MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Project for Control Engineering II					
Student	Șerban Iustinian-Bogdan	Group	30135	Grade	

Self-Balancing Cube Using Reaction Wheels and Gyroscope

University Year: 2024-2025

Author: Șerban Iustinian-Bogdan
Group: 30135/2

Project for Control Engineering II					
Student	Șerban Iustinian-Bogdan	Group	30135	Grade	

Contents

Project for Control Engineering II					
Student	Șerban Iustinian-Bogdan	Group	30135	Grade	

Figures

Tables

Equations

Acronyms

Symbol	Description
ADC	Analog to Digital Converter
ARX	AutoRegressive with eXogenous input
BLE	Bluetooth Low Energy
BLDC	Brushless Direct Current
COTS	Commercial Off-the-Self
DAC	Digital to Analog Converter
DOT	Degrees of Freedom
I2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
NMSE	Normalized Mean Square Error
PID	Proportional – Integral – Derivative
PLA	Polylactic Acid
PPR	Pulses per Revolution
PWM	Pulse Width Modulation
RPM	Rotations Per Minute
SPI	Serial Peripheral Interface
TF	Transfer Function
UART	Universal Asynchronous Receiver/Transmitter
ZOH	Zero-Order Hold

Project for Control Engineering II					
Student	Șerban Iustinian-Bogdan	Group	30135	Grade	

1. Project Overview

This project presents the design and implementation of a self-balancing cube (Cubli) that uses reaction wheels and a gyroscope to maintain balance on one of its edges or corners. The system is controlled using PID controllers that process sensor data and apply corrective torques via reaction wheels.

The goal is to demonstrate how dynamic actuation and feedback control can stabilize an inherently unstable system, achieving a visually impressive and technically challenging balance.

2. Objectives

- Design and assemble a medium-scale Cubli.
- Implement a control system to balance the Cubli on its edge using feedback from sensors.
- Tune and optimize the PID controllers to ensure stability and fast recovery from perturbations.
- Demonstrate the cube's ability to balance autonomously after disturbances.

3. Methodology

The Cubli operates based on the principle of dynamic balancing using reaction wheels. Unlike passive mechanical stabilization, this system actively controls its orientation by accelerating or decelerating internal flywheels, generating the necessary reaction torques to counteract external disturbances.

3.1. Reaction Wheels

A reaction wheel is a spinning mass mounted inside the cube. When the motor changes the wheel's rotational speed, the cube experiences an equal and opposite torque (due to the conservation of angular momentum), allowing it to adjust its body orientation without external supports:

- Accelerating a reaction wheel clockwise applies a counterclockwise torque to the cube.
- Decelerating or braking the wheel applies torque in the opposite direction.

With three orthogonal reaction wheels (aligned along X, Y, and Z axes), the system can generate controlled torques in all rotational directions, enabling balance on an edge or even a corner.

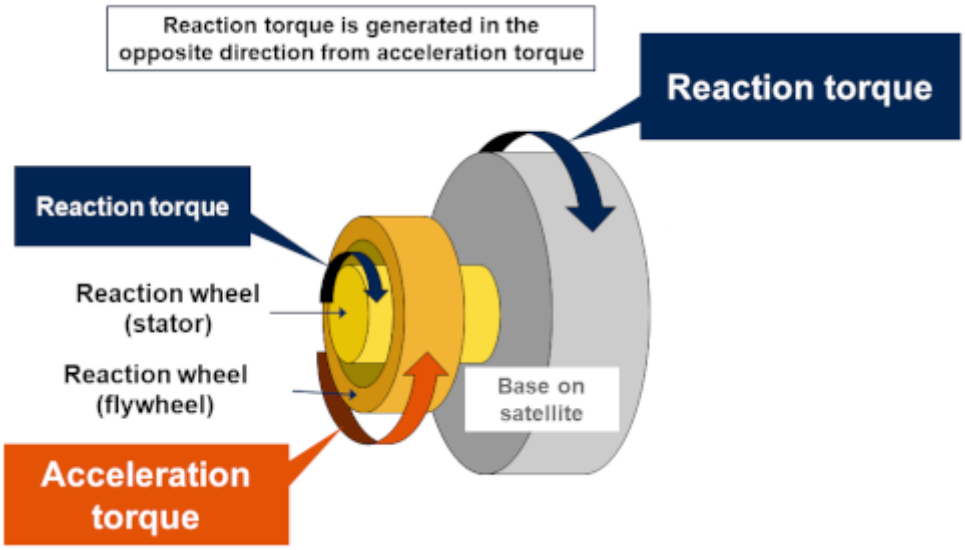

Figure 1 - Reaction Wheel

3.2. PID Controller

A PID controller is a widely used feedback control mechanism that calculates an error value between a desired setpoint and a measured process variable and then applies a corrective action to minimize that error over time.

It works by combining three control terms:

- **Proportional (P):** Corrects the error in proportion to its current size.
- **Integral (I):** Corrects accumulated past errors by integrating over time.
- **Derivative (D):** Predicts future errors by calculating the rate of change.

The controller's output is computed using the formula:

$$u(t) = Kp \cdot e(t) + Ki \cdot \int e(t)dt + Kd \cdot dtde(t)$$

Equation 1 - PID Formula

Where:

- $u(t)$ = control output (e.g., motor command)
- $e(t)$ = error signal (desired value – measured value)
- Kp, Ki, Kd = tunable proportional, integral, and derivative gains

In the Cubli project, the controller continuously adjusts the speed of the reaction wheels to apply the right torque and maintain the cube's balance, even in the presence of disturbances or noise.

3.2.1. Controller Choice

Although a full PID controller includes proportional, integral, and derivative terms, for this project a PI controller was implemented instead.

$$u(t) = Kp \cdot e(t) + Ki \cdot \int e(t)dt$$

Equation 2 - PI Formula

The derivative term (D) was found to be unnecessary because:

- The system dynamics were relatively slow, and proportional plus integral control was sufficient to achieve stable balance.
- The derivative term tended to amplify noise from the sensors (especially the IMU gyroscope), which could introduce instability rather than improve performance.
- Simplifying to a PI controller reduced computational complexity on the ESP32, leaving more resources available for real-time processing.

Thus, the PI controller effectively balances the Cubli by combining:

- Proportional action (P) to respond quickly to current errors, and
- Integral action (I) to eliminate steady-state errors over time.

This approach provided a robust and efficient control solution without the added complexity or sensitivity of the derivative term.

4. System Architecture

4.1. Electronic Components

The Cubli's electronic system integrates several carefully selected components that work together to achieve precise sensing, control, and actuation. The following subsections describe the key electronic components used in the Cubli's assembly and their specific functions within the system.

4.1.1. Microcontroller

The microcontroller used was the ESP32-DevKitC, a compact development board featuring the ESP32-WROOM-32D module, equipped with a dual-core 32-bit Xtensa® microprocessor, Wi-Fi, and Bluetooth (Classic + BLE) connectivity.

Figure 2 - ESP32-DevKitC

Key highlights:

- 38-pin variant for more I/O access

- Rich set of digital and analog peripherals (ADC, DAC, PWM, SPI, I2C, UART)
- Integrated antenna and RF balun

4.1.2. Inertial Measurement Unit (IMU)

The IMU used for this project is the DF-SEN0142, a compact IMU module based on the MPU-6050 chip, offering 6 DOF - combining a 3-axis gyroscope and a 3-axis accelerometer on a single chip.

Figure 3 - DF-SEN0142 IMU

4.1.3. Expansion board for ESP32

The project uses an expansion board, designed to simplify prototyping and development with the ESP32-DevKitC modules. This expansion board provides easy access to all GPIO pins through labeled headers and includes onboard power regulation.

Figure 4 - Expansion Board for ESP32

4.1.4. Motors

The project uses Nidec 24H-series brushless DC motors, known for their high precision, compact design, and reliable performance in applications requiring precise speed and torque control.

Key features:

- 3-Phase, 12-Pole BLDC design
- Logic-controlled clockwise or counterclockwise
- Quiet operation and low inertia
- PWM speed control and brake function

- Dual-channel phase-tracking encoder, a high-resolution encoder output (quadrature signals) allows for detailed phase tracking and fine-grained position/speed measurement.

Figure 5 - Nidec 24H Motor

4.1.5. DC/DC Step-Down Converter

The DF-DFR1015 is a compact step-down (buck) converter module designed to efficiently reduce higher input voltages to stable lower output voltages, making it ideal for supplying power to sensitive electronic components like microcontrollers and sensors. In this project, it is used to step down the 14.8V Li-Po battery voltage to 3.3V, ensuring reliable and safe power delivery to the IMU.

Figure 6 - DF-DFR1015 Buck Converter

4.1.6. Battery

The Tattu R-Line Version 3.0 LiPo battery is used in this project to provide a high-performance power source for the entire Cubli system. With a 4S1P configuration (four cells in series), it delivers 14.8V nominal voltage and supports extremely high discharge rates (120C continuous).

Figure 7 - Tattu R-Line Version 3.0 1800mAh 14.8V LiPo Battery

4.2. Mechanical

The Cubli's mechanical system was entirely modeled from scratch using Autodesk Fusion 360. All structural components were 3D printed using PLA material, allowing for rapid prototyping and iterative improvement. Each mechanical part has been engineered to interface with the electronic and control systems. The following subsections provide a detailed overview of the key mechanical components used in the Cubli's assembly.

4.2.1. 3D Printed Parts

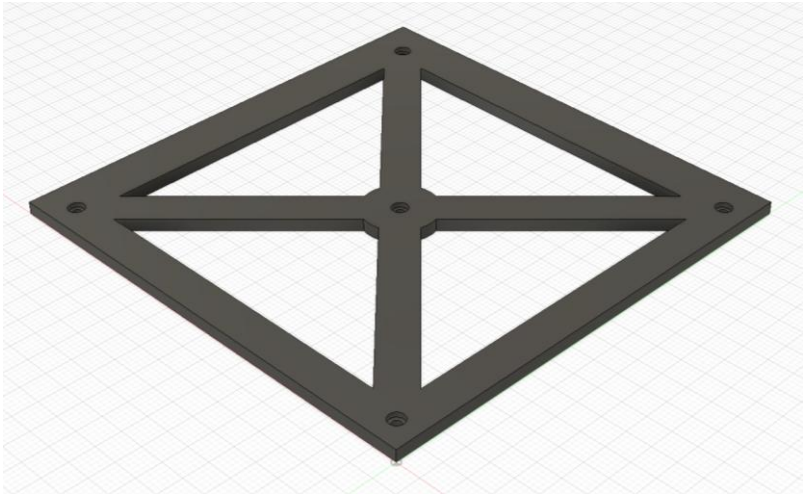

Figure 8 – Plate

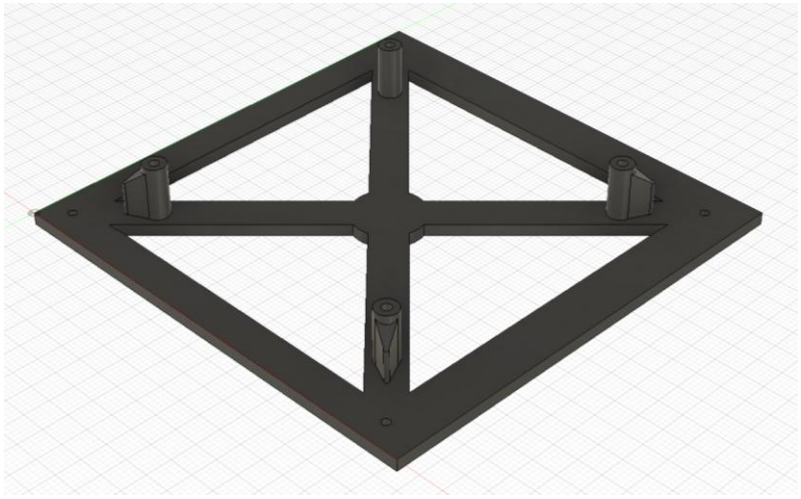

Figure 9 - Plate with Slots for Mounting Brackets

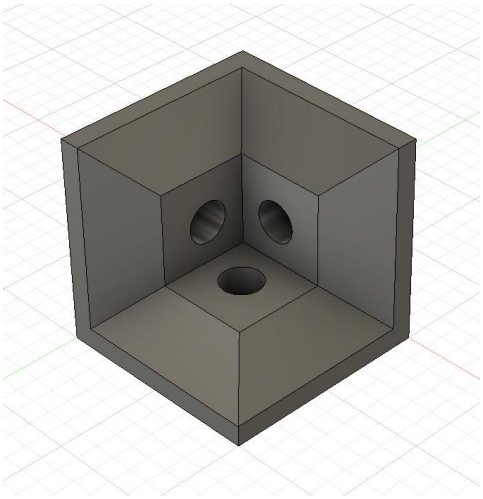

Figure 10 - Corner Piece

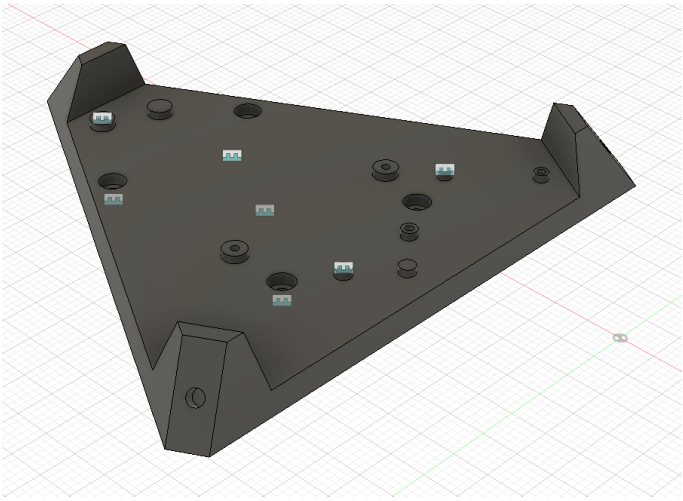

Figure 11 - Electronics Support

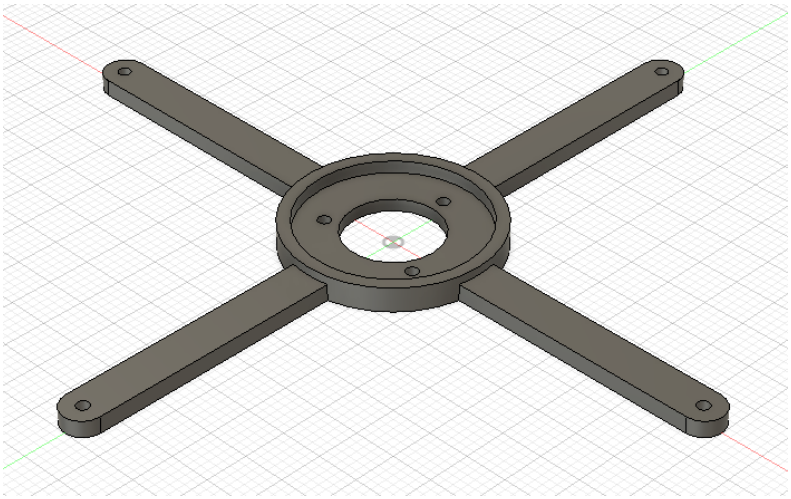

Figure 12 - Motor Mounting Bracket

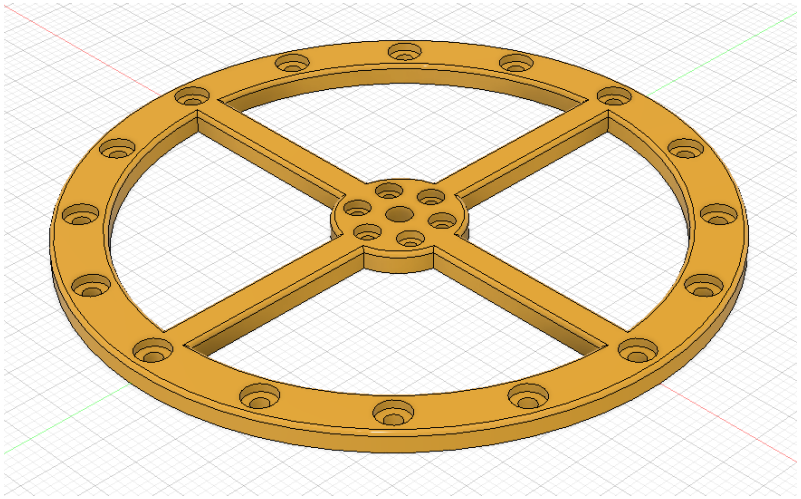

Figure 13 - Reaction Wheel

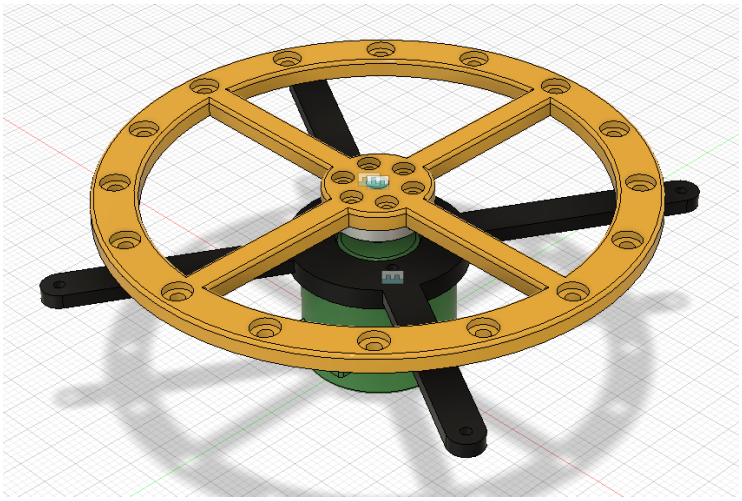

Figure 14 - Motor Assembly

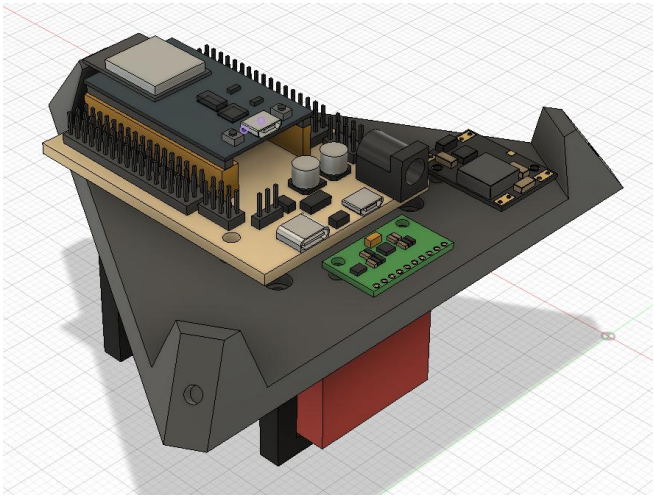

Figure 15 - Electronics Support Assembly

4.2.2. COTS Parts

Figure 16 - Aluminum Mounting Hub

Figure 18 - M3 Brass Inserts

Figure 17 - M3/M4 Screws

In the mechanical assembly of the Cubli, several commercial off-the-shelf components were used to ensure reliable performance and secure integration. Brass inserts (Figure 18) were embedded into the 3D-printed PLA parts to provide durable, wear-resistant threading for repeated assembly and disassembly. M3 and M4 screws and nuts (Figure 17) were employed throughout the build, serving both to fasten structural elements and to add to the weight of the reaction wheels. To interface the motor shafts with the custom-designed reaction wheels, an aluminium mounting hub (Figure 16) was used, ensuring a strong and precise mechanical connection between the motor and the rotating mass. All screws and critical fasteners were secured using Loctite threadlocker to prevent loosening under vibration and dynamic loads, ensuring mechanical stability during operation.

4.3. Connection Diagram

The following diagrams (Figure 19 and Figure 20) illustrate the power and signal distribution of the Cubli system. The main power is supplied by a 14.8V 4S1P LiPo battery. A DC-DC Step-Down Module (DFR1015) reduces the voltage to 3.3V, providing stable power to the DF-SEN0142 IMU. The ESP32 DevKitC acts as the central controller, receiving orientation data from the IMU and encoder feedback from the Nidec 24H motors, running the PI control algorithm, and sending control commands to the three motors. Both the motors and the ESP32 draw their primary power directly from the battery's high-voltage line, while the integrated motor encoders receive regulated 5V power from the ESP32 expansion board. All components share a common ground to ensure a consistent electrical reference and prevent signal disruptions. This tightly integrated system enables real-time sensing, precise control, and rapid actuation, allowing the Cubli to maintain stable dynamic balance.

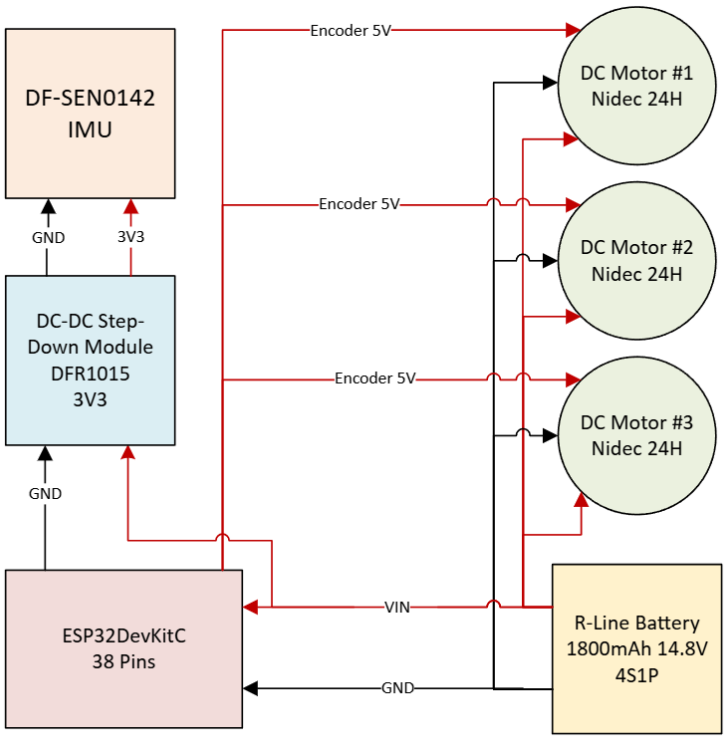

Figure 19 - Cubli's Power Diagram

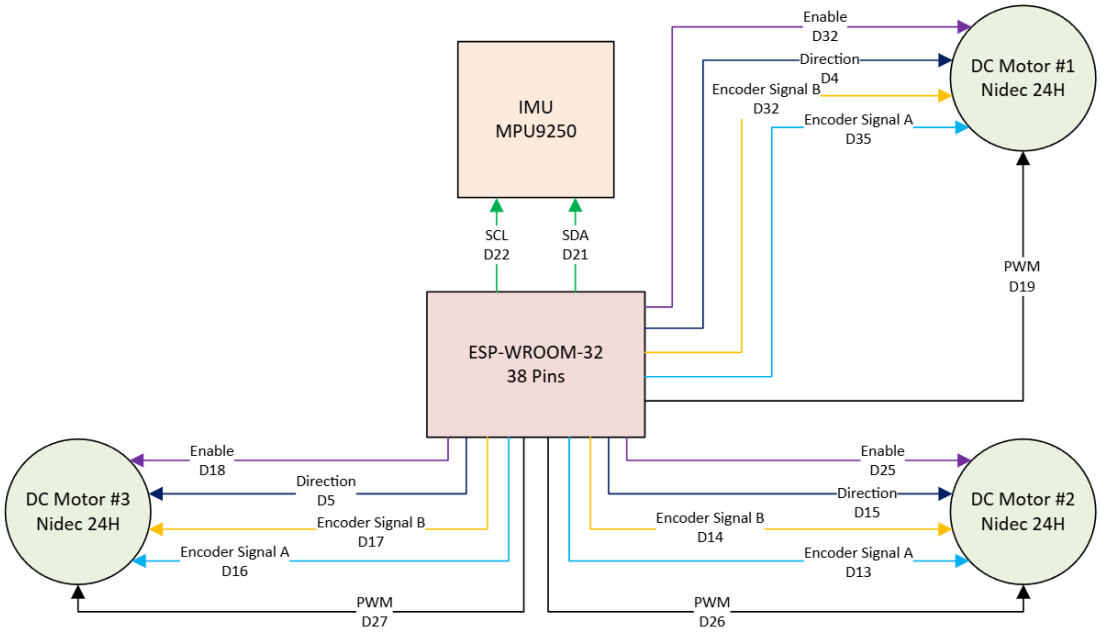

Figure 20 - Cubli's Data Diagram

4.4. Main Assembly

Figure 21 and Figure 22 present the complete 3D CAD model of the Cubli, designed and assembled entirely in Autodesk Fusion 360. The frame is a robust, lightweight cubic structure, designed to house all critical subsystems. Inside the cube, three orthogonally

mounted reaction wheel assemblies—driven by Nidec 24H motors—enable precise torque generation along the X, Y, and Z axes. The ESP32 controller board and connected electronics are securely mounted on an internal platform, ensuring compact integration and minimal wiring complexity. The design prioritizes both mechanical rigidity and easy access for maintenance and adjustments, providing an optimized layout for balancing operations and control experiments.

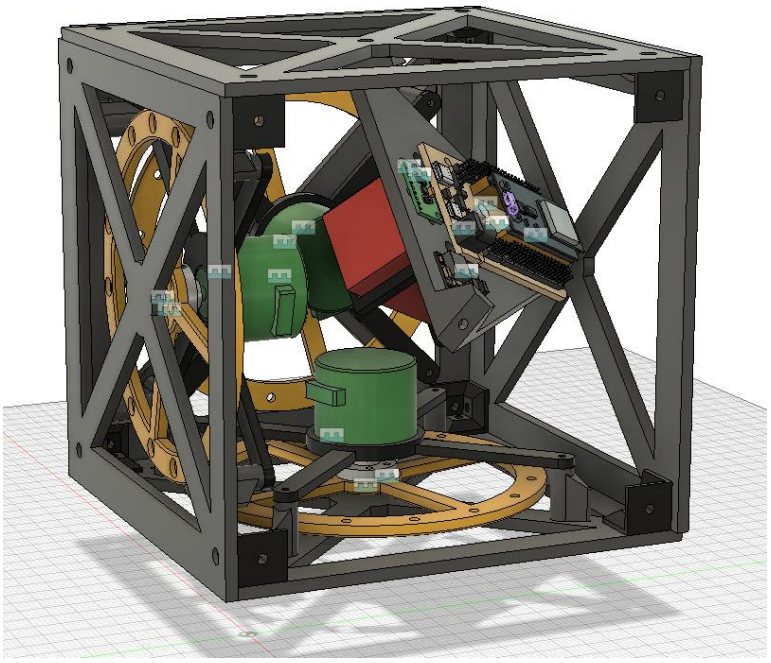

Figure 21 - Cubli's Main Assembly View 1

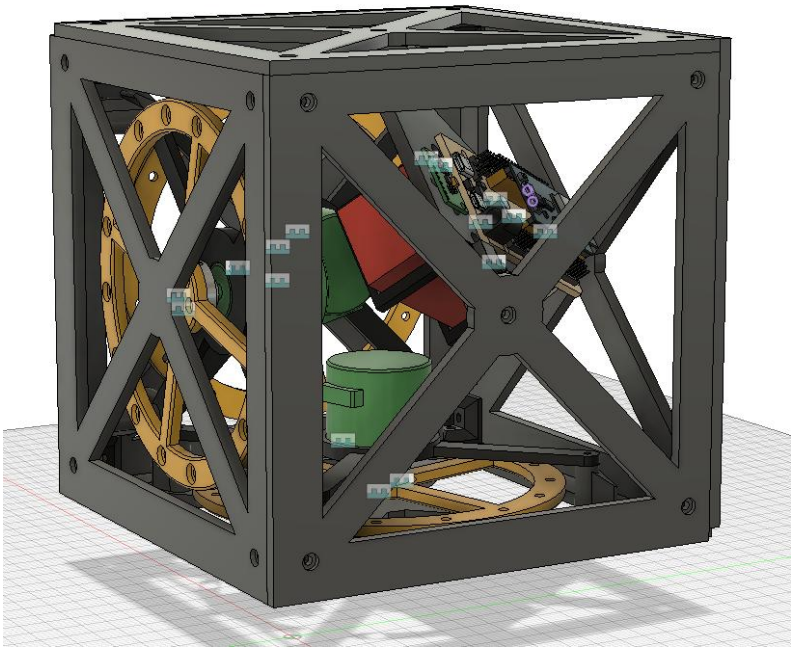

Figure 22 - Cubli's Main Assembly View 2

5. Control System Design

5.1. Motor Data Acquisition

In this section, a MATLAB script was implemented to acquire real-time motor performance data over a serial connection. The system connects to the microcontroller through a specified COM port and baud rate. The script creates a 'serialport' object to handle incoming data and listens for a set duration of 25 seconds.

Within the main loop, the script continuously checks for available bytes in the serial buffer. When a line is received, it expects a comma-separated string with three values: input command (control signal sent to the motor), elapsed time (in milliseconds), and measured motor speed (in RPM). These values are parsed, converted to appropriate units, and stored in MATLAB arrays.

After the logging duration ends, the serial connection is properly cleared to free system resources. The collected data is then organized into a MATLAB table with three columns—Input, Time_s, and Speed_RPM—and exported as a CSV file, which serves as a structured log of the motor's response over time and can be further analyzed or visualized in MATLAB or other data processing tools.

```
comPort = "COM8";
baudRate = 115200;
duration = 25;

s = serialport(comPort, baudRate);
flush(s);

input = [];
time = [];
rpm = [];
startTime = datetime('now');

disp("Logging...");

while seconds(datetime('now') - startTime) < duration
    if s.NumBytesAvailable > 0
        line = readline(s);
        data = sscanf(line, '%f,%f,%f');
        if numel(data) == 3
            input(end+1) = data(1);
            time(end+1) = data(2) / 1000; % ms to seconds
            rpm(end+1) = data(3);
        end
    end
end

clear s
disp("Logging complete.");

T = table(input', time', rpm', 'VariableNames', {'Input', 'Time_s', 'Speed_RPM'});
writetable(T, 'motor_data.csv');
```

Table 1 - Motor Data Acquisition MATLAB Code

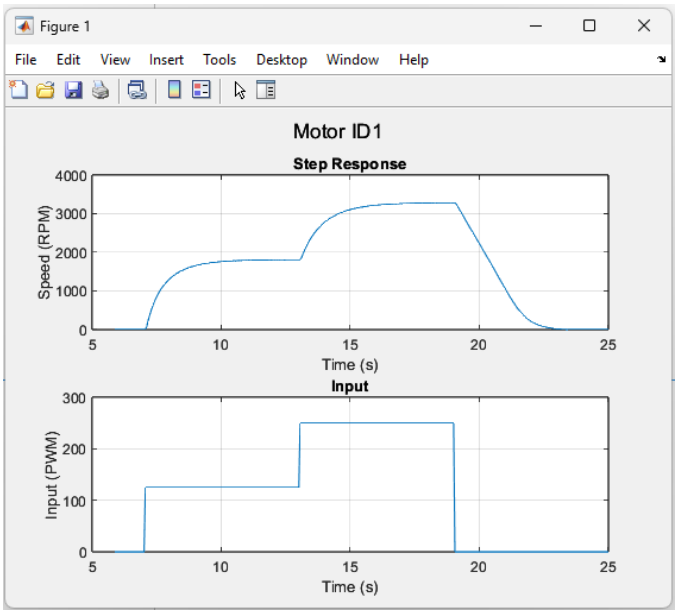

Figure 23 - MotorID1 Data

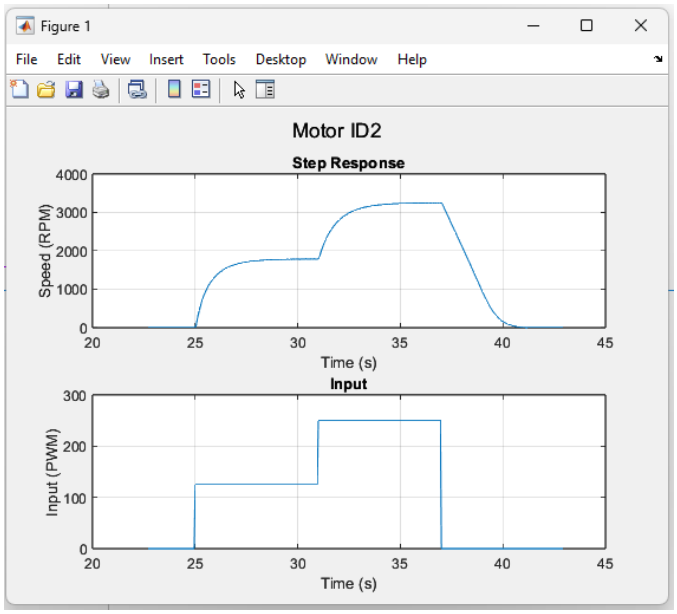

Figure 24 - MotorID2 Data

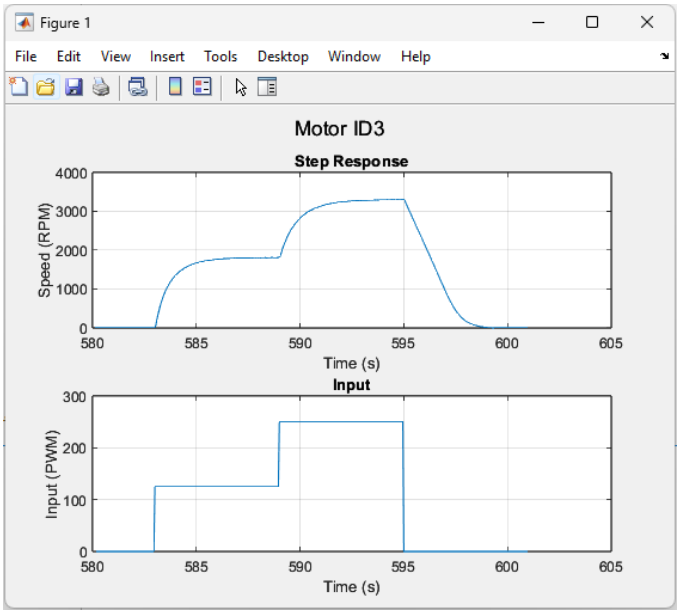

Figure 25 - MotorID3 Data

5.2. Motor Transfer Function Estimation

```
clc; close all; clear all;

dataTable_motorID1 = readtable("motor_data_ID1.csv");
dataTable_motorID2 = readtable("motor_data_ID2.csv");
dataTable_motorID3 = readtable("motor_data_ID3.csv");

inputArray_motorID1 = dataTable_motorID1.Input(5:260);
valueArray_motorID1 = dataTable_motorID1.Speed_RPM(5:260);

inputArray_motorID2 = dataTable_motorID2.Input(27:282);
valueArray_motorID2 = dataTable_motorID2.Speed_RPM(27:282);

inputArray_motorID3 = dataTable_motorID3.Input(38:293);
valueArray_motorID3 = dataTable_motorID3.Speed_RPM(38:293);

inputArray_motorID1_id = inputArray_motorID1(145:end);
valueArray_motorID1_id = valueArray_motorID1(145:end);

inputArray_motorID2_id = inputArray_motorID2(141:end);
valueArray_motorID2_id = valueArray_motorID2(141:end);

inputArray_motorID3_id = inputArray_motorID3(141:end);
valueArray_motorID3_id = valueArray_motorID3(141:end);

iddata_motorID1_id = iddata(valueArray_motorID1_id, inputArray_motorID1_id, 0.05);
iddata_motorID2_id = iddata(valueArray_motorID2_id, inputArray_motorID2_id, 0.05);
iddata_motorID3_id = iddata(valueArray_motorID3_id, inputArray_motorID3_id, 0.05);

TF_motorID1 = calculateTF(inputArray_motorID1_id, valueArray_motorID1_id, 2100, 1);
TF_motorID2 = calculateTF(inputArray_motorID2_id, valueArray_motorID2_id, 1782, 2);
TF_motorID3 = calculateTF(inputArray_motorID3_id, valueArray_motorID3_id, 1806, 3);

function TF = calculateTF(input_data, output_data, initial_cond, motorID)
    iddata_motor = iddata(output_data, input_data, 0.05);
    arx_model = arx(iddata_motor, [1 1 1]);
    [num, den] = tfdata(arx_model, 'v');
    [A, B, C, D] = tf2ss(num, den);
```

```
    y_calc = dlsim(A', C', B', D', input_data, initial_cond);
    TF = minreal(tf(arx_model));

    figure;
    plot(1:length(y_calc), [y_calc, output_data]);
    grid on;
    legend("Y\_calc", "Y\_real");
    title("MotorID" + motorID);
    xlabel("Sample Time");
    ylabel("RPM");
    nmse = norm(output_data - y_calc) / norm(output_data - mean(output_data)) * 100;
end
```

Table 2 - Motor Transfer Function Estimation MATLAB Code

In this section, a MATLAB code was created to estimate the transfer functions of the three motors using recorded experimental data. The script begins by reading CSV files that contain the input signals and measured RPM outputs for each motor.

To focus on the most relevant data segments, specific index ranges are selected for each motor, trimming out initial transients or noise. These refined datasets are then converted into MATLAB's 'iddata' format, which is used for system identification tasks.

The core of the analysis is handled by the custom calculateTF function. For each motor, this function:

1. Constructs an ARX model using the input and output data.
2. Converts the resulting discrete transfer function into state-space form.
3. Simulates the system response using 'dlsim' and compares the simulated (Y_calc) vs. actual (Y_real) motor speeds.
4. Plots the comparison for visual inspection, showing how well the identified model fits the real system behavior.

Finally, the transfer function is simplified using 'minreal' to remove insignificant states, making the model easier to interpret and use in further control system design or simulation.

To quantify how well the simulated model output matches the real measured system output, the Normalized Mean Square Error (NMSE) metric was used. The NMSE compares the squared error between the actual output y and the model's simulated output $\hat{y}$, normalized by the variance of the actual output.

The NMSE is calculated using the formula:

$$NMSE = \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y}_i)^2}$$

Equation 3 - NMSE Formula

where:

- y_i = actual measured output at sample i

- $\hat{y}_i$ = simulated model output at sample i
- $\bar{y}$ = mean value of the measured output y
- N = total number of samples

This chapter provides an essential step in characterizing the dynamic behavior of each motor, which is crucial for tasks like controller design, performance prediction, and closed-loop system optimization.

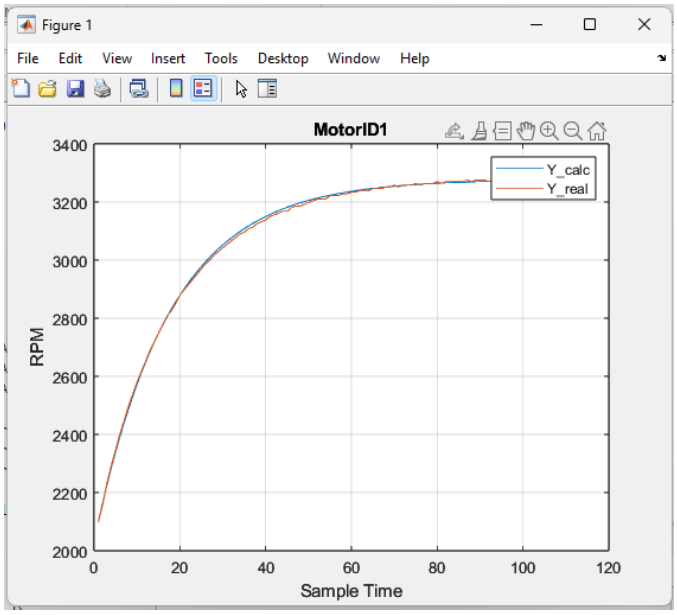

Figure 26 - Simulated vs. Measured RPM for MotorID1

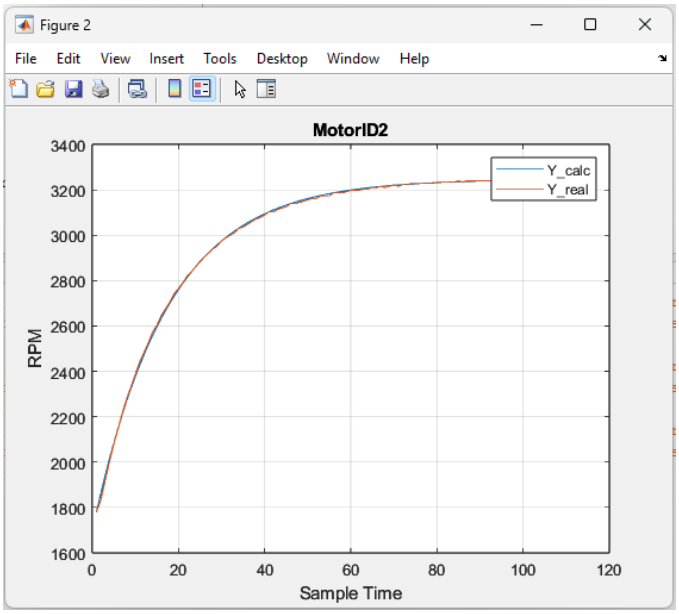

Figure 27 - Simulated vs. Measured RPM for MotorID2

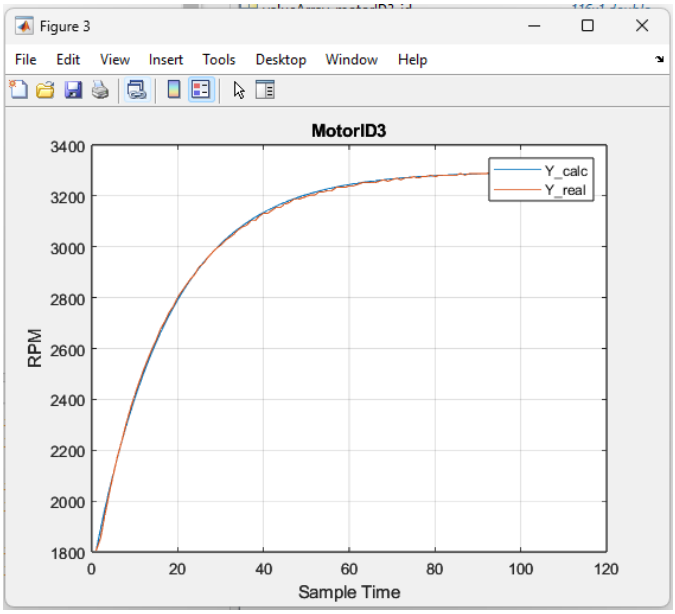

Figure 28 - Simulated vs. Measured RPM for MotorID3

The transfer functions $H_f(s)$ for all three motors show very similar system dynamics, with nearly identical coefficients, indicating consistent motor behavior across units. The discrete transfer functions $H_f(z^{-1})$ were calculated using the zero-order hold (ZOH) method with a sampling time T_e= 0.05. The NMSE values (2.30%, 1.84%, 2.05%) reflect low normalized errors, confirming that the identified models accurately capture each motor's response with good fit quality.

MotorID	ID1	ID2	ID3
TF $H_f(s)$	$\frac{14.96}{s+1.141}$	$\frac{15.08}{s+1.161}$	$\frac{15.03}{s+1.141}$
Discrete TF $H_f(z^{-1})$	$\frac{0.7271 \cdot z^{-1}}{1-0.9445 \cdot z^{-1}}$	$\frac{0.7325 \cdot z^{-1}}{1-0.9436 \cdot z^{-1}}$	$\frac{0.7304 \cdot z^{-1}}{1-0.9446 \cdot z^{-1}}$
NMSE	2.3033	1.8369	2.0452

Table 3 - TF and NMSE of Motors

5.3. PI Tuning

Using MATLAB's 'pidtune' function, a PI controller is automatically tuned for the discretized TF of each motor to achieve a desired closed-loop bandwidth (or the natural frequency ω_n) of 13.55 rad/s. The resulting continuous controller is then converted into discrete form, yielding $H_c(z^{-1})$, which is simplified using 'minreal' to remove unnecessary dynamics.

Finally, the script simulates the closed-loop step response of the combined system $H_c(z^{-1}) \times H_f(z^{-1})$ with unity feedback over 1 second, providing a visual evaluation of the controller's performance.

The formula for the closed-loop system H_o is:

$$H_o = \frac{H_c \times H_f}{1 + H_c \times H_f}$$

Equation 4 - Closed-Loop System Formula

```
Hf_discret = c2d(Hf, 0.05, 'zoh');
Hf_discret.Variable = "z^-1";

% Use pidtune to get PI controller
[c_tuned, info] = pidtune(Hf_discret, 'PI', 13.33) % 13.33 rad/s desired bandwidth
% Discretize controller
Hc_discret = minreal(tf(c_tuned));
Hc_discret.Variable = 'z^-1'
figure;
step(feedback(series(Hf_discret, Hc_discret), 1), 1);
title("Step Response for Motor ID1");
```

Table 4 - PID Tuning MATLAB Code

For this project, the following performances for the closed-loop system were imposed:

- Settling time of $T_s \leq 0.5s$
- Overshoot $\sigma \leq 10\%$

The required damping ratio corresponding to the specified overshoot is calculated using the below formula, which implies a damping ration of $\zeta \approx 0.6$.

$$\sigma = e^{\frac{-\pi\zeta}{\sqrt{1-\zeta^2}}}$$

Equation 5 - Overshoot Formula

For a second-order system, the settling time can be approximated using the formula:

$$T_s \approx \frac{4}{\zeta\omega_n}$$

Equation 6 - Setting Time Formula for a Second-Order System

Given the imposed performances, the resulting natural frequency can be calculated as:

$$\omega_n = \frac{4}{\zeta T_s} \geq \frac{4}{0.6 \times 0.5} = 13.33\, rad/s$$

Thus, the target natural frequency for the system is approximately 13.33 rad/s.

The resulting controllers for each motor can be visualized in the following table:

MotorID	**ID1**	**ID2**	**ID3**
Discrete Controller $H_c(z^{-1})$	$\frac{0.9272 - 0.7636 \cdot z^{-1}}{1 - z^{-1}}$	$\frac{0.9201 - 0.7569 \cdot z^{-1}}{1 - z^{-1}}$	$\frac{0.9229 - 0.7601 \cdot z^{-1}}{1 - z^{-1}}$

H_o Settling Time T_s [s]	0.651	0.648	0.651
H_o Overshoot σ [%]	13.1	13.1	13.1

Table 5 - Controllers Transfer Function and H_0 Settling Time and Overshoot

Although the continuous-time design may meet the imposed settling time and overshoot specifications, the performances in the discrete implementations show deviations — notably, slightly longer settling times and higher overshoot. This discrepancy is often caused by the process of converting continuous-time controllers to discrete form, which introduces approximation errors that subtly change the system's dynamic behavior.

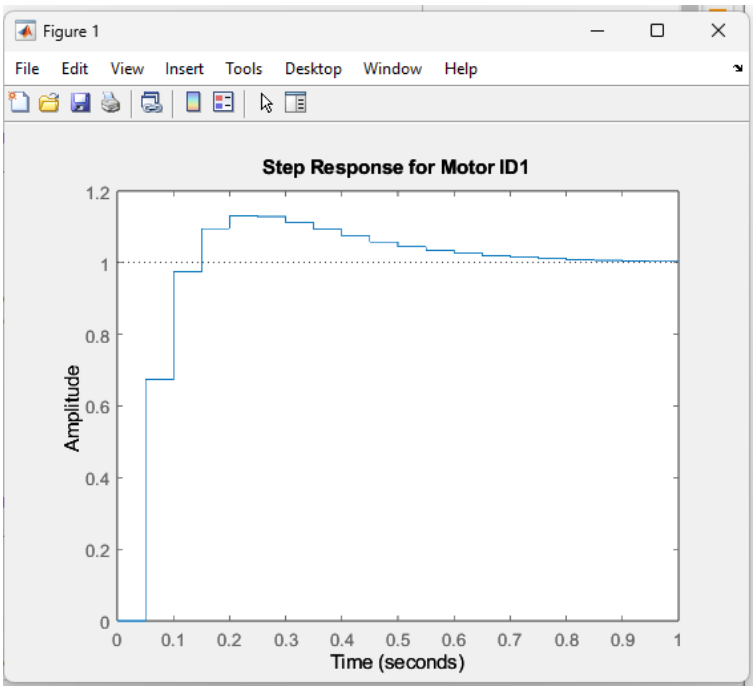

Figure 29 - Step Response of H_0 MotorID1

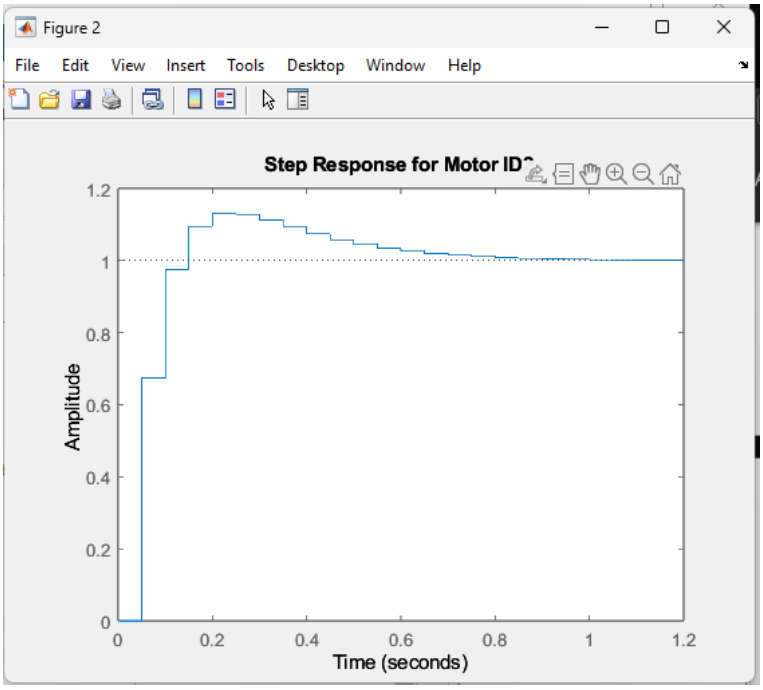

Figure 30 - Step Response of H_0 MotorID2

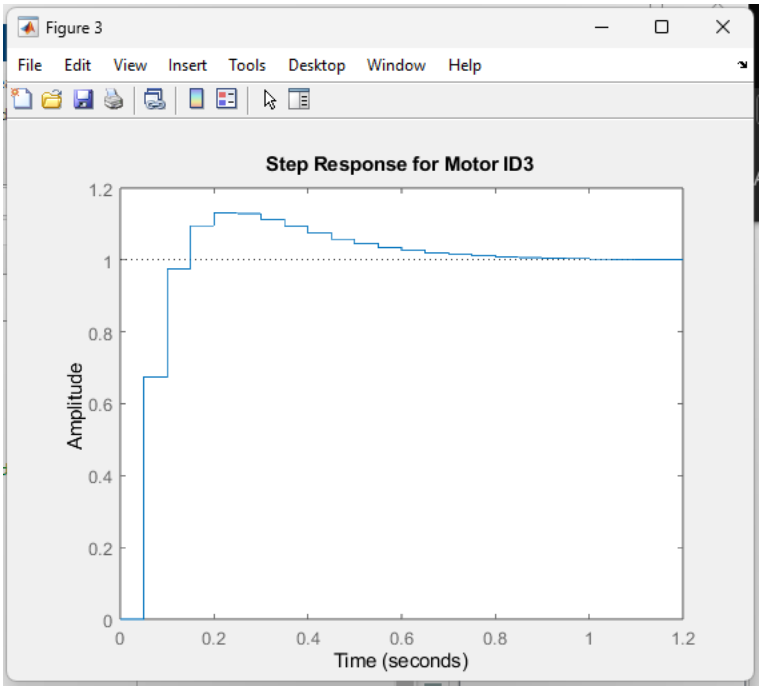

Figure 31 - Step Response of H_0 MotorID3

5.4. Arduino Controller Implementation

```
volatile double M1_command[2];
volatile double M1_error[2];
const double M1_PID_coef[3] = {1, 0.9272, -0.7636};

void M1_PID(int target_rpm, int current_rpm){
```

```
  M1_error[1] = M1_error[0];
  M1_error[0] = abs(target_rpm) - abs(current_rpm);

  M1_command[1] = M1_command[0];
  M1_command[0] = M1_PID_coef[0] * M1_command[1] + M1_PID_coef[1] * M1_error[0] +
M1_PID_coef[2] * M1_error[1];

  Serial.print("Command: ");
  Serial.println(M1_command[0]);

  if(M1_command[0] > 255) M1_command[0] = 255;
  else if(M1_command[0] < -255) M1_command[0] = -255;

  if(target_rpm < 0)
    M1_Control(-M1_command[0]);
  else
    M1_Control(M1_command[0]);
}

void M1_Control(int sp) {
  if (sp > 0) digitalWrite(M1_DIR, LOW);
    else digitalWrite(M1_DIR, HIGH);
  ledcWrite(M1_PWM, abs(sp));
}
```

Table 6 - Controller Implementation in Arduino

This Arduino code implements a discrete PI controller for motor M1, where it calculates the control command based on the difference between the target RPM and the measured current RPM. It updates the current and previous error values, applies a recursive control equation using predefined PID coefficients, and ensures the resulting command stays within the allowed PWM range of −255 to 255. Depending on the sign of the target RPM, it adjusts the control signal's direction and magnitude, calling the 'M1_Control' function to set the motor's direction pin (M1_DIR) and apply the PWM output (M1_PWM). Overall, this setup is implemented for each motor.

The current motor's RPM is calculated based on the motor encoder counts. The encoder generates PPR (pulses per revolution), and the system samples the counts every 50 milliseconds. The time step 'dt' is calculated in seconds. The formula for 'M1_rpm' converts the count rate into RPM, where the factor of 2 accounts for quadrature encoding (since each encoder pulse gives two counts).

The 'M1_UpdateEncoder()' function updates 'M1_enc_count' by reading encoder signals 'M1_ENC_SIG_A' and 'M1_ENC_SIG_B'; if both signals match, it increments the count, and if they differ, it decrements the count, effectively tracking both position and direction of the motor shaft.

```
int PPR = 100; // Pulses per revolution
unsigned long sampleInterval = 50; // ms
float dt = sampleInterval / 1000.0;
float M1_rpm = (M1_enc_count / dt) * (60.0 / (PPR * 2));

void M1_UpdateEncoder() {
  int A = digitalRead(M1_ENC_SIG_A);
  int B = digitalRead(M1_ENC_SIG_B);

  if (A == B) {
    M1_enc_count++;
  } else {
    M1_enc_count--;
  }
}
```

Table 7 - Motor RPM Calculation

6. Testing

The provided code is designed as part of the testing setup to validate the performance of the PI controller for the three independent motors. During testing, Bluetooth commands are received via 'SerialBT', allowing real-time control instructions to be sent to each motor. The command format *<motor_number> <value>* is used to set a target RPM or to issue start/stop commands for each motor individually.

At defined intervals, the code reads the encoder pulse counts for each motor, computes their actual RPM, and applies the PI control algorithm if a target RPM is active. This closed-loop feedback ensures that the motors dynamically adjust their speed to match the desired setpoint.

To facilitate testing and tuning, the system outputs the measured RPMs and current control signal values over the serial interface. This feedback enables observation of how the PI controller responds to setpoint changes, external disturbances, or start/stop commands, ensuring stable and accurate motor speed control prior to integration into the full Cubli system.

```
void loop() {
   if (SerialBT.available()) {
    String incoming = SerialBT.readStringUntil('\n');  // Read until newline
    incoming.trim();  // Remove any extra \r or spaces

    Serial.print("Received: ");
    Serial.println(incoming);

    int spaceIndex = incoming.indexOf(' ');
    if (spaceIndex != -1) {
```

```
    String firstPart = incoming.substring(0, spaceIndex);
    String secondPart = incoming.substring(spaceIndex + 1);

    int motor = firstPart.toInt();
    String valueStr = secondPart;

    // Check if the second part is a number
    bool isNumber = true;
    for (size_t i = 0; i < valueStr.length(); i++) {
      char c = valueStr.charAt(i);
      if (!isDigit(c) && !(i == 0 && c == '-')) {
        isNumber = false;
        break;
      }
    }

    switch(motor){
      case 1:{
        if(isNumber){
          int RPM = valueStr.toInt();
          M1_target_RPM = RPM;
        }
        else if(valueStr == "stop"){
          M1_target_RPM = 0;
          M1_Control(0);
          digitalWrite(M1_EN, HIGH);
        }
        else if(valueStr == "start"){
          digitalWrite(M1_EN, LOW);
        }
        break;
      }
      case 2:{
        if(isNumber){
          int RPM = valueStr.toInt();
          M2_target_RPM = RPM;
        }
        else if(valueStr == "stop"){
          M2_target_RPM = 0;
          M2_Control(0);
          digitalWrite(M2_EN, HIGH);
        }
        else if(valueStr == "start"){
          digitalWrite(M2_EN, LOW);
        }
        break;
      }
```

```
      case 3:{
        if(isNumber){
          int RPM = valueStr.toInt();
          M3_target_RPM = RPM;
        }
        else if(valueStr == "stop"){
          M3_target_RPM = 0;
          M3_Control(0);
          digitalWrite(M3_EN, HIGH);
        }
        else if(valueStr == "start"){
          digitalWrite(M3_EN, LOW);
        }
        break;
      }
    }
  } else {
    Serial.println("Invalid format, expected: <command> <value>");
  }
}
unsigned long currentTime = millis();

if (currentTime - lastTime >= sampleInterval) {
  noInterrupts();
  long M1_count = M1_enc_count;
  long M2_count = M2_enc_count;
  long M3_count = M3_enc_count;

  M1_enc_count = 0;
  M2_enc_count = 0;
  M3_enc_count = 0;
  interrupts();

  float dt = sampleInterval / 1000.0;
  float M1_rpm = (M1_count / dt) * (60.0 / (PPR * 2));
  float M2_rpm = (M2_count / dt) * (60.0 / (PPR * 2));
  float M3_rpm = (M3_count / dt) * (60.0 / (PPR * 2));

  Serial.print("M1_rpm:");
  Serial.print(M1_rpm);
  Serial.print(", ");
  Serial.print("M2_rpm:");
  Serial.print(M2_rpm);
  Serial.print(", ");
  Serial.print("M3_rpm:");
  Serial.print(M3_rpm);
  Serial.print(", ");
```

```
    Serial.print("M1_com:");
    Serial.print(M1_com);
    Serial.print(", ");
    Serial.print("M2_com:");
    Serial.print(M2_com);
    Serial.print(", ");
    Serial.print("M3_com:");
    Serial.println(M3_com);

    if(M1_target_RPM != 0) M1_PID(M1_target_RPM, M1_rpm);
    if(M2_target_RPM != 0) M2_PID(M2_target_RPM, M2_rpm);
    if(M3_target_RPM != 0) M3_PID(M3_target_RPM, M3_rpm);
    lastTime = currentTime;
  }
}
```

Table 8 - Testing Arduino Code

As a result of the testing process, the motors were confirmed to be functioning perfectly, maintaining stable and accurate speed control across all test conditions, and meeting the required performance for integration into the full Cubli system.

Figure 32 - Arduino Serial Plotter

7. Next Steps

The next steps will focus on testing and integrating the IMU into the control system to enable full closed-loop balancing control. While the current setup has validated the performance of the PI controller and motor actuation, the integration of real-time orientation data from the IMU is essential for allowing the Cubli to sense its tilt and apply

corrective actions dynamically. This phase will involve calibrating the IMU, validating sensor readings, and incorporating its data into the control loop to enable the Cubli to balance stably on an edge or corner.

For future improvements, the project aims to make the Cubli more compact and fabricate its structure from aluminium to improve mechanical strength and reduce weight. Additionally, a more robust control strategy will be explored, potentially moving beyond the current PI controller to advanced methods such as adaptive control or model predictive control, providing better disturbance rejection and faster response under complex dynamic conditions. Another improvement is to develop the capability for the Cubli to self-position onto its corner from a flat face, adding an advanced level of autonomy and showcasing its ability to perform dynamic maneuvers beyond static balancing.

8. Conclusions

This project successfully demonstrated the development of a functional Cubli prototype, capable of precision motor control using a PI feedback system. Through the integration of brushless DC motors, encoders, a microcontroller, and Bluetooth communication, the system achieved stable and accurate speed regulation, providing a solid base for future balance control. The mechanical structure, fully designed from scratch and made using 3D-printed PLA components, provided a lightweight and modular platform for testing and iteration.

While significant progress was made, particularly in validating the motor control subsystem, the integration of the IMU remains a critical next step to enable full dynamic balancing. Looking ahead, the planned improvements should make the Cubli more powerful, robust, and independent.

Bibliography

- ETH Zürich, Cubli: A Cube That Can Jump Up, Balance, and Walk Across Your Desk. [Online Video]. Available: https://www.youtube.com/watch?v=n_6p-1J551Y
- Åström, K. J., & Hägglund, T., PID Controllers: Theory, Design, and Tuning. Instrument Society of America, 1995.
- MathWorks, *pidtune (PID Tuning) - MATLAB Documentation*. [Online]. Available: https://www.mathworks.com/help/control/ref/pidtune.html
- Texas Instruments, Brushless DC Motor Fundamentals. [White Paper]. Available: https://www.ti.com/lit/an/slyp173/slyp173.pdf
- DFRobot, DF-SEN0142 MPU6050 6DOF IMU Datasheet. [Online]. Available: https://wiki.dfrobot.com
- DFRobot, *DFR1015 Buck Converter Module*. [Product Page]. Available: https://wiki.dfrobot.com/DFR1015
- Espressif Systems, ESP32 Technical Reference Manual. [Online PDF]. Available: https://www.espressif.com/en/products/socs/esp32/resources
- Nidec Corporation, Nidec 24H Series Brushless DC Motor Specifications. [Datasheet]. Available: https://www.nidec.com