Universität Siegen
Naturwissenschaftlich-Technische Fakultät
Department Elektrotechnik und Informatik
Lehrstuhl für Digital Integrated Systems

Masterarbeit

Evaluation model of a parametrizable ReRAM core in a FPGA architecture

by
Hrishikesh Balkrishna Karande
Matriculation number
1701995

Examiner
Prof. Dr.-Ing. Michael Wahl
2nd Examiner
Prof. Dr.-Ing. Bing Li

Date of delivery
30.10.2025

This report details the design and implementation of a two-wheeled self-balancing robot, a system that embodies the classic inverted pendulum challenge. The robot employs an MPU6050 inertial measurement unit (IMU) for precise state estimation, with sensor data fusion accomplished using a Kalman filter. A Proportional-Integral-Derivative (PID) control algorithm is implemented to ensure robust balance and stability. The system is designed to support remote monitoring and control, broadening its applicability across diverse scenarios. The firmware for the Arduino Nano was developed using PlatformIO, and the full project, including source code and comprehensive documentation, is accessible in this GitHub repository: `https://github.com/haris-mujeeb/Self-Balancing-Robot`.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Background & Motivation

Two-wheeled vehicles are generally more agile, allowing easier navigation through tight spaces, making them ideal for congested environments. Their lighter weight and compact size facilitate easier handling while also enhancing energy efficiency. In addition, they are typically less expensive to purchase and maintain, increasing accessibility for a wider range of users. A good base model to build such robot is ELEGOO Tumbler (shown in Fig. 1), which provided nearly all the hardware required as a DIY kit.

Figure 1: ELEGOO Tumbler which was used for this project **tumblller**.

1.2. Project Objectives

The primary objectives of this project are as follows:

- To design and implement a two-wheeled self-balancing robot capable of maintaining stability using sensor feedback and control algorithms.
- To develop a cascaded PID control system for real-time balance and motion control, ensuring robustness and responsiveness.
- To integrate an MPU6050 inertial measurement unit (IMU) for accurate state estimation, utilizing sensor fusion techniques such as a Kalman filter.
- To enable remote monitoring and control of the robot, enhancing its versatility for various applications.
- To document the design, implementation, and tuning processes, providing a comprehensive resource for future development and replication.

2. Literature Review

Two-wheeled self-balancing robots have been widely studied as a variant of the classic inverted pendulum problem, requiring advanced control strategies to maintain stability. Various approaches, including

Proportional-Integral-Derivative (PID) control **matlab_inverted_pendulum**, Linear Quadratic Regulator (LQR) **10193276** have been explored to achieve robust balancing and motion control.

Xu and Duan **1174486** demonstrated that the Linear Quadratic Regulator (LQR) controller outperforms the pole-placement controller in both simulation and real-time control scenarios.

Kalman filtering has been extensively used for sensor fusion in such systems, particularly for integrating accelerometer and gyroscope data to obtain accurate state estimates. Studies have demonstrated that complementary and extended Kalman filters significantly improve stability and noise rejection in sensor-driven control systems.

Recent advancements in autonomous navigation for self-balancing robots have incorporated Simultaneous Localization and Mapping (SLAM) techniques, LiDAR-based obstacle detection, and machine learning-based adaptive control methods **Tsai_Chih_2008 Ranasinghe_Vidanapathirana_2019**. These improvements enable real-time path planning and environmental interaction, making self-balancing robots more suitable for real-world applications such as personal mobility, surveillance, and industrial automation.

This project builds upon these existing methodologies by implementing a Kalman filter for sensor fusion and employing PID-based control to achieve stable balancing and maneuverability. Future work aims to integrate more advanced control and navigation techniques for enhanced autonomy and performance.

2.1. Scope of Work

The scope of this project encompasses the following key areas:

- **Hardware Integration:** Assembling and configuring the ELEGOO Tumbler platform, including the MPU6050 IMU, motor drivers, and encoders.
- **Software Development:** Implementing firmware for the Arduino Nano microcontroller using PlatformIO, focusing on sensor data acquisition, control algorithms, and communication protocols.
- **Control System Design:** Designing and tuning a cascaded PID control loop for pitch, yaw, and position control, ensuring stable and precise operation.
- **Sensor Fusion:** Utilizing a Kalman filter to fuse data from the IMU, improving the accuracy of state estimation and control.
- **Testing and Validation:** Conducting extensive testing to evaluate the robot's performance under various conditions, followed by iterative refinement of the control parameters.
- **Documentation and Open-Source Contribution:** Providing detailed documentation, including schematics, code, and tuning guidelines, and making the project available on a GitHub repository for open-source collaboration **opensource**.

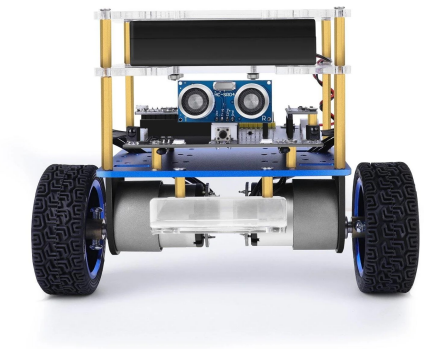


Figure 2: ELEGOO Tumbler **tumblller**.

2.2. Mathematical Modelling

The physical problem of the balancing robot is well described by the widely analyzed inverted pendulum model. This system is typically represented as a rigid rod attached to a joint, mounted on a rigid cart that moves in a single direction.

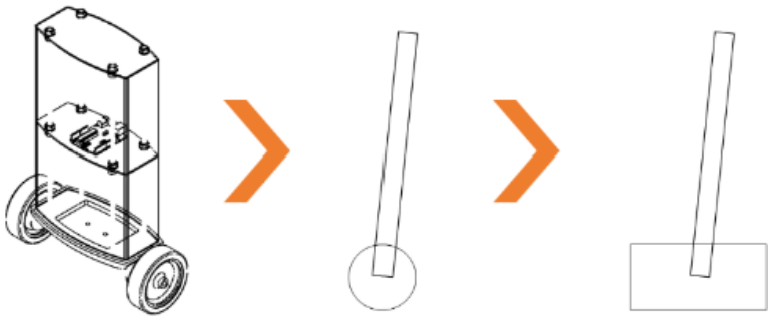


Figure 3: Simplified inverted pendulum model for the balancing robot **10193276**.

For simplification, the wheelbase of the robot is assumed to behave like a cart sliding on a frictionless surface. This modeling approach follows Prasad and Tyagi **prasad_optimal_2014**, and the MathWorks tutorial on the inverted pendulum **matlab_inverted_pendulum**.

To ease mathematical formulation, the pendulum's motion is restricted to one degree of freedom, with the angle θ evolving in the xy-plane, see Figure 3.

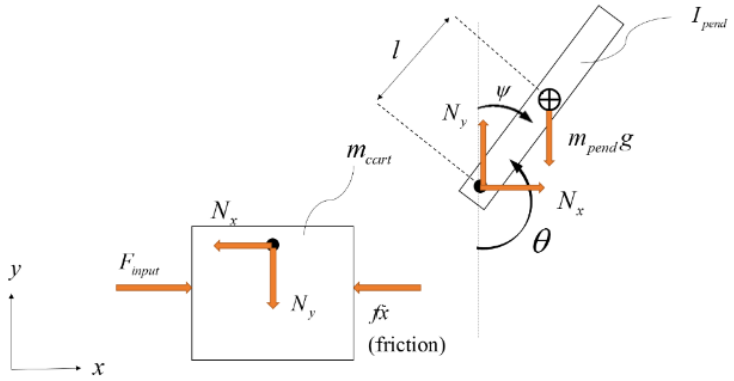


Figure 4: Free-body diagram for inverted pendulum**10193276**.

2.2.1. State Equations

The state equations for the system, describing the evolution of the state variables over time, are given by

$$F_{input} = (m_{cart} + m_{pend})\ddot{x} + f\dot{x} + m_{pend}l\ddot{\theta}\cos\theta - m_{pend}l\dot{\theta}^2\sin\theta \tag{1}$$

$$(I_{pend} + m_{pend}l^2)\ddot{\theta} + m_{pend}gl\sin\theta = -m_{pend}l\ddot{x}\cos\theta \tag{2}$$

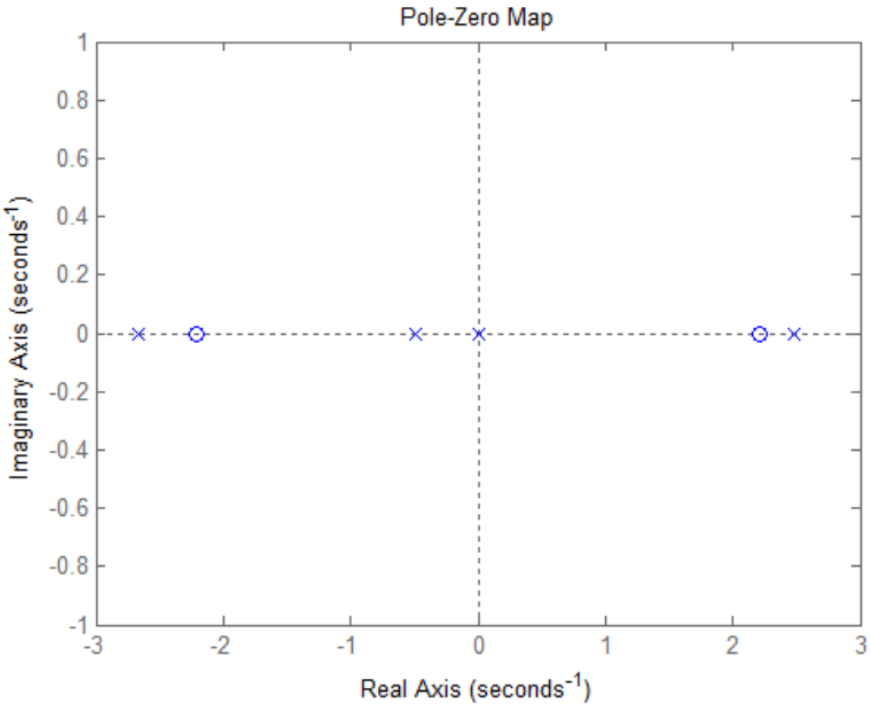


Figure 5: MATLAB-generated pole-zero map of the open loop system G_θ revealing a pole in the right half plane

$$\Theta(s) = \underbrace{\frac{\frac{m_{pend}l}{q}s}{s^3 + \frac{f(I_{pend}+m_{pend}l^2)}{q}s^2 - \frac{(m_{cart}+m_{pend})m_{pend}gl}{q}s - \frac{fm_{pend}gl}{q}}}_{G_\theta(s)} U_{input}(s) \tag{3}$$

It is concluded that for all positive value of l, I_{pend}, m_{cart}, m_{pend} and g the system in itself is unstable since it has a pole in the right half plane as can be seen in the pole-zero map in Figure 5.

2.2.2. Measurement Equation

The system's measurement output, which reflects the measured angle, is given by:

$$y(t) = \theta(t) \tag{4}$$

Here, $\mathbf{y}(t)$ denotes the measurement output, which directly corresponds to the system's angle $\theta(t)$, with no contribution from the gyroscope bias. This measurement model assumes that the system's only observable state is the angle.

2.2.3. State-space Representation

The state-space representation of the system is given by:

$$\begin{aligned} \underline{\dot{x}}(t) &= \underline{A}.\underline{x}(t) + \underline{B}.\underline{u}(t) \\ \underline{y}(t) &= \underline{C}.\underline{x}(t) + \underline{D}.\underline{u}(t) \end{aligned} \tag{5}$$

- **State Vector $\mathbf{x}(t)$**: This vector encapsulates the internal state of the system at time t. In this

case, it is defined as:

$$\mathbf{x}(t) = \begin{bmatrix} \theta(t) \\ \dot{\theta}(t) \end{bmatrix} \tag{6}$$

Where $\theta(t)$ represents the measured angle of the system, and $\dot{\theta}_{bias}(t)$ denotes the bias of the gyroscope.

- **Input Vector $\mathbf{u}(t)$**: This vector represents the external input given to the system at time t. In this case, it is defined as:

$$\mathbf{u}(t) = u_{input} \tag{7}$$

- **State Transition Matrix A**: This matrix describes how the state evolves over time. It is defined as:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ \frac{m_{pend} g l (m_{cart}+m_{pend})}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} & 0 \end{bmatrix} \tag{8}$$

The first row indicates that the angle is updated based on its previous value, and the second row shows that the gyroscope bias remains constant in this model.

- **Control Input Matrix B**: This matrix relates the control inputs to the state. In this case, it is defined as:

$$\mathbf{B} = \begin{bmatrix} 0 \\ \frac{m_{pend} l}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} \end{bmatrix} \tag{9}$$

This indicates that there are no direct control inputs affecting the state in this model.

- **Measurement Matrix C**: This matrix maps the state vector to the measurement output. It is defined as:

$$\mathbf{C} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{10}$$

This means that the measurement output directly reflects the angle, with no contribution from the gyroscope bias.

- **Feed-forward Matrix D**: This matrix relates the control input directly to the measurement output. In this case, it is defined as:

$$\mathbf{D} = 0 \tag{11}$$

This indicates that there is no direct influence of the control input on the measurement output.

- **Controllability and Observability**: The system is controllable observable if the matrices S and O have full rank.

$$\mathbf{S} = \begin{bmatrix} B & AB \end{bmatrix} \tag{12}$$

$$\mathbf{O} = \begin{bmatrix} C \\ CA \end{bmatrix} \tag{13}$$

The rank of S and O confirms that **the system is controllable and observable** for any positive value of l, I_{pend}, m_{cart}, m_{pend} and g.

2.3. Software Implementation Overview

Figure 6 illustrates the overall system architecture and control flow.

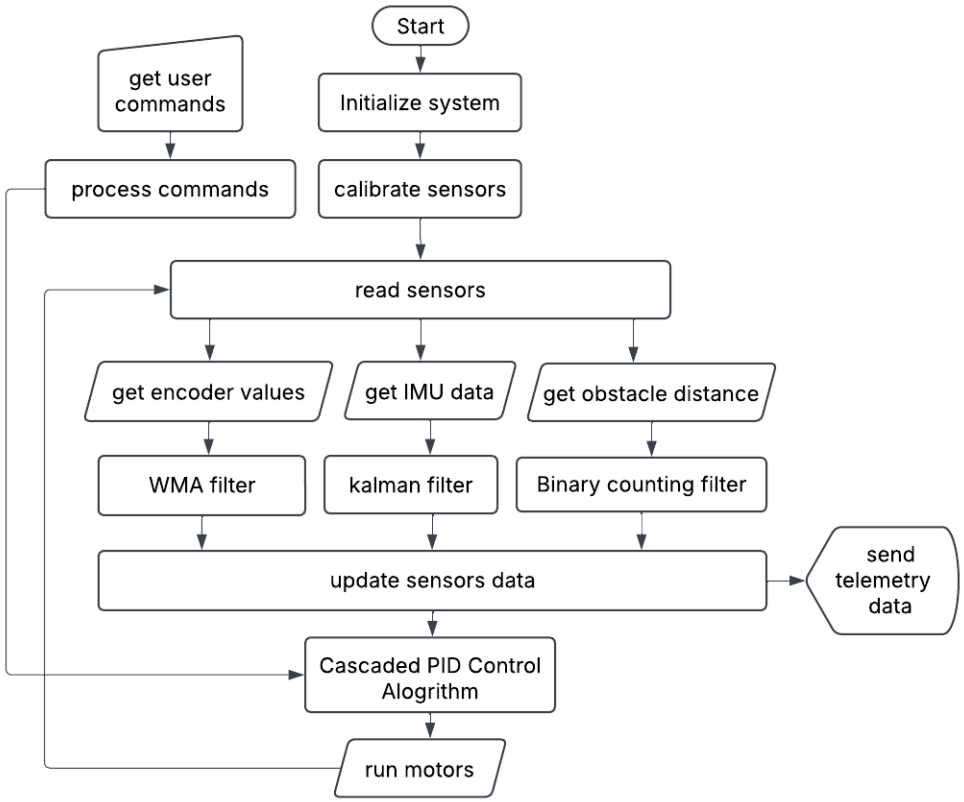


Figure 6: System operational flowchart.

The process begins with the reception of user commands, which are then parsed and processed. Subsequently, the system initializes and calibrates its various sensors. Sensor data, including encoder values, IMU data, and obstacle distance, is acquired and passed through respective filters (WMA, Kalman, and Binary Counting) to reduce noise and improve accuracy. The filtered sensor data is then aggregated and used to update the robot's internal state.

To stabilize the inverted pendulum in the upright position and to control the robot at the desired position using the PID control approach, two PID controllers with Linear–quadratic regulator (LQR) control : Yaw angle PID controller and position PID controller have been designed for the two control loops of the system.

A cascaded PID control algorithm (shown in Fig. 7) utilizes the updated sensor data to calculate appropriate motor commands. These commands are then sent to the motors, driving the robot's movement. Throughout this process, telemetry data is generated and transmitted, providing feedback on the robot's status and performance. This closed-loop control and monitoring system ensures precise and reliable operation. Further details on the individual components and algorithms are provided in the following subsections.

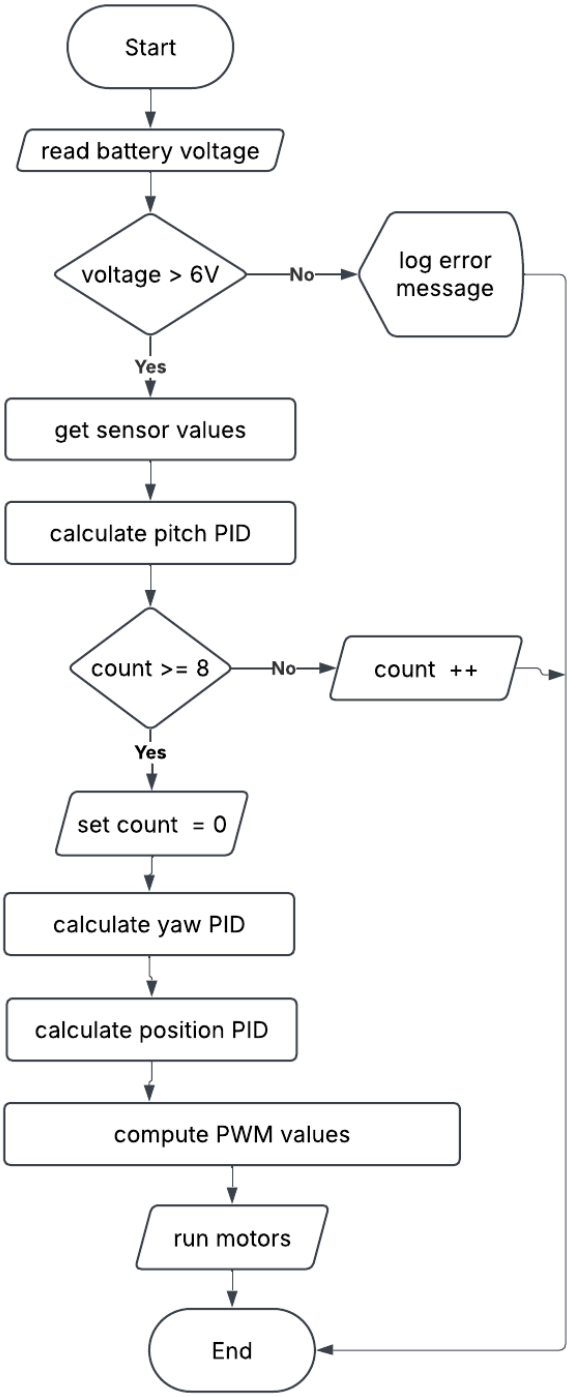


Figure 7: A simplified block diagram of the cascaded control loop used.

3. Sensor Fusion for Self-Balancing Robot using MPU6050

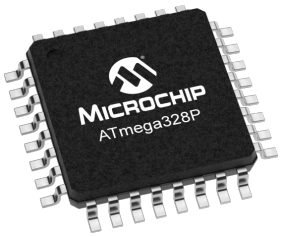
3.1. Introduction

Self-balancing robots require accurate estimation of their tilt angle for effective control. The MPU6050, a widely used Inertial Measurement Unit (IMU), provides raw accelerometer and gyroscope data. However, due to individual sensor limitations, direct usage of these readings is unreliable. Sensor fusion techniques like the Complementary Filter and Kalman Filter help in obtaining a stable and accurate tilt angle

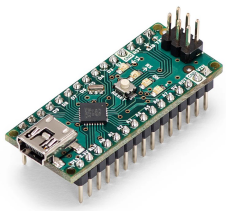
3.1.1. Microcontroller

The ATmega328P (shown in Fig. 8a) is a popular microcontroller from Microchip Technology, widely used in embedded systems and electronics projects. With a 16 MHz clock speed, 32 KB of flash memory, 2 KB of SRAM, and 1 KB of EEPROM **atmega_microchip**, the ATmega328P provides ample resources for this projects application.

The ATMEGA328P is also used in the Arduino Nano (shown in Fig. 8b), a widely adopted development board known for its low cost and open-source ecosystem **arduino_nano**. The combination of affordability and extensive community support makes it an ideal choice for rapid prototyping and academic research, ensuring easy integration with various sensors and motor drivers.



(a) ATMEGA328P **atmega_microchip**.



(b) Arduino Nano **arduino_nano**.

Figure 8

3.1.2. Inertial Measuring Unit

The MPU6050 is a widely used six-axis sensor that integrates a three-axis gyroscope and a three-axis accelerometer on a single chip, making it essential for motion tracking and stabilization applications (shown in Fig. 9). Its compact design and built-in Digital Motion Processor (DMP) enable real-time processing of sensor data, which is crucial for robotics, drones, and wearable devices. In applications like self-balancing robots, it provides accurate orientation and acceleration data necessary for maintaining stability.

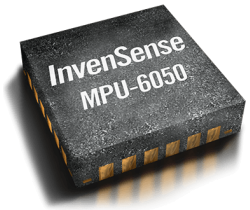


Figure 9: MPU-6050 **mpu6050**.

3.2. Working of MPU6050

The MPU6050 is a 6-axis IMU that provides:

- Accelerometer Data: Measures acceleration in X, Y, and Z axes. It helps in estimating tilt based on gravitational force.
- Gyroscope Data: Measures angular velocity around X, Y, and Z axes. Integration of gyroscope readings over time provides the tilt angle.

3.2.1. Reading Raw Sensor Data

The `read_mpu_6050_data()` function reads raw data from the MPU6050 sensor:

- It requests 14 bytes of data from the sensor, covering the accelerometer, gyroscope, and temperature sensor readings.
- The data is stored in respective variables after combining the high and low bytes.
- If data retrieval fails, an error message is displayed.

```
void mpu6050Base::read_mpu_6050_data() {
        Wire.beginTransmission(mpu6050_addr);
        Wire.write(0x3B);
        Wire.endTransmission();
        Wire.requestFrom(mpu6050_addr, (uint8_t)14);

        if (Wire.available() >= 14) {
                acc_x = Wire.read() << 8 | Wire.read();
                acc_y = Wire.read() << 8 | Wire.read();
                acc_z = Wire.read() << 8 | Wire.read();
                temperature = Wire.read() << 8 | Wire.read();
                gyro_x = Wire.read() << 8 | Wire.read();
                gyro_y = Wire.read() << 8 | Wire.read();
                gyro_z = Wire.read() << 8 | Wire.read();
        } else {
                ERROR_PRINT("Data cannot be read from MPU6050!");
        }

}
```

3.2.2. Issues with Raw Sensor Data

- **Accelerometer Noise**: While providing an absolute angle reference, accelerometer readings are noisy and susceptible to external forces.

- **Gyroscope Drift**: Over time, integration errors cause drift, leading to inaccurate angle estimation. To overcome these issues, sensor fusion techniques are applied.

3.3. Complementary Filter

The complementary filter is a simple yet effective method for sensor fusion **rabbany_design_2021 10193276 1174486**. It blends high-frequency data from the gyroscope with low-frequency data from the accelerometer:

3.3.1. Mathematical Model

Raw accelerometer readings A_y and Az can be used to calculate tilt angle using following equation:

$$\theta_{acc} = \arctan\left(A_y/Az\right) \tag{14}$$

Similarly, using raw angular velocity data from the gyroscope ω_x,

$$\begin{aligned} \dot{\theta}_{gyro}(t) &= -\omega_{x,bias}(t) \\ \dot{\omega}_{x,bias}(t) &= 0 \end{aligned} \tag{15}$$

In these equations, $\theta(t)$ represents the measured angle of the system, and $\omega_{\text{gyro,bias}}(t)$ represents the bias of the gyroscope. The first equation models how the angle evolves over time, influenced by the constant gyroscope bias. The second equation indicates that the gyroscope bias remains constant over time. The sampling time interval Δt and previous angle value $\theta_{previous}$ can be used to calculate tilt angle using following equation:

$$\theta_{gyro} = \theta_{previous,gyro} + \omega_{gyro}\Delta t \tag{16}$$

Then the final estimated angle $\theta_{estimate}$ is calculated as:

$$\theta_{estimate} = \alpha\ \theta_{gyro} + (1-\alpha)\ \theta_{acc} \tag{17}$$

Where α is the filter coefficient, ω is the angular velocity from the gyroscope, and θ_{acc} is the angle from the accelerometer.

3.3.2. Initial Calibration

The `init()` function is responsible for initializing the MPU6050 sensor and performing gyroscope calibration:

- The I2C communication is established with the sensor, and an acknowledgment is checked to ensure proper connection.
- The function sets up the MPU6050 registers using `setup_mpu_6050_registers()`, configuring the power management and sensitivity ranges for both the accelerometer and gyroscope.
- Gyroscope calibration is performed by collecting multiple readings and averaging them to calculate offsets for the X, Y, and Z axes. These offsets help reduce sensor drift.

```
void mpu6050Base::init() {
        Wire.beginTransmission(mpu6050_addr);
        if (Wire.endTransmission() != 0) {
                ERROR_PRINT("MPU6050 not connected!");
                return;
        }
        setup_mpu_6050_registers();

        for (int i = 0; i < CALIBRATION_SAMPLES; i++) {
                read_mpu_6050_data();
                gyro_x_cal += gyro_x;
                gyro_y_cal += gyro_y;
                gyro_z_cal += gyro_z;
                delay(3);
        }
        gyro_x_cal /= CALIBRATION_SAMPLES;
        gyro_y_cal /= CALIBRATION_SAMPLES;
        gyro_z_cal /= CALIBRATION_SAMPLES;
}
```

3.3.3. Calculating Angles

The `calculate()` function processes raw sensor data to determine pitch and roll angles:

- Gyroscope readings are corrected using calibration offsets to remove bias.
- Angular velocity is integrated over time to estimate orientation changes.
- The accelerometer-derived angles are computed from the total acceleration vector using trigonometric transformations.
- A complementary filter is applied to blend gyroscope and accelerometer readings, mitigating drift and noise while enhancing stability.

```
void mpu6050Base::calculate() {
        read_mpu_6050_data();

        gyro_x -= gyro_x_cal;
        gyro_y -= gyro_y_cal;
        gyro_z -= gyro_z_cal;

        angle_pitch += gyro_x * 0.0000611;
        angle_roll += gyro_y * 0.0000611;

        angle_pitch += angle_roll * sin(gyro_z * 0.000001066);
        angle_roll -= angle_pitch * sin(gyro_z * 0.000001066);

        acc_total_vector = sqrt((acc_x * acc_x) + (acc_y * acc_y) + (acc_z * acc_z));

        angle_pitch_acc = asin((float)acc_y / acc_total_vector) * 57.296;
        angle_roll_acc = asin((float)acc_x / acc_total_vector) * -57.296;

        if (set_gyro_angles) {
                angle_pitch = angle_pitch * 0.9996 + angle_pitch_acc * 0.0004;
                angle_roll = angle_roll * 0.9996 + angle_roll_acc * 0.0004;
        } else {
                angle_pitch = angle_pitch_acc;
```

```
24                  angle_roll = angle_roll_acc;
25                  set_gyro_angles = true;
26          }
27
28          angle_pitch_output = angle_pitch_output * 0.9 + angle_pitch * 0.1;
29          angle_roll_output = angle_roll_output * 0.9 + angle_roll * 0.1;
30
31  }
```

3.3.4. Results

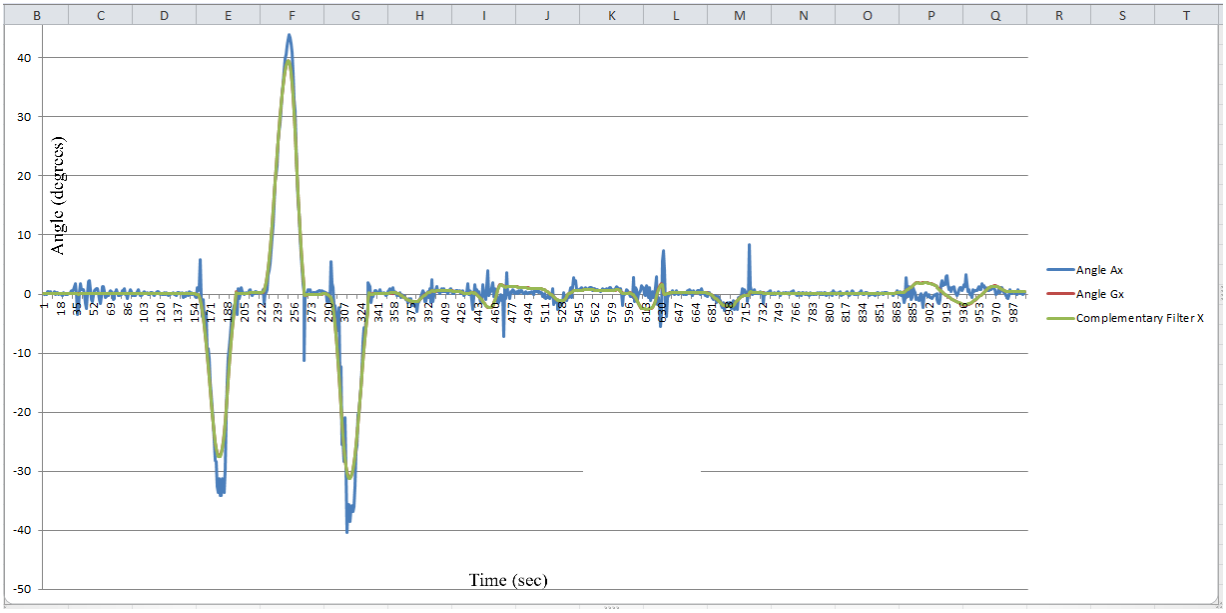


Figure 10: Pitch angle caluclated using a complementary filter with $\omega = 0\ 9$.

3.4. Extended Kalman Filter (EKF)

The Kalman filter (KF) provides a more sophisticated approach, estimating the true state of the system by minimizing the mean of the squared error. It involves prediction and update steps. However, assumes that both the process and measurement models are linear. However, many real-world systems, such as robotic motion and sensor fusion, exhibit inherent nonlinearities. When linearization is not feasible, the standard KF becomes inaccurate.

To address this limitation, the Extended Kalman Filter (EKF) extends the KF to nonlinear systems by employing a first-order Taylor series expansion. This method approximates the nonlinear system dynamics and measurement models with locally linearized representations, allowing the filter to be applied in scenarios where exact linearization is not feasible.

The EKF is an extension of the Kalman Filter for nonlinear systems, utilizing first-order Taylor series expansion to linearize process and measurement models. EKF maintains a Gaussian belief over the state, updating it through a prediction-correction cycle. The Jacobian matrices of the system dynamics and measurement functions are used to approximate state transitions and measurement updates. Its advantages include handling nonlinearities, fusing multi-sensor data, and improving estimation accuracy in noisy environments.

3.4.1. General State Equation

For non-linear system, with Stochastic disturbances:

$$\begin{aligned} \dot{\underline{x}}(t) &= f\left(\underline{x}(t), \underline{u}(t)\right) + \underline{d}(t) \\ \underline{y}(t) &= h\left(\underline{x}(t)\right) + \underline{n}(t) \end{aligned} \tag{18}$$

where,

- $\dot{\underline{x}}(t)$: This represents the time derivative of the state vector $\underline{x}(t)$, indicating how the state evolves over time.
- f: This is a nonlinear function that describes the system dynamics, taking the current state $\underline{x}(t)$ and the control input $\underline{u}(t)$ as arguments. It captures how the state changes based on the current state and control inputs.
- $\underline{d}(t)$: This term represents stochastic disturbances (or process noise) affecting the state dynamics, typically modeled as a zero-mean Gaussian noise.
- $y(t)$: This is the measurement vector at time t, representing the observed outputs of the system. It is the data collected from sensors or measurement devices.
- h: This is a nonlinear measurement function that maps the true state vector $\underline{x}(t)$ to the measurement space. It describes how the state influences the measurements. The function h can be complex and may involve various transformations of the state variables.
- $n(t)$: This term represents measurement noise, which is also typically modeled as zero-mean Gaussian noise. It accounts for inaccuracies in the measurements due to sensor errors, environmental conditions, or other random factors that can affect the observed data.

3.4.2. State Estimation

For a non-linear system the state form is as follows,

$$\begin{aligned} \dot{\underline{\hat{x}}}(t) &= f\left(\underline{\hat{x}}(t), \underline{u}(t)\right) + \underline{K}\left(y(t) - \hat{y}(t)\right) \\ \hat{y}(t) &= h\left(\underline{\hat{x}}(t)\right) \end{aligned} \tag{19}$$

The Kalman gain $\underline{K}$ is computed to optimally balance estimation uncertainty and measurement noise. To achieve this, the system is first linearized around the current state estimate by computing the Jacobians of the process and measurement models. To linearize the system around the current state estimate, the Jacobian matrices, which represent the first-order partial derivatives of the nonlinear functions, are computed as follows:

$$\underline{A}(t) = \left.\frac{\mathrm{d}f}{\mathrm{d}\underline{x}}\right|_{\underline{\hat{x}}(t), \underline{u}(t)} \quad \text{and} \quad \underline{C}(t) = \left.\frac{\mathrm{d}h}{\mathrm{d}\underline{x}}\right|_{\underline{\hat{x}}(t)} \tag{20}$$

where $\underline{A}(t)$ represents the partial derivatives of the state dynamics function f and $\underline{C}(t)$ represents the partial derivatives of the measurement function h.

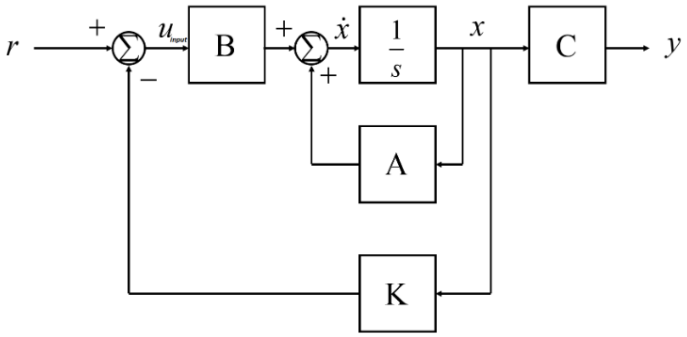


Figure 11: Block representation of state space system showing the system matrices A, B and C as well as the gain matrix K.

3.4.3. Covraince Matrix

The evolution of the state estimation covariance matrix follows the continuous-time Riccati equation, given by:

$$\dot{P}(t) = \underline{A}(t)P(t) + P(t)\underline{A}^T(t) + Q - P(t)\underline{C}^T(t)R^{-1}\underline{C}(t)P(t) \tag{21}$$

Where,

- $\dot{P}(t)$: his represents the time derivative of the covariance matrix $\dot{P}(t)$, which quantifies the uncertainty in the state estimate over time.
- $\underline{A}(t)$: This is the state transition matrix, which describes how the state evolves from one time step to the next.
- Q: This is the process noise covariance matrix, representing the uncertainty in the process model.
- $\underline{C}(t)$: This is the measurement matrix, which relates the state to the measurements.

- R: This is the measurement noise covariance matrix, representing the uncertainty in the measurements.

The Kalman filter is initialized with an initial covariance matrix:

$$P(0) = \mathbf{E}(\Delta\underline{x}(0)\Delta\underline{x}^T(0)) \tag{22}$$

The optimal Kalman gain, balancing estimation uncertainty and measurement noise, is given by:

$$\underline{K}(t) = P(t)\underline{C}^T R^{-1} \tag{23}$$

The time-discrete Kalman filter state update equations are:

$$\begin{aligned} \underline{x}_k &= \underline{A}\underline{x}_{k-1} + \underline{B}\underline{u}_k + \underline{d}_{k-1} \\ \underline{y}_k &= \underline{C}\underline{x}_k + \underline{n}_k \end{aligned} \tag{24}$$

where $\underline{x}_k$ is the state vector, $\underline{B}$ the control input matrix, and $\underline{y}_k$ the measurement vector. The discrete state-space representation is:

$$\begin{aligned} \dot{\underline{x}}(t) &= \underline{A}.\underline{x}(t) + \underline{B}.\underline{u}(t) \\ \underline{y}(t) &= \underline{C}.\underline{x}(t) + \underline{D}.\underline{u}(t) \end{aligned} \tag{25}$$

The state covariance matrix is:

$$\mathbf{P}_k = \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \tag{26}$$

where P_{00} represents uncertainty in the angle estimate and P_{11} in gyroscope bias. The Kalman gain is computed as:

$$\begin{aligned} \underline{K}_k &= \underline{P}_k^- \ \underline{C}^T(\underline{C}\ \underline{P}_k^-\ \underline{C}^T + \underline{R})^{-1} \\ \mathbf{K}_k &= \begin{bmatrix} \frac{P_{00}}{P_{00}+R_{angle}} \\ \frac{P_{10}}{P_{00}+R_{angle}} \end{bmatrix} \end{aligned} \tag{27}$$

Following the prediction step, the measurement update step refines the state estimate using the Kalman gain, which is derived to minimize the posterior estimation error covariance (see Appendix B for detailed calculations):

$$\underline{P}_k = \ (\underline{I} - \underline{K}_k\ \underline{C})\ \underline{P}_k^- \tag{28}$$

3.5. Software Implementation of EKF

For the implementation of the Extended Kalman Filter (EKF), we utilized the Kalman filter library developed by Kristian Lauszus **github_kalman_filter**. This library was modified in accordance with the GNU General Public License to meet the specific requirements of our project.

```
#include <Arduino.h>

class KalmanFilter {
 private:
  float m_dt, m_Q_angle, m_Q_gyro, m_R_angle, m_C_0;
  float q_bias = 0, angle_err = 0;
  float P[2][2] = {{1, 0}, {0, 1}}; // Covariance matrix
```

```
  float K_0 = 0, K_1 = 0;

 public:
  float angle = 0;

KalmanFilter(float dt, float Q_angle, float Q_gyro, float R_angle, float C_0)
: m_dt(dt), m_Q_angle(Q_angle), m_Q_gyro(Q_gyro), m_R_angle(R_angle), m_C_0(C_0) {}

float getAngle(float measured_angle, float measured_gyro) {
        // Predict
        angle += (measured_gyro - q_bias) * m_dt;
        angle_err = measured_angle - angle;

        // Update covariance matrix
        P[0][0] += m_Q_angle - P[0][1] - P[1][0];
        P[0][1] -= P[1][1];
        P[1][0] -= P[1][1];
        P[1][1] += m_Q_gyro;

        // Compute Kalman gain
        float E = m_R_angle + m_C_0 * P[0][0];
        K_0 = (m_C_0 * P[0][0]) / E;
        K_1 = (m_C_0 * P[1][0]) / E;

        // Update state
        angle += K_0 * angle_err;
        q_bias += K_1 * angle_err;

        // Update covariance matrix
        float C0_P00 = m_C_0 * P[0][0];
        P[0][0] -= K_0 * C0_P00;
        P[0][1] -= K_0 * P[0][1];
        P[1][0] -= K_1 * P[0][0];
        P[1][1] -= K_1 * P[0][1];

                return angle;
        }
};
```

4. Motor Speed Control

The system employs a cascaded Proportional-Integral-Derivative (PID) control loop to achieve real-time balance and motion control of the robot. This control architecture ensures stability by continuously monitoring and adjusting the robot's state using feedback from multiple sensors **jamil_modeling_2014**. Key inputs to the control loop include: the robot's pitch angle, gyroscope data, and motor encoder values, which provide critical information on orientation, angular velocities, and position.

- **Pitch angle**: Provides information about the robot's tilt relative to the vertical axis.
- **Gyroscope data**: Supplies angular velocity measurements for dynamic stabilization.
- **Motor encoder values**: Tracks wheel position and velocity for precise motion control.

The control algorithm computes outputs for **pitch**, **yaw**, and **position** at regular intervals, generating **pulse-width-modulation (PWM)** signals to adjust motor speeds via the motor driver (see Fig. 12). To prioritize system stability, the pitch angle is updated at a higher frequency (e.g., every control cycle), while the yaw angle and position control outputs are updated less frequently (e.g., every 8th cycle). This hierarchical approach ensures efficient resource utilization and robust performance.

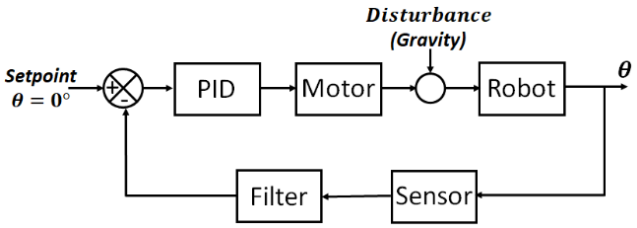
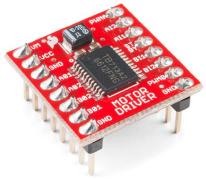


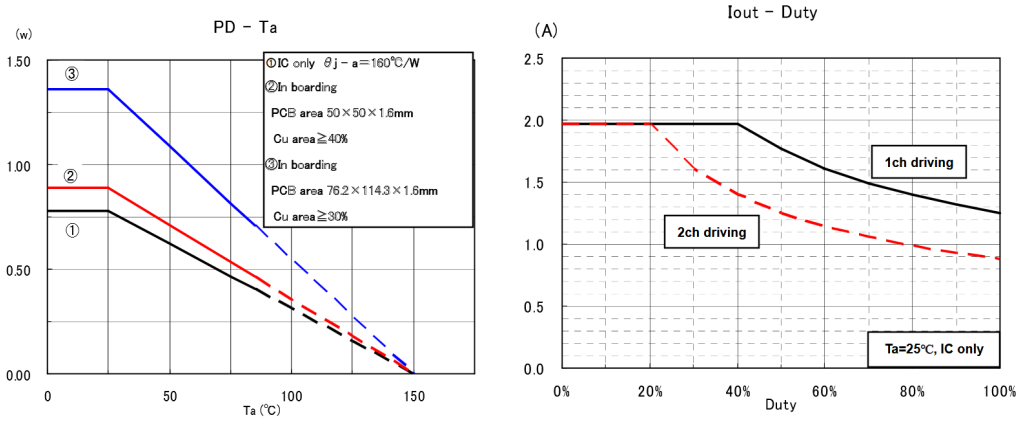
Figure 12: Robot's control block diagram.

4.0.1. Motor Driver

The TB6612FNG dual motor driver (shown in Fig. 13a) allows independent control of two DC motors. It uses a MOSFET H-bridge for bidirectional and Pulse Width Modulation (PWM) based motor speed control. Fig. 13b shows that each channel of the TB6612FNG can deliver up to 0.85A of current continuously. Furthermore, it consists of integrated over-current protection and thermal shutdown features for enhance reliability **TB6612FNG**.



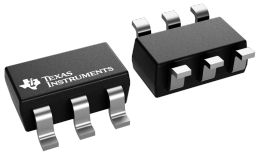
(a) TB6612FNG
TB6612FNG_photo.



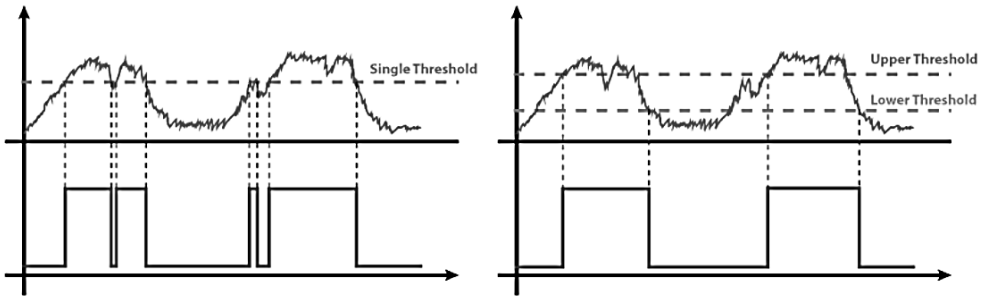
(b) Target characteristics for TB6612FNG **TB6612FNG**.

Figure 13

Instead of using two separate MCU pins for direction control, a single pin can be used with an inverted Schmitt trigger to generate the complementary signal automatically. For this purspose SN74LVC2G14 (shown in Fig. 14a) is used. It also enhances motor control by improving signal stability (similar to what is shown in Fig. 14b).



(a) SN74LVC2G14
SN74LVC2G14.



(b) Schmitt trigger output without hysteresis (left) and with hysteresis (right)
schmitt_trigger_hysteresis.

Figure 14

4.0.2. Drive Motors

The drive system employs NNHYTECH GA37 520 DC motors (37mm diameter, 12V, 360 RPM) equipped with Hall effect encoders (shown in Fig. 15) **dc_motor**. These motors feature a reduction gearbox, which enhances torque output while maintaining controlled rotational speed, making them well-suited for applications requiring precise motion control. The Hall effect encoders generate quadrature signals, enabling accurate measurement of speed and position. Motors from NHYTech were in this case.

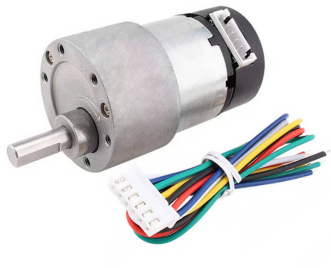


Figure 15: Similar construction motors were used in this project **dc_motor**.

4.1. Hierarchical Control Strategy

To optimize system performance and resource utilization, the control loop employs a hierarchical update strategy:

- **Pitch Control**: Updated at the highest frequency (e.g., every control cycle) to ensure rapid response to changes in the robot's tilt and maintain balance.
- **Yaw and Position Control**: Updated at a lower frequency (e.g., every 8th cycle) to reduce computational load while still providing adequate motion control.

This approach prioritizes critical tasks (e.g., maintaining balance) while efficiently managing system resources, ensuring robust and stable operation.

4.2. Basic PID Structure

The PID controller generates a control signal $u(t)$ based on the error signal $e(t)$, which is the difference between the desired setpoint and the measured process variable (see Fig. 12). The control signal is computed as:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \tag{29}$$

Where:

- $u(t)$: Control signal applied to the system.
- $e(t)$: Error signal, representing the deviation from the setpoint.
- K_p: Proportional gain, which responds to the current error.

- K_i: Integral gain, which addresses accumulated past errors.
- K_d: Derivative gain, which predicts future error trends based on the rate of change.

4.3. Discrete Time Implementation

For digital implementation, the continuous-time PID equation is discretized to suit microcontroller-based systems. The discrete-time PID control signal u[n]u[n] is calculated as:

$$u[n] = K_p e[n] + K_i T_s \sum_{k=0}^{n} e[k] + K_d \frac{e[n] - e[n-1]}{T_s} \tag{30}$$

where:

- T_s: Sampling period, representing the time interval between consecutive control updates.
- $e[n]$: Error signal at the nn-th sampling instant.
- $u[n]$: Control signal at the nn-th sampling instant.

This formulation ensures compatibility with real-time embedded systems while maintaining control precision.

4.3.1. Power Supply considerations

The custom-designed battery box (shown in Fig. 16) provides a portable and rechargeable power solution for the Elegoo robot **battery**. It houses two 18650 LiPo batteries, likely connected in series, to deliver a regulated voltage suitable for powering the robot's components. An integrated Battery Management System (BMS) ensures safe operation by protecting against overcharge, over-discharge, over-current, and short circuits. A power switch enables complete disconnection, while a USB charging port and status indicator LED facilitate easy recharging and monitoring.

Figure 16: ELEGOO battery pack with charger.

To monitor battery voltage, the Arduino Nano employs a voltage divider circuit, scaling the battery voltage to a safe range for its analog-to-digital converter (ADC). The divider consists of two resistors

in series, and the output voltage is given by:

$$V_{\text{out}} = V_{\text{battery}} \times \frac{R_2}{R_1 + R_2} \tag{31}$$

where V_{out} is the scaled-down voltage read by the Arduino's ADC, and R_1 and R_2 are the resistor values chosen to ensure the measured voltage remains within the 0–5V range.

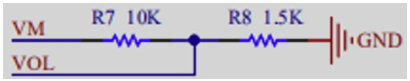


Figure 17: Voltage divider circuit used for in the robot **tumbller**.

The A2 pin is used for measurement, and the internal reference voltage is set to 1.1V, ensuring stable readings independent of supply voltage fluctuations. However, since 1.1V is too low to directly measure the battery voltage, a resistor divider is implemented. A 10kΩ (R1) and 1.5kΩ (R2) resistor pair (schematic shown in Fig. 23) is used, forming a 1/11 voltage division ratio:

$$V_{\text{out}} = V_{\text{battery}} \times \frac{1\ 5}{10 + 1\ 5} = V_{\text{battery}} \times 0\ 136 \tag{32}$$

For a fully charged 8.4V LiPo battery, the resulting $V_{\text{out}} \approx 1.14$ V, which is safely within the 1.1 V reference range but may introduce minor ADC saturation at peak voltage.

The following function reads the battery voltage using the voltage divider and determines the battery status:

```
void Voltage_Measure()
{
        if (millis() - vol_measure_time > 1000) //Measured every 1000 milliseconds
        {
                vol_measure_time = millis();
                double voltage = (analogRead(VOL_MEASURE_PIN) * 1.1 / 1024) * ((10 + 1.5) /
                    1.5); //Read voltage value
                Serial.print("Current voltage value :");
                Serial.println(voltage);
                if(voltage >7.8)
                Serial.println("The battery is fully charged");
                else
                Serial.println("Low battery");
        }
}
```

4.4. Tuning Methodology

The Ziegler-Nichols method is a widely used and systematic approach for tuning PID controllers. It provides a reliable framework for determining initial controller parameters, which can then be fine-tuned for optimal performance. The method involves inducing controlled oscillations in the system to identify critical parameters, which are then used to calculate the proportional, integral, and derivative gains.

4.4.1. Ziegler-Nichols Method

The Ziegler-Nichols tuning procedure consists of the following steps:

1. **Initialize Parameters**: Set the integral gain K_i and derivative gain $K_d$to zero, leaving only the proportional gain K_p active.

2. **Induce Oscillations**: Gradually increase K_p until the system exhibits sustained oscillations. At this point, the system is at the threshold of stability, and the proportional gain is referred to as the ultimate gain K_u. The period of these oscillations is denoted as T_u.

3. **Record Critical Values**: Note the values of K_u and T_u, as they are essential for calculating the final PID parameters.

4. **Calculate PID Gains:** Using the recorded values, compute the PID parameters as follows:

$$\begin{aligned} K_p &= 0\ 6K_u \\ T_i &= 0\ 5T_u \\ T_d &= 0\ 125T_u \end{aligned} \tag{33}$$

   Here, T_i and T_d represent the integral and derivative time constants, respectively. These values are then used to determine the integral and derivative gains:

$$K_i = \frac{K_p}{T_i} \quad \text{and} \quad K_d = K_p \cdot T_d. \tag{34}$$

This method provides a robust starting point for achieving stable control. However, it is important to note that the Ziegler-Nichols method may require additional fine-tuning to account for system-specific dynamics and performance requirements. The calculated parameters serve as an initial baseline, which can be further optimized through iterative testing and adjustment.

4.4.2. Practical Tuning Guidelines

In addition to the Ziegler-Nichols method, practical tuning guidelines were applied to refine the controller performance:

- Start with a small proportional gain K_p (e.g. $K_p = 10$) to avoid instability.
- Introduce the derivative term K_d to dampen oscillations, typically setting $K_d = 0\ 1K_p$.
- Fine-tune K_p, K_i, and K_d iteratively to achieve optimal stability and responsiveness.

4.5. Pitch PID Control:

The pitch control loop ensures the robot maintains its upright position. The primary objective of the pitch controller is to minimize the deviation of the robot's pitch angle from a set-point, which is ideally zero degrees (i.e., upright). The pitch control output is calculated using the PD algorithm, where the error is the difference between the current pitch angle and the desired pitch angle.

$$\tau_{\theta,pid} = K_{p\theta}(\theta_{desired} - \theta_{measured}) + K_{d\theta}\frac{d}{dt}(\theta_{desired} - \theta_{measured}) \tag{35}$$

Below is its code implementation:

```
inline void runPitchControl() {
        pitch_pid_output = (kp_balance * (kalman.angle - 0)) + (kd_balance * gyro_x);
}
```

The final results are shown in Fig. 18.

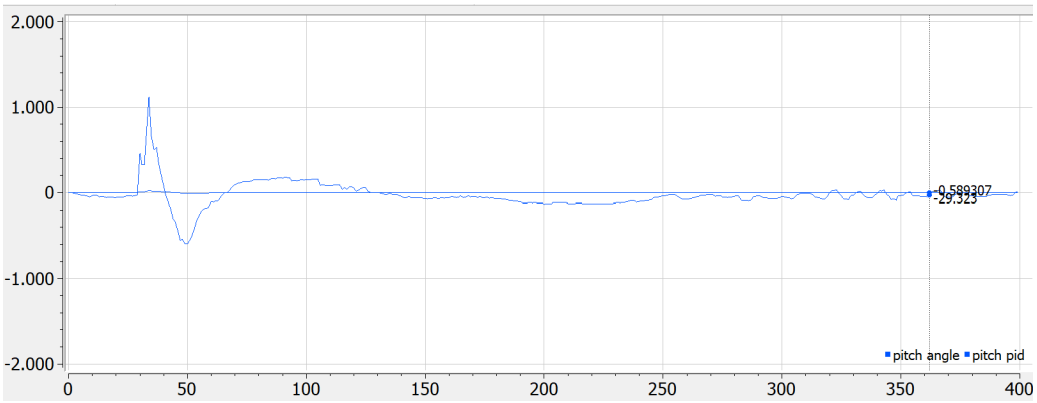


Figure 18: Pitch angle control system response at $K_{p\theta} = 55$ and $K_{d\theta} = 1.25$. The data was recorded at baud rate of 115200.

4.6. Eliminating Jitter

Once the system reaches stability, it begins to exhibit jitter, as observed in the PID output between data points 250 and 400 in Fig. 18. This jitter arises due to the high sensitivity of the derivative gain ($K_{d\theta}$) to fluctuations in the gyroscope readings ($\dot{\theta}$). While reducing $K_{d\theta}$ mitigates the jitter, it also diminishes the system's ability to dampen oscillations, leading to excessive overshoot and potential instability. Fig. 19 illustrates the system's response when using $K_{p\theta} = 55$ and $K_{d\theta} = 0.75$ where significant overshoot is evident.

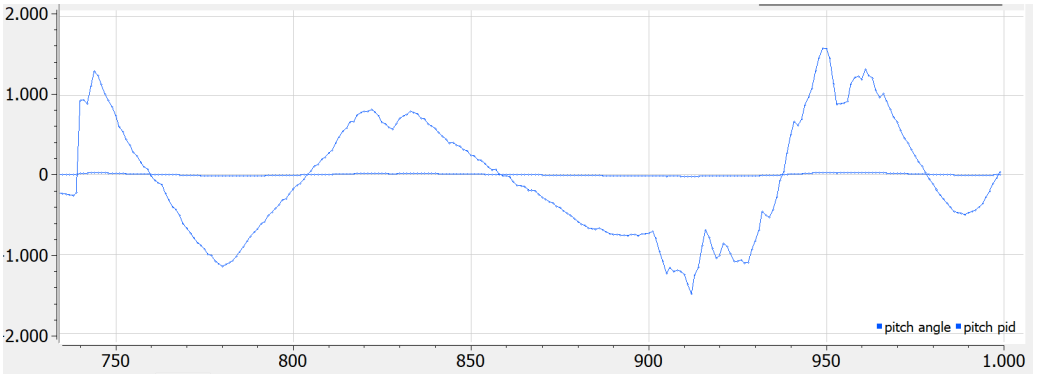


Figure 19: Pitch angle control system response at $K_{p\theta} = 55$ and $K_{d\theta} = 0.75$. Data recorded at baud rate of 115200.

To address this issue, an **adaptive derivative gain approach** is implemented, where $K_{d\theta}$ is adjusted dynamically based on the pitch angle of the robot. When the absolute pitch angle exceeds a predefined threshold, a higher derivative gain is used to provide stronger correction, while a lower gain is applied for small angles to minimize jitter. The corresponding implementation is as follows:

```
if (pitch_angle > 8) { kd_balance = kd_balance_large_angle; }
```

```
else { kd_balance = kd_balance_small_angle; }

inline void runPitchControl() {
        pitch_pid_output = (kp_balance * (kalman.angle - 0)) + (kd_balance * gyro_x);
}
```

This adaptive control strategy effectively stabilizes the system while reducing excessive oscillations. Fig. 20 presents the improved system response, demonstrating a smoother transition and enhanced overall stability.

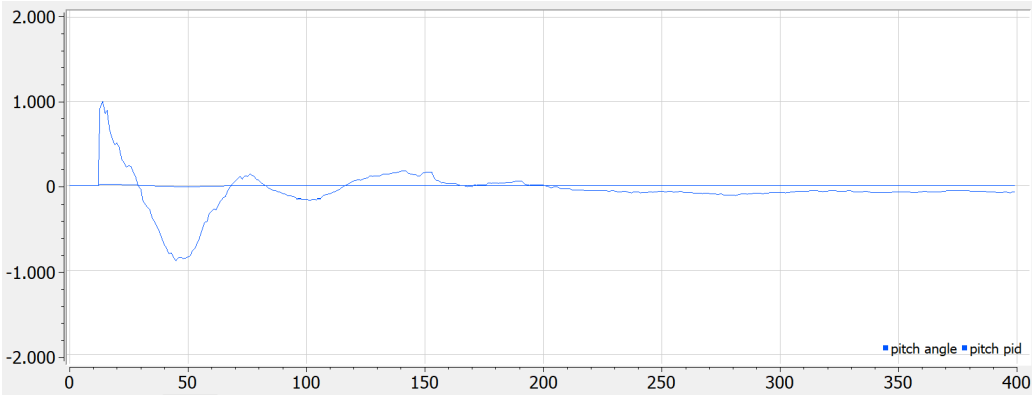


Figure 20: Pitch angle control system response. The data was recorded at baud rate of 115200.

4.7. Yaw Control:

Yaw control is responsible for controlling the robot's rotational movement around its vertical axis. The yaw PID controller computes the control output based on the robot's angular velocity, which is measured by the gyroscope along the z-axis.

$$\tau_{\phi,pid} = K_{p\phi}(\phi_{desired} - \phi_{measured}) + K_{d\phi}\frac{d}{dt}(\phi_{desired} - \phi_{measured}) \tag{36}$$

Below is its code implementation:

```
inline void runYawControl(){
        float delta_yaw_angle = yaw_angle_degrees - desired_yaw_angle;
        yaw_pid_output = (kp_turn * delta_yaw_angle) + (kd_turn * gyro_z);
}
```

The yaw control adjusts the motor speeds to achieve the desired angle, ensuring the robot maintains a stable heading. Fig. 21 illustrates the system's response with $K_{p\phi} = 2.5$ and $K_{i\phi} = 0.5$ when transitioning from an initial angle of -20 *deg* to final angle of $+20$ *deg*.

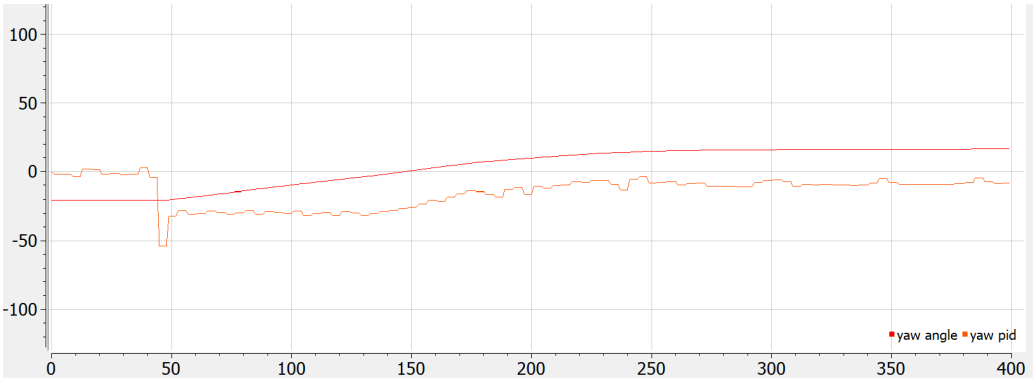


Figure 21: Yaw angle control system response at $K_p = 2.5$ and $K_i = 0.5$ for an angle transition from -20 *deg* to $+20$ *deg*. The data was recorded at baud rate of 115200.

4.8. Position Control:

Position control is implemented to ensure the robot moves smoothly and accurately along a path or to a target location. The encoder feedback from the left and right wheels is used to calculate the robot's displacement and speed. The position PID controller adjusts the motor speeds to minimize the error in position and velocity.

$$\tau_{x,pid} = K_{px}(x_{desired} - x_{measured}) + K_{dx}\frac{d}{dt}(x_{desired} - x_{measured}) \tag{37}$$

Below is its code implementation:

```
inline void runPositionControl(){
        position_pid_output = - (kp_position * (current_position - move_to_position)) - (
            kd_position * encoder_speed_filtered);
}
```

The position controller continuously adjusts motor speeds to reduce position error while maintaining a stable heading. Fig. 22 presents the system's response when $K_{px} = 0.26$ and $K_{dx} = 20$ when moving from relative distance of -20 *cm* to $+20$ *cm*.

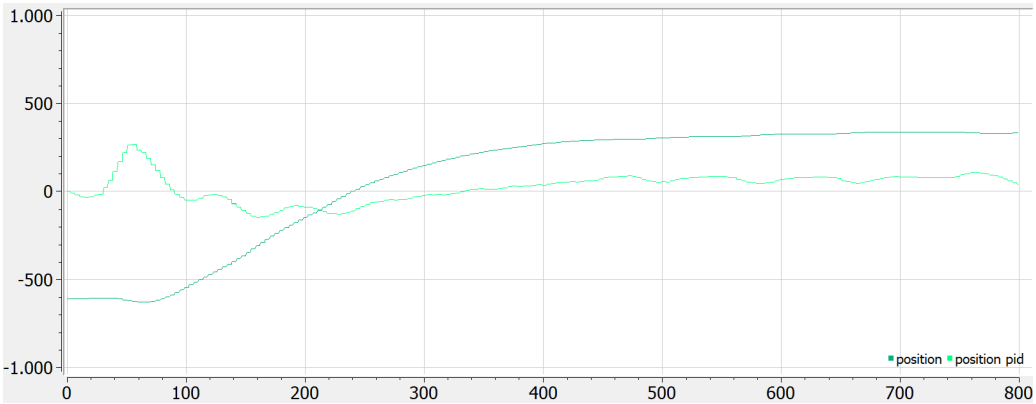


Figure 22: Position control system response with $K_{px} = 0.26$ and $K_{dx} = 20$ for a displacement of -20 *cm* to $+20$ *cm*. Data recorded at baud rate of 115200.

4.9. Combining Control Outputs

The final motor control is achieved by combining the outputs from all three PID controllers. The outputs from the pitch, yaw, and position PID controllers are used to calculate the motor speeds, which determine the robot's motion. Specifically, the following equation is used to compute the PWM values for the left and right motors:

$$\tau_{left,motor} = \tau_{\theta,pid} - \tau_{\phi,pid} - \tau_{x,pid} \tag{38}$$

$$\tau_{right,motor} = \tau_{\theta,pid} + \tau_{\phi,pid} - \tau_{x,pid} \tag{39}$$

Below is its code implementation:

```
void balance(){
...

        pwm_left = pitch_pid_output - yaw_pid_output - position_pid_output;
        pwm_right = pitch_pid_output + yaw_pid_output - position_pid_output;

...
}
```

The computed Pulse-Width-Modulation (PWM) are transmitted to the motor drivers, which adjust the robot's movement and balance in real time. This closed-loop control mechanism ensures precise and stable operation by continuously refining motor outputs based on sensor feedback. A simplified diagram of the motor control loop is illustrated in Fig. 23.

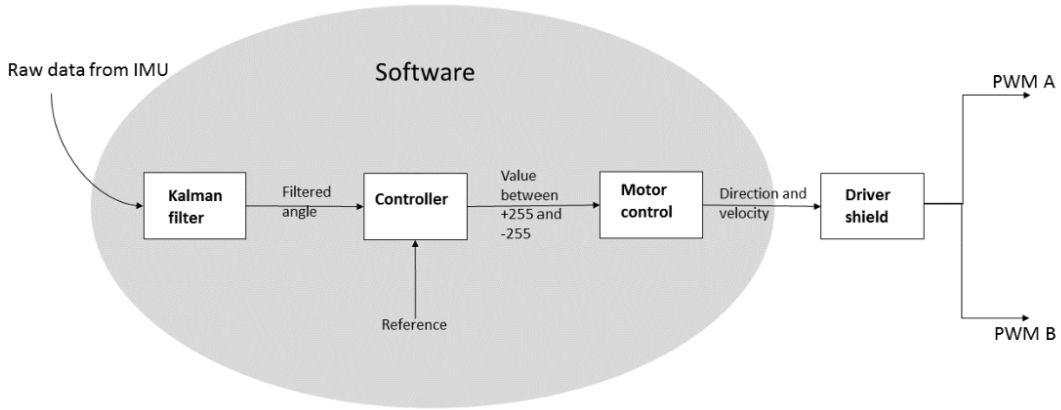


Figure 23: A simplified diagram of the motor control loop **10193276**.

4.10. Results

The use of PID controllers for pitch, yaw, and position control enables the robot to maintain balance and navigate effectively (see Fig. 25). The proportional, integral, and derivative terms in each PID

loop allow the system to respond to real-time errors, minimize steady-state deviations, and anticipate future errors, leading to smooth and precise control of the robot's motion. The integration of these PID controllers is fundamental to the robot's stability and performance. Final results are shown in Fig. 24

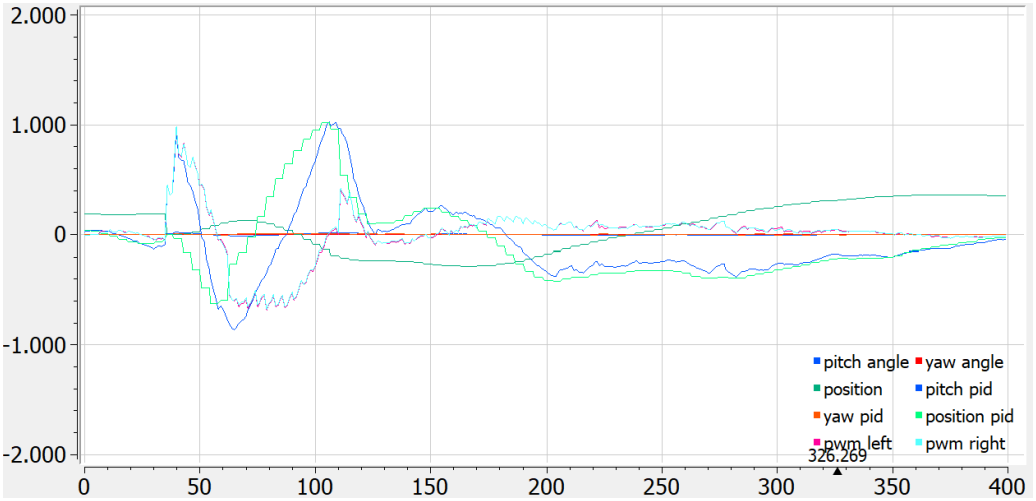


Figure 24: The final control system individual output values recorded at baud rate of 115200.

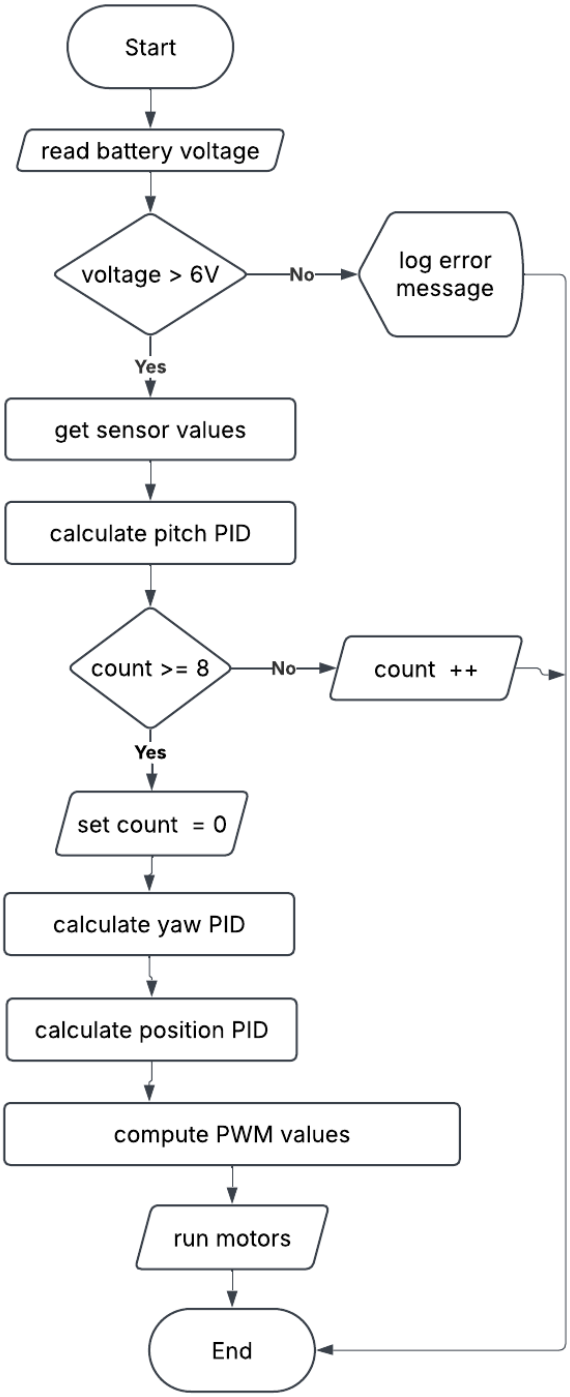


Figure 25: A simplified block diagram of the cascaded control loop used.

5. Front Obstacle Detection

To robustly detect obstacles in its forward path, the robot employs a combination of an **ultrasonic distance-measuring sensor** and **infrared (IR) proximity sensors**. A simplified representation of the obstacle detection algorithm is depicted in Fig. 26.

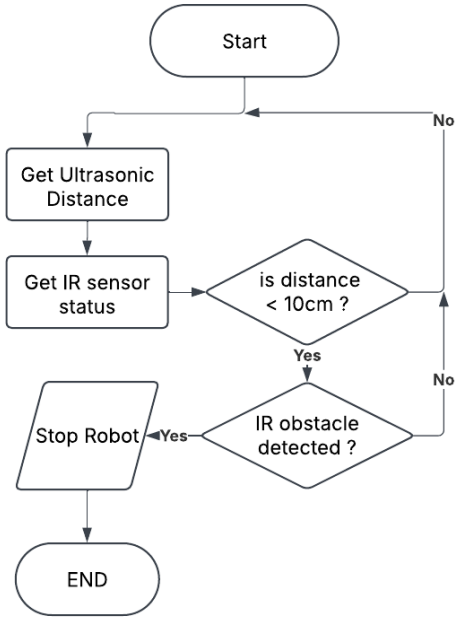


Figure 26: Flowchart for a robot's obstacle avoidance alogrithm using ultrasonic and infrared sensors.

5.0.1. Ultrasonic Distance Sensor

The HC-SR04 is an ultrasonic distance sensor used for measuring the distance to an obstacle by sending an ultrasonic pulse and measuring the time it takes for the echo to return. It operates based on the principle of time-of-flight of sound waves, with a known speed of sound in air.

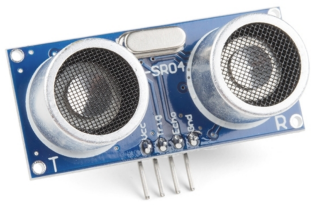


Figure 27: Ultrasonic distance sensor by Sparkfun electronics **ultrasonic_sensor**.

5.0.2. Infrared Sensing

The robot is equipped with infrared proximity sensors at the front-left and front-right directions using the Everlight Elec IR Receiver (IRM-56384) and the Infrared LED (IR204C-A). These sensors detect obstacles by transmitting a modulated infrared signal and detecting its reflection.

(a) IR204C-A **ir_led**.

(b) IRM-56384 **ir_receiver**.

Figure 28

5.1. Ultrasonic Working Principle

As discussed in Section 5.0.1, the ultrasonic sensor operates based on the **time-of-flight** principle, utilizing sound waves to determine the distance to an obstacle. The sensor comprises two key components:

- A **transmitter** that emits high-frequency ultrasonic pulses (typically at 40 kHz).
- A **receiver** that detects the reflected pulses after they bounce off an obstacle.

The distance to the obstacle is calculated by measuring the time delay between the transmission of the ultrasonic pulse and its reception. The formula used to compute the distance is:

$$d = \frac{t \times v}{2} \tag{40}$$

where:

- d is the measured distance to the obstacle (in meters),
- t is the time delay between transmission and reception (in microseconds),
- v is the speed of sound in air (approximately 343 m/s at room temperature, 20°C).

5.1.1. Ultrasonic Implementation

The HC-SR04 requires control signals to be sent from a microcontroller:

1. A short **trigger pulse** (at least 10 µs) is sent to the `TRIG` pin.
2. The sensor responds with a high signal on the `ECHO` pin, the duration of which corresponds to the distance.
3. The duration of the `ECHO` signal is measured to determine the distance.

5.1.2. Code for Distance Measurement

Based on the datasheet **ultrasonic_sensor**, an operating frequency of 20 Hz (corresponding to a 50 ms interval) is selected for distance measurements. The speed of sound is taken as 340.29 m/s, leading to the following constants in the code:

```
constexpr uint8_t USONIC_GET_DISTANCE_DELAY_MS = 50; // Measurement interval
constexpr float SPEED_OF_SOUND_HALVED = (340.29 * 100.0) / (2 * 1000 * 1000); // Speed of
    sound in cm/us, divided by 2
```

Code Explanation:

- `USONIC_GET_DISTANCE_DELAY_MS`: Ensures measurements are taken at 20 Hz intervals.
- `SPEED_OF_SOUND_HALVED`: Converts the speed of sound to cm/µs and accounts for the round-trip travel time of the ultrasonic pulse.

The following C++ function initiates distance measurement using the HC-SR04 ultrasonic sensor:

```
void StartUltrasonicMeasurement() {
        if (millis() - usonicGetDistancePrevTime > USONIC_GET_DISTANCE_DELAY_MS) {
                usonicMeasureFlag = SEND;
                usonicGetDistancePrevTime = millis(); // Update timestamp

                attachPinChangeInterrupt(ECHO_PIN, HandleUltrasonicMeasurementInterrupt,
                    RISING); // Attach interrupt for rising edge

                digitalWrite(TRIG_PIN, LOW);    // Reset TRIG pin
                delayMicroseconds(2);
                digitalWrite(TRIG_PIN, HIGH);   // Send 10 us trigger pulse
                delayMicroseconds(10);
                digitalWrite(TRIG_PIN, LOW);    // Reset TRIG pin
        }
}
```

Code Explanation:

- The function `StartUltrasonicMeasurement()` ensures that the measurement is taken at regular intervals.
- A global flag `usonicMeasureFlag` is set to `SEND`, indicating that the trigger pulse is sent.
- The function `attachPinChangeInterrupt()` attaches an interrupt to detect when the `ECHO` pin goes HIGH.
- The `TRIG` pin is first set LOW (to reset), then HIGH (to trigger the pulse), and then set LOW again.

This setup enables precise distance measurement by capturing the time delay between sending and receiving the ultrasonic pulse.

5.1.3. Interrupt Service Routine

The following function handles the interrupt to measure the distance using the ultrasonic sensor:

```
void HandleUltrasonicMeasurementInterrupt() {
        if (usonicMeasureFlag == SEND) {
                usonicMeasurePrevTime = micros();        // Record timestamp of rising edge
                attachPinChangeInterrupt(ECHO_PIN, HandleUltrasonicMeasurementInterrupt,
                    FALLING);
                usonicMeasureFlag = RECEIVED;    // Update flag to indicate echo reception
        } else if (usonicMeasureFlag == RECEIVED) {
                usonicDistanceValue = (uint8_t)((micros() - usonicMeasurePrevTime) *
                    SPEED_OF_SOUND_HALVED);     // Compute distance
                usonicMeasureFlag = IDLE;        // Reset flag for next measurement
        }
}
```

Code Explanation:

1. **Rising Edge Detection**:
   - When the `ECHO` signal rises (RISING edge), the current time is recorded using `micros()`.
   - The interrupt is reattached to detect the falling edge of the `ECHO` signal.
   - The flag `usonicMeasureFlag` is updated to `RECEIVED` to indicate that the echo signal is being processed.
2. **Falling Edge Detection**:
   - When the falling edge is detected, the elapsed time is computed as the difference between the current time and the previously recorded timestamp.
   - The distance is calculated using the formula:

$$d = \frac{\Delta t \times v}{2} \tag{41}$$

   - The flag `usonicMeasureFlag` is reset to `IDLE` to prepare for the next measurement cycle.

- When the `ECHO` signal rises (RISING edge), the timestamp is recorded using `micros()`.
- The interrupt is reattached to detect the falling edge of the `ECHO` signal.
- When the falling edge is detected, the elapsed time is computed and converted to distance using the speed of sound formula.
- The system resets for the next measurement by setting `usonicMeasureFlag` to `IDLE`.

5.2. Infrared Sensing Implementation

As discussed in Section 5.0.2, Infrared LED and Reciever. The following code implements the control and processing logic for the **IR proximity sensors**:

```
void IRSesorSend38KPule(unsigned char ir_pin){
        for( int i = 0; i < 39; i++) {
                digitalWrite(ir_pin, LOW);
                delayMicroseconds(9);
                digitalWrite(ir_pin, HIGH);
                delayMicroseconds(9);
```

```
        }
}

void ProcessLeftIRSensor(){
        if (millis() - irLeftCountTime > IR_COUNT_DELAY_MS) {
                UpdateSlidingWindow(irLeftPulseCount >= 3, irLeftHistory, irLeftIndex,
                    irLeftRunningCount);
                irLeftIsObstacle = (irLeftRunningCount >= 5); // Check if obstacle is detected
                irLeftPulseCount = 0;
                irLeftCountTime = millis(); // Update timestamp
        }
}

void ProcessRightIRSensor(){
        if (millis() - irRightCountTime > IR_COUNT_DELAY_MS) {
                UpdateSlidingWindow((irRightPulseCount >= 3), irRightHistory, irRightIndex,
                    irRightRunningCount);
                irRightIsObstacle = (irRightRunningCount >= 5); // Check if obstacle is
                    detected
                irRightPulseCount = 0; // Reset pulse coun
                irRightCountTime = millis(); // Update timestamp
        }
}
```

Code Explanation:

1. **IR Pulse Generation**:
   - The function `IRSensorSend38KPulse()` generates a 38 kHz modulated signal by toggling the IR LED pin at a specific frequency.
   - Each cycle consists of 9 µs LOW and 9 µs HIGH states, repeated 39 times to ensure reliable signal transmission.
2. **Sensor Data Processing**:
   - The functions `ProcessLeftIRSensor()` and `ProcessRightIRSensor()` process the data from the left and right IR sensors, respectively.
   - A sliding window algorithm is used to filter out noise and improve detection reliability:
   - The function `UpdateSlidingWindow()` updates a history buffer and running count of detected pulses.
   - An obstacle is considered detected if the running count exceeds a threshold (e.g., 5 out of 10 readings).
3. **Obstacle Detection**:
   - The flags `irLeftIsObstacle` and `irRightIsObstacle` indicate whether an obstacle is detected on the left or right side, respectively.
   - The pulse counts and timestamps are reset after each processing cycle to prepare for the next measurement.

6. Remote Control and Communication

The remote control and communication system of the two-wheeled self-balancing robot was designed to enable seamless and reliable interaction between the user and the robot. For this purpose, we integrated the BT16 Bluetooth UART Module, which provided a robust wireless communication link.

6.1. Bluetooth Module Integration

The BT16 Bluetooth UART Module is a high-performance component based on the Airoha ABI 602 chipset, compliant with the Bluetooth 4.2 BLE standard. It interfaces with the microcontroller via UART, ensuring efficient bidirectional data transmission. The module supports GATT-based communication through an integrated transparent data transmission service, enabling seamless and reliable Bluetooth connectivity.

A key feature of the BT16 module is its *serial command mode*, which allows direct configuration and control via UART commands. Users can modify essential parameters, such as the UUID, device name, and connection settings, facilitating flexible integration into various applications.

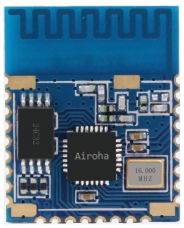


Figure 29: Ultrasonic distance sensor by Sparkfun electronics **bluetooth_module**.

6.2. Custom Communication Protocol

A custom communication protocol was developed to manage the exchange of control commands and telemetry data, which is structured around well-defined command and telemetry packets, implemented in `comm.hpp`. These packets support multiple data formats, ensuring compatibility with various communication protocols. The functions responsible for sending and receiving telemetry data are listed in Tables 1 and 2.

6.3. User Interfaces

The user interface (UI) for the robot is designed to facilitate interactive control and real-time monitoring. It integrates command parsing, telemetry data handling, and communication protocols such as Inter-Integrated Circuit (I2C) and Universal Asynchronous Receiver-Transmitter (UART). Users can send commands to the robot and receive real-time telemetry feedback, ensuring efficient control and monitoring.

6.4. UART Communication

The Universal Asynchronous Receiver-Transmitter (UART) protocol is used for serial communication, typically with a Bluetooth module or a computer for debugging and remote control. The com-

munication is initialized at a baud rate of `9600`, which is commonly used for Bluetooth modules **bluetooth_module**.

Table 1: UART Communication Methods

Packet Type	Function	Description
Command Packets	`readUartBytes()`	Reads command data as raw bytes.
	`readUartASCII()`	Reads command data as an ASCII string.
	`sendUartBytes()`	Sends command data as raw bytes.
	`sendUartASCII()`	Sends command data as an ASCII string.
Telemetry Packets	`sendUartBytes()`	Sends telemetry data as raw bytes.
	`sendUartASCII()`	Sends telemetry data as an ASCII string.
	`readUartBytes()`	Reads telemetry data as raw bytes.
	`readUartASCII()`	Reads telemetry data as an ASCII string.

6.5. I2C Communication

Inter-Integrated Circuit (I2C) communication is used for short-range, two-wire serial communication. The robot operates as an I2C slave with a predefined address (SLAVE_ADDR = 8), allowing it to receive commands and transmit telemetry data over the I2C bus.

Table 2: I2C Communication Methods

Packet Type	Function	Description
Command Packets	`sendI2CBytes(addr)`	Sends command data as raw bytes to the address.
	`sendI2CASCII(addr)`	Sends command data as ASCII to the address.
	`readI2CBytes(addr)`	Reads command data as raw bytes from the address.
	`readI2CASCII(addr)`	Reads command data as ASCII from the address.
Telemetry Packets	`sendI2CBytes()`	Sends telemetry data as raw bytes.
	`sendI2CASCII()`	Sends telemetry data as ASCII.
	`readI2CBytes(addr)`	Reads telemetry data as raw bytes from the address.
	`readI2CASCII(addr)`	Reads telemetry data as ASCII from the address.

6.6. Sending Commands

Users can send commands to control the robot's movement, rotation, or stop function. Commands follow a predefined structured format and can be transmitted via either I2C or UART. The available commands are listed in Table 3.

ASCII Format: The commands sent in ASCII is formatted as a comma-separated string:

<command>,<command_value>,<command_speed>

Table 3: List of Commands and Corresponding Values

Command	Value	Description
`Stop`	0	Stops the robot's movement.
`Move`	1	Moves forward or backward (in cm) at a given speed.
`Rotate`	2	Rotates the robot by a angle (in degrees) at a given speed.
`INVALID`	3	Represents an invalid or unrecognized command.

Example Command:

- Stop the robot: `0`
- Move forward 100 cm at 50% speed: `1,100,50`
- Rotate 90° at 30 speed%: `2,90,30`

6.7. Receiving Telemetry Data

The robot continuously monitors its state and environment using onboard sensors. This data is packaged into a structured format and transmitted back to the user for real-time monitoring and feedback. Telemetry data includes:

- **Yaw Angle**: The robot's current orientation in degrees.
- **Distance Traveled**: The total distance traveled by the robot in centimeters.
- **Ultrasonic Distance**: The distance to the nearest obstacle, as measured by the ultrasonic sensor in centimeters.

ASCII Format: The telemetry data recieved in ASCII is formatted as a comma-separated string:

<yaw_angle>,<distance_traveled>,<ultrasonic_distance>

Example Output:

45,200,30

This indicates a yaw angle of 45 degrees, a distance traveled of 200 cm, and an ultrasonic reading of 30 cm.

7. Safety Features

Ensuring the safety of both the robot and its environment was a priority in the design process. Multiple safety features were integrated into the system.

7.1. Emergency Stop Mechanism

A robust emergency stop system allows the robot to be immediately powered down in critical situations. This can be triggered either by a physical emergency stop button or remotely via a wireless command. This feature ensures quick intervention in the event of a malfunction or hazardous condition.

7.2. Overcurrent and Overvoltage Protection

To prevent electrical damage, the system incorporates dedicated protection circuits that monitor current and voltage levels in real time. If an overcurrent or overvoltage condition is detected, power is automatically cut off to safeguard the motors and microcontroller, preventing potential failures or overheating.

7.3. Fall Detection and Recovery

The robot is equipped with sensors that continuously monitor its tilt angle. If the robot tips beyond a predefined threshold, indicating a potential fall, the motors are automatically disabled to prevent further damage. Additionally, a self-recovery sequence can be initiated, allowing the robot to regain an upright position without human intervention.

8. Assembly

Using instruction provided by the manufacturer (see the user manual **tumbller**), the harware is assembled together as shown in following figures.

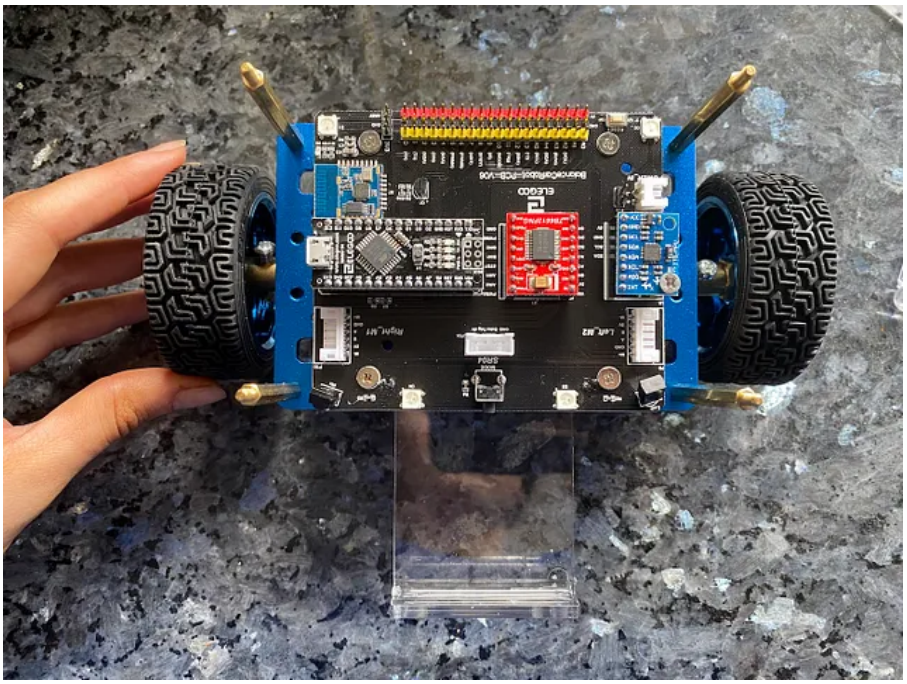


Figure 30: Assembling arduino nano, motor driver, mpu6050 sensor and motors on the robot base.

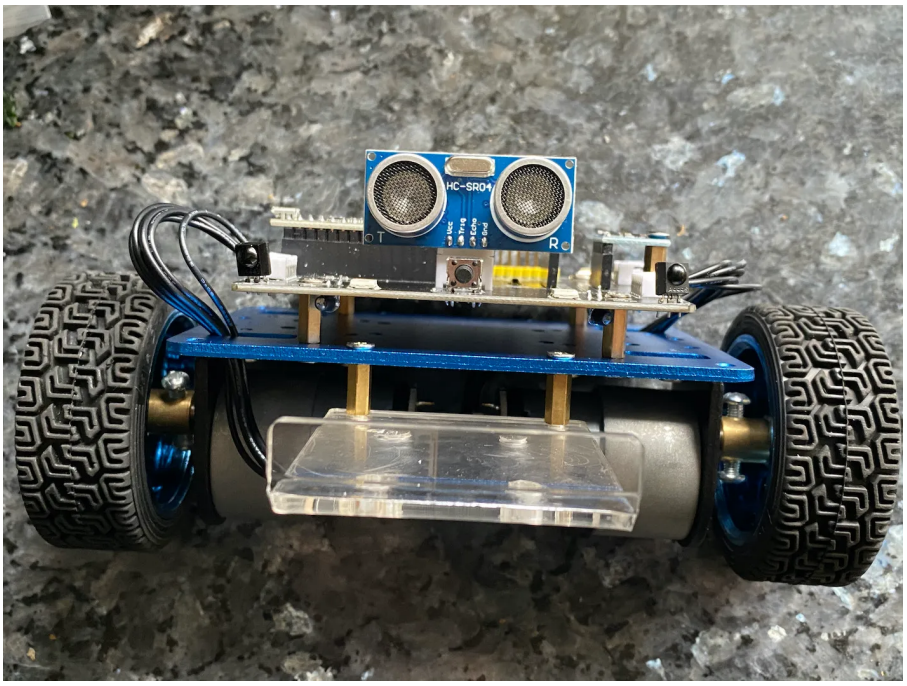


Figure 31: Attaching the ultrasonic sensor.

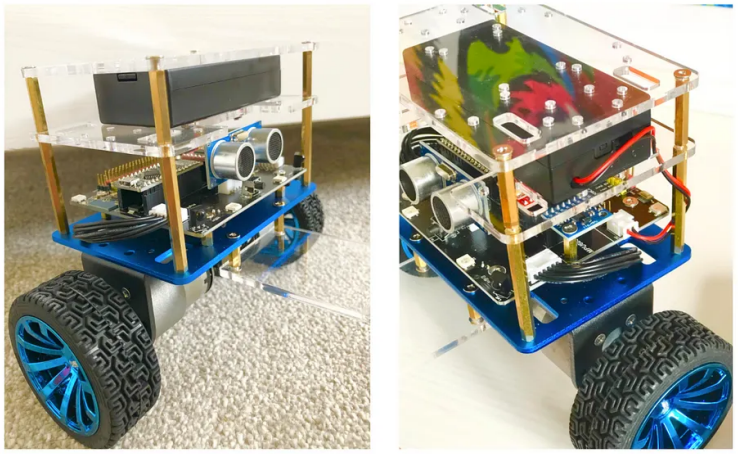


Figure 32: Final Assembly.

9. TerraRanger Multiflex

The TeraRanger Multiflex is a modular Time-of-Flight (ToF) (shown in Fig. 33) sensor array designed for robotics applications, including SLAM and maze-solving. The system includes eight ToF sensors, a Multiflex Hub, flex cables, and connectors, enabling flexible deployment **terra_mount**.

Figure 33: TeraRanger Multiflex circular mount **terra_mount**.

Key Features and Considerations

- **Compact and Modular Design**: Each sensor is enclosed in a polycarbonate cover and can be mounted using M3 screws.
- **Safe Laser Emission**: The sensors utilize a low-power laser emitter; optical modifications are not recommended.
- **Stable Power Requirements**: Operates at 5V ±0.25V with minimal ripple and noise.
- **Reliable Connectivity**: Utilizes a 7-pin Hirose DF13 connector for UART/I2C communication; designed for robust vibration-resistant connections.
- **Mounting Precautions**: Avoid exposure to heat, dust, and strong electromagnetic fields for optimal performance.

9.1. Connector and Data Interfaces

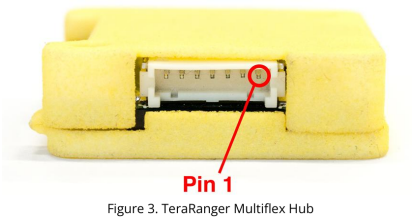


Figure 34: TeraRanger Multiflex Hub pinout **terra_mount**.

The **TeraRanger Multiflex Hub** connects to external equipment via a **7-pin Hirose DF13 connector** (female part: **DF13-7S-1.25C**). A compatible connector cable is included in the package. The pin configuration is as follows:

Pin	Function
7	GND
6	RX Serial In (UART)
5	TX Serial Out (UART)
4	Interrupt Pin
3	SDA (I2C)
2	SCL (I2C)
1	5 V Power Supply

Table 4: TeraRanger Multiflex Hub connector pinout

9.2. UART Data Interface (Default)

The **UART interface** operates at **3.3V output levels** (accepting inputs from **3.3V to 5V**). To connect the hub to a computer, a **USB-to-serial adapter** (e.g., FTDI breakout board) should be used. The default UART configuration is:

- **Baud Rate:** 115200 bit/s
- **Data Bits:** 8
- **Parity:** None
- **Stop Bits:** 1

9.3. I2C Data Interface

The **I2C interface** allows the hub to function as a slave device, connecting to an I2C master. The default **I2C base address** is **0x55**, and only one **TeraRanger Multiflex** can be used per I2C bus. The signal levels are **3.3V**, and the **maximum bus speed is 400 kHz**. Integrated **1.5k Ohm pull-up resistors** ensure proper bus communication.

9.4. USB Interface

A **micro-USB** port provides both **power and data communication**. On **Linux and macOS**, the hub appears as a **virtual COM port** without additional drivers. On **Windows**, a driver must be installed from:

`http://www.st.com/en/development-tools/stsw-stm32102.html`

After installation, reconnect the device to enable the COM port. The communication settings remain **115200-8N1**.

10. Maze Solving Alogrithms

Maze Solving Using Right-then-Left and Left-then-Right Navigation

Maze solving is a fundamental capability for autonomous mobile robots, requiring efficient navigation strategies to explore and determine the optimal path to an exit or target. The two-wheeled self-balancing robot employs two heuristic-based approaches for maze traversal: Right-then-Left (RTL) navigation and Left-then-Right (LTR) navigation. Both methods operate under the assumption that the maze consists of walls and corridors, where the robot continuously follows one side until reaching a dead-end or an unexplored path.

10.1. Right-Then-Left (RTL) Navigation

In the RTL approach, the robot prioritizes right turns whenever a choice is available. If no right turn is possible, it proceeds forward. If forward movement is blocked, it turns left. If no left turn is possible, it executes a U-turn. The algorithm follows these steps:

- If a right turn is available, turn right.
- If a right turn is not possible, move straight.
- If a dead-end is encountered, turn left.
- If all directions are blocked, perform a U-turn.

Dead-End Handling: The robot detects dead-ends using sensor feedback and reverses until a possible turn is detected. It switches to the left wall when navigating out of loops or returning from an incorrect path.

Cycle Detection and Backtracking: To prevent getting stuck in loops, the robot stores visited junctions and alters its strategy if it encounters a repeated state. If a previously visited path is identified, it backtracks and prioritizes alternative routes.

10.2. Left-Then-Right (LTR) Navigation

The LTR approach mirrors the RTL method but prioritizes left turns instead of right turns. This strategy is useful in mazes where a left-biased path leads to a shorter exit route. The algorithm follows these steps:

- If a left turn is available, turn left.
- If a left turn is not possible, move straight.
- If a dead-end is encountered, turn right.
- If all directions are blocked, perform a U-turn.

Dead-End Handling and Backtracking: Similar to RTL, the robot maintains a memory of visited locations to avoid infinite loops. If an already explored junction is reached again, the algorithm forces a deviation.

Cycle Detection and Backtracking: To prevent getting stuck in loops, the robot stores visited junctions and alters its strategy if it encounters a repeated state. If a previously visited path is identified, it backtracks and prioritizes alternative routes.

10.3. Algorithm Implementation

Both RTL and LTR methods can be implemented using a combination of wall-following, decision-tree logic, and sensor-based navigation. The robot uses:

Infrared or ultrasonic sensors to detect walls and available paths. Odometry and IMU data to maintain orientation and track previously visited locations. A finite-state machine (FSM) to switch between different navigation states.

A pseudo-code representation of both strategies is as follows:

```
void navigate_maze_RTL() {
        if (right_available()) {
                turn_right();
        } else if (forward_available()) {
                move_forward();
        } else if (left_available()) {
                turn_left();
        } else {
                u_turn();
        }
}
```

```
void navigate_maze_LTR() {
        if (left_available()) {
                turn_left();
        } else if (forward_available()) {
                move_forward();
        } else if (right_available()) {
                turn_right();
        } else {
                u_turn();
        }
}
```

11. Results

12. Future Work and Improvements

While the current implementation of the two-wheeled self-balancing robot demonstrates robust performance, several areas can be further improved to enhance functionality, autonomy, and user experience.

12.1. Advanced Control Strategies

Implementing more sophisticated control algorithms, such as adaptive controllers or reinforcement learning-based approaches, could improve stability and dynamic response under varying conditions. Model predictive control (MPC) could also be explored to optimize real-time decision-making.

12.2. Autonomous Navigation and Perception

Enhancing the robot's autonomy by integrating additional sensors such as LiDAR or depth cameras would enable precise environmental perception. Implementing SLAM (Simultaneous Localization and Mapping) algorithms would allow the robot to navigate and map complex environments independently.

12.3. Seamless Mobile Integration

Developing a dedicated mobile application with an intuitive user interface and real-time data visualization could significantly improve remote operation. Features such as wireless control, customizable movement presets, and telemetry monitoring would enhance usability.

12.4. Optimized Power Management

Exploring high-efficiency battery technologies, regenerative braking mechanisms, and smart power management systems could extend the robot's operational lifespan. Additionally, integrating energy-efficient motor drivers and optimizing power consumption could further enhance performance.

By addressing these areas, the robot's capabilities could be significantly expanded, paving the way for more advanced applications in research, automation, and real-world deployment.y management systems could extend the robot's operational time.

Appendices

A. Calculation of the system's transfer functions

To simplify the model, we assume that it can be represented as an inverted pendulum attached to a cart. Calculations about inverted pendulum attached to a cart is common and can be found on several websites on the internet, this will help ensure that these calculations are correct. The following calculations were inspired by MathWorks **matlab_inverted_pendulum**.

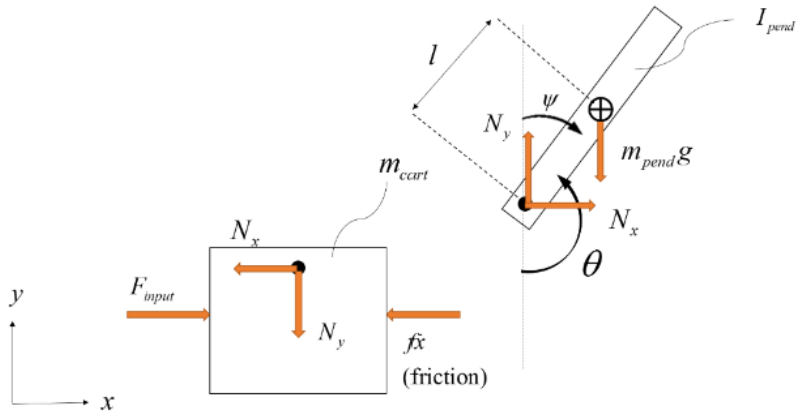


Figure 35: Exposure of the simplified system with cart and pendulum **10193276**.

A.1. Equation for the cart

The only equation needed from the cart is the one who sums the forces in the x direction

$$F_{input} = m_{cart}\ddot{x} + f\dot{x} + N_x \tag{42}$$

where F_{input} is an applied force, m_{cart} is the mass of the cart, f is a constant of friction and N_x is the contact force in the axis between cart and pendulum in x-direction (see Fig. 35).

A.2. Equation for the the pendulum

The sum of forces in the x direction on the pendulum is

$$N_x = m_{pend}\ddot{x} + m_{pend}l\ddot{\theta}\cos\theta - m_{pend}l\dot{\theta}^2\sin\theta \tag{43}$$

where m_{pend} is the mass and l is the distance to the center of mass. Variable θ is the angle between a vertical line and the pendulum. See Figure for details.

The sum of all forces perpendicular to the pendulum is

$$N_y\sin\theta + N_x\cos\theta = m_{pend}g\sin\theta - m_{pend}l\ddot{\theta} + m_{pend}\dot{\theta}^2\cos\theta \tag{44}$$

where N_y is the force in the y direction and g is the gravity constant.

Summarizing the torque acting on the center of the pendulum gives the following equation

$$-N_y l \sin\theta - N_x l \cos\theta = I_{pend}\ddot{\theta} \tag{45}$$

where I_{pend} is the moment of inertia of the pendulum.

=

A.3. Combining the equations

Insert equation (43) in (44) gives the following equation

$$F_{input} = (m_{cart} + m_{pend})\ddot{x} + f\dot{x} + m_{pend}l\ddot{\theta}\cos\theta - m_{pend}l\dot{\theta}^2\sin\theta \tag{46}$$

Combine equations (45) and (46)

$$(I_{pend} + m_{pend}l^2)\ddot{\theta} + m_{pend}gl\sin\theta = -m_{pend}l\ddot{x}\cos\theta \tag{47}$$

A.4. Linearising the equations

Equation (52) and (51) is necessary to get transfer functions for the position x and the angle deviation ψ. To compute the transfer functions the equations need to be linearized. A proper equilibrium point would be when the pendulum is in upright position. The angle will represent the deviation of the pendulum from the equilibrium. The following approximations for small deviations will be used in the nonlinear equations (46) and (47).

$$\cos\theta = \cos(\pi + \psi) \approx -1 \tag{48}$$

$$\sin\theta = \sin(\pi + \psi) \approx -\psi \tag{49}$$

$$\dot{\theta}^2 = \dot{\psi}^2 \approx 0 \tag{50}$$

Linearization with (48), (49) and (50) in (46) and (47) leads to the following approximated linear equations where F_{input} has been substituted for the more general control effort u_{input}.

$$(I_{pend} + m_{pend}l^2)\ddot{\psi} - m_{pend}gl\psi = m_{pend}l\ddot{x} \tag{51}$$

$$u_{input} = (m_{cart} + m_{pend})\ddot{x} + f\dot{x} - m_{pend}l\ddot{\psi} \tag{52}$$

A.5. Laplace transform

To obtain the transfer functions, equations (51) and (52) is transformed to the Laplace domain, the transformation is here denoted by upper case letters.

$$(I_{pend} + m_{pend}l^2)\Psi(s)s^2 - m_{pend}gl\Psi(s) = m_{pend}lX(s)s^2 \tag{53}$$

$$U_{input}(s) = (m_{cart} + m_{pend})X(s)s^2 + fX(s)s - m_{pend}l\Psi(s)s^2 \tag{54}$$

A transfer function is a relationship between a single input and a single output, therefore it is needed to solve $X(s)$ from equation (53).

$$X(s) = \left[\frac{I_{pend} + m_{pend}l^2}{m_{pend}l} - \frac{g}{s^2}\right]\Psi(s) \tag{55}$$

Substitute (55) into (54) gives

$$\begin{aligned} U_{input}(s) =& (m_{cart} + m_{pend})\left[\frac{I_{pend} + m_{pend}l^2}{m_{pend}l} - \frac{g}{s^2}\right]\Psi(s)s^2 \\ &+ f\left[\frac{I_{pend} + m_{pend}l^2}{m_{pend}l} - \frac{g}{s^2}\right]\Psi(s)s - m_{pend}l\Psi(s)s^2 \end{aligned} \tag{56}$$

If equation (56) is rearranged we get the transfer function $G_\Psi(s)$ as the relation between $\Psi(s)$ and $U_{input}(s)$ as seen in (57).

$$G_\Psi(s) = \frac{\Psi(s)}{U_{input}(s)} \tag{57}$$

$$\Psi(s) = \underbrace{\frac{\frac{m_{pend}l}{q}s}{s^3 + \frac{f(I_{pend}+m_{pend}l^2)}{q}s^2 - \frac{(m_{cart}+m_{pend})m_{pend}gl}{q}s - \frac{fm_{pend}gl}{q}}}_{G_\Psi(s)} U_{input}(s) \tag{58}$$

where

$$q = [(m_{cart} + m_{pend})(I_{pend} + m_{pend}l^2) - (m_{pend}l)^2] \tag{59}$$

The transfer function $G_x(s)$ that describes the cart position $X(s)$ looks as

$$X(s) = \underbrace{\frac{\frac{(I_{pend}+m_{pend}l^2)s^2 - gm_{pend}l}{q}}{s^4 + \frac{f(I_{pend}+m_{pend}l^2)}{q}s^3 - \frac{(m_{cart}+m_{pend})m_{pend}gl}{q}s^2 - \frac{fm_{pend}gl}{q}s}}_{G_x(s)} U_{input}(s) \tag{60}$$

A.6. State Space Modelling

It is possible to present the system in state space form. The matrix form is

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = A \begin{bmatrix} x \\ \dot{x} \\ \psi \\ \dot{\psi} \end{bmatrix} + Bu_{input} \tag{61}$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{-(I_{pend}+m_{pend}l^2)f}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} & \frac{m_{pend}^2 gl^2}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{-m_{pend}lf}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} & \frac{m_{pend}gl(m_{cart}+m_{pend})}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} & 0 \end{bmatrix} \tag{62}$$

$$B = \begin{bmatrix} 0 \\ \frac{I_{pend}+m_{pend}l^2}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} \\ 0 \\ \frac{m_{pend}l}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} \end{bmatrix} \tag{63}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{64}$$

It is possible to present the system in state space form. The matrix form, ignoring position and velocity states, is:

$$\begin{bmatrix} \dot{\psi} \\ \ddot{\psi} \end{bmatrix} = A \begin{bmatrix} \psi \\ \dot{\psi} \end{bmatrix} + Bu_{input} \tag{65}$$

where

$$A = \begin{bmatrix} 0 & 1 \\ \frac{m_{pend}gl(m_{cart}+m_{pend})}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} & 0 \end{bmatrix} \tag{66}$$

$$B = \begin{bmatrix} 0 \\ \frac{m_{pend}l}{I_{pend}(m_{cart}+m_{pend})+m_{cart}m_{pend}l^2} \end{bmatrix} \tag{67}$$

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{68}$$

B. Calculation for Kalman Filter

B.1. Initialization

Defining an initialization according to:

$$P(0) = \mathbf{E}(\Delta \underline{x}(0) \Delta \underline{x}^T(0)) \tag{69}$$

where $P(0)$ is the initial covariance matrix, presenting the expected uncertainty in the initial state estimate. It is calculated based on the expected error in the initial state.

B.2. Optimal Kalman Gain

We can find optimal Kalman gain matrix $\underline{K}(t)$ as following:

$$\underline{K}(t) = P(t)\underline{C}^T(t)R^{-1} \tag{70}$$

It determines much the state estimate should be adjusted based on the measurement residual. It balances the uncertainty in the state estimate and the measurement noise.

B.3. Time-Discrete Kalman Filter

The time-discrete Kalman filter equations are expressed as:

$$\begin{aligned} \underline{x}_k &= \underline{A}\underline{x}_{k-1} + \underline{B}\underline{u}_k + \underline{d}_{k-1} \\ \underline{y}_k &= \underline{C}\underline{x}_k + \underline{n}_k \end{aligned} \tag{71}$$

where,

- $\underline{x}_k$: This is the state vector at time step k.
- $\underline{B}$: This is the control input matrix, which relates the control inputs $\underline{u}_k$ to the state.
- $\underline{y}_k$: This is the measurement vector at time step k.
- $\underline{d}_{k-1}$: This represents process noise at the previous time step.
- $\underline{n}_k$: This represents measurement noise at time step k.

B.4. Estimated states:

The estimated states are given by:

$$\begin{aligned} \underline{\hat{x}}_k &= \underline{A}\ \underline{\hat{x}}_{k-1} + \underline{B}\ \underline{u}_k + \underline{d}_{k-1} \\ \underline{\hat{y}}_k &= \underline{C}\ \underline{\hat{x}}_k + \underline{n}_k \end{aligned} \tag{72}$$

B.5. Discrete State Equation

The state-space representation of the system is given by:

$$\begin{aligned} \underline{\dot{x}}(t) &= \underline{A}.\underline{x}(t) + \underline{B}.\underline{u}(t) \\ \underline{y}(t) &= \underline{C}.\underline{x}(t) + \underline{D}.\underline{u}(t) \end{aligned} \tag{73}$$

where the components of the State-Space Representation are,

- **State Vector $\underline{\mathbf{x}}(\mathbf{t})$**: This vector encapsulates the internal state of the system at time t. It this case, it is defined as $\mathbf{x}_k = \begin{bmatrix} \text{angle} \\ \text{q\_bias} \end{bmatrix}$. Where angle represents the measured angle of the system, while q_bias denotes the bias of the gyroscope.
- **State Transition Matrix $\underline{\mathbf{A}}$**: This matrix describes how the state evolves over time. It is defined as: $\mathbf{A}_k = \begin{bmatrix} 1 & -dt \\ 0 & 1 \end{bmatrix}$ The first row indicates that the angle is updated based on its previous value and the time step dt, while the second row shows that the gyroscope bias remains constant in this model.

- **Control Input Matrix $\underline{\mathbf{B}}$**: This matrix relates the control inputs to the state. In this case, it is defined as: $\mathbf{B}_k = 0$. This indicates that there are no direct control inputs affecting the state in this model.
- **Measurement Matrix $\underline{\mathbf{C}}$**: This matrix maps the state vector to the measurement output. It is defined as: $\mathbf{C}_k = \begin{bmatrix} 1 & 0 \end{bmatrix}$. This means that the measurement output directly reflects the angle, with no contribution from the gyroscope bias.
- **Feedforward Matrix $\underline{\mathbf{D}}$**: This matrix relates the control input directly to the measurement output. In this case, it is defined as: $\mathbf{D}_k = 0$. This indicates that there is no direct influence of the control input on the measurement output.

B.6. Measurement Noise Covariance Matrix R

The measurement noise variance for the angle sensor is defined as:

$$\mathbf{R}_k = R_{angle} \tag{74}$$

B.7. Process/System Noise Covariance Matrix Q

The process noise covariance matrix is given by:

$$\mathbf{Q} = \begin{bmatrix} Q_{\text{angle}} & 0 \\ 0 & Q_{\text{gyro bias}} \end{bmatrix} * \Delta t \tag{75}$$

B.8. State Covariance Matrix P

The state covariance matrix is represented as:

$$\mathbf{P}_k = \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \tag{76}$$

- P_{00} represents the uncertainty in the angle estimate.
- P_{11} represents the uncertainty in the gyroscope bias estimate.
- $P_{01} = P_{10}$ represent the covariance between the angle and gyroscope bias.

B.9. Kalman Gain K:

The Kalman gain is defined as:

$$\mathbf{K}_k = \begin{bmatrix} K_0 \\ K_1 \end{bmatrix} \tag{77}$$

B.10. Estimated states

The estimated states are updated as follows:

$$\theta_{measured} = \theta_{measured} + (\omega_{measured} - \omega_{bias}) * \Delta t \tag{78}$$

B.11. Error Calculation

The error in the angle estimate is calculated as:

$$\theta_{error} = \theta_{measured} - \theta_{desired} \tag{79}$$

B.12. Time Update (prediction)

The time update for the state covariance matrix is given by:

$$\mathbf{P}_k = \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \tag{80}$$

The matrix P reflects the uncertainties in the angle estimate (P_{00}), the gyroscope bias (P_{11}), and the cross-covariance terms (P_{01}, P_{10}).

B.13. Initialization

The initial state covariance matrix is defined as:

$$\mathbf{P}_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{81}$$

The prediction step for the covariance matrix is:

$$\mathbf{P}_k = \mathbf{A}\mathbf{P}_{k-1}\mathbf{A}^{\mathbf{T}} + \mathbf{Q} \tag{82}$$

This expands to:

$$\mathbf{P}_k = \begin{bmatrix} P_{00}^- + [(Q_{\text{angle}} - P_{01}^- - P_{10}^-) * \Delta t] & P_{01}^- - (P_{11}^- * \Delta t) \\ P_{10}^- - (P_{11}^- * \Delta t) & P_{11}^- + (Q_{\text{gyro bias}} * \Delta t) \end{bmatrix} \tag{83}$$

B.14. Kalman Gain Calculation

The Kalman gain $\underline{K}_k$ is computed as:

$$\begin{aligned} \underline{K}_k &= \underline{P}_k^- \ \underline{C}^T (\underline{C} \ \underline{P}_k^- \ \underline{C}^T + \underline{R})^{-1} \\ &= \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + R_{angle} \right)^{-1} \\ &= \begin{bmatrix} P_{00} \\ P_{10} \end{bmatrix} (P_{00} + R_{angle})^{-1} \end{aligned} \tag{84}$$

$$\mathbf{K}_k = \begin{bmatrix} \frac{P_{00}}{P_{00}+R_{angle}} \\ \frac{P_{10}}{P_{00}+R_{angle}} \end{bmatrix}$$

B.15. Measurement Update (Correction)

The measurement update for the state covariance matrix is given by:

$$\begin{aligned}\underline{P}_k = &\ (\underline{I} - \underline{K}_k\ \underline{C})\ \underline{P}_k^- \\ = &\ \left(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} - \begin{bmatrix}K_0\\ K_1\end{bmatrix}\begin{bmatrix}1 & 0\end{bmatrix}\right)\begin{bmatrix}P_{00} & P_{01}\\ P_{10} & P_{11}\end{bmatrix} = \begin{bmatrix}1-K_0 & 0\\ -K_1 & 1\end{bmatrix}\begin{bmatrix}P_{00} & P_{01}\\ P_{10} & P_{11}\end{bmatrix} \\ = &\ \begin{bmatrix}P_{00} - K_0 \cdot P_{00} & P_{01} - K_0 \cdot P_{01}\\ P_{10} - K_1 \cdot P_{00} & P_{11} - K_1 \cdot P_{01}\end{bmatrix}\end{aligned} \tag{85}$$

C. Third Appendix