

HARDWARE-IN-THE-LOOP SIMULATION FOR VERIFICATION OF
CUBESAT ATTITUDE DETERMINATION AND CONTROL SUBSYSTEMS

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 Computer Science

by

Andrew Sorensen

March 2018

© 2018
Andrew Sorensen
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Hardware-in-the-Loop Simulation for Verification of CubeSat Attitude Determination and Control Subsystems

AUTHOR: Andrew Sorensen

DATE SUBMITTED: March 2018

COMMITTEE CHAIR: John Bellardo, Ph.D.
Associate Professor of Computer Science

COMMITTEE MEMBER: Hugh Smith, Ph.D.
Associate Professor of Computer Science

COMMITTEE MEMBER: Jordi Puig-Suari, Ph.D.
Professor of Aerospace Engineering

ABSTRACT

Hardware-in-the-Loop Simulation for Verification of CubeSat Attitude Determination and Control Subsystems

Andrew Sorensen

Since their creation in 1999, CubeSats have quickly erupted into a worldwide phenomenon. With missions growing in complexity, it is often required to keep the satellite in a certain orientation appropriate to that mission. To achieve this desired orientation, CubeSats must use an attitude determination and control subsystem, commonly known as an ADCS. Tasked with creating and maintaining an appropriate orientation for the satellite, the ADCS is a complex system, and often there are difficulties verifying the functionality of such algorithms.

This thesis investigates a hardware-in-the-loop simulation based off of PolySat's software architecture that allows for a higher granularity of ADCS testing than standard unit tests do. A space environment model is simulated, and virtual sensors and actuators are created to interface the simulation with an actual ADCS process from PolySat. This allows the ADCS to attempt to orient the satellite in an environment similar to its actual operating environment. A variety of tests are conducted on the system to verify the functionality and performance of this hardware-in-the-loop system. Results reveal that the apparatus is able to provide a suitable testing environment for a CubeSat ADCS algorithm utilizing flight hardware and software.

ACKNOWLEDGMENTS

Thanks to:

- My advisor, Dr. Bellardo, as well as the rest of my committee, for giving me this opportunity
- The various members of PolySat who assisted and provided input into the project, notably: Luc Bouchard, Chandler Gifford, Michael Salinas, Mazzin Ajamia, Liam Bruno, and Kent Rush
- All my friends and family who supported me throughout this endeavor, especially my parents, Alex Bartlett, Noah Weitz, and Marisa Diaz

TABLE OF CONTENTS

	Page
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER	
1 Introduction	1
1.1 Motivation	2
1.1.1 PolySat	2
1.1.2 ExoCube II	3
1.2 Thesis Scope	3
1.3 Thesis Overview	4
2 Background	5
2.1 Sensors	5
2.1.1 Magnetometers	5
2.1.2 Solar Angle Sensors	6
2.2 Actuators	6
2.2.1 Magnetorquers	7
2.2.2 Reaction Wheels	7
2.3 Frames of Reference	8
2.3.1 Earth Centered Inertial	8
2.3.2 Body Fixed	9
2.3.3 Local Vertical, Local Horizontal	10
2.4 Relevant Vocabulary	11
2.4.1 Quarternions	11
2.5 Attitude Determination and Control Subsystem	12
2.5.1 Attitude Determination	12
2.5.2 Attitude Control	13
3 Related Work	16
3.1 Matlab-In-The-Loop	16
3.2 Simulink-Based Hardware-in-the-Loop Simulation	17

3.3	Multiple Boards	18
3.4	NASA 42	18
4	System Requirements and Design	21
4.1	Requirements	21
4.2	Overview of PolySat Software	22
4.2.1	Libpolydrivers	22
4.2.2	Libproc	23
4.2.3	ADCS Process	24
4.3	Design	25
4.3.1	Virtual Time	26
4.3.2	Simulator	28
4.3.2.1	Dynamics Model	28
4.3.2.2	Sensor Reading Generation	29
4.3.3	Virtual Sensors	29
5	System Implementation	32
5.1	Sensor Framework Implementation	32
5.1.1	ZeroMQ	32
5.1.2	Google Protocol Buffers	32
5.1.3	Virtual Sensors	34
5.1.3.1	Magnetometers	36
5.1.3.2	Solar Angle Sensors	36
5.1.3.3	Magnetorquers	36
5.1.3.4	Reaction Wheel	37
5.2	Simulation Implementation	38
5.2.1	Simulink	38
5.2.1.1	Dynamics Model	39
5.2.1.2	Sensor Reading Generation	40
5.2.2	MATLAB	41
6	Evaluation and Discussion	43
6.1	Simulation	44
6.1.1	Discussion	44
6.2	Functional Verification	45

6.2.1	Evaluation	45
6.2.2	Results	47
6.3	Hardware-in-the-Loop Verification	48
6.3.1	Evaluation	48
6.3.2	Results	49
6.4	Performance Bottlenecking	50
6.4.1	Evaluation	50
6.4.2	Results	52
7	Future Work	54
7.1	NASA 42	54
8	Conclusion	55
	BIBLIOGRAPHY	57

LIST OF TABLES

Table		Page
6.1	Summary of Tests Conducted	48
6.2	Timing Results of each of the Simulations	52

LIST OF FIGURES

Figure		Page
2.1	Earth Centered Inertial Frame [20]	9
2.2	Body and Orbital Frames [20]	10
2.3	Flow of an ADCS Process	12
3.1	Tapsawat Simulator Architecture [25]	19
4.1	Libpolydrivers Sensor Hierarchy [20]	23
4.2	High Level Overview of the Hardware-in-the-Loop Simulation . . .	26
4.3	Flow Diagram of the Hardware-in-the-Loop Simulation	31
5.1	Example of a 'GET' Request Protocol Buffer	34
5.2	Flow of the 'SET'/'GET' Functions	35
5.3	Flow of the Dynamics Model	40

Chapter 1

INTRODUCTION

From their creation in 1999, CubeSats have quickly evolved into a worldwide phenomenon. A CubeSat can be described as a 10cm x 10cm x 10cm small satellite, usually weighing around 1.33 kg [11]. This is commonly referred to as a one-unit, or "1U" CubeSat, however larger 2U, 3U, and even 6U CubeSats are commonly used as well, with some even as large as 12U [11][15]. These small satellites provide space access to university programs as well as small firms and agencies [3]. While missions vary in scope in complexity, they often require the satellite to be in a certain orientation in orbit, usually to utilize a payload to its full potential. One such example is PolySat's ExoCube II mission, where the payload must be facing in the direction of the satellite's movement in order to measure particles within the exosphere. In order to achieve this desired orientation, the CubeSat must use an attitude determination and control subsystem, commonly known as ADCS. The attitude determination and control subsystem will be referred to as ADCS for the duration of this paper.

The purpose of the ADCS is to orient the spacecraft's frame relative to an external frame or body, such as the Earth [15]. As the name suggests, this is a two-stage process: first the attitude determination and then the attitude control. Attitude determination is the process of obtaining the satellite's current orientation relative to a frame of reference. This is usually done with on-board sensors to take measurements, as well as software that makes sense of those measurements [15]. Once the attitude has been determined, the control portion of the system comes into play. This is the process of actuating the spacecraft from its current orientation to its desired orientation. This is accomplished with actuators and control laws, which vary depending on the needs of the mission. The ADCS process will be discussed in greater detail in Chapter 2.

While ADCS testing has become a common procedure for large satellite makers, the process can still be very challenging for a CubeSat [16]. CubeSat developers usually lack the enormous testing capabilities of programs such as NASA, diminishing their abilities to thoroughly test the ADCS. The ADCS has traditionally been a difficult system to test due to the fact that it cannot be tested in its operating environment. As a result the space environment in which the CubeSat will be operating needs to be modeled accurately in order to properly test the ADCS. Sensors and actuators that the ADCS relies on often don't work or have limited capabilities on the surface [16]. For example, magnetometers, which measure magnetic fields, will not give the ADCS accurate readings to work with as the magnetic field strength on the Earth varies from that in the atmosphere. While equipment such as Helmholtz cages and air bearing test-beds allow for a more accurate environment to test in, these are often not practical for CubeSat developers. This thesis aims to provide a MATLAB and Simulink-based hardware-in-the-loop simulation that can provide accurate system-level testing of an ADCS that standard unit tests simply cannot provide.

1.1 Motivation

1.1.1 PolySat

PolySat is a student-run research lab on Cal Poly's campus which focuses on developing and launching CubeSats. PolySat uses multidisciplinary teams including but not limited to software, electrical, aerospace, and mechanical engineering to complete various missions. Missions are often a collaborative effort with programs such as NASA ELaNa, Goddard Space Flight Center, JPL, SRI, and many others [7]. One such mission currently in development is ExoCube II, which will be described in more detail in Section 1.1.2. This mission is highly dependent on a functioning ADCS because its particle measuring payload requires the satellite to be in the correct orientation

and as a result provides the motivation for this thesis. Because the project is being completed as a part of the PolySat program, they will be the customer to which the system is delivered to. While PolySat has developed ADCS algorithms in the past they currently lack the ability to provide a fine granularity of ADCS verification outside of normal unit tests, which alone are not sufficient for testing a system as complex as an ADCS.

1.1.2 ExoCube II

ExoCube II is one of the various missions currently in development at the PolySat lab. As mentioned earlier, the purpose of this mission is to measure the density of various particles in the atmosphere, including Hydrogen, Oxygen, Helium, and Nitrogen. This is accomplished through a payload supplied by the NASA Goddard Space Flight Center [7]. Accurate measurements from the payload are dependent on the CubeSat maintaining a proper orientation for the payload, which is done via the ADCS. Active satellite control is required to orient the payload in the RAM-facing direction (direction of the satellite's motion) and maintain that orientation once it is initially achieved. This is currently the second iteration of this mission at PolySat. When ExoCube I was launched back in 2015, confidence about the ADCS functionality was not high, as mainly unit tests were used to exercise the ADCS. For a process as complex as the ADCS unit tests are insufficient, and a hardware-in-the-loop simulation should be proficient to provide ADCS verification to ensure success of missions like ExoCube II and beyond.

1.2 Thesis Scope

The purpose of this thesis is to provide a hardware-in-the-loop environment in which an ADCS can be verified at a system level. A simulation in Simulink and MATLAB

is designed to provide the space environment and dynamics that the CubeSat will experience. Virtual sensors are created to interface with the simulation and the ADCS itself, passing data to the ADCS just as physical sensors would do. The ADCS makes decisions based on these simulated readings, at which point commands will be sent to the virtual actuators, just as though physical actuators were in use. The virtual actuators then deliver the ADCS commands back into the simulation, and simulation will propagate the satellite based on those commands. This allows the ADCS to operate in an environment close to the environment it will be interacting with in space, enabling testing of a finer granularity than of unit tests. Various testing has been conducting on the system, verifying that the simulation can interface properly with the ADCS and provide a hardware-in-the-loop testing methodology.

1.3 Thesis Overview

Chapter 2 explains background information relevant to the rest of the paper. This includes more information on the attitude determination and control process, different types of sensors and actuators used, and general information about the field that readers may find helpful. Chapter 3 discusses works that are similar to what is described in this thesis. Chapters 4 and 5 go over the design and implementation of the hardware-in-the-loop system, respectively. System evaluation as well as analysis of various test cases run can be located in Chapter 6, while a discussion of future work can be found in Chapter 7.

Chapter 2

BACKGROUND

The following section contains background information that the reader might find useful. Included is an explanation of various sensors and actuators utilized by the hardware-in-the-loop simulation, relevant information such as frames of reference and vocabulary, and finally a more in-depth look at the ADCS algorithm.

2.1 Sensors

Sensors are used to gain an understanding of the surrounding environment. When considering a spacecraft such as a CubeSat, these sensors are utilized to determine the position of the satellite. Examples of sensors often employed in satellites include gyroscopes, inertial measurement units, sun sensors, earth sensors, magnetometers, and star trackers. Only sensors employed in the PolySat ADCS algorithm will be examined in greater detail, notably magnetometers and solar angle sensors.

2.1.1 Magnetometers

A magnetometer measures the magnitude and direction of the local magnetic field in the form of a magnetic field vector. A common setup will usually include several magnetometers to paint a more accurate picture of the satellite's attitude. Magnetic field readings gathered by the magnetometers can be compared to on-board data about the Earth's magnetic field to determine the attitude of the CubeSat. A drawback with magnetometers is that measurements can often be corrupted with data from local magnetic fields created by the magnetic properties of the CubeSat's material or the on-board electronics [12]. As a result it is often common practice to mount

magnetometers on deployables so that they only measure the Earth’s magnetic field. One advantage they hold over other common sensors is that there are no field-of-view issues that sun sensors or earth sensors might have, making them more useful in situations where these other sensors cannot see their intended targets [12].

2.1.2 Solar Angle Sensors

Solar angle sensors are used to determine the orientation of the sun relative to the sensor. Solar angle sensors consist of a chamber with a small window at the top to allow sunlight inside the chamber, and a photoelectric material at the bottom, which induces current when hit with sunlight. The current induced in the photoelectric material is directly correlated with the angle at which the panel is struck with sunlight [27]. This angle, paired with the location on the panel at which the sun is striking, allow for the calculation of a vector from the spacecraft to the sun, based upon the sensor’s location on the spacecraft. Solar angle sensors are not without their drawbacks. They have a limited field-of-view, meaning the sensors cannot produce accurate measurements if the sun is not within this field-of-view. Expanding off this, these sensors become useless when the satellite is in an eclipsed state, due to the fact that no sunlight is reaching the spacecraft [27][15].

2.2 Actuators

Actuators provide movement to the satellite and control to the satellite’s attitude. Actuators come in many forms, including thrusters, solar sails, magnetorquers, reaction / momentum wheels, and many more. Different actuators provide different types of control, and which actuators are used depends on the mission parameters. As with the sensors in the previous section, only actuators used by the PolySat ADCS will be covered, those being magnetorquers and reaction wheels.

2.2.1 Magnetorquers

A magnetic torquer, sometimes referred to as a magnetorquer, is an actuator often used for CubeSat attitude control. A magnetorquer consists of a wire coil mounted on the face of a CubeSat. Current can be run through this wire coil to create a local magnetic field, which will create a torque when the local magnetic field interacts with the magnetic field of the earth. This behavior can be explained by the equation

$$\vec{T} = \vec{m} \times \vec{b}$$

where \vec{m} is the magnetic field vector from the torquer, \vec{b} is the Earth's magnetic field vector, and \vec{T} is the resulting torque. However, this equation highlights the fact that torque can only be generated about an axis perpendicular to the Earth's magnetic field vector[12]. While coarse attitude control is possible in a setup consisting of only magnetorquers, it is often practice to use reaction wheels in conjunction with magnetorquers to achieve a more fine-grained control [12]. Some advantages of magnetorquers include their low power consumption and simple design [15]. The main disadvantage of using a magnetorquer is that it is very difficult to achieve fine three-axis control, due to the problem previously mentioned. They also have the potential to corrupt magnetometer readings if there is current running through the coils while the magnetometers are attempting to take readings [12].

2.2.2 Reaction Wheels

A reaction wheel is another type of actuator commonly used for CubeSat attitude control. Much like magnetorquers, reaction wheels are used to control satellite attitude via torque. Rather than using wire coils, the reaction wheel is mounted within the spacecraft, spinning on a brush-less motor to achieve torque. The law of momentum conservation defines the wheel's behavior. For instance, when the wheel creates

a torque by spinning, an opposite torque of equal magnitude affects the spacecraft, meaning $T_{wheel} = -T_{CubeSat}$ [26]. Both the torque of the wheel and the torque of the CubeSat will be about the spin-axis of the wheel. This allows for fine control about whichever axis the wheel is mounted. Mounting three wheels about three different axes within the CubeSat would allow for full three-axis attitude control. An advantage that reaction wheels have over magnetorquers is that better attitude control can be achieved using only wheels. However as mentioned in the previous section, reaction wheels are usually augmented with magnetorquers or thrusters. External disturbance torques that are applied to the spacecraft can change the angular momentum to the wheel, which can be compensated for using the magnetorquers to "offload" the extra torque in a technique known as "momentum dumping" [12].

2.3 Frames of Reference

Many different frames of reference are used in the field of attitude control. It is often advantageous to describe an object's position or motion in one frame as opposed to another. This section contains a description of a few frames of reference that the reader may find useful, and an example of what each coordinate system is used for.

2.3.1 Earth Centered Inertial

The Earth Centered Inertial frame is a frame of reference centered around the Earth. This frame will be referred to as the ECI frame for the rest of the paper. The origin of this frame is the center of the Earth, with the Z axis directed towards the North Pole. The X axis is pointed at the vernal equinox (the point at which the equator passes through the center of the sun's disk), while the Y axis completes the Cartesian coordinate system [15]. This frame can be seen in Figure 2.1. The position of a spacecraft relative to the Earth is usually measured as a vector in this frame.

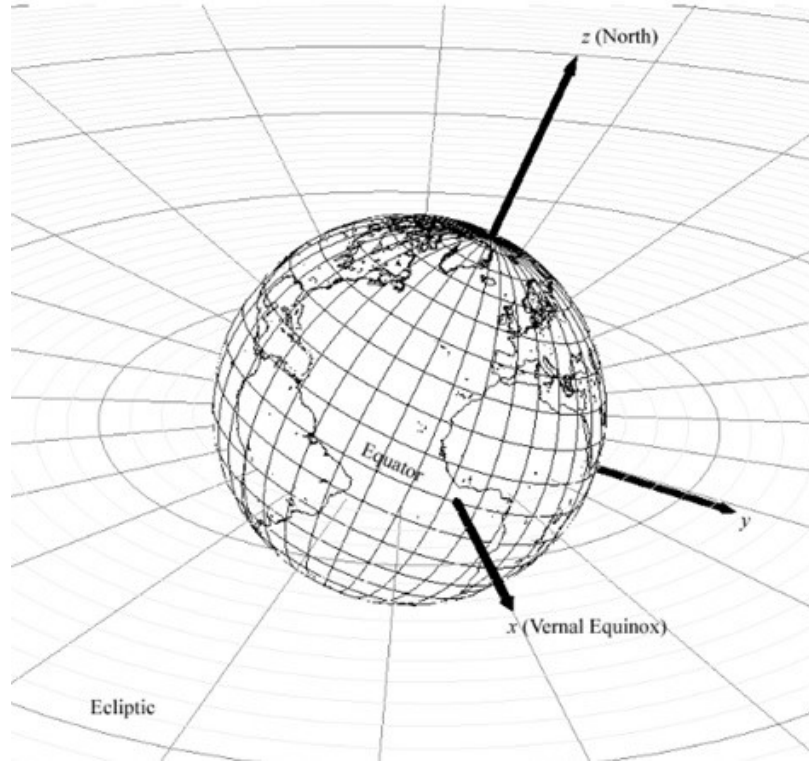


Figure 2.1: Earth Centered Inertial Frame [20]

2.3.2 Body Fixed

The body fixed frame is the frame geometrically aligned with the spacecraft. The origin of the frame is defined as the origin of the CubeSat. The X, Y, and Z axes are then defined to be normal to their respective side of the CubeSat. This can be as the red body frame in Figure 2.2. This frame can then be used to define pitch, roll, and yaw as rotations about these axes, with pitch being a rotation of the Y axis, roll being a rotation of the X axis, and yaw being a rotation about the Z axis. A vector in this frame describes the position of an object (such as the Earth or the sun) in relation to the spacecraft, or the rotations that the spacecraft is currently experiencing.

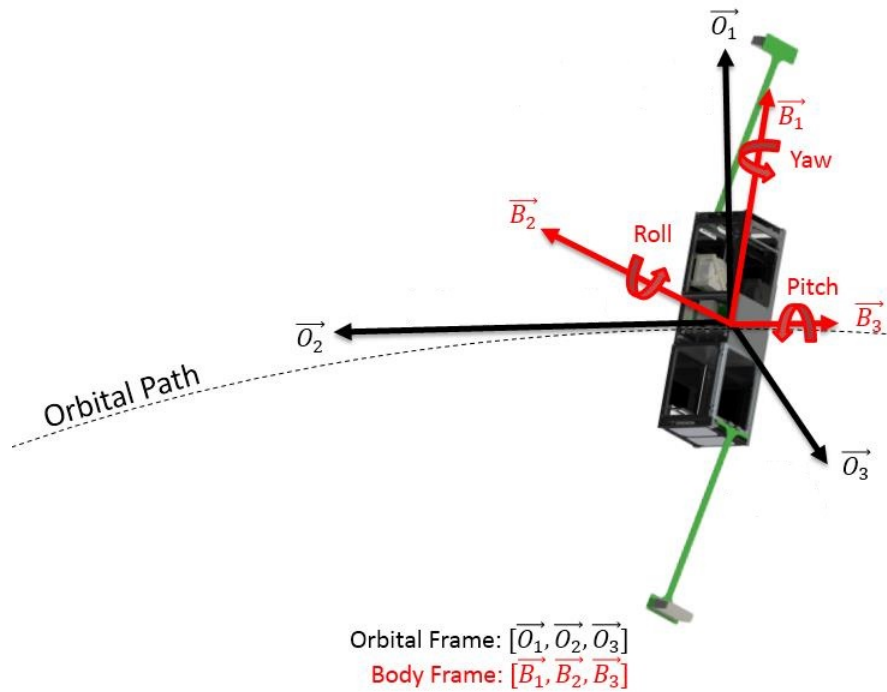


Figure 2.2: Body and Orbital Frames [20]

2.3.3 Local Vertical, Local Horizontal

The Local Vertical Local Horizontal frame, also known as the LVLH frame or even the orbital frame, is a more complex frame that rotates with the body of the spacecraft [19]. The origin of the coordinate system is the origin of the spacecraft. The X axis points in the radial direction (known as the 'local vertical'), the Z axis is normal to radial direction (local horizontal), and the Y direction completes the Cartesian coordinate system and is in the direction of travel. This can be more clearly denoted as the black orbital frame in Figure 2.2. It is often advantageous to describe a satellite's orientation in this frame, especially when the satellite is required to stabilize about a certain axis for mission purposes.

2.4 Relevant Vocabulary

2.4.1 Quaternions

Quaternions build off the complex number system [14]. A quaternion can be described as

$$a + bi + cj + dk$$

where a , b , c , and d are real numbers and i , j , and k are imaginary numbers. Similar to complex numbers, quaternions have the property such that $i^2 = j^2 = k^2 = ijk = -1$ [13]. Quaternions were discovered by an English researcher named William Hamilton, who was struggling to find a three-dimensional interpolation of the relationship between complex numbers and two-dimensional space [13][14].

Quaternions have many applications across multiple fields including graph and number theory, computer graphics, and physics, however one of their most attractive applications is being able to describe rotations in three-dimensional space [13]. This allows for describing the satellite's orientation and rotation within a particular frame of reference, or describing it in multiple frames of reference, such as the ones mentioned in Section 2.3. With a satellite's orientation described in vector X , and a quaternion that describes the satellite's rotation from frame X to frame Y , one can describe the orientation of the satellite in terms of frame Y . Because it can be advantageous to describe a satellite's orientation in one frame or another, quaternions are used throughout the simulation to both describe the satellite's orientation and transform between multiple frames such as the ECI frame, the LVLH frame, and the body frame.

2.5 Attitude Determination and Control Subsystem

As mentioned in Chapter 1, the ADCS is a complex process, whose purpose is to control the satellite's attitude. Both the determination and control aspects of this system will be explored in the next two sections. A flow of the ADCS process can be seen in Figure 2.3.

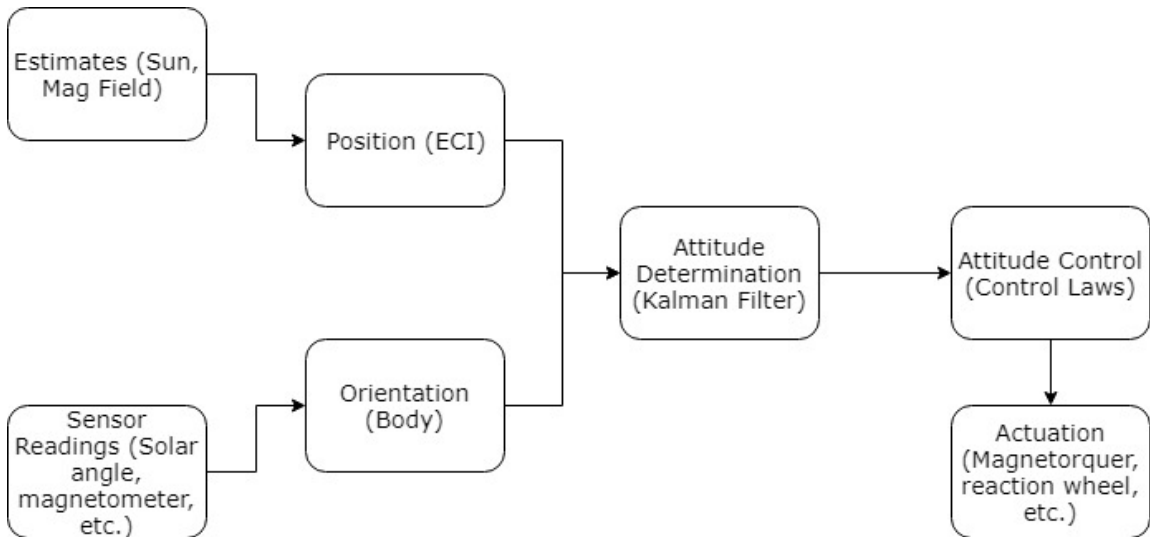


Figure 2.3: Flow of an ADCS Process

2.5.1 Attitude Determination

Attitude determination is the process of determining both where the satellite is and where it is going. An accurate picture of the satellite's attitude must be painted in order for the ADCS control laws to be utilized efficiently. In order to determine the satellite's attitude, its position relative to the earth (usually in the ECI frame) as well as its orientation (usually in the body frame) are needed. The satellite's orientation within the body frame is determined through various types of sensor readings. These sensors return magnetic and sun vectors in the body frame, which together can give a reasonable estimation of the satellite's current orientation. The satellite's position

in the ECI frame comes from computations by the ADCS. Based upon initial launch conditions and how much time elapsed since that launch, the ADCS attempts to extrapolate a position estimate. This estimate is used with an on-board model of the Earth's magnetic field to determine where the satellite is in relation to that magnetic field. This estimate is also used to calculate the position of the sun, and together with the magnetic field estimate, the position of the satellite in the ECI frame is gained [9][20]. The satellite's position in the ECI frame and its orientation in the body frame form the satellite's attitude.

However the task isn't complete once the position and orientation of the satellite are obtained. This information unfortunately doesn't divulge the rotation that the satellite is currently experiencing, knowledge that is critical to successful attitude control. A Kalman filter is used to both refine these attitude estimates and provide an estimate of the satellite's current rotation [9]. The Kalman filter, though a feedback loop, attempts to estimate the satellite's rotation through a series of measurements taken over time, in this case the measurements made by the sensors and ADCS software. Predicted attitude values, often coming from an on-board dynamics model of the satellite, are combined with the sensor measurements to obtain a more accurate value, with the weight of the prediction changing per iteration [9]. The estimated rotation, along with the satellite's current and desired attitude, are then utilized by an attitude control law to attempt to move the satellite into the desired orientation.

2.5.2 Attitude Control

Attitude control is the process of using the data from the attitude control and translating it into actuator commands. This is accomplished through what are known as control laws. There are several types of control laws. A few examples include the PID control law, the PD control law, and the B-Dot algorithm. Through various processes,

these algorithms are charged with calculating the proper torques that will move the satellite from its current attitude to the desired attitude. While the methods may be different, the outcome of these control laws are the same: torque commands are sent to the actuators to orient the satellite to a more desired attitude. A very brief overview of some of these control laws can be found below.

- **B-Dot** The B-Dot algorithm is mainly used to initially de-tumble the satellite when it is launched. This is accomplished by taking multiple magnetometer readings, and then providing a torque that is in the opposite direction of the measured change in magnetic field [20].
- **Proportional** The proportional control law is a very simple control law where the controller output is proportional to an error. The error is defined as the difference between the measured position of the satellite and its desired position [12]. This law can be written as

$$u(t) = K_p * e(t)$$

where $u(t)$ is the controller output, $e(t)$ is the error, and K_p is the proportional gain, a non-zero constant. This can be imagined as applying a reactionary torque proportional to the measured error. This can often result in un-dampened oscillatory motion when examined in terms of satellite attitude control, as the satellite can "overshoot" its desired attitude [12]. While simple, this control law can be developed further to get more complex control laws, as described below.

- **Proportional Derivative** The proportional derivative control law, also known as the "PD" control law, builds off the proportional law by adding a derivative term. The purpose of this derivative term is to provide dampening to the oscillatory motion of the proportional law, providing a more refined control

[12]. With this dampening, the satellite is less likely to overshoot its desired attitude.

- **Proportional Integral Derivative** The proportional integral derivative control law, or "PID" for short, builds off the PD law by adding an integral term. This integral term allows for the elimination of error that comes with constant disturbances, such as external torque on the spacecraft [12]. This is a very common control law and allows for even more refined control than the previous laws.

Chapter 3

RELATED WORK

This chapter will highlight some related work in the field of CubeSat ADCS verification. Mainly, it will focus on works that utilize a software-based approach to ADCS testing, as many methods that use hardware-based test beds to simulate an accurate environment already exist. This section will focus mainly on software-based methods that somewhat resemble what is being described in the remainder of this thesis.

3.1 Matlab-In-The-Loop

Rossi et al. utilized a strategy known as 'Matlab-In-the-Loop', or 'MIL', when verifying the ADCS design for the CubETH satellite [23]. This comprised of a closed loop simulation where many aspects of the ADCS were tested on an EFM32 evaluation board, while the environment was simulated in MATLAB, similar to the approach that will be discussed in this paper. The goal of this testing strategy was to verify that the ADCS C-code was executing the same algorithm originally designed in MATLAB [23]. As such, this was never intended to provide full functional verification to the ADCS. A full-fledged hardware-in-the-loop testing scheme was created for this which utilized various equipment such as air-bearing test beds and Helmholtz cages to simulate an environment that sensors and actuators could interact with.

A few issues were experienced with this testing method, however. The authors had trouble defining sensor inputs that the ADCS algorithm could properly utilize, as well as actuator commands that were fed back into the MATLAB environment. Additionally, precision with floating-point and double-precision numbers was lost when transmitted serially to the ADCS algorithm. Ultimately, this setup was never intended to

provide system-level verification, which contrasts the method to be described in this paper. Besides a Helmholtz cage, there is no reasonable way to test physical sensors with the ADCS at PolySat, so virtual sensors are used to emulate physical sensors that have either been flight-proven or will undergo separate testing.

3.2 Simulink-Based Hardware-in-the-Loop Simulation

Craig Clark and Dr. Simone Chesi presented on a MATLAB and Simulink based hardware-in-the-loop simulator at the 2015 Cal Poly CubeSat Workshop. The system utilizes a software-based environmental model to deliver sensor readings to a system board running a stand alone ADCS program written in C [10]. The simulation allows for the hardware to make real decisions based on sensor inputs from the model and transmit actuator data back into the simulation to propagate the satellite. All data transmitted to and from the board is through a serial connection. According to the authors, this system allows for easy ADCS software implementation and new control laws to be tested with ease [10].

This approach to CubeSat ADCS verification is closely inline with what this paper is trying to accomplish. While the system to be described in this paper is designed with PolySat software architecture in mind, the overall flow of the system is the same. Sensor and actuator data are communicated between an ADCS system and a simulation of a space environment, allowing the ADCS to make independent decisions based on the available data. The system in this thesis will hopefully bring with it some of the advantages previously mentioned in the presentation, such as quick and reliable ADCS development and debugging.

3.3 Multiple Boards

A process similar to that detailed in the previous section is explored by Tapsawat et al. While based on a numerical environmental model to deliver readings to and from the ADCS, more hardware is involved than in previously mentioned methods. The environmental model uses a small suite of MATLAB functions to simulate and propagate the satellite's orbit [25]. The model, however, only exports information such as angular velocity and magnetic field information. Rather than generate sensor readings from the simulation, this information is sent serially to four different boards, each one of them representing a sensor (in this case, two magnetometers and two rate-gyroscopes) [25]. These boards handle processing this information into sensor readings, then sending them to the ADCS in a method that mimics their sensors. The ADCS then carries out its task and calculates magnetic torquer commands. These commands are sent to three actual magnetorquers and the fields they create are measured by a magnetometer, which then feeds that information back into the numerical simulation [25]. This architecture can be seen in Figure 3.1. It is unclear how the additional overhead of all these additional boards and sensors affected the performance of the testing apparatus, however the authors state that this system was able to effectively test their ADCS's de-tumbling capabilities. The flow of this system is similar to that to be described in the paper, with the main difference that actual boards are being used to handle sensor calculations and transmission, as opposed to the virtual sensors used in this paper.

3.4 NASA 42

NASA 42 is a simulator designed to provide an environment for satellite attitude control design and verification. 42 is an open source software designed to be customized

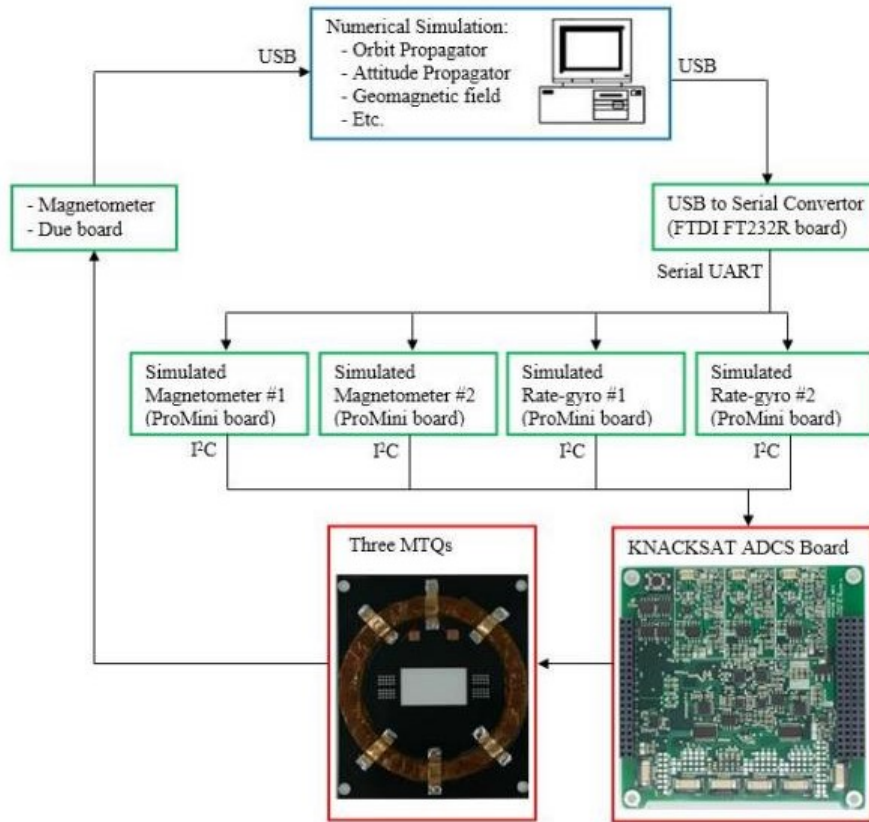


Figure 3.1: Tapsawat Simulator Architecture [25]

across many devices and platforms. It allows for testing across the entire development cycle, from prototyping to flight software verification [1]. A list of some of the provided features can be found below.

- Full modeling of spacecraft dynamics.
- Orbits comprised of more than one body.
- Modeling support for multiple spacecraft at once.
- Socket-based IPC API to communicate with outside applications.

One of the main benefits of using 42 to test ADCS algorithms is that the environment is already completed and is a trusted model, since it was developed and verified

by NASA. This ensures the quality and accuracy of any ADCS verification. A disadvantage however is that it is not plug-and-play, and the amount of modifications needed to integrate it with other systems depends on the architecture of the user system. 42 was researched about halfway into the development cycle of this thesis. While promising, it was ultimately unclear how much modification would be needed to make it work with the PolySat ADCS. Additionally, the simulation model was already developed at this time. 42 and its future role with PolySat ADCS software will be discussed more in Chapter 7.

Chapter 4

SYSTEM REQUIREMENTS AND DESIGN

This chapter will give an overview of the system from a design standpoint. This will include a brief description of some requirements for the simulation, an overview of some relevant PolySat software, and finally details about the hardware-in-the-loop simulation. Finer implementation details can be found in Chapter 5.

4.1 Requirements

While the purpose of the hardware-in-the-loop simulation is to provide system level testing capabilities for CubeSat ADCS algorithms, there are a couple of requirements that the system should adhere to. The first requirement is that the ADCS functionality should not be altered in any way. This means that the binaries that are being tested within the simulation should be the same binaries that are used on flight. The ADCS is a highly critical process imperative to mission success; a non-functioning ADCS results in the satellite not being able to stabilize itself for proper use of the payload it is carrying. As such, it is important that the ADCS process be tested in the state it will operate in. This will ensure the quality of all system tests conducted on the ADCS. While code had to be added to the ADCS to support a virtual clock, which will be discussed in Section 4.3.1, this option can be enabled or disabled via a configuration file, meaning that behavior isn't affected and the binary can still be used for flight.

The second requirement is that the simulation should be able to successfully deliver sensor readings and actuator commands to and from the ADCS. It was mentioned in Chapter 1 that the ADCS is a difficult system to test due to the the need of ad-

ditional equipment needed to provide a more realistic environment. The simulation ultimately cannot be considered a ground-truth model compared to a model developed and verified by NASA or similar organization. However, utilizing virtual sensors that can successfully transfer information, hypothetically any environmental could be used to exercise the ADCS. As such most of the testing conducted in Chapter 6 will focus on the functionality of these sensors rather than the simulation itself.

4.2 Overview of PolySat Software

Because the hardware-in-the-loop system is intended to PolySat missions, the hardware-in-the-loop simulation is designed with the PolySat software architecture in mind. While the simulation is written in MATLAB and Simulink and doesn't necessarily make use of the PolySat software, many supporting elements heavily utilize it, mainly the virtual sensors. As such it is important to understand how this software architecture works on a high level in order to effectively detail how these supporting elements come into play. PolySat software aspects to be discussed include Libpolydrivers, Libproc, and the ADCS core process itself.

4.2.1 Libpolydrivers

Libpolydrivers is PolySat's low level device interface API, which allows processes like the ADCS to easily interface with sensors and actuators. This ensures that process code does not need to constantly change in order to accommodate new sensors. Although implemented in C, Libpolydrivers follows a three-level object oriented design including sensors, classes, and drivers. Sensors are the top layer and are representative of base devices. This level includes functions that apply to all devices. The child of the sensor level is the class level. Classes determine what type of device is being used, such as a solar angle sensor, a power sensor, or a magnetorquer, for example. Classes

contain functions that are pertinent to all sensors of a certain type. This is also the level that the user interacts with, as it is not important for the user to know how details of specific devices work. Finally at the lowest level are drivers. Drivers are written per device and are unique to it. For example, if two different temperature sensors are being used, a separate driver needs to be created for each device in order to interface properly with the parent class. The relationship between all these levels can be seen in Figure 4.1.

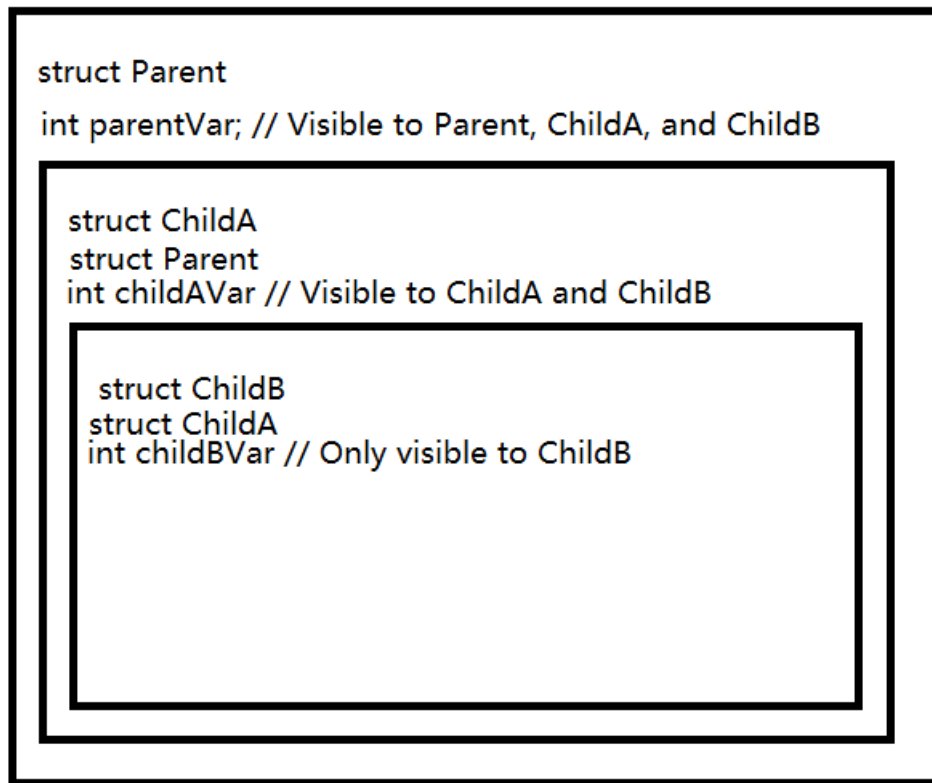


Figure 4.1: Libpolydrivers Sensor Hierarchy [20]

4.2.2 Libproc

Libproc is the PolySat library tasked for scheduling and managing processes and events. PolySat has a number of processes that run on the CubeSat, including the ADCS, system manager, watchdog, and data logger. Processes are often comprised

of events that need to occur at a specific time or regular interval. Libproc allows for an event loop within a process that handles the scheduling of various events that need to occur. The two types of commonly seen events are timed events and file-descriptor events. Timed events occur at a time in the future or an interval, while file-descriptor events occur if there is new data to be read from or written to a file descriptor. This allows for reading without blocking, without which would interfere with the timed event schedule. In addition Libproc supports inter-process communication through sockets. This is especially useful for a process like ADCS which often receives commands from the ground station. Libproc is vital to the PolySat software architecture and many missions and processes would not be possible without it.

The fact that Libproc is event-driven is highly critical to the success of the simulation (to be elaborated in Section 4.3.1). As such it is important to understand some of the design decisions that made Libproc event-driven. As described by Greg Manyak in [18], a satellite is very event-driven in nature, and thus lends itself well to using an event-handler rather than multi-threading. Most events experienced by the satellite are either time-driven or prompted [18]. Additionally, a multitude of different processes are run on the satellite, which rely heavily on sockets for inter-process communication. As a result, an event handler was created that was able to wait periods of time for timed events, as well as utilizes processor sockets to poll file descriptors in the form of *select()* calls [18].

4.2.3 ADCS Process

The ADCS process is tasked with executing all stages of the ADCS technique described in section 2.5. Through use of Libproc, a main command loop coordinates various events and function calls that together make the ADCS successful. This in-

cludes estimating position, reading various sensors through Libpolydrivers, computing exact position through these readings in conjunction with a Kalman filter, and calling a control algorithm to correct the satellite's attitude. While the ADCS can essentially be considered a black box in the hardware-in-the-loop simulation, understanding how the ADCS handles its inputs and outputs (sensor readings and actuator commands) is an important factor when creating an accurate simulation. For example, when determining the spacecraft's position relative to the sun, only the sensor with the best view of the sun should be used in the calculations. In addition the median magnetic sensor vector is used to decide with magnetometer to use. These are examples of sensor subtleties that need to be accounted for in the simulation.

4.3 Design

As previously mentioned, the aim of the hardware-in-the-loop simulation is to provide an environment to test the ADCS on a system level. This is accomplished with two components: the environmental simulation, which propagates the satellite in a simulation of the space environment, and the sensors to interface this dynamics model with the ADCS process. Very limited information is actually exchanged between the the ADCS process and the model. The model sends magnetometer and solar angle sensor readings to the ADCS, at which point it blocks until it receives information back from the ADCS in the form of magnetorquer commands and reaction wheel speed. Outside of various configuration files, sensor data and actuator commands are the only information the ADCS shall send and receive under normal operating conditions, thus in an attempt to create a realistic simulation additional information is limited as much as possible. However in order to get the simulation and the ADCS process to coordinate properly, the ADCS clock time needs to be communicated as well. This will be explained further in section 4.3.1. Additionally, there is metadata

included in the packets that are sent back and forth, however this information is not seen by the ADCS process nor the environmental model and is mainly used to parse readings on either side. This meta-data will be discussed more in section 5.1.2. A high level overview of the system can be seen in Figure 4.2. Sections 4.3.2 and 4.3.3 will go through the two components of the system in greater detail.

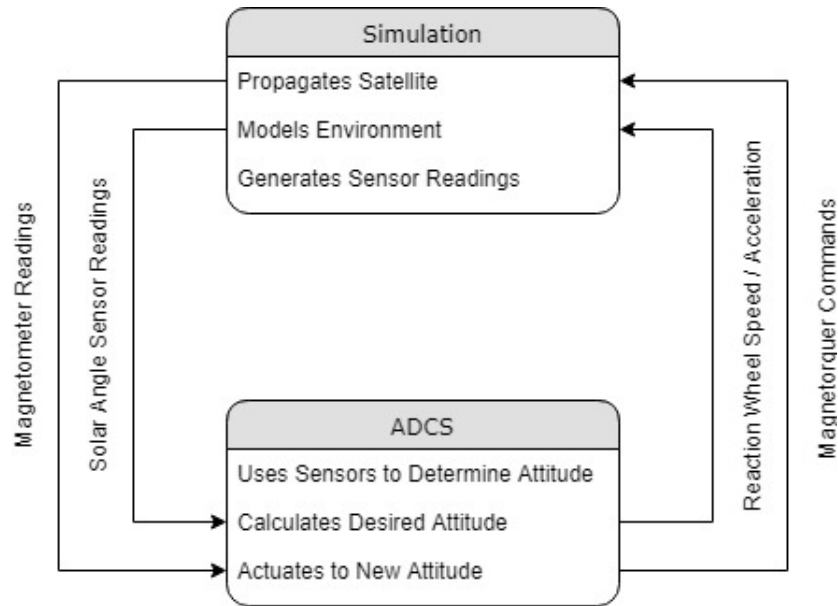


Figure 4.2: High Level Overview of the Hardware-in-the-Loop Simulation

4.3.1 Virtual Time

Ensuring that the ADCS and the simulation stay synced and communicate accurate information is difficult due to many reasons. The ADCS command loop operates at a regular interval (usually one second), and utilizes the system time for many of its calculations. Additionally, due to overhead caused by Simulink, a one second iteration of the simulation may or may not take one second. If sensors on the ADCS block to wait for readings from the simulation, by the time the information is received the system time will have advanced and the ADCS will associate those readings with a potentially wrong time. Running tests in "real time" would also be impractical, as the

CubeSat usually takes close to ninety minutes to complete an orbit around the sun, and multiple orbits are needed to properly look at trends. Thus a test that consisted of ten orbits would take hours to complete, which can be a huge bottleneck when using the hardware-in-the-loop simulation to diagnose problems with the ADCS. To combat all these potential issues, a "virtual time" mode was added to Libproc.

While Libproc normally executes timed events at their respective intervals, the virtual time mode augments Libproc and serves as a discrete event simulation mode. A discrete event simulator models a system as a discrete series of events. Events are associated with a scheduled time, and are executed sequentially based upon this time. Since there are no changes to the system between these events, the simulation can instantaneously jump to the next event, with the simulation time updated to the event's scheduled time [22]. The virtual time mode is emulating this functionality, attempting to execute the ADCS process instantaneously and updating the virtual clock to the ADCS's scheduled execution time. This prevents the clock from elapsing while the ADCS is blocking for sensor readings from the environmental model. The simulation and ADCS are initialized to the same time, and the simulation will only advance when it receives a time from the ADCS making a sensor request. So if, for example, the ADCS is at $t=5$ when a sensor request is made, the simulation will propagate to $t=5$ as well and return the appropriate sensor readings for that time step. Because the virtual clock doesn't increment until the process is complete, the virtual time, as understood by the ADCS, will still be at $t=5$ when the readings are received. When the ADCS completes an iteration, the virtual time will be updated to $t=6$ and the process will start over again. The run time of the simulation may vary, however the ADCS will always associate the correct readings with the correct time. The ADCS ends up driving time for the entire simulation, as the simulation will not propagate unless the time received from the ADCS is greater than its own. The flow of how virtual time works in conjunction with the rest of the hardware-in-the-loop

system can be seen in Figure 4.3.

4.3.2 Simulator

The simulator itself can be thought of as two components: the dynamics models, which propagates the satellite and models the environment that the satellite experiences, and the sensor reading generation, which characterizes this environment into sensor readings that can be passed through the virtual sensors to the ADCS. Design aspects of the two components will be explored further in the following two sections.

4.3.2.1 Dynamics Model

The dynamics model is responsible for propagating the satellite through space. It is also tasked with simulating the environment by calculating all the forces acting upon the satellite and changing its orientation appropriately. This includes forces such as drag, gravity gradient, torques from the earth's magnetic field, solar pressure, and forces resulting from the ADCS commanding the magnetorquers and reaction wheel. This allows the simulation to not only take into account the environment but the actions of the ADCS as well. Thus if the ADCS is functioning properly, the satellite will eventually stabilize, which will be reflected in the data logged by the simulation. Conversely, if the ADCS is not functioning properly then the simulation will reflect that as well. This ensures that the ADCS has the opportunity to make meaningful decisions based on the sensor readings it has gathered.

The main pieces of information the model keeps track of during this process are position and velocity vectors, the disturbance torques on the satellite, the satellite's angular velocity, and quaternions relating the satellite's body frame to the ECI and LVLH frames. Propagation of the position and velocity vectors occur first, after which the disturbance torques are calculated. The satellite's angular velocity resulting from

these disturbance torques are then calculated, which allows for the propagation of the quaternions that describe the satellite's orientation based on this angular velocity. Once this is complete, the position vector and quaternion are used to generate sensor readings for that specific instance in time.

4.3.2.2 Sensor Reading Generation

The second stage of the simulation is the generation of sensor readings for the ADCS. Using the satellite's position relative to the earth, a quaternion describing the satellite's orientation, and the UTC time (initialized to a specific date along with the ADCS), both magnetometer and solar angle sensor readings can be generated. Magnetometer readings come in the form of a magnetic field vector in the satellite's body frame, based on a model of the Earth's magnetic field. Solar angle sensor readings come in the form of a vector that describes the satellite's position relative to the sun, or nothing if the satellite is in eclipse (the earth is in between the satellite and the sun, so no sensors would have a view of the sun). Based on the satellite's orientation, the simulation will decide which sensors can actually see the sun and create the vector appropriately. A small amount of white noise is added to each reading to exercise the Kalman filter's ability to filter out this noise. At this point sensor readings are ready to be delivered to the ADCS.

4.3.3 Virtual Sensors

The sensors serve as the bridge between the simulation and the ADCS. In most cases the sensor calculations have been abstracted to the simulation. As a result, the sensors receive readings that are already usable by the ADCS and in most cases simply pass them along. This contrasts normal sensors, which have various methods of gathering readings and often need to translate them into a more usable state.

This design decision was made because this simulation is intended to test the ADCS and not the sensors, which have different methods of verification. This also makes the creation of these virtual sensors much simpler. The same thing is done with the virtual actuators as well, with the exception being the reaction wheel. Since the wheel needs to accelerate to its determined final speed, the amount of time that the wheel has been accelerating needs to be recorded so the speed can be updated by the driver appropriately. Specifics of how this is done will be covered in chapter 5.

The sensors communicate with the simulation via a request response model. The ADCS will request magnetometer and solar angle readings through the sensors in the form of a 'GET' request, at which point the sensors will block until they receive information. Additionally, the actuators will request to send information back to the simulation in the form of a 'SET' request. The simulation is blocking until it receives either type of request, meaning that the simulation is only executing the dynamic model and sensor generation when necessary. The flow of this can be seen in Figure 4.3.

As mentioned in Section 4.2.1, the sensors communicate to the ADCS through Libpolydrivers. All sensors and actuators are packed into one 'mega' sensor for simplicity. This new sensor is an example of multiple inheritance within Libpolydriver's object oriented design, as this sensor can interface with multiple class APIs such as the magnetometer and solar angle classes. This allows the resulting driver to implement the functionality of every single sensor and actuator utilized, as well as contain the individual sensor objects for the ADCS to interface with. This will be explored in greater depth in Chapter 5. When paired with a configuration file telling the ADCS to use these sensors, the ADCS can operate as if it were using normal sensors. For example, if the ADCS calls the `read()` function to read data from a magnetometer, the `read()` function from the virtual sensor will be called and return the correct information. Due to the modular nature of Libpolydrivers, the ADCS is able to utilize

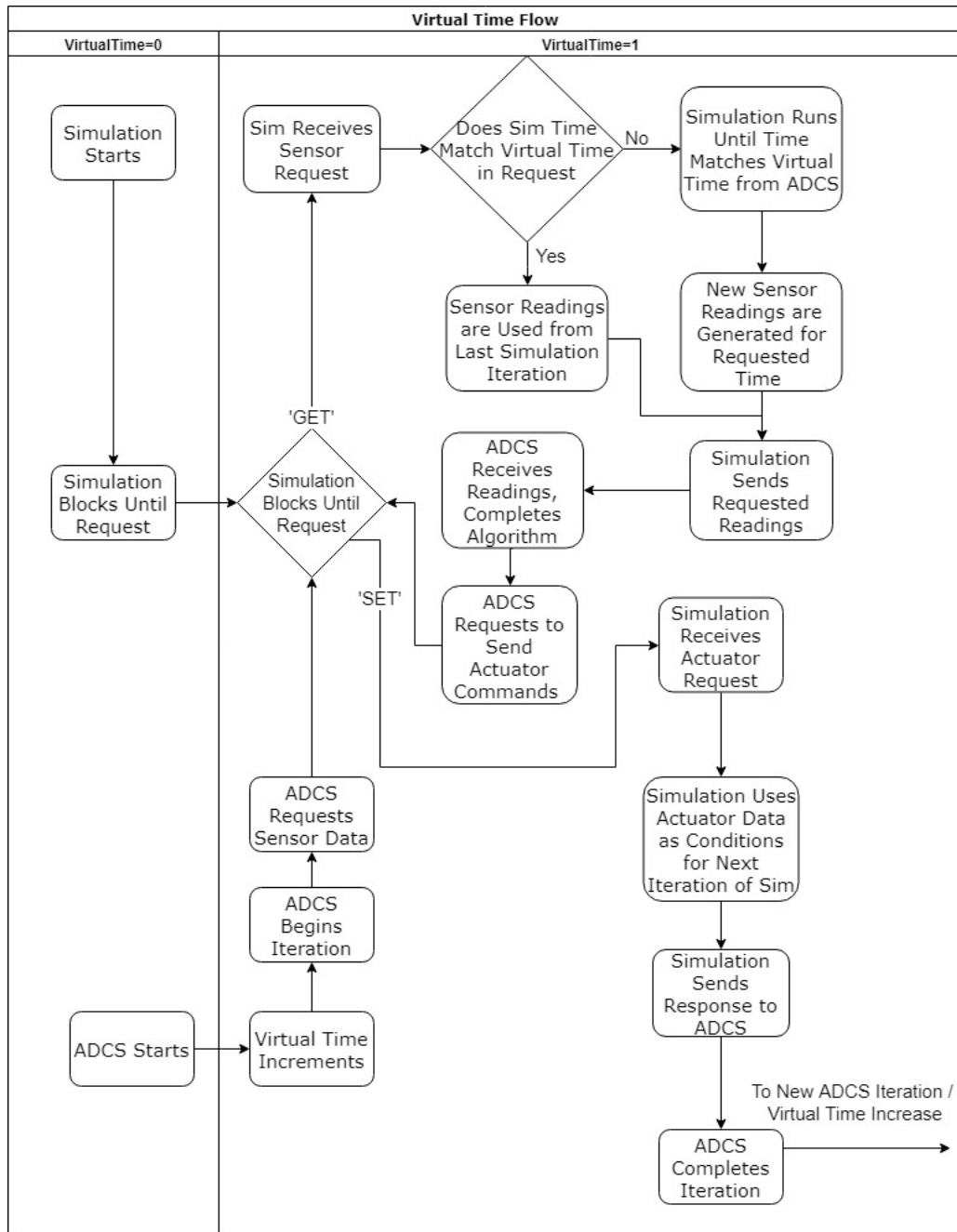


Figure 4.3: Flow Diagram of the Hardware-in-the-Loop Simulation

these virtual sensors without needing to be recompiled.

SYSTEM IMPLEMENTATION

5.1 Sensor Framework Implementation

5.1.1 ZeroMQ

ZeroMQ is an asynchronous messaging library and provides the medium through which the virtual sensors communicate with the simulation. This distributed messaging system allows for connecting code across multiple languages and platforms through the use of various communications protocols like TCP and UDP, all within a robust and high-speed API [8]. This allows for the C code of the virtual sensors to communicate with the MATLAB code (via Java) of the simulation. ZeroMQ utilizes sockets which can be optimized for particular message patterns, including request-response, publish-subscribe, push-pull, and exclusive pair [8][24]. The model being used in this case is request response, as the simulation relies on requesting sensor data and awaiting a response, which falls in line with this model. There is only one send and receive call per sensor data transaction. ZeroMQ is used in the hardware-in-the-loop simulation to provide a request response model to the communication between the sensors and the simulation.

5.1.2 Google Protocol Buffers

While ZeroMQ is used to facilitate communication throughout the hardware-in-the-loop simulation, it views data as just a byte string, and does nothing to format the data. Due to the complex nature of the data being communicated and the importance of its integrity, Google Protocol Buffers are responsible for formatting the data for

usage with ZeroMQ. Protocol Buffers are similar to XML, however they are "smaller, faster, and simpler" according to Google [4]. Protocol Buffers allow users to define the structure of their data and are easy to manipulate in any language [4]. This makes them very useful, as they can be compiled in C++ on the sensor side of the simulation, and in Java on the MATLAB side. Protocol Buffers ensure new information can easily be added or removed from a packet without having to alter any actual code.

Two classes of protocol buffers are used in the hardware-in-the-loop simulation. The 'sensor' class describes sensor readings for a specific device. For example a magnetometer protocol buffer would include an 'x' value, a 'y' value, and a 'z' value, representing the 'x', 'y', and 'z' components of the magnetic field vector that it measures. Additionally, a 'DeviceInfo' protocol buffer describes the type of device and location. Finally, a 'DeviceValue' protocol buffer contains a 'DeviceInfo' buffer and one of the sensor buffers. This 'DeviceValue' ultimately contains both information about the sensor and the readings of the actual sensor. The second class is a 'simulation' class. These provide overhead that make the packets easier to parse on either side of the simulation. The two types of simulation buffers include request and response buffers. Request buffers contain information about the request ('SET' vs 'GET'), time at which the request was made, and a 'DeviceValue' buffer. Response buffers simply indicate either success or error. Originally all information was packed and sent serially, however protocol buffers make information management much more modular. This allows for sensor data and metadata to easily be formatted. Additionally, a function can be called to automatically serialize the protocol buffer for transport via ZeroMQ. An example of a typical request packet can be seen in Figure 5.1.

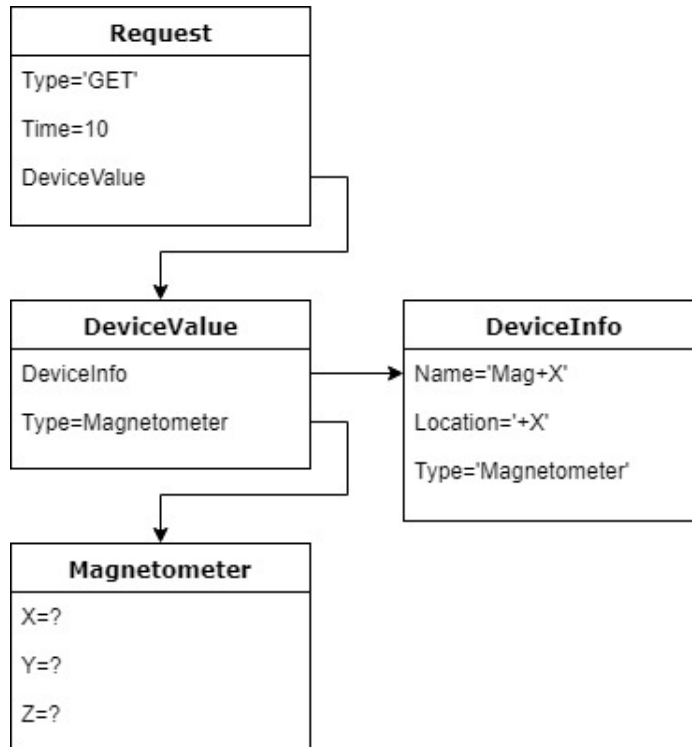


Figure 5.1: Example of a 'GET' Request Protocol Buffer

5.1.3 Virtual Sensors

As mentioned in Chapter 4, the virtual sensor is technically one sensor with the functionality of many sensors and actuators. The virtual sensor is not seen by the ADCS however, only the sensors and actuators mimicked by the virtual sensor. Mainly, it implements objects of the magnetometer, solar angle, magnetorquer, and reaction wheel classes. The virtual sensor takes care of initializing these individual sensors as well as establishing the connection to the ZeroMQ socket with the simulation. Additionally, it implements the 'GET' and 'SET' functions that are utilized by the sensors and actuators, respectively. The 'GET' function sends a request for data to the simulation and blocks until it receives the requested information. The information requested varies depending on which sensor makes the call. For example, a magnetometer making a 'GET' call will receive a magnetic field vector, while a solar

angle sensor making the same call will receive a vector describing the sun’s position in the spacecraft’s body frame. A protocol buffer is passed into this function that describes the type of device making the request, the location of that sensor on the spacecraft, and the time that call is made. This way the simulation knows what sensor data to respond with. The simulation will advance to the requested time if it’s not there already. The ‘SET’ function operates in a similar fashion. The actuator will make a ‘SET’ call when it has data to communicate, usually occurring when a torque command is sent to the magnetorquer, or the reaction wheel changes speed or acceleration. A protocol buffer is created that describes the type of actuator, location of the actuator on the spacecraft, and system time. This data is sent to the simulation, at which point it blocks until it receives a response from the simulation, either a success or failure. The flow of these two functions can be seen in Figure 5.2. Additional implementation details for individual sensor and actuator functionality can be found in the following sections.

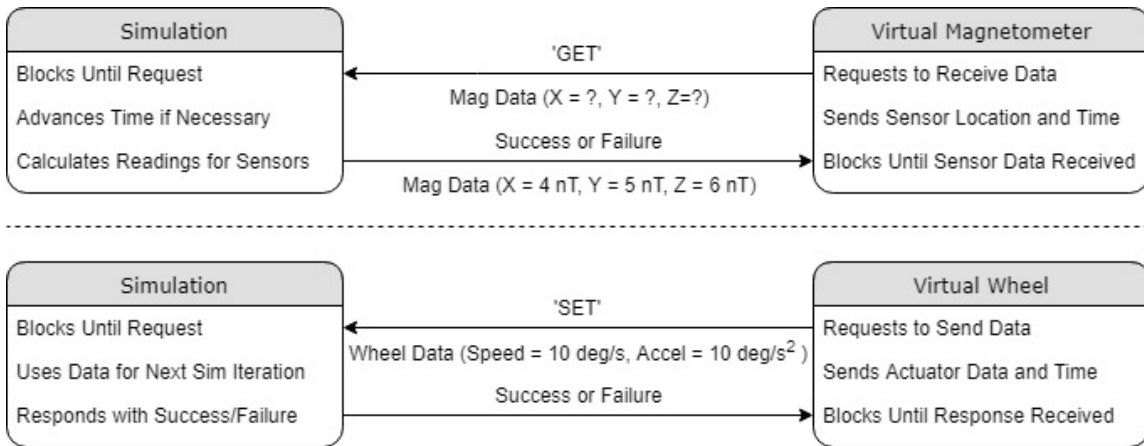


Figure 5.2: Flow of the ‘SET’/‘GET’ Functions

5.1.3.1 Magnetometers

The 'virtual' magnetometer is a fairly straight forward sensor to implement. The only calls to the magnetometer come in the form of *read()*. Where a standard magnetometer's *read()* function would collect data via the hardware, the virtual *read()* collects data from the simulation through ZeroMQ. This function creates a protocol buffer with the appropriate information, makes a 'GET' call, and then returns the appropriate readings. These are the x, y, and z components of the magnetic field vector in the body frame, expressed in units of [nT] (nanoTeslas).

5.1.3.2 Solar Angle Sensors

The 'virtual' solar angle sensor closely resembles the magnetometer in that it also only needs a *read()* function. A 'GET' call is made and the function returns the appropriate data. These include x, y, and z components of a solar position vector in the body frame, as well as an intensity. A sensor with a higher intensity is presumed to have a better view of the sun, meaning the sensor with the highest intensity has its readings used in the ADCS. In physical solar angle sensors, this intensity is representative of the magnitude of the current generated when the sun's rays interact with the photoelectric material of the sensor. However the simulation simply decides which sensors are facing the sun and gives those sensors higher intensity readings.

5.1.3.3 Magnetorquers

The ADCS makes calls to the magnetorquer in the form of several different calls, including *setTorque()*, *startTorque()* and *stopTorque()*, requiring virtual functions of all of these functions.

- *setTorque()* This function emulates that of standard magnetorquers. This

function sets the magnetorquers output to the value that is passed in, usually a magnetic dipole. It is called by the ADCS in preparation for commanding the magnetorquers 'on'. When sent to the simulation via *startTorque()*, these values need to be converted to [Nm] (Newton-meters), so the dynamics model can use these readings to calculate total torque on the spacecraft.

- ***startTorque()*** This is the call made when the ADCS desires to activate the magnetorquers, which then produce the output set in *setTorque()*. The 'SET' call is made in this function to emulate commanding the magnetorquers 'on'. A protocol buffer is made containing the actuator information as well as the values set in *setTorque()*, the 'SET' call is made, and the function returns either a success or error state, depending on the response from the simulation.
- ***stopTorque()*** This function is called by the ADCS when it wants to command the magnetorquers off. Since the virtual magnetorquers aren't actually torquing, the output value is set to zero to represent the magnetorquers being 'off'.

5.1.3.4 Reaction Wheel

The ADCS interfaces with the reaction wheel through several function calls, including *setSpeed()*, *setAccel()*, *readSpeed()*, and *readAccel()*. An interesting fact to note about the reaction wheel is that it has to manually be commanded on from the ground station. The wheel is intended to slowly accelerate to a determined speed, where it will spin continually at that speed. Additionally, unlike the rest of the sensors and actuators, the 'virtual' wheel cannot just pass along information from the ADCS to the simulation. Due to acceleration, the wheel's speed can change over time, which must be accounted for. As a result, the dynamics of a spinning wheel are modeled within the sensor, so speed can be updated appropriately while the wheel is experiencing non-zero acceleration.

- ***setSpeed()*** This function sets the speed of the wheel in $[\frac{deg}{s}]$ (degrees per second). When this information reaches the simulation, it is then converted to $[\frac{rads}{s}]$ (radians per second).
- ***setAccel()*** This function sets the acceleration of the wheel in $[\frac{deg}{s^2}]$. It also records the time at which this acceleration was set, so changes in the wheel speed can be calculated. Should a new acceleration be set, this function will change the speed appropriately based on how much time has passed and reset the timer. Upon reaching the simulation the readings are converted to $[\frac{rads}{s^2}]$
- ***readSpeed()*** This function communicates the speed of the wheel to the ADCS. In addition, the wheel speed is sent to the simulation as well whenever this call is made. As such, it creates a protocol buffer with the wheel information and speed, and makes the appropriate 'SET' call, returning a status based on the response of the simulation. Before this takes place, however, the speed will be updated based on how much time has passed since the current acceleration was set.
- ***readAccel()*** This function operates very similarly to *readSpeed()* in that it uses the 'SET' call to communicate the wheel data with the simulation. As acceleration stays constant, this function can simply pass on the acceleration without keeping track of additional information.

5.2 Simulation Implementation

5.2.1 Simulink

Simulink is a tool integrated into MATLAB that allows for block diagram modeling and simulations [6]. While a space environment simulation could have theoretically been done entirely in MATLAB, utilizing Simulink allows for better modeling and

simulation execution. Each block has its own function, allowing for a more modular simulation. The main simulation contains blocks that model the dynamics and torques applied to the spacecraft, while a separate block is used to calculate sensor readings with various sub-blocks.

5.2.1.1 Dynamics Model

The dynamics model is implemented in a modular nature, with different blocks serving different purposes. A brief overview of each block and its function can be found below.

- **Orbit Propagator** This block is responsible for propagating the orbit of the satellite. This comes in the form of the satellite's position in the ECI frame ([km]), and the satellite's velocity in [km/s]. These are initially determined by initial conditions set by the user.
- **External Torques** This block calculates all the external torques that the spacecraft experiences during flight. Utilizing the spacecraft's position, velocity, and current angular velocity, the external torques being calculated include torque due to the interaction with the Earth's magnetic field, gravity gradient torque, torque from solar pressure, and torque from aerodynamic drag.
- **Total Angular Velocity** Utilizing the information calculated from the previous block, this block calculates and propagates the spacecraft's angular velocity relating the body to the ECI frame. A low angular velocity corresponds to a more stable spacecraft. This block is also where the ADCS's magnetorquers and reaction wheel are taken into account.
- **Quaternion Propagator** This block uses the angular velocity calculated in the previous block to update the quaternion relating the satellite's body frame to the ECI frame. A flow of these blocks can be found in Figure 5.3.

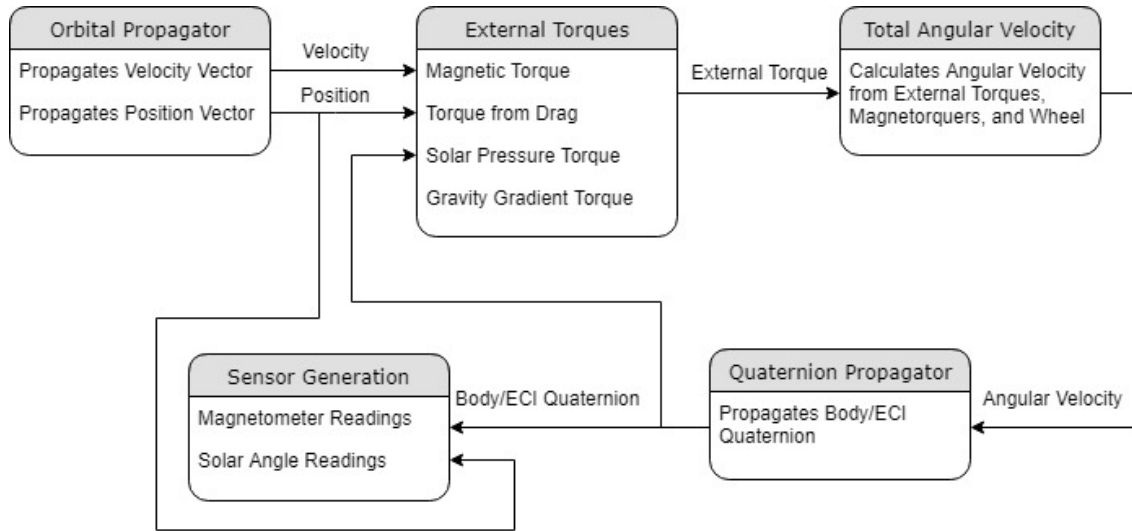


Figure 5.3: Flow of the Dynamics Model

There are additional blocks to propagate a quaternion relating the ECI frame to the LVLH frame, as well as angular velocity with respect to the LVLH frame, however these blocks do not affect the functionality of the simulation and mainly serve to collect additional data for the simulation.

5.2.1.2 Sensor Reading Generation

The sensor generation component relies on a quaternion describing the satellite’s body frame to the ECI frame, as well as the position of the satellite. The position comes in the form of a vector from the center of the Earth the spacecraft in the ECI frame. These two pieces are used in conjunction to accomplish two independent functions: generating solar angle vectors and magnetometer vectors.

To generate solar angle readings, first the position of the sun is calculated in the form of a vector from the Earth to the sun. A vector from the spacecraft to the sun in ECI coordinates can be extrapolated from this and the satellite’s position in ECI coordinates. Using the quaternion, this vector can be translated into the body frame. Using body frame vectors that described the locations of the sensors relative

to the spacecraft, the sensors that were facing the sun (if any) were extrapolated. Higher intensity is granted to the sensors with the better view of the sun. Finally, white noise was added to these readings. While standard solar angle sensors utilize a process mentioned in section 2.1.2 to calculate these vectors, this process was skipped and ADCS useable readings are sent directly to the sensor to keep the sensor implementation simple. To add to this, it is much easier to generate positional vectors than it is to simulate currents generated by solar angle sensors.

In order to generate magnetometer readings, the simulation date is used with the Earth's position to determine where the CubeSat is in Earth's magnetic field. There are functions provided in MATLAB's Aerospace Toolbox that calculate the Earth's magnetic field such as *wrldmagm()* and *igrfmagm()* [5]. A magnetic field vector is returned in the ECI frame, which is then transformed into the satellite's body frame utilizing the provided quaternion. Noise is added, and this one vector is passed to every single magnetometer since variations between the magnetometers is very minimal.

5.2.2 MATLAB

MATLAB is used to implement support functions that are used in the model, as well as the main script that drives the entire hardware-in-the-loop system. Some examples of these supporting functions include *sunfinder()*, which describes the sun's position to the Earth on any given date and time, *drag()*, used to calculate torque on the spacecraft due to aerodynamic drag, and *jDataCalc()*, which converts the UTC time into a Julian time. These support functions serve to limit the code in the Simulink blocks to keep it concise.

The main script is responsible for coordinating the simulation and communicating to and from the sensors. This script consists of two parts: the main control loop and

the 'preamble'. The preamble is responsible for creating the ZeroMQ socket to which the virtual sensor will connect, as well as setting up scripts for data logging and setting initial conditions on the simulation. With the help of MATLAB's workspace functionality, a multitude of variables can be recorded and graphed over time to further analyze the behavior of the ADCS. The main control loop blocks immediately until it receives a packet from the virtual sensor. At this point, if the time from the ADCS is greater than the time on the simulation, the Simulink model will propagate to that time, utilizing the output conditions from the last iterations as input conditions to the next iteration. Since the ADCS usually operates at 1Hz, the simulation will be run in intervals of one time-step. Once this is accomplished, the packet is parsed and the appropriate information is acquired from the protocol buffer. If a sensor requests data, the sensor output from the simulation correlating to the requesting sensor is collected and sent to the ADCS. If an actuator is attempting to send data, the appropriate torque commands or wheel speeds are collected and converted and set as conditions for the next time the Simulink model is executed. Once the simulation is over, the socket is closed and data is charted.

Chapter 6

EVALUATION AND DISCUSSION

This chapter is divided into four sections, with each section dedicated to a different test. Section 6.1 investigates the simulation itself and explores the limitations of the dynamics model. Section 6.2 focuses on verifying the functionality of the hardware-in-the-loop system by examining how the simulation uses virtual sensors to communicate with the ADCS. Section 6.3 is very similar to Section 6.2, however the ADCS is run on flight hardware rather than a virtual machine. This is done to ensure that the hardware-in-the-loop system can function as the name suggests, rather than as a software-in-the-loop simulation. Section 6.4 looks into the performance of the system and investigates where bottlenecking occurs within the simulation.

It should be noted that besides the tests conducted in Section 6.3, the ADCS process is executed on a virtual machine running Ubuntu 16.04. Besides configuring the process to use the virtual clock instead of the system clock, the functionality of the ADCS was not modified in any way. The simulation itself was run on a variety of Windows machines, with distinctions being noted in the following sections when appropriate.

Tests are executed in time units known as orbits, which is the estimated time that the CubeSat takes to orbit the Earth. In the hardware-in-the-loop simulation this has been estimated to be around ninety minutes. There is a distinction that needs to be made here: the amount of virtual time the test is running for doesn't necessarily indicate how long the test will take to complete. For example, a test that executes for ten orbits will run for about fifteen simulated hours, however it may take twenty hours to complete in real time due to the speed of the system. The amount of real time required to execute these tests is investigated in Section 6.4.

6.1 Simulation

The dynamics model is very difficult to verify, as there was not a model that was known to be true that was available for comparison. And while it is believed to be reasonably accurate for the purpose of this thesis, the simulation has its limitations, and cannot compare to models developed and verified over many years. The accuracy to which the ADCS can be verified is dependent on how accurately the environment can be modeled.

6.1.1 Discussion

For evaluation purposes, the dynamics model was run as a stand-alone system. This resembles a satellite being propagated through space with no attempt to stabilize it. Additionally it removes the overhead of generating sensor readings, as well as communicating with the ADCS. This allows for the model to execute at a very fast rate, resulting in multiple orbits calculated within minutes. This makes it easier to examine long-run trends within the dynamics model. It should be noted that the simulation is not verified, due to the reasons mentioned above. The simulation has passed through code review and tested, however this verification consisted of various spot checks with specific inputs and outputs. Additionally, the math behind the orbit of a satellite is well understood, so this was reviewed as well to ensure correctness. While the math is correct, it doesn't necessarily imply that the simulation is a correct implementation of those mathematical models and laws. This fact isn't of high consequence however because any simulation can technically be used in the hardware-in-the-loop simulation.

6.2 Functional Verification

This section is aimed at verifying the functionality of the hardware-in-the-loop simulation. This involves ensuring the pipeline is working properly; the simulation should be able to accurately pass sensor readings to the ADCS, with the sensor readings received by the ADCS matching those readings sent by the simulation. This is the same case with the actuators, where the same commands that are sent by the ADCS match what is received by the simulation. Ensuring these virtual sensors allows for other simulations to easily be integrated in the future.

6.2.1 Evaluation

The simulation was run with many different initial conditions to exercise the system. This ensured that a variety of data was sent through the virtual sensors, as well as that edge cases were being tested as well. Tests were usually run for about half an orbit to ensure that data was still being communicated appropriately. Below is a summary of what was examined for each of the virtual sensors.

- **Virtual Magnetometer** To verify that the magnetometers are communicating the correct data, the magnetic field in nanoTeslas needs to be confirmed on both sides. This means that the magnetic field vector sent by the simulation must match the magnetic field that is registered for the sensor that read it. There are no edge-cases to look out for with the virtual magnetometer sensors.
- **Virtual Solar Angle Sensor** The verification process for the solar angle sensors is similar to that of the magnetometer. The sun positional vector as well as the light intensity sent by the simulation must match the vector and intensity read by the ADCS, however there are a few more things to watch out for in this case. Only sensors that are facing the sun should be reading data, so these

sensors (as determined by the simulation) should be the only ones registering actual readings on the ADCS side. These readings should have non-zero sun vectors and intensities, while the sensors facing away from the sun should be reading zeroes. Additionally, should the satellite be in eclipse (meaning that the Earth is blocking the satellite's view of the sun), all sensor readings must be reading zero. All of these cases must be confirmed on both the simulation side as well as the ADCS side.

- **Virtual Magnetorquer** Virtual Magnetorquers are pretty simple to verify. The torque magnetic dipole commands that are calculated by the ADCS must be received by the simulation. Similar to the magnetometers, there aren't any edge cases to worry about. However, when the magnetorquers are not being used, the simulation registers zero values for the command torque.
- **Virtual Reaction Wheel** The speed and acceleration of the wheel, in degrees per second and degrees per second squared respectively, need to be accurately communicated to the simulation. This includes when the wheel is off, meaning no speed or acceleration, the wheel is spinning up, corresponding to constant acceleration and changing speed, and finally constant speed. All of these cases need to be verified on both the simulation and ADCS side.

To verify all this data was being communicated correctly, the data was logged on both the ADCS side as well as the simulation side. The ADCS has a debug file that logs all relevant information that passes through the ADCS including sensor and actuator readings. This file was compared to the sensor and actuator readings that were logged on the simulation side for accuracy. Again initial conditions of the simulation were varied in order to ensure all the cases previously mentioned were examined. The standard initial conditions were taken from the original ExoCube. The second set of initial conditions were calculated to start the satellite in an eclipsed state. Finally,

there is also a set of random initial conditions for good measure. These are tests for correctness rather than performance tests which will follow later. A table summarizing the test results can be found in Table 6.1. The table describes the test run as well as the results for the system on both the virtual machine and the Intrepid system board. Listed are what initial conditions (ICs) were used, what sensors and actuators were tested (abbreviated 'mag' for magnetometer, 'sol' for solar angle sensor, and 'tor' for magnetorquer), and whether the system passed or failed.

6.2.2 Results

The results of the various tests revealed the virtual sensors are an excellent way of transmitting environmental data from a simulation to the ADCS, and vice-versa. The data expected was always the data received, with accuracy only being lost within the ADCS itself (for example converting a float value to an int). ZeroMQ is very reliable, with no dropped packets or corrupt data experienced within any of the tests. The ADCS was able to use the information from these virtual sensors as if they were standard sensors, and the system believed that it was operating within an actual space environment. This was confirmed by the comparison of 'secondary data'. The ADCS has propagators similar to that used by the simulation to gain an estimate of its position. Testing showed not only was the correct data being conveyed by the sensors and actuators, but the estimated position and velocity as computed by the ADCS matched that computed by the simulation. All edge cases were verified to be correct, with the ADCS acting appropriately to each one. These sensors should allow any process that interfaces with them successfully utilize virtual data from a simulated environment.

Test Description					ADCS Environment	
ICs	Mag	Sol	Tor	Wheel	Virtual Machine	Intrepid Board
Standard	✓	✓	✓	X	Pass	Pass
Standard	✓	✓	✓	Constant	Pass	Pass
Standard	✓	✓	✓	Accelerating	Pass	Pass
Eclipse	✓	✓	✓	X	Pass	Pass
Eclipse	✓	✓	✓	Constant	Pass	Pass
Eclipse	✓	✓	✓	Accelerating	Pass	Pass
Random	✓	✓	✓	X	Pass	Pass
Random	✓	✓	✓	Constant	Pass	Pass
Random	✓	✓	✓	Accelerating	Pass	Pass

Table 6.1: Summary of Tests Conducted

6.3 Hardware-in-the-Loop Verification

The purpose of the tests conducted in this section is identical to that in Section 6.2, however rather than having the ADCS execute on a virtual machine, it instead runs on a system board. This board is identical to the boards that will be running the software on the actual ExoCube II when it comes time to launch. This is important to verify as the ADCS should be tested on its hardware environment. While it is important to simulate an accurate space environment for the ADCS, the importance of replicating the hardware environment cannot be overlooked.

6.3.1 Evaluation

As this section falls in line with Section 6.2, the testing procedure in that section will be replicated, the only difference being the hardware environment that the ADCS

is run on (in this case an Intrepid system board). The ADCS and virtual sensors software is recompiled for the ARM architecture of the board and then transferred onto the board itself. The simulation itself will still be run on a Windows machine. Again, initial conditions are varied to ensure diverse readings pass through the ADCS. The functionality of the ADCS itself is not being examined, but more importantly the ability of the virtual sensors to transmit this data to and from the ADCS. The magnetometers, solar angle sensors, magnetorquers, and wheel are all being tested the same way as they were in the previous section, with all edge cases being examined. It should be noted that while the software is compiled for a different architecture and running on a different environment, the software is not expected to behave differently, nor should the software be edited to specifically work with the board. A summary of the test results can be found in Table 6.1.

6.3.2 Results

These tests reaffirm the results of the systems functional verification done in Section 6.2. Simulation length varied from a quarter to a half of an orbit to ensure data integrity maintained constant. The sensors were able to seamlessly communicate to the simulation from the board just as easily as they had from the virtual machine. ZeroMQ and Google Protocol Buffers also behaved as expected with no observed issues; the data sent by the simulation matched the data received by the ADCS, and vice-versa. This comes as no surprise as previous PolySat missions utilized both ZeroMQ and Google Protocol Buffers. While the ADCS will most likely be run on a virtual machine for the duration of the debugging period in the future, final ADCS functional verification will occur on the system board.

6.4 Performance Bottlenecking

The hardware-in-the-loop simulation did not run as fast as expected, taking about four hours to complete one orbit. For longer tests involving five to ten orbits, the simulation took up to forty hours to execute. This raises the question as to where the bottlenecking is occurring. The most likely candidates are the simulation, the Simulink program itself or the ADCS. Since the ADCS functionality cannot be modified for the sake of the hardware-in-the-loop testing, even if the ADCS is causing the bottlenecking, there isn't much that can be done about it. This leaves the simulation, where complex logic could be taking a long time to compute, or Simulink, whose overhead may be causing the simulation to run slower than expected. This section runs a multitude of different tests in order to ascertain where the bottleneck is occurring.

6.4.1 Evaluation

Four simulations are run in conjunction with the ADCS, each with different properties that are meant to individually test components of the original simulation. Each of these simulations are described below.

- **Simulation - Original** This is the original simulation used for the hardware-in-the-loop simulation and will serve as a baseline to which all the other simulations will be compared to in terms of execution time.
- **Simulation - Dynamics Only** This simulation contains only the dynamics model of the original simulation, while removing the sensor generation logic. Since the speed of the dynamics model is being tested here, the accuracy of the ADCS's inputs and outputs are not a concern. As a result, constant readings are sent to the ADCS every time step to maintain the integrity of the loop.

- **Simulation - Sensor Generation Only** This simulation is similar to the previous one, however this time the dynamics model is stripped out and only the sensor generation remains. This simulation reads in actuator data from the ADCS, however it does nothing with it. Sensor readings are continually calculated from a single position vector and quaternion and sent to the ADCS each time step. Again, accuracy is not as concerning as the speed at which these tests execute. The speed of this simulation should reveal if logic in the sensor generation portion is slowing the simulation down.
- **Simulation - Stripped Down** This is the simplest of the simulations being tested. Both the dynamics model and the sensor reading generation portions are stripped out. This simulation simply reads in data from the ADCS and outputs constant sensor readings for the ADCS to use. Since there are no potentially complex calculations done in this model, it should reveal whether or not Simulink is the cause of the bottlenecking.

Each of these simulations are used in place of the original, and are executed for a varying number of orbits. It should be noted again that the estimated time it takes for the satellite to complete one orbit around the Earth is about ninety-five minutes. The time that these tests take to complete is recorded and compared. Each test is run three times per simulation per number of orbits, and an average of that time is taken. Due to the length of time required to run the original simulation however, that simulation is only run once per number of orbits. Simulations that offer remarkable speed-up compared to the original may indicate that the bottleneck is no longer present within that simulation. Examining the reverse, simulations that don't offer as much improvement may indicate the bottleneck could still be present within that simulation. It should also be noted that the simulation is run on a machine comprised of an Intel i7 quad-core processor operating at 4.5GHz and 16GB of RAM. This has

been noted because the simulation has been observed to perform at different speeds depending on the hardware available. Results of these tests, as well as an analysis of them, can be found in the next section.

6.4.2 Results

Timing results of each simulation can be found in Table 6.2. These tests reveal some interesting trends. Comparing the times between the original and switched simulations, it is clear that the simulation itself was causing a majority of the bottlenecking, as the time to complete a one-orbit test went from a little over four hours to about fifteen minutes. It should be noted that one orbit for the stripped simulation still took fifteen minutes, meaning that Simulink and the process of passing information to and from the ADCS, as well as executing one step of the ADCS itself, takes a small amount of time to execute. Because of this, the time estimates for the stripped simulation could be considered the theoretical limit of how quickly the hardware-in-the-loop simulation could run using a Simulink model. This however would be highly unreasonable as a simulation running that quickly would likely be useless.

Number of Orbits	Original Sim	Dynamics Sim	Sensor Sim	Stripped Sim
1	4h 12m 27s	2h 53m 1s	0h 40m 15s	0h 15m 31s
3	12h 16m 29s	8h 42m 16s	2h 1m 30s	0h 46m 38s
5	20h 33m 35s	14h 41m 28s	3h 23m 3s	1h 13m 13s

Table 6.2: Timing Results of each of the Simulations

The middle two simulations were utilized to determine which aspect of the simulation was eating up time. Based on the results, both the dynamics model and the sensor generation are computationally heavy, with the dynamics model taking longer to execute than the sensor generation by about a factor of four. This is where the

simulation is taking up the most time, however the sensor generation portion had non-negligible execution time as well. One thing to note was that for this simulation, sensor readings were continually being generated for the same position vector and quaternion, so in reality the sensor generation portion of the simulation would take a bit longer to execute since these values are not normally constant. While it is interesting to see where the bottlenecks are occurring, ultimately there is very little that can be done to optimize these values without damaging the integrity of the simulation. Based on this data, however, the dynamics model was able to be optimized by removing redundant calls to expensive MATLAB functions, for example the *wrldmagm()* function that computed the Earth's magnetic field. This brought the run time of the entire simulation to about two hours on average. While this essentially doubled the speed of the simulation, unfortunately this is still very slow. One solution to address this would be to possibly convert the entire simulation into many MATLAB functions, however this makes the simulation much less modular and more difficult to manage from a design perspective. This was partially investigated when the sensor readings code was temporarily moved to MATLAB, however this offered no notable speed increase.

Chapter 7

FUTURE WORK

7.1 NASA 42

While the hardware-in-the-loop simulation works as intended, it is clear that the dynamics model is the could be improved upon. Not only does it slow the entire system down, it is also not verified to be correct. While believed to be reasonable, ultimately the verification process the model went through is less than ideal. The most logical approach to this project in the future is to replace the dynamics model with the most appropriate candidate: NASA 42.

Previously mentioned in Chapter 3, NASA 42 is a open-source program designed to provide an environment for creating and testing attitude control algorithms. This is the direction that PolySat should take for hardware-in-the-loop ADCS testing. With this, students could focus less on creating an accurate dynamics model and instead shift focus to the ADCS itself, while using a trusted and flushed-out model from NASA. As touched upon in Chapters 3 and 6, there is no simple way to validate an environmental model. While editing NASA 42 to be compatible with PolySat's software architecture would certainly be a significant undertaking, the integration of the program would greatly improve the accuracy and speed of the hardware-in-the-loop simulation.

Chapter 8

CONCLUSION

The Attitude Determination and Control Subsystem is a critical component to any CubeSat with an attitude-dependent payload. Without such a system, a CubeSat such as ExoCube II would not be able to stabilize itself nor orient itself to the correct attitude. Such an important system requires copious amounts of testing, however this depends on an accurate environment to be simulated and the use of emulated sensors and actuators. Physical sensors that the ADCS relies on for data collection such as magnetometers and solar angle sensors don't operate correctly on the ground without being augmented by additional equipment [16]. Without access to such equipment, many software-related approaches to testing ADCS algorithms have arisen, many of which can be found in Chapter 3.

This thesis has presented a hardware-in-the-loop simulation that, through the use of virtual sensors, is able to provide a simulated environment with which to test the functionality of an ADCS process. Based upon the thorough testing that was conducted, the virtual sensors have demonstrated to be an effective way of interfacing a process such as the ADCS with a simulated environment. Additionally, the system allows for the ADCS to be tested as it would run on the satellite, with exception of a virtual clock rather than the system clock. Both requirements explained in Chapter 4 are met: the hardware-in-the-loop system can operate without affecting the behavior of the ADCS, and the virtual sensors can successfully deliver data to and from the ADCS.

While functional, the hardware-in-the-loop system is not free of issues. As mentioned upon numerous times throughout the paper, uncertainty exists within the model. With no true model as a basis for comparison, it is difficult to remove this

uncertainty. Additionally, the system is also heavily bottle-necked by the simulation. The results in Chapter 6 show that even simulating the satellite for one orbit can take hours to complete on a higher-end machine. While this bottleneck testing allowed for that execution time to be cut in almost half, the time it takes to simulate an entire orbit is still around two hours, which is unfortunately slower than real time. The way forward for these various issues was discussed in Chapter 7.

Despite all these issues, the hardware-in-the-loop simulation can be used as a useful tool to see trends within the ADCS when testing. Most of the issues that occur are with the simulation itself, so with a different simulation in place, the tool becomes even more powerful. Additionally, the virtual sensors could be used with any PolySat process in the future should there be a need. Thorough system testing of the ADCS is not an easy problem to solve, and ultimately this hardware-in-the-loop simulation provides PolySat with testing capability that it did not previously have.

BIBLIOGRAPHY

- [1] 42: A comprehensive general-purpose simulation of attitude and trajectory dynamics and control of multiple spacecraft composed of multiple rigid or flexible bodies. <https://software.nasa.gov/software/GSC-16720-1>.
- [2] Cal Poly Github. <http://www.github.com/CalPoly>.
- [3] CubeSat. <http://www.cubesat.org>.
- [4] Google Protocol Buffers.
<https://developers.google.com/protocol-buffers>.
- [5] MathWorks Aerospace Toolbox. <https://www.mathworks.com/help/aerotbx>.
- [6] MathWorks Simulink.
<https://www.mathworks.com/help/simulink/index.html>.
- [7] PolySat. <http://www.polysat.org>.
- [8] ZeroMQ. <http://zeromq.org>.
- [9] D. Bertolino. Implementing a matlab based attitude determination algorithm in c within the polySat software architecture. Master's thesis, California Polytechnic State University, San Luis Obispo, 2013.
- [10] C. Clark and S. Chesi. Effective hardware-in-the-loop cubesat attitude control verification and test. In *CubeSat Developers Workshop*, 2015.
- [11] The CubeSat Program, Cal Poly SLO. *CubeSat Design Specification*.
- [12] A. H. de Ruiter, C. J. Damaren, and J. R. Forbes. *Spacecraft Dynamics and Control An Introduction*. John Wiley & Sons, Ltd, 2013.

- [13] R. Eimerl. The quaternions and their applications, 2015.
- [14] J. Familton. Quaternions: A history of complex noncommutative rotation groups in theoretical physics. Master's thesis, Columbia University, 2015.
- [15] V. Francois-Lavet. Study of passive and active attitude control systems for the oufti nanosatellites. Master's thesis, Univeristy of Liège, 2010.
- [16] I. Gavrilovich, S. Krut, M. Gouttefarde, F. Pierrot, and L. Dusseau. Test bench for nanosatellite attitude determination and control system ground tests. In *The 4S Symposium*, 2014.
- [17] R. Kinnett. System integration and control of a low-cost spacecraft attitude dynamics simulator. Master's thesis, California Polytechnic State University, San Luis Obispo, 2009.
- [18] G. Manyak. Fault tolerant and flexible cubesat software architecture. Master's thesis, California Polytechnic State University, San Luis Obispo, 2011.
- [19] A. Nicholas and D. Miller. Attitude and formation control design and system simulation for a three-satellite cubesat mission. Master's thesis, Massachusetts Institute of Technology, 2013.
- [20] PolySat. Polysat Internal Documentation.
- [21] M. Quadrino, D. Miller, and K. Cahoy. Testing the attitude determination and control of a cubesat with hardware-in-the-loop. Master's thesis, Massachusetts Institute of Technology, 2014.
- [22] S. Robinson. *Simulation - The Practice of Model Development and Use*. John Wiley & Sons, Ltd, 2004.

- [23] S. Rossi, A. Ivanov, G. Faure, B. Geissman, J. Amiguet, R. Valceschini, M. Starein, R. Zufferey, and G. Burri. Cubeth adcs design, implementation and validation tests. In *66th International Astronautical Congress*, 2015.
- [24] M. Sústrik. Scalability later hits the internet stack. In *Linux Kongress*, 2010.
- [25] W. Tapsawat, T. Sangpet, and S. Kuntanapreeda. Development of a hardware-in-loop attitude control simulator for a cubesat satellite. *IOP Conference Series: Materials Science and Engineering*, 297(1), 2018.
- [26] R. Votel and D. Sinclair. Comparison of control moment gyros and reaction wheels for small earth-observing satellites. In *AIAA/USU Conference on Small Satellites*, 2012.
- [27] Y. Winetraub and A. B. Heller. Attitude determination advanced sun sensors for pico-satellites. 2005.