TU DUBLIN BLANCHARDSTOWN

SCHOOL OF INFORMATICS AND ENGINEERING

DEVELOPMENT OF AN OPTIMIZED ATTITUDE CORRECTION & CONTROL SYSTEM FOR A 1U CUBESAT

by

Ethan Lawlor

A report submitted in partial fulfilment of the

requirements for the degree

BACHELOR OF ENGINEERING (HONOURS) IN MECHATRONICS

SUPERVISOR:
SUBMISSION DATE:

ABSTRACT

This report presents the end-to-end development of a reaction-wheel attitude-determination-and-control system for a one-unit CubeSat. The work addresses the dual challenge faced by small spacecraft, rapid detumbling after deployment and precise pointing during routine operations, using only low-cost, commercial-off-the-shelf hardware and firmware written from first principles .

A dynamic model of the inverted-pendulum system is derived with Lagrangian mechanics and verified in simulation to guide controller design . Real-time attitude estimates fuse accelerometer and gyroscope data first with a complementary filter and later with a Kalman filter to reduce noise and drift while remaining computationally light . Stability is achieved with a cascaded PID architecture in which an inner wheel-velocity loop feeds an outer attitude loop, a structure selected for its ability to balance fast actuator dynamics against slower body motions .

The report documents the electrical and mechanical builds, software flow charts, and laboratory test methodology, detailing how iterative tuning, on-board data logging, sensor filtering and a watchdog routine were used to refine performance and prevent controller saturation . All design drawings and software codes are contained in the appendices to lower the entry barrier for future attitude correction projects.

ACKNOWLEDGEMENTS

I would like to thank my lecturer, Niall Bell, for his guidance and support throughout this project. His expertise and feedback were invaluable in shaping my work.

I also appreciate the encouragement and support from my friends and family, whose belief in me kept me motivated during challenging moments.

Lastly, thank you to everyone who contributed to my education. Your help made this journey possible.

DECLARATION

The work submitted in this report is the results of the candidate's own investigations and has not been submitted for any other award. Where use has been made of the work of other people it has been fully acknowledged and referenced.

Ethan Lawlor

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

1 U	One-Unit CubeSat standard (10 × 10 × 10 cm)
ADCS	Attitude Determination and Control System
ADS	Attitude Determination System
COTS	Commercial Off-The-Shelf
DC	Direct Current (brushless DC motor)
DOF	Degrees Of Freedom (e.g., 6-DOF sensor)
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPS	Electrical Power System
GNC	Guidance, Navigation and Control
GPS	Global Positioning System
GUI	Graphical User Interface
IMU	Inertial Measurement Unit
I²C / I2C	Inter-Integrated Circuit serial bus
ISS	International Space Station
LQR	Linear-Quadratic Regulator
MCU	Microcontroller Unit (used for on-board processing)
MPPT	Maximum Power Point Tracking (solar-array control)
PCB	Printed Circuit Board
PID	Proportional-Integral-Derivative (controller)
PPR	Pulses Per Revolution (encoder spec)

PWM Pulse-Width Modulation

RPM Revolutions Per Minute

RPS Revolutions Per Second

RWS Reaction Wheel System

RWIP Reaction Wheel Inverted Pendulum

SMC Sliding Mode Control

USB Universal Serial Bus

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

This chapter presents a brief introduction to the need for attitude control in satellites, the relevant background information and the scope of this project's design solution.

Figure 1. NanoRacks CubeSats being launched from the NanoRacks CubeSat Deployer on the ISS on February 25, 2014 (NASA Science Editorial Team, 2019).

1.1 An Introduction to CubeSats

A CubeSat is a class of small satellite designed to be deployed into orbit as secondary payloads on a launch vehicle, or from the International Space Station, as seen in figure 1. CubeSats are designed with a standardized size of 10x10x10 cm, also known as a "1U" unit, to simplify their integration into deployment vehicles, but are modular and can consist of multiple units, as seen in figure 2, and are often manufactured with commercial off-the-shelf (COTS) components.

First conceived in 1999 to be a platform for scientific research and developing new space technologies (NASA, 2024), as of 2024 there are now 2396 CubeSats launched, including 16 interplanetary missions. Academia was responsible for most CubeSat launches initially,

but by 2014 the majority of newly deployed CubeSats were for commercial or amateur projects (Kulu, 2024).

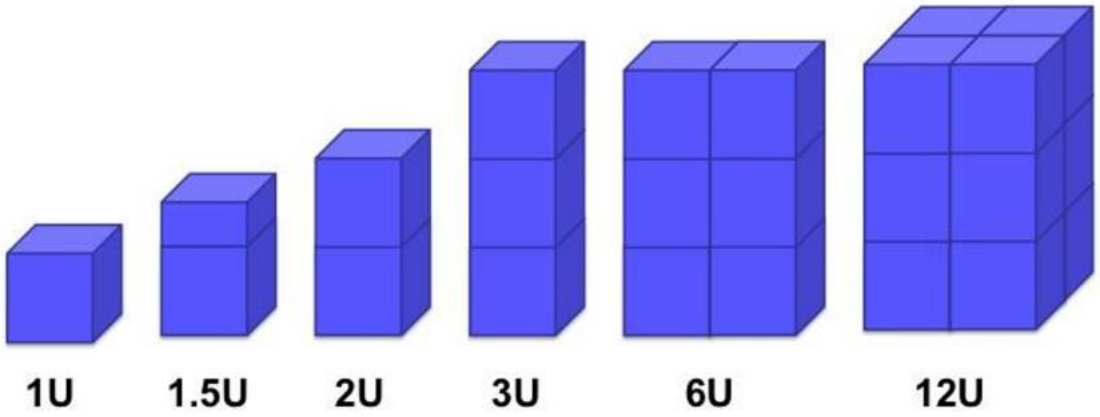


Figure 2. A Sample of CubeSat Arrangements (NASA Science Editorial Team, 2019).

Most CubeSats deployed today are used for Earth observation, communications, and scientific research, including climate monitoring, remote sensing, and technology testing. Additionally, they are increasingly employed for commercial purposes, such as providing data for telecommunication and internet services.

The expediated rate of growth in the nanosatellite commercial sector invites the need for continuous development of increasingly simplified and COTS platforms for integration into CubeSats.

1.2 Problem

Like larger satellites, CubeSats often feature multiple subsystems handling different tasks in parallel including the flight computer, electrical power system (EPS), payload operation, and primary control systems. A satellite's Guidance, Navigation and Control (GNC) subsystem encompasses the components used for position determination and the components used by the Attitude Determination and Control Systems (ADCS) and is crucial for a successful mission (NASA, 2024). Components used for ADCS include

reaction wheels, magnetorquers, thrusters, star trackers, sun sensors, Earth sensors, angular rate sensors, and GPS receivers and antennas.

Due to asymmetric deployment forces and bumping with other CubeSats, tumbling typically occurs during deployment. Some CubeSats operate normally while in a spin, but others that require precise heading must be detumbled. Reaction wheels are commonly used for their ability to impart relatively large amounts of moments on the overall system. Once detumbled a CubeSats orientation may still needed to be controlled to align its sensors for data acquisition or to optimize solar gain through its solar panel array. These operations present the need to develop a control system for precise attitude control in three axes for CubeSats, that doesn't rely on exerting a force on an object external to the satellite.

1.3 Background

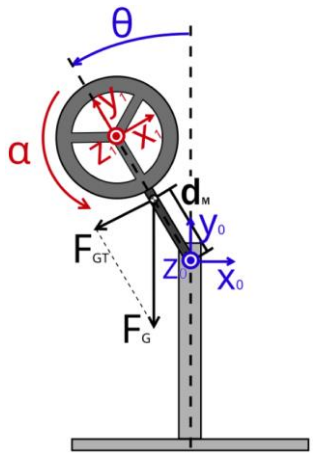


Figure 3. A Reaction Wheel being used to Control Orientation in one Axis (Dominik Zaborniak, 2024).

The use of reaction wheels to control the orientation of a system is well-documented in projects addressing the inverted pendulum problem, as illustrated in Figure 3 (Dominik Zaborniak, 2024). In this case, the torque generated by the reaction wheel is sufficient to counteract external disturbances and adjust the angular momentum of the system, including the combined effects of the system's moment of inertia, existing angular momentums and external disturbances. The reaction wheel imparts torque to the system, which is critical for maintaining stability in an otherwise unstable configuration. An optimized control

system would regulate the torque applied by the reaction wheel based on the magnitude and direction of the system's overall angular momentum, effectively stabilizing the system and keeping it in an upright position.

To control the orientation of the CubeSat in three axes, three reaction wheels can be arranged along its orthogonal axis, as in figure 4. This arrangement aligns each wheel to impart a moment on the system around a primary axis of rotation, either the x-, y-, or z-axis.

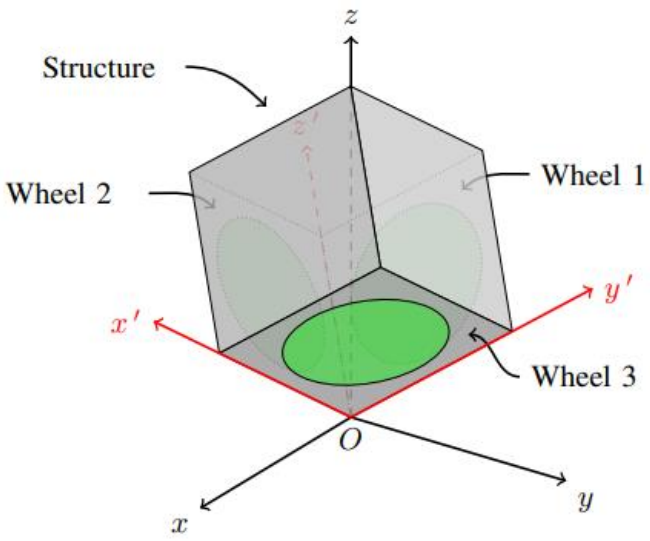


Figure 4. Three Axis Inverted Pendulum forming a cube (Bobrow, et al., 2020).

The Cubli, developed by Mohanarajah Gajamohan & colleagues in 2012, is an excellent example of this three-axis configuration. In Cubli, the system dynamics are modelled using Kane's equations, and orientation of the system is described using Euler angles, and brakes are incorporated to generate large torques from collisions with the wheel. These exert such a large force on the system that Cubli can "jump" from a face to an edge, then from balancing on the edge can jump again up onto a vertex, "catching" itself there and maintaining balance (Mohanarajah, et al., 2012).

Other iterations of Cubli use various means of describing the angular displacement and modelling the system dynamics. Papers by Fabio Bobrow and colleagues documents the

use of complex numbers to describe the angular displacement of a single cube face in 2D (Bobrow, et al., 2020), and, by extension, unit ultra-complex numbers (quaternions) to describe the angular displacement of the cube in 3D (Bobrow, et al., 2020). Using quaternions avoids encountering singularities found in Euler angle mechanics, singularities occur when the algorithm cannot find a solution to the required movements. The Lagrange's equations are used in these papers to derive a set of equations that describe the motion of the system.

Once an equation of motion is determined, a control strategy is necessary to reduce and eliminate the angle of displacement in the three dimensions and maintain stable orientation. An example of a control algorithm reducing the angular velocity of a system in 3 dimensions to zero, i.e. trying to keep it still, is shown in figure 5. Proportional-Integral-Derivative (PID) control is one of the most used methods for correcting orientation errors. It adjusts the control input based on the current error, accumulated past error, and the rate of change of the error. This method is effective for systems with relatively linear dynamics and ensures precise orientation adjustments.

Another useful strategy is the Linear Quadratic Regulator (LQR), which minimizes a cost function that penalizes both deviations from the desired state and the control effort. This is particularly suitable when the system can be linearized around an equilibrium point, providing optimal control for CubeSat attitude control around a point, while minimizing both the angle state error and revolutions per minute (RPM) of the reaction wheel, keeping the wheel speed to a minimum (Mohanarajah, et al., 2013).

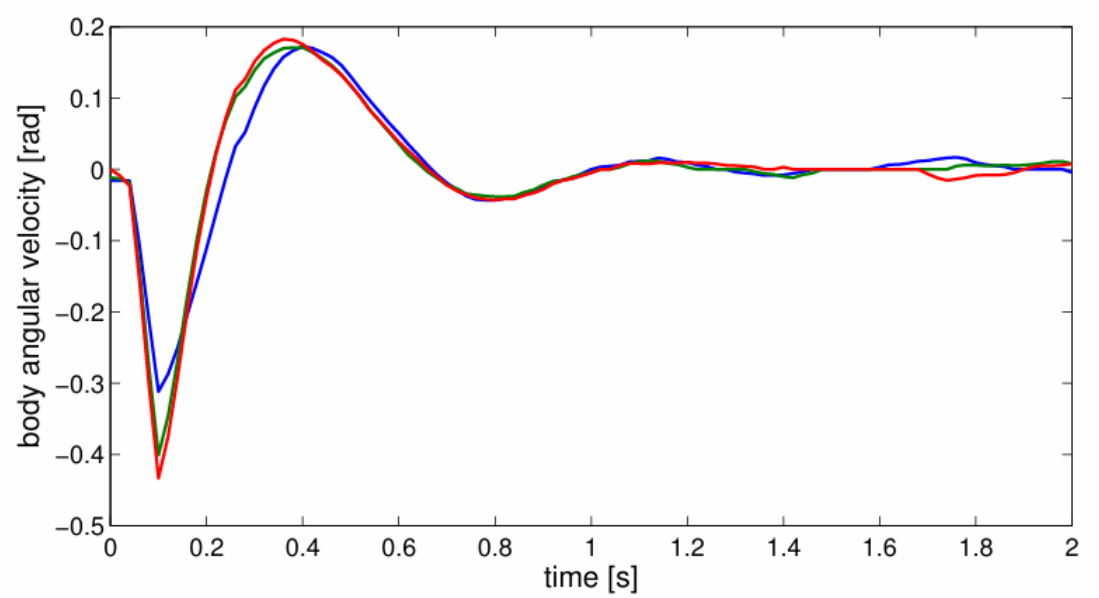


Figure 5. An Example of PID control settling the angular velocity of a system (Mohanarajah, et al., 2013)

These control strategies, when implemented appropriately, ensure that CubeSats maintain precise orientation and stability in space.

1.4 Scope and Objectives

The scope of this project focuses on the design and development of an optimized attitude correction and control system for a 1U CubeSat. The primary objective is to create a model capable of stabilizing and orienting itself using reaction wheels as the primary actuators. The model will simulate a CubeSat's behaviour, maintaining its orientation against external factors such as gravitational forces, sensor noise, and the varying centre of mass and thus the varying of moments of inertia configurations, not unlike when a CubeSat would deploy a solar array. The project aims to ensure that the CubeSat remains stable, and can be orientated about its yaw axis, enabling precise heading for data collection, optimal solar panel alignment, and communication tasks.

Key technical objectives include conducting an in-depth literature review to investigate current CubeSat attitude control technologies, reviewing flywheel system dynamics, and analysing existing studies on control design for orientation. The project will then move on

to the design phase, including both mechanical and electronic components, with detailed engineering drawings and circuit schematics. A comprehensive software flow diagram outlining the control algorithms will also be developed.

Following the design, the project will simulate the proposed system using relevant software tools to validate its functionality. Once validated, a prototype will be built, first for one dimension then moving onto three dimensions, to verify the measurements and movements of the attitude control system. The prototypes will be rigorously tested, and iterative adjustments will be made to optimize the control algorithms based on performance feedback. The final phase will involve comprehensive testing and evaluation to ensure the CubeSat's attitude control system can stabilize the satellite and orient it to the desired direction autonomously.

In addition to the core objectives, the project will explore the integration of a solar array with Maximum Power Point Tracking (MPPT) to simulate varying configurations of mass of the satellite. A graphical user interface (GUI) to display heading, pitch, roll, and a live camera feed is also considered as a potential bonus feature to simulate data capture.

By the end of this project, a robust, low-cost, and efficient attitude control system for CubeSats will be developed, with applications extending to public, private and commercial sectors.

1.5 Document Overview

This document outlines the design, modelling, and testing of an optimized attitude correction and control system for a 1U CubeSat. Chapter 1 introduces the key challenges CubeSats face, particularly the need for precise attitude control to maintain proper alignment of solar panels, sensors, and communication systems. The background section discusses the use of reaction wheels for orientation control and explores various control strategies that are employed to stabilize the CubeSat’s attitude in three dimensions.

Chapter 2 provides a comprehensive technical foundation for the CubeSat system. It begins with the physical components of the CubeSat prototype, including the microcontroller,

motors, and reaction wheels. The chapter continues with a discussion on the derivation of the equations of motion, which describe the CubeSat's dynamics and form the basis for the attitude control system along with the PID error control. The different methods for representing attitude, such as quaternions and Euler angles, highlighting their respective advantages and challenges in CubeSat applications, are also detailed. This section explains how these components are integrated to achieve stable and efficient attitude control.

In Chapter 3, the methodology for the construction of the CubeSat model is outlined, followed by the implementation of the control system. This chapter also describes the testing process, including simulations and real-world validation of the CubeSat's performance. Chapter 4 presents the results from these tests, evaluating the effectiveness of the attitude control system in stabilizing the model’s orientation. Finally, Chapter 5 concludes the report by summarizing the findings and offering recommendations for further improvements in CubeSat attitude control systems.

Chapter 2 Literature Review

This chapter presents a review of the technologies and methodologies for attitude control in CubeSats. CubeSats require efficient and reliable systems to ensure precise attitude control in orbit. This review covers essential topics, including the construction of a system with manipulative dynamics, mathematical frameworks for attitude representation, and linear and nonlinear control strategies. The work presented here is crucial to the development of the CubeSat attitude correction system.

2.1 Reaction Wheel-Based Systems for Attitude Control

A reaction wheel system (RWS) is a key component in small spacecraft such as CubeSats for precise attitude control, using spinning wheels to generate torques that modify the spacecraft's orientation. As the wheels spin, they generate angular momentum, which is transferred to the spacecraft due to the conservation of momentum. The principle can be described by Newton’s Second Law of Motion when applied to rotating bodies, every action having an equal but opposite reaction. This system does not require external thrusters or fuel reservoirs, making it well-suited for small spacecraft like CubeSats, where minimizing mass and maximizing efficiency are essential.

The Cubli is a prominent example of a 3D inverted pendulum system that uses three reaction wheels mounted on the cube’s faces to control its orientation. By applying controlled torques from these wheels, the Cubli can balance on its edge or corner. This design, although originally developed for a ground-based balancing robot, provides the relevant framework for CubeSat design due to the similarities in mechanics and constraints on mass, volume, and power.

The mechanical structure of such RWS’s consists of a lightweight yet rigid housing, with brushless DC motors driving the reaction wheels. For CubeSat applications, a similar design can be adapted as the Cubli balancing in +1g gravity on Earth, proves the system’s ability to work for space-specific, 0g, requirements.

In a CubeSat reaction wheel system, the reaction wheels are typically mounted on three orthogonal axes. This setup provides full control over the spacecraft's attitude, allowing for rotations around all three axes (yaw, pitch, and roll). The reaction wheels are controlled by a feedback system that adjusts the wheel speeds based on input from sensors that measure the spacecraft's orientation.

2.1.1 Mechanical Build of a Reaction Wheel Inverted Pendulum

For the development of a CubeSat model with a reaction wheel attitude control system, the mechanical build would include:

1. **Three Reaction Wheels**: Each mounted on a face of the CubeSat. These wheels will be coupled with motors to generate the required torques for attitude control.

2. **Brushless DC Motors**: These motors are ideal for CubeSats due to their efficiency, precise control, and low power consumption. The motors must be mounted in such a way that the CubeSat's centre of mass is close to the axis of rotation to minimize the need for large control torques.

3. **Structural Housing**: The CubeSat housing must be designed to be lightweight and durable, with a rigid framework that can withstand the forces generated by the reaction wheels.

4. **Sensors**: IMUs (Inertial Measurement Units) are typically used for real-time measurement of the CubeSat’s orientation. These units often combine accelerometers, gyroscopes, and sometimes magnetometers to provide precise attitude determination.

5. **Microcontroller**: A microcontroller unit (MCU) will be used to read the sensor input and then output to the motor drivers accordingly based on the devised control system. A development board with native Bluetooth and Wi-Fi connectivity will be considered.

6. **Electrical Power System:** The electrical power system (EPS) will consist of a battery bank with large enough capacity to support the three motors and other electronics, a charge regulator for the battery and two voltage regulators, one for the motors and one for the MCU and sensor

This mechanical setup allows for a compact and efficient attitude control system for CubeSats, with minimal power consumption and no need for external propulsion systems. The compact nature of CubeSats, however, imposes stringent limitations on mass and power, requiring optimization of both the mechanical and electrical systems.

2.1.2 Reaction Wheel

The reaction wheel is a critical component in CubeSat attitude control systems, used to generate torques that alter the spacecraft's orientation. The fundamental principle behind a reaction wheel is the conservation of angular momentum: when the wheel spins, it creates angular momentum, and any change in the wheel's angular velocity results in a corresponding change in the CubeSat's attitude. This key principle is essential for precisely controlling the model's orientation through the adjustment of the wheel's speed and torque.

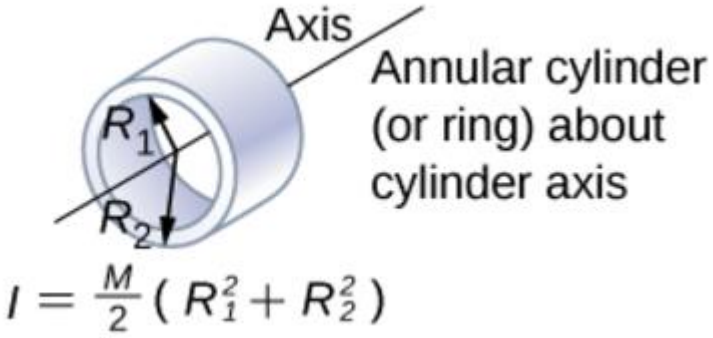


Figure 6. Inertia of an Annular Cylinder about Centre Axis (Moebs, et al., 2016)

The moment of inertia (I) of the reaction wheel is given by:

$$I = \frac{m}{2}(R_1^2 + R_2^2) \tag{2.1}$$

Where m is the mass, and R_1 and R_2 are the inner and outer radii of the wheel. This parameter determines the required torque to achieve specific angular accelerations (Moebs, et al., 2016).

Key Parameters for Reaction Wheel Design:

1. **Mass (m):**

   The mass of the reaction wheel affects its moment of inertia and torque generation. A lighter wheel reduces the CubeSat's overall weight but may limit torque generation, requiring a balance between sufficient torque and weight.

2. **Radius (r):**

   The radius influences the moment of inertia. A larger radius increases torque capability but also raises mass and power requirements.

3. **Torque (τ):**

   The torque generated by the reaction wheel is related to angular acceleration by:

$$\tau = I \cdot \alpha \tag{2.2}$$

   Where α is angular acceleration. This equation governs how the reaction wheel's change in speed affects the model's orientation (Beer & Johnston, 1989).

4. **Angular Velocity (ω):**

   The angular velocity relates to angular momentum L, given by:

$$L = I \cdot \omega \tag{2.3}$$

   Where ω is the angular velocity. This stored angular momentum is transferred to the CubeSat, enabling attitude control (State Examinations Commission, 2011).

5. **Kinetic Energy (E_k):**

The kinetic energy in the reaction wheel is:

$$E_k = \frac{1}{2} I\omega^2 \tag{2.4}$$

This energy affects the CubeSat's power consumption and how efficiently it can maintain orientation (State Examinations Commission, 2011).

6. **Potential Energy (E_p):**

The potential energy, especially in varying gravitational fields or when adjusting orientation relative to a force, is given by:

$$E_p = mgh \tag{2.5}$$

Where h is the height or displacement relative to the gravitational source. This energy must be accounted for in attitude adjustments (State Examinations Commission, 2011).

7. **Control Speed Limit:**

The reaction wheel has physical limits, such as the maximum angular velocity and the maximum torque it can generate. These limits must be factored into the control system to avoid overdriving the wheel and ensure the CubeSat operates within its capabilities.

8. **Energy Efficiency:**

Energy consumption is a key factor in CubeSat design. The reaction wheel's energy use, including kinetic and potential energy, should be optimized to balance power requirements for attitude adjustments within the CubeSat's limited power budget.

9. **Rotational Inertia and Response Time:**

   The CubeSat's overall inertia determines how quickly it can change its orientation. A higher CubeSat inertia requires more torque to achieve the same angular acceleration, influencing the system's response time and stability (Mohanarajah, et al., 2013).

By selecting appropriate values for these design parameters—such as mass, radius, moment of inertia, torque, and angular velocity—the reaction wheel can be optimized for precise CubeSat attitude control. These parameters are essential for ensuring the wheel can generate the required torques while maintaining energy efficiency and operating within the CubeSat's power constraints. Modelling these parameters accurately is key to integrating the reaction wheel into the CubeSat’s attitude control system, ensuring stable and effective orientation adjustments in space.

2.1.3 Motor

The reaction wheel system will use the Nidec24H brushless DC motor (model NCJ-24H-12-01), which integrates the motor, motor driver, and encoder into a single unit. This integrated design simplifies the CubeSat's attitude control system by reducing the number of components and minimizing space, weight, and power consumption—critical considerations for CubeSat applications. This model was chosen as it’s a readily available COTS component.

Figure 7. Brushless DC Motor Nidec 24H (Nidec Corp., 2024)

Key Parameters for Motor Design:

1. **Operating Voltage (V):**

   The motor operates at 12 V, which directly impacts the power consumption and overall energy budget. This voltage level is chosen to balance power requirements with the CubeSat's limited energy supply.

2. **Continuous Torque (τ):**

   The motor generates a continuous torque of 25 mN·m, which is essential for the reaction wheel to produce the necessary forces for attitude control. This torque value is a key parameter for determining the wheel's ability to alter the model's orientation.

3. **Maximum Speed (ω_{max}):**

   The motor has a maximum speed of 2,500 RPM, which dictates how quickly the CubeSat can adjust its orientation. The speed determines how fast the reaction wheel can be spun to achieve desired attitude changes.

4. **Encoder Resolution (PPR):**

   The motor includes an integrated dual line encoder with 200 pulses per revolution (PPR), providing high-resolution feedback on the motor's speed and position. This allows feedback for precise control of the reaction wheel's rotational velocity and accurate feedback for attitude control.

5. **Maximum Current (I_{max}):**

   The motor has a maximum current rating of 0.9 A, ensuring that it will not exceed the model's power supply capacity. This is important for managing the CubeSat's energy usage and ensuring that the motor operates within safe limits.

6. **Efficiency and Design:**

   The brushless design of the motor ensures high efficiency, which is critical for long-duration space missions where energy conservation is essential. The motor's design also minimizes maintenance needs, as it has no brushes that could wear out over time.

7. **Control via PWM:**

   The motor speed is regulated by PWM (Pulse Width Modulation) control, which adjusts the duty cycle of the signal sent to the motor driver. By varying the duty cycle, the average voltage applied to the motor is controlled, which in turn controls the speed and torque of the reaction wheel. This method efficiently regulates the motor's performance while minimizing power loss, allowing for fine control over the reaction wheel's speed.

These parameters are crucial for determining system power requirements and motor behaviour, ensuring the reaction wheel system operates efficiently, while the integrated encoder allows for precise feedback, enabling the model to maintain and accurately adjust wheel speed (Nidec Corp., 2024).

2.1.4 Structure

The CubeSat structure is designed to support the reaction wheel and other critical components while maintaining overall system integrity. It's built to house the motor, IMU, and other subsystems, ensuring everything is secure and well-balanced.

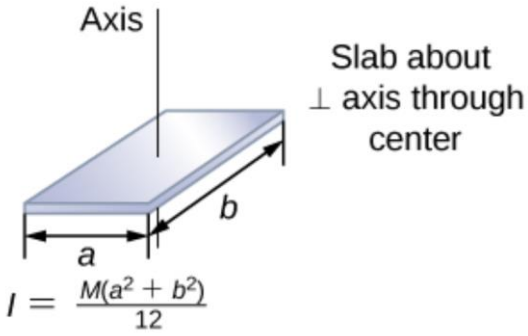


Figure 8. Inertia for Slab About Centre Axis (Moebs, et al., 2016).

The main job of the structure is to provide a solid foundation for the reaction wheel and motor, while also keeping the system's centre of mass balanced. To get an accurate sense of the system's dynamics in one dimension, the moment of inertia of the structure is calculated using a simple slab model, as follows:

$$I = \frac{m(a^2 + b^2)}{12} \tag{2.6}$$

where m is the mass, and a and b are the dimensions of the model. This gives a starting point for modelling the rotational behaviour of the CubeSat. As the CubeSat will be balancing on a vertex, due to the limitations of gravity, the Parallel Axis Theorem is used to account for the offset of the centre of rotation from the CubeSat's centre of mass:

$$I_b = I_c + md^2 \tag{2.7}$$

When dealing with three axis of rotation, the inertia tensor is a key parameter needed to model the CubeSat's rotational dynamics. The inertia tensor, I, describes how the mass is distributed relative to the CubeSat's three principal axes of rotation. For a simple CubeSat model, the inertia tensor can be approximated using the parallel axis theorem, taking into account the mass distribution and the geometry of the CubeSat. For a cube, the inertia tensor is diagonal and can be expressed as:

$$I = \begin{bmatrix} \frac{1}{6}m_{total}(a^2 + b^2) & 0 & 0 \\ 0 & \frac{1}{6}m_{total}(a^2 + b^2) & 0 \\ 0 & 0 & \frac{1}{6}m_{total}(a^2 + b^2) \end{bmatrix} \quad (2.8)$$

Where *a*, *b*, and *c* are the dimensions of the CubeSat, and m_{total} is the total mass. The diagonal elements represent the moments of inertia about the principal axes, while the off-diagonal elements represent the products of inertia, which are assumed to be zero for a symmetric CubeSat with a simple geometry (Beer & Johnston, 1989).

These parameters—total mass, centre of mass, and inertia tensor—are essential for deriving the equations of motion for the CubeSat's attitude control system. The equations of motion describe the CubeSat's rotational dynamics under the influence of external torques (e.g., from the reaction wheel) and disturbances (e.g., gravitational gradients, solar pressure). By accurately modelling these parameters, the dynamics of the CubeSat can be simulated and controlled effectively using the reaction wheel system.

2.1.5 Sensors

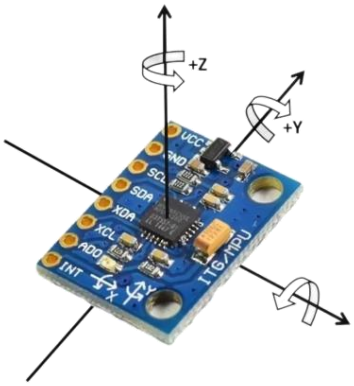


Figure 9. MPU6050 6 Axis Accelerometer/Gyroscope (Arrendondo, 2023)

The CubeSat model utilizes the MPU6050 sensor, an Inertial Measurement Unit (IMU) that integrates both a 3-axis accelerometer and a 3-axis gyroscope. This sensor is essential for determining the CubeSat's orientation and angular velocity in each axis, which are key for the attitude control system. The MPU6050 provides real-time feedback on the

CubeSat's position and motion, enabling the precise measurements required to correctly adjust the reaction wheel's torque in order to stabilize the model's orientation.

The accelerometer measures linear acceleration along the X, Y, and Z axes, providing information on the gravitational forces acting on the CubeSat. This data is used to determine the CubeSat's tilt or attitude relative to the Earth's surface. The gyroscope measures the angular velocity along the same three axes, allowing the CubeSat's rotational motion to be tracked and enabling the system to compensate for changes in orientation.

The MPU6050 communicates via I2C protocol, allowing for efficient data transmission between the sensor and the CubeSat's microcontroller (ESP32). The sensor offers high precision, with a range of $\pm 2g$ to $\pm 16g$ for the accelerometer and ± 250 to ± 2000 degrees per second for the gyroscope, making it suitable for precise attitude measurement in small spacecraft applications like CubeSats. (DFRobot, 2024)

This sensor is integral to the CubeSat's attitude determination system (ADS), providing the necessary data for the control algorithms to adjust the reaction wheel's speed and thus, its torque. By integrating the MPU6050 with the CubeSat's attitude control system, the satellite can measure accurate orientation during its mission.

2.1.6 Microcontrollers

Microcontrollers serve as the central processing units in CubeSat systems, managing sensor data processing, control algorithms, and communication tasks. The choice of microcontroller depends on various factors such as processing power, memory capacity, power consumption, and interface availability. Two commonly used microcontrollers are the Arduino Uno and the ESP32.

2.1.6.1 Arduino Uno

The Arduino Uno R3 development board, a widely used and easy-to-program microcontroller. It is built around the ATmega328P 8-bit microcontroller, running at a clock speed of 16 MHz. Although less powerful than other modern microcontrollers, it is capable of handling essential data processing from the sensors and controlling the reaction wheel motor using PWM signals.

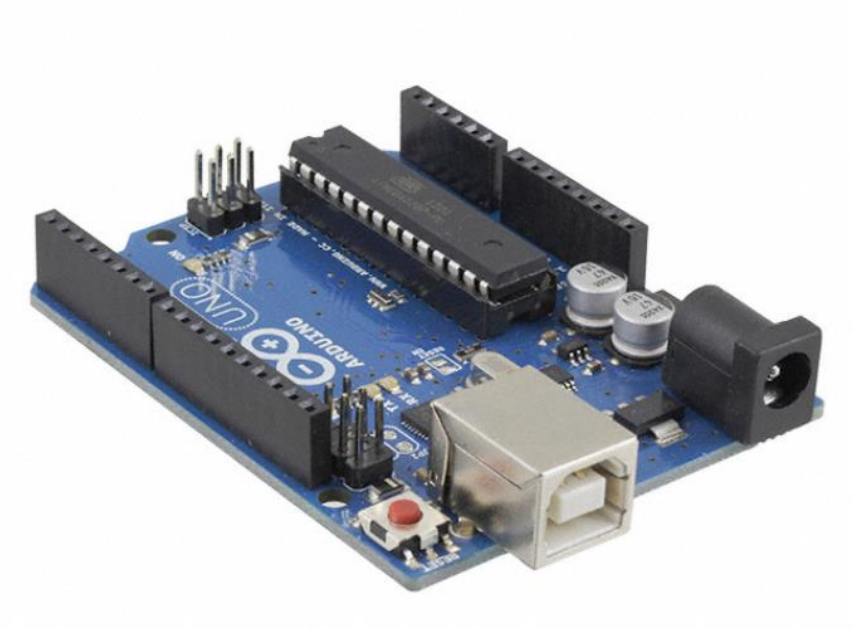


Figure 10. Arduino Uno Development Board

The Arduino Uno includes 14 digital input/output pins (of which 6 can provide PWM output), 6 analog inputs, and a USB connection for programming and serial communication. Its simplicity and reliability make it a suitable choice for early-stage CubeSat designs and developmental projects. The microcontroller offers 32KB of flash memory, 2KB of SRAM, and 1KB of EEPROM, which are sufficient for basic control algorithms and sensor interfacing tasks.

The I2C interface on the Arduino Uno is used to communicate with the MPU6050 sensor, while the GPIO pins are used to drive the motor interface, enabling precise control of the reaction wheel. While the Arduino Uno consumes slightly more power relative to more advanced boards, it remains an accessible and well-supported platform for prototyping CubeSat control systems (Arduino, 2025).

2.1.6.2 ESP32

The ESP32-DevKitC development board, a powerful and energy-efficient microcontroller. It features a dual-core 32-bit processor running at up to 240 MHz, which is capable of handling real-time sensor data processing from the MPU6050 and controlling the reaction wheel motor via PWM signals.

Figure 11. ESP32 Development Board (DigiKey, 2024)

The ESP32-WROOM-DA chip used for this development board includes Wi-Fi and Bluetooth connectivity, useful for testing and debugging, as well as communication with a ground station during development. The microcontroller has 4MB of flash memory and 520KB of SRAM, providing enough resources to store control algorithms and handle data processing tasks. (DigiKey, 2024)

The I2C interface on the ESP32 is used to communicate with the MPU6050 sensor, while its GPIO pins interface with the motor driver, allowing precise control of the reaction wheel. With low power consumption, the ESP32-WROOM is well-suited for CubeSat applications, balancing performance and power efficiency within the CubeSat's limited resources.

2.1.7 Electrical Power System

The CubeSat's Electrical Power System (EPS) is designed to manage the power distribution and charging of the batteries. The system was developed using KiCAD, a widely used open-source electronic design automation (EDA) tool, to create a custom printed computer board (PCB) that meets the specific power requirements of the CubeSat's components.

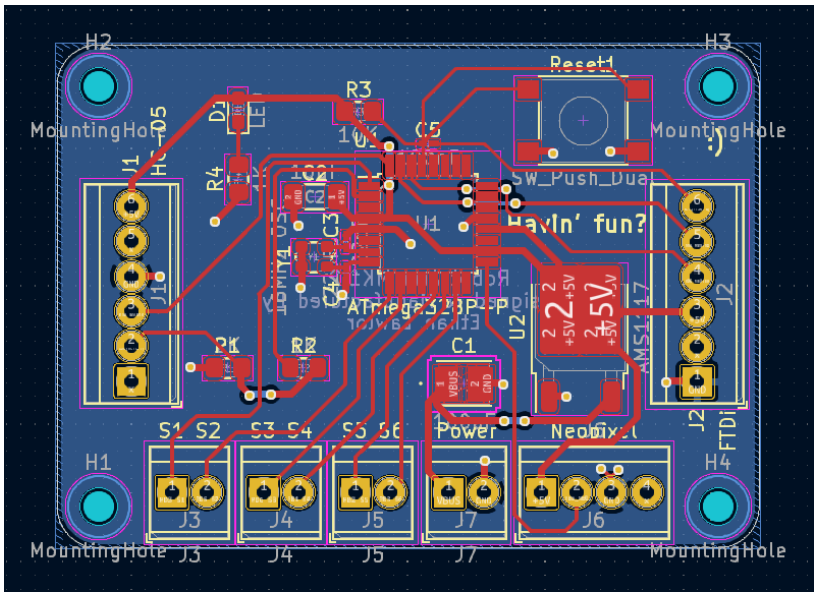


Figure 12. Custom PCB for Electrical Power System

The PCB is designed to regulate power from the CubeSat's solar panels and distribute it to critical components, including the microcontroller, sensors, and reaction wheel motor. The system incorporates a charging circuit for the CubeSat’s battery, ensuring that the battery remains charged during operation while preventing overcharging or deep discharge, which could damage the battery. Voltage regulators on the custom PCB provide stable power to different subsystems, such as the ESP32 microcontroller and the MPU6050 sensor, ensuring that each component receives the appropriate voltage for safe operation.

The EPS also includes power monitoring capabilities, allowing the CubeSat to track battery voltage and current draw to optimize energy usage and ensure that the system remains within its power limits during the mission. By integrating all these power management functions into a single custom PCB, the system minimizes size, weight, and complexity while providing reliable power to the CubeSat’s attitude control system and other subsystems.

2.2 Mathematical Methods for Attitude Representation

Precise modelling of spacecraft dynamics is a crucial aspect of attitude control. CubeSat attitude determination relies on mathematical methods for representing rotations and calculating the required torques to control the spacecraft's orientation. The following mathematical methods are commonly used in spacecraft attitude determination and control:

2.2.1 Euler Angles

Euler angles provide a simple method for describing the orientation of a rigid body in three-dimensional space. Using three rotational angles—yaw, pitch, and roll—Euler angles define the orientation of the spacecraft relative to a reference coordinate system. The rotations are typically performed about fixed axes, such as:

$$R = R_z(\phi)R_y(\theta)R_x(\psi) \tag{2.9}$$

where:

- ϕ is the yaw angle (rotation about the z-axis),
- θ is the pitch angle (rotation about the y-axis),
- ψ is the roll angle (rotation about the x-axis).

Although Euler angles are easy to understand and implement, they suffer from gimbal lock, where two of the rotation axes become aligned, causing a loss of one degree of freedom. This limitation makes Euler angles unsuitable for continuous and precise control over large rotations.

2.2.2 Quaternions

Quaternions offer a more robust solution for representing rotations in three-dimensional space. A quaternion consists of one scalar and three vector components:

$$q = q_0 + q_1 i + q_2 j + q_3 k \tag{2.10}$$

where q_0 is the scalar part and q_1, q_2, q_3 are the vector components. Quaternions are particularly advantageous in CubeSat systems because they do not suffer from singularities and allow for smooth and continuous rotations.

The relationship between the quaternion and rotation matrix is given by:

$$R = \begin{bmatrix} 1-2({q_2}^2+{q_3}^2) & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_1q_2+q_0q_3) & 1-2({q_1}^2+{q_3}^2) & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_2q_3+q_0q_1) & 1-2({q_1}^2+{q_2}^2) \end{bmatrix} \quad (2.11)$$

This matrix can be used to transform vectors from the body frame to the inertial frame, ensuring precise attitude control. (Kuipers, 1999)

2.2.3 Lagrangian Mechanics

Lagrangian mechanics provides a framework for modelling the dynamics of complex systems, including spacecraft. The Lagrangian L is defined as the difference between the kinetic energy T and potential energy U of the system:

$$L = T - U \quad (2.12)$$

The equations of motion are derived using the Euler-Lagrange equations, which describe the dynamics of the system in terms of generalized coordinates (such as orientation angles) and velocities:

$$\frac{d}{dt}\left(\frac{dL}{d\dot{q}_i}\right) - \frac{dL}{dq_i} = 0 \quad (2.13)$$

These equations allow the spacecraft's rotational behaviour to be predicted, providing the necessary models for control system design (Goldstein, et al., 2002).

2.3 Sensor Integration for Attitude Determination

Inertial Measurement Units (IMUs) are essential for accurate attitude determination in CubeSats, providing real-time data on orientation, angular velocity, and acceleration. A typical IMU combines an accelerometer and a gyroscope to deliver comprehensive

orientation information. The MPU-6050 is a widely used IMU that integrates a 3-axis accelerometer and a 3-axis gyroscope, offering reliable attitude data.

The accelerometer measures the force of gravity along each axis, enabling the determination of tilt angles, while the gyroscope measures angular velocity, allowing for the tracking of changes in orientation. To enhance the accuracy and reduce noise in attitude estimates, these measurements are often fused using filters such as the Kalman filter or Complementary filter.

2.3.1 Complementary Filter

$$\theta_{est}(t) = \alpha\left[\theta_{est}(t - \Delta t) + \omega_{gyro} \cdot \Delta t\right] + (1 - \alpha) \cdot \theta_{acc} \quad (2.14)$$

The complementary filter is a computationally efficient method that combines accelerometer and gyroscope data to estimate orientation. It blends the two sensors by giving more weight to the gyroscope for short-term, high-frequency changes and more weight to the accelerometer for long-term, low-frequency stability. This makes it effective in many applications, especially when computational resources are limited, as it helps to minimize the gyroscope's drift over time while maintaining accurate orientation (Halder, et al., 2025).

2.3.2 Kalman Filter

$$Angle_{kalman}(k) = Angle_{kalman}(k) + Gain_{kalman} \cdot (Angle(k) - Angle_{kalman}) \quad (2.15)$$

The Kalman filter is an optimal estimation algorithm that combines sensor measurements over time to produce more accurate estimates of a system's state. It works by continuously updating the state estimate based on both the current sensor readings and the predicted state, effectively minimizing the impact of noise. This makes it particularly effective in systems with noisy data, and it is widely used in attitude estimation applications where precise and reliable estimates are crucial (Candan & Soken, 2021).

By effectively integrating accelerometer and gyroscope data using these filtering techniques, CubeSats can achieve precise and reliable attitude determination, which is crucial for their navigation and control systems.

2.4 Control Algorithms for Attitude Stabilization

To achieve precise attitude control for CubeSats, several control algorithms are employed depending on the system's complexity and the desired performance. These control methods adjust the reaction wheel torque in real-time based on the CubeSat's orientation and angular velocity, enabling the CubeSat to stabilize and eliminate any angular displacement.

2.4.1 Proportional-Integral-Derivative

Proportional-Integral-Derivative (PID) controllers are a widely used approach for maintaining stability and reducing displacement, thanks to their simplicity and ease of implementation. By tuning the proportional, integral, and derivative gains, a PID controller can correct angular displacement and minimize oscillations effectively. This approach is particularly useful in systems with relatively linear dynamics, where the model is straightforward, and the system is responsive to simple feedback. The controller's output adjusts based on three terms: the current error, the accumulation of past errors, and the rate of change of the error.

The PID control law is:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{de(t)}{dt} \tag{2.16}$$

where:

- $u(t)$ is the control input (torque),
- $e(t)$ is the orientation error,
- K_p, K_i, K_d are the proportional, integral, and derivative gains, respectively.
- Proportional gain helps reduce large errors quickly.

- Integral gain eliminates steady state error.
- Derivative gain helps reduce overshoot and improve stability.

PID control is especially effective in applications like CubeSat attitude control, where precise tracking of angular displacement is required over time (Beer & Johnston, 1989).

2.4.2 Cascade Loop

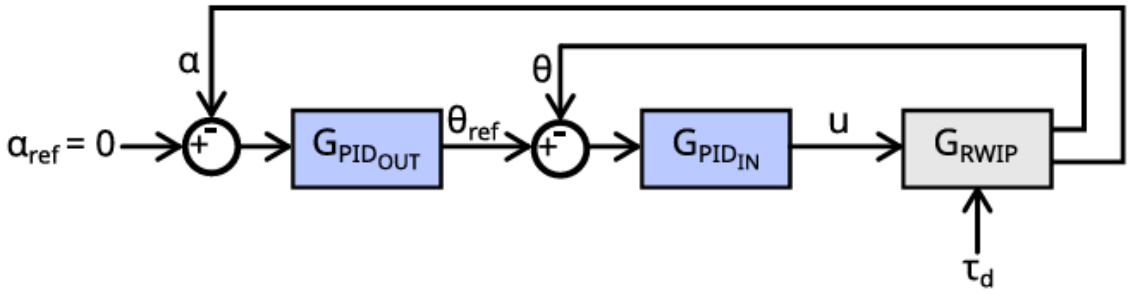


Figure 13. Cascade Loop Control Diagram (Dominik Zaborniak, 2024).

The cascade control loop is a commonly used strategy for controlling systems that require both fast response and precise stabilization, such as the reaction wheel inverted pendulum (RWIP). In this system, two nested feedback loops are implemented: an inner velocity loop and an outer position loop. The inner loop controls the speed or torque of the reaction wheel based on feedback from the motor's encoder, allowing for quick adjustments and stabilization. The outer loop, on the other hand, manages the pendulum's angle by adjusting the desired motor speed according to the pendulum's angular position and velocity, typically using a PID controller. This structure allows the inner loop to handle fast dynamics while the outer loop focuses on maintaining the desired orientation. By decoupling these two tasks, cascade control improves the overall performance and stability of the system, making it ideal for applications like CubeSat attitude control where precise orientation is essential (Dominik Zaborniak, 2024).

2.4.3 Linear-Quadratic Regulator

Linear-Quadratic Regulator (LQR) is another effective control strategy, particularly when the system dynamics can be linearized around an equilibrium point. LQR aims to minimize a cost function that penalizes both deviations from the desired state and excessive control

input. It is especially suitable for linear systems where a model of the system is available, as it provides a systematic way to determine optimal control inputs.

The general LQR cost function is:

$$J = \int_0^\infty (x(t)^T Q x(t) + u(t)^T R u(t)) dt \tag{2.17}$$

Where:

- $x(t)$ is the state vector (which includes the orientation and angular velocity),
- $u(t)$ is the control input (reaction wheel torque),
- Q and R are positive definite weighting matrices that penalize state deviations and control efforts, respectively.

The optimal control law is then given by:

$$u(t) = -Kx(t) \tag{2.18}$$

Where K is the gain matrix, which can be computed from the solution to the Riccati equation. LQR controllers are particularly useful when precise, optimal control is needed, such as in maintaining the CubeSat's orientation on its edge or corner (Mohanarajah, et al., 2013).

2.4.4 Nonlinear Control

For systems where nonlinear dynamics cannot be ignored, nonlinear control strategies are required. Nonlinear controllers account for the system's full dynamics, including the interaction between reaction wheels and the CubeSat's rigid body. These controllers are particularly useful when dealing with large angular displacements or when the system operates outside of linear ranges.

Nonlinear control laws can be derived from the system's equations of motion using methods such as Lyapunov’s direct method or by incorporating generalized momentum. These

methods are effective in stabilizing systems exhibiting significant nonlinearities, like the Cubli, where the system must transition from a face to an edge or corner. The control law typically takes the form:

$$u(t) = f\big(x(t), \dot{x}(t)\big) \tag{2.19}$$

Where:

- $f(x(t), \dot{x}(t))$ represents a nonlinear function based on the system's state and its time derivatives (Mohanarajah, et al., 2013).

Nonlinear control is particularly useful for achieving stable, smooth transitions in attitude control, especially when the CubeSat experiences large external disturbances.

2.4.4 Sliding Mode Control (SMC)

Sliding Mode Control (SMC) is a robust control technique that can handle system uncertainties and external disturbances effectively. It works by forcing the system to "slide" along a predefined surface, called the sliding surface, where the system's behaviour is constrained to follow a desired trajectory. This is particularly useful for attitude control in CubeSats, where external disturbances like gravitational forces or aerodynamic drag can affect the satellite's orientation.

The sliding surface is typically defined as a function of the system's states:

$$\sigma\big(x(t)\big) = Cx(t) \tag{2.20}$$

Where CCC is a matrix that defines the sliding surface. The control law forces the system's trajectory to reach and stay on this surface, which is designed to drive the system to a desired equilibrium state. The SMC control law is then given by:

$$u(t) = -Ksgn\left(\sigma\big(x(t)\big)\right) \tag{2.21}$$

Where:

- K is a constant gain,

- $sgn(\sigma(x(t)))$ is the sign function, ensuring that the system stays on the sliding surface (Liao, et al., 2020).

SMC is particularly advantageous in nonlinear systems with significant uncertainties or disturbances, providing excellent robustness and stability.

2.4.5 Back-Stepping Control

Back-Stepping Control is a recursive control design method used to stabilize nonlinear systems, especially those with multiple interacting states, such as CubeSats with reaction wheels. It works by progressively designing controllers for different subsystems, starting from simpler ones and working towards more complex parts of the system. At each step, a Lyapunov function is constructed to ensure stability, and the control input is adjusted accordingly.

The general approach involves choosing a candidate Lyapunov function for each subsystem:

$$V_i(x_i) = \frac{1}{2} x_i^T P_i x_i \tag{2.22}$$

Where $V_i(x_i)$ is the Lyapunov function for the i-th subsystem, and P_i is a positive definite matrix. The control law is then derived to ensure that the derivative of the total Lyapunov function is negative, ensuring asymptotic stability.

This technique is particularly effective in systems like CubeSats, where the attitude control involves multiple interacting components, such as the CubeSat body and the reaction wheels (Mohanarajah, et al., 2012).

These control strategies are essential for stabilizing CubeSats, ensuring that they can maintain precise orientation for sensor alignment, optimal solar panel positioning, and other critical operations. The choice of control algorithm depends on the specific application and the complexity of the system's dynamics.

2.5 Considerations for Space Deployment

The components used in this project—such as the Arduino Uno, MPU6050 IMU, and Nidec brushless motor—are ideal for Earth-based prototyping due to their low cost and ease of integration. However, these commercial off-the-shelf (COTS) parts are not suitable for direct use in space, where conditions are far more demanding. Exposure to vacuum, microgravity, and ionizing radiation can cause unshielded electronics to malfunction or fail entirely.

For a space-deployable version of this system, components with space heritage and radiation tolerance would be required. Microcontrollers, sensors, and power systems must be tested for resistance to single-event upsets and thermal extremes. Materials must also be selected for structural integrity under launch vibration and thermal cycling, replacing 3D-printed plastics with aerospace-grade metals or composites.

Finally, power management must be more robust. While the custom EPS works for lab testing, space systems need radiation-hardened regulators, reliable battery protection, and efficient solar energy capture. While the model is suitable for terrestrial-based attitude control, transitioning this prototype to a real CubeSat would require significant upgrades in electronics, thermal management, radiation shielding, and structural design.

Chapter 3 Materials and Methods

This chapter offers an in-depth explanation of the project's design and implementation process. It outlines the various techniques and procedures employed to construct and achieve the project's goals.

3.1 Project Block Diagram

The project block diagram, shown in Figure 14, illustrates the architecture of the reaction wheel inverted pendulum system, which models an attitude correction and control system for CubeSats. This diagram shows the interconnection between key components, outlining the flow of power, data, and control signals throughout the system.

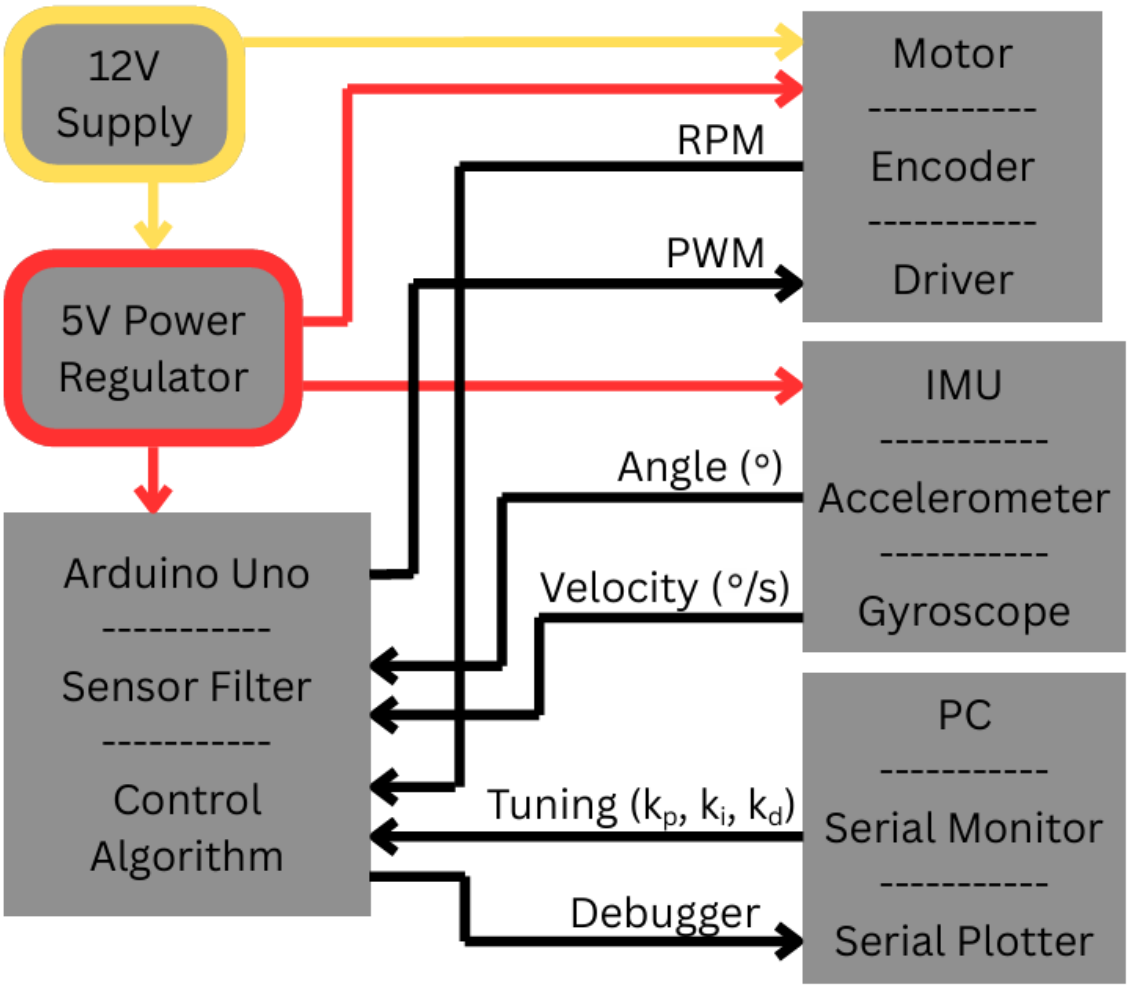


Figure 14. Project Block Diagram

At the top of the diagram, the 12V supply provides power to the motor through the driver. The supply voltage is stepped down to 5V through a regulator, which then in turn supplies the IMU and Arduino, and the motor's driver and encoder. A common ground between all components is necessary for correct communication.

The IMU communicates with the Arduino via the I2C (Inter-Integrated Circuit) interface on the A4 and A5 pins of the MCU. The I2C protocol allows one data line to communicate both the data from accelerometer and the gyroscope to the microcontroller. This enables both the angular position, in degrees, and angular velocity, in degrees per second, to be processed by a filter in the Arduino's sketch, firstly trying a Complementary filter, and then, depending on its performance, a Kalman filter. The motor, when moving, will introduce a lot of noise to the sensor data through vibrations, making accurate filtering vital for the project.

The Arduino Uno serves as the central control unit, processing sensor data and implementing the PID control algorithm. The Arduino generates a PWM and a digital signal to control the motor through the driver, adjusting the speed and direction of the Nidec 24H motor to adjust the torque of the reaction wheel. The frequency of the PWM signal is configured to match the requirements of the motor, best operating between 20-30kHz, ensuring efficient operation and precise control.

The encoder attached to the motor provides feedback on the wheel's RPM, which is sent to the Arduino through digital pulses. The count of pulses is converted into RPM by dividing the count by the PPR and the time. The encoder data, along with the IMU's angular position and velocity readings, allows the system to continuously fine-tune the motor's performance and maintain the pendulum's balance.

Communication with the control system is facilitated via the Serial Monitor and Serial Plotter on a connected PC, where the parameters can be tuned and their effect viewed in real-time. This interface provides insights into control system performance and allows for the adjustment of control parameters to ensure the optimal operation of the attitude correction system.

In summary, the block diagram demonstrates how the system components interact—providing power to the motor, collecting sensor data, executing control algorithms, and adjusting motor behaviour in real time to stabilize the pendulum. This architecture ensures efficient operation, precise control, and the ability to fine-tune system performance during testing.

3.2 Electrical Build

The electrical design for the reaction wheel inverted pendulum focuses on detailing the power distribution and communication.

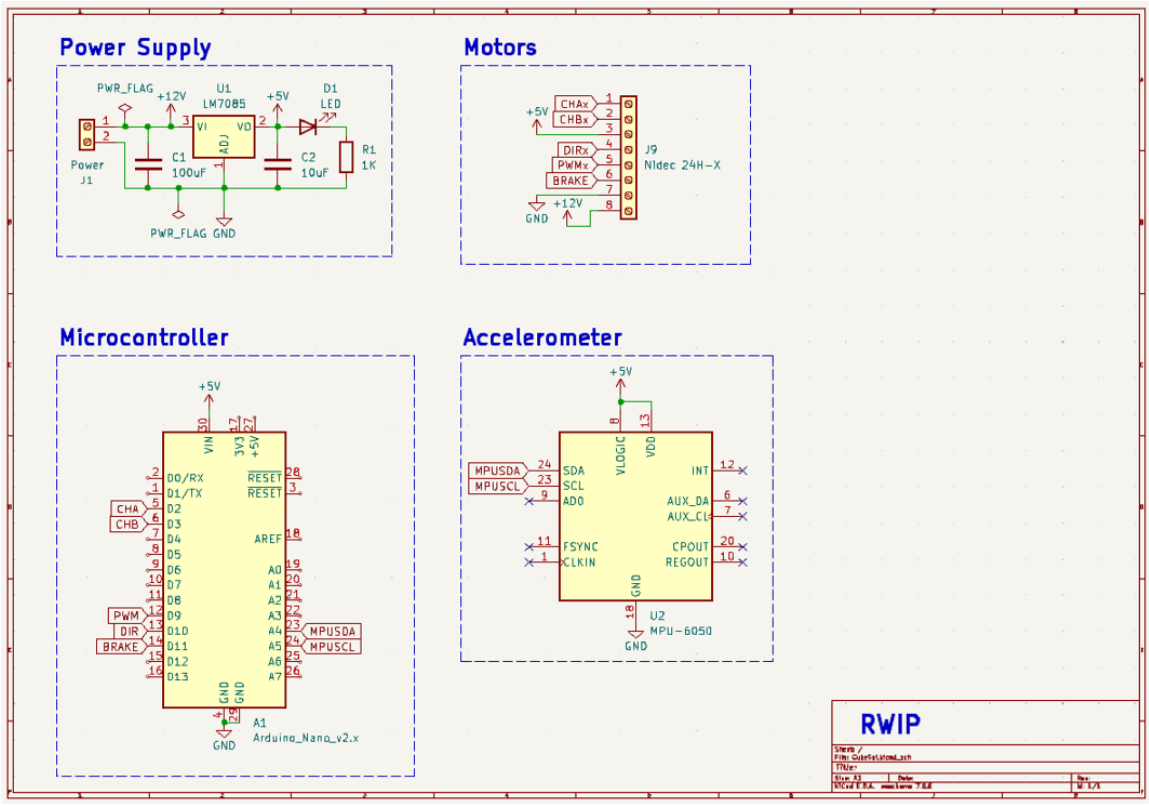


Figure 15. Electrical Schematic.

The system is powered by a 12V supply, supplies the Nidec 24H brushless DC motor. The supply is regulated through a LM7805 voltage regulator, stepping down the voltage to 5V to power the Arduino Uno, the MPU6050 IMU, and the motors encoder. To ensure stable operation, capacitors are placed on both the input and output sides of the LM7805: a 100µF capacitor on the input and a 10µF capacitor on the output, as recommended for optimal performance (Texas Instruments, 2016).

The Arduino Uno controls the motor through the PWM, Brake, and Direction lines, and it receives digital pulses from the encoder on Channel A and Channel B lines, and orientation data from the MPU6050 via the SDA and SCL pins, A4 and A5, which are connected to the respective data and clock lines for I2C communication (DFRobot, 2024).

Communication and tuning of the system are facilitated using the Serial Monitor through the USB-A port on the Arduino, where the control algorithm parameters are adjusted in real-time to fine-tune motor control and attain optimal performance.

This electrical design ensures effective power distribution and stable operation, with the necessary components to enable real-time control through the Arduino.

3.3 Mechanical Build

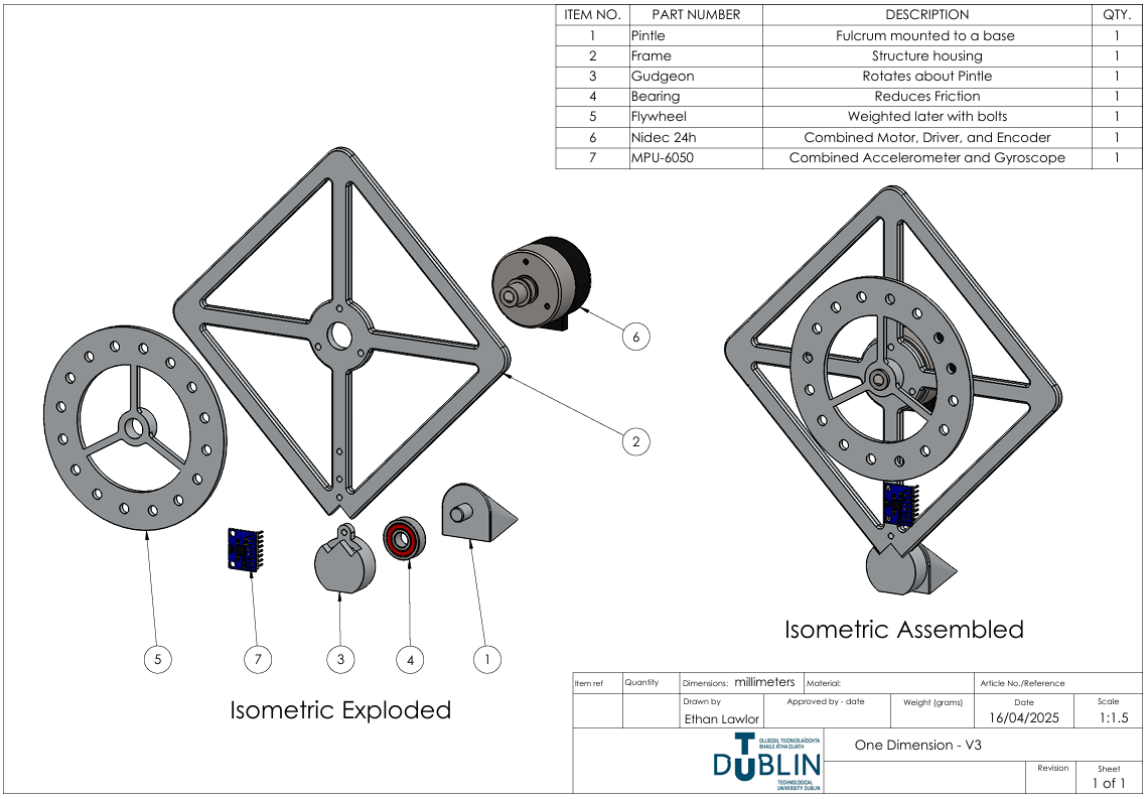


Figure 16. CAD Assembly Drawing.

The mechanical build consists of the flywheel, motor, IMU, frame, and pivot point. The Nidec24H, MPU6050, and bearing being COTS components while the wheel, frame, gudgeon, and pintle are designed using SolidWorks and 3D printed in PLA. Note the holes in the flywheel where bolts of varying lengths can be added to increase or decrease the mass, and thus the inertial moment, of the wheel.

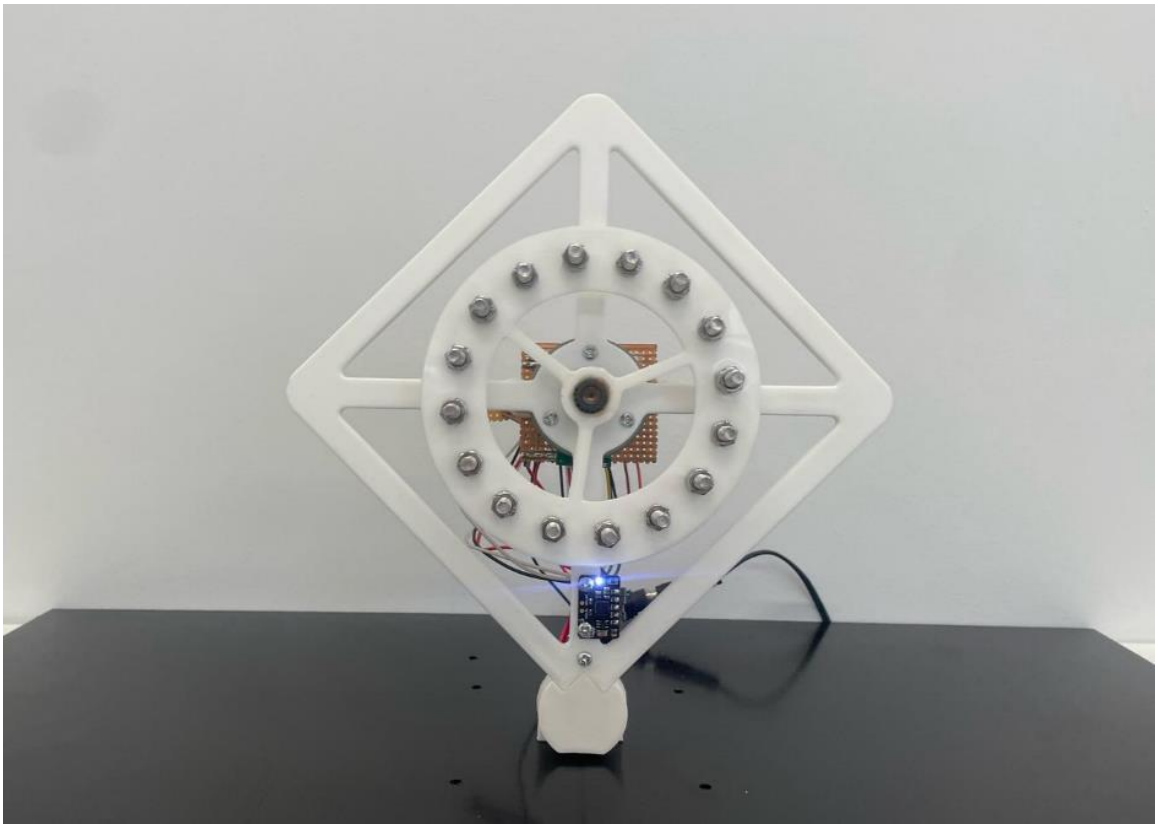


Figure 17. Project Mechanical Build.

3.3.1 Design Considerations

The reaction wheel inverted pendulum was designed to simulate an attitude correction system for CubeSats. Key design factors included weight distribution, wheel size, motor torque, IMU placement, and system stability. The system is built to CubeSat size, specifically a 1U form factor (10x10x10 cm), ensuring relevance for CubeSat applications.

A brushless DC motor, controlled via PWM signals, drives the reaction wheel, positioned near the centre of mass to minimize torque requirements and improve efficiency. The wheel size was selected to be the reasonably largest to better balance torque generation and power consumption, with a 90mm diameter providing sufficient torque while staying within the power budget.

The IMU, an MPU6050, is placed close to the centre of rotation to minimize vibrations and ensure accurate data. An Arduino Uno microcontroller processes this data and adjusts the motor speed in real-time, maintaining balance and simulating CubeSat attitude control.

The structure is constructed using 3D-printed components to minimize mass, which is critical for CubeSat applications. This compact design allows for easy adjustments during testing while reflecting the constraints CubeSats face in terms of space and power.

3.4 Simulink Simulation

In this section the mechanical build is modelled mathematically using Euler-Lagrange and simulated as a transfer function in Simulink. An appropriate control system can be modelled around this equation to verify if the system is indeed controllable based on the mechanical builds specific parameters, including weight, axis of rotation, and where the centre of mass is in relation to that axis.

3.4.1 Deriving the Equations of Motion

To derive the equation of motion for the system the Euler-Lagrange Equation will be used. This approach investigates the energy state transfer from kinetic to potential, describing it in forces in an equation of motion.

3.4.1.1 Structure Inertia and Energy:

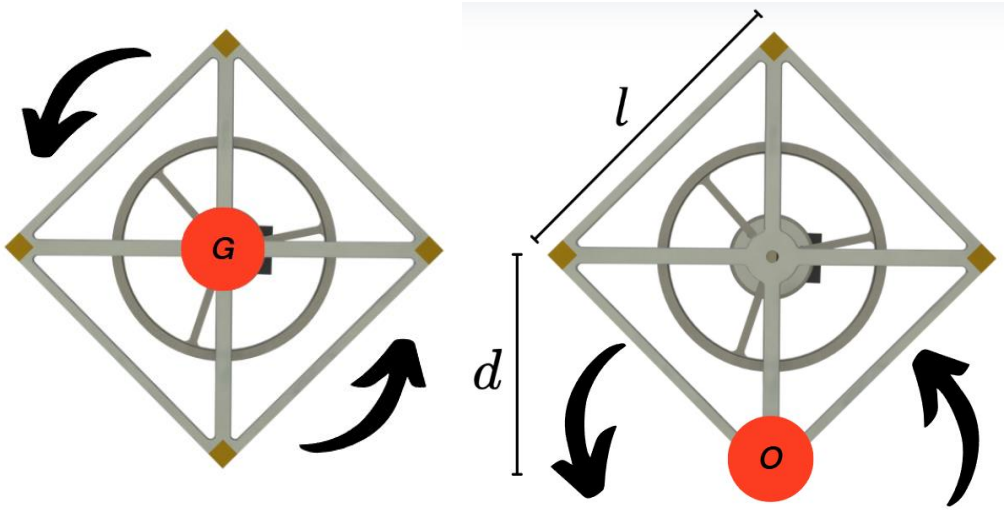


Figure 18. Parallel Axis Theorem Adjusts Inertia Based on Axis of Rotation.

Beginning with the structure, its inertia can be estimated as a uniform cuboid of side length l, where the depth is irrelevant.

Moment of Inertia of a Slab:

$$I = \frac{1}{12}m(a^2 + b^2) \tag{2.6}$$

$$I_sG = \frac{1}{12}m_s 2l^2 \tag{3.1}$$

Where $a^2 = b^2 = l^2$ for a cube.

$$I_sG = \frac{1}{6}m_s l^2 \tag{3.2}$$

Where I_sG is the moment of inertia of the structure about its centre of mass, m_s is the mass of the structure and l is the length of a side of the CubeSat's one-dimensional model. The Parallel Axis Theorem is now used to account for the difference in inertia from rotating about its centre to rotating about a vertex, where *d* is that distance. That distance is related to the length of the side of the cube through the basic trigonometry:

$$Sin\theta = \frac{A}{H} \tag{3.3}$$

$$Sin45° = \frac{d}{l} \tag{3.4}$$

$$d = \frac{l\sqrt{2}}{2} \tag{3.5}$$

Parallel Axis Theorem:

$$I_b = I_c + md^2 \tag{2.7}$$

$$I_sO = I_sG + m_s\left(\frac{l\sqrt{2}}{2}\right)^2 \tag{3.6}$$

$$I_sO = \frac{1}{6}m_s l^2 + m_s\left(\frac{l\sqrt{2}}{2}\right)^2 \tag{3.7}$$

$$I_sO = \frac{1}{6}m_s l^2 + m_s\left(\frac{2l^2}{4}\right) \tag{3.8}$$

$$I_sO = \frac{2}{12}m_s l^2 + \frac{6}{12}m_s l^2 \tag{3.9}$$

$$I_sO = \frac{2}{3}m_s l^2 \tag{3.10}$$

Where I_sO is the inertia of the structure about its vertex. Now that the inertia has been investigated, the energy of this body in the system can be modelled.

Kinetic Energy:

$$T = \frac{1}{2}I\omega^2 \tag{2.4}$$

$$T_s = \frac{1}{2}I_sO\dot{\theta}^2 \tag{3.11}$$

Where $\dot{\theta}$ is the angular velocity of the structure, $\omega = \theta/dt$, and I_sO is the moment of inertia of the structure about the axis of rotation.

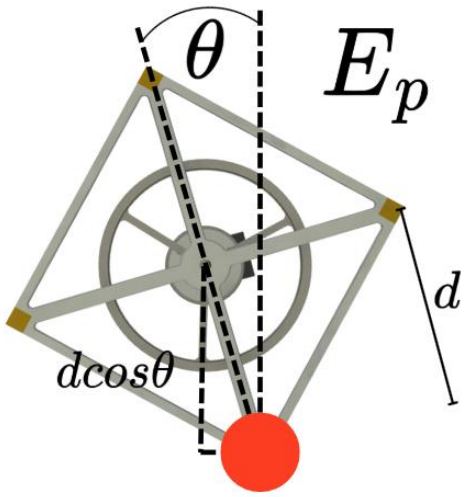


Figure 19. Varying Potential Energy Based on Position.

Potential Energy:

$$U = mgh \tag{2.5}$$

$$U_s = -m_s g d Cos\theta \tag{3.12}$$

Where m_s, is the mass of the structure excluding the wheel and motors rotor, d is the difference in length from the centre of mass to the axis of rotation, g is the gravitational constant and θ is the structures offset from the vertical plane in radians.

3.4.1.2 Wheel Inertia and Energy:

The same order of operations can be followed to determine an expression for the inertia, and thus the energy of the wheel.

Moment of Inertia of Annular Cylinder:

$$I = \frac{1}{2}m\left({R_1}^2 + {R_2}^2\right) \tag{2.1}$$

$$I_wG = \frac{1}{2}m_wR \tag{3.13}$$

Where I_wG is the moment of inertia of the wheel about its centre of mass, m_w is the mass of the wheel and R is the effective radius of the reaction wheels mass, i.e. ${R_1}^2 + {R_2}^2$.

Parallel Axis Theorem:

$$I_b = I_c + md^2 \tag{2.7}$$

$$I_wO = I_wG + m_s\left(\frac{l\sqrt{2}}{2}\right)^2 \tag{3.14}$$

$$I_wO = \frac{1}{2}m_wR + m_w\left(\frac{l\sqrt{2}}{2}\right)^2 \tag{3.15}$$

$$I_wO = \frac{1}{2}m_wR + m_w\frac{2l^2}{4} \tag{3.15}$$

$$I_wO = \frac{1}{2}m_wR + \frac{1}{2}m_wl^2 \tag{3.16}$$

$$I_wO = \frac{1}{2}m_w(R + l^2) \tag{3.17}$$

Where I_wO is the moment of inertia of the wheel about the axis of rotation of the system, m_w is the mass of the reaction wheel including the motors rotor, *R* is the effective radius of the rings mass, from Figure 6, and *l* is the length of the side of the cube.

Kinetic Energy:

$$T = \frac{1}{2} I \omega^2 \tag{2.4}$$

$$T_w = \frac{1}{2} I_w O \dot{\varphi}^2 \tag{3.17}$$

Where $\dot{\varphi}$ is the angular velocity of the wheel and I_wO is its moment of inertia.

Potential Energy:

$$U = mgh \tag{2.5}$$

$$U_w = -m_w g d Cos\theta \tag{3.18}$$

Where m_w, is the mass of the wheel, *d* is the difference in length from the centre of mass to the axis of rotation, *g* is the gravitational constant and θ is the structures offset from the vertical plane in radians.

3.4.1.3 The Lagrangian:

The Lagrangian is the difference between the sum of kinetic and sum of potential energies in the system and is the foundation of the Euler-Lagrange equation.

$$L = T - U \tag{3.19}$$

$$L = T_s + T_w - U_s - U_w \tag{3.20}$$

$$L = \frac{1}{2} I_s O \dot{\theta}^2 + \frac{1}{2} I_w O \dot{\dot{\varphi}}^2 - (-m_s g d Cos\theta) - (-m_w g d Cos\theta) \tag{3.21}$$

$$L = \frac{1}{2} I_s O \dot{\theta}^2 + \frac{1}{2} I_w O \dot{\dot{\varphi}}^2 + m_s g d Cos\theta + m_w g d Cos\theta \tag{3.22}$$

$$L = \frac{1}{2} I_s O \dot{\theta}^2 + \frac{1}{2} I_w O \dot{\varphi}^2 + (m_s + m_w) g d Cos\theta \tag{3.23}$$

Euler-Lagrange Equation:

$$\frac{d}{dt}\left(\frac{dL}{d\dot{\theta}}\right) - \frac{dL}{d\theta} = \tau_{reaction} \tag{2.13}$$

Expression in brackets:

$$\frac{dL}{d\dot{\theta}} = \frac{d}{d\dot{\theta}}\left(\frac{1}{2}I_s O\dot{\theta}^2 + \frac{1}{2}I_w O\dot{\varphi}^2 + (m_s + m_w)gdCos\theta\right) \tag{3.24}$$

$$\frac{dL}{d\dot{\theta}} = \frac{d}{d\dot{\theta}}\left(\frac{1}{2}I_s O\dot{\theta}^2\right) \tag{3.25}$$

$$\frac{dL}{d\dot{\theta}} = 2\left(\frac{1}{2}I_s O\dot{\theta}^1\right) \tag{3.26}$$

$$\frac{dL}{d\dot{\theta}} = I_s O\dot{\theta} \tag{3.27}$$

First term of equation:

$$\frac{d}{dt}\left(\frac{dL}{d\dot{\theta}}\right) = \frac{d}{dt}\left(I_s O\dot{\theta}\right) \tag{3.28}$$

$$\frac{d}{dt}\left(\frac{dL}{d\dot{\theta}}\right) = I_s O\ddot{\theta} \tag{3.29}$$

Second term of equation:

$$\frac{dL}{d\theta} = \frac{d}{d\theta}\left(\frac{1}{2}I_s O\dot{\theta}^2 + \frac{1}{2}I_w O\dot{\varphi}^2 + (m_s + m_w)gdCos\theta\right) \tag{3.30}$$

$$\frac{dL}{d\theta} = \frac{d}{d\theta}\left((m_s + m_w)gdCos\theta\right) \tag{3.31}$$

$$\frac{dL}{d\theta} = -(m_s + m_w)gdSin\theta \tag{3.32}$$

Resulting equation of motion for the system:

$$\therefore I_s O\ddot{\theta} + (m_s + m_w)gdSin\theta = \tau_{reaction} \tag{3.33}$$

I_sO is the moment of inertia of the structure about the axis of rotation of the system, and is $\ddot{\theta}$ is the angular acceleration of the structure, m_s is the mass of the structure, m_w is the mass of the wheel including the motors rotor, *g* is the gravitational constant, *d* is the distance from the centre of gravity to the pivot point, θ is the structures offset angle from the vertical axis, and $\tau_{reaction}$ is the reaction torque required by the wheel to align the structure vertically.

Thus, the reaction torque generated by the flywheel needs to overcome the torque of the structure moving about the axis of rotation, and the potential energy of the two bodies in their current position.

The resulting equation of motion can be simplified further by substituting I_sO with its expressing in terms of m_s and *l,* from Equation 3.10, along with *d* being substituted for its expression in terms of *l,* from Equation 3.5. Note that the resulting Equation 3.34 can be changed with Equation 3.33.

$$I_sO = \frac{2}{3}m_s l^2 \tag{3.10}$$

$$d = \frac{l\sqrt{2}}{2} \tag{3.5}$$

Substituting into the result of the Euler-Lagrange equation:

$$I_sO\ddot{\theta} + (m_s + m_w)gdSin\theta = \tau_{reaction} \tag{3.33}$$

$$\frac{2}{3}m_s l^2\ddot{\theta} + (m_s + m_w)gl\frac{\sqrt{2}}{2}Sin\theta = \tau_{reaction} \tag{3.34}$$

The values are substituted for one's measure from the mechanical build;

- m_w – mass of the wheel $= 0.036kg$.
- m_s – mass of the structure $= 0.156kg$.
- $l -$ is the length of the side of the cube $= 0.120m$.
- $\theta -$ is the position of the structure from vertical in rad.
- $\ddot{\theta} -$ is the angular acceleration of the structure in $rads^{-2}$, determined from taking a running sample from the IMU's accelerometer where $\ddot{\theta} = \dot{\theta}/dt$.
- $g -$ is the gravitational constant, on Earth $= 9.81\ ms^{-2}$.

3.4.2 Obtaining the Transfer Function

To graph the function in Simulink, its transfer function needs to be obtained.

The equation being used for ease is:

$$I_s O\ddot{\theta} + (m_s + m_w)gdSin\theta = \tau_{reaction} \tag{3.33}$$

For small-angle approximations, we can linearize the system by assuming that $Sin\theta \approx \theta$, when θ is a small value. This leads to the linearized equation:

$$I_s O\ddot{\theta} + (m_s + m_w)gd\theta = \tau_{reaction} \tag{3.35}$$

Laplace Domain:

$$I_s Os^2\theta(s) + (m_s + m_w)gd\theta(s) = \tau(s) \tag{3.36}$$

$$\theta(s)[I_s Os^2 + (m_s + m_w)gd] = \tau(s) \tag{3.37}$$

$$G(s) = \frac{\theta(s)}{\tau(s)} = \frac{1}{I_s Os^2 + (m_s + m_w)gd} \tag{3.38}$$

The denominator can be equated to Equation 3.33, and equally, Equation 3.34, leading to the following expression:

$$G(s) = \frac{1}{\frac{2}{3}m_s l^2 s^2 + (m_s + m_w)gl\frac{\sqrt{2}}{2}} \tag{3.39}$$

Substituting the known parameters and simplifying

$$G(s) = \frac{1}{\frac{2}{3}(0.156)(0.120^2)s^2 + (0.156 + 0.036)(9.81)(0.120)\frac{\sqrt{2}}{2}} \tag{3.40}$$

$$G(s) = \frac{1}{0.0015s^2 + 0.1598} \tag{3.41}$$

$$G(s) = \frac{k}{\frac{1}{{\omega_n}^2}s^2 + \frac{2\xi}{\omega_n}s + 1} \tag{3.42}$$

Comparing this to the general form of a second order transfer function shown in Equation 3.42, the need to divide each term on the right-hand side by 0.1598 in apparent to bring the constant in denominator to 1:

$$G(s) = \frac{6.267}{0.009s^2 + 1} \tag{3.43}$$

Plugging this function into a Simulink model give the following result:

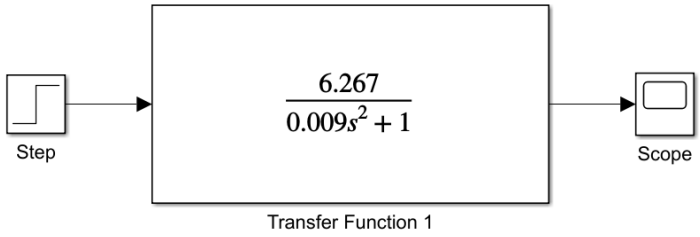


Figure 20. Simulink Model of Equation 3.43.

Figure 21 shows that the system is inherently unstable without control. The system oscillates indefinitely, never settling at the desired output of 1, this reveals the necessity of a control algorithm to direct the output towards stability.

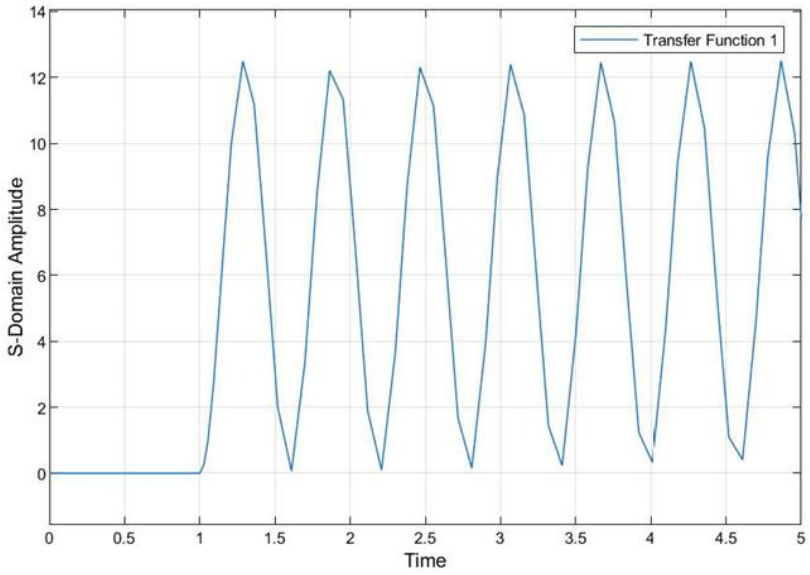


Figure 21. Step-Response to Transfer Function 1.

3.4.3 Adding Derivative Control

The first term implemented from PID control, from Equation 2.16, is derivative control. Derivative control helps reduce overshoot and improves stability by predicting the future state of the system from the previous error order:

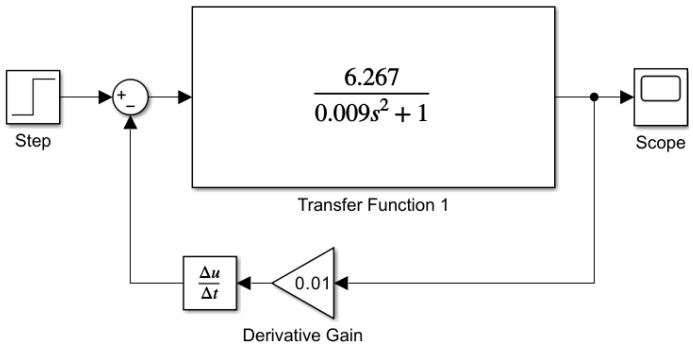


Figure 22. Derivative Control of System.

Figure 23 shows that a derivative gain can produce an inaccurate yet stable output. The output settles in a reasonable time but fails to approach the desired output. The need for more control is evident.

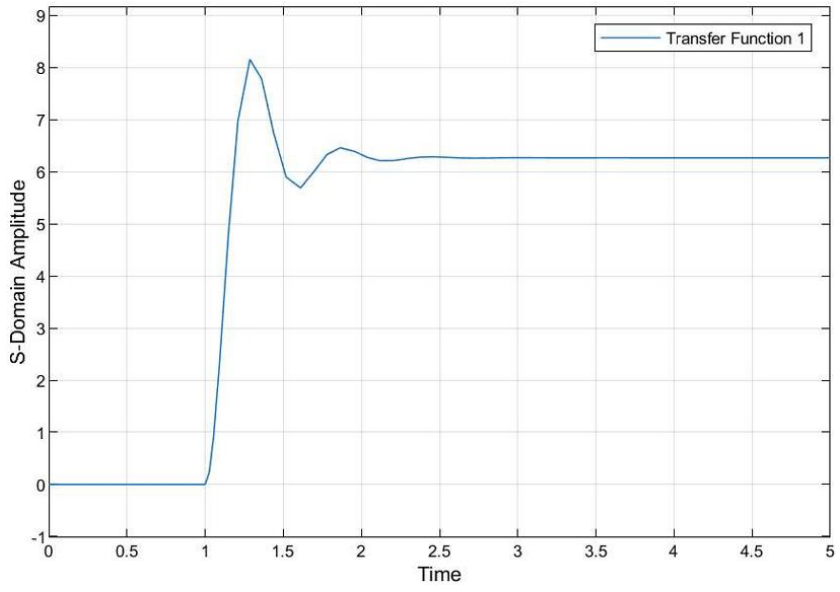


Figure 23. Step-Response to Derivative Control

3.4.4 Adding Proportional Control

The P from PID control, from Equation 2.16, is proportional control. Proportional control helps reduce reduces large errors quickly, by providing a control output that is directly proportional to the size of the error:

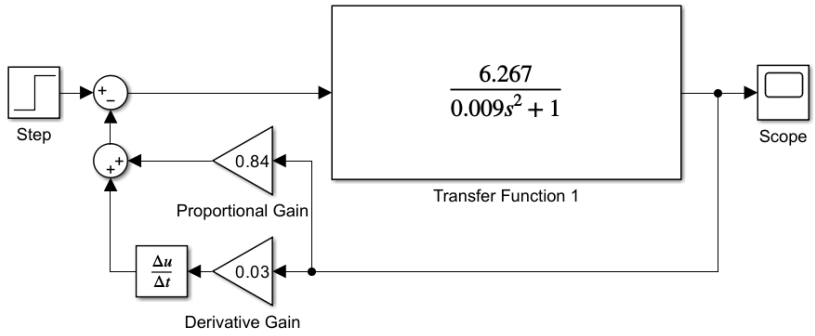


Figure 24. Proportional-Derivative Control of System.

Figure 25 shows how the inclusion of proportional control has drastically diminished the initial error, and thus the error of every timestep following it. The combination of proportional and derivative gain has resulted in achieving the desired steady state output in a controlled and steady manner.

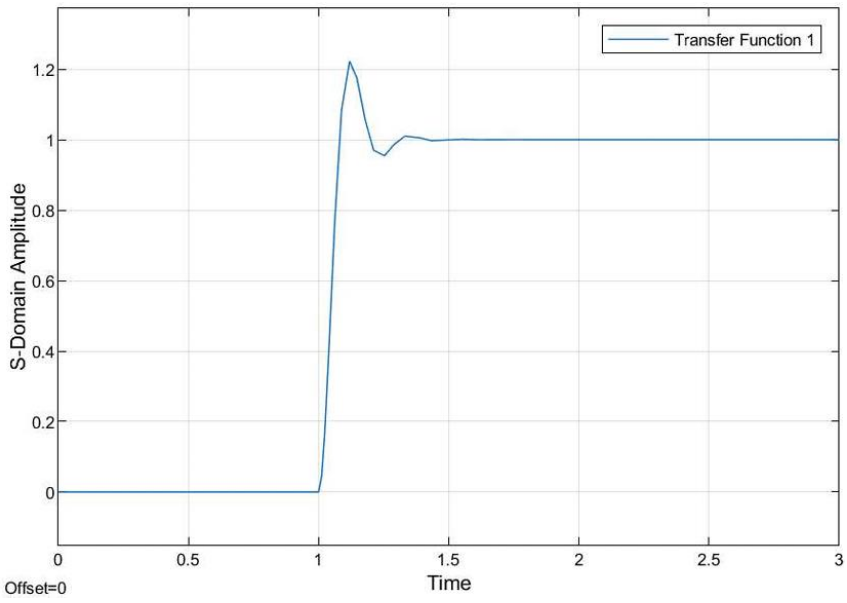


Figure 25. Step-Response of Proportional-Derivative Control.

3.4.5 Simulation Results and Conclusions

The model proposed in Figure 25 yielded a satisfactory output, indicating that the inherently instable system represented by the mechanical design, with its specific masses and moments of inertia for the wheel and structure, can be controlled appropriately.

3.5 Software Design

With the mechanical design being verified through means of mathematical simulation, the software implementation can begin. This section shows how to integrate the motor's driver and encoder, and the IMU's accelerometer and gyroscope, with the microcontroller's code, to be used in the control algorithm.

3.5.1 Software Flow Charts

The flow charts in this section offer an overview of the flow of data and the decision-making process of the software for this project.

3.5.1.1 Single Loop Control

The flow chart in Figure 26 shows how the Simulink block diagram, from Figure 24, can be achieved with coding logic, a target value is set, a sensor gives input data allowing an error to be determined, the PID control system applies its gains to the various error interpretations and a value is calculated for the motor speed. This change in motor speed leads to a change in angle and ultimately a change in error for the next iteration of the loop.

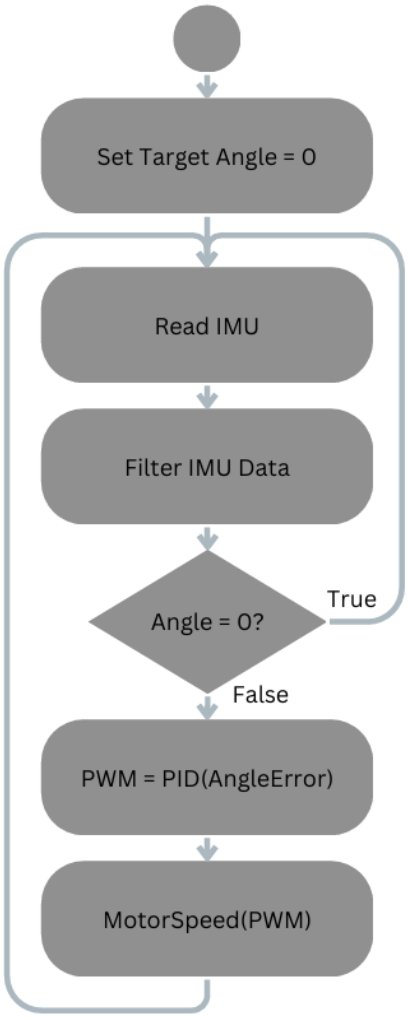


Figure 26. Single Loop PID Control.

3.5.1.2 Cascade Loop Control

The single loop has been verified to work in a simulated virtual environment but this does not guarantee success so an alternative must be prepared. A dual PID controller, often called a cascade loop, uses an inner loop to control the speed of the motor and an outer loop to control the angle of the system.

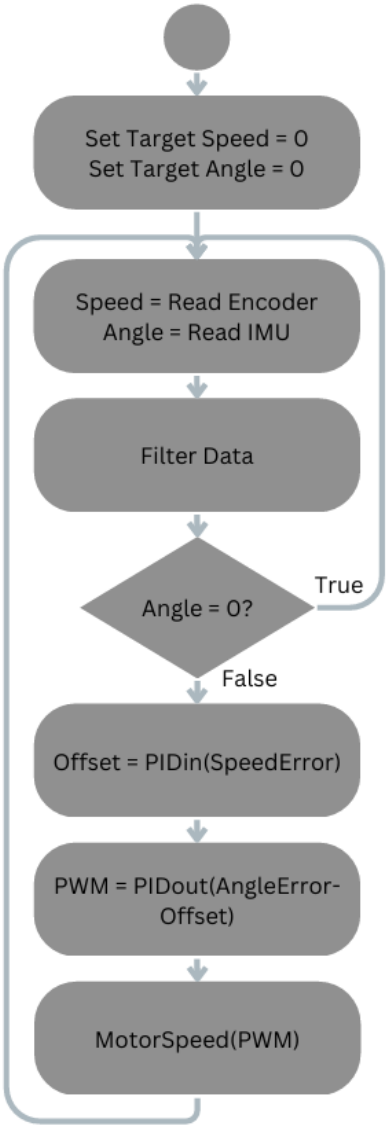


Figure 27. Cascade Loop PID Control.

The inner loop adjusts the setpoint of the outer loop, this increases the responsiveness of the outer loop and limits the output speed of the motor preventing its saturation, an effect where the wheel is spinning so fast a large change in velocity can't be induced to correct the output angle.

3.5.2 Motor Driver

The motor driver is inbuilt into the Nidec 24H motor, it is very simple in that it takes a binary signal, High or Low, for the direction and brake and a PWM signal, 0-255, for the motor speed. This specific motor works best when its pulse width modulated duty cycle is modulated at a

frequency of 20-30kHz. This is attained through use of a library to change the Arduino's default frequency for PWM signals of 480Hz to the desired rate.

```
#include <PWM.h>  // Include the PWM library to control PWM outputs
#define Nidec_Frequency 30000  // Set the PWM frequency to 30kHz
#define motorPWM_Pin 9  // Define the PWM pin for motor control
#define motorDIR_Pin 10  // Define the direction pin for motor control
#define motorBrake_Pin 11  // Define the brake pin for motor control

void setup() {
  // put your setup code here, to run once:

  InitTimersSafe();  // Initialize safe timer operations for PWM control
  SetPinFrequencySafe(motorPWM_Pin, Nidec_Frequency);  // Set PWM frequency for the motor

  pinMode(motorPWM_Pin, OUTPUT);  // Set motorPWM_Pin as an output pin
  pinMode(motorDIR_Pin, OUTPUT);  // Set motorDIR_Pin as an output pin
  pinMode(motorBrake_Pin, OUTPUT);  // Set motorBrake_Pin as an output pin

  pwmWrite(motorPWM_Pin, 255);  // Set the motor's PWM signal to zero power (255)
  digitalWrite(motorDIR_Pin, HIGH);  // Set the motor's direction to forward (HIGH)
  digitalWrite(motorBrake_Pin, LOW);  // Enable the brake (LOW)
}
```

Figure 28. Motor Driver Code.

Once the motor's brake, direction and pwm pins are assigned and declared as outputs, control is achieved from the following:

Table 1. Nidec24H Motor Pin Controls.

motorPWM_Pin	0	Max Speed	255	Min Speed
motorDir_Pin	LOW	Clockwise	HIGH	Counterclockwise
motorBrake_Pin	LOW	Brake Engaged	HIGH	Brake Disengaged

Note that the output of the control algorithm will have to be contained between 0 and 255 for the motor speed. It will also have to be inverted where the largest output of the control algorithm maps to 0 and the lowest output aligns with a 255 input for the motor to speed up and slowdown in the correct direction. This is seen later in Figure 41.

3.5.3 Encoder

The encoder is also inbuilt into the Nidec 24H motor, it has 2 lines which both send a signal and need to be declared as inputs.

```
#define CHA 2  // Define pin 2 for encoder A (CHA)
#define CHB 3  // Define pin 3 for encoder B (CHB)

struct encoderData {  // Struct to hold encoder data (RPM and direction)
  float rpm;  // Motor RPM (revolutions per minute)
  int direction;  // Motor direction (1 for clockwise, -1 for counterclockwise)
};

void setup() {
  // put your setup code here, to run once:
  pinMode(CHA, INPUT_PULLUP); // Set pin 2 (CHA) as input with internal pull-up resistor
  pinMode(CHB, INPUT_PULLUP); // Set pin 3 (CHB) as input with internal pull-up resistor
  // Attach interrupts to pins CHA and CHB for detecting rising edges of the encoder signals
  attachInterrupt(digitalPinToInterrupt(CHA), ai0, RISING);  // Interrupt on pin 2 (encoder A)
  attachInterrupt(digitalPinToInterrupt(CHB), ai1, RISING);  // Interrupt on pin 3 (encoder B)
}
```

Figure 29. Encoder Set Up Code.

The device has two optical encoders, on two digital input lines, that are out of phase with each other. These encoders are triggered one after the other in one direction, then, once the direction is reversed, the initial signal that was lagging becomes the leading signal, allowing the code to determine the direction as well as the angular velocity of the reaction wheel. This logic is exercised in the two interrupt service routines, seen in Figure 30. The use of interrupts allows the processor to effectively multi-task by pausing its main program execution to execute a service routine.

```
void ai0() {
  // Interrupt for pin 2 (A signal from encoder)

  if (digitalRead(CHB) == LOW) {
    counter++;  // Increment counter if B signal is LOW (forward direction)
  } else {
    counter--;  // Decrement counter if B signal is HIGH (reverse direction)
  }
}
void ai1() {
  // Interrupt for pin 3 (B signal from encoder)

  if (digitalRead(CHA) == LOW) {
    counter--;  // Decrement counter if A signal is LOW (reverse direction)
  } else {
    counter++;  // Increment counter if A signal is HIGH (forward direction)
  }
}
```

Figure 30. Encoder Interrupt Service Routines.

The counter value is passed to a custom function on each iteration of the main loop to determine the speed and the direction, from its current value, previous value, and the time between function calls, factoring in the pulses per revolution.

```
encoderData getEncoderData() {
  unsigned long currentMillis = millis();  // Get the current time in milliseconds
  unsigned long time = (currentMillis - lastMillis);  // Calculate time difference
  lastMillis = currentMillis;  // Update lastMillis to the current time

  long count = (counter - lastCounter);  // Calculate the change in counter since the last measurement
  lastCounter = counter;  // Update lastCounter to the current counter value

  float seconds = (time / 1000.0);  // Convert time difference from milliseconds to seconds
  float revolutions = count / 200.0;  // Convert counter value to revolutions (200 pulses per revolution)
  float rps = revolutions / seconds;  // Calculate revolutions per second (RPS)

  // Determine the direction of rotation based on counter value
  int encDIR;
  if (counter > lastCounter) {
    encDIR = 1;  // Clockwise
  } else {
    encDIR = -1;  // Counterclockwise
  }

  // Store and return the encoder data (RPM and direction)
  encoderData result;
  result.rpm = rps * 60;  // Convert RPS to RPM (multiply by 60)
  result.direction = encDIR;  // Store direction

  return result;  // Return the encoder data
}
```

Figure 31. Encoder Speed and Direction Code.

The results are stored in a struct variable, a class of variable container that can be arranged to hold multiple variable of different data types as seen in Figure 29. This allows the main loop to call this function to update the values in a struct to be used in calculations or printed to the serial monitor.

```
void loop() {
  // put your main code here, to run repeatedly:
  encoderData encoderData = getEncoderData();  // Get encoder data (RPM and direction)
  // Print RPM and direction to the Serial Monitor
  Serial.print("RPM:"); Serial.print(encoderData.rpm); Serial.print(",");
  Serial.print("Direction:"); Serial.println(encoderData.direction ? "Clockwise" : "Counterclockwise");
}
```

Figure 32. Using the Encoder Custom Function.

3.5.4 IMU

The microcontroller uses its I2C bus to communicate with the MPU6050's accelerometer and gyroscope. This is facilitated by a library that's abstracts the use of addresses and bit registers, simplifying the code. This is used in conjunction with a library for the specific MPU6050 IMU that avoids involves bit masking and shifting of these addresses.

```
#include <Wire.h>          // Include the Wire library for I2C communication
#include <MPU6050.h>        // Include the MPU6050 library

MPU6050 mpu;
int16_t ax, ay, az, gx, gy, gz;
float gz_offset = 0;

struct mpuData {
  float theta_z;
  float omega_z;
};

void setup() {
  // put your setup code here, to run once:
  Wire.begin();
  mpu.initialize();
  mpuCalibrate();
}
```

Figure 33. MPU6050 Set Up Code.

A sample of values need to be taken when the sensor is still to calibrate its reading. Several sensor readings are taken and their average found. This average is then subtracted from each reading, when still this results in output values closer to zero, which is ideal for a stationary sensor that reads motion.

```
void mpuCalibrate() {
  int num_samples = 1000;  // Take the sum of n samples
  for (int i = 0; i < num_samples; i++) {
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
    gz_offset += gz;
    delay(2);
  }

  // Offset equals the average of the sample i.e. the average distance it is away from origin
  gz_offset /= num_samples;

  Serial.print("Gyro_Z Offset: ");  Serial.println(gz_offset);
}
```

Figure 34. MPU6050 Calibration Code.

A custom function is then implemented to call on the IMU to update sensor reading values. These values are scaled appropriately and used to calculate the angle and velocity of the system. The accelerometer is ranged to the highest sensitivity option, ±2g, for the slow falling system, and since its output is in radians, it is multiplied by 180/PI to convert it into degrees. To determine the system's orientation, the code $float\ theta\_z\ =\ atan2(Ay, Ax)$ is used to compute the angle between the 2D vector formed by the Ax and Ay components, with respect to the positive x-axis, once the correct quadrant has been considered. The gyroscope is also scaled to the highest sensitivity range of ±250°/s while being orientated correctly, and then the previously calculated value from the calibration function is subtracted to get a more accurate reading.

```
mpuData getMpuData() {
  const float ACCEL_SCALE = 16384.0;  // ±2g range
  const float GYRO_SCALE = 131.0;     // ±250°/s range
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);  // Read sensor data

  // Convert accelerometer raw values to g-force (after offset correction)
  float Ax = (ax) / ACCEL_SCALE;
  float Ay = (ay) / ACCEL_SCALE;
  // Compute tilt angles (theta_x, theta_y, theta_z) in degrees
  float theta_z = ((atan2(Ay, Ax) * 180.0) / PI);

  // Convert gyroscope raw values to angular velocity in °/s (after offset correction)
  float omega_z = (gz - gz_offset) / -GYRO_SCALE;

  mpuData result;
  result.theta_z = theta_z;
  result.omega_z = omega_z;

  return result;
}
```

Figure 35. MPU6050 Angle and Angular Velocity Code.

These values for the angular position and velocity are then stored in a struct to be used by the main loop. Again, the main loop creates a new ‘mpuData’ struct variable called ‘mpuData’, and calls ‘getMpuData()’ to fill the struct with the angle and velocity values, identical to the encoder data exchange from custom function to main loop.

```
void loop() {
  // put your main code here, to run repeatedly:
  mpuData mpuData = getMpuData();
  // Print Angle and Velocity to the Serial Monitor
  Serial.print("Theta_z:"); Serial.print(mpuData.theta_z); Serial.print(",");
  Serial.print("Omega_z:"); Serial.println(mpuData.omega_z);
}
```

Figure 36. Using the MPU6050 Custom Function.

3.5.5 Complementary Filter

The complementary filter is executed in the main loop. It takes both the angle and the acceleration and combines them in accordance with Equation 2.14. This combines a large amount from the gyroscope reading and a small amount from the accelerometer reading to give an output angle that is less susceptible to noise to and drift.

```
void loop() {
  // put your main code here, to run repeatedly:
  mpuData mpuData = getMpuData();

  unsigned long compTime = millis() - compTimeLast; compTimeLast = millis();
  angle = alpha*(angle + (mpuData.omega_z * (compTime/1000.0))) + ((1-alpha)*mpuData.theta_z);

  // Print Angle and Filtered Angle to the Serial Monitor
  Serial.print("Theta_z:"); Serial.print(mpuData.theta_z); Serial.print(",");
  Serial.print("Angle:"); Serial.println(angle);
}
```

Figure 37. Complementary Filter Code.

3.5.6 Kalman Filter

The Kalman filter runs in the main loop. It takes the predicted angle and the measured angle, blending them using the Kalman gain, which is calculated based on the uncertainty of both the prediction and the measurement. The filter gives more weight to the gyroscope for quick changes, while the accelerometer helps correct drift and noise. The uncertainty decreases as the filter trusts the measurements more, leading to a more accurate and stable angle estimate over time. This dynamic adjustment between the gyroscope and accelerometer ensures that the filter adapts to the noise levels and provides reliable results.

```
void kalman_1d(float KalmanState, float KalmanUncertainty, float KalmanInput, float KalmanMeasurement) {
  KalmanState = KalmanState + 0.004 * KalmanInput;  // Predict the new state (angle) based on the rate input

  KalmanUncertainty = KalmanUncertainty + 0.000256;  // Update uncertainty based on the input rate (process noise)

  float KalmanGain = KalmanUncertainty / (KalmanUncertainty + 9);  // Calculate Kalman Gain

  KalmanState = KalmanState + KalmanGain * (KalmanMeasurement - KalmanState);  // Update state from measurement and Kalman Gain

  KalmanUncertainty = (1 - KalmanGain) * KalmanUncertainty;  // Update uncertainty after measurement

  // Store the filtered state (angle) and uncertainty in the output array
  Kalman1DOutput[0] = KalmanState;
  Kalman1DOutput[1] = KalmanUncertainty;
}
```

Figure 38. Kalman Filter Custom Function.

The main loop calls this function after communicating with the IMU to filter its data into an accurate sensor reading free of noise, acceptable for use in situations where fine control is needed. The logic of this function is based on one used by an open-source DIY drone created by Carbon Aeronautics and is sufficient for flight controls, proving its effectiveness in turbulent environments (CarbonAeronautics, 2022).

```
void loop() {
  // Main loop for continuous operation
  mpuData mpuData = getMpuData();  // Get data from the MPU (accelerometer/gyroscope)

  // Apply the Kalman filter to the Yaw angle and update its state and uncertainty
  kalman_1d(KalmanAngleYaw, KalmanUncertaintyAngleYaw, mpuData.omega_z, mpuData.theta_z);

  // Update Kalman Angle and Uncertainty for Yaw from the filter output
  KalmanAngleYaw = Kalman1DOutput[0];
  KalmanUncertaintyAngleYaw = Kalman1DOutput[1];

  // Print the filtered Yaw angle and the raw angle to the Serial Monitor
  Serial.print("KalmanAngleYaw:"); Serial.print(KalmanAngleYaw); Serial.print(",");
  Serial.print("Angle:"); Serial.println(angle);  // Print the raw angle value (from sensors)
}
```

Figure 39. Using the Kalman Filter Custom Function

3.5.7 PID Control

The filtered angle is processed by a PID control algorithm to determine an output for the motor. The error is defined as the difference between the filtered angle and the desired angle. Proportional, integral, and derivative gains are applied to their various evaluations of the error, Equation 2.16.

```
float PID(float angle){
  unsigned long currentTime = millis();
  float time =(currentTime - lastTime)/1000; // Calculate Timestep
  lastTime = currentTime;

  float error = setPoint - angle; // Error for P Gain

  errSum += error * time; // Error Sum for I Gain
  errSum = constrain(errSum, -50, 50); // Constrain Value

  float errOrder = (error - lastErr) / time; // Error Order for D Gain
  errOrderF = (0.9 * errOrderFprev) + (0.1 * errOrder); // Low Pass Filter
  errOrderFprev = errOrderF;
  lastErr = error;

  float pTerm = kP * error;
  pTerm = constrain(pTerm, -255, 255);
  float iTerm = kI * errSum;
  float dTerm = kD * errOrderF;

  float controlSignal = pTerm + iTerm + dTerm;
  controlSignal = constrain(controlSignal, -255, 255);

  return controlSignal;
}
```

Figure 40. PID Control Code.

The proportional term has its gain applied directly to the error. This term is constrained to the max motor output value, in either direction, this helps with responsiveness to the effect of other terms whereby if the value was excessively large the other terms values would be negligible. The integral term has it gain applied to the sum all previous errors. The error sum is also constrained to cap its effect on the output. The derivative term applies its gain to the rate of change of the error over time, or the error order. A low pass filter is applied to the error order, this allows the value to move steadily without large deviations overly effecting the output.

```
void loop() {
  // put your main code here, to run repeatedly:
  mpuData mpuData = getMpuData(); // Get Data from the MPU

  angle = filter(mpuData.theta_z, mpuData.omega_z); // Apply Filter (i.e. complementary or kalman)

  controlSignal = PID(angle);  // Apply PID Control

  digitalWrite(motorDIR_Pin, (controlSignal >= 0) ? 0 : 1);
  analogWrite(motorPWM_Pin, (255-abs(int(controlSignal))));
  digitalWrite(motorBrake_Pin, HIGH);

}
```

Figure 41. Using the PID Custom Function.

The combination of the different terms given as a control signal to the motor. Angle deviation could be positive or negative so the resulting value from the PID control could also be either. As the motor only accepts positive integer values and moves at top speed when the input signal is 0, the following conversions need to be done. Assume the output of the PID control is -255, requiring max speed in the counterclockwise direction, considering Table 1 for Nidec 24H motor control:

- -255<0, therefore the motor direction is set to counterclockwise.
- The motor driver only accepts positive integers, the absolute value of -255 is 255.
- 255 is minimum speed but 255-255=0 gives the maximum speed.

This logic works with any value from -255 to +255:

- Control signal = +5. This is greater than zero, so direction is clockwise.
- The absolute value is +5.
- 255-5=250, resulting in a slow motor speed.

Both this code implementation of the PID control algorithm and the following tuning function and cascade loop are highly inspired by Pavlo Bohdan, fellow Technological University Dublin alumni (Bohdan, 2016).

3.5.8 Cascade Loop

In the cascade loop, two PID control functions are nested one within the other. The inner speed control loop works off the RPM of the wheel to determine a setpoint for the outer orientation control loop. The output from the encode has a low-pass filter to prevent large deviations or potential noise, leading to a smoother input for the speed PID control loop.

```
void loop() {
  // put your main code here, to run repeatedly:
  mpuData mpuData = getMpuData(); // Get Data from the MPU
  angle = filter(mpuData.theta_z, mpuData.omega_z); // Apply Filter (i.e. complementary or kalman)

  encoderData encoderData = getEncoderData(); // Get Data from the Encoder
  rpm = rpm - (beta * (rpm - encoderData.rpm));  // Apply Low Pass Filter

  controlSpeed = speedPID(rpm); // Apply Inner Speed PID

  controlSignal = anglePID(controlSpeed, angle); // Apply Outer Angle PID

  digitalWrite(motorDIR_Pin, (controlAngle >= 0) ? 0 : 1);
  analogWrite(motorPWM_Pin, (255-abs(int(controlSignal))));
  digitalWrite(motorBrake_Pin, HIGH);

}
```

Figure 42. Using a Cascaded PID Loop.

The error for the outer loop is the speed of the wheel, the larger the speed the bigger the error. This alters the setpoint of the inner loop by a certain degree, increase the response and effectiveness of the inner loop's gains.

```
float sError = sTarget - rpm; // Inner Loop Error

float aError = controlSpeed - angle; // Outer Loop Error
```

Figure 43. Errors for Inner and Outer Loops.

3.5.9 PID Tuning

The Arduino's has non-volatile memory, memory that doesn't erase after power down, called Electrically Erasable Programmable Read-Only Memory, or EEPROM for short. It is not innately capable of storing float variables, so a library is utilised to spread the value of the float over four storage addresses, enabling it to be written to and read from the EEPROM.

```
#include <EEPROM.h>
float kP, kI, kD; // Declare Float Variables for Gains

void setup() {
  // put your setup code here, to run once:
  EEPROM.get(0, kP); // Read kP (float) from EEPROM starting at address 0
  EEPROM.get(4, kI); // Read kI (float) from EEPROM starting at address 4
  EEPROM.get(8, kD); // Read kD (float) from EEPROM starting at address 8

  // Print Values to Serial Monitor
  Serial.print("kP = "); Serial.print(kP, 3); Serial.print(" , ");
  Serial.print("kI = "); Serial.print(kI, 3); Serial.print(" , ");
  Serial.print("kD = "); Serial.println(kD, 3);

}

void loop() {
  if (Serial.available()){ // When Command Detected
    tunePID(); // Match Command to Value
  }

  PWM = PID(angle); // Apply PID Control

}
```

Figure 44. Using PID Tuning Custom Function.

A character is sent from the serial monitor by the user, the main loop detects the available data and calls the PID tuning function. The tuning function reads the available character and executes a specific routine based on the characters corresponding switch case. The gains can be varied individually or written to all at once to clear them, program parameters can be altered in this way also to start or stop the motor or print values for debugging. A default case is executed if no match is found for the command character

```
void tunePID(){
  char cmd = Serial.read(); // Read Character from Serial Monitor

  switch (cmd){
    case 'q': kP += 1; EEPROM.put(0, kP); // Increment P Gain
    break;

    case 'w': kI += 1; EEPROM.put(4, kI); // Increment I Gain
    break;

    case 'e': kD += 1; EEPROM.put(8, kD); // Increment D Gain
    break;

    case 'z':
      kP = 0; kI = 0; kD = 0; // Reset Gains
      for (int i = 0; i < 12; i++){
        EEPROM.write(i, 0); // Clear Values in EEPROM
      }
    break;

    case 'l': run = !run; // Toggle Status of Run Boolean
    break;

    default: Serial.println("<Error - no tuning value>");
    cmd = Serial.read(); // Wipe Current Command
  }
}
```

Figure 45. PID Tuning Code

3.5.10 Overview

The final version of the project code included the Kalman filter to combine the MPU6050's accelerometer and gyroscope readings into an accurate angular position, a Cascaded PID control loop where the inner loop adjusts the set point for the outer loop based on the speed of the reaction wheel, an IF statement that checks for acceptable run and position values before moving the motor, a custom function that accepts commands through the Serial Monitor to change PID gains, start and stop the motor, and to start and stop streaming data, and a watchdog timer that only sends data to be streamed after an acceptable interval.

```
void loop() {
  if (Serial.available()){ // Detects Serial Data
    tunePID(); // Check for Commands In Serial Data
  }

  mpuData mpuData = getMpuData(); // Get Angle from the MPU
  // Apply kalman Filter
  kalmanData kalmanData = kalman(kalmanData.angle, kalmanData.uncertainty, mpuData.omega_z, mpuData.theta_z);

  encoderData encoderData = getEncoderData(); // Get Speed from Encoder
  rpm = rpm - (beta * (rpm - encoderData.rpm)); // Apply Low Pass Filter

  float setPoint = speedPID(encoderData.rpm); // Apply Inner Speed PID
  float controlSignal = anglePID(setPoint, kalmanData.angle); // Apply Outer Angle PID

  if (run && abs(kalmanData.angle) < 20){ // Run Command and Upright?
    digitalWrite(motorDIR_Pin, (controlSignal >= 0) ? 0 : 1); // Set Direction
    analogWrite(motorPWM_Pin, (255-abs(int(controlSignal)))); // Set Speed
    digitalWrite(motorBrake_Pin, HIGH); // Disengage Brake
  }else{                              // No Run Command or Angle Too Large?
    analogWrite(motorPWM_Pin, 255); // Minimum Speed
    digitalWrite(motorBrake_Pin, LOW); // Engage Brake
  }

  unsigned long currentMillis = millis(); // Get the current time
  // At Least 60ms Since the Last Print?
  if (currentMillis - lastStreamTime >= streamInterval) {
    if (stream){ // Stream Data Command?
    Serial.print("SET:"); Serial.print(0); Serial.print(",");
    Serial.print("Angle:"); Serial.print(kalmanData.angle, 4); Serial.print(",");
    Serial.print("RPM:"); Serial.print(rpm, 4); Serial.print(",");
    Serial.print("PIDspeed:"); Serial.print(setPoint, 4); Serial.print(",");
    Serial.print("PIDangle:"); Serial.println(controlSignal, 4);
    }
    lastStreamTime = currentMillis; // Update the last execution time
  }
}
```

Figure 46. Overview of Project Code.

The watchdog timer stops the program from being slowed down by excessive communication. It was found that the printing data on every loop iteration would prevent the sensor reading from moving smoothly as the microcontroller would miss live data as it was busy printing data.

Chapter 4 Results and Discussion

This chapter offers an in-depth analysis and discussion of the various results derived from the methods and techniques implemented throughout the project. The data is presented through visual figures, and a comparison of the different methods and their outcomes is provided.

4.1 Complementary Filter

The complementary filter is designed to take a high sample rate from the gyroscope, to reduce sensor noise, and a low sample rate from the accelerometer, to reduce sensor drift. The ratio between these sample rates is called the alpha coefficient, from Equation 2.14, and is best defined through experimentation on the specific rig. The results for this mechanical build are shown in Figure 47.

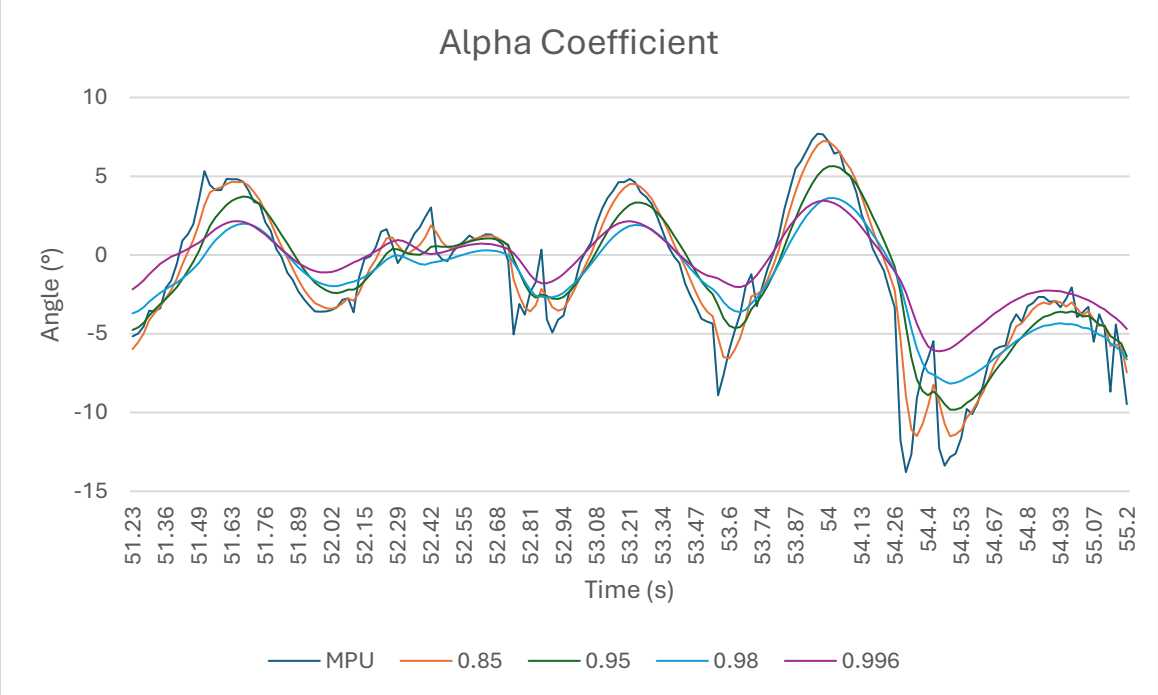


Figure 47. Complementary Filter Alpha Coefficient Comparison.

From the data in the Alpha Coefficient graph, it is clear to see that the raw angle read from the MPU6050s accelerometer, without input from the gyroscope, is too noisy. The Complementary Filter goes a good job at combining the two sensor readings, but the sensor reading was still affected by the accelerometers noise at 0.85 and 0.95. Not until a very high ratio 0.98, was a clear reading

attained, showing responsive and smooth transitions. Higher values like 0.996 are shown to lose response time, and thus not respond to the same magnitude.

4.2 Kalman Filter

The Kalman filter also combines high-frequency gyroscope readings with low-frequency accelerometer data to reduce noise and drift. However, unlike the complementary filter, it further improves accuracy by incorporating the uncertainty from the previous step, allowing it to predict the next state with greater confidence. This prediction step helps refine sensor fusion and provides more reliable estimates of the system's state.

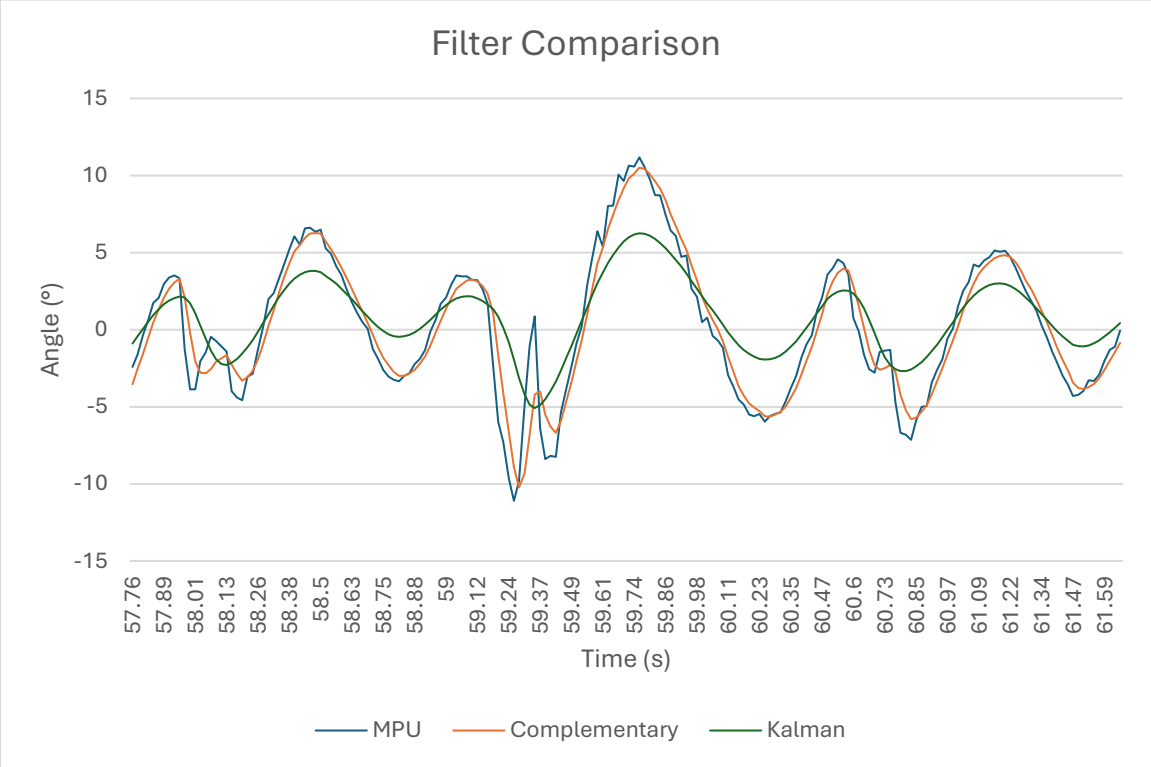


Figure 48. Comparison of Complementary and Kalman Filters.

From the graph comparing filters in Figure 48, it can be seen that the complementary filter is still affected by noise at times, this will affect the output of the control algorithm leading to instability. The Kalman filter, although appearing less responsive, provides a smooth output without sudden spikes in opposing directions, mimicking the motion of the mechanical build during testing. The lower magnitude of the Kalman filter could be due to the removal of the noise from the system, resulting in a less exaggerated, more accurate positional reading during and after motion.

4.3 Single Loop PID Control

Through excruciating trial and error, the best response to PID control was achieved by increasing the magnitude of the positional error by ten-fold. Slightly counterintuitive because if the gains were ten-fold then the output should be the same, but this is not the case due to the nature of the floating-point variable the error is stored as. Multiplying the error by ten gives the lower order numbers, those further to the right of the decimal, in the error more weight when the gains are applied, leading to a more precise input and thus result.

```
float aError = (aTarget - angle)*10; // Error for Single Loop PID Control
```

Figure 49. Error for Single Loop PID Control

The option to set the target angle was also necessitated during this phase, as the IMU was discovered to be displaying an offset of its own, either through the mounting of the sensors on the breakout or through the mounting of the module itself onto the structure. Either way, the ability to adjust the target angle by a fraction of a degree through Serial Monitor commands helped in being able to define zero for the PID positional control loop.

4.3.1 Single Loop Proportional Control

Figure 50 shows the angle from the true upright position of the mechanical build. The angle is mapped against time for increasing values of proportional gain. Figure 51 shows the corresponding proportional control output for the angles. The range of values for gain were chosen for their obvious visual impact on the system. Note the inversion of the output signal to the angle, to counteract the falling of the system into the opposite direction. Note also how the lower gain graph remains somewhat the same on both graphs, whereas the larger gain is clearly exaggerated on the proportional output graph compared to the angle graph. This highlights the effect the proportional gain has on the control system.

A proportional gain of 2.85 was shown to be the largest value to have a negligible effect on the falling system. Upping the gain to between 3.5 and 4.5, the system starts to show signs of control, but unfortunately a value couldn’t be found to provide stability to the system. The value was increased until an obviously overly proportional response was attained, in which the build was not freely falling, but clearly being pushed out of balance by the reaction wheel, seen by 6.45. Proportional gain alone was not enough to provide stability, aligning with simulations.

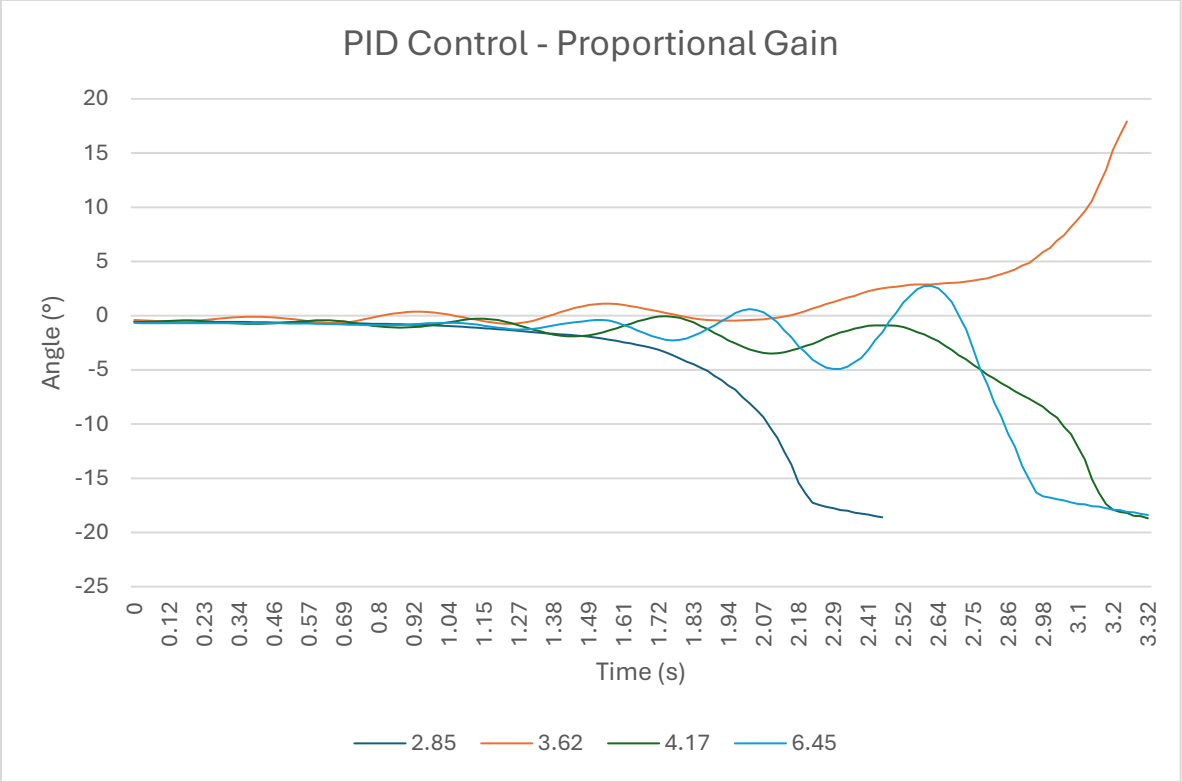


Figure 50. Single Loop PID Control Angle Error.

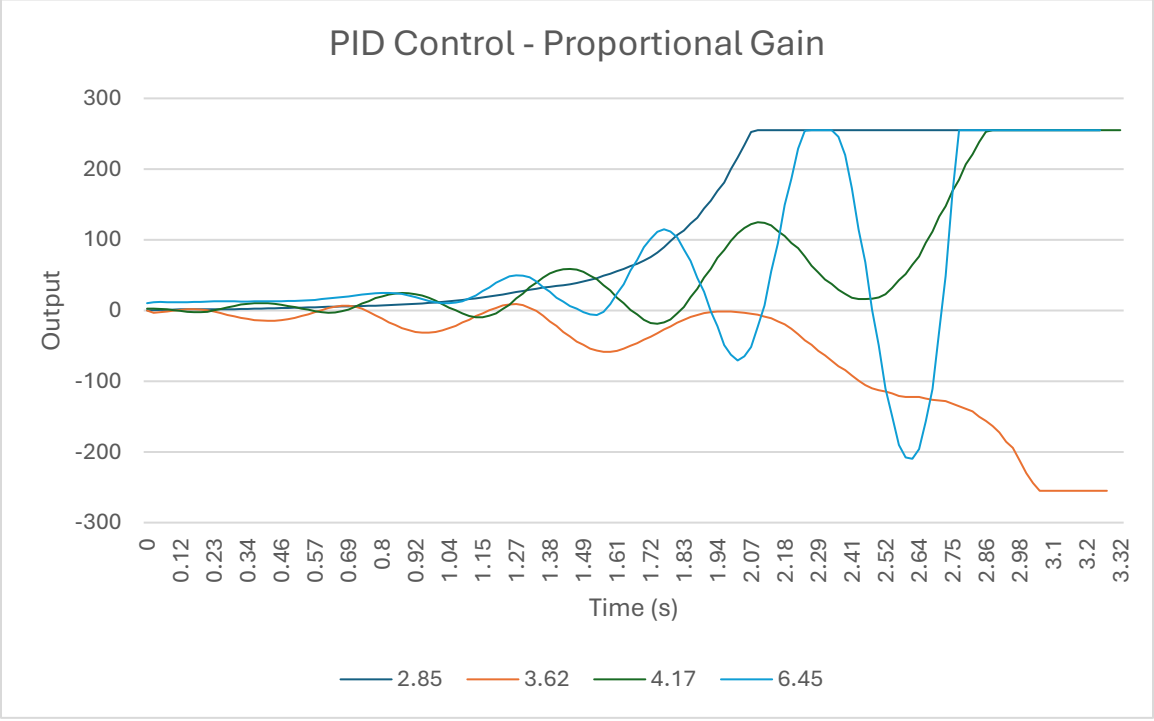


Figure 51. PID Control - Proportional Gain.

4.3.2 Single Loop Derivative Control

Derivative control was employed to add to the response by pre-empting the future state of the system. Derivative control does this by applying a gain to the rate of change of the error. The results for varying derivative gains are shown below in Figure 52. The graph shows both the Proportional, and the Proportional-Derivative control response, highlighting the effect of the derivative gain on the system to smooth the output.

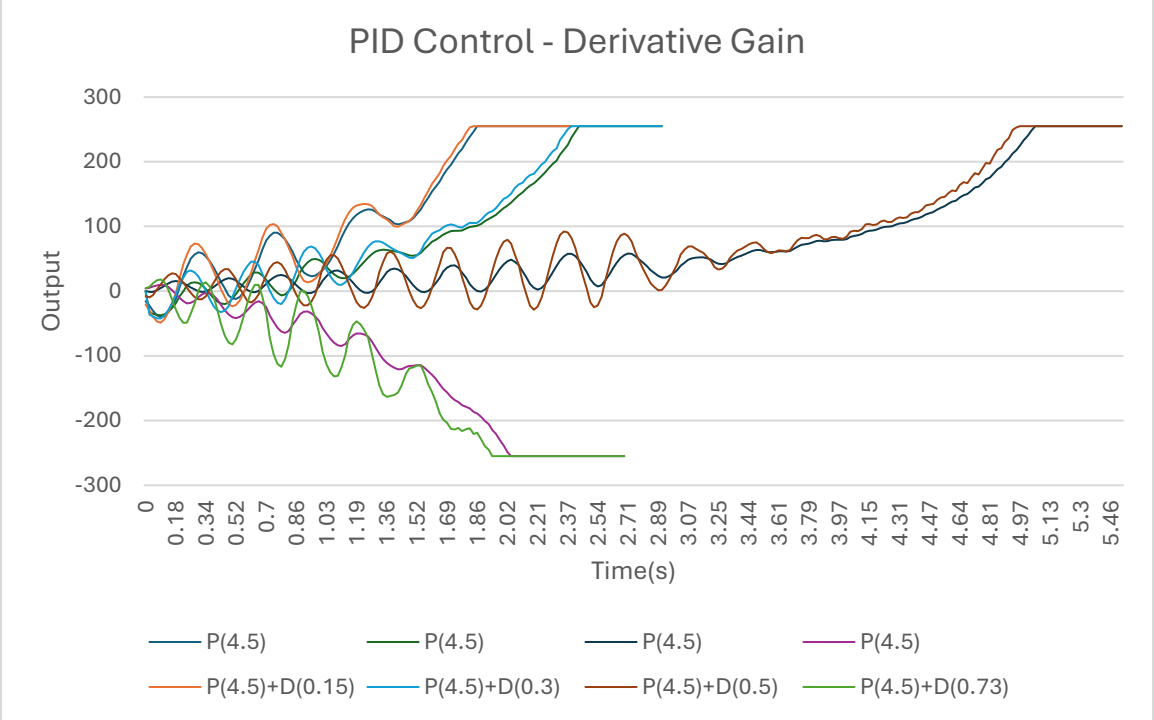


Figure 52. PID Control - Derivative Gain.

The application of the lower derivative gain, 0.15, shows minimal deviation from the solely proportional response. Increasing the gain to 0.3 shows a slight improvement in stability, while a gain 0.5 resulted in the most stable output. Increasing the derivative gain to values larger than 0.7 resulted in more oscillations and inhibited the proportional gain from responding at a magnitude that was sufficient to orientate the system.

Although the simulation showed Proportional-Derivative control would be substantial enough to orientate the system, this is not the case with the mechanical build. Further control must be explored.

4.3.3 Single Loop Integral Control

Integral control was included to improve stability. Integral gain is applied to the sum of the error, usually to counteract steady state error in the output, in the case of a reaction wheel inverted pendulum, it adds to the responses magnitude the more the pendulum leans in that direction. The Proportional-Integral-Derivative control and the corresponding Proportional-Derivative control outputs are graphed in Figure 53, showing the effect of the integral gain on the control systems output.

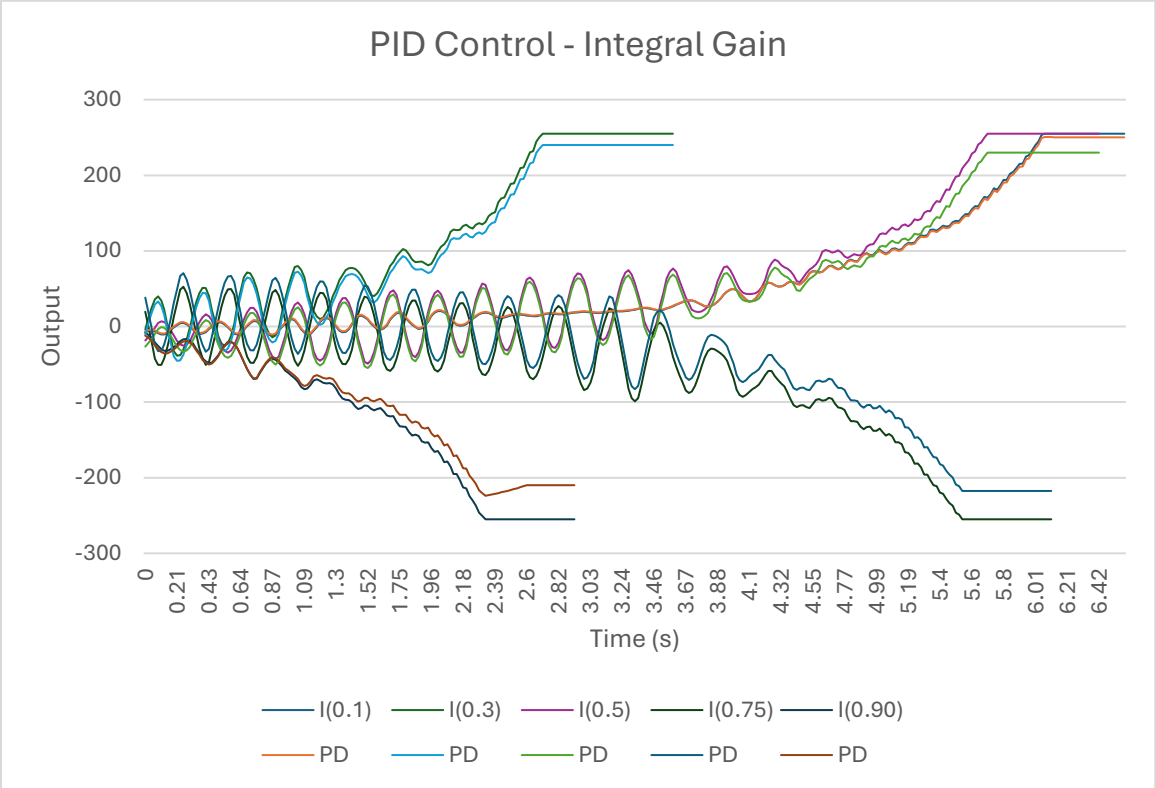


Figure 53. PID Control - Integral Gain.

The addition of the integral gain to the system shows a clear increase in response, deviating more from the Proportional-Derivative control as sum of the errors increases. The larger the integral gain, the greater the deviation from the Proportional-Derivative curves as time goes on.

Increasing the integral gain hasn't presented a clear pattern, a lower value of 0.1 and mid-range values of 0.5 and 0.75 has produced a similar balancing time, while 0.3 and 0.9 also present a similar balancing time. Incorporating Integral control hasn't noticeably improved the overall stability of the system, when compared to Proportional-Derivative control.

4.3.4 Single Loop PID Control

A number of trials of varying proportional, integral, and derivative gains were attempted and their results graphed in Figure 54. A start point was taken from the previous values and manually tuned from there. Other tuning methods require determining the critical gain and the period for the system, which were unobtainable from the proportional gain alone.

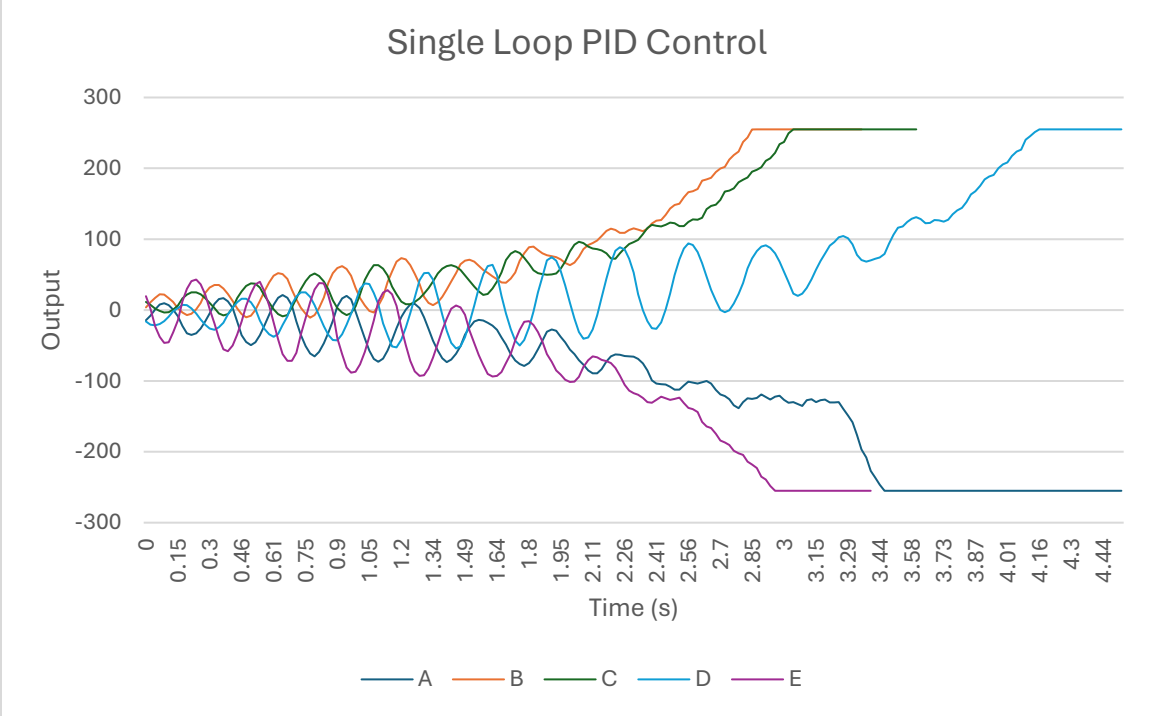


Figure 54. Single Loop PID Control.

After sufficient trials and time, it was clear the Single Loop PID Control would not be enough to stabilize the pendulum. The model is shown to balance for a brief period while swinging about zero but eventually the wheels speed increases to such an extent that it cannot provide any more change in speed to correct the alignment and the system falls. The wheel speed problem is called saturation and indicates the wheel speed needs to be taken into consideration to make the control algorithm more effective. In a zero-gravity environment the loop might be able to work, due to the removal of the alternating potential energy, but this would need to be tested to be confirmed. It was decided to move forward, building on this strategy with a Cascade loop.

4.4 Encoder Low-Pass Filter

The Cascade loop builds on the previous Single loop by altering its setpoint with a second nested PID control loop. The error for the new PID loop is the speed of the reaction wheel as read by the encoder. The encoder raw data was seen to be affected by noise, either from interrupts service routines overlapping each other, or due to the vibration caused by the motor, and therefore required filtering. A Low-Pass filter was implemented to remove smooth the data from the sensor. The results of various coefficients for the low-pass filter are shown below.

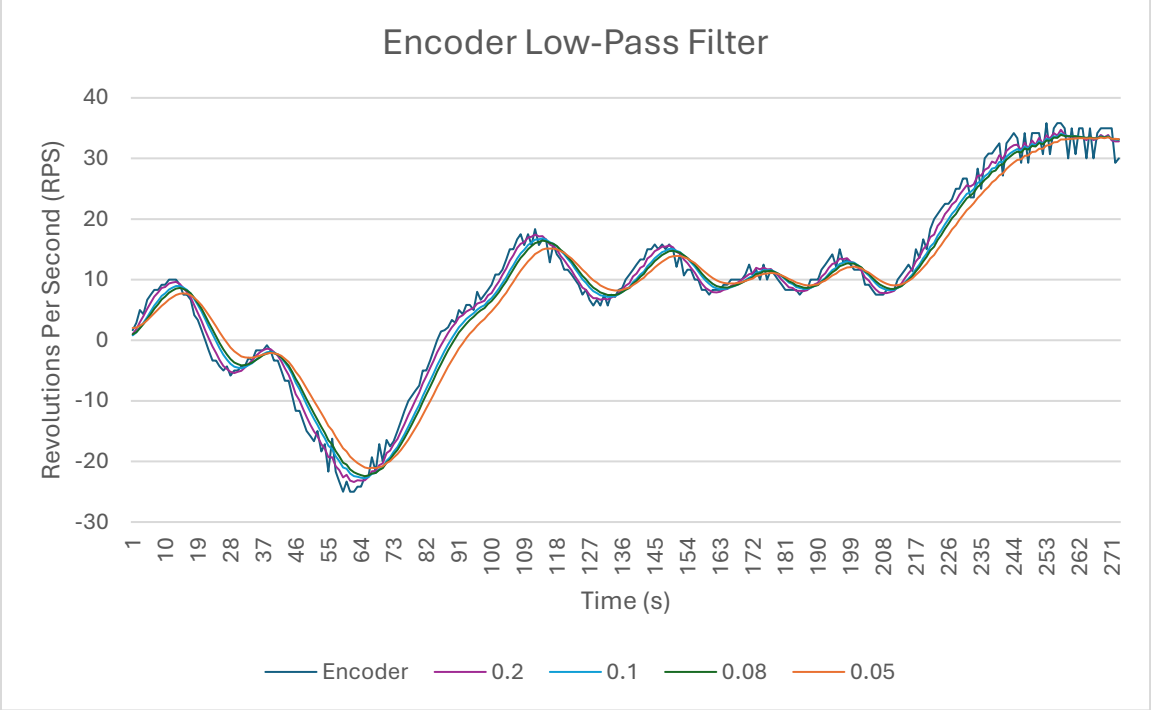


Figure 55. Encoder Low-Pass Filter Coefficient Comparison.

The reading from the encoder is seen to be quite rough, a value of 0.2 does have an immediate effect on the output but is still quite jagged in places where the motor speed is levelling out to change direction or when the continuous changing speed starts to match the harmonic frequency of the system and induces exceptionally large vibrations. This is also seen in a lesser degree in the output of the 0.1 and 0.08 values. Not until the coefficient decreases to 0.05 does the data truly begin to smooth out. Some responsiveness is lost, seen by 0.05 lagging behind the other curves on the graph, but this was shown to not affect the overall performance of the system, while the smoothness of the input data was shown to lead to a smooth control system output.

4.5 Cascade Loop PID Control

Now that the encoder data has been filtered and the wheel speed is free from noise, the data can be used in the Inner Speed PID Loop as its error.

```
float sError = sTarget - rpm; // Error for Inner Speed PID Loop

float aError = (controlSpeed - angle + aTarget)*10; // Error for Outer Angle PID Loop
```

Figure 56. Errors For Cascade Loop PID Control.

The graph in Figure 57 shows how the speed of the reaction wheel is translated into a setpoint for the Outer Angle PID Loop. The curves clearly show how the setpoint, in degrees, reacts proportionally to the wheel speed. The wheel speed increases as the angle of deflection increases, this leads to the Inner Speed PID Loop to move the setpoint of the Outer Angle PID Loop in the opposite direction, increasing the inner loop's error and thus its response.

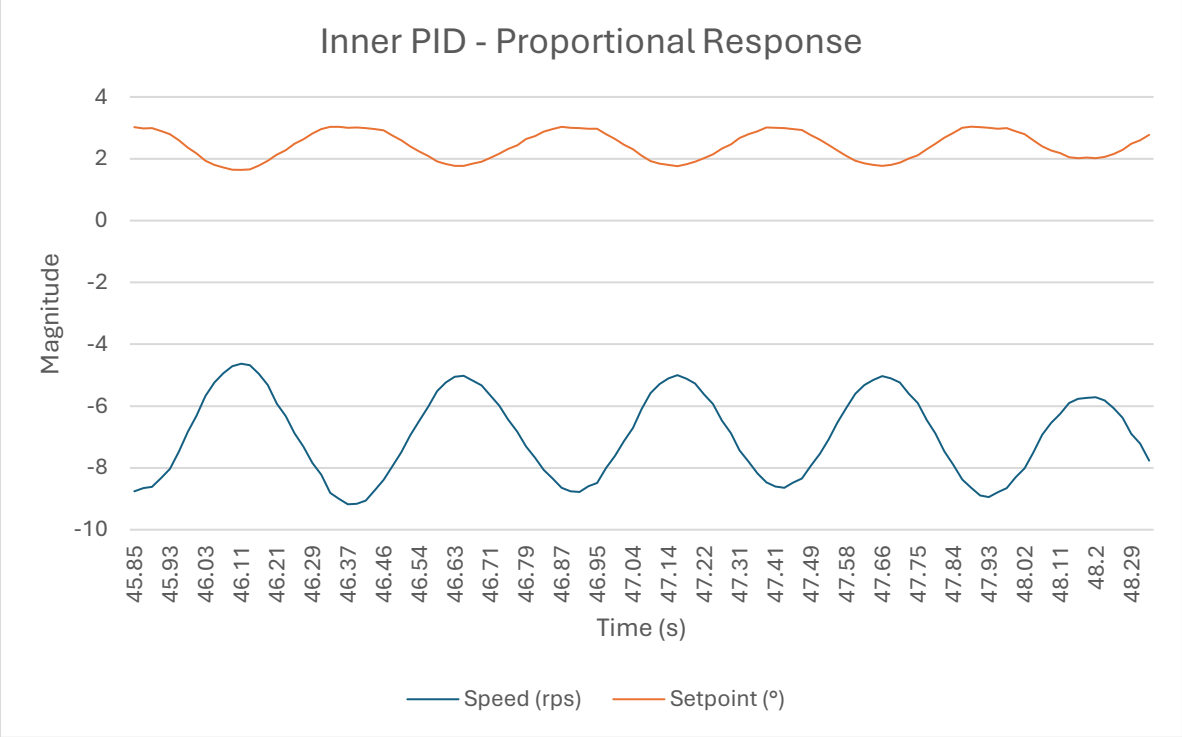


Figure 57. Cascade Loop Input to Output.

4.5.1 Cascade Loop Proportional Control

The introduction of the proportional gain for the Inner Speed PID Loop immediately provided a stable orientation. This stable orientation was maintained through an enormous range of values for the inner loops proportional gain, not until a disturbance is introduced into the system does the different gains produce different outputs. The Figure 58 displays the output of the Cascade Loop for various inner loop gains during a disturbance. After brief experimentation it was found that the Outer Angle PID Loop gains were best set as follows: $aKp = 3.20,\ aKi = 0.00,\ aKd = 0.31$.

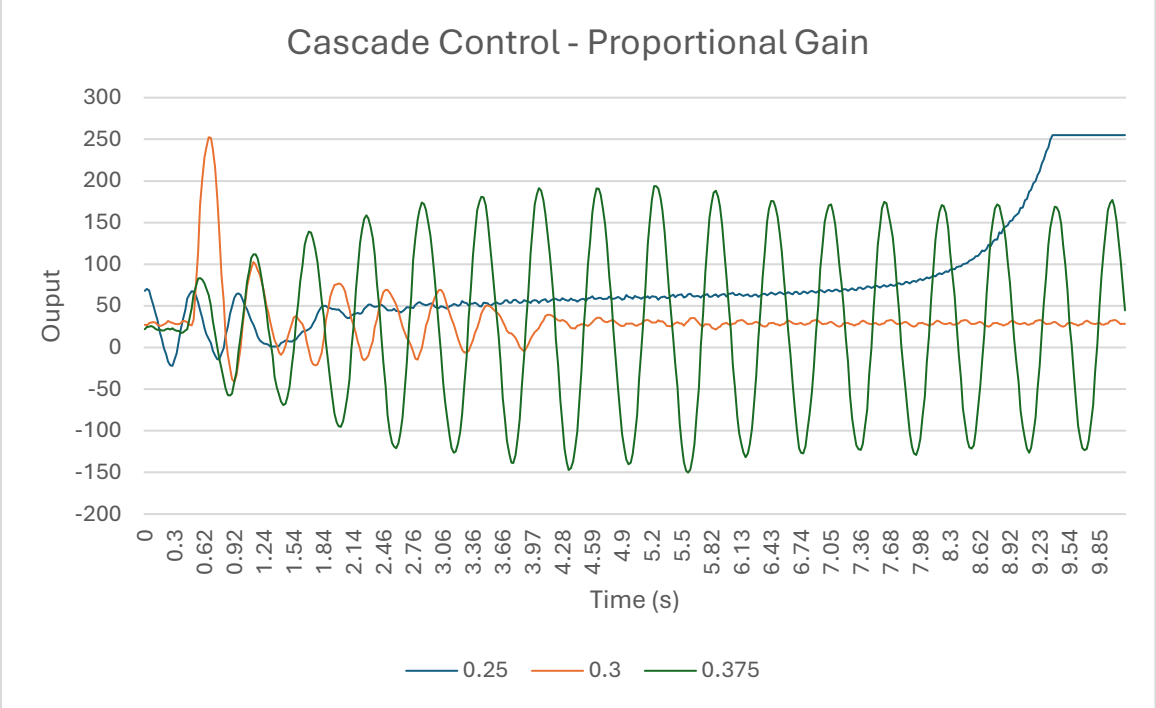


Figure 58. Cascade Control - Proportional Gain.

The lower gain of 0.25 was shown to be able to correct a small initial disturbance, but wheel speed saturation was still a problem. Increasing the gain to 0.3 solved the issue with wheel speed saturation while also significantly increasing the size of the disturbance it was able to correct. Increasing the gain again to 0.375 shows how higher values can magnify small disturbances to induce oscillations in the system.

4.5.2 Cascade Loop Derivative Control

The Outer Speed PID Loops derivative gain was employed also to add to the response by pre-empting the future state of the system to diminish the oscillations of the system and produce a quicker settling time. The results for various derivative gains are shown in Figure 59, the other gains were set as follows: $sKp = 0.30,\ aKp = 3.20,\ aKi = 0.00,\ aKd = 0.31$.

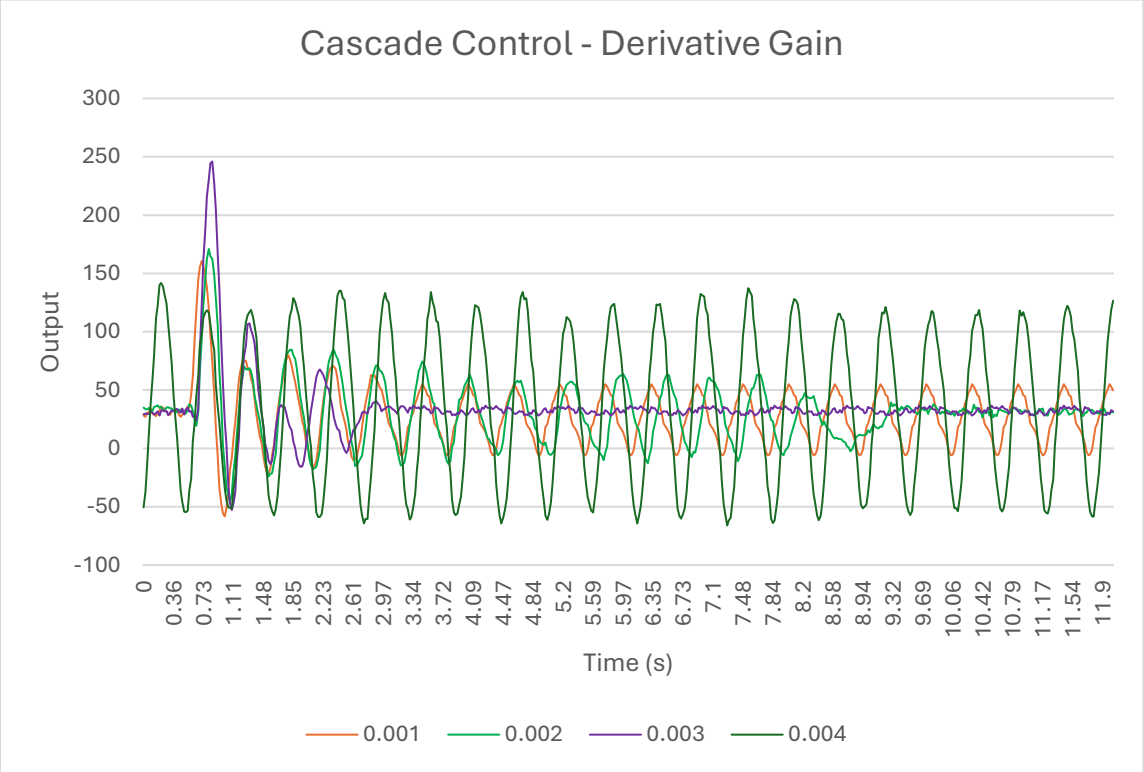


Figure 59. Cascade Control - Derivative Gain.

A low gain of 0.001 was too small to completely dampen oscillations. Increasing the gain to 0.002 showed the oscillations gradually reduce in size and die out. A larger gain of 0.003 results in an even larger initial response to the disturbance and was able to dampen it in an even quicker settling time. Increasing the gain to large values induces oscillations even before the system is disturbed, as shown by the 0.004 derivative gain curve.

4.5.3 Cascade Loop Integral Control

Integral control was again applied, this time to the Inner Speed PID Loop, to try improving overall system stability. The results for the varying integral gains are shown in Figure 60, the results were obtained while the other gains were set as follows: $sKp = 0.30$, $sKd = 0.003$, $aKp = 3.20$, $aKi = 0.00$, $aKd = 0.31$.

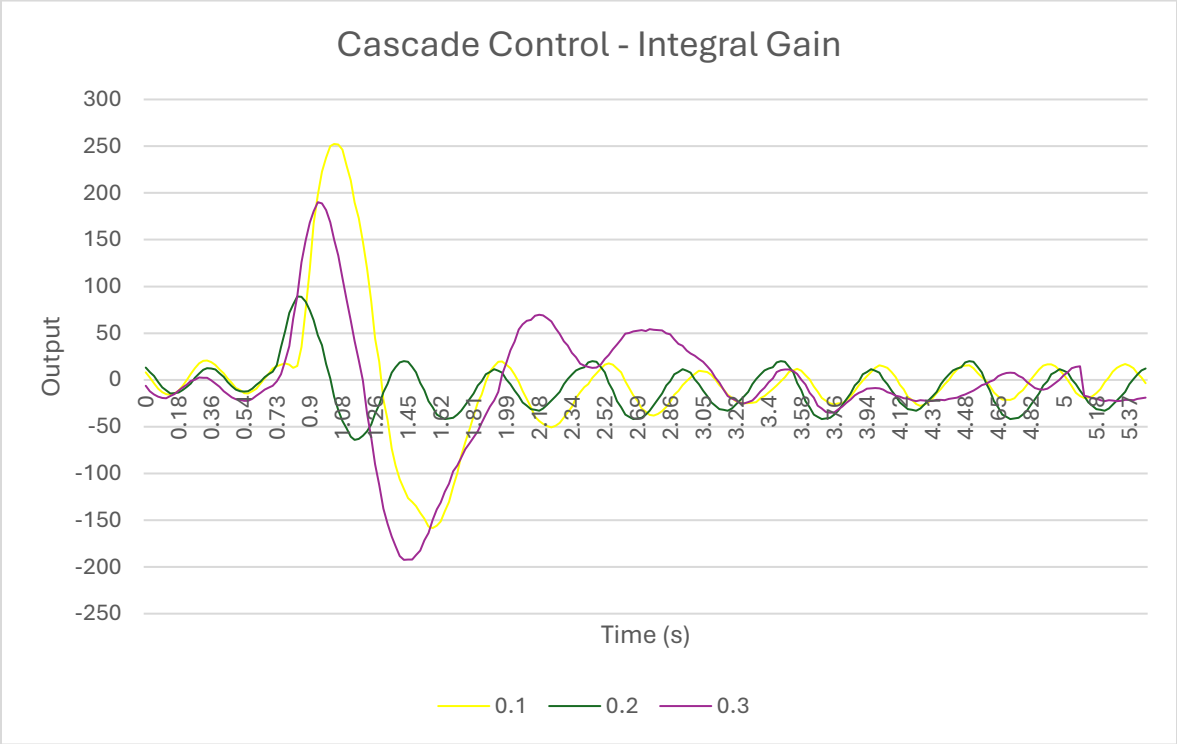


Figure 60. Cascade Control - Integral Gain.

The introduction of the integral control into the inner loop, even at a small gain, has made improved the response of the system but at the cost of its stability. While the disturbances are diminished quickly by all values, the system continues to oscillate slightly, not settling to the same extent as without integral control. This could be due to the ramping up and down of the sum of the errors of the wheel speed as the pendulum sways past true zero. This ramping down of the value requires the system to spend time past the upright position in the opposite direction, thus overshooting the mark on every approach of zero.

4.5.4 Cascade Loop PID Control

The Cascade Loop, utilizing an Inner Speed and an Outer Angle PID loop, has been shown to exert excellent control over the system. After going through each of the gains individually and observing their effect on the overall system, the best parameters for maintaining a vertical position were found to be: $sKp = 0.30$, $sKi = 0.00$, $sKd = 0.003$, $aKp = 3.20$, $aKi = 0.00$, $aKd = 0.31$. This combination of gains keeps the model upright with minimal deviation from the centre and offering a certain amount of disturbance rejection. The range of the data is 0.41°, as seen in Figure 61, while the disturbance rejection only has capacity to 100 out the possible 255 output for motor speed, seen in Figure 62.

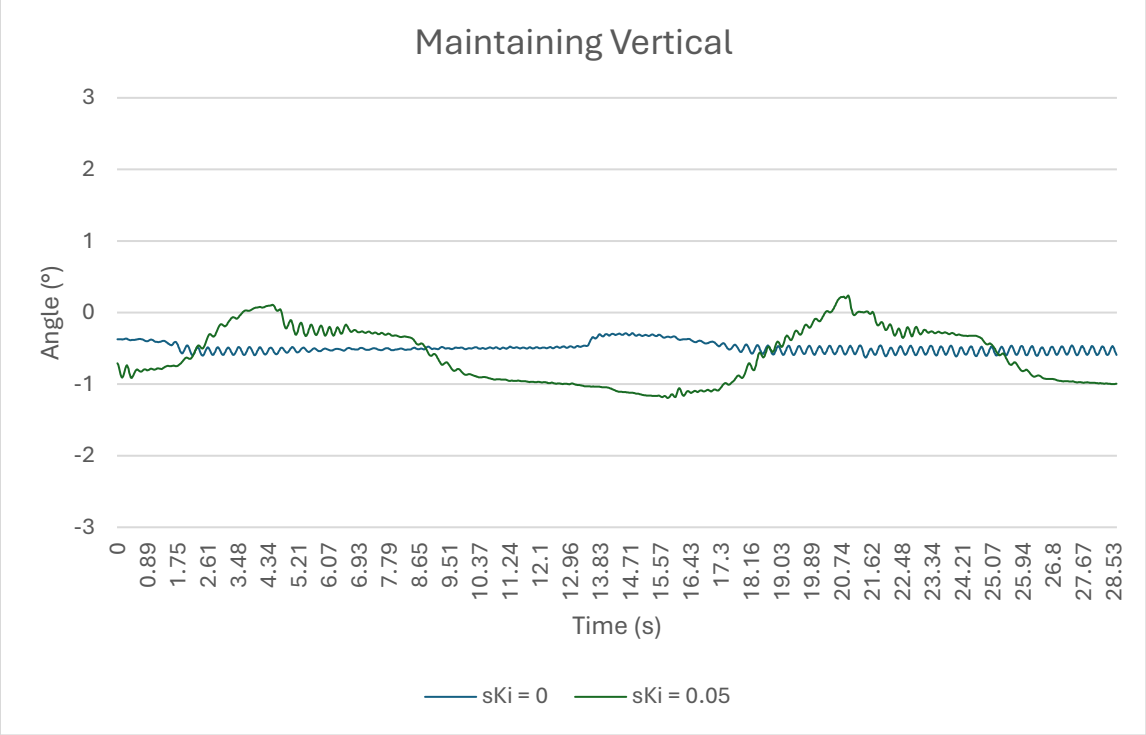


Figure 61. Maintaining Vertical - Cascade Loop Gain Comparison.

However, increasing the integral gain of the inner speed loop, even to just $sKi = 0.05$, will offer better disturbance rejection at the cost of a slight drift about the vertical axis. The control loop now has access to the full 255 output and can reject much larger disturbances, shown in Figure 61. Unfortunately, the trade-off is the overall stillness of the system, seen by the range of the data in Figure 61 increasing to 1.34°.

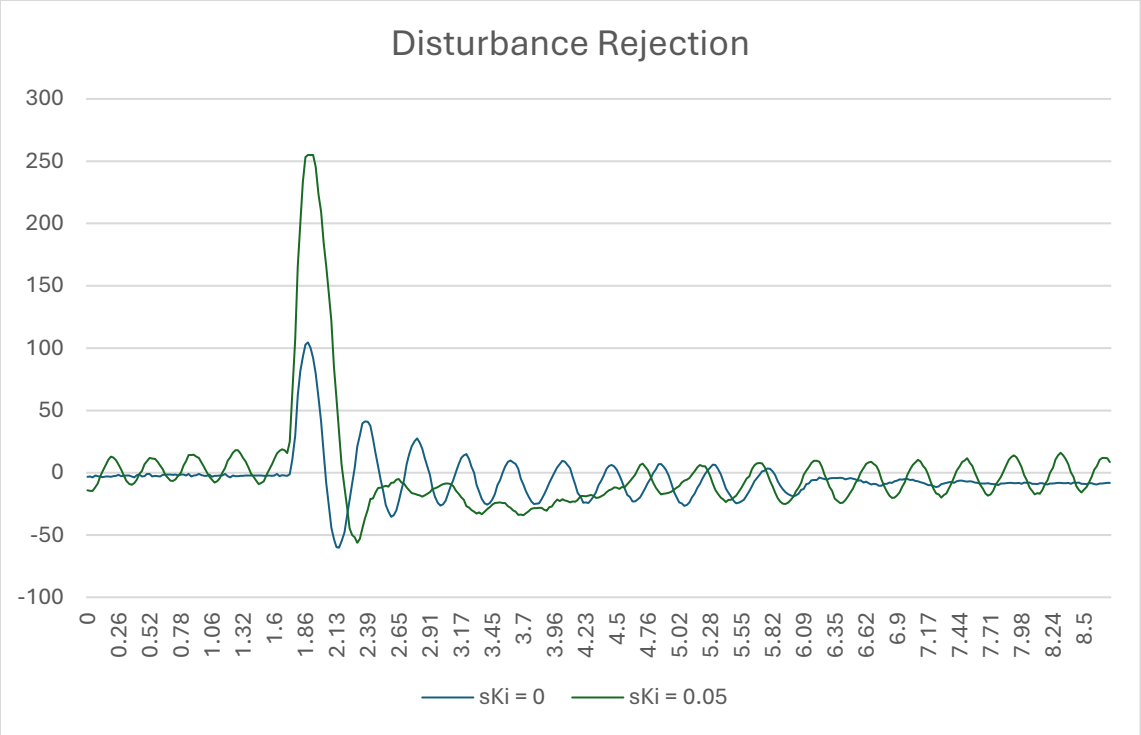


Figure 62. Disturbance Rejection - Cascade Loop Gain Comparison.

Chapter 5 Conclusions and Recommendations

5.1 Conclusions

The primary aim of the project was to demonstrate that an attitude-correction and control system for a one-unit CubeSat can be built almost entirely from off-the-shelf hardware and software written from first principles. The work began by defining the dual challenge faced by small spacecraft: rapid detumbling after deployment and fine pointing during routine operations. A reaction-wheel, printed and weighted with nuts and bolts, driven by a direct current motor and mounted inside a printed CubeSat frame, provided the actuation; an MPU-series inertial-measurement unit linked to a low-power micro-controller supplied the real-time state data required for feedback.

Early testing with a complementary filter confirmed the concept but revealed a tendency for the attitude estimate to spike from motor induced vibrations and whenever the model crossed its zero-position. Replacing that filter with a discrete Kalman implementation removed the spikes and produced a steadier estimate, allowing tighter control gains without risking instability. At this stage a single-loop proportional–integral–derivative controller could hold the rig upright for a very short amount of time, yet its wheel speed drifted steadily towards saturation because the control system had no awareness of it and therefore no way to minimize it.

Introducing a cascaded architecture was the turning point. In the inner loop the wheel-speed error is converted into a desired body-position setpoint; the outer loop then increases the motor speed until the body reaches its target. Because the wheel state is now part of the control decision, the wheel no longer saturates and can tolerate deliberate knocks without losing stability.

With the cascaded loop tuned, the model consistently rejected moderate disturbances. The result is a sub-degree pointing platform that can be reproduced by hobbyists and students for a fraction of the cost of professional platforms, a strong validation of the original objective. All mechanical models, circuit schematics, firmware and tuning scripts have been archived under open licences so future teams can build directly on the work rather than starting from scratch. In that sense the project not only meets its own goals but also lowers the barrier for others who wish to explore precision control on very small satellites. Key design files and firmware have been archived in the appendices ensuring that subsequent work can build on the current foundation.

5.2 Recommendations

The biggest recommendation to be given is that the undertaking of any project involving such precise control should be undertaken with due patience and time. The scope of this project was ambitious, and time was the main constraint when trying to meet all objectives. The following recommendations would be made in trying to further the progress of this project:

- Expand to full three-axis control by adding two orthogonal wheels and upgrading the attitude estimator to quaternion form so that any orientation can be commanded directly.
- Explore adaptive control extensions such as gain scheduling or model-predictive control to accommodate inertia shifts introduced by deployable payloads or fuel movement.
- Migrate to space-qualified electronics and complete vibration and thermal-vacuum campaigns so the hardware can transition from laboratory rig to flight-ready payload.
- Finish the ground-segment interface so live attitude, wheel-speed and power telemetry stream to operators, enabling on-orbit gain refinement and rapid anomaly detection.

Pursuing these avenues will evolve the current laboratory model into a versatile open-hardware platform capable of supporting real missions and helping small-satellite teams point their instruments with greater accuracy on every orbit.

References

Arduino, 2025. *Arduino Uno R3 Datasheet.* [Online]
Available at: https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf
[Accessed 14 04 2025].

Arrendondo, M., 2023. *MPU6050 Wiki.* [Online]
Available at: https://github.com/ElectronicCats/mpu6050/wiki
[Accessed 02 December 2024].

Beer, F. P. & Johnston, E. R., 1989. *Engineering Mechanics: Statics & Dynamincs.* 4th ed. New York: McGraw-Hill College.

Bobrow, F., A. Angelico, B. & P. R. Martins, F., 2020. *The Cubli: Modeling Utilizing Quarternions.* [Online]
Available at: https://arxiv.org/pdf/2009.14626
[Accessed 20 November 2024].

Bobrow, F., A. Angelico, B. & S. P. da Silva, P., 2020. *The Cubli: Modeling and Nonlinear Control.* [Online]
Available at: https://arxiv.org/pdf/2009.14625
[Accessed 20 November 2024].

Bohdan, P., 2016. *Self-Balancing System for a Two-Wheel Machine,* Dublin: Institute of Technology Blanchardstown.

Candan, B. & Soken, H., 2021. *Robust Attitude Estimation Using IMU-Only Measurements.* [Online]
Available at: https://ieeexplore.ieee.org/document/9511443
[Accessed 10 December 2024].

CarbonAeronautics, 2022. *Part XV: One-dimensional Kalman filter.* [Online]
Available at: https://github.com/CarbonAeronautics/Part-XV-1DKalmanFilter
[Accessed 2025 04 22].

DFRobot, 2024. *Fermion: MPU-6050 6 DOF Sensor (Breakout).* [Online]
Available at: https://www.dfrobot.com/product-880.html
[Accessed 02 December 2024].

DigiKey, 2024. *ESP32-DEVKITC-DA.* [Online]
Available at: https://www.digikey.ie/en/products/detail/espressif-systems/ESP32-DEVKITC-DA/15295934
[Accessed 02 December 2024].

Dominik Zaborniak, K. P. a. M. W., 2024. *Design, Implementation, and Control of a Wheel-Based Inverted Pendulum.* [Online]
Available at: https://www.mdpi.com/2079-9292/13/3/514
[Accessed 19 November 2024].

Goldstein, H., Poole, C. & Safko, J., 2002. *Classical Mechaincs.* 3rd ed. San Francisco: Addison-Wesley.

Halder, T., Singh, J. & Nandy, S., 2025. Implementation of Complmentary Filter. *World Scientific News,* 199(https://worldscientificnews.com/wp-content/uploads/2025/01/WSN-199-2025-218-234.pdf), pp. 218-234.

Henderson, D. M., 1977. *Euler Angles, Quarternions, and Transformations,* Washington, D.C.: NASA.

Kuipers, J., 1999. *Quarternions and Rotation Sequences: A Primer with Applications to Orbits, Aerospace, and Virtual Reality.* 1st ed. Princeton: Princeton University Press.

Kulu, E., 2024. *Nanosats Database.* [Online]
Available at: https://www.nanosats.eu/
[Accessed 18 November 2024].

Liao, T.-L., Chen, S.-J., Chiu, C.-C. & Yan, J.-J., 2020. *Nonlinear Dynamics and Control of a Cube Robot.* [Online]
Available at: https://www.mdpi.com/2227-7390/8/10/1840
[Accessed 20 November 2024].

Messier, D., 2015. *Tiny 'Cubesats' Gaining Bigger Role in Space.* [Online]
Available at: https://www.space.com/29464-cubesats-space-science-missions.html
[Accessed 18 November 2024].

Moebs, W., J. Ling, S. & Sanny, J., 2016. *University Physics Volume 1.* 1st ed. Houston, Texas: OpenStax.

Mohanarajah, G., Merz, M., Thommen, I. & D'Andrea, R., 2012. *The Cubli: A Cube that can Jump Up and Balance.* [Online]
Available at: https://ieeexplore.ieee.org/document/6385896
[Accessed 20 November 2024].

Mohanarajah, G., Muehlebach, M., Widmer, T. & D'Andrea, R., 2013. *The Cubli: A Reaction Wheel Based 3D Inverted Pendulum.* [Online]
Available at: https://www.semanticscholar.org/paper/The-Cubli%3A-A-reaction-wheel-based-3D-inverted-Gajamohan-Muehlebach/f87a6c7d30608a0425d7c5de76a4df7b23c0af96
[Accessed 20 November 2024].

Mohanarajah, G., Muehleback, M. & D'Andrea, R., 2013. *Nonlinear Analysis and Control of a Reaction Wheel-based 3D Inverted Pendulum.* [Online]
Available at: https://ieeexplore.ieee.org/document/6760059
[Accessed 20 November 2024].

NASA Science Editorial Team, 2019. *CubeSats - Going Farther.* [Online]
Available at: https://science.nasa.gov/solar-system/10-things-cubesats-going-farther/
[Accessed 18 November 2024].

NASA, 2024. *State-of-the-Art of Small Spacecraft Technology.* [Online]
Available at: https://www.nasa.gov/smallsat-institute/sst-soa/guidance-navigation-and-control/
[Accessed 19 November 2024].

Nidec Corp., 2024. *Nidel 24H.* [Online]
Available at: https://www.nidec.com/en/product/search/category/B101/M102/S100/NCJ-24H-12-01/
[Accessed 14 04 2025].

OlliW's Bastelseiten, 2018. *IMU Data Fusing: Complementary, Kalman, and Mahony Filter.* [Online]
Available at: https://www.olliw.eu/2013/imu-data-fusing/
[Accessed 10 December 2024].

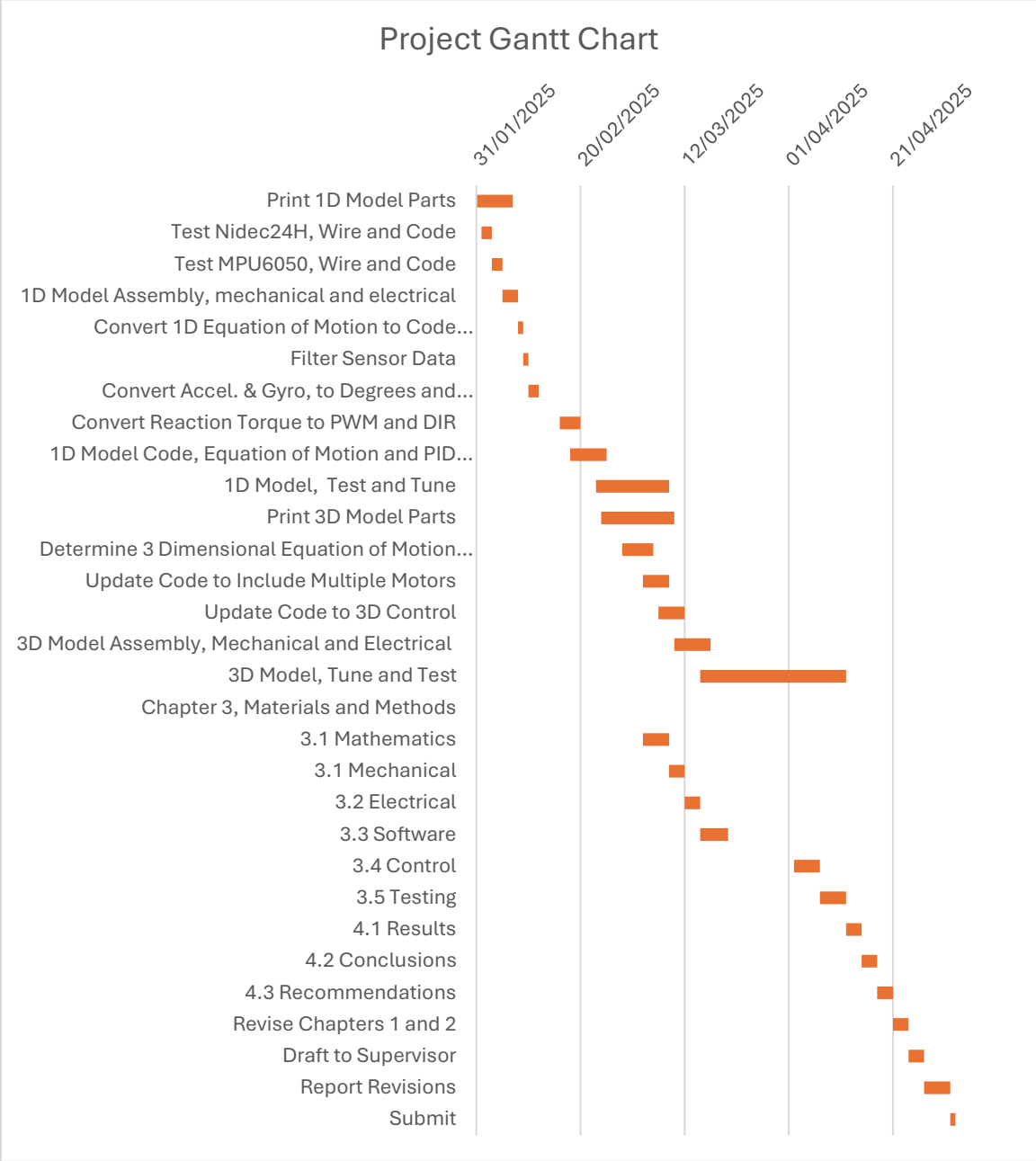
Salmony, P., 2018. *Attitude Estimation.* [Online]
Available at: https://github.com/pms67/Attitude-Estimation
[Accessed 10 December 2024].

State Examinations Commission, 2011. *Formulae and Tables.* 3rd ed. Dublin: Government Publications.

Texas Instruments, 2016. *LM340, LM340A and LM7805 Family Wide VIN 1.5-A Fixed Voltage Regulators,* s.l.: Texas Instruments Inc.

Zhang, Q., n.d. *Self-Balanced Cube Robot Modeling and Control.* [Online]
Available at: https://www.columbia.edu/~ja3451/courses/E6602/projects/balanced_cube.pdf
[Accessed 20 November 2024].

Appendix A Project Planning



Appendix B Salient Extracts from Project Diary

Extract 1 31 Jan – 2 Feb 2025

- Completed the final slicer checks and printed the entire one-axis frame overnight; no reprints needed.
- Wired the Nidec 24H motor to the bench supply and wrote a 40-line test sketch.
- Logged motor current at three duty-cycles; Least noise and vibration at 30kHz.

Extract 2 3 Feb – 10 Feb 2025

- Soldered the MPU6050 breakout, mounted it on the frame and verified I²C comms.
- Translated the one-dimensional equation of motion into C.
- Converted raw accel and gyro counts to degrees & deg s^{-1}; plotted them in real time with Processing to confirm axis mapping.
- Added a complementary filter.

Extract 3 10 Feb – 18 Feb 2025

- Derived the motor-torque-to-PWM map (six-point linear fit) and pushed it into a dedicated function.

Extract 4 18 Feb – 23 Feb 2025

- Assembled the full one-axis rig: frame, wheel, electronics, wiring harness.
- Laid the unit on the air-bearing table; confirmed detumble from 6 deg s^{-1} to <0.05 deg s^{-1} in 19 s (best run).
- Captured high-speed video, annotated the response curve and exported frames for Chapter 3 figures.

Extract 5 24 Feb – 3 Mar 2025

- Printed the three-axis frame parts; one bearing pocket came out 0.3 mm undersize—reprinted.
- Started the full 3-D dynamics derivation; inertia tensor finished but gravity-gradient term still messy (progress ~25 %).
- Sketched three-wheel orientation in SolidWorks and ran a quick mass-budget check: within 60 g of the ration determined from equations of motion.

Extract 6 4 Mar – 12 Mar 2025

- Began restructuring the firmware for multiple motors; IRQ conflicts with the second encoder stalled progress.
- Shifted focus to the report: wrote the Mathematics and Mechanical sections in a blitz, then split the Electrical draft over two evenings.
- Confirmed with the supervisor that finishing the control theory write-up takes precedence over 3-D code this week.

Extract 7 12 Mar – 21 Mar 2025

- Added the Control subsection and dropped the first block diagrams into Chapter 3.
- Ordered smaller Hall sensors to test as backup encoders—lead time two weeks, so flagged as “stretch goal.”
- Started an issues list for the stalled 3-D model; main blocker is time rather than concept.

Extract 8 21 Mar – 7 Apr 2025

- Ran extended one-axis simulations at three disturbance levels; exported CSVs directly into the Results chapter.
- Drafted Chapter 4 “Results” and “Conclusions”.
- Revisited Chapters 1 & 2, trimming five pages and modernising the literature review citations.

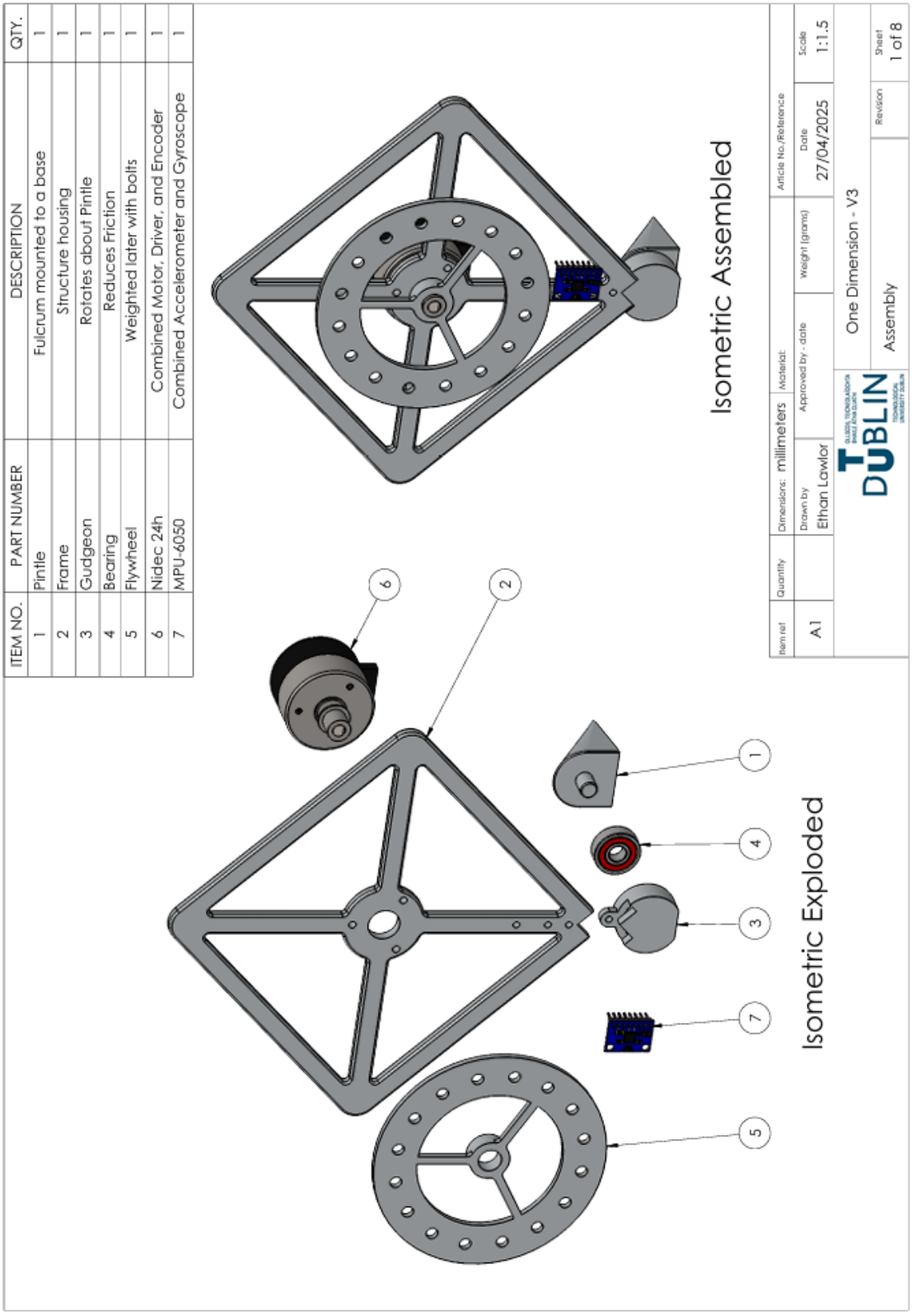
Extract 9 7 Apr – 18 Apr 2025

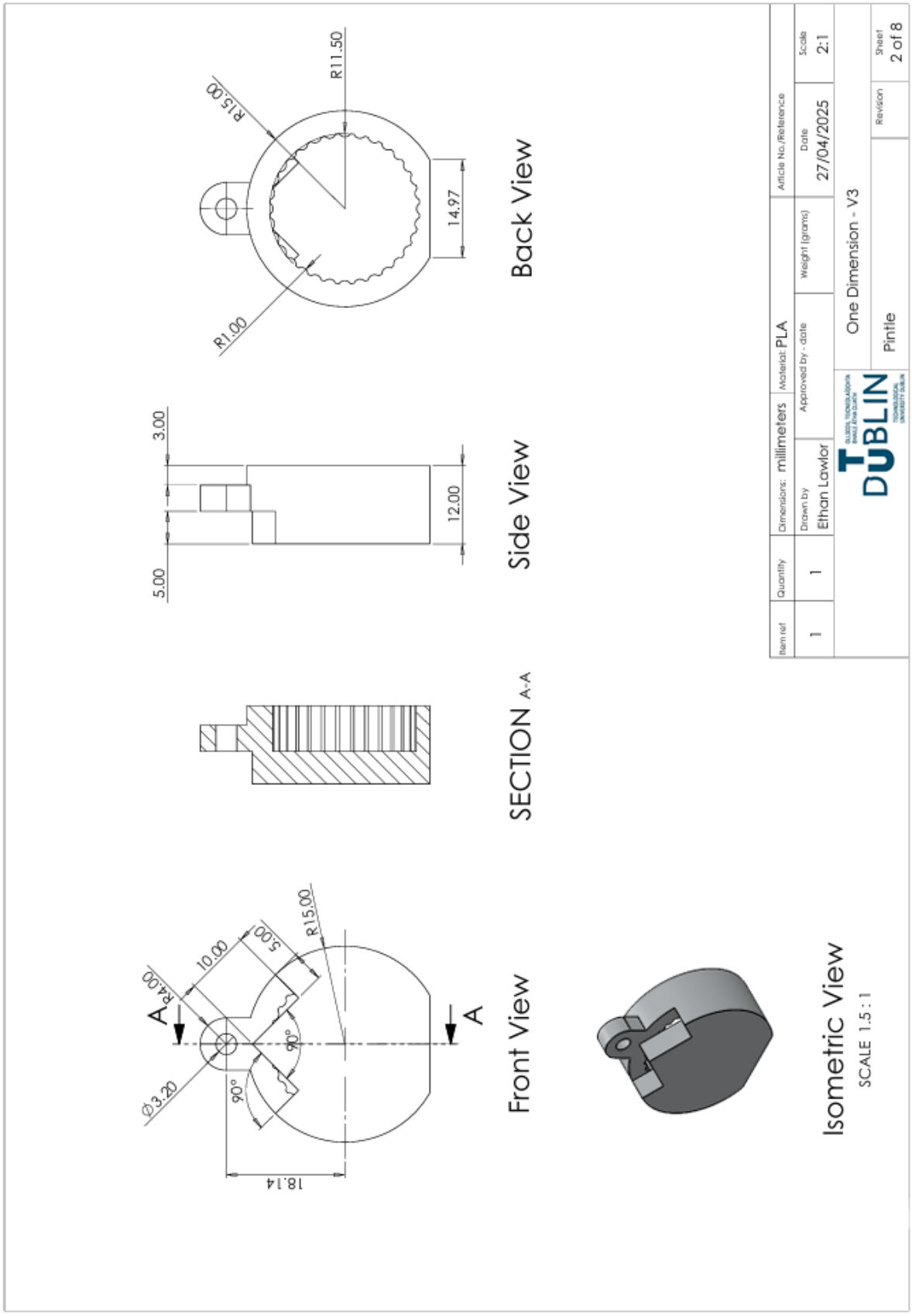
- Noticed long-term IMU drift at high wheel speeds—changed to a Kalman filter.
- 3-hour tuning marathon produced a rock-steady 0.4 deg peak-to-peak—but only on fresh batteries. Changed to wall plug.
- Polished every figure: consistent font, unified colour palette, caption cross-references checked.

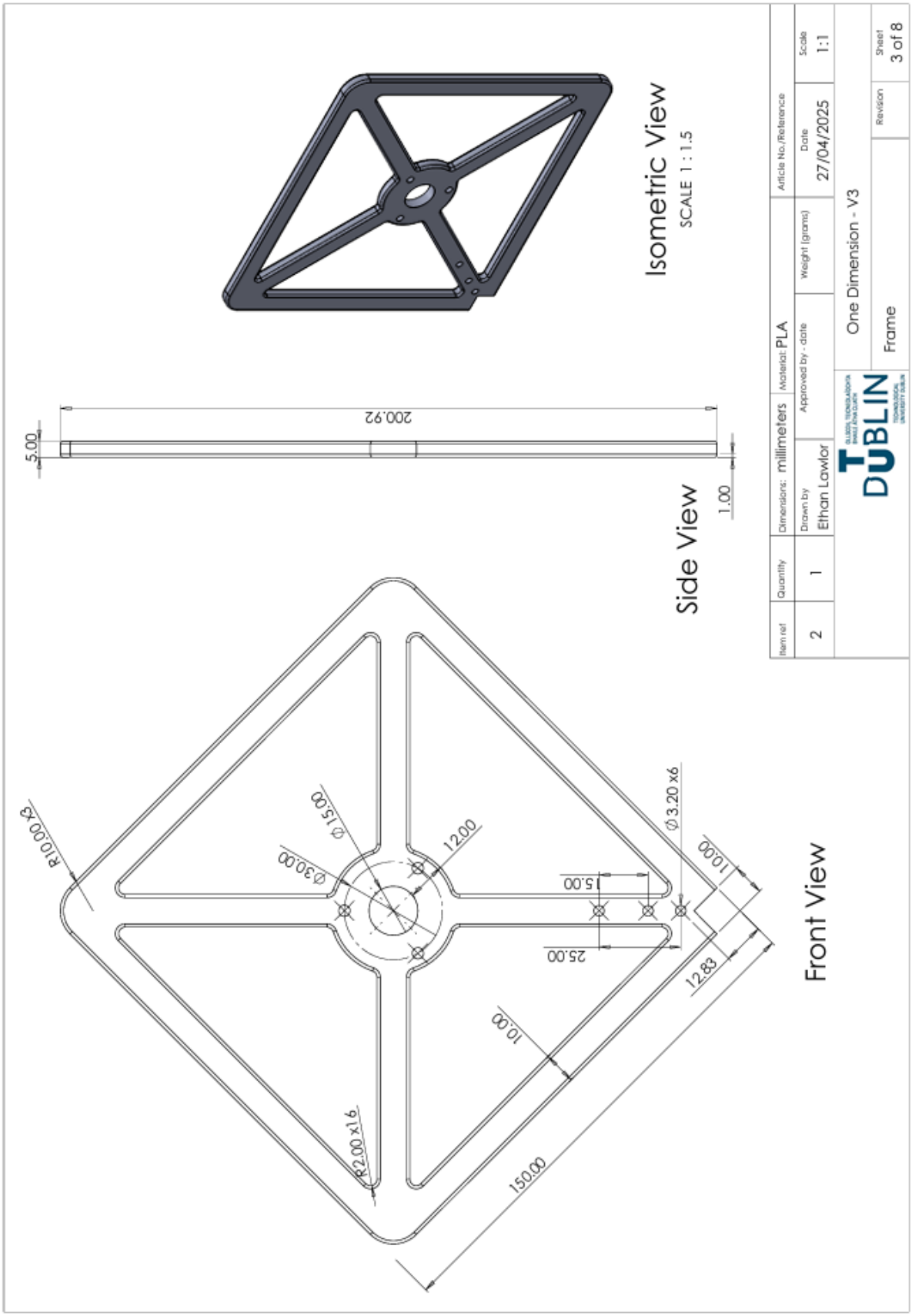
Extract 10 18 Apr – 27 Apr 2025

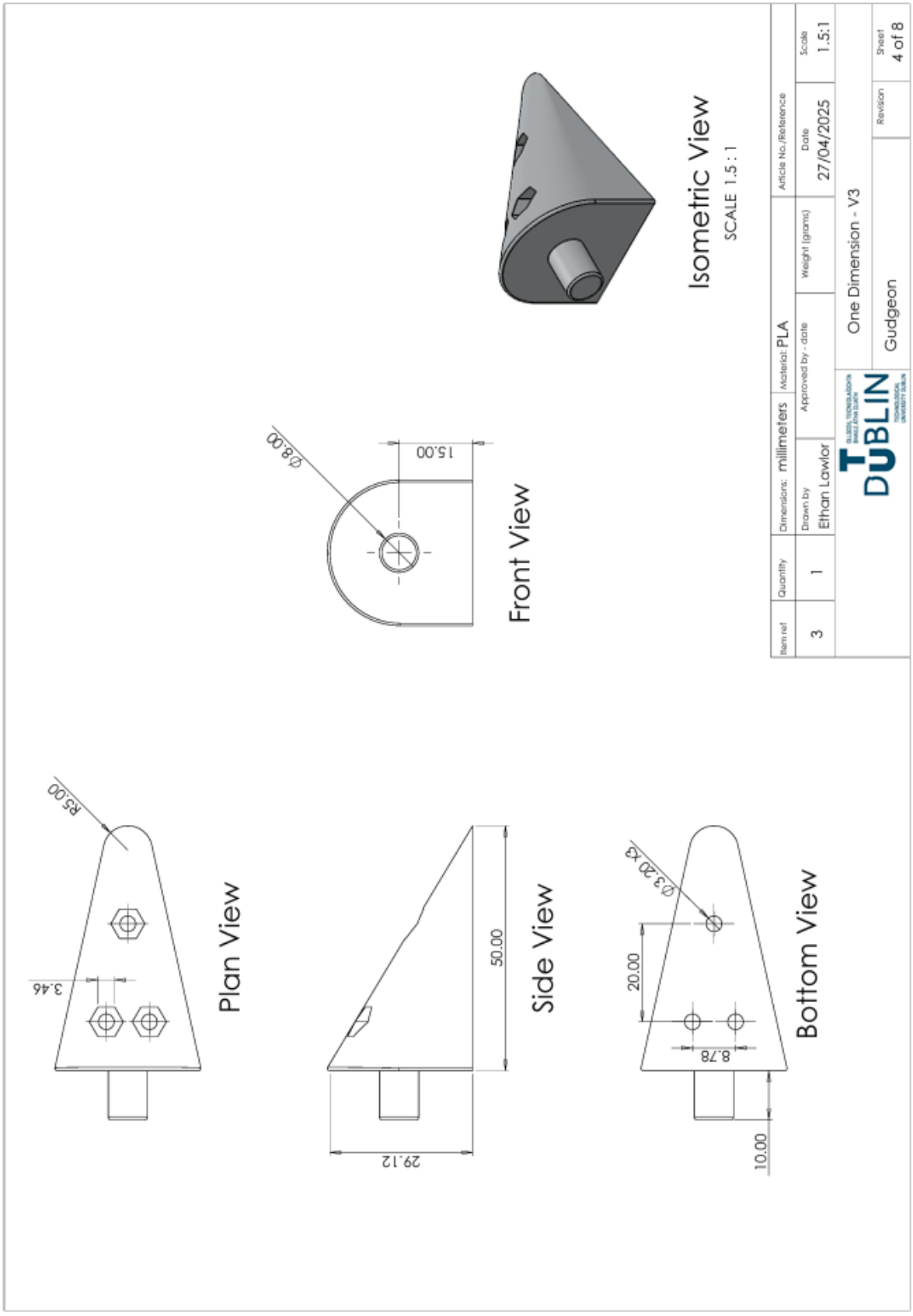
- Delivered draft to supervisor on the 24th; feedback mainly formatting, nothing structural—huge relief.
- Archived CAD, firmware and log files in the appendices with QR links; flagged availability "on request" rather than open-licence release.
- Completed last-pass proof-read, generated the PDF, and submitted the print order—ready for final hand-in on 2 May.

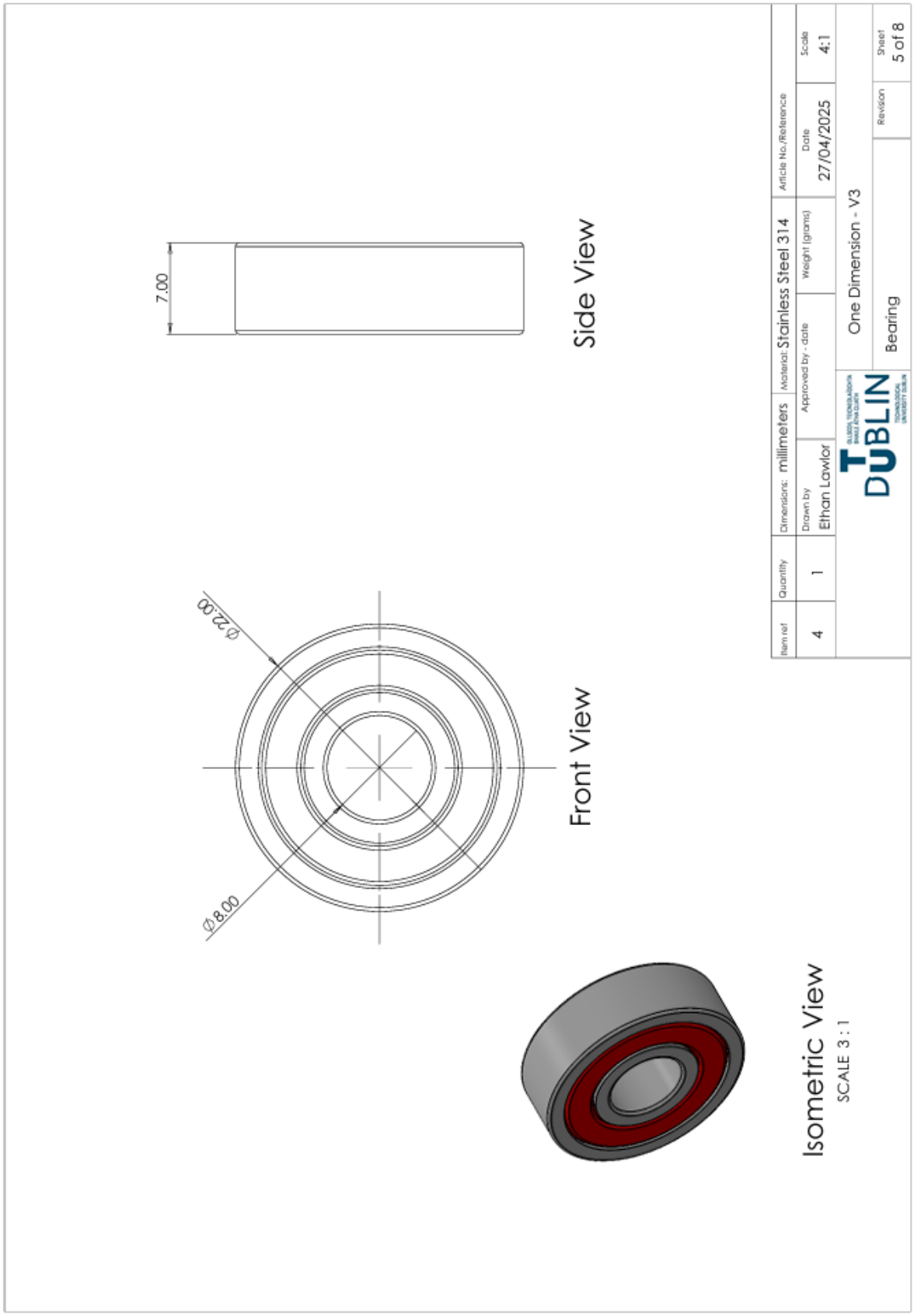
Appendix C Design Drawings & Component Specifications

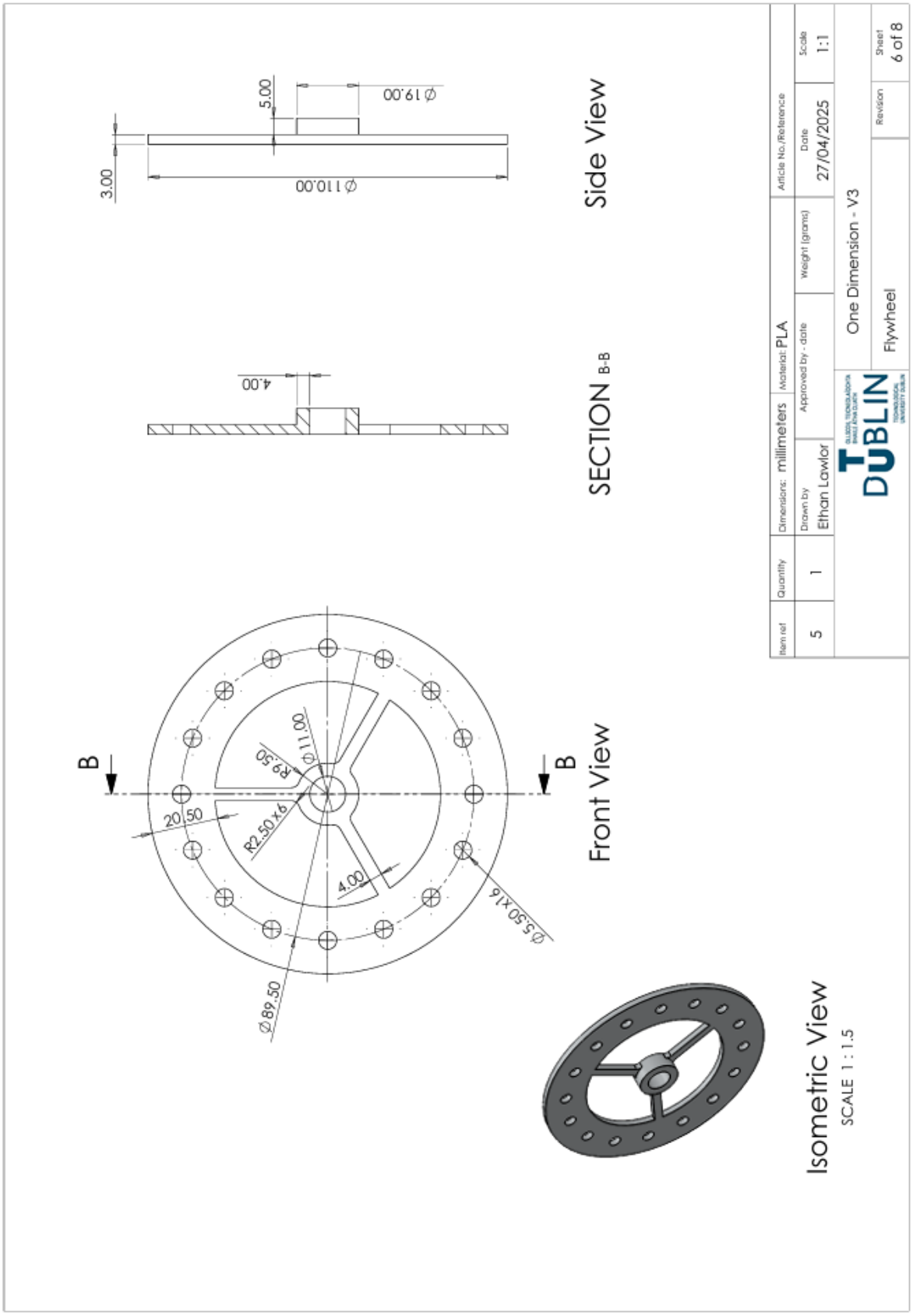


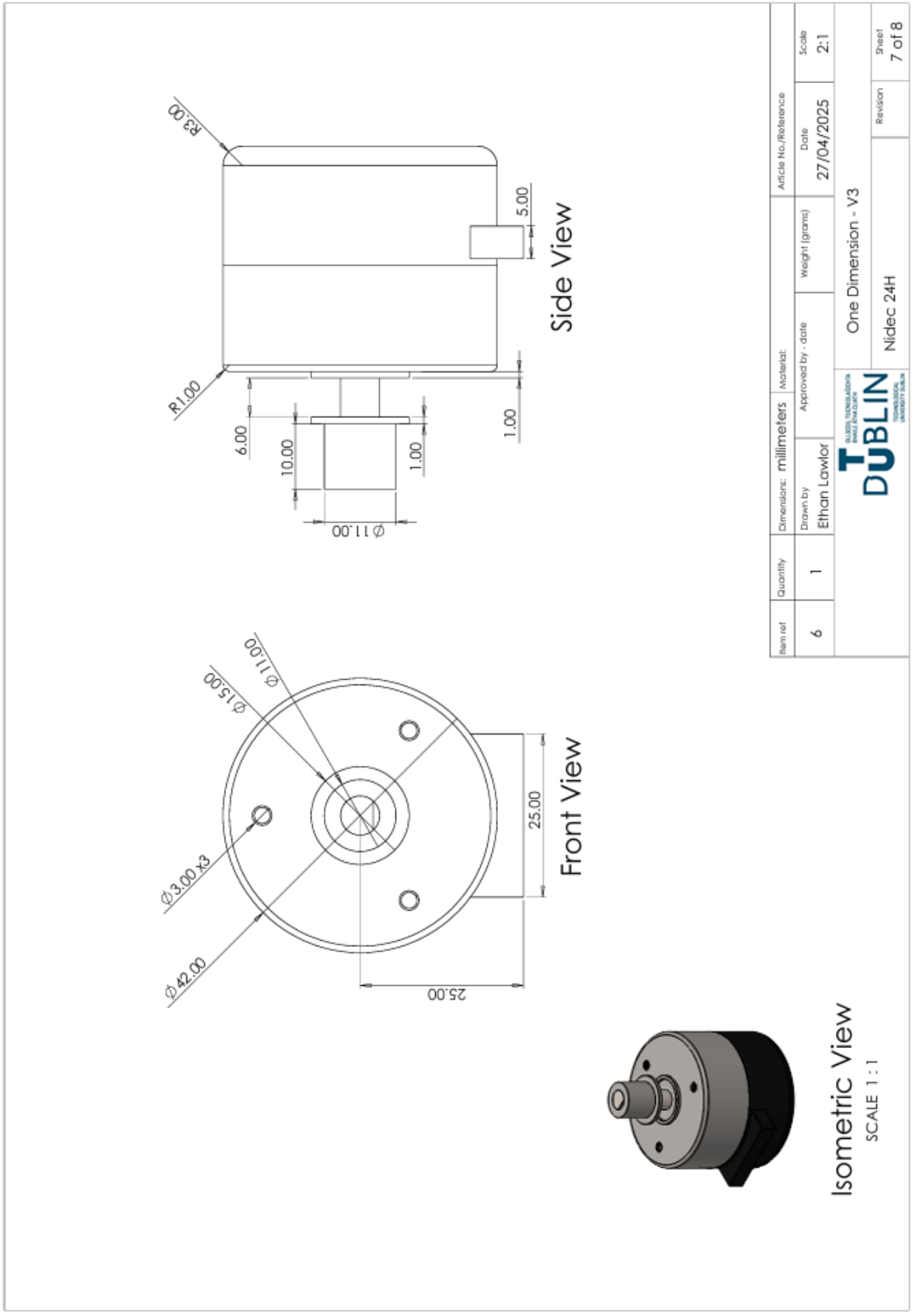

R11.50
R15.00
R1.00
14.97
Back View
3.00
5.00
12.00
Side View
SECTION A-A
R15.00
10.00
5.00
R4.00
Ø3.20
90°
90°
18.14
A
A
Front View
Isometric View
SCALE 1.5 : 1
Item ref
Quantity
1
1
Dimensions: millimeters
Drawn by
Ethan Lawlor
Material: PLA
Approved by - date
Weight (grams)
Article No./Reference
Date
27/04/2025
Scale
2:1
One Dimension - V3
Revision
Sheet
2 of 8
Pintle
TU DUBLIN

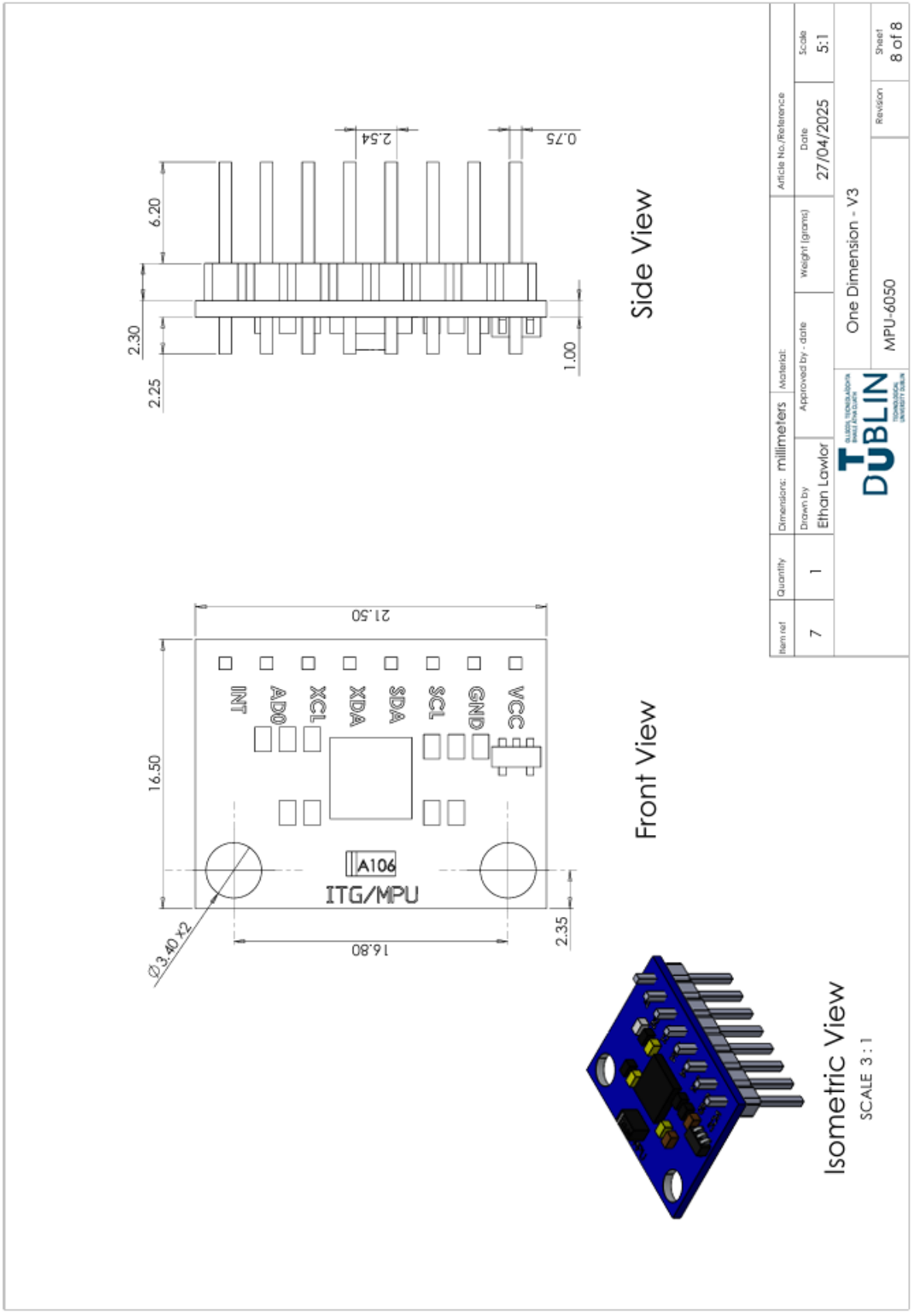

Isometric View
SCALE 1 : 1.5
Side View
200.92
5.00
1.00
Front View
R10.00 x3
Ø15.00
Ø30.00
12.00
Ø3.20 x6
10.00
5.00
25.00
12.83
10.00
R2.00 x16
150.00
Item ref
Quantity
Dimensions: millimeters
Material: PLA
Article No./Reference
2
1
Drawn by
Ethan Lawlor
Approved by - date
Weight (grams)
Date
27/04/2025
Scale
1:1
One Dimension - V3
Frame
Revision
Sheet
3 of 8
DUBLIN

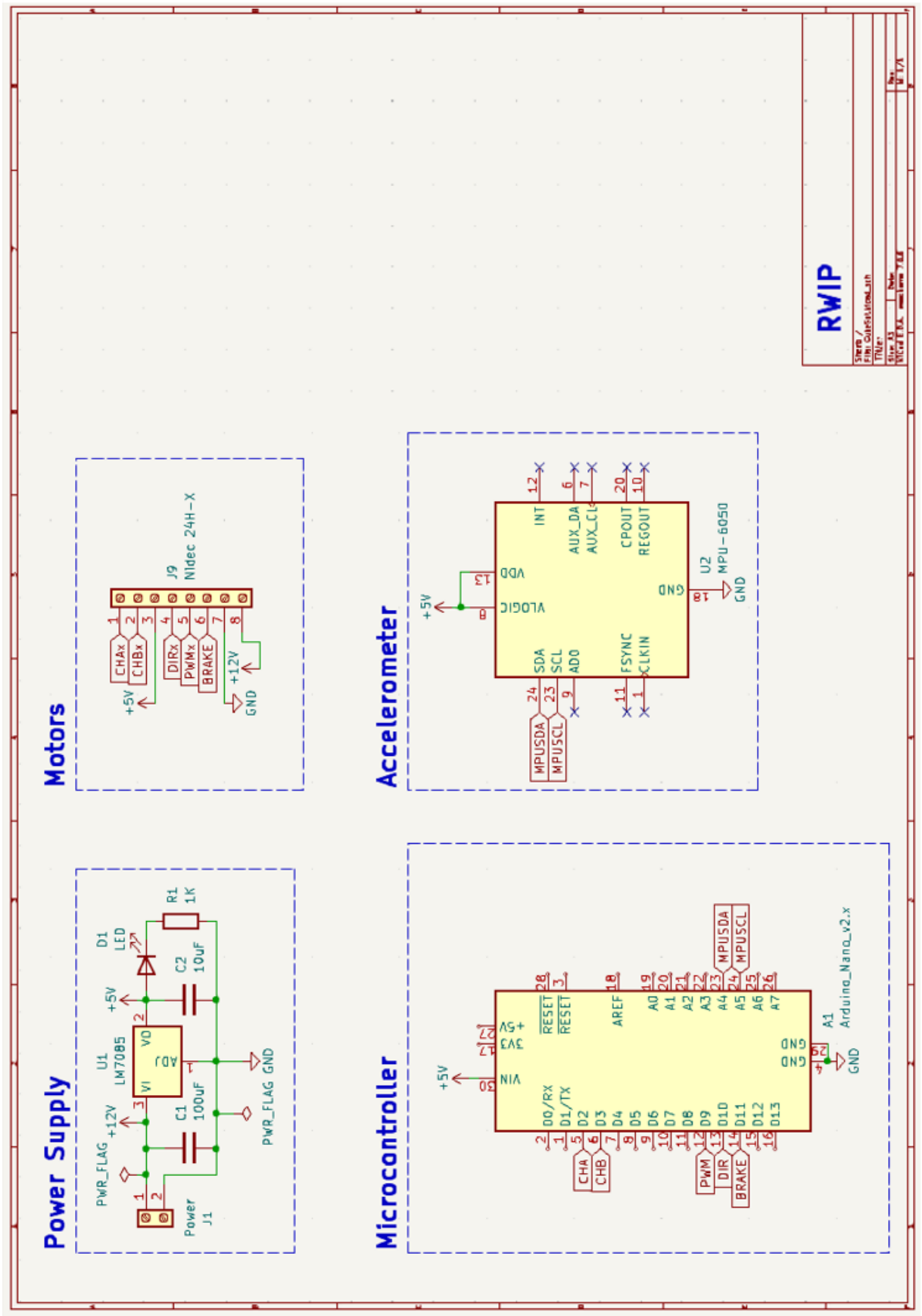

Isometric View
SCALE 1.5 : 1
Front View
Ø8.00
15.00
Plan View
R5.00
3.46
Side View
50.00
29.12
Bottom View
Ø3.20
20.00
8.78
10.00
Item ref
Quantity
Dimensions: millimeters
Material: PLA
Article No./Reference
3
1
Drawn by
Ethan Lawlor
Approved by - date
Weight (grams)
Date
27/04/2025
Scale
1.5:1
One Dimension - V3
Revision
Sheet
4 of 8
Gudgeon


7.00
Side View
Ø22.00
Ø8.00
Front View
Isometric View
SCALE 3 : 1
Item ref
Quantity
Dimensions: millimeters
Material: Stainless Steel 314
Article No./Reference
4
1
Drawn by
Ethan Lawlor
Approved by - date
Weight (grams)
Date
27/04/2025
Scale
4:1
One Dimension - V3
Bearing
Revision
Sheet
5 of 8
DUBLIN


Side View
5.00
Ø19.00
3.00
Ø110.00
SECTION B-B
4.00
Front View
B
B
Ø11.00
R9.50
20.50
R2.50 x 6
4.00
Ø5.50 x 16
Ø89.50
Isometric View
SCALE 1 : 1.5
Item ref
Quantity
Dimensions: millimeters
Material: PLA
Article No./Reference
5
1
Drawn by
Ethan Lawlor
Approved by - date
Weight (grams)
Date
27/04/2025
Scale
1:1
One Dimension - V3
Revision
Sheet
6 of 8
Flywheel
DUBLIN


Side View
R3.00
5.00
R1.00
6.00
10.00
1.00
1.00
Ø11.00
Front View
Ø11.00
Ø15.00
25.00
Ø3.00 x3
Ø42.00
25.00
Isometric View
SCALE 1 : 1
Item ref
Quantity
Dimensions: millimeters
Material:
Article No./Reference
6
1
Drawn by
Ethan Lawlor
Approved by - date
Weight (grams)
Date
27/04/2025
Scale
2:1
One Dimension - V3
Nidec 24H
Revision
Sheet
7 of 8
DUBLIN


Side View
2.54
0.75
6.20
2.30
2.25
1.00
Front View
21.50
16.50
INT
ADO
XCL
XDA
SDA
SCL
GND
VCC
A106
ITG/MPU
Ø3.40 x2
16.80
2.35
Isometric View
SCALE 3 : 1
Item ref
7
Quantity
1
Dimensions: millimeters
Drawn by
Ethan Lawlor
Material:
Approved by - date
Weight (grams)
Article No./Reference
Date
27/04/2025
Scale
5:1
One Dimension - V3
MPU-6050
Revision
Sheet
8 of 8
DUBLIN


RWIP
Motors
Accelerometer
Power Supply
Microcontroller
J9
Nidec 24H-X
CHAx
CHBx
DIRx
PWMx
BRAKE
+12V
GND
+5V
U2
MPU-6050
INT
AUX_DA
AUX_CL
CPOUT
REGOUT
VDD
VLOGIC
GND
SDA
SCL
ADO
FSYNC
XCLKIN
MPUSDA
MPUSCL
D1
LED
R1
1K
C2
10uF
U1
LM7805
VI
VO
ADJ
C1
100uF
PWR_FLAG
PWR_FLAG GND
Power
J1
A1
Arduino_Nano_v2.x
VIN
+5V
3V3
RESET
AREF
D0/RX
D1/TX
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
A0
A1
A2
A3
A4
A5
A6
A7
CHA
CHB
PWM
DIR
BRAKE

Appendix D List of Software Code

```
/*
  Reaction Wheel Inverted Pendulum

  Cascade Loop - Speed of Motor as Outer, Angle from Vertical as Inner

  Ethan Lawlor - B00149346
  27/04/2025
*/

//#define DEBUG_MPU
//#define DEBUG_ENCODER
//#define DEBUG_SPEED_PID
//#define DEBUG_ANGLE_PID

// MPU6050 Globals ---------------------------------------------------------------------
#include <Wire.h>          // Include the Wire library for I2C communication
#include <MPU6050.h>        // Include the MPU6050 library

MPU6050 mpu;
int16_t ax, ay, az, gx, gy, gz;
float gz_offset = 0;

struct mpuData {
  float theta_z;
  float omega_z;
};
// -------------------------------------------------------------------------------------

// Encoder Globals ---------------------------------------------------------------------
volatile long counter = 0;
long lastCounter = 0;
unsigned long lastMillis = 0;

struct encoderData {
  float rpm;
  int direction;
};
// -------------------------------------------------------------------------------------

// Motor Globals -----------------------------------------------------------------------
#include <PWM.h>
#define Nidec_Frequency 30000
#define motorPWM_Pin 9
#define motorDIR_Pin 10
#define motorBrake_Pin 11
int motorPWM = 255;
// -------------------------------------------------------------------------------------

// LED Globals -------------------------------------------------------------------------
#define LED_PIN 13
bool blinkState = false;
// -------------------------------------------------------------------------------------

// Kalman Filter Globals ---------------------------------------------------------------
```

```
float KalmanAngleYaw=0, KalmanUncertaintyAngleYaw=2*2;

struct kalmanData {
  float angle;
  float uncertainty;
};
// ---------------------------------------------------------------------------------------------

// Speed PID Globals ---------------------------------------------------------------------------
float sKp, sKi, sKd;
float sTarget = 0, sLastErr = 0, sErrSum = 0, sErrOrderF = 0, sErrOrderFprev = 0, sErrorPrev
= 0;
double sLastTime;
// ---------------------------------------------------------------------------------------------

// Angle PID Globals ---------------------------------------------------------------------------
--------
float aKp, aKi, aKd;
float aLastErr = 0, aErrSum = 0, aErrOrderF = 0, aErrOrderFprev = 0, aErrorPrev = 0;
double aLastTime;
// ---------------------------------------------------------------------------------------------

// Main Loop Globals ---------------------------------------------------------------------------
#include <EEPROM.h>
float beta = 0.996;
float rpm;
bool run = true;
bool stream = false;
unsigned long lastStreamTime, streamInterval;
// ---------------------------------------------------------------------------------------------

void setup() {
  Serial.begin(38400);
  Serial.println("<Arduino Initializing>");

  motorInit();  // Initialize motor
  encoderInit();  // Initialize encoder and interrupts
  mpuInit();  // Initialize mpu
  pidInit();  // Initialize PID variables with EEPROM

  Serial.println("<Arduino Ready>");
  delay(500);
}

void loop() {
  if (Serial.available()){ // Detects Serial Data
    tunePID(); // Check for Commands In Serial Data
  }

  mpuData mpuData = getMpuData(); // Get Angle from the MPU
  // Apply kalman Filter
  kalmanData kalmanData = kalman(kalmanData.angle, kalmanData.uncertainty, mpuData.omega_z,
mpuData.theta_z);

  encoderData encoderData = getEncoderData(); // Get Speed from Encoder
  rpm = rpm - (beta * (rpm - encoderData.rpm)); // Apply Low Pass Filter
```

```
  float controlSpeed = speedPID(encoderData.rpm); // Apply Inner Speed PID
  float controlSignal = anglePID(controlSpeed, kalmanData.angle); // Apply Outer Angle PID

  if (run && abs(kalmanData.angle) < 20){ // Run Command and Upright?
    digitalWrite(motorDIR_Pin, (controlSignal >= 0) ? 0 : 1); // Set Direction
    analogWrite(motorPWM_Pin, (255-abs(int(controlSignal)))); // Set Speed
    digitalWrite(motorBrake_Pin, HIGH); // Disengage Brake
  }else{                              // No Run Command or Angle Too Large?
    analogWrite(motorPWM_Pin, 255); // Minimum Speed
    digitalWrite(motorBrake_Pin, LOW); // Engage Brake
  }

  unsigned long currentMillis = millis(); // Get the current time
  // At Least 60ms Since the Last Print?
  if (currentMillis - lastStreamTime >= streamInterval) {
    if (stream){ // Stream Data Command?
    Serial.print("SET:"); Serial.print(0); Serial.print(",");
    Serial.print("Angle:"); Serial.print(kalmanData.angle, 4); Serial.print(",");
    Serial.print("RPM:"); Serial.print(rpm, 4); Serial.print(",");
    Serial.print("PIDspeed:"); Serial.print(setPoint, 4); Serial.print(",");
    Serial.print("PIDangle:"); Serial.println(controlSignal, 4);
    }
    lastStreamTime = currentMillis; // Update the last execution time
  }
}

void ai0() {
  // Interrupt for pin 2 (ChA signal)
  if (digitalRead(3) == LOW) {
    counter++;  // Counter increments if B signal is LOW
  } else {
    counter--;  // Counter decrements if B signal is HIGH
  }
}

void ai1() {
  // Interrupt for pin 3 (ChB signal)
  if (digitalRead(2) == LOW) {
    counter--;  // Counter decrements if A signal is LOW
  } else {
    counter++;  // Counter increments if A signal is HIGH
  }
}

float speedPID(float rpm){
  unsigned long sCurrentTime = millis();
  float sTime = (sCurrentTime - sLastTime)/1000;

  float sError = sTarget - rpm;

  sErrSum += sError * sTime;
  sErrSum = constrain(sErrSum, -15, 15);

  float sErrOrder = (sError - sLastErr) / sTime;
  sErrOrderF = (0.9 * sErrOrderFprev) + (0.1 * sErrOrder);
  sErrOrderFprev = sErrOrderF;
```

```
  float sPterm = sKp * sError;
  sPterm = constrain(sPterm, -3, 3);
  float sIterm = sKi * sErrSum;
  float sDterm = sKd * sErrOrder;

  float setPoint = sPterm + sIterm + sDterm;

  sLastErr = sError;
  sLastTime = sCurrentTime;

  #ifdef DEBUG_SPEED_PID
  Serial.print("sPterm:"); Serial.print(sPterm, 8); Serial.print(",");
  Serial.print("sIterm:"); Serial.print(sIterm); Serial.print(",");
  Serial.print("sDterm:"); Serial.println(sDterm);
  #endif

  return setPoint;
}

float anglePID(float setPoint, float angle){
  unsigned long aCurrentTime = millis();
  float aTime = (aCurrentTime - aLastTime)/1000;

  float aError = (controlSpeed - angle)*10;

  aErrSum += aError * aTime;
  aErrSum = constrain(aErrSum, -50, 50);

  float aErrOrder = ((aError - aLastErr) / aTime);
  aErrOrderF = (0.9 * aErrOrderFprev) + (0.1 * aErrOrder);
  aErrOrderFprev = aErrOrderF;

  float aPterm = aKp * aError;
  aPterm = constrain(aPterm, -255, 255);

  float aIterm = aKi * aErrSum;
  float aDterm = aKd * aErrOrderF;

  float controlSignal1 = aPterm + aIterm + aDterm;
  controlSignal1 = constrain(controlSignal1, -255, 255);

  aLastErr = aError;
  aLastTime = aCurrentTime;

  #ifdef DEBUG_ANGLE_PID
  Serial.print("aIterm:"); Serial.print(aIterm); Serial.print(",");
  Serial.print("sIterm:"); Serial.print(sIterm); Serial.print(",");
  Serial.print("sDterm:"); Serial.println(sDterm);
  #endif

  return controlSignal1;
}

kalmanData kalman(float KalmanState, float KalmanUncertainty, float KalmanInput, float
KalmanMeasurement) {
  KalmanState=KalmanState+0.004*KalmanInput;
  KalmanUncertainty=KalmanUncertainty + 0.004 * 0.004 * 4 * 4;
```

```
  float KalmanGain=KalmanUncertainty * 1/(1*KalmanUncertainty + 3 * 3);
  KalmanState=KalmanState+KalmanGain * (KalmanMeasurement-KalmanState);
  KalmanUncertainty=(1-KalmanGain) * KalmanUncertainty;

  kalmanData result;
  result.angle = KalmanState;
  result.uncertainty = KalmanUncertainty;

  #ifdef DEBUG_KALMAN
  Serial.print("theta_z:"); Serial.print(theta_z); Serial.print(",");
  Serial.print("KalmanState:"); Serial.print(KalmanState); Serial.print(",");
  Serial.print("KalmanUncertainty:"); Serial.println(KalmanUncertainty);
  #endif

  return result;
}

mpuData getMpuData() {
  const float ACCEL_SCALE = 16384.0;  // ±2g range
  const float GYRO_SCALE = 131.0;     // ±250°/s range
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);  // Read sensor data

  // Convert accelerometer raw values to g-force (after offset correction)
  float Ax = (ax) / ACCEL_SCALE;
  float Ay = (ay) / ACCEL_SCALE;

  // Compute tilt angles (theta_x, theta_y, theta_z) in degrees
  float theta_z = ((atan2(Ay, Ax) * 180.0) / PI);

  // Convert gyroscope raw values to angular velocity in °/s (after offset correction)
  float omega_z = (gz - gz_offset) / -GYRO_SCALE;

  #ifdef DEBUG_MPU
  Serial.print("theta_z:"); Serial.print(theta_z); Serial.print(",");
  Serial.print("omega_z:"); Serial.println(omega_z);
  #endif

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

  mpuData result;
  result.theta_z = theta_z;
  result.omega_z = omega_z;

  return result;
}

encoderData getEncoderData(){
  unsigned long currentMillis = millis();
  unsigned long time = (currentMillis - lastMillis);

  if (time > 10){
    long count = (counter - lastCounter);

    bool encDIR;
```

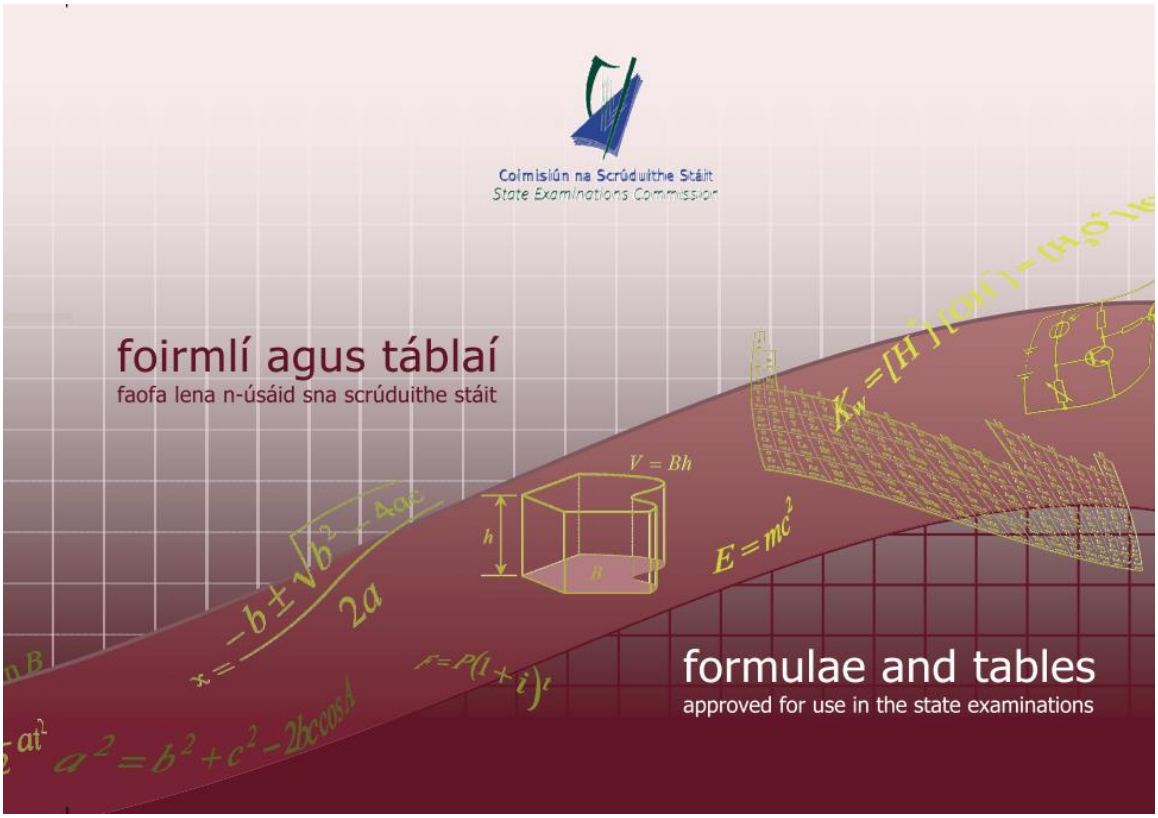
```
    float seconds = (time/1000.0);

    float revolutions = count/200.0;

    float rps = revolutions/seconds;

    if (counter > lastCounter){
      encDIR = 1;
    } else{
      encDIR = -1;
    }

    lastMillis = currentMillis;
    lastCounter = counter;

    encoderData result;
    result.rpm = rps;// * encDIR;
    result.direction = encDIR;

    #ifdef DEBUG_ENCODER
    Serial.print("rps:");  Serial.print(rps, 3); Serial.print(",");
    Serial.print("encDIR:");  Serial.print(encDIR); Serial.print(",");
    Serial.print("counter:");  Serial.println(counter);
    #endif

    return result;
  }
}

void mpuCalibrate() {
  int num_samples = 1000;  // Take the sum of n samples
  for (int i = 0; i < num_samples; i++) {
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
    gz_offset += gz;
    delay(2);
  }

  // Offset equals the average of the sample i.e. the average distance it is away from
origin
  gz_offset /= num_samples;

  Serial.print("Gyro_Z Offset: ");  Serial.println(gz_offset);
}

void pidInit(){
  // Read float values from EEPROM
  EEPROM.get(0, sKp); // Read sKp (float) from EEPROM starting at address 0
  EEPROM.get(4, sKi); // Read sKi (float) from EEPROM starting at address 4
  EEPROM.get(8, sKd); // Read sKd (float) from EEPROM starting at address 8
  EEPROM.get(12, aKp); // Read aKp (float) from EEPROM starting at address 12
  EEPROM.get(16, aKi); // Read aKi (float) from EEPROM starting at address 16
  EEPROM.get(20, aKd); // Read aKd (float) from EEPROM starting at address 20

  Serial.print("sKp = "); Serial.print(sKp, 3); Serial.print(" , ");
  Serial.print("sKi = "); Serial.print(sKi); Serial.print(" , ");
  Serial.print("sKd = "); Serial.print(sKd); Serial.print(" , ");
  Serial.print("aKp = "); Serial.print(aKp, 3); Serial.print(" , ");
```

```
  Serial.print("aKi = "); Serial.print(aKi); Serial.print(" , ");
  Serial.print("aKd = "); Serial.println(aKd, 3);
}

void mpuInit(){
  Wire.begin();
  mpu.initialize();
  Serial.println("Testing device connections...");
  Serial.println(mpu.testConnection() ? "MPU6050 connection successful" : "MPU6050
connection failed");
  mpuCalibrate();

  pinMode(LED_PIN, OUTPUT);
}

void encoderInit(){
  pinMode(2, INPUT_PULLUP); // Internal pull-up for pin 2
  pinMode(3, INPUT_PULLUP); // Internal pull-up for pin 3

  attachInterrupt(digitalPinToInterrupt(2), ai0, RISING);  // Interrupt on pin 2 (encoder A)
  attachInterrupt(digitalPinToInterrupt(3), ai1, RISING);  // Interrupt on pin 3 (encoder B)

  Serial.println("Encoder connection successful");
}

void motorInit(){
  InitTimersSafe();
  SetPinFrequencySafe(motorPWM_Pin, Nidec_Frequency);

  pinMode(motorPWM_Pin, OUTPUT);
  pinMode(motorDIR_Pin, OUTPUT);
  pinMode(motorBrake_Pin, OUTPUT);

  pwmWrite(motorPWM_Pin, 255);
  digitalWrite(motorDIR_Pin, HIGH);
  digitalWrite(motorBrake_Pin, LOW);

  Serial.println("Motor connection successful");
}

void tunePID(){
  char cmd = Serial.read();

  switch (cmd){
    case 'q': sKp += 0.001; EEPROM.put(0, sKp);
    break;

    case 'a': sKp -= 0.001; EEPROM.put(0, sKp);
    break;

    case 'w': sKi += 0.01; EEPROM.put(4, sKi);
    break;

    case 's': sKi -= 0.01; EEPROM.put(4, sKi);
    break;

    case 'e': sKd += 0.01; EEPROM.put(8, sKd);
```

```
    break;

    case 'd': sKd -= 0.01; EEPROM.put(8, sKd);
    break;

    case 'r': aKp += 0.01; EEPROM.put(12, aKp);
    break;

    case 'f': aKp -= 0.01; EEPROM.put(12, aKp);
    break;

    case 't': aKi += 0.01; EEPROM.put(16, aKi);
    break;

    case 'g': aKi -= 0.01; EEPROM.put(16, aKi);
    break;

    case 'y': aKd += 0.001; EEPROM.put(20, aKd);
    break;

    case 'h': aKd -= 0.001; EEPROM.put(20, aKd);
    break;

    case 'z':
      sKp = 0; sKi = 0; sKd = 0; aKp = 0; aKi = 0; aKd = 0;
      for (int i = 0; i < 25; i++){
        EEPROM.write(i, 0);
      }
    break;

    case 'x':
      sKp = 0.276; sKi = 0.02; sKd = 0; aKp = 2.470; aKi = 0; aKd = 0.200;
      EEPROM.put(0, sKp); EEPROM.put(4, sKi); EEPROM.put(8, sKd); EEPROM.put(12, aKp);
EEPROM.put(16, aKi); EEPROM.put(20, aKd);
    break;

    case 'c':
      sKp = 0.375; sKi = 0.02; sKd = 0; aKp = 2.070; aKi = 0.01; aKd = 0.300;
      EEPROM.put(0, sKp); EEPROM.put(4, sKi); EEPROM.put(8, sKd); EEPROM.put(12, aKp);
EEPROM.put(16, aKi); EEPROM.put(20, aKd);
    break;

    case 'l': run = !run;
    break;

    case 'p': stream = !stream;
    break;

    default: Serial.println("<Error - no tuning value>");
    cmd = Serial.read();
  }

  if (stream == false){
    Serial.print("sKp = "); Serial.print(sKp, 3); Serial.print(" , ");
    Serial.print("sKi = "); Serial.print(sKi); Serial.print(" , ");
    Serial.print("sKd = "); Serial.print(sKd); Serial.print(" , ");
    Serial.print("aKp = "); Serial.print(aKp, 3); Serial.print(" , ");
```

```
    Serial.print("aKi = "); Serial.print(aKi); Serial.print(" , ");
    Serial.print("aKd = "); Serial.println(aKd, 3);
  }
}
```

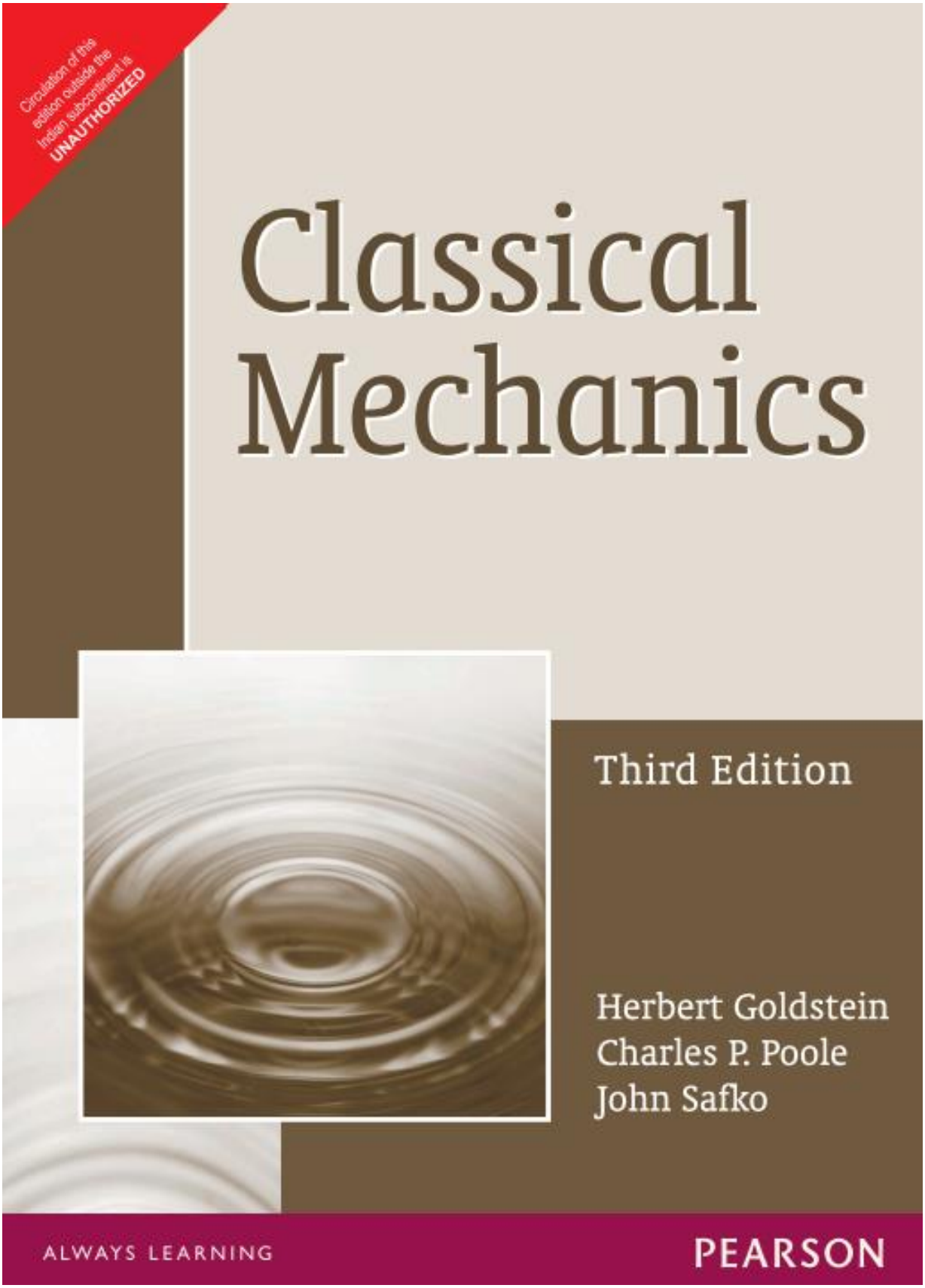
Appendix E Formulae Sheets and Rough Work



Móimintí táimhe		**Moments of inertia**
slat aonfhoirmeach, fad $2l$ timpeall aise trí lárphointe ingearach leis an tslat	$\frac{1}{3}ml^2$	uniform rod, length $2l$ about axis through centre perpendicular to rod
timpeall aise ag foirceann amháin ingearach leis an tslat	$\frac{4}{3}ml^2$	about axis at one end perpendicular to rod
diosca aonfhoirmeach, ga r timpeall aise trí lárphointe ingearach leis an diosca	$\frac{1}{2}mr^2$	uniform disc, radius r about axis through centre perpendicular to disc
timpeall trastomhais	$\frac{1}{4}mr^2$	about diameter
fonsa aonfhoirmeach, ga r timpeall aise trí lárphointe ingearach leis an bhfoinse	mr^2	uniform hoop, radius r about axis through centre perpendicular to hoop
timpeall trastomhais	$\frac{1}{2}mr^2$	about diameter
sféar soladach aonfhoirmeach, ga r timpeall trastomhais	$\frac{2}{5}mr^2$	uniform solid sphere, radius r about diameter
teoirim na n-aiseanna comhthreomhara	$I_b = I_c + md^2$	parallel axis theorem
teoirim na n-aiseanna ingearacha	$I_z = I_x + I_y$	perpendicular axis theorem

Coirp rothlacha		**Rotating bodies**
móiminteam uilleach	$L = I\omega = rmv$	angular momentum
móimint fórsa	$M = Fd$	moment of a force
torc cúpla	$T = Fd$	torque of a couple
dara dlí Newton don rothlú	$T = \frac{dL}{dt}$	Newton's 2nd law for rotation
fuinneamh cinéiteach rothlach	$E = \frac{1}{2}I\omega^2$	rotational kinetic energy
Gluaisne armónach shimplí		**Simple harmonic motion**
	$a = -\omega^2 s$	
	$T = \frac{1}{f} = \frac{2\pi}{\omega}$	
	$s = A\sin(\omega t + \alpha)$	
	$v^2 = \omega^2(A^2 - s^2)$	
luascadán simplí	$T = 2\pi\sqrt{\frac{l}{g}}$	simple pendulum
comhluascadán	$T = 2\pi\sqrt{\frac{I}{mgh}}$	compound pendulum

Fuinneamh agus obair		**Energy and work**
obair	$W = Fs = \int F ds$	work
cumhacht	$P = \frac{W}{t} = Fv$	power
céatadán éifeachtachta	$\frac{P_o \times 100}{P_i}$	percentage efficiency
fuinneamh poitéinsiúil (imtharraingthe)	$E_p = mgh$	potential energy (gravitational)
fuinneamh cinéiteach	$E_k = \frac{1}{2}mv^2$	kinetic energy
prionsabal imchoimeád an fhuinnimh (faoi fhórsaí meicniúla imchoimeádacha)	$\Delta E_p + \Delta E_k = 0$	principle of conservation of energy (under conservative mechanical forces)
coibhéis mhaise is fuinnimh	$E = mc^2$	mass-energy equivalence


Circulation of this edition outside the Indian subcontinent is UNAUTHORIZED
Classical Mechanics
Third Edition
Herbert Goldstein
Charles P. Poole
John Safko
ALWAYS LEARNING
PEARSON

See Eq. (1.47). Equations (1.53) can then be rewritten as

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}_j}\right) - \frac{\partial (T - V)}{\partial q_j} = 0. \tag{1.55}$$

The equations of motion in the form (1.55) are not necessarily restricted to conservative systems; only if V is not an explicit function of time is the system conservative (cf. p. 4). As here defined, the potential V does not depend on the generalized velocities. Hence, we can include a term in V in the partial derivative with respect to $\dot{q}_j$:

$$\frac{d}{dt}\left(\frac{\partial (T - V)}{\partial \dot{q}_j}\right) - \frac{\partial (T - V)}{\partial q_j} = 0.$$

Or, defining a new function, the *Lagrangian* L, as

$$L = T - V, \tag{1.56}$$

the Eqs. (1.53) become

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_j}\right) - \frac{\partial L}{\partial q_j} = 0, \tag{1.57}$$

expressions referred to as "Lagrange's equations."

Note that for a particular set of equations of motion there is no unique choice of Lagrangian such that Eqs. (1.57) lead to the equations of motion in the given generalized coordinates. Thus, in Derivations 8 and 10 it is shown that if $L(q, \dot{q}, t)$ is an approximate Lagrangian and $F(q, t)$ is *any* differentiable function of the generalized coordinates and time, then

$$L'(q, \dot{q}, t) = L(q, \dot{q}, t) + \frac{dF}{dt} \tag{1.57'}$$

is a Lagrangian also resulting in the same equations of motion. It is also often possible to find alternative Lagrangians beside those constructed by this prescription (see Exercise 20). While Eq. (1.56) is always a suitable way to construct a Lagrangian for a conservative system, it does not provide the *only* Lagrangian suitable for the given system.

1.5 ■ VELOCITY-DEPENDENT POTENTIALS AND THE DISSIPATION FUNCTION

Lagrange's equations can be put in the form (1.57) even if there is no potential function, V, in the usual sense, providing the generalized forces are obtained from a function $U(q_j, \dot{q}_j)$ by the prescription

$$Q_j = -\frac{\partial U}{\partial q_j} + \frac{d}{dt}\left(\frac{\partial U}{\partial \dot{q}_j}\right). \tag{1.58}$$

University Physics Volume 1

Senior Contributing Authors

Samuel J. Ling, Truman State University
Jeff Sanny, Loyola Marymount University
Bill Moebs, PhD

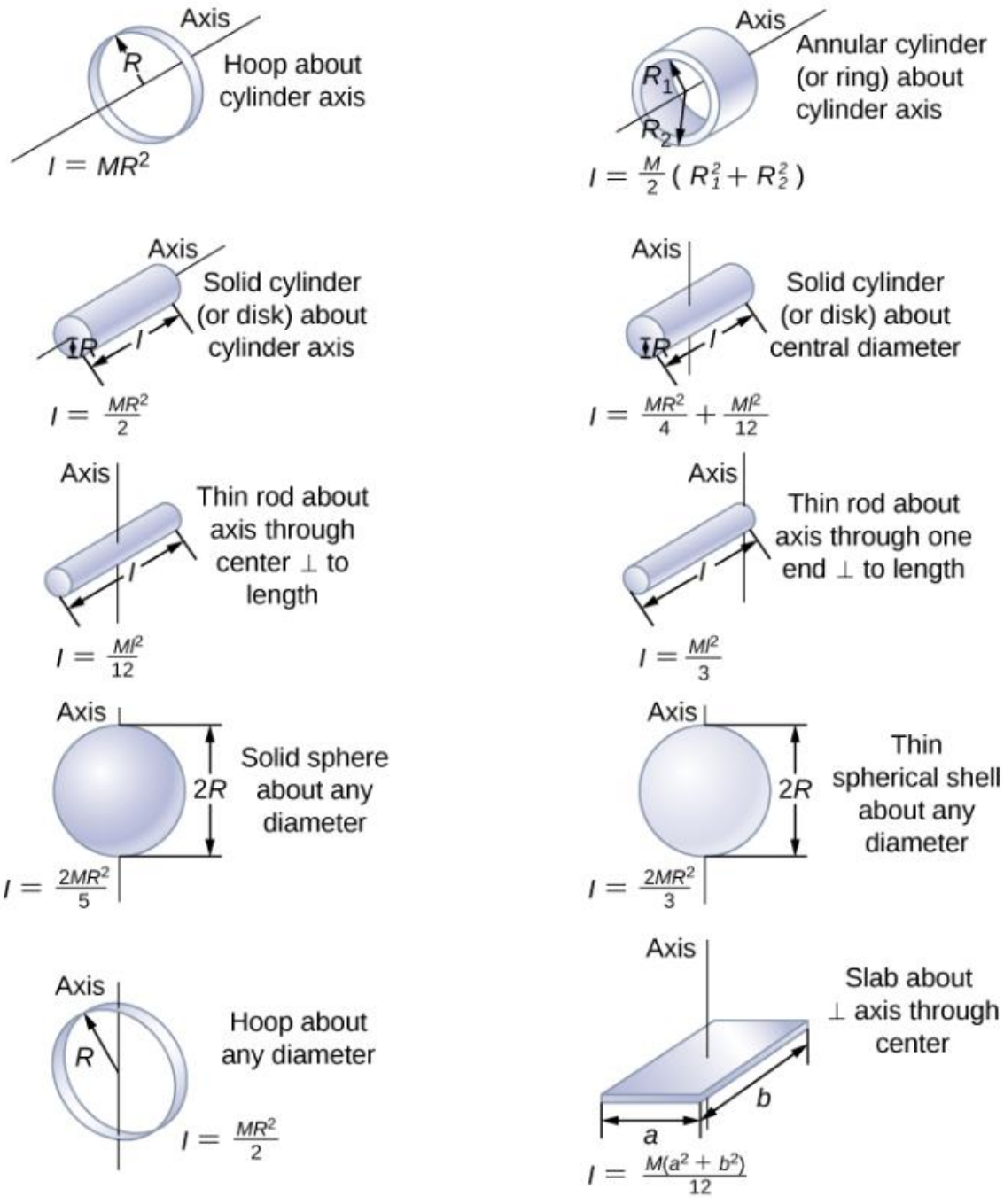
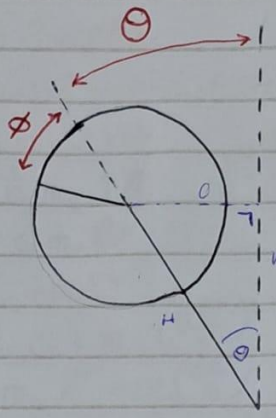
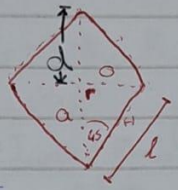


Figure 10.20 Values of rotational inertia for common shapes of objects



$\cos\theta = \frac{x}{d}$

$x = d\cos\theta$



$\sin\theta = \frac{O}{H}$

$\sin 45 = \frac{d}{\ell}$

$\therefore d = \frac{\ell\sqrt{2}}{2}$

Rotational Kinetic Energy

Tables $T = E_K = \frac{1}{2} I \omega^2$

Angular Momentum.

Tables $L = I\omega = RmV$

Moment of Inertia for Annular Cylinder & for slab through centre

University Physics $I = \frac{1}{2} m(R_1^2 + R_2^2)$ $I = \frac{1}{12} m(a^2 + b^2)$

Parallel Axis Theorem.

Tables $I_b = I_c + md^2$

Potential Energy

Tables $U = E_p = mgh$

The Lagrange

Harvard $L = T - V$

Euler Lagrange Equation

Harvard $\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) = \frac{\partial L}{\partial \theta} \Rightarrow \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \left(\frac{\partial L}{\partial \theta}\right) = \tau_{Reaction}$

Moments of Inertia

I_{SG} = Moment of Inertia of structure about its centre of mass ($kg.m^2$)

structure

$I = \frac{1}{12} m(a^2 + b^2)$

$I_{SG} = \frac{1}{12} m_S 2\ell^2$

$I_{SG} = \frac{1}{6} m_S \ell^2$

$I_b = I_c + md^2$

$I_{SO} = I_{SG} + m_S \left(\frac{\ell\sqrt{2}}{2}\right)^2$

$I_{SO} = \frac{1}{6} m_S \ell^2 + m_S \left(\frac{2\ell^2}{4}\right)$

$I_{SO} = \frac{2}{12} m_S \ell^2 + \frac{6}{12} m_S \ell^2$

$I_{SO} = \frac{2}{3} m_S \ell^2$

wheel

$I = \frac{1}{2} m(R_1^2 + R_2^2)$

$I_{\omega G} = \frac{1}{2} m_\omega R$

$I_b = I_c + md^2$

$I_{\omega O} = I_{\omega G} + m_\omega \left(\frac{\ell\sqrt{2}}{2}\right)^2$

$I_{\omega O} = \frac{1}{2} m_\omega R + m_\omega \left(\frac{2\ell^2}{4}\right)$

$I_{\omega O} = \frac{1}{2} m_\omega R + m_\omega \frac{1}{2} \ell^2$

$I_{\omega O} = \frac{1}{2} m_\omega (R + \ell^2)$

Kinetic Energy

$T = \frac{1}{2} I \omega^2$

$T_S = \frac{1}{2} I_{SO} \dot{\theta}^2$

$T = \frac{1}{2} I \omega^2$

$T_\omega = \frac{1}{2} I_{\omega O} \dot{\phi}^2$

Potential Energy

$U = mgh$

$U_S = -m_S g d \cos\theta$

$U = mgh$

$U_\omega = -m_\omega g d \cos\theta$

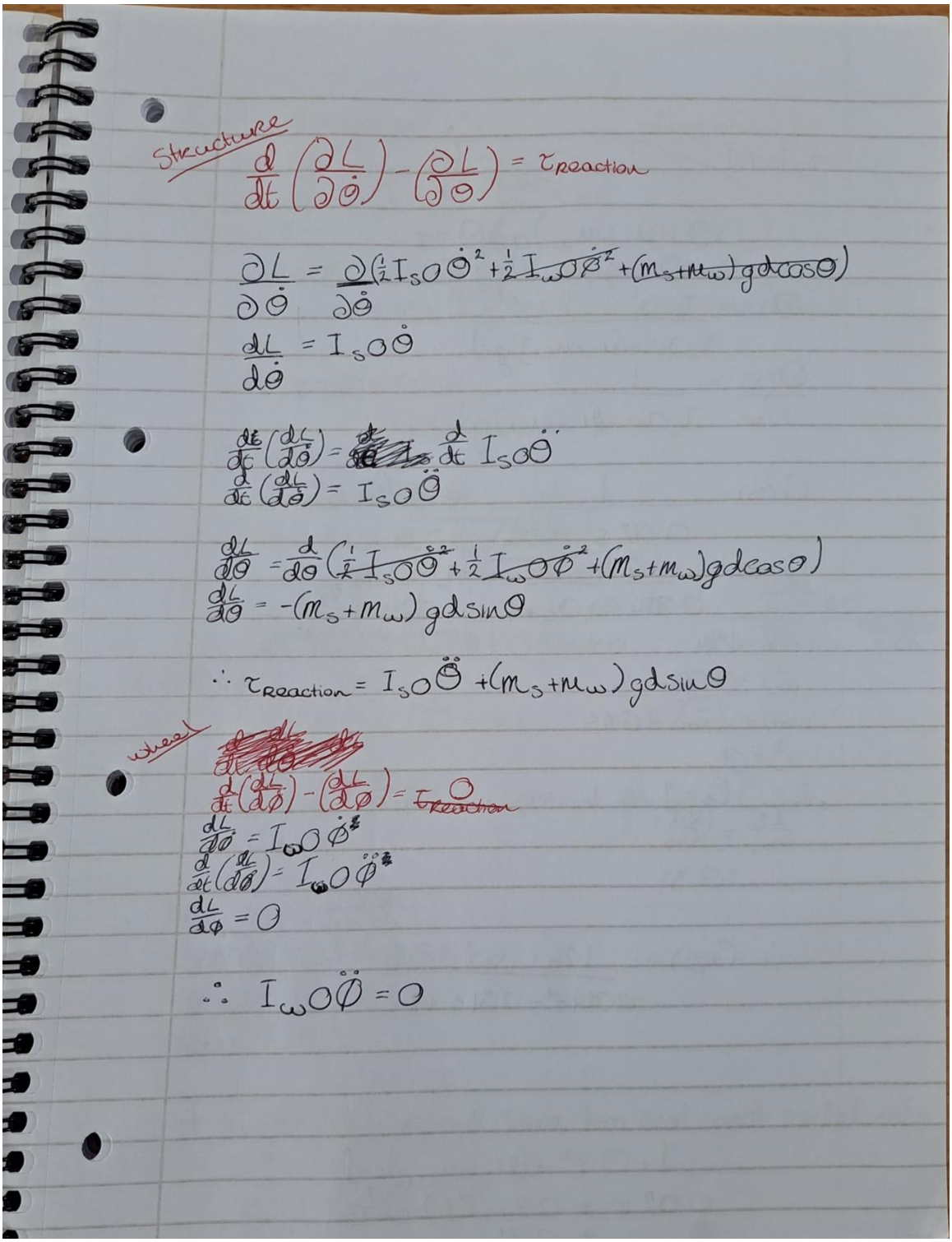
Equation of Motion (Lagrangian)

$L = T - U$

$L = T_S + T_\omega - U_S - U_\omega$

$L = \frac{1}{2} I_{SO} \dot{\theta}^2 + \frac{1}{2} I_{\omega O} \dot{\phi}^2 - (-m_S g d \cos\theta) - (-m_\omega g d \cos\theta)$

$L = \frac{1}{2} I_{SO} \dot{\theta}^2 + \frac{1}{2} I_{\omega O} \dot{\phi}^2 + (m_S + m_\omega) g d \cos\theta$



Structure

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \left(\frac{\partial L}{\partial \theta}\right) = \tau_{reaction}$$

$$\frac{\partial L}{\partial \dot{\theta}} = \frac{\partial}{\partial \dot{\theta}}\left(\tfrac{1}{2} I_s \dot{\theta}^2 + \tfrac{1}{2} I_\omega \dot{\phi}^2 + (m_s + m_\omega) g d \cos\theta\right)$$

$$\frac{dL}{d\dot{\theta}} = I_s \dot{\theta}$$

$$\frac{d}{dt}\left(\frac{dL}{d\dot{\theta}}\right) = \frac{d}{dt} I_s \dot{\theta}$$

$$\frac{d}{dt}\left(\frac{dL}{d\dot{\theta}}\right) = I_s \ddot{\theta}$$

$$\frac{dL}{d\theta} = \frac{d}{d\theta}\left(\tfrac{1}{2} I_s \dot{\theta}^2 + \tfrac{1}{2} I_\omega \dot{\phi}^2 + (m_s + m_\omega) g d \cos\theta\right)$$

$$\frac{dL}{d\theta} = -(m_s + m_\omega) g d \sin\theta$$

$$\therefore \tau_{reaction} = I_s \ddot{\theta} + (m_s + m_\omega) g d \sin\theta$$

wheel

$$\frac{d}{dt}\left(\frac{dL}{d\dot{\phi}}\right) - \left(\frac{dL}{d\phi}\right) = 0$$

$$\frac{dL}{d\dot{\phi}} = I_\omega \dot{\phi}$$

$$\frac{d}{dt}\left(\frac{dL}{d\dot{\phi}}\right) = I_\omega \ddot{\phi}$$

$$\frac{dL}{d\phi} = 0$$

$$\therefore I_\omega \ddot{\phi} = 0$$