Two Wheeled Path Following Self Balancing Robot

Submitted by:

Kishor Kumar
2022EE11706

Rohit L
2024VST9019
(National Institute of Technology Karnataka, Suratkal)

Under the guidance of
Prof. SUBASHISH DATTA

In fulfilment of the Summer Research Internship

Department of Electrical Engineering
INDIAN INSTITUTE OF TECHNOLOGY DELHI

May 2025-July 2025

Contents

1 Introduction

Self-balancing robots, often resembling miniature Segways, are a fascinating blend of engineering and autonomy. What makes them stand out is their ability to stay upright on just two wheel, mimicking how humans balance while walking. This seemingly simple feat has opened doors to a wide range of real world applications. From personal mobility devices like hover boards and electric scooters to smart delivery robots navigating crowded streets, the core technology behind self balancing systems is increasingly shaping modern transportation and automation. They are also used in camera stabilization rigs, agile warehouse bots, and even assistive robots for the elderly, where smooth, controlled motion is essential. By mastering balance, these robots unlock safe, compact, and highly manoeuvrable solutions in environments where traditional four wheeled designs fall short.

A number of studies have explored the design and control of two wheeled, inverted pendulum robots augmented with path tracking capabilities. Early work by Ha and Yuta demonstrated a trajectory-tracking algorithm that decouples balance and steering control using gyroscopes and wheel encoders, validating performance at moderate speeds in real indoor environments [1]. Goring's comprehensive thesis further detailed mechanical design choices and PID based stabilization, emphasizing the importance of accurate sensor fusion and chassis rigidity for reliable operation [2]. More recent research has extended classical PID approaches with optimal control techniques: Asali et al. applied a linear quadratic regulator (LQR) to a full state space model, achieving improved stability margins under varying payload conditions [3].

Complementary efforts by Arefin illustrated a cyber physical implementation using stepper motors and Bluetooth enabled remote guidance, highlighting practical challenges in power distribution and real time PID tuning for both balance and path following tasks [4]. Together, these works establish a foundation for low cost, robust self balancing, path following platforms and inform the hybrid PID and rule based approach adopted in this report.

This work is motivated by several under explored yet critical questions in two wheeled, self balancing path following robots. First, while existing models assume a fixed Centre of Gravity (CoG), there is little experimental data on how battery placement (front vs. rear, high vs. low) quantitatively affects stability margins, rise time, overshoot, and

settling time [2] [3]. Second, although chassis flexibility is known to introduce disturbances, the relationship between measured deflection under load and controller performance remains uncharacterised. Third, most cascaded PID implementations are validated only for a single payload configuration, leaving open the question of real time control robustness across varying mass distributions. Fourth, even though infrared reflectance sensors are widely adopted for cost effective path tracking, their reliability under changing ambient light conditions has not been rigorously tested or optimized. Finally, practical measurements of actuator non linearities such as dead zones, motor saturation limits, and their effects on control responsiveness are scarce in the literature.

By systematically addressing these gaps through Center of Gravity (CoG) relocation experiments, structural deflection mapping, multi payload PID evaluation, IR sensor ambient light trials, and actuator characterization studies, this research aims to deliver actionable guidelines for designing more robust, adaptive, and low cost self balancing, path following platforms.

2 Literature Review

Extensive research has established a solid foundation for two wheeled, self balancing path following robots, beginning with rigorous dynamic modelling and culminating in low cost, real time implementations. Seminal works by Asali et al. and Frankovský et al. derive the full non linear equations of motion via the Euler–Lagrange formalism which accounts for wheel inertia, body mass distribution, and rolling friction and then linearize about the upright equilibrium to produce compact state space models that drive controller synthesis and stability analysis [3] [5]. Parallel studies highlight the critical role of mechanical rigidity: Goring's finite element verified aluminium chassis minimizes flex under payload shifts, while Halawa's investigations into rapid prototype acrylic frames demonstrate how structural compliance dynamically shifts the centre of gravity and injects disturbances that hinder controller performance [2] [6]. On the control side, although optimal strategies such as LQR and model predictive control offer theoretical performance gains, cascaded PID loops remain the de facto standard for resource constrained microcontrollers where Arefin details an Arduino based implementation where empirical tuning of proportional, integral, and derivative gains coupled with anti windup and derivative filtering yields fast recovery from perturbations without excessive overshoot,

further integrate a dual loop PID architecture on an ATMega Microcontroller Unit (MCU) to coordinate tilt stabilization and velocity control, carefully balancing responsiveness against PWM saturation and torque limitations [2][1][4]. Finally, for autonomous guidance, the simplicity and speed of infrared reflectance sensors make them the preferred choice over computationally intensive camera systems; multi element IR arrays deliver sub millisecond contrast detection with straightforward threshold logic, enabling reliable path tracking on curved paths with minimal processing overhead.

Drawing on these insights, our work plans to implement an Euler–Lagrange derived model, a stiff acrylic based frame, a cascaded PID controller with finely tuned adjustment, and a three sensor IR array together forming a practical, cost effective platform that bridges theoretical rigour and real-world robustness.

3 Objective

- Derive the non linear mathematical model of the two wheeled self balancing robot.
- Linearize the system dynamics around the upright equilibrium about 0° tilt angle.
- Design and tune a PID controller based on the linearized model.
- Validate system behaviour through simulation in MATLAB.
- Integrate IR based path following functionality into the balancing platform.

4 Structural Design

The robot has a multi tiered structure, designed to maintain balance by continuously adjusting its wheels based on readings from both the IMU and IR sensors.

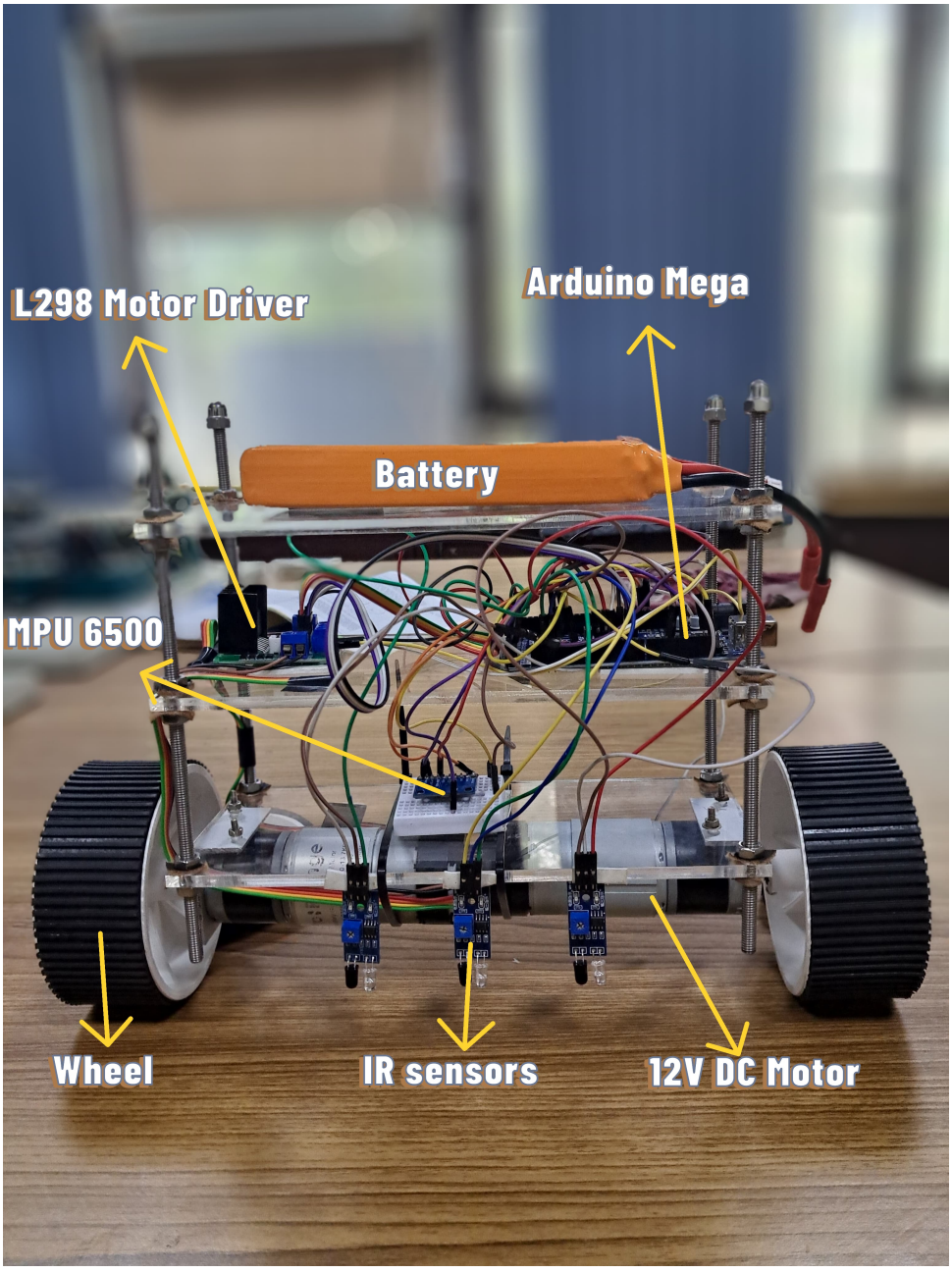


Figure 1: Front View

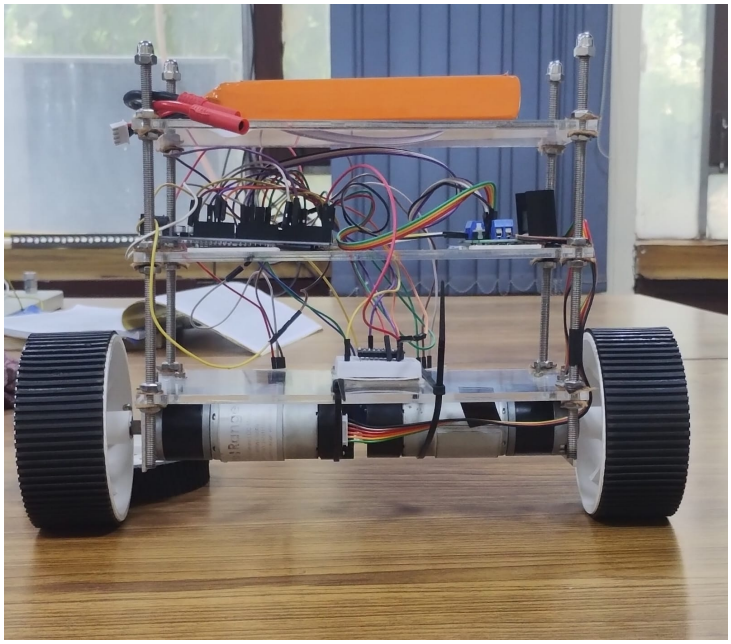


Figure 2: Back View

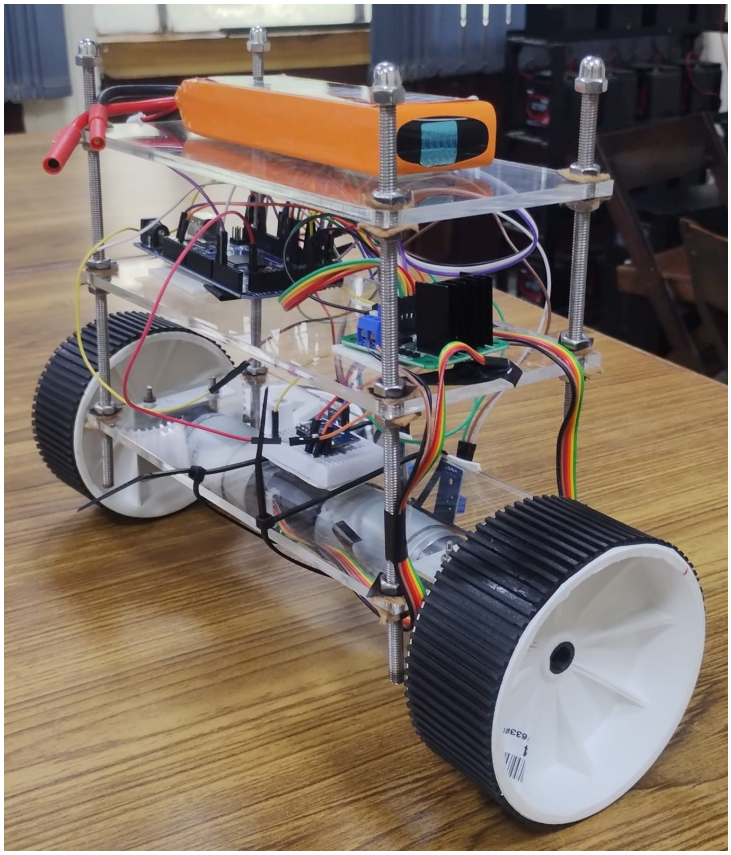


Figure 3: Isometric View

4.1 Chassis

- **Multi Tiered Acrylic Plates:** The chassis is constructed using three layers of acrylic plates of dimensions (230 mm × 100 mm × 6 mm) that provide the mounting surfaces for all electronic components.
  - **Bottom Plate:** This plate mounts the IMU sensor: MPU6500. It is also where the motors and the 3 IR sensors are mounted.
  - **Middle Plate:** It mounts the motor driver and the microcontroller unit.
  - **Top Plate:** This plate primarily holds the battery, positioning it higher to raise the robot's CoG.
- **Threaded Rods:** Four long, metallic threaded rods, 6 mm (diameter) × 150 mm (length) are used to connect and provide structural support between the different plates. This design offers flexibility in adjusting the robot's height and, consequently, its CoG.
- **Wheels:** Two 10 cm (diameter) × 4.4 cm (width) wheels are used.

4.2 Power Supply

LiPo Battery: A Lithium Polymer (LiPo) battery with 3 cells (11.1 V) is used to power the system.

Figure 4: 3S LiPo Battery

4.3 Actuators

- **Geared DC Motors:** Two 12 V DC motors are mounted to the bottom plate, directly driving the wheels. The gearing increases the output torque which is essential for a self-balancing robot to quickly accelerate and decelerate the wheels to counteract tilting, as well as to propel the robot forward/backward.
- **Custom Motor Mounting:** The motors are securely mounted on the bottom plate, ensuring stable power transmission to the wheels.

4.4 Controller and Sensors

- **Microcontroller Unit:**

  The Arduino Mega 2560 board is the central processing unit. Its Arduino Mega2560 Microcontroller will run the main program loop. It reads data from the MPU6500 via I^2C, executes the PID control algorithm based on the IMU data. It is also responsible for reading the IR sensor readings and generating the necessary PWM signals and direction signals for the L298N motor driver to move the robot accordingly.

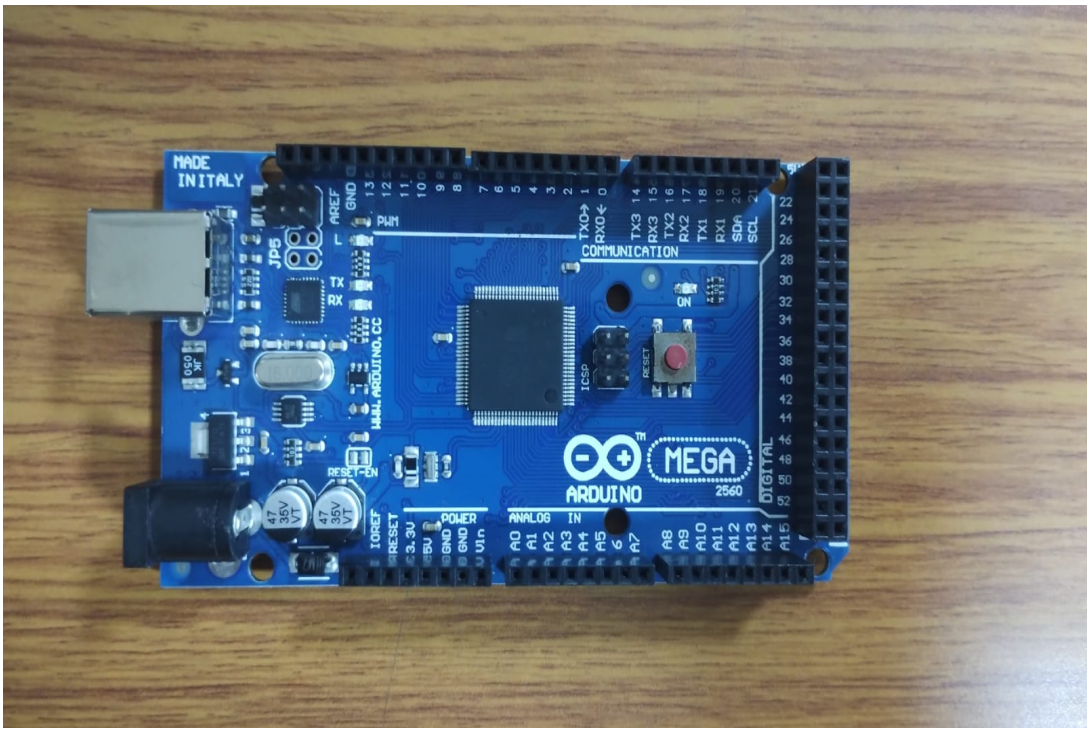


Figure 5: Arduino Mega2560 MCU

- **Motor Driver Module:**

The L298N is a dual H-bridge motor driver, which means that it can control two DC motors independently. It is designed to handle higher voltages and currents than a microcontroller. The Arduino Mega digital output pins (PWM pins for speed control) connect to the L298N IN1-IN4 pins (two for each motor) to control the direction and enable pins for PWM speed control. The L298N's INA,INB,INC and IND terminals connect directly to the positive and negative terminals of the two 12 V DC motors.

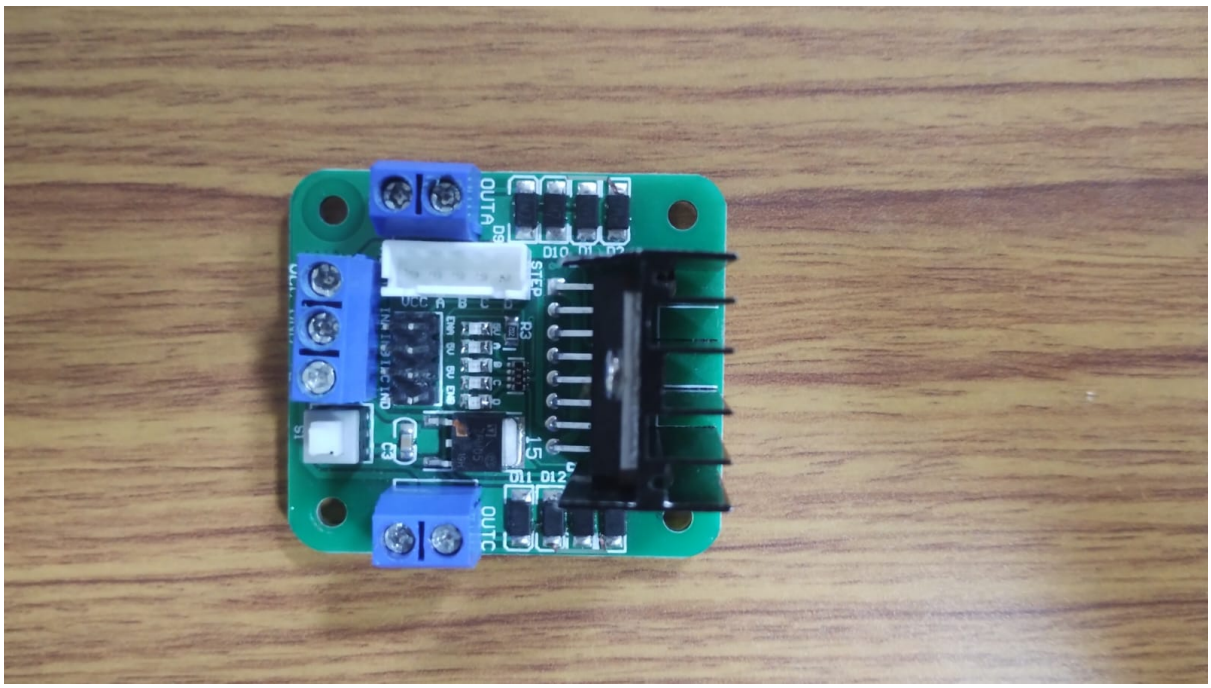


Figure 6: L298N Motor Driver

- **Inertial Measurement Unit (IMU):**

  MPU6500 sensor is fundamental for self-balancing. It combines:

  - **3-axis Accelerometer:**Measures linear acceleration, used to determine the static tilt angle of the robot relative to gravity.
  - **3-axis Gyroscope:**Measures angular velocity (rate of rotation), used to determine how fast the robot is tilting.

  The MPU6500 typically communicates with the Arduino Mega using the I^2C (Inter-Integrated Circuit) protocol. The raw data from the accelerometer and gyroscope will be processed (using a Kalman Filter) on the Arduino Mega to obtain a more accurate and stable estimate of the robot's current tilt angle and angular velocity. This filtered data is the primary input to the PID controller. The MPU6500 must

be mounted rigidly to the robot's chassis, preferably near the robot's geometric centre, to accurately sense its overall movement and orientation on a small white breadboard on the middle plate, connected to the microcontroller with various jumper wires.

Figure 7: Inertial Measurement Unit (MPU6500)

- **Infra Red (IR) Sensors:**
  - We place three IR sensors with 3 cm distance between them on the bottom plate calibrated correctly to detect the presence or absence of a black path in the environment from a distance of 3.5 cm above ground level.
  - The sensor reads 0 when it detects white and 1 when it detects black.
  - These sensor readings are digitally read by the Arduino Mega through its digital pins.

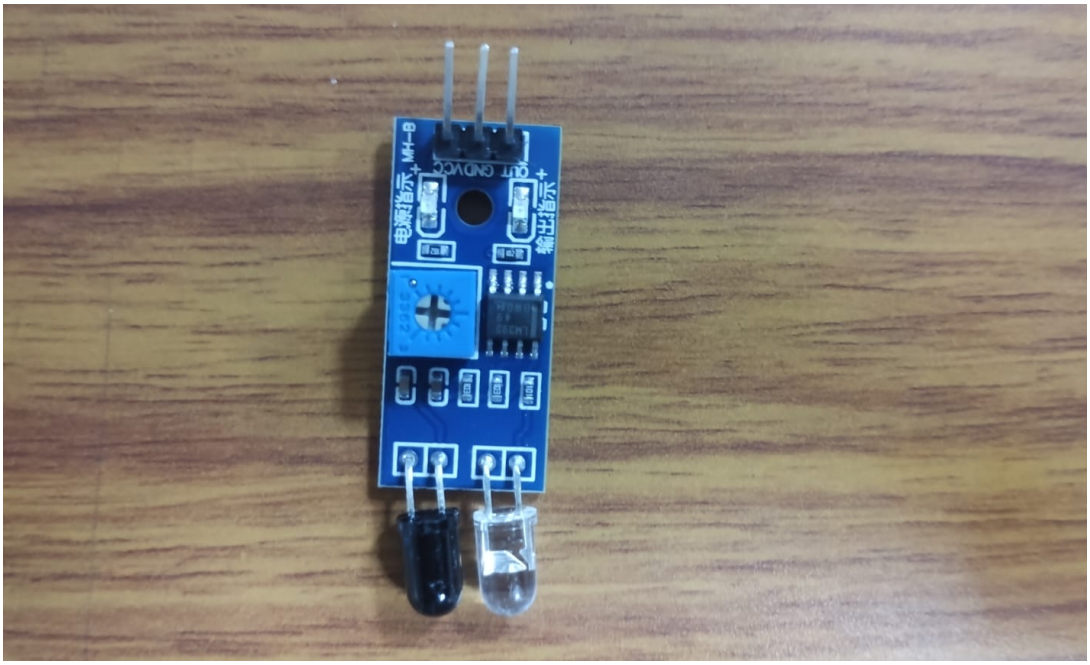


Figure 8: Infra Red Sensor

4.5 Choice of Motor

Motors tested and challenges faced:

4.5.1 24 V DC motor:

Figure 9: 24 V DC Motor

Motor Rating vs. Supply Voltage:

- The motors are rated for 178 RPM at 24 V.

- Initially powered using a 12 V Lithium Polymer battery, which significantly reduced available torque.

Insufficient Torque at 12 V:

- At 12 V, the motors failed to generate sufficient torque for stable balancing, especially during large tilt disturbances or when the battery was mounted high on the chassis.

PID Gain Compensation and Saturation:

- To compensate for low torque, the PID controller had to be tuned with high gains—particularly the integral gain (K_i), which sometimes exceeded 1000.
- These high gains often caused the computed control signal (PWM) to exceed the Arduino's ± 255 limit.

Controller Saturation Effects:

- The PWM output saturated at ± 255 when the error exceeded a small angle ($\sim 1.7^\circ$), resulting in an ON-OFF controller kind of behaviour instead of a smooth, continuous PID-regulated system.

Testing at Rated Voltage (24 V):

- When powered with a 24 V DC regulated supply, the robot showed much better performance:
  - Achieved 23 to 27 seconds of stable balance with the battery mounted on top.
  - Demonstrated even longer stability without the top mounted battery.

Wired Power Supply Limitation:

Direct 24 V DC supply introduced physical limitations due to wire entanglement with the wheels during motion.

Challenges with 24 V Lithium Polymer Battery:

- Though wireless and capable of delivering the rated voltage, the 24 V Lithium Polymer battery was:
  - Too heavy battery which constitutes $\sim$ 40–60% of the robot's total weight.

- Raised the centre of gravity, decreasing system stability.
- More expensive and impractical for compact, lightweight designs.

Overall Trade-Off:

- A clear trade off exists between motor torque, voltage supply, structural load, and controller design.
- Achieving optimal performance requires balancing electrical, mechanical, and control parameters.

4.5.2 12 V DC Servo (Velocity controlled) motor:

Figure 10: 12 V DC Servo Motor

Motor Specifications:

- High torque and high RPM which makes it suitable for self-balancing applications.
- Inbuilt motor driver and encoder.
- Communication via UART(Universal Asynchronous Receiver Transmitter) or I^2C(Inter Integrated Circuit) Protocols.

- Fixed baud rate: 9600 bps (bits per second).

Initial Assessment:

- The motor appeared ideal as a replacement for the earlier 24 V DC motor due to its superior torque and speed.
- Serial communication was necessary for control; UART was chosen over I^2C due to address conflicts.
  - Both motors shared the same I^2C slave address (0x10) with no way to run them independently in parallel.
  - Arduino Mega2560's multiple UART ports enabled UART integration.

UART Integration and Testing:

- UART protocol implemented as per the motor datasheet.
- Motors tested individually and together, outside and within the PID control framework.

Issues Encountered During Use:

- **Persistent Command Memory:** On Arduino reset, the motor retained the last received command, causing unintentional movement.
- **Unstable Behaviour from Loose Connections:** Loose power or ground connections caused the motor to behave erratically, often making the robot fall abruptly.
- **Idle Timeout Response Failure:** Leaving the robot idle for a few minutes led to unresponsiveness; re-adjusting power pins was required to restore functionality.
- **Startup Conflicts:** Uploading code after power was supplied caused the Arduino to shut down or the motors to malfunction unexpectedly.

Troubleshooting Measures Taken:

- Wiring was thoroughly cross-checked.
- Ensured that code upload was done before powering the system.

Critical Limitation – Low Baud Rate:

- A significant delay of 2–3 seconds was observed between command issuance and motor response.
- This delay was traced to the low fixed baud rate (9600 bps) of the UART interface.
- Such latency made the motor unsuitable for self-balancing applications that require real-time, low-latency control.

Conclusion:

- Despite its high torque and integrated features, the motor's slow serial communication and unpredictable behaviour made it impractical for our application.
- A more suitable alternative would be a conventional high-torque DC motor paired with a fast-switching external driver, ensuring both low latency and sufficient power for balance control.

4.5.3 12 V DC Planetary Geared Motor:

4.5.4 12 V DC motor:

Figure 11: 12 V DC Motor

Overall Performance:

- Proven to be the most suitable motor used in this project.
- Delivered sufficient torque to stabilize the robot under typical operating conditions, provided that appropriate control signals were applied.

Ease of Integration:

- Simpler to interface and control compared to the previously used 12 V DC servo motor.
- Did not require complex serial protocols or special communication setups.

Signal Sensitivity:

- Capable of responding to low PWM values (~ 60), indicating higher sensitivity and better control resolution compared to the 24 V DC motor used earlier.
- Enabled finer control for small corrections during balance maintenance.

Stabilization Capability:

- The robot exhibited reliable stability during small tilt deviations.
- However, the motor lacked the torque and speed required to recover from larger tilt angles, leading to failed stabilization in such cases.

Conclusion:

- An optimal choice for initial testing and moderate control tasks.
- Designed for applications where fast, lightweight actuation is needed, although not ideal for aggressive recovery or high-disturbance balancing scenarios.

4.6 Improvements

4.6.1 Acrylic plates

The initial chassis design used acrylic plates with dimensions of 260 mm $\times$ 100 mm $\times$ 4 mm, supported by four threaded rods and equipped with lightweight 24 V DC motors (approximately 200 g each). This configuration offered a low overall mass, which was beneficial for early stage testing. However, the torque output of these motors was insufficient to achieve stable self-balancing behaviour, prompting a transition to more powerful 12 V DC servo motors.

1. The servo motors, each weighing approximately 480 g, introduced a significant increase in load. This added weight caused noticeable inward bending of the long acrylic plates, leading to misalignment of the motor shafts and wheels. As a result, the wheels were no longer parallel to the ground plane, introducing additional instability and complicating the control task.

2. Anticipating similar structural issues with another set of motors weighing 458 g each, we revised the mechanical design to enhance structural rigidity. A more compact chassis layout was adopted using shorter, sturdier acrylic plates of the same thickness.

3. The redesigned frame offered significantly improved resistance to bending, maintaining proper wheel alignment even under high loads. This structural modification contributed to improved control performance by eliminating mechanical deformations that previously introduced undesired dynamics.

4.6.2 Wheels

Wheel size and mass were also found to significantly influence the robot's stability and control characteristics. To evaluate this, we tested four different wheel configurations, each varying in diameter, width, and mass:

- **Small wheel**: 50 mm diameter × 15 mm width, 40 g

Figure 12: Small Wheel

- **Medium wheel**: 80 mm diameter × 20 mm width, 87 g

Figure 13: Medium Wheel

- **Large wheel**: 100 mm diameter × 22 mm width, 96 g

Figure 14: Large Wheel

- **Largest wheel**: 100 mm diameter × 44 mm width, 106 g

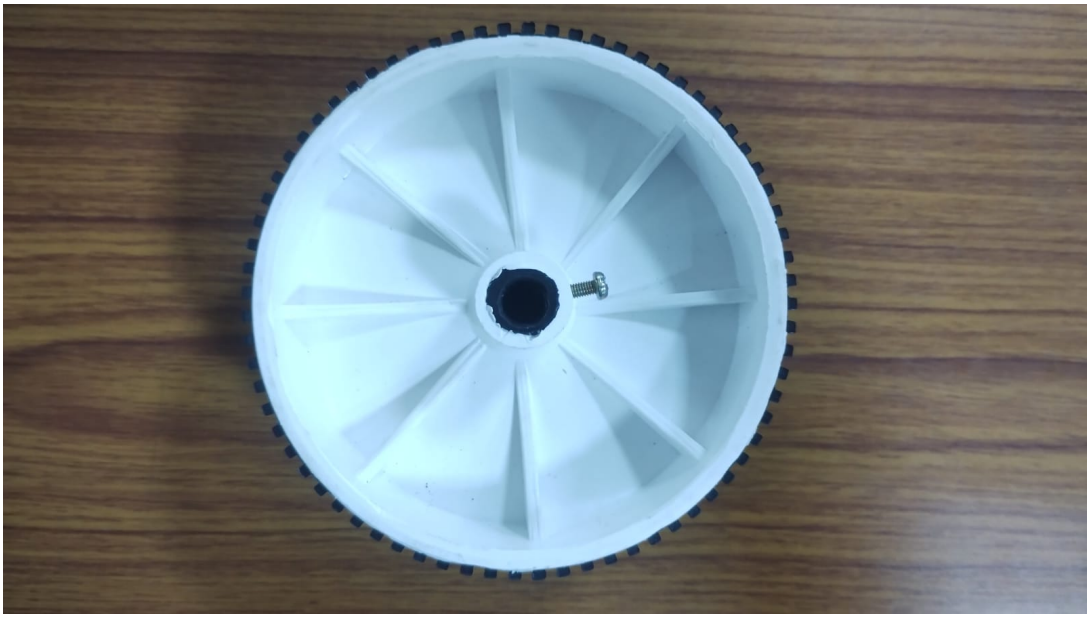


Figure 15: Largest Wheel

1. Experimental trials revealed that, despite the increased mass, the largest wheels with greater width and ground contact area provided the highest degree of stability during dynamic balancing. This configuration was especially effective when combined with the 12 V DC servo motors, which possessed sufficient torque to rapidly accelerate the heavier wheels and respond effectively to small angle deviations.

2. Furthermore, the use of these wide wheels introduced a unique unstable equilibrium behaviour not observed with narrower or smaller diameter wheels. This characteristic appeared to contribute positively to the control system's responsiveness, allowing the robot to maintain balance more effectively under perturbations.

4.6.3 Position of battery

Through experimental observations and control system analysis, we identified that the vertical orientation stability of the robot is primarily influenced by two critical factors: **motor torque** and the **position of the battery**. Among all components on the chassis, the battery measured to weigh approximately 0.432 kg constitutes the single heaviest element and thus significantly affects the overall mass distribution.

1. Drawing parallels from the classic inverted pendulum on cart system, the centre of gravity (CoG) must be carefully considered during system modelling, as it directly influences the system's dynamics.

Depending on the battery's placement, the CoG shifts accordingly, resulting in different system behaviours.

   (a) When the battery is mounted on the **top plate**, the CoG lies approximately **0.108 m above** the motor's rotational axis.

   (b) When placed on the **bottom plate**, the CoG is lowered to **0.059 m above** the motor axis.

2. This change in CoG height notably alters the dynamics of the system and therefore modifies the plant's transfer function. As a consequence, the optimal PID controller gains also differ between configurations. Specifically, the higher placement of the battery (and hence a higher CoG) resulted in **higher PID gain values**, likely due to the increased torque required to counteract larger moments of inertia and gravitational destabilization.

3. Interestingly, contrary to intuitive expectations, empirical testing revealed that the robot exhibited **improved balancing performance when the battery was placed on the top plate**. Across multiple PID tuning sessions, the robot consistently achieved faster response and better damping in this configuration. A plausible explanation may lie in the system's enhanced sensitivity and quicker corrective action due to the higher CoG, which promotes more active stabilization dynamics.

Based on these observations, the final design adopted the **top-mounted battery configuration**, as it yielded **superior stability and control response** in repeated experimental trials.

5 Electronic System Configurations

5.1 Power Distribution

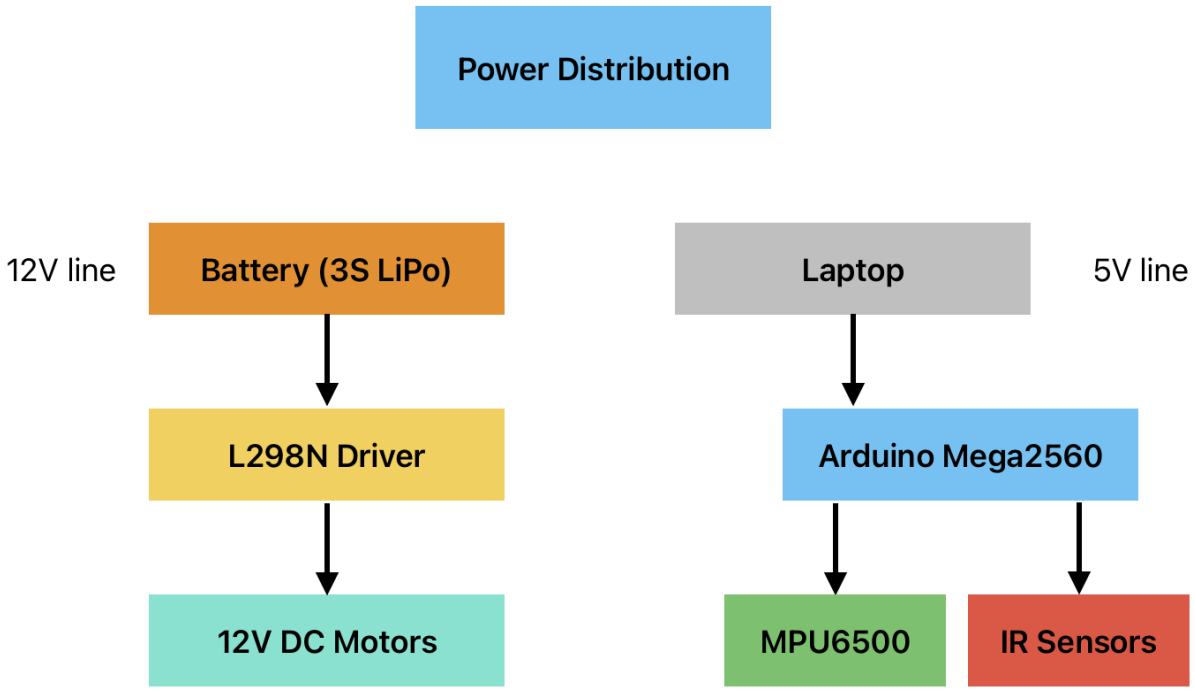


Figure 16: Power Distribution flowchart

5.2 Data and Communication

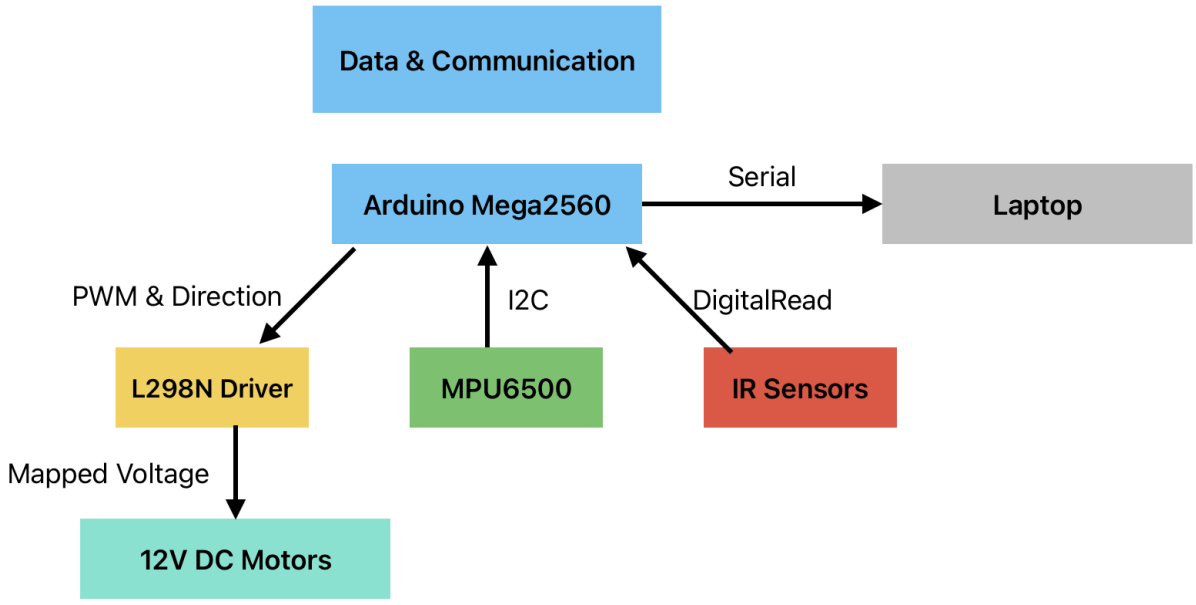


Figure 17: Data flow path

5.3 Sensor Placement

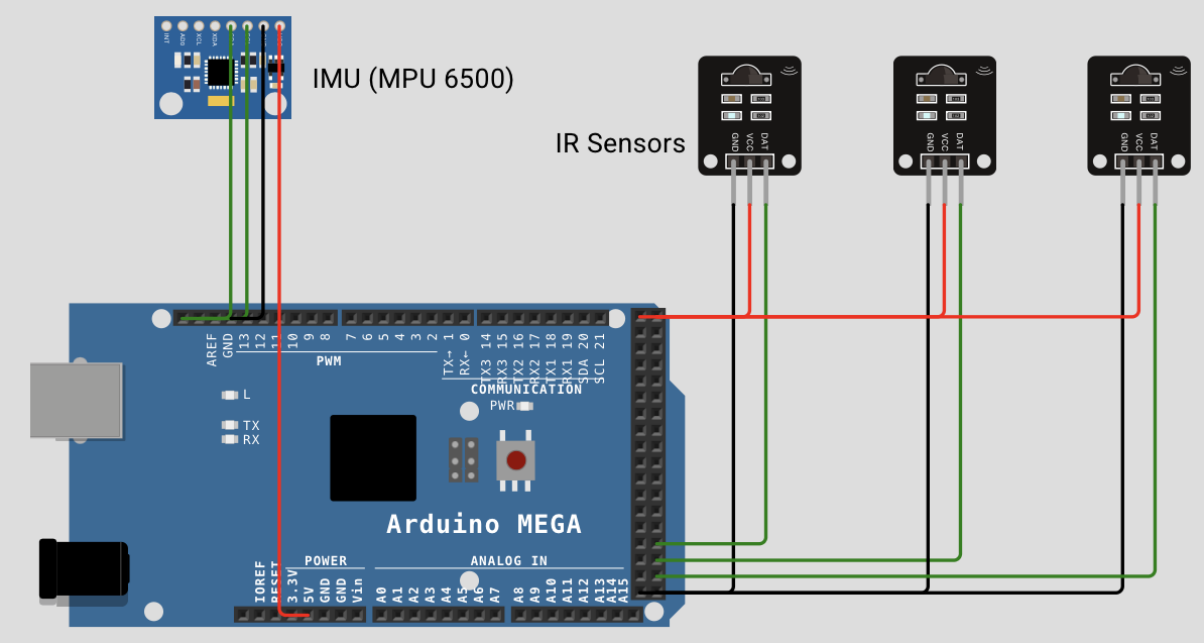


Figure 18: Sensor connections

6 System Modelling:

To mathematically model the robot, the principles of classical mechanics, specifically Newton's second law and Euler-Lagrange equations are used [1][2][5]. The modelling is done by treating the system as an inverted pendulum on wheels, which is a classic control problem.

6.1 Assumptions:

1. Where appropriate, components will be modelled as point masses or bodies with uniformly distributed mass.
2. The wheels are assumed to roll without slipping on the ground. [5]
3. Rolling friction is assumed to be negligible between the wheels and the ground. [5]
4. The robot can move in a 2D plane (forward/backward). The primary motion of interest is the balancing act, so we'll focus on the pitch angle and horizontal displacement.

6.2 State Variables:

The state variables can be defined as:
$x_1 = \text{p}$ (horizontal position of the wheel axle)
$x_2 = \dot{p}$ (horizontal velocity of the wheel axle)
$x_3 = \theta$ (pitch angle of robot body)
$x_4 = \dot{\theta}$ (pitch angular velocity of robot body)
The input is $\text{u} = V_{in}$ and the output is $\text{y} = \theta$.

6.3 Parameters:

Parameter	**Value and Description**
Battery Mass (M_{battery})	0.432 kg
Robot Body Mass (M_{body})	1.15 kg (excluding wheel and motor)
Back EMF Constant ($K_e = K_t$)	0.118
Terminal Resistance (R_m)	5.6 Ω
Wheel Radius (R_w)	0.05 m
Back EMF Reflected (K_b)	$\frac{K_e^2}{R_m R_w} = 0.0348$
Centre of Gravity Height (h)	0.108 m (from wheel centre)
Body Inertia (I_{body})	0.01306 kg·m^2
Total Inertia (I_{total})	$I_{\text{body}} + M_{\text{body}} h^2$
Wheel Mass (M_w)	0.106 kg
Wheel Inertia (J_w)	$\frac{1}{2} M_w R_w^2$
Rotor Inertia (J_m)	2.66×10^{-5}
Equivalent Mass (M_{eq})	$M_{\text{body}} + 2M_w + \frac{2(J_w + J_m)}{R_w^2} = 1.454$ kg
Motor Constant (K_b)	$\frac{K_e}{R_m} = 0.0148$
Δ	$M_{\text{eq}} I_{\text{total}} - (M_{\text{body}} h)^2 = 0.0237$ kg^2·m^2
Gravity (g)	9.81 m/s^2

Table 1: System Parameters Used in Modelling

6.4 Deriving Equations of Motion:

The Lagrangian L is defined as L=T-V where T is total kinetic energy and V is total potential energy of the system [2] [5]. The Euler-Lagrange equations are given by:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = Q_i \tag{1}$$

Where q_i are generalised coordinates, $\dot{q}_i$ are their time derivatives and Q_i are the generalised non conservative forces.

6.4.1 Kinetic Energy(T):

1) **Kinetic Energy of Robot Body (T_{body})**:

Translation:

$T_{body,trans}$=$\frac{1}{2}M_{body}(\dot{p} - (hcos\theta)\dot{\theta})^2$

Rotational:

T_{rot}=$\frac{1}{2}I_{body}\dot{\theta}^2$

T_{body}=$\frac{1}{2}M_{body}(\dot{p} - (hcos\theta)\dot{\theta})^2 + \frac{1}{2}I_{body}\dot{\theta}^2$

2) **Kinetic Energy of Wheels (T_w)**:

Translation:

$T_{w,trans}$=$2 \times \frac{1}{2}M_w(\dot{p})^2$

Rotational:

T_{rot}=$2 \times \frac{1}{2}J_w\dot{\varphi}^2$

$T_w = M_w(\dot{p})^2 + J_w\dot{\varphi}^2$

3) **Kinetic Energy of Motor rotors (T_m)**:

$T_m = 2 \times \frac{1}{2}J_m\dot{\varphi}^2$

Total Kinetic Energy T= $T_{body} + T_w + T_m$

$$T = \frac{1}{2}M_{body}(\dot{p} - (hcos\theta)\dot{\theta})^2 + \frac{1}{2}I_{body}\dot{\theta}^2 + M_w(\dot{p})^2 + J_w\dot{\varphi}^2 + J_m\dot{\varphi}^2 \qquad (2)$$

6.4.2 Potential Energy (V):

The potential energy is due to the height of the body's CoG. Taking ground level V=0.

$V = M_{body}g(R_w + hcos\theta)$

6.4.3 Lagrangian (L):

L=T-V

$$L = \frac{1}{2}M_{body}(\dot{p}-(hcos\theta)\dot{\theta})^2+\frac{1}{2}I_{body}\dot{\theta}^2+M_w(\dot{p})^2+J_w\dot{\varphi}^2+J_m\dot{\varphi}^2-M_{body}g(R_w+hcos\theta) \tag{3}$$

The generalised forces are:

Q_p**:** Force acting in the p vector's direction. This is the sum of the horizontal forces at the contact point between the wheels and the ground due to motor torque. Let $F_{contact}$ be the horizontal force at the contact point of each wheel.

$Q_p = 2F_{contact}$

Q_θ**:** Net torque acting on the body to change θ. This is the reaction torque from the motor on the body. If the motor applies τ_M to the wheel, it applies $-\tau_M$ to the body. So,

$Q_\theta = -2\tau_M$.

Q_φ**:** Net torque acting on the wheels to change. This is the motor torque applied to the wheels, minus friction.

$Q_\varphi = 2\tau_M$

6.4.4 Non-slip Condition:

The horizontal velocity of the wheel's contact point with the ground is 0.

This condition is mathematically expressed as: $\dot{p} - R_w\dot{\varphi} = 0$

$\dot{p} = R_w\dot{\varphi}$

$\therefore$ $\dot{\varphi}$ is substituted with $\frac{\dot{p}}{R_w}$ in the equation of total kinetic energy (T).

Now we have two generalized coordinates: p and θ.

$$T = \frac{1}{2}M_{body}(\dot{p} - (hcos\theta)\dot{\theta})^2 + \frac{1}{2}I_{body}\dot{\theta}^2 + (M_w + \frac{J_w + J_m}{R_w^2})\dot{p}^2 \tag{4}$$

Equation for p:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{p}}\right) - \frac{\partial L}{\partial p} = Q_p \tag{5}$$

$$2F_{contact} = \frac{1}{2}M_{body}(\ddot{p}+(hsin\theta)\dot{\theta}^2-(hcos\theta)\ddot{\theta})+\frac{1}{2}I_{body}\dot{\theta}^2+2(M_w+\frac{J_w+J_m}{R_w^2})\ddot{p} \tag{6}$$

$M_{eq} = M_{body} + 2(M_w + \frac{J_w+J_m}{R_w^2})$

Equation for θ:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = Q_\theta \tag{7}$$

$$\begin{aligned} -2\tau_M = &-M_{\text{body}}h\cos\theta\,\ddot{p} + \left(I_{\text{body}} + M_{\text{body}}\left(h\cos\theta\right)^2\right)\ddot{\theta} \\ &+ M_{\text{body}}h^2\cos\theta\sin\theta\,\dot{\theta}^2 - M_{\text{body}}hg\sin\theta \end{aligned} \tag{8}$$

6.5 Electrical Dynamics of Motor:

For each motor, the voltage equation is:

$V_{in} - V_b = I_a R_m + L_M \frac{\partial I_a}{\partial t}$

$V_{in} - K_e\dot{\varphi} = I_a R_m + L_M \frac{\partial I_a}{\partial t}$

And the motor torque,

$\tau_M = K_t I_a$

L_M is often neglected and $\dot{\varphi} = \frac{\dot{p}}{R_w}$

$\tau_M = \frac{K_t}{R_m}V_{in} - \frac{K_t K_e}{R_m R_w}\dot{p}$

$K_m = \frac{K_t}{R_m}$ (Motor constant)

$K_b = \frac{K_t K_e}{R_m R_w}$ (Back EMF constant reflected to wheel velocity)

$\tau_M = K_m V_{in} - K_b\dot{p}$

After substituting,

$$\begin{aligned} -2(K_m V_{\text{in}} - K_b\dot{p}) = &-M_{\text{body}}h\cos\theta\,\ddot{p} + (I_{\text{body}} + M_{\text{body}}(h\cos\theta)^2)\ddot{\theta} \\ &+ \tfrac{1}{2}M_{\text{body}}h^2\sin(2\theta)\dot{\theta}^2 - M_{\text{body}}hg\sin\theta \end{aligned} \tag{9}$$

6.6 Linearized Model:

For control system design, we linearize the model around the upright equilibrium position (θ=0, $\dot{\theta} = 0$, $\ddot{\theta} = 0$, p=0, $\dot{p} = 0$).

6.6.1 Assumptions for linearization:

$sin\theta \approx 0$, $cos\theta \approx 1$, $\dot{\theta}^2 \approx 0$

6.6.2 Linearized Horizontal Motion:

$2F_{contact} = M_{eq}\ddot{p} - (M_{body}h)\ddot{\theta}$

6.6.3 Linearized Pitch Motion:

$-2\tau_M = -M_{body}h\ddot{p} + (I_{body} + M_{body}h^2)\ddot{\theta} - M_{body}hg\theta$

$I_{total} = I_{body} + M_{body}h^2$

$-2(K_mV_{in} - K_b\dot{p}) = -M_{body}h\ddot{p} + I_{total}\ddot{\theta} - M_{body}hg\theta$

Ideally,

$2F_{contact} = \frac{2\tau_M}{R_w}$

6.7 State Variables:

$$x = \begin{bmatrix} p \\ \dot{p} \\ \theta \\ \dot{\theta} \end{bmatrix}$$

On rewriting the previously found coupled linear equations;

$M_{eq}\ddot{p} - (M_{body}h)\ddot{\theta} = \frac{2(K_mV_{in} - K_b\dot{p})}{R_w}$

$-(M_{body}h)\ddot{p} + I_{total}\ddot{\theta} = M_{body}gh\theta + 2K_b\dot{p} - 2K_mV_{in}$

$$\begin{bmatrix} M_{eq} & -M_{body}h \\ -M_{body}h & I_{\text{total}} \end{bmatrix} \begin{bmatrix} \ddot{p} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{2(K_mV_{in} - K_b\dot{p})}{R_w} \\ M_{body}gh\theta + 2K_b\dot{p} - 2K_mV_{\text{in}} \end{bmatrix} \tag{10}$$

Let $M = \begin{bmatrix} M_{eq} & -M_{body}h \\ -M_{body}h & I_{\text{total}} \end{bmatrix}$

$$\begin{bmatrix} \ddot{p} \\ \ddot{\theta} \end{bmatrix} = M^{-1} \begin{bmatrix} \frac{2(K_m V_{in} - K_b \dot{p})}{R_w} \\ M_{body} g h \theta + 2K_b \dot{p} - 2K_m V_{\text{in}} \end{bmatrix} \tag{11}$$

The determinant of M is $\Delta = M_{eq} I_{total} - (M_{body} h)^2$

$$M^{-1} = \frac{1}{\Delta} \begin{bmatrix} I_{total} & M_{body} h \\ M_{body} h & M_{\text{eq}} \end{bmatrix} \tag{12}$$

Finding expressions for state variables and arranging them in state-space form:

$\dot{X} = AX + BU$ and $Y = CX + DU$ where $U = V_{in}$.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{1}{\Delta}(2K_b M_{\text{body}} h - \frac{2K_b I_{\text{total}}}{R_w}) & \frac{(M_{\text{body}} g h)^2}{\Delta} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{1}{\Delta}\left(2K_b M_{\text{eq}} - \frac{2K_b M_{\text{body}} h}{R_w}\right) & \frac{M_{\text{eq}} M_{\text{body}} g h}{\Delta} & 0 \end{bmatrix} \tag{13}$$

$$B = \begin{bmatrix} \frac{1}{\Delta}(\left(\frac{2K_m I_{\text{total}}}{R_w}\right) - 2K_m M_{\text{body}} h) \\ 0 \\ \frac{1}{\Delta}(\frac{2K_m M_{body} h}{R_w} - 2K_m M_{\text{eq}}) \\ 0 \end{bmatrix} \tag{14}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{15}$$

$$D = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{16}$$

But it must be clear that we are interested only with the tilt angle (θ). Hence we consider only the required part of matrices C and D.

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \tag{17}$$

$$D = \begin{bmatrix} 0 \end{bmatrix} \tag{18}$$

The transfer function is calculated as:

$$V = \frac{1.819s - 1.596^{-15}}{s^3 + 1.733s^2 - 74.65s - 102.1} \tag{19}$$

We neglect the constant term in the numerator, since it is a very small quantity. This also eases our calculations later, as seen in Section 8.0.1. We finally get;

$$V = \frac{1.819s}{s^3 + 1.733s^2 - 74.65s - 102.1} \tag{20}$$

7 Control Architecture and System Objectives:

The overall control objective is bifurcated into two primary components: (1) achieving vertical stabilization of the robot, and (2) enabling reliable path following behaviour. These two functionalities are implemented concurrently, with the final control input to the actuators obtained by superimposing the control signals generated by each subsystem.

7.1 Vertical Stabilization

To maintain the robot's upright orientation, a classical Proportional–Integral –Derivative (PID) control strategy is employed. The controller is designed to minimize the deviation of the system's output typically the tilt angle from a zero reference set point. Real time feedback of the system states, including angular displacement and its derivatives, is used to compute the control signal. The PID controller parameters were empirically tuned to achieve optimal performance, with the following gains:

- **Integral Gain ($K_i = 749.905$):** Applied post integration block ($\frac{1}{s}$), this term accumulates the historical error, thus reducing the steady state error and ensuring convergence to the desired orientation.
- **Proportional Gain ($K_p = 441.475$):** Scales the instantaneous error to generate a control response proportional to the magnitude of the deviation from the set point.

- **Derivative Gain ($K_d = 0.01$):** Acts on the rate of change of error ($\frac{\Delta u}{\Delta t}$), providing predictive damping to suppress oscillatory behaviour and enhance system stability.

7.2 Path Following Control

The path following functionality is implemented using a rule based control scheme based on discrete sensor readings. The system distinguishes among four distinct sensor input cases, each corresponding to a particular alignment of the robot with respect to the path. For each case, a predefined PWM adjustment is computed and added to the output of the PID stabilization controller as specified in the table given below. This modular control strategy allows for real time trajectory tracking while preserving the integrity of the vertical stabilization loop. By combining these two control actions, the robot is capable of maintaining balance while accurately following a predefined path, enabling robust locomotion in dynamic environments.

State	IR (left)	IR (middle)	IR (right)	Left motor PWM	Right motor PWM
Straight	0	1	0	60	60
Left	1	1	0	-50	50
Right	0	1	1	50	-50
Sharp Left	1	0	0	-75	75
Sharp Right	0	0	1	75	-75
Stability	1	1	1	0	0
Stability	1	0	1	0	0
Stability	0	0	0	0	0

Sensor usage decisions:

1. **MPU6500** :

   (a) The Inertial Measurement Unit (IMU) employed in this system integrates a 3-axis accelerometer and a 3-axis gyroscope to estimate the orientation of the robot in terms of pitch, roll, and yaw. For the purpose of balance control, the pitch angle is the primary variable of interest, as it directly informs the control signal generated by the microcontroller (MCU).

   (b) While the gyroscope provides accurate short-term angular velocity measurements, it suffers from cumulative drift over time

due to sensor noise and bias, making it unreliable for long duration measurements when used independently. Conversely, the accelerometer offers stable long term orientation estimation, but its readings are susceptible to noise and transient disturbances, especially during motion.

(c) To address the limitations of each sensor, a **digital Kalman filter** is employed for sensor fusion. The Kalman filter continuously integrates data from both the gyroscope and the accelerometer, dynamically estimating and correcting the system's state. It effectively mitigates gyroscopic drift using the accelerometer as a reference, while also smoothing out accelerometer noise using gyroscopic trends. This fusion process yields a more accurate and stable pitch estimate, enabling robust real time control of the robot's vertical orientation.

2. **IR Sensors**:

(a) The initial design of the robot incorporated a three sensor infrared (IR) array, with each sensor spaced 3 cm apart. This configuration was strategically selected to enable reliable detection of the central path, with the lateral sensors assisting in identifying deviations and supporting directional corrections. During testing, this setup demonstrated high accuracy in trajectory tracking. The robot performed effectively on the first trial, requiring only minor velocity adjustments for sharp and gentle turns, which were refined through iterative experimentation.

(b) To explore potential hardware simplification, a two sensor IR configuration was subsequently tested. The trajectory tracking algorithm and condition checking logic were revised to accommodate the reduced input. While the two sensor system was functional under ideal conditions, it exhibited a significantly narrower detection range. In scenarios where the robot lost alignment with the path, it often defaulted to a passive stabilization mode and failed to reacquire the path, leading to tracking failure. This limitation was primarily attributed to the reduced field of view and lack of redundancy in the sensing mechanism.

(c) Given these findings, the design reverted to the original three sensor configuration. This arrangement provided a more robust and reliable solution for path detection and trajectory mainte-

nance, particularly in dynamic and curved path segments where continuous correction was essential.

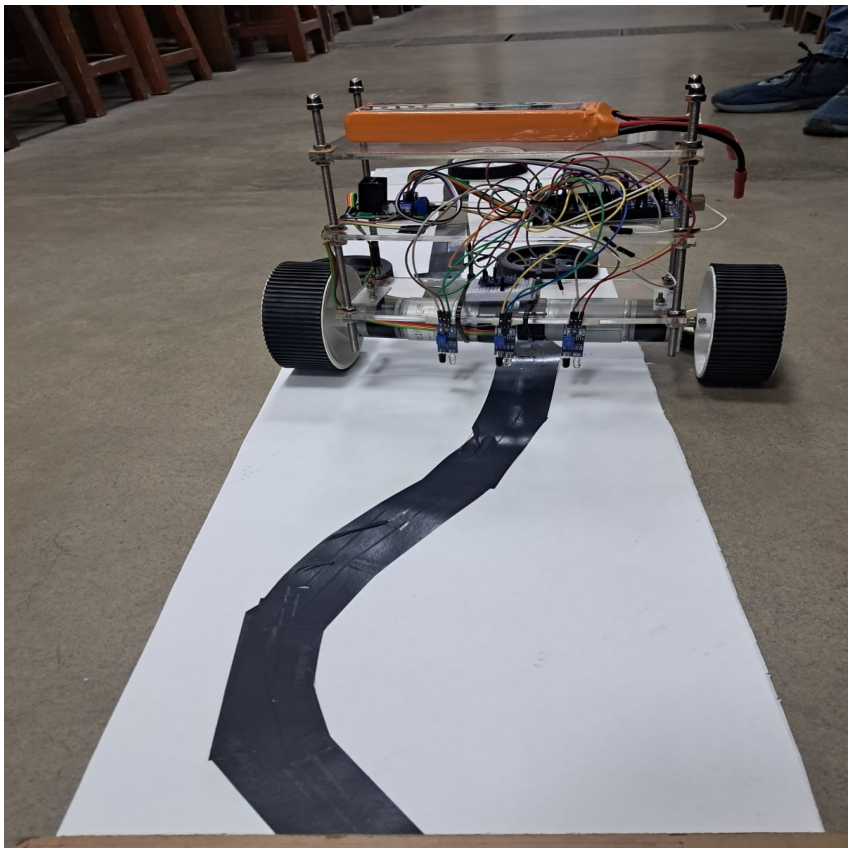


Figure 19: Front View

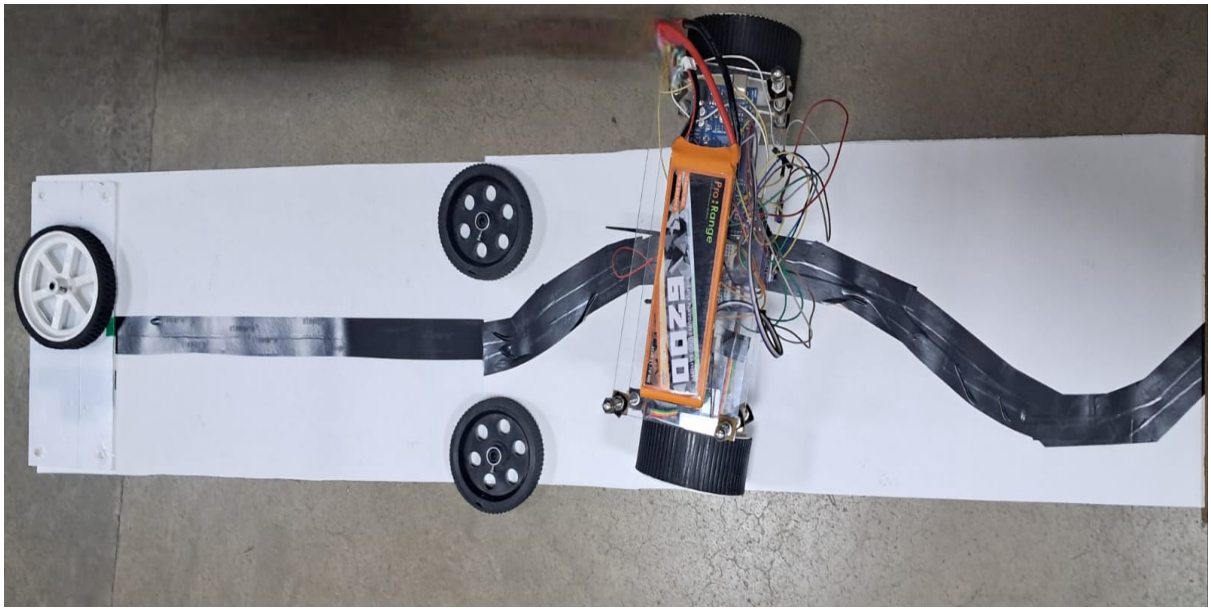


Figure 20: Top View

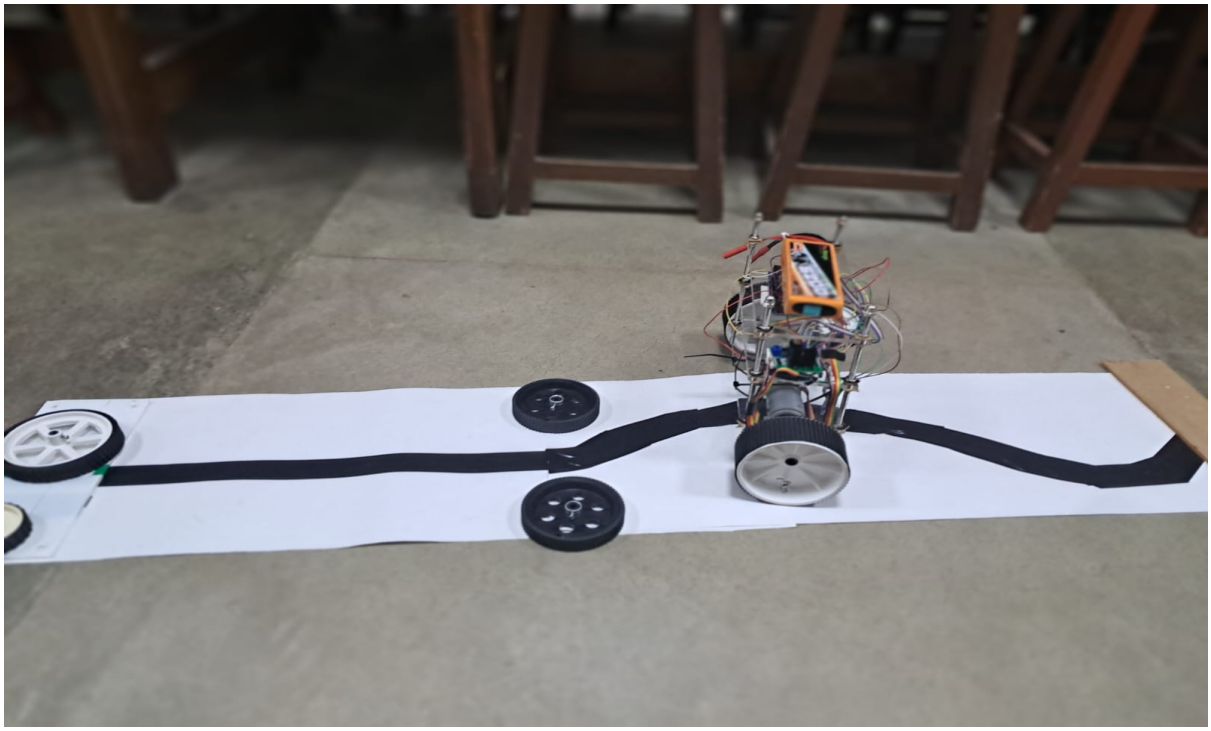


Figure 21: Side View

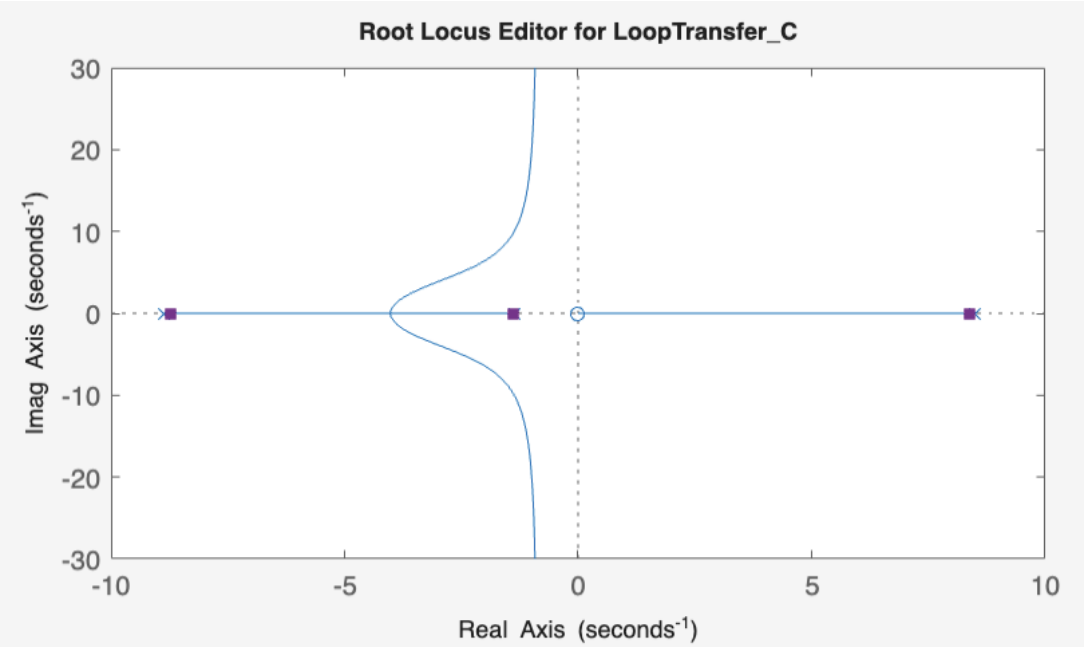
8 Implementation in Matlab:

Two distinct approaches were employed using MATLAB to design and validate the control system for the robot.

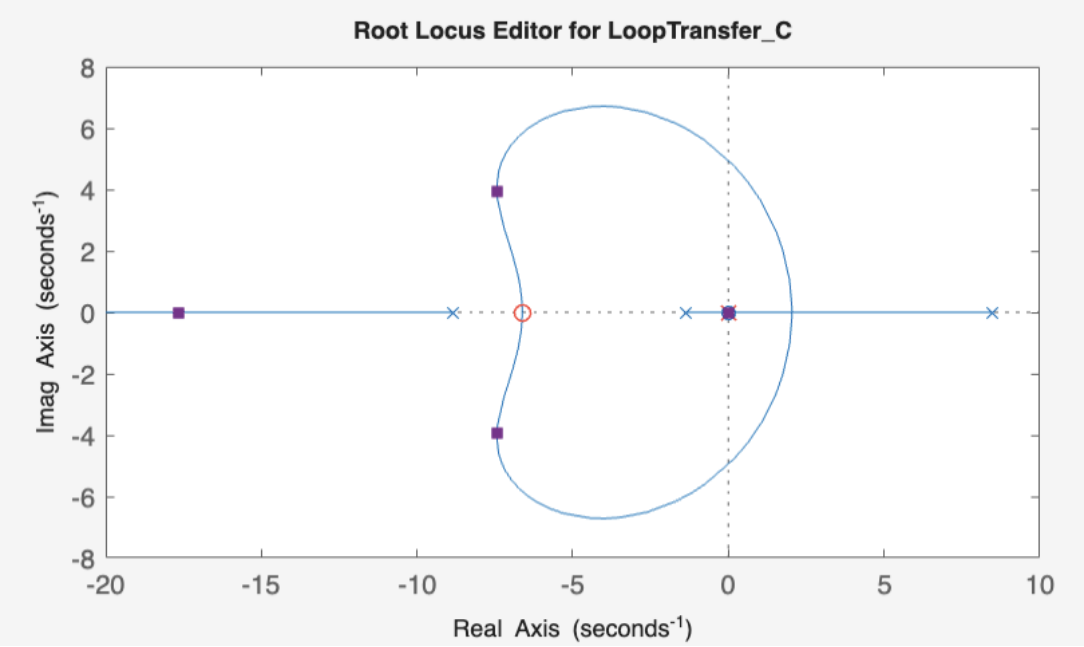
8.1 State-Space Modelling and Analytical PID Tuning

- The initial approach involved deriving a linearized state space representation of the robot using the Euler–Lagrange formulation. This linear model served as the basis for defining the plant system function, which was subsequently used to analytically tune a PID controller.
- The Routh–Hurwitz stability criterion was applied to determine the stability conditions and establish lower bounds for the controller parameters. Initial parameter estimates were obtained through manual calculations, and the system's step response was analysed to refine the gains.
- Once a satisfactory transient and steady state response was observed, the controller was implemented on the physical robot. Although the controller performed adequately for small angular deviations, it failed

to maintain balance during large tilts, indicating limitations of the linearized model in capturing the robot's full dynamics.

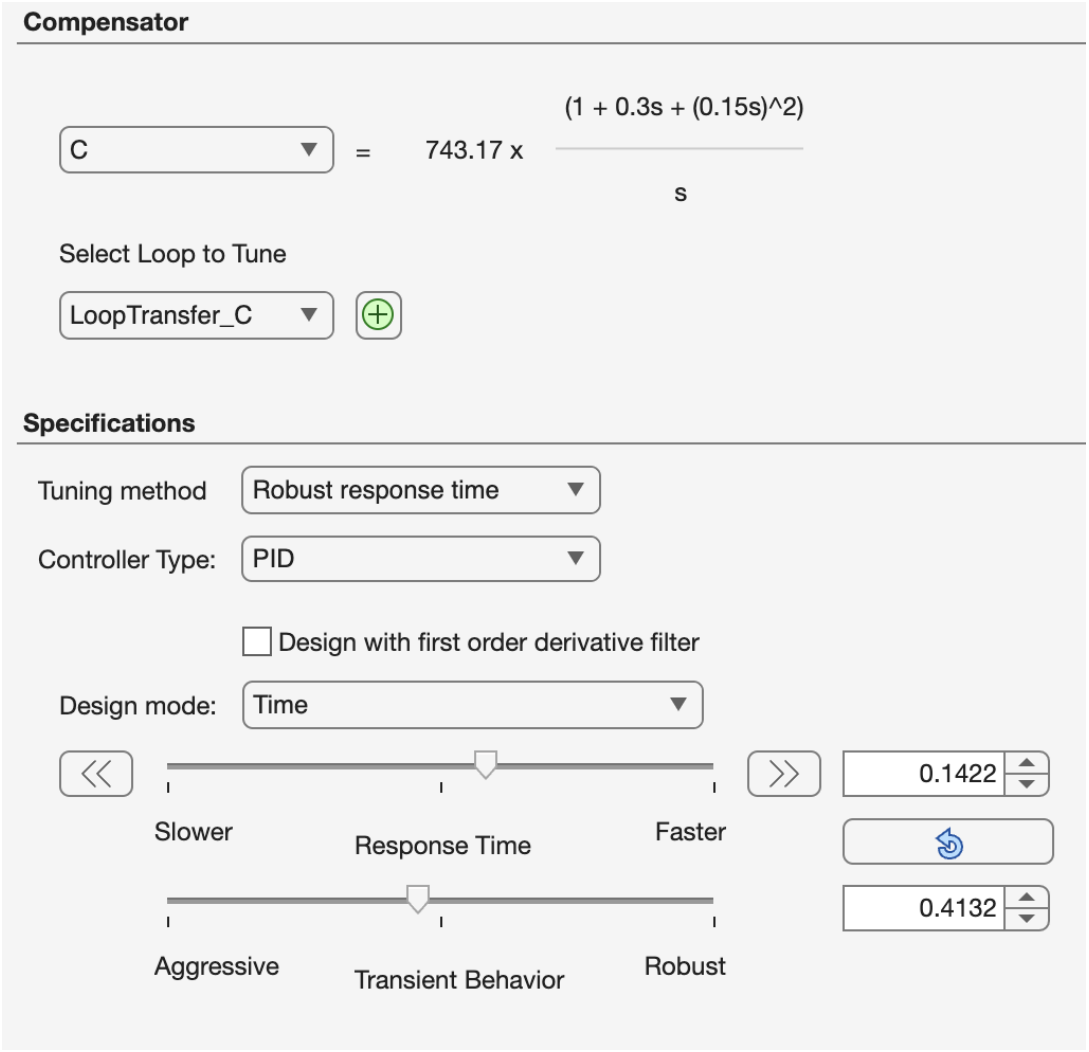


(a) Root Locus plot of Plant

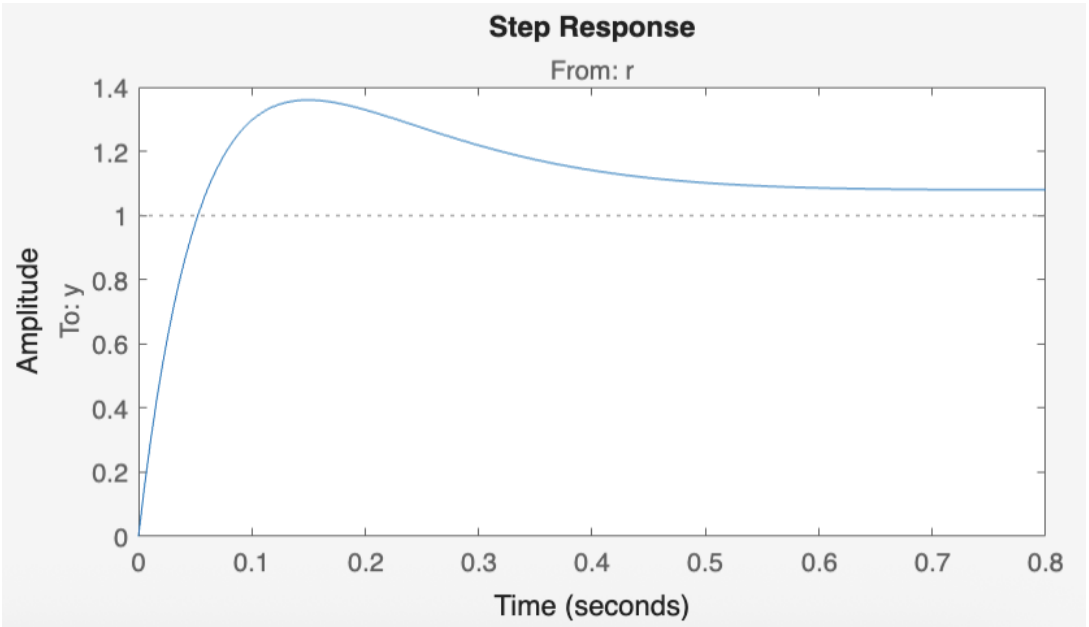


(b) Root Locus plot with Controller

Figure 22: Root Locus Plots



(a) Controller Settings

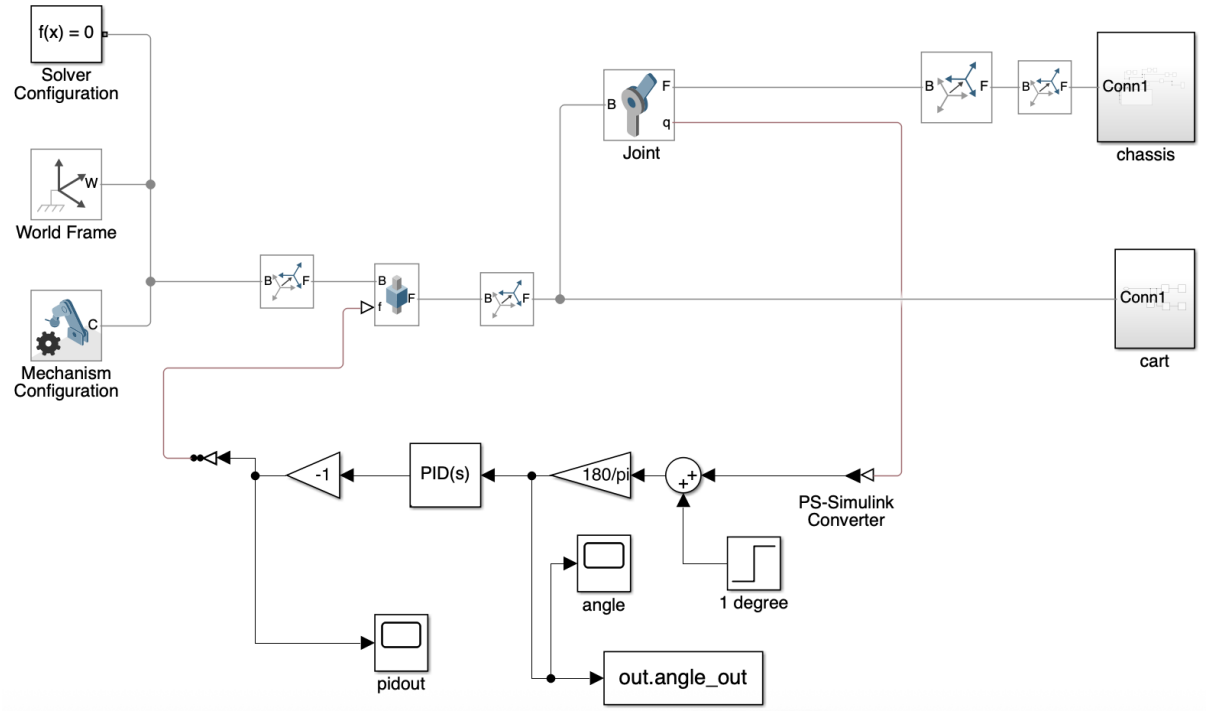


(b) Step Response with Controller

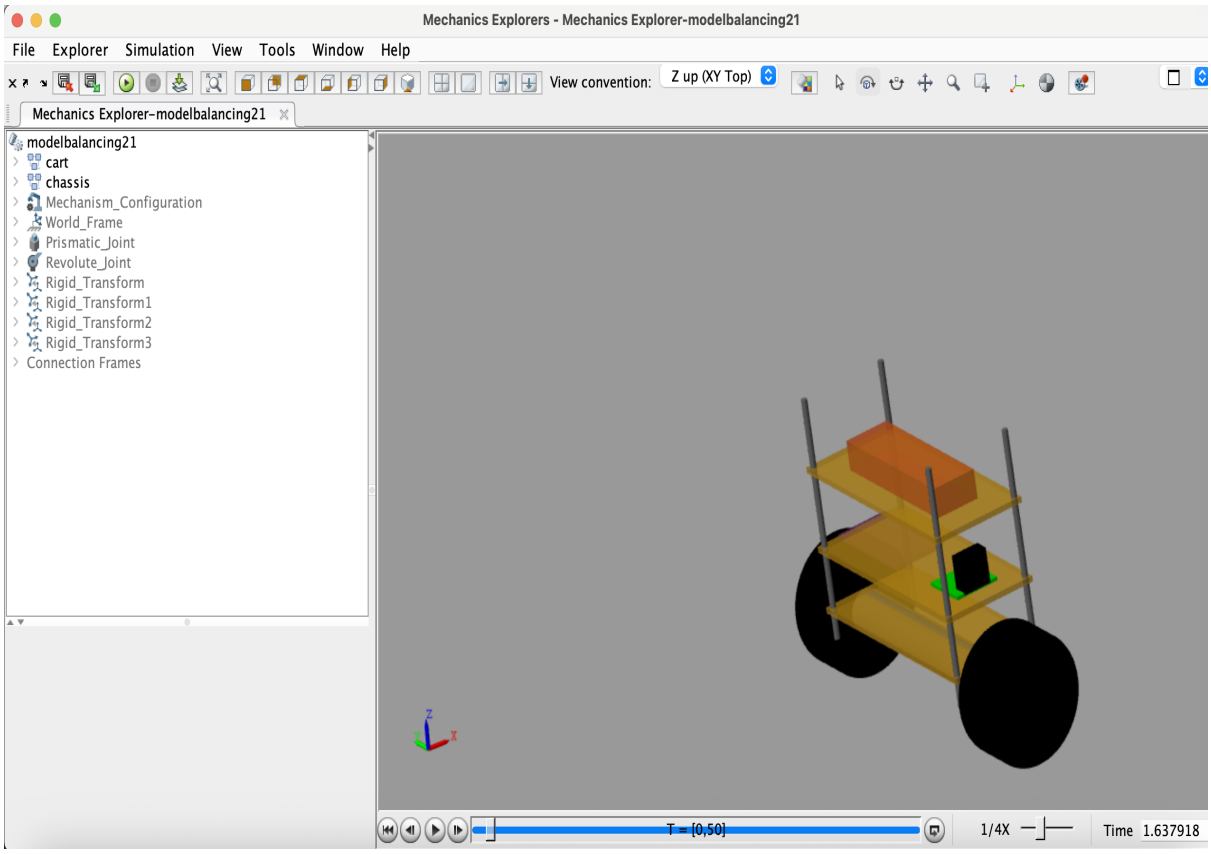
Figure 23: Controller Configuration and System Response

8.2 Simscape-Based Physical Modelling

- The second approach leveraged MATLAB's Simscape environment to construct a detailed physical model of the robot. The model incorporated all relevant physical properties, including mass, rotational inertia, and geometric dimensions of each component.
- An attempt was made to extract a transfer function for the plant from the Simscape model; however, due to the inherent nonlinearity of the system, this process was unsuccessful.
- As a result, PID parameters were tuned heuristically using a trial and error method.
- Despite this limitation, Simscape provided valuable visualization capabilities. It enabled real time observation of the robot's angular deviation and controller output, facilitating an intuitive understanding of the control system's performance in a simulated environment.
- More information regarding the modelling procedure can be referred to in Appendix C

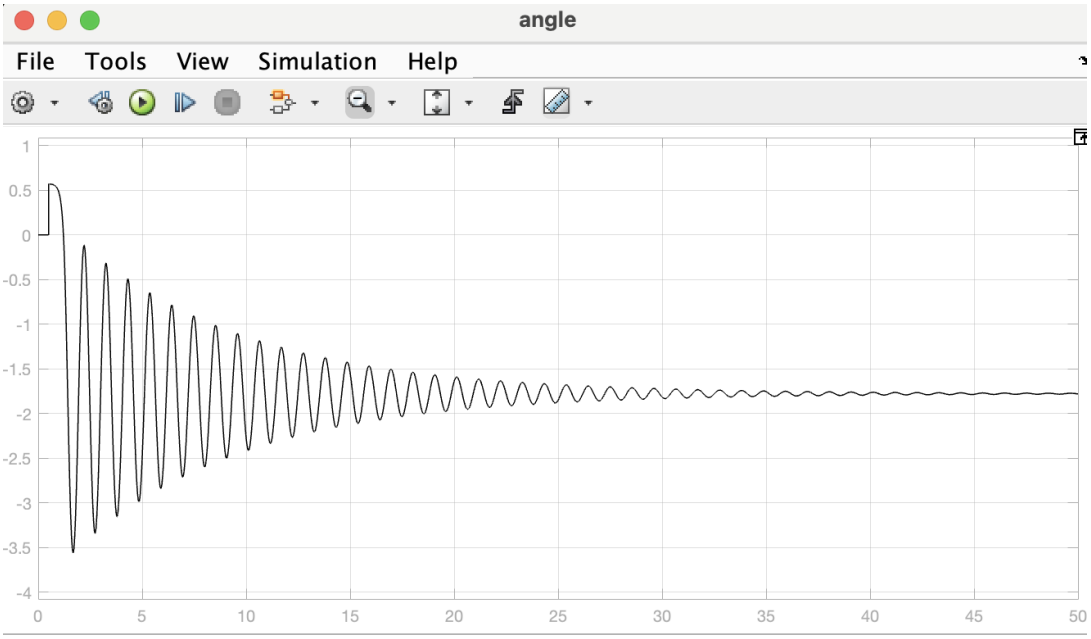


(a) Simulink modelling

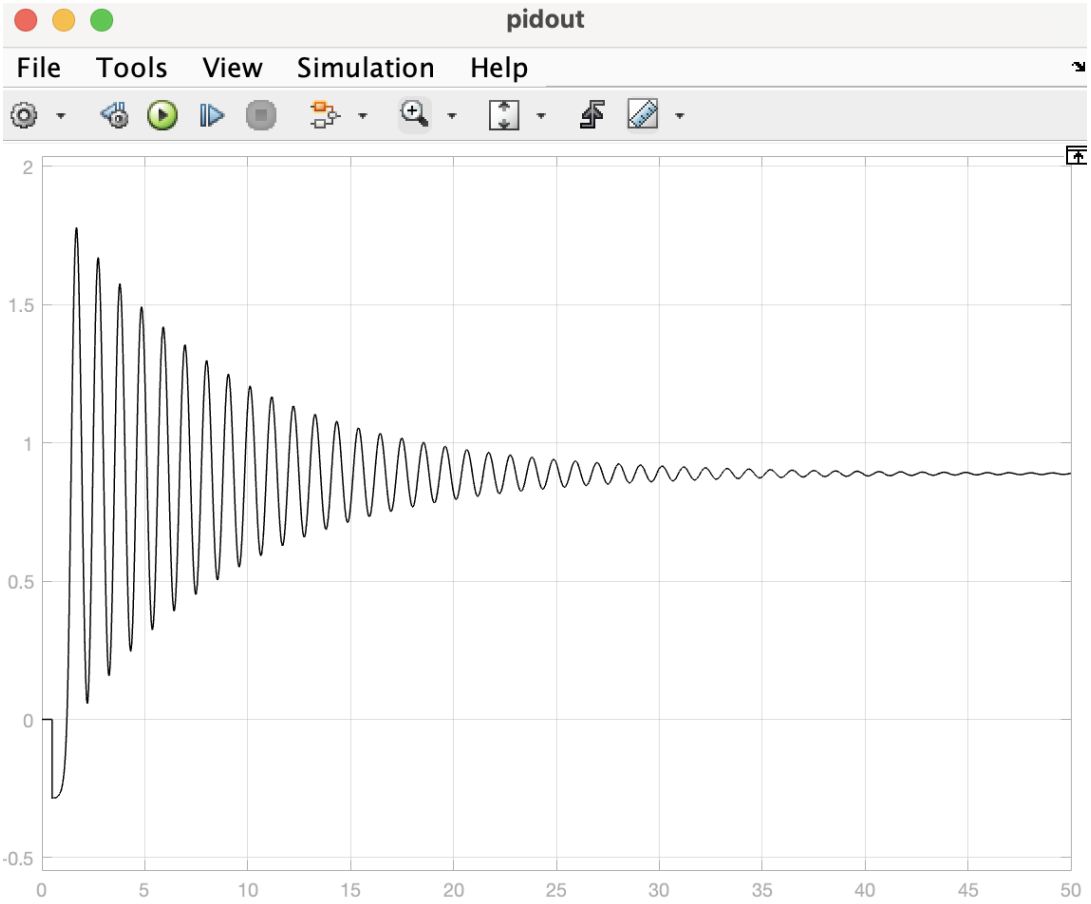


(b) Simulation

Figure 24: Simscape-based simulation setup of the Robot

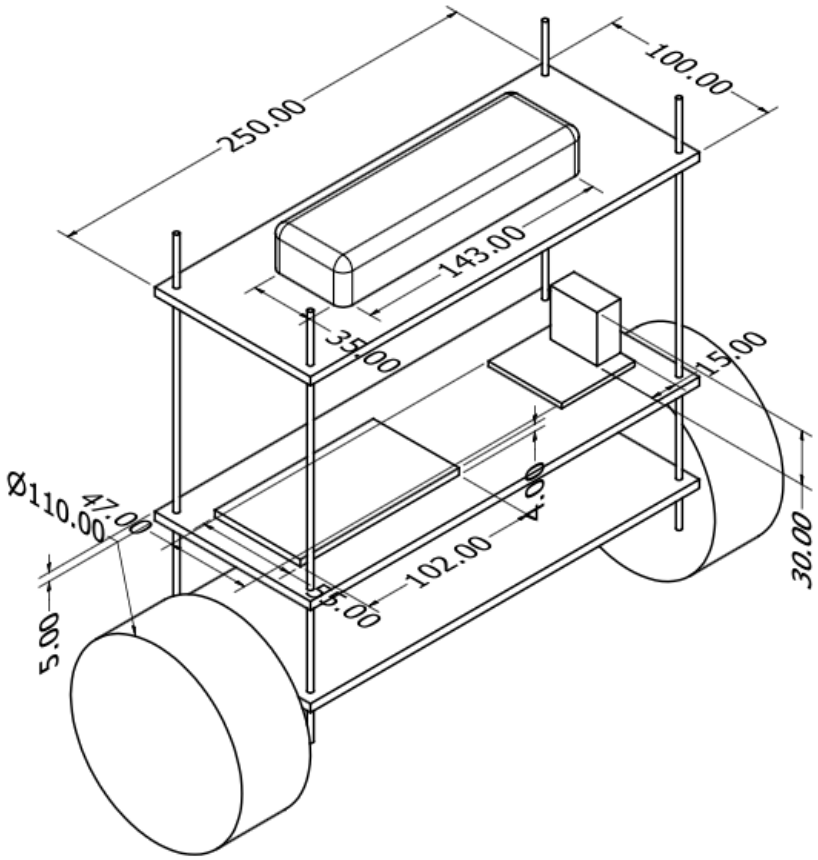


(a) Angular Response

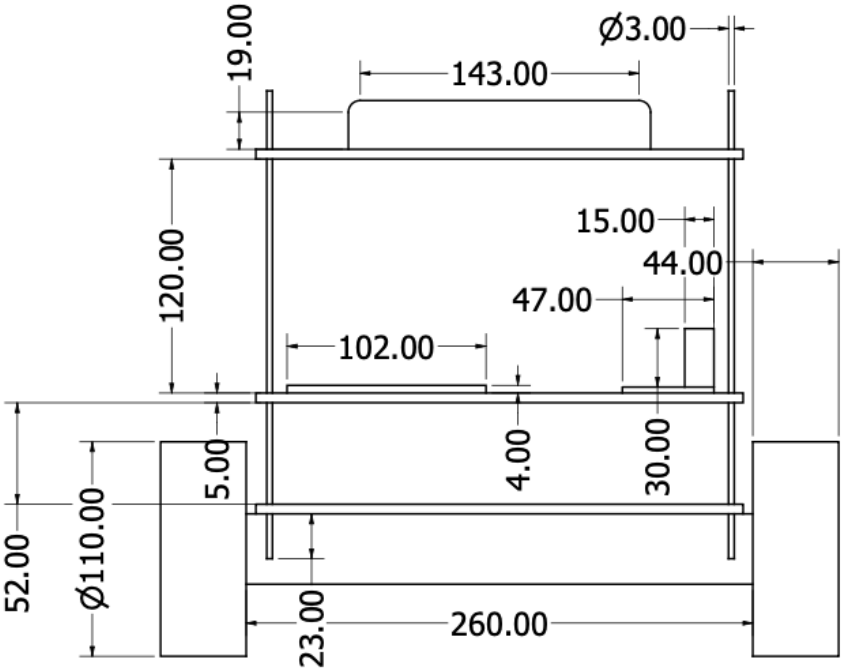


(b) PID Output Response

Figure 25: System Response and PID Output

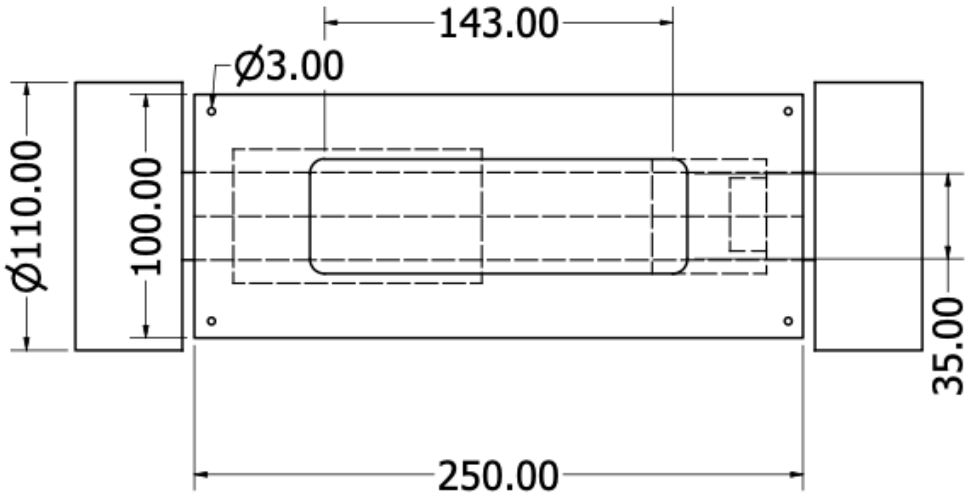


(a) Isometric View

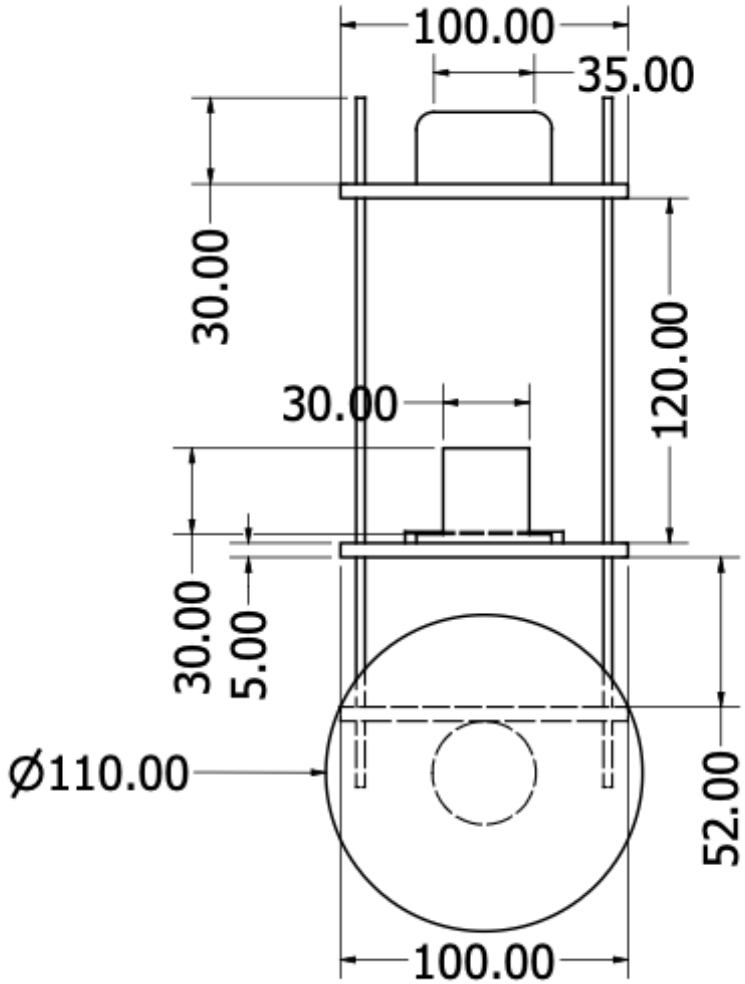


(b) Front View

Figure 26: Isometric and Front Views of the Robot



(a) Top View



(b) Side View

Figure 27: Top and Side Views of the Robot

9 Code Overview

The code given above implements a real time control system for a two wheeled self balancing robot equipped with an MPU6500 inertial measurement unit (IMU) and a set of three infrared (IR) sensors for path following. The system integrates sensor calibration, Kalman filtering for tilt estimation, PID control for vertical stabilization, and rule based trajectory tracking for path navigation.

Pseudo Code:

```
## Algorithm Overview
```
INITIALIZE:
    Configure motors, IR sensors, IMU
    Calibrate IMU offsets (200 samples)
    Initialize Kalman filters and PID variables

MAIN LOOP:
    Read IMU -> Apply offsets -> Calculate angles
    Filter angles using Kalman filter
    Balance control using PID
    IF balanced: Read IR sensors for path following
    ELSE: Stability mode
    Set motor speeds and actuate
```

## Detailed Algorithm

### Setup Phase
```
BEGIN Setup
    Initialize hardware (motors, sensors, IMU)
    Calibrate IMU:
        FOR 200 samples: sum accelerometer readings
        Calculate offsets = averages - gravity compensation
    SET calibrated = TRUE
END Setup
```

### Main Control Loop
```
BEGIN Main Loop
    // Sensor Reading & Processing
    Read IMU data (accel, gyro)
    Apply calibration offsets
    Calculate pitch/roll from accelerometer
    Filter angles using Kalman filter

```
```

```
    // Balance Control
    CALL Balance_PID(pitch)

    // Line Following (if balanced)
    IF tilt_excessive THEN
        mode = STABILITY
    ELSE
        Read IR sensors (left, mid, right)
        mode = Line_Detection(IR_values)
    END IF

    // Motor Control
    Calculate motor speeds based on mode + balance_output
    Actuate motors
END Main Loop
```

```
### Balance Control Function
```
```
FUNCTION Balance_PID(pitch)
    IF |pitch| > 35 THEN
        motor_output = 0, mode = EMERGENCY_STOP
    ELSE IF |pitch| > 17.5 THEN
        motor_output = MAX_SPEED * sign(pitch), mode =
           EXCESSIVE_TILT
    ELSE IF |pitch| > 8.5 THEN
        motor_output = HIGH_SPEED * sign(pitch), mode = HIGH_TILT
    ELSE IF |pitch| > 1.2 THEN
        // Standard PID Control
        error = -pitch * PI/180
        integral += Ki * error
        derivative = (error - last_error) / dt
        motor_output = Kp*error + Ki*integral*dt + Kd*derivative
        motor_output = CONSTRAIN(motor_output, -250, 250)
    ELSE
        motor_output = 0, integral = 0
    END IF
    RETURN motor_output
END FUNCTION
```

```
### Line Following Function
```
```
FUNCTION Line_Detection(left, mid, right)
    SWITCH (left, mid, right)
        CASE (0,0,1): RETURN SHARP_RIGHT
        CASE (0,1,0): RETURN STRAIGHT
        CASE (0,1,1): RETURN RIGHT
        CASE (1,0,0): RETURN SHARP_LEFT
        CASE (1,1,0): RETURN LEFT
        DEFAULT: RETURN STOP
```

```
    END SWITCH
END FUNCTION
```

Motor Speed Control
```
FUNCTION Motor_Control(mode, balance_output)
    base_left = base_right = balance_output

    SWITCH mode
        CASE STRAIGHT:
            left_speed = base_left + straight_speed
            right_speed = base_right + straight_speed
        CASE LEFT/SHARP_LEFT:
            left_speed = base_left - turn_speed
            right_speed = base_right + turn_speed
        CASE RIGHT/SHARP_RIGHT:
            left_speed = base_left + turn_speed
            right_speed = base_right - turn_speed
        DEFAULT: // STABILITY
            left_speed = base_left
            right_speed = base_right
    END SWITCH

    Set_Motors(left_speed, right_speed)
END FUNCTION
```

Kalman Filter (Simplified)
```
FUNCTION Kalman_Filter(measured_angle, gyro_rate, dt)
    // Predict
    angle += (gyro_rate - bias) * dt

    // Update covariance
    Update prediction covariance matrix

    // Kalman gain
    innovation = measured_angle - angle
    kalman_gain = Calculate from covariance and noise

    // Correct
    angle += kalman_gain * innovation
    bias += kalman_gain * innovation

    Update covariance matrix
    RETURN filtered_angle
END FUNCTION
```

Key Parameters

```
```
PID Constants: Kp=450, Ki=750, Kd=0.1
Angle Thresholds: 1.2, 8.5, 17.5, 35
Motor Speeds: straight=60, turn=75, max=250
Sample Rate: 50Hz (20ms loop)
```
```

Listing 1: Pseudo code

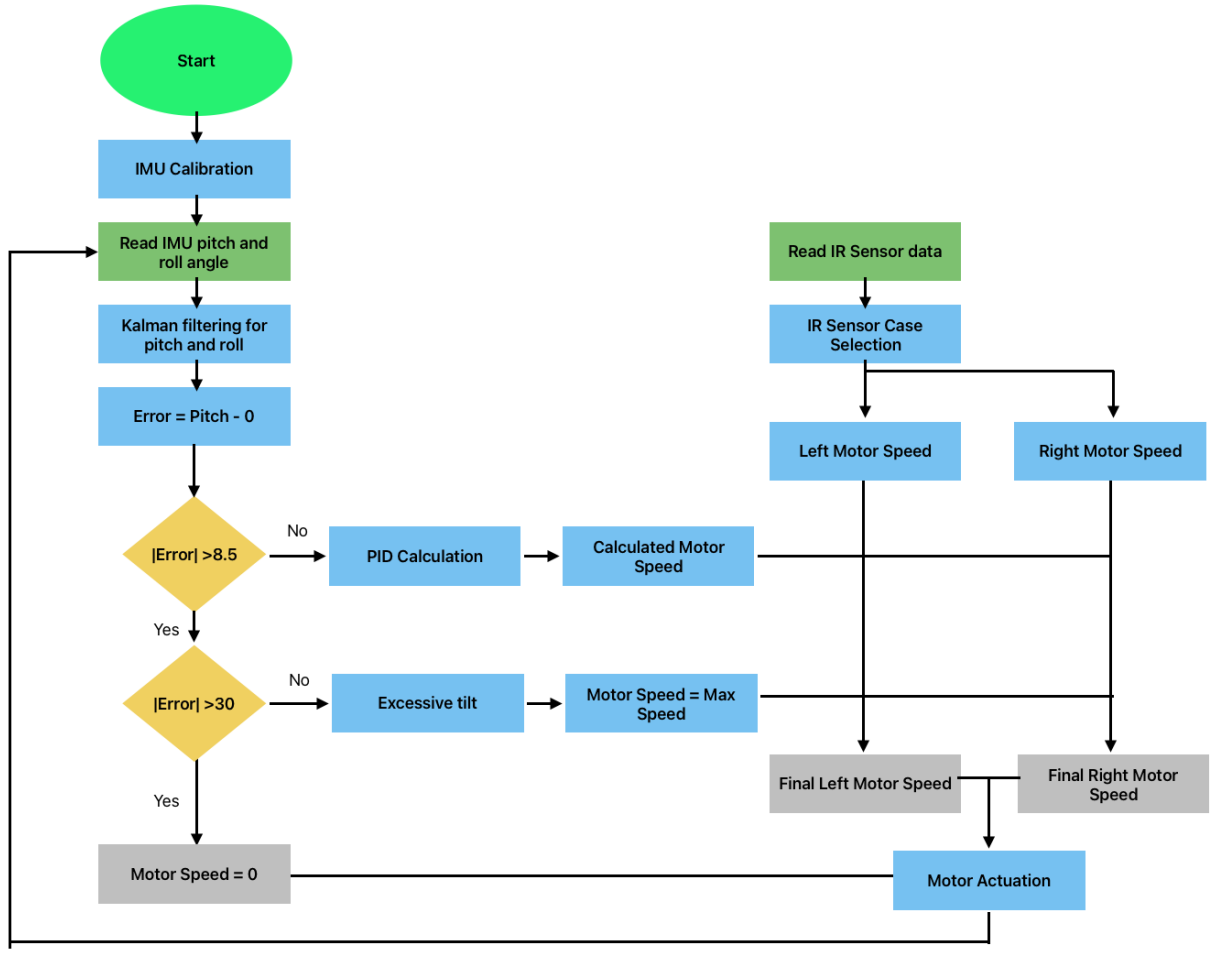


Figure 28: Code Overview flowchart

9.1 System Components

- **Actuation:** The robot utilizes two DC motors, controlled through H-bridge configurations with PWM modulation. Separate pins are configured for direction and speed control of each motor.
- **Sensing:**
  - **IMU (MPU6500):** Provides 6-DOF motion sensing. Accelerometer and gyroscope data are used to compute pitch and roll.

– **IR Sensors:** Positioned left, centre, and right to detect the position of a path on the ground, enabling trajectory decisions.

9.2 Kalman Filtering

Two Kalman filters (for pitch and roll) are implemented to fuse accelerometer and gyroscope data, yielding robust tilt angle estimates. The filter parameters ($Q_{angle}, Q_{bias}, R_{measure}$) were empirically tuned for real time estimation.

9.3 Sensor Calibration

Upon initialization, the accelerometer is calibrated by averaging 200 samples to compute static bias values for each axis. These offsets are subtracted from future readings to improve accuracy in tilt calculations.

9.4 PID Control for Balance

A PID controller stabilizes the robot's vertical orientation using the estimated pitch angle. The control signal is computed as:

$$\text{Output} = K_p \cdot e(t) + K_i \cdot \int e(t)\,dt + K_d \cdot \frac{de(t)}{dt}$$

Where e(t) is the angular error. Additional logic handles anti windup and safety by modifying motor speed or triggering an emergency stop under extreme tilt conditions.

9.5 Path Following Strategy

The robot classifies path alignment into six discrete cases based on IR sensor input. Each case maps to a specific motion directive (e.g., go straight, turn left/right sharply), implemented by modifying motor speeds in a differential manner. These commands are superimposed on the PID generated motor speed to ensure simultaneous path following and balance.

9.6 Control Loop Execution

In the main loop:

- IMU data is read and preprocessed.
- Kalman filters estimate pitch and roll.

- The pitch is used for PID based motor speed control.
- Line following logic selects a directional response.
- The final motor commands are generated and applied.
- Diagnostic information (e.g., angles, speeds, mode) is printed via serial interface.

10 Performance Evaluation

10.1 Vertical Stability

The robot exhibits satisfactory balancing behaviour for moderate angular deviations (pitch $<\sim 8.5°$). The Kalman filter significantly smoothens sensor noise and provides reliable tilt estimation. The PID controller, with tuned gains (K_p=441.4747, K_i=749.905, K_d=0.01), responds effectively to small to medium perturbations, restoring the robot to upright posture.

- **Response time (to $\pm 5°$ tilt):** < 0.3 seconds
- **Settling time:**$\sim$ 0.6–1.0 seconds depending on inertia and terrain
- **Steady state error:** $0.3°$

However, in the case of large deviations (pitch $> 17.5°$), the robot transitions to a safety mechanism that includes either:

- A high-speed corrective drive in the fall direction
- A complete emergency stop (beyond $35°$)

This prevents hardware damage but also indicates that the system's linearized PID design is limited to small-angle dynamics.

10.2 Path Following Performance

The robot accurately tracks a black path over a white background using a three-sensor IR module. The path control logic operates in six discrete states (stop, straight, turn right, turn left, sharp turns), and adjusts motor speeds accordingly.

- **Path reacquisition time:** < 100 ms after deviation
- **Tracking precision:** $> 75\%$ accuracy on curved paths under uniform lighting

- **Switching delay between modes:** $\sim$ 1–2 loop cycles ($\sim$ 2–4 ms)

The combined use of PID control and path following does not significantly degrade performance of either subsystem due to intelligent signal superposition and conditional logic.

10.3 System Robustness

- **Sensor calibration:** Averaging over 200 samples at startup ensures reliable accelerometer offsets
- **Noise resilience:** Kalman filter efficiently suppresses high-frequency jitter

10.4 Limitations

- Performance degrades in uneven terrain due to absence of dynamic gain adaptation.
- IR sensor thresholds are sensitive to ambient light and may require tuning for different environments.
- High PID gains may lead to overshoot in low-friction or high-speed conditions.

11 Challenges faced

Designing and stabilizing a self-balancing two-wheeled robot that can also follow a path autonomously is inherently multi-disciplinary and non-trivial. During the course of this project, we encountered a number of significant challenges across the domains of mechanical design, control algorithm implementation, sensor fusion, and system integration. These are detailed below:

- **Structural Flex and Material Limitations**

  The chassis was constructed using acrylic sheets for ease of prototyping. However, under the weight of the onboard battery and during abrupt motor torque changes, the structure exhibited visible flexing. This led to misalignment between the wheels and caused the center of gravity (CoG) to dynamically shift during operation. The resultant mechanical deformation introduced disturbances that the control algorithm was not designed to handle, degrading both balance and manoeuvrability. A stiffer material or reinforced frame would alleviate this issue in future iterations.

- **Center of Gravity Instability Due to Battery Position**

  The battery pack, being the heaviest component, significantly influenced the CoG. Different placements (high vs. low, forward vs. backward) changed the robot's dynamic behaviour and stability margins. This affected the accuracy of the linearized control model and demanded different PID gain values for stability. Since real-time gain tuning is impractical during operation, this highlighted the need for either a fixed optimized CoG or the use of a gain-scheduled control strategy that adapts to system parameters.

- **Actuator Saturation and Non linear Behaviour**

  The chosen motors, while cost-effective, provided limited torque. To achieve acceptable corrective action during large tilt disturbances, high PID gains were required. However, this often caused the PWM signal to saturate, effectively rendering the controller non linear and reducing its performance to near bang-bang control in some regimes. Moreover, dead zones and backlash further complicated precise motion. Anti-windup mechanisms and torque-linearizing compensators were identified as potential improvements.

- **Kalman Filter Tuning and Sensor Drift**

  Implementing a Kalman filter for tilt estimation was necessary due to significant noise in raw IMU readings. However, tuning the filter—specifically the process noise (Q) and measurement noise (R) covariance matrices—proved difficult. Poor tuning either led to noisy estimates or sluggish response. Additionally, gyro drift and vibration-induced bias worsened over time. These issues highlighted the need for a more advanced filtering approach, potentially involving adaptive or extended Kalman filtering, along with physical damping of the IMU mount.

- **Timing Constraints and Limited Processing Power**

  The Arduino Mega 2560, while convenient, struggled to handle high rate control loop execution, sensor acquisition, Kalman filtering, and path following logic simultaneously. This led to variable loop timing (jitter), degrading control loop determinism and introducing phase lag. Interrupt-based timing and optimization of loop efficiency were partially effective, but a more powerful microcontroller would be necessary for robust real-time performance.

- **Conflict Between Balance and Path Following**

  Balance control required high responsiveness and authority over motor speeds. However, simultaneous path following control introduced disturbances-especially sharp turning commands that counteracted balance corrections. This coupling resulted in wobbling or instability, particularly during aggressive path corrections. The current approach used simple superposition of signals; a hierarchical or coordinated control structure would be better suited to handle such multi-objective control tasks.

- **Infrared Sensor Limitations and Environmental Sensitivity**

  The IR sensors used for path detection were susceptible to ambient light fluctuations and surface reflectivity changes. These variations occasionally caused false detections or missed path boundaries. In addition, the sensor spacing and threshold logic, while functional on gentle curves, struggled with tight turns or complex intersections. Robust threshold, shielding against light, and possibly switching to camera-based systems were considered for future development.

- **Wheel Misalignment and Assembly Tolerances**

Minor inaccuracies during mechanical assembly led to slight misalignment in the wheel axes. This introduced subtle but persistent lateral drift or asymmetric motor behaviour. Because the robot's dynamics are highly sensitive to symmetry, this mechanical imperfection had a non-negligible impact on stability. Improved fixtures and alignment procedures are necessary for precision builds.

12 Conclusion

This report presented the design, implementation, and evaluation of a two wheeled self balancing robot equipped with an IR-based path following system. By integrating a classical PID controller for tilt stabilization with a rule based approach for path guidance, the system achieved reliable performance on a low cost embedded platform. Empirical tests demonstrated that with appropriate gain tuning and sensor fusion via a Kalman filter, the robot could maintain upright balance and follow moderately curved paths under varying surface and lighting conditions.

However, the study also highlighted several challenges, including the effects of structural flex under load, sensor noise, and controller limitations at large tilt angles. These observations underscore the importance of mechanical rigidity and robust estimation in control oriented robotic systems.

The results reinforce that classical control strategies, when carefully implemented and tuned, remain effective for real time embedded applications. Future work will explore gain scheduling techniques, adaptive filtering, and more robust path tracking algorithms to extend system stability and path tracking accuracy under a wider range of operating conditions. Additionally, quantitative comparisons against non linear and model predictive controllers could further contextualize the efficacy of the proposed approach.

A Appendix

A.1 Appendix A: MATLAB Code for Review

```
clc
clear all
close all

mbattery = 0.432;
m_motor = 0.458;
Mw = 0.106; % mass of wheel
Mbody = 2.243-(2*(m_motor+Mw)); % mass of robot(including battery
   ) excluding wheel and motor
Ke = 0.118; %back emf constant
Rm = 5.6; %terminal resistance
wheeldia = 0.100;
rw = wheeldia/2; %radius of wheel
Kb = Ke^2/(Rm*rw); % back emf related something
hcom = 0.111; %cog from the centre of wheel
Ibody = 0.01306; % body inertia
Itotal = Ibody + Mbody*(hcom^2); % total pitch

Jw = 0.5*Mw*(rw^2); % wheel inertia
Jm = 2.66*(10^-5); % rotor inertia
Meq = Mbody + (2*Mw)+ (2*(Jw+Jm)/(rw^2)); % total mass
Km = Ke/Rm ; % motor constant
delta = Meq*Itotal - (Mbody*hcom)^2;
g = 9.81; % acc due to gravity

A= [0 1 0 0;
    0 ((1/delta)*(2*Kb*Mbody*hcom - (2*Kb*Itotal/rw))) (Mbody*
       hcom)^2*g/delta 0;
    0 0 0 1;
    0 ((1/delta)*(2*Kb*Meq - (2*Kb*Mbody*hcom/rw))) Meq*Mbody*g*
       hcom/delta 0];

B = [0;
    ((1/delta)*(-2*Km*Mbody*hcom + (2*Km*Itotal/rw)));
    0;
    ((1/delta)*(-2*Km*Meq + (2*Km*Mbody*hcom/rw)))];
C = [0 0 1 0];
D =0;

sys=ss(A,B,C,D);
systf=tf(sys)

ki =749.905;
kp=0.1 +(27.2684+1.819*ki)/3.1523;
kd=0;

den = [1 0];
```

```
num = [kd kp ki];
controller_tf = tf(num,den);
full_tf = systf * controller_tf;
closedloop_tf = feedback(full_tf,1);

stepplot(closedloop_tf)
sisotool(systf)
```

Listing 2: PID Controller Code

A.2 Appendix B: Arduino Code for Motor Control

```
#include <Wire.h>
#include "mpu6500.h"

#define ENA 2 //motor 0 (right)
#define INA 12
#define INB 11
#define ENB 3 //motor 1(left)
#define INC 6
#define IND 7
#define pin_left 49
#define pin_mid 53
#define pin_right 51

// MPU6500 object
bfs::Mpu6500 imu;
// Calibration offsets
float accel_x_offset = 0, accel_y_offset = 0, accel_z_offset = 0;
bool calibrated = false;

// Kalman filter class
class KalmanFilter {
  public:
    float Q_angle = 0.001;
    float Q_bias = 0.003;
    float R_measure = 0.03;

    float angle = 0;
    float bias = 0;
    float rate = 0;

    float P[2][2] = {{0, 0}, {0, 0}};

    float getAngle(float newAngle, float newRate, float dt) {
      rate = newRate - bias;
      angle += dt * rate;

      P[0][0] += dt * (dt*P[1][1] - P[1][0] - P[0][1] + Q_angle);
      P[0][1] -= dt * P[1][1];
```

```
39        P[1][0] -= dt * P[1][1];
40        P[1][1] += Q_bias * dt;
41
42        float y = newAngle - angle;
43        float S = P[0][0] + R_measure;
44        float K[2];
45        K[0] = P[0][0] / S;
46        K[1] = P[1][0] / S;
47
48        angle += K[0] * y;
49        bias += K[1] * y;
50
51        float P00_temp = P[0][0];
52        float P01_temp = P[0][1];
53
54        P[0][0] -= K[0] * P00_temp;
55        P[0][1] -= K[0] * P01_temp;
56        P[1][0] -= K[1] * P00_temp;
57        P[1][1] -= K[1] * P01_temp;
58
59        return angle;
60    }
61};
62
63KalmanFilter kalmanPitch;
64KalmanFilter kalmanRoll;
65
66void setupMotors() {
67  pinMode(ENA, OUTPUT);
68  pinMode(INA, OUTPUT);
69  pinMode(INB, OUTPUT);
70  pinMode(ENB, OUTPUT);
71  pinMode(INC, OUTPUT);
72  pinMode(IND, OUTPUT);
73}
74
75void setIRs_line(){
76  pinMode(pin_left, INPUT);
77  pinMode(pin_mid, INPUT);
78  pinMode(pin_right, INPUT);
79  delay(50);
80}
81int line_checker_ir(bool ir_left,bool ir_mid,bool ir_right){
82  if(ir_left==0 && ir_mid==0 && ir_right==1){return 4;} // turn
       sharp right
83  else if(ir_left==0 && ir_mid==1 && ir_right==0){return 1;} //
       go straight
84  else if(ir_left==0 && ir_mid==1 && ir_right==1){return 5;} //
       turn right
85  else if(ir_left==1 && ir_mid==0 && ir_right==0){return 2;} //
       turn sharp left
```

```
  else if(ir_left==1 && ir_mid==1 && ir_right==0){return 3;} //
     turn left
  else{return 0;} // stop
}

//0 : ena motor (right motor)
//1 : enb motor (left motor)
void setMotorSpeed(int speed,bool motor) {
  speed = int(speed);
  speed = constrain(speed, -250, 250);

  if (motor){
  if (speed < 0) {
    digitalWrite(INA, HIGH);
    digitalWrite(INB, LOW);
  } else {
    digitalWrite(INA, LOW);
    digitalWrite(INB, HIGH);
  }
  analogWrite(ENA, abs(speed));
}
else {
  if (speed < 0) {
    digitalWrite(INC, HIGH);
    digitalWrite(IND, LOW);
  } else{
    digitalWrite(INC, LOW);
    digitalWrite(IND, HIGH);
  }
  analogWrite(ENB, abs(speed));
}
}
int left_or_right = 1; //1 or -1 based on testing
int straight_speed = 60; // Speed for straight movement
int low_turn_speed = 50; // Speed for turns
int  sharp_turn_speed = 75; // Speed for sharp turns
int motorSpeed = 0;
int delayTime = 3600;

int left_speed=0;
int right_speed=0;

void motor_speed_ctrl(int line_case,int motorSpeed){
  switch(line_case) {
    case 0: // stability
    left_speed =motorSpeed;
    right_speed =motorSpeed;
    break;
    case 1: // straight
    left_speed = motorSpeed + (left_or_right * straight_speed);
    right_speed = motorSpeed + (left_or_right * straight_speed);
```

```
136     break;
137     case 2: // turn right
138     left_speed = motorSpeed + (left_or_right * low_turn_speed);
139     right_speed = motorSpeed - (left_or_right * low_turn_speed);
140     break;
141     case 3: // turn sharp right
142     left_speed = motorSpeed + (left_or_right * sharp_turn_speed);
143     right_speed = motorSpeed - (left_or_right * sharp_turn_speed)
            ;
144     break;
145     case 4: // turn left
146     left_speed = motorSpeed - (left_or_right * low_turn_speed);
147     right_speed = motorSpeed + (left_or_right * low_turn_speed);
148     break;
149     case 5: // turn sharp left
150     left_speed = motorSpeed - (left_or_right * sharp_turn_speed);
151     right_speed = motorSpeed + (left_or_right * sharp_turn_speed)
            ;
152     break;
153     default: // stability
154     left_speed = motorSpeed;
155     right_speed = motorSpeed;
156     break;
157   }
158   setMotorSpeed(left_speed,1);
159   setMotorSpeed(right_speed,0);
160   delayMicroseconds(delayTime);
161 }
162
163 // PID control parameters
164 float Kt = 0;// anti windup term
165 float Kp = 450.0, Ki = 750.0, Kd = 0.1;
166
167 // Angle Errors
168 float targetAngle = 0.0;
169 float error =0.0;
170 float lastError = 0;
171
172 // Time
173 unsigned int current_time = 0;
174 unsigned int lastTime = 0;
175 float dt =0;
176
177 // I/D components
178 float derivative =0;
179 float integral = 0;
180 float I =0;
181 float output =0;
182
183
184 // Angle thresholds
```

```
int tilt_marker =0; //0: no tilt, 1: tilt detected
int debug_reader =0;
float low_threshold = 1.2; //1.7;
float low2_threshold = 2.5; // for PI
float mid_threshold = 8.5;
float high_threshold = 17.5;
float highest_threshold = 35.0; // for emergency stop

float factor =1.0;
float x = 0.0;
short unsigned int first_max_speed = 230;
short unsigned int final_max_speed = 250;

void debug_print(int debug_reader){
switch(debug_reader){
  case 0: Serial.println("\tSTABILITY"); break;
  case 1: Serial.println("\tGOING STRAIGHT"); break;
  case 3: Serial.println("\tTURNING SHARP RIGHT"); break;
  case 2: Serial.println("\tTURNING RIGHT"); break;
  case 5: Serial.println("\tTURNING SHARP LEFT"); break;
  case 4: Serial.println("\tTURNING LEFT"); break;
  default: Serial.println("\tINVALID CASE: STABILITY"); break;
}
}

void pid(float pitch,float dt) {
  // Emergency stop if tilt too large
  if (abs(pitch) > mid_threshold && abs(pitch) < high_threshold)
     {
    tilt_marker = 1;
    float sign = pitch/(abs(pitch));
    motorSpeed = sign*first_max_speed;
    delayTime = 2900;
    Serial.println("\tEXCESSIVE TILT 1");
  }
  else if (abs(pitch) > high_threshold && abs(pitch) <
     highest_threshold) {
    tilt_marker = 1;
    float sign = pitch/(abs(pitch));
    motorSpeed = sign*final_max_speed;
    delayTime = 2900;
    Serial.println("\tEXCESSIVE TILT 2");
  }
  else if (abs(pitch) > highest_threshold) {
    tilt_marker = 1;
    motorSpeed = 0;
    delayTime = 2000;
    Serial.println("\tEMERGENCY STOP");
     }
  // PID control
```

```
233   else if (abs(pitch) > low_threshold && abs(pitch)<mid_threshold
        ) {
234     tilt_marker = 0;
235     delayTime = 3600;
236     if(pitch <0) {factor =1.0;}
237     else {factor = 1.0;}
238     if (abs(pitch) > low2_threshold || (pitch<0))
239       {
240         error = (targetAngle - pitch)*PI/180.0; // making it
              radians
241       integral += Ki*(error);
242       derivative = (error - lastError) / dt;
243       I = integral+ Kt*(x);
244       output = Kp * error + I*dt + Kd * derivative;
245       output*=(-1*factor); // -1 due to imu orientation
246       lastError = error;
247       motorSpeed = constrain(output, -250, 250);
248       //x = output - motorSpeed; // back calculation error for
            integral term
249       }
250       else motorSpeed = 0;
251     }
252   else {
253     tilt_marker = 0;
254     delayTime = 3600;
255     motorSpeed = 0;
256     integral = 0;
257      }
258 }
259
260 void setup() {
261   setIRs_line();
262   setupMotors();
263   Serial.begin(115200);
264   Wire.begin();
265   Wire.setClock(400000);
266
267   imu.Config(&Wire, bfs::Mpu6500::I2C_ADDR_PRIM);
268   if (!imu.Begin()) {
269     Serial.println("Error␣initializing␣IMU");
270     while (1);
271   }
272   // Set the sample rate divider
273   if (!imu.ConfigSrd(19)) {
274     Serial.println("Error␣configuring␣SRD");
275     while (1);
276   }
277
278   Serial.println("Calibrating...");
279   float ax_sum = 0, ay_sum = 0, az_sum = 0;
280   for (int i = 0; i < 200; i++) {
```

```
    while (!imu.Read()) {}
    ax_sum+= imu.accel_x_mps2();
    ay_sum+= imu.accel_y_mps2();
    az_sum+= imu.accel_z_mps2();
    delay(10);
  }
  accel_x_offset = ax_sum/200;
  accel_y_offset = ay_sum/200;
  accel_z_offset = az_sum/200 - 9.81;
  calibrated = true;

  Serial.println("Calibration complete.");
  delay(500);
  Serial.print("Accel X Offset: ");
  Serial.println(accel_x_offset, 6);
  Serial.print("Accel Y Offset: ");
  Serial.println(accel_y_offset, 6);
  Serial.print("Accel Z Offset (gravity removed): ");
  Serial.println(accel_z_offset, 6);
delay(500);
lastTime = millis();
}

void loop() {
  if (imu.Read() && calibrated) {
    current_time = millis();
    dt = (current_time - lastTime) / 1000.0; //seconds
    lastTime = current_time;
    float ax = imu.accel_x_mps2() - accel_x_offset;
    float ay = imu.accel_y_mps2() - accel_y_offset;
    float az = imu.accel_z_mps2() - accel_z_offset;
    float gx = imu.gyro_x_radps() * 180.0/PI;
    float gy = imu.gyro_y_radps() * 180.0/PI;
    // Accelerometer-based pitch and roll (degrees)
    float pitchAcc = atan2(ay, sqrt(ax*ax + az*az)) * 180.0/PI;
    float rollAcc  = atan2(-ax, az) * 180.0/PI;

    // Kalman filter output
    float pitch = kalmanPitch.getAngle(pitchAcc, gx, dt); //
       Adjusted for initial tilt
    float roll = kalmanRoll.getAngle(rollAcc, gy, dt);
    // Output
    Serial.print("\tPitch: ");
    Serial.print(pitch, 3);
    Serial.print("   \tRoll: ");
    Serial.print(roll, 2);
    pid(pitch,dt); // (pitch in deg). converted to rad in pid
       func
    if (tilt_marker ==1){
      debug_reader = 0;
    }
```

```
330     else {
331       debug_reader = line_checker_ir(digitalRead(pin_left),
             digitalRead(pin_mid),digitalRead(pin_right));
332     }
333     motor_speed_ctrl(debug_reader,motorSpeed);
334     Serial.print("\tleft speed: ");
335     Serial.print(left_speed);
336     Serial.print("\tright speed: ");
337     Serial.print(right_speed);
338     debug_print(debug_reader);
339     Serial.println();
340   }
341 }
```

Listing 3: Arduino PWM Motor Control

A.3 Appendix C: Modelling in Simscape

1. Chassis Assembly Construction

   Create Main Frame:

   - Use Brick Solid block for acrylic plates
   - Define dimensions (length × width × thickness)
   - Set mass properties for acrylic material
   - Position at appropriate height above wheels using rigid transformation blocks

   Add Structural Support:

   - Use Cylinder Solid blocks for threaded rods
   - Create 4 vertical support posts
   - Use Rigid Transform blocks for precise positioning
   - Connect rods to main frame with rigid transforms

   Battery Pack (Orange Block):

   - Brick Solid with appropriate mass (heaviest component)
   - Position for optimal centre of gravity
   - Add mass properties based on actual battery specs

Arduino (Purple Block):

- Smaller Brick Solid representing microcontroller
- Position for realistic placement

Motor Driver (Green + Black Block):

- Brick Solid for L298N driver
- Appropriate dimensions and mass

Create Subsystem named "Chassis"

2. Cart/Wheel Assembly Wheel Creation:

- Use Cylinder Solid blocks for black wheels
- Define radius, width, and tire properties
- Set rotational inertia for realistic dynamics

Motor System:

- Gray Cylinder Solid for connecting axle
- Proper diameter and length
- Connect wheels to axle with appropriate rigid transforms

3. Joint Configuration - Critical Element Revolute Joint Setup:

- Connect chassis to cart through single revolute joint
- Set rotation axis (typically about wheel axle)
- This joint provides the tilt angle measurement
- Configure joint limits to -89 to 89 degrees for best performance
- Add damping for realistic behaviour

4. Sensor Implementation Tilt Angle Sensing:

- Use Transform Sensor on revolute joint
- Extract angular position (theta)

- Convert to appropriate units (radians)

5. PID controller

- Use the PID controller block and adjust the necessary coefficients for the stabilising output
- The input is the tilt angle and the output is the force required to move the robot front and back

Refer Figure 18(a) for control loop

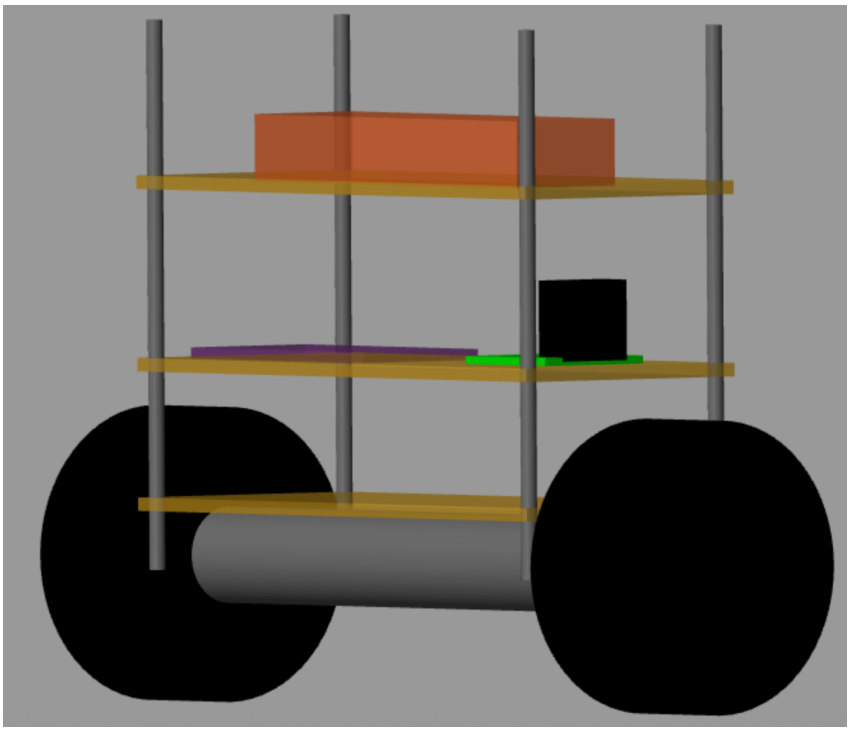


Figure 29: Simscape Model (A)

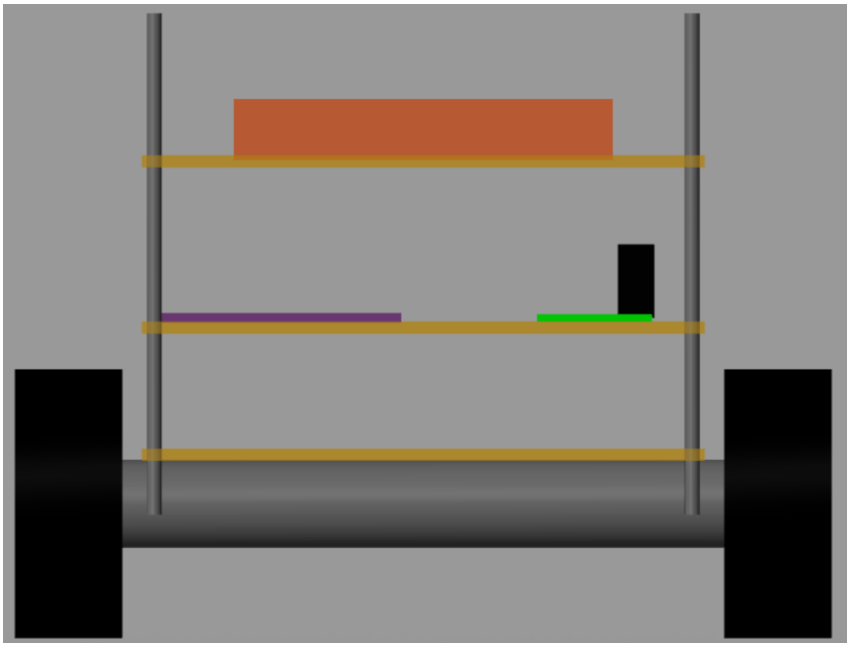


Figure 30: Simscape Model (B)

References

[1] Y.-S. Ha and S. Yuta, "Trajectory tracking control for navigation of the inverse pendulum type self-contained mobile robot," *Elsevier, Robotics and Autonomous Systems 17 (1996) 65-80*, 1996.

[2] J. Goring, "The design, modelling, construction and testing of a two wheeled self-balancing robot," April 2019. PhD Dissertation.

[3] M. O. Asali, F. Hadary, and B. W. Sanjaya, "Modeling, simulation, and optimal control for two-wheeled self-balancing robot," *International Journal of Electrical and Computer Engineering (IJECE) Vol. 7, No. 4, August 2017, pp. 2008 2017*, 2017.

[4] S. E. Arefin, "Simple two-wheel self-balancing robot implementation," 2017.

[5] P. Frankovský, L. Dominik, A. Gmiterko, I. Virgala, P. Kurylo, and O. Perminova, "Modeling of two-wheeled self-balancing robot driven by dc gearmotors," *Int. J. of Applied Mechanics and Engineering, 2017, vol.22, No.3, pp.739-747*, 2017.

[6] O. Halawa, A. Maged, A. Sabry, A. Gaber, A. Ramadan, and E. Tarik, "Mechatronics project: Two wheeled self-balancing robot," 2023.