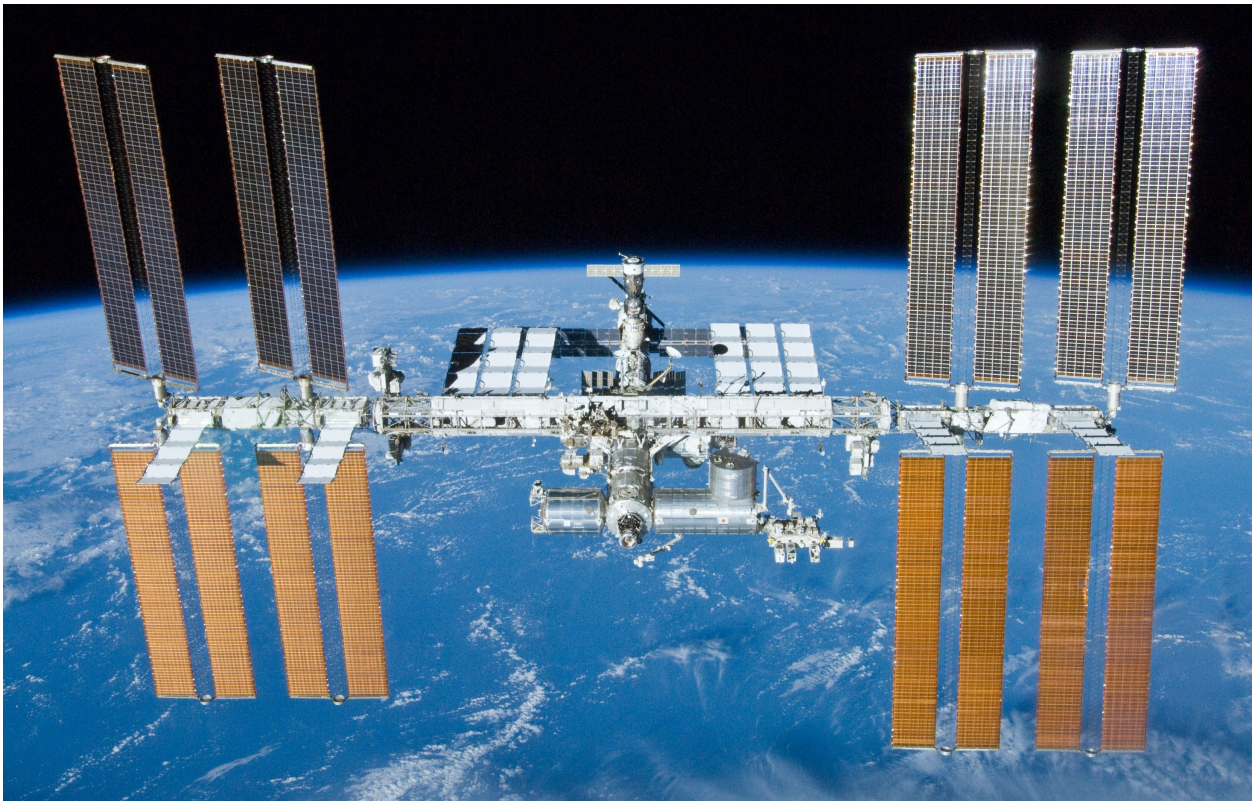
AA279c Project Report: Modeling the ISS

Toby J. Buckley

June 8, 2017



1 Problem Set 1 - Mission Setup and Details.

1.1 Project Overview

For this project the International Space Station (ISS) is modeled and explored, which orbits around the earth in low-earth-orbit (LEO). The ISS was first launched in 1998 and only fully completed in 2009. The full cost is estimated around 150 billion USD, making it the single-most expensive structure every created by man. In the nearly 20 years of operation, countless experiments have taken place onboard, dozens of astronauts have visited, and the total scientific value is immeasurable. The ISS is a tribute to humanity's ability to work together and has solidified its place in human history.

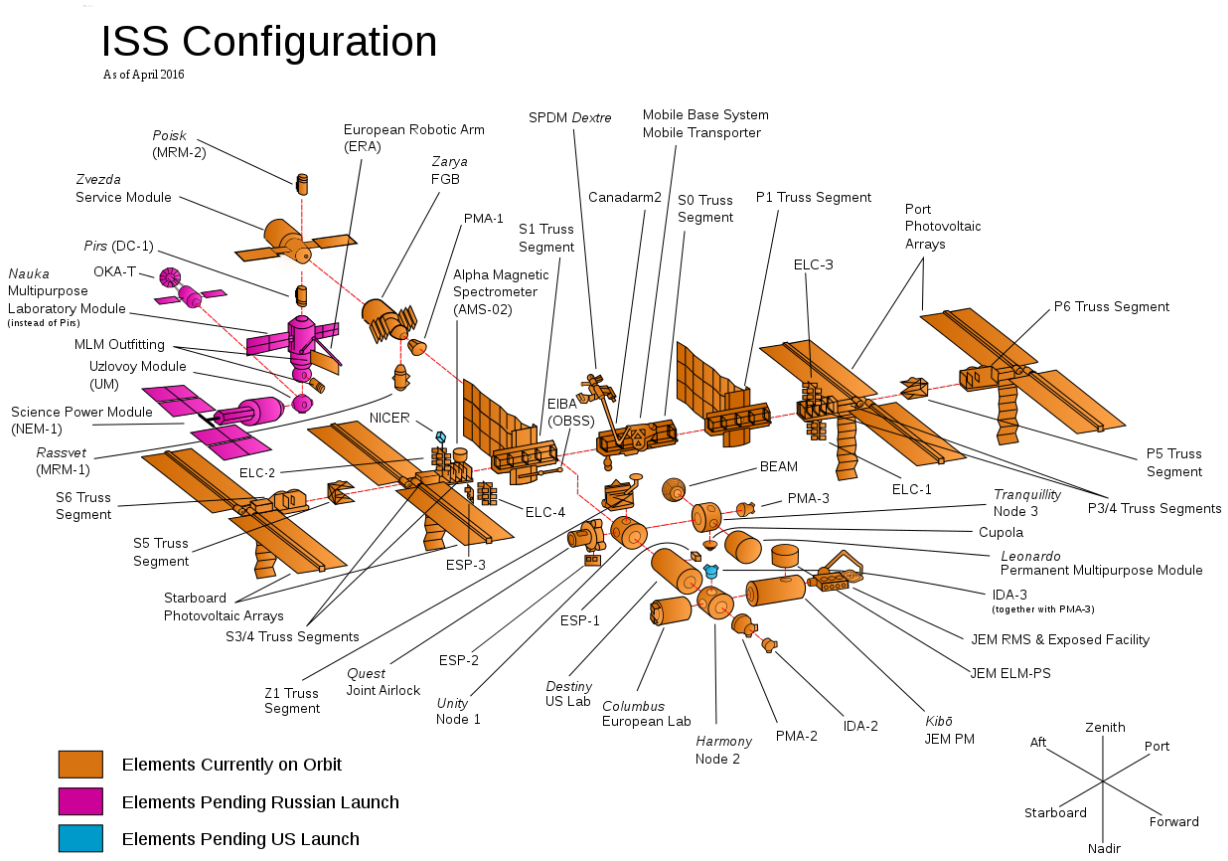


Figure 1: ISS component breakdown.

Fig. 1 shows the exploded component view of the ISS. Over the course of its assembly parts were brought up by different rockets, the two most well known being the US Space Shuttle, and the Russian Soyuz. Sizing was done by referencing the ISS's wikipedia page[1]

[1] `https://en.wikipedia.org/wiki/Assembly_of_the_International_Space_Station`

as well as documents released by NASA[2].

```
chd = {};
%main truss
chd{end+1} = mkChild(3,108.5,3,1000,-1.5,-108.5/2,-1.5,0,0,0);
%pressurized area
chd{end+1} = mkChild(3,3,45,1000,1.5,-1.5,-15,0,0,0);
%solar panel
chd{end+1} = mkChild(3, 17, 0.1, 1000, 1.5, 2, 25, 0,0,0);
chd{end+1} = mkChild(3, 17, 0.1, 1000, 1.5, -19, 25, 0,0,0);

chd{end+1} = mkChild(3, 14, 0.1, 1000, 1.5, 2, 15, 0,0,0);
chd{end+1} = mkChild(3, 14, 0.1, 1000, 1.5, -16, 15, 0,0,0);

%top solar panels
chd{end+1} = mkChild(35,12,0.5,1000,2,-50,0,0,0,0);
chd{end+1} = mkChild(35,12,0.5,1000,2,-37,0,0,0,0);
chd{end+1} = mkChild(35,12,0.5,1000,2, 38,0,0,0,0);
chd{end+1} = mkChild(35,12,0.5,1000,2, 25,0,0,0,0);

%bottom solar panels
chd{end+1} = mkChild(35,12,0.5,1000,-37,-50,0,0,0,0);
chd{end+1} = mkChild(35,12,0.5,1000,-37,-37,0,0,0,0);
chd{end+1} = mkChild(35,12,0.5,1000,-37, 38,0,0,0,0);
chd{end+1} = mkChild(35,12,0.5,1000,-37, 25,0,0,0,0);
```

Figure 2: Child creation code.

When putting together the data structure for the vehicle, I implemented a nested parent-child system, where each component of the ISS could be defined and attached with respect to the parent center. Fig. 2 shows how the different main components are defined. When information was missing, educated guesses on size were used.

In order to reduce complexity, only the largest components of the ISS were modeled. Structural elements such as the main truss, the pressurized cabin area, and solar panels were kept. Additionally, all body components are axis-aligned for ease of transformations and plotting.

Orbit propagation is performed using the two-body problem differential equation (below). This is derived using the standard Newton's second law approach assuming that the only bodies affecting each-other are the satellite and the orbiting body (Earth).

$$(m_1 + m_2)\frac{d^2}{dt^2}\vec{\boldsymbol{R}} = m_1\frac{d^2}{dt^2}\vec{\boldsymbol{x}_1} + m_2\frac{d^2}{dt^2}\vec{\boldsymbol{x}_2} = \vec{\mathbf{0}}$$

[2] https://www.nasa.gov/pdf/508318main_ISS_ref_guide_nov2010.pdf

The two-body problem is modeled in Matlab Simulink which serves as the core of the environment simulator. The state is output in the earth-centered-inertial (ECI) frame and fed into the rest of the model. Fig. 3 shows the model, which includes J2 effects and atmospheric drag. Due to the ISS's proximity to Earth, both of these greatly affect the orbit, and within only a couple orbits its clear that the satellite begins falling to earth.

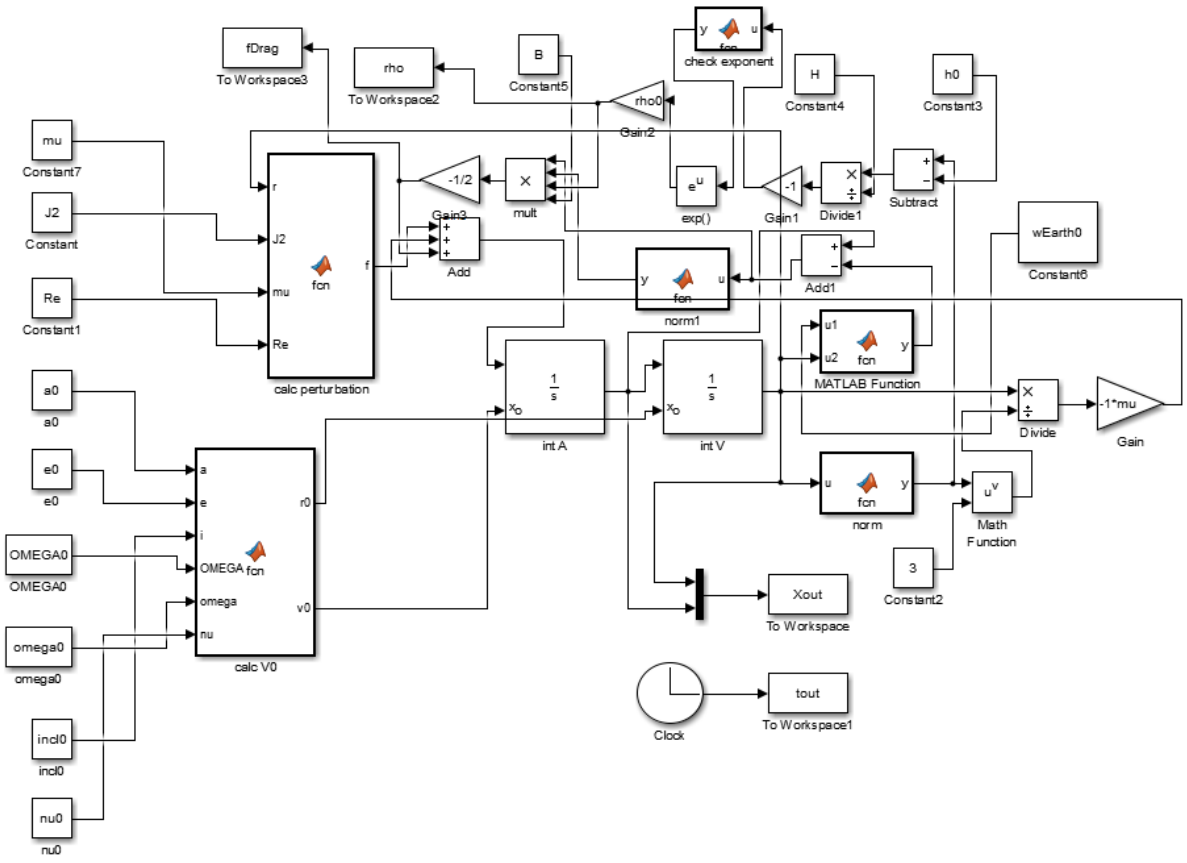


Figure 3: Simulink model of the orbit propagator. Drag and J2 perturbation are included.

When setting up the vehicle, the moment of inertia must be calculated for each component about a fixed point. In my code, each child automatically reports its inertia relative to the parent center, which is not necessarily the center of mass. Two transformations then take place; the first transforms the inertia relative to the true center of mass, and the second solves the principle axis eigenvalue problem. Eq. 1 & 2 show how these transformations are accomplished. For the eigenvalue problem, the eigenvalues are the principle axis inertia values, while the eigenvectors make up the unit triad which defines the rotation between the body and principle axes.

$$J_{ij} = I_{ij} + m(|R|^2\delta_{ij} - R_i R_j) \tag{1}$$

$$(J - I * \lambda)A = 0 \tag{2}$$

Below the moment of inertia tensor for the simplified ISS is shown, in both body and principle axes with respect to the center of mass. All computations are done using km, so these values are in $kg * km^2$.

$$J_b = \begin{pmatrix} 15.63 & 0 & -0.36 \\ 0 & 6.43 & 0 \\ -0.36 & 0 & 16.40 \end{pmatrix}, J_p = \begin{pmatrix} 6.43 & 0 & 0 \\ 0 & 15.49 & 0 \\ 0 & 0 & 16.54 \end{pmatrix} \tag{3}$$

Fig. 4 shows the full simplified vehicle. In the middle, at the center of mass there is the unit triad which rotates any vector from body to principle axis. Because the ISS is symmetric about the XZ plane, the y-component is hidden along the main truss.

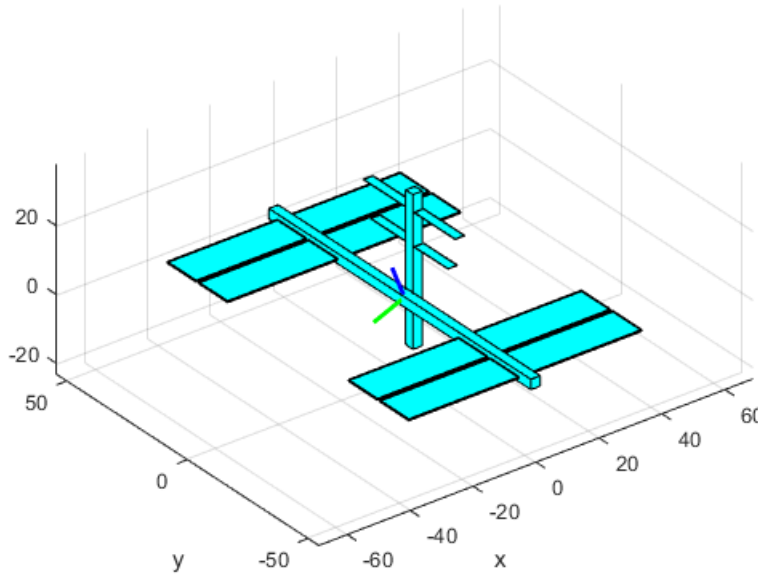


Figure 4: Model of the ISS. Shown in blue, green, and red is the unit triad from body to principle.

In order to properly model atmospheric drag and gravity gradient, it's necessary to know the location of each children surface's barycenter and normal direction. Fig. 5 shows the full vehicle with normal directions plotted at each surface' barycenter.

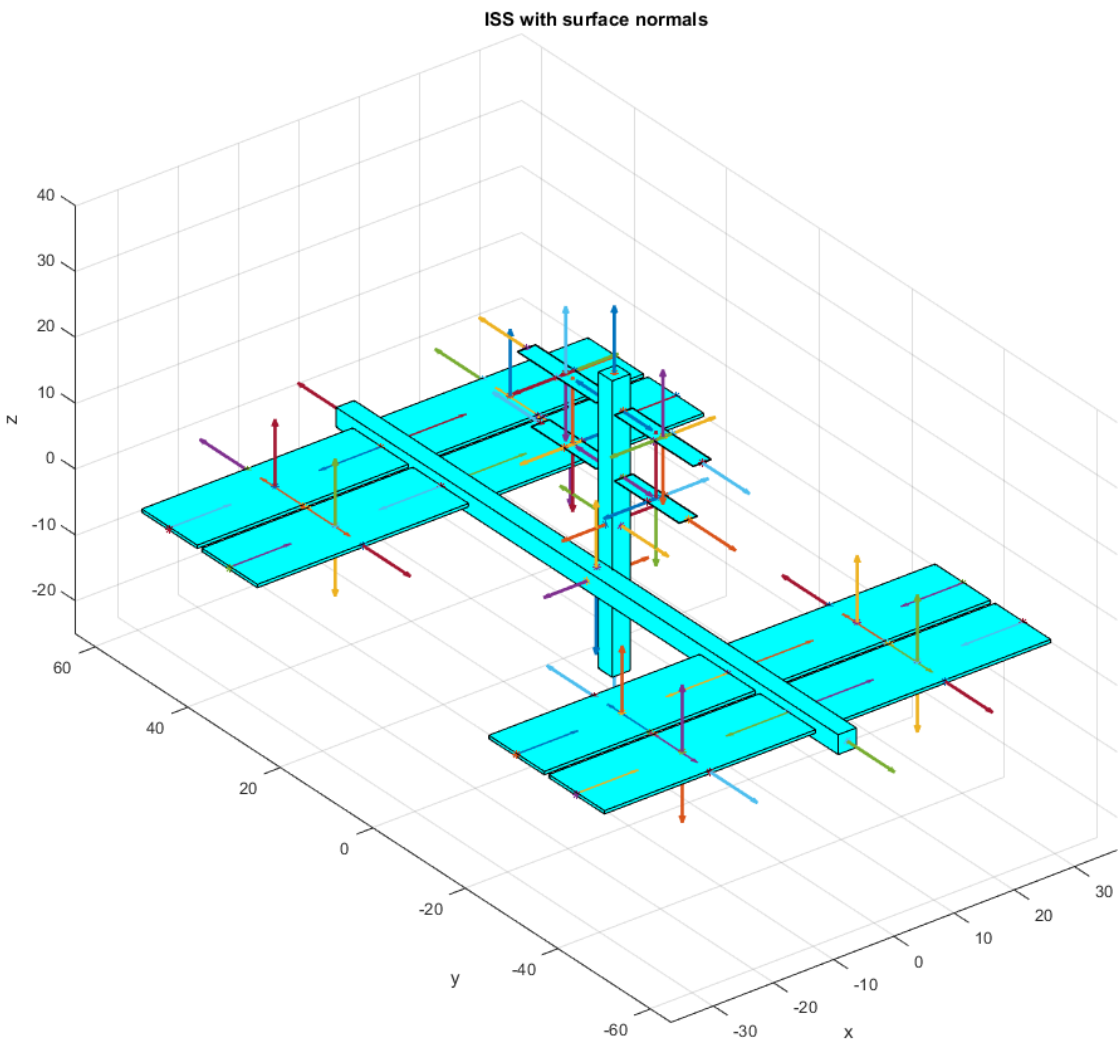


Figure 5: The surface normals for each child are shown.

The orbit propagation is demonstrated in Fig. 6. Due to the proximity to Earth, the ISS travels at 4.76 miles/s and completes a single orbit in just around 90 minutes.

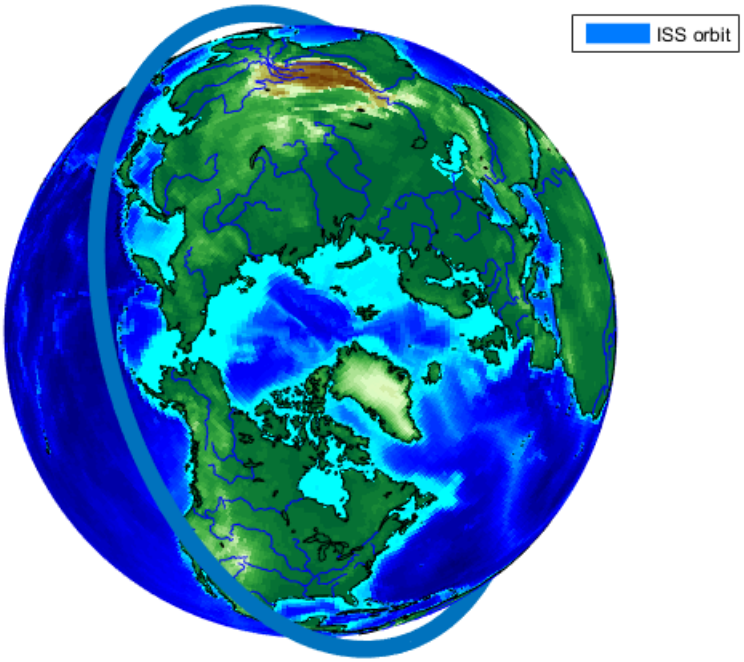


Figure 6: Orbit propagation of ISS in LEO.

2 Problem Set 2 - Conservation Laws, Ellipsoids of Rotational Motion, Euler equations

The governing equation for the angular rates of a spacecraft is derived using the angular momentum version of Newton's second law, $M = \dot{H}$, where H is the angular momentum. Below is the expanded version, as a function of the moment of inertia, I, and the angular velocity, ω. It is assumed that the moment of inertia does not vary through the flight.

$$\mathbf{I} \cdot \dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times (\mathbf{I} \cdot \boldsymbol{\omega}) = \mathbf{M}.$$

The explicit relation between $\dot{\omega}$ and ω is shown below. The moments, M_i are modeled as perturbations. Notice that only the diagonal elements of moment of inertia are present. That's because we perform our analysis in the principle frame, in which all off-diagonal elements of the I matrix are zero. This is the power of maintaining a transformation to the principle axis.

$$\begin{aligned} I_1\dot{\omega}_1 + (I_3 - I_2)\omega_2\omega_3 &= M_1 \\ I_2\dot{\omega}_2 + (I_1 - I_3)\omega_3\omega_1 &= M_2 \\ I_3\dot{\omega}_3 + (I_2 - I_1)\omega_1\omega_2 &= M_3 \end{aligned}$$

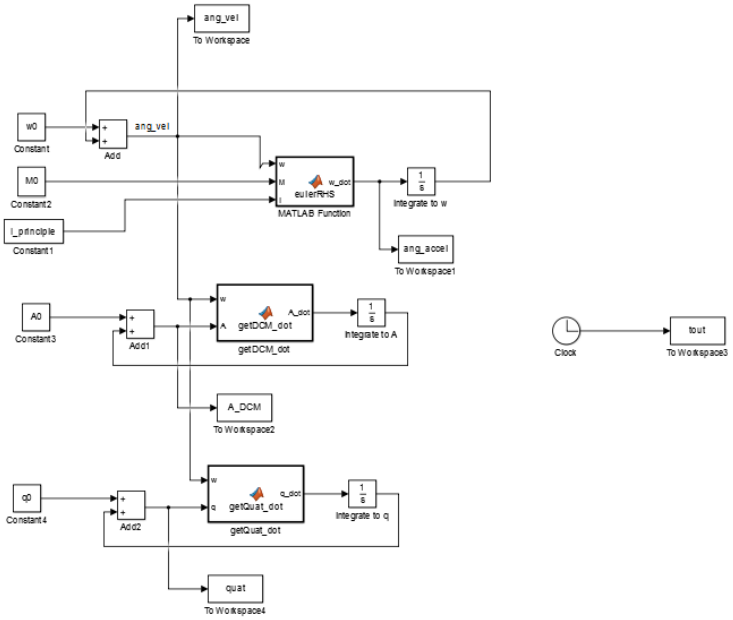


Figure 7: Simulink model for the Euler equations.

Fig. 7 shows how the Euler equations are implemented in the simulink model. All calculations are done in the principle frame, so post-processing requires transformation to inertial frame.

When validating the Euler equations, one can utilize the conservation of energy and angular momentum to ensure the integration is performed correctly. Below shows the energy and momentum ellipsoid equations. Motion is only physically possible when two closed lines form at the intersection of the two ellipsoids.

$$2T = \omega_y^2 I_y + \omega_z^2 I_z + \omega_x^2 I_x \Rightarrow$$
$$\frac{(\omega_y)^2}{2T/I_y} + \frac{(\omega_z)^2}{2T/I_z} + \frac{(\omega_x)^2}{2T/I_x} = \frac{c_y^2}{b^2} + \frac{c_z^2}{c^2} + \frac{c_x^2}{a^2} = 1$$

$$L^2 = \omega_y^2 I_y^2 + \omega_z^2 I_z^2 + \omega_x^2 I_x^2 \Rightarrow$$
$$\frac{(\omega_y)^2}{L^2/I_y^2} + \frac{(\omega_z)^2}{L^2/I_z^2} + \frac{(\omega_x)^2}{L^2/I_x^2} = \frac{c_y^2}{b^2} + \frac{c_z^2}{c^2} + \frac{c_x^2}{a^2} = 1$$

2.1 Rotation about arbitary direction

For validation, the satellite is initiated with a random ang. vel. and allowed to propagate. After running the simulation, the output is post-processed to validate the results. A random starting velocity is generated and the euler equations propagate naturally. With no external

torques, energy is conserved as shown in Fig. 8. The noise is due to numerical error from the simulation.

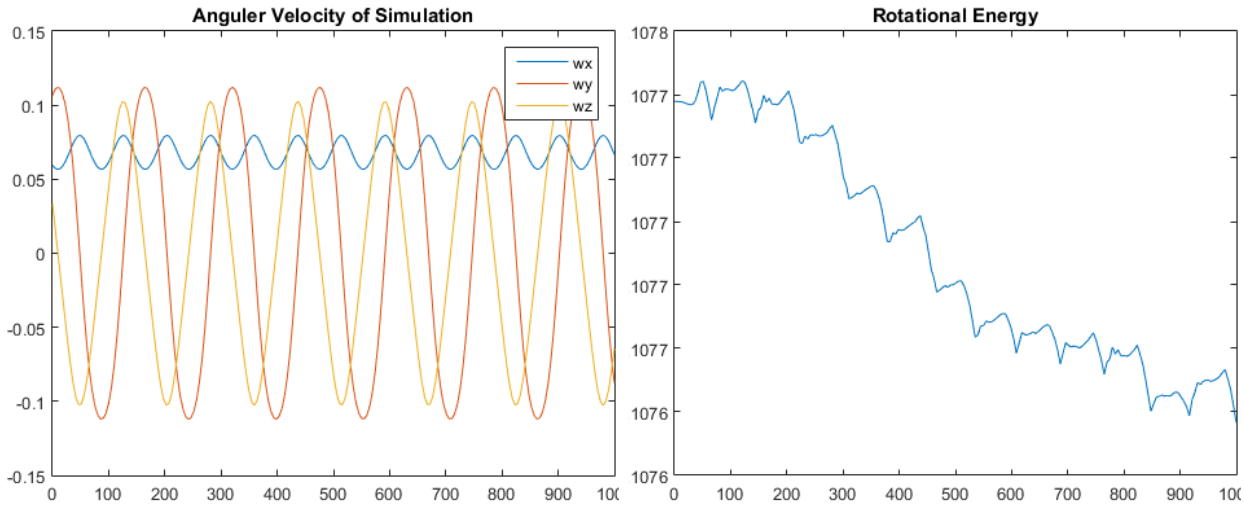


Figure 8: Ang vel. and rotational energy.

In Fig. 9 the momentum and energy ellipsoids are plotted for this simulation. Highlighted in orange and blue is the intersection.

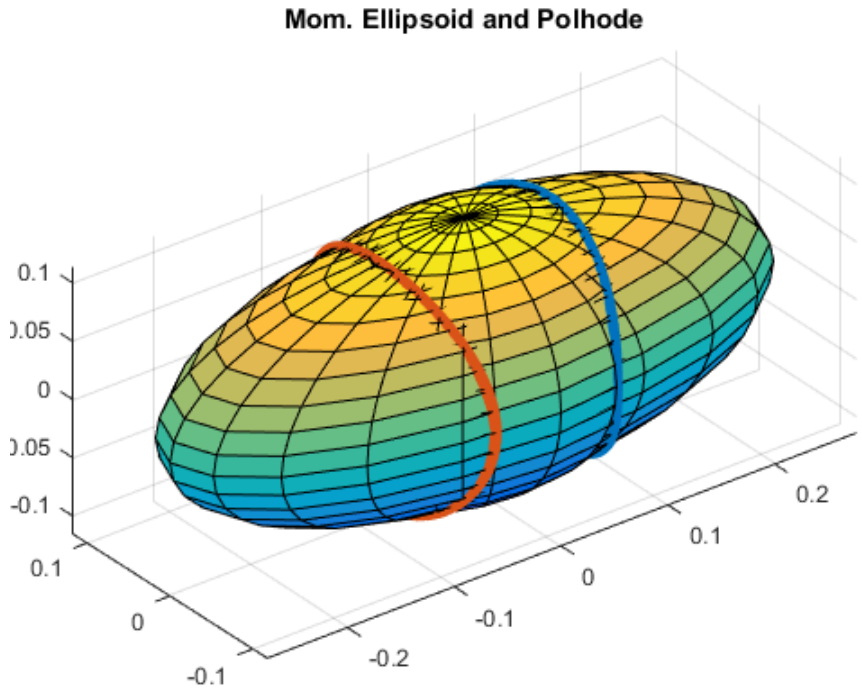


Figure 9: Momentum and energy ellipsoids. The intersection is physically realizable.

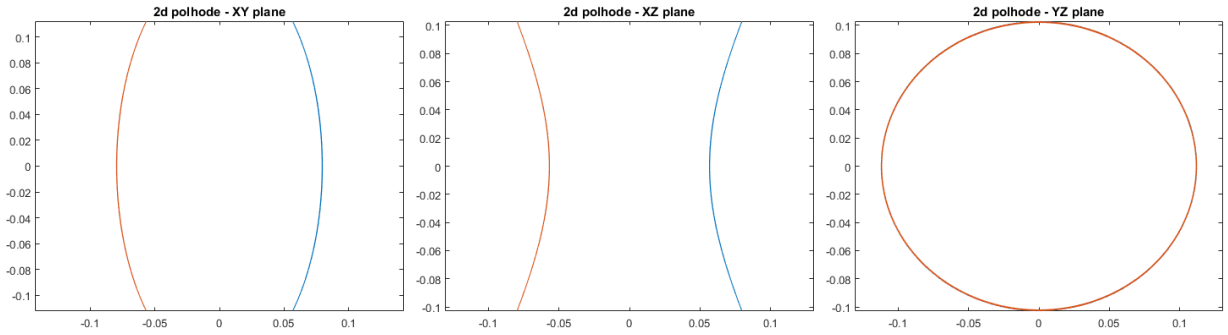


Figure 10: Polhode in each plane.

When the two ellipsoids are set equal to each-other, the resulting shape is described by eq. 4-6. This produces an ellipse in the XY & YZ planes, and a hyperbola in the XZ plane, as confirmed in Fig. 10.

$$(I_y - I_x)I_y w_y^2 + (I_z - I_x)I_z w_z^2 = L^2 - 2TI_x \tag{4}$$

$$(I_z - I_y)I_x w_x^2 + (I_z - I_y)I_z w_z^2 = L^2 - 2TI_y \tag{5}$$

$$(I_x - I_z)I_x w_x^2 + (I_y - I_z)I_y w_y^2 = L^2 - 2TI_z \tag{6}$$

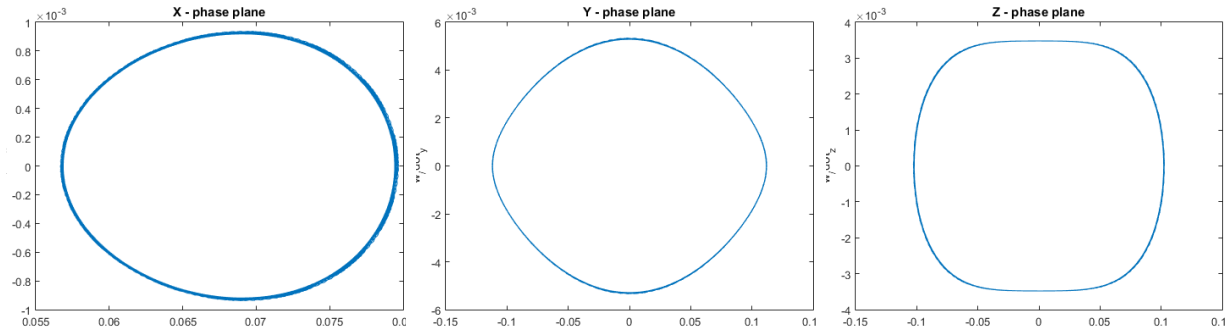


Figure 11: Phase plane in each direction.

Along with analyzing the energy and momentum, we can directly plot the phase of the angular velocity by leaving the Euler equations in differential form and setting $\dot{\omega} = fcn(\omega)$. The phase plots (Fig. 11) have a near-sinusoidal solution, but with perturbations. As shown, this means shapes which are close to ellipses, but not quite.

2.2 Rotation along principle direction

In the following section, conservation analysis is repeated but the initial velocity is constrained to be in one of the three principle directions. This is in fact an equilibrium state

when no external moments are present, as having any two components of ω set to zero reduces the Euler equations to $\dot{\omega} = 0$.

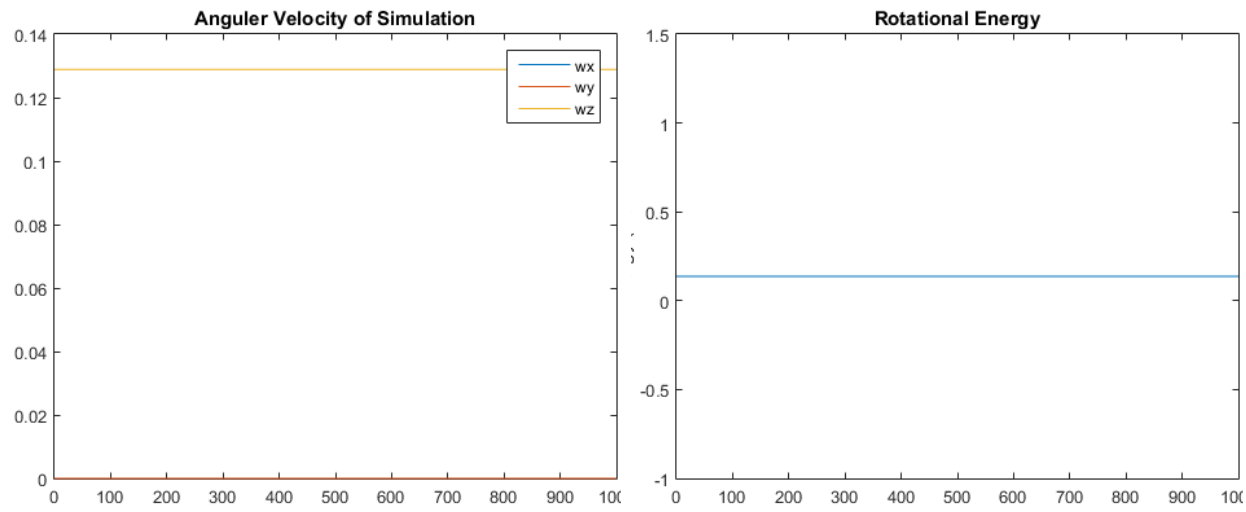


Figure 12: Ang. vel. and energy when rotating about principle axes.

The result is rather boring, as it should be. In the absence of extreme numerical error, nothing changes and the satellite's attitude remains in equilibrium. Rotational energy remains constant as expected. Below is the polhode again, but when aligned in principle axes, the two ellipsoids are exactly tangent.

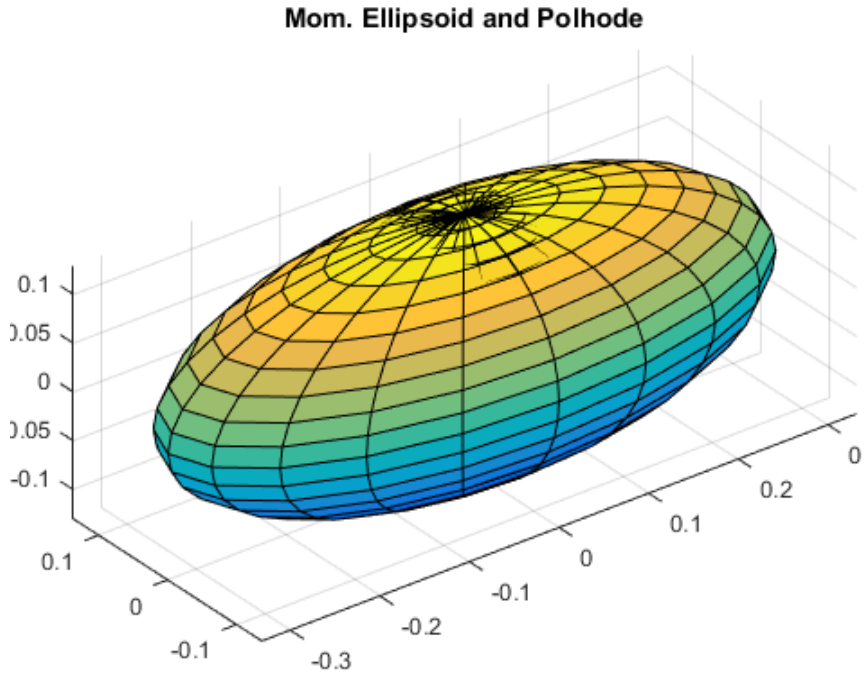


Figure 13: Ellipsoids and polhode in principle axes.

2.3 Comparison to analytic solution for axi-symmetric satellites

Next, the moment of inertia is constrained such that $I_x = I_y$. In this case, the rotation about z-axis is constant, and the Euler equations can be simplified.

$$\begin{aligned} \dot{\omega}_x + \lambda\omega_y &= 0 \\ \dot{\omega}_y - \lambda\omega_x &= 0 \end{aligned}, \lambda = \frac{(I_z - I_x)}{I_x}\bar{\omega}_z$$

Figure 14: Reduced Euler equations.

Using the simpler differential equation, the partial-analytic is found using a Matlab ode solver, ode113 (Fig. 15). By solving the system using both this solver and the full Euler equations, the numerical accuracy can be examined.

```
%axially symmetric - assumes Ix == Iy
if (I(1,1) - I(2,2)) < 1e-3
  lambda = (I(3,3) - I(1,1)) * w0(3) / I(1,1);
  odefun = @(t,y) [-lambda*y(2), lambda*y(1)]'; %x-y euler eq soln
  [~, y] = ode113( odefun ,tspan, w0(1:2));

end
end
```

Figure 15: Code for ode113.

Fig. 16 shows the error over time for each ang. vel. Over time, numerical errors build up and the difference between true ang. vel. and numeric ang. vel. grows. Because of the nature of numerical error, there is no way to recover from this. Proper setting of simulation time-step and solver type can reduce the impact it has on simulations.

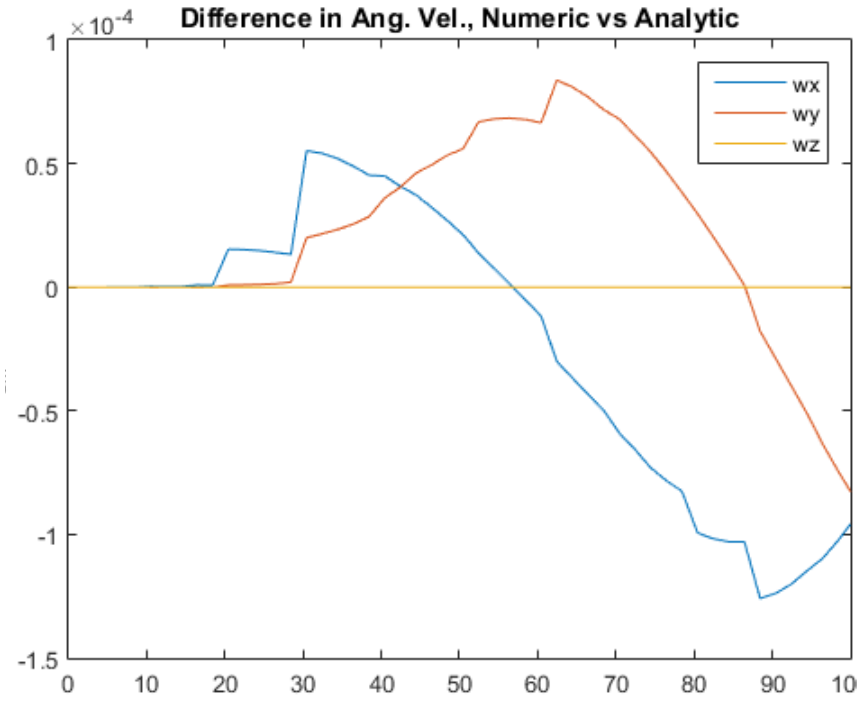


Figure 16: Numerical accuracy of the Euler equation propagator.

3 Problem Set 3 - Kinematic Equations & Stability

In this section, the kinematic equations which govern the attitude propagation are explored, as well as the stability of satellites when operating in different states.

3.1 Direction Cosine Matrix

The direction cosine matrix (DCM) allows vectors to be transformed from one coordinate system to another. Specifically, it is a generalized 3-dimensional rotation which does not change the scale of the input vector. Below shows how a vector is transformed, along with how the DCM is generated.

$$\vec{v}_{xyz} = \vec{A}\vec{v}_{123} \;\; ; \;\; \vec{v}_{123} = \vec{A}^t\vec{v}_{xyz} \;\; ; \;\; \vec{A} = \begin{bmatrix} \vec{x}\cdot\vec{1} & \vec{x}\cdot\vec{2} & \vec{x}\cdot\vec{3} \\ \vec{y}\cdot\vec{1} & \vec{y}\cdot\vec{2} & \vec{y}\cdot\vec{3} \\ \vec{z}\cdot\vec{1} & \vec{z}\cdot\vec{2} & \vec{z}\cdot\vec{3} \end{bmatrix}$$

Attitude of an aircraft can be expressed using DCM's and as a simulation progresses, the DCM must be updated accordingly. The kinematic equations describe how an attitude

representation evolves in time. Below is the time-derivative of a DCM, A, as a function of the ang. vel. expressed in principle axes.

$$\frac{d\vec{A}}{dt} = \lim_{\Delta t \to 0} \frac{-\Delta t[\vec{\omega}\mathrm{x}]\vec{A}(t)}{\Delta t} = -[\vec{\omega}\mathrm{x}]\vec{A}(t)$$

The left side multiplication uses shorthand for a tensor which is expanded in Eq. 7.

$$[\omega x] = \begin{pmatrix} 0 & -\mathrm{w_z} & \mathrm{w_y} \\ \mathrm{w_z} & 0 & -\mathrm{w_x} \\ -\mathrm{w_y} & \mathrm{w_x} & 0 \end{pmatrix} \tag{7}$$

DCM's are powerful as there are no discontinuities nor singularities, they are easy to physically interpret, and they maintain a built-in redundancy. Indeed, a full 6 parameters are redundant. The downside however is they are far more computationally intensive, requiring a full matrix-to-matrix multiplication each time-step.

3.2 Quaternions

Quaternions solve the problem of attitude determination while being less dense as DCM's. Four numbers are required, and only a single redundant parameter remains. Quaternions also have no singularities but they can become discontinuous, because $-q = q$.

In any case, having at least two attitude parameterizations creates a very robust system that allows checks to be implemented to ensure correct attitude determination.

The quaternion evolves as a function of ang. vel. expressed in principle axis as well, shown below.

$$\frac{d\vec{q}}{dt} = \lim_{\Delta t \to 0} \frac{\left(\vec{I} + \frac{1}{2}\vec{\Omega}\Delta t\right)\vec{q}(t) - \vec{q}(t)}{\Delta t} = \frac{1}{2}\vec{\Omega}\vec{q}(t)$$

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

3.3 Analysis of momentum in inertial frame.

For this test, a randomized initial ang. vel. was generated.

Now that the spacecraft's attitude is known, the angular momentum can be transformed into an intertial frame and analyzed. With no external moments, the ang. mom. vector remains constant, as shown in **??**.

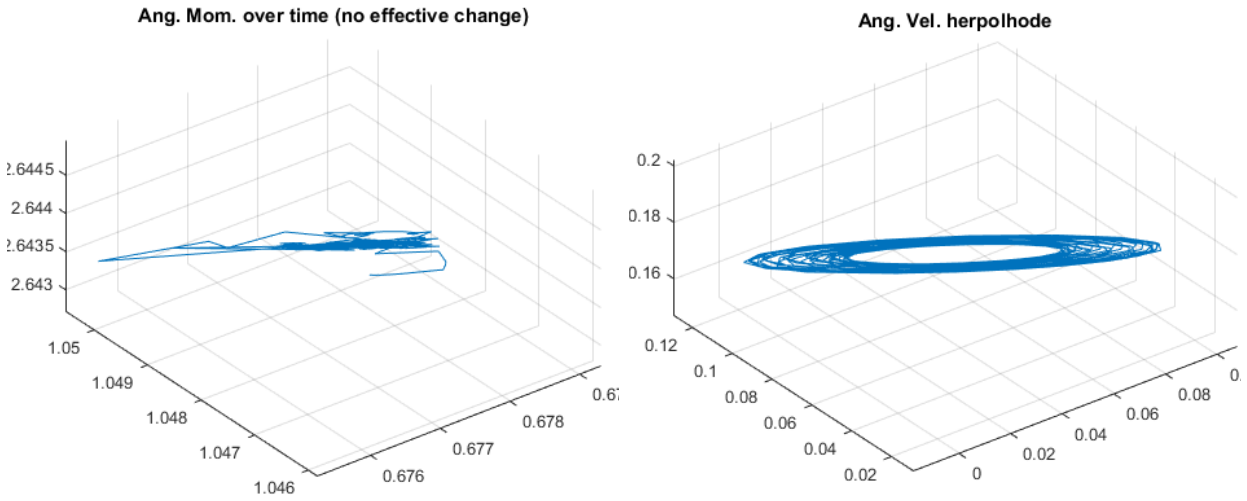


Figure 17: Ang. mom. vector and ang. vel. herpolhode.

When external moments are zero and the moment of inertia is diagonal, the resultant angular velocity herpolhode is contained within a plane perpendicular to the angular momentum vector. Fig. **??** demonstrates this.

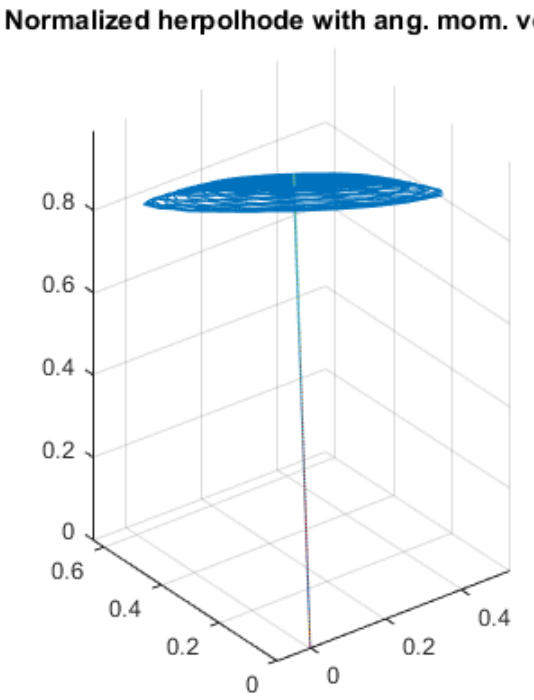


Figure 18: Normalized herpholhode relative to angular momentum vector.

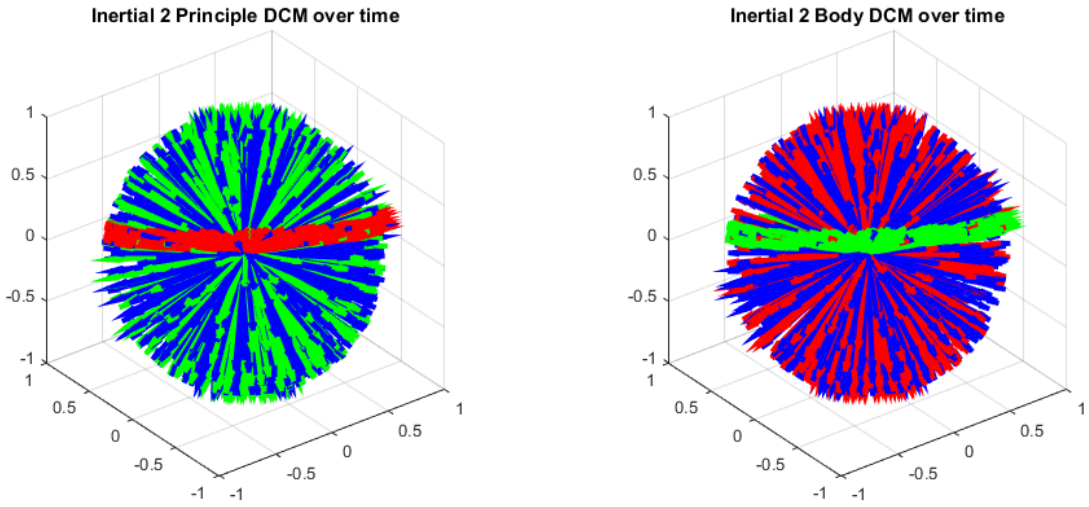


Figure 19: Ang. vel. when the x-axis is non-zero.

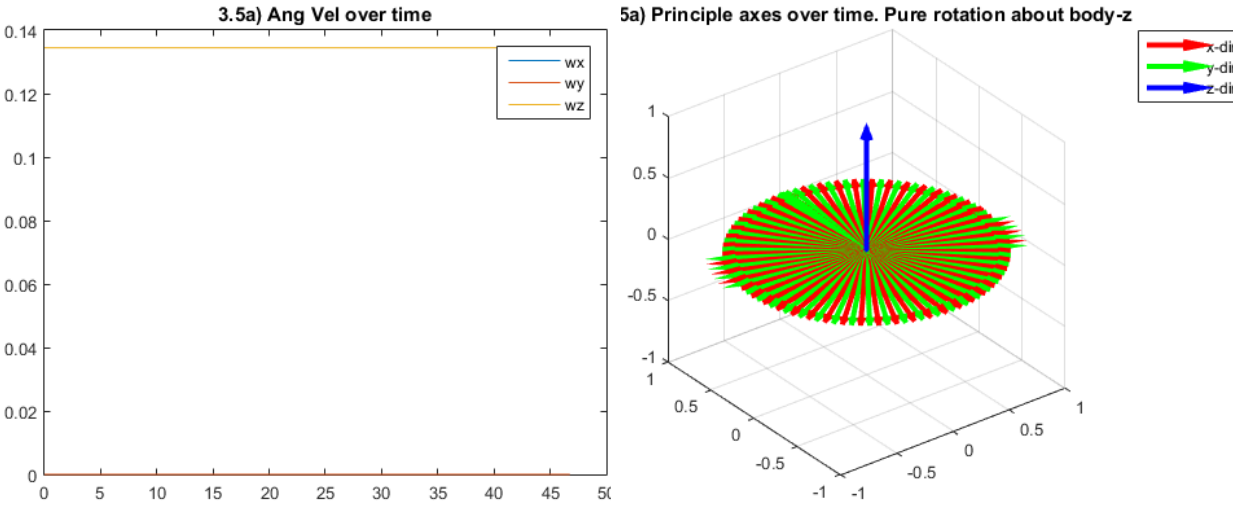


Figure 20: Ang. vel. when the x-axis is non-zero.

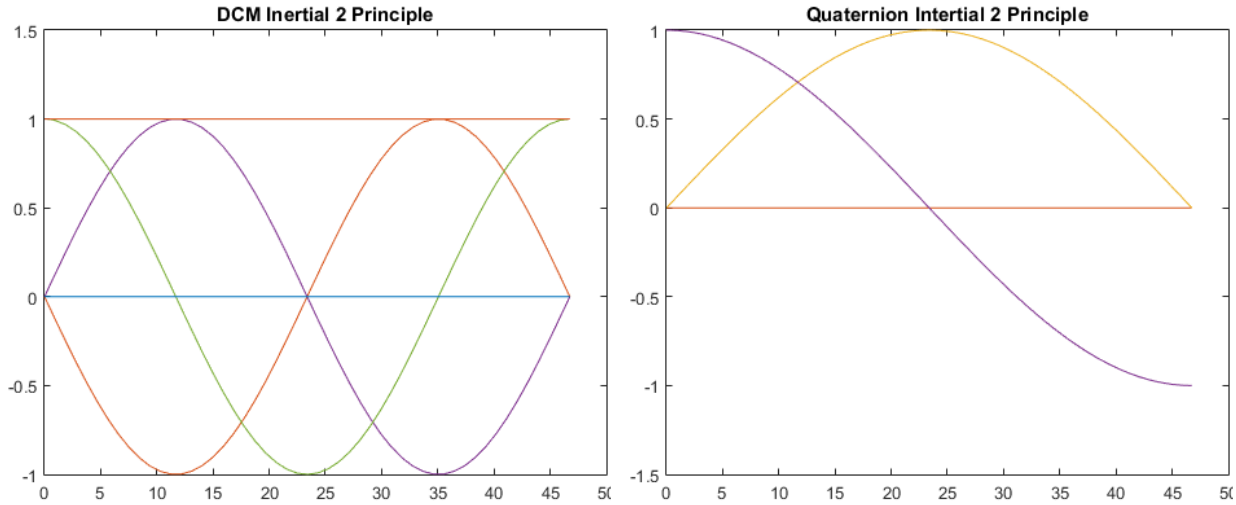


Figure 21: Ang. vel. when the x-axis is non-zero.

4 Problem Set 4 - Gravity Gradient and Rotational Stability

don't forget to take the workspace output from matlab

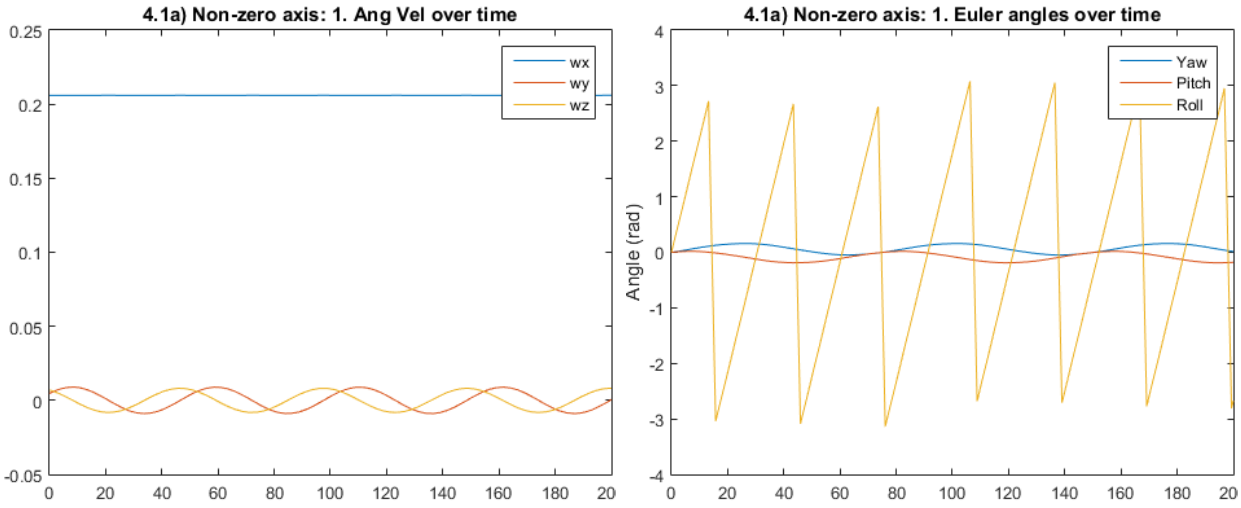


Figure 22: Ang. vel. when the x-axis is non-zero.

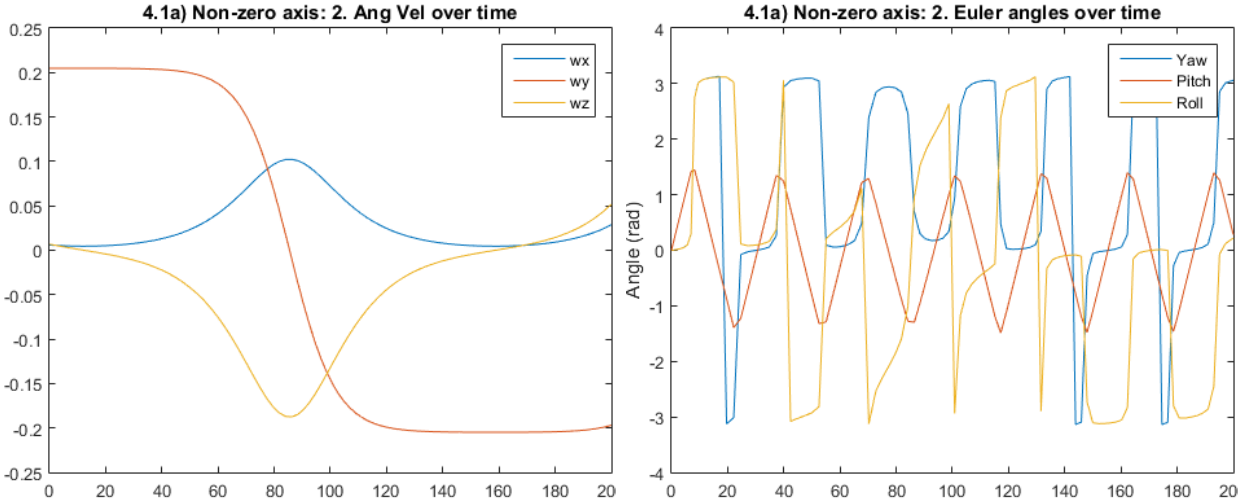


Figure 23: Ang. vel. when the y-axis is non-zero.

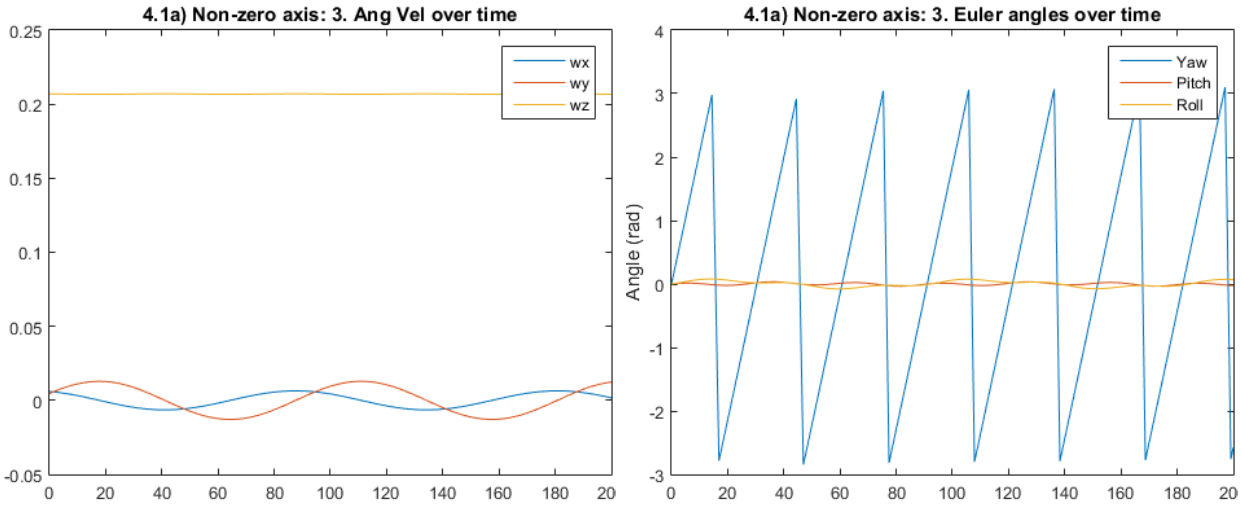


Figure 24: Ang. vel. when the z-axis is non-zero.

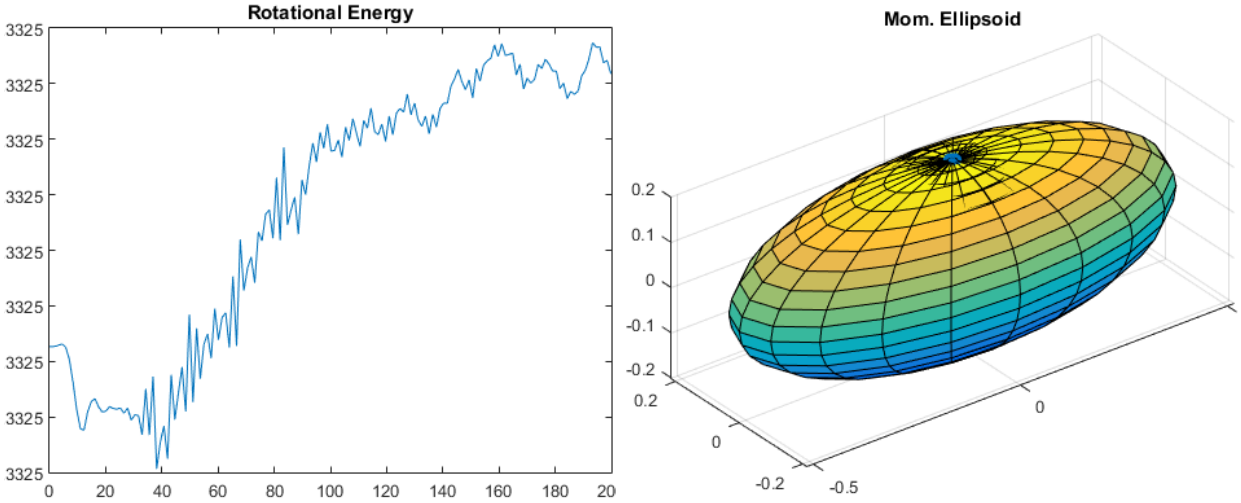


Figure 25: Ang. vel. when the x-axis is non-zero.

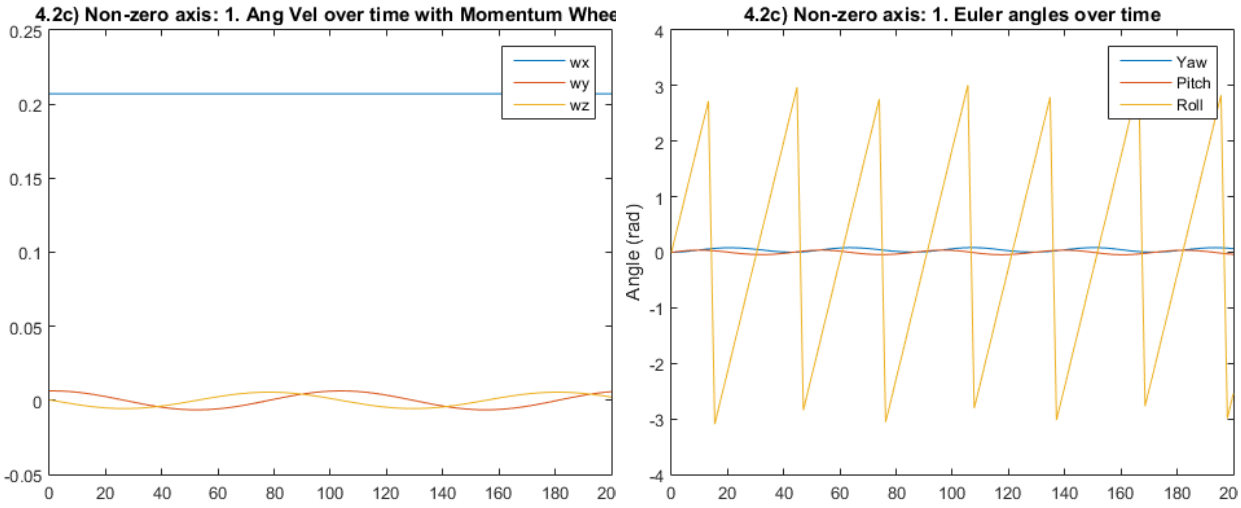


Figure 26: Ang. vel. when the x-axis is non-zero.

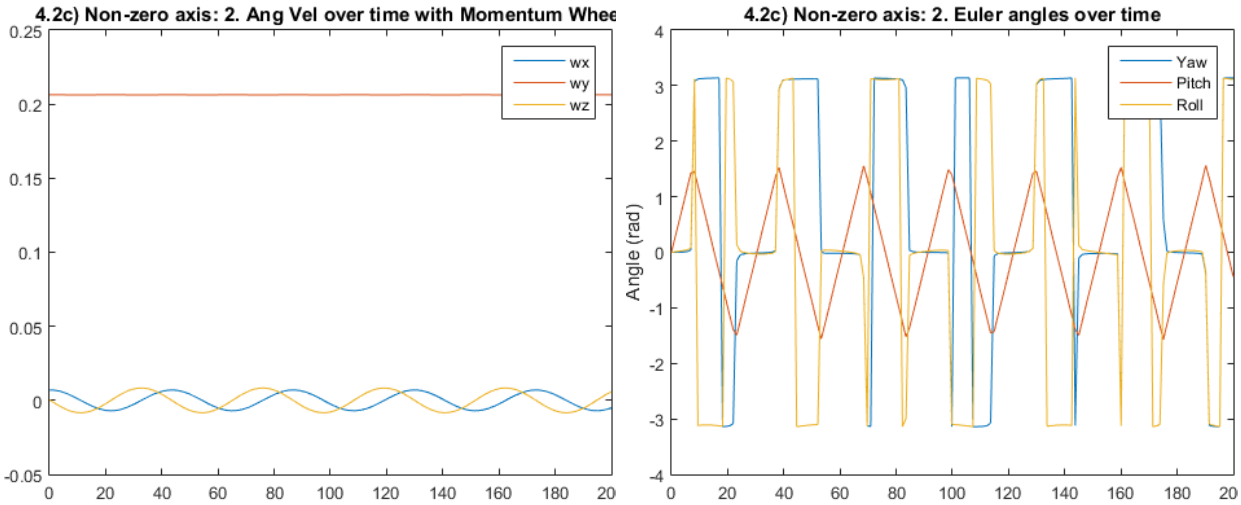


Figure 27: Ang. vel. when the x-axis is non-zero.

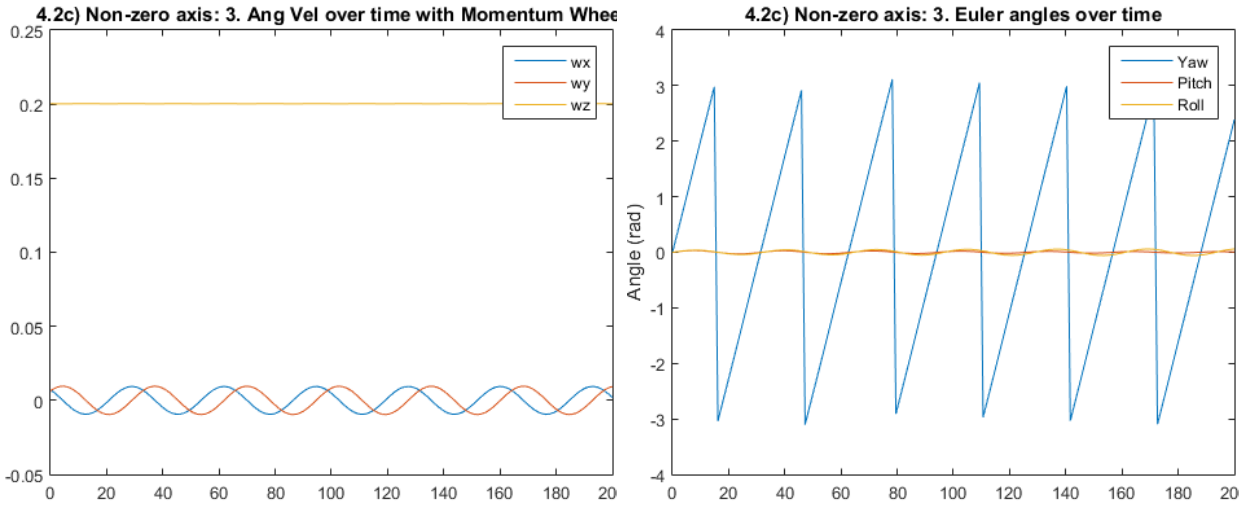


Figure 28: Ang. vel. when the x-axis is non-zero.

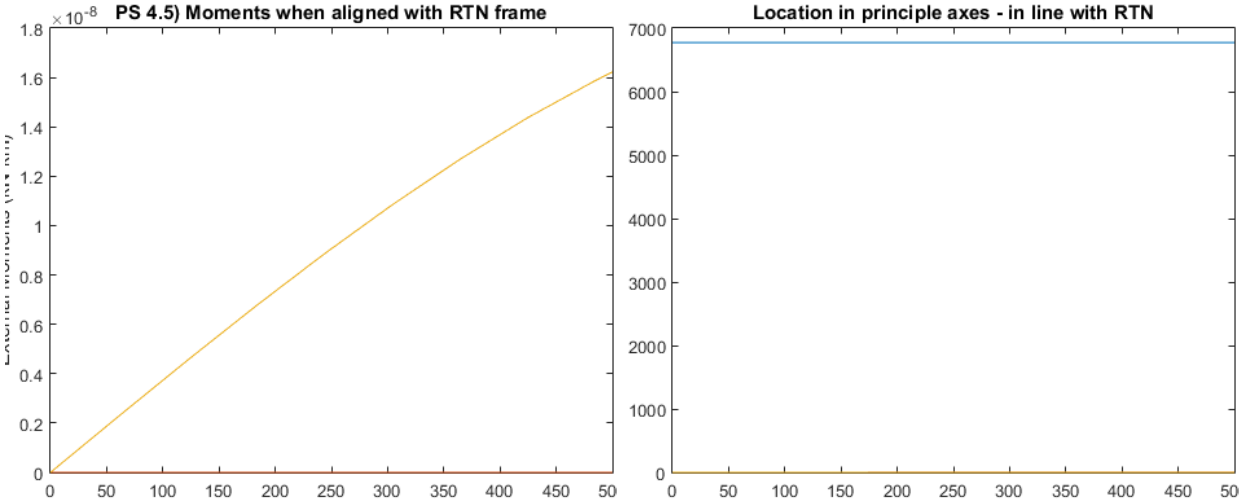


Figure 29: Ang. vel. when the x-axis is non-zero.

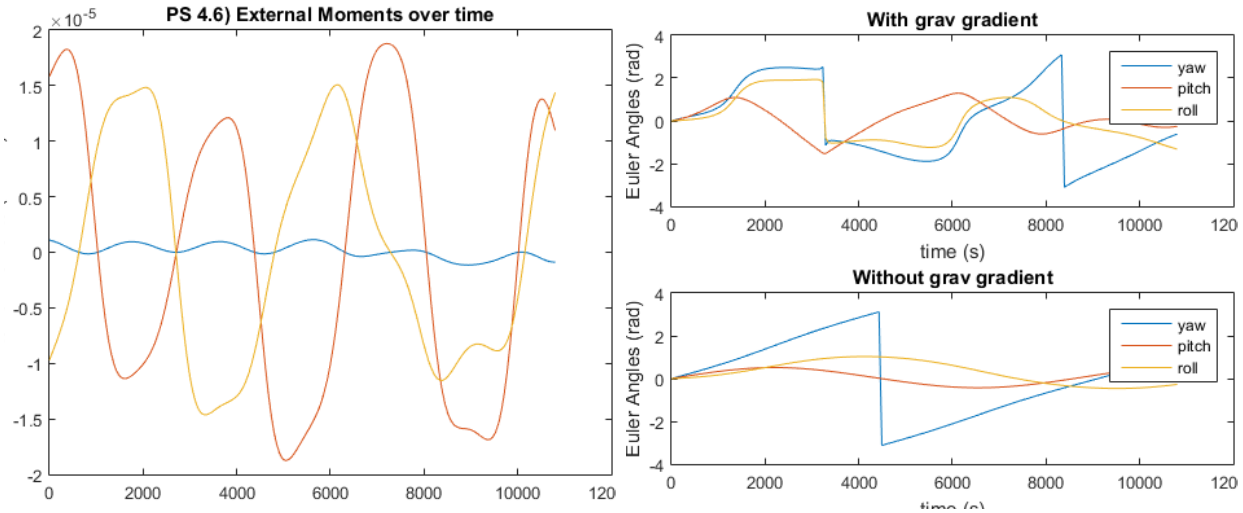


Figure 30: Ang. vel. when the x-axis is non-zero.

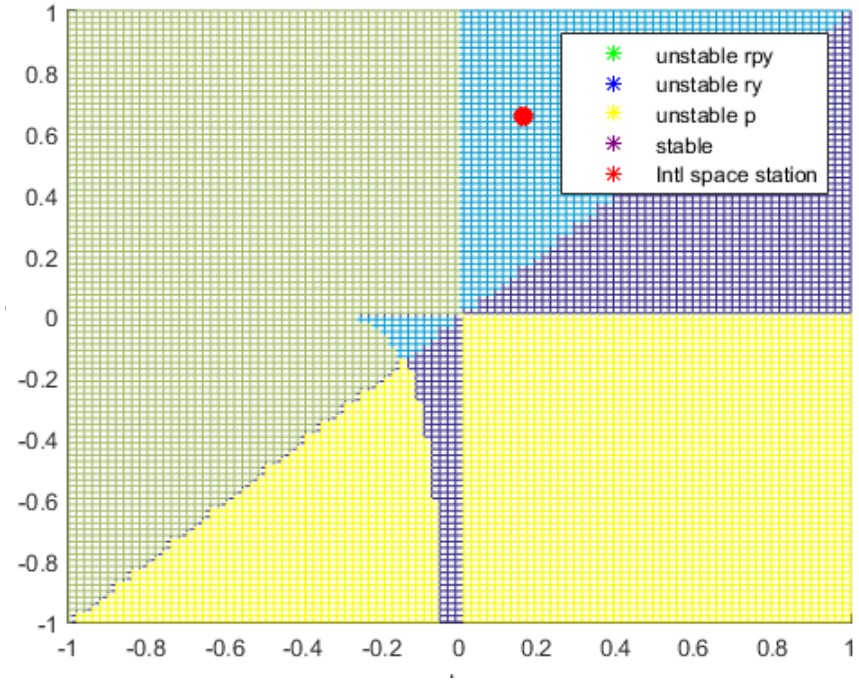


Figure 31: EKF performance.

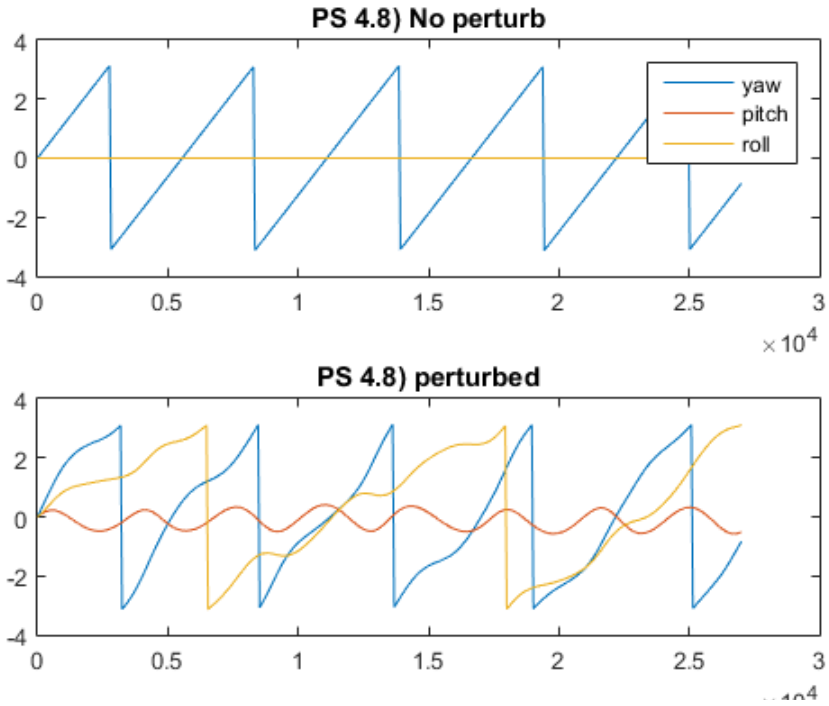


Figure 32: EKF performance.

5 Problem Set 5 - something

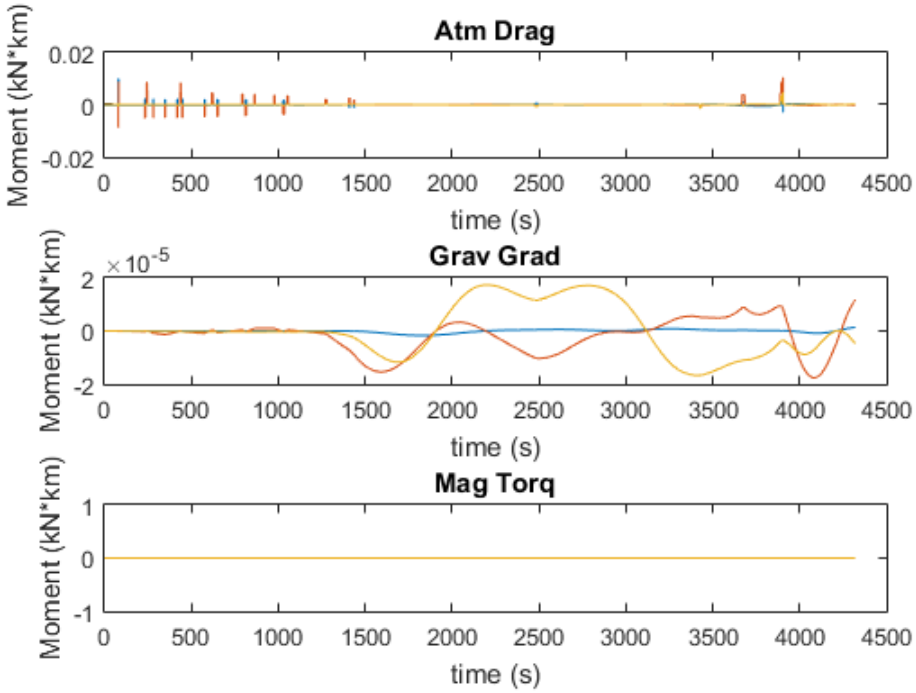


Figure 33: EKF performance.

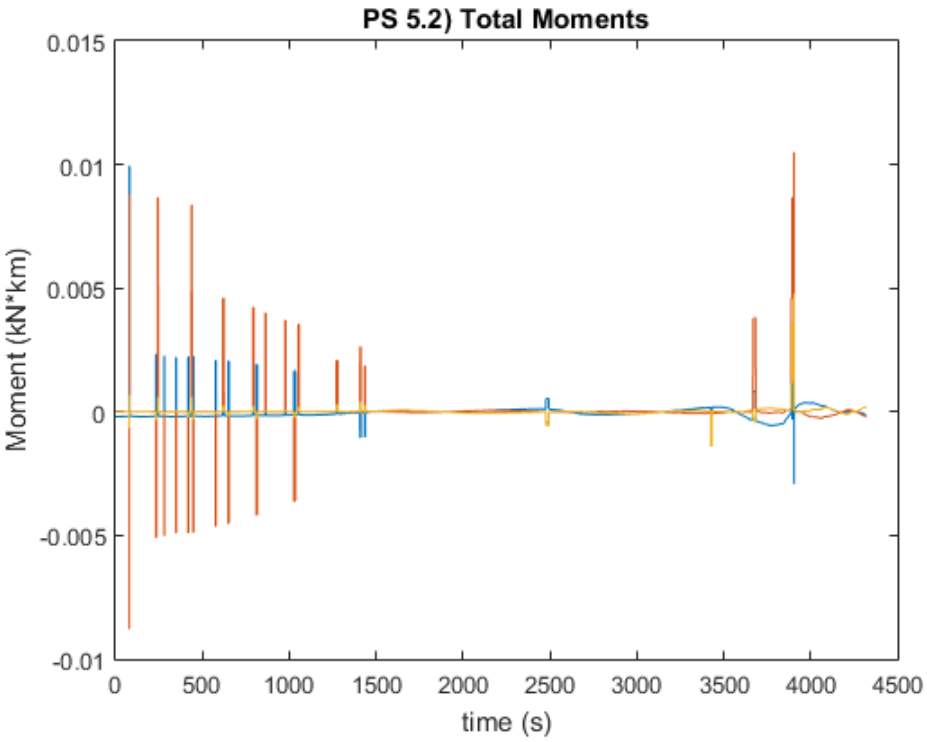


Figure 34: EKF performance.

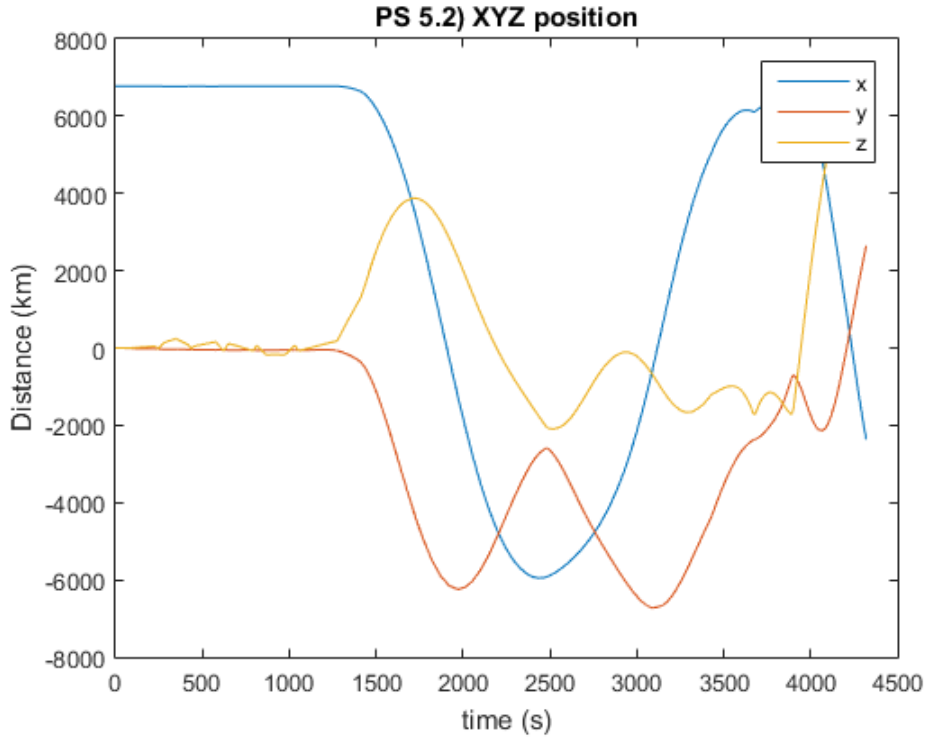


Figure 35: EKF performance.

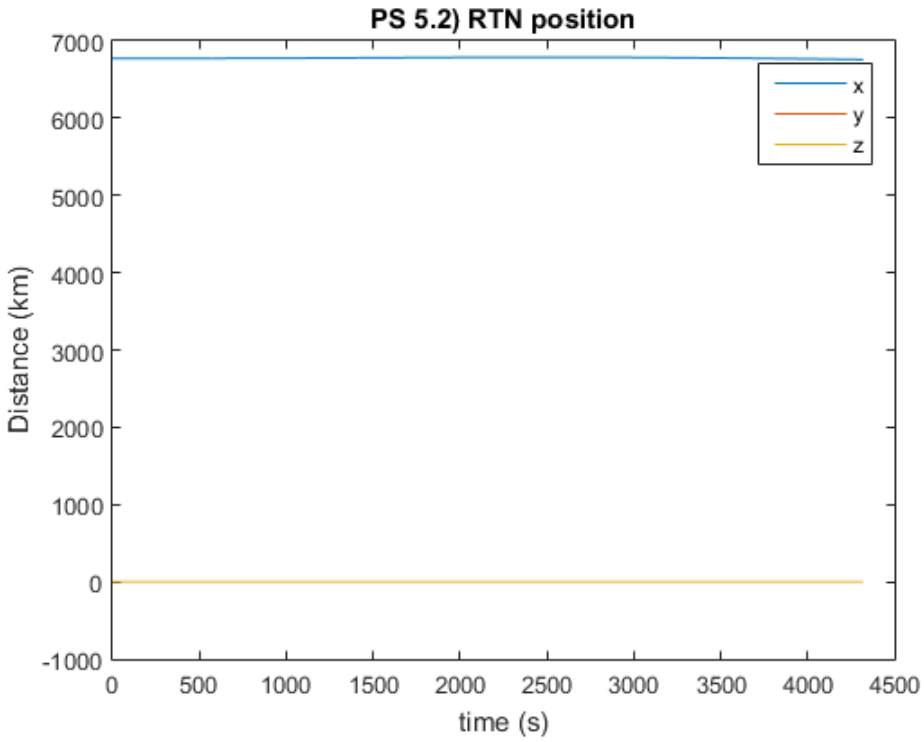


Figure 36: EKF performance.

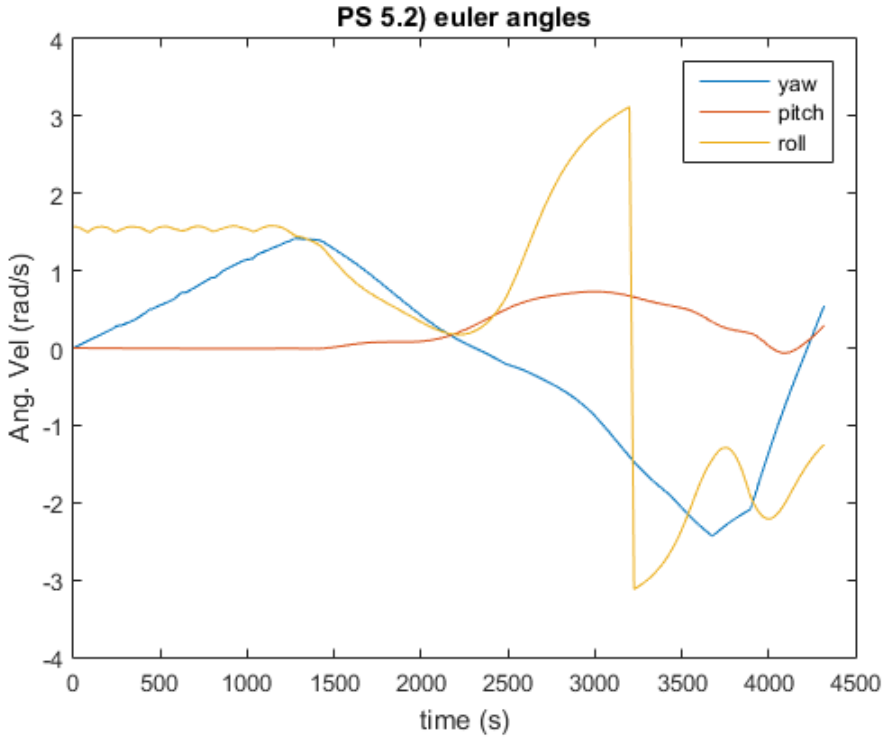


Figure 37: EKF performance.

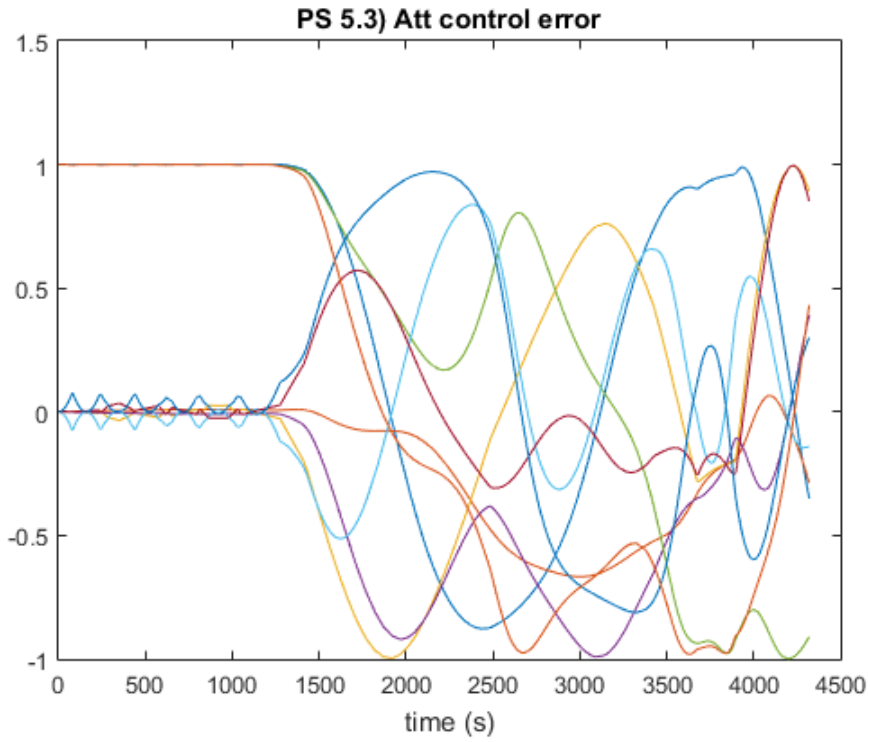


Figure 38: EKF performance.

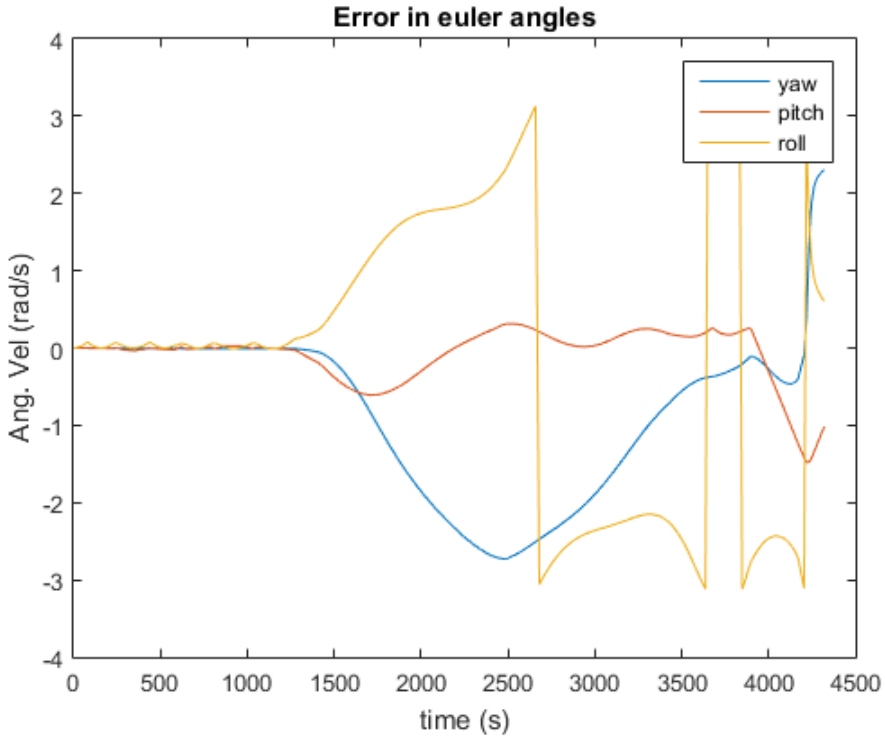


Figure 39: EKF performance.

6 Problem Set 6 - Attitude Determination with Sensor Errors

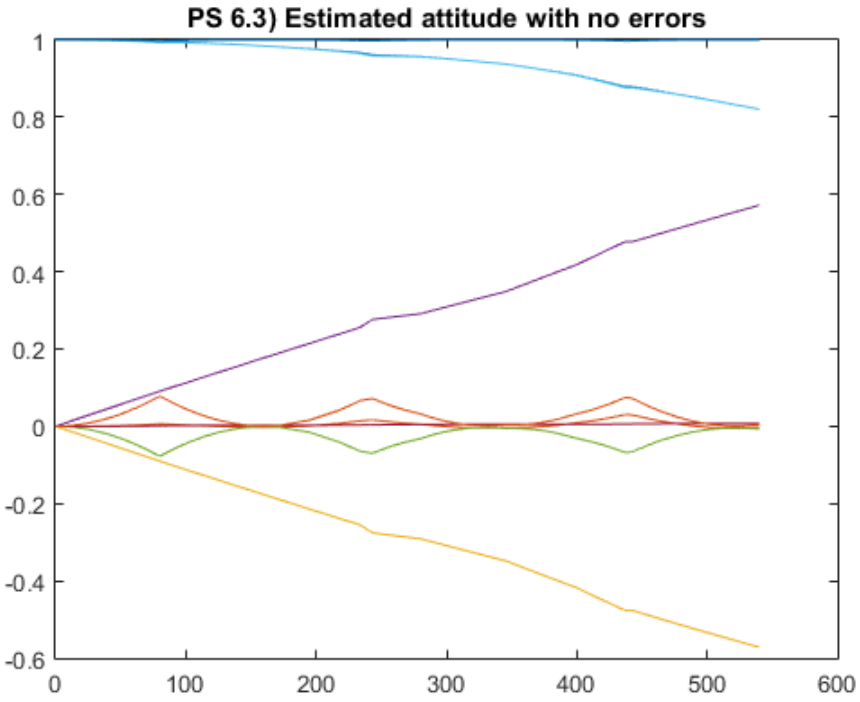


Figure 40: EKF performance.

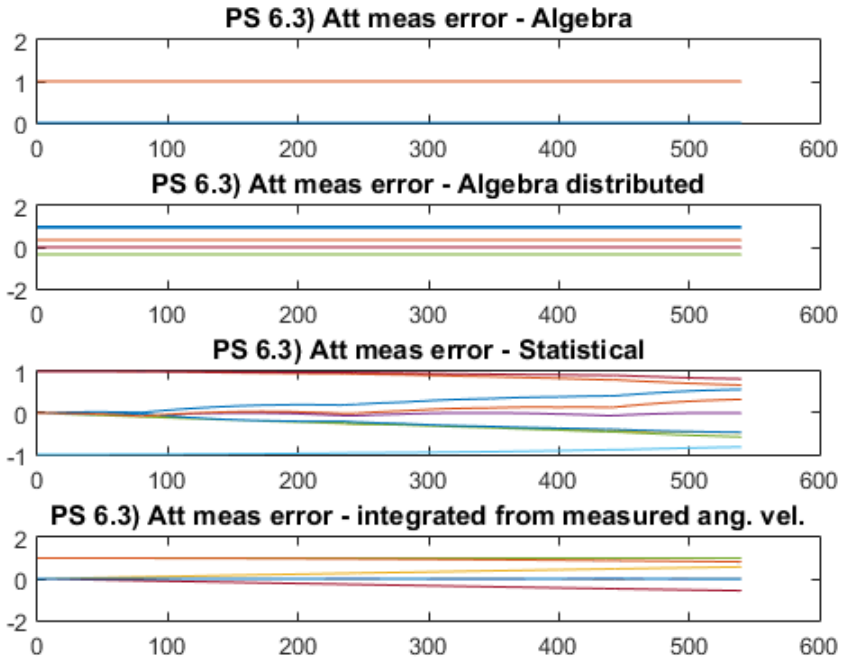


Figure 41: EKF performance.

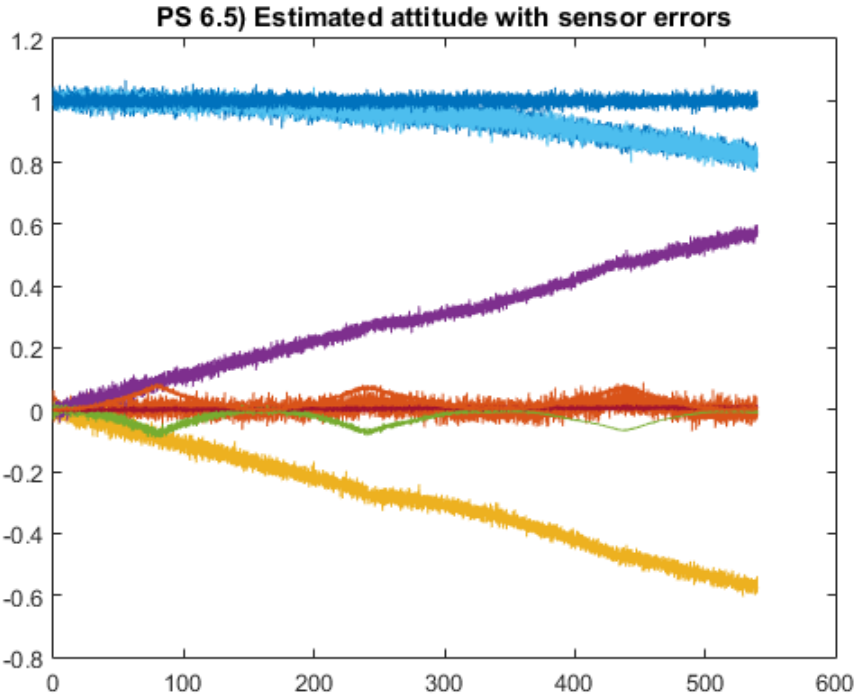


Figure 42: EKF performance.

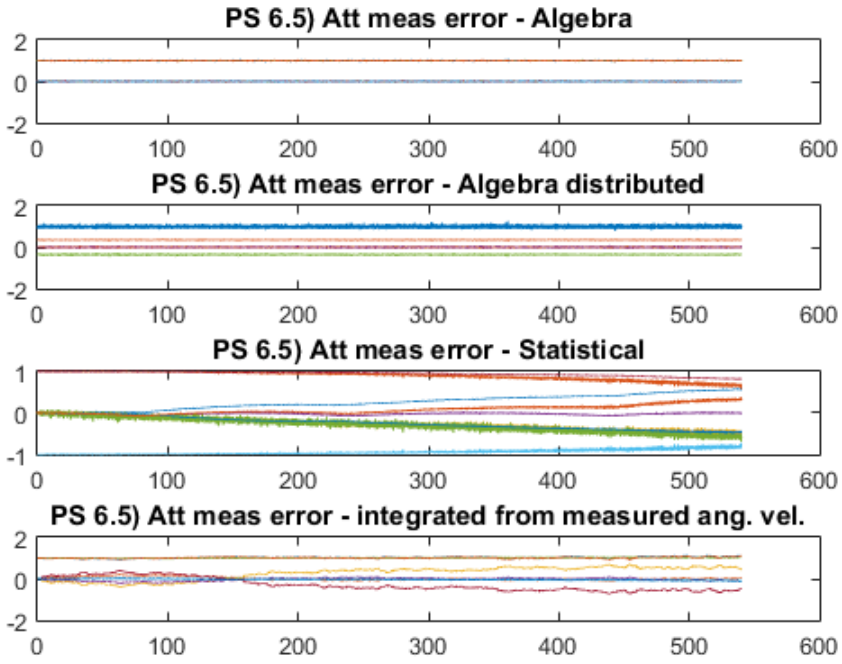


Figure 43: EKF performance.

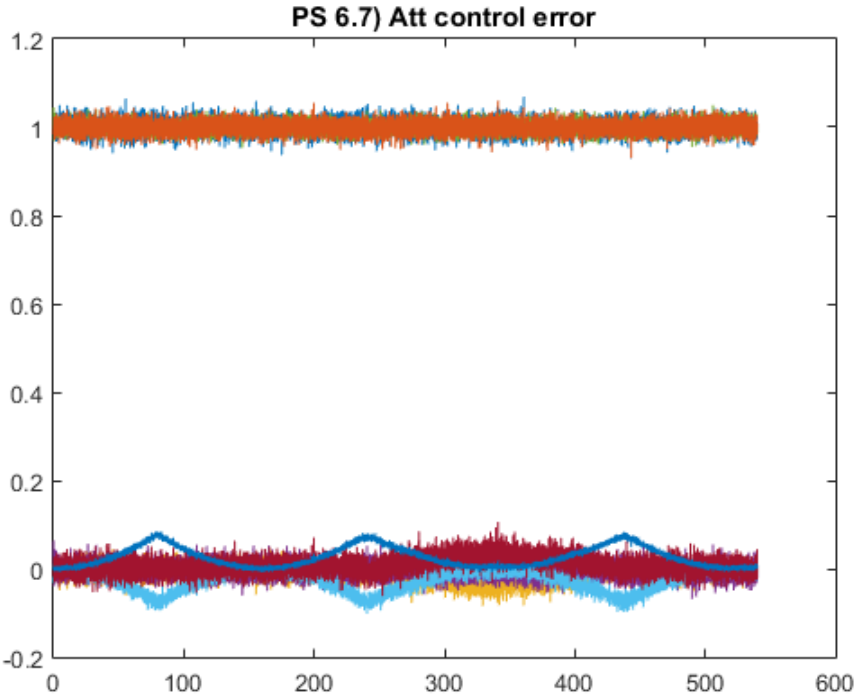


Figure 44: EKF performance.

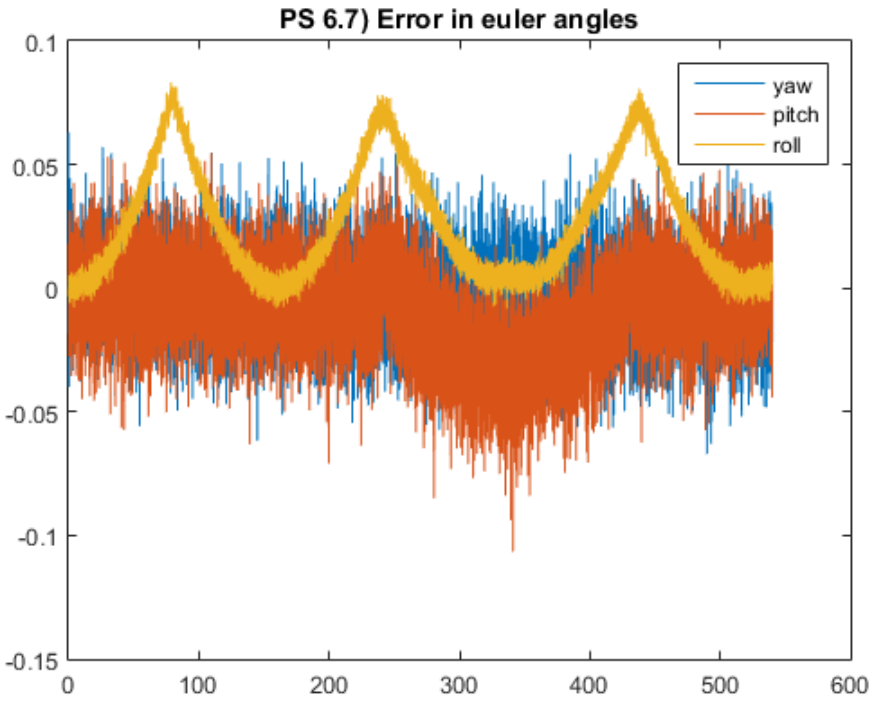


Figure 45: EKF performance.

7 Problem Set 7 - Implement Extended Kalman Filter

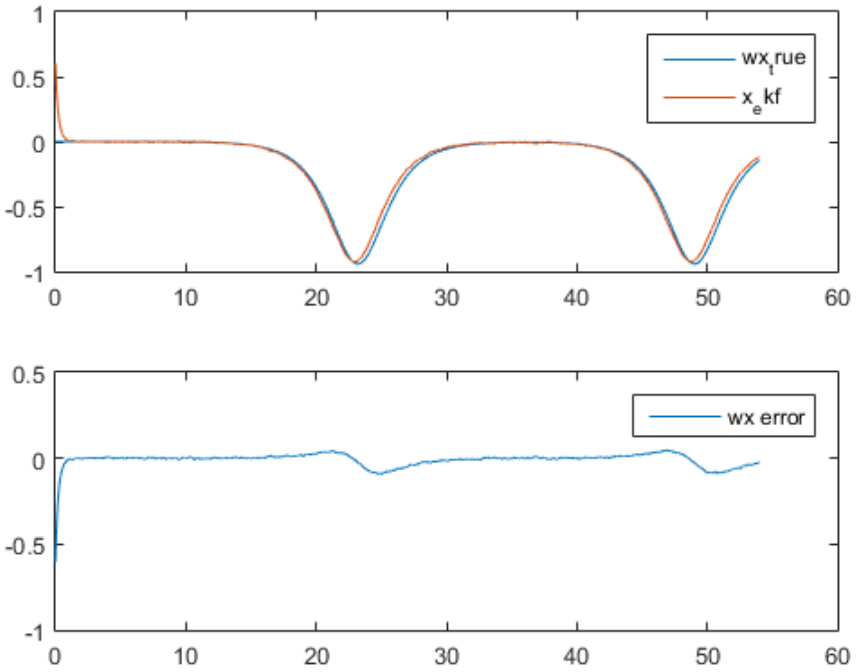


Figure 46: EKF performance.

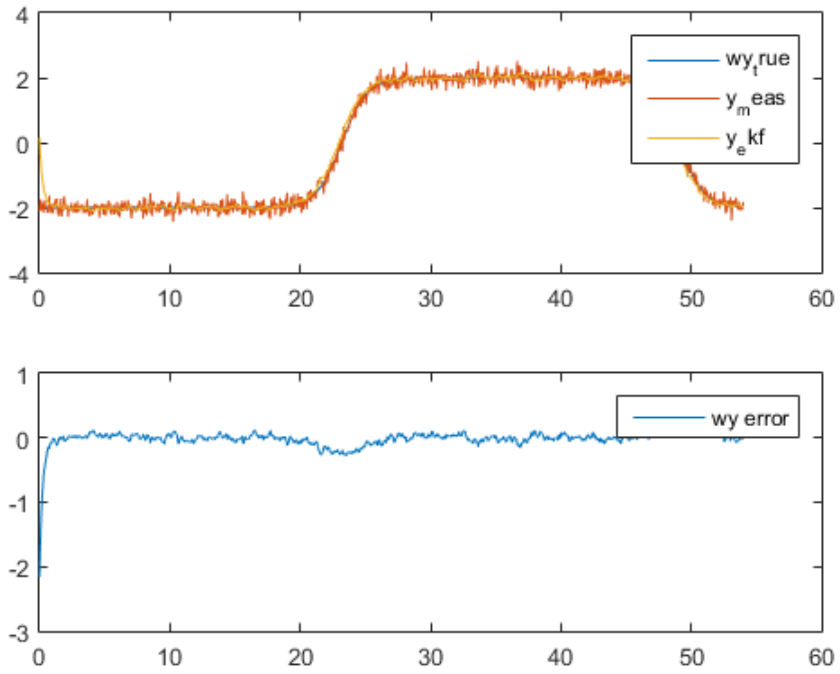


Figure 47: EKF performance.

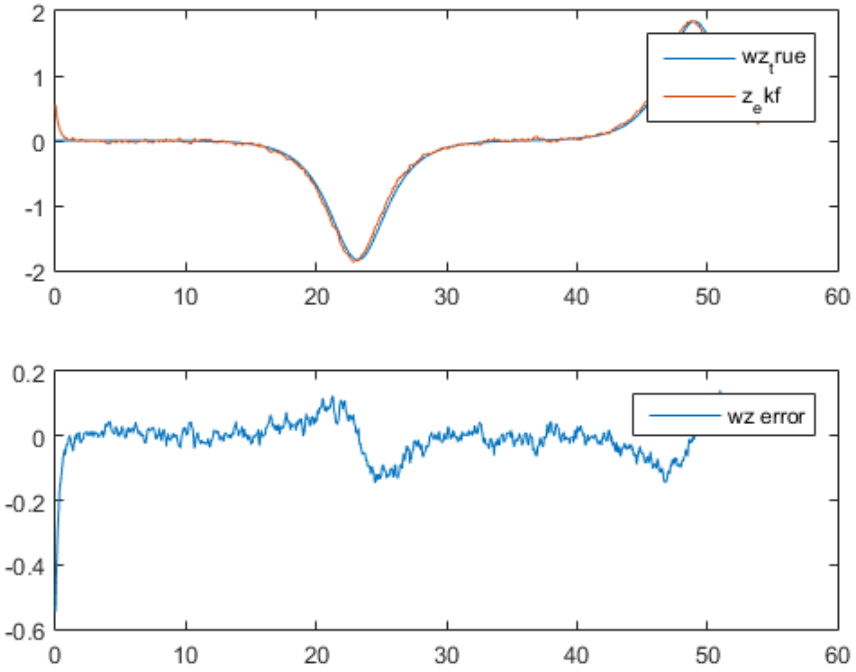


Figure 48: EKF performance.

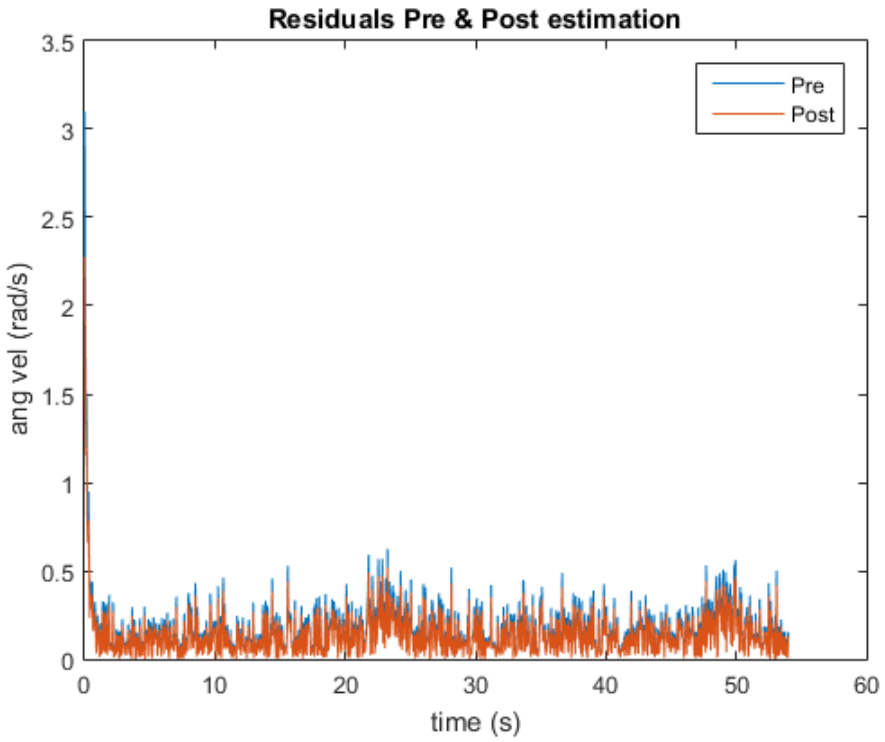


Figure 49: EKF performance.

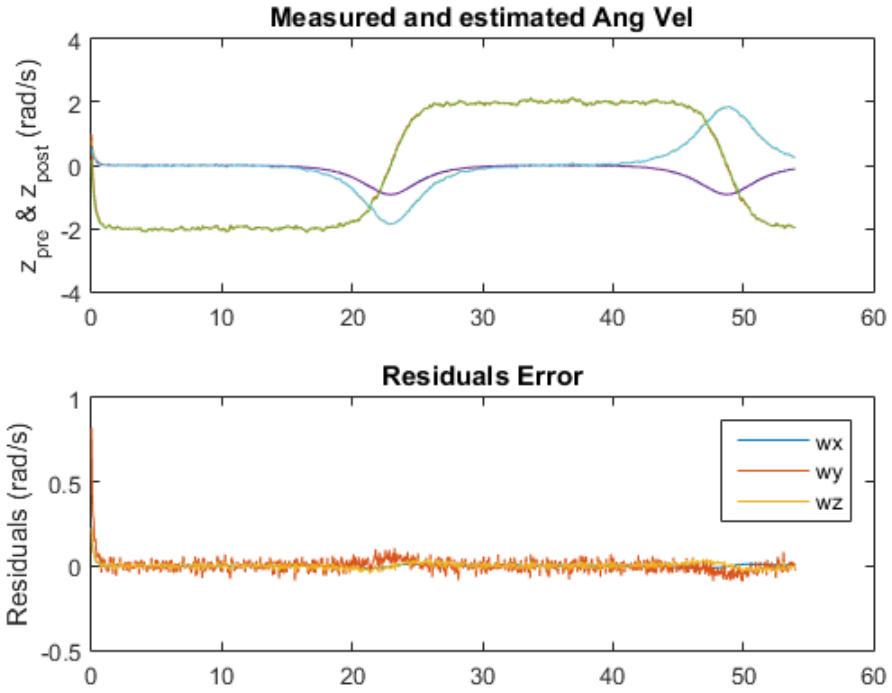


Figure 50: EKF performance.

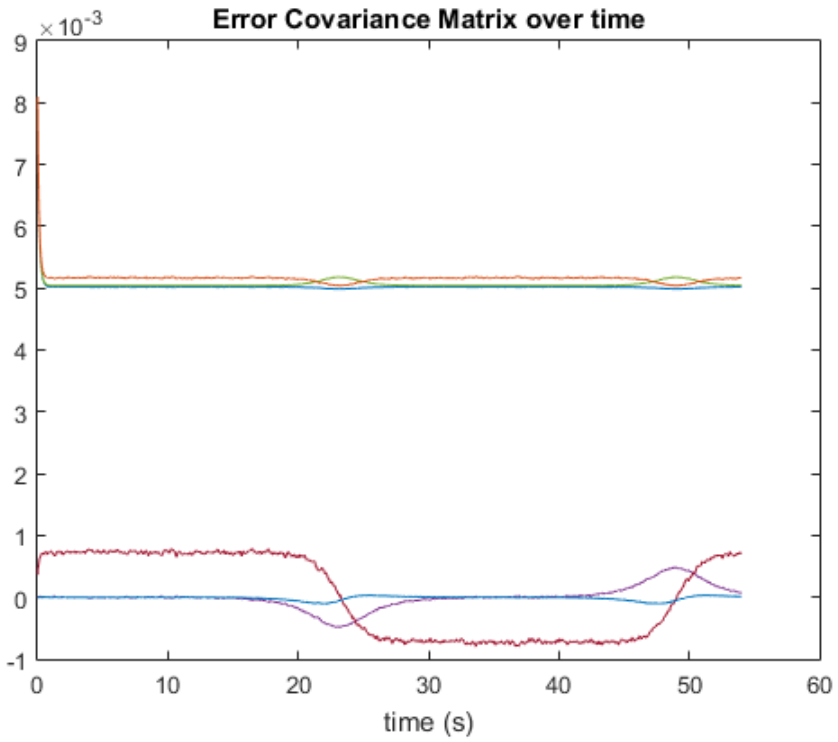


Figure 51: EKF performance.

8 Problem Set 8 - Implement Actuators and Controllers

The International Space Station uses 4 control-moment-gyros it uses for day-to-day station keeping. Each CMG can produce 4760 N*m*s, or 258 N*m, and spins at 691 rad/s. In reality, the ISS has all four aligned the in the same direction, but for simplicity I've decided to model three CMG's each aligned in a different body axis direction.

The control moment gyros operate by running at a fixed very high speed rotation, with large mass and radius. By gimbaling the CMG, the angular momentum vector is shifted which causes a reaction torque on the rest of the satellite due to the conservation of angular momentum.

$$I\dot{\omega} + \omega x I\omega = M + M_c;\ M_c = -A\dot{L_w} - \dot{A}L_w - \omega x A L_w \tag{8}$$

Because we have changed the dynamics of the vehicle by adding additional large rotating masses, the Euler equations must also change. Eq. 8 shows how the CMG's are taken into account, by grouping them together into a control moment term, which we are able to take advantage of when determining a control sequence.

Control of the aircraft is performed using two successive loops (Fig. 52). First, a high-

level objective attitude is created (example: follow the RTN frame) and passed to the attitude controller. A first controller generates commanded angular velocity, which is passed to the second controller. The output of the inner-loop are the desired moments that the CMG's must impart on the vehicle. Those are perturbed slightly by assuming imperfect actuators, and then passed into the environment simulator. The required CMG angles are computed from the control-loop output by solving for $\dot{A}$ and performing a 1st order integration.

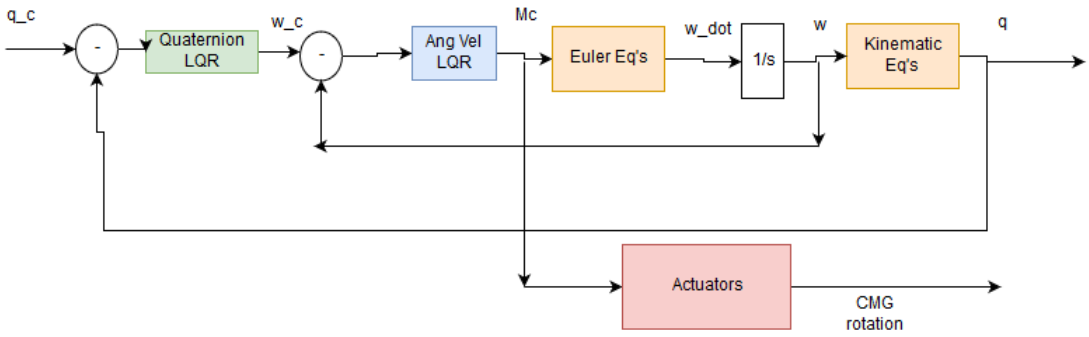


Figure 52: Block diagram for the controller used onboard the satellite. Two LQR's are utilized, one for attitude and the other for a lower-level ang. vel. controller.

The block diagram for the controller (Fig. 52) shows how the different controllers pass information to each-other and the environment. In order to obtain the controller gains, a linear-quadratic controller is implemented.

$$x_{k+1} = Ax_k + Bu_k \tag{9}$$

$$u = -Kx \tag{10}$$

The linear-quadratic controller, or LQR, is well-documented and used widely in the controls community. Eq. 9 shows the discrete state space model, where the next state is a linear function of the current state and controls. It's assumed a linear gain is applied to the state and used to control the system (Eq. 10). By setting a weight matrix for the state (Q), for the control (R), and for the combination state and control (N), an optimal gain matrix can be solved for which minimizes the cost of regulating the system (shown below). The matlab function $K = dlqr(A, B, Q, R)$ is used to obtain the gain matrix.

$$J = \boldsymbol{x}_N^T \boldsymbol{Q} \boldsymbol{x}_N + \sum_{k=0}^{N-1} \left(\boldsymbol{x}_k^T \boldsymbol{Q} \boldsymbol{x}_k + \boldsymbol{u}_k^T \boldsymbol{R} \boldsymbol{u}_k + 2 \boldsymbol{x}_k^T \boldsymbol{N} \boldsymbol{u}_k \right)$$

The plant matrix, A, was obtained by linearizing the euler and kinematic equations. Taylor series expansion (shown below) was used to obtain a first order approximation of the differential equations.

$$f(x) \approx f(a) + f'(a)(x-a)$$

For attitude, $\dot{q} = 1/2\Omega q$ was linearized with respect to angular velocity. Below is the jacobian of q, as well as the discrete next-step quaternion.

$$\nabla_\omega q = \begin{bmatrix} q4 & -q3 & q2 \\ q3 & q4 & -q1 \\ -q2 & q1 & q4 \\ -q1 & -q2 & -q3 \end{bmatrix};\ q_{k+1} \approx dt * 1/2 * \nabla_\omega q|_q * \omega + q_k$$

Eq. 11 places the known quantities into the usual discrete linear system form. For the inner-loop, the same linear model used in the EKF is used here.

$$q_{k+1} = Aq_k + B\omega;\ A = eye(4),\ B = dt * 1/2 * \nabla_\omega q|_q \tag{11}$$

In the simulink model, the control loop is ran once every ten time-steps. This is valid because linearized dynamics are accurate within an area surrounding the point of linearization. Satellite's dynamics evolve relatively slow compared to the time-step, so the gain matrix is only updated sparingly.

Initially, I had significant trouble getting the LQR controllers to function properly. The satellite could control its ang. vel., but not the attitude. After careful inspection, I figured out the issue: the weight matrices on each LQR were not properly adjusted after implementing the successive control-loop structure. With no feed-forward terms or direct coupling between attitude and control moment, there is a natural delay whenever an attitude is commanded. If the attitude controller doesn't allow enough time for the ang. vel. to 'catch up', then it will change the commanded ang. vel. before anything even happens. This was resolved by weighting the attitude rather low, and the ang. vel. higher, so that the inner-loop reaches the commanded velocity much quicker and is able to affect the attitude as desired.

In testing the controller, a nominal trajectory was defined for the ISS to follow. The satellite begins in the RTN frame with $\omega = [0 - n00]'$ and is subject to perturbations. The goal is to maintain the desired attitude.

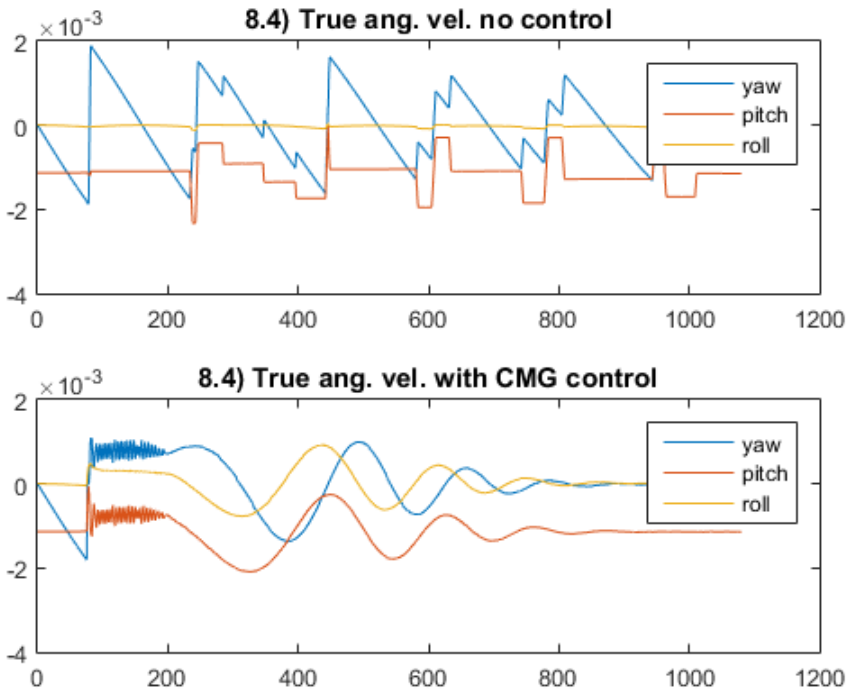


Figure 53: Angular velocity with and without control.

Fig. 57 shows how the euler angles evolve during the flight. With no control the yaw and pitch vary sinusoidally due to the perturbations present. With active control the euler angles converge onto their nominal value and retain them for the rest of the flight. The jitter in the beginning is due to the EKF's random initialization. It hasn't yet converged onto the true value so when $\omega_{measured}$ is fed into the controller it produces garbage control moments.

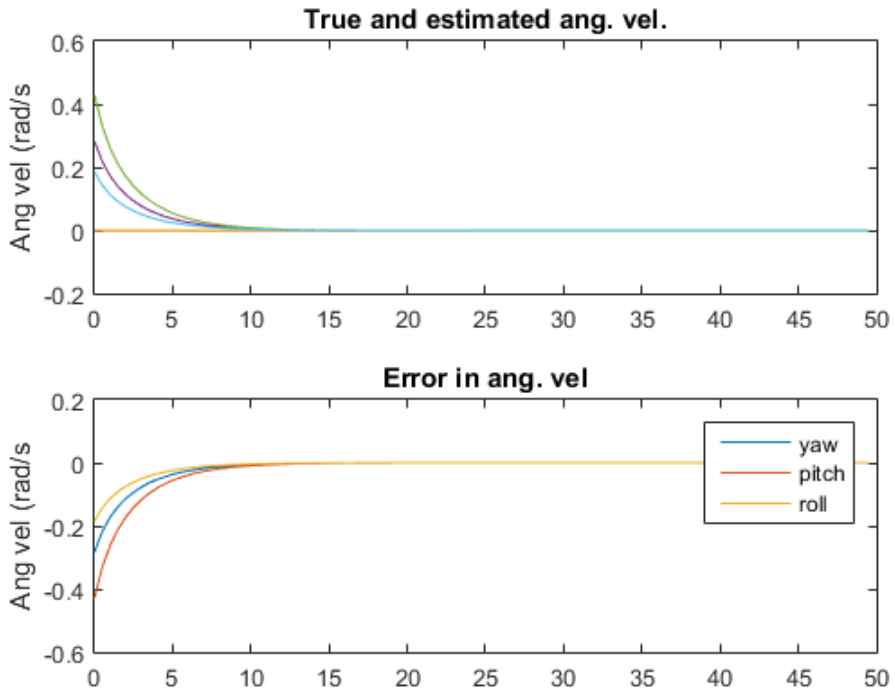


Figure 54: EKF performance.

Fig. 54 shows how the extended kalman filter behaves near the beginning of the flight. After randomly initializing the state, new observations allow the EKF to quickly determine the optimal P matrix and converge onto the true observations. Within 10 seconds, the error in angular velocity approaches zero.

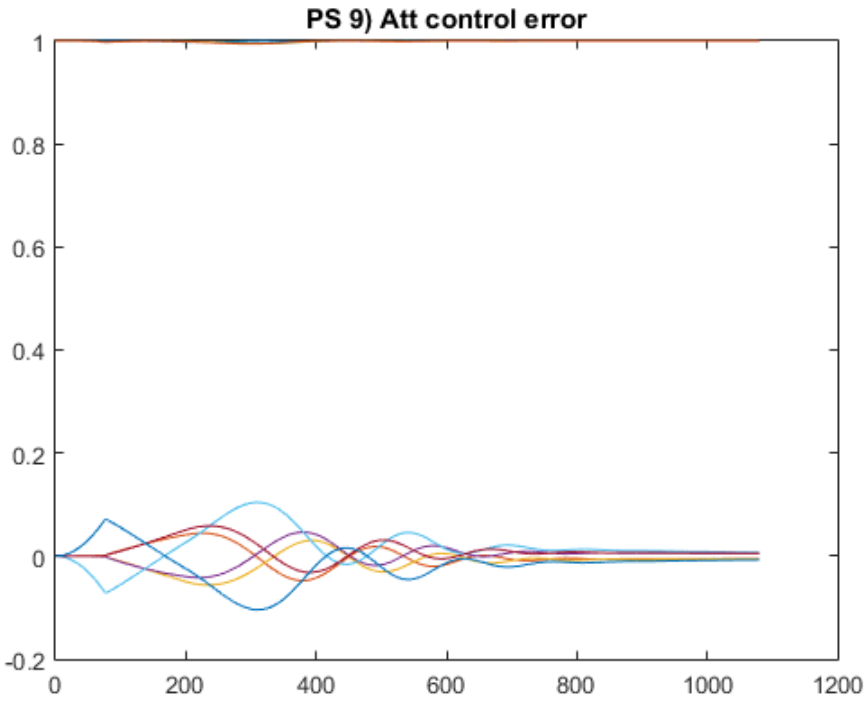


Figure 55: Error in attitude control between principle and RTN frames.

Above is the attitude control error during the flight. This represents the direction cosine matrix between principle axes and RTN frame. Perturbations and state uncertainty lead to some drift between the frames, but the active controller quickly overcomes them and returns the ISS to its nominal attitude.

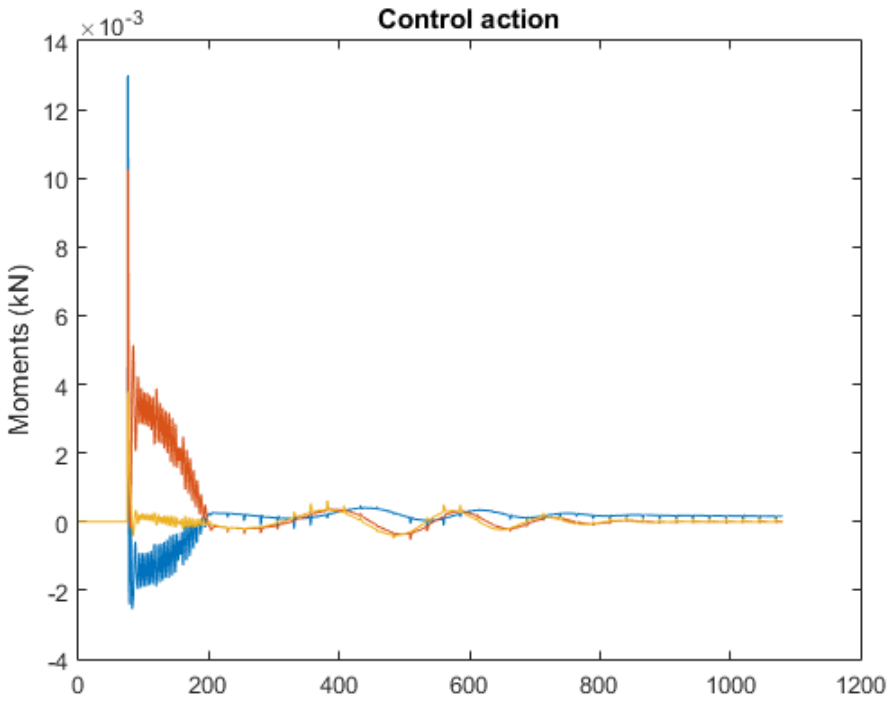


Figure 56: Angular velocity while de-tumbling.

The output of the controller is how much moment to impart in each principle direction. Besides the initial jitter, very little moment is required to keep the orbit.

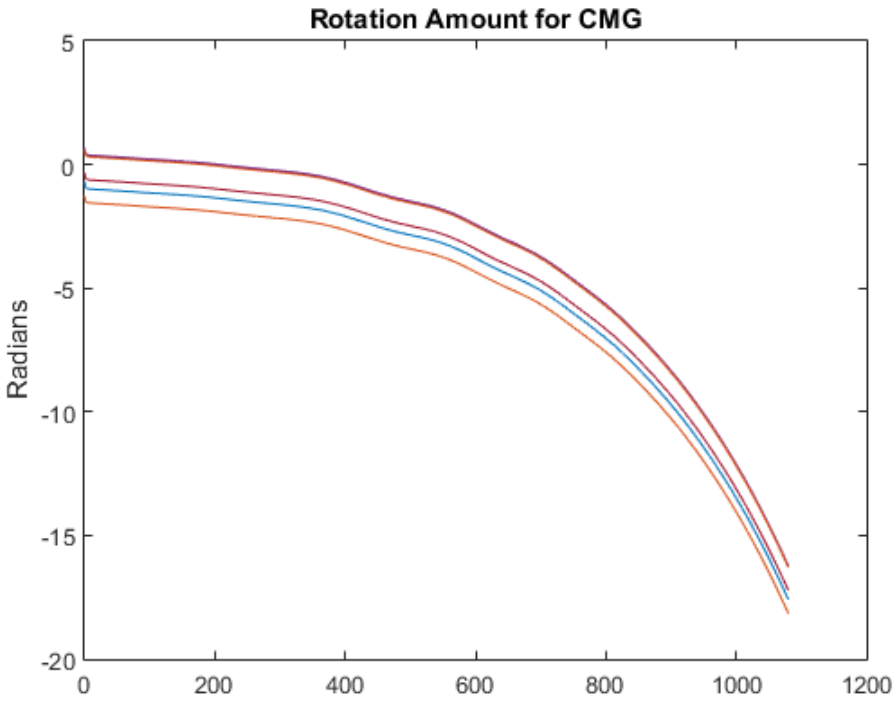


Figure 57: Angular velocity while de-tumbling.

After control moments are generated, the amount of gimbal required to provide them is calculated. Above shows how the combined CMG system must rotate in order to satisfy the controller's requirements. This result is troubling, as each CMG needs to rotate multiple revolutions to keep the station for part of an orbit. I suspect a bug or a sizing issue.

9 Problem Set 9 - Define and Execute a Slew Maneuver

A problem all satellites deal with is de-tumbling. When launched into orbit the satellite is harshly ejected from its rocket into a target orbit and must immediately begin to align itself into its nominal position. I've simulated a tumbling environment by setting $w0 = rand(3, 1)$ and allowing the dynamics and controller to stabilize the system naturally. The target attitude for the de-tumble is the earth-facing RTN frame.

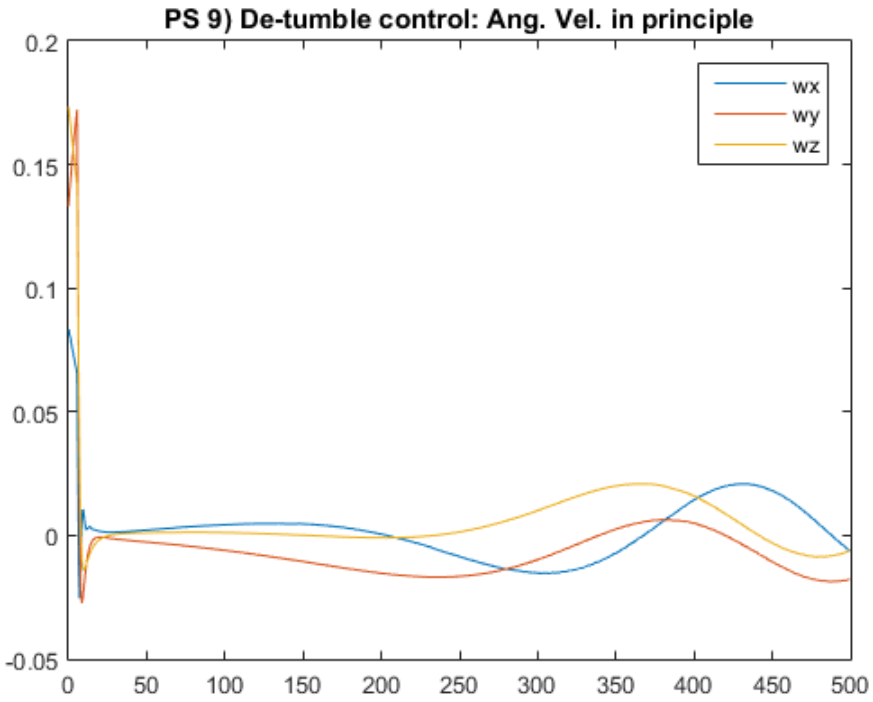


Figure 58: Angular velocity while de-tumbling.

Fig. 58 shows the angular velocity while de-tumbling. Because of the high priority on matching w to $w_{desired}$ the velocities stabilize quickly. After that all changes are due to the attitude controller commanding certain angular velocity in order to align itself with the RTN frame.

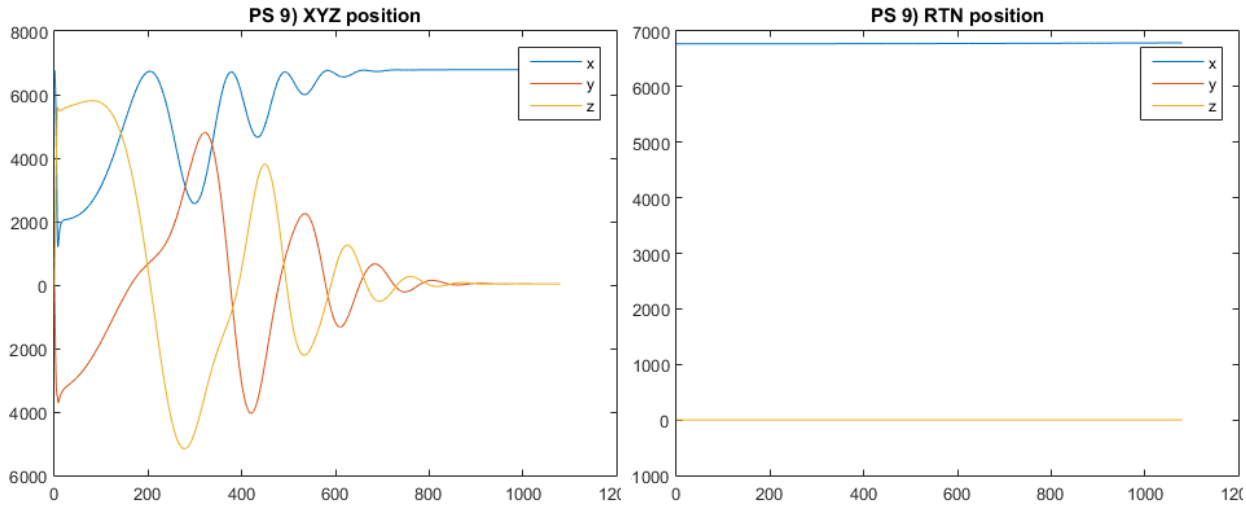


Figure 59: The position during a de-tumble. The Target is earth-facing in the RTN frame.

Fig. 59 shows how the principle axes aligns itself with the RTN frame over time. Oscillations are present due to the sinusoidal dynamics of the rotation cosine matrices.

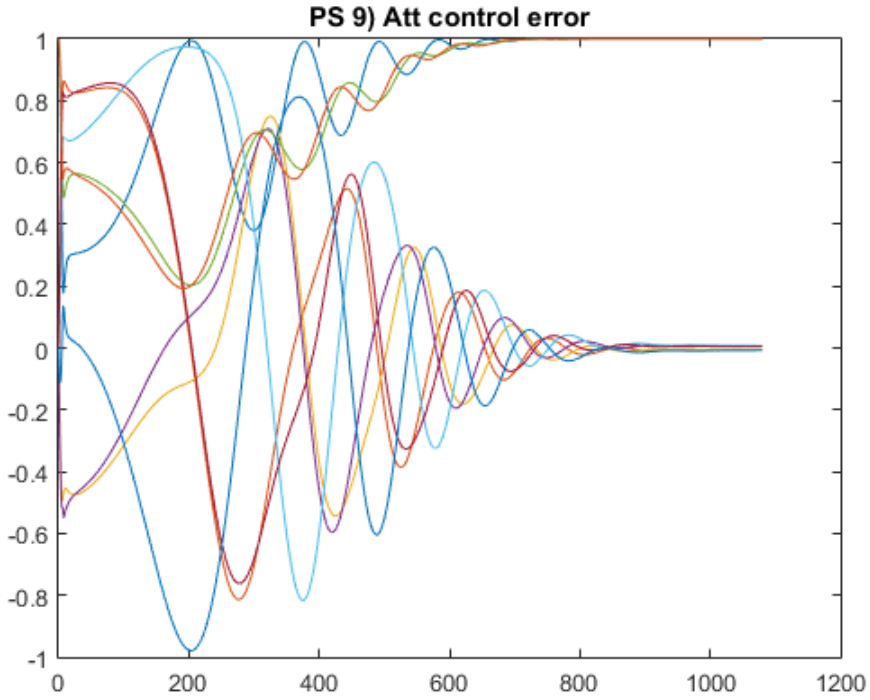


Figure 60: Attitude control error until convergence.

Fig 60 shows the evolution of the attitude control error over time. It's clear the the tumbling has a huge effect on the relative alignment of the principle axes with the desired RTN frame. But this problem is overcome within 800 seconds of ejection with the satellite's onboard controller.

10 Appendices A - Matlab Code

```
function [ surfaceAreasList, surfaceCM_bodyList, normalsList ] = S_top2Lists( S_top )
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

% extract surface areas, surfaceCM_body, and normals from S_top and put
% into big long list
surfaceAreasList = [];
surfaceCM_bodyList = [];
normalsList = [];
for i=1:length(S_top.children)
  chd = S_top.children{i};
  surfaceAreasList = [surfaceAreasList; chd.surfaceAreas];
  surfaceCM_bodyList = [surfaceCM_bodyList, chd.surfaceCM_body];
  normalsList = [normalsList, chd.normals];
end
```

Published with MATLAB® R2015a

```
function [ S_top ] = genGeom( chd )
%GENGEOM Summary of this function goes here
%   Detailed explanation goes here

% top parent
S_top = struct;


S_top.children = chd;
% calculate DCM's between cuboids and parent center
% -- %
numCubs = length(S_top.children);


% calculate inertia tensor for each cuboid w.r.t. parent center
S_top.tauGlobal = 0; %global inertia tensor
for i=1:numCubs
  S_top.children{i}.tauLocal = cuboidInertia(S_top.children{i}.L, ...
      S_top.children{i}.W, S_top.children{i}.H, S_top.children{i}.mass);
  S_top.children{i}.tauGlobal = convertToGlobalInertia(S_top.children{i}.mass, ...
      S_top.children{i}.tauLocal, -S_top.children{i}.cm_x, -S_top.children{i}.cm_y, ...
      -S_top.children{i}.cm_z, S_top.children{i}.r_x, S_top.children{i}.r_y, S_top.children{i}.r_z);
  S_top.tauGlobal = S_top.tauGlobal + S_top.children{i}.tauGlobal;
end
% calculate center of mass of entire system
S_top = calcSystemMass(S_top);

%body axis surface CM coords
for i=1:numCubs
  S_top.children{i}.surfaceCM_body = S_top.children{i}.surfaceCM_parent + ...
          -1*repmat([S_top.cm_x; S_top.cm_y; S_top.cm_z], [1,6]);
end
```

```
% translate final tauGlobal to center of mass
S_top.tauCM = convertToGlobalInertia(S_top.totMass, S_top.tauGlobal, ...
    S_top.cm_x, S_top.cm_y, S_top.cm_z, 0, 0, 0);

% calculate principle axes (eigenvalue/vector problem)
[V,D] = eig(S_top.tauCM);
S_top.tauCM_P = D;
S_top.DCM_P2B = V; %direction cosine matrix - principle to body
S_top.DCM_B2P = S_top.DCM_P2B';

% convert m to km
S_top.tauGlobal = S_top.tauGlobal / 1000^2;
S_top.tauCM = S_top.tauCM / 1000^2;
S_top.tauCM_P = S_top.tauCM_P / 1000^2;
end
```

Published with MATLAB® R2015a

```
function betterPlotEarth()
%UNTITLED17 Summary of this function goes here
%   Detailed explanation goes here
grs80 = referenceEllipsoid('grs80','km');
domeRadius =  3000;  % km
domeLat =  39;       % degrees
domeLon = -77;       % degrees
domeAlt = 0;         % km

[x,y,z] = sphere(20);
xEast  = domeRadius * x;
yNorth = domeRadius * y;
zUp    = domeRadius * z;
zUp(zUp < 0) = 0;

figure('Renderer','opengl')
ax = axesm('globe','Geoid',grs80,'Grid','off', ...
    'GLineWidth',1,'GLineStyle','-',...
    'Gcolor',[0.9 0.9 0.1],'Galtitude',100);
ax.Position = [0 0 1 1];
axis equal off
view(3)
load topo
geoshow(topo,topolegend,'DisplayType','texturemap')
demcmap(topo)
land = shaperead('landareas','UseGeoCoords',true);
plotm([land.Lat],[land.Lon],'Color','black')
rivers = shaperead('worldrivers','UseGeoCoords',true);
plotm([rivers.Lat],[rivers.Lon],'Color','blue')

end
```

Published with MATLAB® R2015a

```
function [ chd ] = mkChild( L,W,H,m,t_x,t_y,t_z,r_x,r_y,r_z )
%MKCHILD Summary of this function goes here
%   Detailed explanation goes here
% generate cuboids of correct size
chd = struct;
chd.L = L;
chd.W = W;
chd.H = H;
[x,y,z,area,CM] = getCuboid(L, W, H); %meters
chd.x = x;
chd.y = y;
chd.z = z;
chd.surfaceAreas = area;
chd.surfaceCM_local = CM;
chd.mass = m; %kg
```

```
% define position of cuboids w.r.t. parent center
count = 1;
chd.t_x = t_x; %translate x - from cuboid to parent
chd.t_y = t_y; %translate y
chd.t_z = t_z; %translate z
chd.cm_x = chd.t_x + chd.L/2;
chd.cm_y = chd.t_y + chd.W/2;
chd.cm_z = chd.t_z + chd.H/2;
chd.r_x = r_x; %rotate x
chd.r_y = r_y; %rotate y
chd.r_z = r_z; %rotate z
%generate surface normals & positions
chd.normals = getNormals(x,y,z);
chd.surfaceCM_parent = chd.surfaceCM_local + ...
      repmat([chd.t_x; chd.t_y; chd.t_z], [1 6]); %this is in parent center coords

end
```

Published with MATLAB® R2015a

```
function [ fig1 ] = plotAxesTriads( fig1, S_top, mult )
%PLOTAXESTRIADS Summary of this function goes here
%   Detailed explanation goes here

figure(fig1);
P = S_top.DCM_P2B;
colors = {'red','green','blue'};
for i=1:3
  Pi = mult.*P(:,i)';
  plot3([0; Pi(1)], [0; Pi(2)], [0; Pi(3)], colors{i},'linewidth',2)
end
```

Published with MATLAB® R2015a

```
function fig1 = plotSC(fig1, S_top )
%PLOTSC Plot spacecraft
%   Detailed explanation goes here
numCubs = length(S_top.children);
figure(fig1);
for i=1:numCubs
  x = S_top.children{i}.x + S_top.children{i}.t_x;
  y = S_top.children{i}.y + S_top.children{i}.t_y;
  z = S_top.children{i}.z + S_top.children{i}.t_z;
  patch(x,y,z,'cyan')
end
axis equal
view(3)
```

```
xlabel('x'); ylabel('y'); zlabel('z');
end
```

Published with MATLAB® R2015a