CubeSat Attitude Determination and Control System (ADCS) Design, Analysis, and Validation

Govind Chari (gmc93@cornell.edu)
December 15, 2021

Abstract

The Pathfinder for Pitch Momentum Bias (PMB) is a satellite project undertaken by the Space Systems Design Studio in partnership with Cateni. The project seeks to develop a 6U proto-flight article that shall contain a payload and will be the first small spacecraft to demonstrate pitch-momentum bias stability. For this project, the satellite's net angular momentum will point parallel to the orbit normal. This project seeks to reuse as much code and hardware from the Pathfinder for Autonomous Navigation (PAN) project as possible. A feasibility study has been completed which has demonstrated pitch momentum bias of a small spacecraft in a simulation. Additionally, the team has spent this past semester working on System Requirements Review. This paper focuses on how to design, analyze, and verify the attitude determination and control system (ADCS) thoroughly and includes a guide on the structure of our in-house simulation tool: psim.

Table of Contents

Introduction 3
Design 3
Pointing controller 5
Momentum Dumping 7
Orbit Estimator 7
Attitude Estimator 9
Analysis 11
Psim 12
Monte-Carlo 14
Controller Analysis 15
Estimator Analysis 15
Characterization 18
Validation 19
Conclusion 20
References 20

Introduction
For spacecrafts, ADCS falls under the broader umbrella of guidance, navigation, and controls (GNC). Navigation seeks to answer the question, "What is my state?" Guidance answers the question "What does my state need to be?" Controls answers the question "How do I get my state where it needs to be?" In this context, a state is the minimum set of information needed to describe the evolution of a system. A common state is position, velocity, attitude, and angular rate. ADCS seeks to answer these three questions for attitude and angular velocity.

In general, GNC work should be some of the first things to be considered in a satellite or aerospace program. The sensors, actuators, and a good amount of hardware should be driven by GNC requirements. If this is not the case, you could end up with having to make software fixes for problems that could have had simple hardware solutions. For example, if the magnetometers are placed too close to the magnetorquers you would have to characterize the bias and scaling error magnetorquer commands have on the magnetometer readings or else the magnetometer readings would be highly inaccurate leading to significant attitude estimation error. This kind of problem could be solved by simply by placing the magnetorquers far from the magnetometers.

This paper focuses on how to design, assess, and test a robust and stable ADCS for the PMB project given the ADCS requirements. This report is structured in three main sections: design, analysis, and verification. The design section focuses on how to design the controller and estimator for all satellite modes as well as all the assumptions embedded in this design. The analysis section focuses on how to test the controller and estimator in a realistic simulated environment. In this section we will also discuss how psim is structured and how to develop psim models and simulations. We will also discuss how to characterize sensors, actuators, and satellite mass properties, all of which are critical for accurately analyzing ADCS performance. Finally, in the verification section we will discuss how to validate ADCS software in software-in-the-loop and hardware-in-the-loop simulations

Design
In this section we will consider how to design the detumbling control law, pointing controller, and attitude estimator for the satellite. To determine the attitude of the satellite we need to use a magnetometer and compare the reading to how the Earth's magnetic field points at the location that the satellite is at. Thus, we need to know where the satellite is, so we need an orbit estimator. We will also consider the design of this orbit estimator in this section.

Detumble Controller
When the satellite is released from the dispenser, the ADCS is inactive for the first thirty minutes to ensure the safety of the launch provider's second stage. During this time, disturbance torques can build up and cause the satellite to have a significant angular rate. The first controller seeks to

null out this angular rate using magnetometers. The torque commanded by a magnetorquer can be modeled as the following

$$\mathrm{T} = m \times B$$

where $\boldsymbol{m}$ is the satellite's commanded net dipole moment, and $\boldsymbol{B}$ is the local magnetic field in the body frame.

An intuitive first cut at designing a control law for this application would be to command a torque from the magnetorquer in a direction opposite to the satellite's angular rate. The commanded torque magnitude should also be larger if the satellite is spinning faster. This controller is called a b-dot controller, since the derivative of the magnetic field in the body frame (magnetometer readings) is proportional to the angular rate of the satellite: if the satellite is spinning faster, the magnetometer reading will be changing faster. Additionally, the derivative of the magnetometer readings points in a direction perpendicular to both the local magnetic field and the of angular rate of the satellite. This can mathematically be represented as the following equation [1].

$$m = -\frac{k}{\|\boldsymbol{B}\|}\dot{\boldsymbol{B}}$$

In this equation, $\dot{\boldsymbol{B}}$ is the derivative of the magnetometer readings and *k* is a scalar gain that can be tuned. A common variant of this algorithm is the b-dot bang-bang controller, which as its name suggests actuates the magnetorquers at full power all the time. This control law takes the form

$$m_i = -m_i^{max}\, sgn\,(\dot{B} \cdot u_i)$$

Where m_i^{max} is the maximum magnetic dipole moment that the ith magnetorquer can generate, $\boldsymbol{u}_i$ is the direction of the ith magnetorquer's dipole moment, and sgn() is the sign function that returns -1 for any negative inputs and 1 for any positive inputs. Since the derivative of the magnetorquer readings are taken via finite differences and the magnetorquer readings have noise, the derivative should be passed through a low pass filter before computing the net dipole moment. This low pass filter will take the following form:

$$y_i = \alpha x_i + (1 - \alpha) y_{i-1}$$

where y_i is the filtered $\dot{\boldsymbol{B}}$, x_i is the current reading of $\dot{\boldsymbol{B}}$, y_{i-1} is the filtered $\dot{\boldsymbol{B}}$ from the previous timestep, and α is smoothing parameter.

For this controller, the only parameter that needs to be tuned is the smoothing parameter α.

Pointing controller

The pointing controller is the controller that is active during most of the mission. This is the controller that will point the satellite wherever the mission dictates. For our mission, this controller will keep the payload pointing nadir, or towards Earth. Due to our PMB stabilization, we only have single axis pointing capability. In other words, we can only control our orientation about an axis parallel to our PMB wheel. It is also worth mentioning that the direction of the PMB will be parallel to the orbit normal vector. This section will take a first cut at designing the attitude controller using linear system analysis. There is a great deal of nonlinear behavior in practice, but all of that will be tackled in the analysis section.

Since we want to always be pointing towards Earth, the satellite must track a reference attitude trajectory with a period of roughly 90 * 60 = 5400 seconds/cycle. This is because a LEO orbit has a period of roughly 90 minutes. This corresponds to a frequency of around 0.0002 Hz. This is an extremely low frequency signal which is good for our controller design. The higher frequency reference signal that a controller must track, the more sensitive it is to sensor noise and the less robust it will be.

A simple attitude controller design would be a PD controller that would produce a torque proportional to how far the satellite is from its desired attitude. To implement this, we would define the attitude error as the vector portion of the error quaternion. The error quaternion encodes the rotation that we would rotate the satellite from its desired attitude to the current attitude. The derivative term would simply be the satellite's angular rate. From Markley & Crassidis, this can be mathematically formulated as the following [1]:

$$\delta q = \begin{bmatrix} \delta q_{1:3} \\ \delta q_4 \end{bmatrix} = q_{current} \otimes q_{desired}^{-1}$$

The formal control law for pointing then becomes

$$\tilde{T}_{des} = -k_p sgn(\delta q_4)\delta q_{1:3} - k_d \omega$$

where k_p and k_d are the P and D gains respectively, ω is the satellite's angular rate in the body frame and $\tilde{T}_{des}$ is the desired torque in the body frame. The $sgn(\delta q_4)$ term is in place to guarantee that the shortest path is taken to the desired attitude. This control law needs one addition to account for the fact that we can only command a wheel torque in the direction of our pitch momentum bias. This modification is given below

$$T_{des} = \tilde{T}_{des} \cdot \hat{w}$$

where $\hat{w}$ is the unit vector pointing parallel to the PMB wheel's angular momentum vector in the body frame. Now that we have the form of our control law, we need to derive the transfer function from our pointing error to our pointing angle. The block diagram for this system is given below

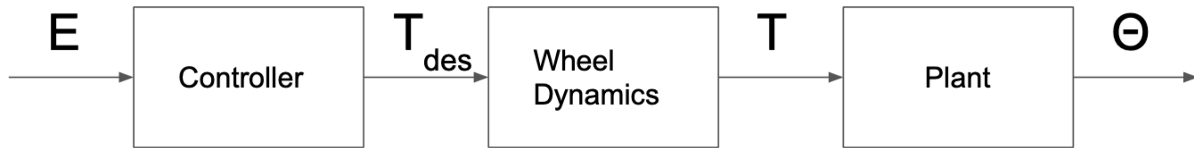

Figure 1: 1D ACS Block Diagram

Transfer functions are most easily defined for linear Single Input Single Output (SISO) systems, but our control law involves torque along three axes and involves the sine of an angle in the quaternion. Thus, we will simplify the model to one dimension which yields the following

$$T_{des} = k_p sin\left(\frac{E}{2}\right) + k_d E$$

where E is the angle error. We can now use the small angle approximation to get the transfer function for our controller which is the following

$$\frac{T_{des}(s)}{E(s)} = \frac{k_p}{2} + sk_d$$

The wheels cannot spin up instantaneously, so there is some delay between the desired torque and the actual torque. We can model this as a first order system. The transfer function for this system would be the following

$$\frac{T(s)}{T_{des}(s)} = \frac{1}{1 + s\tau}$$

where τ is the wheel's time constant. The larger this constant, the longer the delay between desired and actual torque.

Finally, we need the transfer function for the plant. In the 1D case which we are using, the transfer function for the plant is

$$\frac{\Theta(s)}{T(s)} = \frac{1}{Is^2}$$

where Θ is the satellite's attitude about its PMB axis and *I* is the satellite's moment of inertia about its PMB axis.

We now have the loop transfer function

$$L(s) = \left(\frac{k_p}{2} + sk_d\right)\left(\frac{1}{1 + s\tau}\right)\left(\frac{1}{Is^2}\right)$$

We can then plug this loop transfer function into Matlab's SISO tool to use classical control techniques such as Bode plots, root locus, and Nyquist's stability criterion to select the gains k_p and k_d that meet our rise time, settling time, percent overshoot, gain margin, and phase margin requirements. For now, we can treat torques such as residual magnetic dipole, solar radiation pressure, atmospheric drag, and gravity gradient torques as disturbances, and we do not need to explicitly model all of them. In the analysis section we will build up a high-fidelity 6DOF simulation and see how this first cut controller performs and tune the gains accordingly.

Momentum Dumping

With the PMB scheme, attitude is stabilized in the two off axis directions. This means that over time, disturbance torques will build enough on these directions and cause the net angular momentum vector of the satellite to point further away from the orbit normal vector. This will result is growing pointing error over time. Similarly, the PMB wheel will spin faster to reject disturbance torques that act in the PMB direction. Thus, we need a way to remove angular momentum in the off-axis directions and spin down the PMB wheel when its angular velocity becomes too high. The way we achieve this is momentum dumping using our magnetorquers. The control law for momentum dumping is similar to the detumbling control law, except instead of applying a torque opposite to the angular velocity vector, we are now removing a torque opposite to the angular momentum vector. This control law can mathematically be expressed as

$$\mathrm{m} = \frac{k}{\|\mathrm{B}\|} h \times b$$

where $b = \mathrm{B}/\|\mathrm{B}\|$, and h is the satellite's net angular momentum vector. The gain k is a design parameter. With all of this, we have a first cut design of our attitude controllers. In the analysis section we will test their robustness and lock in the final set of controller gains.

Orbit Estimator

To control the satellite's attitude, we must first know its attitude. To do this, we use sun sensors and magnetometer. By comparing the body frame readings of the sun direction and geomagnetic

field to direction of the sun vector and magnetic field vector in an inertial frame, we can come up with an estimate of the satellite's attitude. However, to get the magnetic field vector in the inertial frame, we need to know where the satellite is with respect to the Earth, since the magnetic field is spatially varying. To do this we need an orbit estimator which will give us an estimate of the satellite's position and velocity.

We could simply use GPS readings, but when the satellite is moving at around 7 km/s and the GPS has a refresh rate of 1Hz, our position estimate can be off by at worst 7 km. To get a better estimate, we will use an Extended Kalman Filter (EKF). We will run the predict step every control cycle and the update step whenever we get a GPS reading. The equation of motion for our prediction step is given below

$$\ddot{r} + \frac{\mu}{\|r\|^3} r = 0$$

where r is the satellite's position vector in the ECI frame and μ is the Earth's gravitational parameter. If the predict step were to be implemented in the ECEF frame, additional terms will be needed to capture the non-inertial centrifugal and Coriolis accelerations. When implementing this into the EKF, this equation should be broken up into a system of first order differential equations and then discretized. In this filter, the sensor noise covariance matrix can be determined from the GPS data sheet. However, the process noise covariance matrix must be tuned. We will discuss this filter tuning in the analysis section.

The simplified equation of motion for the predict step should be fine for our purposes, since we will only propagate the satellite's state until we get a GPS reading which should be around 1 second of propagation. However, if more accuracy is desired, we can augment the equation of motion for the predict step with a J2 acceleration term to capture accelerations due to Earth's oblateness. We can also use an Unscented Kalman Filter (UKF), which has the advantage of having the potential to be more accurate and does not require the calculation of a Jacobian. However, it does have more parameters to tune which makes it more non-intuitive and more error prone. With a UKF, we can also use higher order Runge-Kutta integrators for our prediction step. The last modification we could make is to implement the Kalman Filter as a square root filter, where we do all computation with the square root, or more accurately the Cholesky factor, of the covariance matrix. This reduces the condition number of the matrices that we are working with, which makes calculations more numerically stable.

If an orbit estimator is being used for the sole reason of feeding position estimates into the attitude estimator, it may not be worth the effort to make any of these modifications, except the square root implementation since it takes little additional effort and can help guard against numeric instability.

Attitude Estimator

A great deal of time should be spent designing, implementing, testing, and tuning the attitude estimator. Since attitude estimates are fed into the attitude controller, errors in attitude estimates are only amplified by the attitude controller.

Our attitude estimator will use gyroscope readings for the predict step and sun sensor and magnetometer readings for the update step. One assumption that Kalman Filters have is that sensor noise is zero mean with additive Gaussian noise. However, a MEMS gyroscope has a misalignment on installation which manifests as a fixed bias on each axis, time-varying biases, scaling error, and additive noise. If we have a filter which estimates the gyro biases, we can account for all these non-ideal effects that violate our assumptions except scaling error which we really cannot do much about other than characterize on the ground and attempt to account for it in flight. For these reasons, we want an attitude estimator that can simultaneously estimate our attitude as well as our gyroscope biases.

For estimation as important as attitude estimation, we want the highest accuracy estimator within a reasonable computation budget. A very good estimator for this application is the UKF. A properly tuned UKF has better convergence properties than an EKF. As a brief explanation, a UKF has the same predict-update structure as a regular Kalman Filter, but it is adapted to work for nonlinear dynamics. With a regular Kalman Filter, the Gaussian prior distribution can be propagated through the linear state transition equation which results in another Gaussian, but when we have nonlinear dynamics, the Gaussian prior becomes non-Gaussian after being propagated. An EKF solves this issue by linearizing the state transition equation about the previous estimate thus allowing the Gaussian prior to stay Gaussian after propagation. The UKF on the other hand intelligently selects a few key points from the prior distribution (called sigma points), propagates them through the nonlinear dynamics and then fits a Gaussian to these propagated sigma points.

The paper by John Crassidis describes an implementation of the UKF for attitude and gyro bias estimation [2]. This paper has an important addition onto the standard UKF, and it deal with the problem of keeping the attitude quaternion normalized. In this filter, the prediction step propagates the attitude using gyro readings and updates this prediction with sun sensor, magnetometer, or star tracker readings. If a quaternion were used for the attitude formalism, the quaternion will stay normalized in the prediction step (if an accurate integrator were used), but the update step will unnormalize the quaternion since a correction term is added to the predicted quaternion which doesn’t guarantee that unit length is maintained. One could brute force normalize the quaternion after the update step, but this will result in attitude estimation error over long enough timeframes. Crassidis’ paper discusses using generalized Rodrigues parameters (GRP) to represent the local error quaternion. The GRPs are an unconstrained set of three numbers to represent attitude. Using GRPs for computation eliminates the issue of quaternions

becoming unnormalized. The PMB project will likely use the UKF that Crassidis describes for attitude estimation, since this is the same estimator that PAN uses.

The last issue remaining with the attitude estimator is initializing the filter. For this we will use the TRIAD method [3]. Firstly, we assume that we have two linearly independent direction measurements in two different frames. For example, this could be a sun sensor measurement and magnetic field reading in the body frame and then the same two truth vectors in ECI. We then try to solve for the direction cosine matrix (DCM) that rotates the ECI vectors into their corresponding body frame measurements.

Assume the two body measurements are $r_1{}^B$ and $\boldsymbol{r_2}^B$ and their actual representation in ECI coordinates are $\boldsymbol{r_1}^{\mathbf{I}}$ and $\boldsymbol{r_2}^{\mathbf{I}}$. We can then define the following

$$\hat{\boldsymbol{s}}^{\boldsymbol{B}} = \frac{\boldsymbol{r_1}^{\boldsymbol{B}}}{\|\boldsymbol{r_1}^{\boldsymbol{B}}\|} \qquad \hat{\boldsymbol{s}}^{\mathbf{I}} = \frac{\boldsymbol{r_1}^{\mathbf{I}}}{\|\boldsymbol{r_1}^{\mathbf{I}}\|}$$

$$\hat{\boldsymbol{t}}^{\boldsymbol{B}} = \frac{\boldsymbol{r_1}^{\boldsymbol{B}} \times \boldsymbol{r_2}^{\boldsymbol{B}}}{\| \boldsymbol{r_1}^{\boldsymbol{B}} \times \boldsymbol{r_2}^{\boldsymbol{B}}\|} \quad \hat{\boldsymbol{t}}^{\boldsymbol{I}} = \frac{\boldsymbol{r_1}^{\boldsymbol{I}} \times \boldsymbol{r_2}^{\boldsymbol{I}}}{\| \boldsymbol{r_1}^{\boldsymbol{I}} \times \boldsymbol{r_2}^{\boldsymbol{I}}\|}$$

We can now set up the following relation using the definition of a change of basis.

$$[\hat{\boldsymbol{s}}^{\boldsymbol{I}} \quad \hat{\boldsymbol{t}}^{\boldsymbol{I}} \quad \hat{\boldsymbol{s}}^{\mathbf{I}} \times \hat{\boldsymbol{t}}^{\boldsymbol{I}}] = C\,[\hat{\boldsymbol{s}}^{\boldsymbol{B}} \quad \hat{\boldsymbol{t}}^{\boldsymbol{B}} \quad \hat{\boldsymbol{s}}^{\mathbf{B}} \times \hat{\boldsymbol{t}}^{\boldsymbol{B}}]$$

Where C is the change of basis matrix from the body frame to the ECI frame. We can then solve for C as follows.

$$C = [\hat{\boldsymbol{s}}^{\boldsymbol{I}} \quad \hat{\boldsymbol{t}}^{\boldsymbol{I}} \quad \hat{\boldsymbol{s}}^{\mathbf{I}} \times \hat{\boldsymbol{t}}^{\boldsymbol{I}}][\hat{\boldsymbol{s}}^{\boldsymbol{B}} \quad \hat{\boldsymbol{t}}^{\boldsymbol{B}} \quad \hat{\boldsymbol{s}}^{\mathbf{B}} \times \hat{\boldsymbol{t}}^{\boldsymbol{B}}]^T$$

This DCM can then be converted into a quaternion or generalized Rodrigues parameter.

There are several methods we could use for initializing our attitude estimator, for example QUEST or Davenport's q-method. These methods seek to find solutions to the following optimization problem.

$$\min_R \; \frac{1}{2}\sum_{k=1}^{N} a_k \,\|\mathrm{w_k} - R v_k\|^2$$

where we have N measurements, a_k are weights that we can select depending on how much we trust each measurement, $\mathrm{w_k}$ is the kth measurement in the ECI frame, v_k is the kth measurement in the body frame, and R is the DCM that we are trying to find. This optimization problem is

called Wahba's problem, and it seeks to find the optimal DCM, R^* that minimizes the above cost function. Using QUEST or Davenport's q-method would result in a more accurate filter initialization, but it would be quicker to compute the initial attitude using TRIAD, and even though it may be less accurate, the attitude estimator will converge quick enough with a decent initial attitude estimate.

In this section we looked at first cut designs for our attitude controllers and attitude estimators. To fully evaluate the robustness of the estimators and controllers and to do final tuning, we will need to construct a high fidelity 6DOF simulation that mocks reality as well as possible. We will look at how our 6DOF simulation, psim, works and how to use it to evaluate our initial controller and estimator designs.

Analysis

For most space missions, you have one shot at getting the satellite to work. A significant amount of time and money is invested into satellite development. As a result, we want to know exactly how the satellite will operate before launching. The rest of this paper will discuss how to ensure that the satellite operates exactly how it is intended to by leveraging computer simulations.

The controllers and estimators are design based on several assumptions. For example, we used SISO tool earlier to design our attitude controller. This assumes that the system is linear, which is not true. Our actuators can saturate, they have dead bands, and most of all the system's actual dynamics are nonlinear. Similarly, our estimators assume that we have Gaussian, additive white noise. Thus, if we test our controller in an environment with linear dynamics and our estimator with additive white noise, they will obviously perform as expected, but real life does not fit all of our assumptions all the time. The way that we will fully validate that the ADCS will work in real life is by testing it in a simulation that mocks real life as best as possible. It is critical to understand that the better the simulation replicates real life, the more certain you can be that the ADCS will work as intended.

Almost all major aerospace companies have internal simulation tools to test the robustness of controllers and estimators and predict performance under a wide variety of uncertainties from mass uncertainty to sensor noise etc. In my opinion, it is best to write these simulations in C++. At the end of the day, you will want to link your entire flight and ground software stack and test them in your simulation. The ground and flight software will be written in almost exclusively C/C++, so it is easiest link if you simulation is also in C++. Also, these simulations are extremely computationally heavy, and you will want it to run very fast. MATLAB tends to run very slowly, so C++ is a very good choice for a fast-running language. The only downside is that development time is longer for C++, but this drawback is worth it in the long run. PMB will inherit PAN's simulation stack, psim, and make modification as necessary.

Psim

This section will briefly cover how psim is structured and how to go about developing in psim. Psim is too big to make an entire written guide on how it works, but this section seeks to cover its structure at a high level.

Most of psim is written in C++. However, the interface layer is all Python. This python interface makes it very easy to run and interact with the simulation. We can run a simulation by typing into terminal the simulation we want to run, the config files it uses, the number of steps we want to simulate for, and the plots that we want to produce.

Psim is split into two group of code: psim and gnc. The gnc group houses all the gnc code (estimators and controllers). This code consists almost entirely of functions that take in some data and calculate state estimates or desired actuator commands. This directory is linked with our flight software codebase, which means that whenever we prototype and push an algorithm in psim, it is updated in flight software.

The second directory is the psim directory. This group of code includes the integrators, flight computer models, actuator models, truth models, sensor models, and the infrastructure to link the whole simulation together. This is the majority of psim. Psim is still not complete but has most basic functionality. We are seeking to build it up into a very high-fidelity simulation.

A naïve way of building a simulation would be to have a set of functions. For example, have true state would be corrupted with noise then fed into the estimators, which generates estimates that are then fed into the controller, which is then sent to the actuators, which then generates a force or a moment. Then this torque and moment are used in conjunction with the equations of motion to propagate the current state forward, then the cycle repeats. This simulation would work, but it lacks modularity and robustness. What if you wanted to test only the orbital mechanics without the attitude dynamics, or what if you wanted to test the free attitude response without active control. With the simulation architecture described above, code in the simulation would have to be commented out to run these different cases. Whenever source code has to be commented out, we allow an additional failure mode of the code. What if you forget to uncomment the code?

Psim is structured in a modular way that doesn't require source code to be commented out to run different cases. Different cases are run by calling different simulations with different configuration files. The core unit of code in psim is called a model. There are many different types of models: truth models, sensor models, actuator models, and flight computer (fc) models. The truth models model the actual physics for example the attitude dynamics, orbital mechanics, as well as environmental models like the earth's rotation rate, magnetic field and should also include solar activity (but it currently doesn't). These truth models are in attitude_orbit.cpp, environment.cpp, and earth.cpp. The file attitude_orbit.cpp is the file where the equations of

motion for the satellite are housed. If someone were to implement gravity gradient torques, they would have to put it in attitude_orbit.cpp.

The sensor models take in the satellite's state as inputs and return partial noisy state information. For example, the gyro model would take in the true angular rate, then add on scaling error, bias, misalignment, and noise. We currently do not have actuator models, but when we do, they would take in actuator commands as inputs and then output a force or a torque. A reaction wheel model for example would model the actuation dead band, saturation, and lag.

Finally, the flight computer models take care of essential preprocessing of data and decision making that is not in the gnc code. In the actual flight software if the attitude estimator covariance exceeds some bounds, the estimator will reset. This is a functionality that is executed by the flight software code. Essentially, these models try to mock the actual flight software implementation as best as possible.

All these models have three simple functions: get fields, add fields, and step. The get fields and add fields functions act as inputs and outputs. For example, the attitude estimator fc model gets the sun sensor readings, magnetometer readings, and gyro readings and adds the attitude estimate and attitude covariance estimate. The step function is the main block of code that is executed when the model is called. For the attitude estimator fc model, the step function unpacks the data from the get field, assembles it into a struct and calls the attitude estimator from the gnc code.

Normally, transacting data across files can be a pain, because if you forget to include the variable name in the header file or misspell the variables you get errors. To get around this, all models have corresponding .yaml files which allow you to specify what fields the model will get and add and get. These .yaml files are then auto coded into header files which make it easy to access data from the models.
These models are then organized into simulations. There are many simulations that we have for example AttitudeEstimatorTest, AttitudeControllerTest, SingleAttitudeOrbitGnc, SingleOrbitGnc. The simplest is SingleOrbitGnc which simply does orbit propagation and models the satellite as a point mass. SingleAttitudeOrbitGnc propagates the satellites orbit and attitude with no active control. AttitudeEstimatorTest propagates the satellite's orbit and attitude with no active control, but it has the attitude estimator active. AttitudeControllerTest propagates the satellite's orbit and attitude, estimates its attitude, and sends this estimate to the attitude controller.

The last key part of psim is config files. These are text files which contain initial conditions, mass properties, and sensor noise parameters. The goal of config files is to let you modify the behavior of the simulation without touching source code. For example, we can add in two config

files, one containing the initial conditions for a sun synchronous orbit and the other for an ISS orbit.

Now we will discuss how to go about making a model and simulation. For this example, let's assume that we want to make a reaction wheel model and add this into the AttitudeControllerTest simulation. The first step is to create the .yaml file for this model. In this .yaml file we will need to specify the inputs and outputs. The inputs would be the current wheel speed, the commanded wheel speed, the wheel dead band speed, and the maximum wheel speed. The output will be the torque on the satellite and the new wheel speed. Now that we have the .yaml file we will need to create the header file for this model. In this header file we will declare the functions that we will define in the source file. In the source file we will get the fields needed, add the fields needed, and then our actual reaction wheel model will be in the step function. To add this model to the simulation is straightforward. You need to navigate to the attitude_controller_test.cpp simulation file and follow the existing syntax to add the reaction wheel model to the model list.

There are several functionalities that need to be added to psim to make it higher fidelity and I will briefly discuss some of the needed upgrades. These changes will be implemented next semester. The first set of upgrades are physics upgrades. Torques and forces due to solar radiation pressure, atmospheric drag, and gravity gradient should be modeled in. Currently drag force due to atmospheric drag is modeled, but torque is not. Gyro scaling error, gyro misalignment, reaction wheel misalignment, and magnetorquer misalignment should also be added in. Most importantly, Monte-Carlo functionality should be added.

Monte-Carlo

Once we have a simulation that mocks reality as well as possible, our most powerful tool for robustness analysis is Monte-Carlo simulation. In a Monte-Carlo simulation, we disperse all parameters we are uncertain about and run many trials to assess performance. For example, we would run 1000 simulations that simulate the satellite in orbit for a week. For each simulation we would choose different masses, moments of inertia, center of gravity location, and sensor/actuator misalignments. These bounds on these dispersions reflect our knowledge of the parameter. For example, if we are not certain about the satellite's mass, we would randomly choose a mass between 3.00 kg and 3.50 kg. If we were very certain of the mass, then we would select from a range of maybe 3.25 kg to 3.30 kg. For each individual trial we will store the input parameters as well as some performance metric like average pointing error. We then will construct a histogram of pointing errors. If 900 of our 1000 trials have mean pointing errors under our requirement of 2 degrees, then we can conclude that in 90% of cases we will meet our pointing requirement.

This feature needs to be added into psim. This is not too complex though since we can interact with psim through python. We just need to write a python script that chooses a parameter set, write those to the config files, call the simulation, compute the performance metric, then save the parameter set and performance metric to some file. The challenge is how computationally intensive each simulation is and the fact that we usually need 1000 simulations to get a stable histogram. One solution to this is to use multithreading and to run multiple simulations in parallel.

Controller Analysis

Now we will analyze our controller's performance and robustness. Many assumptions that are made when designing a controller are not actually valid but may be good enough. Here we will test whether the controller we designed is acceptable. We will mostly leverage the Monte-Carlo simulations. We will run 1000 trials and see if pointing performance is as expected. If it is not, we will take a closer look at the set of parameters that result in subpar performance and then tune the PID gains until the desired performance is met. This tuning is more of an art than a science and is a very iterative process. In almost all cases, we will not be able to get perfect performance in all Monte-Carlo trials, so we should set a threshold or something like we should see sub two degree pointing accuracy in 99% of trials. Here is also where we will be tuning things like the smoothing parameter from our detumble controller or the gain in our desaturation controller.

Estimator Analysis

Now we will assess estimator performance and tune our estimator as necessary. One of the most important aspects of an estimator is the idea of consistency. Of course, we want the estimate to be accurate, but we also want the estimated covariance to be accurate. For example, we do not want a filter where the actual state significantly deviates outside the 2-sigma bounds of the estimated state covariance. The way that we analyze filter consistency is by looking at a plot of the estimated state error in addition to the 2-sigma bounds. The estimates state error is defined as the state estimate minus the actual state. In this plot, the x=0 line is very important. This line represents the truth value of the state. For a consistent filter, the 2-sigma bounds should not

For a consistent filter the 2-sigma bounds should not intersect with the zero line. This intersection represents the truth value deviating more than 2-sigma from the state estimate. Although this should statistically occur 5% of the time assuming Gaussian noise, it should not occur for extended periods of time. Below is an example of what this plot should look like for a consistent filter.

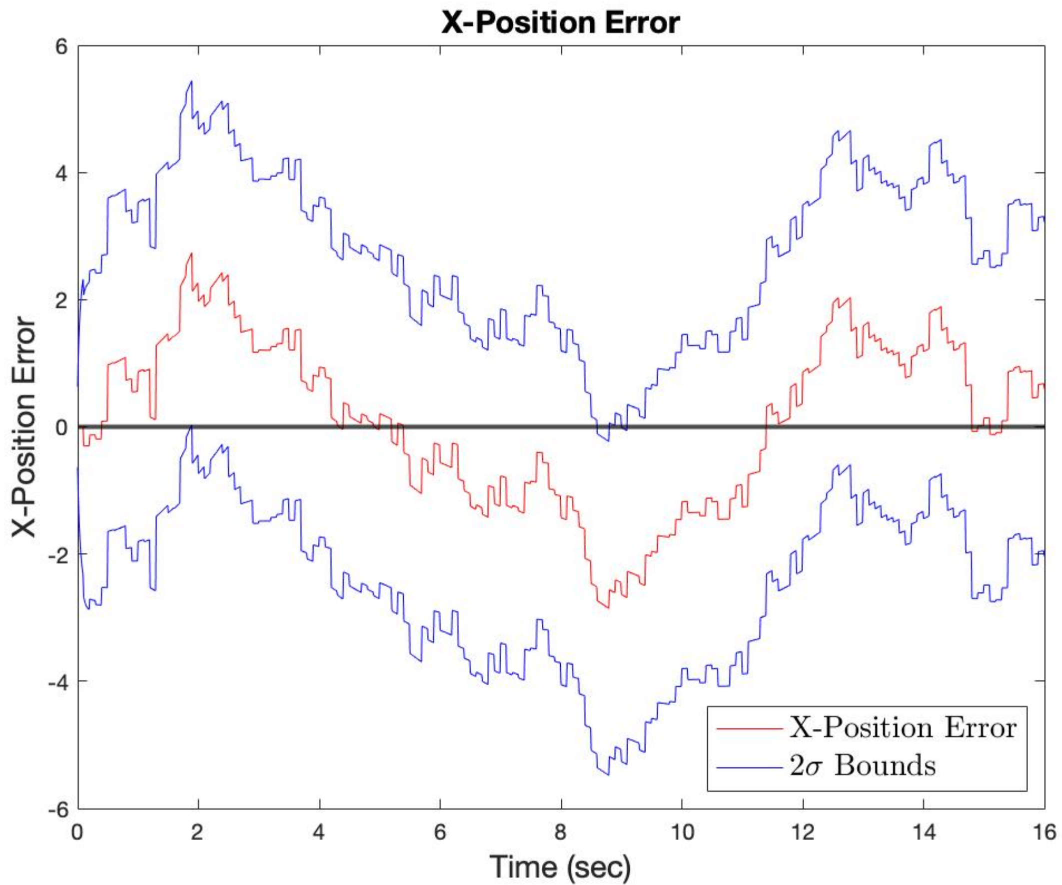

Figure 2: Estimate Error for Consistent Estimator

For this estimator we can see that there is a very brief instant where the 2-sigma bound intersects the zero-error line at around 8 seconds, but this small inconsistency is fine. Below is another error plot, this time for an inconsistent estimator.

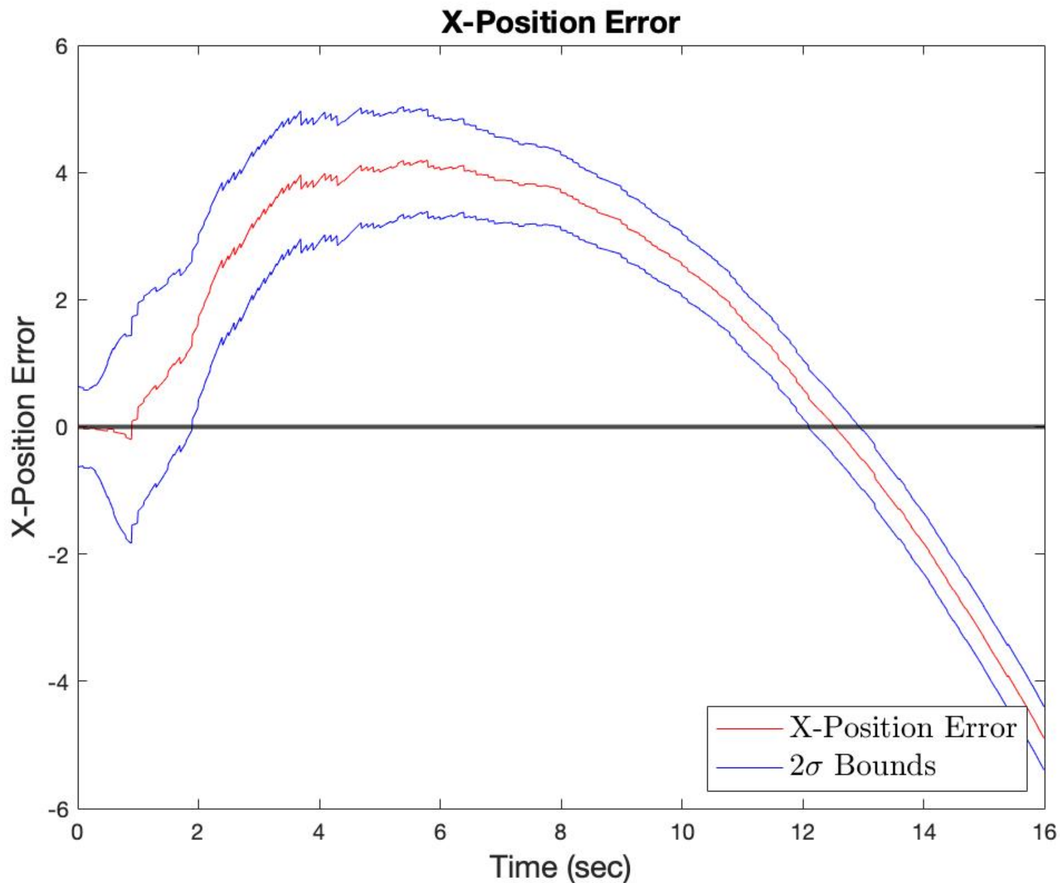

Figure 3: Estimate Error for Inconsistent Estimator

We can see that for this plot the 2-sigma bounds lie well outside the zero-error line for extended periods of time. If we ran an estimator like this on our satellite, not only will we have inaccurate estimates, but we will not know that our estimates are inaccurate.

The way that we can make our estimators consistent is by tuning the process noise covariance matrix. If we make the matrix larger, then we expand the 2-sigma bounds and are more likely to have a consistent estimator. However, we do not want to expand the bounds too much to the point where our estimated covariance does not give us any information. An example of this is shown in a plot below.

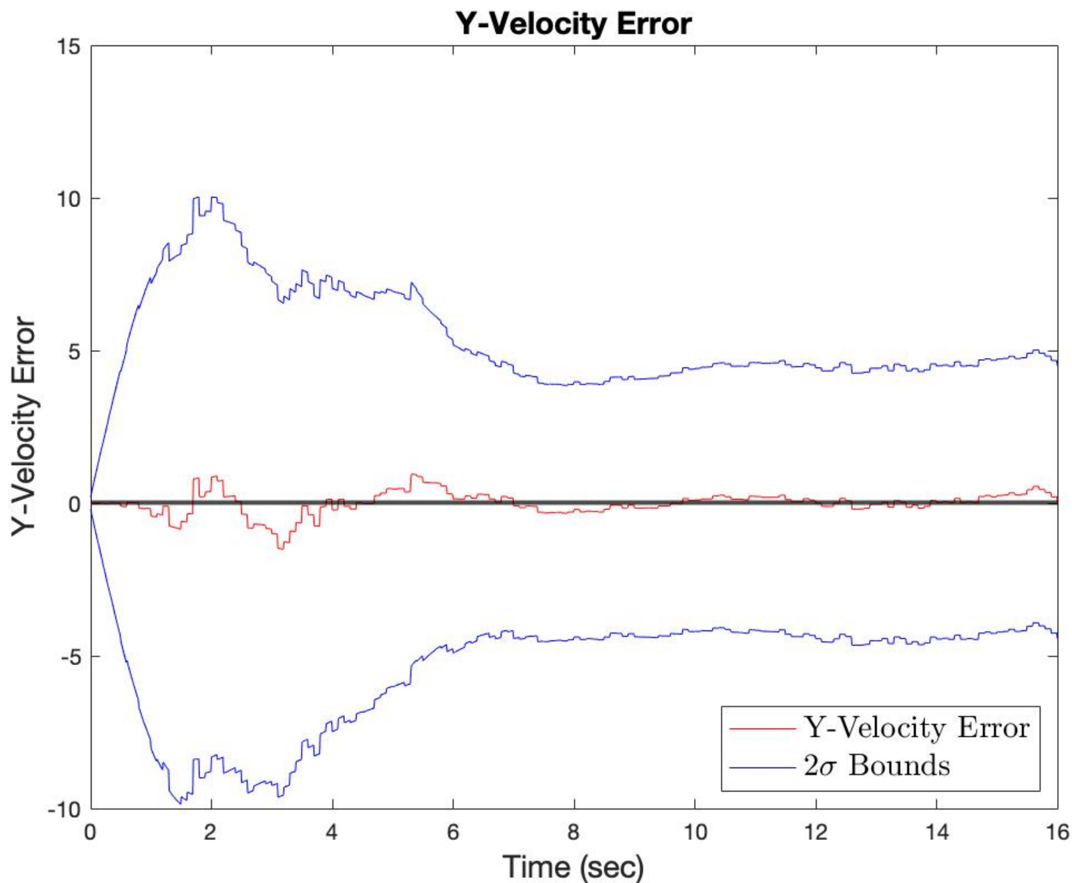

Figure 4: Estimate Error for Overly Conservative Estimator

Characterization

It is important to note that even if all forces and torques are accounted for in the simulation, it is only as accurate as your characterization of the system. This means that you should be spending a good amount of time determining all the mass properties of the satellite: mass, center of gravity, and inertia matrix along with characterizing the sensors and actuators.

For the gyroscope, the noise, bias, and scaling factors should be thoroughly characterized through testing. The same goes for the sun-sensors and magnetometers. Magnetometers like the gyroscope have a bias and bias, so they can be characterized by the following equation

$$y = Ax + b$$

where y is the magnetometer reading, x is the true magnetic field reading, b is the bias of the magnetometer, and A is the scaling factor. From testing, A and b can be determined. If they are not varying, then they can be hardcoded into the satellite and can be accounted for. If they do vary with time, the attitude estimator can be modified to estimate the bias and scaling of the

magnetometer. Similarly, the actuators should be characterized. You should understand the reaction wheel dead band, delay, and time-constant (if it is modeled as a first order system). All of these parameters that are determined experimentally should be plugged into the simulation and the uncertainly in their values should also be characterized and be used to inform the dispersions in Monte-Carlo simulation.

Validation

If the sensors and actuators are characterized perfectly and if the controller and estimators work in simulation, then they will also work in real life. However, we need to validate that this is the case, so we need to perform some tests with the GNC code on the actual satellite. The first set of tests we need to perform are actuator tests. We will first load the satellite onto a testbed that can measure torques. Then we will command the satellite to deliver some torque. We can then measure the actual torque that the actuators are delivering and compare it to the commanded value. If the two values do not match well enough, we need to investigate what causes the discrepancy.

Similarly, we need to test the attitude estimator. We can do this by running the estimator and turning the satellite in predetermined orientations. We can then compare the attitude estimates to the actual attitude.

When your controller, actuators, and estimators perform well in Monte-Carlo and the hardware tests described above this verifies the functionality of the GNC code and hardware, but GNC is only a small fraction of the full software stack. To validate the rest of the software, we need to conduct software-in-the-loop and hardware-in-the-loop tests.

Software-in-the-loop (SITL) also called hardware-out-of-the-loop (HOOTL) tests the entire flight and ground software stack in the simulation. A very small percentage of the full software stack is GNC algorithms. Much of the stack takes care of decision making and communication and thus should be extensively validated.

In the simulation standalone (as described in the previous section) only the controllers and estimators are tested, but SITL is an integrated test that tests the functionality of the state machines and decision making in the flight software as well as the ground software to flight software interaction. These tests can pick up on faulty logic in the state machines such as infinite loops or states that cannot be transitioned out of. To set up SITL, software needs to be written to pipe exchange sensor readings and actuator commands between the software stack and the simulation. The simulation would send noisy sensor readings to flight software and flight software would send actuator commands back to the simulation and step the simulation state forward. Typically, SITL tests are run in real-time where one second in simulation time corresponds to one second in real-life. It is difficult to get SITL to run faster than this, because

the computer that the simulation is running on must execute all the flight software code and simulation stepping.

In hardware-in-the-loop (HITL) tests, the software-hardware interactions are tested. In software and simulations if a valve is commanded to open for one microsecond it would have no problem executing this command. However, a valve cannot open for this short of a time, so this error would be picked up in HITL tests when the valve does not actuate. In HITL tests, the flight software is loaded on to the actual flight computer and all the computation is performed on the flight computer. There must also be a software layer between the flight software and the simulation. The simulation will generate a sensor reading, which is then sent to the actual sensor's register. Then the flight computer will access this sensor reading, run the estimator, make decisions, compute an actuation value then send it to the actuator and the simulation, then the simulation step the state forward. Again, these HITL simulations are run in real time.

Conclusion

Designing, analyzing, and validating a satellite ADCS system is an extremely intensive process that require many hundred or even thousands of hours. Firstly, after the ADCS requirements are set, we will design the controller and estimators using classical feedback control techniques such as Bode plots, root locus, Nyquist plots, and Kalman Filter theory. These techniques have many embedded assumptions such as linear dynamics, and unbiased white sensor noise. If these assumptions were true, then the estimator and controller will work as expected, but we must assess how they will perform in real-life. Thus, we will then test the ADCS code in a simulation that mocks reality as well as possible. We will make this simulation as accurate to real-life physics as possible by characterizing our satellite mass properties, sensors, and actuators at a component. We will then perform Monte-Carlo simulations to see how the ADCS code performs under dispersions. If our simulation and characterization was perfect, then our hardware would be able to actuate the exact torques that we request and estimate the attitude with the same accuracy as it does in simulation. To verify this, we will need to conduct hardware tests using test stands and compare measurements of torque to desired torque and the attitude estimates to actual attitude. Now, we have validated the ADCS code as well as we can, and need to validat the rest of the software stack. To do this, we then conduct SITL and HITL tests.

References

[1] Markley, F. & Crassidis, J. (2014). Fundamentals of Spacecraft Attitude Determination and Control.

[2] Crassidis, John. (2003). Unscented Filtering for Spacecraft Attitude Estimation. Journal of Guidance Control and Dynamics

[3] Black, Harold (July 1964). "A Passive System for Determining the Attitude of a Satellite". AIAA Journal. 2 (7): 1350–1351