2020-03-31

MAD MQP 2001

Andrew M. Montero
Worcester Polytechnic Institute

Christopher Renfro
Worcester Polytechnic Institute

David J. Resmini
Worcester Polytechnic Institute

Oliver D. Hasson
Worcester Polytechnic Institute

MAD 2001

Design and Analysis of a CubeSat

A Major Qualifying Project

Submitted By:

Robaire Galliath

Oliver Hasson

Andrew Montero

Chistopher Renfro

David Resmini

Submitted To:

Professor Michael A. Demetriou

WPI

Submitted to the Faculty of the Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science in Aerospace Engineering.

August 2019 - March 2020

Abstract

This project develops an Attitude Determination and Control (ADC) subsystem for a six-unit CubeSat in an extreme low Earth orbit (eLEO) mission performing atmospheric neutral and ion mass spectrometry. The selection of sensors and actuators were evaluated and updated from a previous mission. The performance of algorithms used for detumble, initial attitude determination, and attitude maintenance was evaluated using MATLAB, Simulink, and Systems Tool Kit (STK) simulations. In order to conduct this evaluation, in-depth Simulink models of spacecraft attitude dynamics and control were developed which consider sensor noise and refresh rates for a GPS receiver, gyroscope, magnetometer, and two-axis sun sensors, as well as actuator limitations for reaction wheels and magnetorquers. In addition, detailed guides to the Simulink models, derivations of ADC algorithms, and recommendations were developed to assist future CubeSat ADC teams. The work developed in this project will allow future groups to simulate different mission profiles, instrument configurations, and control laws in order to improve their design analysis.

Contents

List of Tables

List of Figures

1 Introduction

The goal of this project is to design and conduct analysis of a CubeSat, referred to as the Neutral and Ion Mass Spectrometer eLeo Atmospheric Observer, or "NeAtO", on an extreme low Earth orbit (eLEO) mission. NeAtO will be carrying a mass spectrometer to conduct atmospheric observation. Following deployment from the ISS, NeAtO will enter a 250 - 600 kilometer orbit and maintain this orbit as long as possible. The overall project is composed of three separate MQP teams, each responsible for a different aspect of NeAtO's design and analysis. This portion of the overall project focuses on the attitude determination and control, orbital determination and control, and command subsystems of NeAtO.

1.1 CubeSat Background

Cube satellites (CubeSats) are miniature satellites used for space research and technology development. They are a particular class of nano-satellite that was developed by California Polytechnic State University and Stanford University in 1999 in order to promote the design, manufacture and testing of satellite technology in low Earth orbit [1]. CubeSats are comprised of multiple 10 cm by 10 cm by 10 cm units, referred to as 'U's. Layouts can vary greatly, but the most common form factors are 1U and 3U [1]. In recent years, larger CubeSats have been developed to increase available space for mission payloads. Typically, CubeSats are deployed by a launch mechanism attached tzo the upper stage of a launch vehicle and offer an easy way to deploy CubeSats into Earth orbit. The CubeSat concept began as a plan to provide scientific and military laboratories a tool to grow their operations in space [2]. In recent years, the CubeSat has become a unique tool in the scientific community. NASA's CubeSat Launch Initiative (CSLI) provides opportunities to launch CubeSats aboard larger launch vehicles as secondary payloads [3]. CubeSats are currently in an era of rapid growth; it is estimated that the global CubeSat market was valued at $152 million in 2018, and is projected to rise to nearly $375 million in coming years [4].

1.2 Social, Economic, and Educational Impacts

The continued expansion and development of CubeSat technologies has yielded a variety of positive social, economic, and educational effects. The space industry has been heavily impacted by the surge of CubeSat and related space technology start-ups in the last five years.

From 2000 to 2014 space start-ups have received a combined sum of $1.1 billion in venture capital investments [4]. The number has only increased in recent years, with a total investment of $3.9 billion in 2017 alone [4]. These companies provide satellite components, development, or launch and integration services. As of 2018, 51% of CubeSats are developed by the private sector, demonstrating that CubeSats are not limited to large research institutions or governments [4]. The wide availability of CubeSat components and hardware has significantly reduced the cost and complexity of creating a flight capable system. Many universities and several high schools have launched CubeSats thanks to these advantages [2]. Low development and launch costs also provide an opportunity for cost effective flight testing of new and experimental technologies. In a notable example, the *Mars InSight* mission carried two CubeSats as a secondary payload in order to test new, miniaturized deep space communication equipment [5]. The expansion and promotion of CubeSats has also garnered interest from students at all levels in the space sector. Space agencies such as NASA and ESA are able to use CubeSats to inspire students to pursue an education and career in STEM fields.

Unfortunately, the rapid expansion of CubeSats has had negative effects as well. The large increase in satellites in LEO complicates flight paths and increases the risk of unintended collisions. Unlike larger satellites, CubeSats are generally designed without extensive maneuvering capabilities making it difficult to avoid collisions. Their small size also makes them difficult to track, further increasing the risk posed to other satellites. ESA has already experienced a collision due to CubeSat debris. The Sentinel-1A satellite was struck by debris from a CubeSat. A study conducted by NASA using the LEO to GEO Environment Debris (LEGEND) model estimated significant increases in the number of collisions caused by CubeSats in LEO. The increase is space debris due to satellite collisions would make operations in LEO more difficult. As the accessibility of CubeSats increases a greater effort will be necessary to ensure their operation and control is safe and secure.

1.3 Project Constraints

There are four major constraints on NeAtO:

1. The orbit profile
2. The primary propulsion system
3. The scientific payload

4. The satellite form factor

As mentioned in the introduction, NeAtO must enter and maintain a 250 - 600 kilometer orbit for as long as possible. This is to allow the scientific payload, the miniature Ion Neutral Mass Spectrometer (mini-INMS), to conduct atmospheric analysis in low Earth orbit. The mini-INMS is pictured in Figure 1. A Busek electro-spray thruster has already been selected as the primary propulsion system for this mission profile and cannot be changed. It is expected that NeAtO adheres to the CubeSat form factor, although the final size of NeAtO is flexible. The mini-INMS was developed at NASA's Goddard Space Flight Center. It is capable of detecting ions of densities between $10^3 - 10^8 \text{cm}^3$ and neutrals of densities between $10^4 - 10^9 \text{cm}^3$, with low energies between 0.1eV and 20eV. The apertures of the mini-INMS must be RAM facing to conduct measurements. The instrument occupies nearly 1.5U of space and has a mass of 560 grams [6].

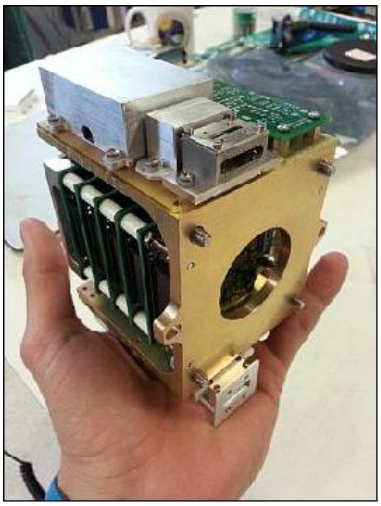

Figure 1: Miniature Ion and Nuetral Mass Spectrometer.

2 Background

2.1 Attitude Determination and Control

The purpose of the attitude determination and control subsystem (ADCS) is to properly position and orient the spacecraft to meet the needs of the mission. The ADCS is responsible for three distinct operations throughout the mission:

1. *Detumbling*
2. *Initial Attitude Determination*
3. *Attitude Maintenance*

Successful operation in all three phases is vital to the overall success of the mission. This is accomplished by combining a variety of sensors and actuators in a closed-loop control system.

2.1.1 Requirements

Each phase of the mission has different requirements. In order to successfully detumble, NeAtO must correct for the angular spin imparted during deployment. It is expected that such an angular velocity would not have a magnitude greater than ten degrees per second about any axis. Once NeAtO has reduced its angular velocity about two of its axes it is considered detumbled. NeAtO must then determine its orientation with respect to the Earth inertial frame. From this point onwards, NeAtO must maintain its attitude within plus or minus five degrees. As the orbit dips lower into the atmosphere the effects of drag become significant. Should the angular orientation deviate further than this limit the torques induced by atmospheric drag risk overcoming the strength of the on-board actuators, causing NeAtO to enter an uncontrollable spin. The torque exerted on NeAtO can be described as a function of atmospheric density, ρ, cross sectional area, A, velocity, V_{rel}, drag coefficient, C_D, center of pressure, c_p, and center of gravity, c_g, as shown in (1).

$$\tau_d = \frac{1}{2}\rho A C_D V_{rel}^2 (c_p - c_g) \tag{1}$$

Velocity is a function of the orbital velocity and the pointing angle. As the velocity increases, it is evident that the torque experienced by NeAtO increases as well. This relationship is shown in Figure 2.

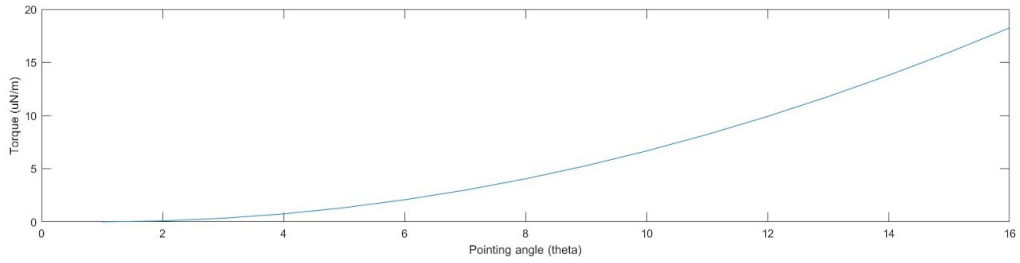

Figure 2: Pointing Angle vs. Torque.

2.1.2 Actuator Selection

Actuators are needed to effect change on NeAtO's orientation in space. Two different types of actuators are used to accomplish this, magnetorquers and reaction wheels. Magnetorquers create a torque using an external magnetic field and a magnetic dipole moment. A unique property of the magnetorquer is that it has no moving parts, unlike a reaction wheel that relies on moving parts. Due to this unique characteristic, magnetorquers are less prone to malfunction, and along with being cheap, lightweight and simple, are great for CubeSats. Magnetorquers are commonly made from a metal rod wrapped in a copper wire. When a current is run through the coil a magnetic moment is created. The interaction of this magnetic moment, ν, with the Earth's magnetic field, B, induces a torque, τ, on NeAtO as shown in (2).

$$\tau = \nu \times B \tag{2}$$

The magnetorquer chosen for this mission is the NCTR-M002 Magnetorquer Rod, which is manufactured by *New Space*. The NCTR-M002 uses a magnetic alloy rod that produces an amplification effect over an air cored magnetorquer. The NCTR-M002 consumes 200mW of power from a 5V power supply while producing a magnetic moment greater than 0.2 Am^2. The residual moment left over from the magnetorquer rod is negligible at less than 0.001 Am^2. Specific performance characteristics for the NCTR-M002 are listed in Table 1.

Table 1: Magnetorquer Parameters

Parameter	Value
Magnetic Moment	0.2 Am^2
Linearity	$< \pm 5\%$

Parameter	Value
Residual Moment	$< 0.005\ Am^2$
Dimensions	$70mm \times \varnothing 10mm$
Mass	$< 30\ g$
Power	200 mW
Operating Temperature	$-20°C$ to $60°C$
Vibration	14 g(rms)

In addition to the magnetorquers, reaction wheels will be included on-board NeAtO. Reaction wheels, also refereed to as momentum wheels, are internal components that store rotational momentum, providing satellites with three-axis control. By adjusting the momentum of a weighted wheel, the reaction wheels induce an equal and opposite torque on NeAtO, as the total momentum of the space craft must remain constant. Reaction wheels provide very precise pointing control without the fuel requirements of conventional attitude control thrusters. The reaction wheel chosen for this mission is the RWP050, pictured in Figure 4. This wheel can create a maximum torque of 0.007 Nm and create a total moment of 0.050 Nms while consuming less that 1W at full power. Operating parameters of RWP050 are available in Table 2.

Table 2: Reaction Wheel Parameters

Parameter	Value
Momentum	0.050 Nms
Mass	0.24 kg
Dimensions	$58 \times 58 \times 25$ mm
Voltage	10 - 14 V
Power	<1 W
Operating Temperature	$-20°C$ to $60°C$

2.1.3 Sensor Selection

NeAtO uses a collection of sensors to accurately determine its attitude and position relative to the Earth. A Global Positioning System (GPS) receiver allows NeAtO to determine its

Figure 3: NCTR-M002 Magnetorquer Rod.

Figure 4: RWP050 Reaction Wheels.

position above the Earth. This information is then used to predict the expected magnetic field and Sun vectors using well proven models. The NGPS-01-422, manufactured by *New Space*, is a great choice for the on-board GPS. It is relatively low powered and provides sufficiently accurate readings. The GPS module includes an external antenna, the NANT-PTCL1. Specifications for the GPS and antenna are available in Tables 3 and 4, respectively.

Figure 5: GPS Module and Antenna.

Table 3: GPS Parameters

Parameter	Value
Mass	<500 g
Power Consumption	1.5 W
Position	< 10 m
Velocity	$< 25\ \frac{cm}{s}$
Operating Temperature	$-10°C$ to $50°C$
Dimensions	$155 \times 76 \times 34$ mm

Table 4: GPS Receiver Parameters

Parameter	Value
Mass	<80 g
Power Consumption	80 mW
Frequency	1575.42 MHz
Bandwidth	20 MHz
Operating Temperature	$-25°C$ to $55°C$

Parameter	Value
Dimensions	$54 \times 54 \times 14.1$ mm
Active Gain (RHC)	> 16dBi
Noise Figure	< 2 dB

A gyroscope is also included to measure the angular rotation rates of NeAtO. The gyroscope chosen is the ADXRS453, manufactured by Analog Devices. This gyroscope consumes very little power and is very light weight.

Figure 6: ADXRS453 Gyroscope.

Table 5: Gyroscope Parameters

Parameter	Value
Mass	<56.7 g
Power Consumption	18.9 mW
Operating Temperature	$-40°C$ to $105°C$
Dimensions	$33 \times 33 \times 3$ mm

The LSM303AGR is a combination triple-axis accelerometer and magnetometer. It is small, lightweight, low cost, and power efficient compared to other accelerometer and magnetometer options. Magnetometer readings allow NeAtO to estimate its orientation based on the expected magnetic field given its position in orbit.

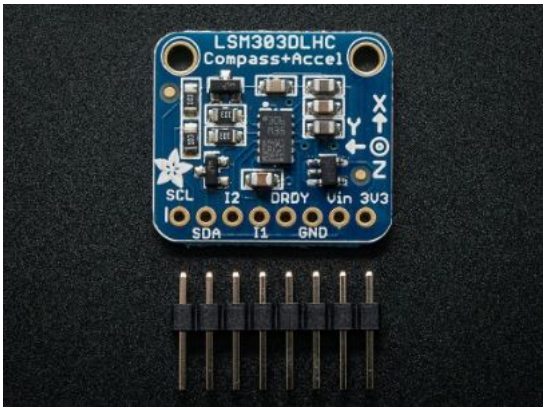

Figure 7: LSM303AGR Triple Axis Accelerometer and Magnetometer.

Table 6: Accelerometer and Magnetometer Parameters

Parameter	Value
Mass	10 mg
Power Consumption	5 mW
Operating Temperature	$-40°C$ to $85°C$
Dimensions	$2 \times 2 \times 1$ mm
Linear Acceleration	$\pm 0.01\%$
Magnetic Sensitivity	$\pm 1\%$

Sun sensors are used to determine the position of the Sun relative to NeAtO. This provides a high accuracy method for determining NeAtO's attitude. Two-axis fine Sun sensors measure the incident angle of the Sun in two orthogonal axes. A total of five Nano-SSOC-A60 Sun sensors are mounted to NeAtO. Five sensors are used to provide near total coverage such that the Sun is always in view of at least one sensor.

Figure 8: Nano-SSOC-A60 Sun Sensor.

Table 7: Sun Sensor Parameters

Parameter	Value
Mass	4 g
Power Consumption	10 mW
Operating Temperature	$-30°C$ to $85°C$
Dimensions	$27.4 \times 14 \times 5.9$ mm
Field of View	$\pm 60°$
Accuracy	$< 0.5°$
Precision	$< 0.1°$

2.1.4 Detumbling

All CubeSat deployment systems will induce some angular rotation to NeAtO as it is deployed. As such, NeAtO will immediately begin tumbling at the beginning of its mission, even before NeAtO can activate any sort of attitude control system. Generally, the angular velocity induced by the deployer is unknown and would interfere with the operation of NeAtO. It becomes necessary to detumble NeAtO before it can begin its main mission. One of the most common methods used to detumble satellites is the B-Dot control algorithm. The B-Dot controller detumbles a spacecraft by commanding a magnetic dipole moment based on the rate of change of the Earth's magnetic field, hence the term 'B-Dot'. The torque produced by the interaction of the magnetorquer's magnetic field with the Earth's magnetic field, as described in (2), acts counter to the angular velocity of NeAtO, slowly reducing its magnitude. If NeAtO's angular velocity, $\vec{\omega}$, is known, as provided by gyroscopes, the command torque, $\vec{m}$, can be expressed in terms of the gain, k, and magnetic field as shown in (4). The normal of the Earth's magnetic field is expressed as $\vec{b}$ in (3).

$$\vec{b} = \frac{\vec{B}}{\|\vec{B}\|} \tag{3}$$

$$\vec{m} = \frac{k}{\|\vec{B}\|}\vec{\omega} \times \vec{b} \tag{4}$$

It is assumed that the change in Earth's magnetic field due the change in orbital position over time occurs much more slowly than the change of the magnetic field in NeAtO's body frame due to the tumbling motion. The B-Dot control law is expressed in (5).

$$\vec{L} = \vec{m} \times \vec{B} \tag{5}$$

To make it simpler to prove the stability of the control law, (5) can be rewritten as (6).

$$\vec{L} = -k(I_3 - \vec{b}\vec{b}^T)\vec{\omega} \tag{6}$$

To prove the stability of the control law, it has to be shown to reduce a positive Lyapunov function asymptotically to zero. This is done showing that the derivative of the function is less than or equal to zero for all cases. For this application, the Lyapunov function in (7) is

used, where J is the spacecraft moment of inertia matrix. This function is analogous to the rotational kinetic energy. This is useful as the goal of the detumbling control law is to reduce the total angular velocity, and thus reduce the rotational kinetic energy.

$$V = \frac{1}{2}\vec{\omega}^T J \vec{\omega} \tag{7}$$

Applying the B-Dot control law written in (7), its derivative is calculated as (8).

$$\dot{V} = -k\vec{\omega}^T (I_3 - \vec{b}\vec{b}^T)\vec{\omega} \tag{8}$$

The eigen values of $(I_3 - \vec{b}\vec{b}^T)$ are always 0, 1, and 1, implying that $(I_3 - \vec{b}\vec{b}^T)$ is positive semidefinite. Therefore, $\dot{V}$ is always less than or equal to zero. The only case where $\dot{V}$ is equal to zero and global asymptotic stability cannot be obtained is when $\vec{\omega}$ is parallel to $\vec{b}$, in which case NeAtO is already not tumbling. In this case, NeAtO would only be spinning around one axis and the mission can move into the attitude determination phase. If it was necessary to use (4) in a B-Dot controller application, it can be rewritten into (9), where N is the number of magnetorquers on NeAtO, m_i^{max} is the maximum moment the i^{th} magnetorquer can deliver, and u_i is the direction of the magnetic moment for the i^{th} magnetorquer. This implementation is less computationally expensive than the other implementation, but less efficient in terms of power consumption.

$$\vec{L} = \sum_{i=1}^{N} -m_i^{\max} \text{sign}(\vec{u_i} \cdot \dot{\vec{B}}) \tag{9}$$

2.1.5 Attitude Determination

There are two main types of methods used to determine the attitude of a spacecraft: recursive and deterministic. Recursive methods work by estimating the current attitude from a previous attitude measurement. To estimate the current attitude quaternion, q_n, the previous attitude quaternion, q_{n-1}, is integrated. An alternative to recursive methods are deterministic methods. Deterministic algorithms estimate the current attitude quaternion by comparing a set of sensor observations with their expected values. Previous MQP teams used the deterministic TRIAD method to determine the attitude of their spacecraft. The TRIAD algorithm is well understood, reliable, and computationally efficient, making it an ideal choice for simple

attitude determination systems.

Harold Black's 1964 TRIAD algorithm was the first of its kind. It utilizes the Sun and Earth magnetic field vectors to determine a spacecraft's orientation. To implement this algorithm four pieces of information must be known: the Sun and magnetic field vectors in the spacecraft body-fixed frame, and the Sun and magnetic field vectors in the Earth-fixed reference frame [7]. The Sun and magnetic field vectors in the body-fixed frame are easily determined using the on-board magnetometer and Sun sensors. Values for the Sun and magnetic field vectors in the Earth-fixed frame can be determined using the position of the satellite in its orbit, provided by the GPS, and mathematical models. The World Magnetic Model (WMM) provides the Earth's magnetic field vector at any point in LEO given a satellite's longitude, latitude and altitude [8]. The Sun vector in the Earth-fixed frame can be calculated using the time of the year with a precision of 0.01° (36") for years between 1950 to 2050.

The TRIAD method allows for coordinates in the body fixed reference frame to be rotated into the Earth-fixed inertial reference frame. However, because vectors must be normalized for this method, it is possible that there could be a decent amount of sensor noise. Later research, specifically by Grace Wahba, poses a potential way of minimizing input sensor error, thus making the TRIAD method more accurate. Solutions to Wahba's problem include Davenport's q-method and QUEST. These methods provide and optimum quaternion for expressing the spacecraft's attitude.

The TRIAD method begins by defining two linearly independent body fixed vectors, b_1 and b_2, as well as their corresponding reference frame vectors, r_1 and r_2. The attitude matrix, A, is defined as a rotation from the body-fixed frame into the Earth-fixed frame in (10).

$$Ar_i = b_i \tag{10}$$

.

Ideally, A would be the same for both $i = 1$ and $i = 2$, however, due to noise in each sensor input, this may not be strictly true. For the TRIAD method to work, it is assumed that the transformation for $i = 1$ is significantly more accurate than its counter part. Two sets of orthonormal right-handed triads of vectors are defined, one for the reference frame, M_{ref},

and one for the body frame, M_{obs}.

$$M_{\text{obs}} = \left\langle r_1, \quad \frac{r_1 \times r_2}{|r_1 \times r_2|}, \quad r_1 \times \frac{r_1 \times r_2}{|r_1 \times r_2|} \right\rangle \tag{11}$$

$$M_{\text{ref}} = \left\langle b_1, \quad \frac{b_1 \times b_2}{|b_1 \times b_2|}, \quad b_1 \times \frac{b_1 \times b_2}{|b_1 \times b_2|} \right\rangle \tag{12}$$

A direct cosine matrix can be obtained by substituting M_{obs} and M_{ref} for b and r in (10). Thus, (10) can be rewritten as (13).

$$AM_{\text{obs}} = M_{ref} \tag{13}$$

Expanding the relation for each individual vector and simplifying the expression for the direct cosine matrix becomes (14).

$$A_{\text{DCM}} = b_1 r_1^{\text{T}} + \left(b_1 \times \frac{b_1 \times b_2}{|b_1 \times b_2|} \right) \left(r_1 \times \frac{r_1 \times r_2}{|r_1 \times r_2|} \right)^{\text{T}} + \left(\frac{b_1 \times b_2}{|b_1 \times b_2|} \right) \left(\frac{r_1 \times r_2}{|r_1 \times r_2|} \right)^{\text{T}} \tag{14}$$

This provides a mechanism for coordinates to be rotated from the body-fixed frame to the Earth-fixed, and back. It is important to note, however, that for cases where either the reference vectors or observed vectors are parallel or anti-parallel (14) becomes undefined.

2.1.5.1 Wahba's Problem

Mathematician Grace Wahba attempted to describe issues associated with using direction cosine matrices in attitude estimation and then provide a way to build upon the TRIAD method to deal with the discovered issues. Improvements included adding a way to weigh sensor measurements and allowing for more than two sets of measurements to be used.

To understand how to improve upon the TRIAD method, one must understand the problem Wahba proposed. Wahba's problem serves to find an orthogonal matrix with a positive determinant that minimizes the loss function in (15).

$$L(A) = \frac{1}{2} \sum_{i=1}^{N} a_i ||b_i - A r_i||^2 \tag{15}$$

This essentially finds the rotation matrix A that brings the first set of unit vectors $(b_1, b_2, \ldots, b_n)$ into the best least squares coincidence with the second set of vectors $(r_1, r_2, \ldots, r_n)$. Where b_i is a set of n-unit vectors in the spacecraft body frame, r_i is the corresponding set of vectors in the reference frame and a_i represents the non-negative weights of each sensor. These weights must be applied according to the accuracy of the sensors. This is necessary to relate Wahba's problem to Maximum Likelihood Estimation, a technique that uses sensor accuracy to help more accurately model a system. Using the orthogonality of A, the unit norm of the unit vectors and the cyclic invariance of the trace, we can rewrite $L(A)$ in (16).

$$L(A) = \lambda_0 - \mathrm{tr}(AB^{\mathrm{T}}) \tag{16}$$

Where

$$\lambda_0 = \sum_{i=1}^{N} a_i \quad \text{and} \quad B = \sum_{i=1}^{N} a_i b_i r_i^{\mathrm{T}}$$

Solutions to Wahba's problem include Davenport's q-method, Singular Value Decomposition, Quaternion Estimator (QUEST) and Estimators of the Optimal Quaternion (ESOQ), and start with the rewritten error estimation above. Through various real world applications, QUEST and ESOQ are deemed significantly faster than their robust counterparts.

2.1.5.2 Quaternions

For the orientation of NeAtO to be determined, controlled, or otherwise used, there must be a way of expressing it. A common way of doing this in other applications is with a sequence of Euler angles. This information can be determined and operated on as a three-element vector and is generally enough to uniquely determine the orientation. For some applications, especially spacecraft, three Euler angles cannot determine the orientation due to singularities in the determination when an angle is near 90 degrees. For this reason, spacecraft often use quaternions, which are four-element vectors with definitions for additional operations. Quaternions are composed of a three element "vector part" and a single "scalar part," and because of the additional element, there are no singularities when expressing orientation with quaternions [9]. Quaternions are also computationally easy to work with because most calculations can be done with quaternion operations instead of trigonometry. Just like most other attitude representations, quaternion attitudes can be converted to other systems,

such as Euler angle representations, which are more intuitive to visualize. Because of these advantages, quaternion attitude representations are great for a CubeSat mission, as the computations are less taxing for the on-board computer (OBC) and the lack of singularities allows for diverse mission profiles.

2.1.5.3 Davenport's q-method

Mathematician Harold Davenport proposed a solution to Wahba's problem that maximizes the following gain function, which is a quantity from (17).

$$g = tr(AB^T) \tag{17}$$

He specified a rotation matrix A that can be expressed in terms of Euler parameters in (18).

$$A = (q_0^2 - \epsilon^{\mathrm{T}}\epsilon) * [I_{3\times 3}] + 2\epsilon\epsilon^{\mathrm{T}} - 2q_0|\epsilon| \tag{18}$$

Where $\epsilon = (q_1, q_2, q_3)$, and q_0 is the scalar term of the quaternion. The gain function (19) can then be written in terms of the 4×4 K-matrix (20).

$$g(\bar{q}) = q^{\mathrm{T}}[K]q \tag{19}$$

$$K = \begin{bmatrix} \sigma & Z^{\mathrm{T}} \\ Z & S - \sigma[I_{3\times 3}] \end{bmatrix} \tag{20}$$

$$B = \sum_{i=1}^{N} a_i b_i r_i^{\mathrm{T}}$$

$$|S| = [B] + [B]^{\mathrm{T}}$$

$$\sigma = \mathrm{tr}([B])$$

$$[Z] = [B_{23} - B_{32}\ B_{31} - B_{13}\ B_{12} - B_{21}]^{\mathrm{T}}$$

In order to maximize the gain function, we must abide by the unit length constraint, which means we cannot set the values to infinity. Due to this, a Lagrange multiplier is needed to

yield a new gain factor, $g\prime$.

$$g\prime(q) = q^{\mathrm{T}}[K]q - \lambda(q^{\mathrm{T}}q - 1) \tag{21}$$

Differentiating and setting the equation equal to zero will find the extrema of the function (22).

$$0 = \frac{d}{dq}(g(q)) = 2[K]q - 2\lambda q \tag{22}$$

Which can alternatively be expressed as:

$$[K]q = \lambda_{max}q \tag{23}$$

Therefore, the desired Euler parameter vector is the eigenvector of the K-matrix. In order to maximize the gain, we have to choose the largest eigenvalue. Through substitution of (22) into (23), and applying some linear algebra, it is easy to show that:

$$g(p) = \lambda \tag{24}$$

2.1.5.4 QUEST

The QUEST method (QUaternion ESTimator) builds off of Davenport's Q method and allows for more frequent attitude computations. Currently, it is the most widely used algorithm for solving Wahba's problem and was first used in 1979 by the MAGSAT spacecraft. The algorithm begins by rewriting into two separate equations:

$$[(\lambda_{max} + \mathrm{tr}B)I_3 - S]\hat{q}_{1:3} = \hat{q}_4 z \tag{25}$$

$$(\lambda_{max} - \mathrm{tr}B)\hat{q}_4 - \hat{q}_{1:3}z = 0 \tag{26}$$

From there, we can rewrite q using the classical adjoint representation, knowing that the

adjoint divided by the determinant gives the inverse of a matrix, as shown in (27).

$$q_{1:3} = q_4((\lambda_{max} + \mathrm{tr}B)I_3 - S)^{-1}z = \frac{q_4(\mathrm{adj}((\lambda_{max} + \mathrm{tr}B)I_3 - S)z)}{\det((\lambda_{max} + \mathrm{tr}B)I_3 - S)} \tag{27}$$

From there, we can use the Cayley-Hamilton theorem, which states that every square matrix over a real or complex field satisfies its own characteristic equation [9], the previous equation can be rewritten in terms of the classical adjoint. This theorem holds for general quaternionic matrices, which is the case for this part of the QUEST algorithm and gives the optimal quaternion estimate as follows:

$$q = \alpha \begin{bmatrix} \mathrm{adj}(\rho I_3 - S)z \\ \det(\rho I_3 - S) \end{bmatrix} \rho = \lambda_{max} + \mathrm{tr}B \tag{28}$$

The term is determined by normalizing the resultant q. This equation assumes we know the value of $\lambda_{\max}$, which is the solution to the characteristic equation of the K-matrix. Finding the maximum eigenvalue of the characteristic equation is complicated for states using more than two observations. However, in the case of our CubeSat, we are only using two observations, which means the characteristic equation for K has a simple closed form solution. Several simplifications result from only having two observations, which ends up yielding the following equation for λ_{max}:

$$\lambda_{max} = a_1^2 + a_2^2 + 2a_1a_2[(b_1 \odot b_2)(r_1 \odot r_2) + ||b_1 \otimes b_2||||r_1 \otimes r_2||]^{\frac{1}{2}} \tag{29}$$

2.1.5.5 Estimator of the Optimal Quaternion (ESOQ)

The main problem with the QUEST method is the ambiguity caused by 180 degree rotations. In order to solve this problem, the method of sequential rotations can be used to effectively permute $\boldsymbol{q}$ components. However, this method is quite computationally expensive. Another way to remedy this problem, is to use the Estimator of the Optimal Quaternion, commonly referred to as ESOQ. This method avoids the need for explicit reference frame rotations, since it treats the four quaternion components more symmetrically than QUEST [9]. The ESOQ algorithm is similar to QUEST in regard to the numerical solution of λ_{max} using Newton-Raphson iteration of the characteristic equation of Davenport's K-matrix. The algorithm also locates the quaternion component with the maximum magnitude, like QUEST.

However, unlike QUEST, the ESOQ method is based on the fact that the optimal quaternion can be computed by normalizing any column of adj$H\lambda_{max}$, not just the fourth column. This realization is based off of an observation by Daniele Mortari, and enables the ESOQ method to be independent of an explicit reference frame. The method works by denoting H_k which is a symmetric 3×3 matrix that is obtained by deleting the kth row and the kth column from the original $H\lambda_{max}$ matrix. Since a quaternion has 4 components, k is indexed between 1 and 4.

$$\hat{q}_k = -\alpha' \det H_k \tag{30}$$

$$\begin{bmatrix} \hat{q}_{1:k-1} \\ \hat{q}_{k+1:4} \end{bmatrix} = \alpha'(\text{adj}H_k)\boldsymbol{h_k}$$

For the previous equation, $\alpha\,'$ is determined by normalizing the resulting quaternion, and the value of k is chosen by finding the largest diagonal element of $\det H_k$ of the adjoint matrix. It is interesting to note that QUEST and ESOQ would yield the same optimal quaternion if the ESOQ index was $k = 4$ assuming the value for λ_{max} was also the same. For our project application, we deemed using ESOQ as an unnecessary measure.

2.2 Command and Data Handling

The Command and Data Handling (C&DH) subsystem is composed of many different components and systems whose purpose is to manage NeAtO throughout its mission. This includes during the startup, deployment, detumble and general mission phases. The key components of the C&DH system are the on-board computer, flight software, radio and data storage systems. The C&DH system is responsible for all aspects of NeAtO's operation. It implements the control system, taking sensor data and issuing commands to the thrusters and ADC systems. It also collects, stores, and transmits sensor and payload data to ground stations. The individual component selection and overall architecture of the C&DH system is dependent on the overall needs of the mission and scientific payload.

2.2.1 Data Handling Requirements

The needs of the scientific payload drives the majority of the data handling requirements. Given the processing power of modern processors, little consideration needs to be given to the needs of the ADC system. The mission profile of NeAtO is relatively simple, requiring only basic slew operations, well within the capabilities of the on-board computer. In comparison, the payload will be producing large amounts of data that needs to be stored, possibly operated on, and eventually transmitted to ground stations. The payload produces data at 13.7 kbps. This data must be stored and eventually down-linked to a ground station. Based on the ground station coverage a total of 237.01 Mb of data may be downloaded each day. Thus, there must be sufficient storage space to collect data over the course of several orbits.

2.2.2 On-board Computer

We have selected the Clyde Space Kryten-M3 flight computer as our on-board computer (OBC). The Kryten-M3 is a flight proven OBC built around a Cortex-M3 processor core running at 50 MHz. Similar OBC's from Clyde Space were also chosen by previous MQPs. The OBC includes 8 MB of MRAM and 4 GB of bulk flash memory. Both the MRAM and bulk storage include automatic error detection and correction (EDAC). This is especially important for correcting potential errors introduced by the high radiation environment of orbit. The system also includes space for an external SD card, increasing the bulk storage capacity.

The Kryten-M3 is designed for use with FreeRTOS, a real-time operating system commonly used in high reliability embedded applications. FreeRTOS is highly configurable and capable of managing NeAtO and the scientific payload.

Figure 9: Kryten-M3 Onboard Computer.

3 ACS Analysis

3.1 Detumbling Simulation in Simulink

This section describes the analysis of our MQP's detumbling simulation, created in Simulink. Detumbling represents the first time NeAtO must actively control its attitude. Without successfully detumbling NeAtO after deployment the mission would not be able to continue to the next phase. The team simulated the detumbling of NeAtO using Simulink. This was done by simulating the on-board magnetometers and applying B-Dot control theory to actuate the magnetometers. An additional system utilizing NeAtO's gyroscope was also simulated.

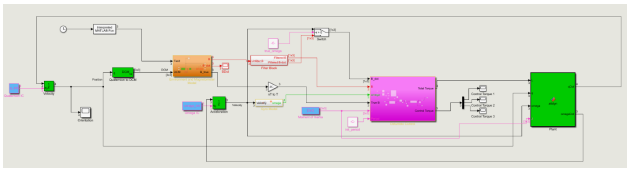

Figure 10: Total Detumbling Simulink Simulation.

Overall, the detumbling of NeAtO was accomplished by providing inputs into our custom Simulink function 'attdyn', or attitude dynamics. The attitude dynamics model block required an input control torque, quaternion, angular velocity, and moment of inertia, in order to output the rate of change of the quaternion $\dot{\boldsymbol{q}}$ and the rate of change of the angular velocity $\dot{\boldsymbol{\omega}}$. Moment of inertia was a constant input into the system and was given by the geometry of our NeAtO.

The attitude dynamics block uses the inputted quaternion to create a quaternion product matrix, denoted $\Xi(\boldsymbol{q})$, that speeds up the quaternion multiplication process.This is because $\Xi(\boldsymbol{q})$ is the same as $\boldsymbol{q}\odot$, which is a common quaternion operator.

From there, expressions for $\dot{\boldsymbol{q}}$ and $\dot{\boldsymbol{\omega}}$ can be derived.

$$\mathbf{L} = \begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix} \tag{31}$$

$$\boldsymbol{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} \tag{32}$$

$$\boldsymbol{\omega} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} \tag{33}$$

$$\Xi(\mathbf{q}) = \begin{bmatrix} q_4 & -q_3 & q_2 \\ q_3 & q_4 & -q_1 \\ -q_2 & q_1 & q_4 \\ -q_1 & -q_2 & -q_3 \end{bmatrix} \tag{34}$$

$$\dot{\boldsymbol{q}} = \frac{1}{2}\Xi_q \boldsymbol{\omega} \tag{35}$$

$$\dot{\boldsymbol{\omega}} = \frac{J}{\boldsymbol{\omega} \cdot J\boldsymbol{\omega}} \tag{36}$$

The quaternion and angular velocity inputs were first given by an initialized value from the simulations initialization file. Then, the outputted $\dot{\boldsymbol{q}}$ and $\dot{\boldsymbol{\omega}}$ got fed back into the Simulink loop, and integrated using a built in continuous integration block in Simulink. The resulting quaternion ($\boldsymbol{q}$) and angular velocity ($\boldsymbol{\omega}$), then got passed back into the 'attdyn' block.

Inputting the control torques into the 'attdyn' block was a little more complicated and required the simulation of the magnetometers and the application of B-Dot control.

3.1.1 Simulating Control Torques

The first step in finding the necessary control torques for our attitude dynamics block was to simulate the magnetometers. This was done with our custom Simulink function 'Environment and Magnetometer Model.' In order to use this function, we needed to take the inertial magnetic field reading and convert it to the body-fixed frame by multiplying it by NeAtO's body-fixed attitude.

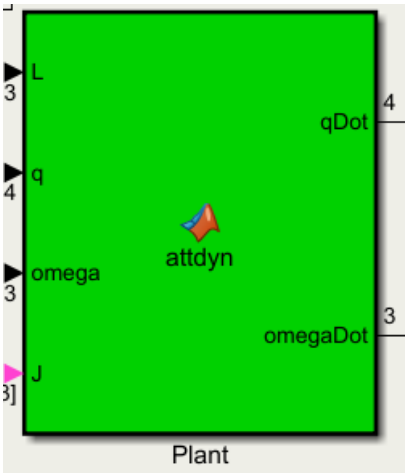

Figure 11: Attitude Dynamics Plant Schematic.

To find the inertial magnetic field vector, we began with a simple orbit propagator within an interpreted Matlab function, shown below:

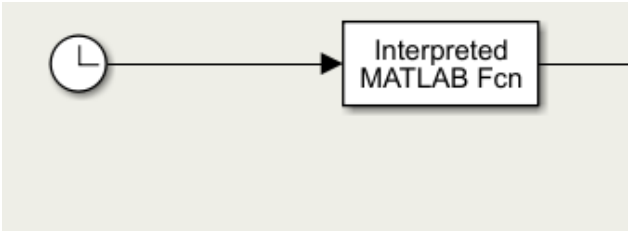

Figure 12: Simulation Path to Find Inertial Magnetic Field Vector.

These are the equations used within the interpreted Matlab function. Reference the *Simulink User Manual* in the appendix for more information on how this interpreted Matlab function was created.

$$\mu = 3.986 \times 10^{14}$$

$$a = \frac{r_a + r_p}{2}$$

$$p_t = 2\pi\sqrt{\frac{a^3}{\mu}}$$

$$n = \frac{2\pi}{p_t}$$

$$TSP = rem(\frac{t}{p_t})$$

After this, the mean anomaly, M, the specific energy, ϵ, and the eccentricity, e, are found.

$$M = nTSP$$
$$\epsilon = \frac{-\mu}{2a}$$
$$e = \frac{r_a - r_p}{2a}$$

After this, we then computed the numerical approximation of the Inverse Kepler Equation. First, we needed to find the eccentric anomaly, E. To do this, we initialized the values E_n and E_{nplus}, being 10 and 0 respectively. Then, we ran E_n and E_{nplus} through a while-loop that states as long as the absolute value of E_n and E_{nplus} is greater than 0.001, E_n and E_{nplus} equal each other, therefore, $E_{nplus} = E_n - (\frac{(E_n - \epsilon\sin(E_n) - M}{1 - \epsilon\cos(E_n)})$. Then, the output of the while-loop, E_{nplus}, is equal to the eccentric anomaly, E. Using E, we then found the true anomaly, ν, the distance, d, the angular momentum, h, and the orbital parameter, p.

$$\nu = 2\tan^{-1}\left(\sqrt{\frac{1+e}{1-e}}\right)\tan\left(\frac{E}{2}\right)$$
$$d = a(1 - e\cos(E))$$
$$h = \sqrt{\mu a(1 - e^2))}$$
$$p = \frac{h^2}{\mu}$$

From here, we then were able to compute the position components of the NeAtO.

$$x = d(\cos(RA)\cos(w + \nu) - \sin(RA)\sin(w + \nu\cos(i))$$
$$y = d(\sin(RA)\cos(w + \nu) - \cos(RA)\sin(w + \nu\cos(i))$$
$$z = d(\sin(i)\sin(w + \nu))$$

Using these, we found NeAtO's position vector:

$$\boldsymbol{P} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

From there, we are then able to find the velocity components.

$$\dot{x} = \frac{xhe}{dp}\sin(\nu) - \frac{h}{d}(\cos(RA)\sin(w+\nu) + \sin(RA)\cos(w+\nu)\cos(i))$$
$$\dot{y} = \frac{yhe}{dp}\sin(\nu) - \frac{h}{d}(\sin(RA)\sin(w+\nu) - \cos(RA)\cos(w+\nu)\cos(i))$$
$$\dot{z} = \frac{zhe}{dp}\sin(\nu) + \frac{h}{d}\sin(i)\cos(w+\nu)$$

From there, we can calculate the velocity vector:

$$\boldsymbol{V} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix}$$

Using ω_{earth}, which is equal to 7.29×10^{-5} rad/s, we then found the true anomaly at time t, θ_t.

$$\theta_t = \omega_{earth}t + \theta_{i_{earth}}$$

Next, we defined a normal rotation matrix that includes θ_t.

$$[RM] = \left(\begin{bmatrix} \cos(\theta_t) & \sin(\theta_t) & 0 \\ -\sin(\theta_t) & \cos(\theta_t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \right)$$

Finally, to find the position of NeAtO in ECEF, we transposed the product of the rotation matrix and the initial position vector defined above. Finding the velocity of the spacecraft in ECEF was a bit more complicated and is defined below.

$$[V_{ecef}] = RM \left(V \begin{bmatrix} \omega_{earthY} \\ \omega_{earthX} \\ 0 \end{bmatrix} \right)$$

After we found V_{ecef} and r_{ecef}, we then converted r_{ecef} from ECEF to Latitude Longitude Altitude (LLA).

To find the body-fixed attitude, the quaternion within the detumbling loop was fed into a built in Simulink function that converted quaternions to a direct cosine matrix. Once we had both the inertial magnetic field reading and the direct cosine matrix representation of NeAtO's attitude, a simple matrix multiplication yielded our true magnetic field in body fixed coordinates.

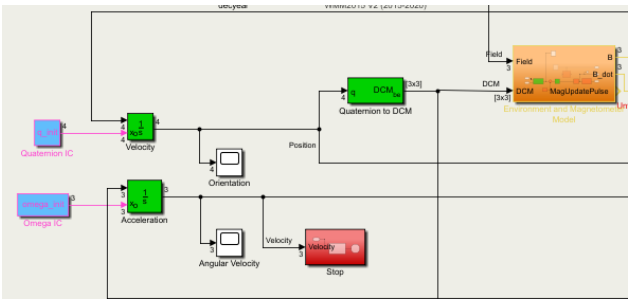

Figure 13: Integration of the q and omega vectors and Simulink q to DCM block.

This output was then seeded with noise via a noise maker created within Simulink to simulate how a real sensor would operate. The noise characteristics we inputted into the simulation are on par with those listed in the spec sheet for our magnetometer and provided a random Gaussian distribution that was added to the true magnetic field vector. Next, the magnetic field with the addition of noise was passed through a Simulink Quantizer that discretizes our input to a specific interval. For our case, the quantization value was set to 200, which was an arbitrary value that we found worked well after some trial and error. The quantization block works by using a quantization algorithm with a round-to-nearest method to map signal values to quantized values at the output that are defined by the quantization interval. After the vector is quantized, it is passed through a built in Simulink function called 'Zero-Order Hold.' This block essentially sets the sample rate of our signal to any specific value. We set our value to $\frac{1}{220}$, or 220 Hz, to be consistent with our sensor's refresh rate. For comparison purposes, a 'Discrete Derivative' block was added in order to see the unfiltered rate of change of the magnetic field, or $\dot{\boldsymbol{B}}$. This was to ensure that once we passed the magnetic field through our

low pass filter, the filter was properly working.

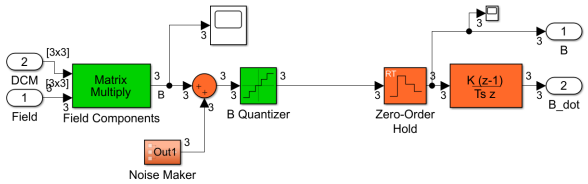

Figure 14: Contents of the Magnetometer Simulation Block.

The next step was to pass our outputted $\boldsymbol{B}$ vector from our 'Environment and Magnetometer Model' through a low pass filter to filter out the noise we previously added.

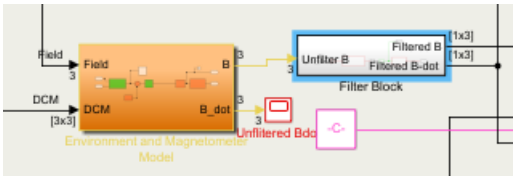

Figure 15: Magnetometer Simulation Block and Low Pass Filter.

Simulink allows for users to specify passband edge frequency and stopband edge frequency in Hz. Those values were set to 90 and 120 Hz, respectively. This was due to the sample rate of our magnetometer being 220 Hz. Any fast, small changes in magnetic field vector are most likely errors, so we want to remove any changes that are not happening slowly.

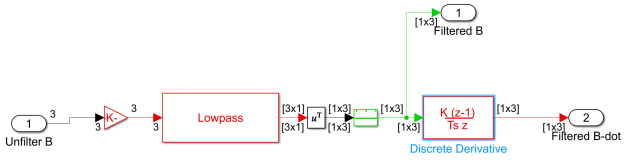

Figure 16: Contents of the Low Pass Filter Block.

In theory, this also means that if we were spinning fast enough, it would filter out that too, and make the satellite spin faster. However, because we are using gyroscope measurements as a second reference, this situation would not occur. The derivative of the signal was then found using another 'discrete derivative' block to give us the filtered rate of change of the magnetic field, or $\dot{\boldsymbol{B}}$.

3.1.2 Simulation of Control Algorithms

This is the subsystem that takes the filtered measurement readings and uses them to command control torques to detumble NeAtO. There are two different control laws that can be chosen in this block by setting a variable in the initialization file depending on the desired control method.

3.1.2.1 B-dot Control

The first control law block made for this model implements the control law described in (9) using an estimate of $\dot{\vec{\boldsymbol{B}}}$ and a preset maximum for magnetorquer dipole moment. The block sums up the individual commanded dipole moments from each of the magnetorquers and the finds the resultant torque using (4). This resultant torque, $\vec{\boldsymbol{L}}$, is then passed to the rest of the subsystem.

3.1.2.2 Proven-Stability B-dot Control

The second control law block, it is a direct improvement on the first control law because of the use of a direct measurement of the angular velocity. This block implements (6) to find a resultant torque given measurements from the gyroscopes and the magnetometers. Because this control law does not use $\dot{\vec{\boldsymbol{B}}}$, which is found by taking discrete differences of a corrupted signal, it is more effective than the other controller. In addition, as was shown in an earlier section, this control law is has proven Lyapunov stability. The advantages of this control law far outweigh the downside of having to also use the gyroscopes for this law, so this is the control law that is used for our analysis. The option of using the other law is still available, although discouraged.

3.1.2.3 Control Activation Delay

In order to meet any potential guidelines for launching from the ISS, this block gives us the option of delaying the activation of the active controls until a set amount of time has passed. This also has the benefit of allowing our controls to ignore the initial transient signals that result from differences between initial conditions and initial estimates upon simulation start up. This block is very simple and does not apply a signal delay to the commanded torques when they are allowed to pass. In addition, it does nothing to any disturbance torques that may be added to the simulation regardless of simulation time or set delay.

3.2 Reaction Wheel Sizing

The reaction wheels selected for NeAtO had to be capable of producing enough torque to counteract the disturbance torques on the satellite. First, the gravitational torque was defined by (37).

$$T_g = \frac{3M}{2R^3}|I_x - I_y|\sin(2\theta) \tag{37}$$

Where θ is equal to 5°, or 0.0872005 radians, M is the mass of the Earth, $3.9816 \times 10^{14}\frac{\mathrm{m}^3}{\mathrm{s}^2}$, and $\boldsymbol{R}$ is the radius of the earth, $6.978 \times 10^6 m$. The next disturbance torque needed to be accounted for was the torque due to solar pressure, defined in (38).

$$T_{sp} = F(C_{ps} - C_g) \tag{38}$$

Where C_{ps} is the center of solar pressure and C_g is the center of gravity. The flux due to solar pressure, F is defined in (39).

$$F = \frac{F_s}{c}A_s(1+q)\cos(l) \tag{39}$$

Where l is the angle of incidence to the Sun, F_s, is the solar flux constant , $1.367\frac{W}{m^2}$, c is the speed of light, $3 \times 10^8 \frac{\mathrm{m}}{\mathrm{s}}$, A_s is the surface area of the RAM facing surface, 0.02 m^2, and q is the coefficient of reflection, 0.6. Next we can then calculate the torque due to the magnetic field in (40).

$$\boldsymbol{T}_m = \boldsymbol{DB} \tag{40}$$

Where $\boldsymbol{D}$ is equal to the dipole moment of NeAtO and $\boldsymbol{B}$ is the Earth's magnetic field. The final disturbance torque to consider was the torque due to aerodynamic pressure. This was defined in (41).

$$T_a = \frac{1}{2}\rho C_d A_s V^2(C_{pa} - C_g) \tag{41}$$

Where V is the velocity of NeAtO and C_{pa} is the center of aerodynamic pressure. With

all of the disturbance torques calculated and summed, we now had a known value for the disturbance torques we needed to overcome. To find the torque for the reaction wheel sizing, T_{rw}, we evaluated (42).

$$T_{rw} = T_D(M_f) \tag{42}$$

Where M_f is the margin factor to help calculate the torque of the reaction wheel for the disturbance rejection and T_D is the reaction wheel torque for the worst case anticipated torque. The Reaction wheel torque, T_{rw}, must be equal to the worst case anticipated disturbance torque plus some margin. Finally, the momentum storage can be calculated with (43).

$$h = T_D \frac{t}{4}(0.707) \tag{43}$$

Where t is the orbital period in seconds and 0.707 is the root mean square average of a sinusoidal function.

3.3 Sun Sensor Orientation

Fine Sun Sensor (FSS) design combines a digital sensor with an analog component that aims to improve observation accuracy. In order to compute the Sun vector, a nonlinear transfer function is used to convert the telemetered FSS counts into Sun angle observations. Normally, two sensors heads, which is what NeAtO incorporates in its components, are combined to measure the Sun angle about orthogonal axes, which combined define the Sun unit vector in the sensor or body frame. Each sensor has an slit that passes a wedge of sunlight through it to a mask that leads to an array of photocells. The angle of the wedge of sunlight determines which slits are illuminated and which photocells are powered. This leads to the determination of the Sun angle. Normally, the accuracy of FSS is limited by the finite angular diameter of the sun, which is 0.53 degrees at the distance of the Earth.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = (1 + \tan^2\alpha + \tan^2\beta)^{-\frac{1}{2}} \begin{bmatrix} \tan\alpha \\ \tan\beta \\ 1 \end{bmatrix}$$

Where α and β are dihedral angles of the Sun vector about the sensor Y and X axes,

respectively. In other terms, they are the angles from the sensor's Z-axis to the projection of the Sun on the X-Z and Y-Z planes. It is assumed the sensor axes are orthogonal.

The sensor measurements are provided in telemetry as counts, N_α and N_β. These are converted to the alpha and beta angles using the transfer function shown above.

$$\alpha = \tan^{-1}(A_1 + A_2 N_\alpha + A_3 \sin(A_4 N_\alpha + A_5) + A_6 \sin(A_7 N_\alpha + A_8)) + A_9$$

$$\beta = \tan^{-1}(B_1 + B_2 N_\beta + B_3 \sin(B_4 N_\beta + B_5) + B_6 \sin(B_7 N_\beta + B_8)) + B_9$$

The expanded transfer function modifies alpha and beta to produce:

$$\alpha' = \alpha + A_{10} + A_{11}\beta + A_{12}\alpha\beta$$

$$\beta' = \beta + B_{10} + B_{11}\alpha + B_{12}\alpha\beta$$

In these equations, N_α and N_β are the telemetry as counts, as mentioned above. Also, A and B are the same value, just independent on which of the two Sun sensors is being measured. A_1 and A_2 are the bias and scale factor for the counts N_α and N_β. A_9 is a parameter that provides a bias for alpha directly. For small angles, A_1 and A_9 are basically identical, and A_9 is often just set to zero. In order to achieve A_3 and A_6, a range of values for $\tan\alpha$ must be plotted, where A_3 and A_6 are the amplitudes. These terms are unique to the FSS used on NeAtO. A_7 is normally set to be equal to two times the magnitude of A_4, which is found through initial guesses. A_{10}, A_{11}, and A_{12} all have geometric interpretations where A_{10} is the bias for the corrected angle $\alpha\prime$, usually similar in value to A_9. A_5 and A_8 are phase coefficients and are normally difficult to estimate and required the FOV of the FSS to be well sampled. It is worth nothing that the A_n and B_n values rely heavily on testing the FSS and usually are done right after the production of the FSS. For the purposes of our MQP, our simulation did not require this testing, as we were not required to order and test a real sun sensor, and were able to calculate the body sun vector in our simulation using a simpler method within Simulink and is described in detail further on in the appendix.

3.4 Inertial Sun Vector Estimation

The method used to determine the inertial Sun vector has a precision up to 0.01°, or 36 inches, for years between 1950 and 2050. To start we need to calculate n, which is the number of days since Greenwich Noon, Terrestrial time, on the 1st of January 2000.

$$n = JD - 2451545 \tag{44}$$

Where JD is the Julian date for the desired time. Once n was found, we calculated L, the mean longitude of the Sun, g, the mean anomaly of the Sun, and ϵ, the obliquity of the ecliptic.

$$L = 280.46^\circ + 0.9856474^\circ(n) \tag{45}$$

$$g = 357.528^\circ + 0.9856003^\circ(n) \tag{46}$$

$$\epsilon = 23.439^\circ - 0.0000004^\circ(n) \tag{47}$$

It is worth noting that $\boldsymbol{L}$ and $\boldsymbol{g}$ must be between 0° and 360°. If these values are not within this range, one can add or subtract 360° to the calculated value until a value between the desired range is achieved. Once we calculated the mean longitude and mean anomaly of the sun, we calculated λ, the ecliptic longitude of the Sun, assuming β, the ecliptic latitude of the Sun, is zero, and R, the distance from the Sun to Earth in AU.

$$\lambda = L + 1.915^\circ \sin(g) + 0.020^\circ \sin(2g) \tag{48}$$

$$R = 1.00014 \breve{} 0.01671 \cos(g) \breve{} 0.00014 \cos(2g) \tag{49}$$

With these values, we can finally calculate the equatorial coordinates of NeAtO with (50), (51), (52).

$$X = R \cos(\epsilon) \cos(\lambda) \tag{50}$$

$$Y = R\cos(\epsilon)\sin(\lambda) \tag{51}$$

$$Z = R\sin(\lambda) \tag{52}$$

3.5 Attitude Maintenance Simulation in Simulink

In order to simulate the attitude maintenance and control of NeAtO, a variety of different methods were used. Magnetometer and Sun sensor data had to be simulated in order to emulate the on board hardware. Inertial values for Sun position and magnetic field were also needed in order to be cross referenced within the QUEST algorithm. A control law was needed in order to use the attitude estimate to generate control torques. Finally, control torques, true position, angular velocity, moments of inertia and angular momentum were fed into the attitude dynamics plant to generate $\dot{\boldsymbol{q}}$, $\dot{\boldsymbol{\omega}}$, and $\dot{\boldsymbol{h}}$.

3.5.1 Simulating Inputs to QUEST

In order to estimate the quaternion, the algorithm requires inertial and body-fixed Sun sensor data as well as inertial and body-fixed magnetometer data. Obtaining the inertial readings is rather trivial. To simulate the inertial Sun vector, a custom Matlab function that takes the inputted time in Julian days. To find the time in Julian days, a Simulink time block that outputs seconds since the simulation started, is fed into a built in Julian date function that is initialized to the start of our mission. To find the inertial magnetic field vector, an interpreted Matlab function was used to find the position and velocity of the CubeSat at a specific time. That information was then used to find the magnetic field vector. To simulate the sensor readings in the body-fixed frame, an integration block is fed quaternion initial conditions. The resulting quaternion is then converted into a direct cosine matrix that transforms vectors from the geodetic Earth frame to the body axes. From there, both inertial Sun position and magnetic field were multiplied with the direct cosine matrix to rotate their reference frames from the geodetic Earth frame to the body frame. The resulting vector is a simulated version of the sensor readings with no noise. With all four of the necessary inputs, the QUEST block could now be used to estimate the attitude quaternion, $\hat{q}$.

3.5.2 Calculating the Control Torques

In order to point NeAtO within the given constraints, a controller was used to calculate the torque needed to rotate NeAtO to its required position. Reaction wheels were used to generate the control torque. Each wheel can provide a maximum torque and angular momentum before the wheel saturates. The controller block we designed takes $\hat{\boldsymbol{q}}$, angular momentum, h, angular velocity, ω, and the desired quaternion, $\boldsymbol{q_c}$, as inputs to the function. The control torque is found by using (53).

$$\boldsymbol{L} = -k_p(\delta\hat{q}_4)\delta\hat{q}_{1:3} - k_d(1 \pm \delta\hat{q}_{1:3}^T\delta\hat{q}_{1:3})\hat{\boldsymbol{\omega}} \tag{53}$$

Where, k_p is the proportional gain, k_d is derivative gain and δ_q is the error quaternion. Gains were tuned by running the simulation many times and changing values until values were found that worked consistently. An expression for the reaction wheel torque, $\dot{h}$ was also found. For this case,

$$\dot{h} = -\hat{\omega} \times h - \overrightarrow{L}$$

$\dot{h}$ is constrained to the maximum torque produced by the reaction wheels. If the calculation calls for it to be higher, the wheels cannot physically achieve this value and the controller is ran at maximum torque. Also, when the angular momentum inputted to the controller is greater than the maximum angular momentum of the reaction wheels and $\dot{h}$ is greater than 0, the control torque $\overrightarrow{L}$ is 0. This means that the reaction wheels are saturated, and cannot produce a control torque. Likewise, if the angular momentum inputted to the controller is less than the negative value of the maximum angular momentum and $\dot{h}$ is 0, the control torque $\overrightarrow{L}$ is zero.

3.5.3 Creating a Closed Loop System

The Attitude Dynamics block takes control torque, $\overrightarrow{L}$, position quaternion, q, angular velocity, ω moments of inertia, J and angular momentum, h, as inputs to the plant. After some math, the block outputs $\dot{q}$, $\dot{\omega}$ and $\dot{h}$. The attitude dynamics of the system were modeled as followed.

$$\dot{\boldsymbol{q}} = \frac{1}{2}\Xi_q\boldsymbol{\omega}$$

$$\dot{\boldsymbol{h}} = -\overrightarrow{\boldsymbol{L}} - \boldsymbol{\omega} \times \boldsymbol{h}$$

$$\dot{\boldsymbol{\omega}} = J^{-1}(-\dot{\boldsymbol{h}} - \boldsymbol{\omega} \times (J\boldsymbol{\omega} + \boldsymbol{h}))$$

Where Ξ_q is a quaternion product matrix defined as:

$$\Xi_q = \begin{bmatrix} q_4 & -q_3 & q_2 \\ q_3 & q_4 & -q_1 \\ q_2 & q_1 & q_4 \\ -q_1 & -q_2 & -q_3 \end{bmatrix}$$

These outputted values can then be fed back into the control loop and integrated to give updated q,ω,and h values.

3.6 Simulation in STK

Systems Tool Kit, or STK, is a physics-based software packaged created by Analytical Graphics (AGI). STK allows engineers to perform analyses of ground, sea, air and space objects in their respective environments. STK is used in government, commercial and defense applications around the world to determine the attitude, spatial relationships and time-dynamic positions of objects under many simultaneous constraints. STK's ability to simulate subsystems like ADC is what makes the software so valuable to the aerospace industry. Our team considered using STK to simulate the detumbling and attitude control of our 6U CubeSat. An example of the simulation can be seen below.

The figure above contains a snapshot of NeAtO in its simulated elliptical orbit. STK has the ability to overlay particular orbital elements as vectors, such as Sun vector and the nadir vector. Attitude determination in STK is done primarily with the use of quaternions. We originally intended on integrating directly between STK and Simulink. Simulink was to be used as our attitude determination and controller, whereas STK was simply to be used as a visual simulation, with the potential of utilizing it for sensor readings such as magnetic field and the Sun vector. Utilizing the work of Xiang Fang and Yunhai Geng of the Research Center of Satellite Technology of Harbin Institute of Technology [10] Fang and Geng laid out detailed steps into the integration of Simulink into a real-time running STK environment. Simulink was to be used to introduce sensor noise, whether that be from

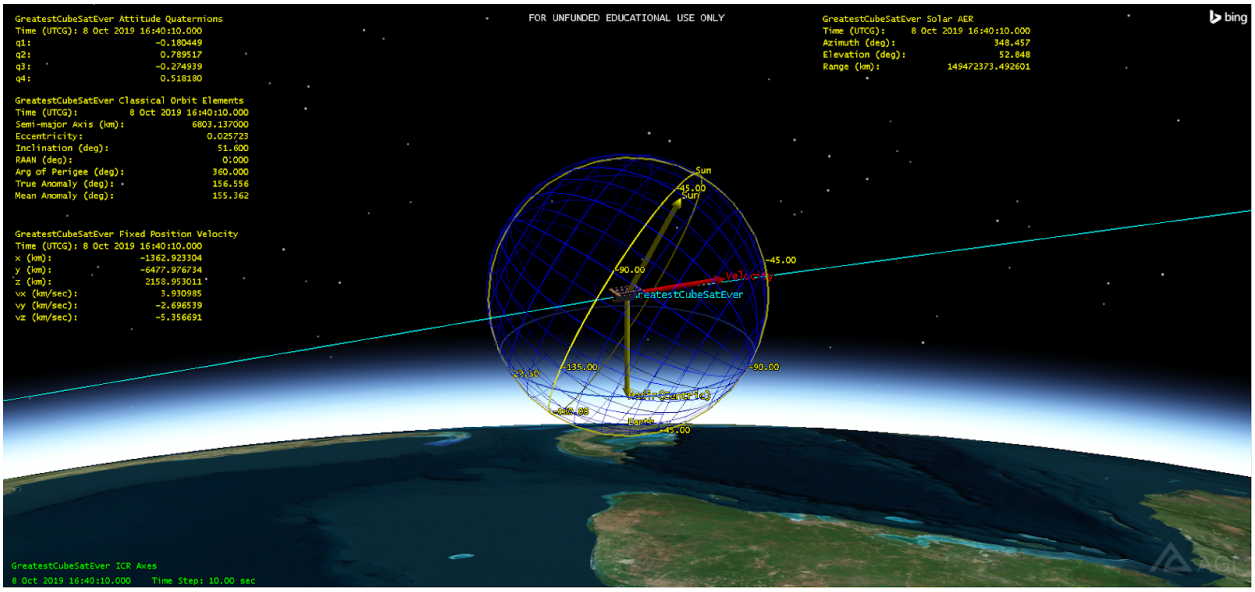

Figure 17: STK Satellite View.

Simulink simulated sensors, or STK sensors with their data passed into Simulink. We would then pass the noisy sensor data through an extended Kalman Filter (EKF) prior to being sent to the controller. This EKF-processed data could be compared to the full state vector to check on the functionality of our EKF. Attitude control commands would then be sent to the controller flow, and then pass the data back into the STK environment for visual simulation, completing the loop.

Although without any code snippets or a codebase to be referenced [11], it is possible to integrate Simulink into STK using the above figure and aforementioned paper as a guide. However, our focus changed from purely using Simulink for our attitude and orbital simulation and control, to primarily utilizing Simulink and its provided 'Aerospace Toolbox' into our control scheme to fully simulate NeAtO.

Since changing our focus to Simulink, the MATLAB to STK integration has become less realistic of a goal, especially with the introduction of the Aerospace Blockset CubeSat Simulation Library, which included an orbital propagator, which simulates in 3-D the orbital and attitude elements of a CubeSat. We will utilize this CubeSat Simulation Library along with using a Level-2 S-Function block, which will connect our Simulink controller to a STK environment, similar to what we had planned with MATLAB. Similar to our approach outlined above, Simulink will do all of the spacecraft control, and STK will merely model and return whatever sensor data we request.

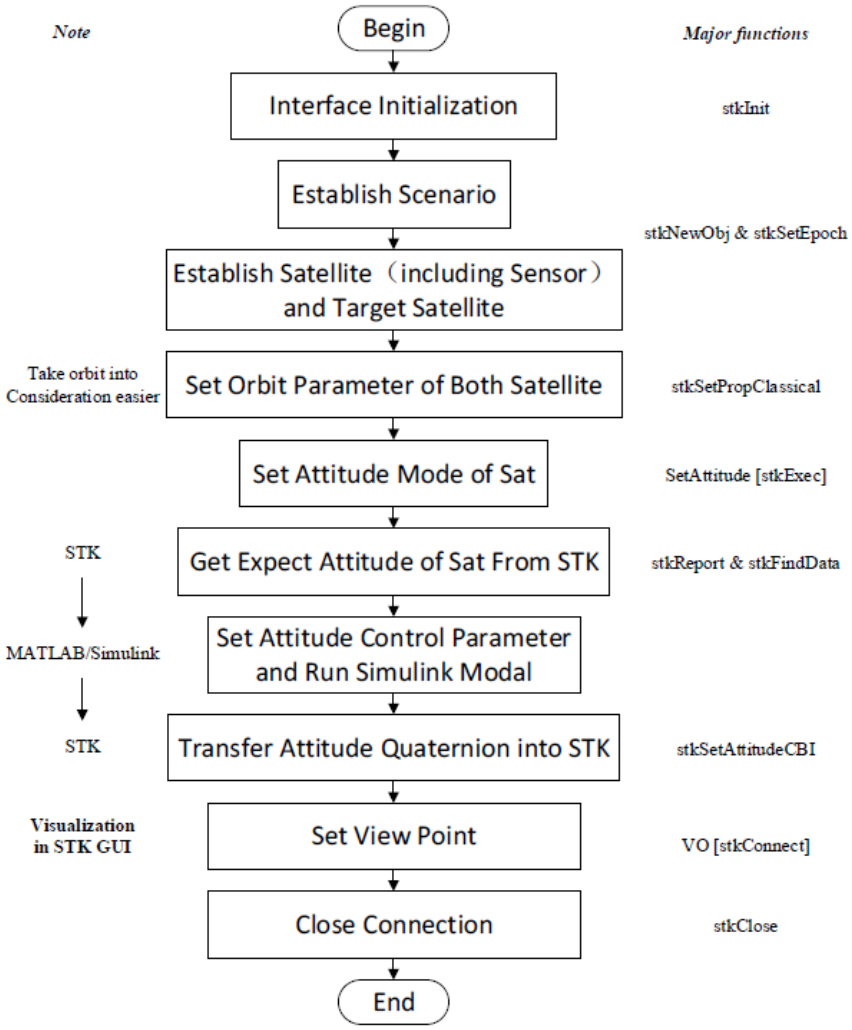

Figure 18: STK-Simulink/Matlab Integration Workflow.

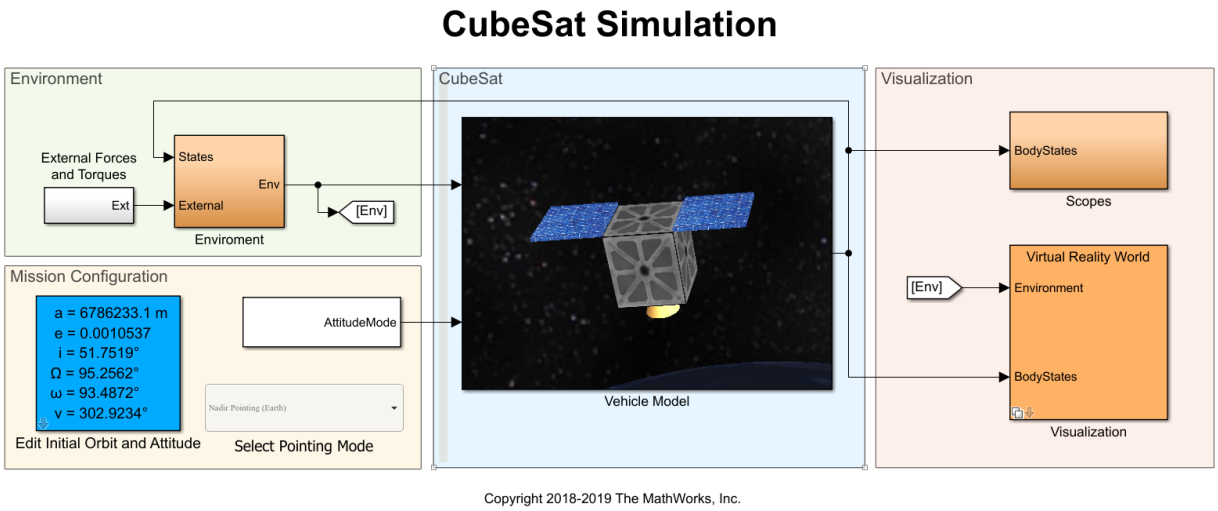

Figure 19: Simulink CubeSat Simulation Toolbox.

Our focus changed to using Simulink for all of our attitude and orbital determination and control. It was determined that the inclusion of STK for use of a simulation tool would be beneficial in presenting our attitude determination and control scheme in a more realistic and professional manner than the simulation provided in the Aerospace Blockset CubeSat Simulation Library. We also found that use of STK to simulate our magnetic field and sun vector sensors with a high degree of accuracy and processor efficiency.

Processes laid out by AGI to connect Simulink with STK were followed to establish a connection between Simulink and STK [12]. Once this connection is established, the STK block within Simulink checks for an existing STK environment, creating one if there is none already running as shown in the code block below. The connection is created using a MATLAB S-function, using input and output blocks to be used within Simulink. Every time step of the STK simulation outputs to Simulink for the use of Simulink noise addition, filtering and controllers. The input and outputs are selected based on the needs of our Simulink control method and simulation.

```
try
    root.CloseScenario();
    root.NewScenario('SimulinkAttitudeControl');
catch
    root.NewScenario('SimulinkAttitudeControl');
end
```

Unfortunately, we were unable to get past the first initialization step with the Simulink and STK integration due to server difficulties. We were able to establish a connection between the two pieces of software and get Simulink to properly initialize an STK environment in which to simulate our spacecraft. Our next steps were to establish STK outputs to give our simulated sensors accurate readings for magnetic field and Sun vector. These measurements were then to have simulated sensor noise added and then filtered through the use of a filter. The stopping point we experienced was due to a licensing issue with either Simulink and MATLAB or STK. We were then unable to establish a connection between Simulink and STK.

The variable matrix "stkParameters" is used to retrieve data from STK to be used within Simulink, where additional columns can be added for additional vectors, satellites, tracking stations or other STK objects. Future use of this Simulink and STK integration assuming resolution of the licensing issues we were confronted with, would include obtaining necessary sensor measurements such as Sun vector, magnetic field vectors, solar wind variables, occlusion times and whatever the current project may require. Unfortunately, with the license and version issues we experienced, and being unable to resolve these issues without the assistance of a server administrator and lots of troubleshooting, we were unable to achieve the desired outcome of a Simulink-STK integrated environment.

Future projects using Simulink that desire to integrate STK into their simulations should follow the examples provided by AGI that are paired with the functional examples provided in training videos [12]. Through the use of the available training material, future project teams will be able to utilize data which is native to the STK environment to simulate sensor data. Examples of such data include magnetic field vectors, Sun vectors, down-link availability, solar flux due to attitude and a multitude of other data.

4 Test Bed

4.1 Similar Work

Several previous Major Qualifying Projects (MQP) at WPI dealt with the development of a CubeSat test stand although none ever made it far beyond a preliminary design phase. The spherical air bearing and stand were initially designed by an MQP in 2017 [13]. The air bearing and stand are reused in this project. Following the recommendations of the 2017 MQP team, a separate MQP in 2018 improved upon the design of the test-stand. A new mounting plate was designed to allow for the precise adjustment of the center of mass of the system as well as to provide positive affixment for a 4U CubeSat model [14]. The team also designed a mock-up CubeSat model composed of a battery, micro-controller, and reaction wheel assembly. Figure 20 shows the extent of the progress made in 2018. The systems developed by the 2018 MQP team proved unsuitable to meet the goals set forth in this project and were thus not used.

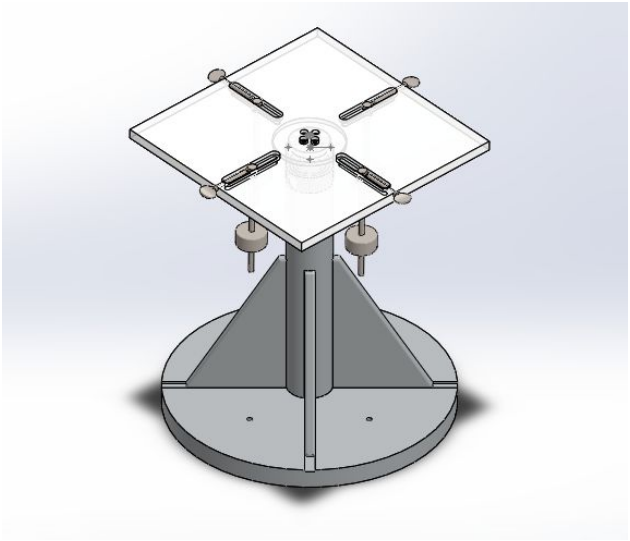

Figure 20: 2018 Test Stand.

Beyond the projects at WPI, multi degree-of-freedom test stands are used extensively in the test and validation of satellite control mechanisms. The complexity and capabilities of such test stands vary greatly depending on the intended application. For example, the Formation Control Testbed, developed by NASA's Jet Propulsion Laboratory is a 6 DOF satellite testbed for the development of formation flying control techniques [15]. On the other end of the spectrum much simpler platforms have been developed for use as educational tools,

such as the 3 DOF *Agilis* platform for simulating satellite dynamics [16].

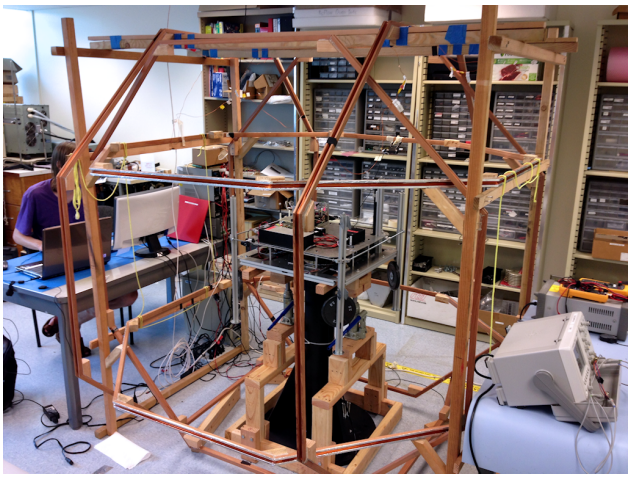

Figure 21: Hawaii Space Flight Laboratory ADCS Test Facility.

The Hawaii Space Flight Laboratory (HSL) ADCS Test Facility has developed a medium size test stand for educational applications. A platform similar to the Hawaii Space Flight Laboratory ADCS Test Facility, albeit at about a quarter of the size, would be ideal for this project. The air bearing and test stand used by the HSL provide 45 degrees of motion in the pitch and roll axis and full 360 degree in the yaw axis and is capable of supporting up to 450 kg [17]. Many aspects of the test stand design are inspired by the HSL project.

4.2 Approach

The goal of this project is to create a three degree-of-freedom (DOF) laboratory testbed that can be used in the analysis of control systems commonly used on spacecraft. The overall system will be an education tool for use in evaluating and demonstrating different spacecraft control schemes. The laboratory testbed can be divided into two primary subsystems:

1. The test stand providing 3 DOF rotational motion
2. The CubeSat model containing an actuation mechanism and control logic

These subsystems are independent of one another. The physical test stand simply provides a platform capable of rotation motion, where as the CubeSat model provides all the command and control functionality. Each component operates independently and can be modified or replaced without affecting the functionality of the other. To aid in the design process and ensure the project meets its goals a number of project constraints and objectives have been identified below:

1. Total project budget of $1250
2. Easily transported and setup
3. Capable of ±45° range of motion in the X and Y axis, and 360° in the Z axis
4. Easily adjustable center of mass, accommodating multiple potential configurations
5. Flexible mounting features
6. Wireless command and control interface
7. Operable with little training or knowledge of the underlying systems

To simplify the design and maintainability of the system as a whole it is desirable to use as many off the shelf components as possible. This is the driving factor in the selection of the motor and motor controllers, as well as the decision to use a Raspberry Pi as the on board computer. Consideration must be given to the future usability of the the system. The use of niche or poorly documented systems would make it difficult for someone else to continue this project in the future.

4.3 Design

4.3.1 3 DOF Platform

The test stand and spherical air bearing are directly reused from the 2018 MQP. This is the 'tee-ball' stand pictured in Figure 22. All other components pictured where purchased or manufactured specifically for this project. The approach to the design of the platform was heavily influenced by the HSL educational test facility.

Overall the design is rather simple and composed of four separate components: air bearing, platform, leg, and leg support bracket. All these components were manufactured in house out of aluminum. Detailed design drawings for each component are available in the appendix. Extensive work went into minimizing the mass of the system and simplifying the process of balancing the platform. The center of mass of the system must directly coincide with the center of rotation of the platform. Were this not the case, gravitationally induced torques would quickly overwhelm any control system. In a real satellite, this does not matter as there is no restriction on the location of the center of rotation. However in this system the center of rotation is always the center of the air bearing. The legs of the platform are constructed out of 80/20 aluminum extrusion so that additional balancing masses may be added, as shown in Figure 23.

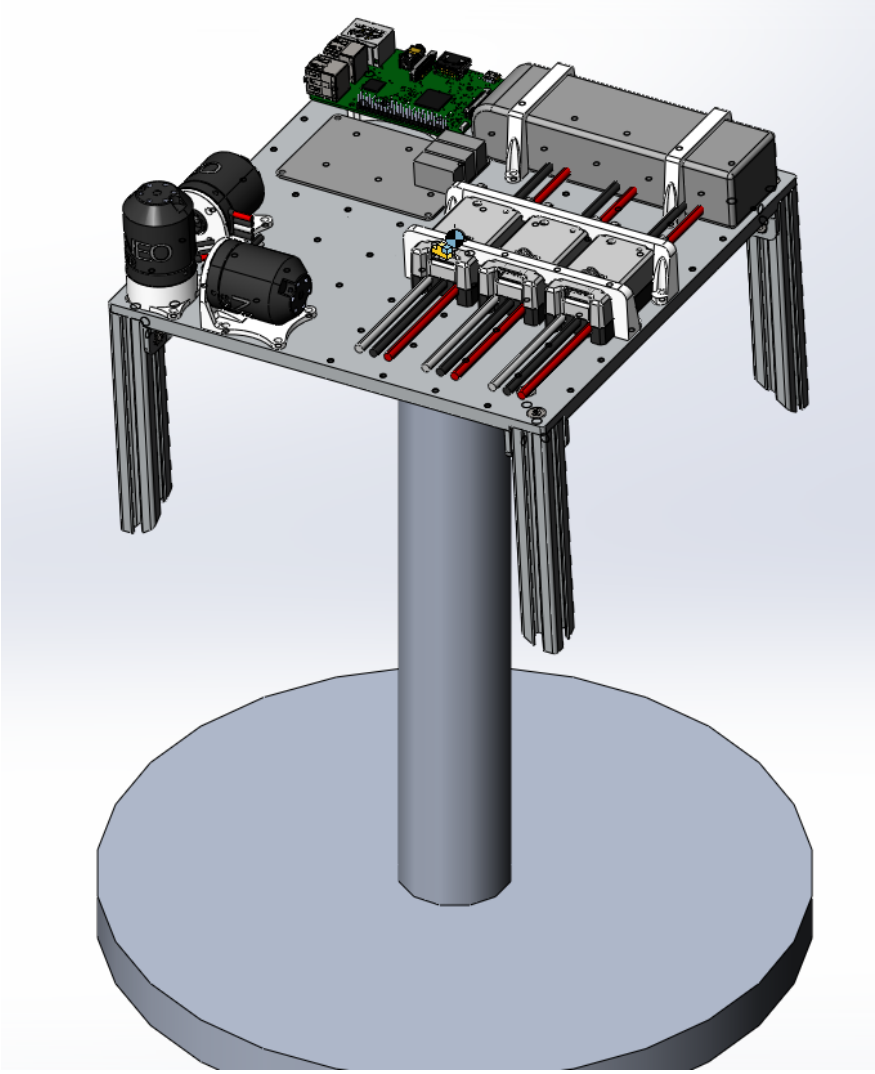

Figure 22: Test Stand 3D Model.

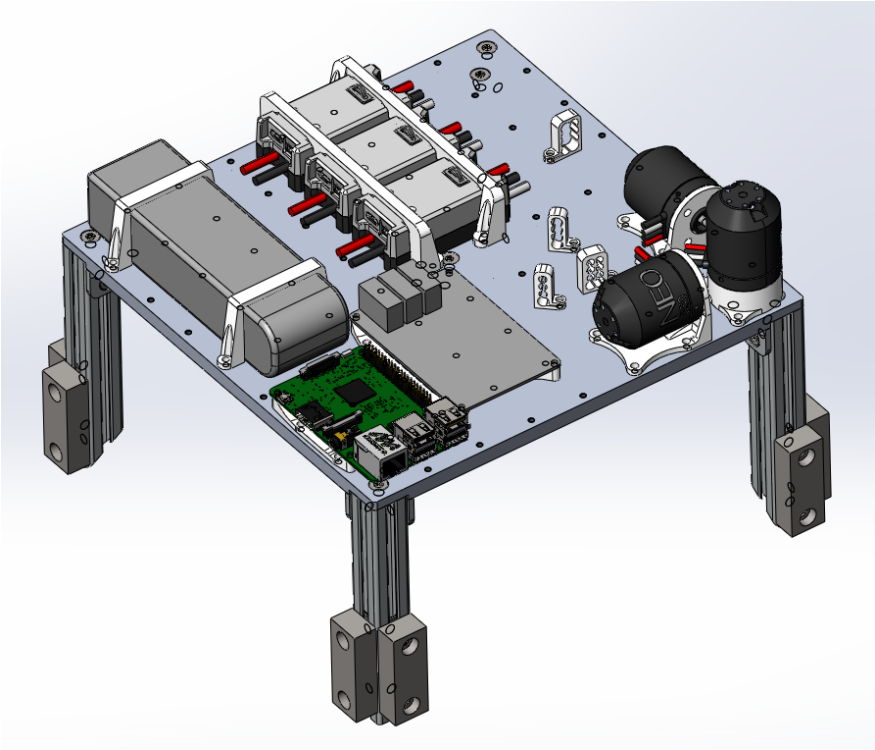

Figure 23: Test Stand Platform Model.

The platform features M3 holes regular spaced at 30mm intervals to provide a variety of mounting options. Figure 23 demonstrates how the holes are used to mount all of the necessary components for the system to operate. All the white components are 3D printed brackets screwed down to the platform.

4.3.2 CubeSat Model

The CubeSat model mimics the control system of simple satellite. The model is composed of many systems working together to create a functional controller. Solidworks was used to design the mounting brackets for each component as well as to determine the effects of each component on the system's center of mass and moments of inertia. Autodesk Eagle was used to design the custom control board. Figure 24 labels all of the main components that comprise the overall system.

The motor controllers and reaction wheel motors are the Spark Max and NEO 550 respectively. They are commercially available components from REV Robotics. While it would have been possible to design and assemble custom brushless motor controllers doing so is outside the scope of this project. The extensive documentation provided by the manufacturer makes the

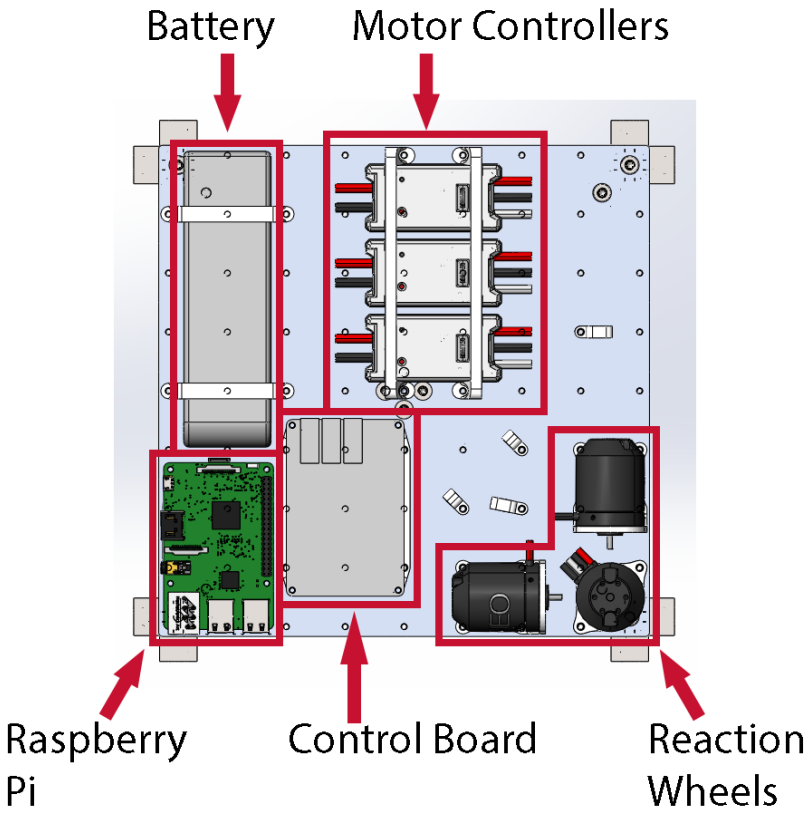

Figure 24: Labeled Platform Components.

integration and use of these components much simpler than if custom or no-name devices had been used instead. Control logic is provided by a Raspberry Pi 3 computer in conjunction with a custom control board that provides power management and hardware interfaces for the Pi to interact with sensors and motor controllers. The assembled control board is pictured in Figure 25. A list of the main integrated circuits on the control board is available in Table 8. The control board design and schematic is available in the appendix. Tables listing all the parts required to assemble the control board are available in the appendix as well.

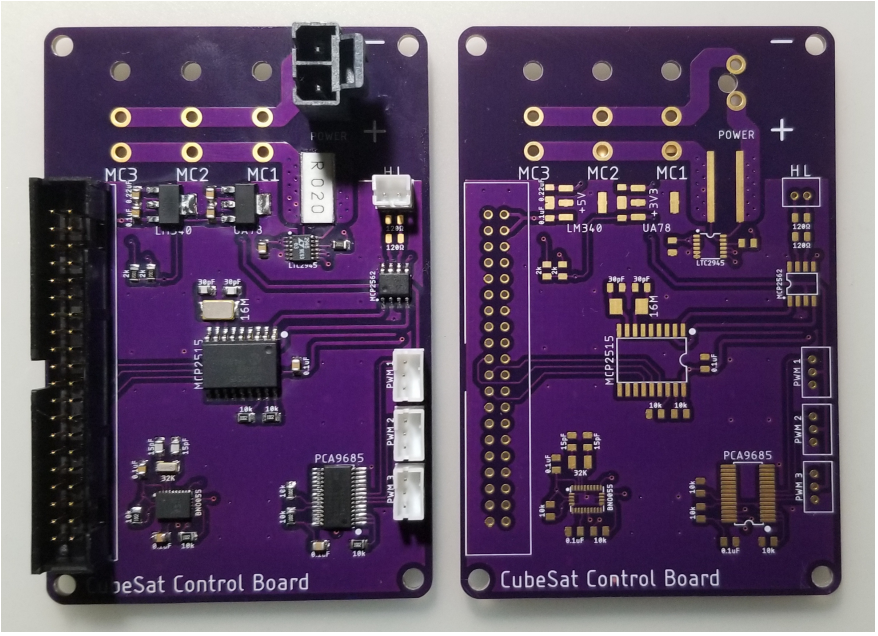

Figure 25: Control Board After and Before Assembly.

Table 8: Major Component List

Component	Role
MCP2515	CAN-BUS Controller
MCP2562	CAN-BUS Transceiver
PCA9685	PWM Driver
BNO055	Inertial Measurement Unit
LTC2945	Voltage & Current Monitor
L78S05CV	Linear 5V Regulator
UA78M	Linear 3.3V Regulator

4.3.2.1 Power & Sensing

Power to all components is provided by a 14.8V lithium polymer battery. The original 5 volt regulator proved incapable of providing the Raspberry Pi with enough current and accordingly had to be replaced, however the design has not yet been updated. A switching DC-DC buck converter would have been a wiser choice than the linear voltage regulator given the high current consumption of the Raspberry Pi 3. Should the control board be redesigned this is a necessary change. Molex Mega-Fit connectors provide convenient connections to the battery and motor controllers. Located near the power input connector is the LTC2945 voltage and current monitor. This integrated circuit provides high accuracy current and voltage measurement across a 20mΩ current sense resistor. With some simple algebra it is possible to determine the instantaneous current draw form the battery. Expressions for the current consumption and power of the system are given below, where V_{sense} is the voltage drop measured across the resistor R_{sense}. V_{bat} is the voltage measured at the batter.

$$I = \frac{V_{\text{sense}}}{R_{\text{sense}}}$$

$$W = V_{\text{bat}} I$$

Orientation and acceleration data is provided by the BNO055 IMU. Developed by Bosch Sensortec, the BNO055 is a highly capable combined 14 bit accelerometer, 16 bit gyroscope, and a geomagnetic sensor [18]. The IMU conveniently provides a sensor fusion mode that uses information from all three sensors to produce a filtered orientation quaternion, eliminating the need for complex data filtering and integration after the fact. The orientation quaternion is then used by the controller in determining the necessary control moments required to adjust the position of the platform. Communication with both the LTC2945 and BNO055 is accomplished using I2C.

4.3.2.2 Motor Control

The control board offers two different options for interfacing with motor controllers depending on the necessary applications. The PCA9685 is capable of driving up to sixteen independent PWM signals. Three of these are available on the control board through 3 pin JST-PH headers. The PCA9685 itself is controlled using I2C. Many commercial brushless motor

controllers will accept a PWM command signal. One disadvantage of PWM control is that it does not provide any sort of feed back. To address this issue, many higher end motor controllers, including those selected for this project, provide a CAN-BUS interface. CAN-BUS is a bi-directional message based protocol. The MCP2515 and MCP2562 work in concert to provide a CAN-BUS interface to the Raspberry Pi. However this proved unreliable for unknown reasons. Instead PWM control was used to communicate with the motor controllers.

4.4 Interface

In order to minimize potential disturbances to the testbed all communication with the system is conducted wirelessly. Upon startup the Pi hosts a web page that can be accessed via a local WiFi network. The interface allows the user to remotely control the testbed and collect sensor data. The web server is built using the Python web framework, *Tornado* while the client is written in javascript using *Vue.Js* to build the interface and *Skeleton* for style and layout. Direct client server communication is accomplish with WebSockets.

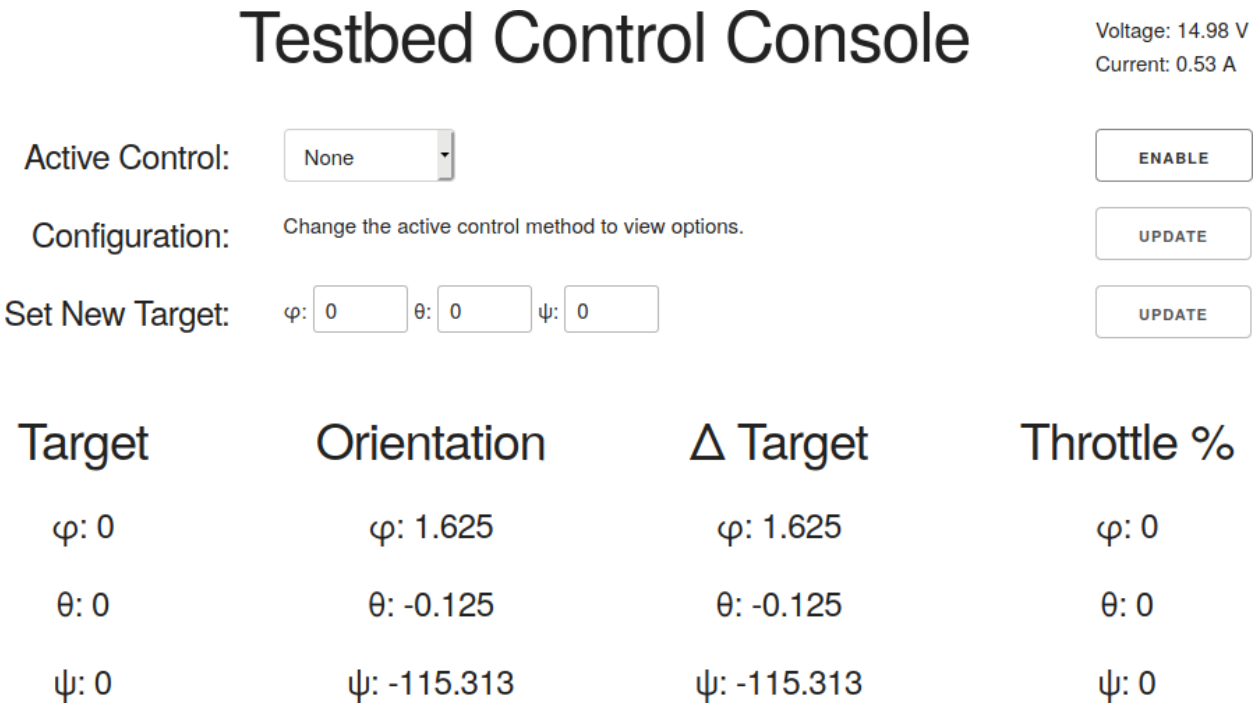

Figure 26: Testbed Control Interface.

The 'state' of the testbed is maintained on the server and pushed to the client as needed. Clients can send messages to the server that cause state changes. This is similar to the reducer pattern used in state management libraries such as Redux. This is useful as it minimizes the

amount of client-server traffic and simplifies the processing required by the client application. When ever the user makes a change on the client, such as selecting an option from the *Active Control* drop down the client sends a message to the server alerting it to the change. The server then sends an updated 'state' to all connected clients. This ensures that even when multiple clients are connected simultaneously they are consistent to one another. In the back ground the server polls the sensors on testbed and pushes updated values to the interface as well, ensuring the information displayed is always up to date.

4.5 User Guide

It is assumed the operator has some degree of familiarity with Linux and the command line interface (CLI). There are numerous resources available to familiarize oneself with the Raspberry Pi and the CLI online if needed. Familiarity with Python and Python virtual environments is recommended but not required. When setting up the Testbed in a new environment it is necessary to connect it to a local WiFi network, this network does not need to be connected to the internet, it merely provides access to the control interface. The easiest way to connect the Raspberry Pi to a network is to remove the SD card and add a file name `wpa_supplicant.conf` to the `/boot` directory. The contents of the file contains the WiFi network name and password.

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=<Insert country code here>

network={
    ssid="<Name of your WiFi>"
    psk="<Password for your WiFi>"
}
```

This method does not work on the WPI network, instead reach out to WPI ITS services for information on how to properly connect to the network. Once the Raspberry Pi is connected to a network it becomes possible to remotely login to it using SSH. On a computer connected to the same network SSH into the Raspberry Pi using the default port, 22. The username is `cube` and the password is `satellite`. Windows does not have a native SSH client and many third party options are available, one popular choice is PuTTY. Once logged in, starting the

web interface is simple. Navigate to the `~/code` directory and enter the command `venv`. This is a shortcut that activates a Python virtual environment with the packages necessary to operate the interface. The command `python main.py` will then start the web interface. Once started the web interface can be accessed from any computer connected to the same network by navigating to the Raspberry Pi's ip address in the browser followed by `:8080`. For example if the ip address was `192.168.1.6` then to access the interface enter `192.168.1.6:8080` in a web browser. The interface should appear as shown in Figure 27. Any number of clients can connect to the interface simultaneously.

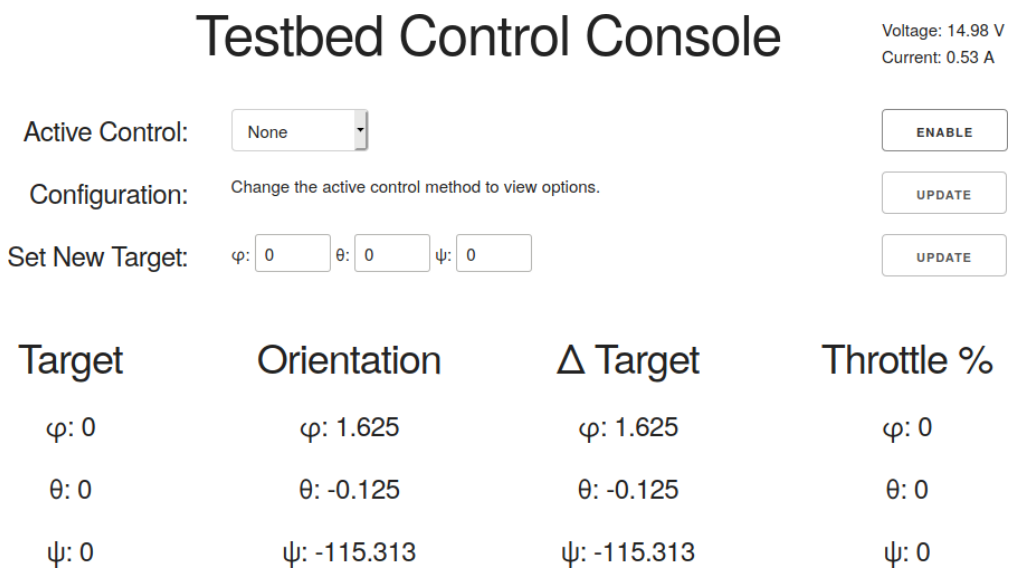

Figure 27: Testbed Control Interface.

This simple interface allows the user to view data from the testbed in real time and adjust the active control method. In total the interface provides four distinct components. Starting from the top and moving down they are:

1. Control Method and Enable Button
2. Controller Configuration
3. Target Settings
4. Data Displays

The `ENABLE` button enables and disables the entire system. Nothing will operate while the system is disabled, although sensor readings are still available. When enabled, the control method selected in the `Active Control:` drop down is used to determine the control algorithm used. Figure 28 shows how the interface looks when Bang-Bang control is enabled. Notice that when Bang-Bang is selected as the active control method, several fields are

introduced next to `Configuration:`. These fields are dependent on the active control method selected and allow the user to adjust the performance of the controller. Beneath the controller configuration are three fields next to `Set New Target:`. These fields are used to set the target state the controller is trying to achieve. Across the bottom of the interface are four data displays. Unless noted otherwise, all measurements are in units of degrees. From left to right there are as follows:

1. Target - Desired orientation
2. Orientation - Measured orientation by the strap-down IMU
3. Δ Target - Difference between the measured orientation and target
4. Throttle % - Commanded motor speed as a percent of max speed

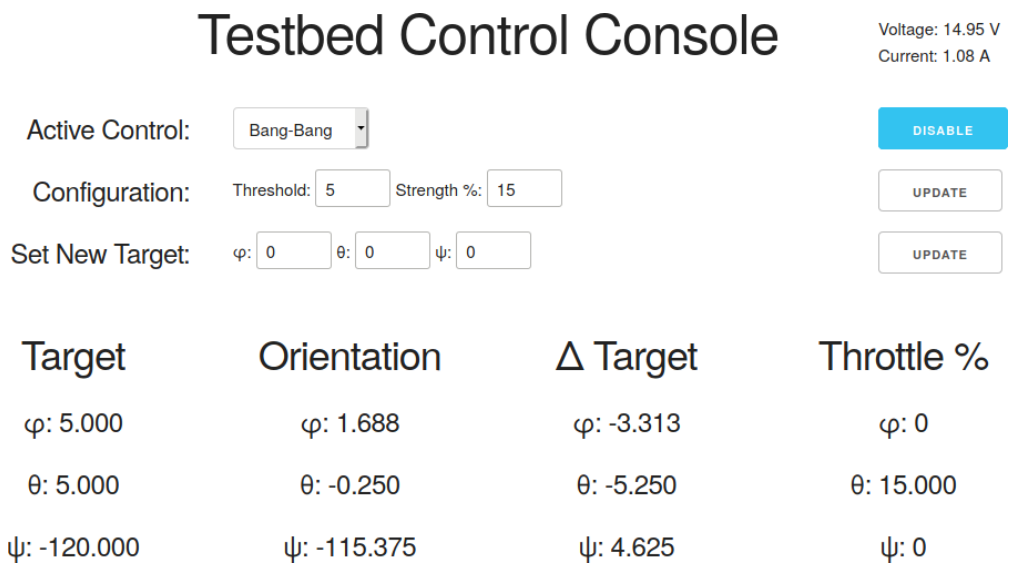

Figure 28: Testbed Enabled with Bang-Bang Control.

Figure 28 is useful in demonstrating how changes made to the configuration fields effect the performance of the system. The `threshold` value represents the minimum difference between the target and orientation positions required to activate the controller. When `threshold` is exceeded, `strength %`, determines the strength of the commanded actuation. The difference between the target and actual orientation in the θ axis is -5.250 degrees. This exceeds the set threshold of 5 and accordingly the controller commands a positive 15% actuation. There is no commanded actuation in the ϕ and ψ axes as the difference between the target and actual orientation in each case is less than the threshold value.

5 Results

5.1 Impacts

The continued expansion and development of CubeSat technologies has yielded a variety of positive social, economic, and educational effects. These impacts had to be considered throughout our design and control process. Firstly, a successful mission would result in new scientific data that can be used to further understand the state of Earth's atmosphere. Since CubeSat components and hardware have significantly reduced in cost and complexity, it was feasible for a group of university students to apply their educational backgrounds to the design of NeAtO. Low development and launch costs also provide an opportunity for cost effective flight testing of new and experimental technologies, which was something this project explored and could realistically be implemented in the future. The expansion and promotion of CubeSats has sparked interest from students of all levels in the space sector. This unique opportunity provided team members with a chance to develop an interest in this type of work going forward. Unfortunately, the rapid expansion of CubeSats has had negative effects as well. Due to the drastic increase in CubeSats in low earth orbit, space debris and collisions with debris are significant problems. Unlike larger satellites, CubeSats are generally designed without extensive maneuvering capabilities making it difficult to avoid collisions. Their small size also makes them difficult to track, further increasing the risk posed to other satellites. These were considered in the development of this project, but while precautions can be taken, there is no way to stop something like this from happening.

5.2 Time to Detumble

With the selected magnetorquers with maximum dipole moment of 0.2 Am^2, it takes 2010 seconds reduce the norm of the angular momentum of NeAtO to 1 g $\frac{m^2}{s}$, which is shown in Figure 29.

The y-axis of the plot is the norm of the angular momentum of NeAtO in g $\frac{m^2}{s}$, while the x-axis is the time since the beginning of detumbling, in seconds. This shows the success of the controller with these magnetorquers as the angular momentum steadily decreases until it reaches a certain point at which it asymptotes. The same behavior is shown in the angular velocities, depicted in Figure 30.

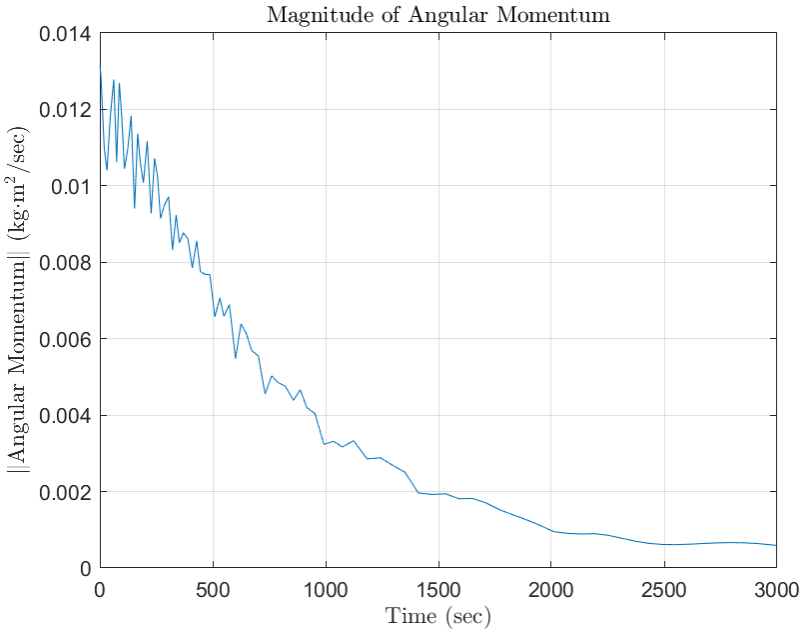

Figure 29: Angular Momentum with Small Magnetorquer.

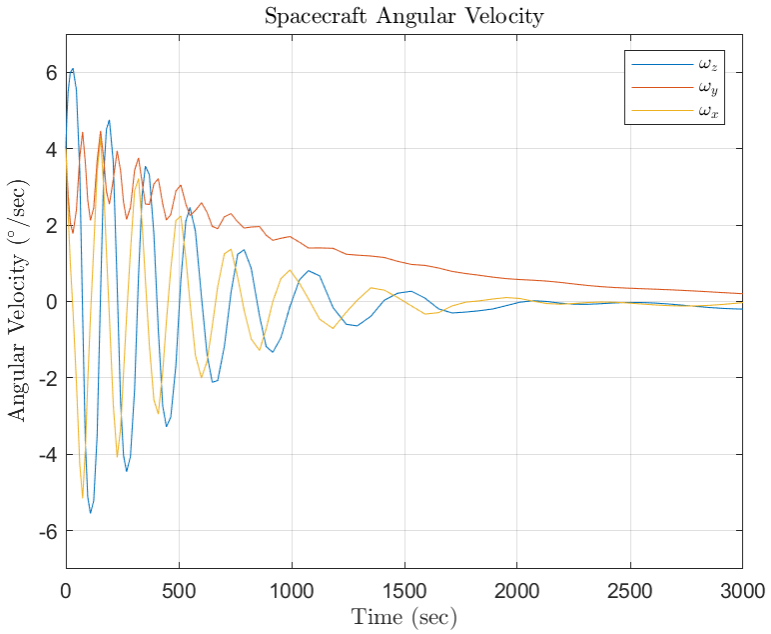

Figure 30: Angular Velocity with Small Magnetorquer.

It is shown in Figure 30 that the magnitudes of the angular velocities are slowly decreased to zero over the course of the simulation. If we change the magnetorquers to stronger ones which operate with a maximum dipole moment of 1.2 Am^2, it takes 2122 seconds to reduce the norm of the angular momentum of NeAtO to 1 g $\frac{m^2}{s}$, which is actually worse. However, it does reduce the angular momentum even further, although this is not necessary for our application. This is shown in Figure 31, which looks very similar to the case of the smaller magnetorquers.

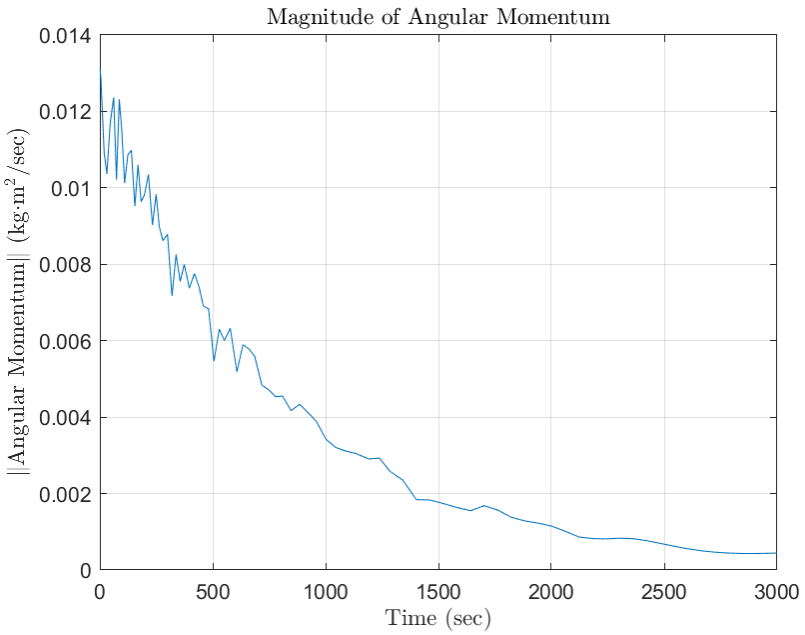

Figure 31: Angular Momentum with Large Magnetorquer.

These results are not much better than the smaller magnetorquers. While angular momentum asymptotes at a lower value, the decrease does not offset the increase in mass and power usage. This is also seen in Figure 32, where the stronger magnetorquers do not reduce any specific angular velocity anymore quickly, so there is no real benefit to this. If it were possible to detumble one of the axes significantly more quickly, there may have been a benefit.

This would use more power than the smaller magnetorquers, for a worse detumbling time, so the smaller magnetorquers are the better choice for this mission. Either way, we are able to detumble NeAtO within one orbital period, which allows for the attitude determination and control to begin quickly. This means NeAtO will not run out of power before orienting itself to collect solar power, so the mission will be able to continue after deployment from nanoracks. The model for filtering the noise is also successful, as the model that includes

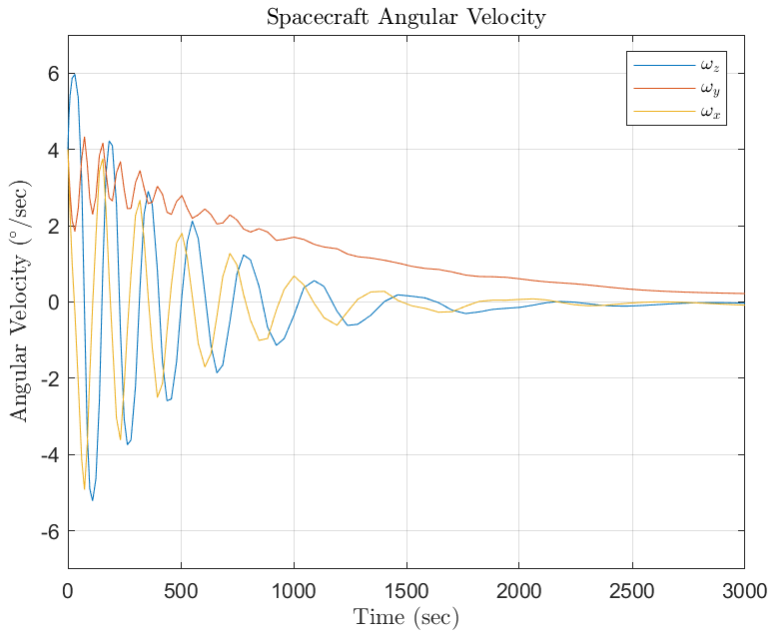

Figure 32: Angular Velocity with Large Magnetorquer.

and then filters the noise also successfully detumbles from the initial conditions. One of the main reasons this is successful is that the control law uses the output of the gyroscopes for the angular velocity along with the measurements from the magnetometers for the magnetic field vector. By using this method, we avoid having to differentiate noisy data, which means that the filter does not have to eliminate all of the noise. Because this is done with lowpass filters rather than an analytical or statistical method, the filtering could be implemented in the hardware, which frees up computational time. Overall, the detumbling is a short portion of the mission, and our models have shown that our control design for NeAtO is able to accomplish its goal of detumbling in less than one orbit with our chosen actuators and sensors.

5.3 Attitude Estimation using QUEST

Figure 33 shows the four attitude quaternion elements and how, over the course of the first 15 seconds of the simulation, the rate of change of each element becomes zero. This plot demonstrates that the QUEST attitude control method is functional and efficient with respect to time. QUEST has minimal overshoot with the quaternion elements, which also proves that QUEST is an efficient method for attitude estimation for our satellite.

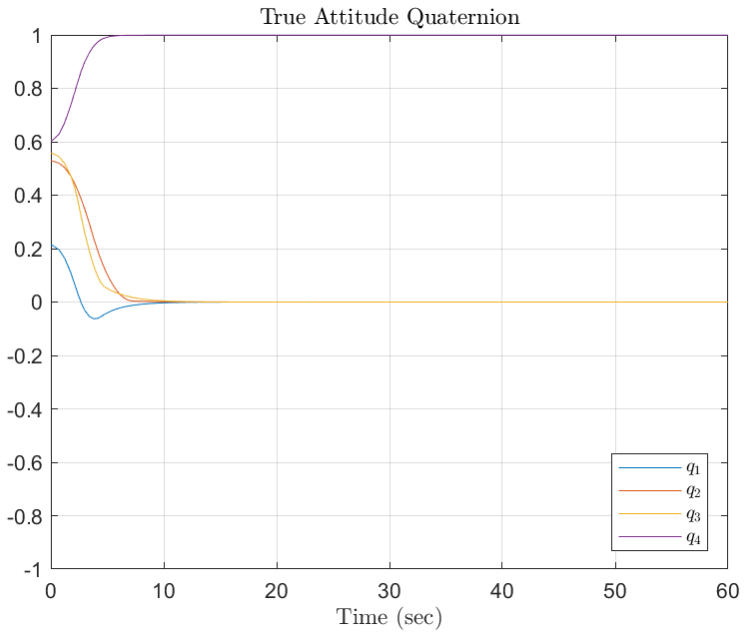

Figure 33: Quaternion Components.

The angular velocities can be seen in Figure 34 with their random initial values due to the tumbling of the spacecraft immediately following deployment. After the attitude determination and control simulation of the spacecraft begins, reaction wheels are used to incur angular velocities along each axis. When the desired pointing requirements have been achieved, the simulation arrests the angular velocities and all elements settle on zero. This can be seen by the zero-slope plot lines at 15 seconds.

Figure 35 shows that the estimate of the quaternion from QUEST is sometimes off by a negative. However, this does not have any significant effect on the control of NeAtO because the controller considers quaternions representing equivalent orientations to be the same.

Figure 36 shows that the maximum torque never exceeds the limits of the wheels because we limit the output torque in our simulations to emulate physical restraints of the chosen reaction wheels. These limits may be changed to emulate different reaction wheels or to emulate differing power constraints which would lower the torque that could be applied. The peaks and valleys of the individual plotlines can be interpreted in the way that the torques would have to reverse to dampen torques that were transferred to the vehicle in the opposite direction. Once the vehicle is brought to a near steady state, the torques required to maintain pointing are minimal and draw little to no power.

Figure 37 shows that the maximum angular momentum never exceeds the limits of the reaction

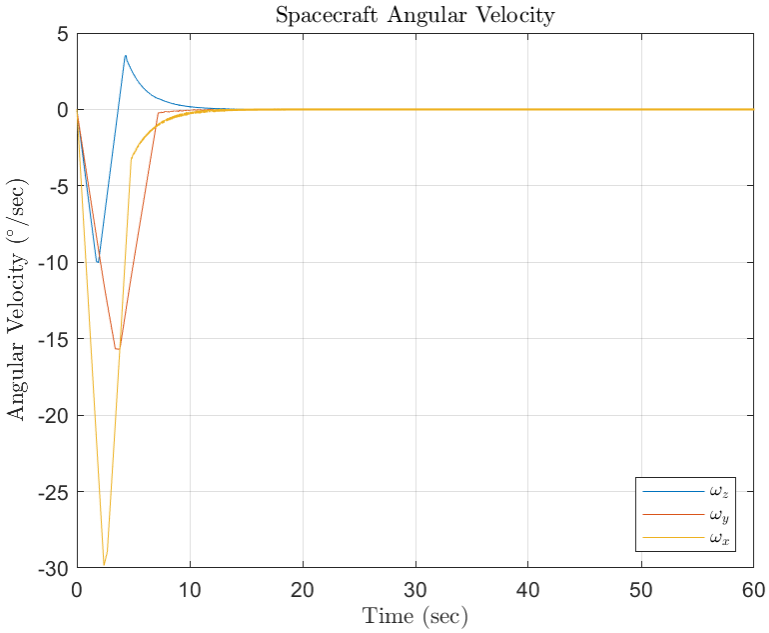

Figure 34: True Angular Velocity.

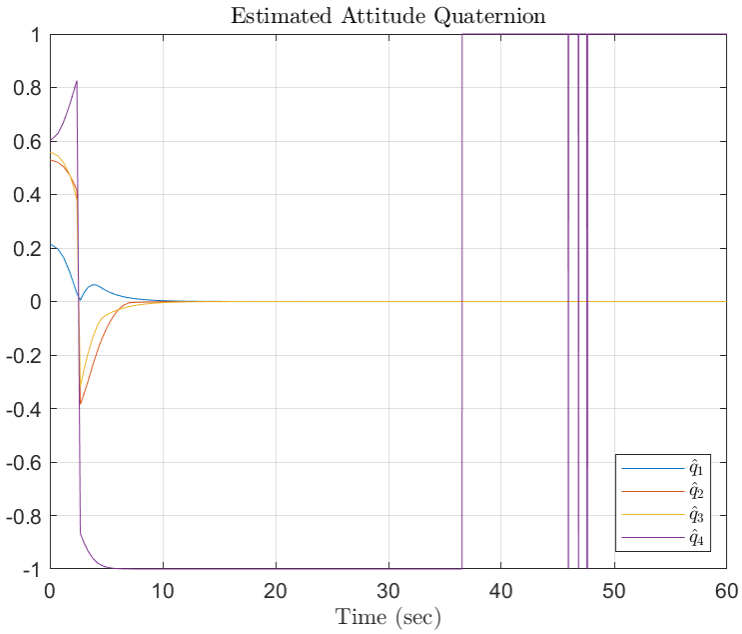

Figure 35: $\hat{\boldsymbol{q}}$ from QUEST Algorithm.

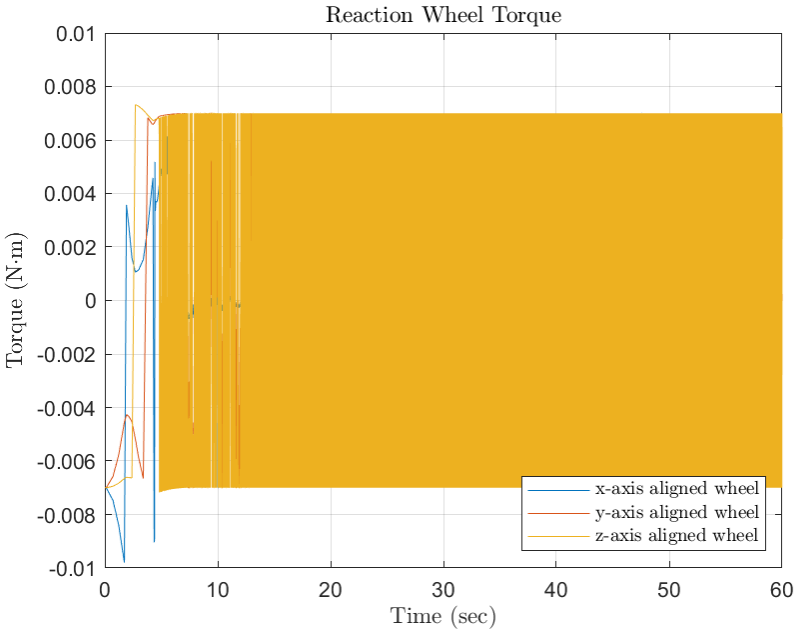

Figure 36: Control Torques.

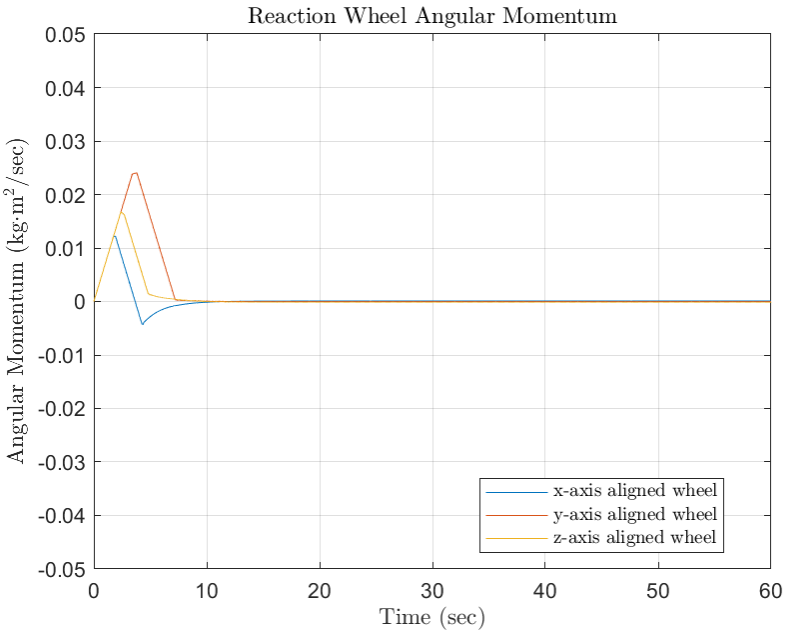

Figure 37: Reaction Wheel Angular Momentum.

wheels, although they do saturate during the maneuver. The wheels can be desaturated by dumping momentum with the magnetorquers. Disturbance torques are not an issue as long as they are significantly smaller than the maximum torque the wheels can provide. Such small disturbance torques have little effect on the controller and need not be accounted for. Changing the gains for the controller drastically affects the pointing time of the satellite. With considerable testing and tweaking of the gains in the system, we came to the conclusion that with gains of 12.5 for both K_p and K_d, we were able to minimize our pointing time to less than 15 seconds. Whereas both smaller and higher values for K_p and K_d result in higher times, our testing peaked around 22 seconds, notably higher than our final choice of gains. Should the moments of inertia of the satellite change or the reaction wheels be switched out this analysis must be repeated to determine new values for K_p and K_d.

6 Conclusion and Future Work

6.1 NeAtO Analysis in Simulink

The Simulink model created by the team is fully functional and will simulate detumbling and attitude control and determination of any small satellite similar in size to NeAtO. Following the *Simulink User Manual*, located in the appendix, another team can easily recreate the simulation and modify the initialization file to fit the specifications of their small satellite. The simulation is able to measure how long it takes NeAtO to detumble, which in the case of our magnetorquers is 2010 seconds, or around 33.5 minutes. This is well within the one-orbit requirement our group set for ourselves, and remains under the power constraint than was designated for the detumbling phase of the overall mission. As discussed in the results section, the magnetorquers with the larger dipole moment that were considered consume more power and detumble NeAtO in more time, proving that the magnetorquers chosen for the mission performed adequately well.

The shorter detumbling time also allows the attitude determination and control portion of the simulation to begin quickly, which allows NeAtO to begin collecting solar power quickly, therefore providing the spacecraft with energy earlier in the mission which makes all the difference when considering a small spacecraft's mission life, especially in an eLEO orbit.

Obviously, with all missions, although very successful, there is room for improvement. Our team spent most of our time troubleshooting the issues we were having with our simulation, which was time well spent since both phases of the mission were completed. However, our team did not have time to simulate both the detumbling and the attitude determination and control of NeAtO together, meaning we did not have enough time to correctly connect both phases of the simulation. Also, noise is present in the detumbling phase of the simulation and correctly filtered, noise is not present in the attitude determination and control phase of the simulation. This was largely due to the fact that once we added noise to the detumbling phase, the simulation slowed down dramatically, and we could not figure out a way to have it run more efficiently with noise. Therefore, in order to have the attitude determination and control phase to run smoothly, we left out noise. It is our hope that future groups could potentially find a way to one, link the two phases together, and two, run the simulation efficiently with noise corrupting the attitude determination and control phase.

6.2 NeAtO Analysis in STK

Systems Tool Kit (STK), as discussed previously, is a very powerful simulation tool that can simulate spacecraft missions both in a high degree of accuracy. Our initial goal was to have a Simulink model with all our initialization values, communicate with STK to simulate the initial environment. With this STK environment initialized, communicate back to Simulink the output values of the STK simulation in a continuous loop, keeping STK running as a visual representation of the mission. While we were able to accomplish the initialization of the STK environment with sensor noise, communicating back with Simulink seemed to be a much bigger task than our group had originally thought. We were unable to overcome this issue as the STK license expired prior to the completion of our project. It is our hope that future groups could use STK's powerful simulation capabilities for CubeSat missions. Utilizing STK's high quality recording software package would deliver a higher quality visualization than one that Simulink can provide.

6.3 Testbed

It is our hope that future project groups will be able to expand on the work conducted in this project, continuing to improve the testbed and use it for real world attitude dynamics simulations. The test platform is designed to be easily modified in order to provide a flexible platform for a variety of projects. There are a number of simple improvements that would increase the reliability and usability of the testbed in its current state. Upgrading or hard anodizing the current air bearing would improve the stability of the system. The SRA100-R45 manufactured by Specialty Components is a intended to be a drop in replacement for the current air bearing. Additionally, placing a proper regulator and air filter on the air supply would improve the reliability of the system. In addition to these simple improvements it is possible to add additional sensors and functionality to the system. However these ideas, listed below, would require extensive research and work to implement.

1. Data Logging
2. Simulated Sun / Star Tracker
3. Simulink / Matlab Integration
4. CAN-BUS Communication
5. Magnetic Field Simulation with Helmholtz Coils
6. Magnetorquer Simulation

References

[1] J. Foley, *The CubeSat Program*. California Polytechnic Institute, San Luis Obispo, 1999.

[2] T. Villela, C. A. Costa, A. M. Brandão, F. T. Bueno, and R. Leonardi, *Towards the Thousandth CubeSat: A Statistical Overview*, vol. 2019. 2019.

[3] S. Jackson, Ed., *NASA's CubeSat Launch Initiative*. NASA, 2019.

[4] "CubeSat Market Report," *Market and Markets.*.

[5] *NASA's InSight Mars Lander*. NASA, 2019.

[6] N. P. Paschalidis, *Mass Spectrometers for Cubesats*. NASA Goddard Space Flight Center, 2017.

[7] H. D. Black, *A passive system for determining the attitude of a satellite*, vol. 2. 1964, pp. 1350–1351.

[8] *The World Magnetic Model*. NOAA.

[9] F. L. Markley and J. L. Crassidis, *Fundamentals of spacecraft attitude determination and control*. Springer, 2014.

[10] X. Fang and Y. Geng, "Simulations of spacecraft attitude control for tracking maneuvers with MATLAB and STK," *2014 IEEE International Conference on Information and Automation (ICIA)*.

[11] "AGI training," *AGI Product Help Center*. AGI, Feb-2020.

[12] "Integrating STK and Simulink for space system design," *STK DIY Series*. AGI.

[13] A. M. Rathbun, J. A. Agolli, and J. Gadoury, *Design and analysis of the sphinx-ng cubesat*. Worcester Polytechnic Institute.

[14] A. M. Koubeck, C. Maki, M. T. Sanchy, and T. Winship, "Design of cubesats for formation flying & for extreme low earth orbit." Worcester Polytechnic Institute.

[15] *DST - formation control testbed*. NASA Jet Propulsion Laboratory, 13AD.

[16] I. R. Bertaska and J. Rakoczy, "An educational platform for small satellite development

with proximity operation capabilities," *AIAA/USU Small Sat Conference 2018*, Aug. 2018.

[17] "ADCS test facility – edu," *Hawaii Space Flight Laboratory.*.

[18] *BNO055 intelligent 9-axis absolute orientation sensor*. Bosch Sensortec, 2016.

Appendix

Simulink User Manual

The first block that we will look at is the "Environment and Magnetometer Model" (E&MM) block, shown in Figure 38

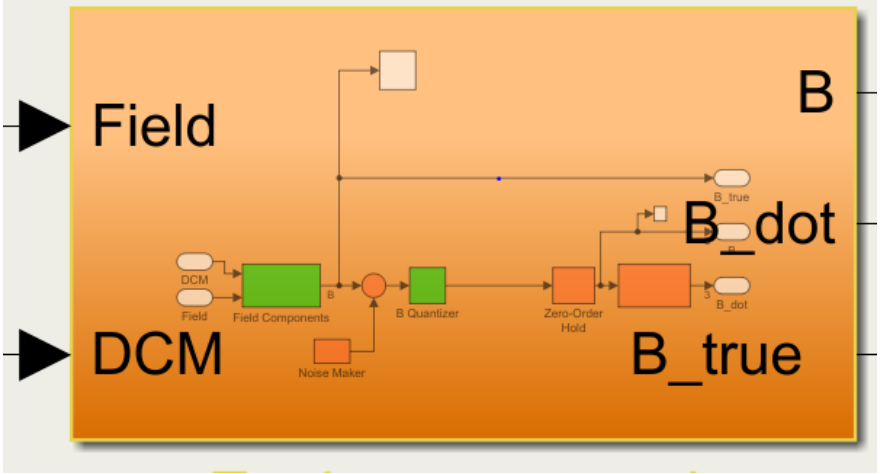

Figure 38: Environment and Magnetometer Block.

As pictured, the E&MM block takes two inputs, the magnetic field in the inertial frame and a direct cosine matrix (DCM) representation of the attitude quaternion of the NeAtO model. Multiplying the magnetic field vector and the DCM, the Environmental and Magnetometer block produce three outputs, B, Bdot and true B, where B is the magnetic field in the body-frame, corrupted by noise and quantized, Bdot is the derivative of the B, and true B is the true body-fixed magnetic field vector, uncorrupted and unquantized. We will discuss where the outputs go in a bit, but first let's discuss where the inputs come from.

In order to find the magnetic field vector in the inertial frame, which is labeled "Field" in the Environment and Magnetometer Block, a very simple and quick orbit propagator was created. In Simulink, the orbit propagator looks like this:

To start the orbit propagation, we start with the built in Simulink tool, the *clock*. This tool outputs the current simulation time in seconds since the simulation started. The *clock* then flows into an *interpreted Matlab function* block. What th *interpreted Matlab function* block does is it allows input values to be passed to a Matlab function for evaluation. However, the function within the *interpreted Matlab function block must return a single value having the*

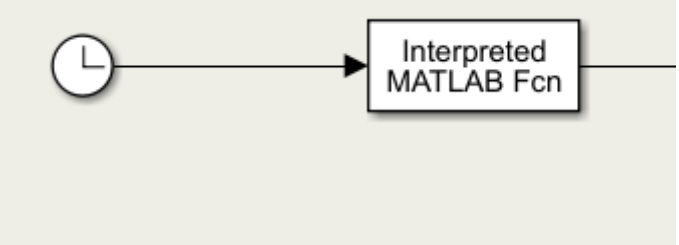

Figure 39: Orbit Propagation in Simulink.

dimensions specified by the output dimensions, which can be modified by double-clicking the block. Our interpreted Matlab function*, which calls the function "OrbitSim", which takes the time input from the clock and has two outputs, the inertial magnetic field vector and the coordinates of NeAtO in LLA. The code for this function is found below.

```
function [mag,lla] = OrbitSim(times)
% This function simulates the orbit of NeAtO.

% Gien Orbital Parameters
start_date = [2019 1 1 0 0 0];  % y, m, d, h, m, s
end_date = [2019 1 2 0 0 0];    % y, m, d, h, m, s
apogee = 600;                   % kilometers
perigee = 250;                  % kilometers
inclination = 51.64;            % degrees
longitude_ascending = 0;        % degrees
argument_periapsis = 0;         % degrees
mean_anomaly = 0;               % degrees

% Derived Orbital Parameters
r_apogee = apogee + 6378.1;
r_perigee = perigee + 6378.1;
semi_major_axis = (r_apogee + r_perigee) ./ 2;                      % ...
    kilometers
eccentricity = (r_apogee - r_perigee) ./ (r_apogee + r_perigee);
standard_gravity = 398600.4418;                                     % km^3/s^2
period = 2 * pi * sqrt(semi_major_axis.^3 ./ standard_gravity);     % seconds
mean_motion = 360 ./ period;

% Calculate the total duration of the mission in seconds
%times = 0:time_step:(decyear(end_date) - decyear(start_date)) * 365 * 24 ...
    * 60 * 60;
```

```
% Expressions for true_anomaly
true_anomaly = @(M, e) M + 2.*e.*sind(M) +  ...
(5/4).*e.^2.*sind(2.*M) +  ...
(1/12).*e.^3.*(13.*sind(3.*M) - 3.*sind(M)) +  ...
(1/96).*e.^4.*(103.*sind(4.*M) - 44.*sind(2.*M));

% Calculate the mean and true anomalies given time
mean_anomalies = mean_anomaly + (mean_motion * (mod(times, period)));
true_anomalies = true_anomaly(mean_anomalies, eccentricity);
clear  mean_anomalies ;

% Calculate the radial position and altitude from anomalies
radial_distance = (semi_major_axis.*(1 - eccentricity.^2)) ./ (1 +
eccentricity.*cosd(true_anomalies));

% Calculate the X, Y position in the plane of the orbit
orbit_x = cosd(true_anomalies) .* radial_distance;
orbit_y = sind(true_anomalies) .* radial_distance;
clear true_anomalies ; clear radial_distance;

% Calculate the X, Y, Z Position in the ECI Frame
w = deg2rad(argument_periapsis);
O = deg2rad(longitude_ascending);
i = deg2rad(inclination);
eci_x = orbit_x .* (cos(w).*cos(O) - sin(w).*cos(i).*sin(O)) -  ...
orbit_y .* (sin(w).*cos(O) + cos(w).*cos(i).*sin(O));
eci_y = orbit_x .* (cos(w).*sin(O) + sin(w).*cos(i).*cos(O)) +  ...
orbit_y .* (cos(w).*cos(i).*cos(O) - sin(w).*sin(O));
eci_z = orbit_x .* (sin(w).*sin(i)) + orbit_y .* (cos(w).*sin(i));
eci = [eci_x; eci_y; eci_z];
clear  orbit_x  orbit_y ;
clear  eci_x  eci_y  eci_z ;

% Get the DCM Transformation from ECI to ECEF
% Get date vectors for all times to evaluate
dates = datevec(datenum (start_date) + times ./ 86400);
dcm = dcmeci2ecef('IAU-2000/2006', dates);
ecef = eci;
ecef(:, 1) = dcm(:,:,1) * eci(:, 1);
clear dcm eci;
```

```
% Compute the Latitude, Longitude, and Altitude of the satellite
lla = ecef2lla(ecef' .* 1000);
clear ecef;

% Compute the magnetic field at the satellite's position for each timestep
mag(1,:) = wrldmagm(lla(1, 3), lla(1, 1), lla(1, 2), decyear(dates(1, :)))';
end
```

OrbitSim is a very simple orbit simulator, as its name suggests. *OribtSim* then outputs the magnetic field at the satellite's position for each time step, which updates the input value that is seen flowing into the E&MM block. The second input seen flowing into the E&MM block is the DCM that stems from the true attitude quaternion of NeAtO. THe flow of the attitude quaternion to the DCM input can be seen in Figure 40.

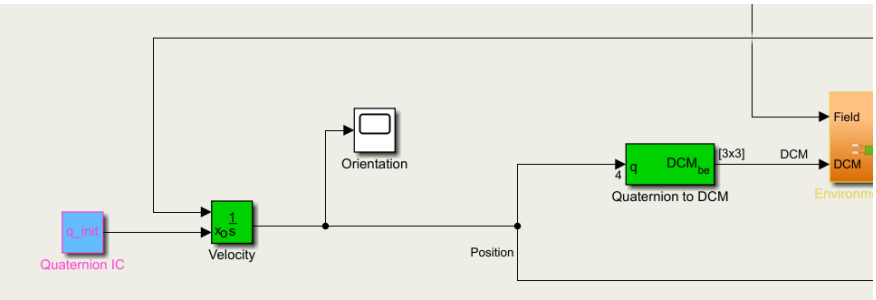

Figure 40: Initial Attitude Quaternion to DCM Input in Environment and Magnetometer Model Block.

To start, our model takes the initial attitude quaternion of NeAtO, defined by a constant block in Simulink. In order to retrieve the initial attitude quaternion of NeAtO, our simulation pulls all initial values in our simulation from an initialization file that is defined in the "Model Explorer" in Simulink. In order to get to the initialization file, you go to the "Modeling" tab, click the drop down arrow directly under the "Modeling" tab, and click on "Model Workspace" in the "Data Repositories" sub group. The initialization file runs once the Simulink simulation opens. If the initialization file is changed when the Simulink file is open, the Simulink file needs to be re-opened. All our initial conditions are defined within this initialization file.

From the initial attitude quaternion, our simulation then flows into an integrator block in Simulink. The *integrator blocks* does continuous-time integrations of the input signal, which is the change in the attitude quaternion, and the initial condition, which in this case is the initial attitude quaternion. What the integrator block does in this case is take the initial

attitude quaternion, which are considered the "+ C" of any integration problem, and then outputs the attitude quaternion at the next time step. From there, the attitude quaternion flows through a quaternion to DCM, which is a built-in Simulink tool used to determine the 3x3 DCM from a 4x1 quaternion orientation vector. This *quaternion to DCM* block also transforms vectors from geodetic earth to body axes. From there, this input combines with the inertial magnetic field as the two inputs in the E&MM block.

We now can go back to Figure 38 and begin describing how the E&MM block outputs B, B-dot and true B. This block is a subsystem. In order to create the subsystem, you simply drag over and highlight the blocks you wish to be in a *subsystem*, and Simulink will ask if you would like to create a subsystem. The contents of the E&MM block is shown in Figure 41.

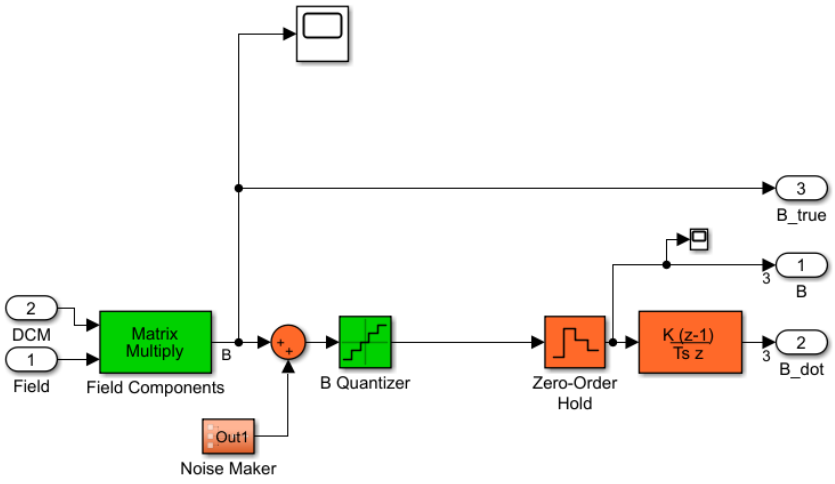

Figure 41: *Subsystem* "Environment and Magnetometer Model".

To start, the inertial magnetic field matrix and DCM flow through a built-in Simulink matrix *multiply* block hat multiplies the two matrices together. After these matrices are multiplied, we are provided with our first output, the true B, which flows up and out. True B, although taken as one output, also continues flowing through a *sum* block that combines the true B with a noise maker that is designed to add noise to the E&MM block. What this *sum* lock does is it adds, or subtracts, vectors stemming from each input port. The *sum* block can also add scalars. As mentioned before, the *sum* block is summing the true magnetic field vector in the body frame with the simulated sensor noise vector in the body frame. The noise is another subsystem block that contains the model in Figure 42.

On the far left of the the noise block are three *random number generator* blocks within Simulink that output a normally distributed random signal. This noise is specifically the

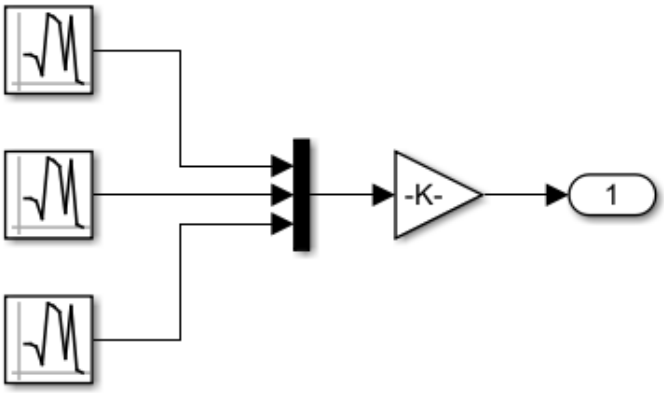

Figure 42: Noise Block.

magnetometer noise. The three random number generator blocks then flow into a *mux* block which combines any number of inputs into one vector signal. The output of this specific *mux* block is the magnetometer noise vector. From there, the noise flows into a *gain* block, which in this simulation is used as a switch to signal either the noise to be turned on or off, or multiplied by one or zero. This gain value is set in the initialization file, and has to be set as zero or one. From there, we have the simulated magnetometer noise output.

Moving back to Figure 41, we are moving on to the block labeled "B Quantizer", which follows when the magnetic field components and noise are summed together. This is a built-in Simulink block called *quantizer* which discretizes input at a given interval. What this *quantizer* does in this simulation is it takes the analog function output from the sum block and turns it into a signal with discrete quanta, which in this case is the resolution of the magnetometer. From there, the quantized magnetic field components combined with the noise flow into a *zero-order hold* block in Simulink that then converts the continuous-time signal into a discrete-time signal, to simulate the sample rate of the magnetometer. After the zero-order hold is placed on the quantized components, we get our second output, B, which is our simulated magnetometer output, corrupted by noise, limited in resolution, and limited in sample rate. Finally, in order to get our third output the now continuous-time components flow into a *discrete derivative* block we a representation of what the Bdot would be calculated as if the magnetometer output was not first filtered. This gives us our third and final output for our E&MM block, B-dot, which is only present to demonstrate the importance of filtering our signal before differentiating it.

We will now discuss where the three outputs of the E&MM block go, and also move on to our second major block labeled, 'Detumble Control'. After the E&MM block is the sequence in Figure 43.

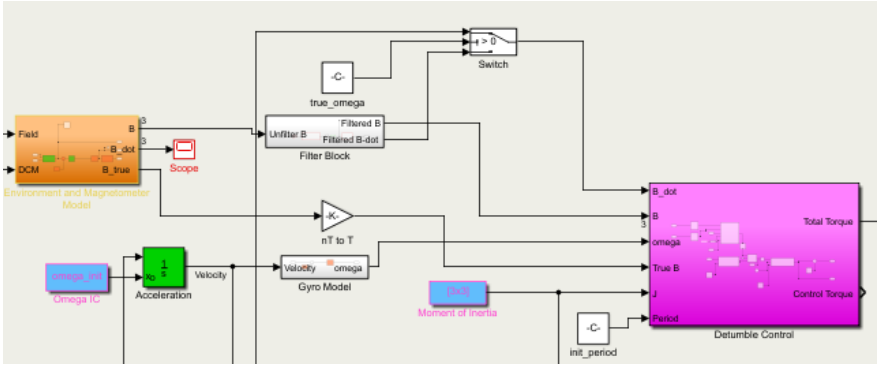

Figure 43: Inputs to Detumble Control Block.

We'll start with the six inputs that flow into the 'Detumble Control' block, B-dot, B, omega, true B, J and the period. The first and second inputs 'Detumble Control' takes are B-dot and B. Now, if you look at the schematic, you may ask, "Why is B-dot and B coming from the grey filter block and not just straight from the B-dot output stemming from the E&MM block?" Great question.The inputs that go into the 'Detumble Control' block must be filtered, which is why B-dot and B come from the filter block instead of directly from E&MM. As mentioned before, B-dot from the E&MM block goes into a scope block in Simulink, which is a time scope block that displays time-domain signals, which we are using to show that without filtering, the result of differentiating B is just noise. The output from the B-dot scope block after running the simulation for 150 simulation seconds is shown in Figure 44.

Once we knew the outputs were correct, we moved on to filtering the unfiltered B output in order to calculate our first two inputs into the 'Detumble Control' block, B and B_dot. The contents of the filter block is shown in Figure 45.

The first step the unfiltered B takes is getting passed through an element-wise gain that multiplies it by 0.001 to convert it from nT to μT. After the unfiltered B is converted, it is then passed through a built-in *lowpass filter* where Simulink filters out the high-frequency noise, therefore outputting a filtered version of B. It is worthwhile to note that Simulink has several filtering methods already built in to the software, making it easier for our group since we did not have to code most of the filters ourselves, groups can also create their own filters by including *integrated Matlab functions* in their simulations instead. After B is filtered, it then runs through a *transpose* block, where it is transposed and continues through an *IC*

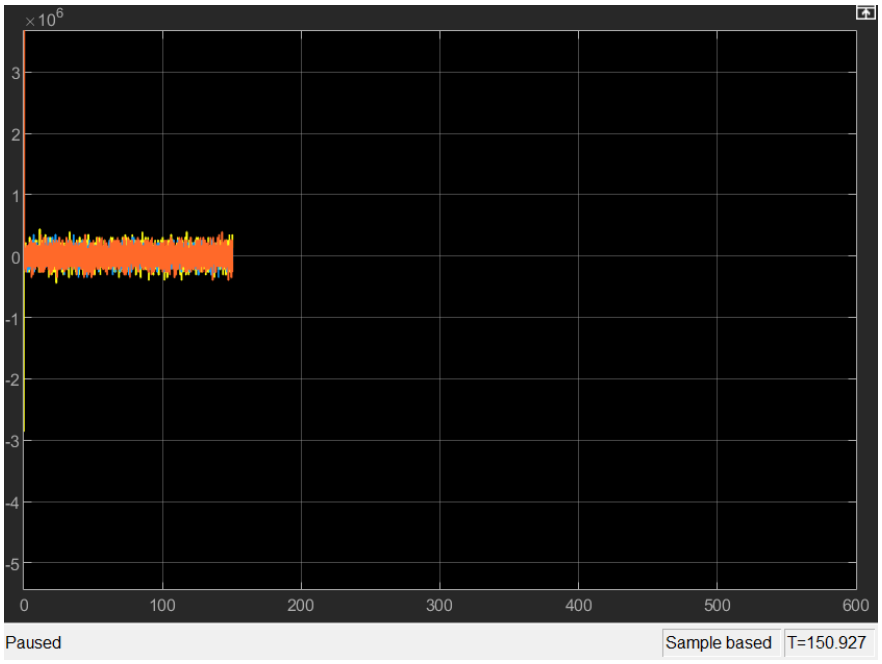

Figure 44: B-Dot Scope.

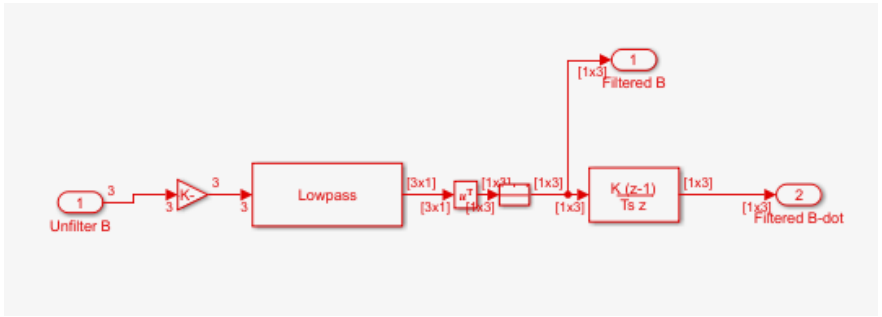

Figure 45: Filter Block.

which sets the initial conditions of the signal for the simulation to prevent errors on start up. What this block does in our simulation is ensure the output from the *transpose* block is a 1x3 matrix. Once filtered, B takes two routes, one route is straight out of the filter block and into the 'Detumble Control' Block and the other route is into a *discrete derivative block* which takes B and differentiates it by the method of finite differences, therefore creating the output B-dot.

Now from here, B-dot does not flow straight into the 'Detumble Control' block, it instead flows into a *switch*, whose sequence can be seen in Figure 46.

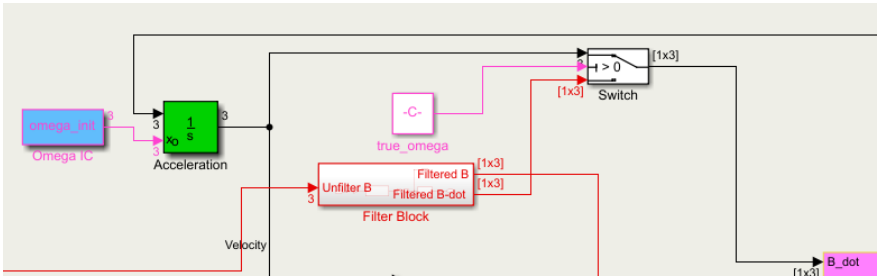

Figure 46: Switch Inputes to Receive B-Dot.

The first input the switch takes is the angular velocity, ω. In order to get ω, our simulation runs through the same process it went through in order to receive the attitude quaternion for the DCM back in Figure 40. It pulls the initial omega from the initialization file and runs them through an integrator block that calculates them at the next time step, which then runs through the switch. The second input into the switch is a flag that determines if the model will use the true omega, which is also defined in the initialization file. Finally, the third input is the filtered B-dot we received from the filter block discussed previously. Once all three inputs are found and run through the switch, we pass either our filtered B-dot or the true angular velocity. The switch determines which signal to pass depending on how the flag is set.

Moving on to our third input for 'Detumble Control', which is ω, we will actually go back to the first part of Figure 46 which shows our initial ω readings going into the integrator block. The process to get the ω input is shown in Figure 47.

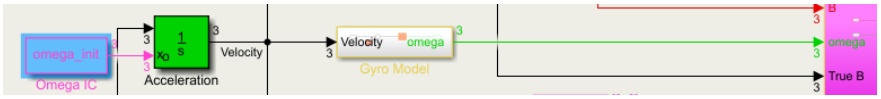

Figure 47: Detumble Control Omega Input.

Our omega input into the 'Detumble Control' model is slightly different than the omega input into the *switch* to receive B-dot in that it flows into a gyroscope model which will corrupt omega with noise instead of acting as a check for our filtered B-dot. The gyro model system is shown in Figure 48.

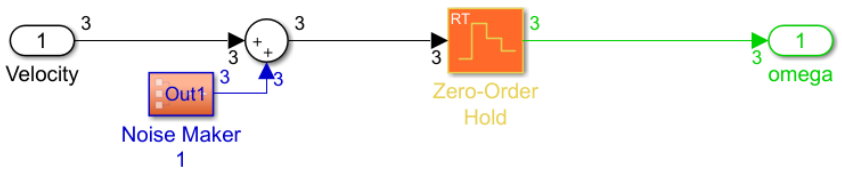

Figure 48: Omega Gryo Model.

The continuous-time omega goes through the same first steps as our magnetic field components from Figure 41, and are affected by the same sum block and noise block, shown in Figure 42. Once noise is added to the omega through the sum block, it runs through a zero-order hold block which holds the input for a specific amount of time, which in this case is 1/60th of a second. This simulates the finite response rate of the gyroscopes. Once this is done, the output, corrupted omega, then flows into the 'Detumble Control' block as our third input.

The fourth input into the 'Detumble Control' model is our true B which flows straight from the E&MM block, through a gain block that converts it from nT to T, and into the 'Detumble Control' block. The process is shown in Figure 49.

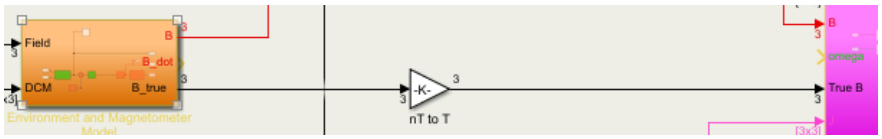

Figure 49: True B Input.

The fifth and sixth inputs into the 'Detumble Control' model are our moments of inertia and our orbital period. We received our moments of inertia from our structures sub-group that is part of our overall MQP team and set our initial period in the model explorer. Those inputs can be seen in Figure 50.

Now that we have all of our inputs, we can now simulated the 'Detumble Control' shown in Figure 51.

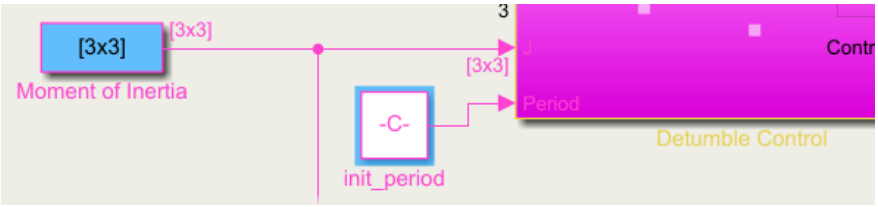

Figure 50: Moments of Inertia and Initial Period Input.

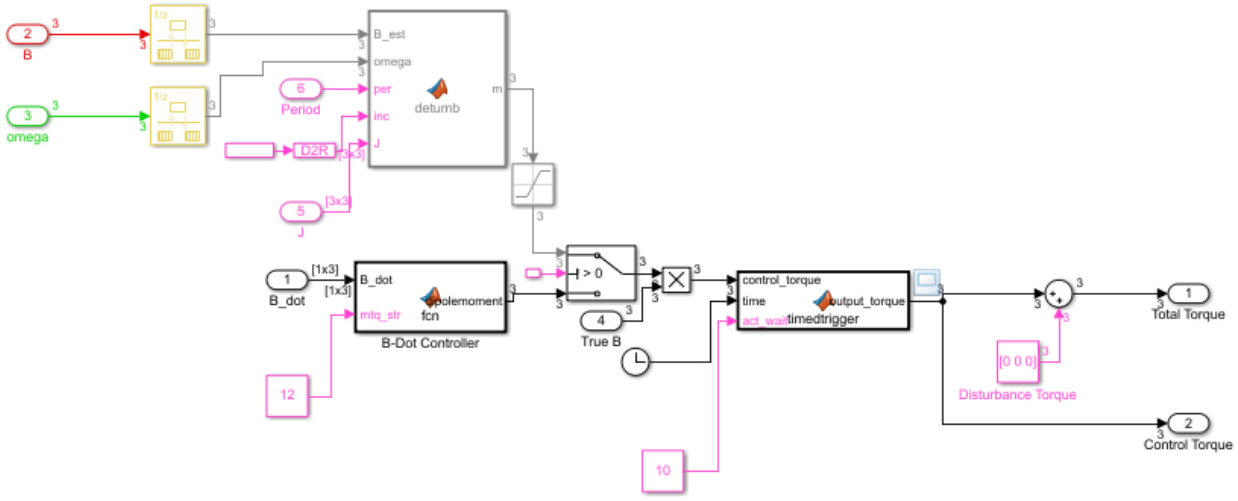

Figure 51: Overall Detumble Control Model

To describe how this Detumble Control system works, we'll split it into two sections, "Before *Switch*" and "After *Switch*". You can see the *switch* directly above "True B" in the model. Obviously, first, we will start with "Before Switch" which can be seen in Figure 52.

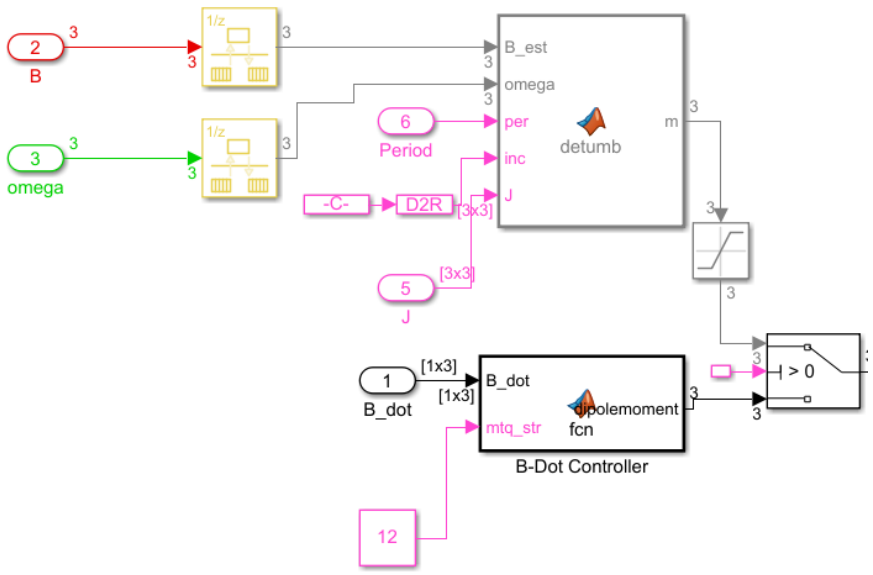

Figure 52: Before *Switch* Detumble Control.

To start, we have our second and third inputs into 'Detumble Control', filtered B and corrupted Omega, which both flow into their own individual rate transition built-in Simulink blocks. What these rate transition blocks do is they handle the data transfer between different rates and tasks with certain parameters, initial conditions and output sample times specified within the block. These are our first two inputs into our "Detumble" Matlab function. The next input, our period, comes straight from the input we specified outside of this system block. The fourth input, our inclination angle, is specified in the initialization file within Simulink and converted from degrees to radians with a converter block in Simulink. Finally, our sixth input, similar to the period, is our moments of inertia and comes straight from the input into the 'Detumble Control' system. We now have all of our inputs for our detumble Matlab function, whose block with inputs and outputs is shown in Figure 53.

The five inputs into the detumble function combine together in order to solve for m, the magnetic dipole moment. Code to accomplish this is shown below.

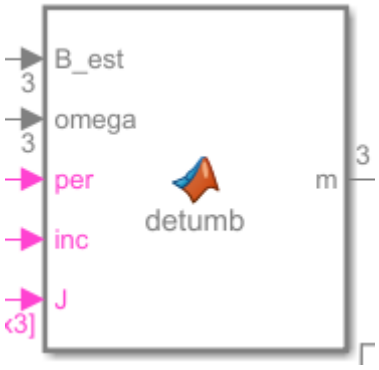

Figure 53: Detumble Function Inputs and Outputs.

```
function m = detumb(B_est,omega,per,inc,J)
%This function solves for the magnetic dipole moment
b = B_est/norm(B_est);
Jmin = min(diag(J));
%This is finds the gain from Eq 7.55 in the Crassidis textbook
k = ((4*pi())/per)*(1+sin(inc))*Jmin;
%Solve for the magnetic dipole moment
m = (k/norm(B_est))*cross(omega,b);
```

You will notice that the block type our group used is different than the *integrated Matlab function* blocks we had been using below to connect Matlab and Simulink in that we typed our function directly into Simulink. In some cases, Simulink responds better to this method, and this was one of those cases. Once m is found, it is then run through a *saturation* block, which basically only outputs m values if it is between a range of two values, which in this case is -1.2 and 1.2. This gives us our first input for the switch. The second input into the switch, or our "control" value, was determined by the detumble control we chose, which in our simulation is "B-Dot Control." The value we wish the threshold to be greater than in this case is set in our initialization file. Finally, the third input is our dipole moment which has the process in Figure 54.

In order to find the dipole moment, we put the B-dot input and our chosen magnetorquer strength of the magnetorquers that were going to be used for detumbling NeAtO, set by a constant block that is not defined in our initialization file, through our B-dot Controller Matlab function, whose code can be seen below.

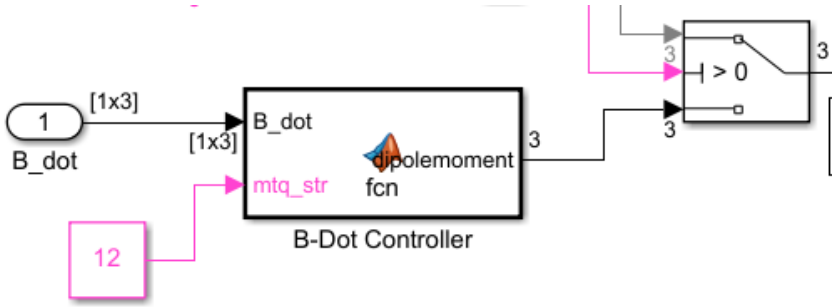

Figure 54: Dipole Moment Switch Input.

```
function dipolemoment = fcn(B_dot,mtq_str)
%This function finds the dipole moment
%Calculate the dipole moment
max_moment = mtq_str; %same in all directions because same MTQs
u = eye(3);
moments = max_moment*u;
%Initialize the dipole moment
dipolemoment = [0 0 0]';
%This loop finds the moment for each magnetorquer
for i = 1:3
    if abs(dot(u(:, i), B_dot)) >= .01
        dipolemoment = dipolemoment  - moments(:,i) * sign(dot(u(:, i), ...
            B_dot));
    end
end
```

Calculating this dipole moment gives us our final input into the switch. What this switch is doing is it is measuring and passing through input one, m, when input two satisfies a certain criterion, which in this case is the constant that determines whether bang-bang control is turned off. If this criterion is not met, it passes through input three, the dipole moment, which is the output of the switch. We can now move onto the "After Switch" part of our 'Detumble Control' block, whose process can be seen in Figure 55.

Once we have the output of the *switch*, the first thing the simulation does is take the cross product of that output with the true B, which is the initial input in the 'Detumble Control' model. This cross product gives us the output torque of the magnetorquers, and is our first input in our next Matlab function which is a timed trigger that measures the output torque of the magnetorquers. The second input is a *clock* that measures the time with a decimation of 10. The third input is the actuator wait time, set to a value of ten seconds, which is an

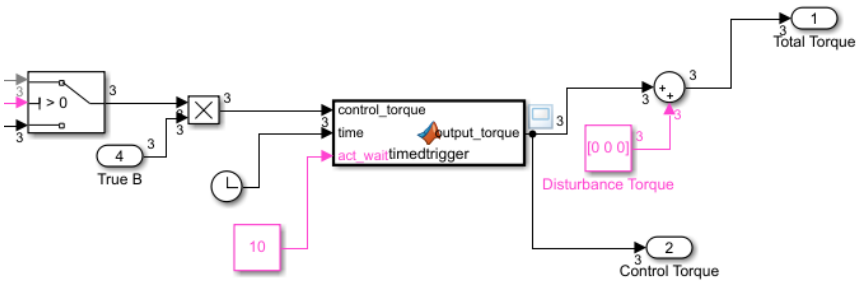

Figure 55: After Switch Detumble Control.

arbitrary value. The code for the timed trigger can be seen below.

```
function output_torque = timedtrigger(control_torque,time,act_wait)
%Suppresses control torque until "act_wait" seconds have passed

if time >= act_wait
    output_torque = control_torque;
else
    output_torque = control_torque*0;
end
```

From this timed trigger function, the actuator wait time, clock and cross product of the true B and switch output come together to create the output torque of the magnetorquers. This output torque then flows two ways: one directly out of 'Detumble Control' as our control torques, which we measure to make sure the 'Detumble Control' model is functioning correctly, and two into a *sum* block where it is combined with the disturbance torques to create our second output, the total torque. Our control torques can be seen in Figure 56, Figure 57, and Figure 58.

We can now move on to our third and final major block within our entire detumble simulation, the attitude dynamics block, shown in Figure 59.

The 'Attitude Dynamics model', which is an in-Simulink Matlab function, rather than a system like the others, takes four inputs, the total torque, the attitude quaternion of NeAtO, the omegas of NeAtO and the moments of inertia of NeAtO. All four of these inputs we have already discussed, and nothing is modified before they reach the 'Attitude Dynamics model' block. The total torque comes from the direct output value from the Detumble Control model, please refer back to Figure 43 for that process, please refer to Figure 40 on how the

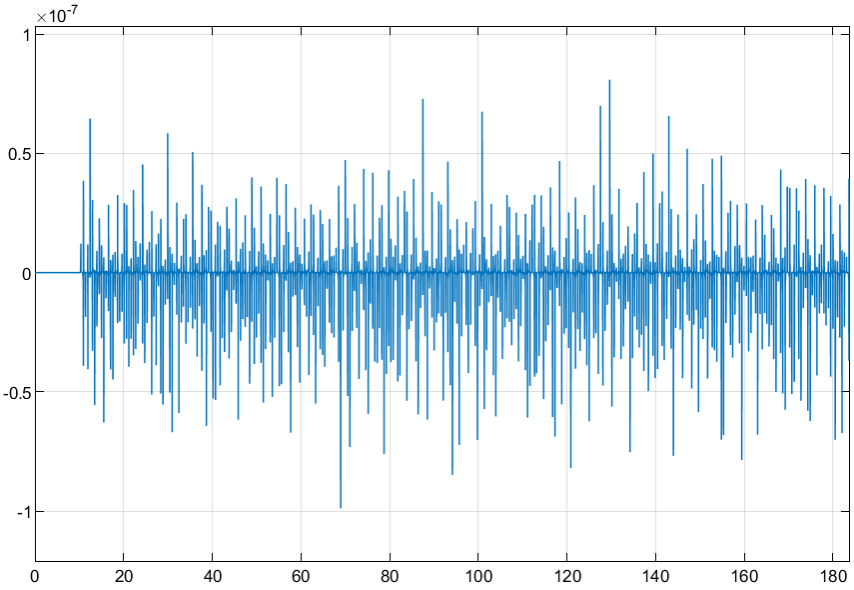

Figure 56: Control Torque 1 of Magnetorquer.

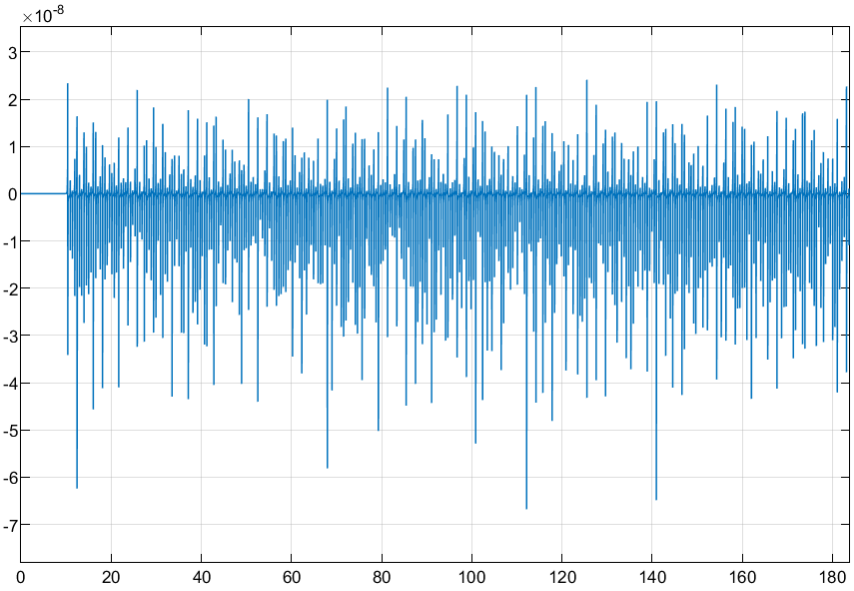

Figure 57: Control Torque 2 of Magnetorquer.

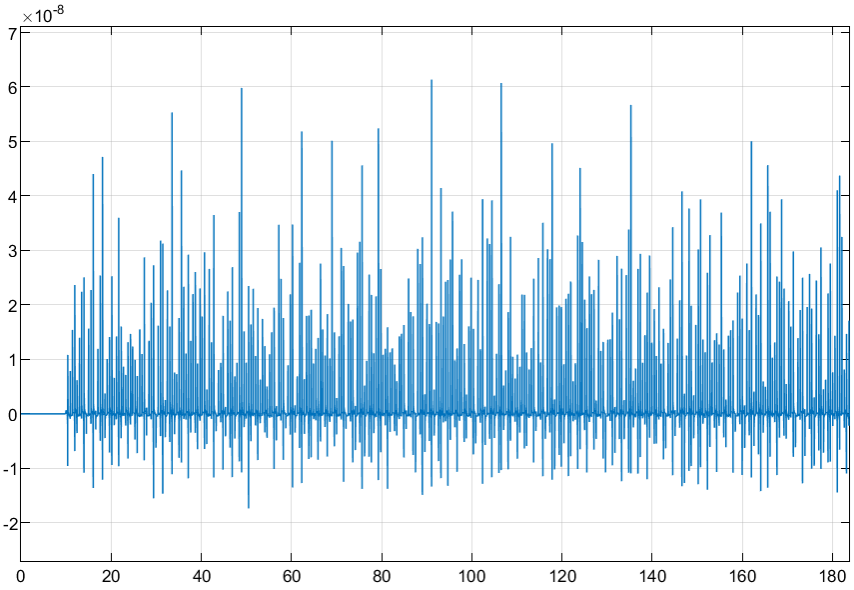

Figure 58: Control Torque 3 of Magnetorquer.

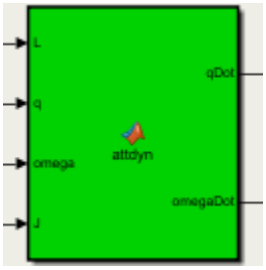

Figure 59: Attitude Dynamics Model Block.

attitude quaternion is found, please refer to Figure 46 for how the omegas are found and please refer to Figure 49 for the moments of inertia. Once all these inputs are found, the 'Attitude Dynamics model' runs the following code.

```
function [qDot,omegaDot] = attdyn(L,q,omega, J)
%This is the plant

%Define the total torque, attitude quaternion and angular velocity of NeAtO
L = [L(1);L(2);L(3)];
q = [q(1);q(2);q(3);q(4)];
omega = [omega(1);omega(2);omega(3)];
Xiq = [ q(4) -q(3)  q(2);
        q(3)  q(4) -q(1);
    -q(2)  q(1)  q(4);
    -q(1) -q(2) -q(3)];
%Quaternion dynamics
qDot = (1/2)*Xiq*omega;
omegaDot = (J)\(L - cross(omega,(J*omega)));
```

After the attitude dynamics script is run within the simulation, it outputs omegaDot and qDot which are the derivatives of the attitude quaternion and omega inputs. Once these are calculated, they then flow back around to the beginning of the simulation where they act as the new inputs into the integrator blocks that are used to find the attitude quaternion and angular velocity values used in the simulation, please refer back to Figure 40 and Figure 46, respectively. The entire simulation is shown in Figure 60.

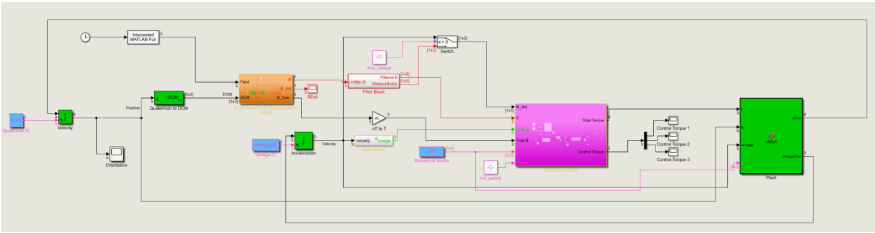

Figure 60: Final Detumble Simulation.

To start the attitude determination and control simulation, our group first had to decide on an attitude estimation method. In our case, we chose quaternion estimation, commonly referred to as QUEST as discussed in the report. The QUEST block can be seen in Figure 61.

Our QUEST Matlab block takes four inputs, the body fixed sun vector, the inertial sun

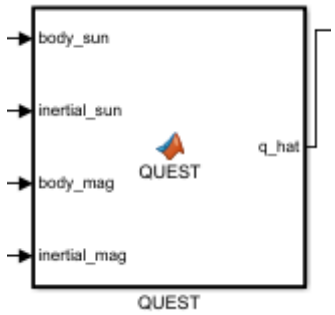

Figure 61: QUEST Matlab Block.

vector, the body fixed magnetic field vector and the inertial magnetic field vector. The flow can be seen in Figure 62.

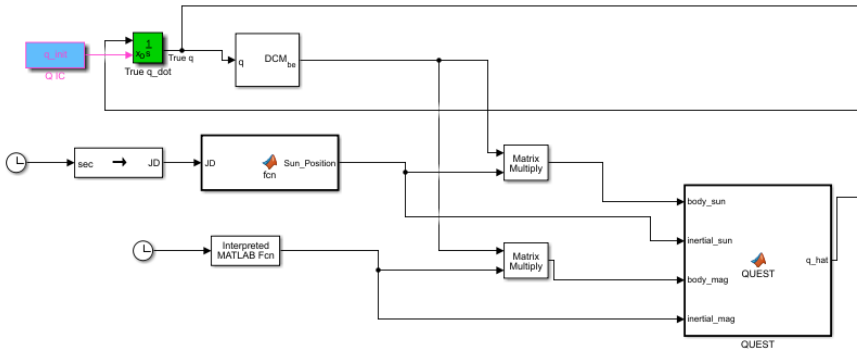

Figure 62: Input Flow to QUEST.

The first input we'll address is the inertial sun vector, which is found by using the a custom Matlab function that takes a given Julian Date (JD) and outputs the sun position at that time. To get the simulation time in the correct JD form, a clock block is fed into an initialized seconds to Julian Date block. This output is directly wired into the QUEST plant. The code for this function can be seen below.

```
function Sun_Position = fcn(JD)
%This function calculates the inertial sun vector from a given Julian Date.
%Calculate the number of days since Greenwich Noon, Terrestrial Time, on ...
    1st January 2000.
n = JD - 2451545.0;

L = 280.46 + 0.985674 *n; %Mean Longitude of the Sun
g = 357.528 + 0.9856003 *n; %Mean Anomaly of the SUn
L = mod(L,360);
```

```
g = mod(g,360);
e = 23.439 + 0.0000004 *n; % Obliquity of the ecliptic

R = 1.00014 - (0.01671*cosd(g)) - (0.00014*cosd(2*g)); %Distance from Sun ...
    to Earth in AU
lambda = L + (1.915*sind(g)) + (0.020*sind(2*g)); %Ecliptic Longitude of ...
    the Sun
%Calculate the Inertial Sun Vector Components
x = R*cosd(lambda); %AU
y= R*cosd(e)*sind(lambda); %AU
z = R*sind(e); %AU
%Compute the Sun Position
Sun_Position_1 = [x,y,z];
Sun_Position_1 = (Sun_Position_1/norm(Sun_Position_1));
Sun_Position = transpose(Sun_Position_1);
```

Next, to get the body fixed sun vector, the integrated initial attitude quaternion, described in Figure 40, is passed through a Quaternion to Direct Cosine Matrix block. Next, a matrix multiplication block is used to multiply the output of the Sun Position function with the DCM to rotate the vector from the inertial frame to the body frame. This emulates the sun sensor and provides body fixed readings for sun position. This value is then passed into QUEST. To get the inertial magnetic field vector, an interpreted Matlab function is used to approximate the magnetic field at a certain time with our given orbital parameters. The interpreted Matlab function, OrbitSim, is the same interpreted Matlab function used as the orbit simulator described in the beginning of the detumble phase of our simulation. This was then inputted into the QUEST algorithm. To achieve the body fixed magnetic field vector, the inertial magnetic field vector flows through a matrix multiply block along with the integrated attitude quaternion to get the body fixed magnetic field. This is the final input to QUEST. With all the inputs now found, the QUEST algorithm can be run to find the estimated attitude quaternion. The code inside the QUEST block can be seen below.

```
function q_hat  = QUEST(body_sun, inertial_sun, body_mag, inertial_mag)
% coder.extrinsic('adjoint');

%Define Sensor Weights
a_i = [1 1];

body_sun = body_sun/norm(body_sun);
```

```
body_mag = body_mag/norm(body_mag);
inertial_sun = inertial_sun/norm(inertial_sun);
inertial_mag = inertial_mag/norm(inertial_mag);

l_0 = sum(a_i);

B = a_i(1)*body_sun*inertial_sun' + a_i(2)*body_mag*inertial_mag';
trB = trace(B);
Z = a_i(1)*cross(body_sun,inertial_sun) + a_i(2)*cross(body_mag,inertial_mag);

K = [B+B'-trB*eye(3) Z;
    Z' trB];

S = B+B';
adjS = det(S)*inv(S);
kappa = trace(adjS);

for i = 1:3
phi_QUEST = ...
    (l_0^2-trB^2+kappa)*(l_0^2-trB^2-norm(Z)^2)-(l_0-trB)*(Z'*S*Z+det(S))-Z'*S^2*Z;
phi_QUESTdot = 2*l_0*(2*l_0^2-2*trB^2+kappa-norm(Z)^2)-(Z'*S*Z+det(S));

l_0 = l_0 - phi_QUEST/phi_QUESTdot;
end
lambda_max = l_0;

rho = lambda_max + trace(B);
q_vec = (det(rho*eye(3) - S)*inv(rho*eye(3) - S))*Z;
q_h = [q_vec;
         det(rho*eye(3)-S)];
q_hat = [q_h(4); q_h(1:3)];
q_hat = q_hat/norm(q_hat);
```

Looking at the script above, you'll quickly note that there is a lot commented out. While the script runs QUEST successfully, it doesn't run for iterations above 180 degrees, which is described in more detail in the main report. In order to solve this issue, our group tried implemented ESOQ into our QUEST method and was close, but didn't get it to work fully. In the end, for the purpose of our simulation, the QUEST script still gives us the correct outputs of qHat.

Once the QUEST script has finished, our simulation then begins calculating the control

torques of NeAtO, whose block can be seen in Figure 63.

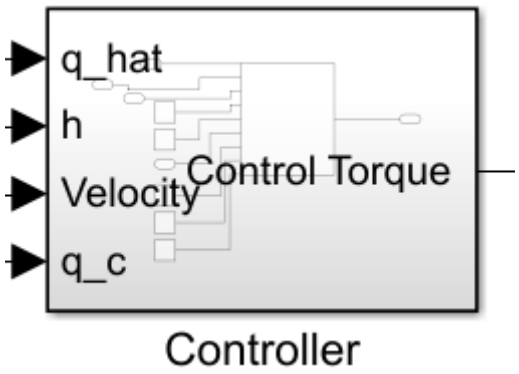

Figure 63: Controller to Calculate Control Torques.

The controller has four inputs, qHat, the angular momentum, h, the angular velocity, ω, and the initial attitude quaternion, q_c. The first input, qHat, comes straight out of the QUEST function block. The angular momentum is found the same way the attitude quaternion and omega has been found throughout the manual, through initializing it and putting it through an integrator block that takes it at the next time-step. The angular velocity, ω, is found similarly to Figure 9, and the initial attitude quaternion is input as a constant. This gives us the four inputs into the controller. Within the controller is the system shown in Figure 64.

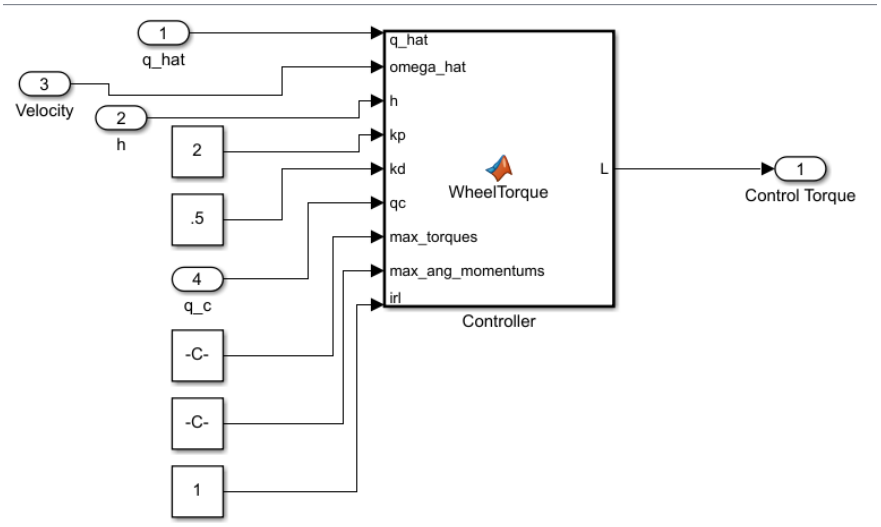

Figure 64: Controller Subsystem Model.

The Controller Matlab function, Wheel Torque, calculates the control torque of the momentum wheels used on NeAtO, and whose output graphs can be seen in the results section of our

report. The Controller function takes nine inputs, qHat, angular velocity, angular momentum, initial attitude quaternion, which were the four calculated inputs from before, along with k_p, which is the proportional gain, kd, which is the derivative gain, qc, which is the error attitude quaternion, irl, which stands for "inertial reaction limits" and is a value set to one or zero that either simulates real reaction wheels or simulates reaction wheels without any limitations, max torques and the max angular momentums for NeAtO's chosen momentum wheels, which are all constant blocks flowing through the controller. With all these inputs, the simulation then can calculate the control torque of the momentum wheels, whose code can be seen below.

```
function L  = ...
    WheelTorque(q_hat,omega_hat,h,kp,kd,qc,max_torques,max_ang_momentums,irl)
%This function does this

%Initialize starting values
Xiqc = [ qc(4) -qc(3)  qc(2);
         qc(3)  qc(4) -qc(1);
     -qc(2)  qc(1)  qc(4);
     -qc(1) -qc(2) -qc(3)];
dq = [0 0 0 0]';
dq(1:3) = Xiqc'*q_hat;
dq(4) = q_hat'*qc;
%Define Control Torque Equation
L = -kp*sign(dq(4))*dq(1:3)-kd*(1-dq(1:3)'*dq(1:3))*omega_hat;

if irl == 1
hDot = - L - cross(omega_hat,h);
for i = 1:3
    if hDot(i) > max_torques(i)
        hDot(i) = max_torques(i);
        temp = cross(omega_hat,h);
        L(i) = -hDot(i) - temp(i);
    end
    if h(i)>= max_ang_momentums(i) && hDot(i)>0
        hDot(i) = 0;
        temp = cross(omega_hat,h);
        L(i) = 0;
    elseif h(i)<= -max_ang_momentums(i) && hDot(i)<0
        hDot(i) = 0;
```

```
        temp = cross(omega_hat,h);
        L(i) = 0;
    end
end
end
```

We now have the control torque of NeAtO, which is our first input out of five into our Attitude Dynamics block within the Attitude Determination and Control portion of our simulation. The other four inputs are the attitude quaternion, angular velocity, moments of inertia and angular momentum. The attitude dynamics block can be seen in Figure 65, along with the code inside the block.

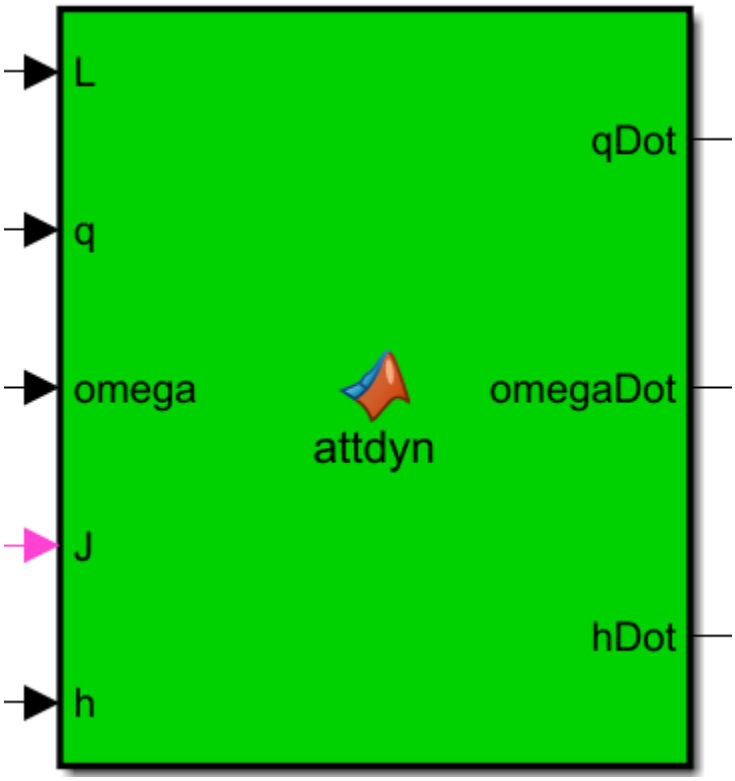

Figure 65: Attitude Dynamics Model within Attitude Determination and Control.

```
function [qDot,omegaDot,hDot] = attdyn(L,q,omega, J, h)

L = [L(1);L(2);L(3)];
q = [q(1);q(2);q(3);q(4)];
omega = [omega(1);omega(2);omega(3)];
```

```
Xiq = [ q(4) -q(3)  q(2);
        q(3)  q(4) -q(1);
    -q(2)  q(1)  q(4);
    -q(1) -q(2) -q(3)];

qDot = (1/2)*Xiq*omega;
hDot = - L - cross(omega,(h));
omegaDot = (J)\(-hDot - cross(omega,(J*omega+h)));
```

This Matlab script is incredibly similar to the attitude dynamics script used in our detumbling portion of the simulation, just with the addition of the angular momentum. After the simulation runs the attitude dynamics script, we are left with the three outputs shown in Figure 65, qDot, ω Dot and hDot. All three of these output values run back through their respective integrator blocks to update the simulation and further update the state of NeAtO. The entire simulation can be seen in Figure 66.

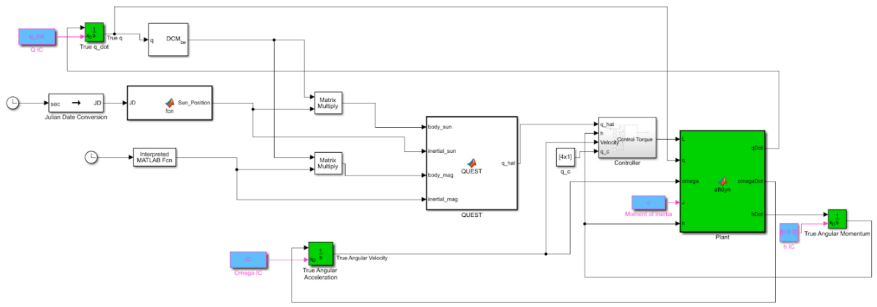

Figure 66: Attitude Determination and Control Simulation.

Detailed Component Lists

Table 9: Logical Component List

Component	Manufacturer	Quantity
LTC2945	Analog Devices	1
MCP2515	Microchip	1
MCP2562	Microchip	1
PCA9685	NXP Semiconductors	1
BNO055	Bosch	1
L78S05CV	STMicroelectronics	1
UA78M	Texas Instruments	1

Table 10: Passive Component List

Component	Manufacturer	Quantity
20mΩ Resistor	Ohmite	1
10KΩ Resistor	Vishay	7
2KΩ Resistor	Vishay	2
120Ω Resistor	Vishay	2
0.33μF Capacitor	Wurth Elektronik	1
0.22μF Capacitor	Wurth Elektronik	1
0.1μF Capacitor	Wurth Elektronik	7
15pF Capacitor	Wurth Elektronik	2
30pF Capacitor	Wurth Elektronik	2
32.786kHz Crystal	ECS	1
16MHz Crystal	ECS	1

Table 11: Connector Component List

Component	Manufacturer	Quantity
IDC Connector	TE Connectivity	2
IDC Header	TE Connectivity	1

Component	Manufacturer	Quantity
PH 24AWG Contact	JST	8
PHR-2 Connector	JST	1
PHR-2 Header	JST	1
PHR-3 Connector	JST	3
PHR-3 Header	JST	3
Mega-Fit 12AWG Contact	Molex	8
Mega-Fit 2Ckt Connector	Molex	1
Mega-Fit 2Ckt Header	Molex	1
Mega-Fit Right Angle Connector	Molex	3
Mega-Fit Right Angle Header	Molex	3

Test Bed Schematics

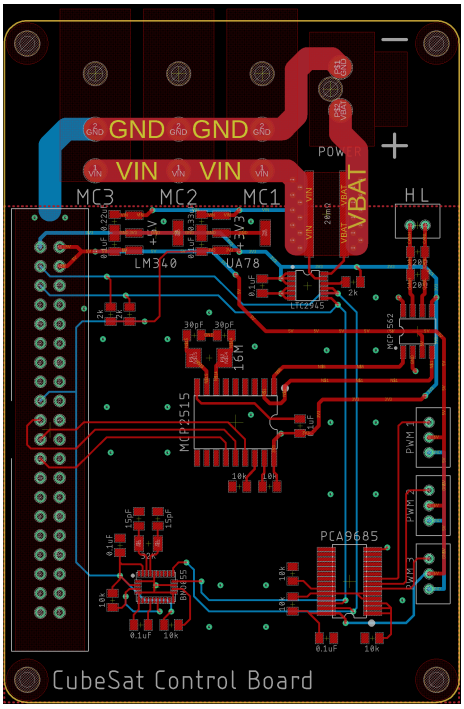

Figure 67: Control Circuit Board.

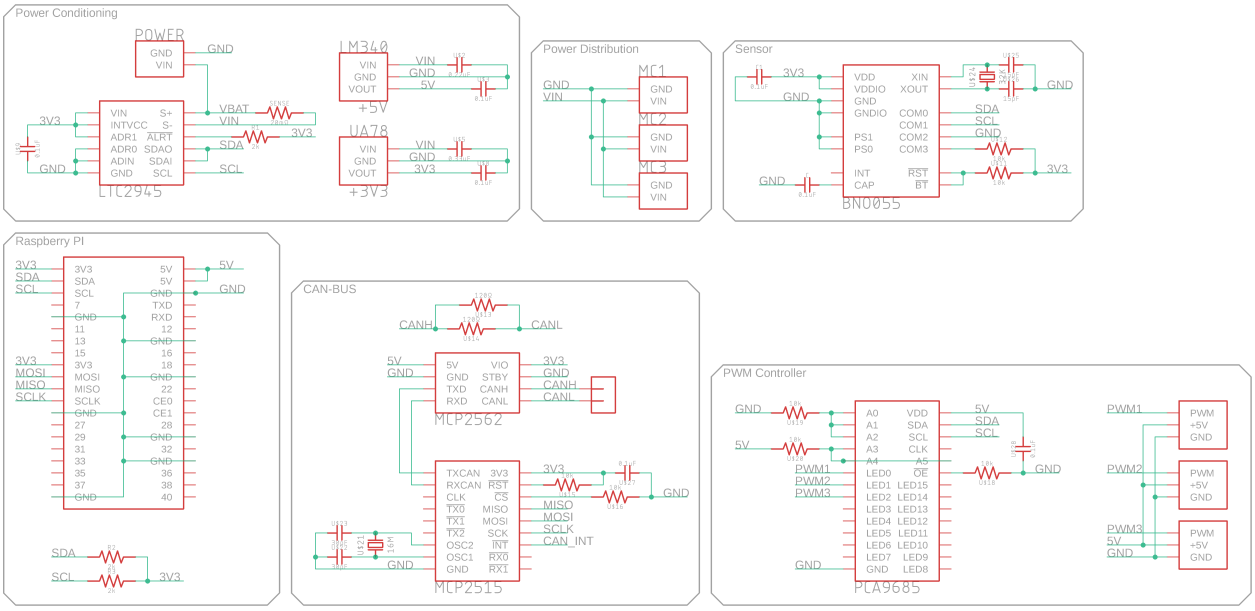

Figure 68: Control Circuit Board Schematic.