THREE AXIS ATTITUDE CONTROL SYSTEM DESIGN AND ANALYSIS TOOL DEVELOPMENT FOR THE CAL POLY CUBESAT LABORATORY

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Aerospace Engineering

by

Liam T. Bruno

June 2020

COMMITTEE MEMBERSHIP

TITLE:	Three Axis Attitude Control System Design and Analysis Tool Development for the Cal Poly CubeSat Laboratory
AUTHOR:	Liam T. Bruno
DATE SUBMITTED:	June 2020
COMMITTEE CHAIR:	Dr. Eric Mehiel, Ph.D. Associate Dean for Diversity and Student Success (Past – Professor of Aerospace Engineering)
COMMITTEE MEMBER:	Dr. John Bellardo, Ph.D. Professor of Computer Science, Computer Engineering, and Aerospace Engineering; Director of the Cal Poly CubeSat Laboratory
COMMITTEE MEMBER:	Dr. Kira Abercromby, Ph.D. Professor of Aerospace Engineering
COMMITTEE MEMBER:	Mr. Dan Wait, M.S. Aerospace Engineering Lecturer

ABSTRACT

Three Axis Attitude Control System Design and Analysis Tool

Development for the Cal Poly CubeSat Laboratory

Liam T. Bruno

The Cal Poly CubeSat Laboratory (CPCL) is currently facing unprecedented engineering challenges—both technically and programmatically—due to the increasing cost and complexity of CubeSat flight missions. In responding to recent RFPs, the CPCL has been forced to find commercially available solutions to entire mission critical spacecraft subsystems such as propulsion and attitude determination & control, because currently no in-house options exist for consideration. The commercially available solutions for these subsystems are often extremely expensive and sometimes provide excessively good performance with respect to mission requirements. Furthermore, use of entire commercial subsystems detracts from the hands-on learning objectives of the CPCL by removing engineering responsibility from students. Therefore, if these particular subsystems can be designed, tested, and integrated in-house at Cal Poly, the result would be twofold: 1) the space of missions supportable by the CPCL under tight budget constraints will grow, and 2) students will be provided with unique, hands-on guidance, navigation, and control learning opportunities. In this thesis, the CPCL's attitude determination and control system design and analysis toolkit is significantly improved to support in-house ADCS development. The toolkit—including the improvements presented in this work—is then used to complete the existing, partially complete CPCL ADCS design. To fill in missing gaps, particular emphasis is placed on guidance and control algorithm design and selection of attitude actuators. Simulation results show that the completed design is competitive for use in a large class of small satellite missions for which pointing accuracy requirements are on the order of a few degrees.

Keywords: Cal Poly CubeSat Laboratory, ADCS, Toolkit, Guidance and Control Algorithm Design, Software-in-the-Loop Simulation, NASA 42

ACKNOWLEDGMENTS

This work would not have been possible without the lifelong support from my family. Thank you for your wholehearted encouragement and financial support.

I would also like to sincerely thank my loving girlfriend, Sierra, for her patience and encouragement during the many hours I spent entrenched in this thesis.

Obviously, a successful thesis requires a good committee. Therefore I would like to extend my sincere thanks to each of the individuals on my thesis committee for their wisdom and guidance throughout this process. Dr. Mehiel, I will not forget our many hours spent in your office discussing the trajectory of this thesis and the many fascinating nooks and crannies of control theory, both classical and modern. Thank you for your hard work this past year. Dr. Bellardo, thank you for supporting me for the past four years in the CubeSat lab as I matured from knowing absolutely nothing to writing this thesis. Dr. A, thank you for dealing with me in each and every one of your orbital mechanics courses, during which I learned almost everything I know about the subject. You truly care about your students, and it shows. For that I am extremely grateful. And finally, Mr. Wait, your keen engineering insight and experience is inspiring to me. You have offered some of the most actionable, perceptive advice to me regarding this work, and for that I am extremely lucky and thankful. Thank you for being a mentor and role model this past year.

Finally, I must give sincere recognition to the many student engineers who have come before me and decided to bravely step up to the challenge of ADCS development for the CPCL. This work would not have been possible without the foundation built by those students. Specifically, I would like to extend my gratitude to Luc Bouchard, Joshua Anderson, Alex MacLean, and Josh Grace for their software engineering expertise and development efforts in support of my thesis. Thank you for inspiring and pushing me to be a better software engineer and programmer. Your time dedicated to helping me out during this past year is sincerely appreciated.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

LITERATURE REVIEW AND GAP ANALYSIS

This thesis will provide enhancements to the CPCL ADCS design and analysis toolkit so that it can be readily employed by members of the CPCL during custom ADCS development. The overall utility of the toolkit will be validated by using it to complete the existing incomplete CPCL ADCS design. This enhanced toolkit is also intended to be applicable in the development of new ADCS designs that must meet specific, currently unknown mission requirements. With access to the enhanced toolkit, members of the CPCL will have the opportunity to design, analyze, and validate their own ADCS.

Overall, the engineering problem of miniaturizing spacecraft attitude determination and control systems has been achieved in the small satellite industry. Most of the necessary hardware and software constituents of a small satellite ADCS can be bought commercially, allowing engineers to simply select from a set of available components in their design process. Furthermore, small satellite attitude control systems are commercially available in modules such as the Blue Canyon XACT or the Maryland Aerospace MAI-400, providing an even more integrated option with relatively little design work necessary. Practically all of this off-the-shelf hardware has achieved TRL 9 [1]. In this first chapter, a historical trend of high level ADCS pointing performance requirements for small satellites is presented, and compared with the corresponding pointing performance offered by commercially available modules. It will be shown that for a large class of small satellite missions, commercially available ADCS modules are a sub-optimal design choice in the sense that they provide overkill performance with respect to typical CubeSat mission requirements. In these cases, their cost also tends to occupy an unreasonably large proportion of mission budgets. Finally, the engineering methodologies associated with the design, analysis, and test of university small satellite attitude control systems are compared and contrasted, highlighting the novelties of the corresponding methodologies that will be proposed in this thesis.

1.1 Historical Perspective on Small Satellite ADCS Performance Requirements

Small satellite missions for which the primary engineering team is comprised of undergraduate students tend to serve as a good model for space missions undertaken by the CPCL in terms of objectives, scope, and budget. Therefore, in this work special attention is given to typical functional and performance requirements driving small satellite ADCS design in university settings. However, before reviewing these requirements, it is important to highlight the distinction between functional and performance requirements as defined in this work. Functional requirements are related to the desired overall behavior of a system, which includes how the system can be commanded and operated, and how it interfaces with other systems. Functional requirements are stated qualitatively. Performance requirements, on the other hand, relate to how well a system does its job, and are usually quantitative. In this section, particular attention is given to ADCS pointing accuracy performance, since those requirements are generally the most influential in terms of the cost and design complexity of the system. ADCS pointing accuracy performance requirements are usually derived from one or more of the following elements of a space mission:

- Payload pointing—primarily imaging, radar, and communications
- High bandwidth communication systems
- Operation of propulsion systems
- Orbit maintenance, rendezvous operations, and formation flight
- Thermally radiating vehicle surfaces

Until recently, most of the above elements were extremely rare in small satellite missions. However, an increasing number of small satellite missions are flying imaging payloads, propulsion systems, and higher bandwidth communication systems. As a result, there is an increasing demand for higher performance and more versatility in an ADCS for small satellites. To illustrate this increasing demand, figure 1 shows a variety of missions along with their year of launch and 3σ pointing accuracy requirement (this information was primarily drawn from the eoPortal Directory of Satellite Missions, as well as [3, 4, 5, 6, and 18]). Since pointing accuracy requirements are formally stated differently dependent on the context and thus should be treated as different types of requirements, what they are ultimately trying to establish—an overall system performance requirement—is context independent. So there will be inevitable information loss

when aggregating pointing requirements from a variety of sources in the literature, but the result will still provide an approximate baseline that can be treated as a good starting goal for an in-house ADCS. One subtle feature of an ADCS system performance requirement is whether it imposes a half-cone pointing accuracy or per-axis pointing accuracies. Although not equivalent, the translation between these requirement definitions does not have a large effect on the numerical values of the requirements. Therefore, these differences will become irrelevant if sufficiently many missions are considered.

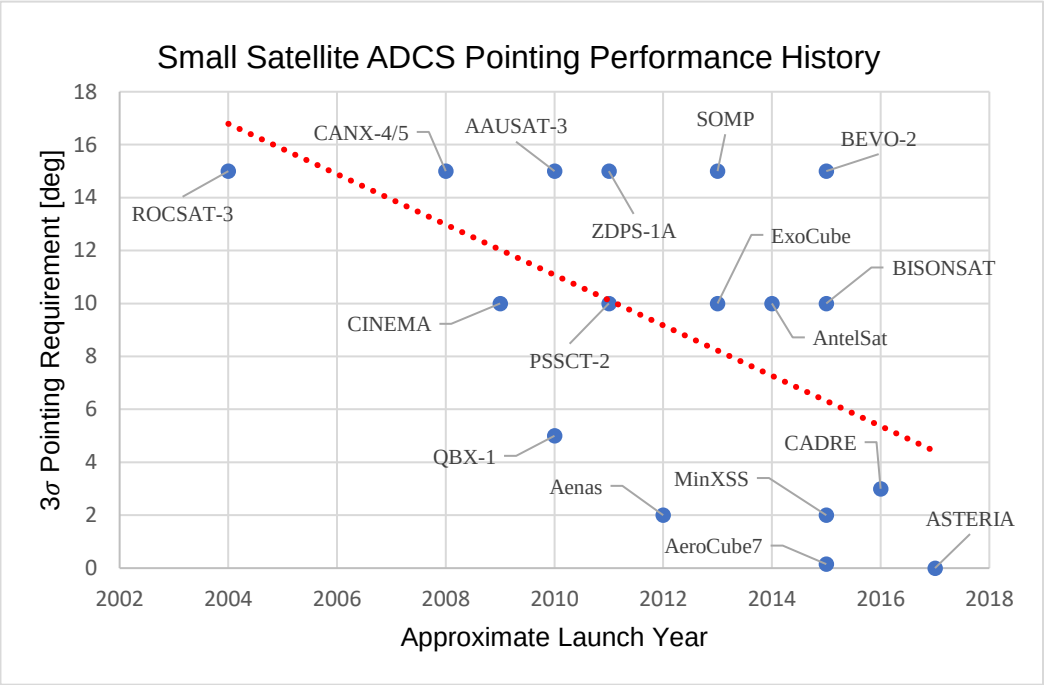

Figure 1: Small Satellite Pointing Performance History

Figure 1 shows a variety of small satellite mission pointing accuracy requirements along with their approximate year of launch. The red dotted line is a linear fit of the data. Although not statistically meaningful, the best fit line provides a notional extrapolation of where modern pointing requirements for small satellites are headed. Figure 1 indicates that a good estimate for modern small satellite 3σ pointing accuracy requirements is around 2°. Note that most of the missions whose 3σ pointing requirements are less than 1˚ are extremely high budget missions which are out of family with university budgets. Such missions are also not primarily designed and operated by universities (e.g. AeroCube7 and ASTERIA). Although this thesis is focused on

university small satellite missions, AeroCube7 and ASTERIA are nevertheless included so as to provide a more holistic picture of the overall trend. Furthermore, including these missions in the data set means that the trend line is more likely to predict an upper bound on modern performance requirements in university settings, which is desirable. In summary, based on an analysis of 17 missions from 2004 to 2017, the most stringent 3σ pointing performance requirements for modern and near future small satellite missions are likely to be around a few degrees, and only in high budget cases with relatively advanced payloads can sub-degree pointing accuracy requirements be expected.

1.2 Commercial ADCS: Cost-Performance Tradeoff

The tradeoff between cost and performance in commercial small satellite attitude control solutions is important to investigate in this context, because if it is not aligned with typical small satellite mission needs as determined in the previous subsection, then there exists a gap in cost-performance space within which new technology may be of significant utility. Figure 2 below provides the approximate unit cost required to achieve the listed pointing accuracy for five attitude determination and control systems that can be purchased commercially. This information was acquired through publicly available information on company websites.

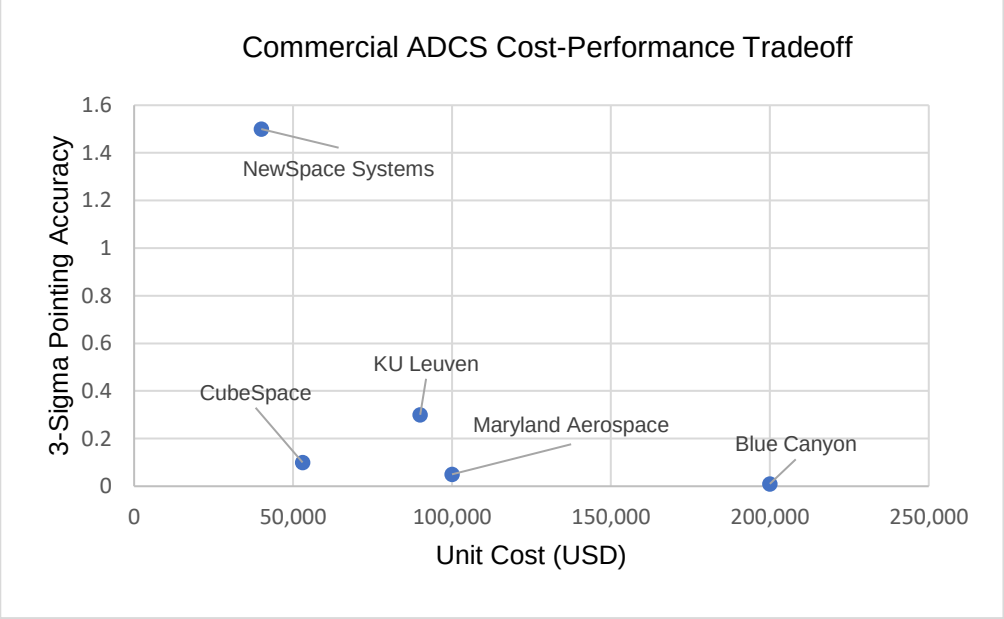

Figure 2: Commercial ADCS Cost vs. Performance

According to figure 2, most of the well-known commercially available ADC solutions for small satellites offer sub-degree 3σ pointing accuracy. From the previous subsection, it was determined that the most stringent modern pointing requirement for small satellite missions is around 2˚. In that case, the system designers are forced to allocate at least ~$80k of the mission budget to the ADCS alone should they select NewSpace Systems ADCS, assuming one test unit and one flight unit. $80k is excessively costly relative to university CubeSat mission budgets. The alternative, higher performance options are all more expensive.

1.3 Inconsistencies between Commercial Performance and Small Satellite Mission Needs

In the previous two subsections, the tradeoff between cost and performance in commercially available attitude control solutions for small satellites was shown to be somewhat inconsistent with the trend in the pointing performance requirements for modern small satellite missions. Such an inconsistency begs the following question: in the case that a university small satellite mission requires ~2˚ 3σ pointing accuracy, is a commercially available solution worth its cost? Obviously, any commercially available solution in figure 2 will satisfy a 3σ pointing requirement of 2˚. However, in university settings, the mission budget is relatively small, and allocating a large percentage of it to the ADCS alone significantly increases the difficulty to close the overall spacecraft design. To avoid spending so much money on one subsystem and constraining the rest of the spacecraft design significantly, many universities—including Cal Poly—have attempted to develop their own ADCS on a per mission basis as an alternative solution. However, such an undertaking is extremely demanding on student engineers who do not have the same level of experience as professional GNC engineers who are designing the commercially available solutions. This thesis aims to drastically alleviate the engineering difficulty associated with designing an in-house ADCS by significantly enhancing the CPCL ADCS design and analysis toolkit.

1.4 Survey of in-house, University Small Satellite ADCS Design and Test Methodologies

In this subsection, a variety of design and test methodologies for university in-house ADCS solutions will be critically analyzed with respect to hardware selection, algorithm design,

and testing framework. It turns out that, among universities, there are a variety of similarities among these criteria when it comes to small satellite ADCS. Table 1 provides a summary of the hardware suite, algorithm selection, and testing framework for various ADCS solutions in a variety of university small satellite missions. This table obviously does not provide an exhaustive analysis of university CubeSat attitude control systems, but it does capture most classes of university CubeSat missions and reflects very common trends.

Table 1: Summary of Small satellite ADC System Level Design and Test

Mission	Sensors*	Actuators*	Algorithms	Testing Framework
MOVE-II [2]	MM, SS, G	MT	Extended Kalman filter, optimal control	Hardware-in-the-loop with in-house MATLAB simulation
CINEMA [3]	MM, SS	MT	Spin stabilized, TRIAD	Standalone in-house MATLAB simulation, autocode
AAUSAT [4]	MM, SS, G	MT	Unscented Kalman filter, model-predictive control	Standalone in-house MATLAB simulation, autocode
FIONA – CANX 4/5 [5]	MM, SS, G	RW, MT	No navigation mentioned, state feedback with integral	Software-in-the-loop with in-house MATLAB simulation
ZDPS-1A [6]	MM, SS, G	RW, MBW	Extended Kalman filter, TRIAD, Bdot, state feedback	Standalone simulation, on-orbit demonstration of ADCS
Mission-Agnostic [7]	MM, SS, G	RW, MT	No navigation mentioned, adaptive nonlinear controller	Hardware-in-the-loop and software-in-the-loop
CPCL ADCS Design	MM, SS, G	RW, MT	Extended Kalman filter, state feedback, PID (developed in this thesis)	Software-in-the-loop with NASA 42, component level testing of sensors and actuators to validate models

*MM = magnetometer, SS = sun sensor, G = rate gyroscope, MT = magnetorquer, RW = reaction wheel, MBW = momentum bias wheel

Table 1 shows that sensor fusion of magnetometers, sun sensors and rate gyros is a common choice for small satellite state estimation hardware. Furthermore, magnetorquers and reaction wheels encompass essentially all choices for attitude actuation hardware. In terms of navigation algorithms, some variant of the Kalman filter is typical. Control algorithms, on the other hand, are more diverse, but they are united by the fact that they all take a modern control theory approach. None of the control algorithms in these five missions can be designed using the

methods and tools of classical control theory, and thus it is much harder to achieve robustness to uncertainty in the plant dynamics. Finally, the strategies used to test the full attitude determination and control system tend to involve the use of MATLAB and/or Simulink to model the environment and algorithm implementation, with the eventual goal of utilizing Simulink's autocode to C feature to generate software which can be compiled into machine code and executed in real time on a flight computer. Although this is a sensible approach, it suffers from the risk that the modeling fidelity may be too low to verify requirements with sufficient confidence. In the CPCL, an open source, high fidelity spacecraft dynamics model is employed to avoid such a risk. Moreover, the Simulink autocode to C feature is not utilized in the CPCL as it would not be possible within the existing flight software architecture.

Overall, the selection of hardware in these missions is sensible for CubeSats as most of it can be purchased commercially, and has a sufficiently small form factor. One issue arising from the choice of magnetorquers as the only actuator—which is not uncommon for university CubeSat missions—is underactuation. Magnetorquers can only impart a torque vector in a plane perpendicular to the instantaneous geomagnetic field vector. While there has been significant research dedicated to dealing with magnetorquer underactuation from a control algorithm perspective, few universities clearly address how they will deal with it. Furthermore, the most prominent issue with these ADCS design and test approaches is the testing framework and, in some cases, the control algorithm design approach. Universities are either deciding to reach extremely high confidence validation with integrated hardware-in-the loop tests, or they are not testing sufficiently. For example, simulations that do not involve flight software-in-the-loop are generally a poor way to verify requirements with high confidence, because they do not invoke the same algorithms that will be executing in real time during flight and thus may behave differently in unexpected ways. On the other hand, since most universities are employing a modern control theory approach to design their control laws, more rigorous simulation and higher modeling fidelity is required to ensure closed loop stability and performance. This is especially problematic when low fidelity models that ignore important dynamics are employed to validate performance.

1.5 Thesis Objective and Outline

Overall, the objective of this thesis is to enhance the CPCL ADCS design and analysis toolkit so that it can be readily employed by members of the CPCL during custom ADCS development. The enhanced toolkit will be validated by using it to create a notional ADCS design from an existing, incomplete ADCS design, and show that the pointing performance of the full notional design is on-par with modern small satellite pointing performance requirements as derived in this chapter. The outline of the rest of this thesis is as follows: first, the current development state of the CPCL ADCS will be outlined in Chapter 2, highlighting some of the elements that require more attention and effort. Then, chapter 3 will provide the formulation of a guidance and control architecture to be applied in the CPCL ADCS toolkit. Chapter 4 will explicitly present the additions to the toolkit as developed in this work, motivated by an overall design iteration strategy presented in chapter 2 and the guidance and control architecture developed in chapter 3. Finally, chapter 5 will show the enhanced CPCL ADCS design and analysis toolkit in action as it is used to create a notional ADCS design and analyze its performance. Chapter 6 will provide concluding remarks and elaborate on future work opportunities.

Chapter 2

BACKGROUND ON CAL POLY CUBESAT LABORATORY ADCS

This section will provide a concise overview of the CPCL ADCS heritage that existed at the beginning of this thesis from a high level perspective. The intent of this chapter is to illustrate what existed before this work in order to clarify why particular attention and effort is dedicated to only certain elements of the overall ADCS design problem in this thesis. Figure 3 below provides a very high-level schematic of the CPCL ADCS. Orange blocks correspond to elements which require significant attention and are currently under development, while green blocks correspond to elements that have either been flown or tested extensively on the ground.

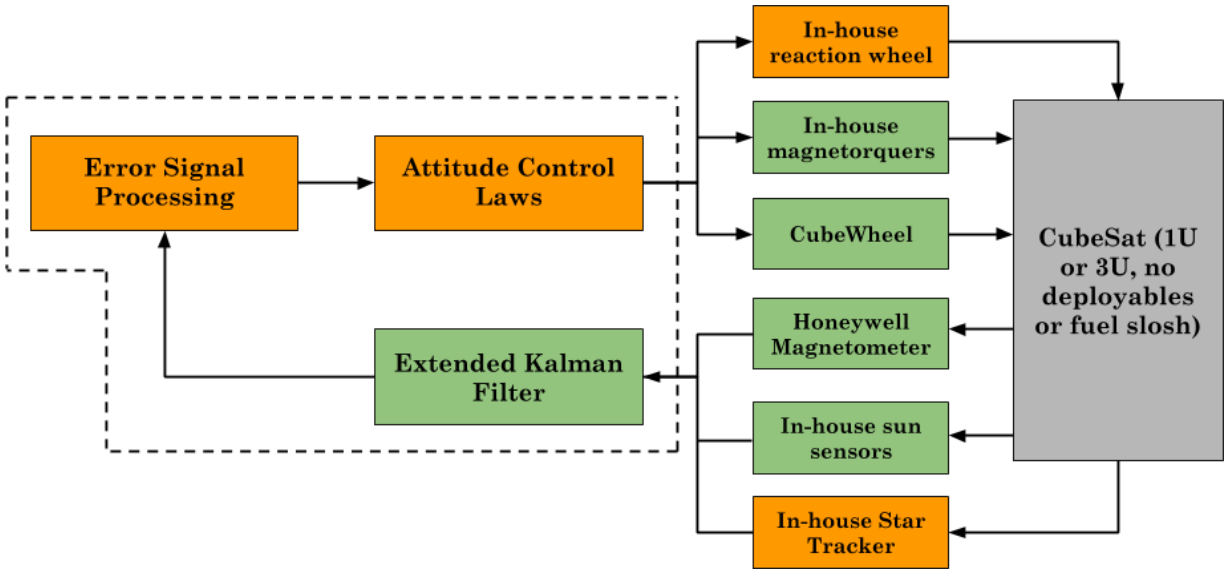

Figure 3: High Level CPCL ADCS Framework

2.1 Sensors and Actuators

In terms of sensors, the CPCL has flight experience with commercial Honeywell magnetometers and will soon be flying in-house designed sun sensors that can be used to measure the sun vector in the spacecraft body frame. Moreover, the CPCL is currently investigating the use of flight software to operate a low-cost camera as a star tracker. Finally, the CPCL in-house main avionics board housing the flight computer also contains a low performance, surface mounted MEMS rate gyroscope. Therefore, from a sensor perspective, the CPCL is

similar to many other universities in its use of magnetometers, sun sensors, and rate gyros, but is beginning to distinguish itself with the use of a low-cost star tracker.

In terms of actuators, the CPCL has flight experience with both in-house magnetorquers and commercial reaction wheels. At the time this thesis is being written, Nick Bonafede (graduate student, mechanical engineering) is developing an in-house reaction wheel for use in the CPCL spacecraft bus. Furthermore, all magnetorquers flown in CPCL missions have been designed, tested, and assembled in-house. From an actuator perspective, the CPCL approach is aligned with the norm in the small satellite community, with the only exception being that most of the hardware is intended to be manufactured and tested in-house instead of purchased commercially.

2.2 Flight Software

In terms of flight algorithms used to generate attitude estimates from sensor measurements, the CPCL has implemented and tested an extended Kalman filter designed to estimate the spacecraft's inertial to body attitude quaternion and total body rates. However, closing the loop on this state estimate has never been done in a rigorous simulation framework, and therefore this thesis will place particular emphasis on guidance and control algorithm design. On the control side, a B-dot detumble control law implementation using the aforementioned magnetometers and magnetorquers has achieved flight heritage. In summary, estimation algorithms at the CPCL are similar to those implemented at other universities, but control algorithms are still in a very early stage and require significant development.

From figure 3 it is apparent that, from an algorithms perspective, more attention has been given to attitude determination than to attitude control. Thanks to the work of multiple insightful master's theses from Sellers [9], Mehrparvar [10], and Bowen [11], as well as a remarkably productive senior project by Bouchard [8], the CPCL has a flight software implementation of a state estimation algorithm as well as the entire software architecture to implement real time GNC algorithms for most conceivable CubeSats missions. Figure 4 below depicts an "object" model of the ADCS flight software architecture as designed and implemented primarily in [8]. Here, "object" is surrounded by quotations because the flight-software is implemented in C, which is not an objected-oriented programming language. However, type abstraction in C is still possible by

appropriately utilizing structs, function pointers, and type casting. Each block in the diagram of figure 4 represents an interface, enforcing consistency across different algorithm implementations, while the arrows represent dependencies. The highest level element in this software architecture is the *ADCS State*, which contains mission specific implementations of *Mission Controller* and *Mission Determination* types. Starting on the control side, the *Mission Controller* is the piece of software responsible for making real time decisions about which *Controller* implementation is to be called each time the *ADCS State* is invoked by the operating system. This enables the spacecraft to switch between control modes in such a way that is consistent with its concept of operations. The *Controllers* contain the flight-software implementations of control algorithms, which calculate a control input and command the spacecraft's actuators to provide that input. The actuator commands as output from *Controller* implementations take place in software as function calls to device drivers implemented in *Actuator* types, which abstractly represent physical hardware. On the determination side, the *Mission Determination* object plays a role analogous to that of the *Mission Controller*: it is responsible for making real time decisions about which *Filters* to invoke. *Filters* are state estimation algorithms like an extended Kalman filter, the implementations of which sometimes require vector references from other sources such an Earth magnetic field model or a sun vector from a sun ephemeris model. Such references are implemented as *Magnetic Model* or *Sun Model* types. Finally, the yellow arrows at the bottom of the diagram represent the flow of data between each branch of the ADCS flight software. Control laws require attitude estimates from the determination algorithm to compute the error signal, but conversely the determination algorithm receives a feedforward signal from the controller (i.e. the torque command) to be used in anticipation of how the state will change as a consequence of the control action. For a more precise and detailed explanation of the flight software architecture, see [8].

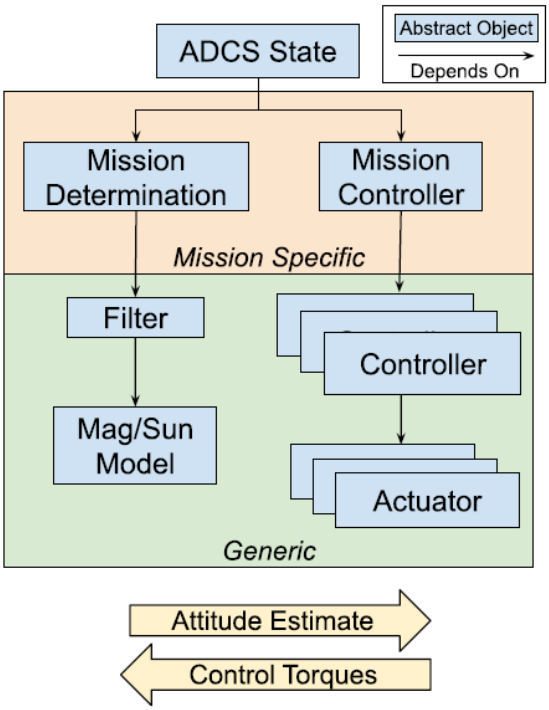

Figure 4: CPCL ADCS Flight Software

At this point, it is assumed that the reader is familiar with the heritage elements—both software and hardware—of the CPCL ADCS. Based on the software infrastructure that currently exists, as well as the sensors and actuators both in development and with heritage, it is clear that as a whole the CPCL is on its way to having a fully functional three axis attitude control system made up of mostly in-house components. However, there is still quite a bit missing. First of all, in terms of flight algorithms, very little thought has been put into the design and testing of attitude control laws and the guidance logic required to compute reference signals. Moreover, at a systems engineering level, there is not a complete set of design and analysis tools that can be readily used by student engineers in the design, test, and integration of an ADCS for any CPCL mission. For example, if different sensors and actuators than those that exist in the aforementioned heritage design are not a good choice, there needs to be a methodology—which includes the appropriate analysis tools—by which new sensors and actuators can be selected and simulated. In the next subsection, a design iteration strategy tailored for use in the CPCL is proposed, with particular emphasis on control law design in the early phases of mission development, and well-informed selection and sizing of sensors and actuators as the design matures. Such a design

iteration strategy will provide engineers with the capacity to make reasonable decisions about sensor and actuator selection as well as algorithm design—perhaps different decisions than those which have been made so far—in the context of the CPCL. Moreover, the overall structure of the CPCL ADCS design and analysis toolkit is motivated by this design iteration strategy.

2.3 Design Iteration Strategy for Future Missions

Figure 5 is a flowchart showing the proposed strategy for iterating on the design of an ADCS for a CPCL space mission, from early concept development to system level validation. Starting in the upper left hand corner with a “conceptual” phase, this is where mission concept of operations development and proposal writing takes place. In this stage, it should be established whether or not the spacecraft even needs an attitude control system, and if so, whether it should be passive, active, or some combination of the two. The mission’s concept of operations will naturally lead to a set of pointing modes that the ADCS shall facilitate. Once the conceptual design is established, more detailed ADCS work can begin in the preliminary design phase. In this phase of mission development, MATLAB/Simulink tools—to be presented and explained thoroughly in this work—can be employed to coarsely size attitude actuators and specify control law behavior in the context of the required pointing modes. In general, these MATLAB/Simulink tools are intended to iteratively bring the design maturity to approximately a PDR level. Once the ADCS satisfies PDR exit criteria, the design and analysis should move into a flight software-in-the-loop environment. It is at this stage in the design process that sensor selection, navigation algorithm design, as well as layout and configuration of sensors and actuators within the spacecraft should be established with more confidence and thoroughly analyzed. Detailed and thorough testing of the sensor and actuator performance should be conducted in order to validate and update models used in the simulation. At the end of this stage, the ADCS sensors and actuators should be well understood in terms of both individual performance and functionality as well as how they interface with the rest of the spacecraft from an electrical, mechanical, and software perspective. Furthermore, flight software implementations of all GNC algorithms should be tested during this phase. The software-in-the-loop simulation environment is intended to iteratively bring the design maturity to approximately a CDR level. Finally, once the ADCS design

has been vetted during CDR and deemed acceptable, Monte Carlo analysis may be required to provide final system level validation. Once the pointing performance of the resulting ADCS design meets is requirement according to Monte Carlo analysis, the design will be mature enough for integration and flight.

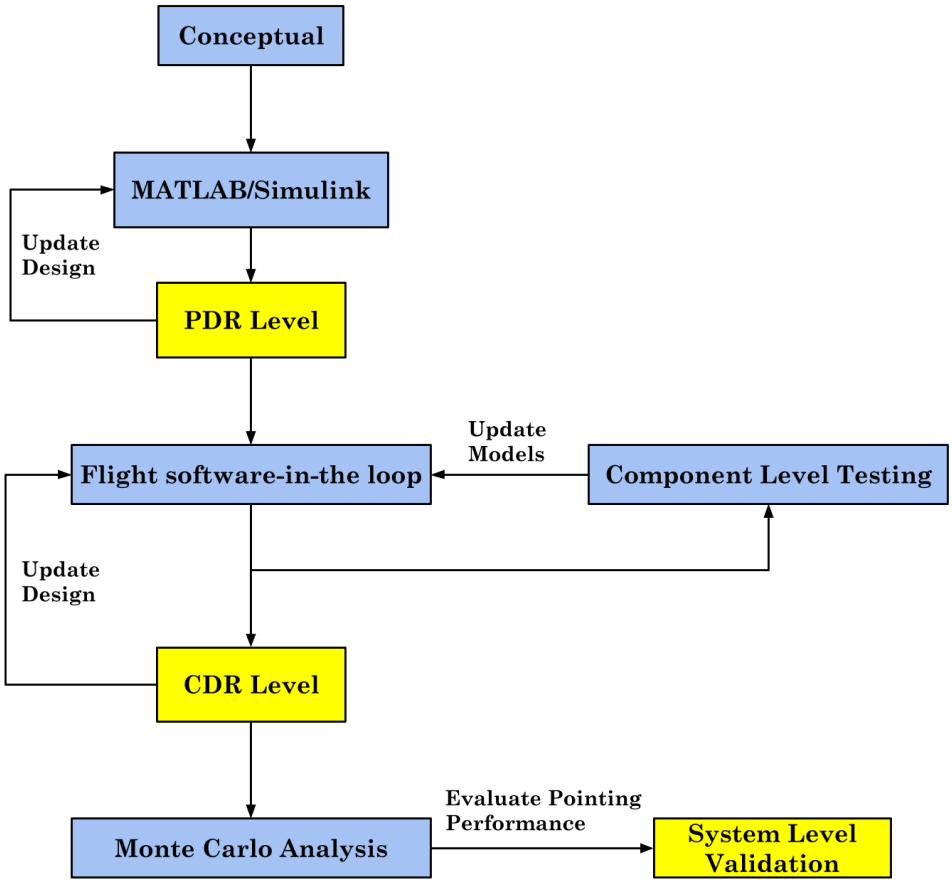

Figure 5: Proposed CPCL ADCS Design Iteration Strategy

As a whole, it should be noted that hesitation to take backward steps in the design process is undesirable. Sometimes it is worthwhile to progress backward in the design in order to correct for poor decisions made earlier in the overall process, or to remove unnecessary elements in the design that add worthless complexity. Such a mentality is sometimes hard to maintain, but in general, it will lead to a more efficient and cheaper system. The author of this thesis strongly recommends that students who design attitude control systems in the CPCL accept this engineering philosophy.

2.4 Software-in-the-loop GNC Simulation Framework

As mentioned in the previous subsection, a software-in-the-loop simulation should be employed during the critical design phase for sensor and actuator selection and layout, as well as eventual verification of flight software functionality and system pointing performance. Fortunately, significant time and effort has been dedicated to developing such a framework for use in the CPCL by past student engineers. Figure 6 provides a block diagram representation of this simulation framework as developed in [8].

A key element in any software-in-the-loop simulation is the truth model. In this case, the CPCL has chosen to take advantage of an open source, high fidelity 6 DOF spacecraft orbit and attitude dynamics simulation called NASA 42, developed by Dr. Eric Stoneking of NASA Goddard. NASA 42 is written in C for speed and portability according to Dr. Stoneking, and can model most conceivable mission scenarios and spacecraft configurations and mass properties, especially for university CubeSat missions. More detail on the simulation's modeling capability and configuration options is provided in chapter 4 of this thesis. The simulation has been intentionally integrated with the flight software so that it interfaces with physical hardware in the same way that it interfaces with simulated hardware in NASA 42. Referring to figure 6, the high level flow of the software-in-the-loop simulation is as follows:

1. Initialize NASA 42 and read all input configuration files
2. Initialize the flight software (all blue boxes)
3. Propagate the state of NASA 42 to a future time and pause the simulation
4. Calculate simulated sensor readings according to sensor models in NASA 42, pass the simulated sensor readings to the PolySat device driver library
5. Execute navigation algorithm with simulated sensor readings as input, pass the resulting state estimate to the control algorithm
6. Execute control algorithm with the state estimate as input and pass the resulting actuator commands to 1) the PolySat device driver library and 2) the navigation algorithm
7. Send actuator commands to NASA 42, go back to step (3)

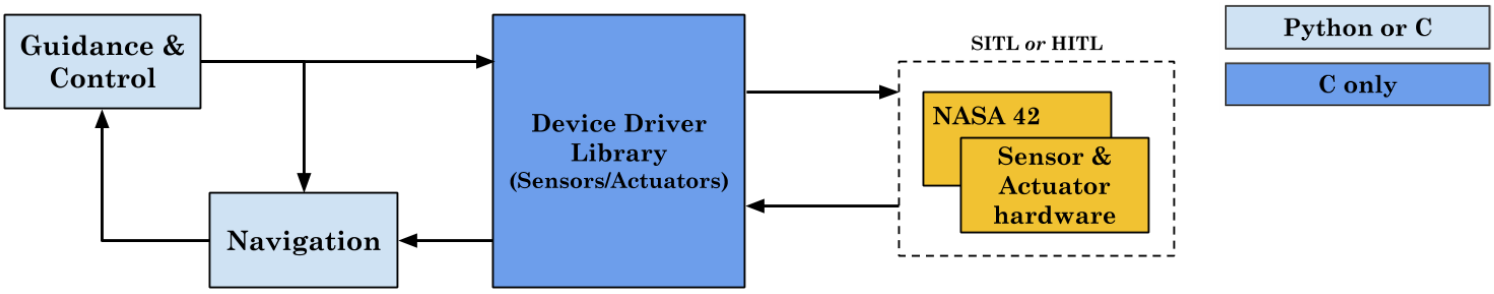

Figure 6: CPCL Software-in-the-loop GNC Simulation Framework

As seen in figure 6, it is also possible to configure the simulation with physical hardware and execute a hardware-in-the-loop test. However, such an activity is beyond the scope of this thesis and will be elaborated upon in the future work section.

An extremely important, high utility feature of this simulation framework is that both navigation and control algorithms can be implemented in python. A binding with python using a C API was established by [8], enabling the algorithms whose fundamental designs are more driven by relatively sophisticated mathematics (e.g. navigation and control) to be implemented in a higher level, more forgiving programming language. If the simulation is built to run with python, then the ADCS flight software makes function calls to python code when it reaches the navigation and/or control steps by utilizing C's python API, and the rest of the process resumes as it would have in C. In this thesis, example python control algorithm implementations for a variety of control modes are successfully implemented in this simulation framework.

In conclusion, it is clear from this chapter that little effort has been given to the formulation of a design methodology for guidance and control algorithms in the CPCL. Instead, much more attention has been given to estimation algorithms, sensor selection, and software-in-the-loop testing. The next chapter will provide formulation of a control algorithm design methodology that will be implemented and utilized in the CPCL ADCS design and analysis toolkit.

Chapter 3

GUIDANCE AND CONTROL ALGORITHM DESIGN PROCEDURE

In this chapter, both guidance and control are addressed within the context of the design and analysis of an ADCS. On the control side, the development of a mathematical procedure to map a set of controller specifications (i.e. design knobs) onto a set of parameters that are necessary in the resulting real-time control algorithm implementations is presented. In terms of guidance, a strategy for computation of the reference quaternion (i.e. the desired attitude) in various control system pointing modes is formulated. In the following chapter, the details of how these guidance and control strategies are embedded into the CPCL ADCS toolkit are provided, and thus the design and implementation of both the single and three axis MATLAB/Simulink tools presented in this work rely heavily on the mathematics to follow. Content in this chapter draws from techniques in [12] and [13] based on classical control theory as applied to the spacecraft attitude control problem. Mathematical reasoning—starting with the nonlinear spacecraft dynamics in three dimensions—followed by linearization and transformation into the frequency domain, will be performed.

3.1 Dynamic Modeling Assumptions and Linearization of the Plant Equations of Motion

In this work, the plant to be controlled is assumed to be an unconstrained rigid body in three dimensions. Although this is not always a good assumption, most CubeSats do not contain deployable features whose flexible dynamics are of importance in the design of the control system. Moreover, if non-negligible flexible modes do exist in a particular spacecraft configuration, they can be readily dealt with in terms of meeting stability margin and pointing performance requirements by appropriate filter design and digital signal processing, which fits very well within the classical control paradigm. Since this work assumes rigid body dynamics in the design of the control laws, care must be taken if the resulting control laws fly on a CubeSat whose structure gives rise to noticeable flexible modes. If that is the case, the designer must consider the following:

- Will flexible modes show up in sensor output or experience excitation by actuator input?

- Given a controller designed with rigid body assumptions, do any flexible modes couple into the open loop frequency response so as to compromise stability margins?

If the answer to either of the above two questions is yes, then it is likely that a digital notch filter will be a necessary addition to the control system. Where this notch filter is placed in the overall architecture of the control system depends on where in the system the flexible modes are more pronounced. For example, if a magnetometer is placed on the edge of a long, flexible solar array, then it is possible that flexible modes of that solar array will be excited by control torques and subsequently show up in the magnetometer output signal. Since the magnetometer readings are used as feedback to the estimation algorithm, these flexible modes may still make it back into the error signal if not removed by the estimator, which is problematic if the controller is not designed to handle them. A flexible body dynamic model, as obtained through appropriate finite element analysis, should inform the designer how changes in control inputs result in changes in mass element deflections at each point on the solar array. With this model, the flexible modes of the solar array at the location of the magnetometer can be recovered, and then removed from the output signal of the magnetometer with an appropriately designed notch filter. Overall, these types of considerations must be made on a per-mission basis, based on seemingly extraneous features of the spacecraft design.

The dynamics of a rigid body are governed by Euler's equations, which describe how torques influence the attitude motion of the rigid body in three dimensions. Euler's equations can be written in vector form as

$$\boldsymbol{T} = \dot{\boldsymbol{h}} + \boldsymbol{\omega}\, \boldsymbol{x}\, \boldsymbol{h}$$

Where $\boldsymbol{h}$ is the total angular momentum vector of the rigid body with respect to inertial space expressed in body coordinates, $\boldsymbol{\omega}$ is the total angular velocity vector of the rigid body with respect to inertial space (also expressed in body coordinates), and $\boldsymbol{T}$ is the net torque vector acting on the rigid body expressed in body coordinates. Several mathematical parameterizations of the attitude exist, but in this work the quaternion is chosen as it provides several advantages—most notably algebraic, nonsingular kinematics—over other parameterizations. The quaternion is a four dimensional quantity that is directly related to the axis around which and the angle through which

one reference frame must be rotated so that it coincides with another reference frame. Therefore it describes the three dimensional orientation of one reference frame with respect to another reference frame. The attitude of a rigid body with respect to the inertial reference frame can be parameterized in terms of a quaternion $\boldsymbol{q}$ as

$$\boldsymbol{q} = \eta + \boldsymbol{\epsilon}$$

Where $\boldsymbol{\epsilon}$ is related to the eigenaxis and angle of the frame transformation from the inertial reference frame to a body fixed reference frame by

$$\boldsymbol{\epsilon} = \sin\left(\frac{\phi}{2}\right)\boldsymbol{a}$$

Where $\boldsymbol{a}$ is the three dimensional unit vector that represents the eigenaxis of the transformation encoded by the quaternion, and ϕ is the angle through which the transformation rotates the inertial frame around $\boldsymbol{a}$. In this work, quaternions are interpreted as *frame rotations*, which means they do not rotate physical vectors with respect to inertial space when transforming them. The scalar part η of the quaternion is given by

$$\eta = \cos\left(\frac{\phi}{2}\right)$$

It is important to note that a quaternion must have unity magnitude in the Euclidean norm sense, otherwise it does not represent a pure rotation. Under the definition described above, the quaternion is guaranteed to have a Euclidean norm of one.

The attitude kinematics associated with the quaternion parameterization are given by

$$\dot{\boldsymbol{\epsilon}} = \frac{1}{2}(\eta \boldsymbol{I} + \boldsymbol{\epsilon}^{x})\boldsymbol{\omega}$$

$$\dot{\eta} = -\frac{1}{2}\boldsymbol{\epsilon}^{T}\boldsymbol{\omega}$$

Where the superscript x over any vector quantity represents the cross-product matrix built from that vector, the T superscript represents the transpose of the vector, and $\boldsymbol{I}$ is the 3 by 3 identity matrix.

Since reaction wheels are likely to be the primary choice of attitude actuator in almost all CubeSat missions that require three axis pointing, it is important to understand how they play into

the rigid body dynamics as described by Euler's equations. In order to fully capture the dynamics of a rigid body under the influence of reaction wheels, the instantaneous momentum vector of the reaction wheel system must be included in the net momentum vector of the wheel plus spacecraft system before applying Euler's equations. For a rigid body containing N reaction wheels mounted in arbitrary orientations with respect to the body frame, the total angular momentum vector of the wheel plus spacecraft system is

$$\boldsymbol{h}_{\text{total}} = J\boldsymbol{\omega} + I_{ws}\boldsymbol{A}_s\boldsymbol{\Omega}$$

Where J is the inertia matrix of the rigid body expressed in a body fixed frame including the reaction wheel mass, I_{ws} is the scalar moment of inertia around the spin axis of each wheel (assuming all wheels are identical), $\boldsymbol{A}_s$ is 3 by N matrix that transforms wheel space angular momentum vectors into body space angular momentum vectors, and $\boldsymbol{\Omega}$ is an N by 1 vector of the angular rates of each wheel. Substituting this expression for the total angular momentum of the rigid body and reaction wheel system into Euler's equations yields

$$\boldsymbol{T} = J\dot{\boldsymbol{\omega}} + I_{ws}\boldsymbol{A}_s\dot{\boldsymbol{\Omega}} + \boldsymbol{\omega}\ x\ (J\boldsymbol{\omega} + I_{ws}\boldsymbol{A}_s\boldsymbol{\Omega})$$

A control law that commands reaction wheels seeks to compute a moment which enters into the rigid body dynamics according to the $I_{ws}\boldsymbol{A}_s\dot{\boldsymbol{\Omega}}$ term in the above nonlinear vector differential equation. An appropriate angular acceleration $\dot{\boldsymbol{\Omega}}$ is then calculated such that the actual momentum exchange applied to the rigid body is as close as possible to the commanded control moment. In most reaction wheel designs, the motor driver attached to the flywheel is configured to control the angular speed of the wheel, not the angular acceleration. Therefore, each time a new torque command is requested from the attitude controller, multiple speed set points must be obtained by discrete-time integration of the torque command divided by the wheel's moment of inertia around its spin axis. The speed set points are then sent to the internal wheel control electronics. Fortunately, this is not difficult to implement in software, and its effects can be captured in simulation by modeling the reaction wheel dynamics from speed commands to true speed using a first order transfer function with time constant τ. The transfer function from wheel

speed command (e.g. the integral of the torque requested by the attitude controller divided by the wheel inertia) to actual wheel speed has a unity DC gain and a single pole at $s = -\frac{1}{\tau}$:

$$G(s) = \frac{1}{\tau s + 1}$$

Where τ is the time required for the true wheel speed to converge to within ~63% of its commanded value.

It is clear that both the rigid body dynamics and the quaternion kinematics are nonlinear. In order to apply classical control theory, the dynamics of the plant to be controlled must be linear and time invariant. Therefore, to obtain a linear time invariant dynamics model, a linearization of the nonlinear dynamics and kinematics about an equilibrium state of particular interest is performed. For analysis purposes, the equilibrium state corresponding to the home quaternion and zero angular velocity is chosen:

$$\boldsymbol{q} = 0\hat{\imath} + 0\hat{\jmath} + 0\hat{k} + 1$$

$$\boldsymbol{\omega} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Although it may not always be desirable to regulate the system to this state, it is still a good approximation of the desirable steady-state behavior in most attitude control modes. Furthermore, any constant, non-zero quaternion relating the attitude of the rigid body to the inertial frame can be represented as the zero quaternion after an appropriate redefinition of the inertial frame. This means that the attitude dynamics are the same local to any constant quaternion relating the body frame to the inertial frame, even if the two frames are not aligned. Such invariance of the attitude dynamics (i.e. the Newtonian physics of the problem) with respect to constant quaternions should not come as a surprise, because a constant quaternion is a purely kinematic quantity.

When the above equilibrium state is substituted into the full nonlinear dynamics, and the gyroscopic torque due to non-zero reaction wheel momentum is neglected (this assumption will be revisited), the following set of differential equations are recovered:

$$\mathbf{0} = J\dot{\boldsymbol{\omega}} + I_{ws}\boldsymbol{A}_s\dot{\boldsymbol{\Omega}}$$

$$\dot{\boldsymbol{\epsilon}} = \frac{1}{2}\boldsymbol{\omega}$$

$$\dot{\eta} = 0$$

The vector differential equations above are decoupled as long as the eigenvectors of the inertia matrix J are nearly parallel to the basis vectors of the body frame. It turns out that it is always possible to find a body fixed frame in which the inertia matrix is diagonal. Such a frame is called the principle frame. However, it is not always convenient to define the body frame to be controlled as the principle frame, especially when the principle axes are not aligned with interesting features of the spacecraft (e.g. sensor boresights, solar array normal vectors, communication beams, or thrust vectors). For spacecraft of very asymmetrical geometry and mass distribution, the principle frame is likely to be significantly misaligned with the control frame of interest. Conveniently, due to the symmetric geometry and mass properties of CubeSats, it is reasonable to assume that J can be well approximated by a diagonal matrix in a body fixed frame whose basis vectors are perpendicular to the faces of the CubeSat. Such a frame is also usually very convenient from the perspective of pointing. A strategy for dealing with cases in which J cannot be assumed diagonal will be dealt with in a future section. With this assumption, the dynamics around each axis can be treated separately to yield three single input single output (SISO) linear dynamic systems. To see how this is possible, notice that in the linearized kinematics, the first derivative of the angular velocity vector is equal to twice the second derivative of the vector part of the quaternion. Substituting twice the second derivative of the vector part of the quaternion for the first derivative of the angular velocity in the linearized dynamics will yield three linear second order systems with two poles at the origin, namely a double integrator process. For the sake of example, focusing on the x-axis gives

$$2I_x\ddot{\epsilon}_x = u_x$$

Where u_x is the x-component of the control input, I_x is the moment of inertia around the principle x axis, and $\ddot{\epsilon}_x$ is the second time derivative of the x component of $\boldsymbol{\epsilon}$. Taking Laplace transforms yields

$$Y(s) = E_x(s) = \frac{1}{2I_x s^2} U_x(s)$$

Where $Y(s)$ is the x-component of the vector part of the quaternion. At this point, the dynamics have been reduced to three SISO systems, all with the same form of plant transfer function (e.g. a double integrator). Therefore, it is really only necessary to solve the control problem once, and then repeat the same procedure for each axis. In the following section, the formulation of the control law design procedure is provided for a single axis.

3.2 Discrete Time PID Controller Design

For this work, the single axis unity feedback block diagram in figure 7 provides the basis for the formulation of the control algorithm design procedure to be implemented in the toolkit. In this diagram, $R(s)$ is the reference signal, T is the sampling period of the digital attitude controller $G_c(z)$, I is the axial moment of inertia, s is the Laplace variable, and $\theta(s)$ is the attitude angle.

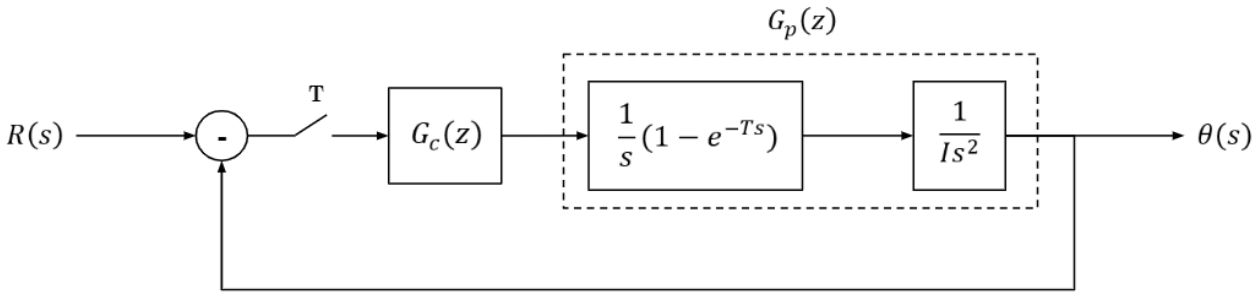

Figure 7: Closed Loop Model for Control Algorithm Design

The design of the digital controller $G_c(z)$ is performed by first obtaining an exact discrete time equivalent (z-domain) model $G_p(z)$ of the zero-order hold plus double integrator plant. The pulse transfer function between the input to the zero-order hold and the ideally sampled version of the output $\theta(s)$ can be obtained with the following sum:

$$G_p(z) = (1 - z^{-1}) \sum_{poles\ of\ X(\lambda)} Res\left[X(\lambda)\frac{1}{1 - z^{-1}e^{T\lambda}}\right]$$

Where the $Res[\,]$ operator denotes taking the residue of the function in the argument. For a derivation of the above, the reader is referred to [13], but a short explanation is that the Residue

Theorem is being applied to evaluate the contour integral of $X(\lambda)$ around its poles. In the case of the control system in figure 7, the transfer function $X(\lambda)$ is given by

$$X(\lambda) = \frac{1}{I\lambda^3}$$

$X(\lambda)$ is a triple integrator because it picked up an extra integrator from the transfer function of the zero order hold. Calculating $G_p(z)$ according to the expression above gives the discrete time transfer function between inputs to the zero order hold and the sampled plant output:

$$G_p(z) = \frac{T^2}{2I}\frac{(z+1)}{(z-1)^2}$$

With an exact discrete time model of the sampled response of the zero-order hold plus double integrator plant, there are two design directions that can be pursued. Either the digital controller $G_c(z)$ can be designed around the discrete time plant directly in the z-domain, or a continuous time approximation to $G_p(z)$ can be obtained, allowing for the controller to be designed in the more familiar continuous time domain, and then implemented digitally after applying an appropriate analog to digital mapping. In this work, the continuous time control design approach is taken as it is more intuitive and much easier to deal with in terms of the resulting open-loop transfer function. During the design process, the continuous frequency domain corresponding to the controller and plant will be referred to as the *w-domain* instead of the *s-domain*. The reason for this is that the continuous time approximation of the exact discrete-time model derived earlier will not be identical to the original dynamic system as defined in the *s-domain*. In fact, the original system in the *s-domain* is nonlinear due to the zero-order hold and thus cannot be treated with classical control techniques. Therefore, using a new variable resolves potential confusion. Moreover, since the frequencies in the *w-domain* are not exact analog frequencies, it is even more natural to utilize a different variable. More specifically, exact analog frequencies in the *s-domain* above the Nyquist frequency are lost upon discretization of the continuous time plant, but reappear as warped versions of themselves when the discrete time model is reconstructed in the *w-domain*. Mathematically, frequencies in the *w-domain* ω_w are related to exact analog frequencies ω after a bilinear transform by

$$\omega_w = \frac{2}{T}\tan\left(\frac{\omega T}{2}\right)$$

From the above expression it is clear that for analog frequencies much less than the Nyquist frequency, the frequency warping is minimal. In other words,

$$\omega_w \approx \omega \text{ when } \omega T << 1$$

This relationship is important, because it tells us that if the bandwidth of the closed loop system is kept sufficiently lower than the Nyquist frequency, the true analog frequency response will be well approximated in the *w-domain*, thus validating the use continuous time design techniques.

Approximation of the discrete time plant $G_p(z)$ in the continuous *w-domain* is performed by utilizing the bilinear transform. The bilinear transform approximates the discrete complex variable $z = e^{wT}$ using a Taylor Expansion to first order, yielding

$$z \approx \frac{1 + \left(\frac{T}{2}\right)w}{1 - \left(\frac{T}{2}\right)w}$$

The advantage of this approximation is that the resulting transfer function can always be written as a ratio of polynomials in w, which is necessary for classical control design techniques. Substituting the above expression for z into $G_p(z)$ gives

$$G_p(w) = \frac{2 - Tw}{2Iw^2}$$

Notice that the *w-domain* expression above, which approximates the discrete time model given by $G_p(z)$, is remarkably similar to a simple double integrator, with the exception of an additional zero that lies below the real axis in the right-half plane, since $T > 0$ by definition. This zero is capturing the phase lag incurred by the zero-order hold in the original nonlinear continuous time system. Although this zero is in the right half plane and thus the *w-domain* system is non-minimum phase, it is after all an approximation. Therefore, the control design techniques that are typically required to deal with non-minimum phase systems do not apply in this context. Moreover, using the above plant $G_p(w)$ in the control design adds sample rate dependence to the computation of the controller gains, which is a desirable property of the overall control design procedure.

In this context, the controller acts on the quaternion error signal at its input and generates body torque commands as its output. The PID controller is designed to achieve a desired phase margin θ_M and open loop gain crossover frequency ω^* in the *w-domain*. The gains K_p, K_D, and K_I are calculated by first considering how the parameters T_{PD} and T_{PI} are related to the frequencies of the two open loop zeros that are added to the open loop transfer function by the controller. In this context, the open loop transfer function $G_O(w)$ is given by the product of the PID controller transfer function and the plant transfer function:

$$G_O(w) = G_P(w)G_C(w) = \frac{K}{2Iw^3}(2 - Tw)(T_{PD}w + 1)(T_{PI}w + 1)$$

Each first order term in the PID controller transfer function of the form $Tw + 1$ adds a zero to the open loop transfer function at frequency $\frac{1}{T}$. Due to the integrator of the controller as well as the double integrator from the plant, the two zeros of the PID controller must collectively compensate for 270° of open loop phase lag at all frequencies, as well as the phase lag from the zero-order hold, to achieve the desired phase margin. The parameters T_{PD}, and T_{PI} can be viewed as knobs that change the shape of the open loop gain and phase frequency response by adjusting the frequency locations of the open loop zeros, since they are just the inverses of the zero frequencies. The gain K can be viewed as a knob that moves the entire open loop gain frequency response up by $20\log_{10} K$ dB. With this in mind, the following strategy is employed in mapping the control system specifications θ_M and ω^* to the PID gains K_p, K_D, and K_I:

1. Place the PI zero one decade before the gain crossover frequency ω^* with

$$T_{PI} = \frac{10}{\omega^*}$$

2. Place the PD zero so that when the open loop gain is ω^*, the open loop phase is θ_M greater than -180° using

$$T_{PD} = \left(\frac{1}{\omega^*}\right)\tan(\,90^\circ + \theta_M + \tan^{-1}\left(\frac{T\omega^*}{2}\right) - \tan^{-1}(T_{PI}\omega^*))$$

3. Finally, compute the gain K so that the open loop gain is $0\ dB$ at frequency ω^* using

$$K = \frac{1}{\left|\frac{T_{PI} i\omega^* + 1}{i\omega^*}\right| \left|T_{PD} i\omega^* + 1\right| \left|\frac{2 - i\omega^*}{2I_a (i\omega^*)^2}\right|}$$

Where i is the imaginary unit, the vertical brackets denote the magnitude of a complex number, and I_a is the moment of inertia around the axis of interest. After steps (1) through (3), the gains K_p, K_D, and K_I are computed using

$$K_I = K$$

$$K_p = K(T_{PD} + T_{PI})$$

$$K_D = K T_{PD} T_{PI}$$

Once the PID gains are calculated, the control law can be implemented in software fairly easily by discretizing the controller transfer function with backward Euler differentiation and trapezoidal integration. After calculation of the above controller gains, the digital control algorithm implements

$$G_c(z) = K_p + \frac{K_I T_s (z+1)}{2(z-1)} + \frac{K_D (z-1)}{T_s z}$$

This is equivalent to applying the bilinear transform to the integral state of the *w-domain* control law and applying a backward Euler discretization to the derivative part of the control law.

3.3 Reference Filter and Derivative Filter Design

Two additional elements of the proposed control system design are two first order digital low pass filters, one after the reference signal and one before the derivative term in the control law. Both of these filters have their own purpose. The derivative filter prevents the controller from amplifying high frequency noise that may be present in the error signal, primarily coming from the feedback path. It can be used to bend the open loop gain response further downward after crossover, which increases noise attenuation performance. The reference filter, on the other hand, removes high frequencies (such as in step changes) in the reference signals to which the controller may respond too aggressively relative to the strength of the actuators in the system. Moreover, the reference filter could also be designed to notch out flexible modes that may be

present in the plant dynamics, but in this work, mitigation of large step changes in attitude commands—which will occur during operational transitions between control modes—is the primary design goal for the reference filter. Both filters are first designed in the continuous frequency domain and then mapped into the discrete frequency domain with the bilinear transform. An analog, first order low pass filter has a transfer function of the form

$$H(s) = \frac{\omega_c}{s + \omega_c}$$

Where ω_c is the cutoff frequency of the filter—the single design parameter at hand—in rad/s. Applying the bilinear transform substitutes the following approximation for s in terms of the discrete frequency variable z into $H(s)$ to yield a discrete time transfer function $H(z)$:

$$s \approx \frac{2(z-1)}{T(z+1)}$$

The resulting discrete time transfer function is given by

$$H(z) = \frac{b(z+1)}{z-a}$$

Where the coefficients a and b are related to the sampling period T and cutoff frequency ω_c by

$$a = \frac{\frac{2}{T} - \omega_c}{\frac{2}{T} + \omega_c}$$

$$b = \frac{1}{1 + \frac{2}{\omega_c T}}$$

The following difference equation in terms of the coefficients a and b is then used to implement the first order filter in discrete time:

$$y_k = a y_{k-1} + b(u_k + u_{k-1})$$

Where the subscript k indexes the time step, y is the filter output, and u is the filter input.

In designing the derivative filter, it is important that the chosen gain crossover frequency be at least one decade less than the filter's cutoff frequency so that the filter only affects the

relatively high frequency portion of the open loop frequency response and does not significantly disrupt the phase margin. In other words,

$$10\omega_{gc} \leq \omega_c$$

The reference filter, on the other hand, has no similar constraint. However, when attempting to filter out high frequencies as a result of large step changes in the attitude reference signal, a reasonable choice for the cutoff frequency of the reference filter is

$$\omega_{gc} \leq 2\omega_c$$

Furthermore, it does not make sense to place the reference filter cutoff frequency significantly lower than the gain crossover frequency, because the interesting frequencies of the *input* signal to the reference filter are designed to be tracked by the closed loop system by virtue of where the gain crossover frequency is placed. In other words, placing the cutoff frequency of the reference filter significantly below the gain crossover frequency could prevent the output of the closed loop system from tracking the interesting part of its input. The above rule of thumb provides a sensible balance between keeping the reference filter cutoff frequency high enough so that the closed loop system tracks frequencies of interest, while also keeping the reference filter cutoff frequency low enough to prevent large discontinuities in actuator commands.

Figure 8 below summarizes the control system architecture as developed in this chapter. The engineer can adjust six control design knobs within this architecture: the sample rate, open loop gain crossover frequency, phase margin target, reference filter cutoff frequency, derivative filter cutoff frequency, and the integral saturation limit.

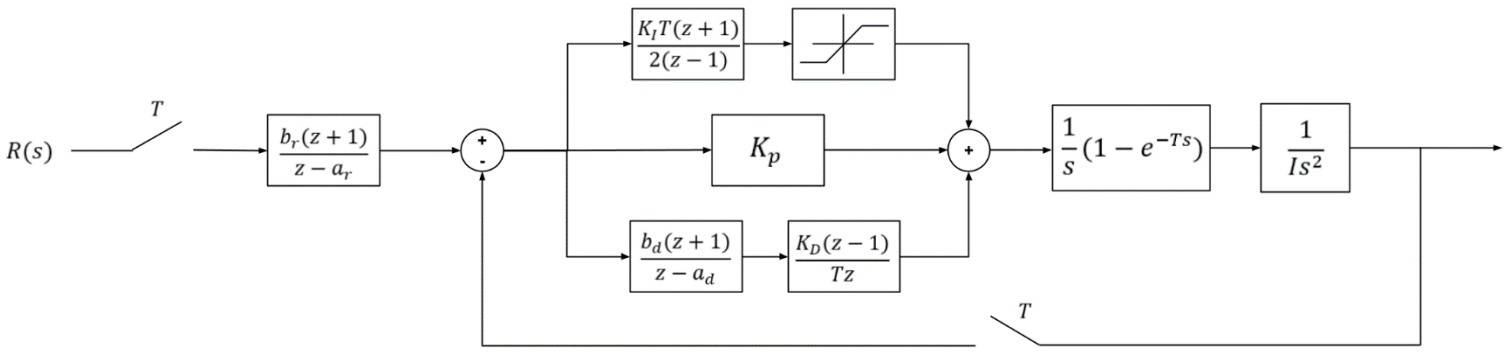

Figure 8: Single Axis Closed-Loop Attitude Control System Architecture

3.4 Feedback Linearization

We will now revisit the previous assumption that the total gyroscopic torque in Euler's equations due to reaction wheel momentum can be neglected, and that J is diagonal. In general, when the spacecraft angular velocity is sufficiently small and the spacecraft mass distribution is symmetric about each plane of the body axes, this assumption is valid. However, if a high control system bandwidth is desired for a particular mission, then spacecraft angular velocities may not stay small enough for a valid dynamics linearization. Furthermore, if the spacecraft mass distribution is asymmetrical enough to give rise to a large rotation angle between the principle frame and the body control frame, the dynamics around each axis can no longer assumed to be decoupled. Fortunately, it is possible to actively "cancel" this nonlinearity in Euler's equations and decouple the system by adding a feedback linearization signal to the control law. For simplicity, the following continuous time representation of such a nonlinear control law would look like

$$\boldsymbol{u} = K_p\boldsymbol{e} + K_I\int \boldsymbol{e}dt + K_D\dot{\boldsymbol{e}} - \boldsymbol{\omega}\, x\, (\boldsymbol{J\omega} + I_{ws}\boldsymbol{A}_s\boldsymbol{\Omega}) = \boldsymbol{u}_L + \boldsymbol{u}_{NL}$$

This makes our assumption about the negligible gyroscope torque more reasonable. In the above, $\boldsymbol{e}$ is the attitude error, and the means by which it is calculated depends on the overall goal of the controller (e.g. the ADCS operational mode). The control law is partitioned into two signals with different design objectives. The first signal $\boldsymbol{u}_L$ is the linear part, whose design is carried out according to the previously mentioned procedure, and its overall goal is to drive the attitude errors to zero. The second signal $\boldsymbol{u}_{NL}$ is the nonlinear part, whose goal is to cancel the nonlinear gyroscopic torques, and force the closed-loop dynamics to behave more similarly to how they were modeled when designing $\boldsymbol{u}_L$. More explicitly,

$$\boldsymbol{u}_L = K_p\boldsymbol{e} + K_I\int \boldsymbol{e}dt + K_d\dot{\boldsymbol{e}}$$

And

$$\boldsymbol{u}_{NL} = -\boldsymbol{\omega}\, x\, (\boldsymbol{J\omega} + I_{ws}\boldsymbol{A}_s\boldsymbol{\Omega})$$

It is important to note that the sign of the nonlinear control $\boldsymbol{u}_{NL}$ depends on the type of actuators being used to generate the control signal. In the above, it is assumed that reaction wheels are the primary control actuator, since in that case

$$I_{ws}\boldsymbol{A}_s\dot{\boldsymbol{\Omega}} = \boldsymbol{u} = \boldsymbol{u}_L + \boldsymbol{u}_{NL}$$

So in order for the $\boldsymbol{\omega}\ x\ (\boldsymbol{J\omega} + I_{ws}\boldsymbol{A}_s\boldsymbol{\Omega})$ term in Euler's equations to vanish, the control should have the opposite sign. Assuming perfect inertial knowledge, angular velocity knowledge, and wheel rate knowledge, the closed loop differential equations become

$$-\boldsymbol{u}_L = 2\boldsymbol{J}\ddot{\boldsymbol{\epsilon}}$$

Which is consistent with the assumptions made when designing three decoupled reaction wheel PID controllers around double integrator plants. Also note that the closed loop differential equations above assume that the control enters into the dynamics on the right-hand-side of Euler's equations, which is only true when momentum exchange actuators like reaction wheels are utilized. This is the reason for the negative sign on the left-hand side. If the actuators provide external torques as opposed to momentum exchanges, the sign of the total control signal needs to be flipped. Keen awareness of the correct sign in this context is a crucial step in avoiding errors which can easily lead to a loss of closed-loop stability.

3.5 Control System Operational Modes

In this section, the mathematical formulation behind the computation of the reference attitude quaternion for each of the following three-axis operational modes will be provided:

- Inertial Pointing
- Nadir Pointing
- Ground Pointing
- Ephemeris Pointing
- Spin Pointing
- Desaturate

Each of these control modes will be dealt with sequentially, starting with inertial pointing. This section addresses the design of the various guidance algorithms required to provide consistent and well defined reference signals to the attitude control system. Preliminary simulation results

show that the PID controller design presented in the previous section provides good closed loop performance for all of these modes.

3.5.1 Inertial Pointing

Inertial pointing is by far the simplest of these operational modes. In inertial pointing, the goal of the control system is to drive the attitude to a constant commanded inertial to body quaternion. The particular quaternion target in question is dependent on the mission and operational scenario, and will be a parameter that can be commanded from the ground or baked into a flight software configuration file. The attitude error vector for inertial pointing is computed as

$$\boldsymbol{e} = 2\boldsymbol{\epsilon}(q_r^* q)$$

Where q_r is the commanded inertial to body quaternion, q is the estimated inertial to body quaternion, and the $\boldsymbol{\epsilon}(\ )$ denotes that only the vector part of the resulting product is taken as the attitude error. The * operator corresponds to quaternion conjugation, which simply involves flipping the sign on the vector part of the quaternion.

The factor of two present in the attitude error computation is to ensure that for small error angles, the attitude error is directly proportional to the rotation angles around each principle axis required to align the two frames. To see this, consider that

$$\boldsymbol{\epsilon}(q_r^* q) = \mathbf{a}\sin(\phi_e/2)$$

For small ϕ_e, this reduces to

$$\boldsymbol{\epsilon}(q_r^* q) \approx \frac{1}{2}\boldsymbol{a}\phi_e$$

The factor of one-half in the above is problematic. It means that if the factor of two is not included in the attitude error computation, then the proportional control signal picks up an unwanted gain of one-half relative to the single axis plant model used in the design of the controller. This will result in closed loop behavior that is inconsistent with the original specifications.

3.5.2 Nadir Pointing

The next mode presented is nadir pointing. This mode is extremely important, because it allows the spacecraft to align itself with a non-inertial reference frame defined by the

instantaneous orbit state. Therefore the spacecraft attitude profile is periodic with respect to the orbit period, permitting a steady state "downward looking" behavior. As a result, the attitude error computation is much more involved than in inertial pointing. First, the rotation matrix from the inertial to orbit frame is constructed by computing the basis vectors of the orbit frame one at a time, each expressed in the inertial frame. The unit vector along the z axis of the orbit frame is defined as

$$\hat{\boldsymbol{z}} = \frac{-\boldsymbol{R}}{|\boldsymbol{R}|}$$

Where $\boldsymbol{R}$ is the inertial position vector of the spacecraft. Based on this calculation, $\hat{z}$ is a unit vector pointing in the geocentric nadir direction. The unit vector along the y axis of the orbit frame is the negative of the orbital angular momentum:

$$\hat{\boldsymbol{y}} = \frac{-\boldsymbol{h}}{|\boldsymbol{h}|} = \frac{-(\boldsymbol{R}\ x\ \boldsymbol{V})}{|\boldsymbol{R}\ x\ \boldsymbol{V}|}$$

And finally, the x axis completes the right-handed orthogonal triad with

$$\hat{\boldsymbol{x}} = \hat{\boldsymbol{y}}\ x\ \hat{\boldsymbol{z}}$$

The rotation matrix from the inertial frame to the orbit frame is given in terms of these three basis vector as

$$C = \begin{bmatrix} \hat{\boldsymbol{x}}^T \\ \hat{\boldsymbol{y}}^T \\ \hat{\boldsymbol{z}}^T \end{bmatrix}$$

The quaternion corresponding to the same frame transformation encoded by C is the reference quaternion. It can be calculated in terms of C with

$$\eta_r = \frac{1}{2}\sqrt{1 + tr(C)}$$

$$\epsilon_{rx} = \frac{C_{23} - C_{32}}{4\eta_r}$$

$$\epsilon_{ry} = \frac{C_{31} - C_{13}}{4\eta_r}$$

$$\epsilon_{rz} = \frac{C_{12} - C_{21}}{4\eta_r}$$

Where the $tr(C)$ operator denotes the trace of the matrix C, and the subscripts C_{ij} denote the element of C at row i and column j. Then the reference quaternion can be built from the elements computed in the previous four equations. However, there is a subtle issue that can arise when computing the reference quaternion in this manner. It turns out that quaternions are not unique, because q and $-q$ correspond to the same physical attitude. Because of this non-uniqueness, computation of the quaternion from the corresponding rotation matrix C along an orbit will necessarily result in a discontinuous transition from q to $-q$ at some point in the orbit. Even though q and $-q$ correspond to the same physical attitude, the attitude error will change sign in one control cycle if this sign change occurs, which will lead to undesirable closed loop behavior and possibly a loss of stability. To ensure that this never happens, a unit delay is applied to the vector part of the reference quaternion computed at one control cycle, and dotted with the vector part of the reference quaternion at the next control cycle. In other words, the following quantity x is computed

$$x = \boldsymbol{\epsilon}_r \bullet (z^{-1}\boldsymbol{\epsilon}_r)$$

If x is less than zero, it means that the aforementioned point in the orbit at which the reference quaternion flips sign has been reached. To force continuity in the reference quaternion profile, the sign of the reference quaternion is flipped if x is less than zero. The next step in computing the error quaternion is identical to the inertial pointing case using the new reference quaternion:

$$\boldsymbol{e} = 2\boldsymbol{\epsilon}(q_r^* q)$$

It should be noted that under this formulation of nadir pointing, the spacecraft's body y axis will be aligned with the negative orbital angular momentum vector, and the body z axis will be the nadir pointing axis. If a different alignment is desired, all that is required is an appropriate re-definition of the basis vectors of the orbit frame expressed in the inertial frame. For example, if it is desired that the body x axis of the spacecraft be aligned with the nadir vector, then the x-axis of the orbital frame should be defined as the nadir vector, while swapping the other two axes so that right-handedness is preserved. Finally, the attitude error computation presented for this

mode corresponds to geocentric pointing, not geodetic pointing. This is because it assumes the Earth is a perfect sphere. Since the Earth is not a perfect sphere, there will be a small pointing error between the body fixed pointing vector and the point on the Earth that intersects with the spacecraft position vector. Implementation of geodetic pointing is beyond the scope of this thesis but will be elaborated upon in the future work chapter.

3.5.3 Ground Pointing

In ground pointing mode, the attitude control system's goal is to align a commanded body fixed pointing axis with a vector pointing from the spacecraft to a commanded location on the surface of the Earth, specified in latitude and longitude coordinates. The reference quaternion is therefore underdefined by one degree of freedom, namely the rotation angle of the spacecraft around the body fixed pointing vector. In this work, the rotation angle around the body fixed pointing vector is defined such that a secondary spacecraft body vector will stay in the orbit plane, and the third axis completes the right-handed orthogonal triad. Defining the third degree of freedom in this manner is somewhat arbitrary, and may be easily redefined by replacing the velocity vector in the following formulation with a different vector of interest

The reference quaternion in this mode is computed by first forming the inertial to body rotation matrix defining the desired inertial to body attitude. The rows of the desired inertial to body rotation matrix are simply the positions of the desired body vectors expressed in inertial coordinates. Therefore, the body pointing axis of interest, which in this work is restricted to either the +X, +Y, or +Z axis, should occupy the corresponding row of the desired inertial to body rotation matrix. The unit vector pointing from the spacecraft to the commanded location on the ground, expressed in inertial coordinates, is given by

$$\widehat{\boldsymbol{T}} = \frac{\boldsymbol{\rho} - \boldsymbol{R}}{|\boldsymbol{\rho} - \boldsymbol{R}|}$$

Where $\boldsymbol{\rho}$ is the inertial position vector of the commanded location on the ground, and $\boldsymbol{R}$ is the spacecraft inertial position vector. In order to obtain $\boldsymbol{\rho}$ given only latitude and longitude coordinates, the Earth centered Earth fixed (ECEF) reference frame must be utilized. In an ECEF frame where the x axis points along zero geodetic longitude and the z axis points outward from

the North Pole, the position $\boldsymbol{\rho}$ is constant and is given by the spherical to rectangular coordinate transformation

$$\boldsymbol{\rho}_{ECEF} = \begin{bmatrix} R_e \cos(\lambda) \cos(\theta) \\ R_e \cos(\lambda) \sin(\theta) \\ R_e \sin(\lambda) \end{bmatrix}$$

Where R_e is the radius of the Earth, λ is latitude of the desired ground location, and θ is the longitude of the desired ground location. Then, $\boldsymbol{\rho}$ is transformed into inertial coordinates according to

$$\boldsymbol{\rho} = R\boldsymbol{\rho}_{ECEF}$$

Where the rotation matrix R projects vectors expressed in the ECEF frame to vectors expressed in the ECI frame. Computation of R depends on which inertial frame is used in the orbit determination paradigm for a given space mission. However, most ECI and ECEF frames are defined in such a way that R has the form

$$R = \begin{bmatrix} \cos \Delta & -\sin \Delta & 0 \\ \sin \Delta & \cos \Delta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Which is simply a rotation about the ECEF z axis. The parameter Δ is an angle that corresponds to a rotation around the ECEF z axis which aligns it with the inertial frame in question.

Now that the target vector of interest has been calculated, it is simply placed in the row of the desired inertial to body rotation matrix corresponding to the body pointing axis (e.g. if +Z axis is body pointing axis, place in 3rd row). In other words, the a'th row of the desired inertial to body rotation matrix is set as

$$r_a = \hat{\boldsymbol{T}}$$

The other two rows of the rotation matrix still remain. They are calculated as

$$r_b = \frac{\boldsymbol{V} \; x \; \hat{\boldsymbol{T}}}{\left| \boldsymbol{V} \; x \; \hat{\boldsymbol{T}} \right|}$$

$$r_c = r_a \; x \; r_b$$

Where $\boldsymbol{V}$ is the inertial velocity vector of the spacecraft. Then the rotation matrix corresponding to the desired inertial to body attitude is constructed as

$$C_r = \begin{bmatrix} r_a{}^T \\ r_b{}^T \\ r_c{}^T \end{bmatrix}$$

It should be noted that the rotation matrix C_r may require a different organization of the rows r_a, r_b, and r_c depending on the body pointing axis. The calculation above assumes that +X is the desired body pointing axis, since r_a occupies the first row of the matrix. Finally, the reference quaternion is computed from the reference rotation matrix C_r in the same way as was described in the Nadir Pointing section (3.5.2). In short,

$$q_r = q(C_r)$$

Finally, the attitude error vector is computed with

$$\boldsymbol{e} = 2\boldsymbol{\epsilon}(q_r^* q)$$

Computing the reference quaternion in this way may result in similar discontinuous behavior as described in the Nadir Pointing section. Therefore, the same logic described in section 3.5.2 is implemented in this mode.

3.5.4 Ephemeris Pointing

In ephemeris pointing, the attitude control system's goal is to align a commanded body fixed pointing axis with a vector pointing from the spacecraft to a celestial body of interest. Pointing towards a celestial body of interest has numerous applications in a large class of space missions. More specifically, many space missions require a sun pointing mode during which the spacecraft aligns its solar array normal vectors with the sun to maximize input power. Ephemeris pointing is meant to generalize beyond sun pointing and allow pointing a body fixed vector at any target, specified by an ephemeris. Therefore, this mode is remarkably similar to the ground pointing mode. However, the difference is that in ephemeris pointing, the target vector does not move as quickly, and therefore the control system bandwidth may not need to be as high to track the lower frequency quaternion reference signals with sufficient accuracy.

The reference quaternion in the ephemeris pointing mode is computed in almost the exact same manner as in ground pointing, with the exception that the target vector is instead known in the inertial frame a priori and does not require computation. Therefore, the exact same procedure as described in 3.5.3 can be used to compute the quaternion reference signal, with the target vector $\hat{\boldsymbol{T}}$ instead being the unit vector pointing from the inertial spacecraft position to the inertial position of the celestial body of interest given by the ephemeris:

$$\hat{\boldsymbol{T}} = \frac{\boldsymbol{R_T} - \boldsymbol{R}}{|\boldsymbol{R_T} - \boldsymbol{R}|}$$

Where $\boldsymbol{R_T}$ is the inertial position vector of the celestial body given by the ephemeris expressed in the ECI frame. Then the rest of the procedure described in 3.5.3 can be used to construct the desired inertial to body rotation matrix, such that the commanded body fixed axis points toward the target, an orthogonal axis stays in the orbit plane, and the third axis completes the orthogonal triad.

3.5.5 Spin Pointing

In spin pointing, the attitude control system aligns a body fixed axis with a fixed direction in inertial space, and imparts an angular velocity around that axis according to a commanded spin rate. This mode is applicable to many conceivable operational scenarios, the most prominent of which is an orbital maneuver or safe mode. Since the desired inertial delta-v vector for an orbital maneuver should be known before the spacecraft initiates the maneuver, this mode could be used to command the spacecraft to place its thrust vector along that desired inertial delta-v vector and begin spinning around that vector at a small rate. Flying in such an attitude during an orbital maneuver can help average out the total accumulated delta-v along the desired inertial burn duration, increasing maneuver accuracy. Further, spinning around a thrust axis can gain stabilize any potentially harmful fuel slosh modes if liquid fuel is used in the propulsion system. On the other hand, the spin pointing mode may be applicable for mission scenarios in which the spacecraft is in an idle state and does not require any sort of active pointing. During this phase of a mission, an appropriately chosen inertial direction and spin rate could place the spacecraft in an extremely safe attitude profile from a thermal perspective. For example, if the spacecraft has

radiators on some of its surfaces, the inertial direction could be chosen such that none of those radiators can see the sun. Then the slow spinning around this direction will evenly distribute incident heat flux onto the rest of the spacecraft surfaces.

The desired attitude quaternion in the spin pointing mode is again computed from a rotation matrix representing the same desired attitude. This rotation matrix is conceptualized as being comprised of two sequential rotations, one that aligns the commanded body axis with the commanded inertial direction, and another that incrementally guides rotation of the spacecraft around its commanded body axis. The desired inertial to body attitude parameterized by a rotation matrix is given by

$$C_r = C_2(t)C_1$$

Where C_1 encodes the rotation required to align the commanded body fixed axis with the commanded inertial direction, and $C_2(t)$ applies the rotation around the commanded body fixed axis at the commanded rate. C_1 is given by a two-step rotation through azimuth and elevation angles, but the sequence of rotation for C_1 depends on the commanded body fixed axis. For example, if the commanded body fixed axis is +X, the rotation sequence is ZY, so that no rotation around the body fixed axis occurs when aligning it with the commanded inertial direction. The azimuth and elevation angles themselves must also be defined in such a way that is consistent with the commanded body fixed axis. Figure 9 provides the geometry for this first rotation sequence in the case that the commanded body fixed axis is +X.

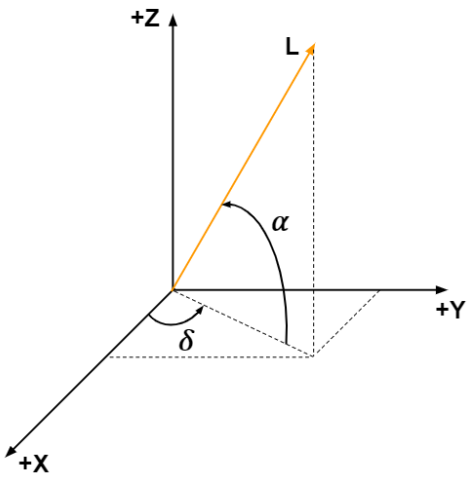

Figure 9: Rotation Angle Geometry for Inertially Fixed Pointing Vector

Figure 9 depicts a commanded inertial direction L along with the relevant azimuth and elevation angles. Notice that if the inertial frame were rotated around its positive Z axis through the angle δ, and then around the new Y axis through the angle $-\alpha$, the +X axis would be aligned with L. Therefore, in this case C_1 is given by two principle frame rotations:

$$C_1 = C_y(-\alpha)C_z(\delta)$$

More generally, the relevant azimuth and elevation angles are first computed from L, and then the appropriate principle rotations are applied to compute C_1 depending on the commanded body fixed axis.

The second rotation $C_2(t)$ is simply a principle rotation around the commanded body fixed axis through an angle that increases linearly with time according to the commanded spin rate. It is computed as

$$C_2(t) = C_a(\omega_s t)$$

Where the subscript a denotes a principle rotation around either the X, Y, or Z axis, depending on which one was commanded, and ω_s is the commanded spin rate. In discrete time, C_2 is given by

$$C_2 = C_a(\theta_0 + \omega_s T)$$

Where T is the sampling period and θ_0 is an arbitrary initial angle. Once the two rotation matrices are computed, their product is taken and then cast to a quaternion to provide the final reference quaternion in spin pointing mode:

$$q_r = q(C_2(t)C_1)$$

Finally, the attitude error vector is computed using the familiar conjugate product

$$\boldsymbol{e} = 2\boldsymbol{\epsilon}(q_r^* q)$$

3.5.6 Desaturate

The final control mode considered in this work is a desaturate mode, in which the control system employs actuators capable of imparting an external torque on the spacecraft to remove angular momentum stored in reaction wheels. A simple desaturation control law commanding a magnetorquer dipole vector is employed, while the wheels are simultaneously commanded by the attitude control loop in the inertial pointing mode. From [14], a magnetorquer desaturation command that can be used to remove stored angular momentum is given by

$$\boldsymbol{m} = \frac{k_{desat}}{|\boldsymbol{B}|}(\boldsymbol{h}\ x\ \boldsymbol{B})$$

Where $\boldsymbol{B}$ is the instantaneous geomagnetic field vector expressed in the body frame, $\boldsymbol{h}$ is the net angular momentum vector of the reaction wheels, also expressed in the body frame. k_{desat} is a gain that controls how aggressively the magnetorquers are commanded for a given amount of stored reaction wheel momentum that exists perpendicular to the instantaneous local magnetic field vector. The choice of the gain is chosen based on the spacecraft orbit, size of magnetorquers, and the desired rate at which momentum is to be removed from the reaction wheels.

During desaturation mode, the reaction wheels are commanded by the attitude controller to hold a constant inertial attitude, while the magnetorquers are commanded according to the above control law. Conceptually, this causes the spacecraft to attempt to “hold onto” inertial space as the magnetorquers try to pull it away from the commanded attitude in the direction of

the wheel angular momentum that is perpendicular to the local magnetic field. Consequently, the momentum stored in the reaction wheels ends up being removed by the external torque from the magnetorquers as the wheels attempt to maintain the fixed inertial attitude.

An important practical consideration behind the use of magnetorquers for desaturation during inertial pointing is that they may significantly corrupt magnetometer readings and thus led to poor estimation algorithm performance. If the estimation algorithm diverges during a desaturation maneuver as a result of bad magnetometer readings, the inaccurate state estimate fed to the inertial pointing controller could lead to egregious closed loop behavior. To avoid this, the commands to the magnetorquers and the sampling of the magnetometers should be sufficiently separated in time.

Chapter 4

CPCL ADCS TOOLKIT ENHANCEMENTS

This chapter will provide an overview of the contributions of this thesis to the overall CPCL ADCS design and analysis toolkit. Additionally, guidance and control design techniques illustrated in the previous chapter will be integrated into the toolkit in this chapter. As illustrated in the design iteration strategy from Chapter 2, the overall engineering philosophy proposed herein is to start with a simplified analysis during preliminary design and slowly reduce the number of assumptions. Then, as the design progresses and decisions are made, assumptions can be rescinded, and the complexity and fidelity of the analysis can be incrementally increased. The overall toolkit is designed to support such an approach, by providing tools of increasing fidelity and complexity. In this chapter, the utility and use cases for three analysis and design tools will be presented, with each tool corresponding to a particular phase within the proposed design iteration strategy.

4.1 Preliminary Design: Single Axis Simulink Tool

Overall, the single axis Simulink tool is designed to provide the engineer with the ability to quickly spec out a reaction wheel relative to the desired control algorithm behavior. For example, suppose the designer wants a relatively high bandwidth attitude controller (commanding reaction wheels) to achieve a sufficiently fast settling time for mission specific reasons. In this case, a stronger reaction wheel with a faster mechanical time response and perhaps a relatively high torque saturation limit will likely be required. This tool can be used to easily establish the range of reaction wheel parameters that will give rise to acceptable closed loop performance with the chosen controller specifications. Furthermore, it could be used in hardware selection trade studies that aim to evaluate which of two off the shelf reaction wheel choices is a better choice. While it is possible to do all of this analysis in a complicated 6DOF simulation environment, doing so is overkill, cumbersome, and inefficient. In many cases, a vastly simplified analysis can get the job done just as well, avoiding the need to deal with a complex simulation environment.

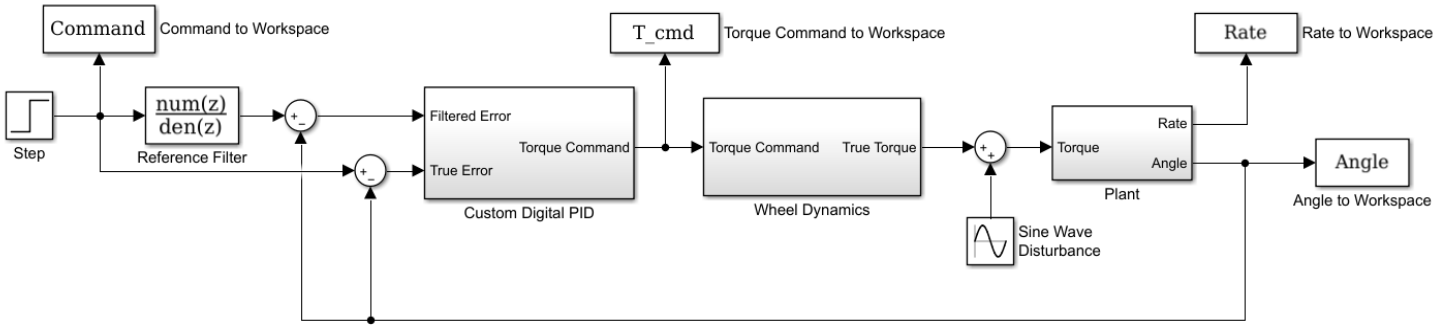

Figure 10: Single Axis Simulink Model

Another important utility of this tool is that it can be used in circumstances where the reaction wheel parameters are already known and can be treated as a constraint on the controller design, or when a reaction wheel must be chosen to support a given controller design. In either of these cases, this single axis Simulink tool would provide utility.

Finally, this tool can be used to perform a preliminary determination of momentum storage capacity requirements. If the spacecraft mass properties and orbit are known, a step disturbance torque of an appropriate magnitude could be applied and the resulting wheel momentum accumulation could be analyzed, providing the designer with insight into how much momentum their reaction wheel should carry. While this same analysis could be performed in three dimensions, doing so is again overkill and not necessary for preliminary sizing calculations.

Once the top-level requirements on the ADCS are reasonably well understood, the first step of the design process should be relatively simple and rapid. Therefore, starting with a simplified model of the dynamics of the system to be controlled and the associated algorithms is a sensible approach. Figure 10 shows the single axis Simulink model, intended be used as a tool in the preliminary design phase. Accompanying this Simulink model is a MATLAB script that allows the designer to define all of the parameters used in the Simulink model as well as invoke the Simulink model and view its outputs. In this tool, reaction wheels are assumed as the primary attitude actuator as single axis analysis with magnetorquers is not possible.

The Simulink model in figure 10 implements a digital PID controller around a double integrator plant. The actuator command output from the controller is sent into the wheel dynamics

block, saturated to the maximum torque value, and held constant until the next control cycle. The wheel dynamics block implements a simplified model of a reaction wheel with an internal motor driver chip responsible for controlling the speed of the wheel. Figure 12 shows the internals of the ‘Custom Digital PID’ block, and figure 11 shows the internals of the ‘Wheel Dynamics Block’. As seen in figure 12, the PID controller contains a MATLAB function which implements deadband logic for the control system. The purpose of the deadband is to prevent the quantized speed commands in the wheel controller from persistently oscillating between zero and their minimum speed bit. The deadband logic works as follows: if the absolute value of the true attitude error—namely the difference between the unfiltered reference signal and the feedback signal—transitions from less than the error deadband to greater than the error deadband, a counter begins incrementing. This timer is reset to zero if the attitude error transitions from being inside the error deadband to outside the error deadband. If the timer exceeds a user-specified convergence time, and the attitude error is within the deadband, then the attitude error is set to zero and the integral state in the controller is reset to zero. This prevents the control system from commanding the actuators if the attitude error has been within the deadband for a specified amount of time. If the attitude error is outside the deadband and the convergence time has not been achieved, the output of the function is simply the filtered attitude error. This logic has been tested in simulation and works as expected, but a well-informed choice of the size of the attitude error deadband and convergence time requires nontrivial analysis that is beyond the scope of this thesis. However, the existence of this logic in this tool provides the user with the means to check whether their choice of deadband width and convergence time yields desirable closed loop behavior.

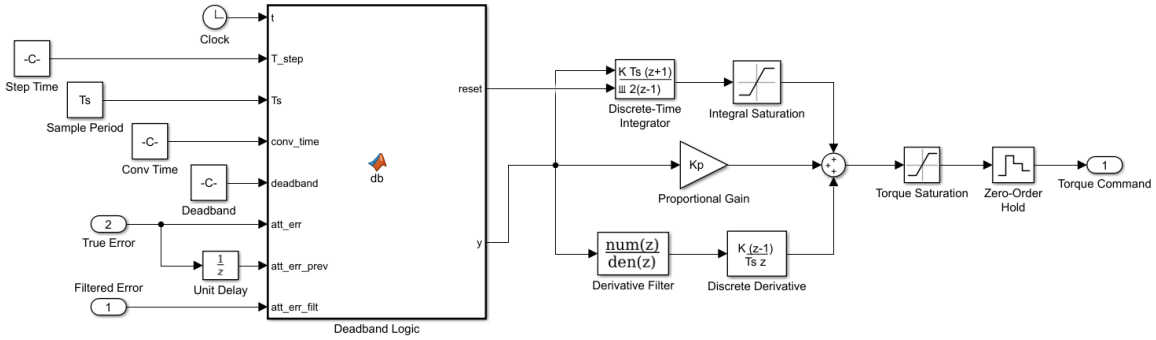

Figure 12: Custom Digital PID Block Contents

The rest of the contents of the ‘Custom Digital PID’ block are straightforward. The filtered attitude error is integrated using the trapezoidal integration technique, multiplied by the integral gain, and saturated to the user specified integral state saturation limit to prevent wind up. Conversely, the filtered attitude error is passed through the derivative filter, differentiated using the backward Euler technique, and multiplied by the derivative gain. Finally, the filtered attitude error is simply multiplied by the proportional gain. Then, all three of these signals are added,

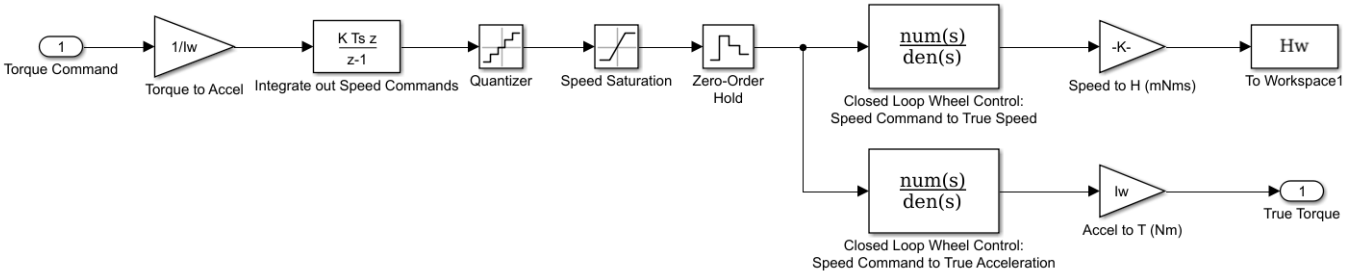

Figure 11: Reaction Wheel Model Block Contents

saturated to the maximum actuator torque, and held constant until the next control cycle.

As seen in figure 11, the wheel dynamics block converts the torque command outputs from the controller into speed command set points by first dividing by the wheel inertia to yield wheel accelerations, and then performing backward Euler integration on these accelerations. This integration must occur at a sufficiently faster sampling rate than the attitude control loop to prevent a non-negligible phase lag between controller torque commands and actuator torque outputs. The wheel speed set points are quantized to the minimum speed bit. The quantizer block in Simulink outputs the input of the block rounded to the nearest integer multiple of the minimum

speed bit. The speed command is then saturated, held constant for the sampling period of the wheel controller, and passed through a first order transfer function of the form

$$G(s) = \frac{1}{\tau s + 1}$$

Where τ is the time constant of the closed loop wheel control system. Here, $G(s)$ is meant to model the closed loop dynamics of the wheel control loop, from speed reference command to speed output. Finally, the speed command output is passed through $sG(s)$ and multiplied by the wheel inertia to yield an instantaneous torque.

To reiterate, the primary utility of the single axis Simulink tool is to provide a vastly simplified analysis environment that supports reaction wheel specification and sizing, and preliminary control law design. Once these parameters have been established with confidence, a three axis Simulink tool should be utilized to verify that the design still meets requirements with coupled, nonlinear dynamics and more sophisticated reference signals.

4.2 Preliminary Design: Three Axis Simulink Tool

To extend the control system design and analysis into three dimensions, a full three axis Simulink tool was developed in this work. Figure 13 shows a top-level view of the model.

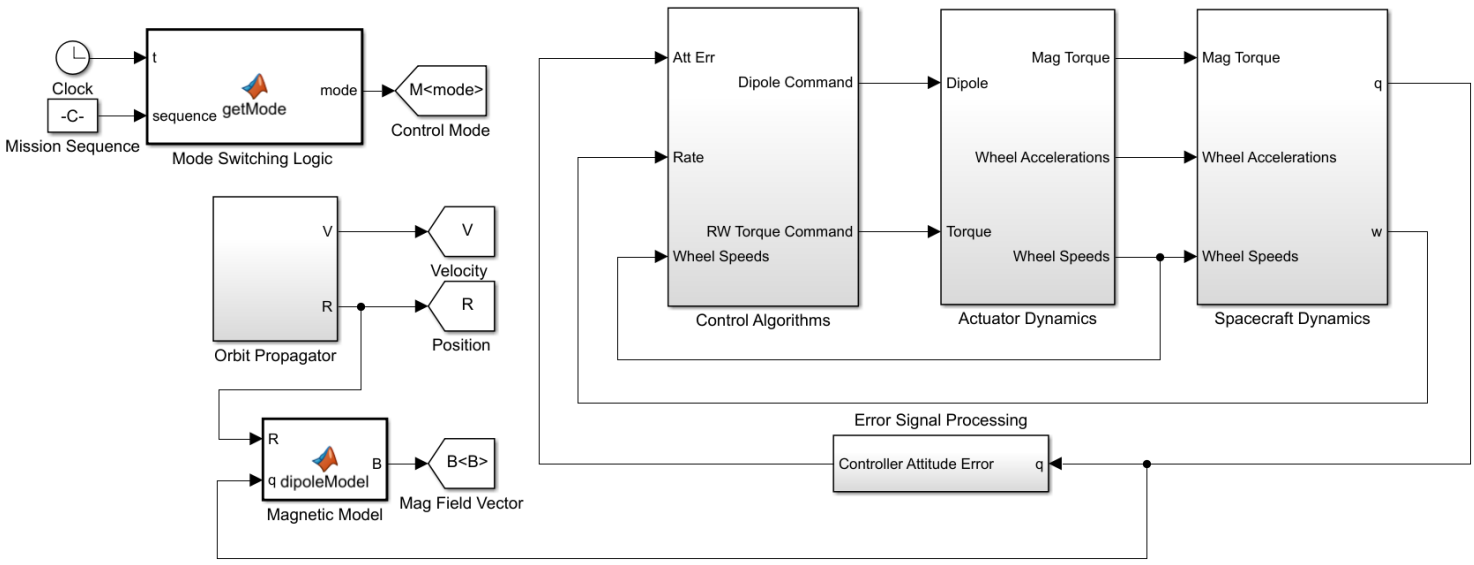

Figure 13: Three Axis Nonlinear Simulink Model

The three axis model is very similar in structure to the single axis model, with the exception that the attitude error is no longer calculated by a simple difference. Instead, it is computed based on the control mode being simulated, the mathematical details of which were provided in Chapter 3. Furthermore, in this model the full three axis nonlinear rigid body dynamics are simulated in the 'Spacecraft Dynamics' Block. In the 'Control Algorithms' block, three digital PID controllers are implemented for each axis, with an option for whether or not the feedback linearization signal is included in the control. The 'Actuator Dynamics' block implements the same wheel dynamics as described in section 4.1, except now with multiple reaction wheels. The 'Error Signal Processing' block takes the true attitude quaternion from the spacecraft dynamics and, depending on the current operational being simulated, computes the reference quaternion and the attitude error vector. Finally, the 'Orbit Propagator' block implements an orbit dynamics model and provides a position and velocity vector to the error signal processing block to be used in the relevant operational modes.

An important goal in the development of this tool was to make it relatively modular and extendable to support future analysis needs. To achieve this, the model was separated into five primary blocks with generic interfaces between them that represent the minimum amount of information that must be at their input or output. These primary blocks include an orbit dynamics model, an attitude dynamics model, an error signal processing block, a control algorithms block, and an actuator dynamics block. This Simulink model is designed in such a way that the overall functionality is robust to changes in the internal dynamics of each of the primary blocks, provided those dynamics are well-defined. For example, if a different model of the reaction wheel dynamics is desired than the one currently being implemented, then any model that takes in a torque command as an input and outputs wheel speeds and wheel accelerations as outputs will be consistent with the overall Simulink model. On the other hand, if a different orbit propagator is required, it simply needs to output the position and velocity vectors so that they can be used by the rest of the model where necessary. Although this model does not support the full range of possible satellite actuator configurations and control structures, it still has the modularity to

support most analysis and design needs for an attitude control system employing reaction wheels and magnetorquers as actuators, which are very likely to be utilized in CubeSat missions.

Another important feature of this tool is its ability to simulate custom ADCS operational profiles. More specifically, the user can build a custom ‘Mission Sequence’, which specifies a set of modes (from the list in Chapter 3) and their start/end times. This allows analysis of the control system behavior under instantaneous transitions between control modes.

Overall, this tool is intended to be used as a follow-up to the single axis tool. Once the single axis tool is used to establish a preliminary control law and reaction wheel specifications, the analysis needs to be extended into three dimensions. Furthermore, understanding how the true nonlinear, multi-input multi-output control system behaves under the influence of more sophisticated reference signals is especially important after conducting a decoupled, linear, single-input single output control design. This tool enables such an investigation.

The contents of each of the blocks in the three axis Simulink model will now be explained, starting with the ‘Error Signal Processing’ block. Figure 14 depicts the contents of this block, which is primarily comprised of a custom MATLAB function implementing the guidance algorithms described in Chapter 3. The inputs to the guidance function are essentially all of the user-facing commands to the attitude control system. These commands include the body fixed

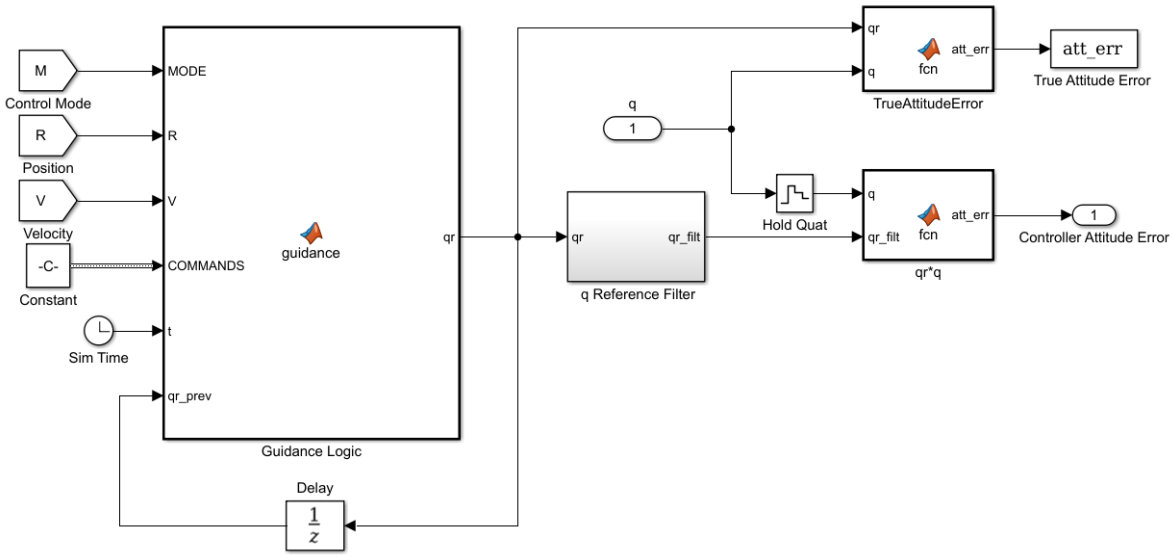

Figure 14: Error Signal Processing Block Contents

pointing axis, the inertial pointing reference quaternion, and the spin rate for the spin pointing mode, to name a few. The guidance function simply checks the control mode of the simulation, and calculates the reference quaternion accordingly. This reference quaternion is then passed through the digital reference filter if it is enabled. The true quaternion from the spacecraft dynamics block is held constant for the sampling period of the control system, and the discrete time attitude error vector is computed from the filtered reference quaternion and the true quaternion. The resulting attitude error vector is the only output of this block.

The attitude error vector as computed by the Error Signal Processing block is then sent to the Control Algorithms block, shown in figure 15. This block implements three digital PID controllers for each body frame axis, each of which are contained within the 'PID + Feedback Linearized Control Law' block. The desaturation control law described in Chapter 3 is implemented as a separate custom MATLAB function, and only outputs non-zero dipole

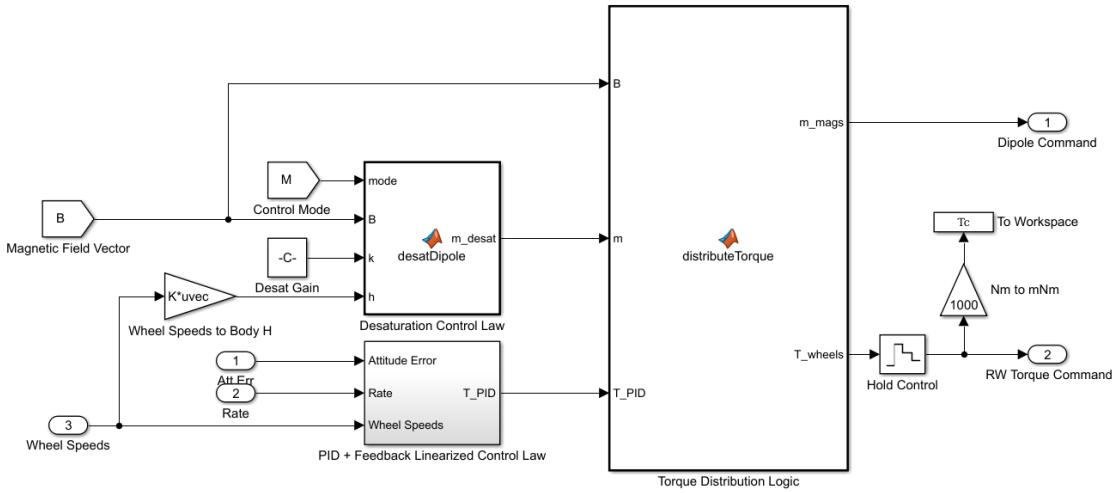

Figure 15: Control Algorithms Block Contents

commands if the simulation is in desaturation mode. Once the actuator commands are computed from the magnetorquer control law (currently only desaturate) and the PID control law, they are sent to the 'Torque Distribution Logic' custom MATLAB function. This function distributes the total control torque—as computed by the controller and magnetorquer control laws—among the available actuators. In the current development state of this tool as this thesis is being written, this function simply gives the reaction wheels all of the PID control torque, and gives the

magnetorquers any dipole commands that come from the desaturation control law. However, other strategies exist for distributing the controller outputs among the available actuators, and this functionality is meant to provide a means of implementing such strategies. More specifically, [19] develops a geometric approach to decompose the total control torque vector into a component parallel to the magnetic field—which can be distributed to reaction wheels—and a component perpendicular to the magnetic field, which can be distributed to magnetorquers. This torque distribution stategy is one possible implementation for the torque distribution logic. A deeper look into whether these strategies are worth considering from a broader engineering perspective would be a productive exersice.

Once the total control signal has been computed by the control laws and distributed among the available actuators, the commands to each of the actuators are sent to the 'Actuator Dynamics' block, shown in figure 16. Notice the simplicity of the top level view inside this block—this is intentional. For each of the avaialable actuators, a model which takes a command as an input and outputs the true relevant dynamic quantities is all that is required in the Actuator Dynamics block as a whole. In the current development state, the tool implements a dynamics model for three reaction wheels that is equivalent to the wheel dynamics model in the single axis tool for each individual wheel. The wheel dynamics model is shown in figure 18.

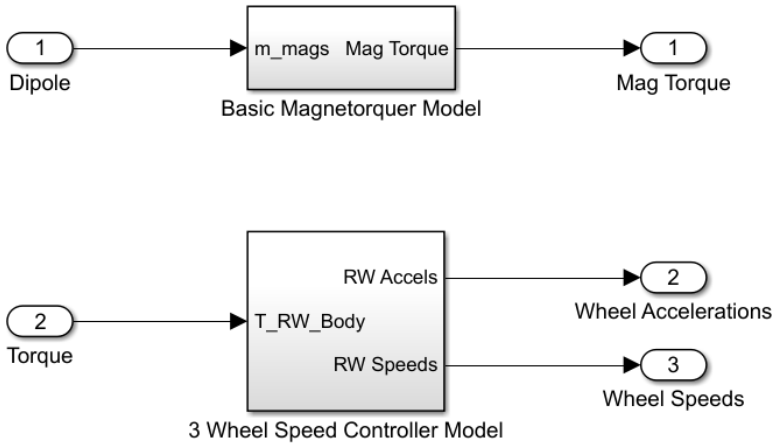

Figure 16: Actuator Dynamics Block Contents

The current magnetorquer model implementation, on the other hand, is even simpler. It simply saturates the dipole command from the desaturate controller, holds the command constant for a sample period, and calculates the cross product of the saturated dipole vector with the local magnetic field to yield an external torque. Figure 17 shows the magnetorquer model.

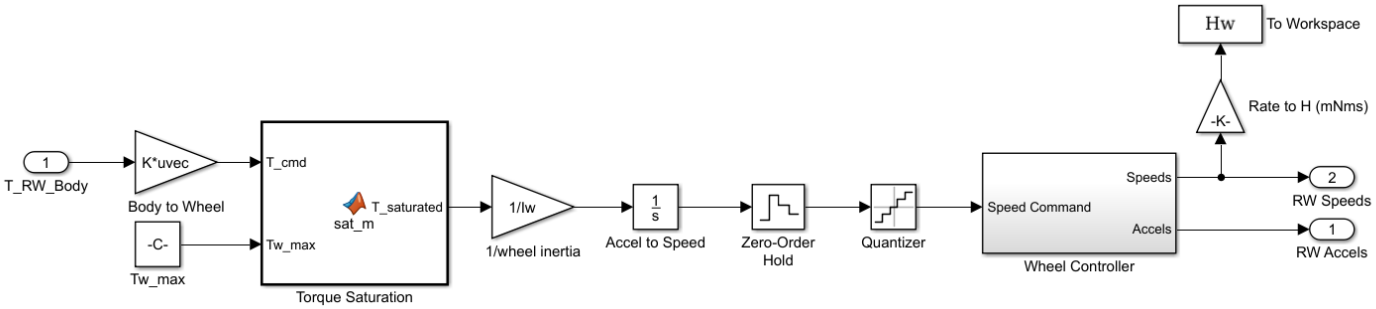

Figure 18: Wheel Dynamics Block Contents

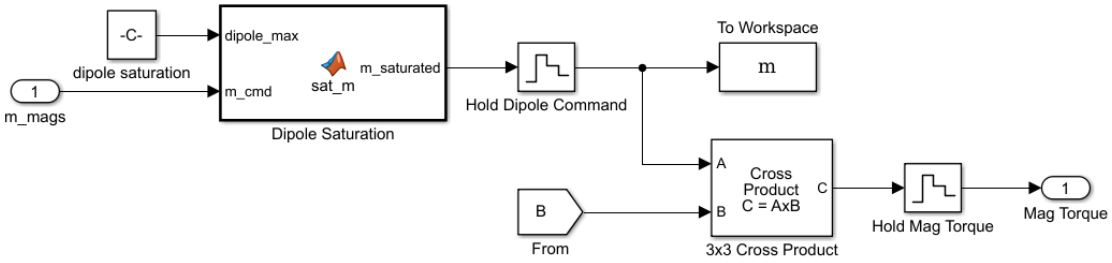

Figure 17: Magnetorquer Block Contents

Once the true dynamic quantities are calculated by the actuator models, they are sent to the ‘Spacecraft Dynamics’ block, where the rigid body equations of motion are integrated. The contents of the Spacecraft Dynamics block are shown in figure 19. The custom MATLAB function in figure 19 simply implements the equations of motion for a rigid body with the wheel accelerations as the control input. A step disturbance or sinusoidal disturbance can be injected into the spacecraft dynamics, with an amplitude and frequency at the discretion of the user. This allows the user to analyze the response of the control system to external disturbances in a nonlinear simulation.

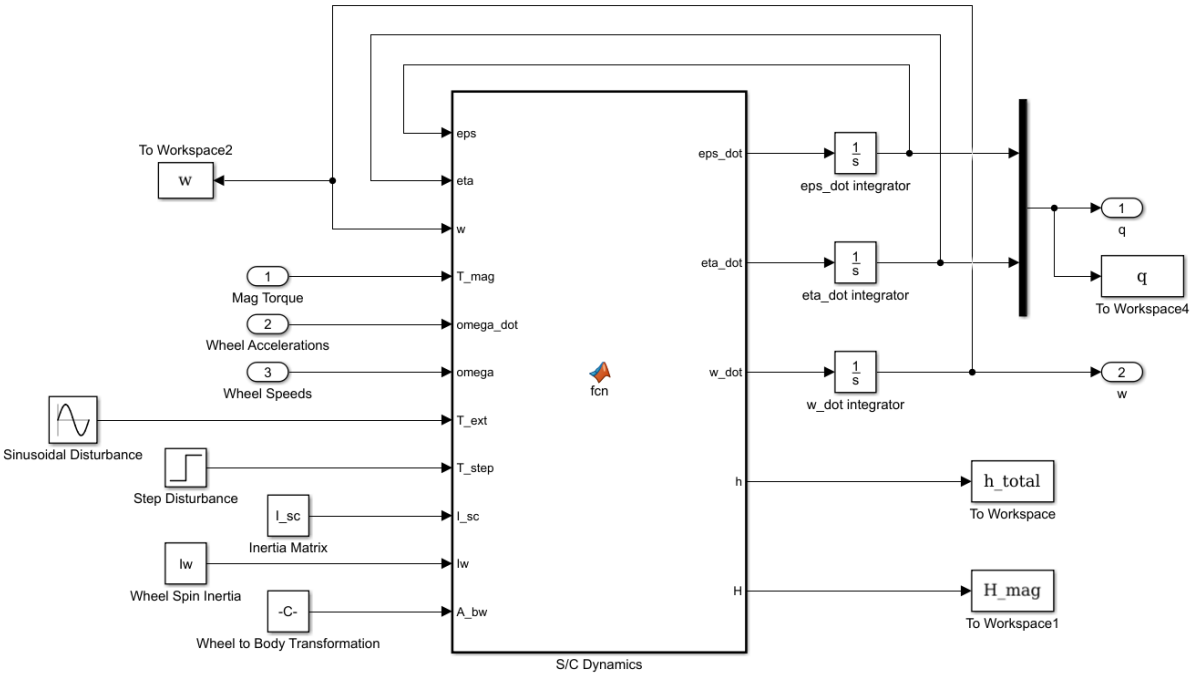

Figure 19: Spacecraft Dynamics Block Contents

Overall, the three axis Simulink tool is intended to be used only in preliminary design, as it does not model much of what is going on during actual flight, nor does it take into account the behavior of the real-time flight software implementations of the GNC algorithms. However, the modeling fidelity is sufficient for preliminary design, and can be increased if necessary with more sophisticated sub-models for the actuators, magnetic field, orbit, or spacecraft dynamics. All that is required to do so is a replacement of the current block implementation with a new implementation containing the same inputs and outputs. In its current state, this tool provides the designer with the means of establishing preliminary control system specifications (e.g. bandwidth, sample rate, phase margin, integral saturation limit, filter cutoff frequencies, etc.), as well as sizing of magnetorquers and reaction wheels. If higher modeling fidelity is required after a preliminary design is established, then the designer should consider moving onto the next and final tool, which is the NASA 42 software-in-the-loop simulation framework.

Finally, the exclusion of any estimation algorithms in the Simulink tools presented in this work is intentional. Overall, the design and tuning of control laws can be performed without an estimator in the loop, as long as the effects of potentially noisy feedback signals are not

completely ignored. This is precisely the reason that a digital low pass filter was added before the derivative term in the control law. Moreover, if the estimation algorithm is well designed and performs as expected, then it should significantly attenuate measurement noise. As long as most of the noise power in the feedback signal is concentrated around frequencies that are significantly above the bandwidth of the control system, its effect on the closed loop dynamics will be minimal. In chapter 5, it will be shown that this in fact that case with the existing extended Kalman filter design by Mehrparvar and implemented in flight software by Bouchard. If a different estimation algorithm than the one currently implemented is required for a given flight mission, then it would need to be prototyped and tested in the NASA 42 software-in-the-loop environment. Development of tools for designing and testing estimation algorithms outside of the NASA 42 software-in-the-loop environment is beyond the scope of this thesis.

4.3 Critical Design: NASA 42 Software-in-the-Loop Environment

Once the preliminary control system design has been established by means of utilizing the Simulink tools outlined in the previous two sections, and the ADCS is roughly at a PDR level maturity, the critical design phase begins. In this phase of development, high modeling fidelity is required to provide increased certainty that the ADCS functional and performance requirements are being satisfied before moving onto hardware procurement and testing. As was outlined in Chapter 2, the CPCL has developed a software-in-the-loop simulation environment employing NASA 42 as the truth model. This section will provide increased detail on the capabilities of the simulation environment and how this thesis has contributed to it, with an emphasis on sensor and actuator models and custom processing of user input files.

Within the NASA 42 software-in-the-loop simulation environment, there are two processes consistently invoking one another during a simulation run. The first of these is processes is the ADCS flight software itself, and the second is NASA 42. If the real time flight software implementations of the GNC algorithms are subject to flight-like signals from NASA 42 sensors, and they command actuators whose dynamics are well modeled, it is possible to achieve a much higher degree of confidence that the system will meet its requirements than with the previously mentioned Simulink tools alone. That is precisely the goal of this environment—to

provide a much higher requirement verification confidence in the critical design phase of the mission development timeline.

NASA 42 is written in a modular fashion such that custom models of sensors and actuators can be implemented in its framework without required modification anywhere else within the code base. In this work, generic sensor models for rate gyros, magnetometers, and sun sensors are utilized. Each model operates in discrete time, and the sensors are sampled at the same rate that the control algorithms are executed. The following sub sections will provide details on how each of these models are implemented in the software-in-the-loop simulation framework.

4.3.1 Digital Rate Gyro Sensor Model

Excluding scale factors and misalignments, the rate gyro measurement model in NASA 42 implements the following widely used rate output and bias equations as given in [14]:

$$\boldsymbol{\omega}_{k+1} = \boldsymbol{\omega}_{k+1}^{true} + \frac{1}{2}(\boldsymbol{\beta}_{k+1}^{true} + \boldsymbol{\beta}_{k}^{true}) + \left(\frac{{\sigma_v}^2}{\Delta t} + \frac{1}{12}{\sigma_u}^2 \Delta t\right)^{1/2} \boldsymbol{N}_v$$

$$\boldsymbol{\beta}_{k+1}^{true} = \boldsymbol{\beta}_{k}^{true} + \sigma_u \sqrt{\Delta t}\, \boldsymbol{N}_u$$

Where the subscript k denotes the k^{th} time step, $\boldsymbol{\omega}$ is an angular velocity vector, $\boldsymbol{\beta}$ is a bias rate vector, σ_v and σ_u are the standard deviations of the Gaussian amplitude probability distributions of the rate output signal and bias rate signal respectively, and Δt is the sample period of the digital rate gyro. Finally, $\boldsymbol{N}_v$ and $\boldsymbol{N}_u$ are zero-mean Gaussian white noise processes with covariance matrices given by the identity matrix. The resulting rate gyro measurement is then passed to the state estimation algorithm.

The derivation of the above model will not be included in this work, but it is essentially a discretized version of the following continuous time rate gyro measurement model:

$$\boldsymbol{\omega}(t) = \boldsymbol{\omega}^{true}(t) + \boldsymbol{\beta}^{true}(t) + \boldsymbol{\eta}_v(t)$$

$$\dot{\boldsymbol{\beta}}^{true} = \boldsymbol{\eta}_u(t)$$

Where $\boldsymbol{\eta}_v$ and $\boldsymbol{\eta}_u$ are both stationary white noise processes. Conceptually, this model of the rate gyro output dynamics says that the bias follows a random walk in discrete time as obtained by

integration of white noise in continuous time, and the angular rate output is corrupted by both the current bias and a different white noise process. The rate gyro scale factor is also modeled, and it simply scales the true angular rate $\boldsymbol{\omega}_{k+1}^{true}$ in the above discrete time model. Fortunately, the datasheets for most commercially available rate gyros contain information about the standard deviation of the output noise as well as the bias random walk and sometimes the scale factor. One subtlety regarding the standard deviation σ_u of the white noise process corresponding to the bias derivative is that it does not have the same units as the corresponding relevant specification listed in most gyro datasheets (e.g. “in-run bias stability”). This is because the datasheets do not make assumptions about the frequency with which the sensor is sampled, and instead list the product $\sigma_u\sqrt{\Delta t}$, which has units of angle per time. Therefore in this context, the user of the software-in-the-loop simulation environment is expected to provide NASA 42 with the product $\sigma_u\sqrt{\Delta t}$, and the standard deviation σ_u is indirectly calculated internally before being initialized by NASA 42.

4.3.2 Sun Sensor Model

In this work, the PolySat in-house sun sensors are modeled as Fine Sun Sensor (FSS) types in NASA 42. Since this thesis presents the first attempt at modeling these in-house sensors with any sort of rigor, it may turn out that Coarse Sun Sensor (CSS) types in NASA 42—or a different customized model for that matter—more accurately reflect the behavior of these sensors. However, for the sake of the analysis presented in this thesis, the Fine Sun Sensor type is deemed an adequate model.

Each sun sensor is assumed to have a pyramidal field of view. When the FSS model executes in NASA 42, the simulation first checks whether the spacecraft is in eclipse, and if so, all solar angle sensor outputs are set to zero, and their readings do not get processed by the state estimation algorithm. If the spacecraft is not in eclipse, NASA 42 will iterate over all existing sun sensors and evaluate whether the sun vector is in each of their fields of view. First, the true body frame sun vector is projected into the sensor frame using the true body to sensor frame transformation with

$$\boldsymbol{r}_{ss} = C_{BS}\boldsymbol{r}_{sb}$$

Where $\boldsymbol{r}_{ss}$ is the true unit vector pointing from the spacecraft to the sun expressed in sensor coordinates, and $\boldsymbol{r}_{sb}$ is the same vector expressed in body coordinates. To determine whether the sun is in the FOV of a sun sensor, the following angles are computed using the x and y components of the sun unit vector expressed in the sensor frame:

$$\theta_x = \sin^{-1}(\boldsymbol{r}_{sx})$$

$$\theta_y = \sin^{-1}(\boldsymbol{r}_{sy})$$

Figure 20 depicts the geometry of the sun unit vector (yellow) expressed in the sensor frame with the angles θ_x and θ_y.

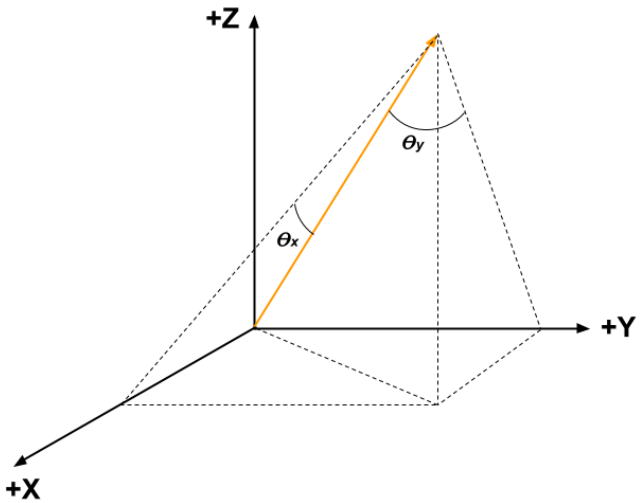

Figure 20: NASA 42 Sun Sensor FOV Geometry

If the angles θ_x and θ_y are both less than the angles defining the pyramidal field of view in both the x and y axes respectively, then the sun is determined to be within the field of view of the sensor. Next, each of these angles are corrupted by a Gaussian white noise process whose standard deviation can be stated in terms of an angle (i.e. the "Noise Equivalent Angle" in NASA 42). Mathematically, the measured angles are computed as

$$\theta_{x,m} = \theta_x + \sigma_\theta N_\theta$$

$$\theta_{y,m} = \theta_y + \sigma_\theta N_\theta$$

Where the subscript m denotes the measured angles, σ_θ is the standard deviation of the Gaussian amplitude probability distribution of the angular measurement noise, and N_θ is a zero-mean Gaussian white noise process with unity variance. The measured angles are then quantized according to the digital resolution of the sensor. Finally, the measured sun unit vector in the sensor frame is computed as

$$\boldsymbol{r}_{ss,m} = \begin{bmatrix} \sin(\theta_{x,m}) \\ \sin(\theta_{y,m}) \\ \sqrt{1 - (\sin(\theta_{x,m}))^2 - (\sin(\theta_{y,m}))^2} \end{bmatrix}$$

The resulting measured sun vector $\boldsymbol{r}_{ss,m}$ is then passed to the state estimation algorithm.

4.3.3 Magnetometer Model

The magnetometer model is the simplest of the sensor models utilized in this work. First, the true magnetic field vector expressed in body frame is first projected into sensor coordinates using the true body to sensor frame transformation with

$$\boldsymbol{r}_{ms} = C_{BS}\boldsymbol{r}_{mb}$$

Where $\boldsymbol{r}_{ms}$ is the true magnetic field vector expressed in the sensor frame, and $\boldsymbol{r}_{mb}$ is the true magnetic field vector expressed in body frame. The vector $\boldsymbol{r}_{ms}$ is then corrupted with Gaussian white noise using

$$\boldsymbol{r}_{ms,m} = \boldsymbol{r}_{ms} + \sigma_m \boldsymbol{N}_m$$

Where the subscript m denotes the measured magnetic field vector, σ_m is the value of the RMS magnetometer output noise, and $\boldsymbol{N}_m$ is a zero-mean Gaussian white noise process with zero-mean and covariance matrix given by the identity matrix. Finally, the readings are saturated and quantized to the resolution of the digital output.

4.3.4 Actuator Models

At the time this thesis is being written, the actuator models in NASA 42 simply implement saturation wherever it is applicable. Higher fidelity actuator models in the NASA 42 simulation framework constitute future work that will improve the overall quality of the software-in-the-loop

framework, and are beyond the scope of this thesis. However, in the previous section outlining the MATLAB/Simulink tools, much more sophisticated models for reaction wheels were developed. This would be a very good starting place for implementing a more sophisticated reaction wheel model in NASA 42. Furthermore, if the actuator dynamics are shown to have minimal impact on the closed loop performance using the Simulink tools, the same should be expected in software-in-the-loop simulations and thus the exclusion of sophisticated actuator dynamics is not unreasonable.

NASA 42 does support modeling internal spacecraft attitude perturbations as a result of static and dynamic wheel imbalance, based off models given in [14]. This work will take advantage of this capability to investigate whether reaction wheel imbalance has an effect on pointing performance or stability. Reaction wheel imbalance forces and torques arise as a result of the fact that the wheels do not spin perfectly around one of their principle axes, and that their center of mass is not colinear with their spin axis.

When the reaction wheel does not spin perfectly around its center of mass, a centripetal force must be imparted on the mass of the flywheel from the bearing assembly in order to maintain circular motion of the center of mass of the flywheel about its axis of rotation. In a reference frame attached to and rotating with the reaction wheel with the z axis pointing along the wheel spin axis, this force is given by

$$\boldsymbol{F} = -m{\omega_w}^2 \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

Where m is the mass of the reaction wheel, ω_w is the rotational velocity of the reaction wheel around its spin axis, and x and y are the instantaneous x and y coordinates of the reaction wheel center of mass in the reaction wheel frame. The constant quantity

$$U_s = m\sqrt{x^2 + y^2}$$

is the static imbalance of the reaction wheel assembly. Assuming that the reaction wheels are not collocated with the center of mass of the spacecraft in which they are mounted—which is

basically always the case—a disturbance torque will arise as a result of this centripetal force, given by

$$\boldsymbol{T}_s = \boldsymbol{r}_w \; x \; C_{bw}\boldsymbol{F}$$

Where $\boldsymbol{r}_w$ is the location of the reaction wheel in the spacecraft body frame whose origin is at the center of mass, and C_{bw} is the matrix that transforms vectors in the wheel frame to vectors in the spacecraft body frame. From this relation, it is apparent that the location and orientation of the wheel with respect to the spacecraft body frame does in fact affect the attitude dynamics. Furthermore, these disturbances show up at frequencies roughly proportional to the rotational speed of the reaction wheels, which is always higher than the bandwidth of the attitude control system for all intents and purposes. Therefore, these disturbances cannot be rejected by the control system. If they are detrimental to the system's pointing performance, the reaction wheels must be deliberately located and oriented within the spacecraft body frame so as to prevent these disturbances from compromising the system performance.

Since the spin axis of a reaction wheel is not perfectly aligned with one of its principle axes, a gyroscopic torque is required to keep the wheel rotating around its spin axis. Assuming the reaction wheel is rotating perfectly around its z axis, this gyroscopic torque in the wheel frame is given by

$$\boldsymbol{T}_d = {\omega_w}^2 \begin{bmatrix} -J_{yz} \\ J_{xz} \\ 0 \end{bmatrix}$$

Where J_{zy} and J_{xz} are products of inertia of the reaction wheel inertia matrix expressed in the reaction wheel frame. The constant quantity

$$U_d = \sqrt{{J_{yz}}^2 + {J_{xz}}^2}$$

Is the dynamic imbalance of the reaction wheel assembly. Similar to torques arising from static imbalance, dynamic imbalance torques also show up at frequencies proportional to the rotational speed of the reaction wheel. However, in this case the location or orientation of the reaction

wheel does not affect the dynamic imbalance torque. Therefore, if this torque compromises pointing performance or stability, it is up to the design of the reaction wheel assembly itself—and the associated manufacturing processes—to reduce the dynamic imbalance down to a tolerable value. Chapter 5 of this thesis will show that the static and dynamic imbalance of CubeSat class reaction wheels does not compromise pointing performance requirements on the order of a few degrees.

4.3.5 Inertia Matrix Uncertainty

Since it is impossible to achieve perfect knowledge of the inertia matrix for a given spacecraft mass distribution, it is necessary to investigate how uncertainty in the inertia matrix may affect closed loop performance and stability. Therefore, in the software-in-the-loop framework, the inertia in the flight software differs from the true inertia used in simulation by a percentage specified as one of the simulation input parameters.

Based on the control algorithm design procedure outlined in Chapter 3, and based on the estimation algorithm that has been implemented in the flight software, the inertia matrix is assumed to be diagonal in the body frame. For CubeSats, this is generally a reasonable assumption. However, it is not exactly the case, and the products of inertia in a body fixed frame whose basis vectors are perpendicular to the CubeSat faces will be very small yet non-zero. Therefore, the nominal diagonal elements of the inertia matrix used in the estimation algorithm and in the design of the control algorithm are increased or decreased by a user-defined percentage. These perturbed quantities are then set as the true diagonal elements in the simulation. The simulation is also given non-zero products of inertia.

4.3.6 Alignment Errors

Alignment errors are simulated separately from the sensor models in NASA 42. For each sensor, the transformation between the sensor output frame and spacecraft body frame must be estimated so that it can be applied to the digital output signal of the sensor in the flight software before being sent to the state estimation algorithm. Therefore, misalignment error for a given sensor is simulated by providing the flight software with an inaccurate sensor to body

transformation relative to the true one that exists in the NASA 42 side of the simulation framework. Since alignment errors can be assumed to be small, the following body to sensor frame transformation is provided to NASA 42:

$$C_{SB} = [C_x(\theta_x + \delta\theta_x)\, C_y(\theta_y + \delta\theta_y) C_z(\theta_z + \delta\theta_z)]^T$$

Where the angles θ_x, θ_y, and θ_z are the nominal ZYX mounting angles from sensor to body frame, and $\delta\theta_x$, $\delta\theta_y$ and $\delta\theta_z$ are the small mounting alignment errors around each sensor axis. When NASA 42 executes the sensor models, it uses the above transformation to project the true body frame value of the physical vector in question into the sensor frame, then applies the output noise. Once the sensor output reaches the flight software, the inverse of the above transformation without the $\delta\theta_x$, $\delta\theta_y$ and $\delta\theta_z$ mounting errors is applied to transform the sensor output into the body frame.

4.3.7 Custom Yaml Input File Design

Due to the inherently complicated and brittle nature of the NASA 42 input configuration files, a more simple interface to the user of the software-in-the-loop simulation tool was desirable. The author of this thesis, in collaboration with software engineers in the Cal Poly CubeSat Laboratory, designed a yaml file structure containing all of the relevant simulation configuration parameters along with a script to parse the Yaml files and convert them into the respective NASA 42 compatible configuration files. To run a simulation, the user must provide the simulation with four Yaml files: (1) a baseline simulation file, (2) a custom simulation file, (3) a baseline spacecraft file, and (4) a custom spacecraft file. The user must also provide the simulation with a fifth configuration file corresponding to the ADCS flight software configuration. The spacecraft Yaml files contain all of the information pertaining the sensor and actuator configuration as well as the mass properties and initial dynamic state of the spacecraft being simulated. On the other hand, the simulation Yaml files contain all simulation specific parameters, such as the start time in UTC, the duration of the simulation, and on/off parameters for spacecraft perturbation models. Baseline files contain every possible simulation parameter, and custom files represent changes or additions to those parameters. Both the baseline and custom Yaml files are parsed into Python

dictionaries, and all of the parameters in the resulting custom dictionary are overlaid onto the baseline dictionary. Then, all of the data in the resulting overlaid dictionary is properly formatted and written to the NASA 42 configuration files. Therefore, the simulation will be configured with all of baseline parameters, with the exception that any parameters specified in the custom Yaml files will overwrite their respective baseline parameters. This input file design not only makes it easier for the user to see all of the simulation parameters in only a few places, but also gives the user the ability to simulate a specific scenario with mostly the same parameters, and perturb some of the parameters off of their baseline values to explore a local region in the parameter space. Appendix A provides an example of the actuator section from a baseline spacecraft file. Although the spacecraft Yaml file contains quite a few parameters, they are organized in such a way that it is relatively easy for the user to find relevant parameters and change them. Furthermore, the order in which these parameters are listed in the file will not change the resulting NASA 42 configuration files, since the parameters are parsed into name based data structures (Python dictionaries). For example, swapping the location of the “Actuators” section with the “Sensors” section in the above file will result in the exact same NASA 42 configuration files, and the simulation will behave identically in either case. Appendix A provides an example of a baseline simulation file. Unlike the baseline spacecraft file, the baseline simulation file has only a few parameters. As seen in the figure, the user can configure the simulation to model specific disturbances to isolate more important ones or simplify the analysis if necessary.

The previous subsections have provided an outline of the relevant NASA 42 capabilities, as well as how this work has contributed to making use of those capabilities in the context of the Cal Poly CubeSat Laboratory ADCS design and analysis workflow. In the next chapter, the full ADCS design and analysis process from conceptual design to critical design will be performed using the tools presented in this work.

A final comment on the use of NASA 42 for a real flight mission: a more rigorous method for developing sensor and actuator models—perhaps utilizing a well-defined system identification procedure—is strongly recommended. Simulating sensors based on generic models and referencing parameters from datasheets is sometimes acceptable, but can be insufficiently

accurate for certain verification applications. However, such an activity is beyond the scope of this thesis.

Chapter 5

VALIDATION OF TOOLKIT UTILITY AND DESIGN APPROACH

In this chapter, the improvements to the ADCS toolkit and the design methodologies outlined in chapters 3 and 4 will be employed to complete the CPCL ADCS design, and show that it could be used to satisfy the system level performance requirements of an ADCS for a large class of modern CubeSat flight missions. Therefore, this chapter will serve as a validation of the toolkit utility by demonstrating that it can be used to both create and validate an example ADCS design. For each control system pointing mode developed in chapter 3, the pointing performance of a notional CPCL ADCS design will be evaluated. Initially, the MATLAB/Simulink tools will be utilized to establish preliminary controller specifications in conjunction with the actuator parameters. Finally, the NASA 42 software-in-the-loop simulation framework will be used to provide a better estimate of the system performance with additional error sources, including sensor noise and estimation dynamics, alignment errors, inertia matrix uncertainty, and orbit knowledge error.

5.1 MATLAB/Simulink: Single Axis Design and Analysis

A reasonable first step in the preliminary design phase of ADCS development is to design a control law and specify actuators that will be capable of supporting such a control law. This task can be performed with the aid of the single axis Simulink tool developed in the previous chapter. As an example, a PID controller is initially designed to achieve the characteristics in table 2. The actuator specifications given in table 3, which are representative of an in-house reaction wheel design current in progress at the CPCL, are used in this analysis. Both the controller and actuator specifications are then used to fully define all the parameters in the single axis Simulink model, and an initial simulation is performed. The PID control law designed for the specifications in table 2 commands reaction wheels with the specifications in table 3 to obtain the closed loop step response. The resulting simulation outputs are then examined to evaluate the combined effectiveness of the controller and reaction wheel.

Table 2: Controller Specifications in Single Axis Simulation

Specification	Value
Open Loop Gain Crossover Frequency	$0.05\ rad/s$
Phase Margin	65°
Sampling Period	$1.0\ sec$
Derivative Filter Cutoff Frequency	$0.5\ rad/s$
Reference Filter Cutoff Frequency	$0.03\ rad/s$
Integral Saturation Limit	$1\ mNm$
Attitude Error Deadband	$0.005\ rad$
Deadband Convergence Time	$120\ sec$

The controller specifications listed in table 2 may look somewhat arbitrary. However, they have in fact been chosen based on engineering judgement and an iterative process.

Figures 21 through 26 provide the system response in both the time and frequency domain as determined by the single axis tool. The axial moment of inertia defining the plant in this simulation is $0.025\ kgm^2$.

Table 3: Actuator Parameters in Single Axis Simulation

Parameter	Value
Maximum Momentum Storage	$5\ mNms$
Spin Axis Inertia	$9.42\ x\ 10^{-7}\ kgm^2$
Maximum Torque	$1.61\ mNm$
Mechanical Speed Time Constant	$0.365\ sec$
Minimum Speed Bit	$12.5\ rpm$
Speed Controller Sampling Frequency	$5.36\ kHz$

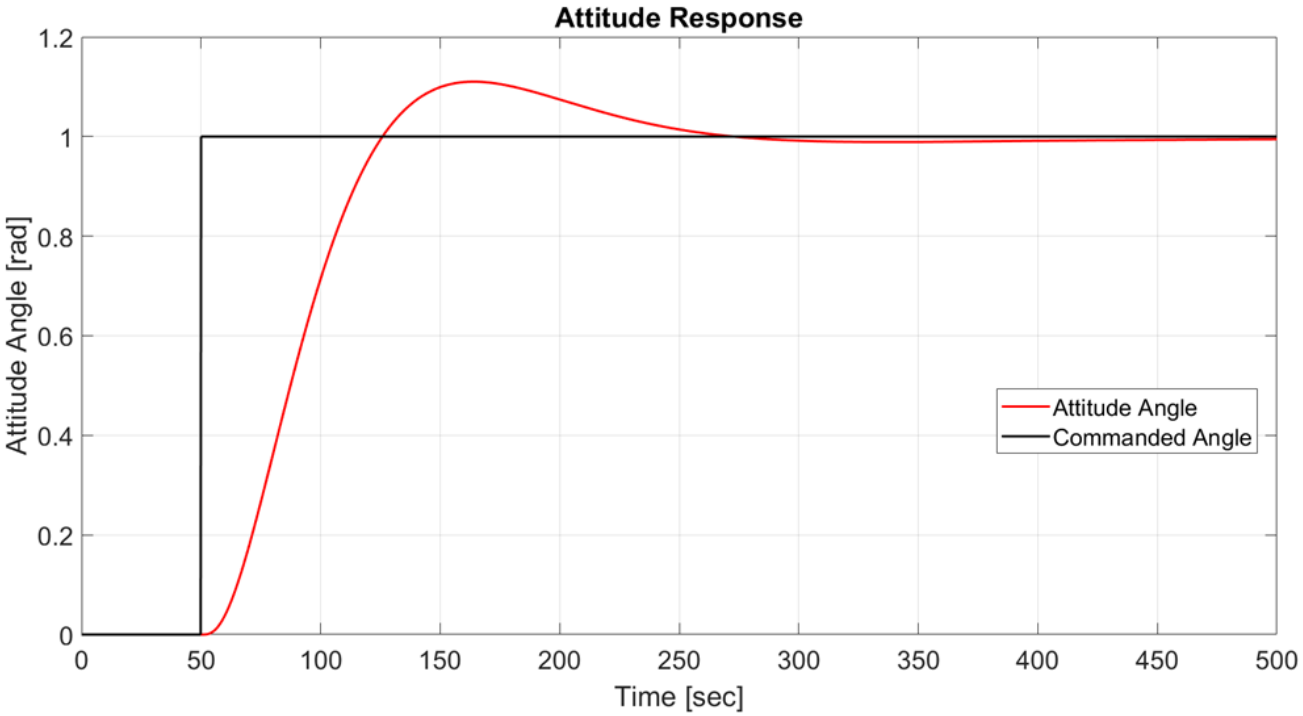

Figure 21: Single Axis Model Closed Loop Attitude Step Response

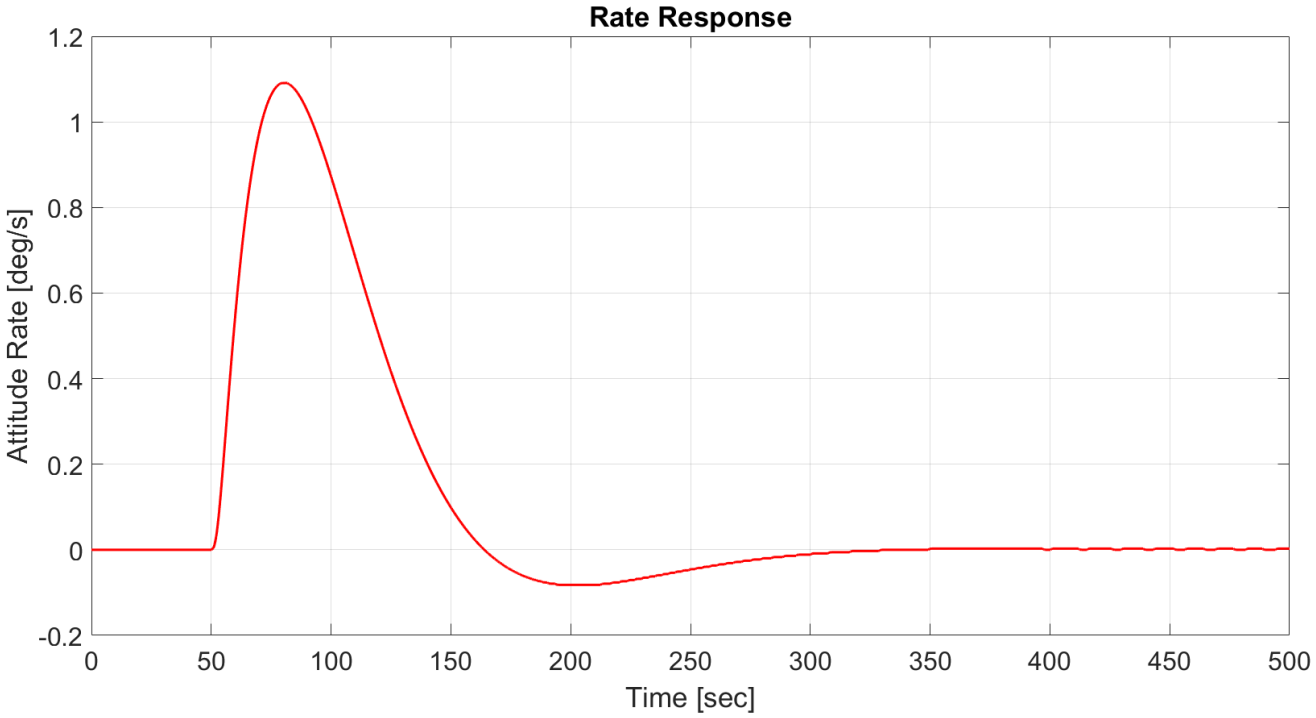

Figure 22: Single Axis Model Closed Loop Rate Response

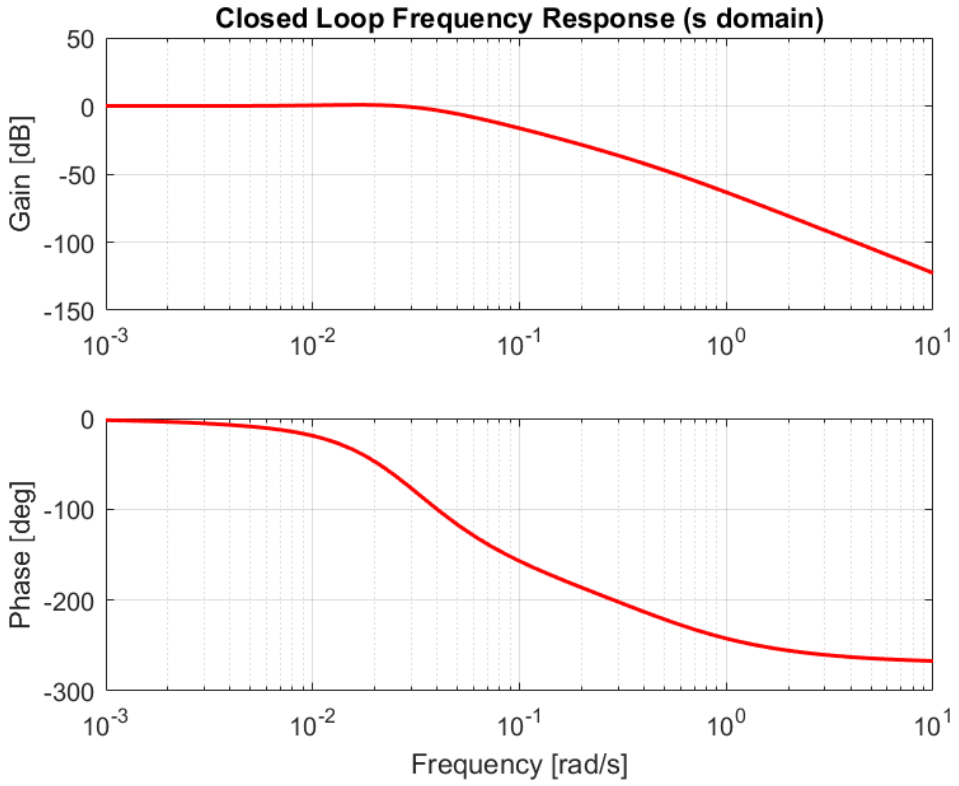

Figure 23: Single Axis Closed Loop Frequency Response

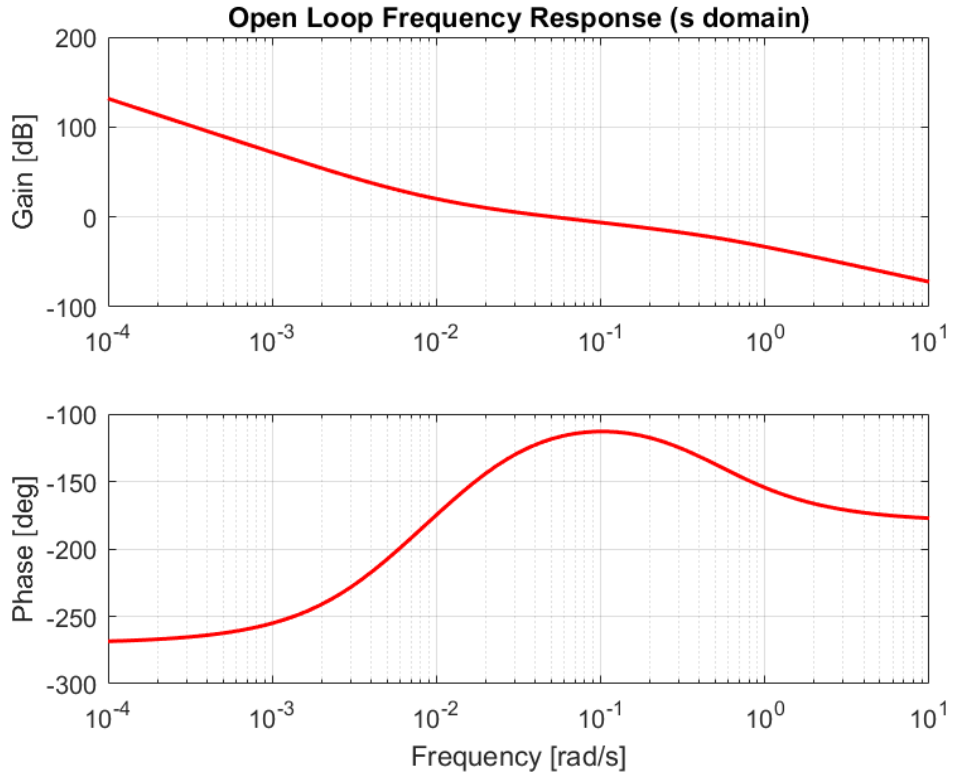

Figure 24: Single Axis Open Loop Frequency Response

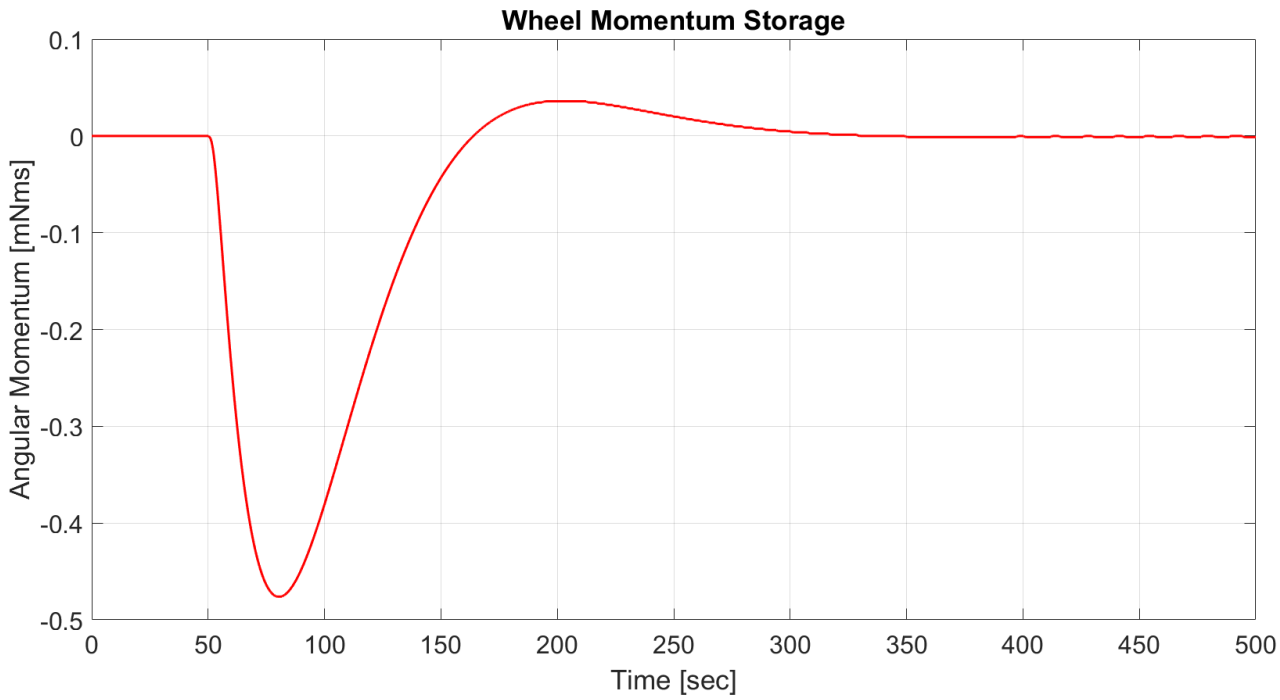

Figure 25: Single-Axis Closed Loop Wheel Momentum Response

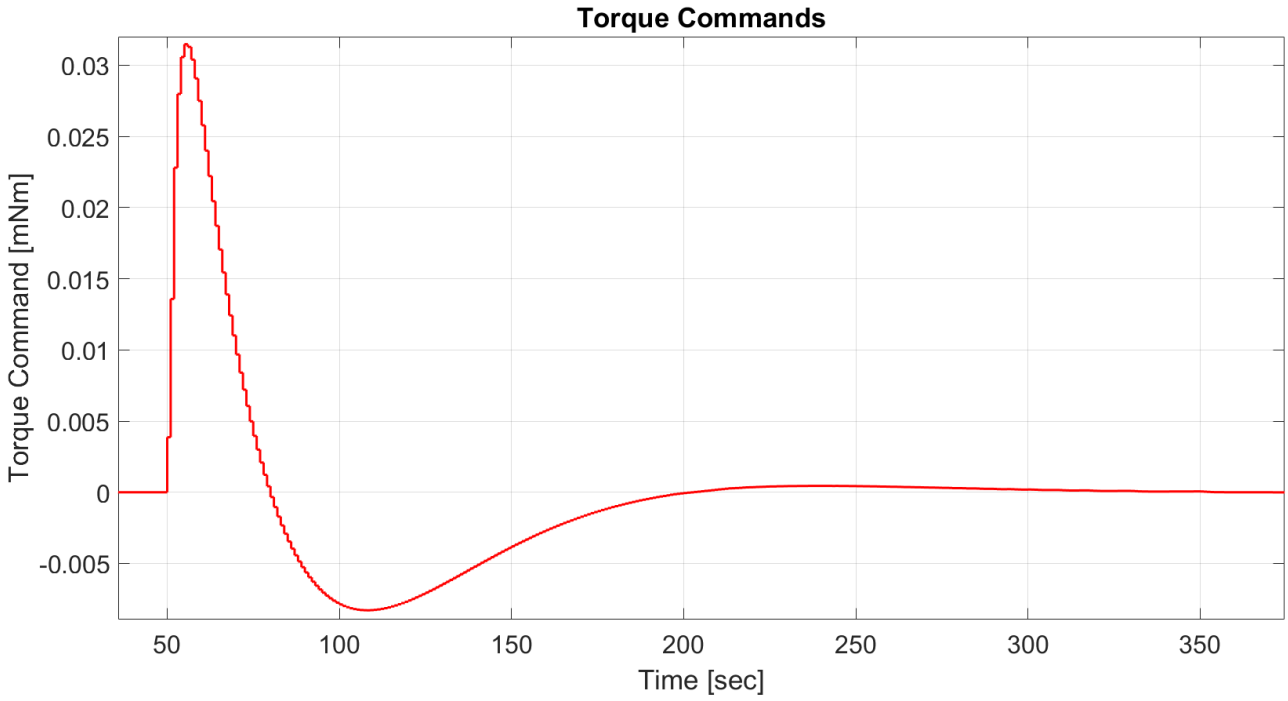

Figure 26: Single Axis Closed-Loop Controller Response to Reference Step

The simulation results in figures 21 through 26 show that a controller with specifications given in table 2, when commanding a reaction wheel with parameters given in table 3, the single axis system performs well. The actuator commands are benign, and the attitude rate does not exceed 1.2 degrees per second, which is well within the linear range of the three axis attitude dynamics. Figure 21 shows that the system takes about 4 minutes to settle on the commanded attitude angle, which is an acceptable amount of time for a large angle attitude maneuver in most mission scenarios. Moreover, the steady state limit cycle behavior is significantly mitigated by the deadband logic, as seen in figure 27. Eventually, the wheel angular momentum stops bouncing between zero and the momentum corresponding to the minimum resolution of the digital wheel speed controller commands. Although this actuator model does not take into account effects such as stiction and other nonlinear behaviors at low wheel speeds, this serves as a proof of concept that the deadband logic presented in this thesis can be used to mitigate undesirable steady state actuator limit cycle behavior.

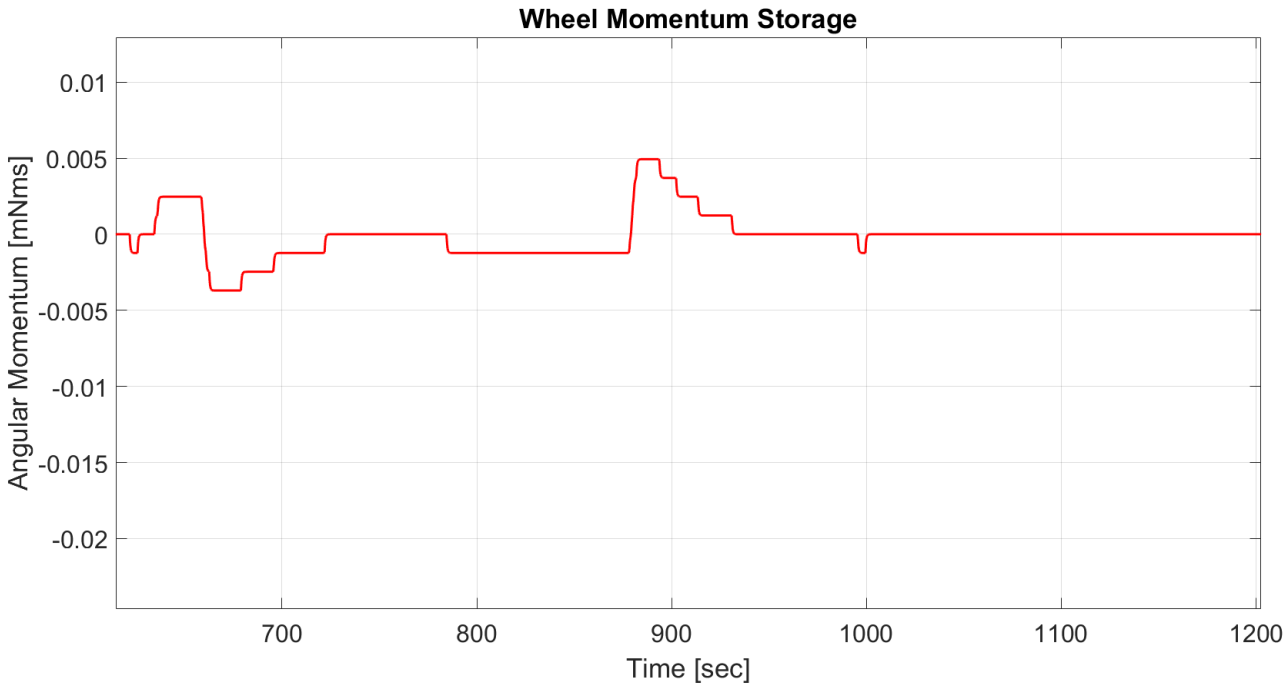

Figure 27: Steady State Wheel Momentum Limit Cycle Suppression

5.2 MATLAB/Simulink: Three Axis Design and Analysis

After establishing preliminary verification that the actuator specified in table 3 can support the corresponding control law designed to achieve the specifications given in table 2, the analysis is extended into the three axis Simulink tool. In this tool, the user has the ability to investigate the control system behavior under the influence of more interesting attitude profiles, not just a step command. For the sake of the analysis presented in this thesis, each of the control modes outlines in chapter 3 except for ephemeris pointing will be analyzed, using the same controller and actuator specifications provided in the previous subsection. A good starting place for analysis using the three axis analysis tool is to initialize the system with zero initial conditions and command a constant attitude (i.e. simulate a rest to rest attitude slew maneuver). In the case that there are any issues with the design, finding those issues in a rest to rest simulation makes debugging and further iteration significantly easier. Figures 28 through 30 show the salient outputs from the three axis Simulink tool for a rest to rest maneuver. In this simulation, the controller and actuator specifications are identical to those given in the single axis case. Furthermore, the inertia matrix is not perfectly diagonal. Instead, the principle frame is a rotated

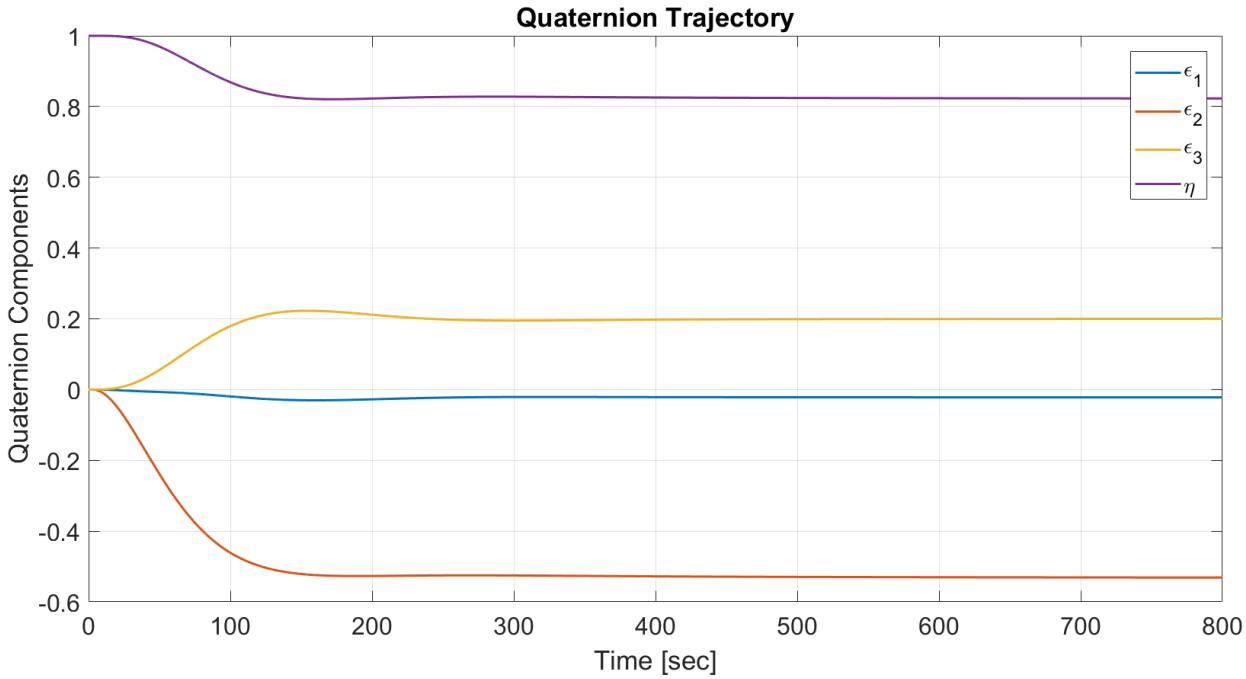

Figure 28: Three Axis Nonlinear Rest-to-Rest Attitude Response

degrees around each body frame axis to analyze the system performance with an inertia matrix that is not given in the principle frame.

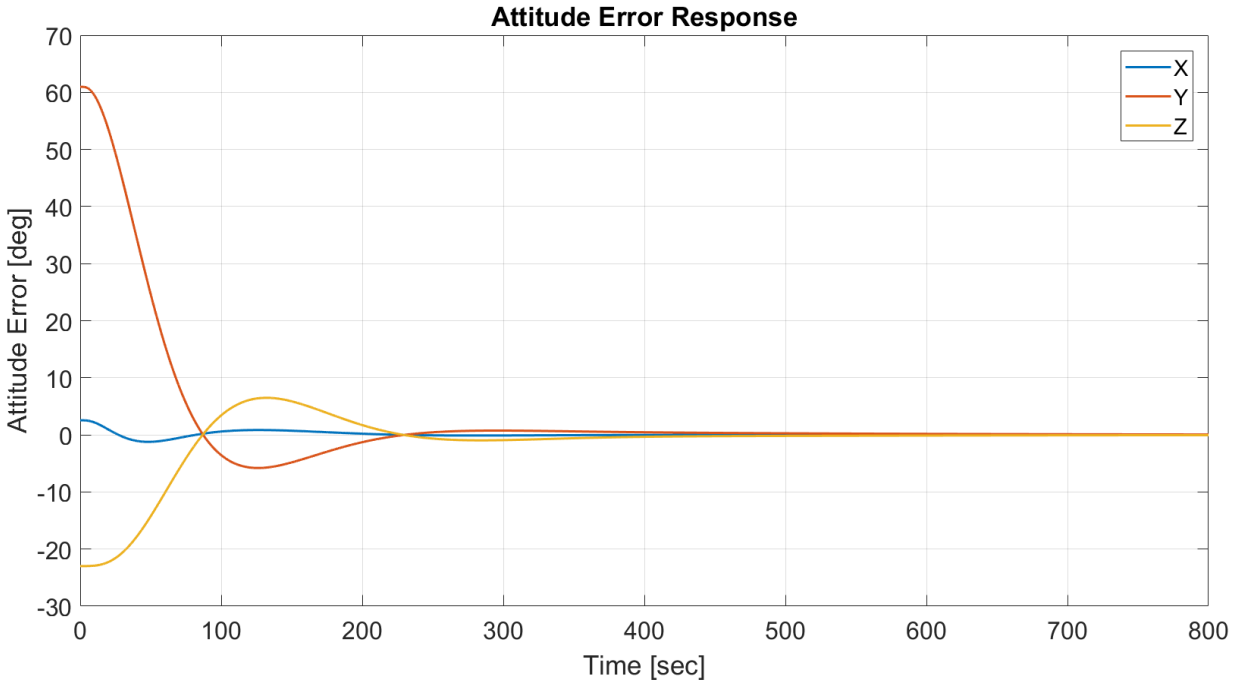

Figure 30: Three Axis Nonlinear Rest-to-Rest Attitude Error Response

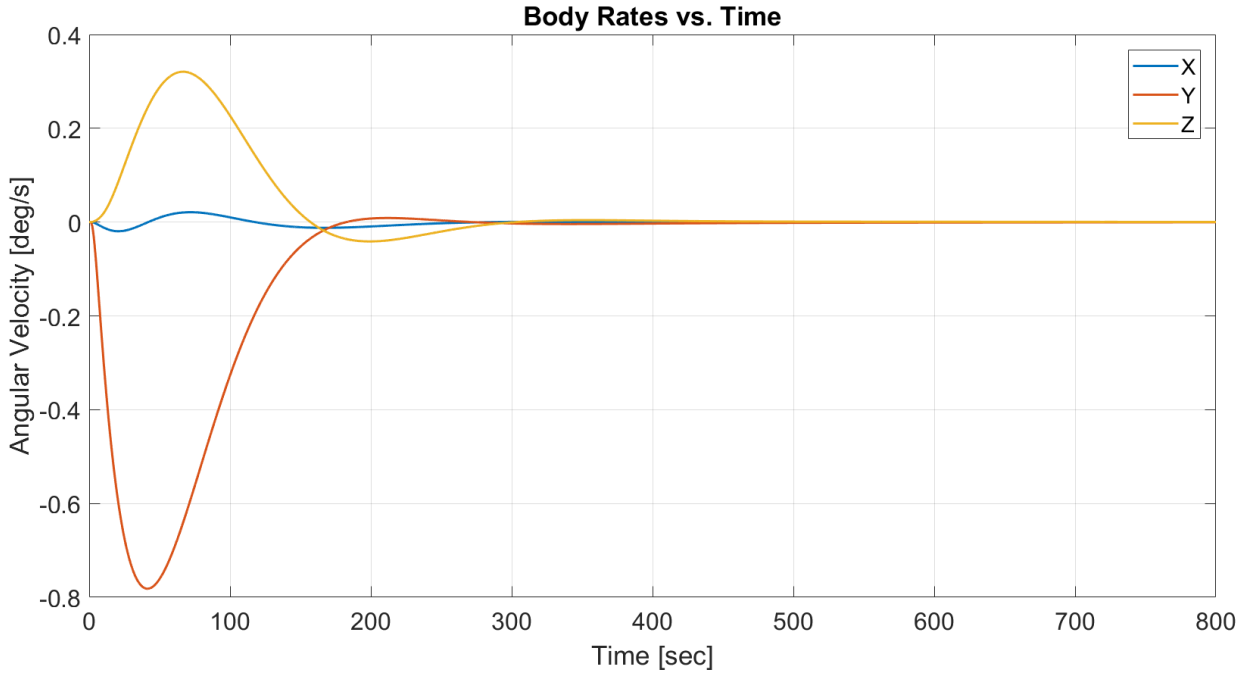

Figure 29: Three Axis Nonlinear Rest-to-Rest Body Rate Response

In figure 28, the commanded quaternion corresponds to a ZYX attitude sequence through 45, -60, and -30 degree angles, respectively. Figure 30 shows the attitude error response.

The rest to rest maneuver performs as expected based on figures 28 through 30. The next pointing mode of interest is nadir pointing. In this mode, the reference filter will be disabled as it gives rise to nonzero steady state attitude error due to the small phase lag at the reference frequency of interest. A potential mitigation to this issue is to stop providing the attitude control algorithm with the filtered reference signal when the absolute value of the unfilitered attitude error around each axis is within a deadband for a specified duration, similar to the deadband logic implemented in the single axis tool. However, this logic has not yet been developed for the three axis tool, and is beyond the scope of this thesis. This topic will be elaborated upon in the future work chapter.

Initially, the controller specifications are left unchanged for the nadir pointing mode to investigate whether the same gains and filter coefficients can achieve good performance. Figures 31 through 33 show the salient outputs from a nadir pointing mode simulation.

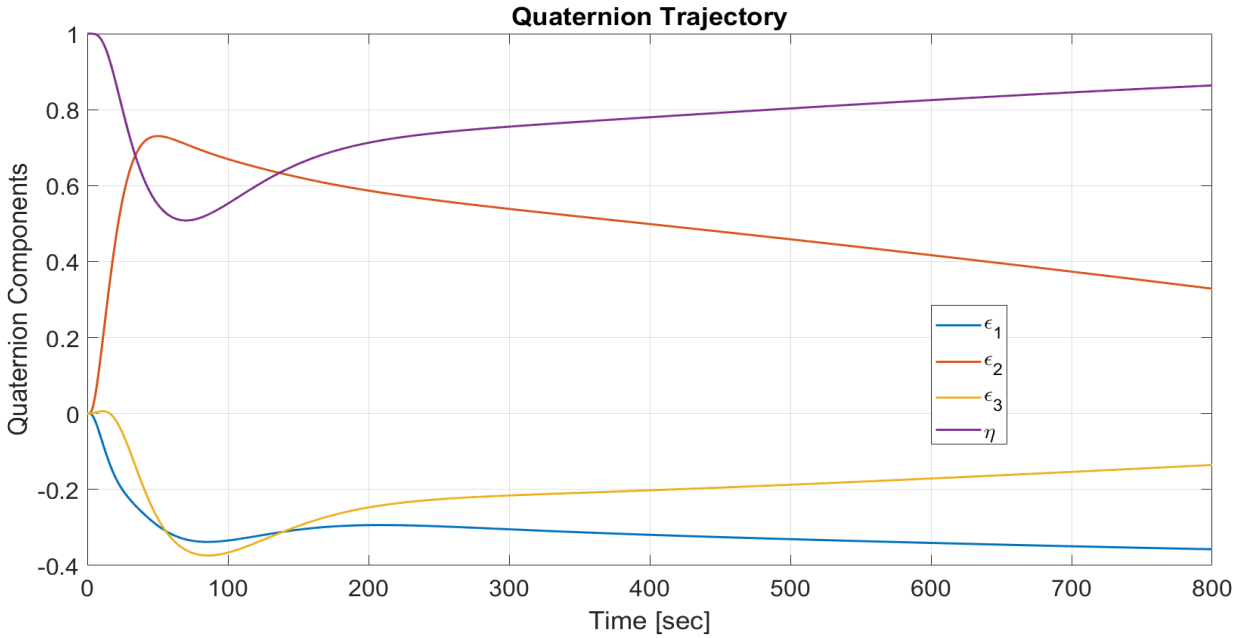

Figure 31: Three Axis Nonlinear Nadir Pointing Attitude Response

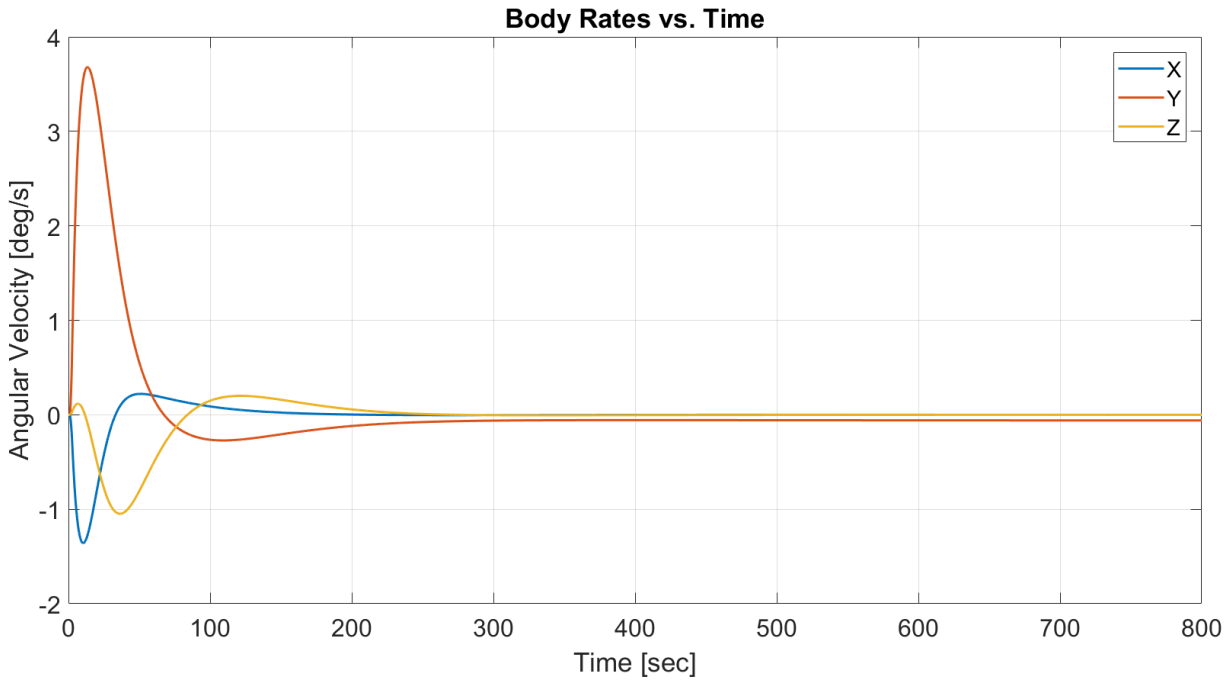

Figure 33: Three Axis Nonlinear Nadir Pointing Body Rate Response

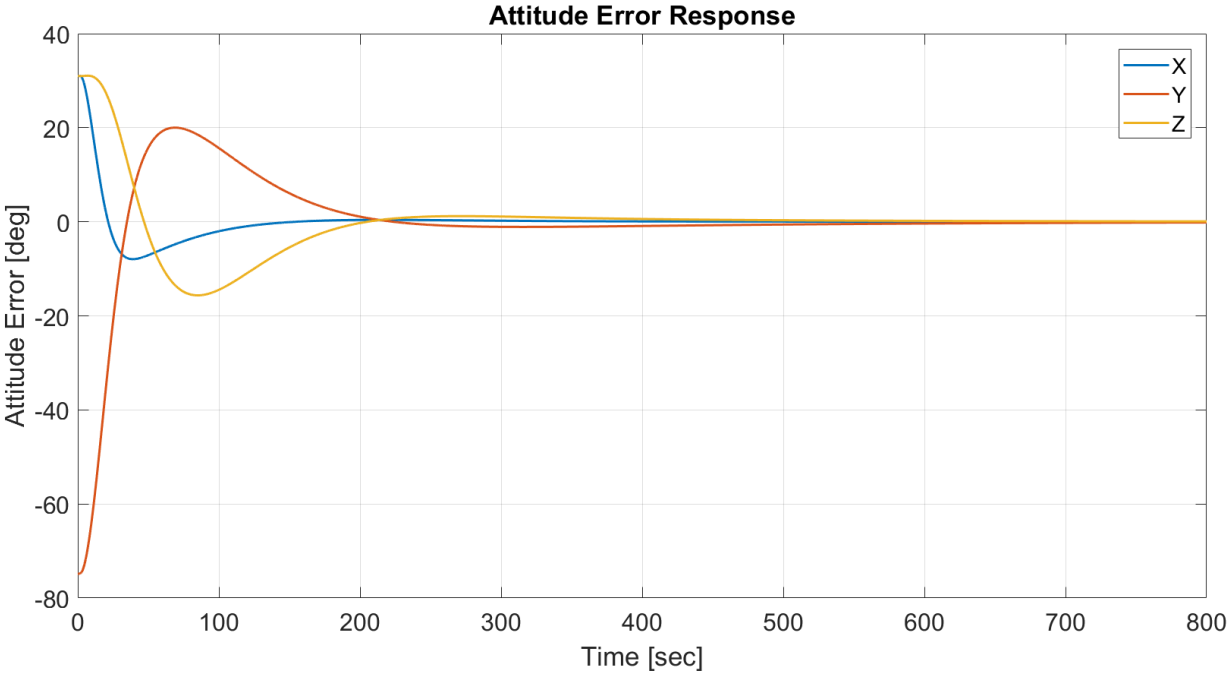

Figure 32: Three Axis Nonlinear Nadir Pointing Attitude Error Response

As seen in figures 31 through 33, the closed loop system still performs well in nadir pointing mode with the same set of controller specifications. Furthermore, notice in figure 31 that the body rates still remain relatively small; in this case less than 4 degrees per second. Therefore,

the small angular rate assumptions made in the design of the control algorithms remain valid in this mode. Figure 33 shows that the attitude errors are successfully driven to zero.

The next pointing mode that will be analyzed is the ground pointing mode. In this mode, the frequency content of the reference signal is much more dependent on the orbit geometry relative to the commanded ground location. Therefore, selection of the control system specifications is much less obvious and will likely require multiple iterations. Furthermore, selection of the body fixed pointing axis in this mode is also important. If an axis that is far away from the major axis of inertia (e.g. the largest principle axis) is commanded, the spacecraft will have a harder time maintaining pointing. In this work, the minor axis of inertia (+Z) is chosen as the commanded body fixed axis for a roughly axisymmetric mass distribution (i.e. a 3 U CubeSat). The spacecraft is simulated in an approximately eccentric (e ≈ 0.02) low Earth orbit with an inclination of 45 degrees. The commanded body fixed location has a longitude of 60 degrees and a latitude of 50 degrees, slightly higher than the inclination. Finally, the open loop gain crossover frequency was increased to $0.075\ rad/s$ and the reference filter was disabled so that the

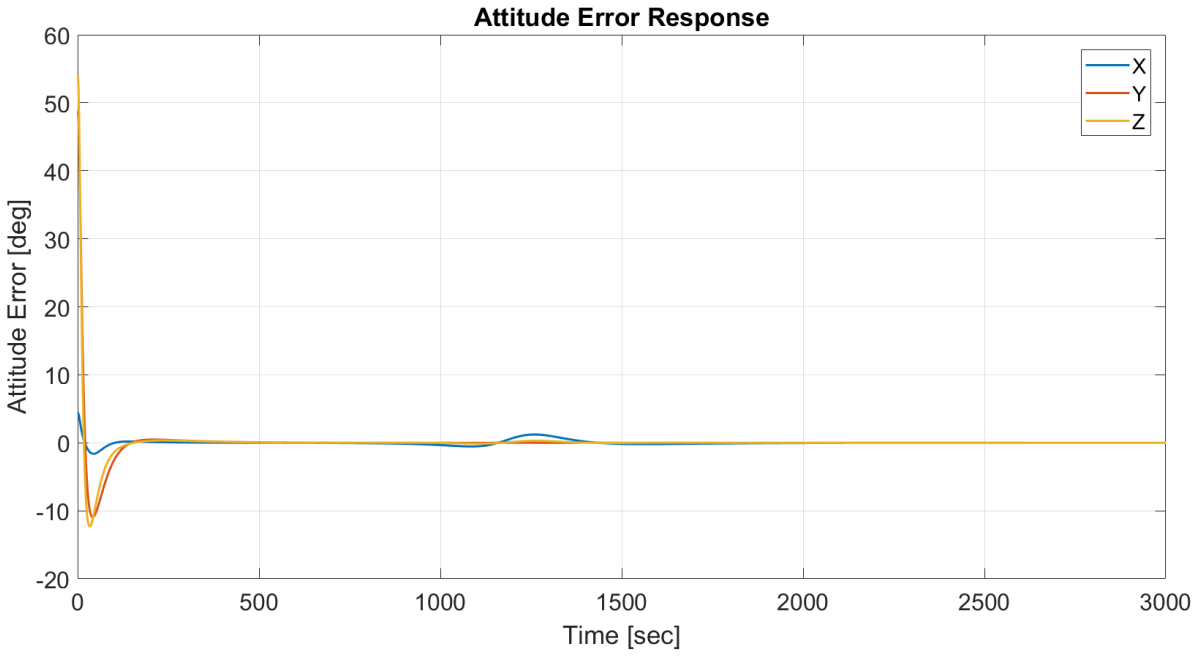

Figure 34: Three Axis Nonlinear Ground Pointing Attitude Error Response

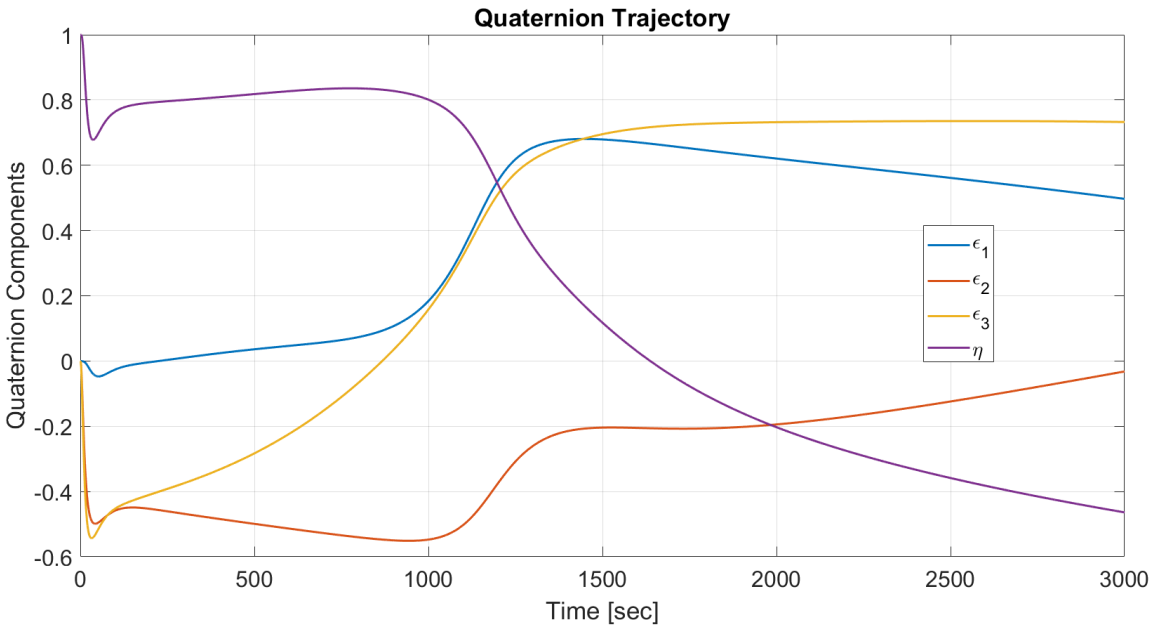

Figure 36: Three Axis Nonlinear Ground Pointing Attitude Response

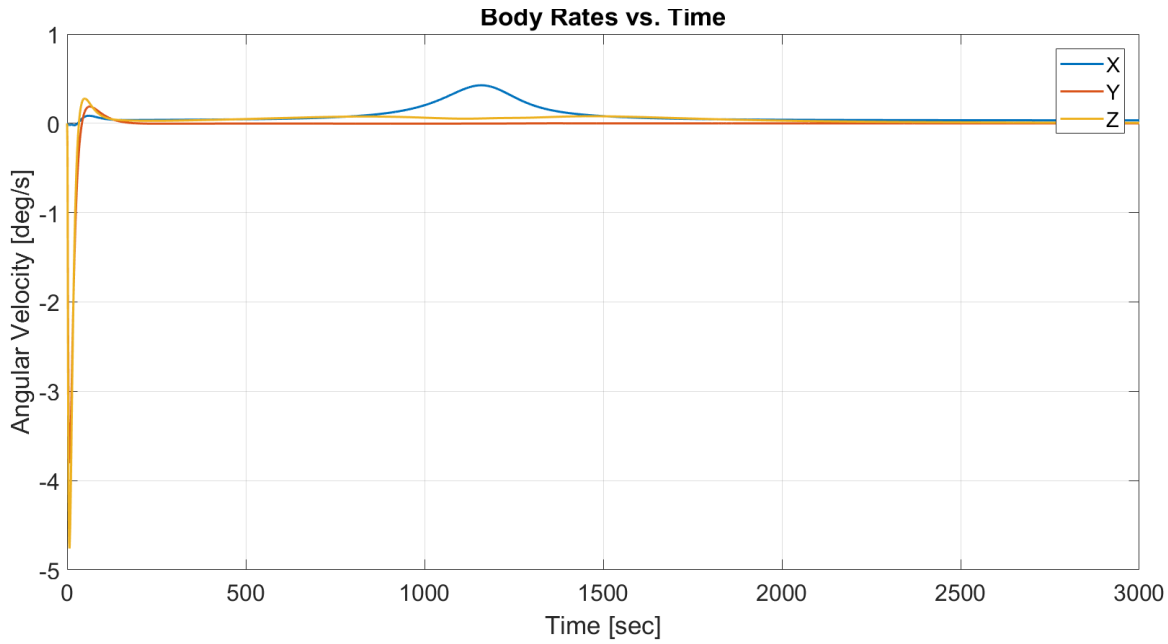

Figure 35: Three Axis Nonlinear Ground Pointing Body Rate Response

controller can better track slightly higher frequency reference signals. Figures 34 through 36 show the outputs for this simulation.

As seen in figure 34, the attitude errors are successfully driven to near zero in the ground pointing mode. At around 1200 seconds into the simulation, the attitude errors grow slightly and

return to zero again – this happens near the same time at which the distance between the spacecraft and the commanded ground location is minimized. When this happens, the reference quaternion starts to change at a frequency for which the closed loop gain and/or phase are nonzero. However, as seen in figure 37, the attitude error around each axis within this time window does not exceed a few degrees, which is tolerable for most CubeSat missions. If a smaller attitude error transient during this time is desired, the control system bandwidth can be increased.

Figure 35 shows that the body rates in the ground pointing mode, for this particular simulation configuration, do not exceed 5 degrees per second around each axis. These rates are still acceptable in terms of staying within the linear region of the spacecraft dynamics.
Finally, it is important to note that in ground pointing mode, very sharp changes in the reference quaternion can occur depending on the orbit geometry relative to the commanded ground location. The three axis tool provides a means of analyzing and mitigating such scenarios.

Figure 37: Brief Attitude Error Transient During Ground Pointing

The final pointing mode that will be analyzed in this work with the three axis tool is the spin pointing mode. The body z axis is chosen as the body fixed pointing axis, with a commanded

spin rate of 1 degree per second. The commanded inertial axis is the x direction of the inertial frame (in most definitions of ECI, the first point of Aires). As seen in figure 38, the body rates

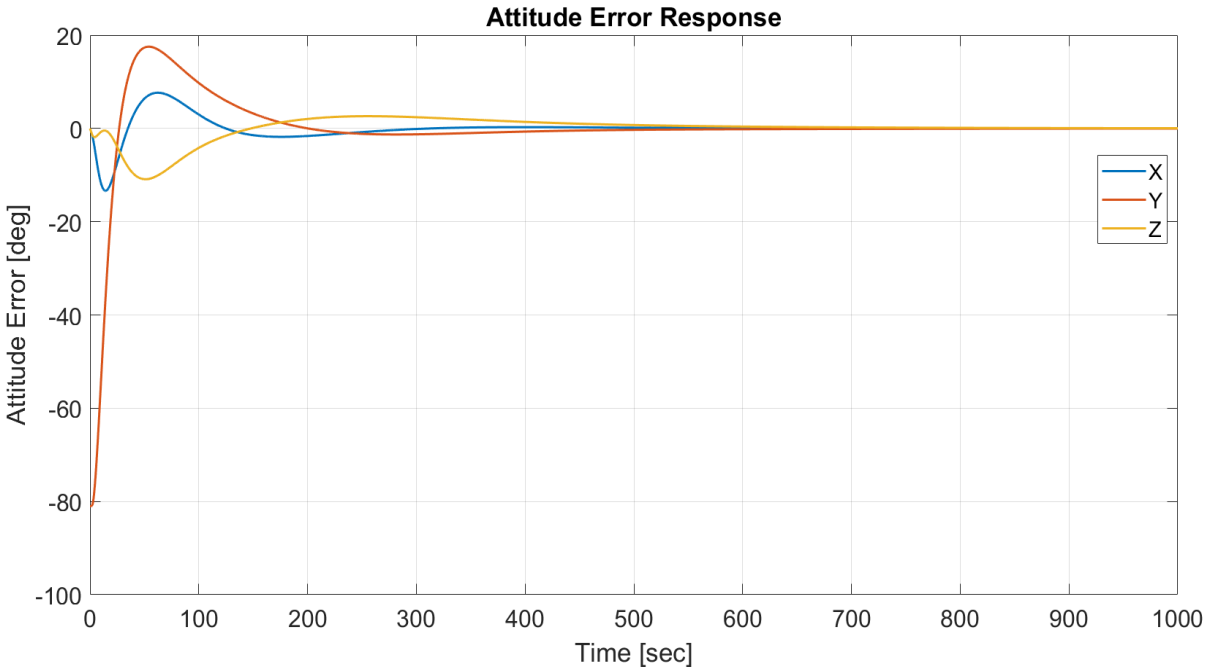

Figure 39: Three Axis Nonlinear Spin Pointing Attitude Error Response

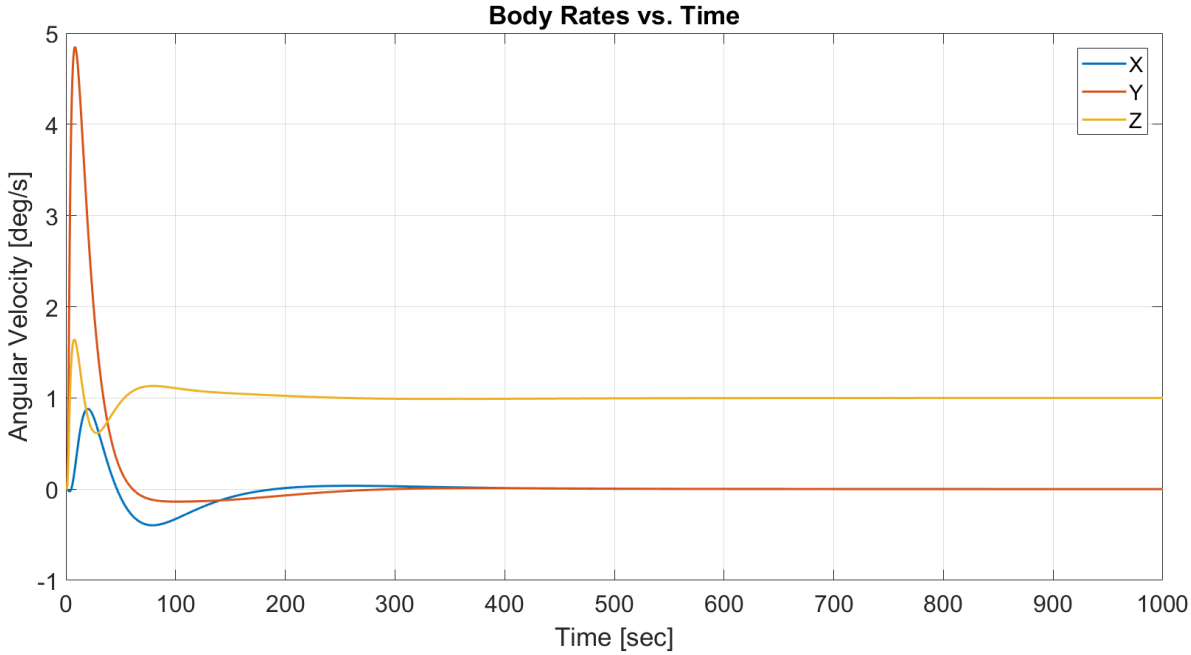

Figure 38: Three Axis Nonlinear Spin Pointing Body Rate Response

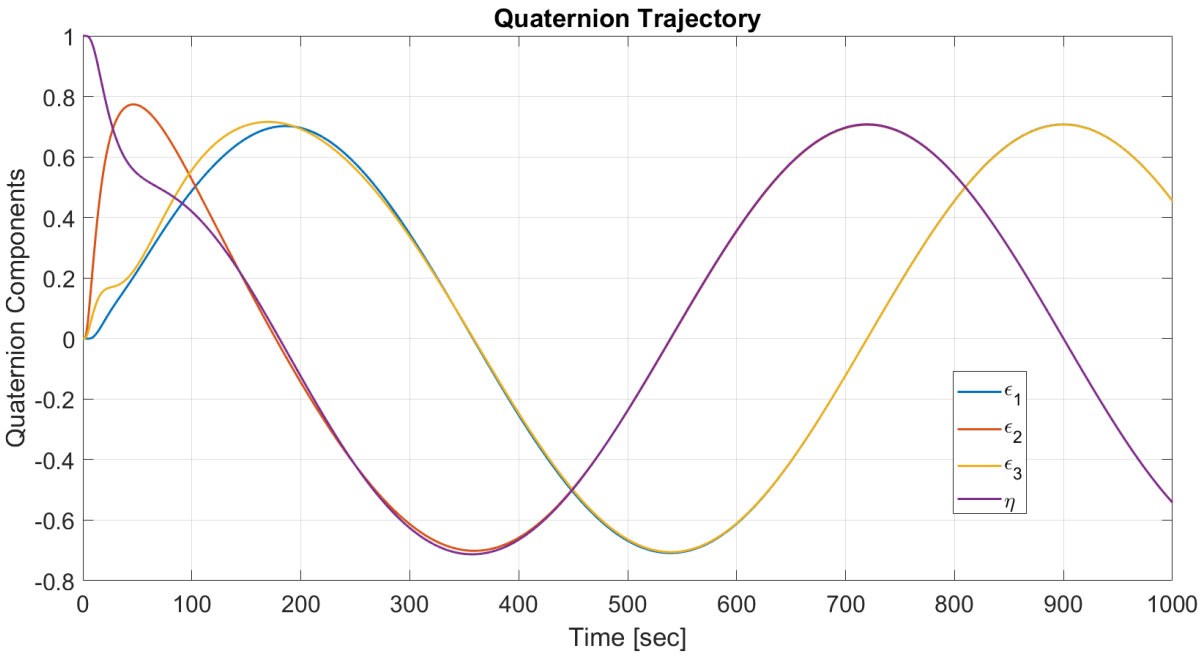

Figure 40: Three Axis Nonlinear Spin Pointing Attitude Response

around the x and y axes converge to zero, while the z axis rate converges to the commanded rate. Furthermore, the attitude errors are successfully driven to zero as seen in figure 39. It is important to note that in this mode, the attitude error around the commanded body fixed axis is not relevant from the perspective of pointing accuracy. Finally, choosing a relatively small (less than about 5 degrees per second) commanded spin rate is critical, since the control laws are designed for a system linearized around small angular rates.

At this point, the success of four pointing modes in a single simulation case has been demonstrated with the three axis MATLAB/Simulink tool. Although this does not provide an exhaustive analysis, it does show that as a whole, this tool has significant utility. Further analysis of each of the pointing modes under multiple different simulation configurations has been performed, and so far, no egregious or unexpected behavior has arisen. These cases will not be shown in this thesis for the sake of conciseness.

Desaturation is the final control system mode to be analyzed within the context of the three axis Simulink tool developed in this work. To test the desaturation control law, the reaction wheel momenta are initialized to non-zero values, and the attitude controller attempts to maintain the initial attitude. Figure 41 shows the wheel momenta across the entire simulation. As expected,

the wheel momenta decrease on average across the desaturation maneuver and settle to near zero. In this case, it took the desaturation controller about 25 minutes to remove approximately 1.6 $mNms$ of angular momentum.

In this simulation, the spacecraft orbit is slightly non-circular with an inclination of 52 degrees. The desaturation control gain is set to 120 Am^2/Nms. It is important to note that with this particular desaturation control law, the spacecraft orbit, reaction wheel characteristics, and desaturation gain largely affect the performance of the desaturation maneuver. The reason for this is that the magnetorquers are underactuated: they can only generate a torque that is perpendicular to the local magnetic field vector. Therefore, for some orbits, the geometry of the magnetic field across the orbit is disadvantageous with respect to the torque directions required to remove the angular momentum of the reaction wheels. Careful and thorough analysis must be performed in order to fully understand how the desaturation control law behaves in different scenarios. This simulation is only one scenario – it is possible that in different orbits and attitudes, the performance will be significantly better or worse.

The spacecraft attitude was affected significantly during this desaturation maneuver as seen in figure 43. However, once the wheel momenta became sufficiently low and the magnetorquers stopped actuating, the attitude controller was able to bring the spacecraft back to the commanded attitude. Such an interaction illustrates a tradeoff with this desaturation control law: if a higher desaturation gain is chosen to remove angular momentum faster, the attitude error transient during the maneuver is more pronounced. One way to mitigate this issue is to feed forward the external torque from the magnetorquers to the PID controller during the maneuver. However, to keep the implementation of algorithms simpler, this feedforward action not included.

Figure 42 shows the dipole moments commanded to the magnetorquers from the desaturation control law during this simulation. Saturation clearly occurs, but this has no adverse effect on the performance and is not a concern from a hardware operation perspective.

At this point, both the single axis and three axis MATLAB/Simulink tools have been utilized to develop and verify a preliminary control system design for the Cal Poly CubeSat Laboratory. The preliminary design consists of actuators whose high level performance

parameters like strength, saturation, and resolution are relatively well known. A discrete time PID control law was then shown to perform effectively when commanding these actuators in simulation for all of the control system operational modes presented in Chapter 3 with the exception of the ephemeris pointing mode, whose behavior is very similar to ground pointing.

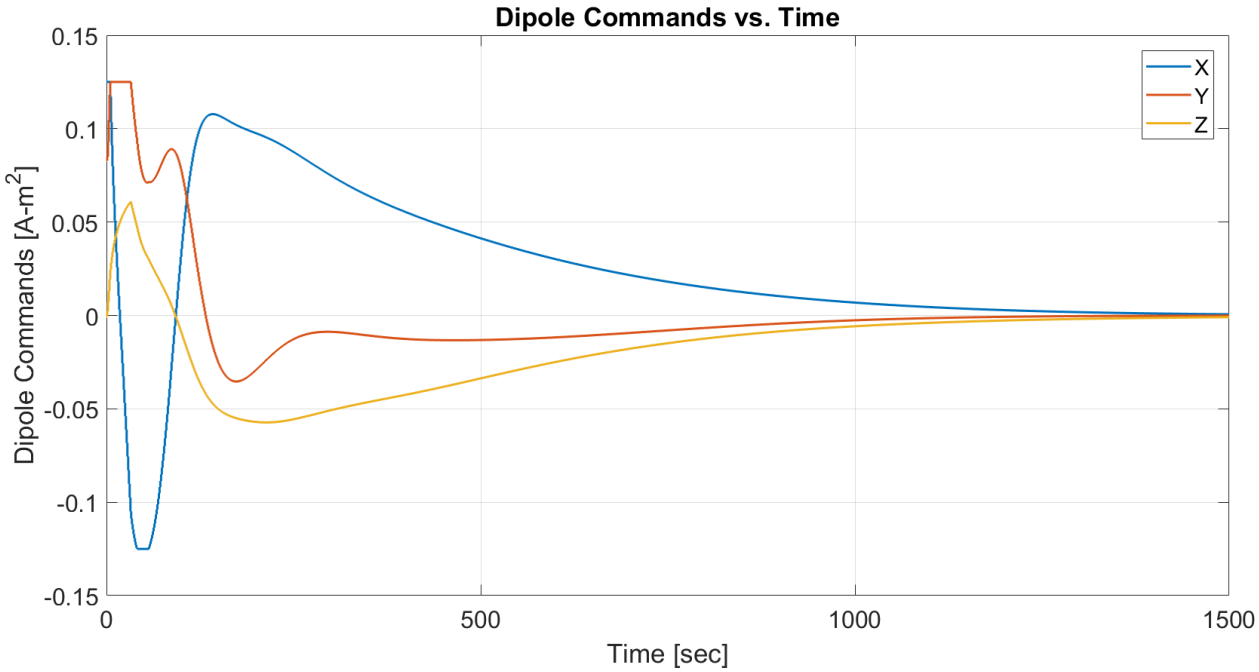

Figure 42: Desaturation Dipole Commands

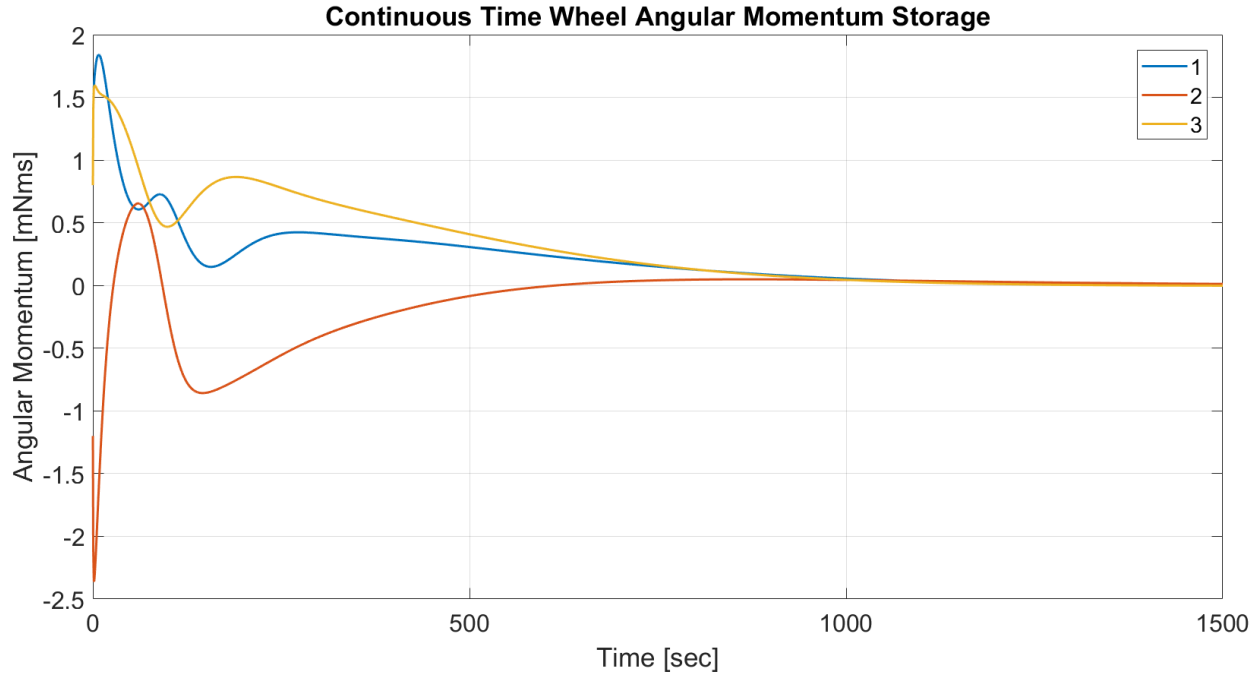

Figure 41: Desaturation Wheel Momentum Profile

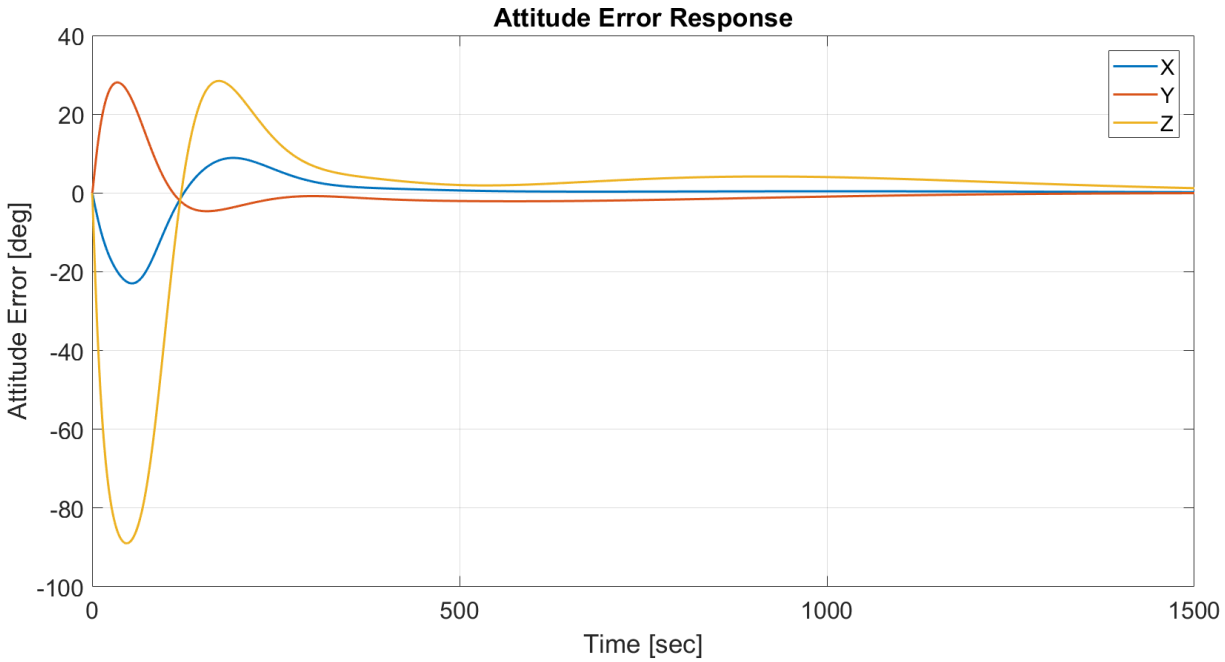

Figure 43: Desaturation Attitude Error Response

The next step in the overall ADCS design process is to incorporate higher fidelity modeling features that capture the effects of sensor noise and additional sources of uncertain parameters that may affect the closed-loop system level performance. Furthermore, transitioning into a simulation environment in which the entire flight-software ADCS process is invoked provides much better validation confidence. In the next section, the software-in-the-loop environment outlined in chapter 2 of this thesis will be utilized to increase the resolution of the overall ADCS design and provide more confidence that it will meet similar pointing performances that were found in Chapter 1.

5.3 Software-in-the-Loop Simulation Framework: Design and Analysis

In this final section of Chapter 5, the software-in-the-loop simulation environment employing NASA 42 as the truth model will be utilized to predict the system level pointing performance of a completed version of the CPCL ADCS design and compare it to the baseline performance requirements derived in Chapter 1. The entire anticipated hardware suite includes reaction wheels and magnetorquers as actuators, and sun sensors, rate gyros, and magnetometers as sensors. The specifications of the sensors and actuators provided to NASA 42 are shown in table 4. The sensor dynamics models used in the simulation environment were provided in Chapter 4. On the other hand, the actuator dynamic models are vastly simplified in the NASA 42 environment relative to the ones currently implemented in this work from Chapter 3. NASA 42 is configured to model the spacecraft as a rigid body under the influence of solar radiation pressure forces and torques, gravity gradient torques, and aerodynamic drag forces and torques. Figure 44 shows the ADCS sensor and actuator layout that is being simulated.

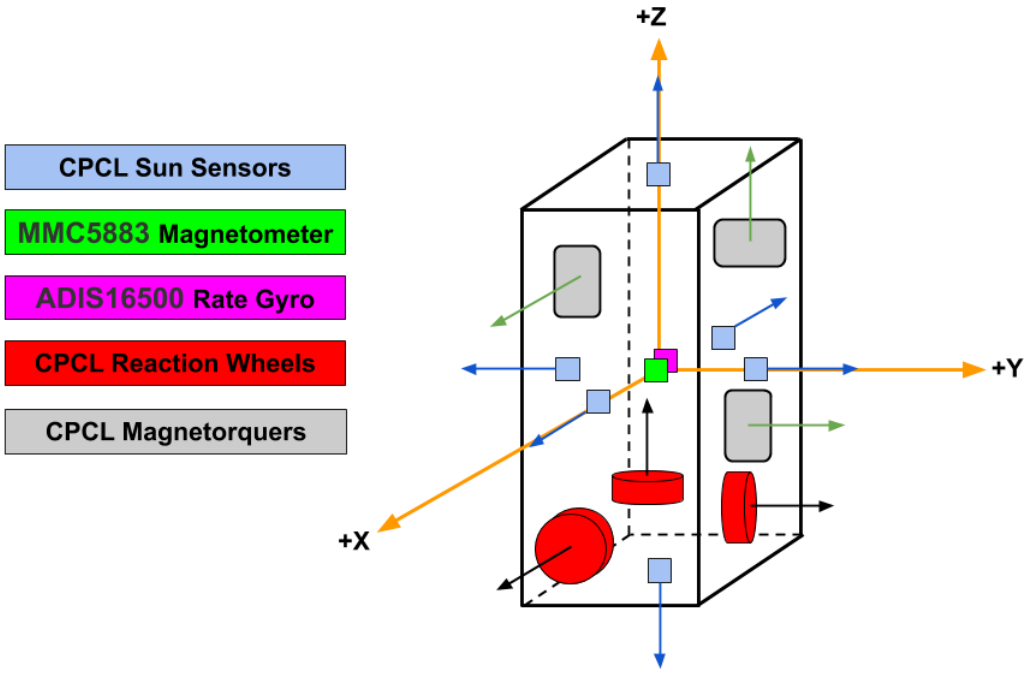

Figure 44: Vehicle ADCS Configuration in Software-in-the-loop Simulation

Table 4: Sensor and Actuator Characteristics and Performance

CPCL Sun Sensors	
Parameter	**Value**
RMS Angle Noise (σ_θ)	0.5°
FOV (sensor x direction)	45°
FOV (sensor y direction)	45°
Mounting Alignment Errors	$\sim 0.5^\circ$ in each axis
Output Resolution	0.1°
Body Frame Locations	Center of each panel facing outward
MMC5883 Magnetometer	
Parameter	**Value**
RMS Output Noise (σ_m)	$4\ x\ 10^{-8}\ Tesla$
Scale Factor	$1000\ ppm$
Saturation	$8\ x\ 10^{-4}\ Tesla$
Mounting Alignment Errors	$\sim 1^\circ$ in each axis
Output Resolution	$2.5\ x\ 10^{-8}\ Tesla$
Body Frame Location [x, y, z]	Origin

ADIS16500 Rate Gyro	
Parameter	**Value**
Angle Random Walk [x, y, z]	$[0.29, 0.29, 0.32]\ ^\circ/\sqrt{hr}$
1σ In-run Bias Stability [x, y, z]	$[8.1, 8.1, 8.1]\ ^\circ/hr$
Scale Factor	$5000\ ppm$ in each axis
Output Range	$+/-125\ ^\circ/s$ in each axis
Mounting Alignment Errors	$\sim 1^\circ$ in each axis
Body Frame Location	Origin
CPCL Reaction Wheels	
Parameter	**Value**
Maximum Torque	$1.61\ mNm$
Momentum Storage	$5\ mNms$
Static Imbalance	$0.003\ gcm$
Dynamic Imbalance	$0.003\ gcm^2$
Mounting Alignment Errors [az, el]	$[0.5\ 0.5]^\circ$
Body Frame Location	Clustered around $[0, 0, -0.15]\ m$

CPCL Magnetorquers	
Parameter	**Value**
Dipole Saturation	$0.07\ Am^2$
Mounting Alignment Errors [az, el]	$[0.5\ 0.5]^\circ$

In the following subsections, simulation outputs are presented for each of the control modes outlined in Chapter 3. In all cases, the spacecraft is initialized at the home quaternion with zero angular rates. Simulation specific parameters are listed in table 5. In each simulation, the spacecraft is in sunlight to analyze the performance when sun sensor measurements are available as inputs to the estimation algorithm. Analysis of closed-loop behavior and performance during which the spacecraft is in eclipse is not included in this work.

In each mode, a python implementation of the control algorithm is handling the same logic that would be handled by a flight software implementation of the *Controller* type. In all cases except for the inertial pointing mode, the digital reference filter does not exist in the algorithm implementation.

Table 5: Simulation Specific Parameters

Parameter	Value
Orbit Epoch (UTC)	11 May 2020, 14:30:15.00
Orbit Geometry	$\sim 400km$ circular, 28° inclined
Inertia Matrix Uncertainty	10% in each principle axis

5.3.1 Inertial Pointing

The salient results of a rest-to-rest attitude slew maneuver are given in figures 45 and 46. The attitude step response in figure 45 exhibits small but noticeably different behavior in the time domain relative to the response in the three axis Simulink tool. These differences are easily attributable the aggregate effects from estimator dynamics and sensor noise, reaction wheel imbalance and misalignment, and external disturbances. The estimation algorithm and control algorithm are interacting successfully in this simulation case, giving rise to desirable closed loop performance. The steady state pointing error achieved in this simulation case is about **3.2°** in the

eigenaxis/angle sense. This pointing error is broken down into the more relevant per axis pointing errors in table 6.

Table 6: Steady State Inertial Pointing Performance in SITL Simulation

Body Axis	Steady State Attitude Error (deg)
X	2.1
Y	1.8
Z	1.6

The per axis pointing errors are all bounded by **2.1°**. Incidentally, this is very similar in magnitude to the pointing performance requirements as investigated in Chapter 1. Therefore, this simulation provides some level of validation that the per-axis inertial pointing accuracy of the CPCL ADCS design is competitive for use in a large class of CubeSat missions. These promising results also provide an initial validation of the guidance and control algorithms presented in this thesis in terms of their robustness to modeling uncertainty, plant parameter uncertainty, noise in the

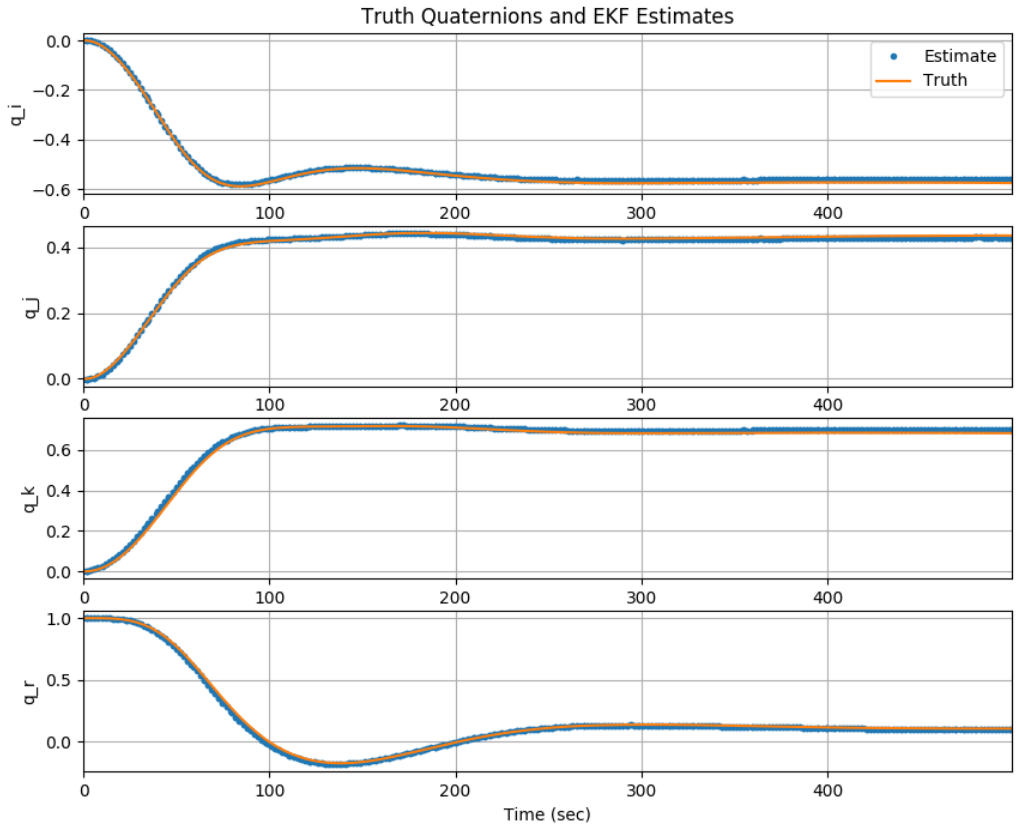

Figure 45: Software-in-the-loop Rest to Rest Attitude Response

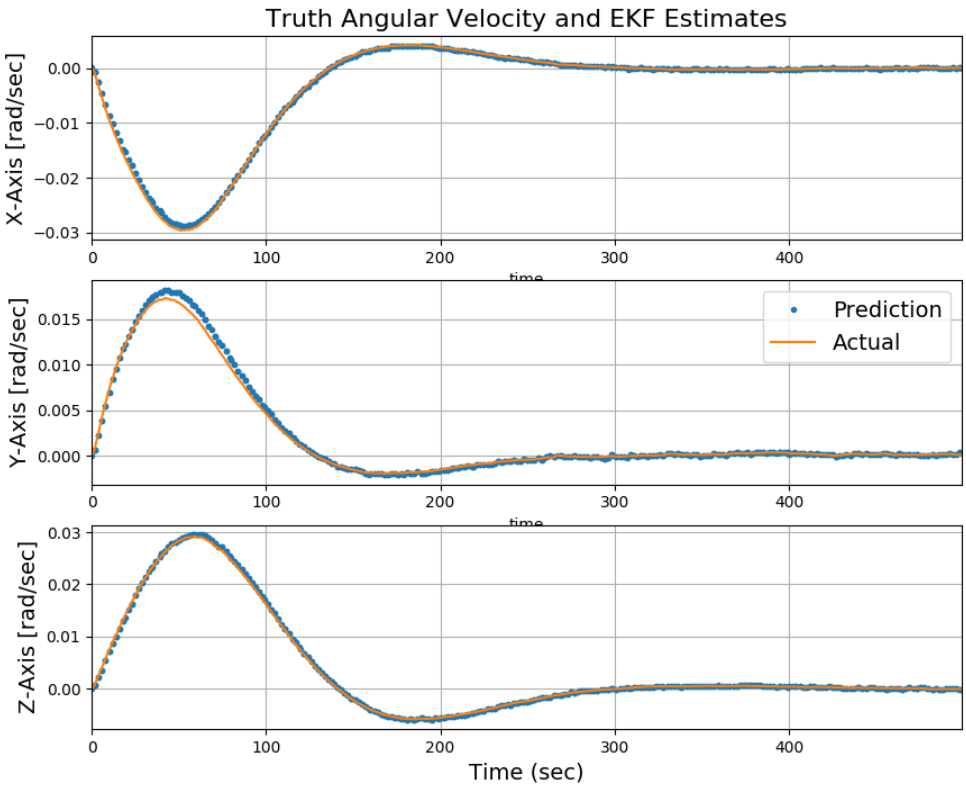

Figure 46: Software-in-the-loop Rest to Rest Body Rate Response

feedback path, and external disturbances. However, a more rigorous exploration of the simulation parameter space should ultimately be conducted in the future.

5.3.2 Nadir Pointing

In the software-in-the-loop simulation, nadir pointing is subject to a few additional sources of pointing error relative to the three-axis Simulink tool. Since the flight software does not have perfect orbit knowledge, there are inherent errors in the reference quaternion that can only be mitigated with higher accuracy orbit determination. Currently, the CPCL flight computer simply propagates its most recent TLE up to the current time of the on-board clock to obtain position and velocity knowledge. NASA 42 does support GPS sensor modeling, but that was not investigated as part of this work and will be elaborated upon in the future work section. Figures 47 and 48 show the closed loop attitude and rate responses, respectively. In this simulation, the steady state pointing error in the eigenaxis/angle sense was about **4.8°**. The steady-state per-axis pointing errors are provided in table 7

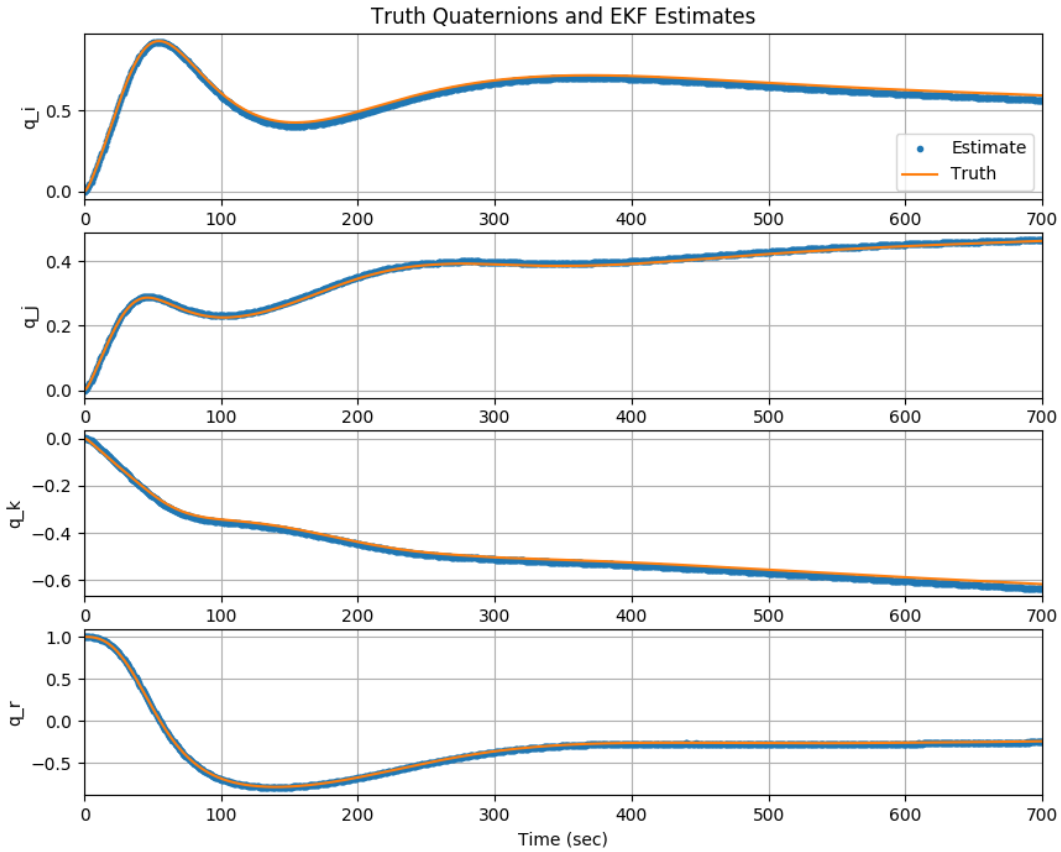

Figure 47: Software-in-the-loop Nadir Pointing Attitude Response

Table 7: Steady State Nadir Pointing Performance in SITL Simulation

Body Axis	Steady State Attitude Error (deg)
X	3.3
Y	2.5
Z	2.4

Figures 47 and 48 show that the spacecraft is successfully maneuvered into a nadir pointing attitude and continues to maintain nadir pointing into steady state. Pointing error in this mode is noticeably worse than inertial pointing. The reason for this is that inaccuracies in orbit position and velocity knowledge are now showing up as errors in the reference quaternion. From a simplified trigonometric analysis, the nadir pointing error induced by a relatively small in-track orbit determination error in a circular orbit is given by

$$\theta_e = \tan^{-1}\left(\frac{\Delta r}{R_e + h}\right)$$

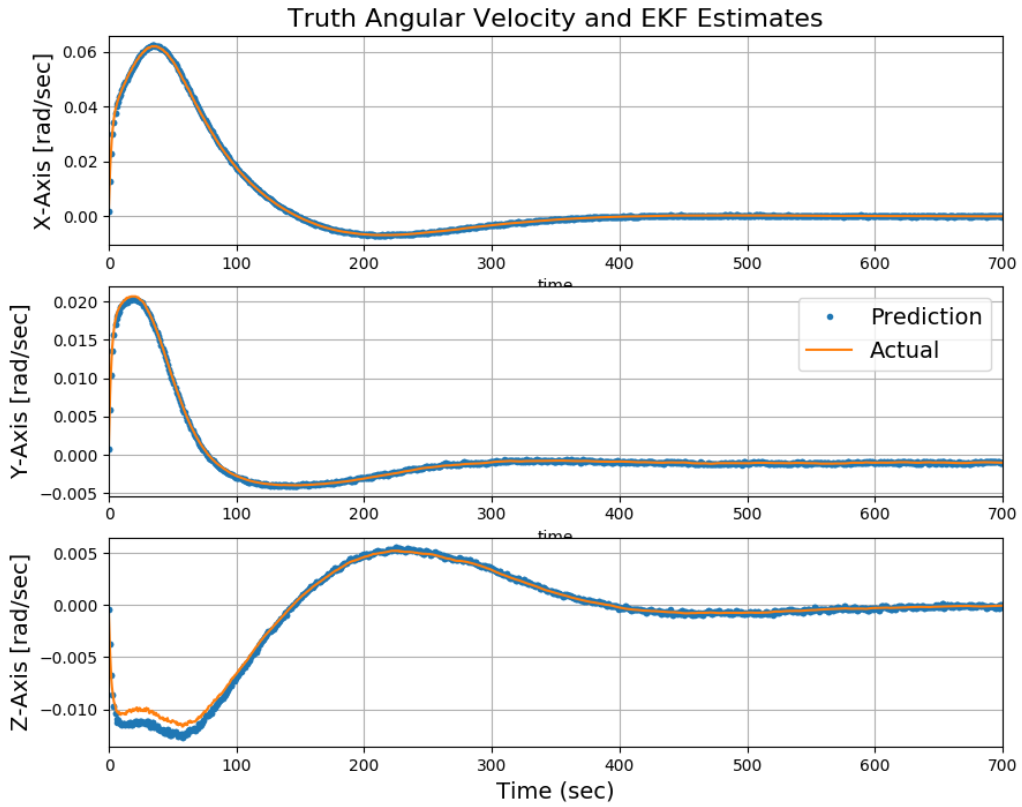

Figure 48: Software-in-the-loop Nadir Pointing Body Rate Response

Where Δr is the magnitude of the in-track orbit knowledge error, R_e is the radius of the Earth, and h is the circular orbit altitude. Simulation results indicate that the worst case in track orbit error between the spacecraft estimate and truth is about 15 km. With an in track orbit error of 15 km at an altitude of 400 km, the resulting attitude error is approximately 0.13˚. Although not very sizable relative to a pointing requirement on the order of a few degrees, this is an absolute lower-bound on pointing errors induced by orbit knowledge errors. As a whole, the attitude error covariance is related to the overall orbit error covariance in a multidimensional sense, and only one of those dimensions is considered in this analysis: in-track uncertainty. Uncertainties in the cross-track and nadir positions and velocities can also show up as contributors to the attitude pointing error. Therefore, to ascertain the sensitivity of pointing error to orbit state uncertainty, significantly more analysis must be performed. In future work, this sensitivity should be investigated in the context of the software-in-the-loop simulation framework, perhaps utilizing GPS sensor models in NASA 42 and a flight software orbit determination algorithm instead of the more simple TLE propagation.

5.3.3 Ground Pointing

Ground pointing is similar to nadir pointing in the sense that attitude pointing error can be induced by error in position and velocity knowledge. However, ground pointing also suffers from error in knowledge of the ECEF to ECI transformation, which can also be thought of as uncertainty in the inertial position of the ground location of interest. Therefore it is expected that the pointing performance in this mode will be no better than nadir pointing performance with all else being equal. Figures 49 and 50 show the attitude and rate response from a software-in-the-loop simulation in ground pointing mode. The positive spacecraft z-axis is commanded to align itself with a vector from the spacecraft to the position on the Earth with latitude 35˚ N and longitude 120˚ W. To verify that these plots are in fact consistent with the spacecraft z-axis pointing toward that location, a computation on the simulation output was performed to compute the angle between the target vector and the spacecraft Z axis in steady state. Table 8 below provides this angle, along with the usual X, Y, and Z attitude errors as computed from the quaternion product. However, the pointing errors in this mode should be interpreted slightly differently. This simulation framework is not quite at the level of maturity where the exact vector from the spacecraft to a specific ground location is accessible to the user. As such, the pointing errors in table 8 are relative to the spacecraft's reference quaternion, not the true reference quaternion. As a crude rule of thumb, about one degree could be added to each axis to account for this additional uncertainty.

Table 8: Steady State Ground Pointing Performance in SITL Simulation

Body Axis	Steady State Attitude Error (deg)
Angle between Pointing Axis and Target Vector	3.5
X	3.4
Y	1.0
Z	1.8

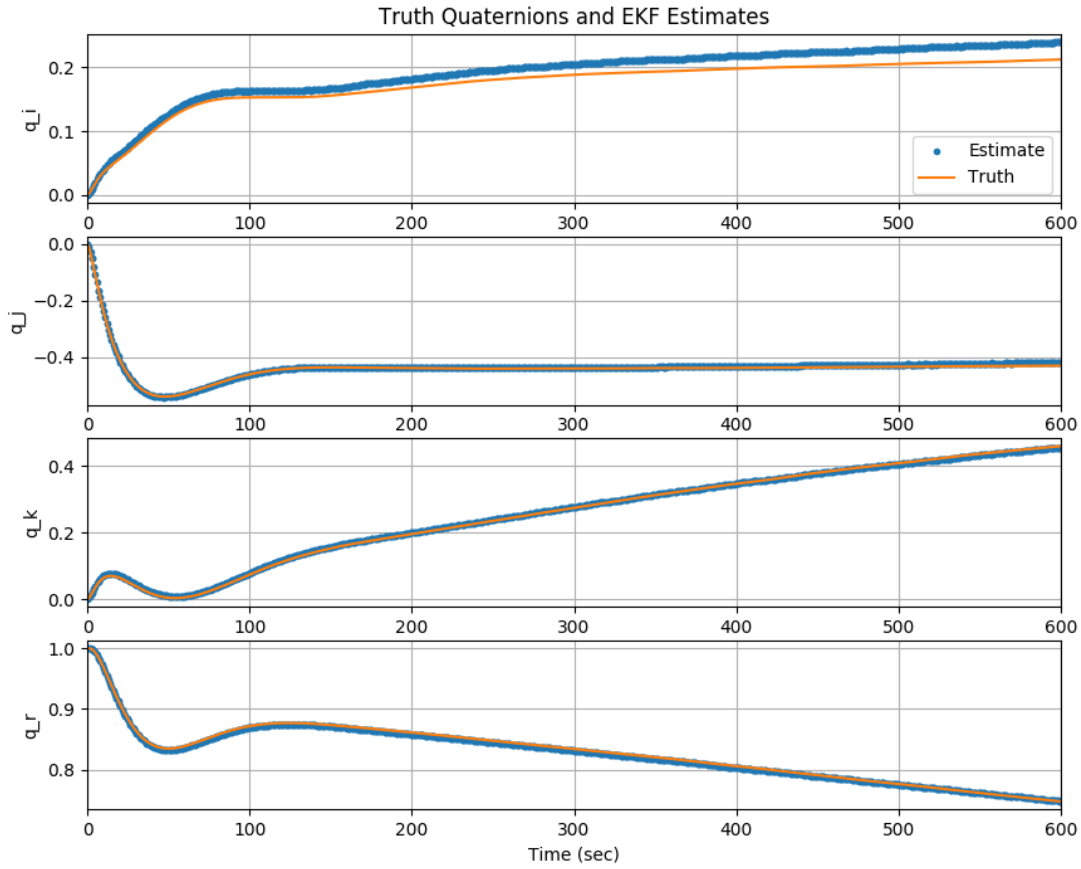

Figure 49: Software-in-the-loop Ground Pointing Attitude Response

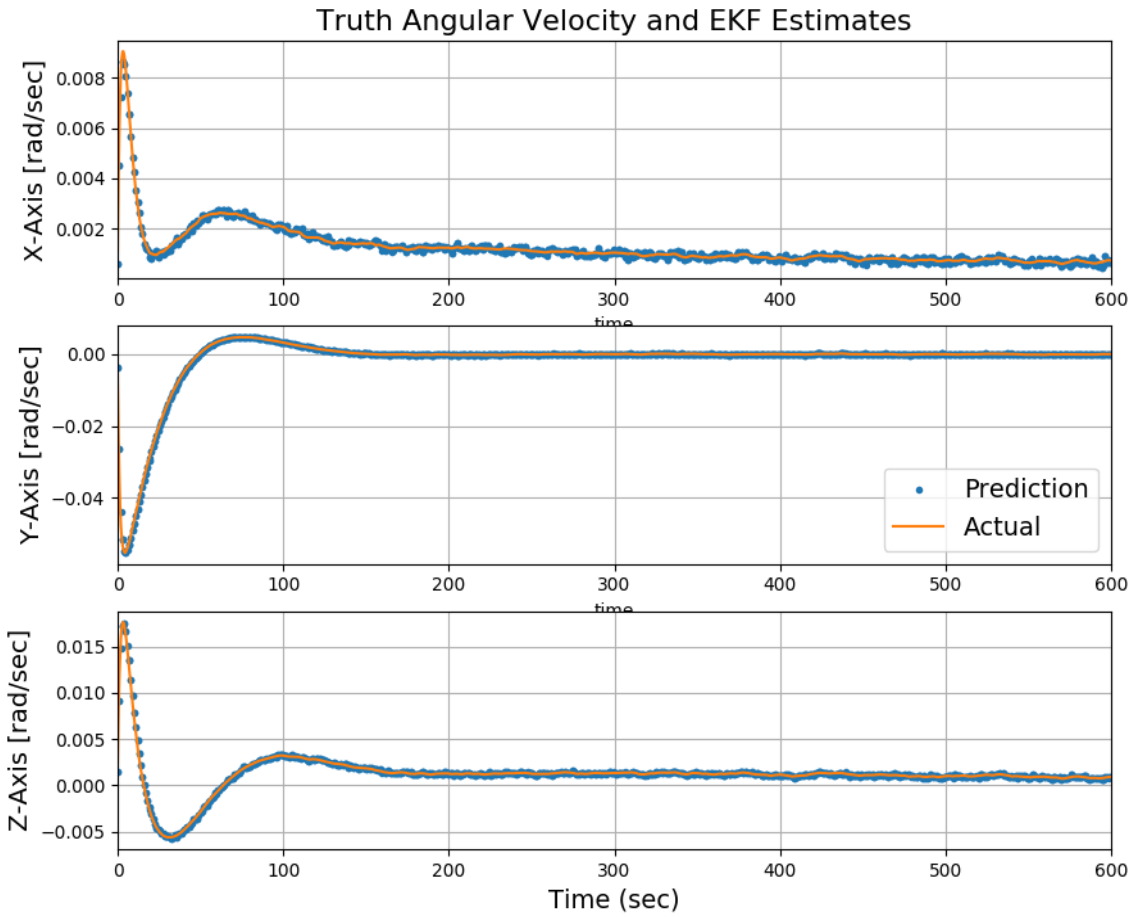

Figure 50: Software-in-the-loop Ground Pointing Body Rate Response

5.3.4 Spin Pointing

Spin pointing is the final pointing mode to be analyzed within the context of the software-in-the-loop simulation framework in this thesis. In this mode, there is no source of pointing error in determining the reference signal. Figures 51 and 52 show the rate and quaternion responses, respectively. The chosen body fixed pointing axis is the +z axis, which is commanded to point along the inertial +x direction with a spin rate of 1 degree per second. Notice that in figure 51, the angular velocities around the x and y axes converge to near zero, while the z body rate converges to the commanded spin rate. Table 9 provides the relevant steady state pointing errors in this simulation. The pointing error around the commanded body axis is not relevant. However, the accuracy of the commanded spin rate around the commanded body axis is an alternative

means of characterizing the control system's performance in this mode. The angle between the inertial x direction and the body z axis in steady state for this simulation is about **2.9˚.**

Table 9: Steady State Spin Pointing Performance in SITL Simulation

Body Axis	Steady State Error
X Attitude	1.1˚
Y Attitude	2.7˚
X Rate	0.0004˚/sec
Y Rate	0.0002˚/sec
Z Rate	0.003˚/sec

As seen in table 9, the per axis steady state pointing errors in the spin pointing mode are all less than 3 degrees. Moreover, the angular rate accuracy is extremely good. The control system brings the angular rate around the commanded axis to within three one-thousandths of a degree per second of the commanded rate, and nulls the rate around the other two axes to within four ten-thousandths of a degree per second.

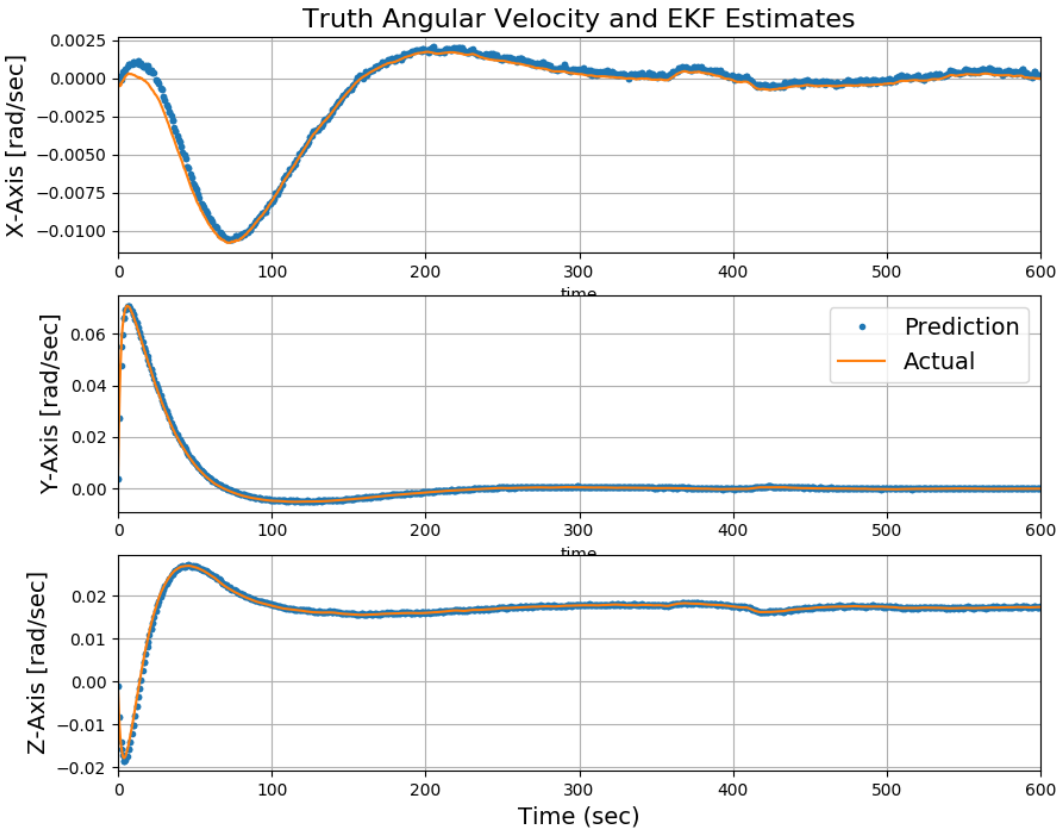

Figure 51: Software-in-the-loop Spin Pointing Rate Response

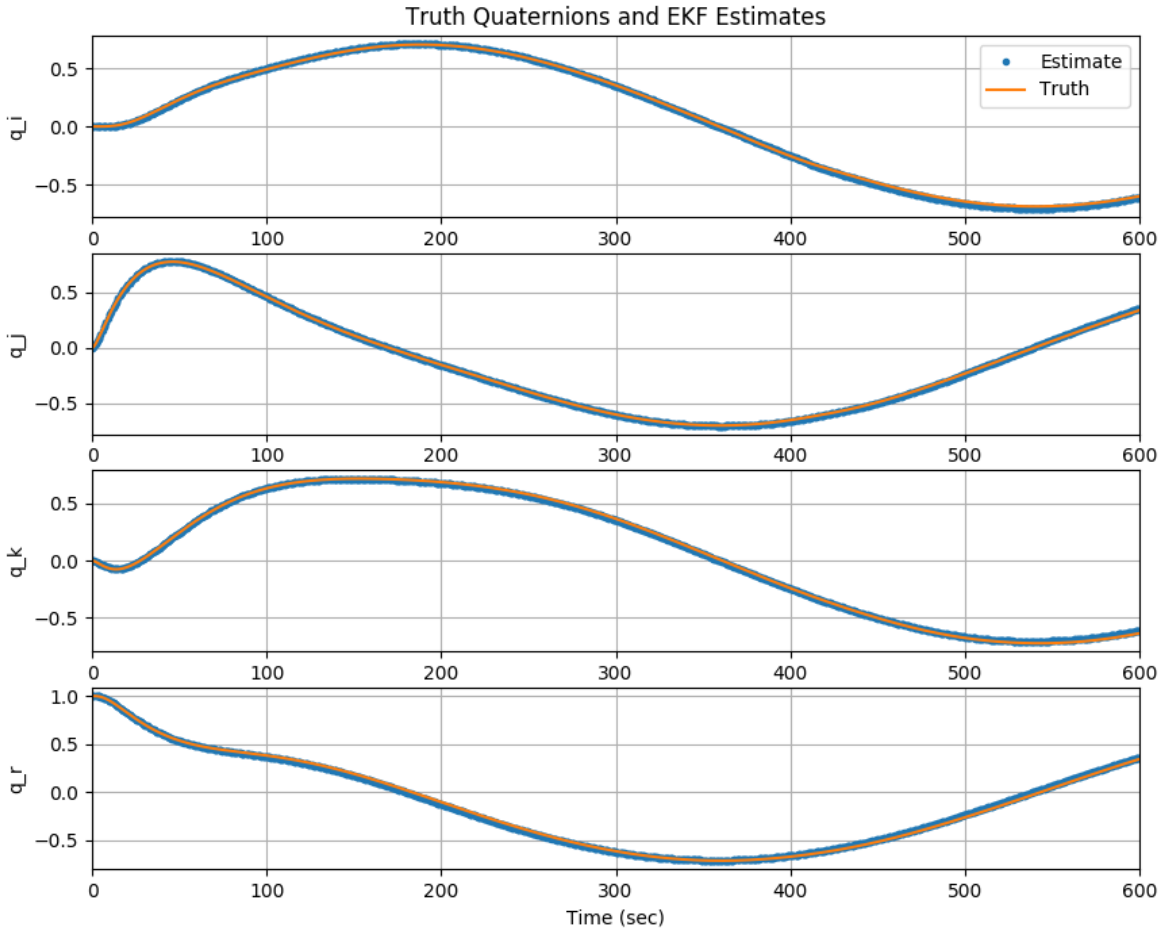

Figure 52: Software-in-the-loop Spin Pointing Attitude Response

Chapter 6

CONCLUSIONS AND FUTURE WORK

This thesis presented the results of an ADCS design and analysis toolkit enhancement effort whose ultimate goal was to solidify the engineering infrastructure required to develop an in-house ADCS for the CPCL. Particular emphasis was placed on the design of attitude guidance and control algorithms to supplement previous effort dedicated to navigation algorithm design performed primarily by Mehrparvar [10] and implemented in flight software by Bouchard [8], completing the overall spacecraft GNC framework. Special attention was also given to actuator sizing and selection since much of the previous work on estimation algorithms was obviously more concerned with sensor performance. Reasonably good performance of a completed, notional CPCL ADCS design employing commercial magnetometers, in-house sun sensors, and commercial rate gyros, was demonstrated in the framework of the enhanced toolkit with the single axis Simulink tool, the three axis Simulink tool, and finally the flight software-in-the-loop simulation employing NASA 42 as the truth model. Along the way, the preliminary specifications of a reaction wheel in development by Nick Bonafede were considered. The results in each of these tools showed reasonably good consistency and similarity, providing some level of independent validation. A preliminary analysis of the pointing performance of the completed CPCL ADCS design in multiple control modes was performed in the software-in-the-loop environment with Python implementations of the control algorithms. In this environment, a multitude of pointing error sources such as sensor and actuator dynamics and misalignment, estimator dynamics, environmental perturbations, digital signal quantization, inertia matrix uncertainty, and orbit knowledge uncertainty were included in the analysis. In each of the modes, pointing performance on the order of a few degrees was observed, providing a preliminary validation that the completed CPCL ADCS design is competitive for use in a large class of CubeSat missions based on figure 1. It is important to note that the largest contributor to pointing error in general is attitude knowledge error, which in turn is largely a function of the quality of the attitude sensing hardware. With the notional design presented in this thesis, the attitude sensor suite does not include a star tracker. Most commercial ADCS designs from figure 2 include a star

tracker, which is likely the primary means by they achieve sub-degree pointing accuracy. Therefore, it is actually not surprising that the pointing performance of the notional ADCS design presented in this work—which does not include a star tracker—was on the order of a few degrees.

In the relatively long history of the CPCL, this thesis is the first successful attempt at performing a rigorous analysis of the *entire* CPCL ADCS functionality and performance within a flight software-in-the-loop environment. Furthermore, the guidance and control algorithm designs presented in this thesis are novel with respect to the entire history of CPCL ADCS development. Therefore, this work has significantly advanced the design maturity of the CPCL ADCS using an enhanced toolkit, moving it much closer to flight demonstration maturity. Nevertheless, significant future work must be performed before flight success should be expected from the current CPCL ADCS design. Furthermore, it is possible that an entirely different design may be required to fit within the context of unique future mission requirements. The enhanced toolkit presented in this work could be used to create such a design.

6.1 Future Work

As mentioned multiple times in this thesis, an abundance of opportunities for future work exist within the context of CPCL ADCS development. In some cases, the future work is relatively simple and quick, but in other cases would likely require an entire master's thesis or senior project. In the following subsections, future work areas previously mentioned in this thesis are elaborated upon in more detail.

6.1.1 Alternatives to Integrated Hardware-in-the-Loop Testing

In this thesis, models of the attitude determination and control system hardware are employed in simulation to capture their effect on the system level functionality and performance. However, all models contain inherent error, and cannot exactly reproduce the behavior they attempt to capture. Therefore, use of the real ADCS hardware to validate the system level functionality and performance is often accomplished within a full hardware-in-the-loop test in which the entire operational environment of the system is replicated. While it may be tempting to

conclude that such a test is the best way to validate system level performance and functionality, it is usually too complicated, costly, time consuming, and inefficient to justify. Furthermore, it is not always necessary with the right alternative verification and validation items.

The difficulty of system-level hardware-in-the-loop testing that cannot be avoided lays in reproducing a flight like environment so that sensor output signals look like they would on orbit. For magnetometers, this requires a functional Helmholtz cage capable of generating magnetic field vector profiles that are consistent with the spacecraft orbit. For sun sensors, a dark room with a light source to simulate the sun would be required. Furthermore, physical simulation of the unconstrained rigid body dynamics requires an air bearing table. This test becomes extremely complicated very fast, so it is critical that the benefits it provides be heavily considered against the cost and schedule impacts caused by it. As mentioned in [16], “fancier integrated tests” can afford descoping in the context of GNC verification and validation for small spacecraft missions, as long as hardware component performance testing is used to update models used in a software-in-the-loop simulation environment.

The reason that hardware-in-the-loop system level testing may be advantageous over software-in-the-loop only simulation is because it does not suffer from nearly as much modeling error, and it captures lower-level functional interactions like electrical interfaces between the central flight computer and the physical sensors and actuators. Furthermore, sign errors that exist in the flight software GNC algorithms which may not appear in a software-in-the-loop simulation are easily detectable in a hardware-in-the-loop test. Sensor and actuator phasing and timing is also captured in a hardware-in-the-loop test. While these are all very good reasons to plan for hardware-in-the-loop testing on a CubeSat ADCS, there are alternative, less complicated means by which the same set of requirements can be verified. Namely, the functionality (polarity, timing, electrical interfaces, etc.) of the individual sensor and actuator components can be tested individually. Furthermore, the performance of each of the sensors and actuators can also be thoroughly characterized during testing. Based on these test results, the models used in NASA 42 can be modified and updated to capture the behavior of the sensors more accurately. More specifically, component-level functionality can be verified independently from the rest of the

system, and simulation models of the hardware behavior and performance can be validated and updated, also through component level testing. If all of these activities are completed successfully, a system level hardware-in-the-loop test becomes unnecessary.

6.1.2 Monte Carlo Analysis

In this thesis, single point simulations were executed for each pointing mode to predict the steady-state system level pointing performance. Although acceptable as a first pass, this is nowhere near as thorough as it should be when applied to a specific mission. Furthermore, depending on the nature of the mission-specific pointing requirement, it could be simply impossible to verify such a pointing requirement with one simulation. For example, if the ensemble interpretation is used as in [17], then by definition one simulation cannot do the job. To illustrate why this is the case, [17] defines the pointing error behavior ε as being a function of both time and the set of parameters $\{A\}$ that could affect it:

$$\varepsilon = f(t, \{A\})$$

Where $\{A\}$ includes the set of all parameters, such as the spacecraft configuration, sensor and actuator layout, orbit geometry, time of day and year, etc. that could conceivably affect the pointing behavior. Now suppose that a pointing requirement levied on the ADCS requires that

$$P(|\varepsilon| < 1^{\circ}) \geq 90\%$$

which specifies that the probability that the absolute value of the pointing error is less than 1° be at least 90%. First of all, the pointing error itself must be well-defined: is it the angle between two vectors? The eigenaxis angle? Per axis angles? Once that is known, an interpretation of the probability must be provided. In other words, the 90% probability could be interpreted as 90% of the time for 100% of $\{A\}$, or it could be interpreted as 90% of $\{A\}$ for 100% of the time. The first interpretation is the *temporal* interpretation and the second is the *ensemble* interpretation. In the case that the ensemble interpretation is used, it is quite literally impossible to verify the requirement without a monte carlo campaign. In order to obtain knowledge that the pointing error is less than 1° in absolute value for 90% of the ensemble, it is necessary to thoroughly explore the

parameter space defined by $\{A\}$ in simulation, which is precisely what a monte carlo campaign does.

Even if a Monte Carlo may not be strictly necessary to verify a requirement, it is still of great utility when there is a known probability distribution associated with some or all of the parameters in $\{A\}$. In the context of this thesis, $\{A\}$ could be considered to be all of the parameters in the Yaml files defining the inputs to NASA 42. A monte carlo program could define probability distributions for some or all of the parameters in $\{A\}$, draw from those distributions and set them as the inputs, run a simulation, and repeat until the simulation outputs are sufficiently bounded. This type of analysis leverages the automated nature of simulations to thoroughly explore a parameter space and obtain a good prediction of how the system would perform on orbit in the face of parameter uncertainty.

6.1.3 Geodetic Pointing

As mentioned in section 3.5.2 of this thesis, geocentric pointing is implemented in the nadir pointing (and ground pointing) modes. This is subtly different than geodetic pointing, in which the non-spherical geometry of the Earth is taken into account to improve pointing accuracy. In some cases, geocentric pointing may actually be preferred, but in the case that a spacecraft boresight must be pointing toward certain features of the Earth, geodetic pointing is more accurate. For example, consider a case in which the ground pointing mode is employed to point a spacecraft boresight at the San Luis Obispo region in an orbit with a different inclination than the latitude of San Luis Obispo. In this case, the spacecraft essentially assumes that the Earth is a perfect sphere with a radius of 6378 km when it computes the vector to the commanded ground location. However, because the distance from the center of the Earth to the San Luis Obispo is likely not exactly 6378 km, there is a small target vector error in the inertial radial direction as a result of assuming that the Earth's radius is uniform across the surface. This small target vector error ends up as pointing error, and in some cases the resulting pointing error may be intolerably large. Nadir pointing suffers from similar error sources if geodetic pointing is required. To recast the formulation of the reference quaternion for geodetic pointing, all that is required is the Earth's radius as a function of latitude and longitude:

$$R_e = f(\lambda, \theta)$$

This function can be implemented as a look-up table in flight software so that during computation of the target vector in ground pointing mode or computation of the nadir vector in nadir pointing mode, the true earth radius at the latitude and longitude of interest is utilized. This strategy could improve pointing accuracy by as much as a few degrees.

6.1.4 Three-Axis Deadband Logic

In section 4.1, attitude error deadband logic was presented in the context of the single axis Simulink tool. This logic is useful in preventing undesirable steady state limit cycle behavior such as actuator chattering. In this work, it was implemented to prevent the speed command to the reaction wheel speed controller from discontinuously switching between zero and the minimum speed bit. This behavior can result in high frequency momentum exchange between the spacecraft and the reaction wheel, which may excite unmodeled, higher frequency flexible modes or compromise pointing performance. The deadband logic was shown to successfully suppress this limit cycle behavior, but only in the single axis tool. Implementation of the deadband logic in the three axis tool as well as the software-in-the-loop simulation framework has yet to be accomplished.

6.1.5 Estimation Algorithm Improvements

Previous estimator algorithm development in the context of CPCL ADCS was dedicated to the design of a discrete time extended Kalman filter to estimate the total spacecraft angular rate as well as the inertial to body attitude quaternion. Although this approach worked well in simulations presented in this work, it does not come without issues. Following along with [14], there are multiple fundamental issues associated with the use of a traditional extended Kalman filter in the estimation of an attitude quaternion and angular rate vector. First of all, due to the non-minimal nature of the quaternion representation of the attitude, the state covariance matrix is prone to being near singular. Second, the update step in the traditional extended Kalman filter can cause issues by violating the quaternion's unit norm constraint. To see this more clearly, the

update step in the extended Kalman filter corresponding to the quaternion portion of the state vector is given by

$$\hat{\boldsymbol{q}}^{+} = \hat{\boldsymbol{q}}^{-} + K[\boldsymbol{y} - \boldsymbol{h}(\hat{\boldsymbol{x}}^{-})]$$

Where $\hat{\boldsymbol{q}}^{+}$ is the a posteriori estimate of the attitude quaternion, $\hat{\boldsymbol{q}}^{-}$ is the a priori estimate of the attitude quaternion, K is the Kalman gain matrix, $\boldsymbol{y}$ is the measurement vector, $\boldsymbol{h}$ is the nonlinear measurement function, and $\hat{\boldsymbol{x}}^{-}$ is the a priori state estimate. Clearly, in order to satisfy the quaternion norm constraint across update steps, some special relationship between the a priori quaternion estimate and the product of the Kalman gain with the measurement residual must hold. According to [14], such a relationship does not exist in the general case.

Furthermore, it is shown in [14] that an unbiased estimate of the four-component quaternion must violate the unit norm constraint. If an unbiased estimator is developed to avoid this issue, a new issue arises, namely that the quaternion covariance matrix becomes ill-conditioned in the limit as the estimation errors go to zero. Ill-conditioned matrices are always undesirable in the context of numerical algorithms that run on digital computers, because any computation involving their inverse is prone to large numerical errors.

To avoid these issues, brute force normalization of the a posteriori attitude quaternion estimate is implemented in flight software by Bouchard [8]. However, such an approach can cause issues if the product of the Kalman gain and the measurement residual is not orthogonal to the a priori quaternion estimate as described in [14].

A more suitable estimation algorithm for use in spacecraft attitude determination and control systems which avoids all of the aforementioned issues is a multiplicative extended Kalman filter (MEKF) designed to estimate the *error-state* as opposed to the *global state* estimated by the extended Kalman filter. This formulation also provides advantages over the extended Kalman filter in that it naturally estimates a rate gyro bias instead of a total angular velocity. Since the rate gyro bias drifts slowly over time, and the extended Kalman filter does not estimate it, very poor estimation performance can be observed over long time scales when using an extended Kalman filter. Furthermore, estimation of a total angular velocity treats rate gyro data as measurements, and uses Euler's equations as the dynamic model in the predict step. If Euler's

equations are used in the predict step, then sufficiently accurate knowledge of the external torque and mass properties of the spacecraft are required. However, this is not always the case, especially when using cold gas thrusters to impart attitude control torques. An alternative approach as outlined in [14] is to use calibrated rate gyro measurements in the dynamic model of the estimation algorithm instead of treating it as part of the measurement vector. Then the expectation of the true angular rate in terms of the sensed angular rate as given by the following relationship:

$$\boldsymbol{\omega}(t) = \boldsymbol{\omega}^{true}(t) + \boldsymbol{\beta}^{true}(t) + \boldsymbol{\eta}_v(t)$$

is used to propagate the global attitude quaternion estimate forward in time until the next measurement is taken. When the next measurement is taken, the error state estimate is updated to a finite value, and subsequently moved into the global sate estimate. Once moved into the global state estimate, the error state quantities are reset to zero.

In summary, the MEKF involves three steps: a measurement update, an error state reset, and a propagation to the next measurement time. Between measurements, the propagation moves the global state estimates forward in time to be used by the attitude controller. Once a measurement is taken, the error sates are updated, pushed into the global state, and rest to zero. This process repeats indefinitely in real-time.

The design and implementation of an MEKF for use in the CPCL is highly recommended by the author of this thesis, and will likely require either an entire master's thesis or collaboration between multiple dedicated student engineers.

Finally, another estimation algorithm issue associated with the simultaneous use of magnetometers and sun sensors first observed by Bouchard [8] is that when the two vector measurements are nearly colinear, the estimator covariance rapidly increases and the filter becomes prone to divergence. The reason this happens it that if the two measurement vectors were exactly colinear, there would be an infinite set of attitudes corresponding to rotations around that line which give rise to the same set of measurements. This non-uniqueness is reflected mathematically in a near singular measurement sensitivity matrix. To avoid this issue, the estimation and control algorithms could remain in an inactive state in anticipation of attitudes and

locations for which these two vector measurements are nearly colinear. This would require careful mission analysis and simulation to verify that it will mitigate the potential issues.

REFERENCES

[1] Shimmin, R. (Ed.). (2015). *Small Spacecraft Technology State of the Art*. NASA Ames Research Center, Mission Design Division.

[2] Messmann, D., Gruebler, T., & Coelho, F. et al (2017). *Advances in the Development of the Attitude Determination and Control System of the CubeSat Move-II*. European Conference for Aeronautics and Aerospace Sciences.

[3] Vega, K., Auslander, D., & Pankow, D. (2009). Design and Modeling of an Active Attitude Control System for CubeSat Class Satellites. *AIAA Modeling and Simulation Technologies Conference*. doi: 10.2514/6.2009-5812

[4] Jensen, K. F., & Vinther, K. F. (2010). *Attitude Determination and Control System for AAUSAT3*. Master's Thesis. Aalborg University.

[5] Philip, A (2008). *Attitude Sensing, Actuation, and Control of the BRITE and CanX-4&5 Satellites*. Master's Thesis. University of Toronto.

[6] Xiang, T., Meng, T., Wang, H., Han, K., & Jin, Z.-H. (2012). Design and on-orbit Performance of the Attitude Determination and Control System for the ZDPS-1A Pico-satellite. *Acta Astronautica*, *77*, 182–196. doi: 10.1016/j.actaastro.2012.03.023

[7] Li, J., Post, M., Wright, T., & Lee, R. (2013). Design of Attitude Control Systems for CubeSat-Class Nanosatellite. *Journal of Control Science and Engineering*, *2013*, 1–15. doi: 10.1155/2013/657182

[8] Bouchard, L (2019). *Reusable and Testable Attitude Determination Software for CubeSats based on Low-Cost Sensors*. Senior Project. California Polytechnic State University, San Luis Obispo.

[9] Sellers, R. (2013). *A Gravity Gradient, Momentum-Biased Attitude Control System for a CubeSat.* Master's Thesis. California Polytechnic State University, San Luis Obispo.

[10] Mehrparvar, A (2013). *Attitude Estimation for a Gravity Gradient Momentum Biased Nanosatellite*. Master's Thesis. California Polytechnic State University, San Luis Obispo.

[11] Bowen, J (2009). *On-Board Orbit Determination and 3-Axis Attitude Determination for Picosatellite Applications.* Master's Thesis. California Polytechnic State University, San Luis Obispo

[12] J., D. R. A. H., Damaren, C., & Forbes, J. R. (2013). *Spacecraft Dynamics and Control: an Introduction*. Chichester, West Sussex, United Kingdom: Wiley.

[13] PHILLIPS, C. L., & NAGLE, H. T. (1998). *Digital Control System Analysis and Design*. New Jersey: Pearson Education.

[14] MARKLEY, F. L. A. N. D. I. S. (2016). *Fundamentals of Spacecraft Attitude Determination and Control*. New York: SPRINGER.

[15] Forbes, J. R., & Damaren, C. J. (2010). Geometric Approach to Spacecraft Attitude Control Using Magnetic and Mechanical Actuation. *Journal of Guidance, Control, and Dynamics*, *33*(2), 590–595. doi: 10.2514/1.46441

[16] Pong, C. M., Sternberg, D. C., and Chen, G. T. (2019). *Adaptations of Guidance, Navigation, and Control Verification and Validation Philosophies for Small Spacecraft*. Guidance and Control 2019, Advances in the Astronautical Sciences, Breckenridge, CO, February 2019.

[17] European Cooperation for Space Standardization. (2008). *Space Engineering Control Performance* (Standard No. E-ST-60-10C).

[18] Jan, Y., & Chiou, J. (2005). Attitude control system for ROCSAT-3 microsatellite: a conceptual design. *Acta Astronautica*, *56*(4), 439–452. doi: 10.1016/j.actaastro.2004.05.066

Appendix A: Example Yaml File Snippets

```yaml
---
Actuators:
  Reaction Wheels:
    Wheel 0:
      Name: wheelX
      Spin Axis: [1.0, 0.0, 0.0]
      Initial Momentum: 0.0
      MaxT: 0.0015
      MaxH: 0.0058
      Rotor Inertia: 9.42e-7
      Static Imbalance: 0.003
      Dynamic Imbalance: 0.003
      Body Frame Location: [0.0, 0.0, 0.07]
      Transverse Alignment Error: 0.1
      Longitudinal Alignment Error: 0.2
    Wheel 1:
      Name: wheelY
      Spin Axis: [0.0, 1.0, 0.0]
      Initial Momentum: 0.0
      MaxT: 0.0015
      MaxH: 0.0058
      Rotor Inertia: 9.42e-7
      Static Imbalance: 0.003
      Dynamic Imbalance: 0.003
      Body Frame Location: [0.0, 0.0, 0.07]
      Transverse Alignment Error: 0.1
      Longitudinal Alignment Error: -0.2
    Wheel 2:
      Name: wheelZ
      Spin Axis: [0.0, 0.0, 1.0]
      Initial Momentum: 0.0
      MaxT: 0.0015
      MaxH: 0.0058
      Rotor Inertia: 9.42e-7
      Static Imbalance: 0.003
      Dynamic Imbalance: 0.003
      Body Frame Location: [0.0, 0.0, 0.07]
      Transverse Alignment Error: 0.1
      Longitudinal Alignment Error: -0.1
```

```yaml
---
Simulation Control:
  Sim_Duration: 5500
  Step_Size: 0.1
  Year: 2020
  Month: 2
  Day: 19
  Hour: 9
  Minute: 23
  Second: 0
Model:
  Aero_F: TRUE
  Aero_T: TRUE
  Grav_Grad: TRUE
  SRP_F: TRUE
  SRP_T: TRUE
  Grav_Pert: TRUE
  RWA_dist: TRUE
```

Appendix B: Example Python Control Algorithm Implementation

```python
def spin_cruise_step(args):

    global controller

    q = np.array(args['quat_body_eci'])

    # Compute attitude error
    controller.updateErrorSignal(q)

    # Filter attitude error
    controller.derivativeFilter()

    # Integrate the attitude error and saturate
    controller.integrateError()
    controller.saturateIntegralState()

    # Calculate actuator commands
    controller.computeControlSignal()

    # Extract control signal
    M_cmd = controller.getControlSignal()

    # Reset previous filter states to current states
    controller.resetStates()

    return { 'step_time_ms': controller.control_rate_ms,
             'run_determination': True,
             'wheel_moment': list(M_cmd)
    }
#spin_cruise_step
```

```python
class SpinCruiseController:

    def __init__(self, Ts, w_gc, PM, w_cd, SAT_I, inertia_matrix, body_axis, inertial_vector, spin_rate):

        self.spin_rate = (np.pi/180)*spin_rate # [rad/s]
        self.control_rate_ms = Ts*1000
        self.SAT_I = SAT_I
        self.ref_quat = np.array([0, 0, 0, 1])
        self.ref_quat_prev = np.array([0, 0, 0, 1])

        Ix = inertia_matrix[0][0]
        Iy = inertia_matrix[1][1]
        Iz = inertia_matrix[2][2]

        # Gain design
        Kp_x, Ki_x, Kd_x = loopShapePID(w_gc, PM, Ix, Ts)
        Kp_y, Ki_y, Kd_y = loopShapePID(w_gc, PM, Iy, Ts)
        Kp_z, Ki_z, Kd_z = loopShapePID(w_gc, PM, Iz, Ts)

        self.Kp = np.diag([Kp_x, Kp_y, Kp_z])
        self.Kd = np.diag([Kd_x, Kd_y, Kd_z])
        self.Ki = np.diag([Ki_x, Ki_y, Ki_z])

        a, b = getFilterCoeffs(Ts, w_cd)
        self.derivative_a = a
        self.derivative_b = b

        # Controller and filter states:
        self.integral_state = np.zeros(3) # Integral of the attitude error up to current control cycle
        self.control_signal = np.zeros(3) # Control signal at current control cycle

        self.att_err = np.zeros(3) # Attitude error at current control cycle
        self.att_err_prev = np.zeros(3) # Attitude error at previous control cycle
```

```
    def computeControlSignal(self):
        self.control_signal = self.Kp @ self.att_err + self.Kd @ (self.att_err_filt - self.att_err_filt_prev) + self.integral_state
    #computeControlSignal

    def saturateIntegralState(self):
        for k in range(len(self.integral_state)):
            if self.integral_state[k] < -self.SAT_I:
                self.integral_state[k] = -self.SAT_I
            elif self.integral_state[k] > self.SAT_I:
                self.integral_state[k] = self.SAT_I
            #elif
        #for
    #saturateIntegralState

    def derivativeFilter(self):
        self.att_err_filt = (self.derivative_a)*(self.att_err_filt_prev) + (self.derivative_b)*(self.att_err + self.att_err_prev)
    #derivativeFilter

    def integrateError(self):
        self.integral_state += self.Ki @ ((1/2)*(self.att_err + self.att_err_prev))
    #integrateError

    def getControlSignal(self):
        return self.control_signal
    #getControlSignal

    def updateErrorSignal(self, q):
        q_ref_pyquat = Quaternion(matrix = (self.getR2(self.theta) @ self.R1).T)
        self.ref_quat = np.array([q_ref_pyquat[1], q_ref_pyquat[2], q_ref_pyquat[3], q_ref_pyquat[0]])
        if np.dot(self.ref_quat[0:3], self.ref_quat_prev[0:3]) < 0:
            self.ref_quat *= -1
            q_ref_pyquat *= -1
        #if
        q_pyquat = Quaternion(q[3], q[0], q[1], q[2]) # [r, i, j, k]
        q_err_pyquat = (q_ref_pyquat.conjugate)*q_pyquat # [r, i, j, k]
        self.theta += self.spin_rate*self.control_rate_ms/1000
        self.att_err = 2*np.array([q_err_pyquat[1], q_err_pyquat[2], q_err_pyquat[3]])
    #updateErrorSignal
#SpinCruiseClass
```

```
        self.att_err_filt = np.zeros(3) # Filtered attitude error at current control cycle
        self.att_err_filt_prev = np.zeros(3) # Filtered attitude error at previous control cycl

        # Build R1, the rotation required to align body fixed vrctor with inertial vector
        # TODO: find a better solution here.
        if body_axis == 'z': # Body Z axis is pointing vector
            delta = np.arctan2(inertial_vector[1], inertial_vector[2])
            alpha = np.arctan2(inertial_vector[0], np.sqrt(inertial_vector[1]**2 + inertial_vector[2]**2))
            self.R1 = Cy(alpha) @ Cx(-delta)
            def getR2(theta):
                return Cz(theta)
            #getR2
            self.getR2 = getR2
        elif body_axis == 'y': # Body Y axis is pointing vector
            delta = np.arctan2(inertial_vector[0], inertial_vector[1])
            alpha = np.arctan2(inertial_vector[2], np.sqrt(inertial_vector[0]**2 + inertial_vector[1]**2))
            self.R1 = Cx(alpha) @ Cz(-delta)
            def getR2(theta):
                return Cy(theta)
            #getR2
            self.getR2 = getR2
        elif body_axis == 'x': # Body X axis is pointing vector
            delta = np.arctan2(inertial_vector[1], inertial_vector[0])
            alpha = np.arctan2(inertial_vector[2], np.sqrt(inertial_vector[0]**2 + inertial_vector[1]**2))
            self.R1 = Cy(-alpha) @ Cz(delta)
            def getR2(theta):
                return Cx(theta)
            #getR2
            self.getR2 = getR2
        #elif
        self.theta = 0 # Rotation angle around body fixed vector [rad]
    #__init__

    def resetStates(self):
        self.att_err_prev = self.att_err
        self.att_err_filt_prev = self.att_err_filt
        self.ref_quat_prev = self.ref_quat
    #resetStates
```