Self Balancing Robot

Muhammad Haris Mujeeb

February 10, 2025

Abstract

This report focuses on the design of a Two-wheeled Self-Balancing Robot, which embodies the classic inverted pendulum problem. Using a Kalman filter for MPU6050 sensor data fusion and implementing PID controllers. This project aims to develop a robust two-wheeled self-balancing robot that can be observed and controlled remotely. The coding for the Arduino Nano has been executed using PlatformIO, with the complete project available on this GitHub repository.

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.

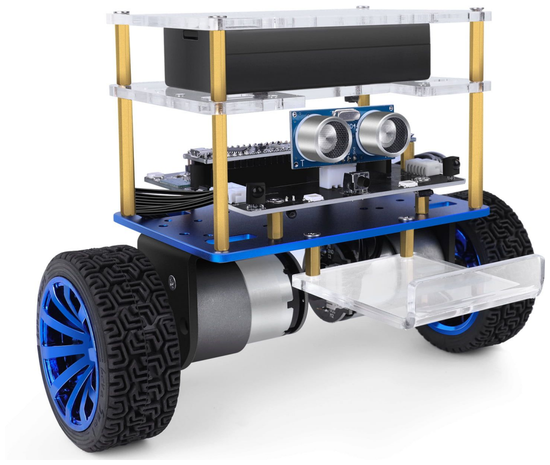


Figure 1: ELEGOO Tumbler which was used for this project [1].

1.2 Project Objectives

1.3 Scope of Work

2 Literature Review

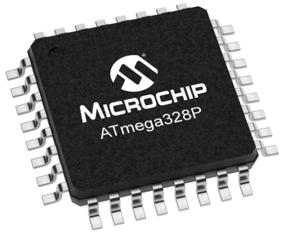
2.1 Overall Architecture

Block diagram of the system Explanation of the control flow

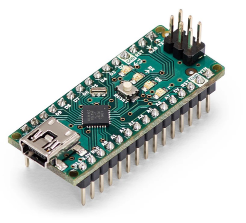
2.2 Hardware Components

2.2.1 Microcontroller

The ATmega328P (shown in Fig. 2a) 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 [10], the ATmega328P provides ample resources for this projects application.



(a) ATMEGA328P ANR [10].



(b) Arduino Nano [11].

Figure 2: ATMEGA328P MCU (a) and Arduino Nano development board (b).

The ATMEGA328P is also used in the Arduino Nano (shown in Fig. 2b), a widely adopted development board known for its low cost and open-source ecosystem [11]. 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.

2.2.2 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. 3). 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.

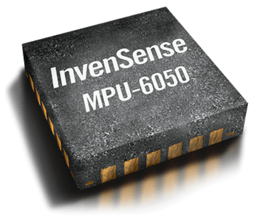


Figure 3: MPU-6050 [2].

2.2.3 Motor Driver

The TB6612FNG dual motor driver (shown in Fig. 4) 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. 5 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 [4].

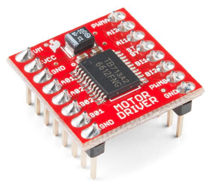


Figure 4: TB6612FNG [3].

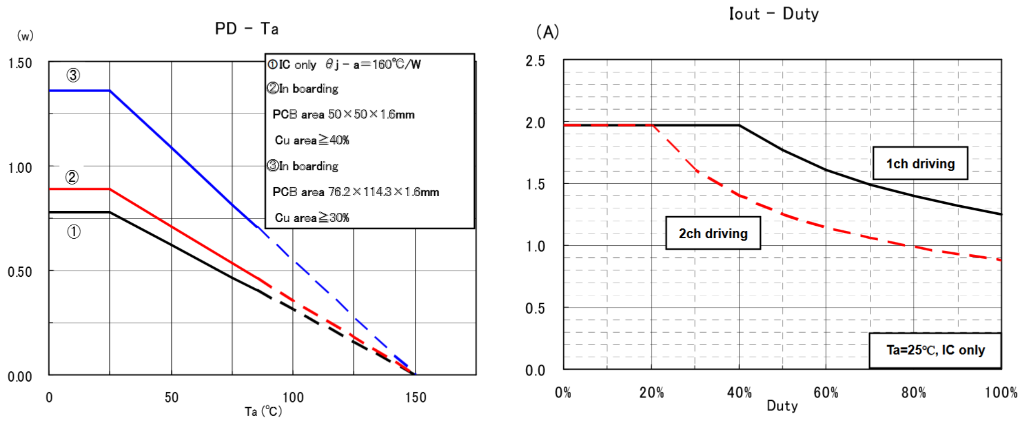


Figure 5: Target characteristics for TB6612FNG [4].

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. 6) is used. It also enhances motor control by improving signal stability (similar to what is shown in Fig. 7).

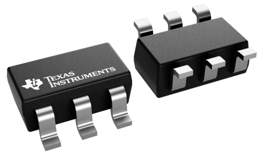


Figure 6: SN74LVC2G14 [5].

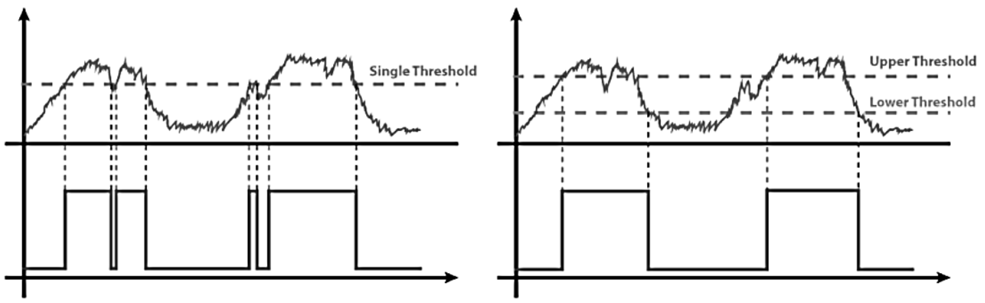


Figure 7: Schmitt trigger output without hysteresis (left) and with hysteresis (right) [6].

2.2.4 Drive Motors

The drive system employs NNHYTECH GA37 520 DC motors (37mm diameter, 12V, 360 RPM) equipped with Hall effect encoders (shown in Fig. 8) [7]. 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.

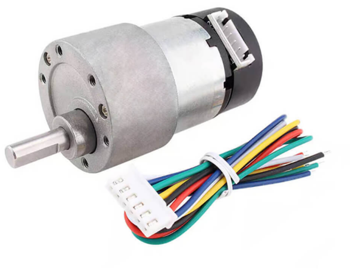


Figure 8: Similar construction motors were used in this project [7].

2.2.5 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.

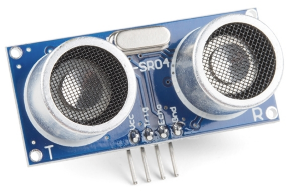
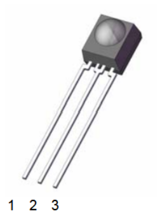


Figure 9: Ultrasonic distance sensor by Sparkfun electronics [8].

2.2.6 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 [12].



(b) IRM-56384 [13].

Figure 10: LED-Emitter (a) and Infrared LED Reciever (b).

2.3 Bluetooth

The BT16 4.2 Bluetooth transparent transmission module is a cutting-edge component that leverages the advanced capabilities of the Airoha ABI 602 single chip, which is compliant with the Bluetooth 4.2 BLE standard. This module facilitates GATT-based Bluetooth data transmission through its integrated data transparent transmission service, ensuring efficient and reliable communication. One of the key features of the BT16 module is its support for serial command mode, which enables seamless interaction between the external microcontroller unit (MCU) and the Bluetooth module. This functionality allows users to configure various parameters and exert control over the module via serial port commands. Users can modify essential settings such as the UUID, change the Bluetooth device name, and manage Bluetooth disconnection processes.

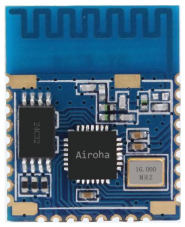


Figure 11: Ultrasonic distance sensor by Sparkfun electronics [9].

2.3.1 Power Supply considerations

This custom-designed battery box provides a portable and rechargeable power solution for the Elegoo robot [14]. Housing two 18650 LiPo batteries (likely connected in series), it delivers a regulated output to power the robot's various components. The integrated Battery Management System (BMS) ensures safe operation by providing overcharge, over-discharge, over-current, and short-circuit protection. A power switch allows for complete disconnection, while a USB charging port and status indicator LED simplify recharging and monitoring.

Figure 12: ELEGOO battery pack with charger.

2.4 Mathematical Modelling

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

3 Software Implementation

3.1 Firmware Overview

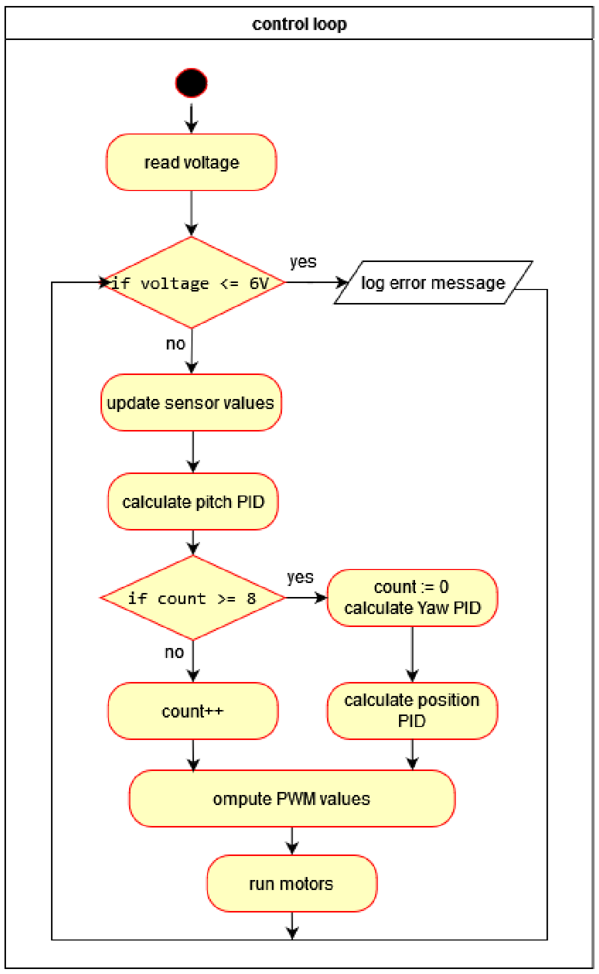


Figure 13: A simplified block diagram of the control loop.

4 Sensor Fusion for Self-Balancing Robot using MPU6050

4.1 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

4.2 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.

4.2.1 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!");
	}

}
```

4.2.2 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.

4.3 Complementary Filter

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

4.3.1 Mathematical Model

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

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

Similarly, using the angular velocity from the gyroscope ω, the sampling time interval Δt and previous angle value $\theta_{previous}$ can be used to calculate tilt angle using following equation:

$$\theta_{gyro} = \theta_{previous} + \omega.\Delta t$$

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

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

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

4.3.2 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;
}
```

4.3.3 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;
                angle_roll = angle_roll_acc;
                set_gyro_angles = true;
        }

        angle_pitch_output = angle_pitch_output * 0.9 + angle_pitch * 0.1;
        angle_roll_output = angle_roll_output * 0.9 + angle_roll * 0.1;

}
```

4.3.4 Results

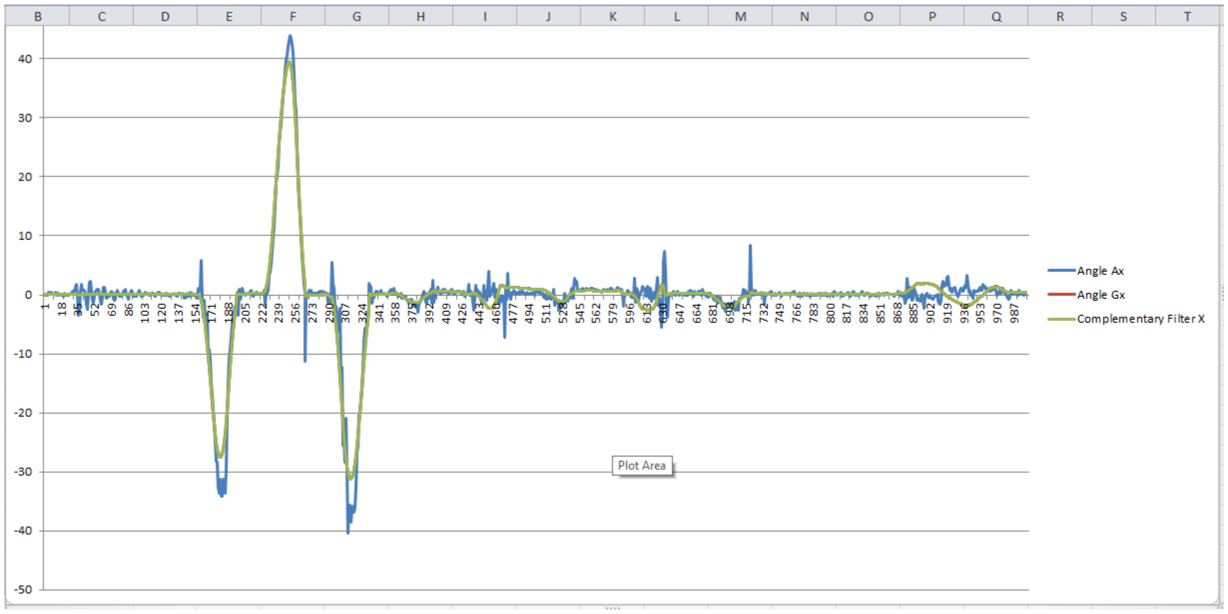


Figure 14: Pitch angle caluclated using a complementary filter with $\omega = 0.9$.

4.4 Extended Kalman Filter

The Kalman filter 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. But Kalman Filter assumes linearity in both the process and measurement models. To solve this issue we will use Extended Kalman Filter.

The Extended Kalman Filter (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.

4.4.1 General State Equation

For non-linear system, with Stochastic disturbances:

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

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.

4.4.2 State Estimation

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

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

To compute the Kalman gain $\underline{K}$ the system must be linearized around the current state estimate. The Jacobain matrices are defined as follows:

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

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.

4.4.3 Covraince Matrix

To find a solution for the covariance matrix, we utilize Differential Riccati Equation (DRE):

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

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.

4.4.4 Initialization

Defining an initialization according to:

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

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.

4.4.5 Optimal Kalman Gain

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

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

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.

4.4.6 Time-Discrete Kalman Filter

The time-discrete Kalman filter equations are expressed as:

$$\begin{aligned} \underline{x}_k &= \underline{Ax}_{k-1} + \underline{Bu}_k + \underline{d}_{k-1} \\ \underline{y}_k &= \underline{Cx}_k + \underline{n}_k \end{aligned}$$

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.

4.4.7 Estimated states:

The estimated states are given by:

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

4.4.8 State Equation for Self-Balancing Robot

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

$$\dot{\underline{x}}(t) = \underline{A}.\underline{x}(t) + \underline{B}.\underline{u}(t)$$
$$\underline{y}(t) = \underline{C}.\underline{x}(t) + \underline{D}.\underline{u}(t)$$

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.

4.4.9 Measurement Noise Covariance Matrix R

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

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

4.4.10 Process/System Noise Covariance Matrix Q

The process noise covariance matrix is given by:

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

4.4.11 State Covariance Matrix P

The state covariance matrix is represented as:

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

- 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.

4.4.12 Kalman Gain K:

The Kalman gain is defined as:

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

4.4.13 Estimated states

The estimated states are updated as follows:

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

4.4.14 Error Calculation

The error in the angle estimate is calculated as:

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

4.4.15 Time Update (prediction)

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

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

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}).

4.4.16 Initialization

The initial state covariance matrix is defined as:

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

The prediction step for the covariance matrix is:

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

This expands to:

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

4.4.17 Kalman Gain Calculation

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

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

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

4.4.18 Measurement Update (Correction)

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

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

4.5 Software Implementation of Extended Kalman Filter

```
#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;
        }
};
```

5 Cascaded PID Control Loop

The algorithm utilizes cascade Proportional–Integral–Derivative (PID) control loop to maintain balance and control the robot's movement in real time. The balancing is achieved by continuously monitoring and adjusting its state using sensor feedback. The key sensor inputs include the robot's pitch angle, gyro data and motor's encoder values, which provide information on the robot's orientation, angular velocities and position.The algorithm utilizes cascade PID control loops to maintain balance and control the robot's movement in real time. The approach includes computing control outputs for pitch, yaw, and position periodically, which are then used to adjust the motor speeds by sending corresponding pulse-width-modulation (PWM) signals to the motor driver (see Fig. 15).

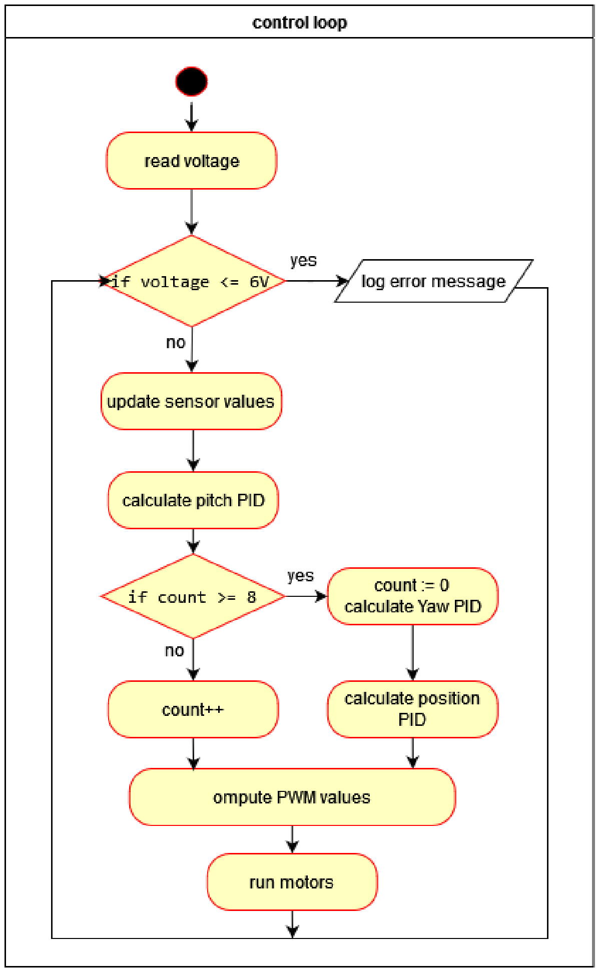


Figure 15: A simplified block diagram of the control loop.

As the pitch angle is more important for system stability, therefore it is updated more frequently (e.g. updated every cycle), while the PID output values for the yaw angle and position control are updated longer duration of time (e.g. after every 8th cycle).

5.1 Basic PID Structure

The PID controller output can be expressed mathematically as:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \tag{2}$$

Where:

- $u(t)$ is the control signal

- $e(t)$ is the error signal
- K_p is the proportional gain
- K_i is the integral gain
- K_d is the derivative gain

5.2 Position Control Loop

The outer loop manages the robot's position:

$$\theta_{desired} = K_{px}(x_{desired} - x_{measured}) + K_{dx}\frac{d}{dt}(x_{desired} - x_{measured}) \tag{3}$$

6 Implementation Considerations

6.1 Discrete Time Implementation

For digital implementation, the PID controller is discretized:

$$u[n] = K_p e[n] + K_i T_s \sum_{k=0}^{n} e[k] + K_d \frac{e[n] - e[n-1]}{T_s} \tag{4}$$

Where T_s is the sampling period.

7 Tuning Methodology

For tuning a general combination of Ziegler-Nichols method and practical tuning guidelines were used.

7.1 Ziegler-Nichols Method

The Ziegler-Nichols tuning method follows these steps:

1. Set K_i and K_d to zero
2. Increase K_p until system oscillates with period T_u
3. Record the ultimate gain K_u
4. Calculate parameters:

$$\begin{aligned} K_p &= 0.6K_u \\ T_i &= 0.5T_u \\ T_d &= 0.125T_u \end{aligned}$$

7.2 Practical Tuning Guidelines

The general tuning guidelines are as follows:

- Begin with small K_p $(= 10)$
- Add derivative term $(K_d = 0.1K_p)$
- Fine-tune until stable

7.3 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.

$$\tau_{\theta,pid} = K_{p\theta}(\theta_{desired} - \theta_{measured}) + K_{d\theta}\frac{d}{dt}(\theta_{desired} - \theta_{measured}) \tag{5}$$

Below is its code implementation:

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

- **Proportional (P):** The proportional term is based on the current error (the pitch angle deviation from the desired value). A higher proportional gain causes the robot to respond more aggressively to larger deviations.
- **Derivative (D):** The derivative term anticipates future errors by considering the rate of change of the error (pitch angular velocity deviation from desired value). It provides a damping effect, which helps to reduce overshooting and oscillations.
- **Integral (I):** Because the system is inherently unstable when at desired upright position (steady state error can never be zero) thus adding the integral term is unnecessary.

The output of the pitch PD controller (pitch_pid_output) is then used to adjust the robot's motor speeds to counteract any tilting or imbalance.

7.4 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. The yaw control adjusts the motor speeds to achieve the desired rotational velocity, ensuring the robot maintains a stable heading.

$$\tau_{\phi,pid} = K_{d\phi}\frac{d}{dt}(\phi_{desired} - \phi_{measured}) \tag{6}$$

Below is its code implementation:

```
inline void runYawControl(){
    yaw_pid_output = kd_turn * gyro_z;
}
```

Similar to pitch control, the yaw controller follows the PID principles:

- **Proportional (P):** In order to calculate yaw angle, a high computational overhead is required, while yaw angle is not critical for balancing thus the proportional term is ignored.
- **Integral (I):** As the yaw angle is not calculated, therefore the integral term cannot be calculated as well.
- **Derivative (D):** The derivative term mitigates any excessive rate of change in yaw, preventing oscillations in the robot's rotation.

The yaw PID output (yaw_pid_output) is then combined with the pitch and position control outputs to compute the final motor PWM values.

7.5 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.

$$\tau_{x,pid} = K_{px}(x_{desired} - x_{measured}) + K_{dx}\frac{d}{dt}(x_{desired} - x_{measured}) \tag{7}$$

Below is its code implementation:

```
inline void runPositionControl(){
    double encoder_speed = (left_encoder_position +
        right_encoder_position) * 0.5
    robot_position += encoder_speed;
    robot_position = constrain(robot_position, -3000, 3000);
    position_pid_output = - ki_position * robot_position - kd_position
         * encoder_speed;
}
```

To prevent integral windup:

$$u_{limited} = \begin{cases} 3000 & \text{if } u > 3000 \\ u & \text{if } -3000 \leq u \leq 3000 \\ -3000 & \text{if } u < -3000 \end{cases} \tag{8}$$

- **Proportional (P):** The proportional term helps correct any immediate error in position by adjusting the motor speeds in response to the current position error.
- **Integral (I):** The integral term addresses any accumulated error in position that may arise due to external factors like friction or uneven terrain.
- **Derivative (D):** Because the position control does not require fast settling time, thus derivative term can be ignored in favour of faster calculation time.

The position control output (position_pid_output) is also factored into the final PWM values for motor control.

7.6 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:

$$\tau_{left,motor} = \tau_{\theta,pid} - \tau_{\phi,pid} - \tau_{x,pid} \tag{9}$$

$$\tau_{right,motor} = \tau_{\theta,pid} + \tau_{\phi,pid} - \tau_{x,pid} \tag{10}$$

Below is its code implementation:

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

These calculated PWM values are then sent to the motor drivers to adjust the robot's movement and balance.

7.7 Conclusion

The use of PID controllers for pitch, yaw, and position control enables the robot to maintain balance and navigate effectively. 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.

7.8 Front Obstacle Detection

The Ultrasonic distance measuring sensor is used to detect the obstacle in front of the robot.

7.9 Ultrasonic Working Principle

The sensor consists of a **transmitter** and a **receiver**:

- The transmitter emits an ultrasonic pulse (40 kHz).
- The pulse reflects off an obstacle and is received by the receiver.
- The time difference between transmission and reception is used to compute the distance using the formula:

$$d = \frac{t \times v}{2} \tag{11}$$

where:

- d is the measured distance,
- t is the time delay (in microseconds),
- v is the speed of sound (approximately 343 m/s at room temperature).

7.9.1 Ultrasonic Implementation

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

1. A short **trigger pulse** is sent to the `TRIG` pin.
2. The sensor responds with a high signal on the `ECHO` pin.
3. The duration of the `ECHO` signal is measured to determine the distance.

7.9.2 Code for Distance Measurement

Based on the datasheet [8], 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;
        constexpr float SPEED_OF_SOUND_HALVED = (340.29 * 100.0) / (2 * 1000 * 1000);
```

Here, the speed of sound is converted to cm/µs, and it is divided by 2 to account for the round-trip travel time of the ultrasonic pulse. The following C++ function is used to initiate distance measurement using the HC-SR04 ultrasonic sensor:

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

                attachPinChangeInterrupt(ECHO_PIN,
                    HandleUltrasonicMeasurementInterrupt, RISING);

                digitalWrite(TRIG_PIN, LOW);
                digitalWrite(TRIG_PIN, HIGH);
                digitalWrite(TRIG_PIN, LOW);
        }
```

- 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.

7.9.3 Interrupt Service Routine

The following function handles the interrupt to measure the distance:

```
	void HandleUltrasonicMeasurementInterrupt() {
		if (usonicMeasureFlag == SEND) {
			usonicMeasurePrevTime = micros();
			attachPinChangeInterrupt(ECHO_PIN,
			    HandleUltrasonicMeasurementInterrupt, FALLING);
			usonicMeasureFlag = RECEIVED;
		} else if (usonicMeasureFlag == RECEIVED) {
			usonicDistanceValue = (uint8_t)((micros() -
			    usonicMeasurePrevTime) * SPEED_OF_SOUND_HALVED);
			usonicMeasureFlag = IDLE;
		}
	}
```

- When the echo signal first rises (RISING edge), the timestamp is recorded using `micros()`.
- The interrupt is then reattached to detect the falling edge.
- When the falling edge is detected, the elapsed time is computed.
- The time is converted to distance using the speed of sound formula.
- Finally, the system resets for the next measurement.

7.9.4 Infrared Sensing Implementation

The IR LED transmits a modulated 38kHz infrared signal. If an obstacle is present, the signal reflects and is received by the IRM-56384, which demodulates the signal and provides a digital output. The following code is used to control and process 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);
			irLeftPulseCount = 0;
			irLeftCountTime = millis();
		}
	}

	void ProcessRightIRSensor(){
		if (millis() - irRightCountTime > IR_COUNT_DELAY_MS) {
			UpdateSlidingWindow((irRightPulseCount >= 3), irRightHistory,
			    irRightIndex, irRightRunningCount);
			irRightIsObstacle = (irRightRunningCount >= 5);
			irRightPulseCount = 0;
```

```
                              irRightCountTime = millis();
                    }
          }
```

8 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.

8.1 Bluetooth Module Integration

The **BT16 Bluetooth UART Module** was chosen due to its compatibility with microcontrollers and its ability to provide stable serial communication over Bluetooth. The module was interfaced with the microcontroller using UART communication, ensuring efficient data transmission and reception.

8.2 Communication Protocol

A custom communication protocol was developed to manage the exchange of control commands and telemetry data. Key features of the protocol included:

- **Command Transmission:** The robot could receive commands for movement control, mode switching, and parameter adjustments from a remote device.
- **Telemetry Feedback:** The robot transmitted real-time data such as tilt angle, battery status, and motor performance back to the controlling device.
- **Error Handling:** Mechanisms were implemented to detect and recover from communication errors, ensuring reliable operation even in noisy environments.

8.3 User Interface for Remote Control

The Bluetooth communication enabled the development of a user-friendly interface for remote control. This interface could be a mobile application or a desktop-based GUI, allowing users to interact with the robot intuitively. Features included:

- **Joystick Control:** For real-time maneuvering of the robot.
- **Status Monitoring:** Display of critical telemetry data.
- **Parameter Tuning:** On-the-fly adjustment of control parameters for performance optimization.

8.4 Testing and Performance

Extensive testing was conducted to ensure the reliability and responsiveness of the Bluetooth communication system. The tests focused on:

- **Range:** Assessing the effective communication distance.
- **Latency:** Measuring the delay between command transmission and robot response.
- **Stability:** Ensuring consistent performance in various environments.

The integration of the Bluetooth module significantly enhanced the robot's usability, providing a flexible and responsive remote control solution that contributed to the overall functionality and user experience of the system.

9 Simulation and Testing

Simulation and testing played a crucial role in the development and refinement of the two-wheeled self-balancing robot. By leveraging computational tools and environments, we were able to model the robot's behavior under various conditions, validate control algorithms, and predict system performance before real-world implementation.

9.1 Simulation Tools and Environment

For simulating the dynamics and control strategies of the robot, we utilized both **Python** and **MATLAB**. These platforms provided robust frameworks for numerical computation, visualization, and algorithm development.

- **Python:** Utilized libraries such as *NumPy*, *SciPy*, and *Matplotlib* for numerical simulations and visualizations. *Control* and *SymPy* libraries were used to model the control systems and analyze the system's response.
- **MATLAB:** Employed for more advanced control design and simulation, including the use of Simulink for block diagram modeling and simulation of dynamic systems. MATLAB's built-in tools facilitated the tuning and optimization of control parameters.

9.2 Control Algorithm Testing

We implemented and tested various control algorithms to ensure the stability and performance of the robot. Key focus was given to the **Complementary Filter** and **Kalman Filter** for sensor fusion and state estimation.

- **Complementary Filter:** Simulations helped fine-tune the filter coefficients to effectively merge accelerometer and gyroscope data, providing a stable estimate of the robot's tilt angle.
- **Kalman Filter:** The filter was tested for its ability to reduce noise and provide accurate state estimation. MATLAB simulations were crucial in visualizing the filter's performance and adjusting the covariance matrices.

9.3 Performance Metrics

Several metrics were used to evaluate the performance of the control algorithms in the simulation environment:

- **Stability:** Assessed by analyzing the system's ability to return to an upright position after disturbances.
- **Response Time:** Measured how quickly the control system could react to changes in tilt and external perturbations.
- **Noise Rejection:** Evaluated the effectiveness of the filters in minimizing sensor noise and maintaining accurate state estimation.

9.4 Real-World Validation

Post-simulation, the control algorithms were transferred to the physical robot for real-world testing. The outcomes from the simulations provided a strong baseline, and discrepancies observed during physical trials were fed back into the simulation models for further refinement. This iterative process ensured a robust and reliable control system.

Overall, the combination of Python and MATLAB simulations significantly accelerated the development process and provided valuable insights into the robot's dynamic behavior and control performance.

10 Simulation and Testing

Simulation and testing played a crucial role in the development and refinement of the two-wheeled self-balancing robot. By leveraging computational tools and environments, we were able to model the robot's behavior under various conditions, validate control algorithms, and predict system performance before real-world implementation.

10.1 Simulation Tools and Environment

For simulating the dynamics and control strategies of the robot, we utilized both **Python** and **MATLAB**. These platforms provided robust frameworks for numerical computation, visualization, and algorithm development.

- **Python:** Utilized libraries such as *NumPy*, *SciPy*, and *Matplotlib* for numerical simulations and visualizations. *Control* and *SymPy* libraries were used to model the control systems and analyze the system's response.
- **MATLAB:** Employed for more advanced control design and simulation, including the use of Simulink for block diagram modeling and simulation of dynamic systems. MATLAB's built-in tools facilitated the tuning and optimization of control parameters.

10.2 Control Algorithm Testing

We implemented and tested various control algorithms to ensure the stability and performance of the robot. Key focus was given to the **Complementary Filter** and **Kalman Filter** for sensor fusion and state estimation.

- **Complementary Filter:** Simulations helped fine-tune the filter coefficients to effectively merge accelerometer and gyroscope data, providing a stable estimate of the robot's tilt angle.
- **Kalman Filter:** The filter was tested for its ability to reduce noise and provide accurate state estimation. MATLAB simulations were crucial in visualizing the filter's performance and adjusting the covariance matrices.

10.3 Performance Metrics

Several metrics were used to evaluate the performance of the control algorithms in the simulation environment:

- **Stability:** Assessed by analyzing the system's ability to return to an upright position after disturbances.
- **Response Time:** Measured how quickly the control system could react to changes in tilt and external perturbations.
- **Noise Rejection:** Evaluated the effectiveness of the filters in minimizing sensor noise and maintaining accurate state estimation.

10.4 Real-World Validation

Post-simulation, the control algorithms were transferred to the physical robot for real-world testing. The outcomes from the simulations provided a strong baseline, and discrepancies observed during physical trials were fed back into the simulation models for further refinement. This iterative process ensured a robust and reliable control system.

To thoroughly test the robustness of the system, we introduced controlled disturbances and varying surface conditions in the real-world environment. This helped identify edge cases and stress points that were not evident in the simulation phase. By iteratively refining the control algorithms based on these findings, we were able to enhance the overall stability and performance of the robot.

Overall, the combination of Python and MATLAB simulations significantly accelerated the development process and provided valuable insights into the robot's dynamic behavior and control performance.

11 Power Management

Efficient power management was critical for ensuring the longevity and reliability of the two-wheeled self-balancing robot. We employed **7.4V Lithium-Ion battery packs** to provide a stable power supply.

11.1 Battery Selection and Integration

The 7.4V Lithium-Ion battery packs were chosen for their high energy density, lightweight properties, and reliable performance. These batteries were integrated with a voltage regulator circuit to ensure consistent voltage levels to the microcontroller and motor drivers.

11.2 Power Monitoring

To monitor the battery status in real-time, voltage sensors were used to track battery levels. This data was integrated into the telemetry feedback system, allowing the user to receive alerts when the battery level was low.

11.3 Charging and Safety

A dedicated charging circuit with overcharge protection was implemented to enhance battery life and safety. Thermal sensors were also included to monitor battery temperature during operation and charging.

12 User Interface

The user interface was designed to provide intuitive control and comprehensive feedback to the operator. Both hardware and software interfaces were developed to enhance the user experience.

12.1 Graphical User Interface (GUI)

A GUI was created for both desktop and mobile platforms, leveraging Bluetooth connectivity for communication. The interface included:

- **Real-Time Monitoring:** Display of sensor data such as tilt angle, speed, and battery status.
- **Control Inputs:** Virtual joystick and buttons for maneuvering the robot.
- **Parameter Adjustment:** Sliders and input fields for tuning control parameters.

12.2 Physical Interface

For manual control, physical buttons and switches were integrated on the robot's chassis. These controls allowed for quick access to basic functions like power on/off and mode switching.

13 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.

13.1 Emergency Stop Mechanism

An emergency stop function was implemented, allowing the robot to be immediately powered down via a physical button or remote command in case of malfunction or hazardous situations.

13.2 Overcurrent and Overvoltage Protection

Electronic protection circuits were included to safeguard the motors and microcontroller from electrical faults. These circuits automatically disconnected power in the event of overcurrent or overvoltage conditions.

13.3 Fall Detection and Recovery

Sensors were programmed to detect when the robot tipped beyond a recoverable angle. In such cases, the motors were disabled to prevent damage, and a recovery sequence could be initiated to return the robot to an upright position.

14 Future Work and Improvements

While the current implementation of the two-wheeled self-balancing robot achieved significant milestones, several areas for future improvement were identified.

14.1 Enhanced Control Algorithms

Exploring advanced control techniques such as adaptive control and machine learning-based approaches could further enhance the robot's stability and performance.

14.2 Autonomous Navigation

Integrating additional sensors like LiDAR and implementing SLAM (Simultaneous Localization and Mapping) algorithms would enable the robot to navigate autonomously in complex environments.

14.3 Mobile App Development

Developing a dedicated mobile application with enhanced features and a more user-friendly interface would improve the overall user experience.

14.4 Extended Battery Life

Investigating alternative power sources or more efficient battery management systems could extend the robot's operational time.

15 Real-World Applications

The technology and design principles behind the two-wheeled self-balancing robot have numerous real-world applications.

15.1 Personal Transportation

Self-balancing technology is commonly used in personal transport devices like hoverboards and Segways, offering convenient and efficient mobility solutions.

15.2 Robotics Research and Education

This project serves as a valuable educational tool for understanding control systems, sensor fusion, and robotics, making it suitable for academic and research purposes.

15.3 Delivery and Logistics

Self-balancing robots can be adapted for use in delivery services, particularly in navigating urban environments where maneuverability is crucial.

15.4 Assistive Technology

In healthcare, self-balancing robots can assist individuals with mobility impairments, providing stability and support in daily activities.

16 Challenges and Limitations

Throughout the development process, several challenges and limitations were encountered.

16.1 Sensor Noise and Drift

Despite the use of Kalman and Complementary filters, sensor noise and drift remained a challenge, affecting the accuracy of state estimation.

16.2 Hardware Constraints

Limitations in processing power and memory on the microcontrollers constrained the complexity of algorithms that could be implemented.

16.3 Battery Life

While the 7.4V Lithium-Ion battery packs provided adequate power, extended operation times required frequent recharging, highlighting the need for more efficient power management.

16.4 Communication Interference

Bluetooth communication was susceptible to interference in noisy environments, occasionally affecting the reliability of remote control.

Addressing these challenges will be key in future iterations to enhance the robustness and functionality of the system.

References

[1] E. Official. Elegoo. Accessed: 2024-11-23. [Online]. Available: https://eu.elegoo.com/de

[2] InvenSense. Mpu-6050. Accessed: 2024-11-23. [Online]. Available: https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/

[3] B. T. M. Shop. Sparkfun motor treiber, dual tb6612fng, mit headern. Accessed: 2025-02-09. [Online]. Available: https://www.berrybase.de/sparkfun-motor-treiber-dual-tb6612fng-mit-headern

[4] Toshiba. Tb6612fng. Accessed: 2024-11-23. [Online]. Available: https://toshiba.semicon-storage.com/us/semiconductor/product/motor-driver-ics/brushed-dc-motor-driver-ics/detail.TB6612FNG.html

[5] T. Instruments. Sn74lvc2g14. Accessed: 2024-11-23. [Online]. Available: https://www.ti.com/product/SN74LVC2G14

[6] C. D. Systems. Schmitt trigger hysteresis provides noise-free switching and output. Accessed: 2025-02-09. [Online]. Available: https://resources.pcb.cadence.com/blog/2021-schmitt-trigger-hysteresis-provides-noise-free-switching-and-output

[7] microdcmotors. 37mm 6v 12v 24v dc gear motor w/encoder model nfp-gm37-520-en. Accessed: 2024-11-23. [Online]. Available: https://microdcmotors.com/product/6v-12v-geared-dc-electric-motor-with-encoder-nfp-jgb37-520-en

[8] sparkfun electronics. Ultrasonic distance sensor - 5v (hc-sr04). Accessed: 2025-02-08. [Online]. Available: https://www.sparkfun.com/ultrasonic-distance-sensor-hc-sr04.html

[9] epow0.org. Elegoo bt16 bluetooth uart module. Accessed: 2025-02-09. [Online]. Available: https://epow0.org/~amki/car_kit/Datasheet/ELEGOO%20BT16%20Bluetooth%20UART%20Module.pdf

[10] microchip. Atmega328p. Accessed: 2024-11-23. [Online]. Available: https://www.microchip.com/en-us/product/ATmega328p

[11] A. S. P. IVA. Arduino nano. Accessed: 2025-02-09. [Online]. Available: https://store.arduino.cc/en-de/products/arduino-nano

[12] DigiKey. Everlight electronics co ltd ir204c-a. Accessed: 2025-02-08. [Online]. Available: https://www.digikey.de/de/products/detail/everlight-electronics-co-ltd/IR204C-A/2675568

[13] X.-O. E. Services. Irm-56384 everlight. Accessed: 2025-02-08. [Online]. Available: https://www.xonelec.com/mpn/everlight/irm56384

[14] E. Official. Elegoo smart robot car kit v4.0 tutorial. Accessed: 2025-02-09. [Online]. Available: https://www.elegoo.com/blogs/arduino-projects/elegoo-smart-robot-car-kit-v4-0-tutorial