Western University
Faculty of Engineering
Mechatronic Systems Engineering Program

CubeSat Attitude Determination and Control System

Final Report

James Brunke (250860834)
Rachelle Cheung (250846623)
Lauren Flanagan (250841678)
John McConkey (250835398)

Executive Summary

CubeSats require a system which can control their orientation in a micro-gravity environment: this is known as an attitude determination and control system. The requirement of this system stems from the need of the CubeSat to point to specific locations in order to transmit data or to maximize power generation. After reviewing existing technologies, a gap was identified for low-cost systems that provide control in all three axes. This led the group to the following problem statement:

Design a low-cost system to stabilize and control the pose of the CubeSat.

The selected concept for the system uses magnetometers, gyroscopes, and sun sensors to attain peripheral information to define its orientation at any point throughout its orbit. The concept uses magnetorquers as actuators, which can interact with the Earth's magnetic field in order to actuate the satellite. The control law is designed using B-dot Bang-Bang control. The purpose of this is to control the current provided to the magnetorquers based on the attitude determined by the sensing system. B-dot Bang-Bang control also sets a limit for the amount of current that is sent to the magnetorquers to ensure that it produces the correct amount of torque.

To determine the orientation, the raw sensor measurements obtained in the Simulink model can be used in conjunction with the TRIAD. The TRIAD is used to extrapolate useful three-dimensional vectors from raw data. The simulation results demonstrated that the selected concept would succeed, and thus validated our design.

After several iterations, the prototype was developed to demonstrate how the ADCS is able to determine the orientation of the CubeSat and how it is able to rotate to a desired position. Since magnetorquers were too expensive to purchase, reaction wheels were built to be used as the actuators in the prototype. These reaction wheels were able to provide the same torque that the magnetorquers would have provided. To determine the position of the prototype, a gyroscope was used for attitude determination.

Table of Contents

List of Figures

List of Tables

1. Introduction

Western University's Faculty of Engineering received a grant from the Canadian Space Agency (CSA) to develop the CubeSat over the course of three years, beginning in May 2018. When a CubeSat is in space, its attitude must be determined and controlled for the entire mission lifetime. The design requirements of the CubeSat led to the following problem statement:

> Design a low-cost system to stabilize and control the pose of the CubeSat.

When the CubeSat is deployed from the International Space Station (ISS), the attitude determination and control system (ADCS) must detumble the CubeSat. During normal operation, the ADCS will orient the CubeSat to desired positions throughout its orbit. The antenna must be pointed towards the ground station to transmit and receive data. To take images, the ADCS should ensure the camera is pointed towards the target location and stabilized well to take clear images. To do this, sensors are located on the CubeSat to acquire knowledge of its current position. With this knowledge, the attitude control algorithm will provide appropriate current to the actuators to orient the CubeSat to the desired position. The sensors and actuators chosen for the design are sun sensors, a magnetometer, a gyroscope, and magnetorquers. Although there are other attitude determination and control systems that have been created for other CubeSat projects, this ADCS will be designed based on the specifications determined for this CubeSat.

This report covers the background information needed to understand the fundamental aspects of the project, the scope, design objectives and constraints, the concept generation and selection process, the analysis and refinement of the selected concept, prototype development, modification and improvements to be made to the prototype and design of the system, and an outline of the resources and budget needed for the selected design concept.

2. Design Requirements & Constraints

2.1. Scope

The ADCS must satisfy many objectives and constraints to efficiently stabilize and control the attitude of the CubeSat. To accomplish this, appropriate sensors and actuators were selected for the design. A control law was developed to take the vectors provided from the sensors that specify the position and compare it to the desired position of the CubeSat. This information was manipulated to create an internal torque using the magnetorquers to move the CubeSat into its desired position. This was to be done while prioritizing a low system cost.

2.2. Objectives

The components of the ADCS should be as compact and lightweight as possible. For the ADCS to last the entire mission, it should have low power consumption. It should be durable and withstand external forces and torques during transportation and deployment. The ADCS should be reliable so it can perform the tasks required. A visual summary of the objectives can be seen in Figure 1.

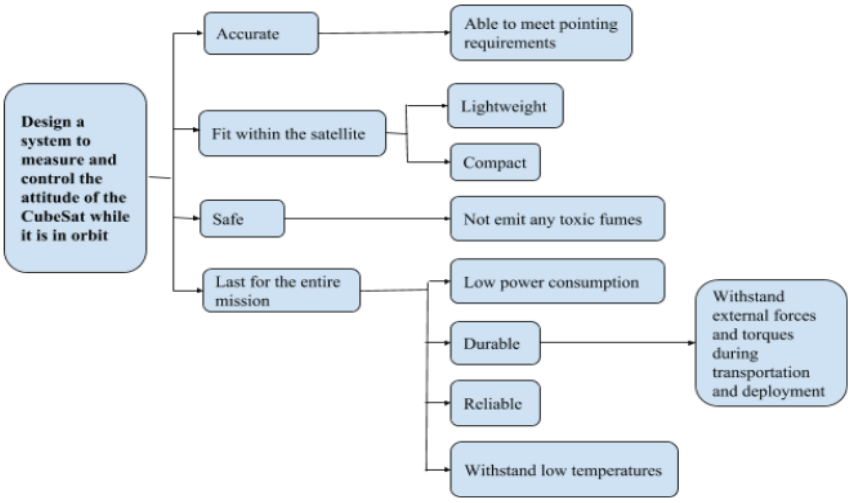

Figure 1: Objectives Chart

2.3. Constraints

A functional constraint of the ADCS is that it must be able to point and stabilize the CubeSat along three axes within 15° accuracy, which is what the ground station requires for communication. The CubeSat must weigh less than 3.6kg, which is an external constraint of the launch provider. All the components of the ADCS must weigh at most 160g, a constraint defined by the project supervisor. The ADCS must fit within the size of the CubeSat, a 20cmx10cmx10cm volume. The cost constraint for the ADCS for the CubeSat is $25,000. The allowable power range of the ADCS must be within 200mW for the CubeSat to function for its mission lifetime of one year. Considering the CubeSat will be operating in space, the materials must be operable in near-vacuum conditions [1]. Components in space are susceptible to erosion due to atomic oxygen present in the environment and must resist approximately 5.2eV of energy. The CubeSat will also be exposed to UV radiation, so any polymers used must withstand both cross-linking (hardening) and chain scission (weakening) [2]. Finally, the CubeSat must not require propellant handling which is a constraint provided by the CSA.

3. State-of-the-art & Emerging Technologies

The CSA CubeSat Project has provided a grant to Western University to design and build a two-unit CubeSat in collaboration with Nunavut Arctic College. The purpose of the CubeSat project is to provide a learning experience and training opportunity for students enrolled in post-secondary education. The CubeSat is to be launched into space in the year 2021 and will be used for demonstrating a novel video-recording camera in orbit.

3.1. PhoneSats

In 2013, NASA started a similar project; they launched a series of nano-satellites called PhoneSats. The project uses Google Android technology and integrates it into a CubeSat. The goal of the project was to build inexpensive satellites to be sent to space. As a result, it is now possible for nano-satellites to be made at a fraction of the cost [3]. This project pertains to Western University's CubeSat project as the design must be cost-effective and will be used for similar purposes such as audio/video recording and radio communication. The PhoneSat varies from other projects as its attitude determination system is based on a star tracker app that each PhoneSat team creates. For attitude control, the PhoneSat uses the phone's vibration function which only allows for one axis control [4]. This ADCS provides insight on how other ADCS could be developed.

3.2. TJ3Sats

Another student-run project was conducted by Thomas Jefferson High School for Science and Technology in Alexandria, Virginia. The satellite, TJ3Sat, has been successful in receiving and transmitting signals [5]. Although the TJ3Sat does not have any actuators to control its attitude, it has telemetry sensors that provide information on current and voltage data for different hardware components [6]. Using this information, the satellite tumbling can be determined. Knowing the

tumble of the CubeSat is necessary to provide information to the actuators for the CubeSat to orient itself.

3.3. Western CubeSat Project

The CubeSat project encompasses students from various disciplines in Western University's engineering program. The subsystem groups will be coordinated by the Project Manager, Matthew Cross, to ensure the CubeSat is designed as a complete system. After reviewing the other satellites, a gap was identified in the ability to control the attitude of the satellite on student-built nanosatellites in more than one axis. Our goal is to produce a low-cost ADCS that will allow for attitude determination and control in all three axes. To accomplish this, the ADCS team will need to use a variety of engineering tools including MATLAB, Simulink, and SolidWorks.

4. Generation & Evaluation of Concepts

4.1. Detailed Description of Selected Concepts

A preliminary analysis of the sensor and actuator options was completed by considering cost, power consumption and weight, as well as the team's ability, resources, and the project restrictions.

4.1.1. Detumbling Mode

When a CubeSat is deployed from the ISS, it undergoes tumbling at a rate of about a few degrees per second around each of its three axes. The CubeSat must reduce its initial rate quickly for normal operation to occur; it does this through detumbling. For the detumbling mode, two concepts were considered.

4.1.1.1. Possible Actuators for Detumbling Mode

4.1.1.1.1. Using Magnetorquers for the Detumbling Mode

To use magnetorquers as actuators for attitude control, three magnetorquers are place perpendicular to each other around the CubeSat's X-, Y- and Z-axes. When the magnetorquers are turned on, they will create a magnetic field that interacts with the Earth's magnetic field, allowing the CubeSat to detumble.

4.1.1.1.2. Using Reaction Wheels for the Detumbling Mode

Upon deployment, the reaction wheel is not moving [7]. To detumble the CubeSat, three reaction wheels positioned with one along each axis must be spun such that the resulting torque acts in the opposite direction to the rate of tumbling to detumble the CubeSat [8].

4.1.1.2. Selection of Actuators for Detumbling Mode

Both magnetorquers and reaction wheels were considered for the detumbling mode and placed into both Go/No-Go and Selection Matrices. There were several requirements considered to determine the most suitable concept for detumbling found in Table 1.

Table 1: Requirements for Modes of Operation

Number	Requirements for Modes of Operation
1	Lightweight
2	Compact
3	Not emit any toxic fumes
4	Not require propellant handling
5	Low power consumption
6	Withstand external forces and torques
7	Reliable
8	Withstand thermal cycling
9	Low cost
10	Meet voltage Requirements

Table 2: Go/No-Go Matrix for the Detumbling Mode

	1	2	3	4	5	6	7	8	9	10
Mai-400 Reaction Wheel	G	G	G	G	M	G	G	G	NG	G
ISIS Magnetorquer Board	NG	NG	G	G	M	G	G	G	NG	G
NCTR-M002 Magnetorquer Rod	G	M	G	G	G	G	G	G	G	G
Cube Torquer and Cube Coil Bundle	G	M	G	G	M	G	G	G	G	G

Table 3: Pairwise Comparison Chart of Detumbling Phase and Normal Operation

	1	2	3	4	5	6	7	8	9	10	Total	Weight
1	1	0.33	3	3	0.33	5	5	5	0.2	7	29.86	0.129
2	3	1	5	3	0.33	7	5	5	0.33	7	36.66	0.158
3	0.33	0.2	1	0.33	0.2	5	3	3	0.2	5	18.26	0.079
4	0.33	0.33	3	1	0.2	5	5	3	0.2	5	23.06	0.100
5	3	3	5	5	1	7	7	5	0.33	7	43.33	0.187
6	0.2	0.14	0.2	0.2	0.14	1	0.33	0.33	0.2	3	5.74	0.025
7	0.2	0.2	0.33	0.2	0.14	3	1	0.33	0.2	3	8.6	0.037
8	0.2	0.2	0.33	0.33	0.2	3	3	1	0.14	5	13.4	0.058
9	5	3	5	5	3	7	5	7	1	9	50	0.216
10	0.14	0.14	0.2	0.2	0.14	0.33	0.33	0.2	0.11	1	2.79	0.012
Total											**231.7**	**1**

Table 4: Selection Matrix for the Detumbling Mode

	Weight	CubeWheel Small	Mai-400 Reaction Wheel	ISIS Magnetorquer Board	*NCTR-M002 Magnetorquer Rod*	Cube Torquer and Cube Coil Bundle
1	0.129	*0*	*0*	*-1*	*0*	*0*
2	0.158	*0*	*0*	*-1*	*-1*	*-1*
3	0.079	*0*	*0*	*0*	*0*	*0*
4	0.100	*0*	*0*	*0*	*0*	*0*
5	0.187	*0*	*-1*	*-1*	*0*	*-1*
6	0.025	*0*	*0*	*0*	*0*	*0*
7	0.037	*0*	*0*	*0*	*0*	*0*
8	0.058	*0*	*0*	*1*	*0*	*0*
9	0.216	*0*	*0*	*1*	*1*	*1*
10	0.012	*0*	*0*	*1*	*0*	*0*
Total	***1***	***0***	***-0.187***	***-0.474***	***0.058***	***-0.129***

From Table 4, it was clear that the NCTR-M002 Magnetorquer Rod was the only option that yielded a positive result. Thus, the development of the detumbling mode progressed using this specific actuator.

4.1.2. Possible Sensors for Normal Operation

4.1.2.1. Normal Operation

There must be two independent vectors measured to determine how the CubeSat is oriented at any given point in time. This would require at least two sensors. The sensors that were taken into consideration include sun sensors, gyroscopes, magnetometers and star trackers.

4.1.2.1.1. Using Sun Sensors for Normal Operation

Six sun sensors would be placed on the CubeSat, one on each face, which would detect where the sun is in relation to the CubeSat, resulting in a sun vector. This vector alone would not be sufficient in determining the exact location of the CubeSat. It is important to note that the sun sensor would be inadequate during solar eclipse.

4.1.2.1.2. Using Magnetometers for Normal Operation

One magnetometer aboard the CubeSat would detect the Earth's magnetic field, resulting in a vector normal to the Earth's magnetic field at the CubeSat's current orbital position. It is important to note that the magnetometers would not be able to operate synchronously with the magnetorquers as the magnetorquers create a magnetic field themselves. The magnetometer would also need to be calibrated to accommodate the residual magnetic field aboard the CubeSat.

4.1.2.1.3. Using Gyroscopes for Normal Operation

There would be three gyroscopes, one placed on each axis, that would determine the angular velocity of the CubeSat. Gyroscopes are subject to a large drift, so they would not be functional at all times during normal operation. Instead, the gyroscopes could be used during an eclipse since the sun sensors would not be able to provide data.

4.1.2.1.4. Using Star Trackers for Normal Operation

Star trackers work by recognizing the pattern of the stars and comparing it to an onboard database to determine how the CubeSat is oriented. The price of the star tracker was over our budget, so it was eliminated from the concept selection stage.

4.1.2.2. Selection of Sensors for Normal Operation

From this evaluation, it was clear that we needed two reliable sensors to provide two linearly independent vectors to determine the orientation of the CubeSat. This was selected to be the sun sensors and magnetometer. However, since the sun sensor would be incapable of providing a valid reading during an eclipse, the gyroscope measurement would be used instead. Although the gyroscope would accumulate drift while in use, it would only be operating for a short period of time and therefore the accumulated drift would not significantly affect the system performance.

Using the same requirements listed in the previous section, we selected specific models of sensors for normal operation using Go/No-Go and Selection Matrices.

Table 5: Go/No-Go Matrix for Sensors During Normal Operation

	1	2	3	4	5	6	7	8	9	10
3-Axis Digital Compass IC HMC5883L Magnetometer	G	G	G	G	G	G	G	NG	M	NG
3-Axis Digital Magnetometer IC	G	G	G	G	G	G	G	M	G	G
Tronics GYPRO2300	NG	G	G	G	G	G	G	M	G	G
Precision Navigation and Pointing Gyroscope	G	G	G	G	G	G	G	M	G	G
Precision Angular Rate Sensor	NG	G	G	G	G	G	G	M	G	G
CubeSense Sun and Nadir Sensor	M	NG	G	G	G	G	G	G	M	G
NSS Sun Sensor	G	G	G	G	M	G	G	G	G	G
Nadir Sensor from MAI	G	M	G	G	M	G	G	G	NG	G

Table 6: Selection Matrix for Sensors During Normal Operation

	Weight	3-Axis Digital Compass IC HMC5883L Magnetometer	3-Axis Digital Magnetometer IC	Tronics GYPRO2300	Precision Navigation and Pointing Gyroscope	Precision Angular Rate Sensor	CubeSense Sun and Nadir Sensor	NSS Sun Sensor	Nadir Sensor from MAI
1	0.129	*0*	*0*	*-1*	*0*	*-1*	*-1*	*0*	*0*
2	0.158	*0*	*0*	*0*	*0*	*0*	*-1*	*0*	*-1*
3	0.079	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
4	0.100	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
5	0.187	*0*	*0*	*0*	*0*	*0*	*0*	*-1*	*-1*
6	0.025	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
7	0.037	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
8	0.058	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
9	0.216	*0*	*1*	*1*	*1*	*1*	*0*	*1*	*-1*
10	0.012	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
Total	***1***	***0***	***0.286***	***0.157***	***0.286***	***0.157***	***-0.217***	***0.099***	***-0.491***

From Tables 5 and 6, requirements detailed in Table 1 were used to select the best sensor of each type: gyroscope, magnetometer and sun sensor. The best sensors were found to be the Precision Navigation and Pointing Gyroscope, the 3-Axis Digital Magnetometer IC, and the NSS Sun Sensor respectively.

4.1.2.3. Possible Actuators for Normal Operation

Similar to the detumbling phase, both reaction wheels and magnetorquers were considered for normal operation. It is important to keep in mind that using one actuator for both modes of operation would be ideal.

Table 7: Go/No-Go Matrix for Actuators During Normal Operation

	1	2	3	4	5	6	7	8	9	10
Mai -400 Reaction Wheel	G	G	G	G	M	G	G	G	NG	G
ISIS Magnetorquer Board	NG	NG	G	G	M	G	G	G	NG	G
NCTR-Moo2 Magnetorquer Rod	G	M	G	G	G	G	G	G	G	G
Cube Torquer and Cube Coil Bundle	G	M	G	G	M	G	G	G	G	G

Table 8: Selection Matrix for Sensors in the Detumbling Phase

	Weight	CubeWheel Small	Mai-400 Reaction Wheel	ISIS Magnetorquer Board	NCTR-M002 Magnetorquer Rod	Cube Torquer and Cube Coil Bundle
1	0.129	0	0	-1	0	0
2	0.158	0	0	-1	-1	-1
3	0.079	0	0	0	0	0
4	0.100	0	0	0	0	0
5	0.187	0	-1	-1	0	-1
6	0.025	0	0	0	0	0
7	0.037	0	0	0	0	0
8	0.058	0	0	1	0	0
9	0.216	0	0	1	1	1
10	0.012	0	0	1	0	0
Total	**1**	**0**	**-0.187**	**-0.474**	**0.058**	**-0.129**

From Tables 7 and 8, it was clear that the NCTR-M002 Magnetorquer was the best option.

5. Engineering Analysis

5.1. Selection & Use of Engineering Tools/Techniques

5.1.1. MATLAB & Simulink

MATLAB was used to create a model of the designed ADCS. Using the Aerospace and Defence toolbox, and by modeling the selected sensors and actuators, accurate simulation results were obtained to demonstrate the performance of the ADCS and CubeSat in space.

5.1.2. Attitude Determination

Figure 2 shows the overall attitude determination system used to determine the attitude throughout the orbit of the CubeSat.

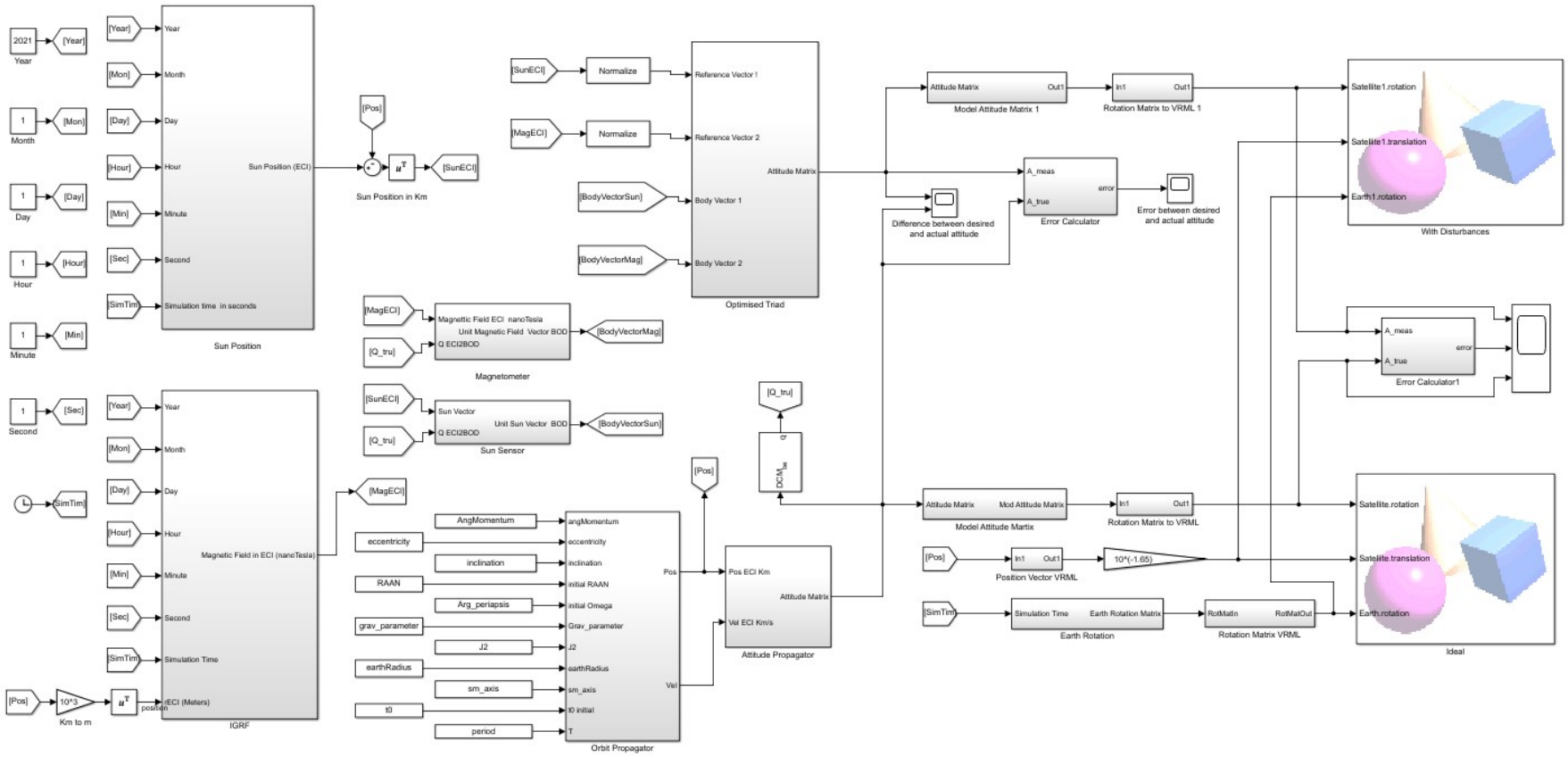

Figure 2: Overall Attitude Determination System in Simulink

5.1.3. Sun Sensor

The sun sensor is modeled in Simulink to accept two inputs (Figure 3): the position of the sun in Earth-Centered Inertial (ECI) coordinates, calculated using the sun position model (Section 5.1.7.1.) and a quaternion that will rotate from ECI coordinates to body coordinates. It outputs the position of the sun as seen by the CubeSat (sun vector body coordinates). In the second level of the sun sensor block (Figure 4), a random quaternion is calculated using the errorquatgen function in MATLAB (Appendix I). This is applied to the sun vector to add noise similar to what will be present in space. Next, the vector is rotated into body coordinates by applying the input quaternion, and normalized to give the output sun vector.

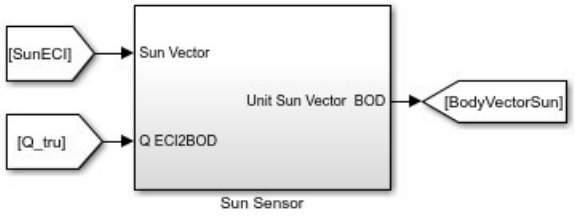

Figure 3: Level 1 of the Sun Sensor Model in Simulink

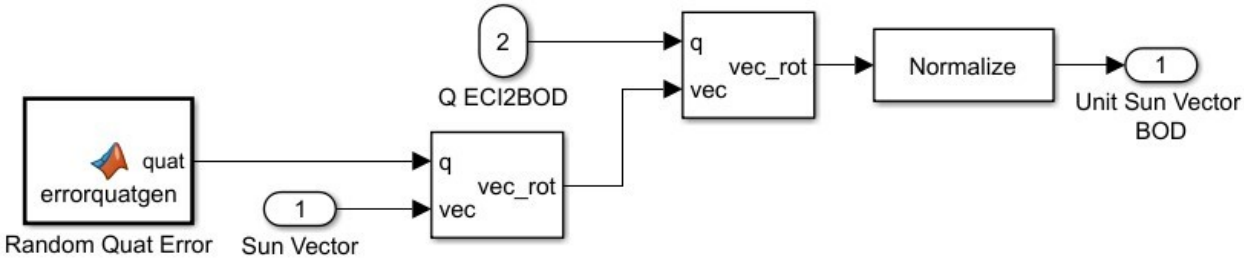

Figure 4: Level 2 of the Sun Sensor Model in Simulink

5.1.4. Magnetometer

The magnetometer has two inputs: the direction of the Earth's magnetic field in ECI coordinates (Section 5.1.7.2.) and a quaternion that will rotate from ECI coordinates to body coordinates. It has one output: the direction of the Earth's magnetic field as seen by the CubeSat (magnetic

vector in body coordinates as seen in Figure 5). In the second level of the sensor (Figure 6), the magnetic vector is rotated to body coordinates and normalized. In the third level of the sensor (Figure 7), noise is added to the sensor using the function NoiseGenerator and the resolution and output range of the sensor are added to accurately model the space performance of the sensor.

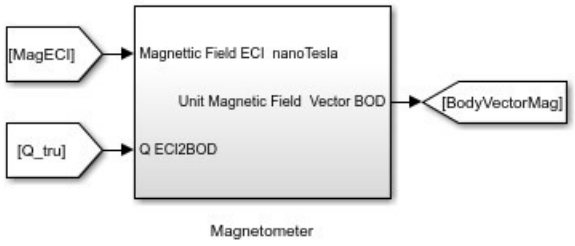

Figure 5: Level 1 of the Magnetometer Model in Simulink

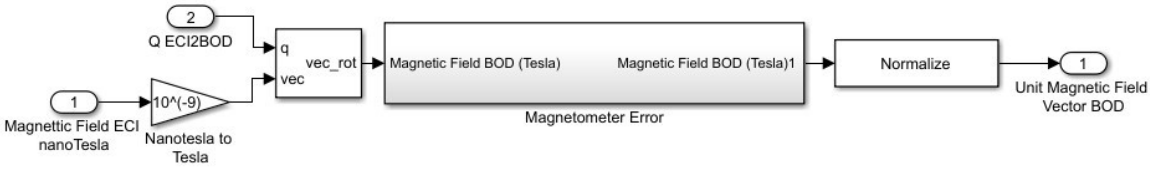

Figure 6: Level 2 of the Magnetometer Model in Simulink

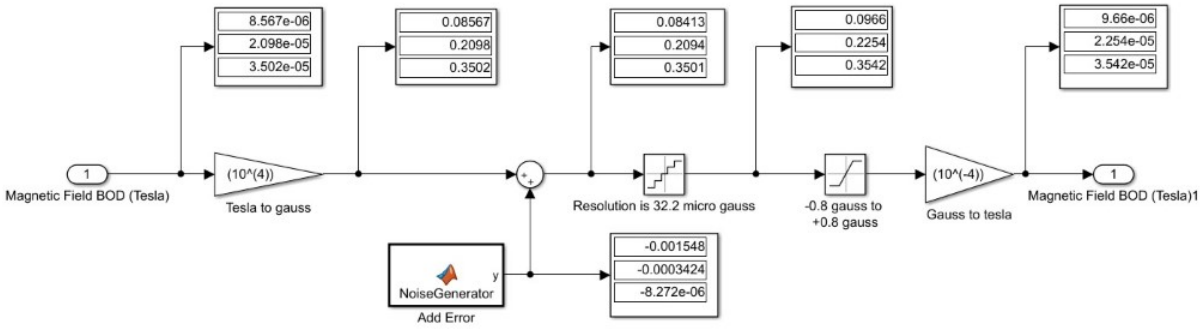

Figure 7: Level 3 of the Magnetometer Model in Simulink

5.1.5. Tri-Axel Attitude Determination (TRIAD)

In the first level of the optimized TRIAD, it takes in two reference vectors, two body vectors and two sigma values as error estimates, and consequently outputs the attitude matrix (Figure 8). In the second level of the optimized TRIAD (Figure 9), two TRIAD functions generate two attitude

matrices. One uses the sun sensor vector as an anchor and the other uses the magnetometer vector as an anchor. By doing this, the two attitude matrices become skewed towards their anchor. To remove this error, the two attitude matrices are passed into the optimized TRIAD function which applies a linear estimator and normalizes the result to calculate the actual attitude matrix.

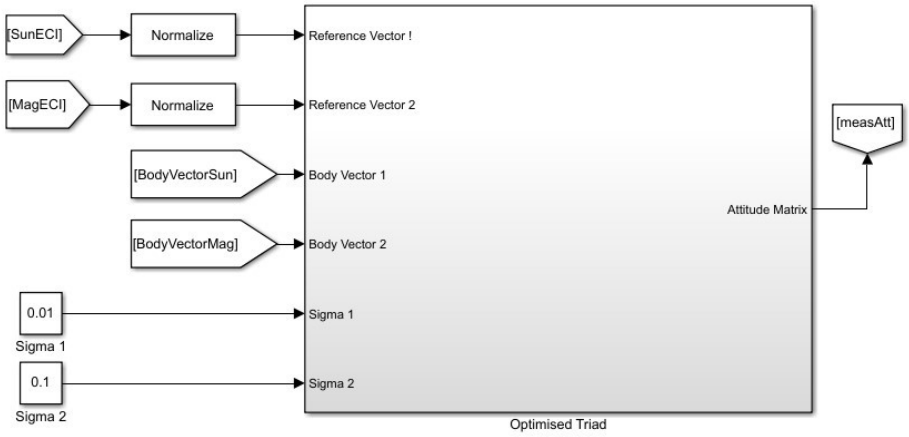

Figure 8: Level 1 of the TRIAD Model in Simulink

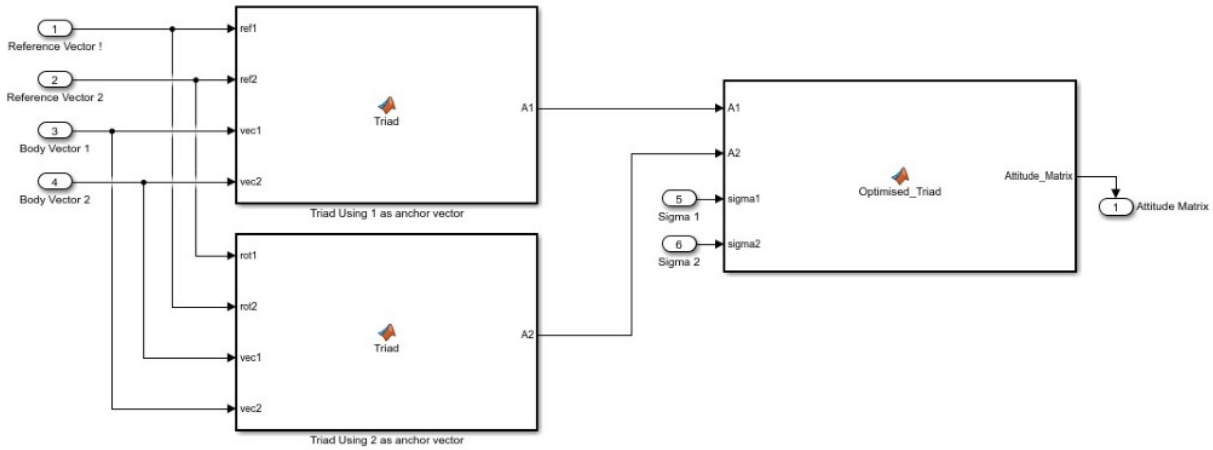

Figure 9: Level 2 of the TRIAD Model in Simulink

5.1.6. Attitude Propagator

The attitude propagator takes in a position vector and a velocity vector in ECI coordinates, and outputs an attitude matrix (Figure 10). It normalizes the two input vectors and passes them along with two reference vectors through a TRIAD to calculate the ideal attitude matrix (Figure 11).

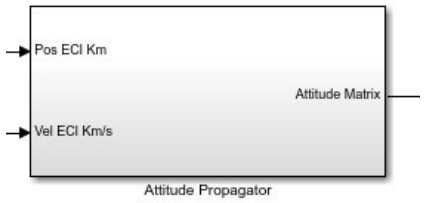

Figure 10: Level 1 of the Attitude Propagator

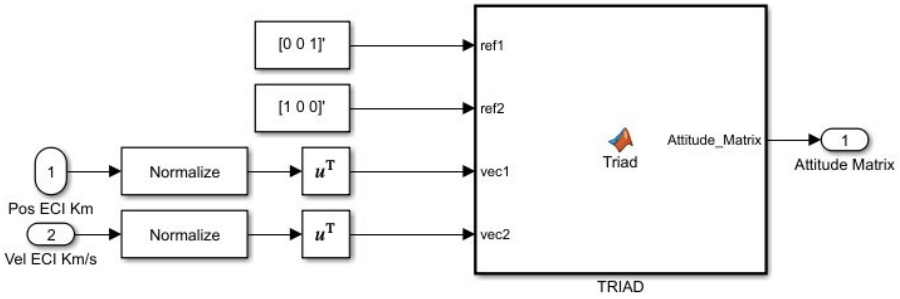

Figure 11: Level 2 of the Attitude Propagator

5.1.7. Helper Blocks

5.1.7.1. Sun Position

Figures 12 and 13 show the sun position block that calculates the position of the sun in ECI coordinates. It takes a date in calendar format and the simulation time as inputs and uses functions julian and fracjday to convert the input date from calendar format to Julian date format [9]. This is then used, along with the desired position of the satellite, to calculate the sun's position in ECI coordinates.

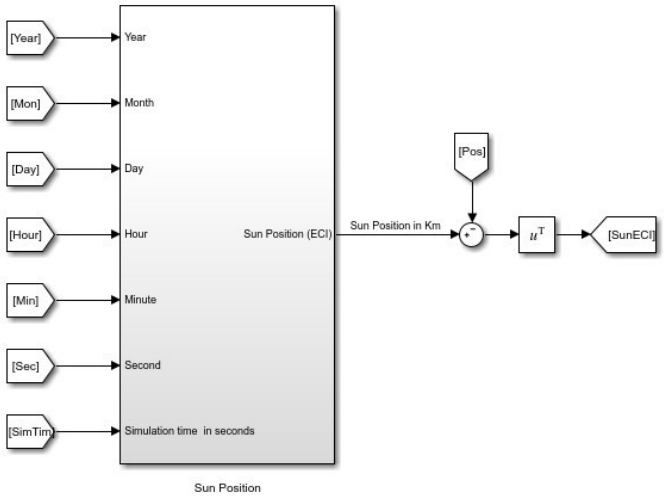

Figure 12: Level 1 of the Sun Position Block

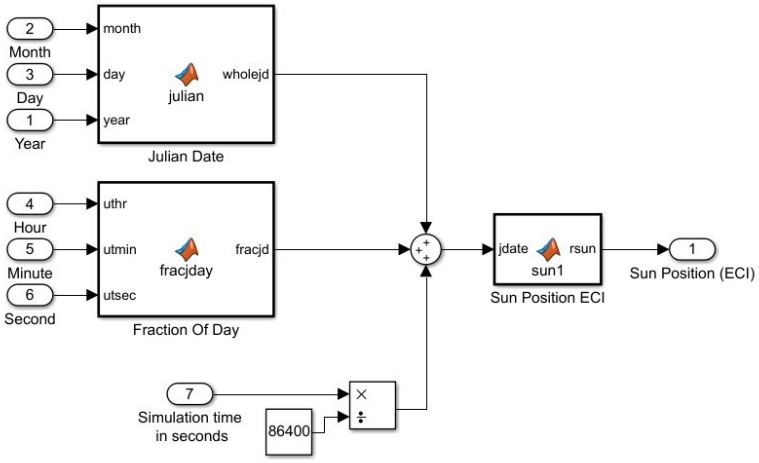

Figure 13: Level 2 of the Sun Position Block

5.1.7.2. International Geomagnetic Reference Field (IGRF)

Figures 14 and 15 show the IGRF model: it takes in the desired position in metres, along with a calendar format date and the simulation time. It outputs the magnetic field in ECI coordinates. The sub-block contains the functions Magfield_spherical, locst, rECI2ECEF and msph2inert. The Magfield_spherical function converts the magnetic field into local spherical coordinates [10]. The locst function takes in the date and the longitude and computes the local sidereal time [11]. The rECI2ECEF function converts from ECI to earth-centered, earth-fixed (ECEF)

coordinates [12]. Finally, the msph2inert converts local spherical coordinates into ECI coordinates.

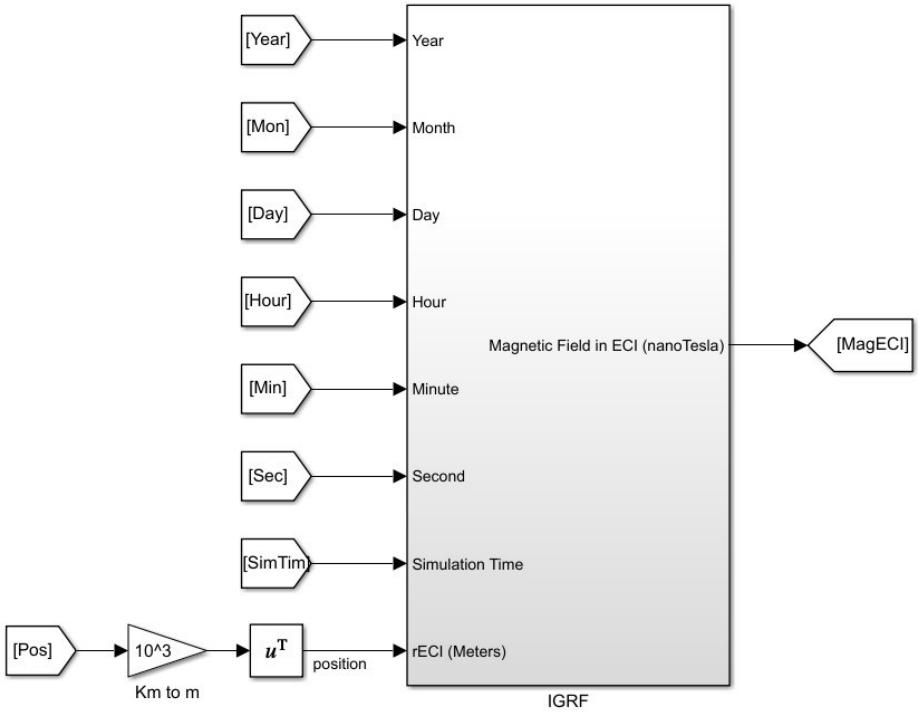

Figure 14: Level 1 of the IGRF Block

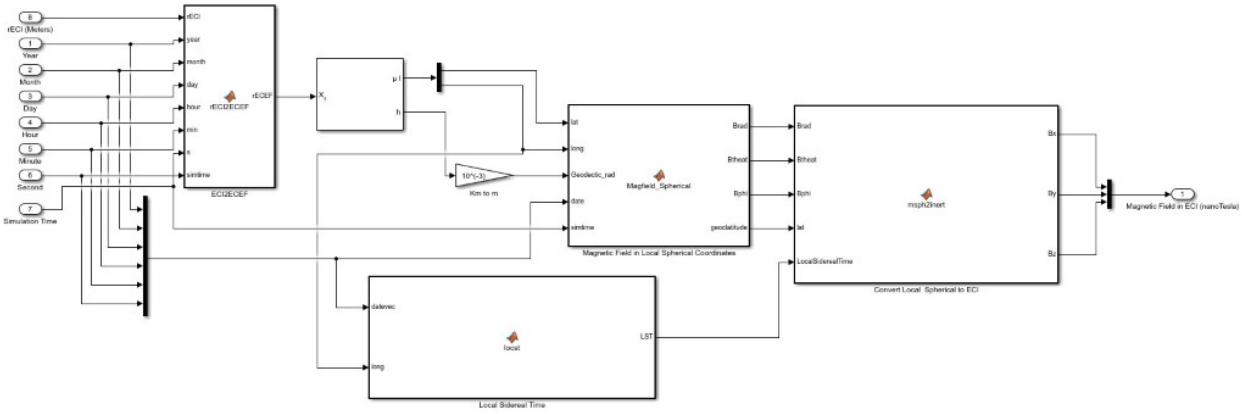

Figure 15: Level 2 of the IGRF Block

5.1.7.3. J2 Propagator

This block generates an ideal orbit of the CubeSat (Figures 16 and 17). This was provided by the orbital CubeSat team. The ideal orbit is required for testing purposes to allow us to generate the output of the sensors by adding noise to the orbit.

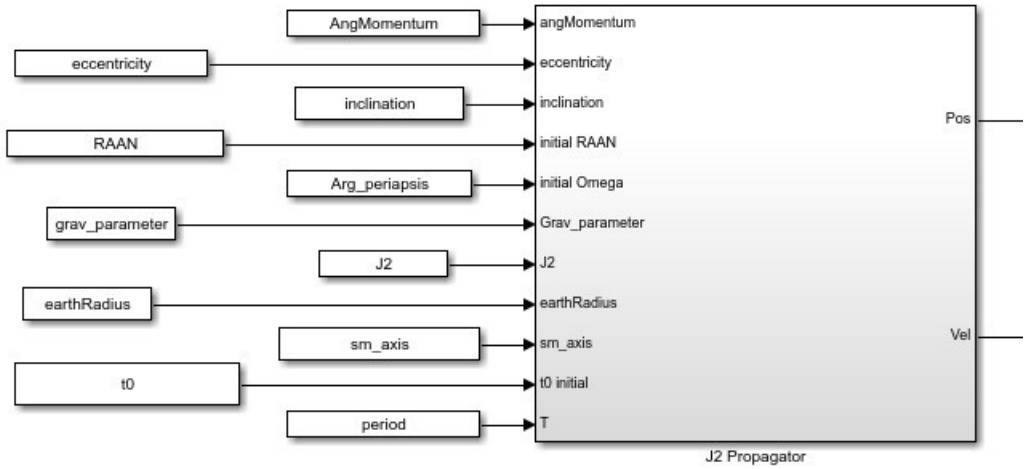

Figure 16: Level 1 of the J2 Propagator

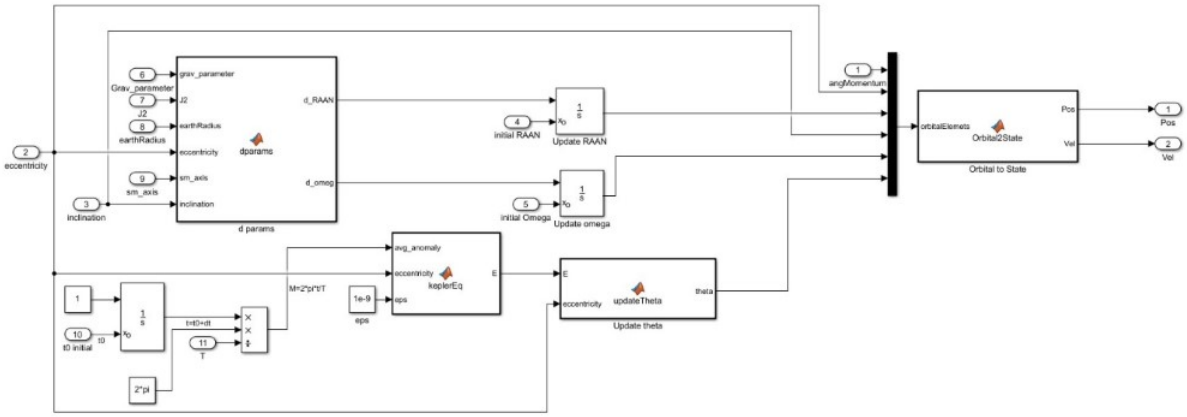

Figure 17: Level 2 of the J2 Propagator

5.1.3.1. Control Law (B-dot Bang Bang)

5.1.3. Attitude Control The B-dot Bang-Bang control law is used to control the attitude of the CubeSat. This commands a magnetic dipole, which is limited by the maximum amount of torque that the selected magnetorquer can produce, ensuring that the model accurately represents the

capabilities of the system. The control torque produced is proportional to the negative angular rate which allows us to oppose the angular rate and control the attitude of the CubeSat.

The decision to use B-dot control was based on its ability to oppose the angular rate of the system and the fact that B-dot control is commonly used in satellites with magnetorquer actuation. Specifically, B-dot Bang-Bang was selected because it can accurately model the torque that the selected magnetorquers will be capable of producing, meaning it can accurately determine the amount of time it will take to detumble the CubeSat. The MATLAB script written for the control law can be seen in Appendix I.

5.1.4. SolidWorks (Selection & Use of Engineering Tool)

SolidWorks will be used to design reaction wheels for the prototype that generate torque that is equivalent to the dipole moment generated by the selected magnetorquers to ensure our prototype accurately represents the designed ADCS. This will require a combination of designing the parts in SolidWorks and using the SolidWorks Analysis tool to determine the torque produced by the reaction wheels.

There are certain design considerations that must be made. First the motors which will be used to spin the flywheels must be able to provide speeds of at least 2000 RPM. To stabilize the system, each flywheel must be kept spinning at a non-zero value, ideally at a value above 500 RPM. Having non-zero flywheel speeds means that the prototype will be able to resist small disturbance torques. Factors that will determine whether or not these speeds are possible are the specifications of the motors and the inertia of the flywheel.

The weight and change in speed of the reaction wheels will determine the resultant torque that is applied to the prototype. The accuracy of the system was more important than a fast system, so a decision was made to use flywheels weighing 67.6 grams. Once the inertia of the flywheel was

determined, the motor which would turn it had to be selected. Using the criteria regarding the minimum acceptable speed and considering only motors which would fit the financial budget and mass budget, the PAN14EE12AA1 standard DC motor was selected. The important characteristics of this motor was the stall torque, which is 4.9N-mm, and the rated 12V no-load speed, which is 12850 RPM.

To verify that this motor would be sufficient for the prototype, it had to be ensured that the motors could provide changes in speed of at least 2000 RPM within one second. This was determined using SolidWorks Motion Analysis, the results of which are shown in Section 5.2.2. From these results, the maximum change in angular speed that can be achieved in one second is 20000deg/second, which is approximately 3500 RPM, well above the minimum requirement. This is the maximum angular acceleration possible as it will require a motor torque of 4.5N-mm, just under the 4.9N-mm stall torque.

5.2. Simulation Results

5.2.1. Attitude Determination Simulations

5.2.1.1. Actual Attitude Compared to Desired Attitude

The actual and desired attitudes are given in Figures 18 to 20. It is shown that the actual attitude follows the desired attitudes very well with only 0.095% error at most. The error between all desired and actual attitude signals is given in Figure 21. The results from these plots imply that the selected sensors will be able to accurately determine the attitude of the CubeSat in space.

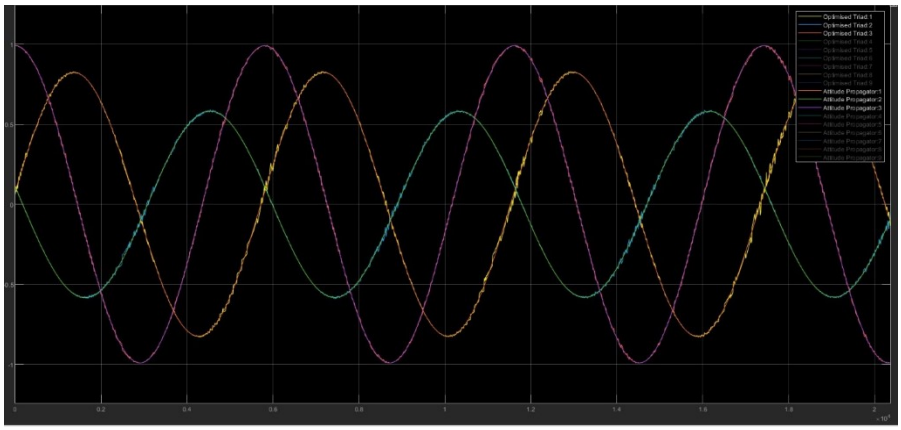

Figure 18: Desired Attitude Compared to Actual Attitude in x-x, x-y and x-z Axes

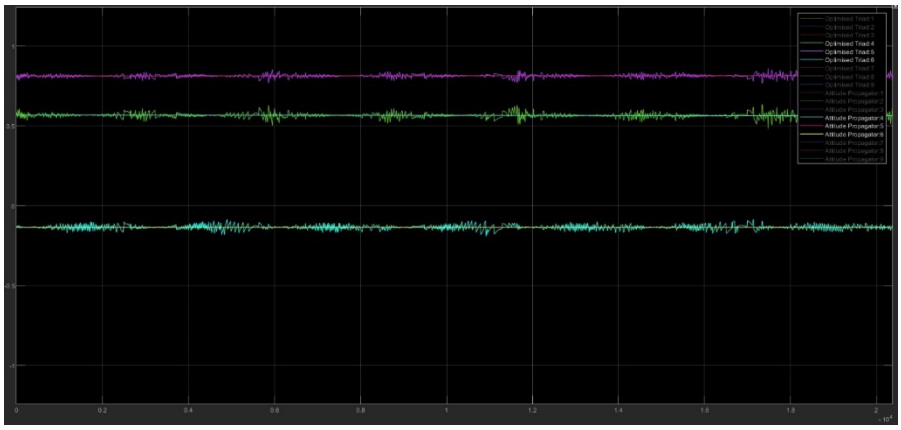

Figure 19: Desired Attitude Compared to Actual Attitude in y-x, y-y and y-z Axes

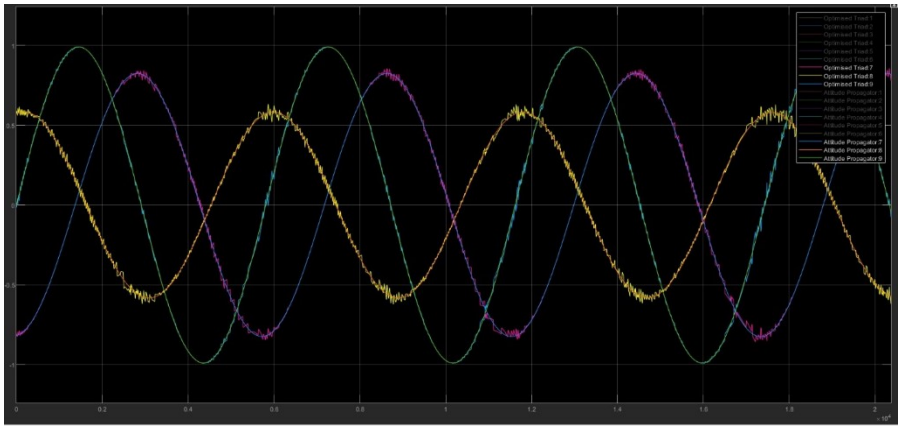

Figure 20: Desired Attitude Compared to Actual Attitude in z-x, z-y and z-z Axes

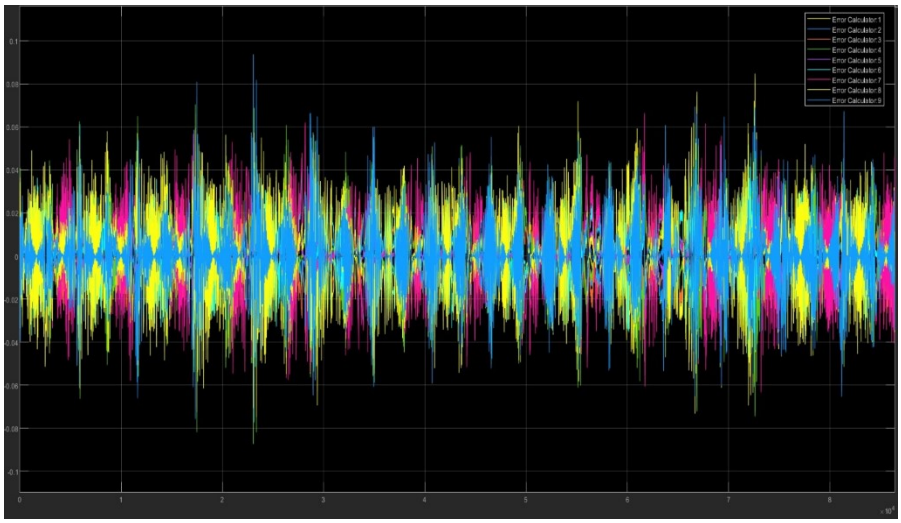

Figure 21: Error Between Desired Attitude and Actual Attitude for Six Attitude Comparisons

5.2.1.2. Attitude Control

After running the MATLAB script for B-dot Bang-Bang control, the simulations shown in Figure 22 was obtained. These simulations show that the ADCS will be able to detumble the CubeSat in four orbits if it is released at a tumbling rate of five degrees per second in each axis. This rate was chosen because it is the maximum rate provided by the CSA. Four orbits is approximately eight hours and is a reasonable amount of time for the CubeSat to detumble.

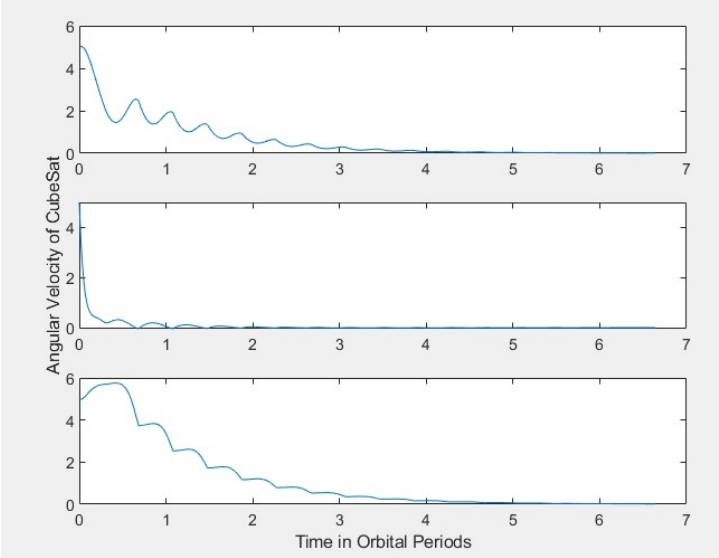

Figure 22: Angular Rate of CubeSat Detumbling with an Initial Rate of 5 Degrees

5.2.2. SolidWorks Simulations

Figures 23 and 24 show the angular velocity and acceleration that would suffice for actuating the prototype. Figure 25 shows the torque which the motor would experience under the angular velocity and acceleration conditions specified by Figures 24 and 25. This figure shows that the motors will not reach their stall torques when loaded with the flywheel. This confirms that the motor and flywheel combination will be suitable for actuating the prototype.

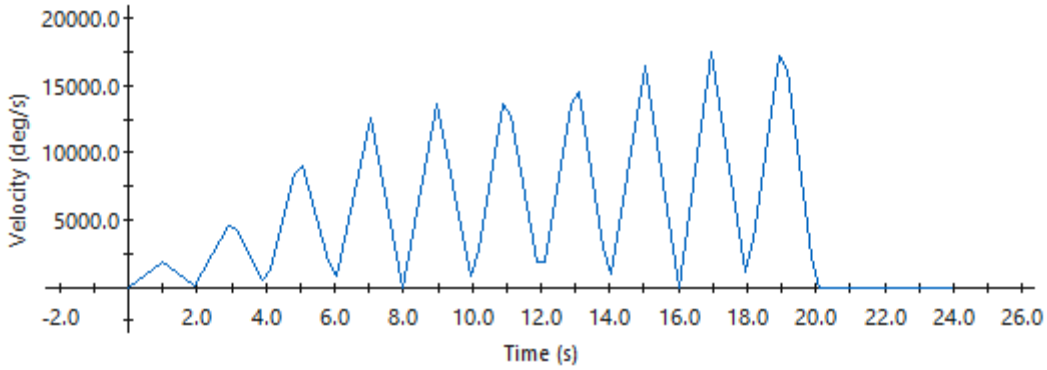

Figure 23: Angular Velocity vs. Time for Reaction Wheel Load Simulation

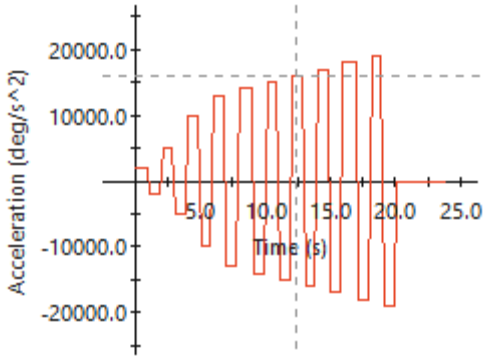

Figure 24: Angular Acceleration vs. Time for Reaction Wheel Load Simulation

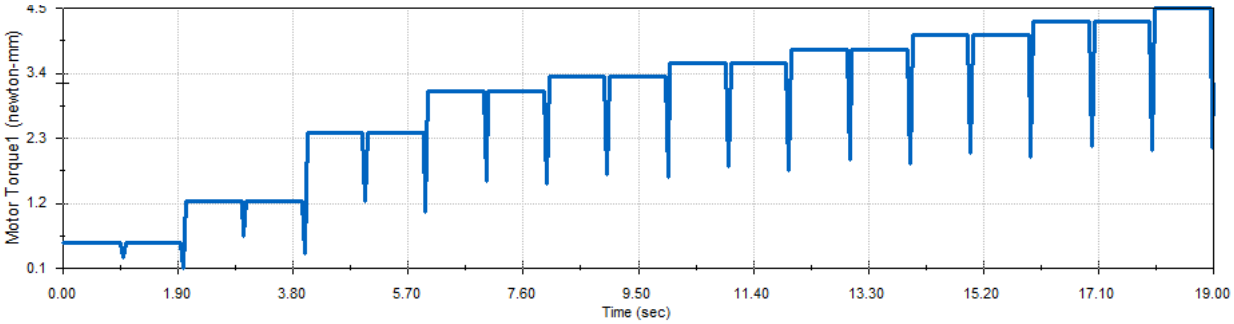

Figure 25: Realized Motor Torque vs. Time for Reaction Wheel Load Simulation

6. Refinement of Design

The preliminary selection of sensors was sun sensors and a magnetometer. It was ensured that the chosen sensors met the required functional, physical, and environmental properties needed. Six sun sensors are needed to cover each face of the CubeSat to ensure that there would be a sun sensor that is providing valuable data no matter which way the CubeSat was oriented. The magnetometer is 3-axis, which means it provides data for all three dimensions. Initially reaction wheels were considered as the actuator for the CubeSat. However, it was determined that the cost of reaction wheels did not fit the constraints of the design. As a result, the actuator chosen for the design are magnetorquers. Three magnetorquers are needed in order to control the CubeSat along all three axes. The magnetorquers interact with the gradient of the Earth's magnetic field to accurately orient the CubeSat. When the CubeSat enters eclipse, the sun sensors no longer provide data necessary for the CubeSat to be controlled. Therefore, the original design was iterated to incorporate a 3-axis gyroscope used to integrate the last set of data provided by the sun sensor before the CubeSat enters an eclipse. The gyroscope acquires drift over time so it can only be used for a short period of time, but since it is only needed when the CubeSat is in eclipse, the amount of drift that it acquires is not significant enough to affect the CubeSat.

7. Prototype Development

7.1. Prototype Structure

The prototype will consist of two parts: the testbed and the ADCS.

7.2. Testbed

The purpose of the testbed is to provide an environment similar to a space environment that allows for rotation in three axes with only small reactional forces. To do this, the testbed will use a spherical air bearing that allows for full rotation in one axis and partial rotation in the other two axes. A platform will be mounted on top of the air bearing that will hold the ADCS, and the base of the air bearing will have an attached air supply to provide air to the bearing. Figure 26 shows the proposed design modeled in SolidWorks.

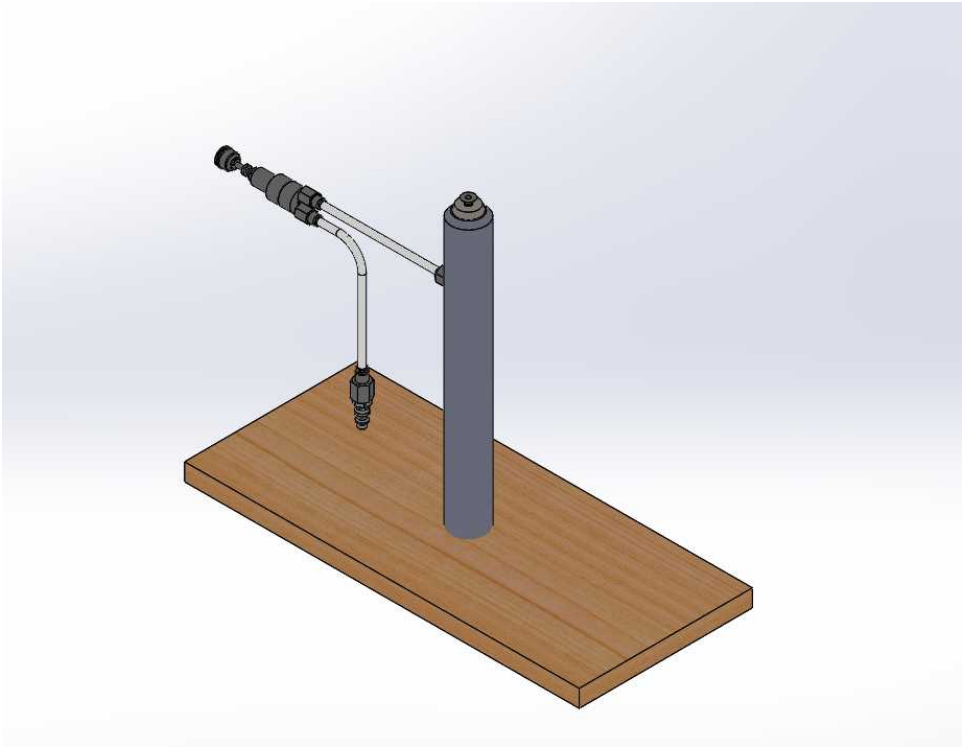

Figure 26: Model of the testbed

The materials required to build the testbed are given in Table 9.

Table 9: Bill of Materials for Prototype

Name	Description	Supplier	Supplier Part	Order Quantity
1’ Black Delrin ® Acetal Resin Rod 1-1/2” Diameter	Resin rod	McMaster-Carr	8497K313	1
Highly Corrosion-Resistant Stainless-Steel Ball 1” Diameter	Stainless steel ball	McMaster-Carr	96415K82	1
Acrylic sheet	Acrylic sheet	Plaskolite	1X09241A	1
Crack-Resist Polypropylene Semi-Clear Tubing	Clear tubing	McMaster-Carr	1979T2	1
Panel-Mount Compressed Air Regulator	Air regulator	McMaster-Carr	41795K3	1
Black-Oxide Alloy Steel Hex Drive Flat Head Screw 1”-8 Thread Size 2-1/2” Long	Flat head screw (1”-8 thread size, 2-1/2” long)	McMaster-Carr	91253A914	6
Black-Oxide Alloy Steel Hex Drive Flat Head Screw ¼”-20 Thread Size, ½” Long	Flat head screw (1/4”-20 thread size, 1/2” long)	McMaster-Carr	91253A537	1
Wooden board	Wooden board	Irving	228288	1
Push-to-Connect Tube Fitting for Air	Tube fitting to air	McMaster-Carr	5779K131	1
Reducing Adapter, 1/8 NPT Female x 1/16 NPT Male	Reducing adapter	McMaster-Carr	2684K21	2
Straight Adapter, for ¼” Tube OD x 1/8 NPT Male	Straight adapter	McMaster-Carr	5779K108	2
Industrial-Shape Hose Coupling Size ¼, Zinc-Plated Steel Ply, ¼ NPTF Male End	Industrial-shape hose	McMaster-Carr	6534K46	1
Push-to-Connect Tube Fitting for Air with Shut-Off, Adapter, for ¼” Tube OD, ¼ NPT Male	Tube fitting for air with shut-off adapter	McMaster-Carr	5779K417	1
PTFE Tape	PTFE tape	Dixon Valve & Coupling	TTB75	1
18.9L Max 125 PSI Portable Air Tank	Air Compressor	Speedway	7296	1

Before these parts can be assembled into the testbed, some modifications are required to be made to the resin rod and the stainless-steel ball. The resin rod requires two holes to be drilled to act as a path for air to enter the bearing, a partial sphere shape to be removed using a lathe to hold the

ball, and two threaded holes to be created: one to attach the rod to a wooden base and the other to attach an air supply. A detailed drawing of the modified Delrin rod can be seen in Figure 27.

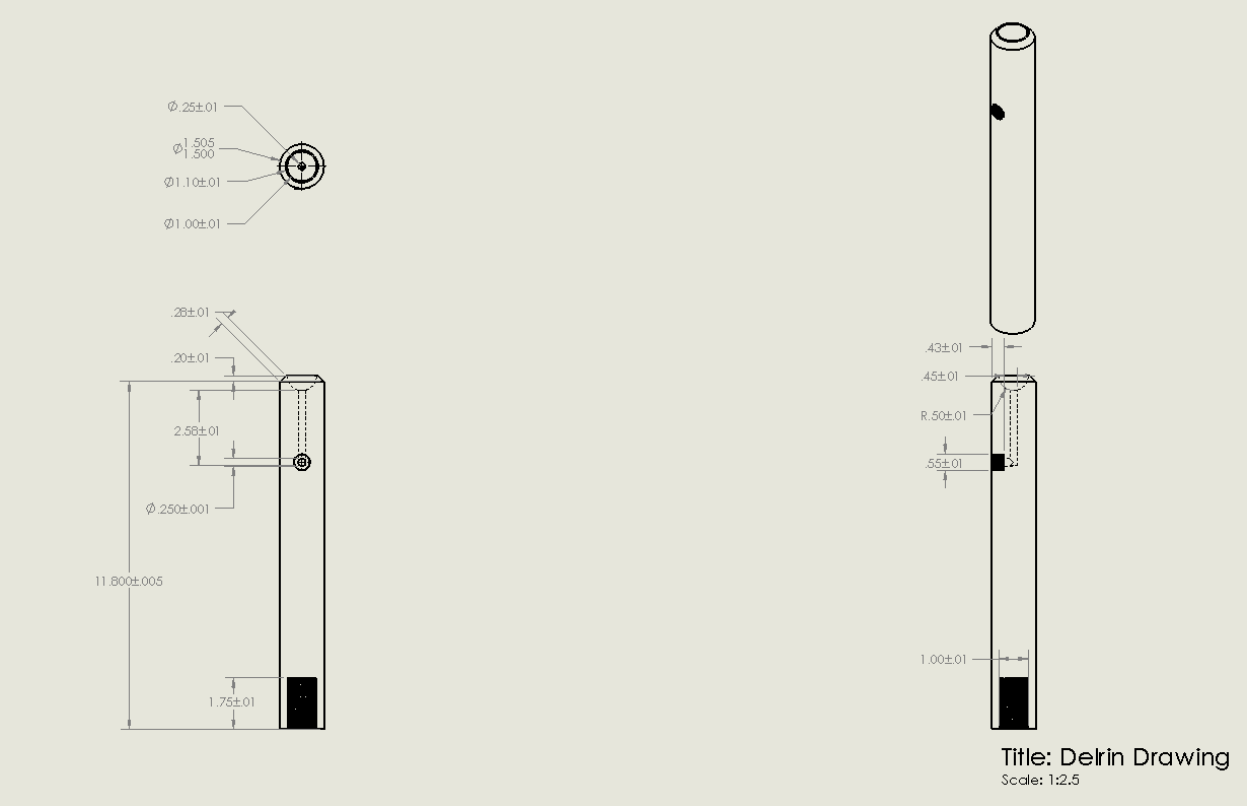

Figure 27: Detailed Drawing of Delrin Rod

The stainless-steel ball must be modified so that the acrylic sheet can be attached to it. This involves flattening one side of the ball using a lathe, and then creating a threaded hole to allow for a screw to secure the ball to the acrylic platform. A detailed drawing of the modifications to the ball can be seen in Figure 28.

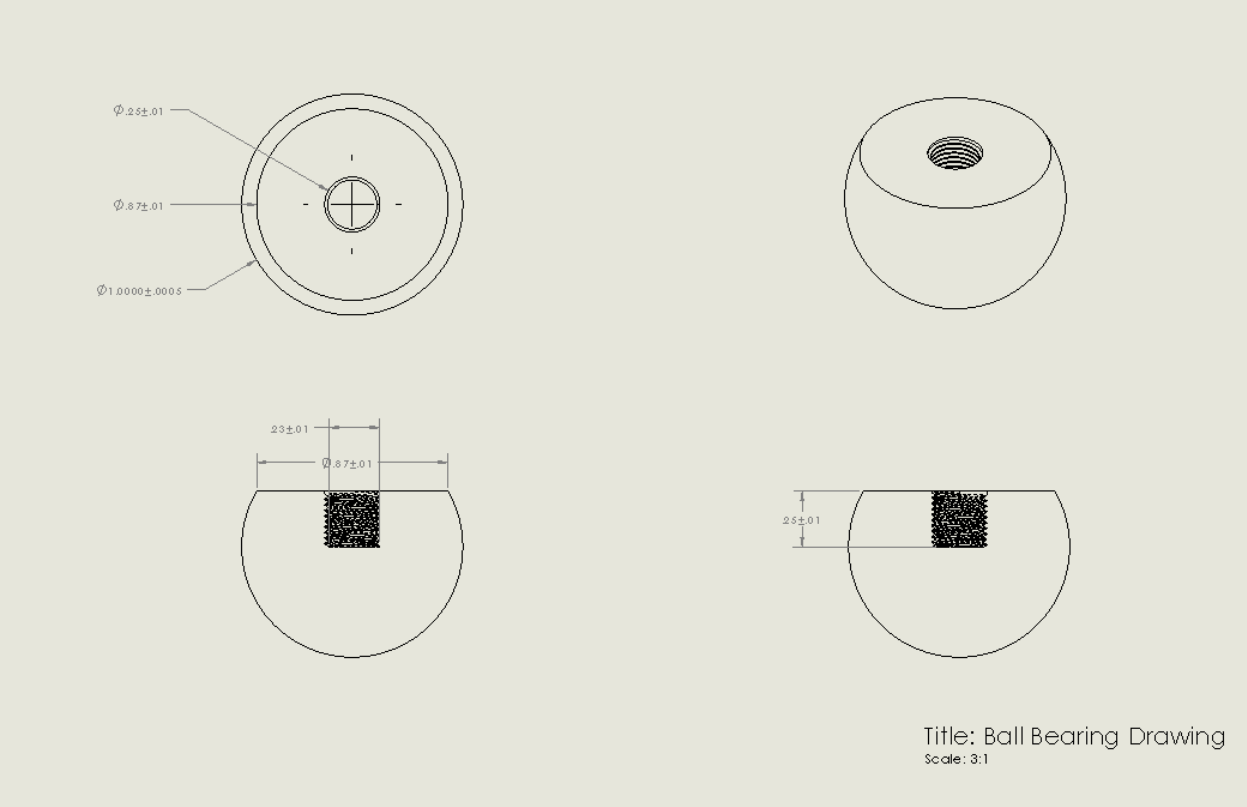

Figure 28: Detailed Drawing of Ball Modifications

A detailed SolidWorks model of the testbed can be found in Figure 26.

The procedure to assemble the testbed is given as:

1. Attach the Delrin rod to the wooden base using a 1'' screw.
2. Attach a 10cm x 20cm x 1cm acrylic sheet to the levelled part of the steel ball using a ¼'' screw.
3. Place the steel ball in the bowl at the top of the Delrin with the levelled part facing upward.
4. Attach a ¼'' Tube OD X 1.4 NPT Male Push-to-Connect Tube Fitting for Air to the Delrin rod at the air inlet.
5. Attach one of each of the ¼'' plastic tubing ends to a ¼'' Tube OD x 1/8 NPT Male Push-to-Connect Tube Fitting for Air each.
6. Attach one of the plastic tubing free ends to the Push-to-Connect Tube Fitting for Air at the Delrin's air inlet.
7. Attach the other plastic tubing free end to a Size ¼'' x ¼ NPTF Male End Industrial-Shape Hose Coupling.
8. Apply sealant tape to all areas where the plastic tubing is connected to a Push-to-Connect Tube Fitting for Air or Industrial-Shape Hose Coupling to ensure no air will escape.
9. Attach the floating Push-to-Connect Tube Fitting for Air adapters to a 1/8 NPT Female x 1/16 NPT Male Miniature Medium-Pressure Stainless Steel Threaded Pipe Fitting each.
10. Attach both Miniature Medium-Pressure Stainless Steel Threaded Pipe Fittings to a 1/16 NPT Female Panel-Mount Compressed Air Regulator.
11. Attach the Industrial-Shape Hose Coupling to an air supply.

7.3. ADCS System

The ADCS prototype will demonstrate the ability to determine the attitude of the CubeSat and actuate the CubeSat to a desired position. Due to the limited budget currently available for the Western CubeSat project, the prototype will not be able to use space-certified magnetorquers or sun sensors. Without the sun sensor to provide a second reference vector, the prototype will use the gyroscope to determine its attitude. This allows for demonstration of the normal operation during eclipse. Since magnetorquers are the only form of actuation on the CubeSat, reaction wheels are being used on the prototype to provide torque that is equivalent to the dipole moment produce by the simulated magnetorquers. By doing this, the prototype will effectively demonstrate how the ADCS will be actuated.

The ADCS prototype consists of the magnetometer, gyroscope, three reaction wheels, batteries and the processing board. Each component is attached to an acrylic sheet representative of the true size of the ADCS. The materials required to build the ADCS are given in Table 10.

Table 10: Bill of Materials for ADCS

Name	Description	Supplier	Supplier Part	Order Quantity
MPU 6050 Gyroscope and Accelerometer Board	Three-Axis Gyroscope	Sunfounder	ARD001	1
Arduino Mega 2560 Rev3	Microprocessor	Arduino	A000067	1
Energizer MAX AA Batteries	Battery	Energizer	E92BP4	6
Wire Wrap	Wires	Adafruit Industries LLC	1446	1
Standard Motor 12850 RPM 12VDC	Motor	NMB Technologies Corporation	PAN14EE12AA1	3
Low-Carbon Steel Disc	Reaction wheels	McMaster-Carr	7786T12	3

The prototype ADCS system is shown in Figure 29.

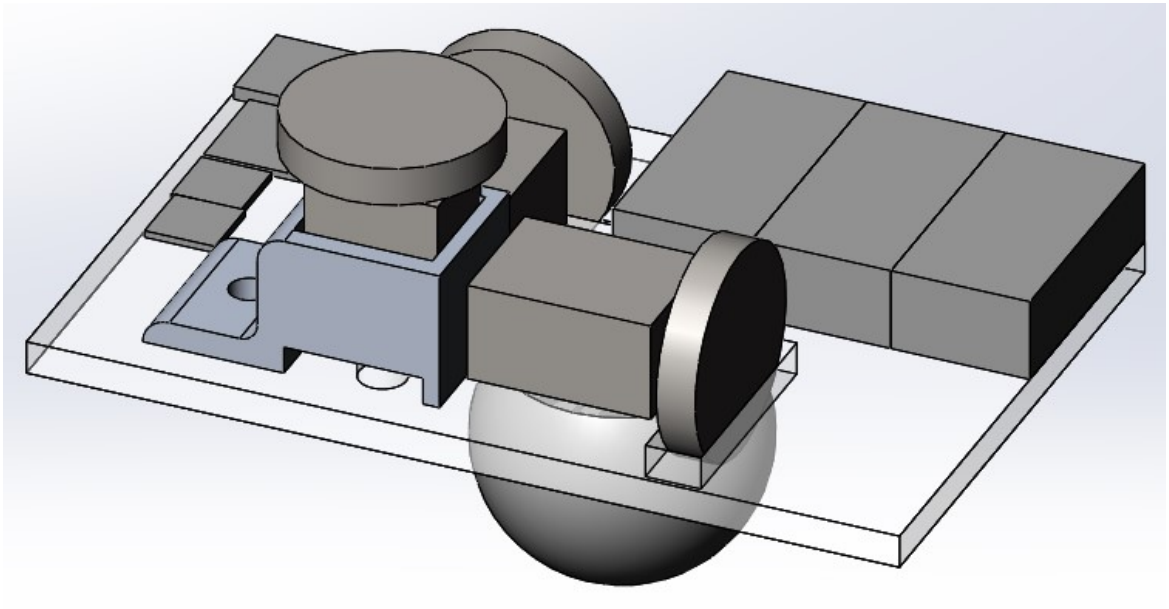

Figure 29: Prototype of the ADCS

The procedure for assembling the prototype ADCS is given as follows:

1. Press fit the reaction wheels onto the motor shafts.
2. Using adhesive strips, secure two motors with attached reaction wheels to the acrylic sheet.
3. Place the third motor and reaction wheel in the 3D printed casing and press fit the motor onto the acrylic.
4. Secure the 3D casing to the acrylic using a screw and a bolt.
5. Secure the gyroscope using adhesive strips.
6. Secure the microcontroller and the motor drivers to the acrylic using adhesive strips.
7. Connect the gyroscope, microcontroller, motor drivers and motors as shown in Figures 30 and 31.

8. Place the batteries on the acrylic and secure them using tape.

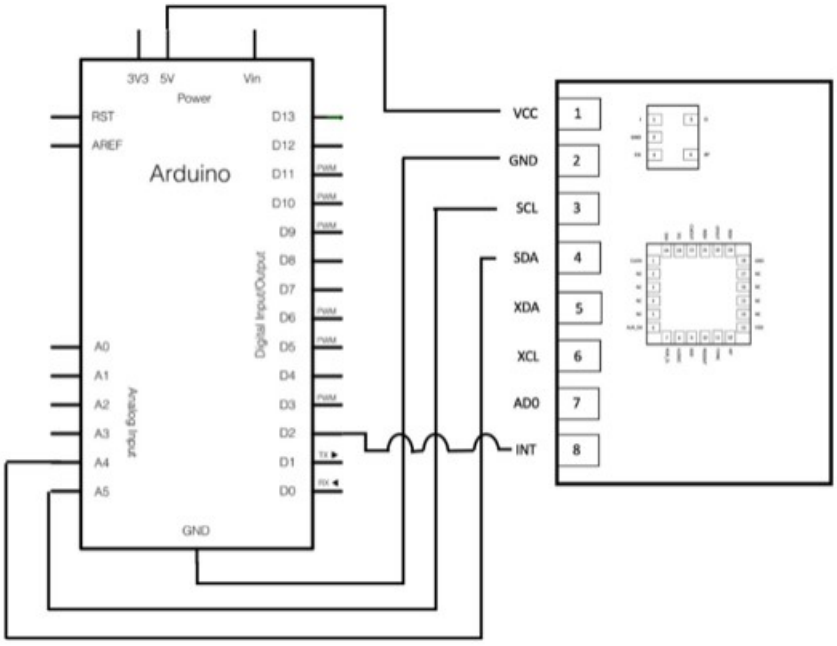

Figure 30: Gyroscope Connected to the Microcontroller

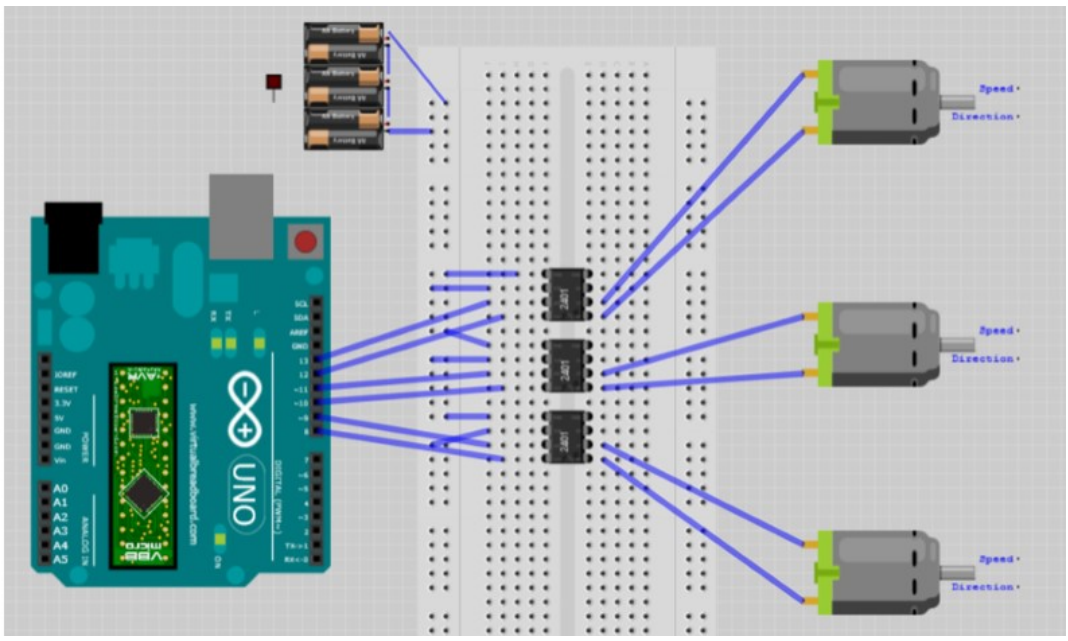

Figure 31: Motor Drivers and Motors Connected to Microcontroller and Power Supply

8.4. Prototype Cost

Table 11 contains the overall prototype budget.

Table 11: CubeSat ADCS Prototype Budget

Item	Cost
Engineering Hours ($80/hr)	$76,800
MATLAB Education License [13]	$665.14
Fundamentals of Spacecraft ADCS Textbook [14]	$107
Materials and Components	$299.62
Laser Cutter [15]	$15.00
Air Compressor [16]	$68.95
University Machine Services Fees ($64.35/hr) [17]	$257.40
Simulink Education Licence [13]	$665.14
Total	$103,906.80

Table 12 contains the Real-Cost Budget. This budget focuses on the actual cost that will be incurred while creating this product.

Table 12: CubeSat ADCS Real-Cost Budget

Item	Quantity	Order Status	Cost
MLX90393 - Magnetic, Magnetometer Sensor Evaluation Board [18]	1	Received	$29.77
MPU6050 Gyroscope and Accelerometer board [19]	1	Received	$5.00
1' Black Delrin Acetal Resin Rod 1-1/2" Diameter [20]	1	Received	$5.13
Highly Corrosion-Resistant 316 Stainless Steel Ball 1" Diameter [21]	1	Received	$19.43
Acrylic sheet [22]	1	Received	$3.28
Crack-Resist Polypropylene Semi-Clear Tubing [23]	1	Received	$7.20
STANDARD MOTOR 12850 RPM 12VDC [24]	3	Received	$21.12
Microprocessor [25]	1	Received	$19.80
Battery [26]	6	Received	$8.15
Panel-Mount Compressed Air Regulator [27]	1	Received	$52.22
Black-Oxide Alloy Steel Hex Drive Flat Head Screw 1"-8 Thread Size, 2-1/2" Long [28]	1	Received	$15.21
Black-Oxide Alloy Steel Hex Drive Flat Head Screw 1/4"-20 Thread Size, 1/2" Long [29]	1	Received	$0.22
Wooden board [30]	1	Received	$12.98
Push-to-Connect Tube Fitting for Air [31]	1	Received	$5.85
Reducing Adapter, 1/8 NPT Female x 1/16 NPT Male [32]	2	Received	$19.86
Straight Adapter, for 1/4" Tube OD x 1/8 NPT Male [33]	2	Received	$6.32
Industrial-Shape Hose Coupling Size 1/4, Zinc-Plated Steel Plug, 1/4 NPTF Male End [34]	1	Received	$2.36
Push-to-Connect Tube Fitting for Air with Shut-Off, Adapter, for 1/4" Tube OD, 1/4 NPT Male [35]	1	Received	$15.53
PTFE Tape [36]	1	Received	$0.05
Low-Carbon Steel Disc [37]	3	Received	$10.32
Total Cost			**$299.62**

8. Testing & Evaluation

The purpose of the prototype was to demonstrate both the ability to measure, determine and control the attitude of the spacecraft. To determine the attitude, the prototype read data from the on-board sensors and based on this, determined its current attitude. To control the attitude, the prototype utilized an actuation technique which would be viable in space.

The prototype was able to successfully determine its attitude at any moment. This was tested by outputting a visual display of the current prototype attitude as determined by the on-board sensors, which was constantly being updated. In this case, the visual display illustrated the measured attitude of the prototype while the true attitude of the prototype was manually manipulated by simply rotating the prototype in three dimensions. The results of this test showed that for any change in the true attitude of the prototype, the visual display followed the true attitude closely. This confirmed that our method for determining the attitude of the spacecraft was valid. The code used to measure and output the attitude of the prototype to a visual display is provided in Appendix II.

9. Modifications & Improvements

9.1. Modifications for Prototype

The prototype that was built aimed to validate the working principle behind the ADCS, specifically the attitude determination and control algorithm. There were some limitations to what could be done given that a space environment could not be replicated on Earth. Instead, a near-frictionless testbed was developed by milling out a hole in the tube and passing air through it where the platform would sit.

Initially, the prototype had two levels, one for the Arduino MEGA, and another for the reaction wheels (Figure 32). However, this iteration resulted in a prototype that was much heavier than expected and the torque generated by the reaction wheels was insufficient in rotating the platform. Thus, the prototype was changed to one level, with the microcontroller replaced by an Arduino Micro, and the reaction wheels placed on the same level. This resulted in a lighter platform, reducing the moment created by gravity, allowing for sufficient torque from the reaction wheels to rotate the platform.

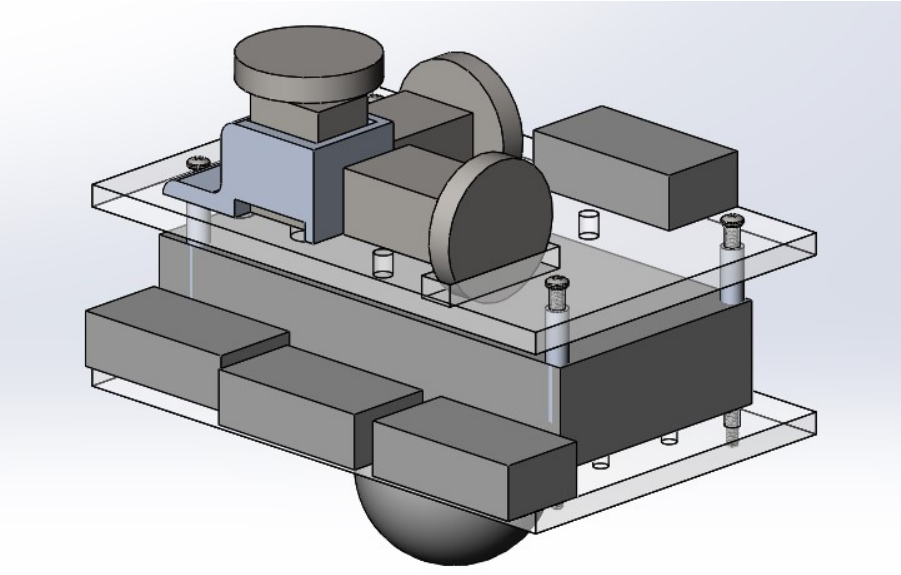

Figure 32: Initial Design of the ADCS prototype

Since the centre of mass was not in the middle of the platform, an extra casing had to be added around the top of the tube to ensure that the platform would not tip over. This resulted in increased friction to the otherwise frictionless testbed, which also led to issues with the platform rotating.

In the future, a different design for the testbed could be used instead. If the testbed were larger and had a large hole milled out in it, and the platform rested on a larger ball, roughly the size of the platform, the platform would be able to balance on the tube better.

The weight of the motors could not be overcome by the torque generated by the motors which caused the whole platform to have issues spinning. This issue was increased by the fact that the reaction wheels were not placed on the centre of mass axes. Since they were not placed on the centre of mass axes, the platform had an even harder time turning. This design flaw is important to keep into consideration during the actual design of the ADCS as it demonstrates the necessity for the magnetorquers to be placed along the centre of mass axes aboard the CubeSat.

9.2. Modifications for Design

As discussed in the previous section, through our prototype, the importance of keeping the magnetorquers along the centre of mass axes was proved to be vital in ensuring effective attitude control. The fact that the prototype did not rotate effectively demonstrates that the centre of mass of the CubeSat must be kept as close to the centre of the CubeSat as possible. This is important in the actual development of the CubeSat as there must be very strict control over where the centre of mass is placed to ensure mission success. This will allow the CubeSat to rotate effectively using the ADCS.

10. Conclusion

Major milestones achieved by the group include the development of the prototype. The testbed was fully constructed, with a prototype that aimed to demonstrate the attitude determination and control algorithm. After building the prototype, there were some issues that needed to be addressed as detailed in Section 9.1. However, the overall design of the attitude determination and control algorithm was validated through this prototype.

The attitude determination and control system has been fully modelled in Simulink, with the NCSS-SA05 as the sun sensors, BM1422AGMV 3-Axis Digital Magnetometer IC as the magnetometer, the CRM 400046-0300 as the gyroscope and the NCTR-M002 as the magnetorquers. Using these sensors and actuators, the design of the system operating in space was validated through Simulink. The magnetorquers were able to effectively detumble the CubeSat within four orbits, the sensors were able to effectively determine the current attitude and the magnetorquers were also proven to be effective for controlling the attitude of the CubeSat.

11. Recommendations

As addressed in the discussion, one of the largest problems the group has encountered is in effectively testing the ability of the prototype to actuate itself. There is a limited range for which the prototype opposes the force of gravity, and the analysis suggests that these problems were due to the prototype being too heavy. These issues result in the prototype requiring the torque provided by the reactions wheel to be far greater than is possible with the current design. The group has come up with the following recommendations to alleviate this problem.

First, the design of the reaction wheels is not ideal. The reaction wheels should be designed to maximize the inertia while minimizing the mass. A wheel and spoke type of design for the wheel would better optimize this relationship. This would reduce the overall weight of the prototype which is beneficial but does not constitute a complete solution on its own.

An additional step that should be taken is iterating the testbed such that the force of gravity on the centre of mass of the prototype is directly opposed by the upwards force provided by the air bearing regardless of the orientation of the prototype in the testbed. If the prototype is at a 60° angle out of the horizontal plane, the centre of mass should still be over-top of the air bearing. This would help negate the effect the gravity when attempting to test the prototype. A potential design could involves increasing the size of the hemispherical air bearing such that the flat area is as large as the prototype platform (roughly 15cmx15cm). This design would require a much larger air bearing than the design currently has and would require a more sophisticated design to evenly disperse the pressurized air to the testbed as having uneven air flows would result in vibrations of the testbed. As this could be a very costly endeavor, it may be worth seeking out students to take this on as a stand-alone capstone project.

12. References

[1] M. Finckenor and K. de Groh, *A Researcher's Guide to: Space Environmental Effects.* Houston, TX: Nasa ISS Program Science Office, 2016.

[2] M. Nehrenz, M. Sorgenfrei, "On The Development of Spacecraft Operating Modes for a Deep Space CubeSat," *emergentspace.com.* 2, Sep. 2015. [Online]. Available: https://www.emergentspace.com/wp-content/uploads/Space15_NehrenzSorg.pdf. [Accessed 8, Oct. 2018].

[3] B. Dunbar, "Phonesat The Smartphone Nanosatellite," NASA, 17, Mar. 2015. [Online]. Available: https://www.nasa.gov/centers/ames/engineering/projects/phonesat.html. [Accessed: 2, Oct. 2018].

[4] "PhoneSat: Convert Your Smartphone Into a Satellite", *Space Apps Challenge.org*. [Online]. Available: https://2014.spaceappschallenge.org/challenge/phonesat-convert-your-smartphone-satellite/. [Accessed: 2, Oct. 2018].

[5] D. Dickinson, "10 Innovative CubeSat and Nanosatellite Projects," Listosaur | Hungry for Knowledge, 30, Apr. 2014. [Online]. Available: https://listosaur.com/science-atechnology/10-innovative-cubesat-nanosatellite-projects/. [Accessed: 2, Oct. 2018].

[6] C. Niederstrasser et al, "TJ3Sat -- The First Satellite Developed and Operated by High School Students," [Online]. Available: https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=1342&context=smallsat. [Accessed: 3, Oct. 2018].

[7] Pignède, A., *"Detumbling of the NTNU Test Satellite,"* [Online]. Available: *http://nuts.cubesat.no/upload/2015/03/06/antoine_detumbling.pdf*. [Accessed 25, Dec. 2018].

[8] Oland, E. and Schlanbusch, R. *"Reaction Wheel Design for CubeSats,"* [Online]. Available: *https://www.researchgate.net/publication/251889729_Reaction_wheel_design_for_CubeSats.* [Accessed Jan. 2019].

[9] D. Eagle, "Cowell's Method for Earth Satellites", *Mathworks.com*, 2013. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/39703-cowell-s-method-for-earth-satellites?focused=3772928&tab=function. [Accessed: 29- Dec- 2018].

[10] D. Compston, “International Geomagnetic Reference Field (IGRF) Model”, *Mathworks.com*, 2016. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/34388-international-geomagnetic-reference-field-igrf-model. [Accessed: 29-Dec-2018].

[11] "Local Sidereal Time", *Small Satellites*. [Online]. Available: https://smallsats.org/2013/04/14/local-sidereal-time/. [Accessed: 28- Dec- 2018].

[12] D. Koblick, "Convert ECI to ECEF Coordinates", *Mathworks.com*, 2012. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/28233-convert-eci-to-ecef-coordinates. [Accessed: 29- Dec- 2018].

[13] "Pricing and Licensing", *Mathworks.com*. [Online]. Available: https://www.mathworks.com/pricing-licensing.html?prodcode=SL&intendeduse=edu. [Accessed: 03- Oct- 2018].

[14] "Fundamentals of Spacecraft Attitude Determination and Control Hardcover", *Amazon.ca*. [Online]. Available: https://www.amazon.ca/Fundamentals-Spacecraft-Attitude-Determination-Control/dp/1493908014/ref=sr_1_fkmr1_1?ie=UTF8&qid=1549571143&sr=8-1-fkmr1&keywords=Fundamentals+of+Spacecraft+ADCS+Textbook+%2813%29. [Accessed: 24- Sep- 2018].

[15] "PCBs / Laser Cutting - Electrical and Computer Engineering - Western University", *Eng.uwo.ca*. [Online]. Available: https://www.eng.uwo.ca/electrical/e-shop/pcb.html. [Accessed: 18- Oct- 2018].

[16] "Portable Air Compressors", *Home Depot*. [Online]. Available: https://www.homedepot.ca/product/speedway-18-9l-max-125-psi-portable-air-tank/1001020126. [Accessed: 12- Jan- 2019].

[17] "Shop Rates - University Machine Services - Faculty of Engineering - Western University", *Eng.uwo.ca*. [Online]. Available: https://www.eng.uwo.ca/departments-units/university-machine-services/rates.html. [Accessed: 17- Oct- 2018].

[18] "Melexis Technologies NV EVB90393", *Digikey.ca*. [Online]. Available: https://www.digikey.ca/products/en?keywords=EVB90393-ND%20. [Accessed: 28- Oct- 2018].

[19] "MPU-6050 Product Specification Revision 3.4", 2013. [Online]. Available: http://43zrtwysvxb2gf29r5o0athu.wpengine.netdna-cdn.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf. [Accessed: 04- Nov- 2018].

[20] "Acetal Rod", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/8497k313. [Accessed: 02- Nov- 2018].

[21] "Highly Corrosion-Resistant 316 Stainless Steel Ball", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/96415k82. [Accessed: 27- Oct- 2018].

[22] "Acrylic Sheets", *The Home Depot*. [Online]. Available: https://www.homedepot.com/p/Plaskolite-8-in-x-10-in-x-0-050-in-Non-Glare-Acrylic-Sheet-1X09241A/301109740. [Accessed: 25- Nov- 2018].

[23] "Crack-Resist Polypropylene Semi-Clear Tubing", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/1979t2. [Accessed: 23- Oct- 2018].

[24] "Motors, Solenoids, Driver Boards/Modules", *Digikey*. [Online]. Available: https://www.digikey.ca/product-detail/en/nmb-technologies-corporation/PAN14EE12AA1/P14346-ND/2417070. [Accessed: 12- Jan- 2019].

[25] "Arduino Micro ", *Arduino*. [Online]. Available: https://store.arduino.cc/usa/arduino-micro. [Accessed: 19- Nov- 2018].

[26] *Amazon.ca*. [Online]. Available: https://www.amazon.ca/Energizer-Batteries-Designed-Prevent-Damaging/dp/B003STJFLA/ref=sr_1_11?hvadid=229978694598&hvdev=c&hvlocphy=9001074&hvnetw=g&hvpos=1t1&hvqmt=e&hvrand=2661137608520591932&hvtargid=kwd-300125690318&keywords=double+a+batteries&qid=1552421108&s=gateway&sr=8-11&tag=googcana-20. [Accessed: 12- Jan- 2019].

[27] "Panel-Mount Compressed Air Regulator", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/41795k3. [Accessed: 12- Jan- 2019].

[28] "Black-Oxide Alloy Steel Hex Drive Flat Head Screw", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/91253a914. [Accessed: 12- Jan- 2019].

[29] "Black-Oxide Alloy Steel Hex Drive Flat Head Screw", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/91253a537. [Accessed: 12- Jan- 2019].

[30] *Home Depot*. [Online]. Available: https://www.homedepot.ca/product/irving-1x12x8-rough-pine/1000113397. [Accessed: 15- Feb- 2019].

[31] "Push-to-Connect Tube Fitting for Air", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/5779K131. [Accessed: 12- Feb- 2019].

[32] "Miniature Medium-Pressure Stainless Steel Threaded Pipe Fitting", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/2684K21. [Accessed: 14- Feb- 2019].

[33] "Push-to-Connect Tube Fitting for Air", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/5779k108. [Accessed: 14- Feb- 2019].

[34] "Industrial-Shape Hose Coupling", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/6534k46. [Accessed: 14- Feb- 2019].

[35] "Push-to-Connect Tube Fitting for Air", *Mcmaster.com*. [Online]. Available: https://www.mcmaster.com/5779k417. [Accessed: 14- Feb- 2019].

[36] "Dixon Valve TTB75 PTFE Industrial Sealant Tape", *Amazon.ca.* [Online]. Available: https://www.amazon.ca/Dixon-Valve-PTFE-Industrial-Sealant/dp/B003D7K8E0/ref=sr_1_5?hvadid=208256087228&hvdev=c&hvlocphy=9001074&hvnetw=g&hvpos=1t1&hvqmt=e&hvrand=14378099485688864281&hvtargid=kwd-295862483499&keywords=ptfe%2Btape&qid=1552424716&s=gateway&sr=8-5&tag=googcana-20&th=1

[37] “Carr,” *McMaster*. [Online]. Available: https://www.mcmaster.com/7786t12. [Accessed: 10-Feb-2019].

Appendix I

Sun Sensor

errorquatgen

```
function quat = errorquatgen()
%generate quaternion for rotation of given angle (rad) around random axis
%computing first term of quaternion
angle=normrnd(0,0.0029);
q0=cos(angle/2);
%random axis
a=rand;
b=rand;
c=rand;
axis=[a b c];
%normalise the random axis
n=norm(axis);
a=a/n;
b=b/n;
c=c/n;
%computing quaternoin
q1=a*sin(angle/2);
q2=b*sin(angle/2);
q3=c*sin(angle/2);
quat=[q0 q1 q2 q3];
end
```

Magnetometer

NoiseGenerator

```
function y = NoiseGenerator()
RSS_err=0.12*1/100;
FullScale=0.8;
y=randn(3,1)*RSS_err*FullScale;
end
```

Optimized TRIAD

Triad

```
function A1=Triad(ref1,ref2,vec1,vec2)
%This function generates an attitude matrix two reference and two measured
%vectors.

%Calculate body tirad
triad1body=vec1./norm(vec1);
triad2body=cross(triad1body,vec2)./norm(cross(triad1body,vec2));
triad3body=cross(triad1body,triad2body)./...
    norm(cross(triad1body,triad2body));

%Calculate reference tirad
triad1reference=ref1./norm(ref1);
triad2reference=cross(triad1reference,ref2)...
    ./norm(cross(triad1reference,ref2));
triad3reference=cross(triad1reference,triad2reference)./...
    norm(cross(triad1reference,triad2reference));
```

```
%Calculate attituce matrix
A1= triad1body*(triad1reference.') + triad2body*(triad2reference.')...
    + triad3body*(triad3reference.');
```

Optimized-Triad

```
function Attitude_Matrix=Optimised_Triad(A1,A2,sigma1,sigma2)

LinearEstimator=((sigma2)^2/((sigma1)^2+(sigma2)^2))...
    *A1+((sigma1)^2/((sigma1)^2+(sigma2)^2))*A2;
%Orthogonalise
Attitude_Matrix=0.5*(LinearEstimator+(inv(LinearEstimator).'));
```

Control Law (B-dot)

```
function m  = bdot(magField, angvel)
%reads in magnetic field vector in body coordinates,and angular velocity

% B-dot Control
detb=sqrt(dot((magField),(magField)));
m=((-1)/detb)*cross(magField,angvel); %actuation level
```

Initialization

InitializeJ2

```
%define parameters
grav_parameter = 398600.4415;  %in km^3/s^2
earthRadius = 6378.137; % in km
J2 = 0.0010836;
earth_rot = 360*(1 + 1/365.25)/(3600*24);  % Earth's rotation [deg/s]

%read in
fname = 'CubeSatTLE.txt';    % TLE file name
% Open the TLE file and read TLE elements
fid = fopen(fname, 'rb');

%while not at end of file
while ~(feof(fid))
% read data form two-line element set for cubesat orbit
D1 = fscanf(fid,'%23c%*s',1);
D2 = fscanf(fid,'%d%6d%*c%5d%*3c%*2f%f%f%5d%*c%*d%5d%*c%*d%d%5d',[1,9]);
D3 = fscanf(fid,'%d%6d%f%f%f%f%f%f%f',[1,9]);
jd_Fraction = D2(1,4)*24*3600; % Epoch Date and Julian Date Fraction
Ballistic_Coeff = D2(1,5);
inclination = D3(1,3); %in degrees
RAAN = D3(1,4);  %Right Ascension of Ascending Node in degrees
eccentricity = D3(1,5)/1e7;
Arg_periapsis = D3(1,6); %in degrees
avg_anomaly = D3(1,7); %in degrees
avg_motion = D3(1,8); % in revolutions per day
```

```
% defineing Orbital parametres
sm_axis = (grav_parameter/(avg_motion*2*pi/(24*3600))^2)^(1/3);%kilometers
period = 2*pi*sqrt(sm_axis^3/grav_parameter); % in minutes
rp = sm_axis*(1-eccentricity);
AngMomentum = (grav_parameter*rp*(1 + eccentricity))^0.5;
E = keplerEq(avg_anomaly*pi/180,eccentricity,2^(-52));
True_anom =  acos((cos(E) -eccentricity)/(1 - eccentricity*cos(E)))...
    *180/pi; %in degrees

E = 2*atan(tand(True_anom/2)*((1-eccentricity)/(1+eccentricity))^0.5);
avg_anomaly = E  - eccentricity*sin(E);
t0 = avg_anomaly/(2*pi)*period;
end
```

KeplerEq

```
function E = keplerEq(avg_anomaly,eccentricity,eps)
En  = avg_anomaly;
Ens = En - (En-eccentricity*sin(En)- avg_anomaly)/...
    (1 - eccentricity*cos(En));
while ( abs(Ens-En) > eps )
    En = Ens;
    Ens = En - (En - eccentricity*sin(En) - avg_anomaly)/...
        (1 - eccentricity*cos(En));
end
E = Ens;
end
```

Sun Position

Julian

```
function wholejd = julian (month, day, year)
wholejd=0;
y = year;
m = month;
b = 0;
c = 0;

if (m <= 2)
   y = y - 1;
   m = m + 12;
end

if (y < 0)
   c = -.75;
end
```

```
% check for valid calendar date
if (year < 1582)
   % null
elseif (year > 1582)
   a = fix(y / 100);
   b = 2 - a + floor(a / 4);
elseif (month < 10)
   % null
elseif (month > 10)
   a = fix(y / 100);
   b = 2 - a + floor(a / 4);
elseif (day <= 4)
   % null
elseif (day > 14)
   a = fix(y / 100);
   b = 2 - a + floor(a / 4);
   fprintf('\n\n  this is an invalid calendar date!!\n');
   return;
end

jd = fix(365.25 * y + c) + fix(30.6001 * (m + 1));
wholejd = jd + day + b + 1720994.5;
```

fracjday

```
function fracjd=fracjday(uthr, utmin, utsec)
fracjd=uthr / 24 + utmin / 1440 + utsec / 86400;
function rsun  = sun1 (jdate)
atr = pi / 648000;
rsun = zeros(3, 1);

% time arguments
djd = jdate - 2451545;
t = (djd / 36525) + 1;

% fundamental arguments (radians)
gs =2.0*pi*((0.993126+0.0027377785*djd)-fix(0.993126+0.0027377785*djd));
lm = 2.0*pi*((0.606434+0.03660110129*djd)-fix(0.606434+0.03660110129*djd));
ls = 2.0*pi*((0.606434+0.03660110129*djd)-fix(0.606434+0.03660110129*djd));
g2 = 2.0*pi*((0.606434+0.03660110129*djd)-fix(0.606434+0.03660110129*djd));
g4 = 2.0*pi*((0.606434+0.03660110129*djd)-fix(0.606434+0.03660110129*djd));
g5 = 2.0*pi*((0.606434+0.03660110129*djd)-fix(0.606434+0.03660110129*djd));
rm = 2.0*pi*((0.606434+0.03660110129*djd)-fix(0.606434+0.03660110129*djd));

% geocentric, ecliptic longitude of the sun (radians)
plon = 6910*sin(gs)+72*sin(2*gs)-17*t*sin(gs);
plon = plon-7*cos(gs-g5)+6*sin(lm-ls)+5*sin(4*gs-8*g4+3*g5);
plon = plon-5*cos(2*(gs-g2))-4*(sin(gs-g2)-cos(4*gs-8*g4+3*g5));
plon = plon+3*(sin(2*(gs-g2))-sin(g5)-sin(2*(gs-g5)));
plon = ls+atr*(plon-17*sin(rm));
```

```
% geocentric distance of the sun (kilometers)
rsm = 149597870.691*(1.00014-0.01675*cos(gs)-0.00014*cos(2*gs));

% obliquity of the ecliptic (radians)
obliq = atr*(84428-47*t+9*cos(rm));
```

sun1

```
% geocentric, equatorial right ascension and declination (radians)
a = sin(plon)*cos(obliq);
b = cos(plon);

epsilon = 0.0000000001;
pidiv2 = 0.5 * pi;
dontenterflag=0;
c=0;
if (abs(a) < epsilon)
    rascy = (1 - sign(b)) * pidiv2;
    dontenterflag=1;
else
    c = (2 - sign(a)) * pidiv2;
end
if ((abs(b) < epsilon)&&dontenterflag~=1)
    rascy = c;
else
    rascy = c + sign(a) * sign(b) * (abs(atan(a / b)) - pidiv2);
end
rasc = rascy;
decl = asin(sin(obliq) * sin(plon));

% geocentric position vector of the sun (kilometers)
rsun(1) = rsm * cos(rasc) * cos(decl);
rsun(2) = rsm * sin(rasc) * cos(decl);
rsun(3) = rsm * sin(decl);
```

Appendix II

Arduino Code for Prototype

```
// 03/31/2019 by Rachelle Cheung
// I2Cdev and MPU6050 must be installed as libraries, or else the .cpp/.h files
// for both classes must be in the include path of your project
#include "I2Cdev.h"

#include "MPU6050_6Axis_MotionApps20.h"
//#include "MPU6050.h"

// Arduino Wire library is required if I2Cdev I2CDEV_ARDUINO_WIRE implementation
// is used in I2Cdev.h
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
   #include "Wire.h"
#endif

// class default I2C address is 0x68
// specific I2C addresses may be passed as a parameter here
// AD0 low = 0x68 (default for SparkFun breakout and InvenSense evaluation board)
// AD0 high = 0x69
MPU6050 mpu;
//MPU6050 mpu(0x69); // <-- use for AD0 high

// uncomment "OUTPUT_READABLE_YAWPITCHROLL" if you want to see the yaw/
// pitch/roll angles (in degrees) calculated from the quaternions coming
// from the FIFO. Note this also requires gravity vector calculations.
// Also note that yaw/pitch/roll angles suffer from gimbal lock (for
// more info, see: http://en.wikipedia.org/wiki/Gimbal_lock)
#define OUTPUT_READABLE_YAWPITCHROLL

// uncomment "OUTPUT_TEAPOT" if you want output that matches the
// format used for the InvenSense teapot demo
```

```
//#define OUTPUT_TEAPOT

#define LED_PIN 13 // (Arduino is 13, Teensy is 11, Teensy++ is 6)
bool blinkState = false;

// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorInt16 aa;         // [x, y, z]            accel sensor measurements
VectorInt16 aaReal;     // [x, y, z]            gravity-free accel sensor measurements
VectorInt16 aaWorld;    // [x, y, z]            world-frame accel sensor measurements
VectorFloat gravity;    // [x, y, z]            gravity vector
float euler[3];         // [psi, theta, phi]    Euler angle container
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and gravity vector

// packet structure for InvenSense teapot demo
uint8_t teapotPacket[14] = { '$', 0x02, 0,0, 0,0, 0,0, 0,0, 0x00, 0x00, '\r', '\n' };

// ================================================================
// ===               INTERRUPT DETECTION ROUTINE                ===
// ================================================================
volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt pin has gone high
void dmpDataReady() {
   mpuInterrupt = true;
}
//added for motors
```

```
const int in1 = 3;
const int in2 = 5;

const int in3 = 6;
const int in4 = 9;

const int in5 = 10;
const int in6 = 11;

//calibration initialization
float yaw;
float pitch;
float roll;
float start;
int count = 1;
float yawdiff;
float pitchdiff;
float rolldiff;

// ================================================================
// ===                  INITIAL SETUP                           ===
// ================================================================

void setup() {
    // join I2C bus (I2Cdev library doesn't do this automatically)
    #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
        Wire.begin();
        TWBR = 24; // 400kHz I2C clock (200kHz if CPU is 8MHz)
    #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
        Fastwire::setup(400, true);
    #endif

    // initialize serial communication
```

```
    // (115200 chosen because it is required for Teapot Demo output, but it's
    // really up to you depending on your project)
    Serial.begin(115200);
    while (!Serial); // wait for Leonardo enumeration, others continue immediately

    // NOTE: 8MHz or slower host processors, like the Teensy @ 3.3v or Ardunio
    // Pro Mini running at 3.3v, cannot handle this baud rate reliably due to
    // the baud timing being too misaligned with processor ticks. You must use
    // 38400 or slower in these cases, or use some kind of external separate
    // crystal solution for the UART timer.

    // initialize device
    Serial.println(F("Initializing I2C devices..."));
    mpu.initialize();

    // verify connection
    Serial.println(F("Testing device connections..."));
    Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") : F("MPU6050
connection failed"));

    // wait for ready
    Serial.println(F("\nSend any character to begin DMP programming and demo: "));
    while (Serial.available() && Serial.read()); // empty buffer
    //while (!Serial.available());                 // wait for data
    //while (Serial.available() && Serial.read()); // empty buffer again

    // load and configure the DMP
    Serial.println(F("Initializing DMP..."));
    devStatus = mpu.dmpInitialize();

    // supply your own gyro offsets here, scaled for min sensitivity
    mpu.setXGyroOffset(220);
    mpu.setYGyroOffset(76);
```

```
mpu.setZGyroOffset(-85);
mpu.setZAccelOffset(1788); // 1688 factory default for my test chip

// make sure it worked (returns 0 if so)
if (devStatus == 0) {
    // turn on the DMP, now that it's ready
    Serial.println(F("Enabling DMP..."));
    mpu.setDMPEnabled(true);

    // enable Arduino interrupt detection
    Serial.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));
    attachInterrupt(0, dmpDataReady, RISING);
    mpuIntStatus = mpu.getIntStatus();

    // set our DMP Ready flag so the main loop() function knows it's okay to use it
    Serial.println(F("DMP ready! Waiting for first interrupt..."));
    dmpReady = true;

    // get expected DMP packet size for later comparison
    packetSize = mpu.dmpGetFIFOPacketSize();
} else {
    // ERROR!
    // 1 = initial memory load failed
    // 2 = DMP configuration updates failed
    // (if it's going to break, usually the code will be 1)
    Serial.print(F("DMP Initialization failed (code "));
    Serial.print(devStatus);
    Serial.println(F(")"));
}

// configure LED for output
pinMode(LED_PIN, OUTPUT);
```

```
    pinMode(in1,OUTPUT);
    pinMode(in2,OUTPUT);
    pinMode(in3,OUTPUT);
    pinMode(in4,OUTPUT);
    pinMode(in5,OUTPUT);
    pinMode(in6,OUTPUT);

    start = millis();
}

// ================================================================
// ===                  MAIN PROGRAM LOOP                     ===
// ================================================================

void loop() {

    float current = millis();
    // if programming failed, don't try to do anything
    if (!dmpReady) return;

    // wait for MPU interrupt or extra packet(s) available
    while (!mpuInterrupt && fifoCount < packetSize) {
    }

    // reset interrupt flag and get INT_STATUS byte
    mpuInterrupt = false;
    mpuIntStatus = mpu.getIntStatus();

    // get current FIFO count
    fifoCount = mpu.getFIFOCount();

    // check for overflow (this should never happen unless our code is too inefficient)
    if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
```

```
    // reset so we can continue cleanly
    mpu.resetFIFO();
    Serial.println(F("FIFO overflow!"));

// otherwise, check for DMP data ready interrupt (this should happen frequently)
}

//otherwise, check for DMP data ready interrupt (this should happen frequently)
else if (mpuIntStatus & 0x02) {
    // wait for correct available data length, should be a VERY short wait
    while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

    // read a packet from FIFO
    mpu.getFIFOBytes(fifoBuffer, packetSize);

    // track FIFO count here in case there is > 1 packet available
    // (this lets us immediately read more without waiting for an interrupt)
    fifoCount -= packetSize;

    if ((current - start) < 20000)
    {
      #ifdef OUTPUT_READABLE_YAWPITCHROLL
        // display Euler angles in degrees
        mpu.dmpGetQuaternion(&q, fifoBuffer);
        mpu.dmpGetGravity(&gravity, &q);
        mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
        Serial.print("ypr\t");
        Serial.print(ypr[0] * 180/M_PI);
        Serial.print("\t");
        Serial.print(ypr[1] * 180/M_PI);
        Serial.print("\t");
        Serial.println(ypr[2] * 180/M_PI);
      #endif
```

```
    #ifdef OUTPUT_TEAPOT
        // display quaternion values in InvenSense Teapot demo format:
        teapotPacket[2] = fifoBuffer[0];
        teapotPacket[3] = fifoBuffer[1];
        teapotPacket[4] = fifoBuffer[4];
        teapotPacket[5] = fifoBuffer[5];
        teapotPacket[6] = fifoBuffer[8];
        teapotPacket[7] = fifoBuffer[9];
        teapotPacket[8] = fifoBuffer[12];
        teapotPacket[9] = fifoBuffer[13];
        Serial.write(teapotPacket, 14);
        teapotPacket[11]++; // packetCount, loops at 0xFF on purpose
    #endif

    // blink LED to indicate activity
    blinkState = !blinkState;
    digitalWrite(LED_PIN, blinkState);
}

else
{
    if (count == 1)
    {
        Serial.print("HERE");
        yaw = ypr[0] * 180/M_PI;
        pitch = ypr[1] * 180/M_PI;
        roll = ypr[2] * 180/M_PI;
        count++;
    }
    else
    {
        Serial.print("THERE");
```

```
#ifdef OUTPUT_READABLE_YAWPITCHROLL
    // display Euler angles in degrees
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
    Serial.print("ypr\t");
    Serial.print(ypr[0] * 180/M_PI);
    Serial.print("YAW DIFF\t");
    yawdiff = yaw-(ypr[0] * 180/M_PI);
    Serial.print(yawdiff);
    Serial.print("\t");
    Serial.print(ypr[1] * 180/M_PI);
    Serial.print("PITCH DIFF\t");
    pitchdiff = pitch-(ypr[1] * 180/M_PI);
    Serial.print(pitchdiff);
    Serial.print("\t");
    Serial.print(ypr[2] * 180/M_PI);
    Serial.print("ROLL DIFF\t");
    rolldiff = roll-(ypr[2] * 180/M_PI);
    Serial.println(rolldiff);

    //rotate about board (z) = yaw (in3,in4)
    //rotate about y - pitch (in1,in2)
    //rotate about x - roll (in5,in6)
    if (yawdiff > 1)
    {
      analogWrite(in4,0);
      analogWrite(in3,100);

      if (pitchdiff > 1)
      {
        analogWrite(in2,0);
        analogWrite(in1,100);
```

```
    if (rolldiff > 1)
    {
       analogWrite(in6,0);
       analogWrite(in5,100);
    }

    else if (rolldiff < -1)
    {
       analogWrite(in5,0);
       analogWrite(in6,100);
    }

    else
    {
       analogWrite(in5,0);
       analogWrite(in6,0);
    }
}

else if (pitchdiff < -1)
{
    analogWrite(in1,0);
    analogWrite(in2,100);

    if (rolldiff > 1)
    {
       analogWrite(in6,0);
       analogWrite(in5,100);
    }

    else if (rolldiff < -1)
    {
```

```
      analogWrite(in5,0);
      analogWrite(in6,100);
   }

   else
   {
      analogWrite(in5,0);
      analogWrite(in6,0);
   }
}

else
{
   analogWrite(in1,0);
   analogWrite(in2,0);

   if (rolldiff > 1)
   {
      analogWrite(in6,0);
      analogWrite(in5,100);
   }

   else if (rolldiff < -1)
   {
      analogWrite(in5,0);
      analogWrite(in6,100);
   }

   else
   {
      analogWrite(in5,0);
      analogWrite(in6,0);
   }
```

```
    }
    delay(20);
}

else if (yawdiff < -1)
{
    analogWrite(in3,0);
    analogWrite(in4,100);

    if (pitchdiff > 1)
    {
        analogWrite(in2,0);
        analogWrite(in1,100);

        if (rolldiff > 1)
        {
            analogWrite(in6,0);
            analogWrite(in5,100);
        }

        else if (rolldiff < -1)
        {
            analogWrite(in5,0);
            analogWrite(in6,100);
        }

        else
        {
            analogWrite(in5,0);
            analogWrite(in6,0);
        }
    }
```

```
else if (pitchdiff < -1)
{
   analogWrite(in1,0);
   analogWrite(in2,100);

   if (rolldiff > 1)
   {
      analogWrite(in6,0);
      analogWrite(in5,100);
   }

   else if (rolldiff < -1)
   {
      analogWrite(in5,0);
      analogWrite(in6,100);
   }

   else
   {
      analogWrite(in5,0);
      analogWrite(in6,0);
   }
}

else
{
   analogWrite(in1,0);
   analogWrite(in2,0);

   if (rolldiff > 1)
   {
      analogWrite(in6,0);
      analogWrite(in5,100);
```

```
        }

        else if (rolldiff < -1)
        {
           analogWrite(in5,0);
           analogWrite(in6,100);
        }

        else
        {
           analogWrite(in5,0);
           analogWrite(in6,0);
        }
     }

     delay(20);
}

else
{
     analogWrite(in3,0);
     analogWrite(in4,0);

     if (pitchdiff > 1)
     {
        analogWrite(in2,0);
        analogWrite(in1,100);

        if (rolldiff > 1)
        {
           analogWrite(in6,0);
           analogWrite(in5,100);
        }
```

```
    else if (rolldiff < -1)
    {
       analogWrite(in5,0);
       analogWrite(in6,100);
    }

    else
    {
       analogWrite(in5,0);
       analogWrite(in6,0);
    }
}

else if (pitchdiff < -1)
{
    analogWrite(in1,0);
    analogWrite(in2,100);

    if (rolldiff > 1)
    {
       analogWrite(in6,0);
       analogWrite(in5,100);
    }

    else if (rolldiff < -1)
    {
       analogWrite(in5,0);
       analogWrite(in6,100);
    }

    else
    {
```

```
            analogWrite(in5,0);
            analogWrite(in6,0);
        }
    }

    else
    {
        analogWrite(in1,0);
        analogWrite(in2,0);

        if (rolldiff > 1)
        {
            analogWrite(in6,0);
            analogWrite(in5,100);
        }

        else if (rolldiff < -1)
        {
            analogWrite(in5,0);
            analogWrite(in6,100);
        }

        else
        {
            analogWrite(in5,0);
            analogWrite(in6,0);
        }
    }

    delay(20);
  }

#endif
```

```
#ifdef OUTPUT_TEAPOT
   // display quaternion values in InvenSense Teapot demo format:
   teapotPacket[2] = fifoBuffer[0];
   teapotPacket[3] = fifoBuffer[1];
   teapotPacket[4] = fifoBuffer[4];
   teapotPacket[5] = fifoBuffer[5];
   teapotPacket[6] = fifoBuffer[8];
   teapotPacket[7] = fifoBuffer[9];
   teapotPacket[8] = fifoBuffer[12];
   teapotPacket[9] = fifoBuffer[13];
   Serial.write(teapotPacket, 14);
   teapotPacket[11]++; // packetCount, loops at 0xFF on purpose

                  mpu.dmpGetQuaternion(&q, fifoBuffer);
   mpu.dmpGetGravity(&gravity, &q);
   mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
   Serial.print("ypr\t");
   Serial.print(ypr[0] * 180/M_PI);
   Serial.print("YAW DIFF\t");
   yawdiff = yaw-(ypr[0] * 180/M_PI);
   Serial.print(yawdiff);
   Serial.print("\t");
   Serial.print(ypr[1] * 180/M_PI);
   Serial.print("PITCH DIFF\t");
   pitchdiff = pitch-(ypr[1] * 180/M_PI);
   Serial.print(pitchdiff);
   Serial.print("\t");
   Serial.print(ypr[2] * 180/M_PI);
   Serial.print("ROLL DIFF\t");
   rolldiff = roll-(ypr[2] * 180/M_PI);
   Serial.println(rolldiff);
```

```
//rotate about board (z) = yaw (in3,in4)
//rotate about y - pitch (in1,in2)
//rotate about x - roll (in5,in6)
if (yawdiff > 1)
{
   analogWrite(in4,0);
   analogWrite(in3,100);

   if (pitchdiff > 1)
   {
      analogWrite(in2,0);
      analogWrite(in1,100);

      if (rolldiff > 1)
      {
         analogWrite(in6,0);
         analogWrite(in5,100);
      }

      else if (rolldiff < -1)
      {
         analogWrite(in5,0);
         analogWrite(in6,100);
      }

      else
      {
         analogWrite(in5,0);
         analogWrite(in6,0);
      }
   }

   else if (pitchdiff < -1)
```

```
{
   analogWrite(in1,0);
   analogWrite(in2,100);

   if (rolldiff > 1)
   {
      analogWrite(in6,0);
      analogWrite(in5,100);
   }

   else if (rolldiff < -1)
   {
      analogWrite(in5,0);
      analogWrite(in6,100);
   }

   else
   {
      analogWrite(in5,0);
      analogWrite(in6,0);
   }
}

else
{
   analogWrite(in1,0);
   analogWrite(in2,0);

   if (rolldiff > 1)
   {
      analogWrite(in6,0);
      analogWrite(in5,100);
   }
```

```
        else if (rolldiff < -1)
        {
           analogWrite(in5,0);
           analogWrite(in6,100);
        }

        else
        {
           analogWrite(in5,0);
           analogWrite(in6,0);
        }
     }
     delay(20);
}

else if (yawdiff < -1)
{
     analogWrite(in3,0);
     analogWrite(in4,100);

     if (pitchdiff > 1)
     {
        analogWrite(in2,0);
        analogWrite(in1,100);

        if (rolldiff > 1)
        {
           analogWrite(in6,0);
           analogWrite(in5,100);
        }

        else if (rolldiff < -1)
```

```
    {
      analogWrite(in5,0);
      analogWrite(in6,100);
    }

    else
    {
      analogWrite(in5,0);
      analogWrite(in6,0);
    }
}

else if (pitchdiff < -1)
{
    analogWrite(in1,0);
    analogWrite(in2,100);

    if (rolldiff > 1)
    {
      analogWrite(in6,0);
      analogWrite(in5,100);
    }

    else if (rolldiff < -1)
    {
      analogWrite(in5,0);
      analogWrite(in6,100);
    }

    else
    {
      analogWrite(in5,0);
      analogWrite(in6,0);
```

```
        }
    }

    else
    {
        analogWrite(in1,0);
        analogWrite(in2,0);

        if (rolldiff > 1)
        {
            analogWrite(in6,0);
            analogWrite(in5,100);
        }

        else if (rolldiff < -1)
        {
            analogWrite(in5,0);
            analogWrite(in6,100);
        }

        else
        {
            analogWrite(in5,0);
            analogWrite(in6,0);
        }
    }

    delay(20);
}

else
{
    analogWrite(in3,0);
```

```
analogWrite(in4,0);

if (pitchdiff > 1)
{
   analogWrite(in2,0);
   analogWrite(in1,100);

   if (rolldiff > 1)
   {
      analogWrite(in6,0);
      analogWrite(in5,100);
   }

   else if (rolldiff < -1)
   {
      analogWrite(in5,0);
      analogWrite(in6,100);
   }

   else
   {
      analogWrite(in5,0);
      analogWrite(in6,0);
   }
}

else if (pitchdiff < -1)
{
   analogWrite(in1,0);
   analogWrite(in2,100);

   if (rolldiff > 1)
   {
```

```
        analogWrite(in6,0);
        analogWrite(in5,100);
    }

    else if (rolldiff < -1)
    {
        analogWrite(in5,0);
        analogWrite(in6,100);
    }

    else
    {
        analogWrite(in5,0);
        analogWrite(in6,0);
    }
}

else
{
    analogWrite(in1,0);
    analogWrite(in2,0);

    if (rolldiff > 1)
    {
        analogWrite(in6,0);
        analogWrite(in5,100);
    }

    else if (rolldiff < -1)
    {
        analogWrite(in5,0);
        analogWrite(in6,100);
    }
```

```
                else
                {
                    analogWrite(in5,0);
                    analogWrite(in6,0);
                }
            }

            delay(20);
        }
    #endif

    // blink LED to indicate activity
    //blinkState = !blinkState;
    //digitalWrite(LED_PIN, blinkState);
  }
  }
 }
}
```

Processing Code for Prototype

```
// Changelog:
//     2019-03-06 – modifications made by Rachelle Cheung
//     2019-03-20 – modifications made by Rachelle Cheung
//     2019-04-01 – modifications made by Rachelle Cheung


import processing.serial.*;
import processing.opengl.*;
import toxi.geom.*;
import toxi.processing.*;

ToxiclibsSupport gfx;

Serial port;                         // The serial port
char[] teapotPacket = new char[14];  // InvenSense Teapot packet
int serialCount = 0;                 // current packet byte position
```

```
int aligned = 0;
int interval = 0;

float[] q = new float[4];
Quaternion quat = new Quaternion(1, 0, 0, 0);

float[] gravity = new float[3];
float[] euler = new float[3];
float[] ypr = new float[3];

void setup() {
    // 300px square viewport using OpenGL rendering
    size(1400, 1000, OPENGL);
    gfx = new ToxiclibsSupport(this);

    // setup lights and antialiasing
    lights();
    smooth();

    // display serial port list for debugging/clarity
    println(Serial.list());

    // get the first available port (use EITHER this OR the specific port code below)
    String portName = "/dev/cu.usbserial-1420";

    // get a specific serial port (use EITHER this OR the first-available code above)
    //String portName = "COM4";

    // open the serial port
    port = new Serial(this, portName, 115200);

    // send single character to trigger DMP init/start
    // (expected by MPU6050_DMP6 example Arduino sketch)
    port.write('r');
}

void draw() {
    if (millis() - interval > 1000) {
        // resend single character to trigger DMP init/start
        // in case the MPU is halted/reset while applet is running
        port.write('r');
        interval = millis();
    }

    // black background
    background(0);
```

```
    // translate everything to the middle of the viewport
    pushMatrix();
    translate(width / 2, height / 2);

    // 3-step rotation from yaw/pitch/roll angles (gimbal lock!)
    // ...and other weirdness I haven't figured out yet
    //rotateY(-ypr[0]);
    //rotateZ(-ypr[1]);
    //rotateX(-ypr[2]);

    // toxiclibs direct angle/axis rotation from quaternion (NO gimbal lock!)
    // (axis order [1, 3, 2] and inversion [-1, +1, +1] is a consequence of
    // different coordinate system orientation assumptions between Processing
    // and InvenSense DMP)
    float[] axis = quat.toAxisAngle();
    rotate(axis[0], -axis[1], axis[3], axis[2]);

    // draw outline of CubeSat
    //fill(0, 0, 255, 1);
    noFill();
    stroke(200);
    strokeWeight(5);
    box(150, 150, 300);

    // cameras
    strokeWeight(1);
    stroke(10);
    fill(255, 0, 0, 200); //bright red
    pushMatrix();
    translate(0, 0, -158); //where base of cylinder starts
    rotateX(PI/2);
    drawCylinder(35, 40, 10, 8); //drawCylinder(tip, radius, height, number of sides of base shape)
    popMatrix();
    stroke(10);
    fill(245, 193, 234, 200); //pink
    pushMatrix();
    translate(0, 0, 158); //where base of cylinder starts
    rotateX(PI/2);
    drawCylinder(-35, -40, -10, 8); //drawCylinder(tip, radius, height, number of sides of base
shape)
    popMatrix();

    // magnetometers
    fill(50,255,50,200); //green
    box(70, 10, 10);
```

```
//translate(50, 100, 0);
fill(242,237,61,200); //yellow
translate(40, 0);
rotateY(radians(90));
box(70, 10, 10);
fill(242,149,61,200); //orange
translate(40,10);
rotateZ(radians(90));
//translate(50, 100, 0);
box(70, 10, 10);

//gyroscope
fill(177,92,211,200); //purple
translate(40,10);
rotateZ(radians(90));
//translate(50, 100, 0);
box(10, 30, 20);

//board1
fill(180,178,178,200); //grey
//translate(0,10);
translate(40,50,-40);
box(10, 150, 150);

//board2
fill(180,178,178,200); //grey
//translate(0,10);
translate(60,0,0);
box(10, 150, 150);

//board3
fill(180,178,178,200); //grey
//translate(0,10);
translate(60,0,0);
box(10, 150, 150);

//board4
fill(180,178,178,200); //grey
//translate(0,10);
translate(-190,0,0);
box(10, 150, 150);

//board5
fill(180,178,178,200); //grey
//translate(0,10);
translate(-70,0,0);
```

```
    box(10, 150, 150);

    //box(40);
    //fill(0, 255, 0, 200);
    //box(90, 10, 10);
    //box(10, 90, 10);
    //beginShape(TRIANGLES);
    //vertex(-100,  2, 30); vertex(0,  2, -80); vertex(100,  2, 30);  // wing top layer
    //vertex(-100, -2, 30); vertex(0, -2, -80); vertex(100, -2, 30);  // wing bottom layer
    //vertex(-2, 0, 98); vertex(-2, -30, 98); vertex(-2, 0, 70);  // tail left layer
    //vertex( 2, 0, 98); vertex( 2, -30, 98); vertex( 2, 0, 70);  // tail right layer
    //endShape();
    //beginShape(QUADS);
    //vertex(-100, 2, 30); vertex(-100, -2, 30); vertex(  0, -2, -80); vertex(  0, 2, -80);
    //vertex( 100, 2, 30); vertex( 100, -2, 30); vertex(  0, -2, -80); vertex(  0, 2, -80);
    //vertex(-100, 2, 30); vertex(-100, -2, 30); vertex(100, -2,  30); vertex(100, 2,  30);
    //vertex(-2,   0, 98); vertex(2,   0, 98); vertex(2, -30, 98); vertex(-2, -30, 98);
    //vertex(-2,   0, 98); vertex(2,   0, 98); vertex(2,   0, 70); vertex(-2,   0, 70);
    //vertex(-2, -30, 98); vertex(2, -30, 98); vertex(2,   0, 70); vertex(-2,   0, 70);
    //endShape();

    popMatrix();
}

void serialEvent(Serial port) {
    interval = millis();
    while (port.available() > 0) {
        int ch = port.read();
        print((char)ch);
        if (ch == '$') {serialCount = 0;} // this will help with alignment
        if (aligned < 4) {
            // make sure we are properly aligned on a 14-byte packet
            if (serialCount == 0) {
                if (ch == '$') aligned++; else aligned = 0;
            } else if (serialCount == 1) {
                if (ch == 2) aligned++; else aligned = 0;
            } else if (serialCount == 12) {
                if (ch == '\r') aligned++; else aligned = 0;
            } else if (serialCount == 13) {
                if (ch == '\n') aligned++; else aligned = 0;
            }
            //println(ch + " " + aligned + " " + serialCount);
            serialCount++;
            if (serialCount == 14) serialCount = 0;
        } else {
            if (serialCount > 0 || ch == '$') {
```

```
            teapotPacket[serialCount++] = (char)ch;
            if (serialCount == 14) {
                serialCount = 0; // restart packet byte position

                // get quaternion from data packet
                q[0] = ((teapotPacket[2] << 8) | teapotPacket[3]) / 16384.0f;
                q[1] = ((teapotPacket[4] << 8) | teapotPacket[5]) / 16384.0f;
                q[2] = ((teapotPacket[6] << 8) | teapotPacket[7]) / 16384.0f;
                q[3] = ((teapotPacket[8] << 8) | teapotPacket[9]) / 16384.0f;
                for (int i = 0; i < 4; i++) if (q[i] >= 2) q[i] = -4 + q[i];

                // set our toxilibs quaternion to new data
                quat.set(q[0], q[1], q[2], q[3]);

                /*
                // below calculations unnecessary for orientation only using toxilibs

                // calculate gravity vector
                gravity[0] = 2 * (q[1]*q[3] - q[0]*q[2]);
                gravity[1] = 2 * (q[0]*q[1] + q[2]*q[3]);
                gravity[2] = q[0]*q[0] - q[1]*q[1] - q[2]*q[2] + q[3]*q[3];

                // calculate Euler angles
                euler[0] = atan2(2*q[1]*q[2] - 2*q[0]*q[3], 2*q[0]*q[0] + 2*q[1]*q[1] - 1);
                euler[1] = -asin(2*q[1]*q[3] + 2*q[0]*q[2]);
                euler[2] = atan2(2*q[2]*q[3] - 2*q[0]*q[1], 2*q[0]*q[0] + 2*q[3]*q[3] - 1);

                // calculate yaw/pitch/roll angles
                ypr[0] = atan2(2*q[1]*q[2] - 2*q[0]*q[3], 2*q[0]*q[0] + 2*q[1]*q[1] - 1);
                ypr[1] = atan(gravity[0] / sqrt(gravity[1]*gravity[1] + gravity[2]*gravity[2]));
                ypr[2] = atan(gravity[1] / sqrt(gravity[0]*gravity[0] + gravity[2]*gravity[2]));

                // output various components for debugging
                //println("q:\t" + round(q[0]*100.0f)/100.0f + "\t" + round(q[1]*100.0f)/100.0f + "\t"
+ round(q[2]*100.0f)/100.0f + "\t" + round(q[3]*100.0f)/100.0f);
                //println("euler:\t" + euler[0]*180.0f/PI + "\t" + euler[1]*180.0f/PI + "\t" +
euler[2]*180.0f/PI);
                //println("ypr:\t" + ypr[0]*180.0f/PI + "\t" + ypr[1]*180.0f/PI + "\t" +
ypr[2]*180.0f/PI);
                */
            }
        }
    }
  }
}
```

```
void drawCylinder(float topRadius, float bottomRadius, float tall, int sides) {
   float angle = 0;
   float angleIncrement = TWO_PI / sides;
   beginShape(QUAD_STRIP);
   for (int i = 0; i < sides + 1; ++i) {
      vertex(topRadius*cos(angle), 0, topRadius*sin(angle));
      vertex(bottomRadius*cos(angle), tall, bottomRadius*sin(angle));
      angle += angleIncrement;
   }
   endShape();

   // If it is not a cone, draw the circular top cap
   if (topRadius != 0) {
      angle = 0;
      beginShape(TRIANGLE_FAN);

      // Center point
      vertex(0, 0, 0);
      for (int i = 0; i < sides + 1; i++) {
         vertex(topRadius * cos(angle), 0, topRadius * sin(angle));
         angle += angleIncrement;
      }
      endShape();
   }

   // If it is not a cone, draw the circular bottom cap
   if (bottomRadius != 0) {
      angle = 0;
      beginShape(TRIANGLE_FAN);

      // Center point
      vertex(0, tall, 0);
      for (int i = 0; i < sides + 1; i++) {
         vertex(bottomRadius * cos(angle), tall, bottomRadius * sin(angle));
         angle += angleIncrement;
      }
      endShape();
   }
}
```