = 'y';
w0Deg = 8*[0.01;1;0.01];
w0 = deg2rad(w0Deg);
tspan = 0:1200;
marker = 'none';

%% Problem 9, z-axis
axis = 'z';
w0Deg = 8*[0.01;0;1];
```

```
w0 = deg2rad(w0Deg);
tspan = 0:120;
marker = 'none';

%% Problem 9
w = eulerPropagator(w0,Ix,Iy,Iz,tspan, ...
    ['Images/ps2_problem9_euler_equations_', axis, '.png']);

[XE,YE,ZE] = ellipsoidEnergy(IPrincipal,w0, ...
    ['Images/ps2_problem9_energy_', axis, '.png']);
[XM,YM,ZM] = ellipsoidMomentum(IPrincipal,w0, ...
    ['Images/ps2_problem9_momentum_', axis, '.png']);

w = polhode(XE,YE,ZE,XM,YM,ZM,w, ...
    ['Images/ps2_problem9_p7_', axis, '.png']);

w = polhode2D(w,marker, ...
    ['Images/ps2_problem9_p8_', axis, '.png']);
```

```
function [t,y] = plotECI(a,e,i,O,w,nu,tspan)
    yECI = oe2eci(a,e,i,O,w,nu);
    options = odeset('RelTol',1e-6,'AbsTol',1e-9);
    [t,y] = ode113(@orbitSimple,tspan,yECI,options);
    plot3(y(:,1),y(:,2),y(:,3),'LineWidth',2,'Color','green')
    xlabel('x [km]')
    ylabel('y [km]')
    zlabel('z [km]')
    axis equal
    grid on
    hold on
    [xE,yE,zE] = ellipsoid(0,0,0,6378.1,6378.1,6378.1,20);
    surface(xE,yE,zE, ...
        'FaceColor','blue', ...
        'EdgeColor','black', ...
        'FaceAlpha',0.1);
    hold off
end
```

```
function w = eulerPropagator(w0,Ix,Iy,Iz,tspan,filename)
    options = odeset('RelTol',1e-6,'AbsTol',1e-9);
    [t,w] = ode113(@(t,w) eulerEquation(t,w,Ix,Iy,Iz),tspan,w0,options);
    wDeg = rad2deg(w);

    figure(1)
    plot(t,wDeg,'LineWidth',2)
    legend('\omega_{x}','\omega_{y}','\omega_{z}', ...
        'Location','southeast')
    xlabel('Time [s]')
    ylabel(['Angular velocity (\omega) [' char(176) '/s]'])
    saveas(1,filename)
end
```

```
function [XE,YE,ZE] = ellipsoidEnergy(IPrincipal,w0,filename)
```

```
    Ix = IPrincipal(1,1);
    Iy = IPrincipal(2,2);
    Iz = IPrincipal(3,3);
    T = sum(IPrincipal * w0.^2,"all") / 2;
    L = sqrt(sum((w0.*IPrincipal).^2,"all"));
    [XE,YE,ZE] = ...
        ellipsoid(0,0,0,sqrt(2*T/Ix),sqrt(2*T/Iy),sqrt(2*T/Iz),50);
    ellipsoidAxes = [sqrt(2*T/Ix), sqrt(2*T/Iy), sqrt(2*T/Iz)];

    % Plot energy ellipsoid
    figure(1)
    surf(XE,YE,ZE, ...
        'FaceAlpha',0.5, ...
        'FaceColor','blue', ...
        'DisplayName','Energy Ellipsoid');
    axis equal
    hold on
    quiver3(0, 0, 0, ellipsoidAxes(1), 0, 0, 'Color', 'r', ...
        'LineWidth', 2)
    quiver3(0, 0, 0, 0, ellipsoidAxes(2), 0, 'Color', 'r', ...
        'LineWidth', 2)
    quiver3(0, 0, 0, 0, 0, ellipsoidAxes(3), 'Color', 'r', ...
        'LineWidth', 2)
    xlabel('\omega_{x} [rad/s]')
    ylabel('\omega_{y} [rad/s]')
    zlabel('\omega_{z} [rad/s]')
    hold off
    saveas(1,filename)

    I = L^2/(2*T);
    if (Ix <= I || ismembertol(Ix, I, 1e-7)) && I <= Iz
        fprintf("The polhode is real!\n")
    else
        error("The polhode is NOT real!\n")
    end
end
```

```
function [XM,YM,ZM] = ellipsoidMomentum(IPrincipal,w0,filename)
    Ix = IPrincipal(1,1);
    Iy = IPrincipal(2,2);
    Iz = IPrincipal(3,3);
    T = sum(IPrincipal * w0.^2,"all") / 2;
    L = sqrt(sum((w0.*IPrincipal).^2,"all"));
    [XM,YM,ZM] = ellipsoid(0,0,0,L/Ix,L/Iy,L/Iz,50);
    momentumAxes = [L/Ix, L/Iy, L/Iz];

    % Plot momentum ellipsoid
    figure(1)
    surf(XM,YM,ZM, ...
        'FaceAlpha',0.5, ...
        'FaceColor','green', ...
        'DisplayName','Momentum Ellipsoid');
    axis equal
    hold on
    quiver3(0, 0, 0, momentumAxes(1), 0, 0, 'Color', 'r', ...
        'LineWidth', 2)
```

```
    quiver3(0, 0, 0, 0, momentumAxes(2), 0, 'Color', 'r', ...
        'LineWidth', 2)
    quiver3(0, 0, 0, 0, 0, momentumAxes(3), 'Color', 'r', ...
        'LineWidth', 2)
    xlabel('\omega_{x} [rad/s]')
    ylabel('\omega_{y} [rad/s]')
    zlabel('\omega_{z} [rad/s]')
    hold off
    saveas(1,filename)

    I = L^2/(2*T);
    if (Ix <= I || ismembertol(Ix, I, 1e-7)) && I <= Iz
        fprintf("The polhode is real!\n")
    else
        error("The polhode is NOT real!\n")
    end
end
```

```
function w = polhode(XE,YE,ZE,XM,YM,ZM,w,filename)
    figure(1)
    surf(XE,YE,ZE, ...
        'FaceAlpha',0.5, ...
        'FaceColor','blue', ...
        'DisplayName','Energy Ellipsoid');
    xlabel('\omega_{x} [rad/s]')
    ylabel('\omega_{y} [rad/s]')
    zlabel('\omega_{z} [rad/s]')
    axis equal
    hold on
    surf(XM,YM,ZM, ...
        'FaceAlpha',0.5, ...
        'FaceColor','green', ...
        'DisplayName','Momentum Ellipsoid');
    plot3(w(:,1),w(:,2),w(:,3), ...
        'LineWidth',2, ...
        'Color','red', ...
        'DisplayName','Polhode')
    legend('Location','northwest')
    hold off
    saveas(1,filename)
end
```

```
function w = polhode2D(w,marker,filename)
    subplot(1,3,1)
    plot(w(:,2),w(:,3),'Marker',marker)
    title('Polhode (along x-axis)')
    xlabel('\omega_{y} [rad/s]')
    ylabel('\omega_{z} [rad/s]')
    axis equal

    subplot(1,3,2)
    plot(w(:,1),w(:,3),'Marker',marker)
    title('Polhode (along y-axis)')
    xlabel('\omega_{x} [rad/s]')
    ylabel('\omega_{z} [rad/s]')
```

```
14      axis equal
15
16      subplot(1,3,3)
17      plot(w(:,1),w(:,2),'Marker',marker)
18      title('Polhode (along z-axis)')
19      xlabel('\omega_{x} [rad/s]')
20      ylabel('\omega_{y} [rad/s]')
21      axis equal
22
23      saveas(1,filename)
24  end
```

A.3 Problem Set 3 – Axial-Symmetric and Kinematics

```
clear; close all; clc;

%% Problem 1
IPrincipal = [7707.07451493673 0 0; ...
    0 7707.0745149367 0; ...
    0 0 18050.0227594212];
Ix = IPrincipal(1,1);
Iy = IPrincipal(2,2);
Iz = IPrincipal(3,3);

w0Deg = [8;4;6];
w0 = deg2rad(w0Deg);
tspan = 0:0.1:120;
w = eulerPropagator(w0,Ix,Iy,Iz,tspan,'Images/ps3_problem1.png');

%% Problem 2
lambda = w0(3) * (Iz - Iy) / Ix;
wxy = (w0(1) + w0(2) * 1j) * exp(1j * lambda * tspan);
wx = real(wxy);
wy = imag(wxy);
wz = w0(3) * ones(size(wxy));
wAnalytical = [wx',wy',wz'];
wDegAnalytical = rad2deg(wAnalytical);

figure(1)
plot(tspan,wDegAnalytical,'LineWidth',2)
legend('\omega_{x}','\omega_{y}','\omega_{z}', ...
    'Location','southeast')
xlabel('Time [s]')
ylabel(['Angular velocity (\omega) [' char(176) '/s]'])
saveas(1,'Images/ps3_problem2.png')

%% Problem 3
% Error plots
error = w - wAnalytical;
plot(tspan,error,'LineWidth',2)
legend('\omega_{x}','\omega_{y}','\omega_{z}', ...
    'Location','southeast')
xlabel('Time [s]')
ylabel('Angular velocity (\omega) [rad/s]')
saveas(gcf,'Images/ps3_problem3.png')

% Verify L and omega
L_principal = [Ix Iy Iz] .* w;
keyTimes = [1, 61, 121, 181, 241, 361];
for n = keyTimes
    figure(1)
    L_unit = L_principal(n,:)/norm(L_principal(n,:));
    w_unit = w(n,:)/norm(w(n,:));
    quiver3(0,0,0,w_unit(1),w_unit(2),w_unit(3),1,'r')
    hold on
    quiver3(0,0,0,L_unit(1),L_unit(2),L_unit(3),1,'b')
    quiver3(0,0,0,0,0,1,'k')
    xlim([-1 1]); ylim([-1 1]); zlim([-1 1]);
    xlabel('x'); ylabel('y'); zlabel('z');
```

```
56      legend('\omega','L','z-axis','Location','northeast')
57      title(sprintf('Unit vectors at t = %.2f s', tspan(n)))
58      hold off
59      saveas(1, sprintf('Images/ps3_problem3_vectors_%i.png', n))
60  end
61
62  %% Non-Axisymmetric Satellite
63  cm = computeCM('res/mass.csv');
64  I = computeMOI('res/mass.csv',cm);
65
66  [rot,IPrincipal] = eig(I);
67  Ix = IPrincipal(1,1);
68  Iy = IPrincipal(2,2);
69  Iz = IPrincipal(3,3);
70  xPrincipal = rot(:,1);
71  yPrincipal = rot(:,2);
72  zPrincipal = rot(:,3);
73
74  %% Problem 6 (Quaternions)
75  axang0 = [sqrt(1/2) sqrt(1/2) 0 pi/4];
76  q0 = axang2quat(axang0).';
77  tFinal = 600;
78  tStep = 0.1;
79  t = 0:tStep:tFinal;
80
81  % Forward Euler
82  % [q,w] = kinQuaternionForwardEuler(q0,w0,Ix,Iy,Iz,tFinal,tStep);
83
84  % RK4
85  [q,w] = kinQuaternionRK4(q0,w0,Ix,Iy,Iz,tFinal,tStep);
86
87  figure(2)
88  hold on
89  plot(t,q,'LineWidth',1)
90  legend('q_{1}','q_{2}','q_{3}','q_{4}', ...
91      'Location','Southeast')
92  xlabel('Time [s]')
93  ylabel('Quaternion')
94  hold off
95  saveas(2,'Images/ps3_problem6_quaternions.png')
96
97  %% Problem 6 (Euler Angles)
98  eulerAngle0 = rotm2eul(axang2rotm(axang0))';
99  state0 = [eulerAngle0;w0];
100
101 tFinal = 600;
102 tStep = 0.1;
103 t = 0:tStep:tFinal;
104
105 % Forward Euler
106 % state = kinEulerAngleForwardEuler(state0,Ix,Iy,Iz,tFinal,tStep);
107
108 % ode113
109 tspan = 0:tStep:tFinal;
110 options = odeset('RelTol',1e-6,'AbsTol',1e-9);
111 [t,state] = ode113(@(t,state) kinEulerAngle(t,state,Ix,Iy,Iz), ...
112     tspan,state0,options);
113
```

```
eulerAngle = wrapTo360(rad2deg(state(:,1:3)));

figure(3)
plot(t,eulerAngle,'LineWidth',1)
legend('\phi','\theta','\psi', ...
    'Location','southwest')
xlabel('Time [s]')
ylabel('Euler Angle [deg]')
saveas(3,'Images/ps3_problem6_euler.png')

%% Problem 7(a)
% Part a: Angular momentum
tLen = length(t);
L_principal = [Ix Iy Iz] .* w;
L_inertial = nan(size(L_principal));
L_norm = nan(1, tLen);

% Part b: Herpolhode
w_inertial = nan(size(w));

for i = 1:tLen
    % Get rotation matrix
    qi = q(i,:);
    A = q2A(qi);

    % Angular momentum
    L_inertial(i,:) = A' * L_principal(i,:)';
    L_norm(i) = norm(L_inertial(i,:));

    % Angular velocity
    w_inertial(i,:) = A' * w(i,:)';
end

figure(4)
hold on
plot(t, L_inertial)
plot(t, L_norm, 'k--')
xlabel('Time [s]')
ylabel('Angular momentum [kg m^{2}/s]')
legend("L_{1}", "L_{2}", "L_{3}", "||L||")
hold off
saveas(4, 'Images/ps3_problem7a.png')

%% Problem 7(b)
figure(5)
plot3(w_inertial(:,1), w_inertial(:,2), w_inertial(:,3), 'r')
grid on
hold on
quiver3(0, 0, 0, ...
    L_inertial(1,1), L_inertial(1,2), L_inertial(1,3), ...
    1e-4)
quiver3(0, 0, 0, ...
    w_inertial(1,1), w_inertial(1,2), w_inertial(1,3), ...
    1)
xlabel('\omega_{x} [rad/s]')
ylabel('\omega_{y} [rad/s]')
zlabel('\omega_{z} [rad/s]')
legend('Herpolhode', ...
```

```
    'Angular momentum (L)', ...
    'Angular velocity (\omega)', ...
    'Location','northwest')
hold off
saveas(5, 'Images/ps3_problem7b.png')

%% For fun kinda thing
saveGif = true;
tGif = 240 / tStep;

L_unit = nan(size(L_inertial));
w_unit = nan(size(w_inertial));
if saveGif == true
    gif = figure;
    for i = 1:tLen
        w_unit(i,:) = w_inertial(i,:)./norm(w_inertial(i,:));
        L_unit(i,:) = L_inertial(i,:)./norm(L_inertial(i,:));
    end

    for i = 1:20:tGif
        plot3(w_unit(1:i,1), w_unit(1:i,2), w_unit(1:i,3), 'r')
        grid on
        hold on
        quiver3(0, 0, 0, w_unit(i,1), w_unit(i,2), w_unit(i,3),1)
        quiver3(0, 0, 0, L_unit(i,1), L_unit(i,2), L_unit(i,3),1)
        hold off
        xlim([-1 1])
        ylim([-1 1])
        zlim([-1 1])
        xlabel('x')
        ylabel('y')
        zlabel('z')
        title('Note: all vectors are normalized')
        legend('Herpolhode','\omega','L','Location','northeast')
        exportgraphics(gif,'Images/ps3_problem7b.gif','Append',true);
    end
end

%% Problem 7(c)
% Generate orbit
a = 7125.48662; % km
e = 0.0011650;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree

days = 0.069;
tFinal = days * 86400;
tStep = 1;
tspan = 0:tStep:tFinal;

figure(6)
[t,y] = plotECI(a,e,i,O,w,nu,tspan);
hold on
figure(7)
plotECI(a,e,i,O,w,nu,tspan);
hold on
```

```
figure(8)
plotECI(a,e,i,O,w,nu,tspan);
hold on

[q,w] = kinQuaternionRK4(q0,w0,Ix,Iy,Iz,tFinal,tStep);

tLen = length(t);
for i = 1:500:tLen
    % Get rotation matrix
    qi = q(i,:);
    A = q2A(qi);
    % Body axes
    B = rot * A * rot';
    % Position
    pos = y(i,1:3);
    radial = pos / norm(pos);
    tangential = y(i,4:6) / norm(y(i,4:6));
    normal = cross(radial,tangential);
    RTN = [radial' tangential' normal'];
    figure(6);
    plotTriad(gca,pos,A,1e3,'r');
    figure(7);
    plotTriad(gca,pos,B,1e3,'m');
    figure(8);
    plotTriad(gca,pos,RTN,1e3,'b');
end
figure(6);
legend('Orbit','Earth','Principal (x-axis)','Location','northwest')
hold off
saveas(gcf,'Images/ps3_problem7c_principal.png');
figure(7);
legend('Orbit','Earth','Body (x-axis)','Location','northwest')
hold off
saveas(gcf,'Images/ps3_problem7c_body.png');
figure(8);
legend('Orbit','Earth','RTN (radial axis)','Location','northwest')
hold off
saveas(gcf,'Images/ps3_problem7c_rtn.png');
```

```
function M = plotTriad(ax,o,M,scale,colorString)
    quiver3(ax, ...
        o(1),o(2),o(3), ...
        M(1,1),M(2,1),M(3,1), ...
        scale, ...
        'LineWidth',1, ...
        'Color','k')
    quiver3(ax, ...
        o(1),o(2),o(3), ...
        M(1,2),M(2,2),M(3,2), ...
        scale, ...
        'LineWidth',1, ...
        'Color',colorString)
    quiver3(ax, ...
        o(1),o(2),o(3), ...
        M(1,3),M(2,3),M(3,3), ...
        scale, ...
```

```
18          'LineWidth',1, ...
19          'Color',colorString)
20  end
```

A.4 Problem Set 4 – Equilibrium and Gravity Gradient

```
close all; clear; clc;
savePlot = false;

%% Import mass properties
cm = computeCM('res/mass.csv');
I = computeMOI('res/mass.csv',cm);

[rot,IPrincipal] = eig(I);
Ix = IPrincipal(1,1);
Iy = IPrincipal(2,2);
Iz = IPrincipal(3,3);

%% Problem 1(a)
tFinal = 60;
tStep = 0.01;
tspan = 0:tStep:tFinal;

eulerAngle0 = [0; 0; 0];
w0 = [0; 0; 1];
state0 = [eulerAngle0;w0];

options = odeset('RelTol',1e-6,'AbsTol',1e-9);
[t,state] = ode113(@(t,state) kinEulerAngle(t,state,Ix,Iy,Iz), ...
    tspan,state0,options);

state = rad2deg(state);

% Plot
figure()
plot(t,state(:,4:6),'LineWidth',1)
legend('\omega_{x}','\omega_{y}','\omega_{z}', ...
    'Location','southeast')
xlabel('Time [s]')
ylabel(['Angular Velocity (\omega) [' char(176) '/s]'])
if savePlot
    saveas(gcf,'Images/ps4_problem1a_angvel.png')
end

figure()
plot(t,wrapTo180(state(:,1:3)),'LineWidth',1)
legend('\phi','\theta','\psi', ...
    'Location','southwest')
xlabel('Time [s]')
ylabel(['Euler Angle [' char(176) ']'])
if savePlot
    saveas(gcf,'Images/ps4_problem1a_angle.png')
end

%% Problem 1(b)
a = 7125.48662; % km
e = 0;
i = 98.40508; % degree
O = -19.61601; % degree
w_deg = 89.99764; % degree
nu = -89.99818; % degree
```

```
56
57 muE = 3.986 * 10^5;
58
59 n = sqrt(muE / a^3);
60
61 tFinal = 6000;
62 tStep = 0.1;
63 tspan = 0:tStep:tFinal;
64 tTrunc = 300;
65 nTrunc = find((tspan == tTrunc) == 1);
66
67 [~,y] = plotECI(a,e,i,O,w_deg,nu,tspan);
68 close all
69 format long
70
71 % Initialize angular velocity aligned with normal
72 r0 = y(1,1:3);
73 v0 = y(1,4:6);
74 h = cross(r0,v0);
75 radial = r0 / norm(r0);
76 normal = h / norm(h);
77 tangential = cross(normal,radial);
78 A_RTN = [radial' tangential' normal']';
79 w0_RTN = [0; 0; 0.1];
80 euler0_RTN = A2e(A_RTN);
81 state0 = [euler0_RTN; w0_RTN];
82 options = odeset('RelTol',1e-6,'AbsTol',1e-9);
83 [t,state] = ode113(@(t,state) kinEulerAngle(t,state,Ix,Iy,Iz), ...
84     tspan,state0,options);
85
86 w_RTN = nan(size(state(:,1:3)));
87 euler_RTN = nan(size(state(:,1:3)));
88
89 for n = 1:length(t)
90     pos = y(n,1:3);
91     vel = y(n,4:6);
92     h = cross(pos,vel);
93     radial = pos / norm(pos);
94     normal = h / norm(h);
95     tangential = cross(normal,radial);
96     tangential = tangential / norm(tangential);
97     A_RTN = [radial' tangential' normal']';
98
99     % Get rotation matrixes (to ECI)
100    euler = state(n,1:3);
101    w_principal = state(n,4:6)';
102    A_principal = e2A(euler);
103    A_P2R = A_RTN * A_principal';
104    w_RTN(n,:) = A_P2R*w_principal;
105    euler_RTN(n,:) = A2e(A_P2R');
106 end
107
108 figure()
109 plot(t, rad2deg(w_RTN))
110 xlabel('Time [s]')
111 ylabel(['Angular Velocity, RTN Frame [' char(176) '/s]'])
112 legend('\omega_{R}','\omega_{T}','\omega_{N}')
113 if savePlot == true
```

```
    saveas(gcf,'Images/ps4_problem1b_angvel.png')
end

figure()
plot(t(1:nTrunc), wrapTo180(rad2deg(euler_RTN(1:nTrunc,:))))
xlabel('Time [s]')
ylabel(['Euler Angle, RTN Frame [' char(176) ']'])
legend('\phi','\theta','\psi')
if savePlot == true
    saveas(gcf,'Images/ps4_problem1b_euler.png')
end

%% Problem 2
% Initial conditions
perturbation = 0.001;
w0x = [1; perturbation; perturbation];
w0y = [perturbation; 1; perturbation];
w0z = [perturbation; perturbation; 1];

w0Mat = {w0x, w0y, w0z};
eulerAngle0 = [0; 0; 0];
tStep = 0.01;
tFinal = 60;

for n = 1:3
    w0 = w0Mat{n};
    state0 = [eulerAngle0;w0];

    tspan = 0:tStep:tFinal;
    options = odeset('RelTol',1e-6,'AbsTol',1e-9);
    [t,state] = ode113(@(t,state) kinEulerAngle(t,state,Ix,Iy,Iz), ...
        tspan,state0,options);

    eulerAngle = wrapTo180(rad2deg(state(:,1:3)));
    w = rad2deg(state(:,4:6));

    figure()
    subplot(2,1,1)
    plot(t,w)
    xlabel('Time [s]')
    ylabel(['Angular Velocity [' char(176) '/s]'])
    legend('\omega_{x}','\omega_{y}','\omega_{z}', ...
        'Location','Southeast')

    subplot(2,1,2)
    plot(t,eulerAngle)
    xlabel('Time [s]')
    ylabel(['Euler Angle [' char(176) ']'])
    legend('\phi','\theta','\psi', 'Location','Southeast')
    if savePlot
        saveas(gcf,['Images/ps4_problem2a_' sprintf('%i',n) '.png'])
    end
end

%% Momentum wheel setup
% Based on RSI 68
mr = 8.9; % kg
r = 0.347 / 2; % m
```

```
Ir = mr * r^2;
wrRPM = 2500; % RPM
wr = wrRPM * 0.1047198;

% Initial Euler angle
eulerAngle0 = [0; 0; 0];

% No external torques
M = [0; 0; 0; 0];

% Time
tFinal = 300;
tStep = 0.1;

%% Problem 3(b)
w0 = [0.01; 0.01; 0.01; wr];
r = [0; 0; 1];

namePlot = 'Images/ps4_problem3b.png';
plotPS4Problem3(eulerAngle0,w0,tStep,tFinal, ...
                M,r,Ix,Iy,Iz,Ir,namePlot,savePlot);

%% Problem 3(c)
w0 = [0.1; 0.001; 0.001; wr + 0.001 * rand()];
r = [1; 0; 0];
namePlot = 'Images/ps4_problem3c_x.png';
plotPS4Problem3(eulerAngle0,w0,tStep,tFinal, ...
                M,r,Ix,Iy,Iz,Ir,namePlot,savePlot);

w0 = [0.001; 0.1; 0.001; wr + 0.001 * rand()];
r = [0; 1; 0];
namePlot = 'Images/ps4_problem3c_y.png';
plotPS4Problem3(eulerAngle0,w0,tStep,tFinal, ...
                M,r,Ix,Iy,Iz,Ir,namePlot,savePlot);

w0 = [0.001; 0.001; 0.1; wr + 0.001 * rand()];
r = [0; 0; 1];
namePlot = 'Images/ps4_problem3c_z.png';
plotPS4Problem3(eulerAngle0,w0,tStep,tFinal, ...
                M,r,Ix,Iy,Iz,Ir,namePlot,savePlot);

%% Problem 3(d)
w0 = [0.001; 0.1; 0.001; wr * 10];
r = [0; 1; 0];

namePlot = 'Images/ps4_problem3d.png';
plotPS4Problem3(eulerAngle0,w0,tStep,tFinal, ...
                M,r,Ix,Iy,Iz,Ir,namePlot,savePlot);

%% Problem 3(e)
w0 = [rot' * [0.1; 0.001; 0.001]; 0];
r = rot' * [1; 0; 0];

namePlot = 'Images/ps4_problem3e_unstable.png';
plotPS4Problem3(eulerAngle0,w0,tStep,tFinal, ...
                M,r,Ix,Iy,Iz,Ir,namePlot,savePlot);

w0 = [rot' * [0.1; 0.001; 0.001]; wr * 10];
```

```
230
231 namePlot = 'Images/ps4_problem3e_stable.png';
232 plotPS4Problem3(eulerAngle0,w0,tStep,tFinal, ...
233                 M,r,Ix,Iy,Iz,Ir,namePlot,savePlot);
234
235 %% Problem 4(d)
236 % Should put this into a function and call it for (d-e)
237 tFinal = 6000;
238 tStep = 1;
239 tspan = 0:tStep:tFinal;
240
241 a = 7125.48662; % km
242 e = 0;
243 i = 98.40508; % degree
244 O = -19.61601; % degree
245 w = 89.99764; % degree
246 nu = -89.99818; % degree
247 muE = 3.986 * 10^5;
248 n = sqrt(muE / a^3);
249
250 y = oe2eci(a,e,i,O,w,nu);
251 r0 = y(1:3);
252 v0 = y(4:6);
253 h = cross(r0,v0);
254 radial = r0 / norm(r0);
255 normal = h / norm(h);
256 tangential = cross(normal,radial);
257 A_RTN = [radial tangential normal]';
258
259 state0 = zeros(12,1);
260 state0(1:6) = y;
261 state0(7:9) = [0; 0; n];
262 state0(10:12) = A2e(A_RTN);
263
264 options = odeset('RelTol',1e-6,'AbsTol',1e-9);
265 [t,state] = ode113(@(t,state) gravGrad(t,state,Ix,Iy,Iz,n), ...
266         tspan,state0,options);
267
268 c = zeros(size(state(:,1:3)));
269 M = zeros(size(state(:,1:3)));
270 for i = 1:length(t)
271     r = state(i,1:3);
272     radial = r / norm(r);
273     A_ECI2P = e2A(state(i,10:12));
274     c(i,1:3) = A_ECI2P * radial';
275     M(i,1:3) = gravGradTorque(Ix,Iy,Iz,n,c(i,1:3));
276 end
277
278 figure()
279 plot(t,M)
280 xlabel('Time [s]')
281 ylabel('Torque in Principal Axes [Nm]')
282 legend('M_{x}','M_{y}','M_{z}')
283 ylim([-1e-5 1e-5])
284 if savePlot == true
285     saveas(gcf,'Images/ps4_problem4d_torque.png')
286 end
287
```

```
figure()
plot(t,state(:,7:9))
xlabel('Time [s]')
ylabel('Angular Velocity in Principal Axes [rad/s]')
legend('\omega_{x}','\omega_{y}','\omega_{z}')
if savePlot == true
    saveas(gcf,'Images/ps4_problem4d_angvel.png')
end

%% Problem 4(e)
tFinal = 6000;
tStep = 1;
tspan = 0:tStep:tFinal;

a = 7125.48662; % km
e = 0;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree
muE = 3.986 * 10^5;
n = sqrt(muE / a^3);

y = oe2eci(a,e,i,O,w,nu);
r0 = y(1:3);
v0 = y(4:6);
h = cross(r0,v0);
radial = r0 / norm(r0);
normal = h / norm(h);
tangential = cross(normal,radial);
A_RTN = [radial tangential normal]';
A_Body = rot' * A_RTN;

state0 = zeros(12,1);
state0(1:6) = y;
state0(7:9) = [0; 0; n];
state0(10:12) = A2e(A_Body);

options = odeset('RelTol',1e-6,'AbsTol',1e-9);
[t,state] = ode113(@(t,state) gravGrad(t,state,Ix,Iy,Iz,n), ...
        tspan,state0,options);

c = zeros(size(state(:,1:3)));
M = zeros(size(state(:,1:3)));
for i = 1:length(t)
    r = state(i,1:3);
    radial = r / norm(r);
    A_ECI2P = e2A(state(i,10:12));
    c(i,1:3) = A_ECI2P * radial';
    M(i,1:3) = gravGradTorque(Ix,Iy,Iz,n,c(i,1:3));
end

figure()
plot(t,M)
xlabel('Time [s]')
ylabel('Torque in Principal Axes [Nm]')
legend('M_{x}','M_{y}','M_{z}')
if savePlot == true
```

```
346     saveas(gcf,'Images/ps4_problem4e_torque.png')
347 end
348
349 figure()
350 plot(t,state(:,7:9))
351 xlabel('Time [s]')
352 ylabel('Angular Velocity in Principal Axes [rad/s]')
353 legend('\omega_{x}','\omega_{y}','\omega_{z}')
354 if savePlot == true
355     saveas(gcf,'Images/ps4_problem4e_angvel.png')
356 end
357
358 figure()
359 plot(t,wrapTo180(rad2deg(state(:,10:12))))
360 xlabel('Time [s]')
361 ylabel(['Euler Angles [' char(176) ']'])
362 legend('\phi','\theta','\psi')
363 if savePlot == true
364     saveas(gcf,'Images/ps4_problem4e_angle.png')
365 end
```

```
1  function stateDot = kinEulerAngleWheel(t,state,M,r,Ix,Iy,Iz,Ir)
2      % Computes state derivative for Euler angles, angular velocity
3      % Adds momentum wheel
4      % Assign variables
5      phi = state(1);
6      theta = state(2);
7      w = state(4:7);
8
9      stateDot = zeros(7,1);
10     % Angular velocity time derivatives
11     wDot = eulerEquationWheel(t,w,M,r,Ix,Iy,Iz,Ir);
12     stateDot = zeros(7,1);
13     stateDot(4) = wDot(1);
14     stateDot(5) = wDot(2);
15     stateDot(6) = wDot(3);
16     stateDot(7) = wDot(4);
17     % Euler angle time derivatives
18     % 312
19     EPrimeInv = [sin(phi)*sin(theta) cos(phi)*sin(theta) cos(theta); ...
20         cos(theta)*cos(phi) -sin(phi)*cos(theta) 0; ...
21         sin(phi) cos(phi) 0] * (1 / cos(theta));
22     % 313
23     % EPrimeInv = [-sin(phi)*cos(theta) -cos(phi)*cos(theta) ...
           sin(theta); ...
24     %     cos(phi)*sin(theta) -sin(phi)*sin(theta) 0; ...
25     %     sin(phi) cos(phi) 0] * (1 / sin(theta));
26     stateDot(1:3) = EPrimeInv * w(1:3);
27 end
```

A.5 Problem Set 5 – Gravity Gradient Stability and Perturbations

```
close all; clear; clc;
savePlot = true;

%% Import mass properties
cm = computeCM('res/mass.csv');
I = computeMOI('res/mass.csv',cm);

[rot,IPrincipal] = eig(I);
Ix = IPrincipal(1,1);
Iy = IPrincipal(2,2);
Iz = IPrincipal(3,3);

%% Problem 1(a)
IR = Ix;
IT = Iz;
IN = Iy;

kT = (IN - IR) / IT;
kR = (IN - IT) / IR;

plotGravGradStability(kR,kT,'Nominal','Images/ps5_problem1a.png');

%% Problem 1(b) (Unstable, Unperturbed)
tFinal = 6000 * 5; % 5 orbits
tStep = 1;
tspan = 0:tStep:tFinal;

a = 7125.48662; % km
e = 0;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree
muE = 3.986 * 10^5;
n = sqrt(muE / a^3);

y = oe2eci(a,e,i,O,w,nu);
r0 = y(1:3);
v0 = y(4:6);
h = cross(r0,v0);
radial = r0 / norm(r0);
normal = h / norm(h);
tangential = cross(normal,radial);
A_Nominal = [-radial -normal -tangential]';

state0 = zeros(12,1);
state0(1:6) = y;
state0(7:9) = [0; -n; 0];
state0(10:12) = A2e(A_Nominal);

options = odeset('RelTol',1e-6,'AbsTol',1e-9);
[t,state] = ode113(@(t,state) gravGrad(t,state,Ix,Iy,Iz,n), ...
        tspan,state0,options);

c = zeros(size(state(:,1:3)));
```

```
M = zeros(size(state(:,1:3)));
for i = 1:length(t)
    r = state(i,1:3);
    radial = r / norm(r);
    A_ECI2P = e2A(state(i,10:12));
    c(i,1:3) = A_ECI2P * radial';
    M(i,1:3) = gravGradTorque(Ix,Iy,Iz,n,c(i,1:3));
end

figure()
plot(t / 3600,state(:,7:9))
xlabel('Time [h]')
ylabel('Angular Velocity in Principal Axes [rad/s]')
legend('\omega_{x}','\omega_{y}','\omega_{z}')
if savePlot == true
    saveas(gcf,'Images/ps5_problem1b_angvel_unperturbed.png')
end

figure()
plot(t / 3600,wrapToPi(state(:,10:12)))
xlabel('Time [h]')
ylabel('Euler Angles in Principal Axes [rad]')
legend('\phi','\theta','\psi')
if savePlot == true
    saveas(gcf,'Images/ps5_problem1b_angle_unperturbed.png')
end

%% Problem 1(b) (Unstable, Perturbed)
tFinal = 6000 * 3; % 2 orbits
tStep = 1;
tspan = 0:tStep:tFinal;

a = 7125.48662; % km
e = 0;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree
muE = 3.986 * 10^5;
n = sqrt(muE / a^3);

y = oe2eci(a,e,i,O,w,nu);
r0 = y(1:3);
v0 = y(4:6);
h = cross(r0,v0);
radial = r0 / norm(r0);
normal = h / norm(h);
tangential = cross(normal,radial);
A_Nominal = [-radial -normal -tangential]';

state0 = zeros(12,1);
state0(1:6) = y;
state0(7:9) = [0; -n; 0] * 1.01;
state0(10:12) = A2e(A_Nominal) + pi * [0.01; 0.01; 0.01];

options = odeset('RelTol',1e-6,'AbsTol',1e-9);
[t,state] = ode113(@(t,state) gravGrad(t,state,Ix,Iy,Iz,n), ...
        tspan,state0,options);
```

```
114
115 c = zeros(size(state(:,1:3)));
116 M = zeros(size(state(:,1:3)));
117 eulerRTN = zeros(size(state(:,1:3)));
118 for i = 1:length(t)
119     r = state(i,1:3);
120     v = state(i,4:6);
121     radial = r / norm(r);
122     h = cross(r,v);
123     normal = h / norm(h);
124     tangential = cross(normal,radial);
125     A_ECI2P = e2A(state(i,10:12));
126     c(i,1:3) = A_ECI2P * radial';
127     M(i,1:3) = gravGradTorque(Ix,Iy,Iz,n,c(i,1:3));
128     eulerRTN(i,1:3) = A2e(A_ECI2P * [-radial' -normal' -tangential']);
129 end
130
131 figure()
132 plot(t / 3600,state(:,7:9))
133 xlabel('Time [h]')
134 ylabel('Angular Velocity in Principal Axes [rad/s]')
135 legend('\omega_{x}','\omega_{y}','\omega_{z}')
136 if savePlot == true
137     saveas(gcf,'Images/ps5_problem1b_angvel.png')
138 end
139
140 figure()
141 plot(t / 3600,wrapToPi(state(:,10:12)))
142 xlabel('Time [h]')
143 ylabel('Euler Angles in Principal Axes [rad]')
144 legend('\phi','\theta','\psi')
145 if savePlot == true
146     saveas(gcf,'Images/ps5_problem1b_angle.png')
147 end
148
149 figure()
150 plot(t / 3600,wrapToPi(eulerRTN))
151 xlabel('Time [h]')
152 ylabel('Euler Angles in RTN Axes [rad]')
153 legend('\phi','\theta','\psi')
154 if savePlot == true
155     saveas(gcf,'Images/ps5_problem1b_angle_rtn.png')
156 end
157
158 %% Problem 1(c)
159 IR = Ix;
160 IT = Iy;
161 IN = Iz;
162 kT = (IN - IR) / IT;
163 kR = (IN - IT) / IR;
164
165 plotGravGradStability(kR,kT, ...
166     'Principal XYZ aligned with RTN', ...
167     'Images/ps5_problem1c.png');
168
169 %% Problem 1(c) (Stable, Perturbed)
170 tFinal = 6000 * 10; % 10 orbits
171 tStep = 1;
```

```
tspan = 0:tStep:tFinal;

a = 7125.48662; % km
e = 0;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree
muE = 3.986 * 10^5;
n = sqrt(muE / a^3);

y = oe2eci(a,e,i,O,w,nu);
r0 = y(1:3);
v0 = y(4:6);
h = cross(r0,v0);
radial = r0 / norm(r0);
normal = h / norm(h);
tangential = cross(normal,radial);
A_RTN = [radial tangential normal]';

state0 = zeros(12,1);
state0(1:6) = y;
state0(7:9) = [0; 0; n] * 1.01;
state0(10:12) = A2e(A_RTN) + pi * [0.01; 0.01; 0.01];

options = odeset('RelTol',1e-6,'AbsTol',1e-9);
[t,state] = ode113(@(t,state) gravGrad(t,state,Ix,Iy,Iz,n), ...
        tspan,state0,options);

c = zeros(size(state(:,1:3)));
M = zeros(size(state(:,1:3)));
eulerRTN = zeros(size(state(:,1:3)));
for i = 1:length(t)
    r = state(i,1:3);
    v = state(i,4:6);
    radial = r / norm(r);
    h = cross(r,v);
    normal = h / norm(h);
    tangential = cross(normal,radial);
    A_ECI2P = e2A(state(i,10:12));
    c(i,1:3) = A_ECI2P * radial';
    M(i,1:3) = gravGradTorque(Ix,Iy,Iz,n,c(i,1:3));
    eulerRTN(i,1:3) = A2e(A_ECI2P * [radial' tangential' normal']);
end

figure()
plot(t / 3600,state(:,7:9))
xlabel('Time [h]')
ylabel('Angular Velocity in Principal Axes [rad/s]')
legend('\omega_{x}','\omega_{y}','\omega_{z}')
if savePlot == true
    saveas(gcf,'Images/ps5_problem1c_angvel.png')
end

figure()
plot(t / 3600,wrapToPi(state(:,10:12)))
xlabel('Time [h]')
ylabel('Euler Angles in Principal Axes [rad]')
```

```
230 legend('\phi','\theta','\psi')
231 if savePlot == true
232     saveas(gcf,'Images/ps5_problem1c_angle.png')
233 end
234
235 figure()
236 plot(t / 3600,wrapToPi(eulerRTN))
237 xlabel('Time [h]')
238 ylabel('Euler Angles in RTN Axes [rad]')
239 legend('\phi','\theta','\psi')
240 if savePlot == true
241     saveas(gcf,'Images/ps5_problem1c_angle_rtn.png')
242 end
243
244 %% Problem 3
245 tFinal = 6000;
246 tStep = 1;
247 tspan = 0:tStep:tFinal;
248
249 % Satellite orbit initial conditions
250 a = 7125.48662; % km
251 e = 0;
252 i = 98.40508; % degree
253 O = -19.61601; % degree
254 w = 89.99764; % degree
255 nu = -89.99818; % degree
256 muE = 3.986 * 10^5; % km^3 / s^2
257 n = sqrt(muE / a^3);
258
259 % Compute initial position and attitude
260 y = oe2eci(a,e,i,O,w,nu);
261 r0 = y(1:3);
262 v0 = y(4:6);
263 h = cross(r0,v0);
264 radial = r0 / norm(r0);
265 normal = h / norm(h);
266 tangential = cross(normal,radial);
267 A_RTN = [radial tangential normal]';
268
269 % Earth orbit initial conditions
270 aE = 149.60E6; % km
271 eE = 0.0167086;
272 iE = 7.155; % degree
273 OE = 174.9; % degree
274 wE = 288.1; % degree
275 nuE = 0;
276 muSun = 1.327E11; % km^3 / s^2
277 nE = sqrt(muSun / aE^3);
278 ySun = oe2eci(aE,eE,iE,OE,wE,nuE);
279
280 % Initial conditions
281 state0 = zeros(12,1);
282 state0(1:6) = y;
283 state0(7:9) = [0; 0; n];
284 state0(10:12) = A2e(A_RTN);
285 state0(13:18) = ySun;
286
287 % Properties
```

```
[barycenter,normal,area] = surfaces('res/area.csv',rot');
cm = computeCM('res/mass.csv');
I = computeMOI('res/mass.csv',cm);
[rot,~] = eig(I);
cmP = rot' * cm;

% Parameters
CD = 2;
Cd = 0; Cs = 0.9;
P = 1358 / 3e8;
S_sat = 24.92;
m_max = 4e-7 * pi * S_sat * 0.1;
m_direction_body = [1; 0; 0];
m_direction = rot * m_direction_body;
m = m_max * m_direction / norm(m_direction); % Arbitrary sat dipole
UT1 = [2024 1 1];

% Run numerical method
options = odeset('RelTol',1e-6,'AbsTol',1e-9);
[t,state] = ode113(@(t,state) orbitTorque(t,state,Ix,Iy,Iz, ...
    CD,Cd,Cs,P,m,UT1, ...
    barycenter,normal,area,cmP,n), ...
    tspan,state0,options);

% Compute torques (since ode113 does not allow returning these)
Rx = [1 0 0; 0 cosd(23.5) -sind(23.5); 0 sind(23.5) cosd(23.5)];
c = zeros(size(state(:,1:3)));
Mgg = zeros(size(state(:,1:3)));
Md = zeros(size(state(:,1:3)));
Msrp = zeros(size(state(:,1:3)));
Mm = zeros(size(state(:,1:3)));

for i = 1:length(t)
    r = state(i,1:3)';
    v = state(i,1:3)';
    radial = r / norm(r);
    rEarth = state(i,13:15)';
    A_ECI2P = e2A(state(i,10:12));

    c(i,1:3) = A_ECI2P * radial;
    Mgg(i,1:3) = gravGradTorque(Ix,Iy,Iz,n,c(i,1:3));

    [~,density] = atmosnrlmsise00(1000 * (norm(r) - ...
        6378.1),0,0,2000,1,0);
    rho = density(6);
    vPrincipal = A_ECI2P * (v + cross([0; 0; 7.2921159E-5],r));
    [~,M] = drag(vPrincipal,rho,CD,barycenter,normal,area,cmP);
    Md(i,1:3) = M;

    s = A_ECI2P * (-Rx * rEarth - r);
    [~,M] = srp(s,P,Cd,Cs,barycenter,normal,area,cmP);
    Msrp(i,1:3) = M;

    M = magFieldTorque(m,r,state(i,10:12),t(i),6378.1,UT1);
    Mm(i,1:3) = M;
end

figure()
```

```
plot(t / 3600,state(:,7:9))
xlabel('Time [h]')
ylabel('Angular Velocity in Principal Axes [rad/s]')
legend('\omega_{x}','\omega_{y}','\omega_{z}')
if savePlot == true
    saveas(gcf,'Images/ps5_problem3_angvel.png')
end

figure()
plot(t / 3600,Mgg)
xlabel('Time [h]')
ylabel('Gravity Gradient Torque in Principal Axes [Nm]')
legend('M_{x}','M_{y}','M_{z}')
if savePlot == true
    saveas(gcf,'Images/ps5_problem3_grav.png')
end

figure()
plot(t / 3600,Md)
xlabel('Time [h]')
ylabel('Drag Torque in Principal Axes [Nm]')
legend('M_{x}','M_{y}','M_{z}')
if savePlot == true
    saveas(gcf,'Images/ps5_problem3_drag.png')
end

figure()
plot(t / 3600,Msrp)
xlabel('Time [h]')
ylabel('Solar Radiation Pressure Torque in Principal Axes [Nm]')
legend('M_{x}','M_{y}','M_{z}')
if savePlot == true
    saveas(gcf,'Images/ps5_problem3_srp.png')
end

figure()
plot(t / 3600,Mm)
xlabel('Time [h]')
ylabel('Magnetic Field Torque in Principal Axes [Nm]')
legend('M_{x}','M_{y}','M_{z}')
if savePlot == true
    saveas(gcf,'Images/ps5_problem3_mag.png')
end

figure()
plot(t / 3600,Mgg + Md + Msrp + Mm)
xlabel('Time [h]')
ylabel('Net Torque in Principal Axes [Nm]')
legend('M_{x}','M_{y}','M_{z}')
if savePlot == true
    saveas(gcf,'Images/ps5_problem3_net.png')
end

%%
figure()
plot(t, state(:,10:12))

%% Problem 3 Maximum Torques
```

```
% Parameters
CD = 2;
Cd = 0; Cs = 0.9;
q = Cd + Cs;
P = 1358 / 3E8;
S_sat = 24.92;
m_max = 4e-7 * pi * S_sat * 0.1;
UT1 = [2024 1 1];
rE3_B0 = 7.943e15; % Wb-km

r_norm = norm(state(1,1:3));
Mgg_max = 3 / 2 * muE / (r_norm^3) * abs(max([Ix Iy Iz]) - min([Ix ...
    Iy Iz]));
Mm_max = 2 * m_max * rE3_B0 / ((r_norm * 1e3)^3);
Msrp_max = 0;
Md_max = 0;

vMax = max(vecnorm(state(:,4:6)')) * 1e3;

for n = 1:length(area)
    Msrp_max = Msrp_max + P * area(n) * (1+q) * norm(barycenter(:,n) ...
        - cmP);
    Md_max = Md_max + 0.5 * rho * CD * vMax^2 * area(n) * ...
        norm(barycenter(:,n) - cmP);
end

fprintf("Maximum expected values: \n" + ...
        "M_gg: %d Nm \n" + ...
        "M_m: %d Nm \n" + ...
        "M_srp: %d Nm \n" + ...
        "M_d: %d Nm\n", ...
        Mgg_max,Mm_max,Msrp_max,Md_max);
```

```
function [kR,kT] = plotGravGradStability(kR,kT,nameText,namePlot)
    % Gather points
    fimplicit(@(x,y) 1 + 3 * x + y * x + 4 * sqrt(y * x),[-1,1,-1,1])
    leftBranch = findobj(gcf,'Type','ImplicitFunctionLine');
    xLeft = leftBranch.XData;
    yLeft = leftBranch.YData;
    close()
    fimplicit(@(x,y) 1 + 3 * x + y * x - 4 * sqrt(y * x),[-1,1,-1,1])
    rightBranch = findobj(gcf,'Type','ImplicitFunctionLine');
    xRight = rightBranch.XData;
    yRight = rightBranch.YData;
    close()
    hold on

    % Plot yaw, roll unstable
    plot(polyshape([0 0 1 1],[-1 0 0 -1]), ...
        'FaceAlpha',1, ...
        'FaceColor','b', ...
        'DisplayName','Unstable yaw, roll')
    plot(polyshape([xRight(27:end) -1],[yRight(27:end) -1]), ...
        'FaceAlpha',1, ...
        'FaceColor','b', ...
        'HandleVisibility','off')
```

```
24
25      % Plot pitch unstable
26      plot(polyshape([0 1 0],[0 1 1]), ...
27          'FaceAlpha',1, ...
28          'FaceColor','y', ...
29          'DisplayName','Unstable pitch')
30      plot(polyshape([xLeft -1],[yLeft 0]), ...
31          'FaceAlpha',1, ...
32          'FaceColor','y', ...
33          'HandleVisibility','off')
34      plot(polyshape([xRight(1:27) 0],[yRight(1:27) 0]), ...
35          'FaceAlpha',1, ...
36          'FaceColor','y', ...
37          'HandleVisibility','off')
38
39      % Plot yaw, roll, pitch unstable
40      plot(polyshape([-1 -1 0 0],[0 1 1 0]), ...
41          'FaceAlpha',1, ...
42          'FaceColor','g', ...
43          'DisplayName', ...
44          'Unstable yaw, roll, pitch')
45      plot(polyshape([flip(xRight(1:27)) xLeft -1], ...
46          [flip(yRight(1:27)) yLeft -1]), ...
47          'FaceAlpha',1, ...
48          'FaceColor','g', ...
49          'HandleVisibility','off')
50
51      % Plot spacecraft location
52      plot(kT,kR,'x','Color','k','LineWidth',2,'DisplayName',nameText)
53
54      axis equal
55      legend()
56      xlabel('k_{T}')
57      ylabel('k_{R}')
58      xlim([-1 1])
59      ylim([-1 1])
60      hold off
61      saveas(gcf,namePlot)
62  end
```

```
1   function [stateDot] = orbitTorque(t,state,Ix,Iy,Iz, ...
2       CD,Cd,Cs,P,m,UT1, ...
3       barycenter,normal,area,cm,n)
4       warning('off','aero:atmosnrlmsise00:setf107af107aph')
5
6       % Orbit position and velocity
7       r = state(1:3);
8       v = state(4:6);
9       rEarth = state(13:15);
10      vEarth = state(16:18);
11
12      % Angular velocity
13      w = state(7:9);
14
15      % Euler angles
16      phi = state(10);
```

```
    theta = state(11);

    % Gravity gradient torque
    radial = r / norm(r);
    A_ECI2P = e2A(state(10:12));
    c = A_ECI2P * radial;
    Mgg = gravGradTorque(Ix,Iy,Iz,n,c);

    % Drag torque
    % Hard-coded with Earth radius for now
    [~,density] = atmosnrlmsise00(1000 * (norm(r) - ...
        6378.1),0,0,2000,1,0);
    rho = density(6);
    vPrincipal = A_ECI2P * (v + cross([0; 0; 7.2921159E-5],r));
    [~,Md] = drag(vPrincipal,rho,CD,barycenter,normal,area,cm);

    % Solar radiation pressure torque
    % Hard-coded with Earth axial tilt for now
    Rx = [1 0 0; 0 cosd(23.5) -sind(23.5); 0 sind(23.5) cosd(23.5)];
    s = A_ECI2P * (-Rx * rEarth - r); % SCI -> ECI -> XYZ
    [~,Msrp] = srp(s,P,Cd,Cs,barycenter,normal,area,cm);

    % Magnetic field torque
    % Hard-coded with Earth radius for now
    Mm = magFieldTorque(m,r,state(10:12),t,6378.1,UT1);

    % Compute net moments
    Mx = Mgg(1) + Md(1) + Msrp(1) + Mm(1);
    My = Mgg(2) + Md(2) + Msrp(2) + Mm(2);
    Mz = Mgg(3) + Md(3) + Msrp(3) + Mm(3);
    % Mx = 0; My = 0; Mz = 0; % Use this to test without disturbances

    % Time derivatives
    stateDot = zeros(12,1);
    stateDot(1:3) = v;
    stateDot(4:6) = (-3.986e5 / norm(r)^2) * r / norm(r); % km/s^2
    stateDot(7) = (Mx - (Iz - Iy) * w(2) * w(3)) / Ix;
    stateDot(8) = (My - (Ix - Iz) * w(3) * w(1)) / Iy;
    stateDot(9) = (Mz - (Iy - Ix) * w(1) * w(2)) / Iz;

    % 312 Euler angle time derivatives
    EPrimeInv = [sin(phi)*sin(theta) cos(phi)*sin(theta) cos(theta); ...
        cos(theta)*cos(phi) -sin(phi)*cos(theta) 0; ...
        sin(phi) cos(phi) 0] * (1 / cos(theta));
    stateDot(10:12) = EPrimeInv * w;

    % Sun position
    stateDot(13:15) = vEarth;
    stateDot(16:18) = (-1.327E11 / norm(rEarth)^2) * ...
        rEarth / norm(rEarth); % km/s^2
end
```

```
function [F,M] = drag(v,rho,CD,barycenter,normal,area,cm)
    % Compute drag in principal axes
    % RE = 6378.1 % km
    u = v / norm(v);
```

```
    N = normal;
    Aeff = ((u' * N) > 0) .* area;
    rC = barycenter - cm;
    D = -0.5 * CD * rho * norm(v)^2 * (u' * (N .* Aeff)) .* u;
    F = sum(D,2);
    M = sum(cross(rC,D),2);

    % Slow loop function (obsolete)
    % u = v / norm(v);
    % F = zeros([3 1]);
    % M = zeros([3 1]);
    % for i = length(area)
    %     n = normal(:,i);
    %     if dot(u,n) < 0
    %         rC = (barycenter(:,i) - cm);
    %         A = area(1,i);
    %         D = -0.5 * CD * rho * norm(v)^2 * dot(u,n) * u * A;
    %         M = M + cross(rC,D);
    %         F = F + D;
    %     end
    % end
end
```

```
function [F,M] = srp(s,P,Cd,Cs,barycenter,normal,area,cm)
    % Compute solar radiation pressure in principal axes
    u = s / norm(s);
    N = normal;
    Aeff = ((u' * N) > 0) .* area;
    rC = barycenter - cm;
    theta = acos((u' * N) ./ (norm(u) * vecnorm(N)));
    SRP = -P * cos(theta) .* Aeff .* ...
        ((1 - Cs) * u + 2 * (Cs * cos(theta) + Cd / 3) .* N);
    F = sum(SRP,2);
    M = sum(cross(rC,SRP),2);

    % Slow loop function (obsolete)
    % F = zeros([3 1]);
    % M = zeros([3 1]);
    % for i = length(area)
    %     n = normal(:,i);
    %     if dot(u,n) > 0
    %         rC = barycenter(:,i) - cm;
    %         theta = acos(dot(u,n) / (norm(u) * norm(n)));
    %         A = area(1,i);
    %         SRP = -P * cos(theta) * A * ...
    %             ((1 - Cs) * u + 2 * (Cs * cos(theta) + Cd / 3) * n);
    %         M = M + cross(rC,SRP);
    %         F = F + SRP;
    %     end
    % end
end
```

The following set of functions are used for our fourth-order magnetic field model.

```
function [M,B_ECEF] = magFieldTorque(m,R,eulerAngle,t,RE,UT1)
    % Calculated the expected torque due to Earth's magnetic field
    % Inputs:
    % - m: magnetic moment of satellite [N * m / T]
    % - R: position vector of satellite [km]
    % - t: time of simulation [s]
    % - RE: radius of Earth [km] (6378 km)
    % - UT1: start time of simulation

    GMST = time2GMST(t,UT12MJD(UT1));
    [lat,lon,~] = ECEF2Geoc(ECI2ECEF(R,GMST),t);
    theta = pi/2 - lat;

    [B_R,B_theta,B_phi] = magFieldEarth(R,lon,theta,RE);

    delta = lat;
    alpha = lon + GMST;

    B_x = (B_R * cos(delta) + B_theta * sin(delta)) * ...
        cos(alpha) - B_phi * sin(alpha);
    B_y = (B_R * cos(delta) + B_theta * sin(delta)) * ...
        sin(alpha) + B_phi * cos(alpha);
    B_z = (B_R * sin(delta) - B_theta * cos(delta));

    B_ECI = [B_x;B_y;B_z];
    B_ECEF = ECI2ECEF(B_ECI, GMST);
    B = e2A(eulerAngle) * B_ECI;

    M = cross(m,B);
end
```

```
function [B_R,B_theta,B_phi] = magFieldEarth(R,phi,theta,RE)
    % Make sure that R is normalized
    R = norm(R);

    % For g & h matrix (row = n, col = m + 1)
    g = [-30186 -2036 0 0 0; ...
        -1898 2997 1551 0 0; ...
        1299 -2144 1296 805 0; ...
        951 807 462 -393 235] * 1e-9; % T
    h = [0 5735 0 0 0; ...
        0 -2124 -37 0 0; ...
        0 -361 249 -253 0; ...
        0 148 -264 37 -307] * 1e-9; % T

    B_R = 0;
    B_theta = 0;
    B_phi = 0;
    for n = 1:4
        BR_temp = 0;
        BTheta_temp = 0;
        BPhi_temp = 0;

```

```
        for mInd = 1:n+1
            m = mInd - 1;

            P_nm = getPnm(theta,n,m);
            dPnm_dtheta = getdPdTheta(theta,n,m);

            BR_temp = BR_temp + ...
                (g(n,mInd) * cos(m * phi) + h(n,mInd) * sin(m * ...
                    phi)) * ...
                P_nm;
            BTheta_temp = BTheta_temp + ...
                (g(n,mInd) * cos(m * phi)+ h(n,mInd) * sin(m * phi)) ...
                    * ...
                dPnm_dtheta;
            BPhi_temp = BPhi_temp + ...
                (-g(n,mInd) * sin(m * phi) + h(n,mInd) * cos(m * ...
                    phi)) * ...
                m * P_nm;
        end

        B_R = B_R + (RE / R)^(n + 2) * (n + 1) * BR_temp;
        B_theta = B_theta + (RE / R)^(n + 2) * BTheta_temp;
        B_phi = B_phi + (RE / R)^(n + 2) * BPhi_temp;
    end

    B_theta = -B_theta;
    B_phi = -1 / sin(theta) * B_phi;
end
```

```
function dPdTheta = getdPdTheta(theta,n,m)
    if n < m
        error("n >= m for Legendre functions")
    end
    if n == m && m == 0
        dPdTheta = 0;
    elseif n == m
        dPdTheta = sin(theta) * getdPdTheta(theta,n-1,n-1) + ...
            cos(theta) * getPnm(theta,n-1,n-1);
    else
        K = getKnm(n,m);
        if K == 0
            dPdTheta = cos(theta) * getdPdTheta(theta,n-1,m) - ...
                sin(theta) * getPnm(theta,n-1,m);
        else
            dPdTheta = cos(theta) * getdPdTheta(theta,n-1,m) - ...
                sin(theta) * getPnm(theta,n-1,m) - ...
                K * getdPdTheta(theta,n-2,m);
        end
    end
end
```

```
function K = getKnm(n,m)
    if n == 1
        K = 0;
    else
```

```
        K = ((n - 1)^2 - m^2)/((2 * n - 1)*(2 * n - 3));
    end
end
```

```
function P = getPnm(theta,n,m)
    if n == 0 && m == 0
        P = 1;
    elseif n == m
        P = sin(theta) * getPnm(theta,n-1,n-1);
    else
        K = getKnm(n,m);
        if K == 0
            P = cos(theta) * getPnm(theta,n-1,m);
        else
            P = cos(theta) * getPnm(theta,n-1,m) - K * ...
                getPnm(theta,n-2,m);
        end
    end
end
```

A.6 Problem Set 6 – Attitude Determination

```
close all; clear; clc;
savePlots = true;

%% Import mass properties
cm = computeCM('res/mass.csv');
I = computeMOI('res/mass.csv',cm);

[rot,IPrincipal] = eig(I);
Ix = IPrincipal(1,1);
Iy = IPrincipal(2,2);
Iz = IPrincipal(3,3);

%% Problem 2-3
% Problem 2 assumes no disturbances. We edit orbitTorque to do this.
tFinal = 6000;
tStep = 1;
tspan = 0:tStep:tFinal;

% Satellite orbit initial conditions
a = 7125.48662; % km
e = 0;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree
muE = 3.986e5; % km^3 / s^2
n = sqrt(muE / a^3);

% Compute initial position and attitude
y = oe2eci(a,e,i,O,w,nu);
r0 = y(1:3);
v0 = y(4:6);
h = cross(r0,v0);
radial = r0 / norm(r0);
normal = h / norm(h);
tangential = cross(normal,radial);
A_Nominal = [-radial -normal -tangential]';

% Earth orbit initial conditions
aE = 149.60E6; % km
eE = 0.0167086;
iE = 7.155; % degree
OE = 174.9; % degree
wE = 288.1; % degree
nuE = 0;
muSun = 1.327E11; % km^3 / s^2
nE = sqrt(muSun / aE^3);
ySun = oe2eci(aE,eE,iE,OE,wE,nuE);

% Initial conditions
state0 = zeros(12,1);
state0(1:6) = y;
state0(7:9) = [0; -n; 0];
state0(10:12) = A2e(A_Nominal);
state0(13:18) = ySun;
```

```
56
57 % Properties
58 [barycenter,normal,area] = surfaces('res/area.csv',rot');
59 cm = computeCM('res/mass.csv');
60 I = computeMOI('res/mass.csv',cm);
61 [rot,~] = eig(I);
62 cmP = rot' * cm;
63
64 % Parameters
65 CD = 2;
66 Cd = 0; Cs = 0.9;
67 P = 1358 / 3e8;
68 S_sat = 24.92;
69 m_max = 4e-7 * pi * S_sat * 0.1;
70 m_direction_body = [1; 0; 0];
71 m_direction = rot * m_direction_body;
72 m = m_max * m_direction / norm(m_direction); % Arbitrary sat dipole
73 UT1 = [2024 1 1];
74
75 % Run numerical method
76 options = odeset('RelTol',1e-6,'AbsTol',1e-9);
77 [t,state] = ode113(@(t,state) orbitTorque(t,state,Ix,Iy,Iz, ...
78     CD,Cd,Cs,P,m,UT1, ...
79     barycenter,normal,area,cmP,n), ...
80     tspan,state0,options);
81
82 eulerError = zeros(size(state(:,10:12)));
83 eulerTarget = zeros(size(state(:,10:12)));
84 A_Sats = zeros([3,3,length(t)]);
85 for i = 1:length(t)
86     r = state(i,1:3)';
87     v = state(i,4:6)';
88     h = cross(r,v);
89     radial = r / norm(r);
90     normal = h / norm(h);
91     tangential = cross(normal,radial);
92     A_Target = [-radial -normal -tangential]'; % ECI -> RTN
93     A_Sat = e2A(state(i,10:12)); % ECI -> Principal
94     A_Sats(:,:,i) = A_Sat;
95     eulerTarget(i,1:3) = A2e(A_Target);
96     eulerError(i,1:3) = A2e(A_Sat * A_Target');
97 end
98
99 figure()
100 plot(t / 3600,wrapToPi(state(:,10:12)))
101 xlabel('Time [h]')
102 ylabel('Euler Angle (Principal) [rad]')
103 legend('\phi','\theta','\psi')
104 saveAsBool(gcf,'Images/ps6_problem2_principal.png',savePlots)
105 % saveAsBool(gcf,'Images/ps6_problem3_principal.png',savePlots)
106
107 figure()
108 plot(t / 3600,wrapToPi(eulerTarget))
109 xlabel('Time [h]')
110 ylabel('Euler Angles (Target) [rad]')
111 legend('\phi','\theta','\psi')
112 saveAsBool(gcf,'Images/ps6_problem2_target.png',savePlots)
113 % saveAsBool(gcf,'Images/ps6_problem3_target.png',savePlots)
```

```
114
115 figure()
116 plot(t / 3600,wrapToPi(eulerError))
117 xlabel('Time [h]')
118 ylabel('Euler Angle Error [rad]')
119 legend('\phi','\theta','\psi')
120 saveAsBool(gcf,'Images/ps6_problem2_error.png',savePlots)
121 % saveAsBool(gcf,'Images/ps6_problem3_error.png',savePlots)
122
123 %% Problem 5
124 numReadings = 5;
125
126 v = rand([3, numReadings]);
127 v  = v ./ vecnorm(v);
128 w = [100 100 ones([1, numReadings-2])];
129
130 eulerReals = state(:,10:12);
131 eulerDAD2 = nan([3, length(t)]);
132 eulerDAD = nan([3, length(t)]);
133 eulerqMethod = nan([3, length(t)]);
134 eulerKin = nan([3, length(t)]);
135
136 eulerKin(:,1) = state0(10:12);
137 omega = [0; -n; 0];
138
139 for n = 1:length(t)
140     m = A_Sats(:,:,n) * v;
141     m1 = m(:,1);
142     m2 = m(:,2);
143     v1 = v(:,1);
144     v2 = v(:,2);
145     A_DAD2 = DAD2Vec(m1, m2, v1, v2);
146     eulerDAD2(:,n) = A2e(A_DAD2);
147     A_DAD = DAD(m,v);
148     eulerDAD(:,n) = A2e(A_DAD);
149     qMeas_qMethod = qMethod(m,v,w);
150     eulerqMethod(:,n) = A2e(q2A(qMeas_qMethod));
151
152     % Propagate
153     [eulerKin(:,n+1), omega] = kinEulerStepRK4(eulerKin(:,1), omega, ...
          Ix, Iy, Iz, tStep);
154 end
155
156 figure()
157 plot(t/3600, wrapToPi(eulerReals))
158 title("Actual Attitude")
159 xlabel('Time [h]')
160 ylabel('Euler Angle (Principal) [rad]')
161 legend('\phi','\theta','\psi')
162 saveAsBool(gcf,'Images/ps6_problem6_actual.png',savePlots)
163
164
165 figure()
166 plot(t/3600, eulerDAD2)
167 title("Deterministic Attitude Determination (Fictitious Measurements)")
168 xlabel('Time [h]')
169 ylabel('Euler Angle (Principal) [rad]')
170 legend('\phi','\theta','\psi')
```

```
saveAsBool(gcf,'Images/ps6_problem6_DADFict.png',savePlots)

figure()
plot(t/3600, eulerDAD)
title("Deterministic Attitude Determination")
xlabel('Time [h]')
ylabel('Euler Angle (Principal) [rad]')
legend('\phi','\theta','\psi')
saveAsBool(gcf,'Images/ps6_problem6_DAD.png',savePlots)

figure()
plot(t/3600, eulerqMethod)
title("q Method")
xlabel('Time [h]')
ylabel('Euler Angle (Principal) [rad]')
legend('\phi','\theta','\psi')
saveAsBool(gcf,'Images/ps6_problem6_qMethod.png',savePlots)
```

```
function A = DAD_twoVecs(m1, m2, v1, v2)
    % Implement the Deterministic Atittude Determination algorithm
    % Inputs:
    % - m1, m2: measured vectors
    % - v1, v2: ground truth vector
    % Outputs:
    % - A: DCM between principal axes and ECI

    m1_tilde = (m1 + m2)/2;
    m2_tilde = (m1 - m2)/2;
    v1_tilde = (v1 + v2)/2;
    v2_tilde = (v1 - v2)/2;

    m1_tilde = m1_tilde / norm(m1_tilde);
    m2_tilde = m2_tilde / norm(m2_tilde);
    v1_tilde = v1_tilde / norm(v1_tilde);
    v2_tilde = v2_tilde / norm(v2_tilde);

    pm = m1_tilde;
    cross_qm = cross(m1_tilde, m2_tilde);
    qm = cross_qm/norm(cross_qm);
    rm = cross(pm, qm);

    pv = v1_tilde;
    cross_qv = cross(v1_tilde, v2_tilde);
    qv = cross_qv/norm(cross_qv);
    rv = cross(pv,qv);

    M = [pm qm rm];
    V = [pv qv rv];
    A = M / V;
end
```

```
function A = DAD(m, v)
    % Implement the Deterministic Atittude Determination algorithm
    % Inputs:
    % - m: measured vectors ([m1 m2 . . . ])
```

```
    % - v: ground truth vector ([v1 v2 . . . ])
    % Outputs:
    % - A: DCM between principal axes and ECI

    A = m * pinv(v);

end
```

```
function q = qMethod(m, v, w)
    % Implement the Deterministic q method
    % Inputs:
    % - m: measured vectors
    % - v: ground truth vector
    % - w: weights of sensor measurements
    % Outputs:
    % - A: DCM between principal axes and ECI

    U = sqrt(w) .* v;
    W = sqrt(w) .* m;

    B = W * U';
    S = B + B';
    Z = [B(2,3)-B(3,2), B(3,1)-B(1,3), B(1,2)-B(2,1)]';
    sigma = trace(B);

    K = [S - eye(size(S))*sigma, Z;
            Z', sigma];

    [V, lamda] = eig(K);

    [~, nMax] = max(diag(lamda));

    q = V(:, nMax);
    q = A2q(q2A(q));
end
```

```
function [eulerAngs, omega] = kinEulerStepRK4(eulerAngs, omega, Ix, ...
    Iy, Iz, tStep)
    state = zeros(6,1);
    state(1:3) = eulerAngs;
    state(4:6) = omega;

    t = 0;

    k1 = kinEulerAngle(t,state,Ix,Iy,Iz);
    k2 = kinEulerAngle(t+tStep/2,state+(k1*tStep/2),Ix,Iy,Iz);
    k3 = kinEulerAngle(t+tStep/2,state+(k2*tStep/2),Ix,Iy,Iz);
    k4 = kinEulerAngle(t+tStep,state+(k3*tStep),Ix,Iy,Iz);
    nextState = state + tStep * (k1/6 + k2/3 + k3/3 + k4/6);

    eulerAngs = nextState(1:3);
    omega = nextState(4:6);
end
```

A.7 Problem Set 7 – MEKF Time Update

```
%%
close all; clear; clc;
savePlots = false;

%% Import mass properties
cm = computeCM('res/mass.csv');
I = computeMOI('res/mass.csv',cm);

[rot,IPrincipal] = eig(I);
Ix = IPrincipal(1,1);
Iy = IPrincipal(2,2);
Iz = IPrincipal(3,3);

%% Problem 1
% NOTE: Requires fixed timestep (0.1s) to get good data
qVals = squeeze(out.q.data);
qMeasVals = squeeze(out.qMeasured.data);
timeVals = squeeze(out.q.time);
timeValsMeas = squeeze(out.qMeasured.time);

eulerVals = quats2Euler(qVals);
eulerValsMeas = quats2Euler(qMeasVals);

figure(1)
hold on
plot(timeValsMeas, rad2deg(eulerValsMeas), 'b')
plot(timeVals, rad2deg(eulerVals), 'r--')
hold off

%% Problem 2
eulerError = zeros(size(eulerVals));
A_error = zeros(3,3,length(timeVals));
for n = 1:length(timeVals)
    A_ECI2true = e2A(eulerVals(:,n));
    A_ECI2err = e2A(eulerValsMeas(:,n));
    A_true2err = A_ECI2err * A_ECI2true';
    eulerError(:,n) = A2e(A_true2err);
    A_error(:,:,n) = A_true2err;
end

figure(1)
plot(timeVals/3600, rad2deg(eulerError))
xlim([0 timeVals(end)/3600])
xlabel("time [hr]")
ylabel("euler angle [deg]")
legend("\phi", "\theta", "\psi")
ylim([-20, 20])
% saveas(1, "Images/ps7_problem2_qMethod.png")
% saveas(1, "Images/ps7_problem2_DAD.png")
% saveas(1, "Images/ps7_problem2_DADFict.png")
% saveas(1, "Images/ps7_problem2_kin.png")

%% Problem 3
frob_norm = nan(1, length(timeVals));

```

```
for n = 1:length(timeVals)
    phi = eulerError(1,n);
    theta = eulerError(2,n);
    psi = eulerError(3,n);
    A_smallAng = [1 phi -psi;
                  -phi 1 theta;
                  psi -theta 1];
    frob_norm(n) = norm(A_error(:,:,n) - A_smallAng);
end

figure(2)
plot(timeVals/3600, frob_norm)
xlabel("time [hr]")
ylabel("||A_{error} - A_{small angle}||_F")
xlim([0 timeVals(end)/3600])

% saveas(2, "Images/ps7_problem3.png")

%% Problem 4

figure(3)
hold on
plot(timeValsMeas, rad2deg(eulerValsMeas), 'b')
plot(timeVals, rad2deg(eulerVals), 'r--')
legend(["ground truth", "", "", "error"])
hold off

% saveas(3, "Images/ps7_problem4_qMethod.png")
% saveas(3, "Images/ps7_problem4_DAD.png")
% saveas(3, "Images/ps7_problem4_DADFict.png")
% saveas(3, "Images/ps7_problem4_kin.png")

%% Problem 5-6
tFinal = 18000;
tStep = 0.1;
tspan = 1:tStep:tFinal;

% Satellite orbit initial conditions
a = 7125.48662; % km
e = 0;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree
muE = 3.986e5; % km^3 / s^2
n = sqrt(muE / a^3);

% Compute initial position and attitude
y = oe2eci(a,e,i,O,w,nu);
r0 = y(1:3);
v0 = y(4:6);
h = cross(r0,v0);
radial = r0 / norm(r0);
normal = h / norm(h);
tangential = cross(normal,radial);
A_Nominal = [-radial -normal -tangential]';

q0 = A2q(A_Nominal);
```

```
w0 = [-n; -n; -n];
x0 = [q0; w0];

q = q0;
w = w0;
euler = A2e(A_Nominal);
P = eye(6);
Q = P / 100;
for i = 1:length(tspan)
    u = zeros([3 1]);
    [q(:,i + 1),w(:,i + 1),P] = ...
        timeUpdate(tStep,q(:,i),w(:,i),P,Q,Ix,Iy,Iz,u);
    euler(:,i + 1) = A2e(q2A(q(:,i + 1)));
end

figure()
plot(tspan,wrapToPi(euler(:,1:end-1)'))
xlabel('Time [s]')
ylabel('Euler Angle (Principal) [rad]')
legend('\phi','\theta','\psi')
saveAsBool(gcf,'Images/ps7_problem5a_angle_est.png',savePlots)

figure()
plot(tspan,wrapToPi(w(:,1:end-1)'))
xlabel('Time [s]')
ylabel('Angular Velocity [rad/s]')
legend('w_{x}','w_{y}','w_{z}')
saveAsBool(gcf,'Images/ps7_problem5a_angvel_est.png',savePlots)

% Simulation state
state0 = [A2e(A_Nominal); w0];
options = odeset('RelTol',1e-6,'AbsTol',1e-9);
[t,state] = ode113(@(t,state) kinEulerAngle(t,state,Ix,Iy,Iz), ...
    tspan,state0,options);

figure()
plot(tspan,wrapToPi(state(:,1:3)))
xlabel('Time [s]')
ylabel('Euler Angle (Principal) [rad]')
legend('\phi','\theta','\psi')
saveAsBool(gcf,'Images/ps7_problem5a_angle_sim.png',savePlots)

figure()
plot(tspan,wrapToPi(state(:,4:6)))
xlabel('Time [s]')
ylabel('Angular Velocity [rad/s]')
legend('w_{x}','w_{y}','w_{z}')
saveAsBool(gcf,'Images/ps7_problem5a_angvel_sim.png',savePlots)

% Errors
eulerError = size(state(:,1:3));
for i = 1:length(t)
    A_MEKF = e2A(euler(:,i));
    A_Sim = e2A(state(i,1:3));
    eulerError(i,1:3) = A2e(A_MEKF * A_Sim');
end

figure()
```

```
171 plot(t,wrapToPi(eulerError))
172 ylim([-0.01 0.01])
173 xlabel('Time [s]')
174 ylabel('Euler Angle Error [rad]')
175 legend('\phi','\theta','\psi')
176 saveAsBool(gcf,'Images/ps7_problem6_angle_err.png',savePlots)
177
178 figure()
179 plot(t,w(:,1:end-1)' - state(:,4:6))
180 xlabel('Time [s]')
181 ylabel('Angular Velocity Error [rad/s]')
182 legend('w_{x}','w_{y}','w_{z}')
183 saveAsBool(gcf,'Images/ps7_problem6_angvel_err.png',savePlots)
```

A.8 Problem Set 8 – MEKF Measurement Update

```
%% Model
close all; clear; clc;
savePlots = false;
modelVars

%%  Plots
qVals = squeeze(out.q.data);
qMeasVals = squeeze(out.qEstimated.data);
wVals = squeeze(out.w.data);
wMeasVals = squeeze(out.wEstimated.data);
timeVals = squeeze(out.q.time);
timeValsMeas = squeeze(out.qEstimated.time);

eulerVals = quats2Euler(qVals);
eulerValsMeas = quats2Euler(qMeasVals);

zNorm = squeeze(vecnorm(out.z.data,2,1));
zPostNorm = squeeze(vecnorm(out.zPost.data,2,1));
zNormVect = vecnorm(zNorm(1:end-1,:), 2);
zPostNormVect = vecnorm(zPostNorm(1:end-1,:), 2);
zNormGyro = squeeze(out.z.data(:,end,:));
zPostNormGyro = squeeze(out.zPost.data(:,end,:));

figure()
hold on
plot(timeVals, zNormVect, 'b')
plot(timeVals, zPostNormVect, 'r')
xlabel('Time [s]')
ylabel('Residual norms')
legend('pre-fit','post-fit')
hold off
saveAsBool(gcf,'Images/ps8_problem7_res_units.png', savePlots)

figure()
hold on
plot(timeVals, zNormGyro, 'b')
plot(timeVals, zPostNormGyro, 'r')
xlabel('Time [s]')
ylabel('Residual norms')
legend('pre-fit','','','post-fit')
hold off
saveAsBool(gcf,'Images/ps8_problem7_res_gyro.png', savePlots)

figure()
hold on
plot(timeValsMeas, rad2deg(eulerValsMeas), 'b')
plot(timeVals, rad2deg(eulerVals), 'r--')
xlabel('Time [s]')
ylabel('Euler Angle (Principal) [rad]')
legend('MEKF','','','','Ground Truth')
hold off
saveAsBool(gcf,'Images/ps8_problem7_state.png', savePlots)

% Get error
figure()
```

```
eulerError = zeros(size(eulerVals));
A_error = zeros(3,3,length(timeVals));
for n = 1:length(timeVals)
    A_ECI2true = e2A(eulerVals(:,n));
    A_ECI2err = e2A(eulerValsMeas(:,n));
    A_true2err = A_ECI2err * A_ECI2true';
    eulerError(:,n) = A2e(A_true2err);
    A_error(:,:,n) = A_true2err;
end
eulerAngNames = ["\phi", "\theta", "\psi"];
for n = 1:3
    subplot(3, 1, n)
    plot(timeVals/3600, rad2deg(eulerError(n,:)))
    xlim([0 timeVals(end)/3600])
    xlabel("time [hr]")
    ylabel(eulerAngNames(n) + " [deg]")
end
saveAsBool(gcf,'Images/ps8_problem7_error.png', savePlots)

%%
figure()
covError = [squeeze(out.Pkplus.Data(1,1,:)), ...
                  squeeze(out.Pkplus.Data(2,2,:)), ...
                  squeeze(out.Pkplus.Data(3,3,:))]';
angleNames = ["\alpha_x", "\alpha_y", "\alpha_z"];
for n = 1:3
    subplot(3, 1, n)
    hold on
    plot(timeVals/3600, 1.96.*rad2deg(covError(n,:)), 'b')
    plot(timeVals/3600, -1.96.*rad2deg(covError(n,:)), 'b')
    xlim([0 timeVals(end)/3600])
    xlabel("time [hr]")
    ylabel(angleNames(n) + " [deg]")
    hold off
end
saveAsBool(gcf,'Images/ps8_problem7_cov.png', savePlots)

figure()
angleNames = ["\alpha_x", "\alpha_y", "\alpha_z"];
alphaError = squeeze([A_error(2,3,:), A_error(3,1,:), A_error(1,2,:)]);
for n = 1:3
    subplot(3, 1, n)
    hold on
    plot(timeVals/3600, rad2deg(alphaError(n,:)), 'r')
    plot(timeVals/3600, 1.96.*rad2deg(covError(n,:)), 'b')
    plot(timeVals/3600, -1.96.*rad2deg(covError(n,:)), 'b')
    xlim([0 timeVals(end)/3600])
    xlabel("time [hr]")
    ylabel(angleNames(n) + " [deg]")
    hold off
end
saveAsBool(gcf,'Images/ps8_problem7_covComp.png', savePlots)
```

```
function measurementStarTracker = ...
    starTracker(qkminus,groundTruthVectors)
    % Extract information
```

```
    A_ECI2P = q2A(qkminus);
    rStarECI = groundTruthVectors(:,1:end-1);
    rStarP = A_ECI2P * rStarECI;

    measurementStarTracker = normVec(rStarP);
end
```

```
function measurementSunSensor = sunSensor(qkminus,groundTruthVectors)
    % Get modeled parameters
    rSCI = groundTruthVectors(:,end);
    A_ECI2P = q2A(qkminus);

    % Get sensor measurement without noise
    rSunSensors = A_ECI2P * rSCI;
    measurementSunSensor = normVec(rSunSensors);
end
```

A.9 Problem Set 9 – Control

```
close all; clear; clc;
savePlots = false;
sinWave = false;
modelVars

%% Problem 1
% Satellite orbit initial conditions
a = 7125.48662; % km
e = 0;
i = 98.40508; % degree
O = -19.61601; % degree
w = 89.99764; % degree
nu = -89.99818; % degree
muE = 3.986 * 10^5; % km^3 / s^2
n = sqrt(muE / a^3);

% Max expected torque
Md = 0.0024; % N*m

% Reaction Wheel
T = 2*pi*sqrt(a^3/muE); %s
L = Md*T*0.707/4; %Nms

% Magnetorquer
B = 4.3578e-05;
D = Md/B;

% Thruster
b = 0.6;
T = Md/b;

%% Problem 2
time = squeeze(out.q.time);
sinWave = squeeze(out.sinWave.Data);
Lw = squeeze(out.Lw.Data);
w_wheel = Lw ./ control.IWheel;

figure()
for n = 1:4
    subplot(2,2,n)
    hold on
    title(sprintf("Wheel %i", n))

    yyaxis left
    ylabel("Angular Velocity [rad/s]")
    plot(time, w_wheel(n,:), 'b')
    ylim([min(w_wheel(:)), max(w_wheel(:))])

    yyaxis right
    ylabel("Control Moment [Nm]")
    plot(time, sinWave, 'r')
    hold off

    xlabel("Time [s]")
end
```

```
56
57 saveAsBool(gcf, 'Images/ps9_problem2.png', savePlots & sinWave)
58
59 %% Question 3
60 % Get attitude determination error
61 time = squeeze(out.q.Time);
62 q = squeeze(out.q.data);
63 qMeas = squeeze(out.qEstimated.data);
64
65 eulerAngs = quats2Euler(q);
66 eulerAngsMeas = quats2Euler(qMeas);
67
68 eulerDetError = zeros(size(eulerAngs));
69 A_error = zeros(3,3,length(time));
70 for n = 1:length(time)
71     A_ECI2true = e2A(eulerAngs(:,n));
72     A_ECI2err = e2A(eulerAngsMeas(:,n));
73     A_true2err = A_ECI2err * A_ECI2true';
74     eulerDetError(:,n) = A2e(A_true2err);
75     A_error(:,:,n) = A_true2err;
76 end
77
78 % Control Error
79 eulerControlError = A2eVec(out.AE.data);
80
81 Lw = squeeze(out.Lw.Data);
82 w_wheel = Lw ./ control.IWheel;
83
84 if control.useLinearModel == true
85     model = 'linear';
86 else
87     model = 'nonlinear';
88 end
89
90 figure()
91 hold on
92 plot(time/3600, rad2deg(eulerDetError))
93 xlabel("Time [hr]"); ylabel("Euler Angles [deg]")
94 legend(["\phi", "\theta", "\psi"])
95 title("Attitude Determination Error")
96 xlim([0, time(end)/3600])
97 hold off
98 saveAsBool(gcf, ['Images/ps9_problem3_determinationError_', model, ...
       '.png'], savePlots)
99
100 figure()
101 hold on
102 plot(time/3600, rad2deg(eulerControlError))
103 xlabel("Time [hr]"); ylabel("Euler Angles [deg]")
104 legend(["\phi", "\theta", "\psi"])
105 title("Attitude Control Error (Ground Truth)")
106 xlim([0, time(end)/3600])
107 hold off
108 saveAsBool(gcf, ['Images/ps9_problem3_controlError_', model, ...
       '.png'], savePlots)
109
110 figure()
111 hold on
```

```
plot(time/3600, rad2deg(squeeze(out.alpha.data)))
xlabel("Time [hr]"); ylabel("Euler Angles [deg]")
legend(["\alpha_x", "\alpha_y", "\alpha_z"])
title("Attitude Control Error (Measured)")
xlim([0, time(end)/3600])
hold off
saveAsBool(gcf, ['Images/ps9_problem3_controlMomentsMeasured_', ...
    model, '.png'], savePlots)

Mc = squeeze(out.Mc.data);
figure()
hold on
plot(time/3600, Mc)
xlabel("Time [hr]"); ylabel("Control Moments [Nm]")
legend(["M_{cx}" "M_{cy}" "M_{cz}"])
title("Control Moments")
xlim([0, time(end)/3600])
hold off
saveAsBool(gcf, ['Images/ps9_problem3_controlMoments_', model, ...
    '.png'], savePlots)

figure()
for n = 1:4
    subplot(2,2,n)
    hold on
    title(sprintf("Wheel %i", n))

    ylabel("Angular Velocity [rad/s]")
    plot(time/3600, w_wheel(n,:), 'b')
    ylim([min(w_wheel(:)), max(w_wheel(:))])
    xlim([0, time(end)/3600])

    xlabel("Time [hr]")
end
saveAsBool(gcf, ['Images/ps9_problem3_wheelMomentum_', model, ...
    '.png'], savePlots)
```

```
function [alpha, AE_GT]= getAlpha(qEstimated, A_ECI2P,rECI,vECI,control)
    h = cross(rECI,vECI);
    radial = rECI / norm(rECI);
    normal = h / norm(h);
    tangential = cross(normal,radial);
    A_Target = [-radial -normal -tangential]';
    AE_GT = A_ECI2P * A_Target';

    AE = q2A(qEstimated) * A_Target';
    % AE = AE_GT;

    if control.useLinearModel == true
        alpha = [AE(2,3); AE(3,1); AE(1,2)];
    else
        alpha = [(AE(2,3) - AE(3,2)) / 2; ...
            (AE(3,1) - AE(1,3)) / 2; ...
            (AE(1,2) - AE(2,1)) / 2];
    end
end
```

```
function [Mw,Lwdot] = reactionWheels(Lw,Mc,control,w)
    A = control.A;
    Lwdot = pinv(A) * (-Mc - cross(w,A * Lw));

    % Check saturation and apply correction
    if any(abs(Lw) > control.LMaxWheel)
        Lwdot = (abs(Lw) > control.LMaxWheel).* Lwdot;
    end

    % Recalculate Mc
    Mw = -A*Lwdot - cross(w,A * Lw);
end
```

```
function Mc = controlLaw(alpha,alphadot,control)
    Mc = -control.Kp .* alpha - control.Kd .* alphadot;
end
```

A.10 Problem Set 10 – Momentum Management

```
close all; clear; clc;
savePlots = false;
modelVars

%% Angular Velocity Plot
time = squeeze(out.q.time);
Lw = squeeze(out.Lw.Data);
w_wheel = Lw ./ control.IWheel;

figure()
for n = 1:4
    subplot(2,2,n)
    hold on
    title(sprintf("Wheel %i", n))

    ylabel("Angular Velocity [rad/s]")
    plot(time/3600, w_wheel(n,:), 'b')
    ylim([min(w_wheel(:)), max(w_wheel(:))])
    xlabel("Time [hr]")
end

%% Control Error Plot
eulerControlError = A2eVec(out.AE.data);
figure()
hold on
plot(time/3600, rad2deg(eulerControlError))
xlabel("Time [hr]"); ylabel("Euler Angles [deg]")
legend(["\phi", "\theta", "\psi"])
title("Attitude Control Error")
xlim([0, time(end)/3600])
hold off

%% Dipole Plot
dipole = squeeze(out.magnetorquerDipole.data);
hold on
plot(time/3600, dipole)
xlabel("Time [hr]"); ylabel("Dipole Moment [Am^{2}]")
legend(["x", "y", "z"])
xlim([0, time(end)/3600])
hold off

%% Torque Plot
Mmag = squeeze(out.Mmagnetorquer.data);
hold on
plot(time/3600, Mmag)
xlabel("Time [hr]"); ylabel("Magnetorquer Torque [N-m]")
legend(["x", "y", "z"])
xlim([0, time(end)/3600])
hold off

%% Duty Cycle
hold on
plot(time/3600, (Mmag(1,:) ~= 0))
xlabel("Time [hr]"); ylabel("Magnetorquer Duty Cycle")
xlim([0, time(end)/3600])
```

```
ylim([-1 2])
hold off
```

```
function B = getMagField(rECI,A_ECI2P,t,constants)
    % Calculated the expected torque due to Earth's magnetic field
    % Inputs:
    % - m: magnetic moment of satellite [N * m / T]
    % - R: position vector of satellite [km]
    % - t: time of simulation [s]
    % - RE: radius of Earth [km] (6378 km)
    % - UT1: start time of simulation

    GMST = time2GMST(t,UT12MJD(constants.UT1));
    [lat,lon,~] = ECEF2Geoc(ECI2ECEF(rECI,GMST),t);
    theta = pi/2 - lat;

    [B_R,B_theta,B_phi] = magFieldEarth(rECI,lon,theta,constants.RE);

    delta = lat;
    alpha = lon + GMST;

    B_x = (B_R * cos(delta) + B_theta * sin(delta)) * ...
        cos(alpha) - B_phi * sin(alpha);
    B_y = (B_R * cos(delta) + B_theta * sin(delta)) * ...
        sin(alpha) + B_phi * cos(alpha);
    B_z = (B_R * sin(delta) - B_theta * cos(delta));

    B_ECI = [B_x;B_y;B_z];
    B = A_ECI2P * B_ECI;
end
```

```
function [mode,m,Mmag] = magnetorquer(mode,B,Lw,control)
    % Check if we need to dump momentum
    if all(abs(Lw) < control.LwLimit * 0.1) && mode == 1
        mode = 0; % Stop desaturation
    end
    if any(abs(Lw) > control.LwLimit) || mode == 1
        mode = 1; % Trigger desaturation
        m = control.KMag / norm(B) * cross(control.A*Lw, normVec(B));

        % Implement magnetorquer saturation
        if any(m > control.magMax)
            magOver = m(m > control.magMax);
            scaleOver = max(magOver ./ control.magMax);
            m = m ./ scaleOver;
        end

        Mmag = cross(m,B);
    else
        mode = 0;
        m = [0;0;0];
        Mmag = [0;0;0];
    end
end
```