Balancing a Robot: A Control Systems Project

Qufei Zhang

October 16, 2024

1 Introduction

In this project, I explored the fundamentals of control systems by simulating the behavior of a self-balancing robot. This robot mimics real-world systems like **inverted pendulum robots**, which require precise control to maintain stability. The focus of the project was on implementing a **PD controller** and utilizing an **Extended Kalman Filter (EKF)** for state estimation.

Through this process, I developed a deeper understanding of control algorithms, the importance of feedback systems, and how real-world noise complicates state estimation. Below, I'll walk you through the logic behind the control system, the algorithms used, and my personal learnings.

2 Problem Overview

The main challenge was to balance an unstable robot, which essentially works like an **inverted pendulum** mounted on a wheel. The robot must remain upright by adjusting the wheel's motion based on the robot's tilt angle and its rate of change. This requires continuous state feedback, achieved using a **control loop**. For this project, I implemented a **PID controller** (Proportional, Integral, Derivative) to manage the balancing and an **Extended Kalman Filter** to estimate the robot's state from noisy measurements.

3 High-Level Overview of the Balancing Robot Algorithm

The goal of the algorithm is to maintain the balance of an unstable, two-wheeled robot by continuously adjusting the wheel's motion based on the robot's tilt and position. The algorithm integrates three major components: **state estimation (Extended Kalman Filter)**, **system dynamics (Equations of Motion)**, and **control (PD Control)**. Together, these components ensure that the robot remains balanced while following a desired trajectory.

3.1 State Estimation (Extended Kalman Filter)

Since real-world sensors are noisy, the algorithm relies on an **Extended Kalman Filter (EKF)** to estimate the robot's true state (i.e., tilt angle and angular velocity) from noisy sensor data. The EKF plays a key role in providing accurate feedback to the controller by filtering out the noise from the accelerometer and gyroscope readings. This allows the system to predict and update the robot's state.

- **Prediction Step**: The EKF predicts the robot's next state using a linearized system model and the previous state estimate. This step uses the robot's equations of motion to forecast where the robot will be.
- **Update Step**: The EKF updates the state estimate based on the new sensor readings. It compares the predicted sensor measurements to the actual measurements and adjusts the state estimate accordingly using the Kalman gain.

3.2 System Dynamics (Equations of Motion)

Once the state is estimated, the algorithm needs to compute how the robot's body and wheels will respond to forces and torques over time. The system's **Equations of Motion (EOM)** describe the dynamics of the robot, capturing the relationship between applied torque and how the robot moves.

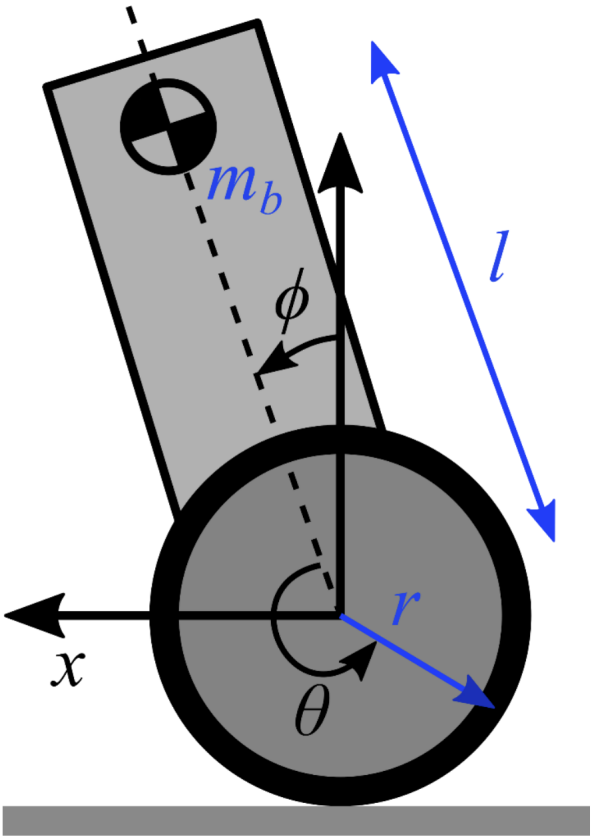

Figure 1: Diagram of the balancing robot system showing key parameters.

- The robot's dynamics are represented in terms of the angular accelerations of the wheel and the body (tilt). These accelerations are computed by solving a matrix equation $Aq = B$, which comes from the **Lagrangian formulation** of the system. The Lagrangian combines the robot's kinetic and potential energy, allowing us to derive the dynamic behavior of the robot.
- Solving the equations of motion gives us the angular accelerations of the wheel and the body, which are critical for simulating the robot's motion in each time step.

3.3 Control (PD Controller)

The controller's job is to apply torque to the wheels in such a way that the robot maintains balance and moves according to a desired trajectory. This is done using a **Proportional-Derivative (PD) Controller** in a cascade structure.

- **Position Control (Outer Loop)**: The outer loop calculates the desired tilt angle (body angle) based on the position error (difference between the robot's actual position and desired position). This is achieved through a PD controller that adjusts the body angle to correct the position.
- **Tilt Control (Inner Loop)**: The inner loop adjusts the actual tilt angle of the robot using another PD controller. It takes the desired tilt from the outer loop and controls the wheel torque to keep the robot upright and aligned with the desired body angle.

3.4 Simulation Integration (ODE Solver)

The simulation integrates the **Equations of Motion** over time using an ODE solver (e.g., `ode45` in MATLAB), which allows the algorithm to simulate the continuous evolution of the robot's state over time.

- For each time step:

1. The `dynamics` function is called to compute the state derivatives (e.g., velocity and acceleration) based on the current state and control input.
2. The ODE solver integrates these derivatives to update the robot's position, tilt, and velocities.
3. Noise is added to the sensor readings to simulate real-world imperfections, and the EKF filters the noise to estimate the true state.
4. The **PD controller** computes the required torque to balance the robot based on the current state estimate.
5. The process repeats for the next time step until the entire simulation time is covered.

3.5 Performance Evaluation

At the end of the simulation, the performance is evaluated by comparing the robot's actual trajectory to the desired trajectory. Metrics like position error and tilt oscillations are computed to assess how well the controller performed in keeping the robot balanced and on track.

- **Position Error**: This metric checks how closely the robot followed the desired trajectory.
- **Tilt Stability**: This measures how much the robot's tilt fluctuated over time, indicating the effectiveness of the tilt control loop.

3.6 Summary

- **EKF**: Estimates the robot's true state (angle and velocity) from noisy sensor data.
- **EOM**: Describes how the robot's body and wheels move in response to the applied torque.
- **PD Controller**: Controls the robot's motion by calculating the appropriate wheel torque to maintain balance and follow the desired trajectory.
- **ODE Solver**: Simulates the continuous evolution of the robot's motion over time.
- **Performance Evaluation**: Assesses the controller's ability to maintain balance and follow the trajectory.

This combination of state estimation, dynamics, and control ensures the robot remains balanced while accurately following a desired path.

4 Extended Kalman Filter (EKF)

In real-world applications, sensor measurements are often noisy, making it difficult to obtain accurate data on the robot's angle and velocity. To address this, I implemented an Extended Kalman Filter (EKF), which estimates the robot's true state (i.e., tilt angle and angular velocity) from noisy sensor readings.

The EKF works in two key steps:

- **Prediction Step**: Predict the next state of the system based on the current state and control input.
- **Update Step**: Update the state estimate based on new sensor data using the Kalman gain.

By leveraging the EKF, the controller receives more accurate feedback even when the sensor data is noisy, leading to better stability in the robot.

4.1 State Vector and Measurements

We have a 2D state vector:

$$\boldsymbol{x} = \begin{bmatrix} \phi \\ \dot{\phi} \end{bmatrix}$$

where:

- ϕ is the roll angle (in radians),
- $\dot{\phi}$ is the roll angular velocity (in radians/second).

Our noisy measurements come from an IMU (Inertial Measurement Unit), and the measurement vector is:

$$\boldsymbol{z} = \begin{bmatrix} a_y \\ a_z \\ \dot{\phi} \end{bmatrix}$$

where:

- a_y and a_z are the accelerometer readings along the y and z axes, respectively (in units of g),
- $\dot{\phi}$ is the gyroscope measurement of the angular velocity.

4.2 EKF Prediction Step

4.2.1 State Prediction

The predicted state is computed using a linear state transition model:

$$\boldsymbol{x}_{k|k-1} = \boldsymbol{A}\boldsymbol{x}_{k-1}$$

with:

$$\boldsymbol{A} = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix}$$

Here, dt is the time step between measurements.

4.2.2 Covariance Prediction

The predicted covariance of the state estimation error is updated as follows:

$$\boldsymbol{P}_{k|k-1} = \boldsymbol{A}_{k-1}\boldsymbol{P}_{k-1}\boldsymbol{A}_{k-1}^{\top} + \boldsymbol{Q}$$

where:

- $\boldsymbol{P}_{k|k-1}$ is the predicted covariance matrix,
- $\boldsymbol{Q}$ is the process noise covariance matrix.

4.3 Measurement Update Step

4.3.1 Measurement Model

The predicted measurement $\boldsymbol{h}(\boldsymbol{x}_{k|k-1})$ maps the predicted state into the measurement space:

$$\boldsymbol{h}(\boldsymbol{x}_{k|k-1}) = \begin{bmatrix} \sin(\phi_{k|k-1}) \\ \cos(\phi_{k|k-1}) \\ \dot{\phi}_{k|k-1} \end{bmatrix}$$

4.3.2 Kalman Gain

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

$$\boldsymbol{K}_k = \boldsymbol{P}_{k|k-1}\boldsymbol{H}_k^{\top}\left(\boldsymbol{H}_k\boldsymbol{P}_{k|k-1}\boldsymbol{H}_k^{\top} + \boldsymbol{R}\right)^{-1}$$

where:

- $\boldsymbol{H}_k$ is the Jacobian of the measurement model,
- $\boldsymbol{R}$ is the measurement noise covariance matrix.

4.3.3 State Update

The innovation or measurement residual is:

$$\boldsymbol{y}_k = \boldsymbol{z}_k - \boldsymbol{h}(\boldsymbol{x}_{k|k-1})$$

The updated state estimate is:

$$\boldsymbol{x}_k = \boldsymbol{x}_{k|k-1} + \boldsymbol{K}_k \boldsymbol{y}_k$$

4.3.4 Covariance Update

The updated covariance matrix is:

$$\boldsymbol{P}_k = (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k)\boldsymbol{P}_{k|k-1}$$

This step reduces the uncertainty in the state estimate after incorporating the new sensor data.

5 Tuning Parameters

The performance of the *Extended Kalman Filter (EKF)* largely depends on the proper tuning of three key matrices: the **process noise covariance** Q, the **measurement noise covariance** R, and the **initial error covariance** P. These matrices control how much weight the EKF places on the model's predictions versus the noisy sensor measurements, and how much uncertainty is attributed to each component.

5.1 Initial Error Covariance (P)

The matrix P represents the uncertainty in the initial state estimate. It is a critical part of the EKF initialization process and dictates how much confidence the EKF has in the starting estimate.

- **Tuning** P:
  - Start with reasonable values based on how confident you are in your initial state estimate.
  - Commonly, P is set to a diagonal matrix with large values (e.g., 10, 100) to reflect uncertainty in the initial state.
  - As the EKF progresses, P will shrink as the filter gains confidence in the state estimates.

Example:

$$P = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$$

A higher initial value in P reflects more initial uncertainty in the state estimate, allowing the filter to rely more on sensor data during the first few steps. Reduce this value if you have a better idea of the initial state.

5.2 Process Noise Covariance (Q)

The matrix Q represents the uncertainty in the system model. It captures variations and uncertainties in the process dynamics, such as unmodeled forces, friction, or external disturbances. It tells the filter how much to trust the model's predictions.

- **Tuning** Q:
  - Start with small values for Q and increase them if the system needs to respond more quickly to dynamic changes.
  - If Q is too small, the EKF may be too slow in responding to actual state changes.
  - If Q is too large, the EKF may overreact to small changes and become too noisy.

Example:

$$Q = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.01 \end{bmatrix}$$

For a system with high dynamics or uncertainties, you might start with larger values like 0.1 or higher.

5.3 Measurement Noise Covariance (R)

The matrix R represents the uncertainty in the sensor measurements. For example, if your sensors are noisy, you'd increase the values in R to reflect that the filter should place less trust in them and more trust in the predicted state from the process model.

- **Tuning** R:
  - Start with values that reflect the expected noise in the sensors. For example, if you expect a variance of 0.01 in the accelerometer readings, set the corresponding element in R to 0.01.
  - Increasing R makes the filter rely more on the model than the measurements, while lowering R increases the reliance on sensor data.

Example:

$$R = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.01 \end{bmatrix}$$

If you know your sensors have higher noise, you might use larger values like 0.1, depending on the specific characteristics of the sensors.

6 Equation of Motion - Derivation

The dynamics of a self-balancing robot can be described using **Lagrangian mechanics**. The **Lagrangian** is a function that combines the system's **kinetic energy** and **potential energy**, allowing us to derive the system's **equations of motion**.

The Lagrangian function L is defined as:

$$L = T - V$$

where:

- T is the **kinetic energy**, and
- V is the **potential energy**.

In our robot, we define two generalized coordinates:

$$\mathbf{q} = \begin{bmatrix} \theta \\ \phi \end{bmatrix}$$

where:

- θ is the wheel/body angle (rotation of the wheel),
- ϕ is the body roll angle (tilt of the robot).

Using the Lagrangian mechanics approach, we can calculate the **equations of motion** by applying the **Euler-Lagrange equations**, which state:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} = \tau$$

This equation helps us derive the robot's dynamics based on the interaction between the wheel and the robot's body. Now, we'll break down how this applies to the robot system.

6.1 Kinetic Energy and Potential Energy

- The **kinetic energy** T consists of both the wheel's and the body's movement. For the body, it is given by:

$$T = \frac{1}{2} m_b \dot{p}^T \dot{p} + \frac{1}{2} I_b \dot{\phi}^2$$

where:

- m_b is the mass of the body,
- $\dot{p}$ is the velocity of the body's center of mass (COM),
- I_b is the body's moment of inertia around the COM,
- $\dot{\phi}$ is the angular velocity of the body.

- The **potential energy** V is due to gravity:

$$V = m_b g l \cos(\phi)$$

where:

- g is the acceleration due to gravity,
- l is the distance from the wheel axis to the body's COM.

6.2 Euler-Lagrange Equation

The next step is to derive the system's equations of motion using the **Euler-Lagrange equation**. For each generalized coordinate q, the equation is given by:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}}\right) - \frac{\partial L}{\partial q} = \tau$$

where τ represents any external torques applied to the system (e.g., torque applied by the motor to the wheel).

This equation expresses the dynamics of the system by relating the rate of change of kinetic and potential energy to the external forces and torques acting on the system.

6.3 The Equation of Motion in Matrix Form

By applying the Euler-Lagrange equation to the generalized coordinates θ and ϕ, we can derive the equations of motion for the system. These equations describe how the angles θ and ϕ evolve over time in response to external torques.

Through this process, we arrive at the final form of the equations of motion:

$$A\mathbf{q}_{dd} = B$$

where:

- A is the **mass matrix**, which captures the inertia of the system and the coupling between the wheel and the body,
- $\mathbf{q}_{dd} = \begin{bmatrix} \ddot{\theta} \\ \ddot{\phi} \end{bmatrix}$ represents the angular accelerations of the wheel and body,
- B is the vector of external forces and torques acting on the system.

This matrix form of the equations of motion is beneficial because it provides a compact representation of the system's dynamics, which can be efficiently solved using numerical methods. The matrix A accounts for the mass, geometry, and inertia of the robot, while the vector B captures the effects of applied torques and gravitational forces.

6.4 Mass Matrix (A) and Forces (B)

The equations of motion can be described using two matrices:

1. The **Mass Matrix** A:

$$A = \begin{bmatrix} mr^2 & mr^2 + mrl\cos(\phi) \\ mr^2 + mrl\cos(\phi) & 2mrl\cos(\phi) + I + ml^2 + mr^2 \end{bmatrix}$$

2. The **Forces/Torques** (B):

$$B = \begin{bmatrix} mrl\dot{\phi}^2\sin(\phi) + u \\ ml\sin(\phi)(r\dot{\phi}^2 + g) \end{bmatrix}$$

6.5 Solving for Angular Accelerations

Finally, we solve for the angular accelerations by using:

$$A\mathbf{q}_{dd} = B$$

This equation gives us the angular accelerations of both the wheel ($\ddot{\theta}$) and the body ($\ddot{\phi}$).

In conclusion, the `eom` function implements the derived equations of motion using the Lagrangian approach to compute the angular accelerations needed for controlling the balancing robot.

6.6 Benefits of Using the Lagrangian Approach

The Lagrangian method is particularly useful in mechanical systems like this because it:

- Simplifies the derivation of the equations of motion, especially for systems with multiple degrees of freedom,
- Automatically accounts for the interaction between different parts of the system (e.g., the coupling between the wheel and the body),
- Can be easily extended to more complex systems by adjusting the kinetic and potential energy terms.

In this project, the final form $A\mathbf{q}_{dd} = B$ allows us to compute the angular accelerations $\ddot{\theta}$ and $\ddot{\phi}$, which are essential for controlling the robot's balance and motion.

7 Cascade Control Strategy

In the context of controlling our self-balancing robot, we employ a **cascade control strategy** using a **PD controller** (Proportional-Derivative) to manage the two interdependent control loops: the inner loop that stabilizes the tilt (or body angle) and the outer loop that controls the robot's position. Cascade control is effective when dealing with systems where the control of one variable (like body angle) indirectly affects another (like position).

7.1 Key Concepts of Cascade Control

- **Body Angle as Input to Control Position:** The **inner loop** controls the tilt of the robot (body angle, ϕ) using a **PD controller** to maintain balance. The **outer loop** controls the **position** of the robot by managing the **body angle**. In other words, the position is adjusted by tilting the body in a way that moves the wheel to the desired position.
- **Relationship Between θ and ϕ:** Recall that the robot's position x is related to both the wheel angle θ and the body angle ϕ:

  $$x = (\theta + \phi)r$$

  where r is the wheel radius.
- **Cascading the Control:** The **outer loop** computes the desired tilt angle ϕ_{desired} based on the position error, and then the **inner loop** adjusts the actual tilt ϕ to match ϕ_{desired}. This structure separates the position control from the tilt control, allowing each to be handled independently by its own control law.
- **PD Control for Position and Tilt:** For the **outer loop**, the desired tilt angle ϕ_{desired} is computed using a **PD control** law:

  $$\phi_{\text{desired}} = k_{p_x}(x_{\text{desired}} - x) + k_{d_x}(0 - \dot{x})$$

  where:
  - x_{desired} is the desired position, typically a trajectory the robot needs to follow.
  - k_{p_x} and k_{d_x} are the proportional and derivative gains for the position control loop.
  - x is the current position, and $\dot{x}$ is the velocity.

The **inner loop** uses another **PD controller** to compute the control input u based on the difference between the actual tilt ϕ and the desired tilt ϕ_{desired}:

$$u = k_p \sin(\phi - \phi_{\text{desired}}) + k_d(\dot{\phi})$$

where:

- k_p is the proportional gain that adjusts the robot's body tilt,
- k_d is the derivative gain that dampens the tilt velocity $\dot{\phi}$,
- ϕ_{desired} is the desired tilt angle.

7.2 Control Flow in Cascade Strategy

- **Step 1:** The controller receives a reference trajectory (e.g., desired position over time). The **outer loop** calculates the position error and determines the required body angle ϕ_{desired} to correct this error.
- **Step 2:** The **inner loop** uses the calculated ϕ_{desired} to adjust the actual body angle ϕ by applying control input u (e.g., torque on the wheel) to bring the robot back to the desired tilt.

7.3 Benefits of Cascade Control for Self-Balancing Robots

- **Decoupling of Dynamics:** By using a cascade structure, the fast-changing inner dynamics (body angle) are handled separately from the slower outer dynamics (position), allowing the system to maintain balance while moving to a desired position.
- **Improved Stability:** Each loop focuses on controlling one key variable (either tilt or position), making the system more stable and easier to tune.
- **Effective Disturbance Rejection:** The inner loop (tilt control) quickly responds to disturbances that could cause the robot to fall, while the outer loop ensures that the robot follows the desired trajectory.

7.4 Tuning the Cascade Controller

- **Tuning the Outer Loop** (Position Control): Start by tuning the **position loop gains** k_{p_x} and k_{d_x} while keeping the tilt control gains low. The outer loop should respond slowly to the desired trajectory while minimizing overshoot.
- **Tuning the Inner Loop** (Tilt Control): Once the outer loop is stable, increase the **inner loop gains** k_p and k_d to control the body tilt. The inner loop should respond quickly to maintain balance while minimizing oscillations.

8 Simulation Result

The above plot represents how the robot's body angle ϕ, angular velocity $\dot{\phi}$, and position x vary over time during the simulation.

Here is a visualization of the robot's movement during the simulation:

9 Thoughts and Conclusion

In this project, the EKF is used to estimate the roll angle ϕ and its time derivative $\dot{\phi}$ (angular velocity) based on noisy sensor measurements from an accelerometer and gyroscope. The predicted state is continuously updated as new measurements arrive, providing a robust estimate of the system's state in real-time.

- **Predicted State vs. Predicted Measurement**: $\hat{x}_{k|k-1}$ is the **predicted state** vector, representing quantities like ϕ (roll angle) and $\dot{\phi}$ (angular velocity). $h(\hat{x}_{k|k-1})$ is the **predicted measurement**, which represents what the sensors (e.g., IMU) would measure based on the predicted state.

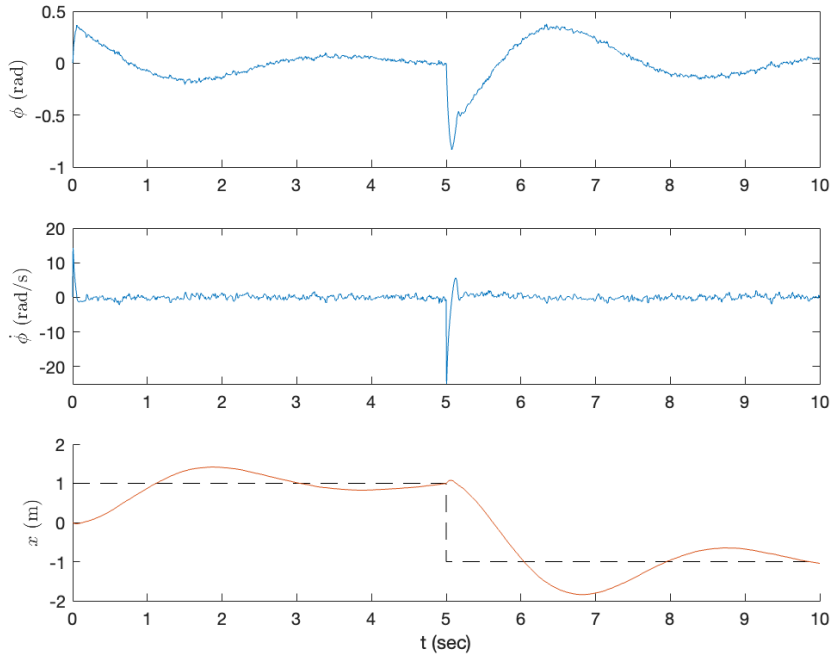

Figure 2: Plot of tilt angle, angular velocity, and position over time.

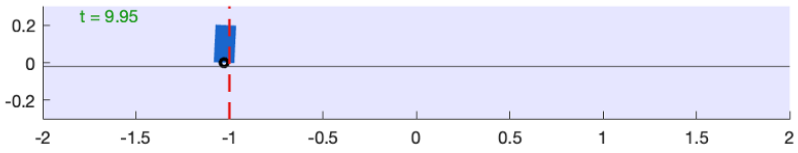

Figure 3: Simulation visualization of the balancing robot at different time steps.

- **Linearization**: Since the measurement model $h(x)$ is typically non-linear, the EKF linearizes it around the current state estimate using the Jacobian H_k.
- **Kalman Gain**: The Kalman Gain K_k determines how much the state estimate is corrected based on the measurement residual y_k. A larger Kalman Gain implies more trust in the new measurement, while a smaller Kalman Gain implies more trust in the predicted state.
- **Tuning**: The performance of the EKF heavily depends on the tuning of the **process noise covariance** Q and the **measurement noise covariance** R. Fine-tuning these parameters allows the EKF to strike the right balance between trusting the model and trusting the measurements.

The cascade control approach is widely used in many control systems where there are hierarchical dynamics, such as in drones (where tilt controls position) or in industrial processes where temperature might be controlled using both fast and slow loops. The self-balancing robot provides a concrete example where body tilt acts as the input for position control, making the system both robust and adaptable.

Overall, this project gave me hands-on experience with both control systems and state estimation techniques, particularly the PID controller and Extended Kalman Filter. By simulating the dynamics of a balancing robot, I gained a deeper understanding of how these techniques apply to real-world robotics and autonomy.