Self Balancing Robot

Muhammad Haris Mujeeb

January 31, 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.

1 Introduction

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].

2 Hardware

2.1 ATmega328P

The ATmega328P is a popular microcontroller from Microchip Technology, widely used in embedded systems and electronics projects. It belongs to the AVR family of microcontrollers and is renowned for its versatility, ease of use, and robust performance (shown in Fig. 2). Its most significant benefits is its abundance of resources and community support, particularly due to its compatibility with the Arduino platform. With a 16 MHz clock speed, 32 KB of flash memory, 2 KB of SRAM, and 1 KB of EEPROM, the ATmega328P provides ample resources for this projects application.

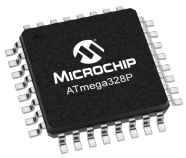

Figure 2: ATMEGA328P ANR [2].

2.2 MPU6050

The MPU6050 is a widely used 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 [3].

2.3 TB6612FNG H-bridge

The TB6612FNG H-bridge [4] together with SN74LVC2G14 schmitt-trigger [5] provide a powerful and reliable motor control solution for self-balancing robots. This combination allows for precise motor operation, enabling the robot to adapt quickly to its environment and maintain stability effectively.

2.4 Geared DC Motors

37mm High Torque 12V Gearbox DC Motors with Speed Encoders were used in this project. Multiple manufacurer are producing these kind of motors (shown in Fig. 4). Motors from NHYTech were in this case.

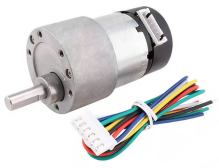

Figure 4: Similar construction motors were used in this project [6].

3 Mathematical Modelling

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

4 Extended Kalman Filter

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 f is a non-linear function of $\underline{x}(t)$, $\underline{u}(t)$.

4.1 State Estimation

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

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

Kalman Gain is calculated after the system is linearized around a working point or a trajectory.

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

where $\underline{A}(t)$ and $\underline{C}(t)$ correspond to the Jacobian matrix.

4.1.1 covraince matrix

We use Differential Riccati Equation (DRE) for finding a solution for covraince matrix:

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

Defining an initialization according to:

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

The optimal Kalman gain matrix is:

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

4.1.2 Time-Discrete Kalman Filter

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

4.1.3 Estimated states:

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

4.1.4 State-space From

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

where,

$$\mathbf{x}_k = \begin{bmatrix}\text{angle}\\ \text{q\_bias}\end{bmatrix}, \qquad \mathbf{A}_k = \begin{bmatrix}1 & -dt\\ 0 & 1\end{bmatrix}$$

$$\mathbf{B}_k = 0, \qquad \mathbf{C}_k = \begin{bmatrix}1 & 0\end{bmatrix}, \qquad \mathbf{D}_k = 0$$

This means you are directly measuring the angle, but you have no direct measurement of the gyroscope bias.

4.1.5 Measurement Noise Covariance Matrix R:

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

the measurement noise variance for the angle sensor.

Process/System Noise Covariance Matrix **Q**:

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

4.1.6 State Covariance Matrix P:

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

- 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.

4.1.7 Kalman Gain K:

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

4.1.8 Estimated states:

$$angle = angle + (gyro_m - gyro_{bias}) * \Delta t$$

4.1.9 Error Calculation:

$$angle_{err} = angle_m - angle$$

4.1.10 Time Update (prediction):

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

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}).

4.1.11 Initialization,

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

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

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

4.1.12 Kalman gain

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

$$= \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + R_{angle} \right)^{-1}$$

$$= \begin{bmatrix} P_{00} \\ P_{10} \end{bmatrix} (P_{00} + R_{angle})^{-1}$$

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

4.1.13 Measurement Update (correction):

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

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. 5).

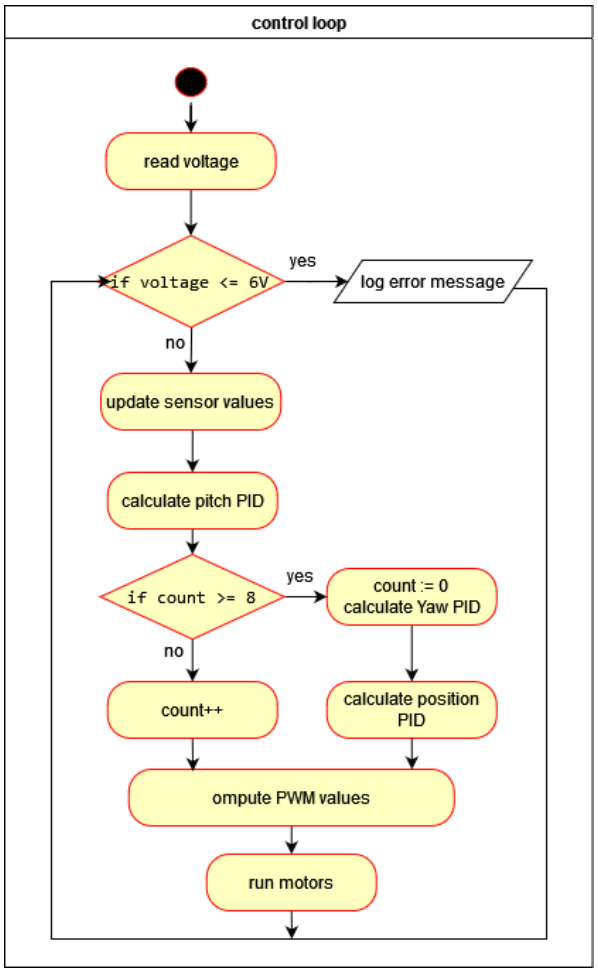

Figure 5: A simplified block diagram of the control loop.

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:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \tag{1}$$

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:

$$\theta_{desired} = K_{px}(x_{desired} - x_{measured}) + K_{dx}\frac{d}{dt}(x_{desired} - x_{measured}) \tag{2}$$

6 Implementation Considerations

6.1 Discrete Time Implementation

For digital implementation, the PID controller is discretized:

$$u[n] = K_p e[n] + K_i T_s \sum_{k=0}^{n} e[k] + K_d \frac{e[n] - e[n-1]}{T_s} \tag{3}$$

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.

$$\tau_{\theta,pid} = K_{p\theta}(\theta_{desired} - \theta_{measured}) + K_{d\theta}\frac{d}{dt}(\theta_{desired} - \theta_{measured}) \tag{4}$$

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.

$$\tau_{\phi,pid} = K_{d\phi}\frac{d}{dt}(\phi_{desired} - \phi_{measured}) \tag{5}$$

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.

$$\tau_{x,pid} = K_{px}(x_{desired} - x_{measured}) + K_{dx}\frac{d}{dt}(x_{desired} - x_{measured}) \tag{6}$$

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:

$$u_{limited} = \begin{cases} 3000 & \text{if } u > 3000 \\ u & \text{if } -3000 \leq u \leq 3000 \\ -3000 & \text{if } u < -3000 \end{cases} \tag{7}$$

- **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:

$$\tau_{left,motor} = \tau_{\theta,pid} - \tau_{\phi,pid} - \tau_{x,pid} \tag{8}$$

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

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.

References

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

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

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

[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] 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