GALLOGLY COLLEGE OF ENGINEERING
SCHOOL OF AEROSPACE AND MECHANICAL ENGINEERING
The UNIVERSITY *of* OKLAHOMA

May 4th, 2024

Dr. Sanchez,

Attached is the final report from the Probe Team of the Constellation Measurement Asteroid Project (CAMP). This report contains the overview, specifications, and design progress as of May 4th, 2024, on designing an attitude control and determination system via reaction wheels and thrusters for a CubeSat-based satellite system. Aspects of the CAMP system accompanying both the Probe and the Prototype are detailed in the report, including background research, design processes, and manufacturing/testing of the Probe and the Prototype. If the content of this report may be used in future CAMP capstone projects, recommendations for manufacturing fully operational systems are included.

We hope that this report contains all necessary information to reach a satisfactory status in relation to your CAMP research goals and provides a robust and detailed design for the attitude control and determination system.

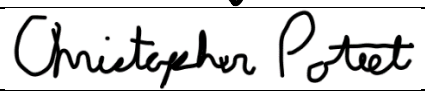
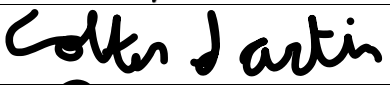
Sincerely, on the behalf of the CAMP Probe Team,

Caden Matthews

Constellation Asteroid Measurement Project (CAMP) Probe Team Final Report

Submitted to: Dr. Bashetty
Submitted by: Caden Matthews, Chris Poteet, Colter Sartin, Mequanent Yohannes
04 May 2024

Signatures

Signatures	
Caden Matthews	
Chris Poteet	
Colter Sartin	
Mequanent Yohannes	

Executive Summary

The Constellation Asteroid Measurement Project (CAMP) is a project seeking to design a mission with all components necessary to gather sensor data and collect samples from Near-Earth asteroids (NEA). The asteroid systems selected for study in this mission include 436724 2011 UW158 and 153958 2002 AM31. UW158 is a single asteroid believed to be coherent due to its high rate of rotation. Spectrometry previously collected on this asteroid suggests it may contain upwards of $5 trillion of platinum. AM31 is a binary asteroid system that could provide useful information regarding the solar system's formation and the study of the gravitational effects in a 2-body orbital system. These systems were selected for study due to their comparable orbital inclinations and semi-major axes, allowing for efficient intercept/study in a single mission.

The CAMP mission is divided into three project teams for this year's Capstone projects. The "Spacecraft" team is responsible for the design of the main spacecraft (or mothership) as well as coordination of the orbital maneuvering necessary to intercept both systems and collect the asteroid samples. The "Sample Collector" team is responsible for the particular design and implementation of the sample collector that will be carried on a probe to collect a physical sample from each asteroid. The "Probe" team is responsible for the design of probes that will be launched from the main spacecraft into each system to gather sensor readings and physical samples from each asteroid [1].

Multiple probe types will complete the desired missions in each system. The types of probes include Visual, Non-Visual, and Lander Probes, and each will be designed using a 6U CubeSat frame. The Visual Probe will gather images and sensor data from the visual light spectrum. The Non-Visual Probe will collect non-visual data such as gravimetry and magnetometry. The Lander Probe will carry the sample collector and return the samples to the main spacecraft. The Visual and Non-Visual Probes will carry Lidar sensors to create a detailed map of the asteroid's surface, and all sensor data will be transmitted and stored on the main spacecraft.

This team will design a navigation system to allow for the orbital maneuvering of the probes in each system. The navigation system will utilize reaction wheels and thrusters to facilitate attitude control and acceleration, respectively. The navigation system will be sufficient to maneuver the Lander Probes into position with enough accuracy to accommodate the sample collection and, additionally, return the Lander Probes from the asteroids to the main spacecraft. The navigation system's design will depend on many constraints, such as mass, power, and size limitations inherent to space missions. At a minimum, acceptable margins for mass and mass distribution of the probes will be defined in relation to the desired performance parameters of the navigation system. Additional design specifications will be defined as necessary to support the missions of the Spacecraft and Sample Collection Teams [2].

Table of Contents

List of Figures

List of Tables

1. Introduction

The Constellation Asteroid Measurement Project (CAMP) is the proposed asteroid-based exploration mission to observe, analyze, and sample three asteroids, 2011 UW158 and 2002 AM31 (binary system) shown in Figure 1. These analyses will include the mapping of these asteroids through the observation of the electromagnetic spectrum by spectrometry and visual data collection of visible, ultraviolet, and infrared processing by the Visual Probe [1]. The Non-visual Probe will observe the physical properties of the asteroids, such as the magnetic field, elemental composition, and gravitational harmonics. The Sample Collection Probe will collect sediment from each of the three asteroids and return it to the mothership. These asteroids were chosen due to their similar orbital inclinations and semi-major axis with respect to Earth. Additionally, previous scans suggest the possibility of 2011 UW158 containing platinum, evaluated up to $5 trillion. The likelihood of this element being found is still uncertain, but there remains invaluable knowledge and insight into the composition of solid-body asteroids and their natural resources.

Figure 1: Current image of UW158 (36m/px) [2]

The explicit scope of the CAMP mission, which is specific to the Probe team, is defined by standardized mission planning in accordance with the NASA Space Mission Architecture Framework. The start of the Probe team's mission is initialized by the launch of the Probe from the central spacecraft within the Terminator orbit. After the injection of the Sample Collection Probe into a circularized low-altitude orbit around the asteroid, the mission phase that is defined is then temporarily suspended as the sample collection of sediment from the asteroid; this is not applicable for the Non-Visual Probe and the Visual Probe. The mission phase is then resumed after the sample collection has reached completion, returning to a semi-stable circularized orbit around the asteroid. The mission ends its phase when the Probe is retrieved by the spacecraft.

A Terminator orbit, as shown in Figure 2, is a specific type of orbit that is in line along the terminator line of the body, which is the boundary between the sun-illuminated side and the dark side of the body of interest. The Terminator orbit was chosen for several advantages, including

Sun synchronization, stable alignment, energy efficiency, and consistent lighting conditions. The orbit is Sun-synchronous, meaning that the spacecraft in the Terminator orbit maintains a constant angle with respect to the Sun throughout the mission. This ensures that the solar panels always remain in view of the Sun and that there is an assurance that the necessary charging capabilities are sufficient. Due to specific alignment with the Sun, the perturbations due to the spherical harmonics caused by the extreme oblateness of the asteroid or the solar radiation pressure are minimized. This means that the orbit can remain stable for the entire mission duration necessary for a successful analysis of the asteroid. By staying in the viewpoint of the Sun consistently, the spacecraft and Probe remain in constant sunlight. In turn, surface feature imaging, mapping, and analysis reach their full potential and can be conducted without significant interferences.

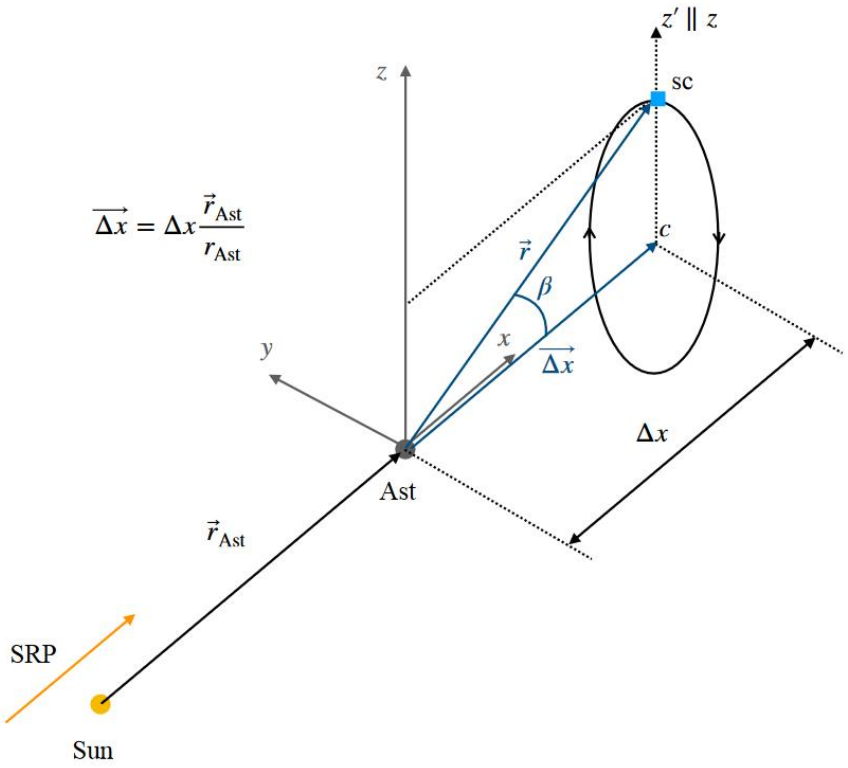


Figure 2: Terminator Orbit of Spacecraft with reference to the Asteroid

The mission will be conducted by nine 6U CubeSats, which are small, standardized spacecraft units offering versatile capabilities in a compact form; these include three Visual Probes, three Non-visual Probes, and three Sample Collection Lander Probes shown in Figure 3. All these components are housed within the main spacecraft, which serves as a mission's launch vehicle and control center. The spacecraft is designed to carry out intricate launch missions to UW158 and perform precise maneuvers to reach AM31.

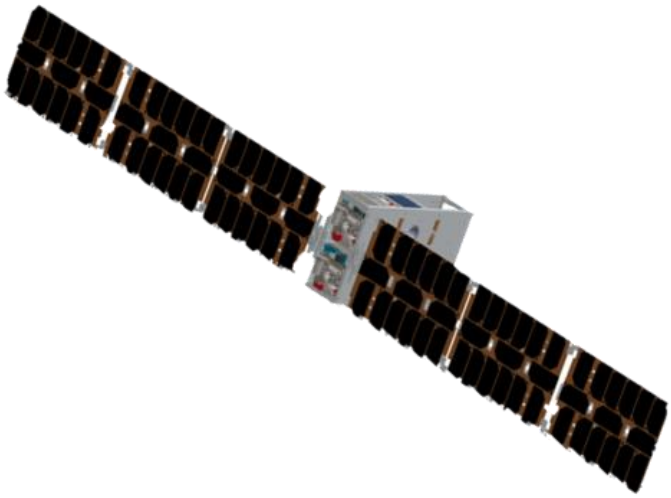


Figure 3: Probe CubeSat with Deployed Solar Panels

The CAMP timeline follows as such:

- Mission Launch - **July 2034**
- UW158 Rendezvous – **Mar. 2040**
- UW158 Depart – **Sep. 2040**
- Earth Fly-by – **Aug. 2042**
    - Return UW158 Samples
- AM31 Rendezvous – **Oct. 2045**

This mission encompasses three primary areas of focus: asteroid exploration, economic advantages, and the assessment of potentially hazardous asteroids. The significance of this mission within the realm of asteroid exploration hinges on the concept of a returnable CubeSat. If such a concept is feasible, it could yield two valuable outcomes. Firstly, it might revolutionize how the space industry utilizes CubeSats by shifting from disposable to reusable systems, resulting in cost savings. Secondly, it could establish the ability to retrieve asteroid samples en route to another asteroid, thereby accelerating the overall timeframe for asteroid exploration.

Furthermore, exploration may pave the way for asteroid mining, presenting economic opportunities for entities engaged in this endeavor. By exploring asteroids rich in rare elements and sending samples back to Earth, the proceeds from such ventures could fund various projects even before full-scale mining operations commence. This, in turn, could fuel space exploration and foster technological advancements. Additionally, the analysis of potentially hazardous asteroids stands to benefit from asteroid exploration. By closely examining the properties of these celestial bodies, we can gather crucial data to determine whether an asteroid is genuinely hazardous or is essential to redirect our efforts to other targets.

Lastly, the utilization of new developments that have yet to be used on small-scale CubeSats could provide massive improvements to the available control systems currently used. These

improvements include using Quaternion-based data streams, the Unscented Kalman Filter to provide cleaner data outputs, the Super-Twisting Sliding Mode control algorithm, and the Inverse Tetrahedral reaction wheel configuration. All these advanced mechanisms provide a robust system of attitude determination and control of CubeSat systems that eliminate a significant number of extensive issues that have plagued rudimentary systems, including unwanted gyroscopic torque, high-frequency vibrations, and inaccurate data streams. In turn, this could improve the longevity of small-scale, high-accuracy spacecraft exploration.

The navigation system's design will depend on many constraints, such as mass, power, and size limitations inherent to space missions. At a minimum, acceptable margins for mass and mass distribution of the probes will be defined concerning the desired performance parameters of the navigation system. Engineering standards such as the NASA Engineering Handbook and the Cal-Poly Institute CubeSat standard define most aspects of the constraints of our Probe design. The Cal-Poly CubeSat standard defines the maximum allowable mass of the assembled CubeSat to be no more than 12 kg [4]. Also, they represent a standard for the location of the center of mass with respect to the overall mass of the CubeSat [5].

In addition to the mass and volume requirements inherent to CubeSats, the power consumption of the Probe components is critical. The power production capability of the Probe is limited by the solar panels mounted externally to the CubeSat. This power is used to charge the onboard battery and is processed for the Probe's components through the power conditioning module in the computing suite. The exposure angle and intensity of the sunlight in contact with the Probe limit power production. The battery pack must be of sufficient capacity to account for sub-optimal solar panel configuration during specific legs of the mission, as well as blackout periods in the shadow of the asteroid or the main spacecraft.

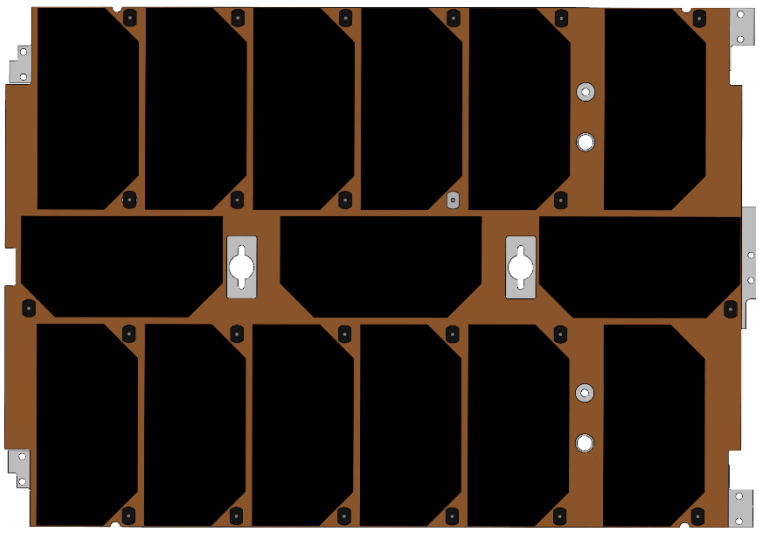


Figure 4: NanoPower MSP and NanoPower TSP Solar Panels

Concerning the design of the probes' solar panels, such as the NanoPower MSP and NanoPower TSP shown in Figure 4, environmental considerations defined in the NASA Handbook for Limiting Orbital Debris limit the options for jettisoning the panels prior to probe retrieval. The determination on whether to jettison the panels into an acceptable orbit, or to design the hinge mechanism such that the panels can be retracted prior to retrieval remains to be decided. Further investigation into feasible options is needed to make this decision.

Our specific constraints are listed below:

- Probe
    - Size limitations of 6U CubeSat Standard and the available storage on the spacecraft mothership. These size limitations are defined according to the Cal-Poly institute CubeSat Standard and the Rocket Lab 6U Satellite Dispenser specifications.
    - The maximum allowable weight for a 6U CubeSat is defined by the Cal-Poly Institute Standard.
    - Distribution of probe components will be limited by the available volume and mass distribution requirements.
    - Power consumption constraints are limited by the solar panel production capability and battery storage capacity. Both of which are also subject to the volume and weight limitations mentioned above.
- Prototype
    - Conforms to the 6U CubeSat standard for sizing and dimensions.
    - Limited budget of $1,500 USD for construction.
    - Limited time for completion, including consideration of time for the ordering and shipping of parts and manufacturing lead times.
    - Subject to gravitational forces in addition to the inertia of the model.
    - The availability of necessary parts for the construction of the Prototype.

These considerations are all weighed against the navigation system's performance capabilities and the Probe's ability to carry out its intended mission. An iterative design process and component selection are necessary to optimize the performance and capabilities of each aspect of the Probe with respect to the mission and design constraints. These design processes and component selections adhered to the necessary objectives for both successful Probe mission phases and Prototype demonstration.

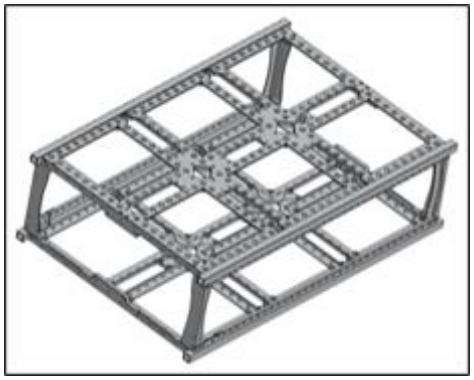


Figure 5: GOMSpace 6U Professional CubeSat Frame

Our specific objectives are listed below:

- Develop a control attitude and determination system for the CubeSat-based Prototype.
- Utilize reaction wheels to control rotational pitch, roll, and yaw components.
- Implement thrusters to control lateral components and provide substantial delta-v.
- Develop control system software to achieve desired outcomes from reaction wheels and thrusters.
- Address communication methods between Spacecraft and Probe to enable navigation management.
- Focus on designing and implementing reaction wheels and associated control software within Probe dimensions.
- Explore thruster options based on mission requirements and constraints.
- Select the most suitable thruster without manufacturing them, establishing design specifications for selection guidance.

2. Background Research

The main component of our research is the design, development, and building of a system of reaction wheels to control the attitude of the probes in orbit around 2011 UW158 and 2002 AM31[1,2]. Due to the necessity to save fuel and energy where possible, there is a need for low-fuel or no-fuel methodologies, limiting the use of thrusters and increasing the reliance on reaction wheels. Reaction wheels are disks that utilize the conservation of angular momentum principles to apply a constant force orthogonal to the direction of rotation. In return, the object, in this case, the Probe, propagates a counteractive force along the components of attitude. These components include pitch, or the rotation around the Probe's x-axis; roll, or the rotation around the z-axis; and yaw, or the rotation around the y-axis. This will provide hypothetically true rotational control of the Probe, with minimal lateral effects that a thruster will counteract. As such, we can perform precise movements with respect to the inertial frame. There are two main categories of reaction wheels: a system of momentum wheels and a control moment gyroscope.

Reaction wheels, such as the one shown in Figure 6, utilizing momentum wheels, would involve high torque and low jitter disks that are positioned in configurations that provide complete 360-

degree freedom of movement. The two configurations for these momentum wheels include the three-axis cartesian configuration and the triangular prismoid configuration. The three-axis cartesian configuration has at least three reaction wheels along the corresponding inertial frame coordinate axes. This configuration allows each wheel to control the components of attitude, being the pitch, roll, and yaw, to be controlled independently without the reliance on additional wheels for singular axis control. The other configuration is a triangular prismoid utilizing inverse kinematics that can use the combination of two reaction wheels to control all components of attitude. This allows the third wheel to act as a redundant addition, providing insurance in cases of mechanical failure.

The control moment gyroscope utilizes the unwanted gyroscopic torque caused by the reaction wheels' perturbations. It can use that torque to transform the coordinates to control the pointing direction or the Probe's x-axis with respect to the reference axis. With this type of attitude control, using only one reaction wheel is necessary alongside a gyroscopic stand containing said reaction wheel. The two configurations for the control moment gyroscope include a single gimbal configuration and a dual gimbal configuration. The single gimbal configuration contains one individual pivotal gimbal to constrain the torque. The dual gimbal configuration consists of two pivotal gimbals which counteract each other to produce angular momentum while saving mass efficiently.

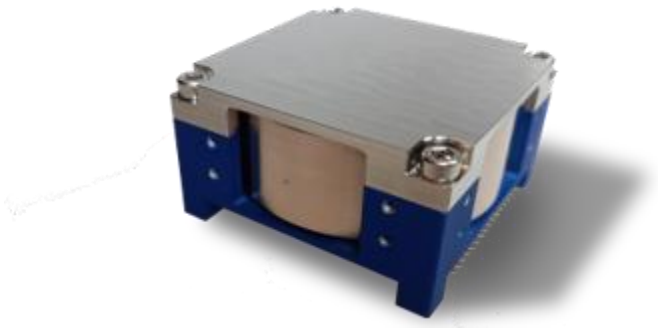


Figure 6: AAC Clyde Space RW400 CubeSat Reaction Wheel [7]

The secondary component of the navigational system will be the thruster, such as the thruster shown in Figure 7, for control of lateral movement. These will be used to supply substantial changes in delta V. They will not be used as often as the reaction wheels but are still important when rendezvousing with the spacecraft. There are two main types of thrusters: chemical and electric. With chemicals, a propellant is accelerated due to a chemical reaction, whereas an electric propellant is accelerated due to electricity and the effects it can cause. The consideration of limited fuel storage has influenced the propellant that can be utilized for the CubeSat thrusters.

Chemical propulsion has three main subcategories. They are liquid, solid, and hybrid. Solid is typically not used because it cannot be turned off once started. Because of this, the most used is liquid; it has two subcategories. One is monopropellant, which means one propellant. Its main advantage is its simplicity, which is why it is the most used form of propulsion. Its primary

disadvantage is that it requires heavy catalyst systems to operate. The other subcategory is bipropellants. As it suggests, there are two propellants used. Its advantages are that it is more suitable for long-term space travel and does not require an ignition source. Its primary disadvantage is that it is corrosive and performs lower than monopropellants.

Electric propulsion also has three main subcategories. They are ion propulsion, hall-effect propulsion, and Thermal/resist jet propulsion. Ion propulsion advantages are high specific impulse and efficiency. Its disadvantages are low thrust, high power consumption, and maintenance. Hall-effects advantages are that they can create more thrust than ion propulsion and have a longer lifespan. Its disadvantages are that it is not as efficient, requires maintenance, and has high power consumption. The last one is thermal/resist jet propulsion, whose main advantage is simplicity, so it finally solves maintenance problems. Its disadvantages, however, are a small thrust generation, and it still has high-power requirements. Since all electric propulsion requires significant amounts of power unavailable to the Probe, electric propulsion will not be used in this design.

Figure 7: Dawn Aerospace 0.8U CubeDrive [22]

The navigation system's control will require an algorithm to resolve the error between the present trajectory, control inputs to the reaction wheels and thrusters, and the desired trajectory at a given instant in the mission. Given the size and mass constraints on the CubeSat Probes, the navigation control computer will be located on the main spacecraft. It will require a source of communication between the probes and the spacecraft. Communication between Probes and the Spacecraft is already a requirement to transmit scan data for storage on the spacecraft and will thus not drive

additional component mass on the probes. Navigational control residing in the main spacecraft will require constraints on the amount of time the Probe can be out of contact with the main spacecraft. There were a multitude of control algorithms in consideration regarding the control of the reaction wheels, including Proportional-Integral-Derivative control (PID), Linear-Quadratic Regulator (LQR), nominal Sliding Modes, and Super-Twisting Sliding Modes.

A Proportional-Integral-Derivative controller is a widely used feedback loop mechanism in control systems. It continuously calculates an error value as the difference between a desired setpoint and a measured process variable and applies a correction based on proportional, integral, and derivative terms. The proportional term produces an output proportional to the current error value, and the integral term integrates the accumulation of past errors. It seeks to eliminate the residual steady-state error over time, and the derivative term predicts system behavior and applies a control output based on the rate of change of the error. This controller algorithm is the simplest to implement, but there were severe drawbacks when used for the Probe team's application. These drawbacks include a linear assumption of the system when our system would be inherently non-linear, significant time delay, sensitivity to unwanted noise in the error signal, and such, which was deemed to be unusable for both the Probe and Prototype.

A Linear Quadratic Regulator (LQR) is an advanced control strategy applied to systems that can be described by linear differential or difference equations in state-space form by minimizing a cost function representing a trade-off between the system's performance and the cost of applying control inputs. The solution to this optimization problem results in a control law that is a linear function of the state where K is the optimal gain matrix computed based on the system dynamics and the matrices Q and R. The matrices Q and R allow the designer to balance the importance of maintaining the system state close to the desired state against the cost of applying control inputs. A high value in Q emphasizes minimizing state errors, while a high value in R focuses on reducing control effort. This controller algorithm is more robust than the PID controller, but there were still severe drawbacks when used for the Probe team's application. These drawbacks include the fact that LQR applies only to linear systems and that the algorithm requires all state variables to be known for complete feedback control, which was also deemed unusable for the Probe team's application.

Sliding Mode Control (SMC) is a robust and efficient control technique used in control systems, particularly for non-linear and uncertain systems. The central component of SMC is the design of a sliding surface, which is a specified condition in the system's state space. When the system's state reaches this surface, it must "slide" along. This, in turn, allows two phases, the Reaching Phase, which drives the system's state to the sliding surface, and the Sliding Phase, where the system dynamics are governed by the sliding condition, to switch infinitesimally fast to ensure that the state trajectory is reached. Since the CubeSat control is a non-linear application, this algorithm would be applicable to the Probe, but there were still a few drawbacks. One significant drawback was the occurrence of chattering, which is a high-frequency oscillation caused by the algorithm that can cause wear to mechanical parts over long periods of time.

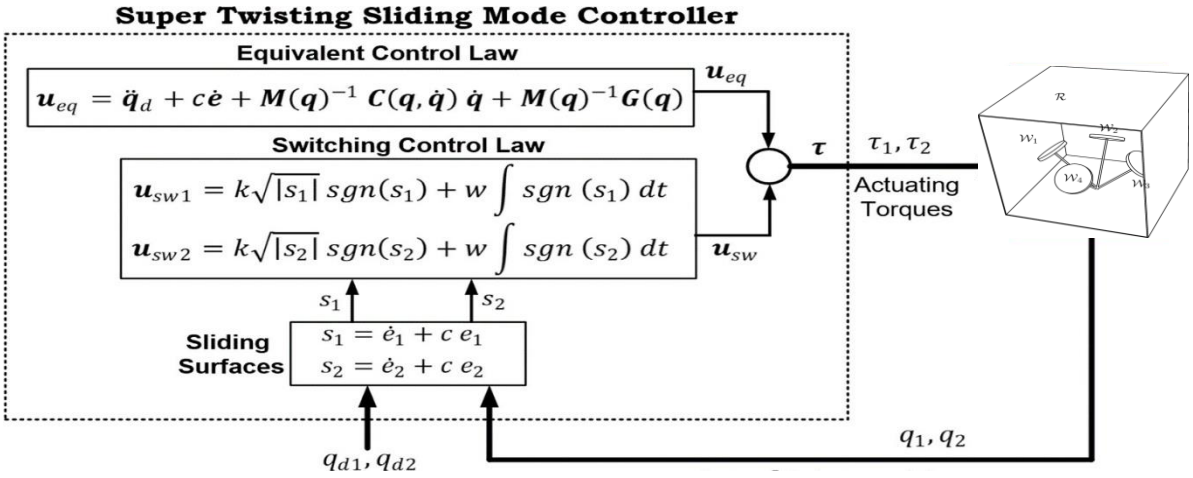


Figure 8: Super-Twisting Slide Mode Block Diagram

Super-Twisting Sliding Mode Control, shown above in Figure 8, is an advanced version of the basic Sliding Mode Control (SMC) technique. It is specifically designed to address one of the primary drawbacks of conventional sliding mode control: the chattering phenomenon. The Super-Twisting Sliding Mode Control is a type of second-order sliding mode control; where unlike the traditional first order Sliding Mode Control, which only considers the state to be on the sliding surface, second order sliding mode control ensures that both the state and its first derivative are on the sliding surface, leading to higher precision and smoother control. Additionally, the algorithm is a continuous control law, unlike the discontinuous control in the basic Sliding Mode Control, meaning it will eliminate all chattering effects entirely. The Super-Twisting Sliding Mode Control is significantly more complex than the standing Sliding Mode Control, Linear Quadratic Regulator, and Proportional-Integral-Derivative controller. Due to its robustness against system uncertainties and external disturbances, making it suitable for systems with modeling inaccuracies or unknown dynamics, and the nonlinearity of the control algorithm, the Probe team decided that the Super-Twisting Sliding Mode Control would be the best option for the application of advanced and accurate CubeSat attitude determination.

3. Design Process

3.1 Probe

This design process began with an existing design for the CAMP probes' components. During the initial review of the existing designs of the three probe types used for this mission, several issues were found with the components and layout proposed by the previous design team. It was discovered that the power consumption of the Lander Probe exceeded the power production estimates for the selected solar panels. The previous iteration of this design justified this decision based on the capacity of the battery onboard the Probe. Upon further review, it was decided that a hard limit would be placed on the power consumption of the Probe's components so that it would not exceed the estimated power production capacity of the solar panels. This decision is justified

based on the probability of battery degradation during the long duration of the flight from launch to asteroid intercept, as well as decreasing the reliance of the entirety of the Probe's components on the performance of the battery.

In addition to the power consumption concerns with the existing design, a volume conflict was found with the magnetometer selected for the Non-Visual Probe. The original magnetometer selected for the mission was the Digital Miniature Magnetometer by Antrix, which measured 182x92x72 mm and weighed 500 g. The original design team made an error in converting the units of measure from centimeters to millimeters and reported the volume of the magnetometer as 18.2x9.2x7.2 mm. This error was reflected in the CAD model of the Probe, and it was discovered that the actual dimensions of the magnetometer would not fit within the volume constraints of the CubeSat. This discovery led to an intensive review of the existing design of the probes and a process of validation of each of the components selected for the mission. Ultimately, this was the only error of this type discovered, and a final selection of probe components was conducted based on market research according to the methods outlined below.

When selecting different components, the Probe team used design matrices. The main factors the design matrices considered were size, weight, integration, power consumption, history, and efficiency. First, let us analyze the different models of magnetometers. Table 1 shows how each factor above was weighed in the decision matrix and the magnetometer model's decision. Everything was ranked 1-5, with 5 being the highest.

Table 1: Magnetometer Decision Matrix

	MAG-3 Satellite Magnetometer	**NMRM-Bn25o485 Magnetometer**	**CGUMM01 - Magnetometer**
Colter			
Size (2.5x)	3	4	1
Weight (2x)	4	5	3
Performance (3x)	4	5	4
History (1.5x)	3	3	3
Power Req (1.5x)	3	4	5
Integration(1x)	4	4	1
Total	40.5	49.5	33.5
Chris			
Size (2.5x)	3	4	1
Weight (2x)	3	4	2
Performance (3x)	4	4	4
History (1.5x)	3	3	3
Power Req (1.5x)	2	3	4
Integration(1x)	3	5	1
Total	36	44	30
Meq			
Size (2.5x)	2	3	1

Weight (2x)	3	4	2
Performance (3x)	3	4	3
History (1.5x)	2	2	2
Power Req (1.5x)	2	3	4
Integration(1x)	2	2	1
Total	28	37	25.5
Caden			
Size (2.5x)	3	4	1
Weight (2x)	3	4	3
Performance (3x)	4	5	4
History (1.5x)	3	3	3
Power Req (1.5x)	3	4	4
Integration(1x)	2	5	1
Total	36.5	48.5	32
Average	35.25	44.75	30.25

From Table 1, it can be seen that the Probe team chose the NMRM-Bn25o485 Magnetometer due to its high performance and critical advantages. It can also be seen from this table that the NMRM-Bn25o485 Magnetometer was chosen because it leads in the categories of size, weight, performance, and integration.

Next, the Probe team noticed that our CubeSat did not have an antenna as part of the communication station. In Table 2, it can be seen how the Probe team decided between the different models through decision matrices.

Table 2: Antenna Model Decision Matrix

	UHF Antenna III	**S-Band Antenna ISM**	**S-Band Antenna Commercial**
Colter			
weight(2x)	3	5	4
Size (3x)	4	5	3
Range(2x)	5	2	3
Power Req (1.5x)	3	3	3
compatibility (1x)	5	2	3
Total	42	40	35
Chris			
weight(2x)	3	4	3
Size (3x)	4	5	3
Range(2x)	5	1	1
Power Req (1.5x)	3	3	3

compatibility (1x)	4	2	2
Total	41	36	28
Meq			
weight(2x)	3	5	3
Size (3x)	3	5	3
Range(2x)	5	3	3
Power Consumption (3x)	4	4	5
compatibility (1x)	5	3	3
Total	42	46	39
Caden			
weight(2x)	3	5	4
Size (3x)	3	5	3
Range(2x)	5	1	2
Power Req (1.5x)	4	3	3
compatibility (1x)	4	3	2
Total	41	39	32
Average	41.5	40.25	33.5

From Table 2, it can be seen that the Probe team chose the UHF Antenna II. It can also be seen from this table that it was chosen because the UHF Antenna II leads in the categories of range and compatibility with our communication system. Even though it only led to these two categories, they were heavily weighted since they were the most important factors when choosing an antenna.

To increase the available space for the sample collector, the Probe team considered the possibility of removing components from the Lander Probe, which was additionally investigated because of the power production concerns mentioned above. We specifically looked at items that were on other probes and were redundant on the Lander Probe. These items were two spectrometers, a lidar sensor, and a visual camera. To make sure this decision was correct, and not needing to keep some of our redundant systems on the Lander Probe, the decision matrix in Table 3 was made.

Table 3: Removal of Redundant Systems Design Matrix

	With	**Without**
Colter		
Size (2.5x)	2	4
Weight (2.5x)	2	3
Redundancy(2x)	5	2
Power Consumption (1x)	2	4
Sample Collector Accommodation(4x)	1	4
Ability to complete mission (2x)	3	4

Total	32	49.5
Chris		
Size (2.5x)	2	5
Weight (2.5x)	3	4
Redundancy(2x)	5	4
Power Consumption (1x)	4	5
Sample Collector Accommodation(4x)	3	5
Ability to complete mission (2x)	3	4
Total	44.5	63.5
Meq		
Size (2.5x)	2	5
Weight (2.5x)	2	4
Redundancy(2x)	5	3
Power Consumption (1x)	3	5
Sample Collector Accommodation(4x)	3	5
Ability to complete mission (2x)	3	4
Total	41	61.5
Caden		
Size (2.5x)	2	5
Weight (2.5x)	2	3
Redundancy(2x)	5	3
Power Consumption (1x)	4	4
Sample Collector Accommodation(4x)	3	5
Ability to complete mission (2x)	3	3
Total	42	56
Average	39.875	57.625

From Table 3, we can see it is viable to remove these redundant systems because it does not affect our ability to complete the mission while freeing up volume, weight, and power consumption. Because of this decision, the sample collection team's available space increased from 1U to 2U. Another byproduct of this decision allowed our power consumption to drop drastically. It made the solar panels that had been selected, along with their dimensions, feasible for the amount of power produced.

Next, the Probe team wanted to ensure the thrusters they used were the optimal type for what we wanted to accomplish in this mission. The propulsion lead evaluated the selection of thrusters using the decision matrix below.

Table 4: Thruster Decision Matrix

	Monopropellant Thruster	Bipropellant Thruster	Ion Thruster	Thermal/Resistojet Thruster
Complexity (2x)	3	3	4	4
Power Requirement (2x)	5	4	1	1
Maintenance (1.5x)	4	2	2	4
Weight (1.5x)	4	3	4	4
Cost (1x)	3	3	3	3
Total	31	24.5	22	25

From Table 4, a monopropellant system is ideal due to the maintenance and power requirements of the other systems. From this, we can see that the thruster that had been selected prior was the correct thruster type. This means no changes will be made to the thruster.

Since our main objective is designing a reaction wheel set, the Probe team chose to replace the ADCS unit selected in the previous design. Replacement of the ADCS unit inadvertently removed a star tracker and a set of magnetometers that were integral to the previous unit. This necessitated the replacement of the integral star tracker unit with an independent unit. The magnetometers were unused in the original design and were thus not considered for replacement. Table 5 shows how the model was selected for the reaction wheels. Table 6 shows how the model for the star tracker was decided.

Table 5: Reaction Wheel Model Decision Matrix

	RW-400	Rwp050	Sinclair 60mNms RW-.06
Colter			
Size (2.5x)	4	3	2
Weight (2x)	2	3	4
Performance (3x)	3	3	4
History (1.5x)	5	3	3
Power Consumption (1.5x)	4	3	1
Integration(1x)	4	3	2
Total	40.5	34.5	33
Chris			
Size (2.5x)	4	3	2
Weight (2x)	3	4	4
Performance (3x)	3	2	4

History (1.5x)	5	3	3
Power Consumption (1.5x)	4	3	2
Integration(1x)	5	3	1
Total	43.5	33.5	33.5
Meq			
Size (2.5x)	4	3	1
Weight (2x)	2	4	4
Performance (3x)	3	2	3
History (1.5x)	5	4	4
Power Consumption (1.5x)	4	3	2
Integration(1x)	4	3	2
Total	40.5	35	30.5
Caden			
Size (2.5x)	4	3	1
Weight (2x)	3	4	4
Performance (3x)	3	3	4
History (1.5x)	5	3	3
Power Consumption (1.5x)	5	3	2
Integration(1x)	5	3	1
Total	45	36.5	31
Average	42.375	34.875	32

Table 6: Star tracker Model Decision Matrix

	ST-200	**ST-400**	**ST-16HP**
Colter			
Weight(2x)	5	3	4
Size (2.5x)	5	3	4
Power Consumption (1.5x)	5	4	3
History(1x)	3	4	5
Total	33	23.5	27.5
Chris			
Weight(2x)	5	3	4
Size (2.5x)	5	3	3
Power Consumption (1.5x)	4	4	3
History(1x)	3	4	5
Total	31.5	23.5	25
Meq			

Weight(2x)	5	3	3
Size (2.5x)	5	4	3
Power Consumption (1.5x)	4	4	2
History(1x)	4	4	5
Total	32.5	26	21.5
Caden			
Weight(2x)	5	3	3
Size (2.5x)	5	3	3
Power Consumption (1.5x)	4	4	2
History(1x)	4	4	5
Total	32.5	23.5	21.5
Average	32.375	24.125	23.875

Table 5 shows that the model that was best for our design was the RW-400. This was because it led to the following categories: size, history, power consumption, and integration. From Table 6, we can see that the selected star tracker model was ST-200. This was chosen because it led or was tied in all the categories that were outlined. This is because, for its size, it is currently the leading star tracker. Also, both these models were the same ones that were in the ADCS unit. Another reason why these were ideal ones that were chosen is because they were already made into a unit.

Next, the Probe team noticed that the radar position was on the same side as the reaction wheels. Since we wanted to allow as much room for the reaction wheels as possible, we made the decision matrix shown in Table 7. The factors that affect this decision are mass distribution, reaction wheel space, and ease with docking. These differ from previous decision matrices because the radar will not undergo any significant alterations.

Table 7: Location Change of Radar Decision Matrix

	Currently	Moved
Colter		
Mass Distributions(1x)	4	3
Reaction Wheel Space (2.5x)	3	4
Ease with docking(2x)	3	3
Total	17.5	19

Chris		
Mass Distributions(1x)	3	3
Reaction Wheel Space (2.5x)	3	4
Ease with docking(2x)	3	3
Total	16.5	19

Meq		
Mass Distributions(1x)	4	4
Reaction Wheel Space (2.5x)	3	4
Ease with docking(2x)	2	4
Total	15.5	22

Caden		
Mass Distributions(1x)	2	3
Reaction Wheel Space (2.5x)	3	4
Ease with docking(2x)	3	3
Total	15.5	19
Average	16.25	19.75

From Table 7, moving the radar is the better decision. It leads or is tied in all categories. This means that there is no negative factor in doing this, so it is the best decision. This would the Probe team to adjust the positioning of the reaction wheels to a location that was determined to be more suitable for proper application.

Table 8: Reaction Wheel Configuration Decision Matrix

	Traditional	**Triangular**	**CMG**
Colter			
Size (2.5x)	3	5	2
Weight (2.5x)	3	4	2
Complexity (2x)	5	3	1
Cost (1x)	4	3	2
Failure Points (2x)	3	5	2
Performance (3x)	4	3	5
Total	47	50.5	33
Chris			
Size (2.5x)	**3**	**5**	**3**
Weight (2.5x)	**4**	**4**	**1**
Complexity (2x)	**5**	**4**	**3**
Cost (1x)	**5**	**4**	**3**
Failure Points (2x)	**3**	**5**	**1**
Performance (3x)	**4**	**4**	**3**
Total	**50.5**	**56.5**	**30**
Meq			
Size (2.5x)	3	5	3
Weight (2.5x)	3	4	1
Complexity (2x)	5	3	2
Cost (1x)	4	4	3
Failure Points (2x)	4	5	2
Performance (3x)	4	5	4

Total	49	57.5	33
Caden			
Size (2.5x)	4	5	3
Weight (2.5x)	3	3	1
Complexity (2x)	5	3	2
Cost (1x)	5	4	3
Failure Points (2x)	2	4	2
Performance (3x)	4	4	3
Total	48.5	50	30
Average	48.75	53.625	31.5

Three configuration designs were deliberated regarding the reaction wheels for the attitude and determination of the CubeSat probe, including a Three-Axis Cartesian configuration, an Inverse-Tetrahedral Triangular configuration, and a Control Moment Gyroscope (CMG) [3].

The Three-Axis Cartesian configuration utilizes three to four reaction wheels orthogonal to the x-y-z axes of the Cartesian Coordination systems, which directly control the Probe's roll, yaw, and pitch. The reaction wheel would then provide a torque that could control these components of attitude and determination. Regarding the Three-Axis Cartesian configuration, the Probe team found that while the implementation of this configuration would be a relatively straightforward process, utilizing the PID control algorithm that would control each axis would not have been an innovative, long-term solution for achieving accurate control of the Probe.

The Inverse-Tetrahedral Triangular configuration utilizes four reaction wheels at an angle, θ, from the reference frame and at an angle, β, of approximately 90 degrees from each reaction wheel. This configuration allows for continuous operation by utilizing fluid inertial frames from the mapping relation of the reference frame, where if one wheel were to break, complete control of the roll, yaw, and pitch of the Probe would still be maintained. We found this capability of redundancy to be appealing while considering options. However, the Probe team had to take into account that this configuration would have to utilize the Super-Twisting Sliding Mode control algorithm, which is much more challenging to implement than the standard PID control algorithm [9].

Lastly, the Control Moment Gyroscope uses one reaction wheel inside a three-axis gyroscopic spheroid to balance perturbations along multiple axes. However, the Probe team found this impractical for what 2qw was ultimately trying to achieve. Commonly, Control Moment Gyroscopes are implemented on large spacecraft and satellites, with miniscule maneuverability when used on CubeSats.

Upon using the Reaction Wheel Configuration Decision Matrix shown in Table 8, the Probe team decided it would be best to proceed with the Inverse-Tetrahedral Triangular configuration for our Probe's reaction wheels. The design, shown in Figure 9, uses four reaction wheels facing away from each other. Quaternions were used to avoid the possibility of singularity issues while computing Euler angles and Rodrigues parameters that could lead to the lack of balanced control

if nominal frames were considered instead [8]. As shown in Figure 31 and Figure 32 in the Appendix, the current stabilization time, without the full control parameters, is approximately 2 seconds, which could be brought down to 1.4 seconds if successful implementation is achieved.

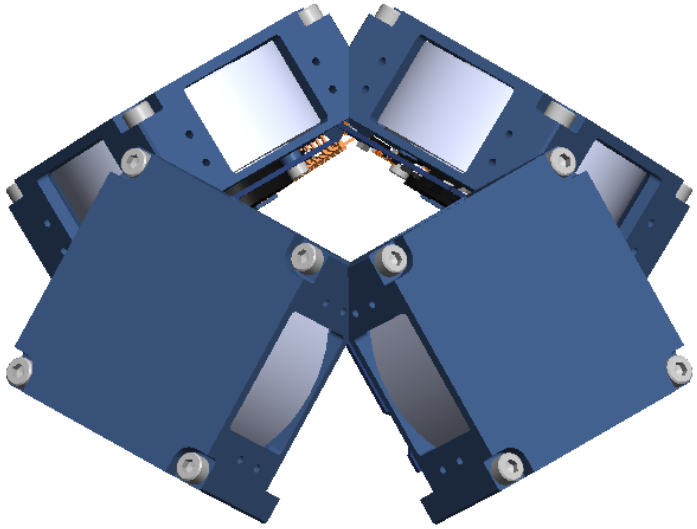


Figure 9: Probe Reaction Wheel Configuration CAD Layout

Once all the Probe's components were selected and the design of the reaction wheels was finalized, a preliminary layout of the probe components was conducted in SolidWorks. The Visual and Non-Visual Probes underwent minimal change, and the only change of layout ultimately performed was replacing the magnetometer in the Non-Visual Probe and adding the UHF antenna for the communication suite. The most intensive design change occurred on the Lander Probe, which houses the sample collector. The space claim for the sample collector team was expanded from 1U to 2U in the front of the Probe, and the sensors listed above were removed to facilitate this expansion and provide a margin for the power production on the Probe. The placement of the antenna on the Lander Probe will be prioritized for mass distribution, and its placement will be replicated on the other two probe types, if possible, to allow for ease of manufacture. The preliminary design for mounting the antenna involves recessing a portion of the outer shell of the CubeSat to minimize the outer thickness while maximizing the range and reception of the antenna within this constraint. This is done to keep the Probe within the volume constraints of the selected RocketLabs dispenser [8,9]. The preliminary layout of the Lander Probe can be seen in Figure 10 below. This drawing has several outer components removed so that the orientation of the internal components can be seen.

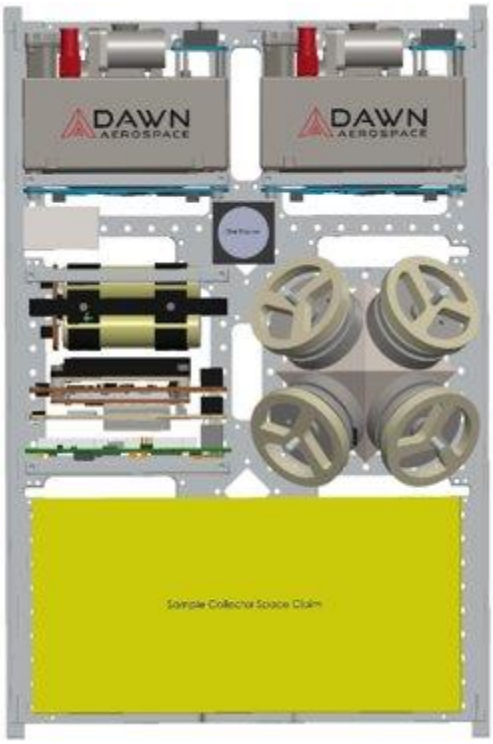


Figure 10: Lander Probe Preliminary Layout

Moving forward, the layout of the Probe's components can be validated when the needed definition from the sample collector is developed. Once the layout is finalized, ancillary equipment such as mounting brackets, shells, electrical cables, and fasteners will be defined. Of particular importance is the design of a "keep-out" box, which will house the reaction wheels and act as a shield to prevent physical interference while the wheels are in operation.

3.2 Prototype and Test Stand

The reaction wheel assembly is the primary design component for this development iteration. The previous section outlines the selection of an inverse-tetrahedral triangular reaction wheel configuration using a quaternion-based super-twisting sliding mode of control. This design concept necessitated designing a unique set of reaction wheels as this design concept has yet to be implemented in commercially available reaction wheels. For this reason, a demonstration of proof of concept was planned to validate the control and performance of the reaction wheel design. To successfully validate the reaction wheel design, the reaction wheels should be able to:

- Maintain a set orientation.
- Perform preset adjustments to attitudinal orientation.
- Compensate for external perturbations and disturbances.

All these tasks should be performed on a probe model comparable in size and weight distribution to the planned mission probes.

To perform this proof of concept within the budget and time allotted for this project, many design compromises needed to be made to the Prototype. The Prototype design simplifications include the following:

- Substituting the commercially selected reaction wheels with
    - Cost-effective EC motors.
    - Steel reaction wheels of simplified design.
    - Cost-effective controllers and battery.
- Substituting the Star Tracker and Radar with an IMU to utilize the Earth's magnetic field for attitude determination.
- Moving the simplified reaction wheel assembly to the center of gravity of the Probe to minimize the effect of Earth's gravity on the model.
- Modeling all components not necessary for reaction wheel function using 3-D printing and weighting.
- Building a test stand to counteract the Prototype's weight while allowing 3 degrees of freedom for attitude maneuvers.

3.2.1 Prototype Reaction Wheel Design

The key differences in the reaction wheel design from the Probe to the Prototype are as follows:

- An Arduino is replacing the command module.
- An IMU is replacing the star tracker.
- The reaction wheels will be of a simpler design than the prefabricated reaction wheels in the Probe.

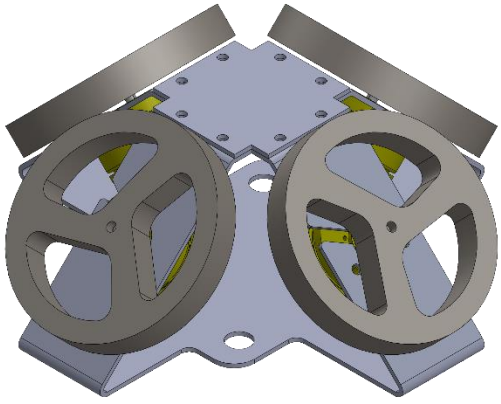


Figure 11: Prototype Reaction Wheel Configuration CAD Layout

The control algorithm utilized remained the same for the Probe and the Prototype. Additionally, since the prefabricated wheels in the Probe included motors, the Probe team had to select motors for our reaction wheel design. The reasons for each specific part replacement and the differences from the Probe are listed below.

Practical considerations drove the replacement of the command module with an Arduino. Arduino was chosen due to its ease of availability compared to the initially selected command module, considering budget, time constraints, and the need for direct coding capability instead of receiving code through an antenna from an external source, which further influenced this decision. Arduino was explicitly selected for its compactness in providing commands to both the motor and IMU, making it a streamlined and efficient solution. Additionally, the cost-effectiveness of Arduino, coupled with its ready availability, enabled the allocation of budget funding to other essential components.

The decision to replace the star tracker with an Inertial Measurement Unit (IMU) in the Prototype was based on the testing environment in Earth's gravitational field. An IMU is more practical for use on Earth compared to a star tracker. The specific choice of the Adafruit BNO055 IMU was deliberate, primarily due to its quaternion design, aligning with the requirements of the reaction wheels. The selection of quaternion design is crucial because Euler angles face limitations, particularly when roll, pitch, and yaw angles surpass 45 degrees due to the invertible nature of the rotation matrix, leading to a singularity. Quaternions, on the other hand, eliminate this singularity issue. Since quaternions were incorporated into the design, the Adafruit BNO055 IMU was chosen for its ability to seamlessly convert these quaternions, saving time and simplifying the integration process.

This quaternion data is fed through an Unscented Kalman Filter, a state estimation filter commonly used in control systems. Unlike the traditional Kalman Filter, which linearizes the nonlinear functions using a first-order Taylor series expansion, the Unscented Kalman Filter utilizes a sampling of sigma points around the mean state estimate propagated through the nonlinear functions of the systems [19]. This is necessary since both the Probe and Prototype are nonlinear applications of attitude determination and control. Traditional filters such as the Low Pass Filter, Madgwick Filter, and Complementary Filter are insufficient for efficient and optimal handling.

While the Unscented Kalman Filter provides a more accurate estimation for nonlinear systems, it comes with increased computational and technical complexity compared to the linear Kalman Filter, but the Probe team ultimately decided that this was the most effective for the Probe and Prototype applications.

Additionally, the Adafruit BNO055 IMU was preferred for its 9 degrees of freedom, encompassing a magnetometer (measuring Earth's magnetic field), a gyroscope (measuring angular velocity), and an accelerometer (measuring linear acceleration) [15]. This comprehensive sensor suite allows the IMU to capture data on roll, pitch, and yaw, along with movement along the x, y, and z axes. This allows the IMU to understand where it is in relation to an artificial horizon and will be able to orient itself with the horizon or any changing degrees about the quaternion-based coordinate system composed of four components: one real part and three imaginary parts, as $q = w + xi + yj + zk$, where $w, x, y,$ and z are real numbers, and $i, j,$ and k are the fundamental quaternion units [20].

Table 9: Comparison of Reaction Wheel Materials and Diameters

1023 Carbon Steel	Part #	Scale	Diameter (mm)	Mass (g)	Inertia (g*mm^2)	Angular Momentum (g*mm^2)/s	Angular Accelaration
	1	1	50	53.6	10537.89	9929853.747	732.6
	2	0.95	47.5	45.94	8153.96	7683476.508	473.39
	3	0.9	45	39.05	6222.43	5863395.789	620.34
	4	0.85	42.5	32.88	4675.61	4405827.303	825.56
	5	0.8	40	27.4	3452.94	3253705.362	1117.89
	6	0.75	37.5	22.56	2500.57	2356287.111	1543.65
	7	0.7	35	18.33	1770.99	1668803.877	2179.57
	8	0.65	32.5	14.66	1222.6	1152055.98	3157.21
	9	0.6	30	11.52	819.33	772054.659	4711.17
	10	0.55	27.5	8.86	530.27	499673.421	7279.31
	11	0.5	25	6.64	329.24	310242.852	11723.97
	12	0.45	22.5	4.83	194.4	183183.12	19855.97
	13	0.4	20	3.38	107.86	101636.478	35787.13
2014-T4 Aluminum	**Part #**	**Scale**	**Diameter (mm)**	**Mass (g)**	**Inertia (g*mm^2)**	**Angular Momentum (g*mm^2)/s**	
	1	1	50	19.1	3754.91	33794190	
	2	0.95	47.5	16.37	2905.46	26149140	
	3	0.9	45	13.91	2217.21	19954890	
	4	0.85	42.5	11.72	1666.04	14994360	
	5	0.8	40	9.76	1230.37	11073330	
	6	0.75	37.5	8.04	891.02	8019180	
	7	0.7	35	6.53	631.05	5679450	
	8	0.65	32.5	5.22	435.64	3920760	
	9	0.6	30	4.1	291.95	2627550	
	10	0.55	27.5	3.16	188.95	1700550	
	11	0.5	25	2.37	117.32	1055880	
	12	0.45	22.5	1.72	69.27	623430	
	13	0.4	20	1.2	38.43	345870	

For the reaction wheel systems, instead of utilizing prefabricated, fully assembled reaction wheels, the wheels will be machined and work in tandem with electronic motors. Multiple factors went into the decision to utilize alternative reaction wheels for the Prototype, including the cost of products, material difficulties, and the differences between Earth's gravitational field and the micro gravitational scenarios that the Probe will undergo while in orbit. The statistical average of machined sets of reaction wheels from companies including Rocket Labs, AAC Clyde Space, and NewSpace Systems was around $30,000 USD to $50,000 USD, which is significantly over the allocated budget constraint of $1,500 USD that is bestowed to the Probe team. Thus, the Probe team concluded that it would be critical that we are able to construct reaction wheels with affordable components.

Additionally, the type of material to be used for reaction wheels was extensively discussed. For Titanium, a single ½ inch sheet would have cost upwards of $1,000 USD to machine. It would have provided an incredible amount of difficulty to cut reaction wheels of a highly accurate standard properly. Hence, this material proved impractical for the Probe team's available equipment and capabilities. For Aluminum and Steel, specifically 1023 Carbon Steel and 2014-T4 Aluminum, the Probe team conducted material analysis for reaction wheels of multiple diameters to analyze the mass, moment of inertia, and angular momentum to compare each material, as seen in Table 9. From this material analysis, the Probe team concluded that Steel would provide optimal performance for counteractive angular momentum against the gravitational acceleration of Earth when the Prototype is in tilt and rotation.

The manufacturing of the reaction wheels consisted of CNC machining a 6.35 mm sheet of Steel to produce wheels that would be 42.5 mm in diameter. The reaction wheels were explicitly designed for the most significant portion of the mass to be placed farthest from the center of the axis of rotation since that would produce the efficiency, accuracy, and necessary counteractive torque produced by the reaction wheels to oppose the Prototype's 3-axis tilt due to the gravitational acceleration of Earth, due to maximizing the moment of inertia of the reaction wheels. The most critical portion of the reaction wheel design is the geometry of the center hole for the motor shaft. The selected motors (mentioned below) depend on a friction fit between the motor shaft and the component being mounted (reaction wheels). Additionally, the motors have a maximum axial load limit of 200 N (approximately 45 lbf). It was found that machining the shaft-hole of the reaction wheels to 1.985 mm in combination with a shrink fit onto the motor shaft allowed for this connection.

To mount the reaction wheels to the motors the following procedure was followed:

- The motors were cooled to –20° C and the reaction wheels were warmed to approximately 100° C.
- The shaft hole on the reaction wheels were coated in Loctite 609 joining compound.
- The motors were pressed down onto the reaction wheels using a press designed to apply no more than 200 N of force. (Note that a spacer made from 22 ga sheet steel was used to maintain a space between the motor and wheel for approximately 0.80 mm.)
- The spacers were removed once cooled and the motors cured at room temperature for at least 48 hours before use.

Unlike the integrated motor/wheel design selected for the probe, the prototype would implement reaction wheels with external motors and controllers. The Probe team finalized the motors to be Maxon EC Flat 20 mm sensor-less motors that would provide upwards of 9450 rpm. The ESCON Module 24/2 controllers were selected based on Maxon’s motor catalogue based on the motors we selected. After the prototype was entirely manufactured and work began on configuring the motors and implementing the control algorithm, it was discovered that the controllers selected were not compatible with sensorless motors contrary to the specifications provided by the manufacturer. Thus, the motor configuration was not completed. Further iterations of the prototype design are recommended to utilize motors with integral Hall-effect sensors along with the ESCON Module 24/2 motor controllers obtained from Maxon [16]. The motors with integral hall sensors provide more precise control, especially at low RPM, than sensor-less motors.

The control algorithm, which is the Super-Twisting Sliding Mode, will be utilized for both the Probe and the Prototype due to the robustness of the controller that the Proportional-Integral-Derivative controller and the Linear Quadratic Regulator could not provide. Not only can this control algorithm counteract the perturbations caused by asteroids due to the spherical harmonics of the irregular shape, but it also eliminates unwanted, high-frequency oscillation in the control signal while providing the same robust system when demonstrating on Earth.

For the Probe attitude determination and control, the Proportional-Integral-Derivative (PID) controller falls short in handling complex dynamics and external disturbances. While the PID controller is capable of providing stable control in linear systems with predictable behavior, its performance significantly degrades in the presence of non-linear dynamics and unpredictable perturbations, such as those encountered in space environments. The Probe team did develop a system using a PID controller in the case of failure regarding the Super-Twisting Sliding Mode, as the Prototype architecture is compatible with the PID controller when used in conjunction with rudimentary Eulerian angle data streams.

While the Linear Quadratic Regulator (LQR) controller excels in minimizing a given cost function, which is typically a combination of control effort and deviation from the desired state, there are significant limitations that arise in dealing with non-linear systems and external disturbances, which are prevalent in space missions. The LQR controller assumes a linear system model, which becomes inadequate for handling a spacecraft's complex and highly non-linear dynamics navigating through an asteroid field, so the Probe team decided not to utilize this algorithm.

The Sliding Mode Controller is a robust control strategy known for its ability to handle non-linearities and uncertainties, which makes it compatible with the Probe's applications. It operates by driving the system's state to a predefined sliding surface, thus ensuring system stability [9]. However, one of the main drawbacks of the traditional Sliding Mode Controller is the phenomenon known as "chattering." This refers to the high-frequency oscillations that can occur in the control signal, which can induce excessive wear and tear on motors and degrade the overall performance of the Probe [8]. Due to the considerable possible length of time that the Probe may be in operation, this wear of components was not an acceptable trait for the Probe team. Thus, it decided not to utilize this algorithm.

After the Probe team evaluated the controllers, as mentioned earlier, the Super-Twisting Sliding Mode Controller was chosen for the Probe attitude determination and control. This decision was driven by the controller's enhanced robustness and ability to counteract the limitations of the other controllers. The Super-Twisting Sliding Mode algorithm, an advanced version of the traditional Sliding Mode Controller, effectively addresses the chattering issue by providing a smooth control action. This is crucial for minimizing wear on the Probe components and ensuring precise attitude control.

3.2.2 Prototype Probe Design

To model the layout and weight distribution of the Lander Probe, the significant components, including the thrusters, battery, computing suite, star tracker, radar transmitter and receiver, and a space claim box for the sample collector, were 3-D printed and weighted proportionally to the true weight. The components were modeled in a simplified space claim and left hollow so that after they were printed, they could be weighted with lead weights suspended in a silicone solution to prevent shifting inside each component. The components were weighted to approximately 20% of their original weight due to the lower torque and power of the cheaper motors used in the Prototype.

The probe's volume constraints were represented using a MakerBeam frame built to contain 6Us of space internally and show the outer dimensions of the CubeSat rails. The frame is mounted to a wooden base to provide structural stability and facilitate mounting the 3-D printed components and the test stand. The Prototype probe model can be seen with the 3-D printed components, wooden base, and MakerBeam frame in Figure 12 below.

Figure 12: Protype Probe Model

3.2.3 Test Stand Design

The test stand serves as a stable base for the Prototype probe to counteract the force of gravity on the probe and allow for 3 degrees of freedom in movement for the probe. The test stand plays a critical role in simulating the conditions that the Prototype probe will encounter in space, partially mimicking the microgravity environment of space. A ball joint connection was selected for the interface between the Prototype probe and the base to achieve this. Market research was conducted, and a RAM 1" mount was selected for this connection. The selected mount can be seen in Figure 13 below.

Figure 13: RAM Ball and Socket Mount

Due to the configuration of the mount, it was determined that the rotational limitations of the ball were 360° about the y-axis, 97° about the x-axis, and 31° about the z-axis of the probe. To avoid maneuvers that could cause the model to get "stuck" in the groove about the z-axis, it was decided to limit the maximum inclination of the Prototype probe on the test stand to 30° in both the x- and z-directions. Ultimately, the simplest way to implement this constraint was to set the height of the test stand such that the probe would contact the table before exceeding the desired maximum inclination. It was determined that a height of 85 mm from the table to the base of the prototype probe would accomplish this. Coincidentally, the height of the RAM mount selected for the ball joint interface was exactly 85 mm. Because of this, the inclination limit was set from the base of the prototype probe to the top surface of the base material, simplifying the requirements for the base of the test stand.

The base was cut from the same wooden sheet as the probe base in an x-configuration where the outer perimeter extended past the outermost reach of the prototype probe by 40 mm in each direction. The ball of the RAM mount was mounted to the probe model's wooden base at the probe's center of gravity, and the RAM mount's socket was mounted to the center of the test stand base. A model of the test stand and Prototype probe can be seen in Figure 14 below.

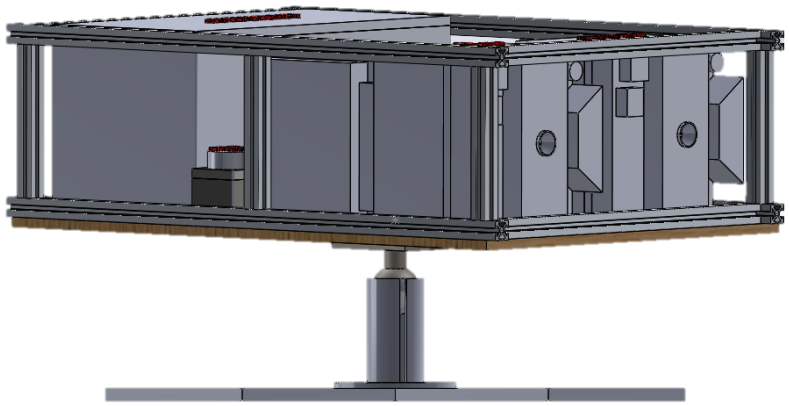


Figure 14: Prototype Test Stand Assembly

In future iterations of prototype testing, it is recommended that a 3-D gyroscopic mount be designed to house the prototype. This would better mitigate the force of gravity on the probe and facilitate 360° of rotation about all 3 prototype axes as friction is minimized with the 3-hoop system when compared to the ball and socket system. If a more complex iteration were to be taken, then the use of an air semi-hemisphere could be implemented to completely minimize friction in the orientation system, though this method of testing would be significantly more complex in terms of manufacturing than the 3-D gyroscopic method.

4. Engineering Standards

As part of this project, the team is expected to follow different engineering standards. Having Engineering standards will ensure Safety, Reliability, Interpretability, Compliance and Regulations, Risk Management, etc. To ensure the successful completion of this project, the Probe team will follow the standards that are listed below:

- **ISO/IEEE 15288: Systems and Software Engineering - System Life-Cycle Processes:** This standard was applied to ensure that all stages of the system life cycle are adequately addressed. Used to ensure that the Probe-specific hardware would remain operatable for the full mission.
- **IEEE 1012: Standard for System, Software, and Hardware Verification and Validation Software Reviews and Audits:** This standard is applied for verification and validation of the system, software, and hardware components of the CubeSat. Used to verify each Probe component is applicable to other standards pertaining to space life cycles and to provide validation that each component would be capable of long-term missions.
- **IEEE 828: Standard for Configuration Management in Systems and Software Engineering:** This standard could be applied for configuration management, ensuring that all configurations of the CubeSat are appropriately documented and controlled. Used this standard to ensure all Probe and Prototype components were adequately documented.

- **ISO/IEEE 29148: Systems and Software Engineering - Requirements Engineering:** This standard was applied to ensure that all requirements for the CubeSat are properly identified, documented, and managed.
- **NASA/SP-2016-6105: NASA Systems Engineering Handbook:** This handbook provides comprehensive guidance on NASA's systems engineering processes and requirements and was used as a reference throughout the project. Used this standard to ensure that the proper conduction of building and design processes were met.
- **NASA-STD-6016C: NASA Standard Materials and Processes Requirements for Spacecraft:** This handbook provides the technical details and regulations for materials utilized for space operations. This standard was used to conduct finalized decisions on proper, structurally integral materials for the Probe.
- **NASA-STD-7009: NASA Standard for Models and Simulations:** This standard was applied for the models and simulations that were used in the design and testing of the Probe and Prototype.
- **NASA-HDBK-1005: NASA Space Mission Architecture Framework (SMAF) Handbook for Uncrewed Space Missions:** Defines the scope of uncrewed space missions, including the comprehensive guide for the planning, design, and execution of uncrewed space missions. Used this standard to provide a framework for the planning phases of the Probe's defined mission.
- **NASA-HDBK-4001: Electrical Ground Architecture for Unmanned Spacecraft:** Defines spacecraft grounding architecture at the system level and focuses on minimizing electromagnetic interference of subsystems. Used this standard when designing the component layout of the Probe to minimize interference.
- **NASA-STD-4009: Space Telecommunications Radio Systems (STRS) Architecture:** Designed for the detailed implementation of space telecommunication systems, providing essential guidance to ensure effective communication in space missions. Used this standard when considering available robust communication systems for the Probe.
- **NASA-GB-8719.13: NASA Software Safety Guidebook:** This standard focuses on the analysis, development, and assurance of safety-critical software, which includes firmware and programmable logic. Used this standard for detailed programming logic in MATLAB, Python, and C++ languages.
- **NASA-HDBK-8719.14: Handbook for Limiting Orbital Debris**: This standard provides background and reference materials to aid in understanding the foundation and science behind predicting and limiting orbital debris. This handbook is particularly useful for any program that launches material into space. This standard was used for the consideration of methods for jettisoning the Probe's solar arrays.
- **ANSI/EIA 632: Processes for Engineering a System:** This standard provides an integrated set of fundamental processes to aid a developer in the engineering or re-engineering of a system. Used this standard in conjunction with ISO 31 NASA/SP-2016-6105.

- **IEEE 1220-1998: Standard for Application and Management of the Systems Engineering Process:** This standard provides guidance on managing a system and includes useful guidance on developing a Systems Engineering Management Plan (SEMP).
- **IEC 60027 Use of the International System of Units:** Standardize the use of quantities and units to ensure consistency. Used this standard to define all measurement quantities of both the Probe and Prototype would be in the International System of Units.
- **ISO 286-1: Basis of Tolerances, Deviations, and Fits:** This standard defines the classes of tolerance fits and methods of calculating tolerances for fits. This standard was used to calculate the tolerances for the holes in the reaction wheels to be press-fit onto the shaft of the reaction wheel motors for the Prototype.
- **ISO 286-2: Tables of Standard Tolerance Classes and Limit Deviations for Holes and Shafts:** This standard contains tables and best practices for determining tolerances for different classes of fits for holes and shafts. This standard was used to verify the tolerances calculated using ISO 286-1 for the press-fit connections between the shafts of the motors and the holes in the reaction wheels for the Prototype.
- **IEEE 1451: Networked Smart Transducer Interface Standard:** This standard addresses the interoperability and interchangeability of sensors, including IMUs. This standard was used when creating the interface for the BNO055 IMU system in Python for the Prototype.
- **IEC 60721: Classification of Environmental Conditions:** This standard classifies the environmental conditions that electrical and electronic equipment can encounter and provides guidance on how to design equipment to withstand these conditions. This standard was used for the wiring of the Radar/Star Tracker system for the Probe.

5. Final Design Details - Chris

5.1Probe

This design iteration began with a comprehensive review of the legacy design definition. Designs for all three probe types (Visual, Non-visual, and Lander) were evaluated against engineering standards relevant to this mission as well as the constraints of the mission itself, including a mass restriction of 12 kg as defined by the Cal Poly CubeSat Standard, a volume constraint of 6U as defined by the CubeSat Standard and selected for this mission, power consumption limitations defined by the production capability of the selected solar panels. The specifications for each component were confirmed against data provided by the manufacturers and validated for the selected purpose of the existing design. From this design review, two significant changes were made. First, the magnetometer selected for the non-visual probe was replaced. The initially selected magnetometer was validated in error by the previous team when the dimensions of the unit were recorded as 18.2 x 7.2 x 9.2 millimeters instead of centimeters. This magnetometer was replaced by the NMRM-Bn25o485 Magnetometer with dimensions of 99 x 43 x 17 mm, which was verified to fit within the non-visual probe alongside all of the other components of the probe.

In addition to replacing the magnetometer on the non-visual probe, major changes were made to the lander probe to facilitate expanding the Sample Collectors team’s space claim from 1U to 2Us

as well as reducing the power consumption of the probe below the production capabilities of the solar panels. The components removed from the lander probe consist of the IR spectrometer, Lidar, and visual camera sensors, which are redundant in the lander probe. The ADCS system from AAC Clyde Space was also removed and replaced with an attitude control system of our unique design. Our design utilized four AAC Clyde Space RW400 reaction wheels in an inverse tetrahedral-triangular configuration designed to fit within 1U of space. This necessitated the addition of an independent star tracker to replace the integral star tracker from the AAC ADCS system. For this, the AAC Clyde Space ST-200 Star Tracker was selected. The navigation system's design and layout, including the Dawn Aerospace Thrusters, Reaction Wheels, and Star Tracker, is compatible with all three probe types. It is implemented in the same configuration and layout as shown in Figure 13 below. Note that some panels and brackets have been removed for clarity.

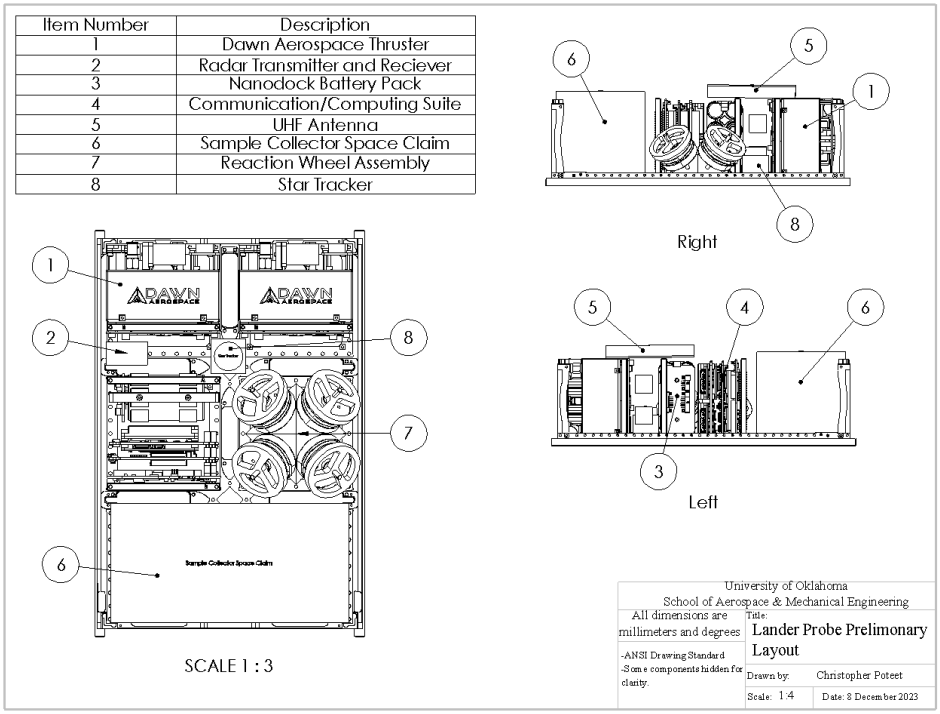


Figure 15: Lander Probe Layout

In addition to component selections and layouts for the probes, a super-twisting-sliding mode control algorithm based on quaternion kinematics was selected to control the probes' navigation system. This algorithm provides highly optimized control while reducing energy requirements and jitter typical to PID-cartesian-based control algorithms.

To complete the design work on the probes, the sample collector design must be completed, the configuration of the solar panels needs to be further defined, and wiring harnesses and assembly definitions need to be completed. To elaborate on the configuration of the solar panels, the mechanism by which the solar panels are either jettisoned or retracted must be defined. This is

essential for the lander probe to be recaptured by the spacecraft and stored within the ship without any interference.

5.2Prototype

The primary purpose of the prototype is to demonstrate the performance of the selected control algorithm and reaction wheel configuration in implementing attitude control and determination on a model of the probe.

The prototype probe consists of reaction wheels (CNC machined) mounted to Maxon Flat EC motors in the same configuration as the reaction wheels defined for the probe. The prototype utilizes independent Maxon motor controllers and an Arduino UNO to implement the control algorithm. Attitude measurements and determination are performed using an IMU.

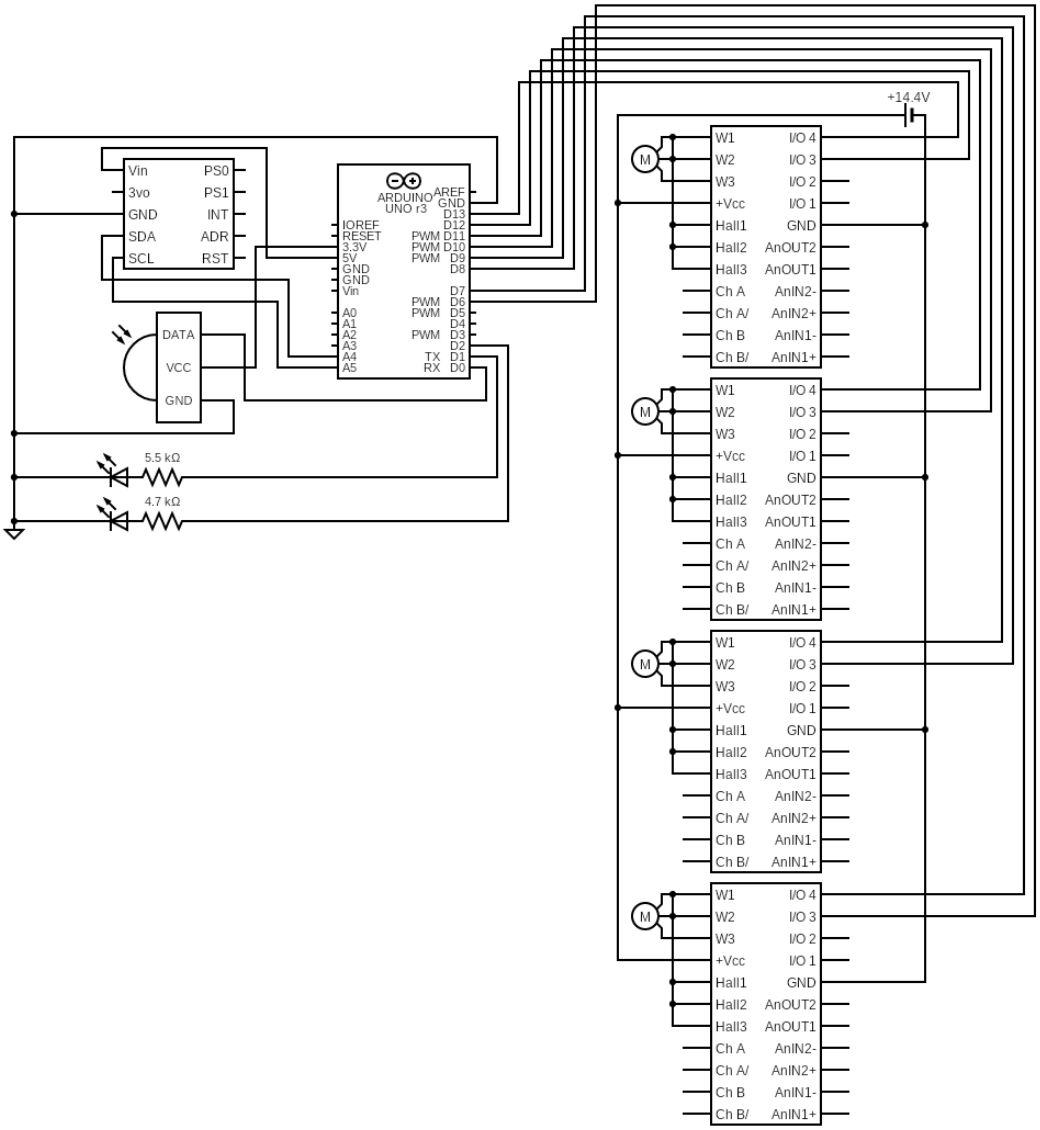


Figure 16: Motor-Controller-IMU-Arduino Integrated Circuit Diagram

In Figure 16 above, the circuitry for four Maxon EC Flat 20mm motors with Hall Sensors, four ESCON Module 24/2 motor controllers, an Adafruit BNO055 IMU, and an Arduino UNO is displayed to show the wiring necessary for a fully operational system. Each of the four motors has individually configured and tuned controllers, with the motor connecting to PIN W1, PIN W2, and PIN W3 for the 3-phase motor windings per Maxon standard configuration and the integrated hall sensors connecting to PIN HALL1, PIN HALL2, and PIN HALL 3. PWM signals can be sent via PWM digital pins on the Arduino UNO; if more PWM digital pins are needed for the resulting system, the Arduino MEGA will be more suitable for the application due to the increased number of digital PWM pins or digital PWM programming. However, this will be of lower quality and would result in a less robust system. The motor controller pairing is powered through an external 14.4V LIPO battery, with the controller being wired in parallel, each containing a built-in voltage step-down unit to provide the system with 5.5V per controller. It has been proven to be safe for operational use.

The Adafruit BNO055 IMU is powered through the Arduino UNO Vout pin, providing a safe 5V for fully operational use. SDA and SCL signals are sent PIN A4 and A5, respectively, which are the only pins capable of receiving the types of values necessary for IMU integration. The system includes an IR sensor connected to PIN D0 for remote configuration and starting sequencing. Additionally, there are two LED lights, a red LED wired in series with a 5.5k ohm resistor to PIN D1 and a green LED wired in series with a 4.7k ohm resistor to PIN D0. This is for the purpose of visualizing the successful calibration and initialization of the Adafruit BNO055 IMU system in the case of the green LED and visualizing an error in the system in the case of the red LED. The ESCON Module 24/2 has integrated LEDs that will signal the same cases as the external LEDs, only pertaining to the controllers and motors themselves. All systems, including the Maxon EC Flat 20mm motors with Hall Sensors, ESCON Module 24/2 motor controllers, Adafruit BNO055 IMU, Arduino UNO, IR sensor, and LED lights, should be ground to a common Earth source. Hence, no voltage leakage occurs, and the three voltage sources, 5V, 5.5V, and 14.4V, are not intermixed with one another.

To ensure consistent performance of the reaction wheels in every axis of rotation, the reaction wheel assembly was moved to the center of the prototype. All of the probe components not required for the operation of the reaction wheels were 3D printed from PLA and weighted such that the model probe weighs approximately 20% of the true probe weight. The prototype was mounted inside a MakerBeam frame to show the volume constraints of the probe, and the model was mounted atop a ball joint/test stand to allow for rotation with 3 degrees of freedom. The prototype is intended to implement the same super-twisting-sliding mode control algorithm as the probe. The prototype test stand configuration can be seen in Figure 14 below.

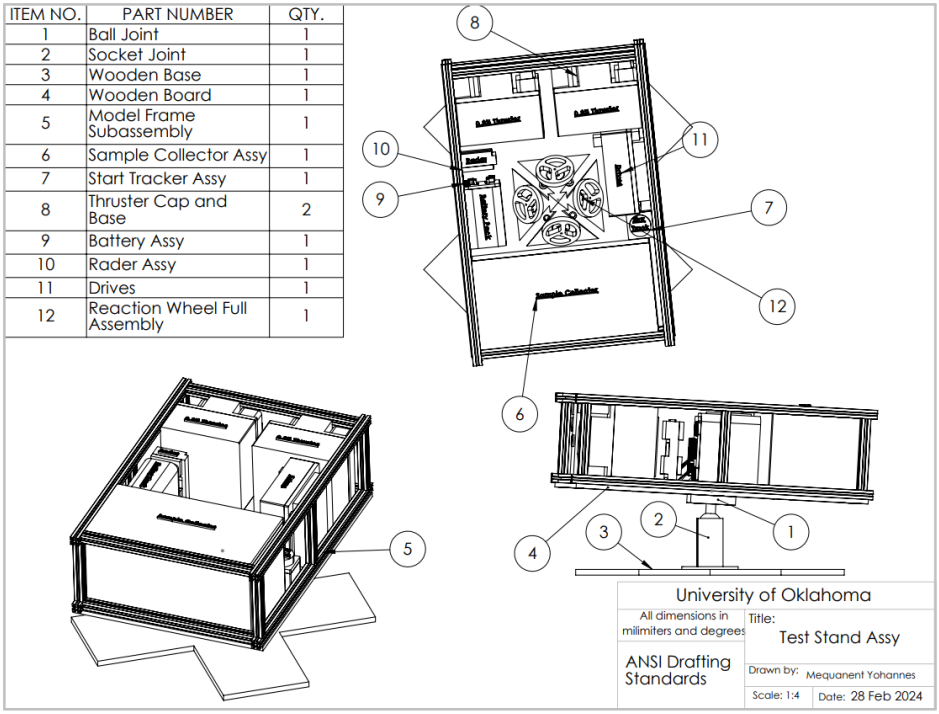


Figure 17: Prototype Test Stand

Some work remains to be completed for the prototype. There is a compatibility issue between the motors and controllers selected for the prototype that this team did not have the budget or time to resolve. To resolve this issue, it is recommended that Maxon's 20 mm flat EC motors with integrated hall sensors be ordered to replace those on the prototype. New reaction wheels will need to be machined and mounted to the new motors, which can then be swapped into the reaction wheel base on the model. This should resolve the configuration error with the motors and allow the control algorithm to be implemented with more precise control of the motors at low RPM compared to the sensorless motors currently installed.

Additional improvements that may be implemented involve the test stand for the prototype. The ball joint stand allows for rotation about 3-axis but is limited by the range of motion of the joint. Additionally, because the joint is small relative to the size of the prototype, the effect of gravity on the performance of the reaction wheels increases exponentially with increasing inclination. This severely limits the effective range of motion for the prototype on this test stand. To mitigate this issue, it is recommended that a gyroscopic test stand be implemented. This would allow for 360° rotation about all three axes and will also mitigate the increase of gravitational effect with increasing inclination. Gravity will still affect the prototype in ways that the probe will not experience, but this should reduce the significant factors being overcome by the reaction wheels to the moment of inertia of the model alone.

6. Design Testing and Results

6.1 Reaction Wheels

To ensure structural integrity of our reaction wheel design and material, ANSYS finite element analysis (FEA) was conducted on the 42.5 mm diameter, 6.35 mm thick reaction wheels made of 1023 Carbon Steel material. By subjecting the reaction wheel design to virtual testing under various operating conditions and environmental scenarios, the Probe team could identify potential stress concentrations. As shown in Figure 18, the maximum total deformation of an individual reaction wheel was 0.00030332 mm at complete failure of the center motor spindle locking in place. This data ensures that the potential deformation of the reaction wheel is negligible and would meet the Probe team's standards and various material/safety-centric engineering standards for safe demonstration.

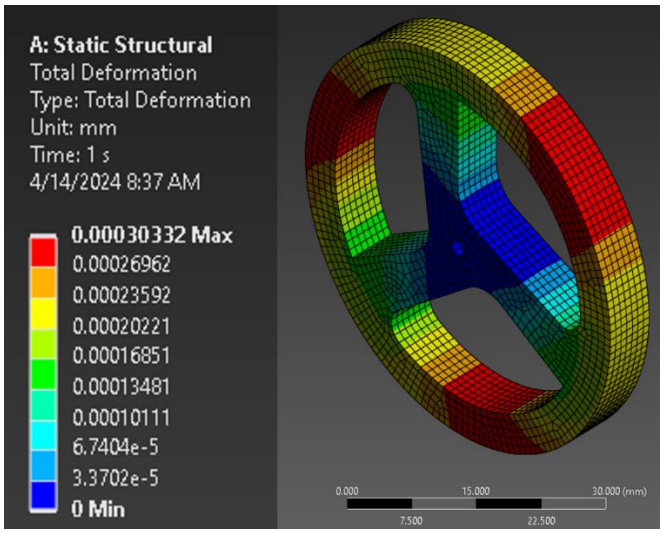


Figure 18: Total Deformation Analysis of Reaction Wheel

As shown in Figure 19, the maximum total deformation of the reaction wheel base was 0.0022788 mm when full rotation torque and static load by the motor-reaction wheel system is applied when mounted with Countersunk M2 screws. This data ensures that the bases could withstand any noticeable forces applied when the reaction wheels were to spin at maximum RPM, even with the stress concentration around the holes as these have minimal deformation under load. The ANSYS analysis provides valuable insights into the behavior of our reaction wheel system to ensure the success of our Prototype.

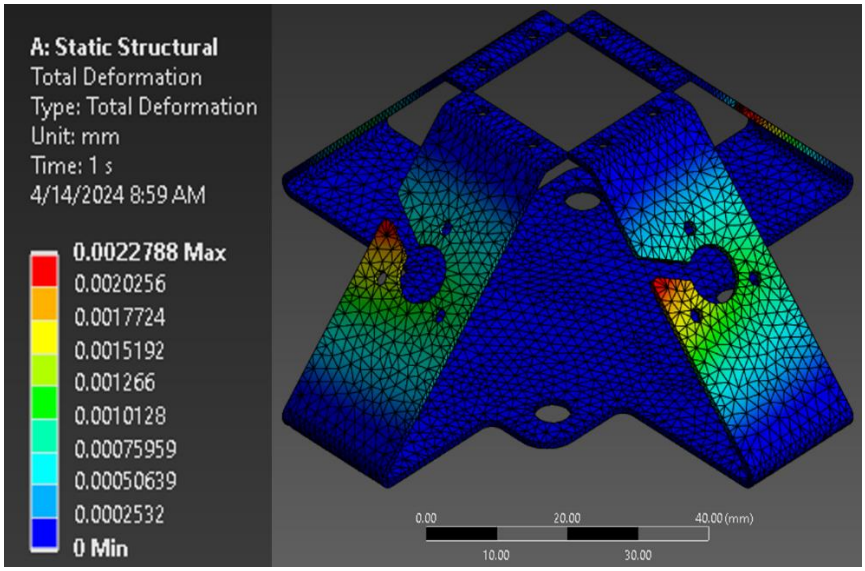


Figure 19: Total Deformation Analysis of Reaction Wheel Base

As seen in the Counteractive Inertia of Reaction Wheels code, a simulation was created to calculate and visualize different diameters and materials' effects on the necessary angular acceleration to provide counteractive torque. The Probe team deemed it essential to enhance and verify the efficiency and effectiveness of the Probe and Prototype's attitude determination and control system. The diameters and angular accelerations were then graphed based on the theoretical time needed for the Prototype and Probe to stabilize. This allowed the Probe team to finalize the reaction wheel diameter to be 42.5 mm and the material to be Steel.

To provide a robust control system for the Prototype, establishing seamless communication between the Arduino UNO and the Adafruit BNO055 Inertial Measurement Unit (IMU) is essential for accurate operation. The Probe team developed software protocols and interfaces to facilitate real-time data exchange between the Arduino and IMU that reads raw quaternion data. Additionally, a visualization of the Probe rotation was created utilizing the data stream to demonstrate the process of how the Arduino reads the raw quaternion data. This would exhibit the ability to rotate 360° in 3D space, fully utilizing quaternion kinematics.

6.2 Asteroid Perturbations

Using four different estimation models for the 2011UW158 asteroid, the gravitation potential-based perturbations were modeled using MATLAB, shown in Code 11. Two types of asteroid spectral types were specifically studied: two S-type asteroid models, where there are indications of a siliceous composition due to the observations of high-density characteristics, and two E-type asteroid models, where there are indications of an enstatite achondrite composition due to high sunlight reflection observations [27].

The mass from the effective volume (kg) and the surface gravity from the effective radius (m/s2) were used to simulate an oscillating lever arm on the cartesian axes to model the effects of gravitational oscillations on the rotation of the Probe, visualized through quaternion kinematics. This was modeled entirely within the proposed Hill radius for direct orbits around the asteroid of 20.9 km to 28.3 km, with the range of orbital radii being 1.5 km to 8 km. Additionally, the model includes orbital radii below the proposed sphere of influence of 2.69 km to 3.8 km [26]. These models do not consider solar radiation pressure as the effects of those perturbations would be minimal due to the small area-to-mass ratio of the Probe. They do not take into account the asteroid's irregular shape due to the inability to accurately model a proper estimation. The resulting Quaternion oscillations are shown in Figures 20-23; the S-type model demonstrates a lower oscillatory effect in relation to the distance (m) of the orbit radius of the Probe, with the 1750 m range providing a volatile region for the q2 quaternion component in particular. However, the 1600 m range shows low oscillatory effects for a relatively stable orbit to minimize the usage of constant angular momentum counteraction provided via the reaction wheels. The E-type model demonstrates extremely high oscillations between lower altitudes of the orbit around the asteroid. However, there are more small distance-based zones for relative stability that would minimize the use of reaction wheels.

These models were used to produce an oscillation function applied to the normal unit quaternion range from 5% to 20% to analyze the capabilities of the inverse tetrahedral reaction wheel configuration and robustness of the Super-Twisting Sliding Mode control algorithm. The models were propagated over the length of one hour, which was estimated to be above the amount of time that the Probe would stay in a parking orbit around the asteroid. This provided a significant amount of insight into the realistic capabilities of the overall system and demonstrated a possible scenario that the Probe could undergo during the mission.

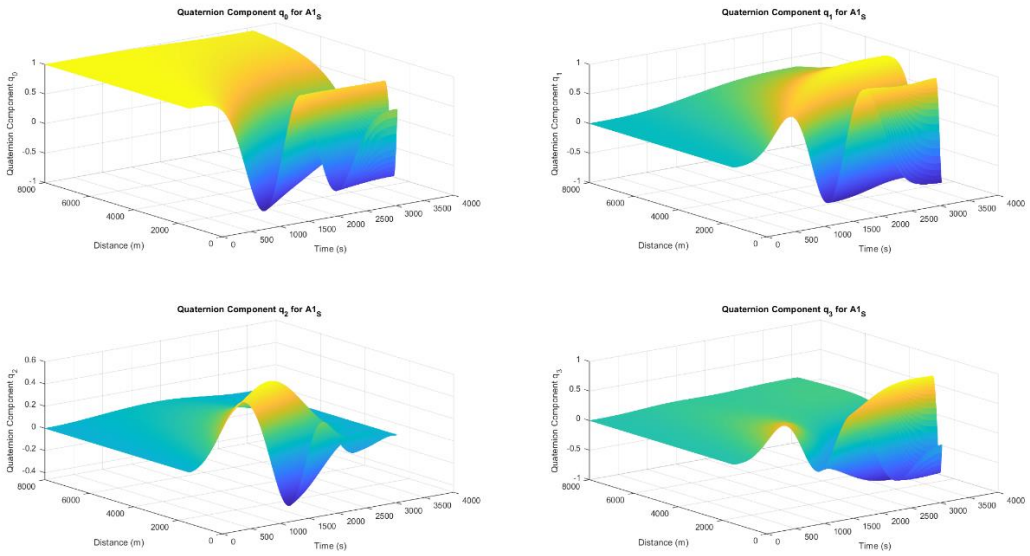


Figure 20: Effects of Gravitational Perturbations of Asteroid Model A1S

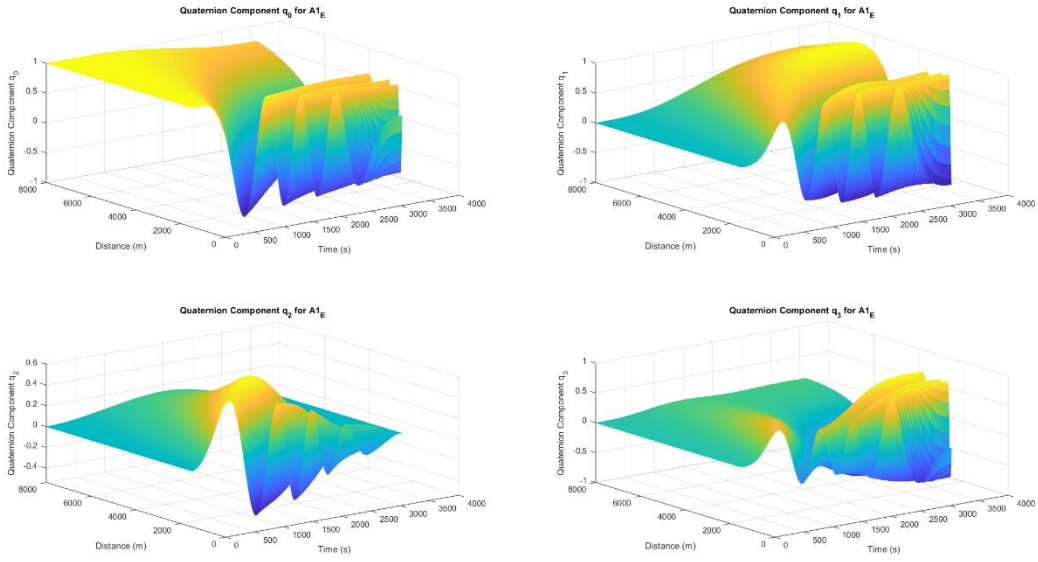


Figure 21: Effects of Gravitational Perturbations of Asteroid Model $A1_E$

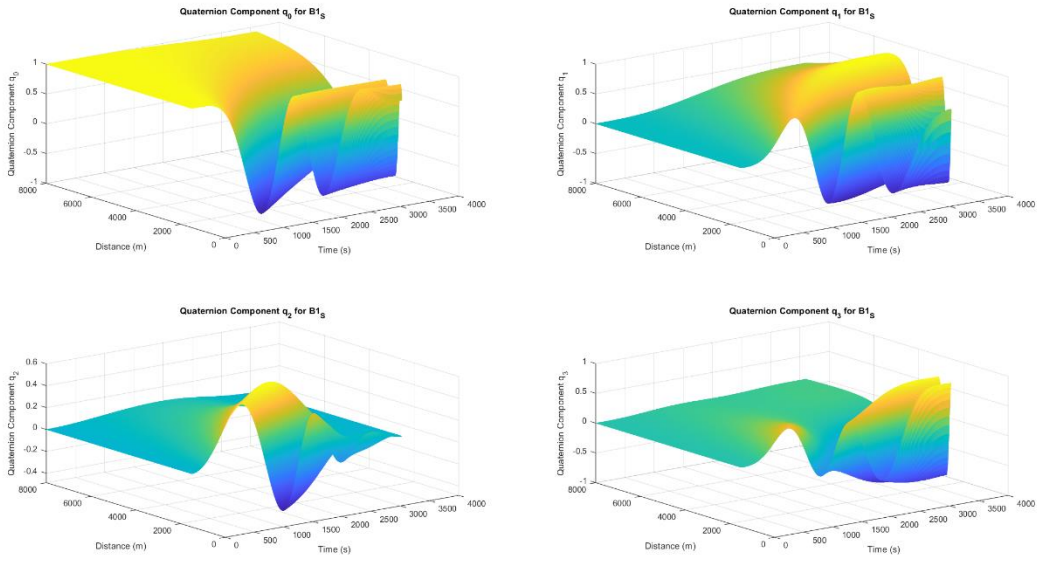


Figure 22: Effects of Gravitational Perturbations of Asteroid Model $B1_S$

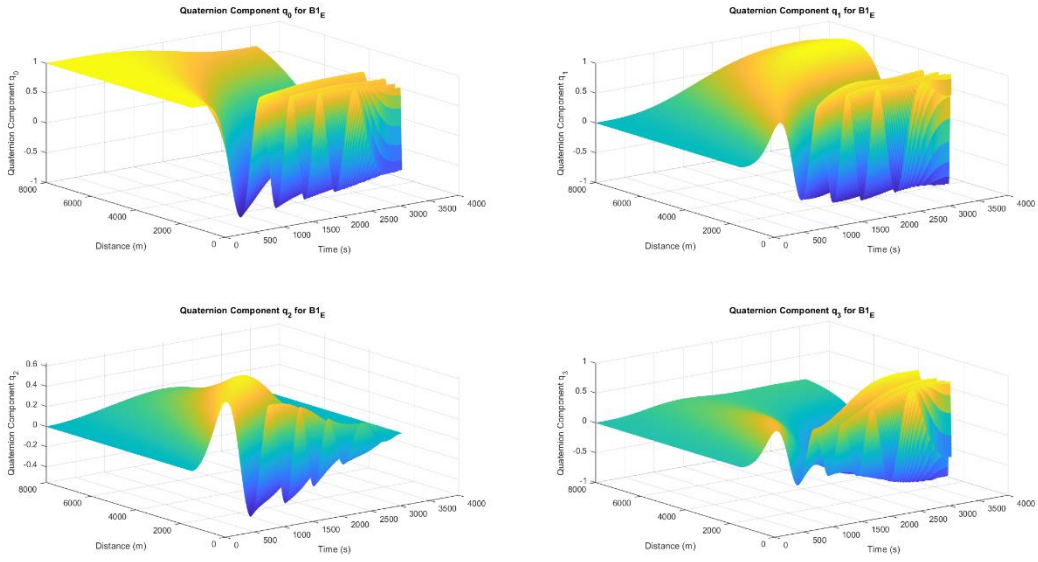


Figure 23: Effects of Gravitational Perturbations of Asteroid Model B1$_E$

A close approach maneuver was simulated using MATLAB, Code 10, to produce the gravitational perturbations when the sample collection process is in operation, where the distance is approximately 0.8 m from the surface of the asteroid. This was done to model the highest possible oscillatory effects that the Probe will undergo, besides any collision or moment torque risks due to the sample collection itself. The results over a time of one hour were collected, parsed, and applied to a normal unit quaternion vector with a weight of approximately 5%.

6.3 Super-Twisting Sliding Mode Control Algorithm

In Figure 24 below, the Quaternion section of the Super-Twisting Sliding Mode control algorithm is visualized and built using MATLAB Simulink. Using the oscillation .mat file, these were compared to a unit quaternion [1, 0, 0, 0] where the Probe would be facing its normal position relative to the initial database of the star tracker. The error of the quaternion vectors is calculated and fed through the quaternion Sliding Manifold, which uses estimated weights of the current quaternion error and the derivative of the error to construct a pathing model for the rest of the controller to follow and be applied to. The control algorithm is now split into two components that operate in tandem with each other: the Continuous Control function and the Super-Twisting algorithm. The Continuous Control function utilizes the integral of the Sliding Manifold to estimate the future quaternion error difference over a continuous portion of time, which allows the system to provide an approximate value for stabilization correction at that current point in time. The sliding surface, $s(x) = 0$, is defined based on the system's desired dynamics. The objective is to drive the system states to reach and sustain on this surface. However, this inherently causes the previously mentioned "chattering" effect, so the Super-Twisting algorithm is applied. The Super-Twisting algorithm utilizes a lambda function to weigh the signum, using Equation 28, that

provides the switching of negative or positive gains, ensuring finite time convergence to the sliding surface and an epsilon function which activates once the surface is reached, counters the effects of perturbations and uncertainties to maintain the trajectory on the surface that dampens the entire system. The two functions are then weighted based on the necessary requirements of the system and combined to provide robust control during the estimated duration of the mission for close approach sample collection.

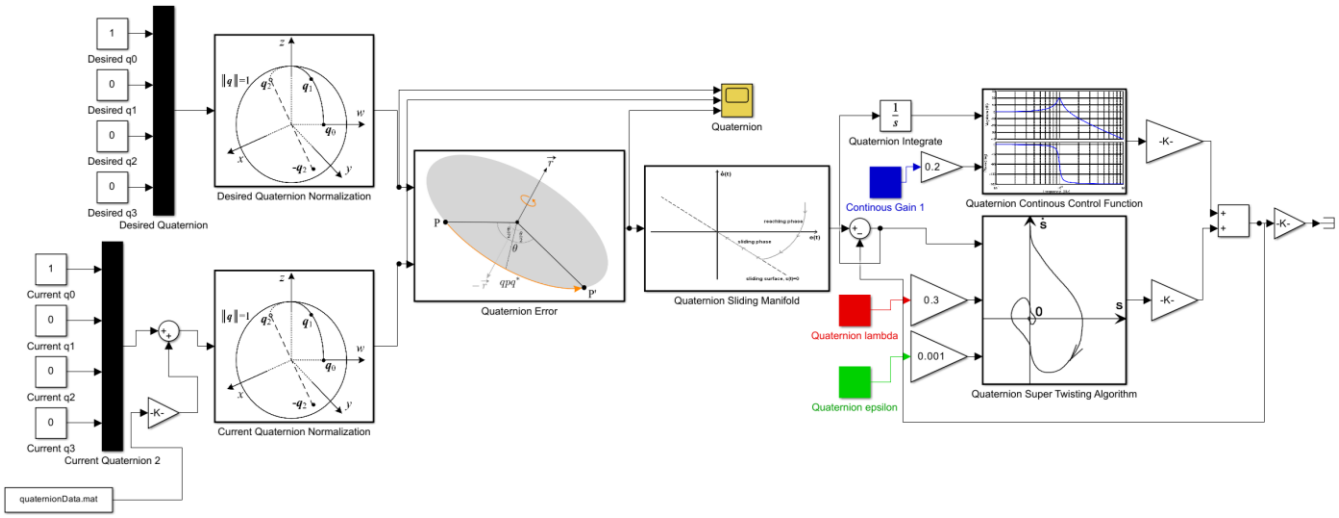


Figure 24: Quaternion Section of the Super-Twisting Sliding Mode Control

In Figure 25 below, the Angular Velocity section of the Super-Twisting Sliding Mode control algorithm is continued from the Quaternion section, where there are conversions applied to both the desired quaternion to the desired angular velocity signal and the current quaternion to the current angular velocity signal. The structure of the Super-Twisting Sliding Mode described previously remains the same. Using the angular velocity output from the control system, the torque applied to the Probe can be calculated using the Probe inertia matrix modeled after a hollow rectangular prism of 300 x 200 x 100 mm. The resulting torque 3x1 matrix can then be transformed inversely into the torques for each individual reaction wheel based on the inverse tetrahedral geometric configuration, using Equation 27.

The system is modeled with an algebraic loop using the summation function to continuously drive the error to approximately the unit quaternion or the ‘zero’. This is done due to the lack of a real hardware-based system and to examine the time to stabilize. Modeling an impulsive force, using a step signal to simulate an impact, the system will reach complete stabilization in approximately 2.41 seconds, proving the significant robustness of the system. In Figure 26 below, the reaction wheel configuration is further expanded with the torque of each reaction wheel relative to the fixed frame of the wheel itself being modeled. This allows the torque values to be converted to voltage values using parameters set by the AAC Clyde RW400 reaction wheels.

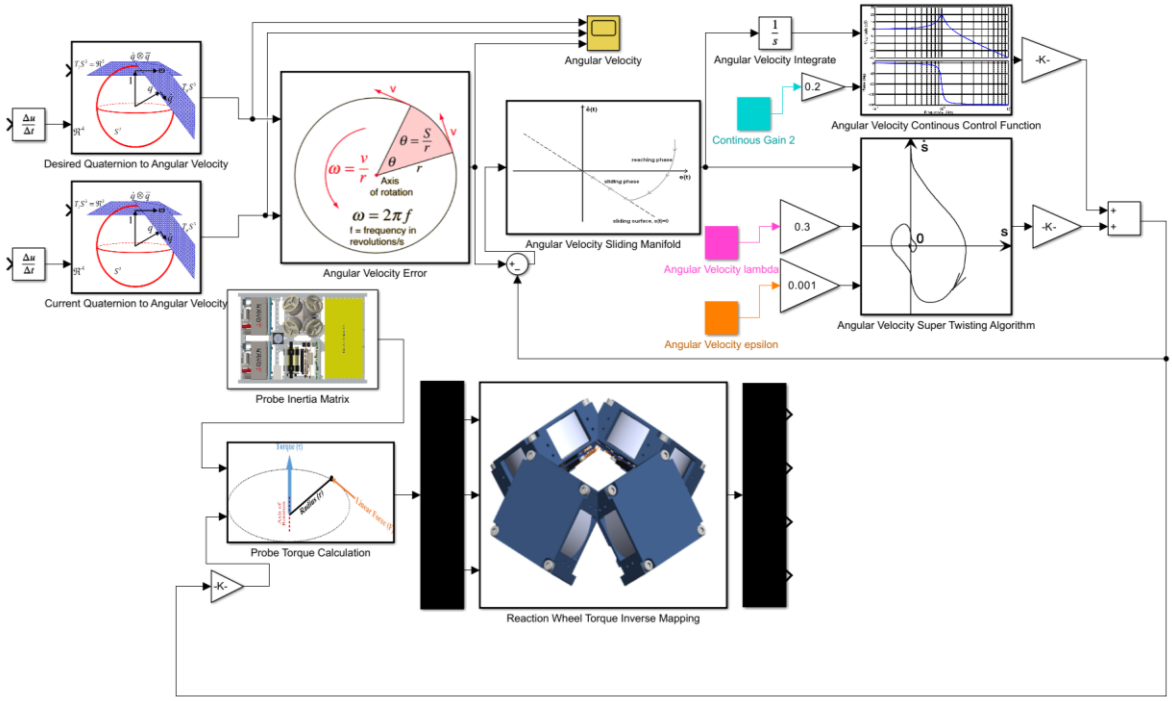


Figure 25: Angular Velocity Section of the Super-Twisting Sliding Mode Control

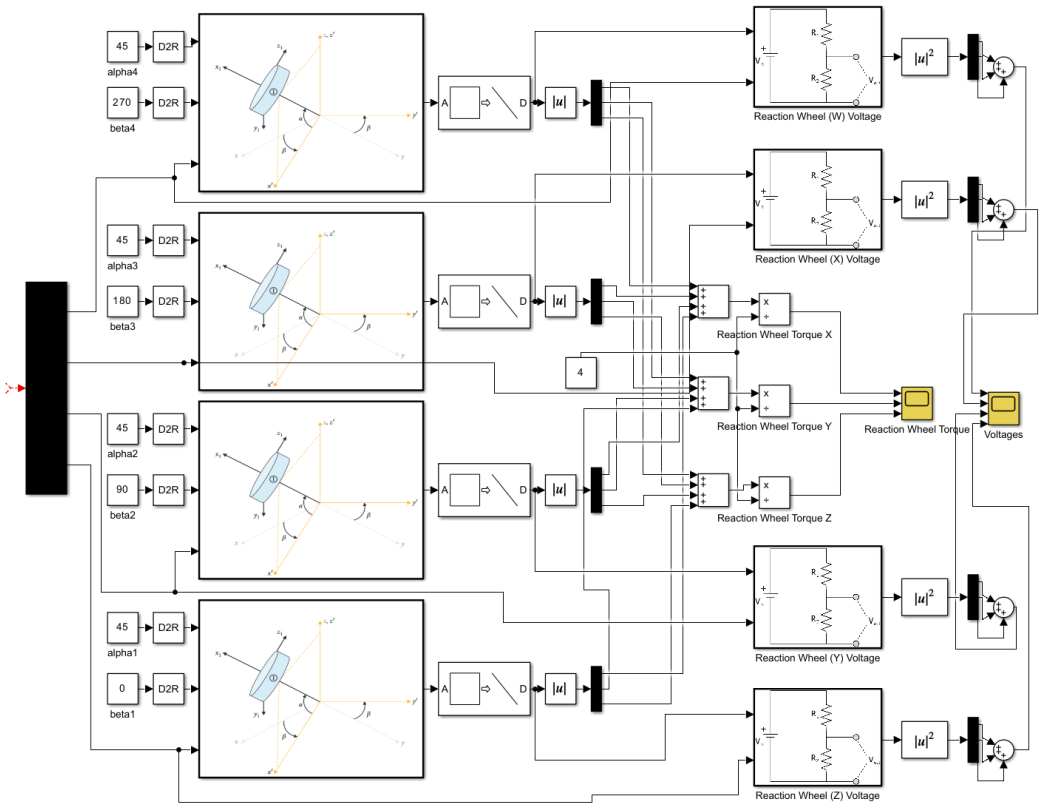


Figure 26: Geometric Configuration of the Inverse Tetrahedral Reaction Wheel Set

The oscillations can be seen in Figure 27, where small oscillations due to the gravitational potential field of the asteroid are applied. The hyper-imaginary vector components, q_1, q_2, and q_3 (colloquially known as q_x, q_y, and q_z) are shown to oscillate -0.2 and 0.2, based on the Eulerian angle direct cosine matrix parameterizations that are converted to a 4D matrix component to change the rotation of the cartesian axis using the resulting real component, q_0 (colloquially known as q_w). Since the normal vector component of the desired quaternion is [0, 0, 0], the error is the same as the current quaternion propagated over a period of one hour.

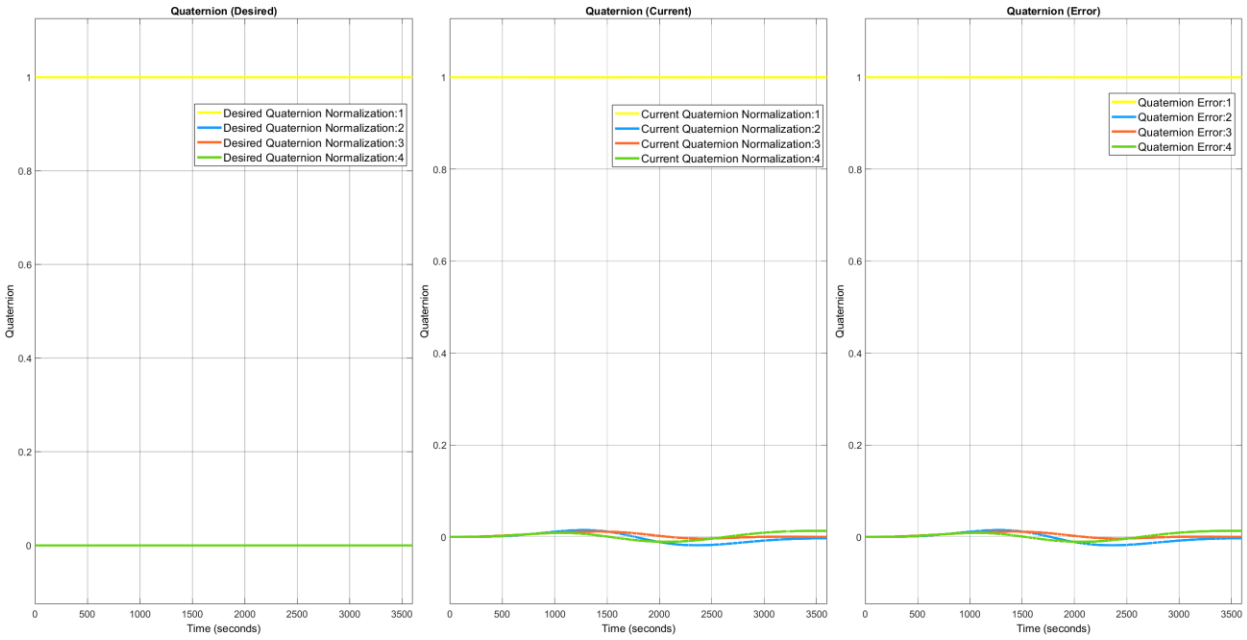


Figure 27: Probe Quaternion Characteristics due to Asteroid Approach

Since the desired state of the system is non-moving, the desired angular velocity results in 0 for the x, y, and z-axis, which is consistent with the conversion from the desired quaternion state to the desired angular velocity, shown in Figure 28. Since the error is compared to the zeroth line of the desired state, the current angular velocity and error are flipped representations of one another. As expected, the angular velocity differentials are relatively small, ranging from approximately $-6 \cdot 10^{-5}$ rpm to $10.3 \cdot 10^{-5}$ rpm, with an oscillatory pattern that matches the perturbation due to the asteroid combined with the orbital motion of the Probe itself. As with the quaternion results, this was propagated over a period of one hour.

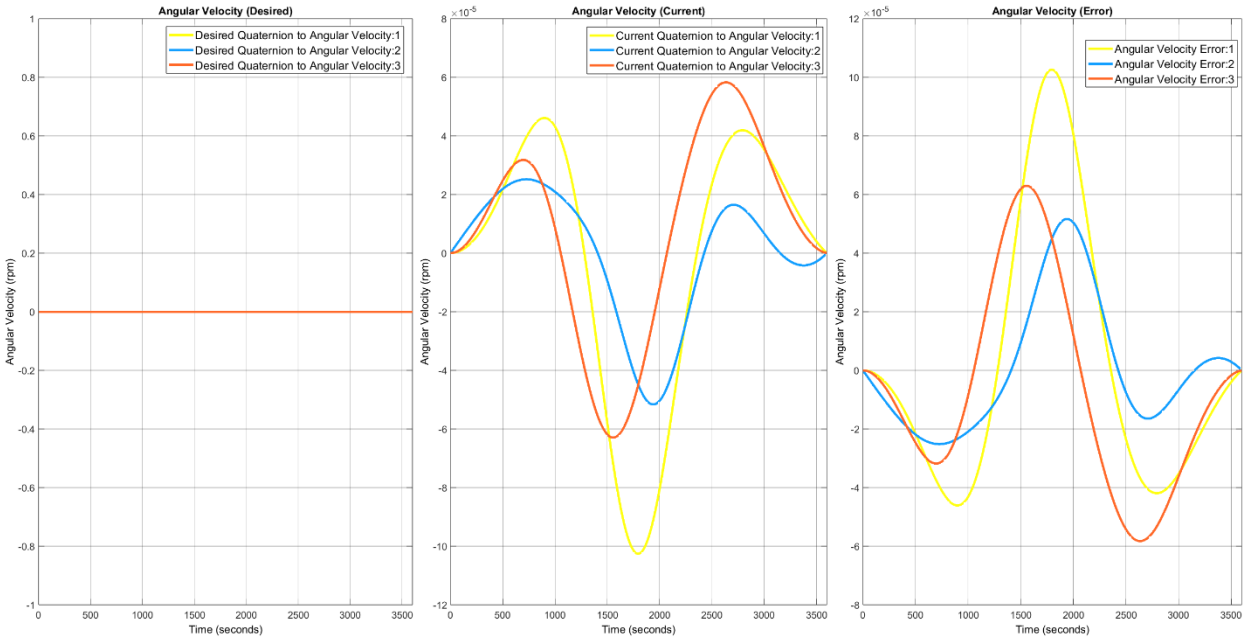


Figure 28: Probe Angular Velocity Characteristics due to Asteroid Approach

Efficient torque management is crucial because reaction wheels have limited operational ranges and can saturate if overloaded. The Super-Twisting Sliding Mode control algorithm efficiently manages the torque applied to the reaction wheels by ensuring that the control actions are precisely adjusted to the real-time requirements of the spacecraft's attitude control. As shown in Figure 29, the torques of the entire reaction wheel set are transformed into Cartesian x, y, and z axes, which demonstrate the system's response to asteroid perturbations. With a slight lag time of approximately 100 milliseconds relative to accurate rotational motion, the torque values provide smooth hills of increasing and decreasing intensity respective to when the slopes of the angular velocity of the probe increase or decrease, with the peaks marking those transitions. Additionally, the torque values have a rapid switch time matching the angular velocity signal's peaks or troughs, providing significant potential for fast-paced orientation control of the Probe.

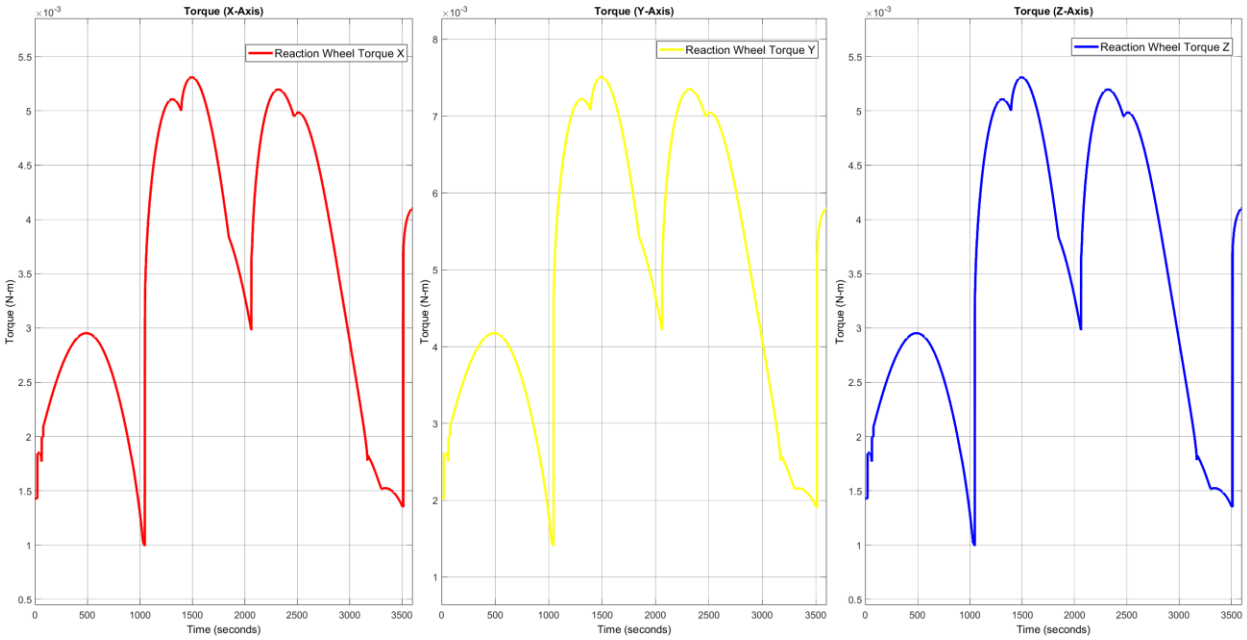


Figure 29: Reaction Wheel Torque Required due to Asteroid Approach

The Super-Twisting Sliding Mode control algorithm's smoother control actions benefit mechanical components and enhance power efficiency. By avoiding the frequent and harsh switching of control inputs standard in traditional sliding mode controls, the Super-Twisting Sliding Mode control algorithm reduces the energy consumption of the entire system. This is particularly important for Probe, where energy resources are limited and must be managed. Traditional PID, LQR, and Sliding Mode controllers can cause significant power fluctuations due to aggressive switching actions. These fluctuations can strain the Probe's power system and lead to inefficient power usage that can cause the solar panel/battery system to fail. As shown in Figure 30, the voltage values are designed to provide efficient gradients specific to the system's state. These are constrained by the AAC Clyde RW400 reaction wheel voltage logic system, with a region of 0V to 5.3V producing logic 0s and 1s that control the positive and negative directions. Additionally, the same constraints create optimal slopes for speed variance necessary to counteract the asteroid's gravitational potential perturbations, leading to a unique voltage-switching system. The continuous nature of the Super-Twisting Sliding Mode control law minimizes these fluctuations, leading to more stable power consumption patterns.

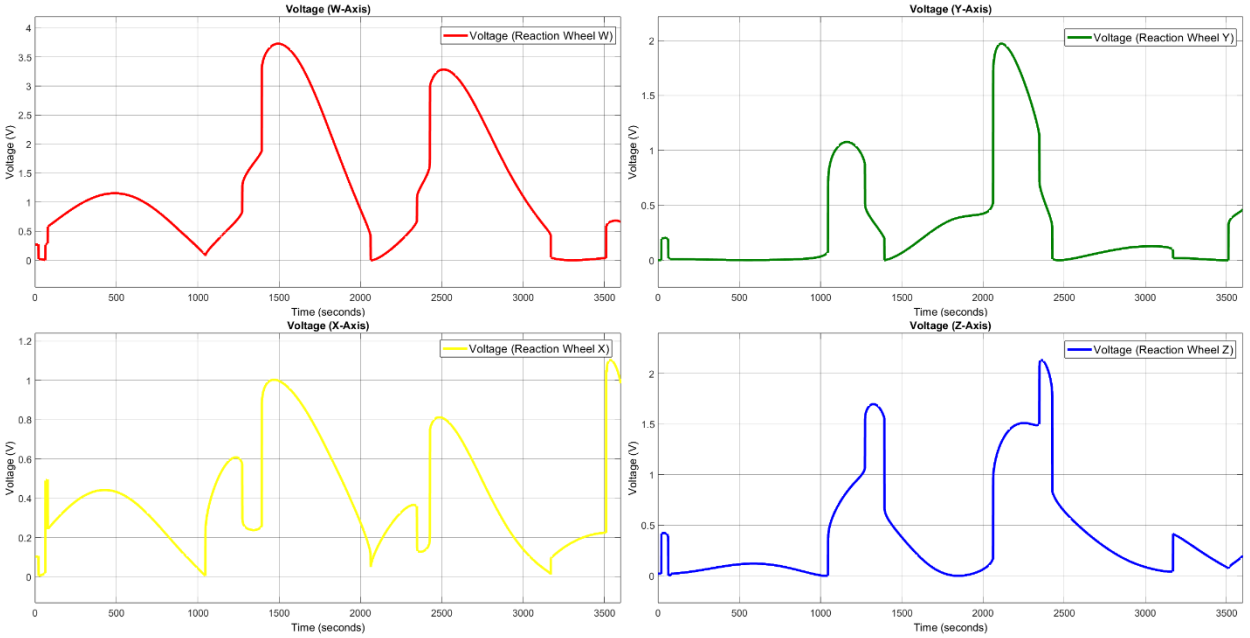


Figure 30: Reaction Wheel Voltage Required due to Asteroid Approach

By minimizing the high-frequency switching actions typical of conventional sliding mode controls, the Super-Twisting Sliding Mode control algorithm reduces the mechanical stress on reaction wheels. The Super-Twisting component of the control algorithm adjusts control actions more continuously, which minimizes wear on the mechanical components over long periods, thereby potentially extending their operational lifespan. The robustness of the Super-Twisting Sliding Mode control algorithm control to disturbances and its ability to maintain the desired control objective on the sliding surface without deviating due to disturbances leads to more precise attitude adjustments. This precision is crucial for missions requiring high attitude accuracy, which is required for the CAMP mission.

7. Conclusions

In summary, the probe team validated the existing design of the probes for the CAMP mission, making corrections as needed, removing redundant components from the lander probe, including the IR spectrometer, Lidar Sensor, and visible light camera, expanding the space claim for the sample collector from 1U to 2U, removed the previously selected AAC Clyde Space ADCS system and replaced with an independent AAC Clyde Space ST-200 Star Tracker, four RW-400 reaction wheels in an inverse tetrahedral configuration, and implemented a super-twisting-sliding mode control algorithm.

A prototype probe and test stand were built to the same size and 20% the weight of the true Probe to demonstrate the control algorithm's performance when implemented with reaction wheels in an inverse tetrahedral configuration. By substituting Maxon motors with integral hall-effect sensors, the prototype and test stand will be ready for testing in their current configuration.

The Super-Twisting Sliding Mode control algorithm offers substantial advantages for managing reaction wheel torque and minimizing voltage usage in CubeSat attitude determination and control systems. Its ability to provide smooth, continuous control actions enhances spacecraft components' operational efficiency, reliability, and lifespan while also ensuring precise control and energy efficiency. By reducing the "chattering" effects and thus the associated voltage spikes, the Super-Twisting Sliding Mode control algorithm is proven, experimentally through realistic asteroid gravitational potential perturbations of four models, to maintain the overall stability of the voltage and torque supplied to the Probe's attitude determination and control system. Combined with the inverse tetrahedral configuration of the reaction wheels and the utilization of quaternion kinematics, a highly efficient system has been achieved and can be implemented on various orbiters, landers, and spacecraft.

This navigation system, based on reaction wheels and thrusters, is a robust and versatile solution that can be implemented on all of the probes in the CAMP mission, and testing shows that it is capable of the orbital maneuvers necessary to orbit the asteroids that are subject to study in this mission, facilitate sample collection through close approach towards the asteroid, and return the lander probe to the spacecraft with the collected sample. The reaction wheel design and control algorithm are an innovative solution in the area of CubeSat control and have the potential to advance mankind's capabilities for deep-space exploration using low-cost CubeSat probes.

8. References

[1] A. Barcio, A. Bicak, C. Nimmo, R. Lane, B. Leppke, and C. Watson, “Constellation Asteroid Measurement Project (CAMP) Spacecraft Team,” May 2023.

[2] I. Stewart, S. Brownlee, C. Lawrence, A. Henderson, and M. Currie, “Constellation Asteroid Measurement Project Probe Report,” May 2023.

[3] James Richard Wertz and Computer Sciences Corporation. Attitude Systems Operation, Spacecraft Attitude Determination and Control. Dordrecht: Kluwer Academic Publishers, 1978.

[4] C. Cappelletti, S. Battistini, and B. Malphrus, CubeSat Handbook. Academic Press, 2020.

[5] State-of-the-Art Small Spacecraft Technology. 2021.

[6] S. J. Kapurch, NASA Systems Engineering Handbook. DIANE Publishing, 2010.

[7] “RW400 - High Performance CubeSat Reaction Wheels | AAC Clyde Space,” www.aac-clyde.space. https://www.aac-clyde.space/what-we-do/space-products components/adcs/rw400

[8] J. Sol`a, "Quaternion kinematics for the error-state Kalman filter," Cornell University, 2017.

[9] C.-C. P. Yang-Rui Li, "Super-Twisting Sliding Mode Control Law Design for Attitude Tracking Task of a Spacecraft via Reaction Wheels," Ministry of Science and Technology, Tainan, 2020.

[10] A. Barrau and S. Bonnabel, “The Invariant Extended Kalman Filter as a Stable Observer,” *IEEE Transactions on Automatic Control*, 62(4):1797–1812, 2017.

[11] M. Brossard, A. Barrau, and S. Bonnabel, “A Code for Unscented Kalman Filtering on Manifolds (UKF-M).” 2019.

[12] G Huang, A Mourikis, and S Roumeliotis, “A Quadratic-Complexity Observability-Constrained Unscented Kalman Filter for SLAM,” *IEEE Transactions on Robotics*, 29(5):1226–1243, 2013.

[13] E. Wan and R. van der Merwe, “The Unscented Kalman Filter for Nonlinear Estimation,” Oregon Graduate Institute of Science & Technology, 2000.

[14] E. Kraft, “A Quaternion-based Unscented Kalman Filter for Orientation Tracking,” Physikalisches Institut, University of Bonn, Germany, 2003.

[15] Bosch Sensortec, BNO055 Intelligent 9-axis absolute orientation sensor data sheet, 2014.

[16] Maxon Motors, EC 20 flat: Ø20 mm, brushless, 3-watt data sheet, 2023.

[17] Ardunio, Arduino UNO R3 Product Reference Manual, 2024.

[18] P. J. Besl and N. D. McKay. A method for registration of 3-D shapes. IEEE Transactions on pattern analysis and machine intelligence, 14(2):239–256, 1992.

[19] O. D. Faugeras and M. Hebert. The representation, recognition, and locating of 3-D objects. International Journal of Robotics Research, 5(3):27–52, 1986.

[20] B. K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. Journal of Optical Society of America A, 4(4):629–642, 1987.

[21] T. Chanut, S. Aljbaae, and V. Carruba, Mascon gravitation model using a shaped polyhedral source, Monthly Notices of the Royal Astronomical Society, 2015.

[22] "Dawn Aerospace 0.8U CubeDrive," https://www.dawnaerospace.com/green-propulsion-cubedrives

[23] C. Matthews, C. Sartin, C. Poteet, M. Yohannes, "CAMP Probe Project Team Progress Report," November 2023.

[24] Maxon Motors, ESCON Module 24/2 Motor Controller Hardware Reference Manual, 2024.

[25] Curtis, H.D., "Orbital Mechanics for Engineering Students," Fourth Revised Edition, 2021

[26] Monteiro, F. et al., "Shape model and spin direction analysis of the PHA (436724) 2011UW158: a large superfast rotator," Monthly Notices of the Royal Astronomical Society 495, pp. 3990-4005, 2020

[27] Yokoyama, T.; Vieira-Neto, E.; Winter, O.C.; Sanchez, D.M.; and Brasil, P.I.O., "On the evection resonance and its connection to the stability of outer satellites," Mathematical Problems in Engineering 2008, Article 251978, pp. 16, 2008.

9. Appendix

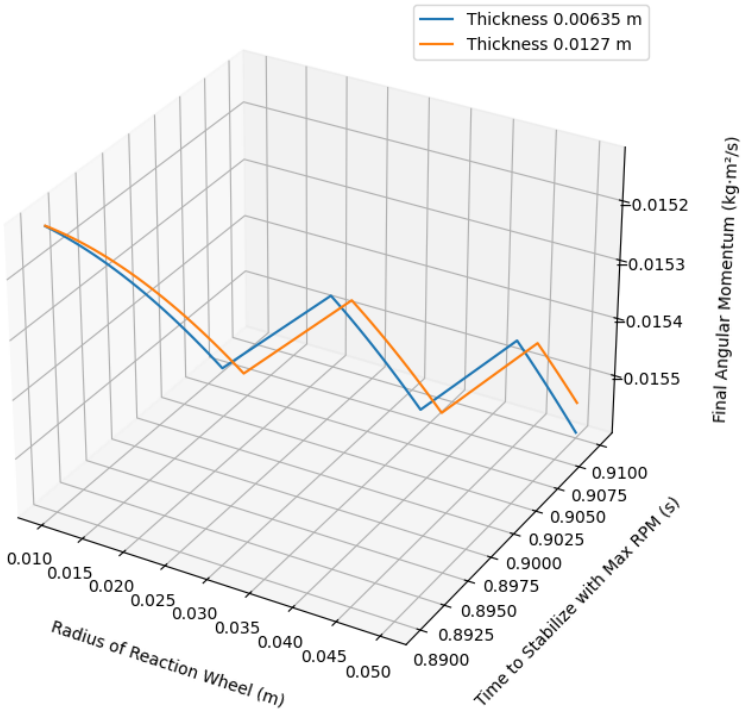


Figure 31: Reaction Wheel Stabilization Analysis Graph

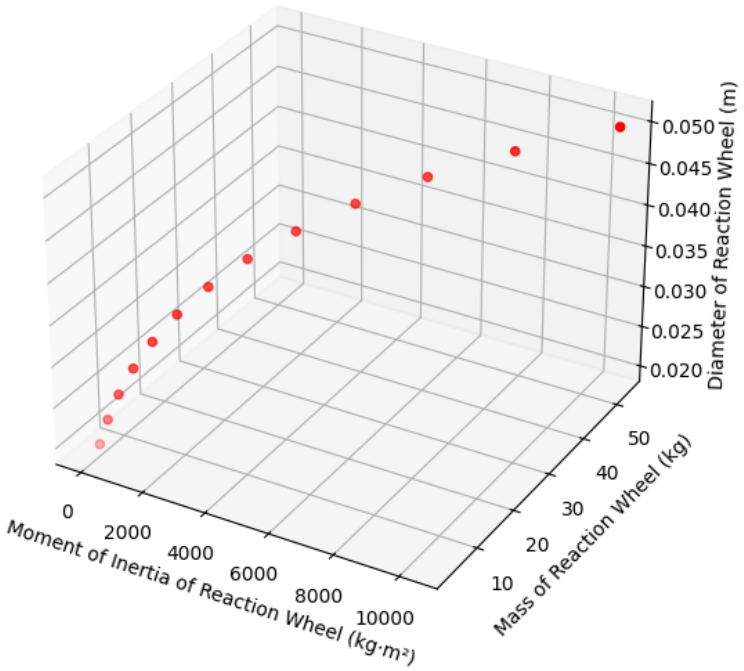


Figure 32: Reaction Wheel Inertial Analysis Graph

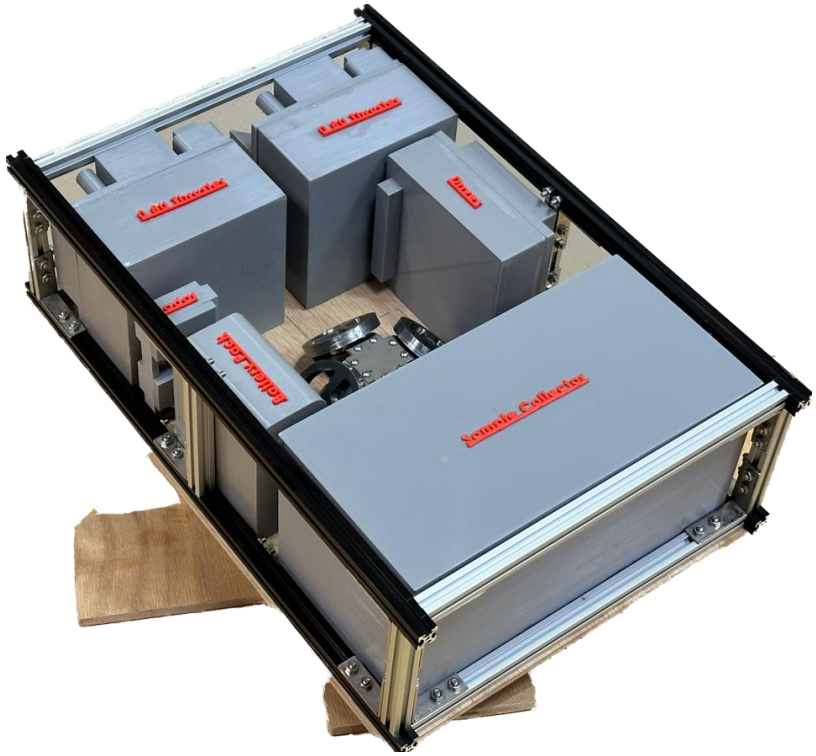


Figure 33: Attitude Control Test Stand

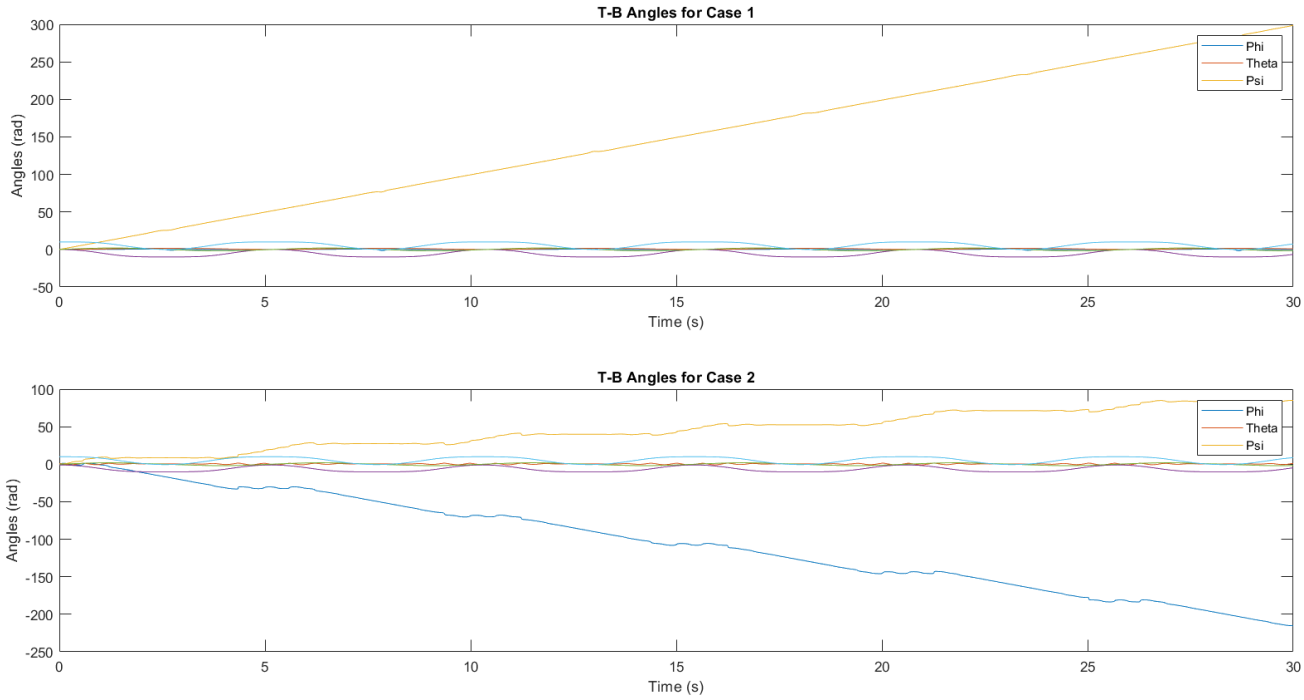


Figure 34: Evolution of Probe Eulerian Angles due to Constant Applied Force

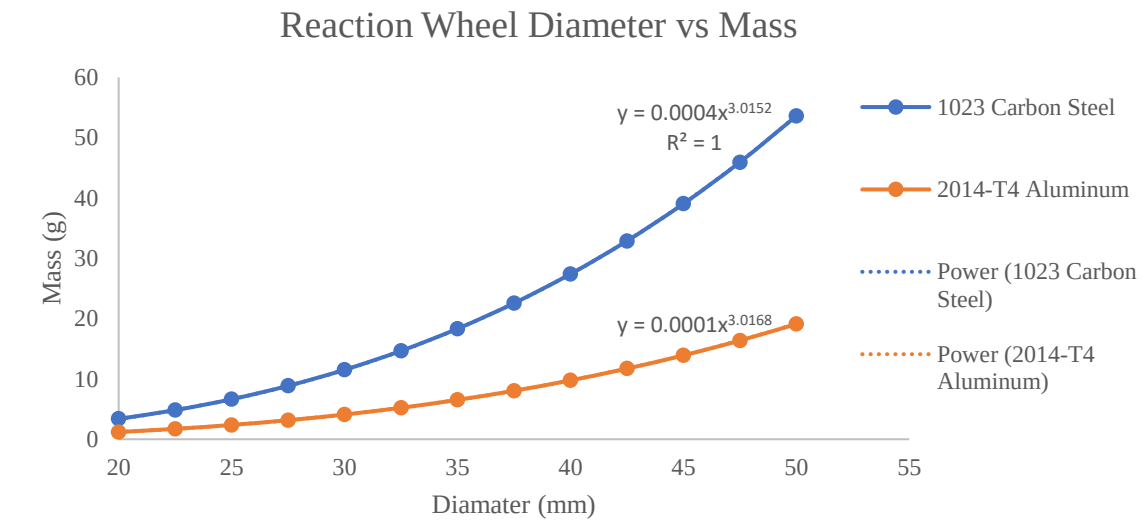


Figure 35: Reaction Wheel Diameter in comparison to Reaction Wheel Mass

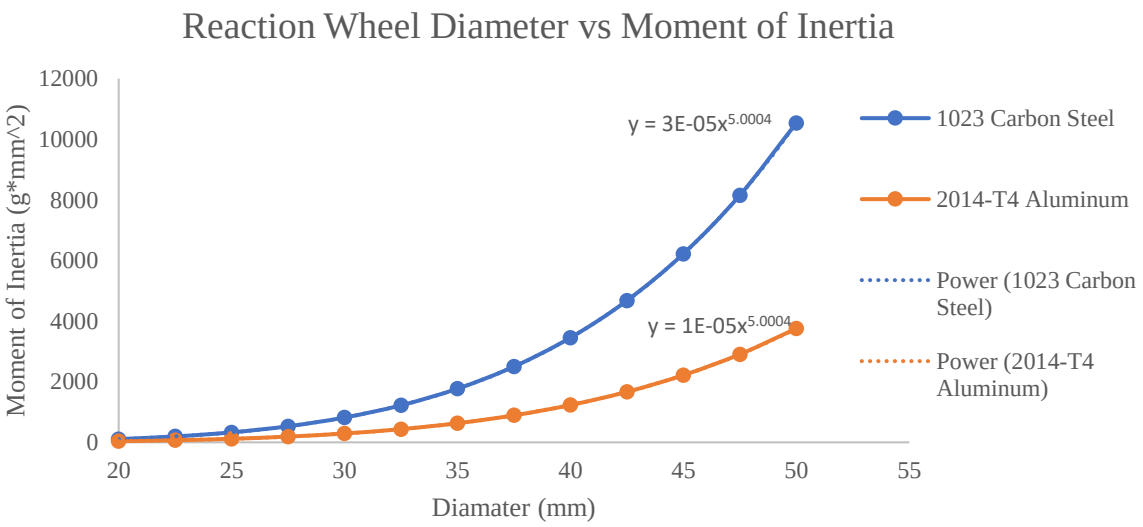


Figure 36: Reaction Wheel Diameter in comparison to Reaction Wheel Inertia

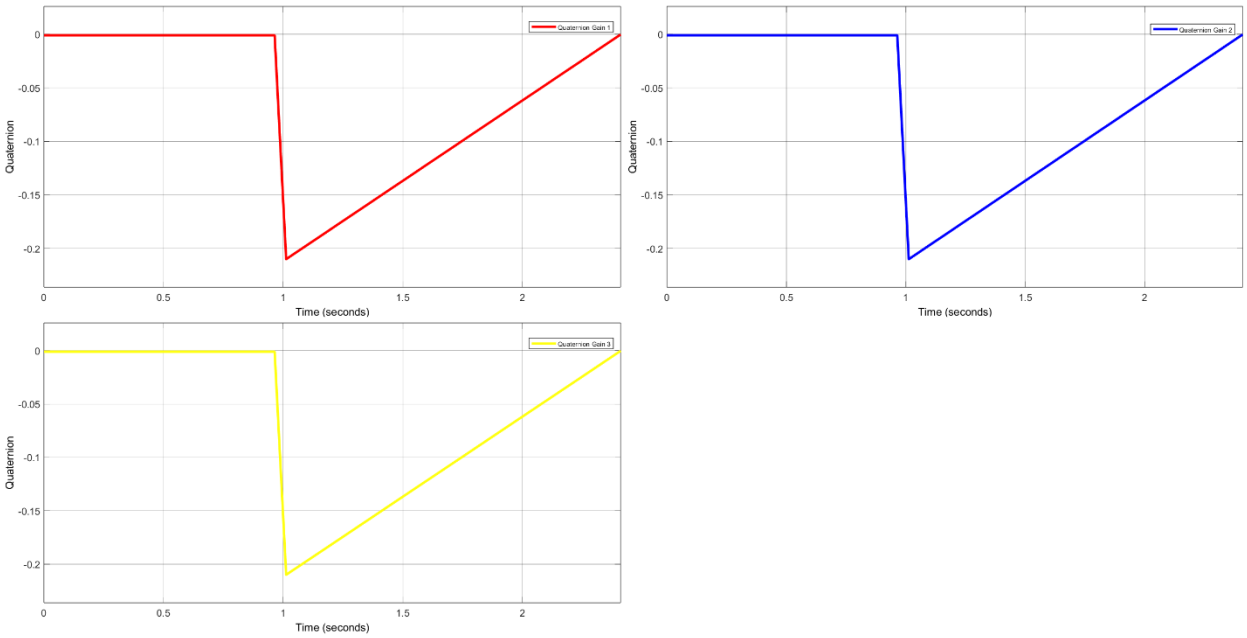


Figure 37: Probe Quaternion Stabilization due to Impact

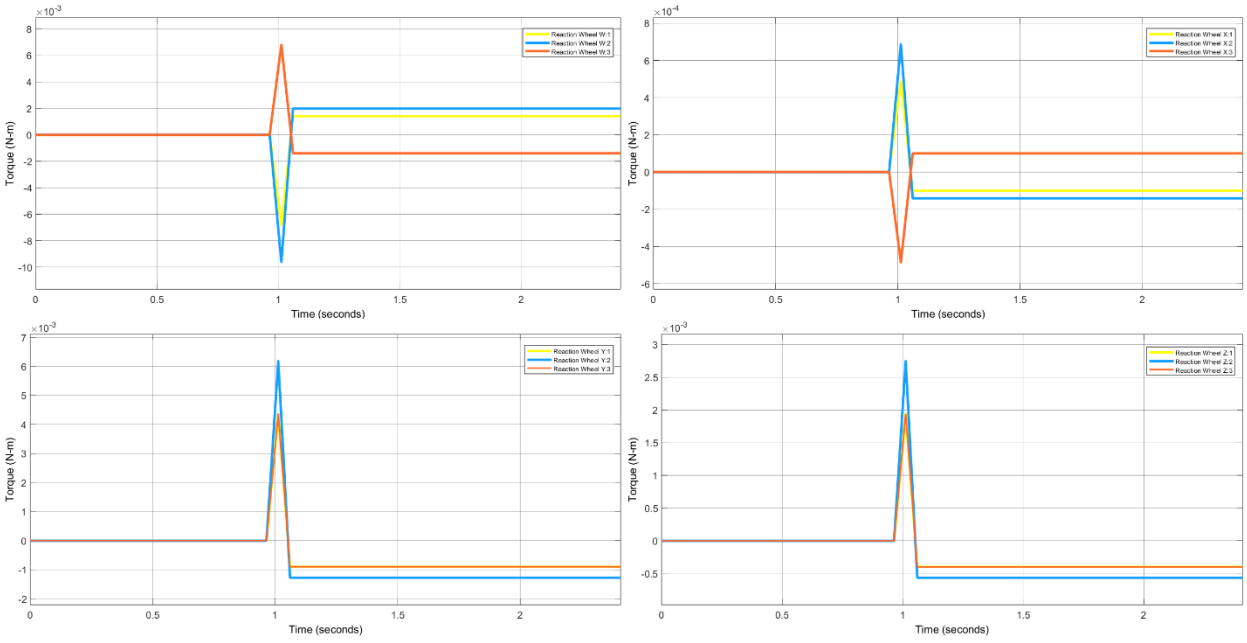


Figure 38: Reaction Wheel Torque Stabilization due to Impact

10. Sample Calculations

$$\frac{SRP}{d^2} = 604 \tag{1}$$

The equation above is the inverse square law. It is used to know how much the total solar radiation is at any given distance from the sun. SRP is the solar radiation constant. d is the distance from the sun in AU.

$$\frac{SRP}{d^2} \cdot A \cdot PR \cdot R = 48.96 \tag{2}$$

The equation above is used to calculate the amount of energy absorbed by the solar panels. A is the area of the solar panels, PR is the performance ratio of the solar panels, the default value for this is 0.75, R is the solar panel specific efficiency, SRP is the solar radiation constant, and d is the distance from the sun in AU.

$$\omega = v \cdot r \tag{3}$$

The equation above is for the calculation of the reaction wheel angular velocity. r is the radius of the individual reaction wheel in millimeters and v is the linear velocity in kilometers per second.

$$L = I \cdot \omega \tag{4}$$

The equation above is for the calculation of the reaction wheel angular momentum. I is the inertia of the reaction wheel, utilizing a simple cylinder in this case, and ω is the angular velocity.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}. \tag{5}$$

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} \cos(-\alpha) & 0 & -\sin(-\alpha) \\ 0 & 1 & 0 \\ \sin(-\alpha) & 0 & \cos(-\alpha) \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \tag{6}$$

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} \cos(-\alpha) & 0 & -\sin(-\alpha) \\ 0 & 1 & 0 \\ \sin(-\alpha) & 0 & \cos(-\alpha) \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos(-\alpha) & 0 & -\sin(-\alpha) \\ 0 & 1 & 0 \\ \sin(-\alpha) & 0 & \cos(-\alpha) \end{bmatrix} \begin{bmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{7}$$

$$\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} = \mathbf{R}_i \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \tag{8}$$

Equations 5-8 are rotation matrixes about the x axis and the negative y axis. Where x, y, and z are the original coordinate system, x´, y´, and z´ is the coordinate system after rotation about the z axis and x´´, y´´, and z´´ is the rotation about the z axis then the negative y axis.

$$\dot{q}_0 = -\frac{1}{2}\mathbf{q}^T\boldsymbol{\omega} \tag{9}$$

$$\dot{\mathbf{q}} = \frac{1}{2}(q_0\mathbf{I}_3 + \mathbf{q}^{\times})\boldsymbol{\omega} \tag{10}$$

Equations 9-10 are used to avoid singularity problems where q and q_0 will be the unit quaternion, I will be the identity matrix, and ω is the angular velocity. They can also be simplified in matrix from Equations 11 – 13.

$$\dot{\mathbf{Q}} = \frac{1}{2}\mathbf{E}(\mathbf{Q})\boldsymbol{\omega}, \tag{11}$$

$$\mathbf{E}(\mathbf{Q}) = \begin{bmatrix} -\mathbf{q}^T \\ q_0\mathbf{I}_3 + \mathbf{q}^{\times} \end{bmatrix} = \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{bmatrix} \tag{12}$$

$$\boldsymbol{\omega}_{w,i} = \mathbf{R}_i\boldsymbol{\omega} + \Omega_i\mathbf{e}_{x_i}, \tag{13}$$

Where Ω is the relative rotation speed of each reaction wheel and e_x is a unit vector.

$$\mathbf{H} = \mathbf{J}\boldsymbol{\omega} = \begin{bmatrix} J_x & -J_{xy} & -J_{xz} \\ -J_{xy} & J_y & -J_{yz} \\ -J_{xz} & -J_{yz} & J_z \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{14}$$

Where H is the angular momentum of spacecraft relative to the body frame and J is the moment of inertia matrix given in Equation 15.

$$\mathbf{J}_{w,i} = \begin{bmatrix} J_{m,i} & 0 & 0 \\ 0 & J_i' & 0 \\ 0 & 0 & J_i' \end{bmatrix}. \tag{15}$$

Where J_m is the axial moment of inertia matrix.

$$\begin{aligned} \mathbf{H}_{w,i} &= \mathbf{J}_{w,i}\boldsymbol{\omega}_{w,i} \\ &= \mathbf{J}_{w,i}\left(\mathbf{R}_i\boldsymbol{\omega} + \Omega_i\mathbf{e}_{x_i}\right) \\ &= \mathbf{J}_{w,i}\mathbf{R}_i\boldsymbol{\omega} + \mathbf{J}_w\Omega_i\mathbf{e}_{x_i} \\ &= \mathbf{J}_{w,i}\mathbf{R}_i\boldsymbol{\omega} + J_{mi}\Omega_i\mathbf{e}_{x_i} \end{aligned} \tag{16}$$

Where $H_{w,i}$ is the angular momentum of each reaction wheel. This is then mapped onto the spacecraft body from yielding Equation 17.

$$\begin{aligned} \mathbf{H}_{w,i}^{B} &= \mathbf{R}_i^T\boldsymbol{\omega}_{w,i} \\ &= \mathbf{R}_i^T\left(\mathbf{J}_{w,i}\mathbf{R}_i\boldsymbol{\omega} + J_{mi}\Omega_i\mathbf{e}_{x_i}\right) \\ &= \mathbf{R}_i^T\mathbf{J}_{w,i}\mathbf{R}_i\boldsymbol{\omega} + \mathbf{R}_i^T J_{mi}\Omega_i\mathbf{e}_{x_i} \end{aligned} \tag{17}$$

This can then be combined with the angular momentum of the spacecraft to obtain Equation 18.

$$\begin{aligned} \mathbf{H}_T &= \mathbf{H} + \sum_{i=1}^{4}\mathbf{H}_{w,i}^{B} \\ &= \mathbf{J}\boldsymbol{\omega} + \sum_{i=1}^{4}\left(\mathbf{R}_i^T\mathbf{J}_{w,i}\mathbf{R}_i\boldsymbol{\omega} + \mathbf{R}_i^T J_{mi}\Omega_i\mathbf{e}_{x_i}\right) \\ &= \left(\mathbf{J} + \sum_{i=1}^{4}\mathbf{R}_i^T\mathbf{J}_{w,i}\mathbf{R}_i\right)\boldsymbol{\omega} + \sum_{i=1}^{4}\mathbf{R}_i^T J_{mi}\Omega_i\mathbf{e}_{x_i} \end{aligned} \tag{18}$$

This can be simplified using Equation 19 which is the Force Distribution matrix to obtain Equation 20.

$$\Gamma = \begin{bmatrix} \cos\alpha\cos\beta & -\cos\alpha\sin\beta & -\cos\alpha\cos\beta & \cos\alpha\sin\beta \\ \cos\alpha\sin\beta & \cos\alpha\cos\beta & -\cos\alpha\sin\beta & -\cos\alpha\cos\beta \\ \sin\alpha & \sin\alpha & \sin\alpha & \sin\alpha \end{bmatrix} \tag{19}$$

$$\mathbf{H}_T = \mathbf{J}_{eq}\boldsymbol{\omega} + \Gamma\mathbf{J}_m\boldsymbol{\Omega} \tag{20}$$

$$\mathbf{M}_G = \left(\frac{\mathrm{d}\mathbf{H}_T}{\mathrm{d}t}\right)_B + \boldsymbol{\omega}^{\times}\mathbf{H}_T \tag{21}$$

Where M_G is the Euler equation of motion. This is als0 equal to the sum of the external control torques (τ_a) and the external disturbance torques (d(t)).

$$\begin{aligned}\boldsymbol{\tau}_c &= \Gamma\boldsymbol{\tau}_w \triangleq \begin{bmatrix} \tau_{cx} & \tau_{cy} & \tau_{cz} \end{bmatrix}^T, \\ \boldsymbol{\tau}_w &= -\mathbf{J}_m\dot{\boldsymbol{\Omega}} = -\begin{bmatrix} J_{m1}\dot{\Omega}_1 & J_{m2}\dot{\Omega}_2 & J_{m3}\dot{\Omega}_3 & J_{m4}\dot{\Omega}_4 \end{bmatrix}^T \\ &\triangleq \begin{bmatrix} \tau_{w1} & \tau_{w2} & \tau_{w3} & \tau_{w4} \end{bmatrix}^T.\end{aligned} \tag{22}$$

Where τ_c is the control torque and τ_w is the reaction torque. So 21can be rewritten as Equation 23.

$$\mathbf{J}_{eq}\dot{\boldsymbol{\omega}} = \boldsymbol{\tau}_a + \boldsymbol{\tau}_c + \mathbf{d} - \boldsymbol{\omega}^{\times}\left(\mathbf{J}_{eq}\boldsymbol{\omega} + \Gamma\mathbf{J}_m\boldsymbol{\Omega}\right) \tag{23}$$

Where τ_a is zero.

$$\begin{bmatrix} \tau_{cx} \\ \tau_{cy} \\ \tau_{cz} \end{bmatrix} = \begin{bmatrix} \cos\alpha\cos\beta & -\cos\alpha\sin\beta & -\cos\alpha\cos\beta & \cos\alpha\sin\beta \\ \cos\alpha\sin\beta & \cos\alpha\cos\beta & -\cos\alpha\sin\beta & -\cos\alpha\cos\beta \\ \sin\alpha & \sin\alpha & \sin\alpha & \sin\alpha \end{bmatrix} \begin{bmatrix} \tau_{w1} \\ \tau_{w2} \\ \tau_{w3} \\ \tau_{w4} \end{bmatrix}. \tag{23}$$

Where $\beta = \pi/4$ and $\sin\alpha = (3)^{1/2}/3$. From this and Equation 19 the following FDM.

$$\Gamma = \frac{\sqrt{3}}{3}\begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{24}$$

$$\min_{\tau_w} \mathcal{T} = \sum_{i=1}^{4} \tau_{wi}^2, \tag{25}$$

This is a static optimization problem.

$$
\begin{aligned}
\left.\frac{\partial \mathscr{L}}{\partial \tau_{w1}}\right|_{\tau^*,\lambda^*} &= 2\tau_{w1}^* + \frac{\sqrt{3}}{3}\lambda_1^* + \frac{\sqrt{3}}{3}\lambda_2^* + \frac{\sqrt{3}}{3}\lambda_3^* = 0 \\
\left.\frac{\partial \mathscr{L}}{\partial \tau_{w2}}\right|_{\tau^*,\lambda^*} &= 2\tau_{w2}^* - \frac{\sqrt{3}}{3}\lambda_1^* + \frac{\sqrt{3}}{3}\lambda_2^* + \frac{\sqrt{3}}{3}\lambda_3^* = 0 \\
\left.\frac{\partial \mathscr{L}}{\partial \tau_{w3}}\right|_{\tau^*,\lambda^*} &= 2\tau_{w3}^* - \frac{\sqrt{3}}{3}\lambda_1^* - \frac{\sqrt{3}}{3}\lambda_2^* + \frac{\sqrt{3}}{3}\lambda_3^* = 0 \\
\left.\frac{\partial \mathscr{L}}{\partial \tau_{w4}}\right|_{\tau^*,\lambda^*} &= 2\tau_{w4}^* + \frac{\sqrt{3}}{3}\lambda_1^* - \frac{\sqrt{3}}{3}\lambda_2^* + \frac{\sqrt{3}}{3}\lambda_3^* = 0 \\
\left.\frac{\partial \mathscr{L}}{\partial \lambda_1}\right|_{\tau^*,\lambda^*} &= \frac{\sqrt{3}}{3}\left(\tau_{w1}^* - \tau_{w2}^* - \tau_{w3}^* + \tau_{w4}^*\right) - \tau_{cx} = 0, \\
\left.\frac{\partial \mathscr{L}}{\partial \lambda_2}\right|_{\tau^*,\lambda^*} &= \frac{\sqrt{3}}{3}\left(\tau_{w1}^* + \tau_{w2}^* - \tau_{w3}^* - \tau_{w4}^*\right) - \tau_{cy} = 0 \\
\left.\frac{\partial \mathscr{L}}{\partial \lambda_3}\right|_{\tau^*,\lambda^*} &= \frac{\sqrt{3}}{3}\left(\tau_{w1}^* + \tau_{w2}^* + \tau_{w3}^* + \tau_{w4}^*\right) - \tau_{cz} = 0
\end{aligned}
\tag{26}
$$

Where τ_w^* and λ^* be to optimal solution for the first order condition to minimize Equation 25. These can then be rewritten as the following distribution matrix.

$$
\begin{bmatrix} \tau_{w1}^* \\ \tau_{w2}^* \\ \tau_{w3}^* \\ \tau_{w4}^* \end{bmatrix} = \frac{1}{4}\begin{bmatrix} \sqrt{3} & \sqrt{3} & \sqrt{3} & 1 \\ -\sqrt{3} & \sqrt{3} & \sqrt{3} & -1 \\ -\sqrt{3} & -\sqrt{3} & \sqrt{3} & 1 \\ \sqrt{3} & -\sqrt{3} & \sqrt{3} & -1 \end{bmatrix}\begin{bmatrix} \tau_{cx} \\ \tau_{cy} \\ \tau_{cz} \\ 0 \end{bmatrix} \tag{27}
$$

$$
\dot{s} = -k_1|s|^{1/2}\operatorname{sign}(s) - k_2\int_0^t \operatorname{sign}(s)\mathrm{d}\tau + \mathrm{d}(t) \tag{28}
$$

Where s is the closed loop sliding dynamics and k_1 and k_2 are gains. It can be expressed in the following state-space form.

$$
\begin{aligned}
z_1 &= s(x,t). \\
z_2 &= -k_2\int_0^t \operatorname{sign}(s)\mathrm{d}\tau + \mathrm{d}(t)
\end{aligned}
\tag{29}
$$

The time derivative of Equations 29 is shown below.

$$\dot{z}_1 = -k_1|z_1|^{1/2}\text{sign}(z_1) + z_2$$
$$\dot{z}_2 = -k_2\text{sign}(z_1) + \rho, \tag{30}$$

Since the above is nonlinear the following transform equation must be done.

$$\zeta_1 = |z_1|^{1/2}\text{sign}(z_1)$$
$$\zeta_2 = z_2. \tag{31}$$

The time derivative of this yields Equation 32.

$$\dot{\zeta}_1 = \frac{\dot{z}_1|z_1|^{1/2} - z_1 \cdot (1/2)|z_1|^{-1/2} \cdot \text{sign}(z_1) \cdot \dot{z}_1}{|z_1|} = \frac{\left[-k_1|z_1|^{1/2}\text{sign}(z_1) + z_2\right] \cdot \left(|z_1|^{1/2} - (1/2)|z_1|^{1/2}\right)}{|z_1|}$$

$$= \frac{1}{|z_1|^{1/2}}\left(-\frac{k_1}{2}|z_1|^{1/2}\text{sign}(z_1) + \frac{1}{2}z_2\right) = \frac{1}{|\zeta_1|}\left(-\frac{k_1}{2}\zeta_1 + \frac{1}{2}\zeta_2\right),$$

$$\dot{\zeta}_2 = -k_2\text{sign}(z_1) + \rho = \frac{1}{|z_1|^{1/2}}\left(-k_2|z_1|^{1/2}\text{sign}(z_1) + |z_1|^{1/2}\rho\right) = \frac{1}{|\zeta_1|}\left(-k_2\zeta_1 + |\zeta_1|\rho\right), \tag{32}$$

These can then be rewritten in the matrix from Equation 33.

$$\dot{\boldsymbol{\zeta}} = \frac{1}{|\zeta_1|}(\mathbf{A}\boldsymbol{\zeta} + \mathbf{B}\tilde{\rho})$$

$$\mathbf{A} = \begin{bmatrix} -0.5k_1 & 0.5 \\ -k_2 & 0 \end{bmatrix},$$

$$\mathbf{B} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \tag{33}$$

Equation 34 and 35 is the confinement of the gains.

$$k_2 > \delta,$$
$$k_1^2 > 4k_2 \tag{34}$$

$$k_1^2\left(\frac{1}{2}k_2 - \frac{1}{16}k_1^2\right) < \delta^2, \quad 4k_2 > k_1^2. \tag{35}$$

Equations 5-35 are for the Super Twisting Sliding Mode Controller

Equation 36 is the states for the Unscented Kalman Filter

$$\begin{gathered} Initialization: \\ x(k+1) = f(x(k), u(k)) + v_k, \\ x \in \mathbb{R}^n, u \in \mathbb{R}^m, z \in \mathbb{R}^z. \\ \Rightarrow Defined\ system \\ y(k) = h(x(k), u(k)) + n_k \end{gathered} \tag{36}$$

11. Code

Code 1: Initialization of IMU Quaternion Raw Data Output

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <math.h>
#include <SimpleKalamFilter.h>
#include "IRremote.hpp"

#define BNO055_SAMPLERATE_DELAY_MS (100)
#define IR_RECEIVE_PIN 5
#define RED_LED_PIN 6 // Define red LED pin
#define GREEN_LED_PIN 7 // Define green LED pin
#define MOTOR_PWM_PIN_1 13 // Define the motor PWM control pin
#define MOTOR_PWM_PIN_2 12 // Define the motor PWM control pin

Adafruit_BNO055 myIMU = Adafruit_BNO055();

// Create a Kalman filter instance for each quaternion component
// Adjusted parameters for tuning

// Measurement Noise, Estimation Error, Process Noise
SimpleKalmanFilter kalmanW(1.0, 1.0, 0.03);
SimpleKalmanFilter kalmanX(1.0, 1.0, 0.03);
SimpleKalmanFilter kalmanY(1.0, 1.0, 0.03);
SimpleKalmanFilter kalmanZ(1.0, 1.0, 0.03);
```

```
void setup() {
  // Initialize serial communication
  Serial.begin(115200);

  // Initialize the LED pins as outputs
  pinMode(RED_LED_PIN, OUTPUT);
  pinMode(GREEN_LED_PIN, OUTPUT);
  pinMode(MOTOR_PWM_PIN_1, OUTPUT);
  pinMode(MOTOR_PWM_PIN_2, OUTPUT);

  // Attempt to start the IMU
  if(!myIMU.begin()) {
    Serial.println("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
    while(1) {
      digitalWrite(RED_LED_PIN, HIGH); // Keep the red LED on if there is an error
    }
  }
  delay(1000);
  // Other setup code
  int8_t temp = myIMU.getTemp();
  myIMU.setExtCrystalUse(true);
}

void loop() {
  // IMU code
  static uint8_t lastSystem = 0, lastGyro = 0, lastAccel = 0, lastMg = 0;
```

```
  uint8_t system, gyro, accel, mg = 0;
  myIMU.getCalibration(&system, &gyro, &accel, &mg);

  // Check if any calibration value has changed
  if(system != lastSystem || gyro != lastGyro || accel != lastAccel
|| mg != lastMg) {
    digitalWrite(GREEN_LED_PIN, HIGH); // Turn the green LED on
    delay(5000);                       // Green LED on for a short
time
    digitalWrite(GREEN_LED_PIN, LOW);  // Turn the green LED off

    // Update last calibration values
    lastSystem = system;
    lastGyro = gyro;
    lastAccel = accel;
    lastMg = mg;
  }

  imu::Quaternion quat = myIMU.getQuat();

  // Apply the Unscented Kalman Filter to each quaternion component
  float filteredW = kalmanW.updateEstimate(quat.w());
  float filteredX = kalmanX.updateEstimate(quat.x());
  float filteredY = kalmanY.updateEstimate(quat.y());
  float filteredZ = kalmanZ.updateEstimate(quat.z());

  // Print IMU data to serial
  Serial.print(filteredW);
```

```
  Serial.print(",");
  Serial.print(filteredX);
  Serial.print(",");
  Serial.print(filteredY);
  Serial.print(",");
  Serial.print(filteredZ);
  Serial.print(",");
  Serial.print(accel);
  Serial.print(",");
  Serial.print(gyro);
  Serial.print(",");
  Serial.print(mg);
  Serial.print(",");
  Serial.println(system);

  // Delay before next loop iteration
  delay(BNO055_SAMPLERATE_DELAY_MS);
}
```

Code 2: Configuration of the Unscented Kalman Filter Matrix Rotations

```
#ifndef KONFIG_H
#define KONFIG_H

#include <stdlib.h>
#include <stdint.h>
#include <math.h>

/* State Space dimension */
#define SS_X_LEN    (2)
#define SS_Z_LEN    (1)
#define SS_U_LEN    (1)
#define SS_DT_MILIS (10)                                /* 10 ms */
#define SS_DT        float_prec(SS_DT_MILIS/1000.)    /* Sampling
time */

/* Change this size based on the biggest matrix you will use */
#define MATRIX_MAXIMUM_SIZE     (5)

/* Define this to enable matrix bound checking */
#define MATRIX_USE_BOUNDS_CHECKING

/* Set this define to choose math precision of the system */
#define PRECISION_SINGLE    1
#define PRECISION_DOUBLE    2
#define FPU_PRECISION       (PRECISION_SINGLE)

#if (FPU_PRECISION == PRECISION_SINGLE)
```

```
    #define float_prec          float
    #define float_prec_ZERO     (1e-7)
    #define float_prec_ZERO_ECO (1e-5)      /* 'Economical' zero*/
#elif (FPU_PRECISION == PRECISION_DOUBLE)
    #define float_prec          double
    #define float_prec_ZERO     (1e-13)
    #define float_prec_ZERO_ECO (1e-8)      /* 'Economical' zero*/
#else
    #error("FPU_PRECISION has not been defined!");
#endif

/* Set this define to choose system implementation */
#define SYSTEM_IMPLEMENTATION_PC                    1
#define SYSTEM_IMPLEMENTATION_EMBEDDED_CUSTOM       2
#define SYSTEM_IMPLEMENTATION_EMBEDDED_ARDUINO      3
#define                                         SYSTEM_IMPLEMENTATION
(SYSTEM_IMPLEMENTATION_EMBEDDED_ARDUINO)
void SPEW_THE_ERROR(char const * str);
#define ASSERT(truth, str) { if (!(truth)) SPEW_THE_ERROR(str); }

#endif // KONFIG_H
```

Code 3: Matricies for Unscented Kalman Filter

```
#ifndef MATRIX_H
#define MATRIX_H

#include "konfig.h"

#if (SYSTEM_IMPLEMENTATION == SYSTEM_IMPLEMENTATION_PC)
    #include <iostream>
    #include <iomanip>      // std::setprecision

    using namespace std;
#elif                   (SYSTEM_IMPLEMENTATION                    ==
SYSTEM_IMPLEMENTATION_EMBEDDED_ARDUINO)
    #include <Wire.h>
#endif

class Matrix
{
public:
    typedef enum {
        InitMatWithZero,    /* Initialize matrix with zero data */
        NoInitMatZero
    } InitZero;

    /* ------------------------------------------- Basic Matrix
Class functions ------------------------------------------- */
    /* Init empty matrix size _i16row x _i16col */
    Matrix(const int16_t _i16row, const int16_t _i16col, const
InitZero _init = InitMatWithZero);
```

```
    /* Init matrix size _i16row x _i16col with entries initData */
    Matrix(const int16_t _i16row, const int16_t _i16col, const
float_prec* initData, const InitZero _init = InitMatWithZero);
    /* Copy constructor (for this operation --> A(B)) (copy B into
A) */
    Matrix(const Matrix& old_obj);
    /* Assignment operator (for this operation --> A = B) (copy B
into A) */
    Matrix& operator = (const Matrix& obj);
    /* Destructor */
    ~Matrix(void);
    /* Get internal state */
    inline int16_t i16getRow(void) const { return this->i16row; }
    inline int16_t i16getCol(void) const { return this->i16col; }


    /* ------------------------------------------ Matrix entry
accessing functions ------------------------------------------
*/
    float_prec& operator () (const int16_t _row, const int16_t
_col);
    float_prec operator () (const int16_t _row, const int16_t
_col) const;
    class Proxy {
        public:
            Proxy(float_prec* _inpArr, const int16_t _maxCol) {
_array.ptr = _inpArr; this->_maxCol = _maxCol; }
            Proxy(const float_prec* _inpArr, const int16_t
_maxCol) { _array.cptr = _inpArr; this->_maxCol = _maxCol; }
            float_prec & operator [] (const int16_t _col);
            float_prec operator [] (const int16_t _col) const;
```

```
        private:
            union {
                const float_prec* cptr;
                float_prec* ptr;
            } _array;
            int16_t _maxCol;
    };
    Proxy operator [] (const int16_t _row);
    const Proxy operator [] (const int16_t _row) const;



    /* ---------------------------------------- Matrix checking
function declaration ---------------------------------------- */
    bool bMatrixIsValid(void);
    void vSetMatrixInvalid(void);
    bool bMatrixIsSquare();
    /*    ------------------------------------------    Matrix
elementary operations ------------------------------------------
-- */
    bool operator == (const Matrix& _compare) const;
    bool operator != (const Matrix& _compare) const;
    Matrix operator - (void) const;
    Matrix operator + (const float_prec _scalar) const;
    Matrix operator - (const float_prec _scalar) const;
    Matrix operator * (const float_prec _scalar) const;
    Matrix operator / (const float_prec _scalar) const;
    Matrix operator + (const Matrix& _matAdd) const;
    Matrix operator - (const Matrix& _matSub) const;
    Matrix operator * (const Matrix& _matMul) const;
```

```
    /* Declared outside class below */

    /* inline Matrix operator + (const float_prec _scalar, Matrix
_mat); */

    /* inline Matrix operator - (const float_prec _scalar, Matrix
_mat); */

    /* inline Matrix operator * (const float_prec _scalar, Matrix
_mat); */

    /*   ---------------------------------------------   Simple
Matrix operations ----------------------------------------------
*/

    void vRoundingElementToZero(const int16_t _i, const int16_t
_j);

    Matrix RoundingMatrixToZero(void);

    void vSetHomogen(const float_prec _val);

    void vSetToZero(void);

    void vSetRandom(const int32_t _maxRand, const int32_t
_minRand);

    void vSetDiag(const float_prec _val);

    void vSetIdentity(void);

    Matrix Transpose(void);

    bool bNormVector(void);

    /*  ----------------------------------------  Matrix/Vector
insertion operations  ----------------------------------------
*/

    Matrix InsertVector(const Matrix& _Vector, const int16_t
_posCol);

    Matrix InsertSubMatrix(const Matrix& _subMatrix, const int16_t
_posRow, const int16_t _posCol);

    Matrix InsertSubMatrix(const Matrix& _subMatrix, const int16_t
_posRow, const int16_t _posCol,
                           const int16_t _lenRow, const int16_t
_lenColumn);
```

```
    Matrix InsertSubMatrix(const Matrix& _subMatrix, const int16_t
_posRow, const int16_t _posCol,
                           const   int16_t   _posRowSub,   const
int16_t _posColSub,
                           const int16_t _lenRow, const int16_t
_lenColumn);
    /*  ------------------------------------------------- Large
operations   -------------------------------------------------
*/
    /* Matrix invertion using Gauss-Jordan algorithm */
    Matrix Invers(void) const;
    /* Check the definiteness of a matrix */
    bool           bMatrixIsPositiveDefinite(const           bool
checkPosSemidefinite = false) const;
    /* Return the vector (Mx1 matrix) correspond with the diagonal
entries */
    Matrix GetDiagonalEntries(void) const;
    /*  Do  the  Cholesky  Decomposition  using  Cholesky-Crout
algorithm, return 'L' matrix */
    Matrix CholeskyDec(void) const;
    /*  Do  Householder  Transformation  for  QR  Decomposition
operation */
    Matrix  HouseholderTransformQR(const  int16_t  _rowTransform,
const int16_t _colTransform);
    /*  Do  QR  Decomposition  for  matrix  using  Householder
Transformation */
    bool QRDec(Matrix& Qt, Matrix& R) const;
    /* Do back-subtitution for upper triangular matrix A & column
matrix B:
     * x = BackSubtitution(&A, &B)          ; for Ax = B
     */
```

```
    Matrix BackSubtitution(const Matrix& A, const Matrix& B)
const;

    /* Do forward-subtitution for lower triangular matrix A &
column matrix B:

     * x = ForwardSubtitution(&A, &B)       ; for Ax = B

     */

    Matrix ForwardSubtitution(const Matrix& A, const Matrix& B)
const;

    /*   ---------------------------------------------   Matrix
printing function ----------------------------------------------
*/

    void vPrint(void);

    void vPrintFull(void);


private:

    /* Data structure of Matrix class:

     *  0 <= i16row <= MATRIX_MAXIMUM_SIZE       ; i16row is the
row of the matrix. i16row is invalid if (i16row == -1)

     *  0 <= i16col <= MATRIX_MAXIMUM_SIZE       ; i16col is the
column of the matrix. i16col is invalid if (i16col == -1)

     *

     * Accessing index start from 0 until i16row/i16col, that is:

     *  (0 <= idxRow < i16row)     and     (0 <= idxCol < i16col).

     * There are 3 ways to access the data:

     *  1. A[idxRow][idxCol]

     *  2. A(idxRow, idxCol)

     *  3. A._at(idxRow, idxCol)

     * floatData[MATRIX_MAXIMUM_SIZE][MATRIX_MAXIMUM_SIZE] is the
memory representation of the matrix. We only use the

     *  first i16row-th and first i16col-th memory for the matrix
data. The rest is unused. */
```

```
    int16_t i16row;
    int16_t i16col;
    float_prec
floatData[MATRIX_MAXIMUM_SIZE][MATRIX_MAXIMUM_SIZE];
    float_prec& _at(const int16_t _row, const int16_t _col) {
return this->floatData[_row][_col]; }
    float_prec _at(const int16_t _row, const int16_t _col) const
{ return this->floatData[_row][_col]; }
};


inline Matrix operator + (const float_prec _scalar, const Matrix&
_mat);
inline Matrix operator - (const float_prec _scalar, const Matrix&
_mat);
inline Matrix operator * (const float_prec _scalar, const Matrix&
_mat);
inline Matrix MatIdentity(const int16_t _i16size);


/*    ================================================    inline
definition                                                 below:
================================================ */
/*    ================================================    inline
definition                                                 below:
================================================ */


/* ---------------------------------------- Basic Matrix Class
functions ---------------------------------------- */
/* ---------------------------------------- Basic Matrix Class
functions ---------------------------------------- */


inline  Matrix::Matrix(const  int16_t  _i16row,  const  int16_t
_i16col, const InitZero _init) {
```

```
    this->i16row = _i16row;
    this->i16col = _i16col;

    if (_init == InitMatWithZero) {
        this->vSetHomogen(0.0);
    }
}

inline Matrix::Matrix(const int16_t _i16row, const int16_t
_i16col, const float_prec* initData, const InitZero _init) {
    this->i16row = _i16row;
    this->i16col = _i16col;

    if (_init == InitMatWithZero) {
        this->vSetHomogen(0.0);
    }
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            (*this)(_i,_j) = *initData;
            initData++;
        }
    }
}

inline Matrix::Matrix(const Matrix& old_obj) {
    this->i16row = old_obj.i16row;
    this->i16col = old_obj.i16col;
```

```
    const float_prec *sourc = old_obj.floatData[0];
    float_prec *desti = this->floatData[0];

    for (int16_t _i = 0; _i < i16row; _i++) {
        /* Still valid with invalid matrix ((i16row == -1) or
(i16col == -1)) */
        memcpy(desti,   sourc,   sizeof(float_prec)*size_t((this-
>i16col)));
        sourc += (MATRIX_MAXIMUM_SIZE);
        desti += (MATRIX_MAXIMUM_SIZE);
    }
}

inline Matrix& Matrix::operator = (const Matrix& obj) {
    this->i16row = obj.i16row;
    this->i16col = obj.i16col;

    const float_prec *sourc = obj.floatData[0];
    float_prec *desti = this->floatData[0];

    for (int16_t _i = 0; _i < i16row; _i++) {
        /* Still valid with invalid matrix ((i16row == -1) or
(i16col == -1)) */
        memcpy(desti,   sourc,   sizeof(float_prec)*size_t((this-
>i16col)));
        sourc += (MATRIX_MAXIMUM_SIZE);
        desti += (MATRIX_MAXIMUM_SIZE);
    }
```

```
    return *this;
}

/*    ----------------------------------------    Matrix    entry
accessing functions ---------------------------------------- */
/*    ----------------------------------------    Matrix    entry
accessing functions ---------------------------------------- */

inline float_prec& Matrix::operator () (const int16_t _row, const
int16_t _col) {
    #if (defined(MATRIX_USE_BOUNDS_CHECKING))
        ASSERT((_row >= 0) && (_row < this->i16row) && (_row <
MATRIX_MAXIMUM_SIZE),
                "Matrix index out-of-bounds (at row evaluation)");
        ASSERT((_col >= 0) && (_col < this->i16col) && (_col <
MATRIX_MAXIMUM_SIZE),
                "Matrix index out-of-bounds (at column _column)");
    #else
        #warning("Matrix bounds checking is disabled”);
    #endif
    return this->floatData[_row][_col];
}
inline float_prec Matrix::operator () (const int16_t _row, const
int16_t _col) const {
    #if (defined(MATRIX_USE_BOUNDS_CHECKING))
        ASSERT((_row >= 0) && (_row < this->i16row) && (_row <
MATRIX_MAXIMUM_SIZE),
                "Matrix index out-of-bounds (at row evaluation)");
        ASSERT((_col >= 0) && (_col < this->i16col) && (_col <
MATRIX_MAXIMUM_SIZE),
                "Matrix index out-of-bounds (at column _column)");
```

```
    #else
        #warning("Matrix bounds checking is disabled”);
    #endif
    return this->floatData[_row][_col];
}
inline float_prec & Matrix::Proxy::operator [] (const int16_t
_col) {
    #if (defined(MATRIX_USE_BOUNDS_CHECKING))
        ASSERT((_col >= 0) && (_col < this->_maxCol) && (_col <
MATRIX_MAXIMUM_SIZE),
                "Matrix index out-of-bounds (at column
evaluation)");
    #else
        #warning("Matrix bounds checking is disabled”);
    #endif
    return _array.ptr[_col];
}
inline float_prec Matrix::Proxy::operator [] (const int16_t _col)
const {
    #if (defined(MATRIX_USE_BOUNDS_CHECKING))
        ASSERT((_col >= 0) && (_col < this->_maxCol) && (_col <
MATRIX_MAXIMUM_SIZE),
                "Matrix index out-of-bounds (at column
evaluation)");
    #else
        #warning("Matrix bounds checking is disabled”);
    #endif
    return _array.cptr[_col];
}
inline Matrix::Proxy Matrix::operator [] (const int16_t _row) {
```

```
    #if (defined(MATRIX_USE_BOUNDS_CHECKING))
        ASSERT((_row >= 0) && (_row < this->i16row) && (_row < MATRIX_MAXIMUM_SIZE),
               "Matrix index out-of-bounds (at row evaluation)");
    #else
        #warning("Matrix bounds checking is disabled”);
    #endif
    return Proxy(floatData[_row], this->i16col);    /* Parsing column index for bound checking */
}
inline const Matrix::Proxy Matrix::operator [] (const int16_t _row) const {
    #if (defined(MATRIX_USE_BOUNDS_CHECKING))
        ASSERT((_row >= 0) && (_row < this->i16row) && (_row < MATRIX_MAXIMUM_SIZE),
               "Matrix index out-of-bounds (at row evaluation)");
    #else
        #warning("Matrix bounds checking is disabled”);
    #endif
    return Proxy(floatData[_row], this->i16col);    /* Parsing column index for bound checking */
}
/*    ---------------------------------------    Matrix    checking function declaration ---------------------------------------- */
/*    ---------------------------------------    Matrix    checking function declaration ---------------------------------------- */

inline bool Matrix::bMatrixIsValid(void) {
    /* Check whether the matrix is valid or not */
    if    ((this->i16row    >    0)    &&    (this->i16row    <= MATRIX_MAXIMUM_SIZE) &&
```

```
        (this->i16col     >     0)     &&     (this->i16col     <=
MATRIX_MAXIMUM_SIZE))
    {
        return true;
    } else {
        return false;
    }
}

inline void Matrix::vSetMatrixInvalid(void) {
    this->i16row = -1;
    this->i16col = -1;
}

inline bool Matrix::bMatrixIsSquare(void) {
    return (this->i16row == this->i16col);
}

/*  ----------------------------------------  Matrix  elementary
operations ----------------------------------------- */
/*  ----------------------------------------  Matrix  elementary
operations ----------------------------------------- */

inline bool Matrix::operator == (const Matrix& _compare) const {
    if  ((this->i16row  !=  _compare.i16row)  ||  (this->i16col  !=
_compare.i16col)) {
        return false;
    }
```

```
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            if (fabs((*this)(_i,_j) - _compare(_i,_j)) >
float_prec(float_prec_ZERO_ECO)) {
                return false;
            }
        }
    }
    return true;
}

inline bool Matrix::operator != (const Matrix& _compare) const {
    return (!(*this == _compare));
}

inline Matrix Matrix::operator - (void) const {
    Matrix _outp(this->i16row, this->i16col, NoInitMatZero);

    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            _outp(_i,_j) = -(*this)(_i,_j);
        }
    }
    return _outp;
}

inline Matrix Matrix::operator + (const float_prec _scalar) const
{
```

```
    Matrix            _outp(this->i16row,           this->i16col,
Matrix::NoInitMatZero);


    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            _outp(_i,_j) = (*this)(_i,_j) + _scalar;
        }
    }
    return _outp;
}


inline Matrix Matrix::operator - (const float_prec _scalar) const
{
    Matrix            _outp(this->i16row,           this->i16col,
Matrix::NoInitMatZero);


    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            _outp(_i,_j) = (*this)(_i,_j) - _scalar;
        }
    }
    return _outp;
}


inline Matrix Matrix::operator * (const float_prec _scalar) const
{
    Matrix            _outp(this->i16row,           this->i16col,
Matrix::NoInitMatZero);


    for (int16_t _i = 0; _i < this->i16row; _i++) {
```

```
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            _outp(_i,_j) = (*this)(_i,_j) * _scalar;
        }
    }
    return _outp;
}

inline Matrix Matrix::operator / (const float_prec _scalar) const
{
    Matrix           _outp(this->i16row,           this->i16col,
Matrix::NoInitMatZero);

    if (fabs(_scalar) < float_prec(float_prec_ZERO_ECO)) {
        _outp.vSetMatrixInvalid();
        return _outp;
    }
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            _outp(_i,_j) = (*this)(_i,_j) / _scalar;
        }
    }
    return _outp;
}


inline Matrix operator + (const float_prec _scalar, const Matrix&
_mat) {
    Matrix       _outp(_mat.i16getRow(),       _mat.i16getCol(),
Matrix::NoInitMatZero);
```

```
    for (int16_t _i = 0; _i < _mat.i16getRow(); _i++) {
        for (int16_t _j = 0; _j < _mat.i16getCol(); _j++) {
            _outp(_i,_j) = _scalar + _mat(_i,_j);
        }
    }
    return _outp;
}

inline Matrix operator - (const float_prec _scalar, const Matrix&
_mat) {
    Matrix       _outp(_mat.i16getRow(),       _mat.i16getCol(),
Matrix::NoInitMatZero);

    for (int16_t _i = 0; _i < _mat.i16getRow(); _i++) {
        for (int16_t _j = 0; _j < _mat.i16getCol(); _j++) {
            _outp(_i,_j) = _scalar - _mat(_i,_j);
        }
    }
    return _outp;
}

inline Matrix operator * (const float_prec _scalar, const Matrix&
_mat) {
    Matrix       _outp(_mat.i16getRow(),       _mat.i16getCol(),
Matrix::NoInitMatZero);

    for (int16_t _i = 0; _i < _mat.i16getRow(); _i++) {
        for (int16_t _j = 0; _j < _mat.i16getCol(); _j++) {
```

```
            _outp(_i,_j) = _scalar * _mat(_i,_j);
        }
    }
    return _outp;
}

inline Matrix Matrix::operator + (const Matrix& _matAdd) const {
    Matrix _outp(this->i16row, this->i16col, NoInitMatZero);
    if ((this->i16row != _matAdd.i16row) || (this->i16col != _matAdd.i16col)) {
        _outp.vSetMatrixInvalid();
        return _outp;
    }

    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            _outp(_i,_j) = (*this)(_i,_j) + _matAdd(_i,_j);
        }
    }
    return _outp;
}

inline Matrix Matrix::operator - (const Matrix& _matSub) const {
    Matrix _outp(this->i16row, this->i16col, NoInitMatZero);
    if ((this->i16row != _matSub.i16row) || (this->i16col != _matSub.i16col)) {
        _outp.vSetMatrixInvalid();
        return _outp;
```

```
    }

    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            _outp(_i,_j) = (*this)(_i,_j) - _matSub(_i,_j);
        }
    }
    return _outp;
}

inline Matrix Matrix::operator * (const Matrix& _matMul) const {
    Matrix _outp(this->i16row, _matMul.i16col, NoInitMatZero);
    if ((this->i16col != _matMul.i16row)) {
        _outp.vSetMatrixInvalid();
        return _outp;
    }

    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < _matMul.i16col; _j++) {
            _outp(_i,_j) = 0.0;
            for (int16_t _k = 0; _k < this->i16col; _k++) {
                _outp(_i,_j)      +=      ((*this)(_i,_k)      *
_matMul(_k,_j));
            }
        }
    }
    return _outp;
}
```

```
/*  ------------------------------------------  Simple  Matrix
operations ------------------------------------------ */
/*  ------------------------------------------  Simple  Matrix
operations ------------------------------------------ */

inline void Matrix::vRoundingElementToZero(const int16_t _i, const
int16_t _j) {
    if (fabs((*this)(_i,_j)) < float_prec(float_prec_ZERO_ECO)) {
        (*this)(_i,_j) = 0.0;
    }
}

inline Matrix Matrix::RoundingMatrixToZero(void) {
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            if               (fabs((*this)(_i,_j))              <
float_prec(float_prec_ZERO_ECO)) {
                (*this)(_i,_j) = 0.0;
            }
        }
    }
    return (*this);
}

inline void Matrix::vSetHomogen(const float_prec _val) {
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            (*this)(_i,_j) = _val;
```

```
        }
    }
}

inline void Matrix::vSetToZero(void) {
    this->vSetHomogen(0.0);
}

inline void Matrix::vSetRandom(const int32_t _maxRand, const
int32_t _minRand) {
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            (*this)(_i,_j) = float_prec((rand() % (_maxRand -
_minRand + 1)) + _minRand);
        }
    }
}

inline void Matrix::vSetDiag(const float_prec _val) {
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            if (_i == _j) {
                (*this)(_i,_j) = _val;
            } else {
                (*this)(_i,_j) = 0.0;
            }
        }
    }
```

```
}

inline void Matrix::vSetIdentity(void) {
    this->vSetDiag(1.0);
}

inline Matrix MatIdentity(const int16_t _i16size) {
    Matrix _outp(_i16size, _i16size, Matrix::NoInitMatZero);
    _outp.vSetDiag(1.0);
    return _outp;
}

/* Return the transpose of the matrix */
inline Matrix Matrix::Transpose(void) {
    Matrix _outp(this->i16col, this->i16row, NoInitMatZero);
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            _outp(_j,_i) = (*this)(_i,_j);
        }
    }
    return _outp;
}

/* Normalize the vector */
inline bool Matrix::bNormVector(void) {
    float_prec _normM = 0.0;
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
```

```
            _normM = _normM + ((*this)(_i,_j) * (*this)(_i,_j));
        }
    }

    /* Rounding to zero to avoid case where sqrt(0-), and _normM
always positive */
    if (_normM < float_prec(float_prec_ZERO)) {
        return false;
    }
    _normM = sqrt(_normM);
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        for (int16_t _j = 0; _j < this->i16col; _j++) {
            (*this)(_i,_j) /= _normM;
        }
    }
    return true;
}
/*     ---------------------------------------     Matrix/Vector
insertion operations ---------------------------------------- */
/*     ---------------------------------------     Matrix/Vector
insertion operations ---------------------------------------- */

/* Insert vector into matrix at _posCol position
inline Matrix Matrix::InsertVector(const Matrix& _Vector, const
int16_t _posCol) {
    Matrix _outp(*this);
    if ((_Vector.i16row > this->i16row) || (_Vector.i16col+_posCol
> this->i16col)) {
        /* Return false */
        _outp.vSetMatrixInvalid();
```

```
        return _outp;
    }
    for (int16_t _i = 0; _i < _Vector.i16row; _i++) {
        _outp(_i,_posCol) = _Vector(_i,0);
    }
    return _outp;
}

inline Matrix Matrix::InsertSubMatrix(const Matrix& _subMatrix,
const int16_t _posRow, const int16_t _posCol) {
    Matrix _outp(*this);
    if (((_subMatrix.i16row+_posRow) > this->i16row) ||
((_subMatrix.i16col+_posCol) > this->i16col)) {
        /* Return false */
        _outp.vSetMatrixInvalid();
        return _outp;
    }
    for (int16_t _i = 0; _i < _subMatrix.i16row; _i++) {
        for (int16_t _j = 0; _j < _subMatrix.i16col; _j++) {
            _outp(_i + _posRow,_j + _posCol) = _subMatrix(_i,_j);
        }
    }
    return _outp;
}

inline Matrix Matrix::InsertSubMatrix(const Matrix& _subMatrix,
const int16_t _posRow, const int16_t _posCol,
                                      const int16_t _lenRow,
const int16_t _lenColumn)
```

```
{
    Matrix _outp(*this);
    if      (((_lenRow+_posRow)      >      this->i16row)      ||
((_lenColumn+_posCol) > this->i16col) ||
        (_lenRow   >   _subMatrix.i16row)   ||   (_lenColumn   >
_subMatrix.i16col))
    {
        /* Return false */
        _outp.vSetMatrixInvalid();
        return _outp;
    }
    for (int16_t _i = 0; _i < _lenRow; _i++) {
        for (int16_t _j = 0; _j < _lenColumn; _j++) {
            _outp(_i + _posRow,_j + _posCol) = _subMatrix(_i,_j);
        }
    }
    return _outp;
}

/* Insert the _lenRow & _lenColumn submatrix, start from _posRowSub
& _posColSub submatrix;
 *  into matrix at the matrix's _posRow and _posCol position.
 *
inline Matrix Matrix::InsertSubMatrix(const Matrix& _subMatrix,
const int16_t _posRow, const int16_t _posCol,
                        const int16_t _posRowSub, const int16_t
_posColSub,
                        const int16_t _lenRow, const int16_t
_lenColumn)
{
```

```
    Matrix _outp(*this);
    if      (((_lenRow+_posRow)      >      this->i16row)      ||
((_lenColumn+_posCol) > this->i16col) ||
        ((_posRowSub+_lenRow)     >     _subMatrix.i16row)     ||
((_posColSub+_lenColumn) > _subMatrix.i16col))
    {
        /* Return false */
        _outp.vSetMatrixInvalid();
        return _outp;
    }
    for (int16_t _i = 0; _i < _lenRow; _i++) {
        for (int16_t _j = 0; _j < _lenColumn; _j++) {
            _outp(_i     +     _posRow,_j     +     _posCol)     =
_subMatrix(_posRowSub+_i,_posColSub+_j);
        }
    }
    return _outp;
}



/*    ------------------------------------------------    Large
operations ------------------------------------------------ */
/*    ------------------------------------------------    Large
operations ------------------------------------------------ */


/* Invers operation using Gauss-Jordan algorithm */
inline Matrix Matrix::Invers(void) const {
    Matrix _outp(this->i16row, this->i16col, NoInitMatZero);
    Matrix _temp(*this);
    _outp.vSetIdentity();
```

```
    /* Gauss Elimination... */
    for (int16_t _j = 0; _j < (_temp.i16row)-1; _j++) {
        for (int16_t _i = _j+1; _i < _temp.i16row; _i++) {
            if (fabs(_temp(_j,_j)) < float_prec(float_prec_ZERO))
{
                /* Matrix is non-invertible */
                _outp.vSetMatrixInvalid();
                return _outp;
            }

            float_prec _tempfloat = _temp(_i,_j) / _temp(_j,_j);

            for (int16_t _k = 0; _k < _temp.i16col; _k++) {
                _temp(_i,_k) -= (_temp(_j,_k) * _tempfloat);
                _outp(_i,_k) -= (_outp(_j,_k) * _tempfloat);

                _temp.vRoundingElementToZero(_i, _k);
                _outp.vRoundingElementToZero(_i, _k);
            }

        }
    }

    #if (1)
        for (int16_t _i = 1; _i < _temp.i16row; _i++) {
            for (int16_t _j = 0; _j < _i; _j++) {
                _temp(_i,_j) = 0.0;
```

```
            }
        }
    #endif


    /* Jordan... */
    for (int16_t _j = (_temp.i16row)-1; _j > 0; _j--) {
        for (int16_t _i = _j-1; _i >= 0; _i--) {
            if (fabs(_temp(_j,_j)) < float_prec(float_prec_ZERO))
{
                /* Matrix is non-invertible */
                _outp.vSetMatrixInvalid();
                return _outp;
            }

            float_prec _tempfloat = _temp(_i,_j) / _temp(_j,_j);
            _temp(_i,_j) -= (_temp(_j,_j) * _tempfloat);
            _temp.vRoundingElementToZero(_i, _j);

            for (int16_t _k = (_temp.i16row - 1); _k >= 0; _k--)
{
                _outp(_i,_k) -= (_outp(_j,_k) * _tempfloat);
                _outp.vRoundingElementToZero(_i, _k);
            }
        }
    }
```

```
    /* Normalization */
    for (int16_t _i = 0; _i < _temp.i16row; _i++) {
        if (fabs(_temp(_i,_i)) < float_prec(float_prec_ZERO)) {
            /* Matrix is non-invertible */
            _outp.vSetMatrixInvalid();
            return _outp;
        }

        float_prec _tempfloat = _temp(_i,_i);
        _temp(_i,_i) = 1.0;

        for (int16_t _j = 0; _j < _temp.i16row; _j++) {
            _outp(_i,_j) /= _tempfloat;
        }
    }
    return _outp;
}

/* Use elemetary row operation to reduce the matrix into upper
triangular form
*
inline    bool    Matrix::bMatrixIsPositiveDefinite(const    bool
checkPosSemidefinite) const {
    bool _posDef, _posSemiDef;
    Matrix _temp(*this);

    /* Gauss Elimination... */
    for (int16_t _j = 0; _j < (_temp.i16row)-1; _j++) {
```

```
        for (int16_t _i = _j+1; _i < _temp.i16row; _i++) {
            if (fabs(_temp(_j,_j)) < float_prec(float_prec_ZERO))
{
                return false;
            }

            float_prec _tempfloat = _temp(_i,_j) / _temp(_j,_j);

            for (int16_t _k = 0; _k < _temp.i16col; _k++) {
                _temp(_i,_k) -= (_temp(_j,_k) * _tempfloat);
                _temp.vRoundingElementToZero(_i, _k);
            }

        }
    }

    _posDef = true;
    _posSemiDef = true;
    for (int16_t _i = 0; _i < _temp.i16row; _i++) {
        if (_temp(_i,_i) < float_prec(float_prec_ZERO)) {
            /* false if less than 0+ (zero included) */
            _posDef = false;
        }
        if (_temp(_i,_i) < -float_prec(float_prec_ZERO)) {
            /* false if less than 0- (zero is not included) */
            _posSemiDef = false;
        }
    }
```

```
    if (checkPosSemidefinite) {
        return _posSemiDef;
    } else {
        return _posDef;
    }
}

inline Matrix Matrix::GetDiagonalEntries(void) const {
    Matrix _temp(this->i16row, 1, NoInitMatZero);

    if (this->i16row != this->i16col) {
        _temp.vSetMatrixInvalid();
        return _temp;
    }
    for (int16_t _i = 0; _i < this->i16row; _i++) {
        _temp(_i,0) = (*this)(_i,_i);
    }
    return _temp;
}

/* Do the Cholesky Decomposition using Cholesky-Crout algorithm.
*/
inline Matrix Matrix::CholeskyDec(void) const {
    float_prec _tempFloat;

    /* Note that _outp need to be initialized as zero matrix */
    Matrix _outp(this->i16row, this->i16col, InitMatWithZero);
```

```
    if (this->i16row != this->i16col) {
        _outp.vSetMatrixInvalid();
        return _outp;
    }
    for (int16_t _j = 0; _j < this->i16col; _j++) {
        for (int16_t _i = _j; _i < this->i16row; _i++) {
            _tempFloat = (*this)(_i,_j);
            if (_i == _j) {
                for (int16_t _k = 0; _k < _j; _k++) {
                    _tempFloat = _tempFloat - (_outp(_i,_k) *
_outp(_i,_k));
                }
                if (_tempFloat < -float_prec(float_prec_ZERO)) {
                    /* Matrix is not positif (semi)definit */
                    _outp.vSetMatrixInvalid();
                    return _outp;
                }
                /* Rounding to zero to avoid case where sqrt(0-)
*/
                if              (fabs(_tempFloat)              <
float_prec(float_prec_ZERO)) {
                    _tempFloat = 0.0;
                }
                _outp(_i,_i) = sqrt(_tempFloat);
            } else {
                for (int16_t _k = 0; _k < _j; _k++) {
                    _tempFloat = _tempFloat - (_outp(_i,_k) *
_outp(_j,_k));
```

```
                }
                if              (fabs(_outp(_j,_j))             <
float_prec(float_prec_ZERO)) {
                    /* Matrix is not positif definit */
                    _outp.vSetMatrixInvalid();
                    return _outp;
                }
                _outp(_i,_j) = _tempFloat / _outp(_j,_j);
            }
        }
    }
    return _outp;
}

/*  Do  the  Householder  Transformation  for  QR  Decomposition
operation.
*/
inline   Matrix   Matrix::HouseholderTransformQR(const   int16_t
_rowTransform, const int16_t _colTransform) {
    float_prec _tempFloat;
    float_prec _xLen;
    float_prec _x1;
    float_prec _u1;
    float_prec _vLen2;

    /* Note that _outp & _vectTemp need to be initialized as zero
matrix */
    Matrix _outp(this->i16row, this->i16row, InitMatWithZero);
    Matrix _vectTemp(this->i16row, 1, InitMatWithZero);
```

```
    if ((_rowTransform >= this->i16row) || (_colTransform >= this-
>i16col)) {
        _outp.vSetMatrixInvalid();
        return _outp;
    }

    _x1 = (*this)(_rowTransform,_colTransform);
    _xLen = _x1*_x1;
    _vLen2 = 0.0;
    for (int16_t _i = _rowTransform+1; _i < this->i16row; _i++) {
        _vectTemp(_i,0) = (*this)(_i,_colTransform);

        _tempFloat = _vectTemp(_i,0) * _vectTemp(_i,0);
        _xLen  += _tempFloat;
        _vLen2 += _tempFloat;
    }
    _xLen = sqrt(_xLen);

    if (_x1 < 0.0) {
        _u1 = _x1+_xLen;
    } else {
        _u1 = _x1-_xLen;
    }
    _vLen2 += (_u1*_u1);
    _vectTemp(_rowTransform,0) = _u1;

    if (fabs(_vLen2) < float_prec(float_prec_ZERO_ECO)) {
```

```
        /* x vector is collinear with basis vector e, return result
= I */
        _outp.vSetIdentity();
    } else {
        for (int16_t _i = 0; _i < this->i16row; _i++) {
            _tempFloat = _vectTemp(_i,0);
            if (fabs(_tempFloat) > float_prec(float_prec_ZERO)) {
                for (int16_t _j = 0; _j < this->i16row; _j++) {
                    if          (fabs(_vectTemp(_j,0))          >
float_prec(float_prec_ZERO)) {
                        _outp(_i,_j) = _vectTemp(_j,0);
                        _outp(_i,_j) = _outp(_i,_j) * _tempFloat;
                        _outp(_i,_j)   =   _outp(_i,_j)   *   (-
2.0/_vLen2);
                    }
                }
            }
            _outp(_i,_i) = _outp(_i,_i) + 1.0;
        }
    }
    return _outp;
}

/*  Do  the  QR  Decomposition  for  matrix  using  Householder
Transformation.
*/
inline bool Matrix::QRDec(Matrix& Qt, Matrix& R) const {
    Matrix Qn(Qt.i16row, Qt.i16col, NoInitMatZero);
    if ((this->i16row < this->i16col) || (!Qt.bMatrixIsSquare())
|| (Qt.i16row != this->i16row) ||
```

```
        (R.i16row != this->i16row) || (R.i16col != this->i16col))
    {
        Qt.vSetMatrixInvalid();
        R.vSetMatrixInvalid();
        return false;
    }
    R = (*this);
    Qt.vSetIdentity();
    for (int16_t _i = 0; (_i < (this->i16row - 1)) && (_i < this-
>i16col-1); _i++) {
        Qn  = R.HouseholderTransformQR(_i, _i);
        if (!Qn.bMatrixIsValid()) {
            Qt.vSetMatrixInvalid();
            R.vSetMatrixInvalid();
            return false;
        }
        Qt = Qn * Qt;
        R  = Qn * R;
    }
#if (0)
    Qt.RoundingMatrixToZero();
    R.RoundingMatrixToZero();
#else
    Qt.RoundingMatrixToZero();
    for (int16_t _i = 1; ((_i < R.i16row) && (_i < R.i16col));
_i++) {
        for (int16_t _j = 0; _j < _i; _j++) {
            R(_i, _j) = 0.0;
```

```
        }
    }
#endif

    /* Q = Qt.Transpose */
    return true;
}

/* Do the back-subtitution operation for upper triangular matrix
 */
inline Matrix Matrix::BackSubtitution(const Matrix& A, const
Matrix& B) const {
    Matrix _outp(A.i16row, 1, NoInitMatZero);
    if ((A.i16row != A.i16col) || (A.i16row != B.i16row)) {
        _outp.vSetMatrixInvalid();
        return _outp;
    }

    for (int16_t _i = A.i16col-1; _i >= 0; _i--) {
        _outp(_i,0) = B(_i,0);
        for (int16_t _j = _i + 1; _j < A.i16col; _j++) {
            _outp(_i,0) = _outp(_i,0) - A(_i,_j)*_outp(_j,0);
        }
        if (fabs(A(_i,_i)) < float_prec(float_prec_ZERO)) {
            _outp.vSetMatrixInvalid();
            return _outp;
        }
        _outp(_i,0) = _outp(_i,0) / A(_i,_i);
```

```
    }
    return _outp;
}
inline Matrix Matrix::ForwardSubtitution(const Matrix& A, const
Matrix& B) const {
    Matrix _outp(A.i16row, 1, NoInitMatZero);
    if ((A.i16row != A.i16col) || (A.i16row != B.i16row)) {
        _outp.vSetMatrixInvalid();
        return _outp;
    }
    for (int16_t _i = 0; _i < A.i16row; _i++) {
        _outp(_i,0) = B(_i,0);
        for (int16_t _j = 0; _j < _i; _j++) {
            _outp(_i,0) = _outp(_i,0) - A(_i,_j)*_outp(_j,0);
        }
        if (fabs(A(_i,_i)) < float_prec(float_prec_ZERO)) {
            _outp.vSetMatrixInvalid();
            return _outp;
        }
        _outp(_i,0) = _outp(_i,0) / A(_i,_i);
    }
    return _outp;
}
/* ------------------------------------------ Matrix printing
function ------------------------------------------ */
/* ------------------------------------------ Matrix printing
function ------------------------------------------ */
#if (SYSTEM_IMPLEMENTATION == SYSTEM_IMPLEMENTATION_PC)
    inline void Matrix::vPrint(void) {
```

```
        for (int16_t _i = 0; _i < this->i16row; _i++) {
            cout << "[ ";
            for (int16_t _j = 0; _j < this->i16col; _j++) {
                cout << std::fixed << std::setprecision(3) <<
(*this)(_i,_j) << " ";
            }
            cout << "]";
            cout << endl;
        }
        cout << endl;
    }
    inline void Matrix::vPrintFull(void) {
        for (int16_t _i = 0; _i < this->i16row; _i++) {
            cout << "[ ";
            for (int16_t _j = 0; _j < this->i16col; _j++) {
                cout << resetiosflags( ios::fixed | ios::showpoint
) << (*this)(_i,_j) << " ";
            }
            cout << "]";
            cout << endl;
        }
        cout << endl;
    }
#elif                   (SYSTEM_IMPLEMENTATION                  ==
SYSTEM_IMPLEMENTATION_EMBEDDED_ARDUINO)
    inline void Matrix::vPrint(void) {
        char _bufSer[10];
        for (int16_t _i = 0; _i < this->i16row; _i++) {
            Serial.print("[ ");
```

```
            for (int16_t _j = 0; _j < this->i16col; _j++) {
                snprintf(_bufSer,  sizeof(_bufSer)-1,  "%2.2f  ",
(*this)(_i,_j));
                Serial.print(_bufSer);
            }
            Serial.println("]");
        }
        Serial.println("");
    }
    inline void Matrix::vPrintFull(void) {
        char _bufSer[32];
        for (int16_t _i = 0; _i < this->i16row; _i++) {
            Serial.print("[ ");
            for (int16_t _j = 0; _j < this->i16col; _j++) {
                snprintf(_bufSer,   sizeof(_bufSer)-1,   "%e   ",
(*this)(_i,_j));
                Serial.print(_bufSer);
            }
            Serial.println("]");
        }
        Serial.println("");
    }
#else
    /* void vPrint(); */
    /* void vPrintFull(); */
#endif

#endif // MATRIX_H
```

Code 4: Unscented Kalman Filter Computation

```
#include "ukf.h"


UKF::UKF(const Matrix& XInit, const Matrix& PInit, const Matrix&
Rv, const Matrix& Rn,
         bool (*bNonlinearUpdateX)(Matrix&, const Matrix&, const
Matrix&),
         bool (*bNonlinearUpdateY)(Matrix&, const Matrix&, const
Matrix&))
{
    /* Initialization: */
    this->X_Est = XInit;
    this->P     = PInit;
    this->Rv    = Rv;
    this->Rn    = Rn;
    this->bNonlinearUpdateX = bNonlinearUpdateX;
    this->bNonlinearUpdateY = bNonlinearUpdateY;
    float_prec _alpha   = 1e-2;
    float_prec _k       = 0.0;
    float_prec _beta    = 2.0;


    /* lambda = (alpha^2)*(N+kappa)-N,       gamma = sqrt(N+alpha)
...{UKF_1} */
    float_prec   _lambda     =   (_alpha*_alpha)*(SS_X_LEN+_k)   -
SS_X_LEN;
    Gamma = sqrt((SS_X_LEN + _lambda));



    /*  Wm  =  [lambda/(N+lambda)              1/(2(N+lambda))  ...
1/(2(N+lambda))]     ...{UKF_2} */
```

```
    Wm[0][0] = _lambda/(SS_X_LEN + _lambda);
    for (int16_t _i = 1; _i < Wm.i16getCol(); _i++) {
        Wm[0][_i] = 0.5/(SS_X_LEN + _lambda);
    }

    /*  Wc  =  [Wm(0)+(1-alpha(^2)+beta)    1/(2(N+lambda))  ...
1/(2(N+lambda))]     ...{UKF_3} */
    Wc = Wm;
    Wc[0][0] = Wc[0][0] + (1.0-(_alpha*_alpha)+_beta);
}

void UKF::vReset(const Matrix& XInit, const Matrix& PInit, const
Matrix& Rv, const Matrix& Rn)
{
    this->X_Est = XInit;
    this->P     = PInit;
    this->Rv    = Rv;
    this->Rn    = Rn;
}

bool UKF::bUpdate(const Matrix& Y, const Matrix& U)
{
    /* Run once every sampling time */

    /*   XSigma(k-1)   =   [x(k-1)   Xs(k-1)+GPsq   Xs(k-1)-GPsq]
...{UKF_4}  */
    if (!bCalculateSigmaPoint()) {
        return false;
    }
```

```
    /*    Unscented    Transform    XSigma    [f,XSigma,u,Rv]    ->
[x,XSigma,P,DX]:       ...{UKF_5a} - {UKF_8a} */
    if     (!bUnscentedTransform(X_Est,     X_Sigma,     P,     DX,
bNonlinearUpdateX, X_Sigma, U, Rv)) {
        return false;
    }
    if     (!bUnscentedTransform(Y_Est,     Y_Sigma,     Py,     DY,
bNonlinearUpdateY, X_Sigma, U, Rn)) {
        return false;
    }

    /* Calculate Cross-Covariance Matrix: */
    for (int16_t _i = 0; _i < DX.i16getRow(); _i++) {
        for (int16_t _j = 0; _j < DX.i16getCol(); _j++) {
            DX[_i][_j] *= Wc[0][_j];
        }
    }
    Pxy = DX * (DY.Transpose());

    /* Calculate the Kalman Gains */
    Matrix PyInv(Py.Invers());
    if (!PyInv.bMatrixIsValid()) {
        return false;
    }
    Gain = Pxy * PyInv;

    /* Update the Estimated State Variable:*/
    Err = Y - Y_Est;
```

```
    X_Est = X_Est + (Gain*Err);

    /* Update the Covariance Matrix:*/
    P = P - (Gain * Py * Gain.Transpose());

    return true;
}

bool UKF::bCalculateSigmaPoint(void)
{
    /* Use Cholesky Decomposition to compute sqrt(P) */
    P_Chol = P.CholeskyDec();
    if (!P_Chol.bMatrixIsValid()) {
        /* System Fail */
        return false;
    }
    P_Chol = P_Chol * Gamma;

    /* Xs(k-1) = [x(k-1) ... x(k-1)]            ; Xs(k-1) = NxN */
    Matrix _Y(SS_X_LEN, SS_X_LEN);
    for (int16_t _i = 0; _i < SS_X_LEN; _i++) {
        _Y = _Y.InsertVector(X_Est, _i);
    }

    X_Sigma.vSetToZero();
    /* XSigma(k-1) = [x(k-1) 0 0] */
    X_Sigma = X_Sigma.InsertVector(X_Est, 0);
    /* XSigma(k-1) = [x(k-1) Xs(k-1)+GPsq  0] */
```

```
    X_Sigma = X_Sigma.InsertSubMatrix((_Y + P_Chol), 0, 1);
    /* XSigma(k-1) = [x(k-1) Xs(k-1)+GPsq Xs(k-1)-GPsq] */
    X_Sigma = X_Sigma.InsertSubMatrix((_Y - P_Chol), 0,
(1+SS_X_LEN));


    return true;
}


bool UKF::bUnscentedTransform(Matrix& Out, Matrix& OutSigma,
Matrix& P, Matrix& DSig,
                              bool (*_vFuncNonLinear)(Matrix&
xOut, const Matrix& xInp, const Matrix& U),
                              const Matrix& InpSigma, const
Matrix& InpVector,
                              const Matrix& _CovNoise)
{
    Out.vSetToZero();
    for (int16_t _j = 0; _j < InpSigma.i16getCol(); _j++) {
        /* Transform the column submatrix of sigma-points input
matrix (InpSigma) */
        Matrix _AuxSigma1(InpSigma.i16getRow(), 1);
        Matrix _AuxSigma2(OutSigma.i16getRow(), 1);
        for (int16_t _i = 0; _i < InpSigma.i16getRow(); _i++) {
            _AuxSigma1[_i][0] = InpSigma[_i][_j];
        }
        if (!_vFuncNonLinear(_AuxSigma2, _AuxSigma1, InpVector))
{
            return false;
        }
```

```
        /* Combine the transformed vector to construct sigma-
points output matrix (OutSigma) */
        OutSigma = OutSigma.InsertVector(_AuxSigma2, _j);
        _AuxSigma2 = _AuxSigma2 * Wm[0][_j];
        Out = Out + _AuxSigma2;
    }
    /* DX = XSigma(k)(i) - Xs(k)   ; Xs(k) = [x(k|k-1) ... x(k|k-
1)]
     *                                        ; Xs(k) = Nx(2N+1)
...{UKF_7a}  */
    Matrix                          _AuxSigma1(OutSigma.i16getRow(),
OutSigma.i16getCol());
    for (int16_t _j = 0; _j < OutSigma.i16getCol(); _j++) {
        _AuxSigma1 = _AuxSigma1.InsertVector(Out, _j);
    }
    DSig = OutSigma - _AuxSigma1;

    /* P(k|k-1) = sum(Wc(i)*DX*DX') + Rv       ; i = 1 ... (2N+1)
...{UKF_8a}  */
    _AuxSigma1 = DSig;
    for (int16_t _i = 0; _i < DSig.i16getRow(); _i++) {
        for (int16_t _j = 0; _j < DSig.i16getCol(); _j++) {
            _AuxSigma1[_i][_j] *= Wc[0][_j];
        }
    }
    P = (_AuxSigma1 * (DSig.Transpose())) + _CovNoise;

    return true;
```

Code 5: Application of Unscented Kalman Filter Configuration and Matrices

```
#ifndef UKF_H
#define UKF_H

#include "konfig.h"
#include "matrix.h"

#if ((2*SS_X_LEN + 1) > MATRIX_MAXIMUM_SIZE)
    #error("The MATRIX_MAXIMUM_SIZE is too small for sigma points (at least need (2*SS_X_LEN + 1))!");
#endif

class UKF
{
public:
    UKF(const Matrix& XInit, const Matrix& PInit, const Matrix& Rv, const Matrix& Rn,
        bool (*bNonlinearUpdateX)(Matrix&, const Matrix&, const Matrix&),
        bool (*bNonlinearUpdateY)(Matrix&, const Matrix&, const Matrix&));
    void vReset(const Matrix& XInit, const Matrix& PInit, const Matrix& Rv, const Matrix& Rn);
    bool bUpdate(const Matrix& Y, const Matrix& U);

    const Matrix GetX()   const { return X_Est; }
    const Matrix GetY()   const { return Y_Est; }
    const Matrix GetP()   const { return P; }
```

```
    const Matrix GetErr() const { return Err; }


protected:

    bool bCalculateSigmaPoint(void);

    bool bUnscentedTransform(Matrix& Out, Matrix& OutSigma,
Matrix& P, Matrix& DSig,

                             bool      (*_vFuncNonLinear)(Matrix&
xOut, const Matrix& xInp, const Matrix& U),

                             const   Matrix&   InpSigma,   const
Matrix& InpVector,

                             const Matrix& _CovNoise);


private:

    bool (*bNonlinearUpdateX) (Matrix& X_dot, const Matrix& X,
const Matrix& U);

    bool (*bNonlinearUpdateY) (Matrix& Y_Est, const Matrix& X,
const Matrix& U);


    Matrix X_Est{SS_X_LEN, 1};

    Matrix X_Sigma{SS_X_LEN, (2*SS_X_LEN + 1)};


    Matrix Y_Est{SS_Z_LEN, 1};

    Matrix Y_Sigma{SS_Z_LEN, (2*SS_X_LEN + 1)};


    Matrix P{SS_X_LEN, SS_X_LEN};

    Matrix P_Chol{SS_X_LEN, SS_X_LEN};


    Matrix DX{SS_X_LEN, (2*SS_X_LEN + 1)};

    Matrix DY{SS_Z_LEN, (2*SS_X_LEN + 1)};
```

```
    Matrix Py{SS_Z_LEN, SS_Z_LEN};
    Matrix Pxy{SS_X_LEN, SS_Z_LEN};

    Matrix Wm{1, (2*SS_X_LEN + 1)};
    Matrix Wc{1, (2*SS_X_LEN + 1)};

    Matrix Rv{SS_X_LEN, SS_X_LEN};
    Matrix Rn{SS_Z_LEN, SS_Z_LEN};

    Matrix Err{SS_Z_LEN, 1};
    Matrix Gain{SS_X_LEN, SS_Z_LEN};
    float_prec Gamma;
};

#endif // UKF_H
```

Code 6: Angular Acceleration and Torque of Probe

```
import numpy as np


# Constants
weight = 3  # in kg
g = 9.81  # gravitational acceleration in m/s^2
length = 0.3  # length of the prism in meters (300 mm)
width = 0.1  # width of the prism in meters (200 mm)
height = 0.1  # height of the prism in meters (100 mm)


# Moment of Inertia for a rectangular prism (I) about its center
# Since the rotation is around its length, we use the formula I = (1/12) * m * (w^2 + h^2)
moment_of_inertia = (1/12) * weight * (width**2 + height**2)


# Function to calculate torque and angular acceleration for a given angle
def calculate_torque_and_acceleration(angle):
    # The distance (r) from pivot to center of mass changes with angle
    r = (length / 2) / np.cos(np.radians(angle))  # the distance from the pivot to the center of mass


    # Torque (τ = r * F * sin(θ)), F is the force due to weight
    force = weight * g
    torque = r * force * np.sin(np.radians(angle))


    # Angular acceleration (α = τ / I)
    angular_acceleration = torque / moment_of_inertia
```

```
    return torque, angular_acceleration


# Calculate torque and angular acceleration for angles from 0 to
45 degrees
angles = range(0, 46)  # 0 to 45 degrees
torques = []
angular_accelerations = []


for angle in angles:
    torque,                 angular_acceleration                 =
calculate_torque_and_acceleration(angle)
    torques.append(torque)
    angular_accelerations.append(angular_acceleration)


for angle, torque, angular_acceleration in zip(angles, torques,
angular_accelerations):
    print(f"Angle: {angle}° - Torque: {torque:.2f} mNm, Angular
Acceleration: {angular_acceleration:.2f} rad/s²")
```

Code 7: Counteractive Inertia of Reaction Wheels

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize_scalar

# Constants and CubeSat properties
mass_cubesat = 8  # kg
dimensions_cubesat = np.array([0.6, 0.2, 0.1])  # meters
initial_angle = np.deg2rad(15)  # radians
mass_wheel = 0.5  # kg
thicknesses = [0.00635, 0.0127]  # meters
tilt_angle = np.deg2rad(45)  # radians

# Define the range of radii
radii = np.linspace(0.01, 0.05, 50)  # 50 values from 10 mm (0.01 m) to 50 mm (0.05 m)

# Define the range of angles for the plot
angles = np.linspace(np.deg2rad(15), 0, 50)  # 50 values from 15 degrees to 0 degrees

# Define the time step
time_step = 0.01  # seconds

# Moment of Inertia of CubeSat [Rectangular Prism: I = 1/12*m*(H^2+D^2)]
def moment_of_inertia_cubesat(mass, dimensions):
    width, height, depth = dimensions
    ix = (1 / 12) * mass * (height ** 2 + depth ** 2)
```

```
    iy = (1 / 12) * mass * (width ** 2 + depth ** 2)
    iz = (1 / 12) * mass * (width ** 2 + height ** 2)
    return np.diag([ix, iy, iz])

I_cubesat = moment_of_inertia_cubesat(mass_cubesat,
dimensions_cubesat)

# Moment of Inertia of a single Reaction Wheel [Hollow Cylinder:
I = 1/2*m*(R^2+r^2)]
def moment_of_inertia_wheel(mass, radius, thickness, tilt_angle):
    # Assuming the mass is evenly distributed over the volume of
the hollow cylinder
    outer_radius = radius
    inner_radius = radius - thickness
    i_wheel = (1 / 2) * mass * (outer_radius ** 2 + inner_radius
** 2)
    # Adjusting for the tilt angle (45 degrees)
    ix = i_wheel * np.sin(tilt_angle) ** 2
    iy = i_wheel * np.cos(tilt_angle) ** 2
    iz = i_wheel  # No tilt in z-axis
    return np.array([ix, iy, iz])

# Constants for the new constraint
max_rpm = 6000  # Maximum revolutions per minute for the reaction
wheels
max_angular_velocity = (max_rpm * 2 * np.pi) / 60

# Re-defining the control strategy with max RPM constraint,
including Kp
Kp = 0.1  # Proportional gain (arbitrary for this simulation)
```

```
# Adjusted function to incorporate different thicknesses
# Modified control strategy function to return final angular momentum as well
def control_strategy_with_angular_momentum(radius, thickness):
    i_wheel = moment_of_inertia_wheel(mass_wheel, radius, thickness, tilt_angle)
    i_total = I_cubesat + 4 * np.diag(i_wheel)

    time_to_stabilize = 0
    angle = initial_angle
    angular_velocity = 0
    while abs(angle) > 0.01:
        torque = -Kp * angle
        angular_acceleration = torque / i_total[0, 0]
        angular_velocity += angular_acceleration * time_step
        angular_velocity = np.clip(angular_velocity, -max_angular_velocity, max_angular_velocity)
        angle += angular_velocity * time_step
        time_to_stabilize += time_step

    # Final angular momentum
    final_angular_momentum = i_total[0, 0] * angular_velocity

    return time_to_stabilize, final_angular_momentum
```

```
# Calculating stabilization times and angular momentum for each thickness
times_and_momentum_for_thicknesses = []
for thickness in thicknesses:
    times_and_momentum = np.array([control_strategy_with_angular_momentum(r, thickness) for r in radii])

times_and_momentum_for_thicknesses.append(times_and_momentum)



# Wrapper function for optimization that only returns time to stabilize
def optimization_wrapper(radius, thickness):
    time_to_stabilize, _ = control_strategy_with_angular_momentum(radius, thickness)
    return time_to_stabilize



# Running optimization with max RPM constraint for each thickness
optimal_results_for_optimization = []
for thickness in thicknesses:
    result = minimize_scalar(optimization_wrapper, args=(thickness,), bounds=(0.01, 0.05), method='bounded')
    optimal_results_for_optimization.append((result.x, result.fun))

# Extracting the optimal radii and times for each thickness
optimal_radii_for_optimization = [result[0] for result in optimal_results_for_optimization]
```

```
optimal_times_for_optimization = [result[1] for result in optimal_results_for_optimization]


for thickness, radius, time in zip(thicknesses, optimal_radii_for_optimization, optimal_times_for_optimization):
    print(f"For a reaction wheel thickness of {thickness*1000} millimeters, "
          f"the optimal radius is approximately {radius*1000:.5f} millimeters, "
          f"resulting in a stabilization time of about {time:.2f} seconds.")



# Create a new 3D plot with adjusted label positions and font
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Font settings
font = {'family': 'Georgia Pro'}

# Plot for each thickness (time to stabilize and angular momentum)
for i, thickness in enumerate(thicknesses):
    times = times_and_momentum_for_thicknesses[i][:, 0]
    angular_momentums = times_and_momentum_for_thicknesses[i][:, 1]
    ax.plot(np.linspace(0.01, 0.05, 50), times, angular_momentums, label=f'Thickness {thickness} m')

# Set labels with increased padding and updated font
ax.set_xlabel('Radius of Reaction Wheel (m)', labelpad=15, fontdict=font)
```

```
ax.set_ylabel('Time to Stabilize with Max RPM (s)', labelpad=15, fontdict=font)

ax.set_zlabel('Final Angular Momentum (kg·m²/s)', labelpad=20, fontdict=font)

ax.set_title('Impact of Wheel Thickness on CubeSat Stabilization and Angular Momentum', fontdict=font)

ax.legend()


# Show the plot with adjustments

plt.show()
```

Code 8: VPython Simulation of IMU

```
from vpython import *
from time import *
import numpy as np
import math
import serial
ad=serial.Serial('com3',115200)
sleep(1)


scene.range=5
scene.background=color.black
toRad=2*np.pi/360
toDeg=1/toRad
scene.forward=vector(-1,-1,-1)


scene.width=800
scene.height=1000


xarrow=arrow(lenght=2,                                  shaftwidth=.1,
color=color.red,axis=vector(1,0,0))
yarrow=arrow(lenght=2,                                  shaftwidth=.1,
color=color.green,axis=vector(0,1,0))
zarrow=arrow(lenght=4,                                  shaftwidth=.1,
color=color.blue,axis=vector(0,0,1))


frontArrow=arrow(length=4,shaftwidth=.1,color=color.purple,axis=
vector(1,0,0))
upArrow=arrow(length=1,shaftwidth=.1,color=color.magenta,axis=ve
ctor(0,1,0))
```

```
sideArrow=arrow(length=2,shaftwidth=.1,color=color.orange,axis=v
ector(0,0,1))


# Dimensions for the CubeSat components
cubeSatLength = 3  # 30 cm
cubeSatWidth = 2   # 20 cm
cubeSatHeight = 0.6  # 6 cm


# Define the components of the CubeSat
cubeSatBody   =   box(length=cubeSatLength,   width=cubeSatWidth,
height=cubeSatHeight, color=color.gray(0.5))


# Create a compound object for the CubeSat
cubeSat = compound([cubeSatBody])
while (True):
    try:
        while (ad.inWaiting()==0):
            pass
        dataPacket=ad.readline()
        dataPacket=str(dataPacket,'utf-8')
        splitPacket=dataPacket.split(",")
        q0=float(splitPacket[0])
        q1=float(splitPacket[1])
        q2=float(splitPacket[2])
        q3=float(splitPacket[3])


        roll=-math.atan2(2*(q0*q1+q2*q3),1-2*(q1*q1+q2*q2))
        pitch=math.asin(2*(q0*q2-q3*q1))
```

```
        yaw=-math.atan2(2*(q0*q3+q1*q2),1-2*(q2*q2+q3*q3))-
np.pi/2


        rate(60)
        k=vector(cos(yaw)*cos(pitch),
sin(pitch),sin(yaw)*cos(pitch))
        y=vector(0,1,0)
        s=cross(k,y)
        v=cross(s,k)
        vrot=v*cos(roll)+cross(k,v)*sin(roll)


        frontArrow.axis=k
        sideArrow.axis=cross(k,vrot)
        upArrow.axis=vrot
        cubeSat.axis = k
        cubeSat.up = vrot
        sideArrow.length=2
        frontArrow.length=4
        upArrow.length=1
    except:
        pass
```

Code 9: Visualization Switcher

```
function switch_visualizer()
    % Close any existing figures to prevent overlap and
confusion
    close all;

    % Create the main GUI figure
    mainFig = figure('Name', 'Quaternion Visualizer Selector',
'Position', [100, 100, 300, 200]);

    % Button for Visualizer 1
    uicontrol('Style', 'pushbutton', 'String', 'Visualizer 1',
...
              'Position', [50 120 200 50], 'Callback',
@startVisualizer1);

    % Button for Visualizer 2
    uicontrol('Style', 'pushbutton', 'String', 'Visualizer 2',
...
              'Position', [50 50 200 50], 'Callback',
@startVisualizer2);

    % Function to start Visualizer 1
    function startVisualizer1(src, event)
        fprintf('Starting Visualizer 1\n');
        quaternion_rotation_visualizer1(); % Assumes this
function exists
    end

    % Function to start Visualizer 2
    function startVisualizer2(src, event)
```

```
        fprintf('Starting Visualizer 2\n');
        quaternion_rotation_visualizer2(); % Assumes this
function exists
    end
end
```

Code 10: Close Approach Asteroid Perturbation Simulation

```
function asteroid_perturbation_simulation()
    % Constants
    G = 6.67430e-11; % Gravitational constant
    deltaTime = 1; % Time step for the simulation in seconds
    totalSimulationTime = 36000; % Total simulation time in
seconds
    initialQuaternion = [1, 0, 0, 0]; % Initial quaternion (no
rotation)

    % Mass and distance parameters
    massAsteroid = 2.58945e10; % Mass of asteroid in kg
    massObject = 0.8; % Mass of the object/spacecraft in kg
    distance = 0.8; % Distance between asteroid and object in
meters

    % Inertia matrix components for a 6U CubeSat
     dimensions = [0.3, 0.2, 0.1]; % Dimensions in meters
    m = massObject; % Mass in kg
    Ixx = m/12 * (dimensions(2)^2 + dimensions(3)^2);
    Iyy = m/12 * (dimensions(1)^2 + dimensions(3)^2);
    Izz = m/12 * (dimensions(1)^2 + dimensions(2)^2);
    InertiaMatrix = diag([Ixx, Iyy, Izz]); % Diagonal inertia
matrix

    % Time vector
    timeVector = 0:deltaTime:totalSimulationTime;

    % Initialize quaternion data for plotting
    quaternionData = zeros(length(timeVector), 4); % Initialize
with zeros
```

```
    % Initial conditions
    angularVelocity = [0, 0, 0];
    quaternion = initialQuaternion;

    % Main simulation loop
    for t = 1:length(timeVector)
        % Oscillating lever arm dynamics
        leverArmX = 0.05 * sin(2 * pi * t / 3600);
        leverArmY = 0.05 * cos(2 * pi * t / 3600);
        leverArmZ = 0.1 * sin(2 * pi * t / 1800);

        % Calculate gravitational force
        F_grav = G * massAsteroid * massObject / distance^2;

        % Compute torques for each axis
        torque = [leverArmX * F_grav, leverArmY * F_grav,
leverArmZ * F_grav];

        % Update angular velocity based on torque and inertia
        angularAcceleration = InertiaMatrix \ torque';
        angularVelocity = angularVelocity + angularAcceleration'
* deltaTime;

        % Update quaternion using the derived angular velocity
        omega_quat = [0, angularVelocity];
        quaternion_dot = 0.5 * quatmultiply(quaternion,
omega_quat);
        quaternion = quaternion + quaternion_dot * deltaTime;

        % Normalize quaternion to maintain unit length
```

```
        quaternion = quaternion / norm(quaternion);

        % Store quaternion data
        quaternionData(t, :) = quaternion;
    end

    % Convert to timeseries object
    ts = timeseries(quaternionData, timeVector, 'Name',
'Quaternion');

    % Save data for use in Simulink
    save('quaternionData.mat', 'ts', '-v7.3');
end
```

Code 11: Parking Orbit Asteroid Perturbation Simulation

```
function asteroid_perturbation_simulation()
    % Constants
    G = 6.67430e-11; % Gravitational constant
    deltaTime = 1; % Time step for the simulation in seconds
    totalSimulationTime = 36000; % Total simulation time in
seconds
    initialQuaternion = [1, 0, 0, 0]; % Initial quaternion (no
rotation)

    % Mass and distance parameters
    massAsteroid = 2.58945e10; % Mass of asteroid in kg
    massObject = 0.8; % Mass of the object/spacecraft in kg
    distance = 0.8; % Distance between asteroid and object in
meters

    % Inertia matrix components for a 6U CubeSat
    dimensions = [0.3, 0.2, 0.1]; % Dimensions in meters
    m = massObject; % Mass in kg
    Ixx = m/12 * (dimensions(2)^2 + dimensions(3)^2);
    Iyy = m/12 * (dimensions(1)^2 + dimensions(3)^2);
    Izz = m/12 * (dimensions(1)^2 + dimensions(2)^2);
    InertiaMatrix = diag([Ixx, Iyy, Izz]); % Diagonal inertia
matrix

    % Time vector
    timeVector = 0:deltaTime:totalSimulationTime;

    % Initialize quaternion data for plotting
    quaternionData = zeros(length(timeVector), 4); % Initialize
with zeros
```

```
    % Initial conditions
    angularVelocity = [0, 0, 0];
    quaternion = initialQuaternion;

    % Main simulation loop
    for t = 1:length(timeVector)
        % Oscillating lever arm dynamics
        leverArmX = 0.05 * sin(2 * pi * t / 3600);
        leverArmY = 0.05 * cos(2 * pi * t / 3600);
        leverArmZ = 0.1 * sin(2 * pi * t / 1800);

        % Calculate gravitational force
        F_grav = G * massAsteroid * massObject / distance^2;

        % Compute torques for each axis
        torque = [leverArmX * F_grav, leverArmY * F_grav,
leverArmZ * F_grav];

        % Update angular velocity based on torque and inertia
        angularAcceleration = InertiaMatrix \ torque';
        angularVelocity = angularVelocity + angularAcceleration'
* deltaTime;

        % Update quaternion using the derived angular velocity
        omega_quat = [0, angularVelocity];
        quaternion_dot = 0.5 * quatmultiply(quaternion,
omega_quat);
        quaternion = quaternion + quaternion_dot * deltaTime;

        % Normalize quaternion to maintain unit length
```

```
        quaternion = quaternion / norm(quaternion);

        % Store quaternion data
        quaternionData(t, :) = quaternion;
    end

    % Convert to timeseries object
    ts = timeseries(quaternionData, timeVector, 'Name',
'Quaternion');

    % Save data for use in Simulink
    save('quaternionData.mat', 'ts', '-v7.3');
end
```

Code 12: Slider-controlled Quaternion Rotation Visualizer

```
function quaternion_rotation_visualizer()
    % Create a figure and axis for visualization
    fig = figure('Name', 'Quaternion Rotation Visualizer', 'Units', 'normalized', 'Position', [0 0 1 0.9], 'Color', [0.1 0.1 0.1]);
    ax = axes('Parent', fig, 'DataAspectRatio', [1 1 1], 'Box', 'off', 'Color', [0 0 0]);
    hold(ax, 'on');
    grid(ax, 'on');
    xlabel(ax, 'X');
    ylabel(ax, 'Y');
    zlabel(ax, 'Z');
    sgtitle('                Quaternion Rotation Visualizer', 'Color', [1 1 1])  % Set title color to white
    view(3);
    % Remove the default axes box
    axis(ax, 'vis3d');
    set(ax, 'Visible', 'off');

    % Plot unit sphere for reference
    [X, Y, Z] = sphere;
    surf(ax, X, Y, Z, 'FaceAlpha', 0.2, 'EdgeColor', 'none');

    % Plot initial axes
    quiver3(0, 0, 0, 1, 0, 0, 'k', 'LineWidth', 2, 'AutoScale', 'off');
    quiver3(0, 0, 0, 0, 1, 0, 'k', 'LineWidth', 2, 'AutoScale', 'off');
    quiver3(0, 0, 0, 0, 0, 1, 'k', 'LineWidth', 2, 'AutoScale', 'off');
```

```
    % Define handles for rotated axes
    rotated_axes_i = quiver3(0, 0, 0, 1, 0, 0, 'r', 'LineWidth',
2, 'AutoScale', 'off');
    rotated_axes_j = quiver3(0, 0, 0, 0, 1, 0, 'g', 'LineWidth',
2, 'AutoScale', 'off');
    rotated_axes_k = quiver3(0, 0, 0, 0, 0, 1, 'b', 'LineWidth',
2, 'AutoScale', 'off');

    % Initialize quaternion and store in the figure's data
    quaternion = [1, 0, 0, 0]; % Unit quaternion
    guidata(fig, quaternion);

    % Setup slider controls and labels
    [slider_q0, slider_q1, slider_q2, slider_q3, az_slider,
el_slider] = setupControls();

    % Real-time Quaternion Display
    quaternion_display = uicontrol('Style', 'text', 'Position',
[150 105 200 30], ...
                                   'String',
sprintf('Quaternion: [1 0 0 0]'), 'ForegroundColor', [1 1 1],
'BackgroundColor', [0.5 0.5 0.5]);

    % Callback function to update rotation based on slider input
    function updateRotation(f, component, value, isReset)
        quaternion = guidata(f); % Retrieve the quaternion from
the figure's data
        quaternion(component) = value;

        if ~isReset
```

```
            % Normalize the quaternion to maintain it as a unit
quaternion
            quaternion = quaternion / norm(quaternion);
            % Update the slider positions to reflect the new
quaternion values
            set(slider_q0, 'Value', quaternion(1));
            set(slider_q1, 'Value', quaternion(2));
            set(slider_q2, 'Value', quaternion(3));
            set(slider_q3, 'Value', quaternion(4));
        end

        guidata(f, quaternion); % Store the updated quaternion
back into the figure's data

        % Update the display
        set(quaternion_display, 'String', sprintf('Quaternion:
[%.2f %.2f %.2f %.2f]', quaternion));

        % Update the rotation of the axes using the quaternion
        updateAxes(quaternion);
    end

    function updateView()
        azimuth = get(az_slider, 'Value');
        elevation = get(el_slider, 'Value');
        view(ax, [azimuth, elevation]);
    end

    function updateAxes(quaternion)
        i_rot = quatrotate(quaternion, [1 0 0]);
        j_rot = quatrotate(quaternion, [0 1 0]);
```

```
        k_rot = quatrotate(quaternion, [0 0 1]);
        set(rotated_axes_i, 'UData', i_rot(1), 'VData',
i_rot(2), 'WData', i_rot(3));
        set(rotated_axes_j, 'UData', j_rot(1), 'VData',
j_rot(2), 'WData', j_rot(3));
        set(rotated_axes_k, 'UData', k_rot(1), 'VData',
k_rot(2), 'WData', k_rot(3));
    end

    function [s0, s1, s2, s3, az, el] = setupControls()
    % Quaternion component sliders
    % Labels for the sliders
    uicontrol('Style', 'text', 'Position', [10 72 956 15],
'String', 'Scalar q_0',     'ForegroundColor', [1 1 1],
'BackgroundColor', [0.1 0.1 0.1]);
    uicontrol('Style', 'text', 'Position', [10 52 968 15],
'String', 'Vector q_1 (i)', 'ForegroundColor', [1 1 1],
'BackgroundColor', [0.1 0.1 0.1]);
    uicontrol('Style', 'text', 'Position', [10 32 968 15],
'String', 'Vector q_2 (j)', 'ForegroundColor', [1 1 1],
'BackgroundColor', [0.1 0.1 0.1]);
    uicontrol('Style', 'text', 'Position', [10 12 972 15],
'String', 'Vector q_3 (k)', 'ForegroundColor', [1 1 1],
'BackgroundColor', [0.1 0.1 0.1]);

    % Sliders for adjusting the quaternion components
    s0 = uicontrol('Style', 'slider', 'Min', -1, 'Max', 1,
'Value', 1, ...
                   'Position', [10 70 450 20],
'BackgroundColor', [0.8 0.8 0.8], ...
```

```
                    'Callback', @(src, event) updateRotation(fig,
1, src.Value, false));
    s1 = uicontrol('Style', 'slider', 'Min', -1, 'Max', 1,
'Value', 0, ...
                    'Position', [10 50 450 20],
'BackgroundColor', [1 0.5 0.5], ...
                    'Callback', @(src, event) updateRotation(fig,
2, src.Value, false));
    s2 = uicontrol('Style', 'slider', 'Min', -1, 'Max', 1,
'Value', 0, ...
                    'Position', [10 30 450 20],
'BackgroundColor', [0.5 1 0.5], ...
                    'Callback', @(src, event) updateRotation(fig,
3, src.Value, false));
    s3 = uicontrol('Style', 'slider', 'Min', -1, 'Max', 1,
'Value', 0, ...
                    'Position', [10 10 450 20],
'BackgroundColor', [0.5 0.5 1], ...
                    'Callback', @(src, event) updateRotation(fig,
4, src.Value, false));

    % Azimuth slider
    az = uicontrol('Style', 'slider', 'Min', -180, 'Max', 180,
'Value', -45, ...
                    'Position', [600 50 300 20],
'BackgroundColor', [0.6 0.6 0.6], ...
                    'Callback', @(src, event) updateView());

    % Elevation slider
    el = uicontrol('Style', 'slider', 'Min', -90, 'Max', 90,
'Value', 30, ...
```

```
                    'Position', [600 10 300 20],
'BackgroundColor', [0.6 0.6 0.6], ...
                    'Callback', @(src, event) updateView());
    end
end
```

Code 13: IMU-controlled Quaternion Rotation Visualizer

```
function quaternion_rotation_visualizer()
    % Create the visualization figure and axes
    fig = figure('Name', 'Quaternion Rotation Visualizer',
'Units', 'normalized', 'Position', [0 0 1 0.9], 'Color', [0.1
0.1 0.1]);
    ax = axes('Parent', fig, 'DataAspectRatio', [1 1 1], 'Box',
'off', 'Color', [0 0 0]);
    hold(ax, 'on');
    grid(ax, 'on');
    xlabel(ax, 'X');
    ylabel(ax, 'Y');
    zlabel(ax, 'Z');
    sgtitle('Quaternion Rotation Visualizer', 'Color', [1 1 1]);
    view(3);
    axis(ax, 'vis3d');
    set(ax, 'Visible', 'off');

    % Plot the unit sphere for reference
    [X, Y, Z] = sphere;
    surf(ax, X, Y, Z, 'FaceAlpha', 0.2, 'EdgeColor', 'none');

    % Plot initial axes
    rotated_axes_i = quiver3(0, 0, 0, 1, 0, 0, 'r', 'LineWidth',
2, 'AutoScale', 'off');
    rotated_axes_j = quiver3(0, 0, 0, 0, 1, 0, 'g', 'LineWidth',
2, 'AutoScale', 'off');
    rotated_axes_k = quiver3(0, 0, 0, 0, 0, 1, 'b', 'LineWidth',
2, 'AutoScale', 'off');
```

```
    % Setup serial port
    s = serialport("COM3", 115200);
    configureTerminator(s, 'LF');
    flush(s);

    % Read and process data from serial port
    while isvalid(fig)  % Continue until the figure is closed
        line = readline(s);
        if line == ""
            continue;
        end

        parsedData = str2double(strsplit(line, ','));
        if length(parsedData) >= 4 && ~any(isnan(parsedData))
            quaternion = parsedData(1:4) /
norm(parsedData(1:4));
            guidata(fig, quaternion);
            updateAxes(quaternion);
            drawnow;
        else
            fprintf('Invalid data received: %s\n', line);
        end
    end

    % Close serial port on exit
    delete(s);
    clear s;

    function updateAxes(quaternion)
        i_rot = quatrotate(quaternion, [1 0 0]);
        j_rot = quatrotate(quaternion, [0 1 0]);
```

```
        k_rot = quatrotate(quaternion, [0 0 1]);
        set(rotated_axes_i, 'UData', i_rot(1), 'VData',
i_rot(2), 'WData', i_rot(3));
        set(rotated_axes_j, 'UData', j_rot(1), 'VData',
j_rot(2), 'WData', j_rot(3));
        set(rotated_axes_k, 'UData', k_rot(1), 'VData',
k_rot(2), 'WData', k_rot(3));
    end
end
```

Code 14: Arduino-IMU MATLAB Input

```
clc; close all; clear;

% Open serial port with the correct baud rate
s = serialport("COM3", 115200);

% Create arrays to store data
data_w = [];
data_x = [];
data_y = [];
data_z = [];
timestamps = [];

% Create a single figure for all subplots
figure;
ax_w = subplot(2, 2, 1);
hLine_w = animatedline(ax_w);
title('Quaternion w-component');
ylabel('w-value');

ax_x = subplot(2, 2, 2);
hLine_x = animatedline(ax_x);
title('Quaternion x-component');
ylabel('x-value');

ax_y = subplot(2, 2, 3);
hLine_y = animatedline(ax_y);
title('Quaternion y-component');
ylabel('y-value');

ax_z = subplot(2, 2, 4);
```

```
hLine_z = animatedline(ax_z);
title('Quaternion z-component');
ylabel('z-value');

% Initialize a counter
i = 1;

% Read and plot live data
while true % loop until the figure is closed
    % Read data from serial port
    line = readline(s);
    if line == ""
        continue;
    end

    % Parse data
    parsedData = str2double(strsplit(line, ','));

    % Check for invalid data
    if any(isnan(parsedData))
        fprintf('Invalid data received: %s\n', line);
        continue;
    end

    % Record timestamp
    timestamps(i) = i - 1;

    % Append new data points to arrays
    addpoints(hLine_w, timestamps(i), parsedData(1));
    addpoints(hLine_x, timestamps(i), parsedData(2));
    addpoints(hLine_y, timestamps(i), parsedData(3));
```

```
    addpoints(hLine_z, timestamps(i), parsedData(4));

    drawnow;

    % Increment counter
    i = i + 1;
end

% Close serial port
fclose(s);
delete(s);
```

Code 15: Spacecraft TB-Angles

```
function tb_angles()
    % Given Inertial Matrix
    I = [10, 0, -0.1;
         0, 5, 0;
         -0.1, 0, 10];

    % Time Span for 30 seconds
    tspan = [0 30];

    % Case 1
    y0_1 = [0; 0; 0; 0; 0; 10];
    % ODE 45 for Case 1
    [t1, y1] = ode45(@(t,y) spacecraft_dynamics(t, y, I, [0; 0;
0]), tspan, y0_1);

    % Case 2
    y0_2 = [pi/4; pi/3; 0; -1; 0; 10];
    % ODE 45 for Case 2
    [t2, y2] = ode45(@(t,y) spacecraft_dynamics(t, y, I, [0; 1;
0]), tspan, y0_2);

    % Plot results
    % Case 1
    subplot(2,1,1);
    plot(t1,y1(:,1:6));
    title('T-B Angles for Case 1');
    xlabel('Time (s)');
    ylabel('Angles (rad)');
    legend('Phi', 'Theta', 'Psi');
```

```
    % Case 2
    subplot(2,1,2);
    plot(t2,y2(:,1:6));
    title('T-B Angles for Case 2');
    xlabel('Time (s)');
    ylabel('Angles (rad)');
    legend('Phi', 'Theta', 'Psi');
end

function dydt = spacecraft_dynamics(t, y, I, M_ext)
    % Extracting states
    phi = y(1);
    theta = y(2);
    psi = y(3);
    p = y(4);
    q = y(5);
    r = y(6);
    M = M_ext;  % Use the passed external moment directly from
cases
    % Angles
    d_angles = [p + q*sin(phi)*tan(theta) +
r*cos(phi)*tan(theta);
                q*cos(phi) - r*sin(phi);
                q*sin(phi)/cos(theta) + r*cos(phi)/cos(theta)];
    % Rates
    d_rates = inv(I) * (M - cross([p; q; r], I*[p; q; r]));

    % Time derivative
    dydt = [d_angles; d_rates];
end
```

Code 16: Super-Twisting Sliding Mode Equations

```
%-------------------------------------------------------------
% GEOMETRY CONFIGURATION ANALYSIS
%-------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq.1 Body Frame Rotation
function BodyFrameRotation
% Define the angle beta in radians
% beta = ...; % replace with the actual value of beta
% Define the rotation matrix
R = [cos(beta) sin(beta) 0;
    -sin(beta) cos(beta) 0;
    0 0 1];

% Define the original vector
v = [x; y; z]; % replace x, y, z with actual values

% Apply the rotation
v_prime = R * v;

% Extract the new coordinates
x_prime = v_prime(1);
y_prime = v_prime(2);
z_prime = v_prime(3);
% v_prime now contains the x', y', z' values
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 2 New Frame Rotation
function NewFrameRotation
% Define the angle alpha in readians
```

```
% alpha = ...; % replace with the actual value of alpha

% Define the rotation matrix
R_prime = [cos(-alpha) 0 -sin(-alpha);
    0 1 0;
    sin(-alpha) 0 cos(-alpha)];

% Apply the rotation
v_double_prime = R_prime * v_prime;

% Extract the new coordinates
x_double_prime = v_double_prime(1);
y_double_prime = v_double_prime(2);
z_double_prime = v_double_prime(3);
% v_double_prime now contains the x'', y'', z'' values
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 3 Mapping Rotational Relation
function MappingRotationalRelation
% Mapping Relation derived from Eq. 1 and Eq. 2
R_mapping = R * R_prime * v;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 5 Inertial Rotational Frame
function InertialRotationalFrame
% Preallocate the array for beta_i values
beta_i = zeros(1, 4);

% Calculate the beta_i values
for i = 1:4
    beta_i(i) = beta + 0.5 * (i - 1) * pi;
```

```
end

% Calculate the rotation matrix R_i for each beta_i
% (assuming you need to do this for each beta_i)
R_rotation_i = cell(1, 4); % Use cell array to store matrices
since they can vary

for i = 1:4
    R_i = [cos(beta_i(i)), sin(beta_i(i)), 0;
        -sin(beta_i(i)), cos(beta_i(i)), 0;
        0,               0,              1];

    % The given multiplication R_prime * [R_i | [0; 0; 1]]
    R_rotation_i{i} = R_prime * [R_i, [0; 0; 1]];
end
% Now R_i{1}, R_i{2}, R_i{3}, R_i{4} contain the four rotation
matrices
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 4 General Mapping Relation
function GeneralMappingRelation
% Mapping relation betwen x y z and x_i y_i z_i
v_i = R_rotation_i * v;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------------------------------------------------
% ATTITUDE DYNAMICS
%-------------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 6 Quaternion Scalar Kinematics
```

```
function QuaternionScalarKinematics
% q: The current quaternion [q0, q1, q2, q3]
% omega: The angular velocity vector [omega_x, omega_y, omega_z]

% Extract the scalar and vector parts of the quaternion
q0 = q(1);
q_vec = q(2:4);

% Quaternion kinematic equation
q0_dot = -0.5 * q_vec' * omega; % Transpose q_vec to make it a
column vector
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 7 Quaternion Vector Kinematics
function QuaternionVectorKinematics
% Derivative of the quaternion
q_dot = [q0_dot; -0.5 * (q0 * omega + cross(q_vec, omega))];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 8 Compact Quaternion Matrix
function CompactQuaternionMatrix
% Calculate the quaternion derivative
Q_dot = 0.5 * E_Q * omega;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 9 Eulerian-based Quaternion
function EulerianBasedQuaternion
% Define the matrix E(Q)
E_Q = [ -Q(2), -Q(3), -Q(4);
    Q(1), -Q(4),  Q(3);
    Q(4),  Q(1), -Q(2);
```

```
    -Q(3),  Q(2),  Q(1)];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 10 Cross Product Matrix
function CrossProductMatrix
% Define the cross product matrix a_x
a_x = [ 0,   -a(3),  a(2);
        a(3),  0,   -a(1);
       -a(2),  a(1),  0 ];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 11 Redundant Kinetic Equations
function RedundantKineticEquations
% Define the relative rotational speed of the i-th reaction
wheel
Omega_i = 0;
% Define the unit vector along the x_i axis
ex_i = [1; 0; 0];
% Define the rotation matrix Ri for the i-th wheel
Ri = eye(3);
% Compute the angular velocity of the i-th reaction wheel
% with respect to x_i y_i z_i
omega_w_i = Ri * omega + Omega_i * ex_i;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 12 Angular Momentum Body Frame
function AngularMomentumBodyFrame
    % angularMomentum calculates the angular momentum of a rigid
body.
    % J is the inertia matrix, a 3x3 matrix.
    % omega is the angular velocity vector, a 3x1 vector.
```

```
    % Calculate the angular momentum
    H = J * omega;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 13 Circle Plate Angular Momentum
function CirclePlateAngularMomentum
    % diagonalInertia creates a diagonal inertia matrix for a
rigid body.
    % Jmi, Ji, Jl are the diagonal elements of the inertia
matrix.

    % Construct the diagonal inertia matrix
    Jw = diag([Jmi, Ji, Jl]);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 14 Angular Momentum Relative Frame
function AngularMomentumRelativeFrame
    % Compute the angular momentum component Hwi
    Hwi = Jwi * R_omega + Jwi * (Omega * e_x);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 15 Angular Momentum Mapping Relation
function AngularMomentumMappingRelation
    % Compute Hwi using the given equations
    Hwi = Jwi * R_omega + Jwi * (Omega * e_x);

    % Transform Hwi to body-fixed frame
    HwiB = Ri' * Hwi;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Eq. 18 Equivalent Moment of Inertia Matrix
function EquivalentMomentofInertiaMatrix
    % Initialize the equivalent inertia matrix
    Jeq = J;

    % Compute contributions from each component
    for i = 1:4
        Ri = R{i};
        Jwi = Jw{i};
        % Compute and accumulate the transformed inertia
contributions
        Jeq = Jeq + Ri' * Jwi * Ri;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 19 Force Distribution Matrix (FDM)
function ForceDistributionMatrix
    % Define the rotation matrix using cosine and sine functions
    R_prime = [
        cos(alpha) * cos(beta), -cos(alpha) * sin(beta), -
cos(alpha) * cos(beta), cos(alpha) * sin(beta);
        cos(alpha) * sin(beta),  cos(alpha) * cos(beta), -
cos(alpha) * sin(beta), -cos(alpha) * cos(beta);
        sin(alpha),              sin(alpha),
sin(alpha),              sin(alpha)
    ];
end
%-------------------------------------------------------------------
% ACTUATOR ANALYSIS
%-------------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Eq. 26 Torque Rotation Matrix
function [tcx, tcy, tcz] = calculate_actuator_forces(tw1, tw2,
tw3, tw4, a, b)
    % Given matrix
    R = [cos(a)*cos(b), -cos(a)*sin(b), -cos(a)*cos(b),
cos(a)*sin(b);
         cos(a)*sin(b), cos(a)*cos(b), -cos(a)*sin(b), -
cos(a)*cos(b);
         sin(a), sin(a), sin(a), sin(a)];

    % Calculate τcx, τcy, and τcz
    tw = [tw1; tw2; tw3; tw4];
    tc = R * tw;

    % Extract individual components
    tcx = tc(1);
    tcy = tc(2);
    tcz = tc(3);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 27 FDM Geometric Constraint Definition 1
function [a,b] = alpha_and_beta_calculations()
    b = pi/4;
    a = asind(sqrt(3)/3);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 29 Geometrically Constrained Force Distribution Model
function A = FDMMatrix()
    A = sqrt(3)/3*[1, -1, -1, 1; 1, 1, -1, 1; 1, 1, 1, 1];
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Eq. 30 Static Optimization
function minT = mulated(tw1,tw2,tw3,tw4)
    twi = [tw1,tw2,tw3,tw4];
    minT =  sum(twi^2);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 31 Static Optimization Subjections
function [g1, g2, g3, g4] = compute_g1(tw1, tw2, tw3, tw4, tcx,
tcy, tcz)
    % Compute g1, g2, and g3 using the given formulas
    g1 = (sqrt(3) / 3) * (tw1 - tw2 - tw3 + tw4) - tcx;
    g2 = (sqrt(3) / 3) * (tw1 + tw2 - tw3 - tw4) - tcy;
    g3 = (sqrt(3) / 3) * (tw1 - tw2 + tw3 - tw4) - tcz;

    % Check if g1, g2, and g3 are all zero
    if g1 == 0 && g2 == 0 && g3 == 0
        g4 = true;
    else
        g4 = false;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 32 Lagrangian Defintion
function L = computeL(T, lambda1,lambda2,lambda3, g1,g2,g3)
    L = T + lambda1*g1 +lambda2*g2 +lambda3*g3;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 35 Static Optimization First-Order Implied Condition
function [tw1_star, tw2_star, tw3_star, tw4_star, lambda1_star,
lambda2_star, lambda3_star,check] = solveOptimalVariables()
    % Define symbolic variables
```

```
    syms tw1 tw2 tw3 tw4 lambda1 lambda2 lambda3
    % Define the equations
    eq1 = 2*tw1 - sqrt(3/lambda1) + sqrt(3/lambda2) +
sqrt(3/lambda3) == 0;
    eq2 = 2*tw2 - sqrt(3/lambda1) + sqrt(3/lambda2) +
sqrt(3/lambda3) == 0;
    eq3 = 2*tw3 - sqrt(3/lambda1) + sqrt(3/lambda2) +
sqrt(3/lambda3) == 0;
    eq4 = 2*tw4 - sqrt(3/lambda1) + sqrt(3/lambda2) +
sqrt(3/lambda3) == 0;

    % Solve the system of equations
    solutions = solve([eq1, eq2, eq3, eq4], [w1, w2, w3, w4,
lambda1, lambda2, lambda3]);

    % Extract the optimal values
    tw1_star = solutions.w1;
    tw2_star = solutions.w2;
    tw3_star = solutions.w3;
    tw4_star = solutions.w4;
    lambda1_star = solutions.lambda1;
    lambda2_star = solutions.lambda2;
    lambda3_star = solutions.lambda3;

    if(tw1_star-tw2_star+tw3_star-tw4_star == 0)
        check = true;
    else
        check = false;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Eq. 36 Augmented Distribution Matrix
function[tw1_star, tw2_star, tw3_star, tw4_star] =
inverse_mapping(tcx,tcy,tcz)
    inverse_map_matrix = 1/4*[sqrt(3),sqrt(3),sqrt(3),1;
                              -sqrt(3),sqrt(3),sqrt(3),-1;
                              -sqrt(3),-sqrt(3),sqrt(3),1;
                              sqrt(3),-sqrt(3),sqrt(3),-1;];
    tc = [tcx;tcy;tcz;0];
    tw_star = inverse_map_matrix * tc;

    tw1_star = tw_star(1)
    tw2_star = tw_star(2)
    tw3_star = tw_star(3)
    tw4_star = tw_star(4)
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 38 Linear Dynamic Reaction Torque
function[j_omega1_dot] = omega_solver(a,b,omega,v_in)
    j_omega1_dot = -a*omega+b*v_in;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Eq. 39 Control Input Voltage
function[v_in] = velocity_solver(b,twi,a,omega)
    v_in = 1/b*(twi+a+omega);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

12. Assembly/Sub-Assembly Drawings

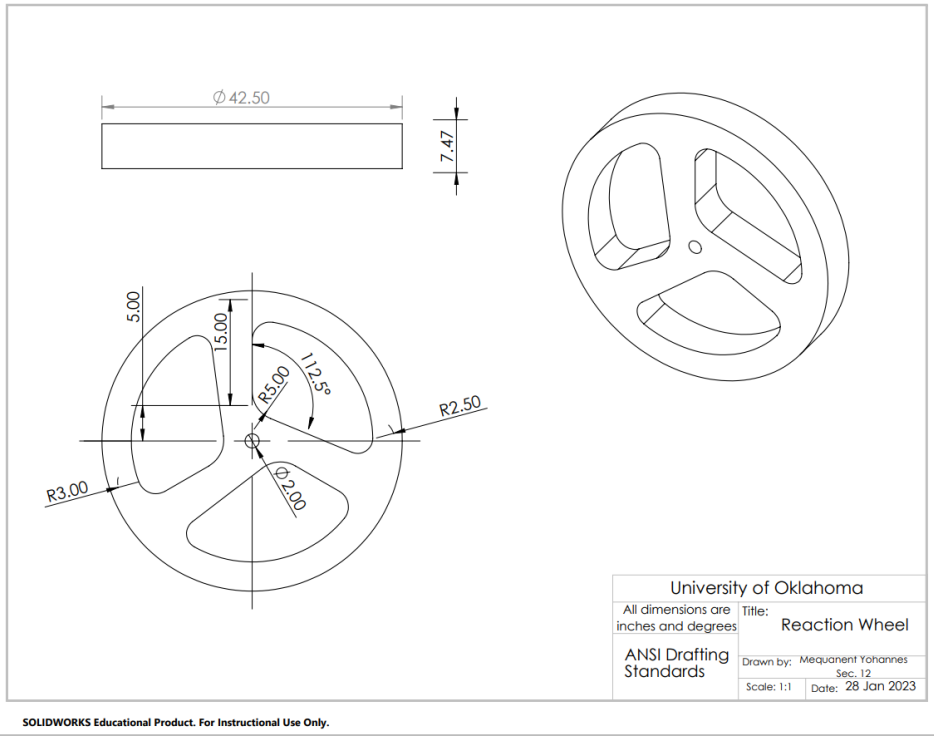


Figure 39: Reaction Wheels Drawing

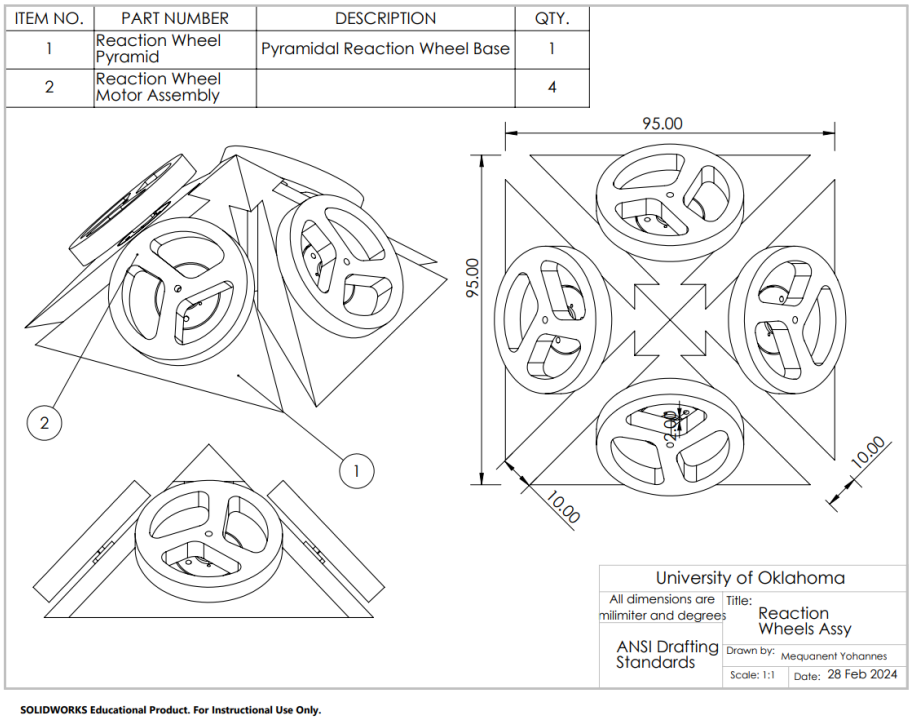
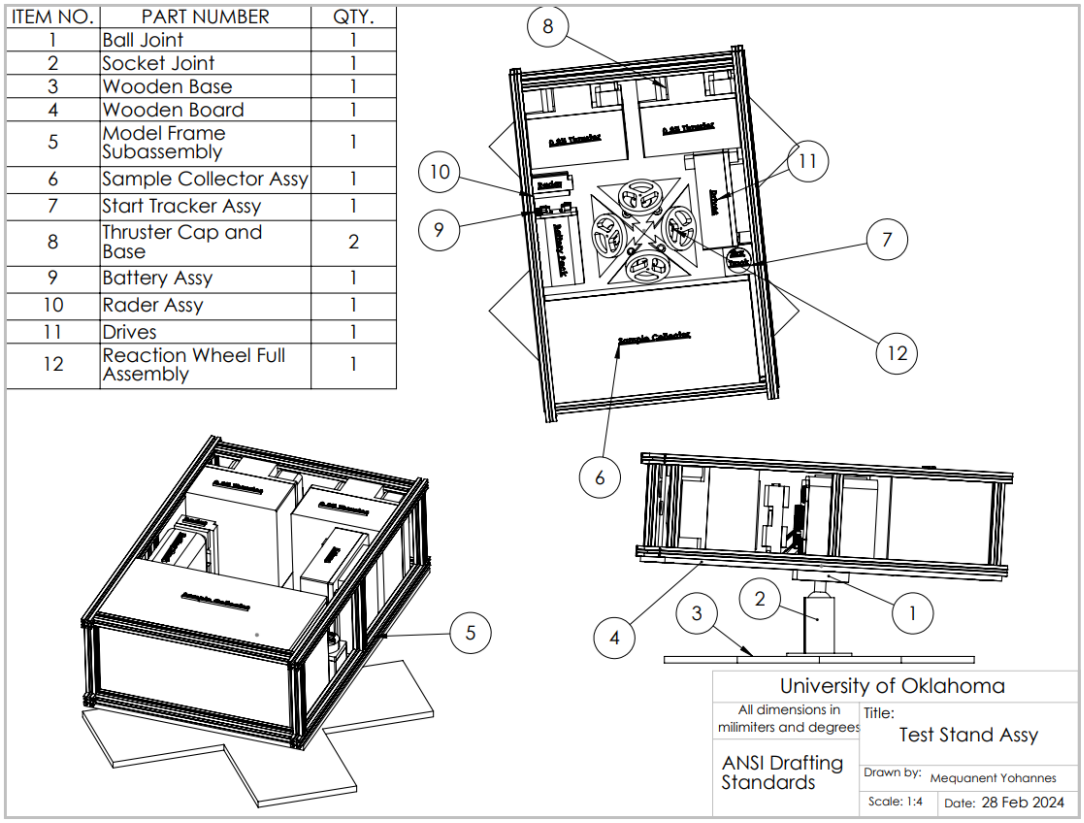


Figure 40: Reaction Wheels Assembly

ITEM NO.	PART NUMBER	QTY.
1	Ball Joint	1
2	Socket Joint	1
3	Wooden Base	1
4	Wooden Board	1
5	Model Frame Subassembly	1
6	Sample Collector Assy	1
7	Start Tracker Assy	1
8	Thruster Cap and Base	2
9	Battery Assy	1
10	Rader Assy	1
11	Drives	1
12	Reaction Wheel Full Assembly	1



SOLIDWORKS Educational Product. For Instructional Use Only.

Figure 41: Probe Assembly with Test Stand

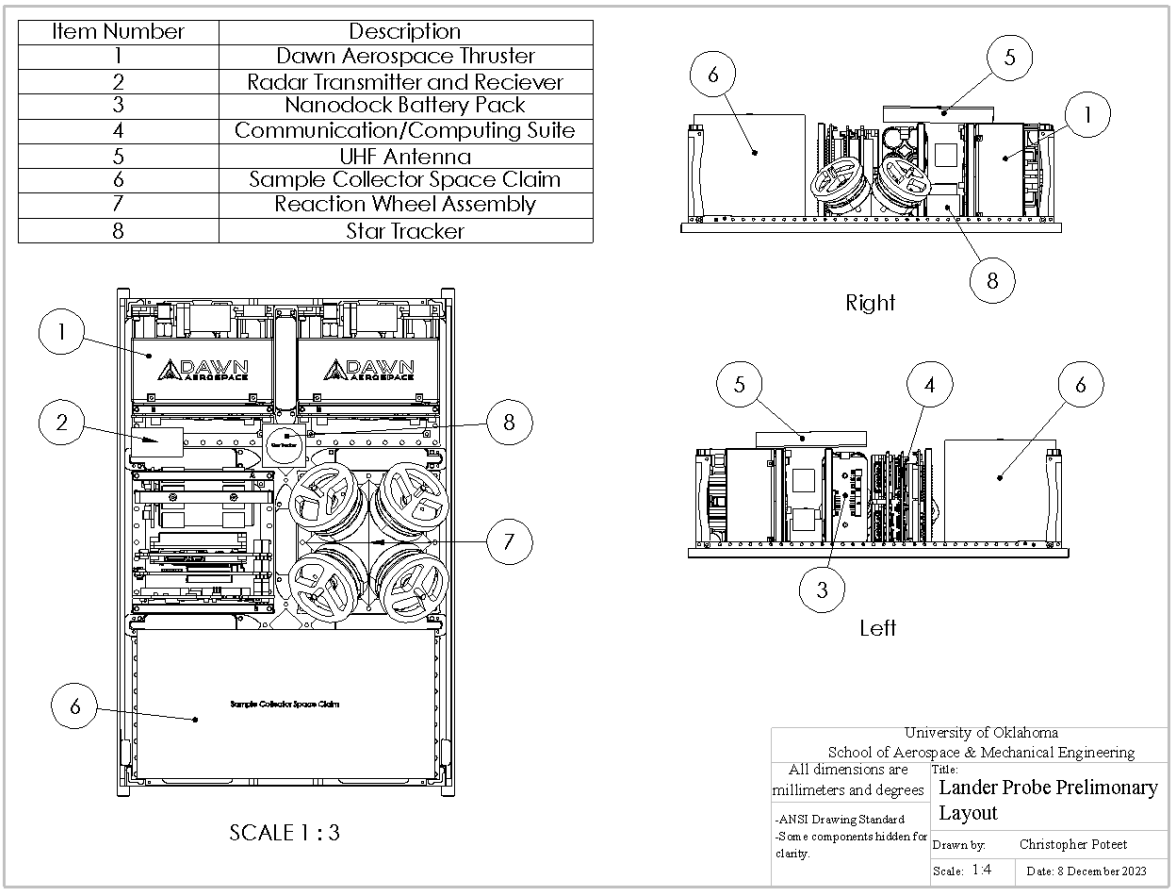


Figure 42: Probe Assembly

13. Facilities/Materials/Parts List

Table 10: Facilities/Materials/Parts Utilization

Facilities/Materials/Parts	**Information**
AME Machine Shop (Facility)	Free service provided by school
Innovation Hub (Facility)	Free service provided by school
Home 3D Printer (Facility)	Service provided by Chris Poteet
Maxon Motors (Part)	Swiss Company (https://www.maxongroup.us/maxon/view/catalog/)
ESCON Controllers (Part)	Swiss Company (https://www.maxongroup.us/maxon/view/catalog/)
Adafruit IMU (Part)	IMU that can handle quaternions obtained on Amazon
BOJACK Breadboard (Part)	Obtained on Amazon
Adapters for Motors to Controller(Parts)	Swiss Company (https://www.maxongroup.us/maxon/view/catalog/)
Ovonic Batteries and Charger (Parts)	Obtained on Amazon
Bluetooth Adapter (Parts)	Obtained on Amazon
¼ inch steel sheet (Material)	Provided by AME Machine Shop
20 gage steel sheet (Material)	Provided by Innovation Hub

14. Budget Statement

Table 11: Total Time/Budget Spent

Total Time Spent on Project	610 Hours
Total Budget Spent	$1,336.54

Table 12: Budget Categorial Allocation

Component Category	**Budget Allocated**
Motors	$402.50
Tools/Equipment	$90.48
Controllers	$582.50
Electronics	$261.06
Unused	$163.46

Table 13: Cost Allocation for Individual Items

Item	**Cost**	**Time**
Maxon EC 20 flat Ø20 mm, brushless, 3 Watt, sensorless	$402.50	2 weeks for shipping
BNO055 9 DOF Absolute Orientation IMU Fusion Breakout Board	$34.95	2 days for shipping
BOJACK 3 Values 130 Pcs Solderless Breadboard	$11.99	2 days for shipping
ESCON Module 24/2 Controller	$582.50	2 weeks for shipping
Adapter 4-pole flexprint connector to 4-pole screw terminal	$98.15	2 weeks for shipping
Reaction Wheels(CNC)	No Cost	3 weeks to get CNC
Reaction Wheel Base (Waterjet)	No Cost	2 days to waterjet and formed into shape
Beslands Digital Electronic Display Micrometer	$40.49	2 days for shipping
OVONIC 4s Lipo Battery 100C 1550mAh 14.8V Lipo Battery	$42.99	2 days for shipping

OVONIC Mate1 Lipo Battery Charger 1s-6s 100W 10A Smart RC Battery Chargers	$62.99	2 days for shipping
DSD TECH HC-05 Bluetooth Serial Pass-through Module	$9.99	2 days for shipping
Drill Bits for CNC	$42.28	1 week for shipping

15. Time and Personnel Management

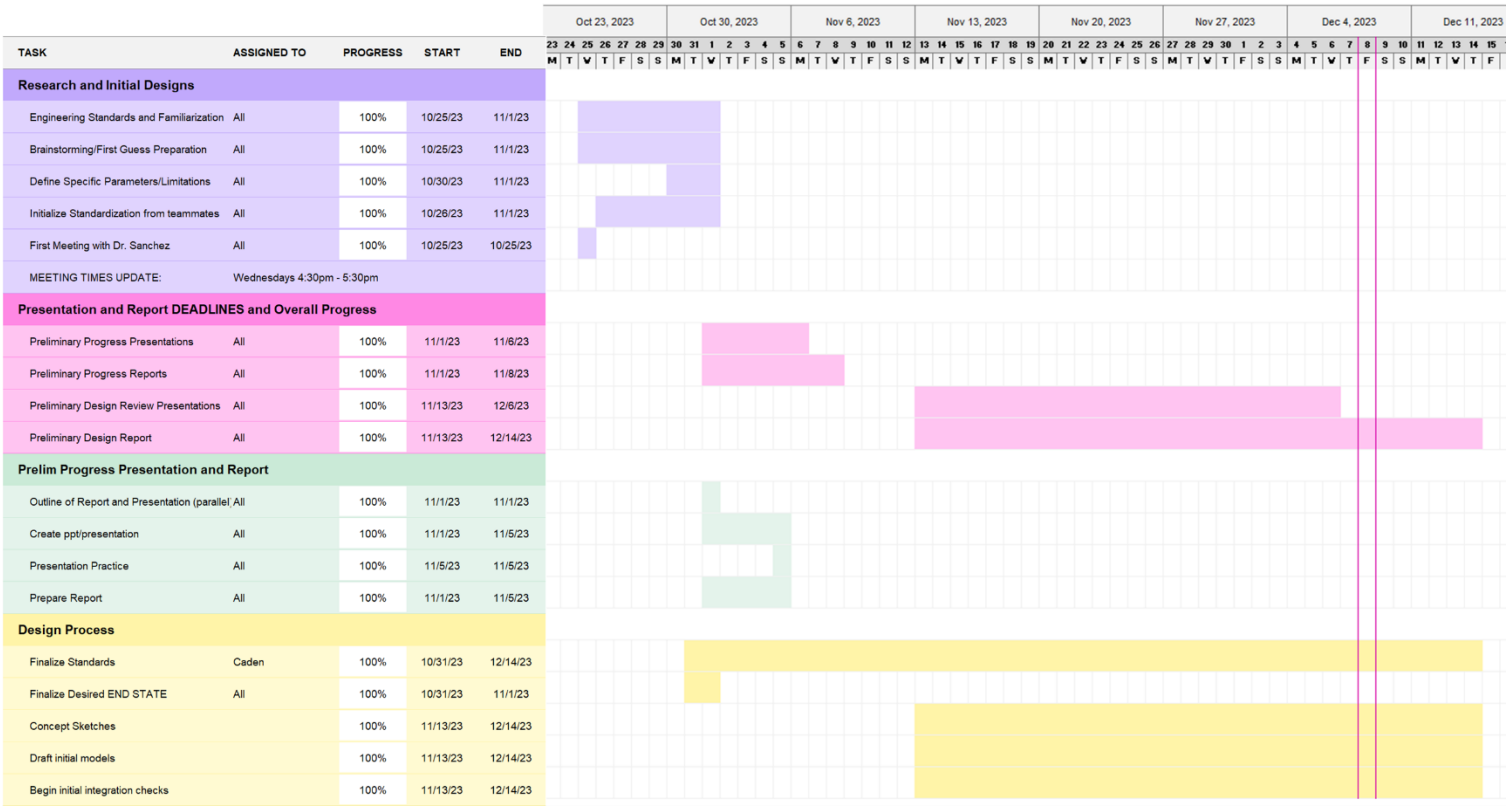
The individual responsibilities are divided as follows:

Caden Matthews - Team Coordinator/Control Systems Lead

Chris Poteet - Manufacturing and Test Lead

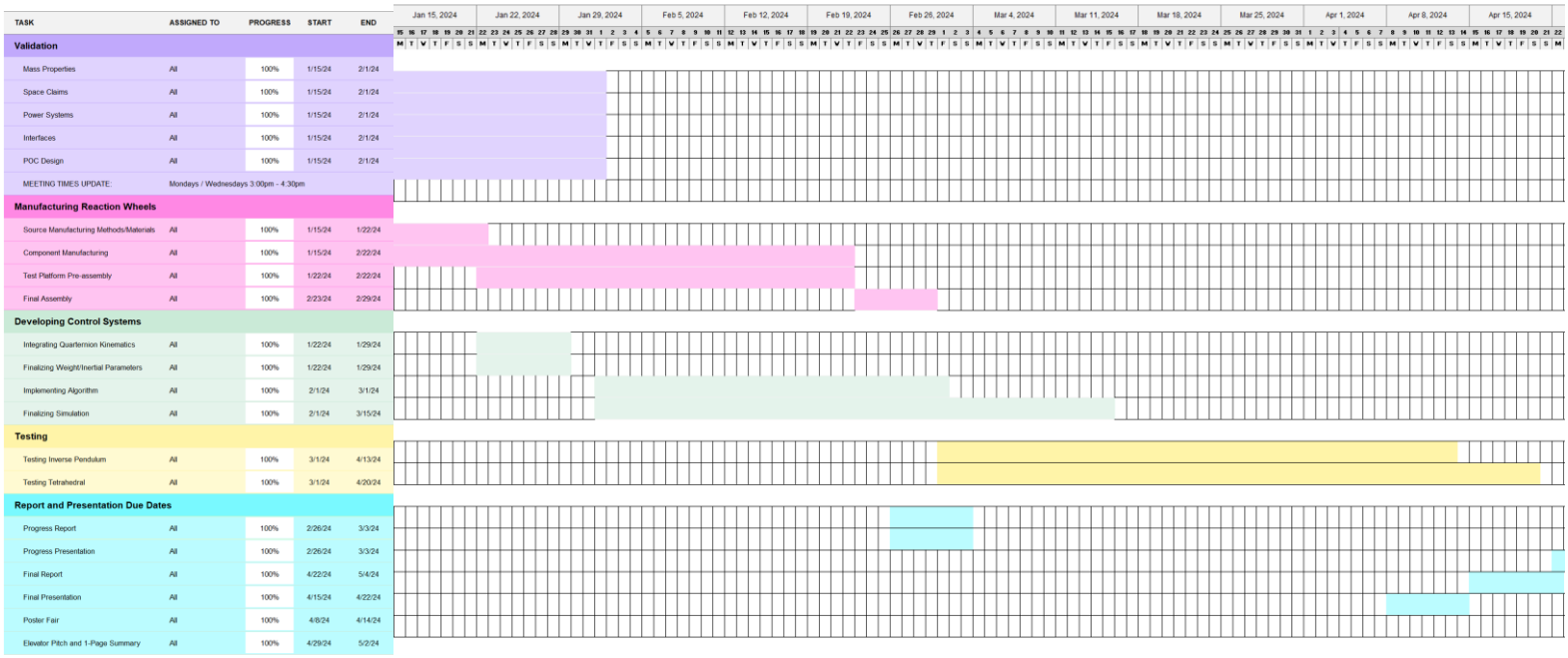
Colter Sartin - Backup Team Coordinator/Propulsion and Power Lead

Mequanent Yohannes - CAD Lead



TASK	ASSIGNED TO	PROGRESS	START	END
Research and Initial Designs				
Engineering Standards and Familiarization	All	100%	10/25/23	11/1/23
Brainstorming/First Guess Preparation	All	100%	10/25/23	11/1/23
Define Specific Parameters/Limitations	All	100%	10/30/23	11/1/23
Initialize Standardization from teammates	All	100%	10/26/23	11/1/23
First Meeting with Dr. Sanchez	All	100%	10/25/23	10/25/23
MEETING TIMES UPDATE:	Wednesdays 4:30pm - 5:30pm			
Presentation and Report DEADLINES and Overall Progress				
Preliminary Progress Presentations	All	100%	11/1/23	11/8/23
Preliminary Progress Reports	All	100%	11/1/23	11/8/23
Preliminary Design Review Presentations	All	100%	11/13/23	12/6/23
Preliminary Design Report	All	100%	11/13/23	12/14/23
Prelim Progress Presentation and Report				
Outline of Report and Presentation (parallel	All	100%	11/1/23	11/1/23
Create ppt/presentation	All	100%	11/1/23	11/5/23
Presentation Practice	All	100%	11/5/23	11/5/23
Prepare Report	All	100%	11/1/23	11/5/23
Design Process				
Finalize Standards	Caden	100%	10/31/23	12/14/23
Finalize Desired END STATE	All	100%	10/31/23	11/1/23
Concept Sketches		100%	11/13/23	12/14/23
Draft initial models		100%	11/13/23	12/14/23
Begin initial integration checks		100%	11/13/23	12/14/23

Figure 43: Fall 2023 Probe Team Gantt Chart



TASK	ASSIGNED TO	PROGRESS	START	END
Validation				
Mass Properties	All	100%	1/15/24	2/1/24
Space Claims	All	100%	1/15/24	2/1/24
Power Systems	All	100%	1/15/24	2/1/24
Interfaces	All	100%	1/15/24	2/1/24
POC Design	All	100%	1/15/24	2/1/24
MEETING TIMES UPDATE:	Mondays / Wednesdays 3:00pm - 4:30pm			
Manufacturing Reaction Wheels				
Source Manufacturing Methods/Materials	All	100%	1/15/24	1/22/24
Component Manufacturing	All	100%	1/15/24	2/22/24
Test Platform Pre-assembly	All	100%	1/22/24	2/22/24
Final Assembly	All	100%	2/23/24	2/29/24
Developing Control Systems				
Integrating Quarternion Kinematics	All	100%	1/22/24	1/29/24
Finalizing Weight/Inertial Parameters	All	100%	1/22/24	1/29/24
Implementing Algorithm	All	100%	2/1/24	3/1/24
Finalizing Simulation	All	100%	2/1/24	3/15/24
Testing				
Testing Inverse Pendulum	All	100%	3/1/24	4/13/24
Testing Tetrahedral	All	100%	3/1/24	4/20/24
Report and Presentation Due Dates				
Progress Report	All	100%	2/26/24	3/3/24
Progress Presentation	All	100%	2/26/24	3/3/24
Final Report	All	100%	4/22/24	5/4/24
Final Presentation	All	100%	4/15/24	4/22/24
Poster Fair	All	100%	4/8/24	4/14/24
Elevator Pitch and 1-Page Summary	All	100%	4/29/24	5/2/24

Figure 44: Spring 2024 Probe Team Gantt Chart