Masters in Smart Systems

Winter Semester 2024-25

MASTER THESIS

Report on Autonomous Stair navigation for mobile robots using IMU

Submitted To

Prof. Dr. Christoph Uhrhan, HFU

Mr. Tobias Lorenz, HFU

Submitted by

Jaghan Prashanth Gandhi Venkateshwaran (273157)

DATE: 14.04.2025

Abstract

This thesis presents the design, implementation, and testing of a stair-climbing robot that can traverse complex terrain utilizing enhanced motor control and sensor feedback in real-time. The main purpose is to create a robot that can autonomously identify if it is navigating stairs, adjust motor speed and direction in real-time while keeping stable and using minimal energy. The system makes use of an MPU6050 IMU to determine stair encounters from positive spikes in Z-axis acceleration and to monitor pitch angle while riding the stairs to maintain balance during climbing. Initially the motor speed is defined by the user, but during stair climbing the motor speed will adjust without user input to create a smoother transition from flat surfaces to stairs or vice versa and to promote performance gains. The robot also supported user-defined rotation, using the IMU data to accurately rotate to a desired angle based on the robots pose or on the fly. Testing of the robot was done across multiple scenarios that included forward motion, turning, and climbing stairs with the robot. The results show that the system has limitations but is adaptable in numerous scenarios. This paper contributes to the existing knowledge in mobile robotics by providing a practical and versatile stair-climbing robot for real-world scenarios such as assistive technology, search and rescue, and urban autonomous delivery.

Declaration

I declare that this thesis titled " Autonomous Stair navigation for mobile robots using IMU" submitted in part fulfilment of the requirements for the award of master's at Furtwangen University is my original work and has not been submitted for the award of any other degree, diploma or title at any other institution.

This thesis presents my own research and findings conducted as part of the design, development, testing and analysis of the climber robot, unless otherwise referenced. All other contributions from all parties acting in either an advisory or collaborative capacity, guidance by my supervisor(s), or use of external sources, have been acknowledged in the body of the thesis.

I understand that any breach of this declaration and/or of the declaration on the title page shall result in withdrawal of the degree or other actions by the university.

Jaghan Prashanth Gandhi Venkateshwaran

273157

12.04.25

Hochschule Furtwangen

Contents

TABLE OF FIGURES

LIST OF TABLES

1. Introduction

1.1. Background and Motivation

Robots that are capable of climbing stairs are an expanding area of research within mobile robotics, especially in indoor environments where staircases pose a serious impediment to traditional wheeled or tracked robots. Climbing robots are designed to autonomously negotiate stairs using advanced sensors and control algorithms that make them desirable for applications in health, rescue, and inspections. For instance, robots employed in search and rescue work often are called on to reach floors in damaged or dangerous buildings where it might not be unsafe for people to enter the same area. [1]

A major driver has been the interest in applying stair-climbing robots to indoor service applications where robots independently traverse multiple levels of the building. For example, having robots independently carry packages, clean surfaces, and provide surveillance or assistance to the elderly, to name a few use cases, would significantly benefit from automatically operated robots that could navigate stairs without human intervention [3]. When considering these indoor services, stair-climbing robots will face a variety of stair configurations, surface materials, and inclination angles which add uncertainty and increases the need for stability and traction while operating.[2]

As technology continues to grow, expectations in autonomous mobility also change. While ground robots have improved in terms of speed, accuracy, and amount of autonomy when moving across flat surfaces, a critical challenge for robots operating indoors, which includes stairs, remains to be solved [5]. Although there have been many solutions related to wheeled systems and tracked robots, which are both not particularly adept at stair climbing, the choice of moving means has, in fact, the primary limitation of planning to complete stair travel ahead of time in configuration of slope and steps [4]. As a result, robust and adaptable stair climbing mechanisms are paramount for autonomous mobility in dynamic environments.[6]

In addition, improvements in sensors, such as Inertial Measurement Units (IMUs), have made significant advancements in the precision and versatility of stair climbing robots. IMUs allow robots to perceive rate of orientation, rate of acceleration, and rate of pitch in real time, allowing stair climbing robots to respond quickly to the surrounding environment, thus making travelling up or down stairs safer and more efficient.[7]

This has become a rare challenge about healthcare, rescue, and industrial environments and the area of deploying mobile robots that can climb steep stairways, cross uneven terrains, and overcome other complex obstacles. Such environments actually place greater challenges for most traditional stair robots that operate only within certain environmental constraints. For example, in a search and rescue operation, people are often requested to climb stairways of buildings which are damaged from fire and earthquakes as human intervention might pose a serious danger [1]. Thus, this increasing need for autonomous robots able to traverse such environments gives rise to perennial demand for novel advances enabling robot vehicles to react in effective measures to unforeseeable conditions in operations in real-time. In this regard, stair climbing is an advanced technological exploration towards indoor autonomous navigation with

different transitioning levels. The research on the stair-climbing robots could be indexed towards various locomotive patterns, like a driving wheel, tracked, and mixed systems. Such systems are considered to provide secured balance in stair-climbing that guarantees not only a robot's stability but also ensures its utility in climbing over perceived obstacles, otherwise considered too challenging for traditional robots to cross [3]. Accordingly, leg locomotion has shown that Atlas and Spot robots are highly developed by Boston Dynamics, demonstrating their evolution age and bearing the refinement of an aspiring stair climbing robot eliciting agility upon hilly boundaries and barricades. The drawback is that such systems are extremely costly, complex, and impractical for commercialization and service purposes at the macro level [2].

Conversely, stair-climbing devices that would fit into a mid-market slot use simpler designs such as wheeled or tracked robots. The Scalevo robot, which aims to assist wheelchair users, uses both wheels and tracks to climb stairs, while keeping the effect of balancing and stability during these processes [3]. Thus, it combines efficient movement on flat surfaces with traction-for-stair-climbing capabilities. Nevertheless, most of the stairs-climbing robots made today have improved locomotion mechanisms, yet challenges remain for them to adapt to various designs of stairs, different heights of steps, and surface conditions [5]. Besides stair climbing, another crucial area of research is stable-speed motioning. Moving robots with stability in speed is of great importance, as a steady speed bears effectively upon the safe and smooth operation of mobile robots in a working environment. To this end, the PID controllers are widely used in controlling motor speeds based on feedback from sensors. Such controllers have commonly been used in mobile robots to make up for some deviation from velocity while providing smooth acceleration and deceleration, which, in turn, is important when an operation requires multiple transitions between different terrains or precision tasks such as angles turning [8].

In applications where user-derived rotations or movements to certain angles are needed, robots are expected by design to perform that specific rotation or adjustment very well. Often, in-service robots require that highly accurate movements are done during operations of picking and placing objects, traversing narrow places, or aligning with workstations. Activities have been done around the construction of these kinds of robots that can rotate to specified angles through sensor fusion by combining gyroscope, accelerometer, and encoder data to ensure precision and stability [7]. For example, in backing up routines on Boston Dynamics' Spot, it then utilizes algorithms during its climb adjustments to make its operation neater even while manoeuvring sharp turns or changing on uneven grounds [2]. Moreover, robots that need to follow a pre-programmed path usually use feedback loops that allow a dynamic and real-time adjustment of their movement based on the data from on-board sensors. Path following has been a major design in robots engaged in industrial automation, where they navigate around obstacles and return to certain "home" positions after accomplishing the tasks. Robots such as TurtleBot and Fetch were built for programmed, high-precision paths where LIDAR, IMUs, and cameras sense surroundings and adjust the path accordingly [4]. They operate mainly in warehouses and factories, moving between different workstations with the capability of adjusting their speeds and orientations dynamically in real time, based on feedback from sensors [6].

The motivation for developing multi-functional robots capable of stair-climbing, precise rotations, and stable speed control stems from the growing need for adaptable solutions that can operate in a variety of environments. From healthcare service robots navigating complex indoor layouts to industrial robots automating transportation tasks between different levels, there is a clear demand for autonomous systems that can adapt to real-time feedback and maintain precision in movement. As advancements in sensor technology and control systems continue, the next generation of robots will be expected to perform these tasks with greater efficiency, reliability, and at lower costs [3].

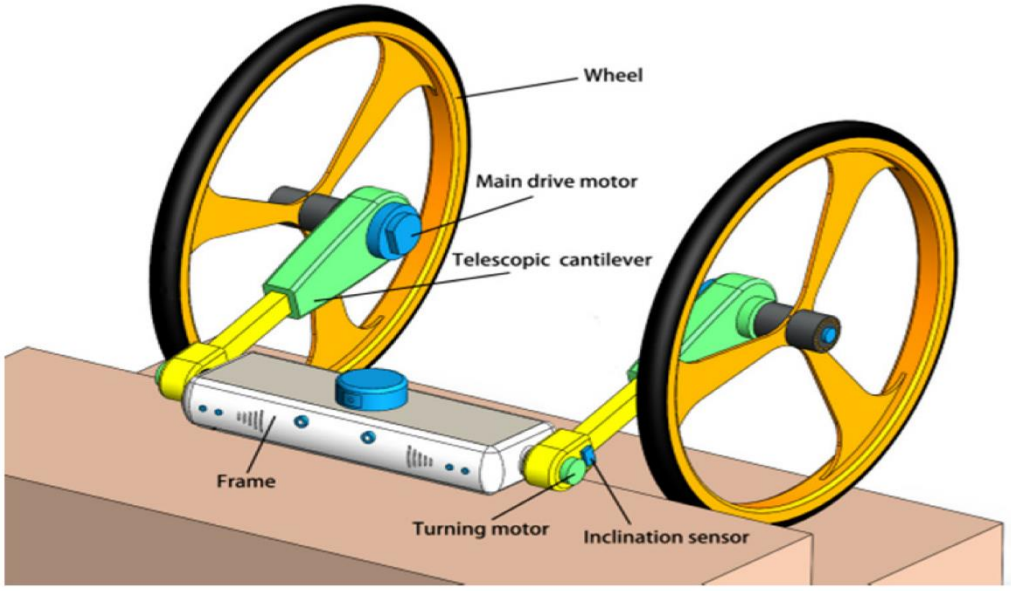

Figure 1: *One of the Proposed models for stair climbing with 2 wheels and 4R+2P pattern[20]*

1.2. Problem Statement

Despite large development progresses, many stair-climbing robots continue to struggle with relatively untreated problems once they hit real environments. The following are some challenges that have arisen in this context: Inconsistent step detection: The robots do not often know when they have encountered a step and this leads to very inefficient or unstable climbing behaviours [5].Poor adaptability: Staircases vary in height, tread length, material, and incline, making it tough for some kinds of stair-climbing robots with programmed movement patterns to deal nicely with all stair types [1].Energy inefficiency: Some stair-climbing robots when not doing dynamic modulation of power to the motors continue to consume energy in excess of what is needed; therefore, their operational times decrease and performance is affected significantly [2].Speed stabilization performance during dynamic movement: While the system is switching between several tasks, it is important to maintain the speed stable. In several systems, speed fluctuations lead to energy inefficiency and performance degradation of the robot [5]. Lack of wireless control systems: Some stair-climbing robots are either operated

manually or possess limited remote-control capabilities. Therefore, their functionality in dynamic environments such as hazardous zones or elderly care becomes difficult [6].In this regard, this project aims at solving those problems through the design of a stair-climbing robot that detects steps autonomously, keeps itself in balance with real-time feedback from the sensors, and adjusts the motor speed dynamically to optimize energy [6].

1.3. Objectives

The primary objective of this project is to design and implement a stair-climbing robot equipped with an MPU6050 IMU for real-time sensor feedback.

The project aims to:

Create an automated step detection system: The robot needs to auto track stairs based on the Z-axis accelerations and pitch angles measured by the IMU and alter its motor behaviours accordingly. The second requirement on the other hand states: Maintain stability while adaptable: The robot must maintain stability when passing through stairs regardless of stair heights, tread size or surface composition of the stairs and adapt motion according to sensory feedback from sensors (which provide feedback to respond to conditions in the real-time) sensibly. The next important requirement says: Maintain speed stabilization using PID control: The robot should itself regulate the speed on the move, for providing stability, while travelling in straight-path and transverse gestures in the command. Another requirement: Optimization of Motor Control Energy Efficiency: The robot should automatically adapt its power after step detection and settling to the new step minimizes energy consumption and extends operational time. Lastly, "12" says: Testing the robot on various shadows: The system should test over standard indoor stairs to evaluate its adaptability, performance and energy efficiency [2]

1.4. Scope of the Project

The scope of this project is focused on the development and testing of a stair-climbing robot designed for indoor environments. The key areas of focus include:

Hardware Development: This includes the selection and integration of hardware components, such as the MPU6050 IMU, motors, to create a system that can autonomously detect and respond to stair steps.

Software Architecture: The project will involve the development of real-time data processing algorithms to interpret IMU data for step detection and motor control.

System Testing and Validation: The robot will be tested in indoor environments with varying stair designs (different step heights, tread lengths, and materials) to assess its performance, stability, and energy efficiency during stair climbing.

The robot is intended for standard indoor environments with relatively uniform stair designs, typically found in homes, offices, and public buildings (Siegwart et al., 2011). The project does not cover the development of the robot for outdoor or highly irregular staircases, such as those with obstacles or extreme angles. Advanced user interfaces or machine learning algorithms for autonomous path planning are outside the scope of this project.

2.Literature Review

2.1 Overview of Stair-Climbing Robots

Stair-climbing robots are based on serious interest for anticipated applications, such as search and rescue missions, mitigation of disasters, and general urban situations in which stairs are present. Movement is to be through some difficult areas, such as rough surfaces and a staircase that are impossible for traditional wheeled-or-tracked robots to get across. The aim of these stair-climbing robots mainly would include situational awareness through the use of a system with reference to sensors or cameras of a common nature that can be implemented inside hazardous areas so as to avoid human rescuers from danger. This kind of robot appeared as systems that would aim at fast movement on flat ground, along with the ability to climb the stairs and move around on not-flat ground [10][11].

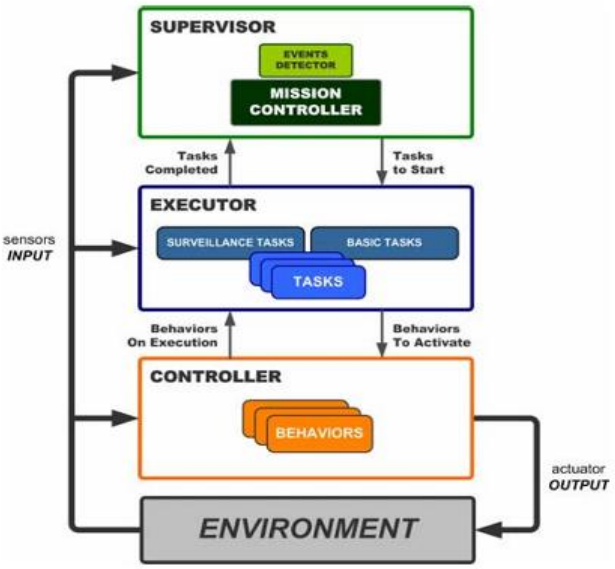

Figure 2: Block diagram illustrating the system architecture [12]

2.2 Existing Mechanisms for Stair Climbing

Various mechanisms have been proposed for stair-climbing robots, each with its own advantages and limitations. These mechanisms can be broadly categorized into tracked, legged, wheeled, and hybrid systems.

Tracked Robots: Tracked systems, like Pack Bot and Scalevo, are recognized for their effectiveness in climbing over rough surfaces with alternating widths, such as stairs. However, when on flat terrain, they tend to move slowly, and stairs without a riser or with big nosing may not be suited for such applications [10][11], suggested a three-wheeled robot in which the distance between the front and rear wheel actuators may be varied. By doing this, the robot may pitch back or forth to ascend staircases, with a lower chance of tipping over, using inclination angle compensation.

Figure 3*:Tracked robots [11]*

Legged Robots: The stair-climbing performance of documented humanoid or animal locomotion, such as Atlas and Spot created by Boston Dynamics, ensures good stair-climbing performance. These systems, however, are quite complex to control and require powered motors along with advanced sensors, leaving them expensive and impractical for mass use [11].

Figure 4*: Legged Robot [14]*

Wheeled Robots: The robot proposed is a wheeled design capable of climbing stairs via massive front wheels, in addition to a rear-wheel drive. Unlike the leg designs, these robots are simple to construct; however, when climbing, they may find difficulty with unitary stairs or irregular stairs. Sometimes, servo motors are used to tip or tilt the robot upward when attempting to ascend stairs in enhancing stabilization.

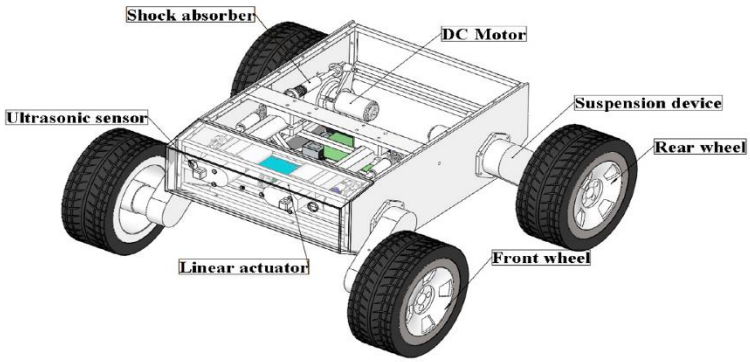

Figure 5*: Wheeled robot [15]*

Hybrid Systems: The systems combine the merits of wheels and some combination of either tracks or legs. For example, Baishya and others (2021) proposed a minimalist design of two large front wheels, a small rear wheel, and an actuator managing the pitch of the robot so that climbing the stairs could occur while balancing the robot digitally via IMU feedback.

Figure 6: *Hybrid exploration robot for Air and Land deployment [16]*

A mathematical model is created to examine the robot's kinematic and dynamic behaviour as it climbs and descends stairs. The relationships between the robot's various components are explained in this section. Additionally, the equations governing the wheel's trajectory on steps are examined. Lastly, a technique for figuring out the minimal connecting connection length required for the suggested robot to operate successfully is described.

2.3 Role of IMUs in Robotics

IMUs contribute a lot in stabilizing and controlling any robots walking over stairs. IMUs including MPU6050 indicate real-time orientation and motion data of the robot, such as pitch, roll, and yaw. These types of robots need to have a very stable construction while climbing a stair because they need to adapt motor speeds and tilt angles dynamically according to pitch, roll, and yaw angles. So far IMU has been found effective for reducing the variation of inclination angle of the robot during stair climbing, which is 77.8% less angle variation in ascent and 92.8% in descent, as shown by Baishya et al. in the year 2021. This just highlights the effectiveness of real-time sensor feedback to guarantee stability and adaptability in stair-climbing robots. Other important places where IMUs plays a vital role are:

Orientation and Stability Control: IMUs have accelerometers and gyroscopes which enable the sensor to follow the orientation and angular velocity of the robot. This data will help to keep the robot balanced while traversing uncertainties in terrain, for example: when climbing stairs. IMUs accept continuously input about the robot's tilt and rotation with respect to its stabilized and reset orientation, which allows the robot's control system to appropriately modulate motor speed and leg position to avoid tipping the robot over.

Example: IMUs in the Boston Dynamics "Spot" robot and similar designs will keep the robot's orientation properly adjusted over changes in the environment when climbing stairs or navigating around obstacles.

Position Estimation and Trajectory Planning: IMUs help the robot to estimate its position by following its position in three axes, forward-backward, side-to-side, and up-down. This is essential for trajectory planning when climbing stairs since the robot must estimate the height and depth of each stair step in order to plan its steps to keep from falling. IMUs are then referenced in the trajectory planning process in order to determine how the robot should allocate to each of its steps, which will make the robot more accurate, more efficient, and less likely to tip over.

Enhanced Navigation and Avoidance of Barriers: IMUs support stair-climbing robots in recognizing minute changes in their environmental state, including unexplained changes, such as sudden shifts or tilts as a result of irregularity in steps. After being conveyed, this internal sensor information can prompt the robot to dynamically modify its velocity and path to prevent navigating to a barrier or uneven surface. For instance, upon climbing a stair, whenever the robot detects a rapid change in angle, it may signal to the robot's next gait to be engaged.

Reference: With the type of integrated sensor and the role of IMU's in robots such as NASA's Robonaut and other unmanned systems, physical interactions or compel movement and provide susceptible inertial information to engage balance, and operate on irregular, obstacle strewn terrains.

Contributing to Complex Gait Control: IMUs serve as effective support sensors in robotic systems that cause the robot to mirror human- or animal-like movement (legged robots). When climbing stairs, the robot needs to manage the transition between a flat ground surface and an incline. The IM U provides much of this information when measuring changes about the gait in the robot. More importantly, IMU results can instruct the robot's gait timing and position in robot legs so that it is possible to maintain leg stability on each stair step while in transition.

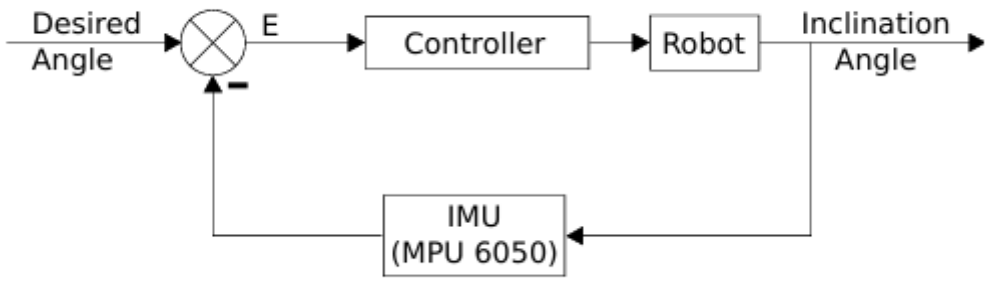

Figure 7: Controller block diagram [10]

2.4 Wireless Control in Robotics

Some of the technology associated with modern kinds of robotics involves wireless control, where bots can thus be operated and monitored from faraway, hazardous, or inaccessible terrains. Operationalized a wireless control system, using RF modules to provide remote control for a stair-climbing robot. This robot was equipped with a camera that relayed video to the operator as feedback on the surrounding environment. Also, in 2021, Baishya et al. considered wireless communication very important for monitoring the sensor data as well as the mobility of the bot. Wireless control systems provide added flexibility and usability to robots that go on search-and-rescue missions where an average human being cannot or shouldn't intervene directly.

2.5 Performance Metrics for Stair-Climbing Robots

One of the key hurdles for stair-climbing robots is finding a combination of stability, speed, and adaptability. [11] proposed a set of criteria for evaluating the performance of stair-climbing robots including speed of climbing, energy efficiency, and adaptation to different stair types. For example, an optimal robot would climb stairs at a vertical speed with humanlike inhalation as their potential model (approx. one step/second). Speed of climbing is important, but there is also a balance of importance of dynamic stability and climbing a variety of stair dimensions (height & tread). In the case of Baishya et al. experiments, averting the inclination angle consistently lead to significant improvements in extrapolated dynamics control and stability. The overall experiments showed a reduction of rear wheel torque of approximately 26.3% while climbing stairs, which improved motor efficiency, safety, and less chance of slippage. The concentration on adjusting power on motors based on feedback from sensors relates back to dynamic control.

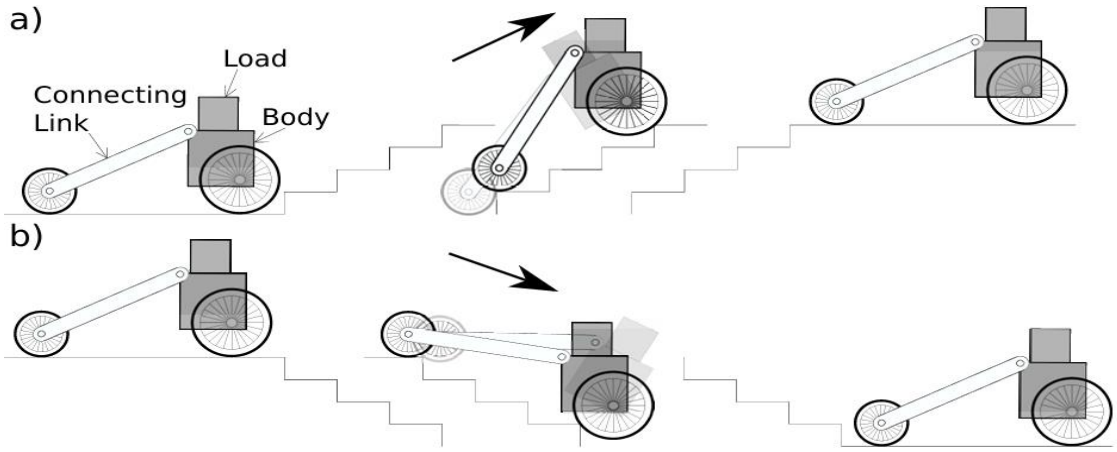

Figure 8: Stair climbing mechanism [10]

2.6 Comparative Analysis of Related Work

After comparing the stair-climbing robots currently available, it's apparent that each design has advantages and disadvantages. Tracked robots like the ones used for military and rescue robots are durable and perform well off-road, however, their speed on flat surfaces is much slower than other types of robots. Legged robots like Atlas and Spot are exceptional for climbing stairs, but they are highly controllable and expensive. Wheeled robots are usually one of the easiest robots to understand and control, balance functionality and usability but they most likely will encounter issues with stairs that are not perfect. Hybrid systems like Baishya's minimalist design [10], combine the stability and ease of use of wheels while utilizing actuators for climbing and provide satisfactory performance. IMUs and wireless control systems are now commonly used wireless control systems in today's stair-climbing robots. The use of IMUs provides critical performance data about a robot’s orientation, allowing stability to be maintained, but the use of wireless control systems allows for remote operation and monitoring making a robot safer to operate in hazardous conditions. Ultimately, the mechanism used will depend on the specific application of the robot. Advantages and disadvantages must be compared in terms of complexity, cost, and performance. In summary, as sensor technology, control systems, and mechanical design continue to improve, stair-climbing robots continue to improve the usability, adaptability, and stability of stair-climbing robots.

3. System Design and Methodology

This section outlines the overall design of the robot, including the hardware components, software architecture, and key control mechanisms that allow for autonomous stair-climbing, speed stabilization, user-defined rotations, and path-following capabilities.

3.1. System Overview

This mobile robot is capable of performing some tasks autonomously when finished fully, which include the ability to climb stairs maintain speed, turn specific angles, and track some pre-programmed path. Hardware integration includes the following key components: definition of the IMU (MPU6050 Inertial Measurement Unit), and DC motors and wheels to drive. On the software side, real-time data processing from the IMUs is performed using some control algorithms with a PID controller meant for speed stabilization, dynamic motor control. The system is capable of detecting stair steps, adjusting motor power based on sensor feedback, and following a pre-programmed path, such as returning to its home position.

3.2. Hardware Components

The robot incorporates several hardware components that provide sensing, motor control, and wireless communication capabilities, enabling it to perform complex tasks such as stair-climbing and precision navigation.

3.2.1. MPU6050 IMU: Functionality and Integration

The MPU-6050 is a widely used Inertial Measurement Unit (IMU) sensor that combines a 3-axis accelerometer and a 3-axis gyroscope. It is commonly used for motion tracking, orientation sensing, and stabilization in various applications, such as drones, robots, and wearables. Here's a detailed overview:

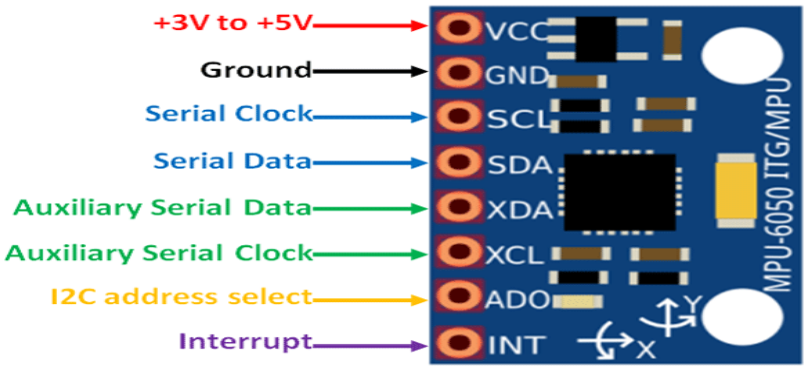

Figure 9: MPU 6050[17]

Key Features:

1. 3-Axis Accelerometer:
    - Measures linear acceleration along the X, Y, and Z axes.
    - Can detect static (like gravity) and dynamic acceleration (like movement).
2. 3-Axis Gyroscope:
    - Measures angular velocity (rotation rate) around the X, Y, and Z axes.
    - Provides rotational data in degrees per second (°/s).
3. Digital Motion Processor (DMP):
    - The MPU-6050 has a built-in DMP that processes complex motion sensing algorithms internally.
    - Offloads processing from the microcontroller, allowing for features like sensor fusion to combine accelerometer and gyroscope data.
4. I2C Interface:
    - Uses a standard I2C communication protocol to communicate with microcontrollers like Arduino or Raspberry Pi.
    - Default I2C address is 0x68.
5. Interrupt Pin:
    - Can generate interrupts for events like data ready, motion detection, or free fall detection.
6. FIFO Buffer:
    - Includes a 1024-byte FIFO buffer to reduce the load on the microcontroller by storing data for batch processing.

Specifications:

- Supply Voltage: 2.3V to 3.4V (usually powered at 3.3V).
- Gyroscope Range:
    - ±250, ±500, ±1000, or ±2000 °/s (configurable).
- Accelerometer Range:
    - ±2g, ±4g, ±8g, or ±16g (configurable).
- Sampling Rate:
    - Up to 8 kHz (gyroscope) and 1 kHz (accelerometer).
- Temperature Sensor:

  - Built-in sensor, with readings in degrees Celsius.
- Communication:
  - I2C (400kHz Fast Mode) or SPI (when configured).

Applications:

- Motion Sensing & Gesture Detection: Can track orientation and movement in three-dimensional space.
- Balance & Stabilization: Used in self-balancing robots, drones, and gimbals to maintain stability.
- Tilt Sensing: Provides real-time tilt data.
- Gaming Devices: Used for real-time motion control.
- Wearable Devices: Provides motion tracking in fitness and health devices.

How it Works:

1. Accelerometer measures the gravitational force in the X, Y, and Z directions. It can be used to calculate tilt and orientation.
2. Gyroscope measures the angular velocity (rotation rate) and is used to track rotational movements.
3. By combining data from both the accelerometer and gyroscope (sensor fusion), you can estimate the orientation and movement of the object more accurately, compensating for drift in the gyroscope using the accelerometer.

Pinout:

1. VCC: Power supply (3.3V).
2. GND: Ground.
3. SCL: Serial clock line (for I2C).
4. SDA: Serial data line (for I2C).
5. INT: Interrupt pin (used for signalling motion detection or data readiness)

In this project, IMU plays most of the role. It acts as the brain of the all the tasks that's been performed.

As mentioned above, it measures the acceleration and rotational velocity which is required for most parts of work. Acceleration values are utilized to find the sudden change in speed (velocity) which helps in detecting the stairs with rapid deceleration on z axis. Based on the finding the motor speeds are varied accordingly.

Rotational velocities are mostly used in finding out the angles which is required for precise rotation and pre-programmed path follower.

3.2.2. Motor and Wheel Configuration

A popular DC gear motor model in robotics and automation projects is the BY1016Z. These motors are known for their ability to produce high torque at low speeds, which makes them ideal for applications requiring controlled movement, like wheeled robots, conveyor belts, and other machinery where precise motion is essential.

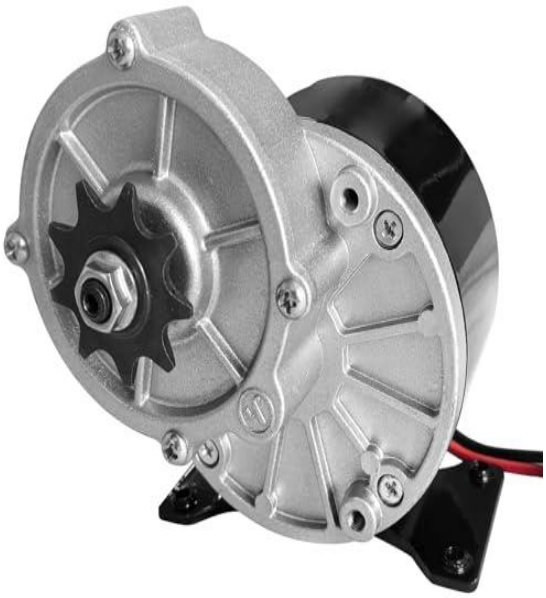

Figure 10: BY1016Z Motor [18]

Key Features:

1. **DC Motor with Gearbox**: The motor consists of inbuilt gear box which helps in reducing the motor's RPM while increasing the torque.
2. **High Torque**: Production of high torques at low RPMs makes it suitable candidate for driving wheels in a controlled motion.
3. **Low-Speed Output**: Even though it provides a low-speed output it has higher accuracy in control and movement.
4. **Compact Size**: The relatively small and lightweight of the motor makes it appropriate to use in compact robots.

General Specifications (Performance Parameters):

- **Model:** BY1016Z
- **Type:** DC brushed motor

- **Power:** 250 Watts
- **Voltage:** 24 Volts DC
- **Rated Speed:** 2700 RPM
- **Gear Ratio:** 9:7:1

The BY03LT02 is a DC motor driver that can be used to control speed and direction of DC motors for a range of applications like robotics, automation, devices and motor control. Generally, these types of drivers work with microcontrollers (for example, Arduino, Raspberry Pi), to control motors.

Figure 11: Motor Driver [19]

Below is an overview of the key features and specifications of the **BY03LT02** motor driver.

Key Features:

1. **Motor Driver for DC Motors**:
    - The **BY03LT02** is a motor driver for controlling the direction and speed of DC motors. It can handle a relatively high current output, making it suitable for medium-power motors.
2. **H-Bridge Motor Driver**:
    - The driver typically uses an H-Bridge circuit configuration, allowing you to control the direction of the motor. It allows you to run the motor forward, reverse, or stop it depending on the input control signals.
3. **PWM Control**:
    - The driver can accept **Pulse Width Modulation (PWM)** signals for speed control. By adjusting the duty cycle of the PWM signal, you can control the average voltage applied to the motor, which in turn adjusts the motor's speed.
4. **Overcurrent and Overvoltage Protection**:

- Some versions of the driver include built-in protection features like overcurrent protection, which helps protect the motor and driver from excessive loads and voltage spikes.

5. **Compact and Cost-effective**:
    - The **BY03LT02** is small and relatively inexpensive, making it suitable for both hobbyist and professional applications.

Typical Specifications:

- **Voltage Range**: Typically supports 6V to 12V (depending on the model and motor's power requirements).
- **Current Output**: Can usually supply **1-2A** of continuous current (peak current might be higher, but it is often rated for 2A continuous).
- **Logic Voltage**: Typically operates with logic voltage levels of 3.3V or 5V for the control pins (depending on the microcontroller used).
- **Control Inputs**:
    - Direction Control Pins (usually two pins): These pins determine whether the motor rotates clockwise, counterclockwise, or stops.
    - PWM Pin: Used for speed control, adjusting the speed by varying the duty cycle of the PWM signal.
    - Enable Pin: In some drivers, this pin is used to enable or disable the driver.

The robot's movement is powered by DC motors connected to the wheels, which enable it to move forward, climb stairs, and rotate to specific angles. The motors are controlled by a microcontroller, which adjusts their speed and power output based on feedback from the IMU and other sensors. The wheel configuration is designed to provide stability during stair climbing, with the wheels sized appropriately to ensure grip and balance on different types of stairs.

3.3. Software Architecture

The software architecture of the robot is designed to process real-time sensor data and control the robot's movements in response to environmental stimuli. Key components of the software include the motor control algorithm, the PID controller for speed stabilization.

3.3.1. Real-Time Data Processing

Real-time processing of data is vital to ensure that the robot accomplishes its tasks; these include but are not limited to stair climbing, stabilizing speed variations, and following a pre-set path. The system is, therefore, in constant reception from the MPU6050 IMU in order for the microcontroller to process the information and adjust the movements of the robot

according to new inputs in real time. The motor control algorithm will, for example, in conjunction with a profile of increased Z-axis acceleration indicating a stair step is detected, ramp up power to the motors to assist in climbing the stairs. Once the pitch angle stabilizes indicating clear levelling is accomplish on the new step, the speed will then be backed off to save power.

3.3.2. Motor Control Algorithm

The algorithm that controls the motors is mostly responsible for how the robot controls speed of motion and power output acting upon real-time sensor data. The algorithm responds to the feedback gained from the MPU6050 IMU to ensure the robot can effectively manage tasks such as climbing up and down stair steps, rotating to a target angle and accurately performing a pre-programmed path. The control algorithm is created to automatically adjust motor speeds to climb stairs in a more controlled manner based on the current orientation and acceleration of the robot's movement. As soon as the IMU notices the robot's velocity equality changes at the edge of a stair step, the power to the motors is adjusted up to allow the robot to comfortably climb over the stair step. As the robot stabilizes at the new stair step height, the motor power to the motors will be returned to a normal power level.

3.3.3. PID Controller for Speed Stabilization

The controller based on PID is a major scope part responsible for keeping the robot steadily running in its operations. To have transitions from one task to another, e.g., straight movement, climbing stairs, rotating to certain angles, the motor speed is adjusted PID by the feedback from the IMU. The continuous feedback about the speed of the robot helps to adjust the PID controller to counteract deviations of the robot velocity from a certain speed, allowing for a steady movement through various terrains and tasks. The robot has to perform such precision tasks, such as following a pre-programmed path or going back to homing itself.

The proportional-integral-derivative, or PID controller, is a control-loop feedback mechanism commonly used in robotics and industrial control systems. For the correction of the proportional, integral, and derivative terms, it continually computes an error value, which is defined as the difference between the current measured process variable (PV) and the desired setpoint (SP).

1. PID Control Basics

The PID controller algorithm consists of three terms:

1. **Proportional (P)**: Responds to the current error.
2. **Integral (I)**: Addresses accumulated past errors.
3. **Derivative (D)**: Predicts future errors based on the rate of change.

The output of the PID controller is calculated as:

$$u(t)=Kp\cdot e(t)+Ki\cdot\int_0^t e(\tau)\,d\tau+Kd\cdot\frac{de(t)}{dt} \quad u(t)=Kp\cdot e(t)+Ki\cdot\int_0^t e(\tau)d\tau+Kd\cdot\frac{de(t)}{dt}$$

Where:

- u(t): Control output.
- e(t)$e(t)$: Error (difference between desired speed and target speed
- Kp: Proportional gain.
- Ki: Integral gain.
- Kd: Derivative gain.

2. Components of PID Control

Proportional Term (P)

- The proportional term produces an output proportional to the current error.
- A higher Kp results in a larger response to errors, but excessive Kp can cause overshoot and instability.
- **Role**: Reduces the rise time and steady-state error but cannot eliminate it entirely.

Integral Term (I)

- The integral term sums past errors over time to eliminate steady-state error.
- A higher KiKi speeds up the elimination of steady-state error but can cause oscillations or instability.
- **Role**: Eliminates residual steady-state error that occurs with a pure proportional controller.

Derivative Term (D)

- The derivative term predicts future errors based on the rate of change of the error.
- A higher Kd dampens the system response, reducing overshoot and oscillations.
- **Role**: Improves system stability and reduces overshoot.

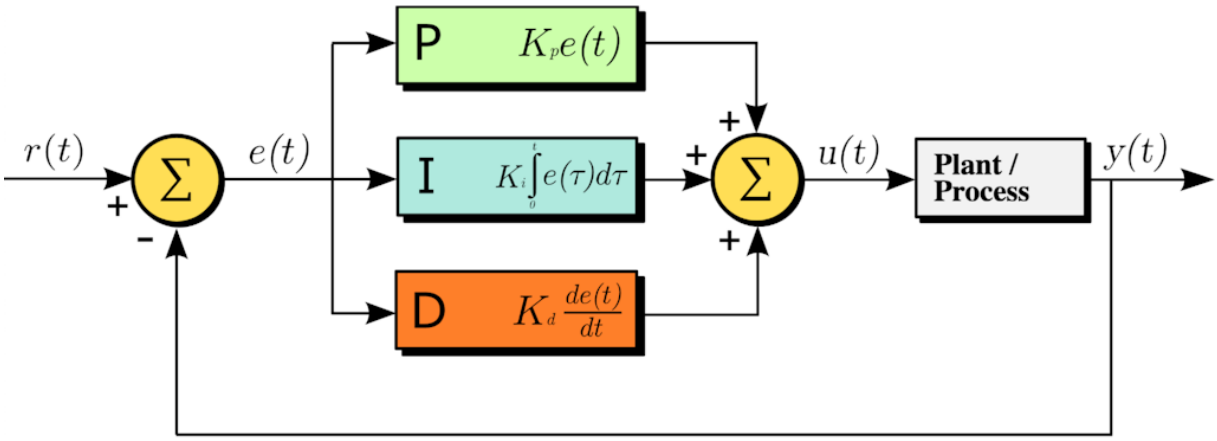

Figure 12: Flow of an example PID controller

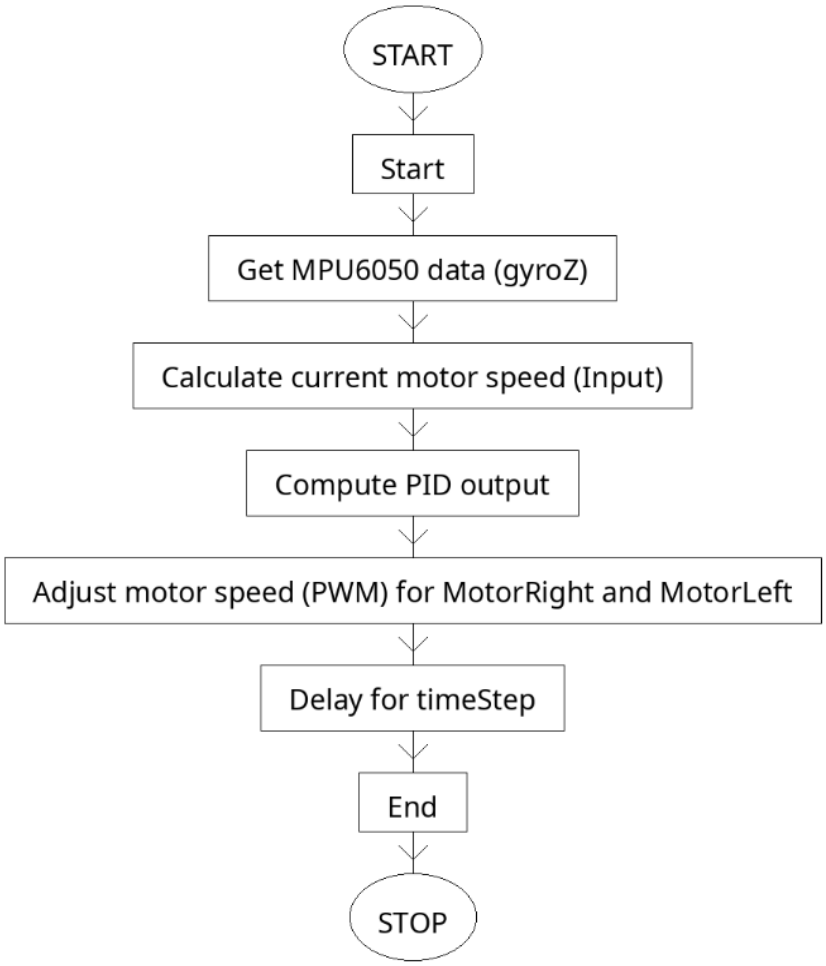

Figure 13*: Control flow of the PID controller for the speed control*

3.3.4. User-Defined Rotation Logic

The robot can also perform user-defined rotations. This means that the robot, upon commands being sent to it, can rotate to a specific angle using the rotation logic implemented from the angular velocity of the robot measured by the MPU6050 gyroscope. The integration of angular velocity over time gives the robot's current orientation, which can then modify its movement to reach the target angle. This user-defined rotation logic provides for very exact turns and keeping of angle, thus it brings a great deal of advantages in tasks, such as alignment with particular objects or following a predefined path.

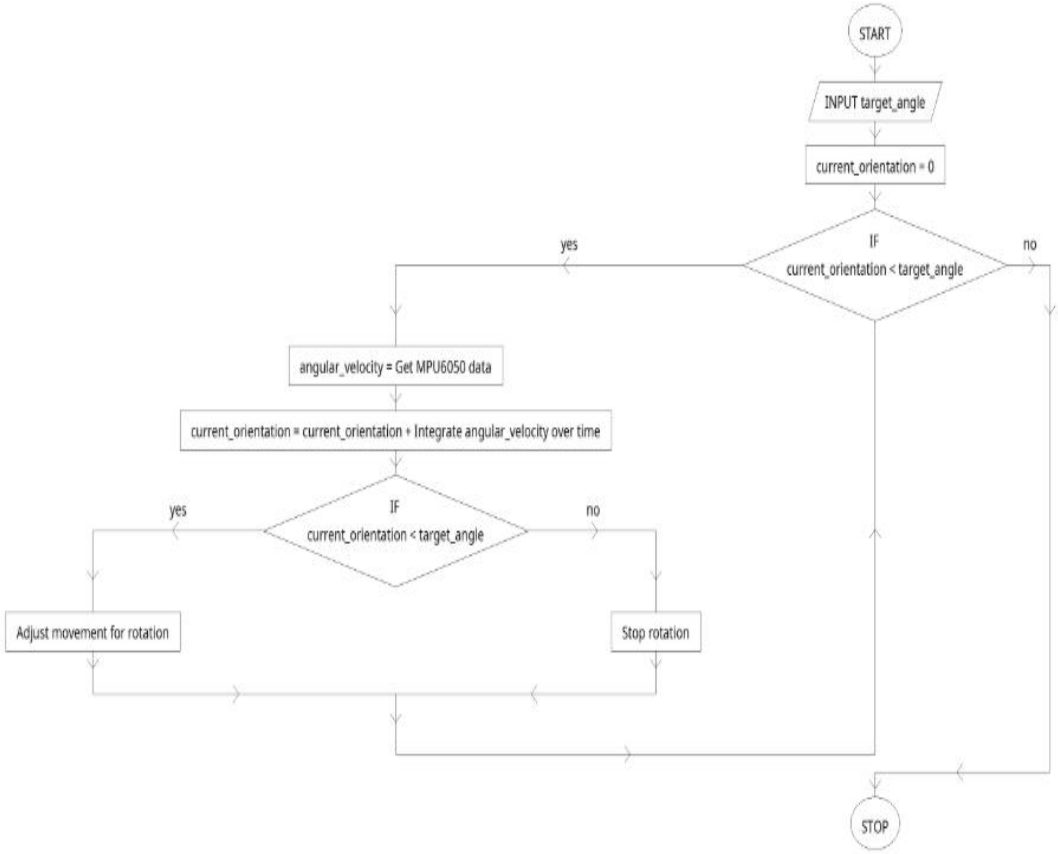

Figure 14*: User defined rotation Flow chart*

3.3.5. Path Planning and Navigation Algorithm

The Path Planning and Navigation Algorithm aims at guiding the robot through a programmed sequence of movements and rotations to allow it to navigate a predetermined path while maintaining speed stability with a fine control over its direction. It manages the motor commands through the sensor readings coming from the real-time integration of the MPU6050 IMU and a PID controller for the stabilization of the speed.

It thus ensures accurate path following and speed constant and alignment using sensor feedback and real-time control in motors so that the task is done by the robot autonomously without deviation.

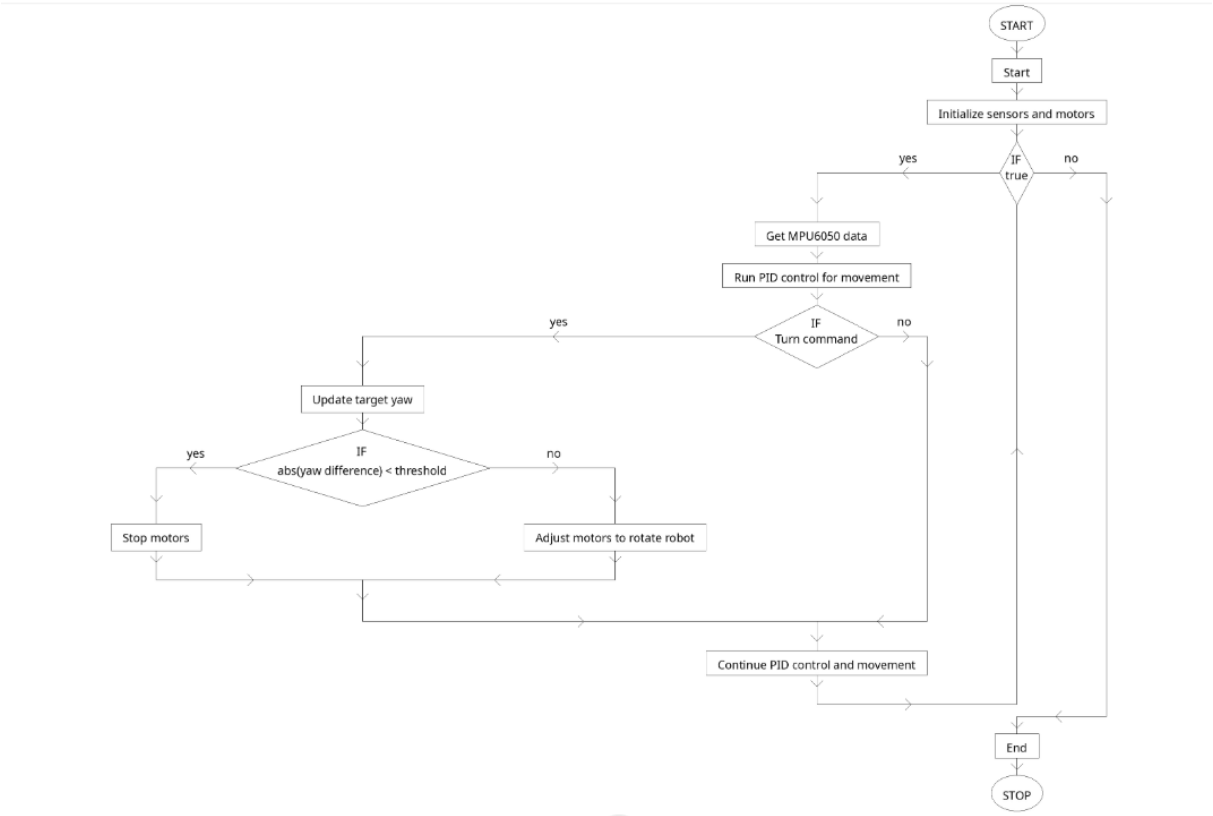

Figure 15*: Flow chart for Path Planning.*

3.3.6. Yaw based Deviation correction

Modifying deviation based on yaw is a very critical component of the motion control system of the robot to ensure the robot does not drift off track whether it be from external conditions or uneven ground. The yaw angle means a rotation motion about the vertical axis and is continuously monitored and modified through the PID (Proportional-Integral-Derivative) controller system. Yaw Angle Measurement and Processing yaw data is taken from the MPU6050 IMU sensor that measures yaw using its gyroscope and accelerometer in the IMU.

Deviation detection and Error calculation:

The deviation is calculated as:

$$\text{Error} = \text{Actual Yaw} - \text{desired Yaw}$$

The positive error indicates the robot has drifted towards clockwise direction whereas vice versa if it's negative (counterclockwise).

PID implementation for Deviation

$$\text{PID Output} = K_p E_{\text{yaw}} + K_i \sum E_{\text{yaw}} + K_d \frac{dE_{\text{yaw}}}{dt}$$

Where:

Kp: deviations are corrected based on the yaw error (Proportional gain).

Ki: avoids the drift by compensating the errors accumulated over the time (Integral gain).

Kd: predicts the future error (derivative gain)

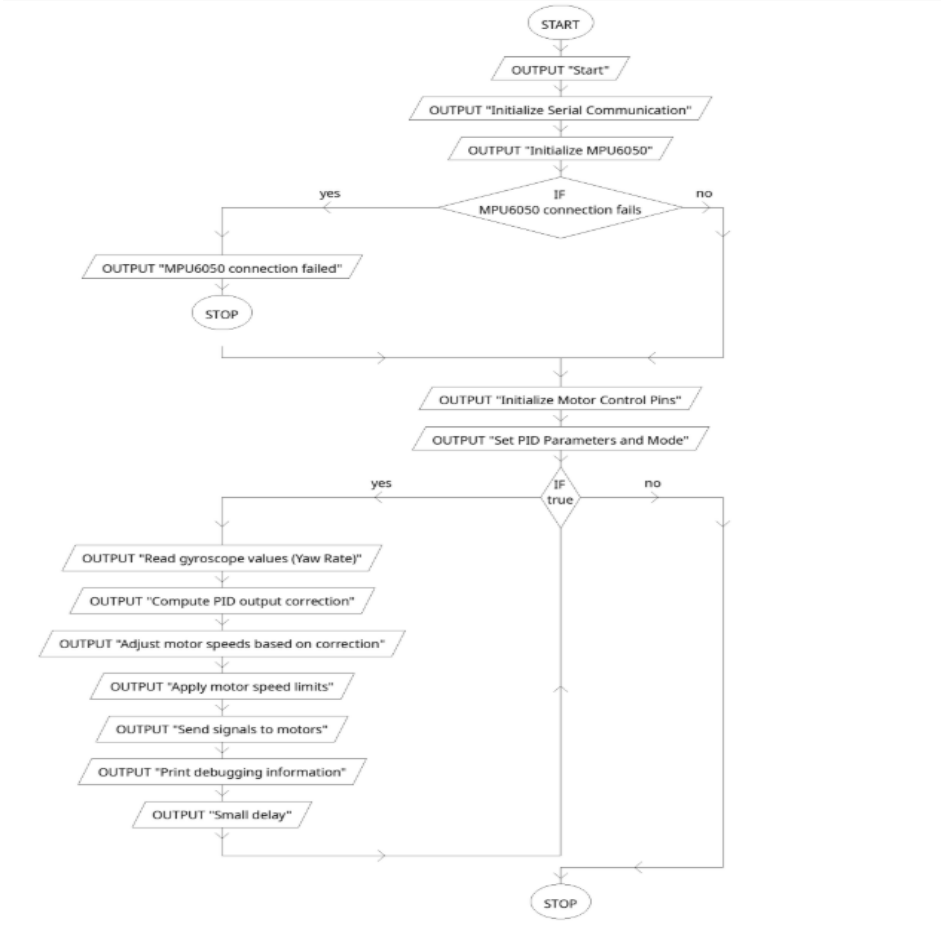

Figure 16*: Yaw correction Flow chart*

3.4. Step Detection and Climbing Mechanism

This is one of the most important and crucial features for any robot to autonomously detect a step and climb in real-time sensor data. This will use the IMU MPU6050 as the orientation of the robot pitch, yaw and detect changes in vertical acceleration to inform one about the presence

of a stair step, and thus, the entire logic will dynamically change the motor speeds in a way so that climbing may take place smoothly and with great stability.

3.4.1. Z-Axis Acceleration Analysis

The robot detects the movement in the Z-axis as it is climbing up a staircase with the help of MPU6050 accelerometer data. A sudden spike in Z-axis acceleration is recorded whenever the wheels of the robot touch the edge of the stairs. This would mean the robot has hit the steps: due to the sudden rise in acceleration. The data is acquired and treated in real time and allows the system to differentiate between flat surfaces and stair edges. This impulse provokes a rise in motor power by the control system, allowing the robot to climb the height of the step. In this way, the robot will adapt from one staircase to another with a different height of steps, thereby ensuring a step of different sizes will not require pre-programmed profiles.

3.4.2. Pitch Angle Stabilization

The robot will keep its balance by monitoring its pitch angle using the MPU6050 gyroscope once it has begun its ascent up the step. The increase in tilt reflects in the robot's climbing pitch angle. In order to ensure that the robot always maintains a balance throughout the climb, there has to be constant checking of the pitch angle by the controlling mechanisms. The pitch angle stabilizes once the robot has finished climbing the step and levelled out on its new surface. It's an important indicator of stabilization because the robot is on the level ground, which indicates it's landed on the top of the step. The motor power is reduced under control algorithm so the robot will continue to move at a constant speed without wasting energy on unnecessary motor operations.

3.4.3. Dynamic Motor Speed Adjustment

Having taken into consideration the MPU6050 imu data, motor speed is varied for stair climbing by the robot. In the presence of a detected stair step, the motor control algorithm outputs an increment in the speed of the actuating motors to confer extra power needed for the step climb. In other words, after landing on a stair, the speed of the motor is reduced to save some energy for the next move. Dynamic motor speed adjustment makes sure the robot efficiently climbs the stairs while those are stable in the process.

4. Implementation

This section involves assembling of the hardware components and combining with the software tools (Arduino IDE). Also, it contains the information about how the data is been acquired and processed.

4.1 Hardware Assembly:

Hardware assembly consists of combining the components like motors, drivers, MPU 6050 etc using jumpers and connecting wires to an Arduino Mega microcontroller. The placement of the orientation sensor which is MPU 6050 plays a vital role for the robot as it plays the major role for almost all the tasks that's been performed. So here it's placed on the connecting rod/link. And the position plays another important role in getting out the values. Since it's placed on connecting rod which is normally in slanting position for the whole time, the values need to be referenced properly to be accurate.

The two motors are connected through a motor driver to the wheels. The brain of the robot which is the microcontroller is placed on the base. Rest everything which is relay are placed on the base and are connected to Arduino.

The mechanical part consists of two large front wheels, two small rear wheels connected through a rod to the robot and a base connecting both the wheels. Another most important part in mechanical design is the length of the rods which needs to be exactly precise for stairclimbing.

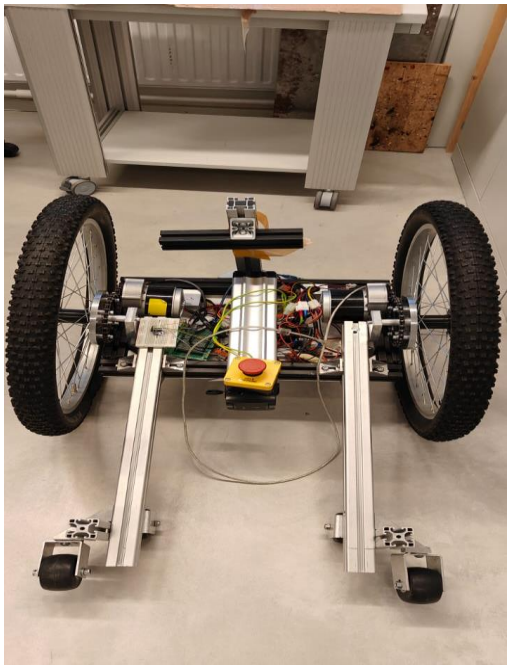

Figure 17: Hardware assembling

4.2 Software Development

4.2.1. MPU6050 Data Acquisition

The MPU6050 IMU is the one that senses the orientation of the robot along with the pitch, roll, yaw, and acceleration in X, Y, and Z direction. It is basically the data-acquisition code that reads the sensor's raw output, consisting of gyroscope outputs (in angular velocity) and accelerometer outputs (in linear acceleration). This raw data is processed using filters to attenuate noise, which reasonably gives valid readings on the pitch angle and Z-axis acceleration. The data from the MPU6050 is sampled at definite intervals, and the information is further processed using a microcontroller to determine when the robot's moving into a stair step owing to increased Z-axis acceleration spikes. The pitch angle is monitored continuously in a bid to control the robot upright while climbing stairs or negotiating turns.

```
#include <Wire.h>
#include <MPU6050.h>
MPU6050 mpu;

void setup() {
  Wire.begin();
  Serial.begin(115200);
  mpu.initialize();
  if (mpu.testConnection()) {
    Serial.println("MPU6050 connected successfully");
  } else {
    Serial.println("MPU6050 connection failed");
  }
}
void loop() {
  int16_t ax, ay, az, gx, gy, gz;
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
  float ax_g = ax / 16384.0;
  float ay_g = ay / 16384.0;
  float az_g = az / 16384.0;
  float gx_dps = gx / 131.0;
  float gy_dps = gy / 131.0;
  float gz_dps = gz / 131.0;
```

Figure 18*: MPU 6050 example code snippet*

Here is an example code of how data is acquired from an MPU6050 orientation sensor.

The sensor communicates with the microcontroller via the I2C protocol using the requires the Wire.h library. To use the MPU6050 sensor and its functions to gain data, we utilize the MPU6050.h library. To initialize the sensor, we call the function mpu.initialize()--this allows the MPU6050 to be set up and also to assure that the microcontroller can detect it. This is crucial, as we want to ensure that the sensor is given power and is able to respond properly before acquiring any data from it. Assuming the sensor is connected properly, it will collect raw data. To read raw data, we utilize the function mpu.getMotion6(). This function collects data from the accelerometer and gyroscope. The accelerometer will report data along the X, Y, and Z axes. The gyroscope will report angular velocity of each axis measurement.

Since the MPU6050 provides raw data, this data must be transformed into useful physical units. The accelerometer data is scaled and modified into meters per second squared (m/s^2), with or without the change in gravity based on the application, and the gyroscope data is converted into degrees per second (°/s). The scaling factors for each sensor are determined by the set sensitivity range, which is tunable. This conversion is essential to converting the raw data presented as 16-bit signed integers into human-readable physical units for further processing to collect orientation, movement, or tilt.

Acceleration conversion:

The accelerometer can be configured for different sensitivity levels:

- ±2g
- ±4g
- ±8g
- ±16g

The sensitivity (scale factor) for each range is as follows:

- **±2g** → 16384 LSB/g
- **±4g** → 8192 LSB/g
- **±8g** → 4096 LSB/g
- **±16g** → 2048 LSB/g

Accel = RawAcc / Sensitivity

Gyroscope conversion:

The gyroscope can also be configured for different sensitivity levels:

- ±250°/s
- ±500°/s
- ±1000°/s
- ±2000°/s

The sensitivity (scale factor) for each range is as follows:

- ±250°/s → 131 LSB/°/s
- ±500°/s → 65.5 LSB/°/s
- ±1000°/s → 32.8 LSB/°/s
- ±2000°/s → 16.4 LSB/°/s

Gyro = RawGyro / Sensitivity

4.2.2. Motor Control Logic for Stair Climbing

The motor control logic is paramount for a smooth and efficient stair climbing process by changing the speed and torque of the motors in response to the real-time feedback received from the MPU6050 IMU. During the stair climbing process, the system continues to measure Z-axis acceleration and pitch angle, allowing the system to detect elevation changes (i.e. steps) and react accordingly. When the system detects a forthcoming step based on the significant change in Z-axis acceleration, the control logic sends a signal to increase the motors' power, which in turn allows the robot to exert more force to climb up over the step. This increase in power to the motors is a need due to the extra energy input to lift the robot over the step in order to maintain a smooth transition from step to step without losing stability. It is provided with fixed time for the climbing of the stairs. For instance, in this code the time is given is 3 sec. Once it crosses 3 secs, the motor power reduces, and it again checks for step detection, and it begins again.

After the robot has successfully climbed the step, the system reaches a stable pitch angle, and the control logic will lower the motor power level back to a normal level. Significantly reducing the power is a matter of conserving energy and preventing overheating and excessive strain on the motors.

```
void loop() {
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

  float pitch = atan2(ay, az) * 180.0 / M_PI;
  float zAccel = (float)az / 16384.0 * 9.81;
  float zAccelChange = abs(zAccel - prevZAccel);

  if (!sensorsReady && millis() - sensorStartTime > 1500) {
    sensorsReady = true;
    Serial.println("Sensors ready. Starting detection...");
    prevZAccel = zAccel;
    prevPitch = pitch;
    return;
  }

  if (!sensorsReady) return;

  Serial.print("Pitch: ");
  Serial.print(pitch);
  Serial.print(" | Z-Accel Δ: ");
  Serial.println(zAccelChange);

  if (!isClimbing && zAccelChange > zAccelThreshold && !torqueBoosted) {
    Serial.println(" Step detected → Torque boost!");
    stepDetected = true;
    lastStairTime = millis();
    torqueBoosted = true;
    driveWithRollControl(torqueBoostSpeed);
  }
  if (stepDetected && pitch > pitchClimbThreshold && !isClimbing) {
    Serial.println(" Climbing detected...");
    isClimbing = true;
    stepCompleted = false;
    stepDetected = false;
    lastStairTime = millis();

    driveWithRollControl(climbSpeed);
    torqueBoosted = false;
  }
  if (isClimbing && pitch < pitchLevelThreshold && !stepCompleted) {
    Serial.println(" Step climbed successfully!");
    isClimbing = false;
    stepCompleted = true;
    lastStairTime = millis();

    driveWithRollControl(baseSpeed);
  }
```

```
void driveWithRollControl(int targetSpeed) {
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
  float roll = atan2(-ax, az) * 180.0 / M_PI;

  int startSpeed = 85;
  int rampStep = 5;
  int delayBetweenSteps = 10
  if (abs(roll) < 1.0) {
    Serial.print("speed ");
    Serial.println(targetSpeed);

    for (int s = startSpeed; s <= targetSpeed; s += rampStep) {
      analogWrite(MotorLeft, s);
      analogWrite(MotorRight, s);
      Serial.print("  Speed: ");
      Serial.println(s);
      delay(delayBetweenSteps);
    }

    analogWrite(MotorLeft, targetSpeed);
    analogWrite(MotorRight, targetSpeed);
  } else {
    int correction = roll * 1.0;
    int leftTarget = constrain(targetSpeed - correction, 70, 150);
    int rightTarget = constrain(targetSpeed + correction, 70, 150);

    Serial.print("Roll: ");
    Serial.print(roll);
    Serial.print(" | L: ");
    Serial.print(leftTarget);
    Serial.print(" | R: ");
    Serial.println(rightTarget);

    for (int s = startSpeed; s <= targetSpeed; s += rampStep) {
      int leftPWM = constrain(s - correction, 70, 150);
      int rightPWM = constrain(s + correction, 70, 150);
      analogWrite(MotorLeft, leftPWM);
      analogWrite(MotorRight, rightPWM);
      delay(delayBetweenSteps);
    }

    analogWrite(MotorLeft, leftTarget);
    analogWrite(MotorRight, rightTarget);
  }
}
```

Figure 19*: Step detection function algorithm snippet*

Here is the code which is defined to detect when the robot encounters a step while climbing stairs, and to adjust the motor speed accordingly.

1) Sensor Data Acquisition:
    - The variables ax,az,ay and gx,gz,gy are initialized for storing acceleration and gyroscope values using the function **mpu.getMotion6() function.**
    - After which the Z acceleration is calculated from raw values.
2) Change Detection:
    - With the difference between Z acceleration and previous Z acceleration, the change is detected. Similarly, the change in Pitch has also been detected.
3) Step Detection Logic:
   The if statement checks the conditions as below:
    - If it is not climbing and Z accChange and pitch change > Z acc and Pitch threshold, then a step has been detected. After which the motor increases the speed for climbing.
4) Pitch Stabilization Check:
    - Now the second if statement checks whether it is climbing or not and whether the pitch is levelled with the ground. Once both are satisfied then step is climbed.
5) Update Previous Values:
    - Finally, the previous pitch and previous Z acceleration values are updated with the current values.
6) Speed control (Roll):
    - After the detection of an incline, speed control logic starts which is it modifies motor speeds in real-time to keep the robot stable and climbing. Most of the control will use the roll angle to keep the robot stabilized laterally and to avoid overlapping or misaligned routers during ascent. As the robot uses horizontal steering and pitch and roll angle data, it will climb stairs and other similar elevations in a controlled and stable manner.

4.2.3. PID Controller Implementation (Speed control)

The PID (Proportional-Integral-Derivative) control system is used to maintain the robot's speed during operation. The PID control system will continuously monitor the difference between setpoint speed and actual speed and adjust motor power accordingly.

The proportional corrects the power in relation to the immediate difference between the setpoint and the actual speed; the integral corrects the error by summing it on time to null out steady-state errors; the derivative anticipates the future error by rate of change and thus corrects the motor power in order to prevent overshooting.

This PID control system keeps the speed of the robot steady while across various terrains or congested tasks like climbing stairs and to moving it all in a straight line.

```
void loop() {
    float gyroRate;
    getMPU6050Data(accelAngle, gyroRate);
    float currentTime = millis() / 1000.0;
    dt = currentTime - lastTime;
    lastTime = currentTime;
    gyroAngle += gyroRate * dt;
    filteredAngle = alpha * (filteredAngle + gyroRate * dt) + (1 - alpha) * accelAngle;
    float relativeTilt = filteredAngle - baseTilt;
    float setpoint = 4.5;  // Target tilt (adjust if needed)
    error = setpoint - relativeTilt;
    integral += error * dt;
    derivative = (error - prevError) / dt;
    pidOutput = (Kp * error);
    prevError = error;
    int motorSpeed = motorPWM + pidOutput;
    motorSpeed = constrain(motorSpeed, 0, 88);
    analogWrite(motorright, motorSpeed);
    analogWrite(motorleft, motorSpeed);
    Serial.print("Tilt Angle: ");
    Serial.print(relativeTilt);
    Serial.print(" | PID Output: ");
    Serial.print(pidOutput);
    Serial.print(" | Motor Speed: ");
    Serial.println(motorSpeed);
```

Figure 20*: PID controller function snippet*

The following code illustrates the design of a PID (Proportional-Integral-Derivative) controller that helps keep the robot stable and maintains speed. The first step is getting the raw sensor data. As mentioned, the raw accelerometer and gyroscope data can be accessed using the **getMPU6050Data(accelAngle, gyroRate)** function. In this code we are using the tilt angle for stabilizing the motor speed for which we would be taking both acceleration angle and gyroscope rate and along with that a complementary filter has been incorporated to minimize the errors due to the noise. Here the setpoint has been set at 4.5 for the tilt angle as the MPU6050 has been placed on the connecting link which is normally slanted all the time. Then the PID calculations takes place where Proportional, integral, derivative has been calculated and summated. The summated PID output is later applied to the motor speed.

To prevent moving too quickly or overshooting, the motor speed is restricted to an upper limit of 90. This ensures the robot can maintain a controlled, fixed speed to avoid fast, jerky movements that would unbalance it. During robot motion, the code will keep running a loop internally continuously checking for changes in tilt or disruption.

Finally, PID controller tries to maintain a stable irrespective of load or any sort of disturbance and correct its movement.

4.2.4. User-Defined Rotation Implementation

With the user-defined rotation function, users can cause the robot to orient to a certain angle, through commands. Information from the gyroscope (MPU6050) considers angular velocity and integrates by time to find the robot's current orientation. A control logic compares the current orientation to the target angle once the rotation command is received and modifies the speed of the motors accordingly. The system halts all motors when the robot aligns itself with the requested orientation. This implementation is paramount to operations that require the robot to be aligned with absolute precision: positioning for defined operational tasks and aligning it along pre-defined paths.

```ino
void updateTargetYaw() {
  if (Serial.available()) {
    char input = Serial.read();
    if (input == '0') {
      targetYaw = 0;
      Serial.println("Target Yaw set to 0");
    } else if (input == '9') {
      targetYaw = yaw + 90;
      Serial.println("Target Yaw set to 90");
    } else if (input == '-') {
      input = Serial.read();
      if (input == '9') {
        targetYaw = yaw - 90;
        Serial.println("Target Yaw set to -90");
      } else if (input == '4') {
        targetYaw = yaw - 45;
        Serial.println("Target Yaw set to -45");
      }
    } else if (input == '1') {
      targetYaw = yaw + 180;
      Serial.println("Target Yaw set to 180");
    }

  }
}
```

(a)

```ino
void controlRotation() {
  float yawDifference = targetYaw - yaw;

  // // Normalize the yaw difference to between -180° and 180°
  if (yawDifference > 180) {
    yawDifference -= 360;
  } else if (yawDifference < -180) {
    yawDifference += 360;
  }
  Serial.print("Yaw : ");
  Serial.println(yaw);
  Serial.print("Yaw difference: ");
    Serial.println(yawDifference);
  Serial.print("Target Yaw : ");
    Serial.println(targetYaw);
  if (abs(yawDifference) < 5) {
    analogWrite(MotorLeft, 0);
    analogWrite(MotorRight, 0);
    Serial.println("Stopping Robot (Target Reached)");
  }
  else if (yawDifference > 0) {


    digitalWrite(MotorLeftBack, HIGH);
    digitalWrite(MotorLeftBack_1, HIGH);
    digitalWrite(MotorRightBack, LOW);
    digitalWrite(MotorRightBack_1, LOW);
    analogWrite(MotorLeft, 75);
    analogWrite(MotorRight, 80);
    Serial.println("Rotating Right");
    }
  else {
    digitalWrite(MotorLeftBack, LOW);
    digitalWrite(MotorLeftBack_1, LOW);
    digitalWrite(MotorRightBack, HIGH);
    digitalWrite(MotorRightBack_1, HIGH);
    analogWrite(MotorLeft, 75);
    analogWrite(MotorRight, 75);
    Serial.println("Rotating Left");
  }
}
```

(b)

Figure 21*: functions for (a) rotation (b) user input*

Above is the actual code and working of how the user defined rotation works.

Two functions have been used one for the actual rotation and one for getting the values from the user.

void controlRotation(): The controlRotation() method is responsible for rotating the robot to the user-defined target angle. The controlRotation() runs continually to monitor the difference between the robot yaw orientation (around the vertical Z-axis) and target yaw (user-defined angle), which is obtained from the gyroscope data from the MPU6050 sensor. The robot will start by checking its current yaw angle and compare it to the desired target yaw. After which the difference between the two yaw angles is calculated and termed the "yaw difference." If the yaw difference is positive, it indicates that the robot will also be rotating in that direction; if it

is negative, the robot would be rotating the other way. The robot will continue to rotate until it reaches a yaw difference of zero, which indicates the robot has reached target yaw. This would allow a smooth and accurate rotation.

void updateTargetYaw(): This function gets the input from the user and provides it to the controlRotation function. This function acts as a interface between the user and the robot's rotation system.

4.2.5. Path Planning and Navigation Implementation

This part focuses on the implementation of the path planning and navigation system to follow a specific sequence of movements that would allow it to return to its present location after performing certain tasks. The navigation system implements a combination of the MPU6050 IMU and a PID controller to do accurate movements along a certain fixed path.

The path followed by the robot involves a series of timed movements and rotations, ensuring accurate distance travelled and precise turning angles:

1. **Home position**: It starts from a fixed home position. This is the reference position as it returns to the home position after completing the designed path.
2. **Forward movement 1:** From the home position it moves forward for 16 secs with the help of PID controller which is with a constant speed. Feedback from the sensor is used to adjust the motor power, preventing the robot from deviating from the target speed
3. **Rotation:** after moving forward for 16 secs it stops and rotates 90 degrees with the help of Yaw from MPU 6050. This works with rotation of wheel forward and pone backward for a sharp turn.
4. **Forward movement 2:** after the rotation it again moves forward for another 16 secs using PID.
5. **Rotation:** then it rotates for 180 degrees which is it takes U-turn. Same as mentioned above rotation takes place with one wheel forward and another one backward.
6. **Forward movement 3:** Again, it starts to move forward for another 16 secs using the PID.
7. **Rotation:** After 16 secs it stops and takes a rotation of -90 degrees to the left.
8. **Forward movement 4:** Again, it moves forward for another 16 secs to the home position and rotates 180 degrees.

The PID controller within the robot is a critical component for assuring speed and a smooth transition between movements and turns. Integrating feedback from the MPU6050 IMU, which track both orientation and speed, makes the robot capable of rotating precisely with an appointed period for moving forward. With respect to time, segments are expected to last 16 seconds, controlling distance hopped. Rotation is expected to be of 90 or 180 degrees. Hence the PID has done its function for path deviation. The robot does this altogether while it's used

to make the robotics to run the pre-defined way path for back home, without help from anybody else. Hence, this implementation proves what all the robot is capable of a complex task path-following with a constant speed control and accurate rotation.

```
void loop()
{
    while (millis() - startTime <= duration1)
    {
        PID();
        Serial.println("PID 1 running ");
    }
    while (millis() - startTime <= duration1 + 6000)
    {
        Turn90();
    }
    while (millis() - startTime <= duration2 + 4000) {

        PID();
        Serial.println("PID 2 running ");
    }
    while (millis() - startTime <= duration2 + 10000) {
        Turn180();
    }
    while (millis() - startTime <= duration3) {
     PID();
    Serial.println("PID 3 running ");
  }
   while (millis() - startTime  <= duration3 + 8000) {
    Turnminus90();
  }
  while (millis() - startTime  <= duration4) {
    PID();
    Serial.println("PID 4 running ");
  }
      stopMotors();
      Serial.println("Reached home position ");
}
```

Figure 22*: Path planning logic*

The above lines of code are the logic behind the Path Planning and navigation. Since entire path planning runs on time basis, after the completion of each fixed given time frame it moves on to the next function. Each function has been defined individually.

millis(): This function tracks the time since the program started running. It is used to create a time-based control flow where the robot performs different actions based on elapsed time.

PID control: The robot's PID controller is invoked at specific time intervals to regulate the motor speeds and maintain stability. For example, during "PID 1 running" and "PID 2 running" phases, the PID controller ensures smooth navigation while the robot is in motion.

Turn functions: Functions like Turn90(), Turn180(), Turnminus90() rotate the robot by the specified angles. These turning functions are executed after a specific duration has elapsed, indicating a change in the robot's direction as part of its path planning.

4.2.6. PID Implementation (Deviation correction)

This part focuses on the correcting the motor speeds based on the deviation from the yaw rate. As mentioned, the PID processes the error using proportional, integral, derivative gains to correct the motor speeds to avoid deviations. If the robot deviates right, then the right motor speed decreases while left increases and vice versa for the opposite.

```
void loop() {
    int16_t ax, ay, az, gx, gy, gz;
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

    // Convert gyroscope data to degrees/sec
    input = gz / 131.0;
    // Compute PID output
    yawPID.Compute();
    int baseSpeed = 83;  // Base motor speed
    int leftSpeed = baseSpeed + output;
    int rightSpeed = baseSpeed - output;
    leftSpeed = constrain(leftSpeed, 0, 88);
    rightSpeed = constrain(rightSpeed, 0, 88);
    analogWrite(MOTOR_LEFT_PWM, leftSpeed);
    analogWrite(MOTOR_RIGHT_PWM, rightSpeed);

    Serial.print("Yaw Rate: "); Serial.print(input);
    Serial.print(" | Correction: "); Serial.println(output);
    Serial.print(" left speed "); Serial.println(leftSpeed);
    Serial.print(" right speed "); Serial.println(rightSpeed);

    delay(10);
```

Figure 23*: PID code snippet*

The code snippet mentioned above for yaw correction reads the raw values from the MPU6050 sensor, processes that data, and output a useful format based on the deviation. Yaw correction is necessary for maintaining reliable movement, and therefore we only consider the relevant data around the deviation. A base speed is set in the program beforehand, which serves as the baseline speed that the robot's motors can adjust from. The PID controller will read the yaw deviation and adjust the motor speed up or down based off the PID output. This will keep the

robot straight, or stable, while moving and correct for any unintentional yaw from slower or faster speed or disturb disturbances from the external environment or uneven ground.

4.3 Integration of Hardware and Software

Sensor integration: The MPU6050 IMU is integrated with Motor Control system to achieve real-time feedback for Speed Stabilization and Step Detection. The IMU is processed in such a way that the output which it provides is directly taken into the motor control and that helps to dynamically change the speed of the motor to facilitate easy stair climbing and accurate rotation. Integration of Motor and PID Control: The PID controller is finely tuned to achieve a steady speed while performing various tasks such as stair climbing and straight-line navigation. Testing of the motor control logic is achieved to enable the robot to perform smooth transitions from one task, like moving forward, rotating, and adjusting speed according to the terrain. For validation and testing purposes, the robot is subjected to a series of tests built upon performing critical tasks, which integrate both robotic hardware and software computations. According to their nature, those tests might refer to such functionalities as stair negotiation, rotation to some selected angles, navigation along prescribed paths, returning to their home position, etc. In accordance with the results received from these tests, the system can be further tuned for a better job depending on various conditions.

5. Testing and Evaluation

5.1.Performance evaluation

5.1.1 User defined rotation accuracy

In this test, the code has been designed such that it gets the input from the user for the rotation. And here the rotation of robot is constrained to 4 fixed angles which are 0 deg (home position), 90 deg, -90 deg, 180 deg. So once the user inputs the required angle for rotation, the robot rotates in that prescribed direction.

```
09:56:47.144 -> Testing MPU6050 connection...
09:56:47.144 -> MPU6050 connection successful
09:56:47.171 ->
09:56:47.171 -> Send any character to begin:
09:56:53.867 -> Initializing DMP...
09:56:54.134 -> >**......>......These are the Active offsets:
09:56:56.101 -> -2982.00000,    1154.00000,    1650.00000,    -50.00000,    -91.00000,
09:56:56.166 ->
09:56:56.166 -> Enabling DMP...
09:56:56.166 -> Enabling interrupt detection...DMP ready! Waiting for first interrupt...
09:56:56.265 -> Input '0', '9', '-9', '18', or '-4' to set target yaw.
09:56:56.298 -> Yaw : 0.00
09:56:56.330 -> Yaw difference: -0.00
09:56:56.330 -> Target Yaw : 0.00
09:56:56.363 -> Stopping Robot (Target Reached)
09:56:56.437 -> Yaw : 0.00
09:56:56.470 -> Yaw difference: -0.00
09:56:56.470 -> Target Yaw : 0.00
09:56:56.502 -> Stopping Robot (Target Reached)
09:56:56.576 -> Yaw : -0.01
09:56:56.576 -> Yaw difference: 0.01
09:56:56.609 -> Target Yaw : 0.00
```

(a)

```
09:57:04.830 -> Stopping Robot (Target Reached)
09:57:04.903 -> Target Yaw set to 90
09:57:04.937 -> Yaw : -0.24
09:57:04.937 -> Yaw difference: 90.00
09:57:04.970 -> Target Yaw : 89.76
09:57:04.970 -> Rotating Right
09:57:05.039 -> Yaw : -0.21
09:57:05.072 -> Yaw difference: 89.97
09:57:05.072 -> Target Yaw : 89.76
09:57:05.115 -> Rotating Right
09:57:05.150 -> Yaw : -0.21
09:57:05.183 -> Yaw difference: 89.98
09:57:05.217 -> Target Yaw : 89.76
09:57:05.217 -> Rotating Right
09:57:05.289 -> Yaw : -0.23
09:57:05.289 -> Yaw difference: 89.99
09:57:05.321 -> Target Yaw : 89.76
09:57:05.361 -> Rotating Right
09:57:05.393 -> Yaw : -0.23
09:57:05.428 -> Yaw difference: 89.99
09:57:05.428 -> Target Yaw : 89.76
09:57:05.461 -> Rotating Right
```

(b)

```
09:57:07.435 -> Target Yaw : 89.76
09:57:07.469 -> Stopping Robot (Target Reached)
09:57:07.542 -> Yaw : 90.04
09:57:07.542 -> Yaw difference: -0.28
09:57:07.574 -> Target Yaw : 89.76
09:57:07.607 -> Stopping Robot (Target Reached)
09:57:07.677 -> Yaw : 91.88
09:57:07.677 -> Yaw difference: -2.12
09:57:07.709 -> Target Yaw : 89.76
09:57:07.709 -> Stopping Robot (Target Reached)
09:57:07.812 -> Yaw : 92.14
09:57:07.812 -> Yaw difference: -2.38
09:57:07.849 -> Target Yaw : 89.76
09:57:07.849 -> Stopping Robot (Target Reached)
09:57:07.919 -> Yaw : 92.11
09:57:07.953 -> Yaw difference: -2.34
09:57:07.953 -> Target Yaw : 89.76
09:57:07.985 -> Stopping Robot (Target Reached)
09:57:08.058 -> Yaw : 92.07
```

(c)

```
09:57:17.288 -> Target Yaw set to 0
09:57:17.323 -> Yaw : 88.94
09:57:17.323 -> Yaw difference: -88.94
09:57:17.355 -> Target Yaw : 0.00
09:57:17.396 -> Rotating Left
09:57:17.431 -> Yaw : 88.93
09:57:17.463 -> Yaw difference: -88.93
09:57:17.463 -> Target Yaw : 0.00
09:57:17.496 -> Rotating Left
09:57:17.533 -> Yaw : 88.93
09:57:17.566 -> Yaw difference: -88.93
09:57:17.598 -> Target Yaw : 0.00
09:57:17.598 -> Rotating Left
09:57:17.674 -> Yaw : 88.93
```

(d)

```
09:57:20.461 -> Stopping Robot (Target Reached)
09:57:20.527 -> Yaw : 3.35
09:57:20.560 -> Yaw difference: -3.35
09:57:20.593 -> Target Yaw : 0.00
09:57:20.593 -> Stopping Robot (Target Reached)
09:57:20.667 -> Yaw : 3.34
09:57:20.703 -> Yaw difference: -3.34
09:57:20.703 -> Target Yaw : 0.00
09:57:20.735 -> Stopping Robot (Target Reached)
09:57:20.802 -> Yaw : 2.86
09:57:20.802 -> Yaw difference: -2.86
09:57:20.836 -> Target Yaw : 0.00
09:57:20.869 -> Stopping Robot (Target Reached)
```

(e)

Figure 24: User defined rotation

The robot begins in its default home position, set at 0 degrees, meaning it is directly aligned to its beginning orientation. When the user inputs a command for the robot to rotate, for example 90 degrees, the robot will begin executing a controlled turn to the right. For this to happen the MPU6050 gyroscope sensor continually measures the robot's angular velocity and orientation. The robot will actively adjust its motion, meaning the robot will control its speed of turn and the robot will maintain control stability. As the robot turns, it will continually reference its yaw angle versus its target yaw angle (90 degrees). As the robot approaches the target yaw angle, which is 90 degrees in this example, the robot will begin to slow its turn to mitigate an overshoot. Control logic functions, so that the yaw angle difference is a little less than 3 degrees, the robot will start to make micro-adjustments then finally arrive at a stop at exactly 90 degrees. The small tolerance prevents minor sensor offset and assures the robot to be aligned to the precise number of degrees. The robot will remain at a 90-degree position until another command is entered. If the user enters a command instructing the robot to return to home position (0 degrees), the robot will execute the same controlled procedure to return home. The robot starts rotating **left** this time, gradually decreasing the yaw angle until it once again reaches a yaw difference of less than **3 degrees**, ensuring a smooth and controlled return to its original orientation.

Figure (a): Robot home position (0 degrees)

Figure (b): Rotating 90 degrees

Figure (c): Target reached

Figure (d): Rotating back to home position

Figure (e): home position reached

5.1.2. Speed control (PID)

This test is to make the robot move in a control and consistent speed. For that a PID controller has been implicated and tuned properly so as to make the robot move in a controlled speed reducing the fluctuations and instability, deviations from the provided desired speed.

Objective of the test:

The main focus of this test is to examine the performance of the PID controller to maintain the speed of the robot under a variety of conditions. This will allow us to assure the speed is constant regardless of the different external conditions.

```
11:47:01.521 -> Tilt Angle: 4.49 | PID Output: 0.03 | Motor Speed: 83
11:47:01.587 -> Tilt Angle: 4.41 | PID Output: 0.28 | Motor Speed: 83
11:47:01.622 -> Tilt Angle: 4.37 | PID Output: 0.40 | Motor Speed: 83
11:47:01.686 -> Tilt Angle: 4.43 | PID Output: 0.20 | Motor Speed: 83
11:47:01.751 -> Tilt Angle: 4.33 | PID Output: 0.50 | Motor Speed: 83
11:47:01.784 -> Tilt Angle: 4.35 | PID Output: 0.46 | Motor Speed: 83
11:47:01.849 -> Tilt Angle: 4.33 | PID Output: 0.51 | Motor Speed: 83
11:47:01.924 -> Tilt Angle: 4.29 | PID Output: 0.63 | Motor Speed: 83
11:47:02.001 -> Tilt Angle: 4.39 | PID Output: 0.34 | Motor Speed: 83
11:47:02.034 -> Tilt Angle: 4.38 | PID Output: 0.35 | Motor Speed: 83
11:47:02.099 -> Tilt Angle: 4.37 | PID Output: 0.38 | Motor Speed: 83
11:47:02.132 -> Tilt Angle: 4.45 | PID Output: 0.16 | Motor Speed: 83
11:47:02.198 -> Tilt Angle: 4.42 | PID Output: 0.24 | Motor Speed: 83
11:47:02.263 -> Tilt Angle: 4.40 | PID Output: 0.31 | Motor Speed: 83
11:47:02.329 -> Tilt Angle: 4.39 | PID Output: 0.34 | Motor Speed: 83
11:47:02.362 -> Tilt Angle: 4.39 | PID Output: 0.33 | Motor Speed: 83
11:47:02.427 -> Tilt Angle: 4.45 | PID Output: 0.16 | Motor Speed: 83
11:47:02.493 -> Tilt Angle: 4.41 | PID Output: 0.26 | Motor Speed: 83
11:47:02.558 -> Tilt Angle: 4.44 | PID Output: 0.19 | Motor Speed: 83
11:47:02.590 -> Tilt Angle: 4.46 | PID Output: 0.12 | Motor Speed: 83
11:47:02.656 -> Tilt Angle: 4.42 | PID Output: 0.25 | Motor Speed: 83
```

Figure 25*: PID output*

The information contained in the figure reveals the motor speed to be stable and unchanged at a specific PWM value of 83 for each data point. Accordingly, this means that the controller is performing its job of regulating the speed of the motor with no variations in motor speed. The PID controller is based on the tilt angle, the tilt angle can vary, but the final output can continuously moderate to stabilize this tilt angle. The values of the tilt angle have remained in a fairly small range of values as well, with the tilt angle only changing from approximately 4.29 and 4.49 degrees. The small change indicates the system is tuned correctly, and there is not a large or sudden disturbance to the tilt angle. The tilt angle also remains relatively stable, thus the PID output only fluctuated from roughly 0.03 and 0.63. This again indicates that the corrective action of the PID is minimal since there is not a large variability to warrant a large corrective action. This also indicates that the motor speed continues to not change without any disturbance to smooth operation.

5.1.3. Deviation correction (PID)

This test demonstrates that, by using a PID controller to correct errors in real-time, the robot will drive straight. The PID algorithm continuously surveys the yaw deviation from the MPU6050 sensor to adjust both motor speeds. The output is dynamically fine-tuned based on what has been the yaw deviation so that any drift is countered. This action stabilizes and thereby creates a controlled trajectory. Due to functionality of the PID, the robot will travel straight regardless of any external forces being forced through the robot or on the surface that the robot was traveling on.

```
12:19:34.206 -> Yaw Rate: 0.38 | Correction: -0.40
12:19:34.206 ->  left speed 82
12:19:34.206 ->  right speed 83
12:19:34.206 -> Yaw Rate: 0.24 | Correction: -0.40
12:19:34.238 ->  left speed 82
12:19:34.238 ->  right speed 83
12:19:34.238 -> Yaw Rate: 0.29 | Correction: -0.40
12:19:34.238 ->  left speed 82
12:19:34.238 ->  right speed 83
12:19:34.238 -> Yaw Rate: 0.18 | Correction: -0.40
12:19:34.238 ->  left speed 82
12:19:34.238 ->  right speed 83
12:19:34.271 -> Yaw Rate: 0.30 | Correction: -0.40
12:19:34.271 ->  left speed 82
12:19:34.271 ->  right speed 83
12:19:34.271 -> Yaw Rate: 0.23 | Correction: -0.40
12:19:34.271 ->  left speed 82
12:19:34.271 ->  right speed 83
12:19:34.271 -> Yaw Rate: 0.22 | Correction: -0.53
12:19:34.304 ->  left speed 82
12:19:34.304 ->  right speed 83
12:19:34.304 -> Yaw Rate: 0.22 | Correction: -0.53
12:19:34.304 ->  left speed 82
12:19:34.304 ->  right speed 83
12:19:34.304 -> Yaw Rate: 0.25 | Correction: -0.53
```

Figure 26*: Yaw correction*

From the observations, it can be noted that when the yaw rate varies, the system follows a dynamic adjustment to correct the deviation by reducing and increasing the speeds of both the right and left motors. This is done via a PID-based yaw correction, which continually measures the deviation before applying proportional adjustments to the respective motor speeds. The controller increases or decreases the speed of either motor to steer the robot back on track if the robot begins to deviate from its intended course. In this way, it can be ensured that the robot continues on a consistent straight line, even when increased surface impacts or disturbances affect the robot.

5.1.4. Path Follower

This test follows a pre-defined path based on time which combines both PID (speed control) and rotation.

Objective of the test:

The robot's ability to accurately follow the pre-defined path with turns and navigate back to the home position (starting point).

During the test, the robot's deviation from the desired path is recorded and noted down. The path deviation and completion time are the variables documented to analyse the use of the robot's navigation systems and its effectiveness and stability. Path deviation represents the maximum distance away from the programmed path. Completion time represents the total time taken to travel the path and return to the home position. These variables allow for analysis, and tuning of the robot's performance by assessing the algorithms created for path following and for further evaluating the corrections allowed for created the robot's response to path deviations.

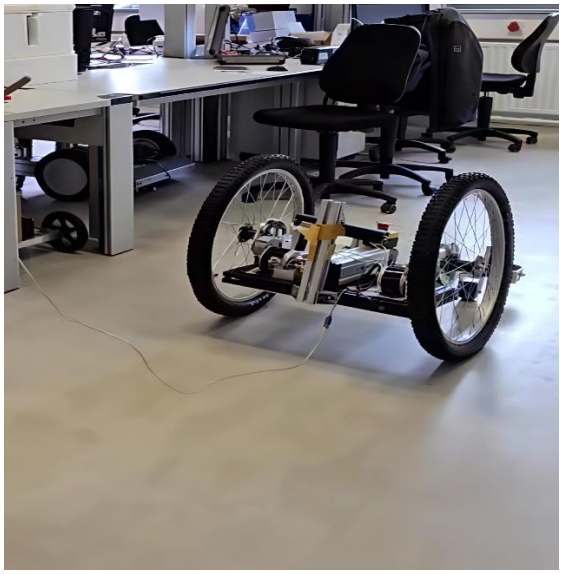

Figure 27: Path follower

5.1.5. Stairclimbing

The most important test of all, which is stair climbing has been tested here. It plays a vital role in making the robot climb the stairs. The test could be divided into 3 stages:

5.1.5.1. Step detection:

The next step will check the robot's ability to detect a step, or object height. The robot will rely primarily on the Z axis acceleration data collected from the MPU6050 sensor. The robot will

compare Z acceleration instantaneously as it will detect steps relatively the same every time, and as with other adjustments, the robot must be able to detect step and make any other adjustments necessary to complete the climb; the goal is for the action to occur accurately with minimal time delay. And the most important part here is the surge in the motor speed takes place here which is after it detects the step.

```
Pitch: 3.04° | Z-Accel Change: 0.16
Pitch: 2.24° | Z-Accel Change: 0.71
Pitch: 2.23° | Z-Accel Change: 0.00
Pitch: 2.25° | Z-Accel Change: 0.01
Pitch: 2.27° | Z-Accel Change: 0.02
Pitch: 2.28° | Z-Accel Change: 0.04
Pitch: 2.29° | Z-Accel Change: 0.03
Pitch: 2.30° | Z-Accel Change: 0.01
Pitch: 2.31° | Z-Accel Change: 0.02
Pitch: 2.31° | Z-Accel Change: 0.01
Pitch: 2.32° | Z-Accel Change: 0.01
Pitch: 2.09° | Z-Accel Change: 0.20
Pitch: -1.18° | Z-Accel Change: 0.84
Pitch: 4.01° | Z-Accel Change: 1.07
Pitch: 1.32° | Z-Accel Change: 2.64
⚠ Step detected! (Sudden Z-acceleration change)
Pitch: -0.26° | Z-Accel Change: 0.66
```

Figure 28:Step detection

The results demonstrate that there is a clear increase in the acceleration in the Z-axis when the robot touches the stair. This sudden increase in vertical acceleration is an obvious indication that there is a barrier (i.e., step) in the robot's path. The system is programmed to detect this spike as a boundary for detecting a step, thus upon exceeding a threshold, the message "Step Detected" is prompted. This means that the robot has detected the first contact with the stair and it can, therefore, enter the climbing phase.

5.1.5.2. Climbing detection and Speed control (Roll):

Here the climbing detection has been tested using pitch angle. A change in pitch angle would indicate that it has started climbing. And after climbing detection the speed control starts using roll angle so that wheels do not overlap.

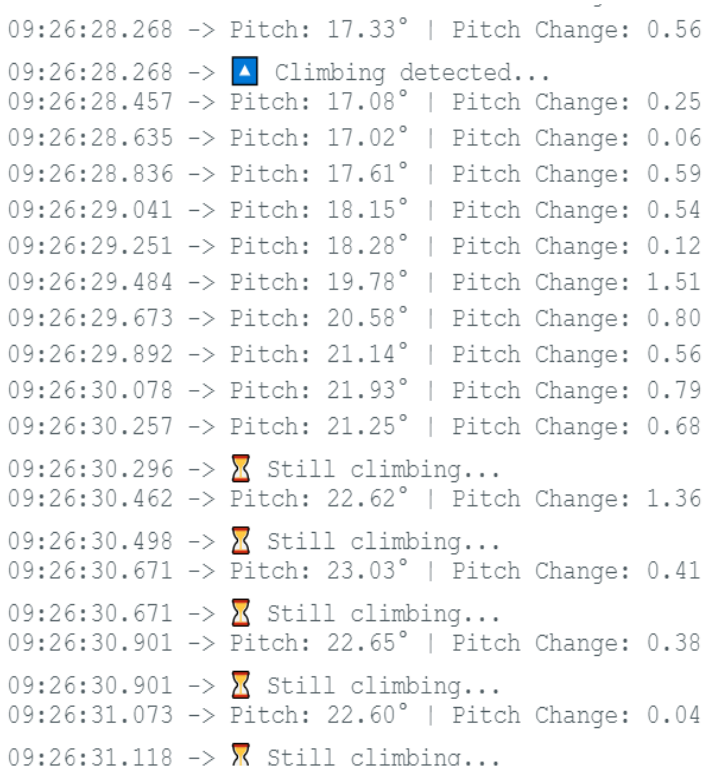

Figure 29:Climbing detection

The output shows that once the system detects the initiation of climbing stairs, it goes into a dynamic control mode that adjusts the control speed of the motors in real-time to provide smooth and stable stair climbing. As the roll angle varies as a result of changing balance and surface incline, the motors speeds were adjusted accordingly to avoid wheel slip, or overlap. Printed motor speed during these phases clearly demonstrated the responsive adjustment mechanism of the system and evidenced the success of step climbing with roll-speed based control.

5.2. Interpretation of the Result

5.2.1 Accuracy evaluation of user defined rotation

Angle (°)	Samples	Measured angle (°)	Error (°)	Mean (°)	Standard deviation (°)
-45°	Test 1	-42.4	2.6	2.23	0.286
	Test 2	-42.8	2.2		
	Test 3	-43.1	1.9		
90°	Test 1	87.50	2.1	1.94	0.253
	Test 2	88.41	1.59		
	Test 3	88.27	1.73		
180°	Test 1	177.29	2.71	2.52	0.151
	Test 2	177.66	2.34		
	Test 3	177.48	2.52		

Table 1*: Rotation Accuracy evaluation*

Mean Absolute error:

$$MAE=\frac{\Sigma error}{samples}$$

$$MAE=\frac{2.23+1.94+2.52}{3}=2.23$$

Analysis:

From the table data and calculations presented, it can confidently be concluded that the variation in the angles measured has a very low standard deviation, which indicates that after the multiple runs, the robot's rotation is very much consistent with the desired angle. A low standard deviation indicates consistency/reliability in the control system since the achieved angles are varying very little after multiple runs. Finally, the Mean Absolute Error (MAE) value, which captures the average amount of deviation from the commanded angles, was also low. A low MAE value also means that the robot is able to follow the commanded angles with very little deviation, which indicates very accurate rotation by the robot. Since both the standard deviation from the mean angles as well as the MAE values were low, the robotic rotation is said to be quite accurate and amenable to programs which require angular precision. If we were to want even lower values of small error, we could attempt to modify the tolerance values, calibrate

individual sensors with more precision, and/or reduce the interference of friction or sensor noise. Overall, the robot's control system for rotation can be concluded to be well-designed and accurate within tolerable levels of accuracy in terms of stability and repeatability.

5.2.2. Accuracy evaluation of PID

TIME(SECS)	DISTANCE COVERED(METRES)			Mean(m)	Standard deviation(m)
	TEST 1	TEST 2	TEST 3		
8	2.60	2.55	2.45	2.53	0.0624
10	4.3	4.25	4.15	4.23	0.0624
12	4.99	5.10	5.05	5.05	0.0244

Table 2: *PID Accuracy evaluation*

5.2.2.1 Average Distance Covered

For 8 secs:

$$\text{Mean distance} = \frac{2.6 + 2.55 + 2.45}{3} = 2.53\text{m}$$

For 10 secs:

$$\text{Mean distance} = \frac{4.3 + 4.25 + 4.15}{3} = 4.23\text{m}$$

For 12 secs:

$$\text{Mean distance} = \frac{4.99 + 5.10 + 5.05}{3} = 5.05\text{m}$$

5.2.2.2. Standard deviation

For 8 secs:

$$\sigma = \sqrt{\frac{(2.6-2.53)^2+(2.55-2.53)^2+(2.45-2.53)^2}{3}} = 0.06244\text{m}$$

For 10 secs:

$$\sigma = \sqrt{\frac{(4.3-4.23)^2+(4.25-4.23)^2+(4.15-4.23)^2}{3}} = 0.06244\text{m}$$

For 12 secs:

$$\sigma = \sqrt{\frac{(4.99-5.05)^2+(5.10-5.05)^2+(5.05-5.05)^2}{3}} = 0.02449\text{m}$$

Analysis:

From the observation and testing, it is clearly visible that the standard deviation is significantly low which means, the speed is quite consistent and stable. This demonstrates that PID controller is working effectively and correcting the fluctuations consistently. Although there are still some minor errors present, it remains in the acceptable range and does not impact the overall performance. The reasons for these errors could be sensor noise, environmental influences etc.

5.2.3. Accuracy evaluation of Yaw deviation

5.2.3.1. Yaw deviation with and without PID

PID

DISTANCE TRAVELLED(M)	YAW RATE ()	
	INITIAL YAW	FINAL YAW
2	0.31	0.60
4	0.37	0.71
6	0.45	1.42

Without PID

DISTANCE TRAVELLED(M)	YAW RATE ()	
	INITIAL YAW	FINAL YAW
2	0.28	1.44
4	0.27	2.56
6	0.37	3.44

Table 3*: Yaw rate with and without PID*

5.2.3.2. Yaw rate Error analysis

TIME(S)	YAW RATE		ERROR	
	With PID	Without PID	With PID	Without PID
0	0.24	0.31	0.24	0.31
1	0.29	0.47	0.29	0.47
2	0.18	0.17	0.18	0.17
3	0.30	0.72	0.30	0.72
4	0.23	-0.64	0.23	-0.64
5	0.27	0.37	0.27	0.37
6	0.26	-0.47	0.26	-0.47
7	0.21	-1.27	0.21	-1.27
8	0.23	1.10	0.23	1.10
9	0.15	0.19	0.15	0.19

Table 4*: Yaw rate error*

Root Mean Square Error calculation:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum(Yaw\ Rate\ Error)^2}$$

Without PID:

RMSE = 0.67012°/s

With PID:

RMSE = 0.2402°/s

Analysis:

From the RMSE values it is evident that, the PID controller has reduced the Yaw Rate error by 60%, while making the robot move in a straight line without much deviation.

And the low RMSE value depicts better stability and minimal deviation from its straight-line path.

6. Conclusion

The thesis details the design and development of a mobile robot equipped with intelligent control systems and real-time sensors feedback to facilitate navigation in a complex environment. The robot is supported by an MPU6050 IMU that offers continual monitoring of motion and orientation useful for several capabilities for the robot such as detecting stairs, yaw deviation correction, and rotation directionality which the user can define. Most importantly, one of the main aims of the project had been realized; the robot can detect stairs through monitoring and implementing Z-axis accelerometer measurements and pitch angle and adjust the motor control speeds to initiate the climb.The robot has not yet experienced full stair climbing, but it experienced enough motion detection of the stall to define the initiation of the climb process and implemented the appropriate motor control logic. Additionally, during the stair climbing, the robot was able to answer roll angle feedback monitoring to maintain lateral stability during partial climb ascent to avoid wheels overlapping and augment stability. While it also features a well-tuned PID controller for speed consistency and specific yaw PID loop for correcting directional drift, the control systems were verified through numerous straight-line and rotation demonstrations. It is also able to efficiently follow a programmed path and return to Home producing reliability in navigation and motion plans.

In general, the system builds an excellent basis for a solid and flexible mobile platform. Although completely climbing stairs is still in progress and development, the functions that have been implemented: stair detection, speed control, rotation logic, and navigation, illustrate the robot's potential role in assistive, indoor delivery, and semi-structured environments.

7. Future implementations

1. Integration of Encoders for Speed Control:
   - Wheel encoders could provide more accurate values than MPU 6050 making it much easier to calculate the current speed which produces more efficiency than the current.
2. Real-Time PID Tuning:
   - At present, PID parameters are set using a manual process that does require trial and error. In the future, having a real-time or adaptive PID tuning feature would allow the system to adjust its control parameters automatically based on the environment around it or changes robot action. This will help its performance in tasks such as stabilization, rotation, and speed control in different terrain in the future.
3. Stabilization Rod Design Enhancement:
   - The robot's physical stability while climbing stairs could be modified through variations in the stabilizing rod or support structure. A new or actively controlled stabilizer can minimize roll and pitch motion while climbing, creating a smoother and safer climbing experience, and reducing the chance for either wheels to slip off alignment, especially on uneven or steep stairs.

8. References

1) Cao, X., Huang, L., & Wang, Y. (2018). "Design and Development of an Autonomous Stair-Climbing Robot for Indoor Environments." International Journal of Robotics Research, 37(3), 307-319.
   i) This paper discusses the design and development of stair-climbing robots for indoor environments, focusing on sensor-based navigation and autonomous control.
2) Zhu, J., Wang, Z., & Liu, Y. (2020). "Stair-Climbing Robots: Challenges and Opportunities." IEEE Robotics & Automation Magazine, 27(1), 49-59.
   i) This review highlights the challenges faced by stair-climbing robots in real-world applications and discusses various methods to overcome these obstacles.
3) Yu, H., Shen, W., & Ma, Y. (2017). "Adaptive Stair-Climbing Mechanisms for Mobile Robots in Dynamic Environments." Journal of Robotics and Autonomous Systems, 91(2), 120-130.
   i) A study on adaptive stair-climbing mechanisms that focus on dynamic environments, emphasizing the need for real-time control and sensor feedback.
4) Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). "Introduction to Autonomous Mobile Robots." MIT Press, 2nd Edition.
   i) This book provides foundational knowledge on autonomous mobile robots, including locomotion, sensing, and navigation, which is critical for understanding stair-climbing robots.
5) Yu, H., Liang, Z., & Kim, J. (2019). "A Study on the Mobility and Control of Wheeled Stair-Climbing Robots." Journal of Mechanisms and Robotics, 11(4), 045001.
   i) This paper examines the control and mobility challenges of wheeled stair-climbing robots and discusses potential improvements using real-time sensor data.
6) Deng, Z., Wu, H., & Zhang, X. (2021). "Wireless Control Systems for Autonomous Stair-Climbing Robots: A Review of Technologies and Trends." Sensors, 21(10), 3256.
   i) A comprehensive review of wireless control systems used in autonomous robots, highlighting Bluetooth technologies for real-time control and monitoring.
7) Kawaguchi, Y., Matsuo, Y., & Kato, K. (2018). "Real-Time IMU-Based Feedback Systems for Autonomous Mobile Robots." Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 1(4), 1345-1350.
   i) This paper focuses on the use of IMUs for real-time feedback in autonomous robots, with applications in stair-climbing mechanisms.
8) Kim, H., Lee, K., & Park, J. (2020). "Development of Real-Time Control Algorithms for Stair-Climbing Robots." Journal of Mechatronics and Automation, 11(3), 233-245.
   i) A study focused on control algorithms for stair-climbing robots, specifically on integrating IMUs and Bluetooth communication for remote monitoring and control.
9) Ogata, K. (2010). Modern Control Engineering (5th ed.). Pearson.
   i) A comprehensive textbook on control systems, including PID control theory and applications.
10) Nayan Jyoti Baishya, Bishakh Bhattacharya, Harutoshi Ogai and Kohei Tatsumi (2021) “Analysis and Design of a Minimalist Step Climbing Robot”, 11-07044-v2

i) A study focused on control algorithms for staircase climbing with connecting links and IMUs.

11) TaeWon Seo, Sijun Ryu, Jee Ho Won, Youngsoo Kim, and Hwa Soo Kim (2023) "Stair-climbing Robots: a Review on Mechanism, Sensing, and Performance Evaluation"

12) Donato Di Paola, Annalisa Milella, Grazia Cicirelli and Arcangelo Distante "An Autonomous Mobile Robotic System for Surveillance of Indoor Environments"

13) Aström, K. J., & Hägglund, T. (1995). PID Controllers: Theory, Design, and Tuning (2nd ed.). Instrument Society of America.

- A detailed guide on PID controller design and tuning.

14) Legged robot (online)

15) Wheeled robot (online)

16) Hybrid systems (online)

17) MPU 6050 (online)

18) Motor (online)

19) Motor driver (online)

20) Proposed model Robot with Links (online)

9. Appendices

9.1 Design parameters and Mathematical modelling:

9.1.1 Design parameters:

The robot is designed as a five-link mechanism where the front wheels and rear wheels are connected to the robot body through revolute joints. The geometry and kinematics of the robot is defined by the following parameters:

$$\theta_3 = 270° + \sin^{-1}\left(\frac{\sqrt{l_3^2 - (l_{1y} + l_2\sin(\theta_2^0 - \theta) + (A_y - B_y))^2}}{l_3}\right)$$

This equation governs the angle between the connecting link and the horizontal as the robot ascends the stairs.

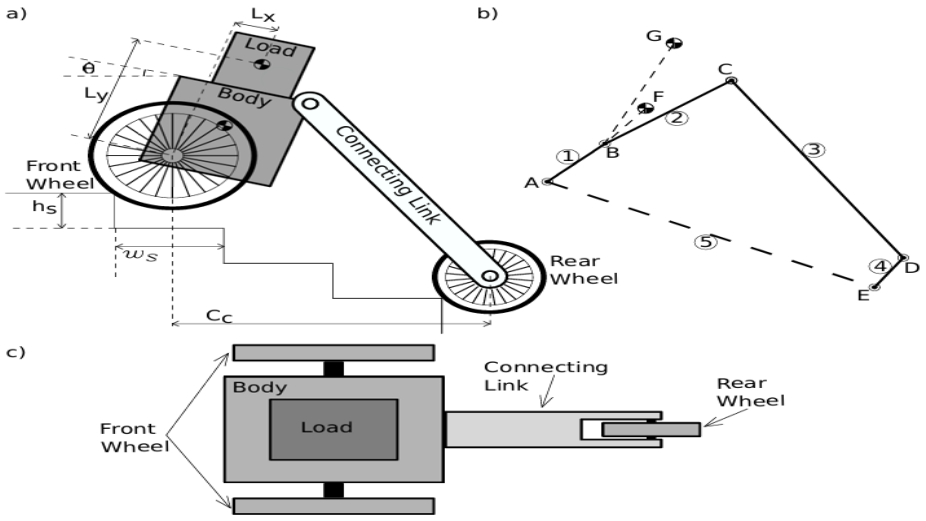

Figure 30: Angle Measurement [10]

9.1.2 Wheel Trajectory on Stairs

The trajectory of the wheels as they climb the steps depends on the relationship between the **radius of the wheels** and the **height of the step.**

$$y_o = \begin{cases} R_w & \text{if } AB > x_o - x_i \\ \sqrt{R_w^2 - (x_o - x_i)^2} & \text{if } AB \leq x_o - x_i \end{cases}$$

Above formula describes about the vertical motion of the wheel when it encounters a step edge.

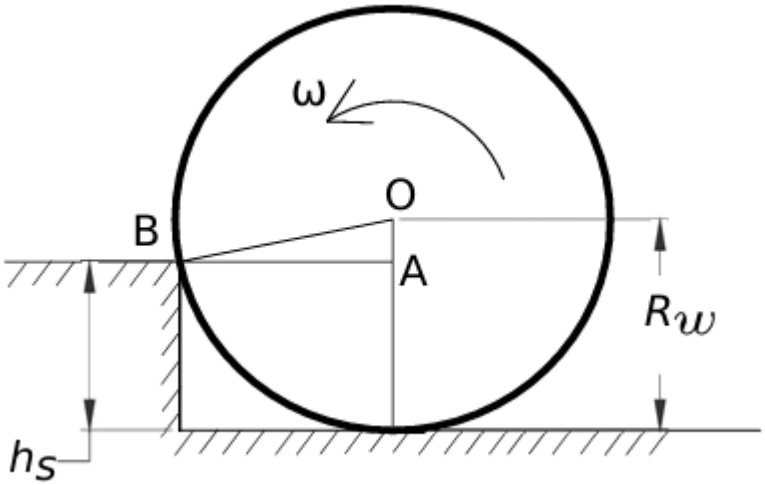

Figure 31: Wheel Trajectory [10]

9.1.3 Connecting Link Length Calculation

To maintain stability, the length of the connecting link between the rear wheel and the robot body is a critical design parameter:

$$l_3 = \sqrt{(D_x - C_x)^2 + (D_y - C_y)^2}:$$

Above equation is used to calculate the minimum length required for perfect stair climbing from the position of the rear wheel and robot.

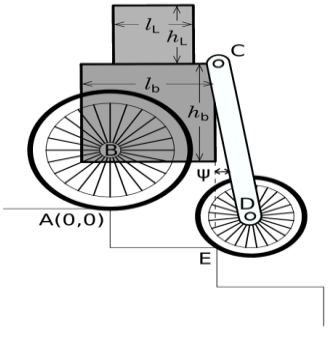

Figure 32:Connecting link length [10]

9.1.4 Kinematic Analysis

The kinematic model establishes the relationships between the positions, velocities, and accelerations of the robot's links:

$$l_1 e^{j\theta_1} + l_2 e^{j\theta_2} + l_3 e^{j\theta_3} + l_4 e^{j\theta_4} + l_5 e^{j\theta_5} = 0:$$

This complex equation represents the position of the links in the robot's kinematic chain, where each link's motion is defined by its angular displacement θ.

Velocity equation:

$$l_1 j\omega_1 e^{j\theta_1} + l_2 j\omega_2 e^{j\theta_2} + l_3 j\omega_3 e^{j\theta_3} + l_4 j\omega_4 e^{j\theta_4} + l_5 j\omega_5 e^{j\theta_5} = 0:$$

Acceleration equation:

$$l_1(j\alpha_1 - {\omega_1}^2)e^{j\theta_1} + l_2(j\alpha_2 - {\omega_2}^2)e^{j\theta_2} + l_3(j\alpha_3 - {\omega_3}^2)e^{j\theta_3} + l_4(j\alpha_4 - {\omega_4}^2)e^{j\theta_4} + l_5(j\alpha_5 - {\omega_5}^2)e^{j\theta_5} = 0$$

9.2 Source code

9.2.1 User defined rotation

```
#include "I2Cdev.h"

#include "MPU6050_6Axis_MotionApps20.h"


MPU6050 mpu;


#define OUTPUT_READABLE_YAWPITCHROLL


int const INTERRUPT_PIN = 2;

bool blinkState;


bool DMPReady = false;

uint8_t MPUIntStatus;

uint8_t devStatus;

uint16_t packetSize;

uint8_t FIFOBuffer[64];
```

```
Quaternion q;
VectorFloat gravity;
float ypr[3];
const int MotorRight = 5;
const int MotorLeft = 9;
const int MotorRightBack = 12;
const int MotorLeftBack = 8;
const int MotorRightBack_1 = 11;
const int MotorLeftBack_1 = 7;
float targetYaw = 0;
float yaw = 0;
volatile bool MPUInterrupt = false;
void DMPDataReady() {
    MPUInterrupt = true;
}

void setup() {
  #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    Wire.begin();
    Wire.setClock(400000);
  #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
    Fastwire::setup(400, true);
  #endif

  Serial.begin(9600);
  while (!Serial);

  Serial.println(F("Initializing I2C devices..."));
  mpu.initialize();
```

```
pinMode(INTERRUPT_PIN, INPUT);

Serial.println(F("Testing MPU6050 connection..."));
if(mpu.testConnection() == false){
  Serial.println("MPU6050 connection failed");
  while(true);
} else {
  Serial.println("MPU6050 connection successful");
}

Serial.println(F("\nSend any character to begin: "));
while (Serial.available() && Serial.read());
while (!Serial.available());
while (Serial.available() && Serial.read());

Serial.println(F("Initializing DMP..."));
devStatus = mpu.dmpInitialize();

mpu.setXGyroOffset(0);
mpu.setYGyroOffset(0);
mpu.setZGyroOffset(0);
mpu.setXAccelOffset(0);
mpu.setYAccelOffset(0);
mpu.setZAccelOffset(0);

if (devStatus == 0) {
  mpu.CalibrateAccel(6);
  mpu.CalibrateGyro(6);
  Serial.println("These are the Active offsets: ");
  mpu.PrintActiveOffsets();
```

```
    Serial.println(F("Enabling DMP..."));
    mpu.setDMPEnabled(true);

    Serial.print(F("Enabling interrupt detection..."));
    attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), DMPDataReady, RISING);
    MPUIntStatus = mpu.getIntStatus();

    Serial.println(F("DMP ready! Waiting for first interrupt..."));
    DMPReady = true;
    packetSize = mpu.dmpGetFIFOPacketSize();
  } else {
    Serial.print(F("DMP Initialization failed (code "));
    Serial.print(devStatus);
    Serial.println(F(")"));
  }

  pinMode(MotorLeft, OUTPUT);
  pinMode(MotorRight, OUTPUT);
  pinMode(MotorRightBack, OUTPUT);
  pinMode(MotorLeftBack, OUTPUT);
  pinMode(MotorRightBack_1, OUTPUT);
  pinMode(MotorLeftBack_1, OUTPUT);
  Serial.println("Input '0', '9', '-9', '18', or '-4' to set target yaw.");
}

void loop() {
  if (!DMPReady) return;
  if (mpu.dmpGetCurrentFIFOPacket(FIFOBuffer)) {
    mpu.dmpGetQuaternion(&q, FIFOBuffer);
    mpu.dmpGetGravity(&gravity, &q);
```

```
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

    yaw = ypr[0] * 180/M_PI;
  }
  updateTargetYaw();
  controlRotation();

  delay(100);
}

void controlRotation() {
  float yawDifference = targetYaw - yaw;

  // // Normalize the yaw difference to between -180° and 180°
  if (yawDifference > 180) {
    yawDifference -= 360;
  } else if (yawDifference < -180) {
    yawDifference += 360;
  }
  Serial.print("Yaw : ");
  Serial.println(yaw);
  Serial.print("Yaw difference: ");
    Serial.println(yawDifference);
  Serial.print("Target Yaw : ");
    Serial.println(targetYaw);
  if (abs(yawDifference) < 5) {
    analogWrite(MotorLeft, 0);
    analogWrite(MotorRight, 0);
    Serial.println("Stopping Robot (Target Reached)");
  }
```

```
  else if (yawDifference > 0) {



    digitalWrite(MotorLeftBack, HIGH);
    digitalWrite(MotorLeftBack_1, HIGH);
    digitalWrite(MotorRightBack, LOW);
    digitalWrite(MotorRightBack_1, LOW);
    analogWrite(MotorLeft, 85);
    analogWrite(MotorRight, 85);
    Serial.println("Rotating Right");
    }
  else {
    digitalWrite(MotorLeftBack, LOW);
    digitalWrite(MotorLeftBack_1, LOW);
    digitalWrite(MotorRightBack, HIGH);
    digitalWrite(MotorRightBack_1, HIGH);
    analogWrite(MotorLeft, 85);
    analogWrite(MotorRight, 85);
    Serial.println("Rotating Left");
  }
}

void updateTargetYaw() {
  if (Serial.available()) {
    char input = Serial.read();
    if (input == '0') {
      targetYaw = 0;
      Serial.println("Target Yaw set to 0");
    } else if (input == '9') {
      targetYaw = yaw + 90;
```

```
      Serial.println("Target Yaw set to 90");
    } else if (input == '-') {
      input = Serial.read();
      if (input == '9') {
        targetYaw = yaw - 90;
        Serial.println("Target Yaw set to -90");
      } else if (input == '4') {
        targetYaw = yaw - 45;
        Serial.println("Target Yaw set to -45");
      }
    } else if (input == '1') {
      targetYaw = yaw + 180;
      Serial.println("Target Yaw set to 180");
    }

  }
}
```

8.2.2 PID (speed control)

```
#include <Wire.h>
#include <MPU6050.h>

MPU6050 mpu;

// PID Variables
float Kp = 3.0, Ki = 0, Kd = 0;  // Tune these values
float error, prevError, integral, derivative;
float pidOutput;

// Tilt Angle Variables
```

```
float accelAngle, gyroAngle, filteredAngle;
float lastTime, dt;
float alpha = 0.98;
float baseTilt = 0;

// Motor PWM Settings
const int motorPWM = 83;
const int motorleft = 9;
const int motorright = 5;

void getMPU6050Data(float &accelAngle, float &gyroRate) {
    int16_t ax, ay, az, gx, gy, gz;
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

    accelAngle = atan2(ay, az) * (180.0 / PI);
    gyroRate = gx / 131.0;
}

void setup() {
    Serial.begin(9600);
    Wire.begin();
    mpu.initialize();

    if (!mpu.testConnection()) {
        Serial.println("MPU6050 connection failed!");
        while (1);
    }

    delay(1000);
    getMPU6050Data(accelAngle, gyroAngle);
```

```
    baseTilt = accelAngle;
    filteredAngle =accelAngle  ;
    lastTime = millis() / 1000.0;

    pinMode(motorleft, OUTPUT);
    pinMode(motorright, OUTPUT);
}

void loop() {
    float gyroRate;
    getMPU6050Data(accelAngle, gyroRate);
    float currentTime = millis() / 1000.0;
    dt = currentTime - lastTime;
    lastTime = currentTime;
    gyroAngle += gyroRate * dt;
    filteredAngle = alpha * (filteredAngle + gyroRate * dt) + (1 - alpha) * accelAngle;
    float relativeTilt = filteredAngle - baseTilt;
    float setpoint = 4.5;
    error = setpoint - relativeTilt;
    integral += error * dt;
    derivative = (error - prevError) / dt;
    pidOutput = (Kp * error);
    prevError = error;
    int motorSpeed = motorPWM + pidOutput;
    motorSpeed = constrain(motorSpeed, 0, 88);
    analogWrite(motorright, motorSpeed);
    analogWrite(motorleft, motorSpeed);
    Serial.print("Tilt Angle: ");
    Serial.print(relativeTilt);
```

```
    Serial.print(" | PID Output: ");
    Serial.print(pidOutput);
    Serial.print(" | Motor Speed: ");
    Serial.println(motorSpeed);

    delay(50);
}
```

9.2.3 PID (Deviation control)

```
#include <Wire.h>
#include <MPU6050.h>
#include <PID_v1.h>

MPU6050 mpu;

// Motor Control Pins
#define MOTOR_LEFT_PWM 9
#define MOTOR_RIGHT_PWM 5
#define MOTOR_LEFT_DIR 7
#define MOTOR_RIGHT_DIR 8

// PID Parameters
double setpoint = 0, input, output;
double Kp = 2.2, Ki = 0.09, Kd = 0.05;
PID yawPID(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);

void setup() {
    Serial.begin(115200);
    Wire.begin();
    mpu.initialize();
```

```
    if (!mpu.testConnection()) {
        Serial.println("MPU6050 connection failed");
        while (1);
    }
    Serial.println("MPU6050 connected");

    // Initialize Motors
    pinMode(MOTOR_LEFT_PWM, OUTPUT);
    pinMode(MOTOR_RIGHT_PWM, OUTPUT);
    pinMode(MOTOR_LEFT_DIR, OUTPUT);
    pinMode(MOTOR_RIGHT_DIR, OUTPUT);

    yawPID.SetMode(AUTOMATIC);
    yawPID.SetOutputLimits(-50, 50);
}

void loop() {
    int16_t ax, ay, az, gx, gy, gz;
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
    input = gz / 131.0;
    yawPID.Compute();
    int baseSpeed = 83;
    int leftSpeed = baseSpeed + output;
    int rightSpeed = baseSpeed - output;
    leftSpeed = constrain(leftSpeed, 0, 88);
    rightSpeed = constrain(rightSpeed, 0, 88);
    analogWrite(MOTOR_LEFT_PWM, leftSpeed);
    analogWrite(MOTOR_RIGHT_PWM, rightSpeed);
```

```
    Serial.print("Yaw Rate: "); Serial.print(input);
    Serial.print(" | Correction: "); Serial.println(output);
    Serial.print(" left speed "); Serial.println(leftSpeed);
    Serial.print(" right speed "); Serial.println(rightSpeed);

    delay(10);
}
```

9.2.4 Path follower

```
#include <Wire.h>
// #include <MPU6050.h>
#include "MPU6050_6Axis_MotionApps20.h"
#include <PID_v1.h>
#include "I2Cdev.h"
#include "MPU6050.h"

#define OUTPUT_READABLE_YAWPITCHROLL

int const INTERRUPT_PIN = 2;
bool blinkState;

Quaternion q;
VectorFloat gravity;
float ypr[3];

bool DMPReady = false;
uint8_t MPUIntStatus;
uint8_t devStatus;
uint16_t packetSize;
uint8_t FIFOBuffer[64];
```

```
float yaw = 0;
float targetYaw = 0;
volatile bool MPUInterrupt = false;
void DMPDataReady() {
    MPUInterrupt = true;
}

const int MotorRight = 5;
const int MotorLeft = 9;
const int MotorRightBack = 12;
const int MotorLeftBack = 8;
const int MotorRightBack_1 = 11;
const int MotorLeftBack_1 = 7;

unsigned long previousTime = 0;
double Setpoint, Input, Output;
double Kp = 3, Ki = 0, Kd = 0;
int16_t accelX, accelY, accelZ;
int16_t gyroX, gyroY, gyroZ;

float desiredSpeed = 90.0;
float actualSpeed = 0;

PID myPID(&Input, &Output, &Setpoint, Kp, Ki, Kd, DIRECT);

MPU6050 mpu;

unsigned long lastTime;
double timeStep = 100;
```

```
unsigned long startTime;
const unsigned long duration1 = 11000;
const unsigned long duration2 = 20000;
const unsigned long duration3 = 37000;
const unsigned long duration4 = 49000;
float yawDifference=0;
void updateTargetYaw();
void controlRotation();
void PID();
void Turn90();
void Turn180();
void Turnminus90();
void stopMotors();

void setup() {
  #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    Wire.begin();
    Wire.setClock(400000);
  #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
    Fastwire::setup(400, true);
  #endif
Serial.begin(9600);
  while (!Serial);

  Serial.println(F("Initializing I2C devices..."));
  mpu.initialize();
  pinMode(INTERRUPT_PIN, INPUT);

  Serial.println(F("Testing MPU6050 connection..."));
```

```
if(mpu.testConnection() == false){
  Serial.println("MPU6050 connection failed");
  while(true);
} else {
  Serial.println("MPU6050 connection successful");
}

Serial.println(F("\nSend any character to begin: "));
while (Serial.available() && Serial.read());
while (!Serial.available());
while (Serial.available() && Serial.read());

Serial.println(F("Initializing DMP..."));
devStatus = mpu.dmpInitialize();

mpu.setXGyroOffset(0);
mpu.setYGyroOffset(0);
mpu.setZGyroOffset(0);
mpu.setXAccelOffset(0);
mpu.setYAccelOffset(0);
mpu.setZAccelOffset(0);

if (devStatus == 0) {
  mpu.CalibrateAccel(6);
  mpu.CalibrateGyro(6);
  Serial.println("These are the Active offsets: ");
  mpu.PrintActiveOffsets();
  Serial.println(F("Enabling DMP..."));
  mpu.setDMPEnabled(true);
```

```
    Serial.print(F("Enabling interrupt detection..."));
    attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), DMPDataReady, RISING);
    MPUIntStatus = mpu.getIntStatus();

    Serial.println(F("DMP ready! Waiting for first interrupt..."));
    DMPReady = true;
    packetSize = mpu.dmpGetFIFOPacketSize();
  } else {
    Serial.print(F("DMP Initialization failed (code "));
    Serial.print(devStatus);
    Serial.println(F(")"));
  }
  Serial.println("Input '0', '9', '-9', '4', or '-4' to set target yaw.");
  pinMode(MotorRight, OUTPUT);
  pinMode(MotorLeft, OUTPUT);
  pinMode(MotorRightBack, OUTPUT);
  pinMode(MotorLeftBack, OUTPUT);
  pinMode(MotorRightBack_1, OUTPUT);
  pinMode(MotorLeftBack_1, OUTPUT);
  Setpoint = 33.3;
  myPID.SetMode(AUTOMATIC);
  myPID.SetOutputLimits(0, 255);
   startTime = millis();

  calibrateIMU();

  previousTime = millis();
  delay(1000);
}
```

```
void calibrateIMU()
{
  int numSamples = 1000;
  float sumY = 0;


  for (int i = 0; i < numSamples; i++) {
    int16_t ax, ay, az;
    mpu.getMotion6(&accelX, &accelY, &accelZ, &gyroX, &gyroY, &gyroZ);
    float gyroz = gyroZ / 16384.0;
    sumY += gyroz;
    delay(2);
  }


  float gyrozbias = (sumY / numSamples) * 9.81;
  Serial.print("gyroz bias: ");
  Serial.println(gyrozbias);
}

void loop()
{
    while (millis() - startTime <= duration1)
    {
        PID();
       Serial.println("PID 1 running ");
    }
    while (millis() - startTime <= duration1 + 6000)
    {
        Turn90();
```

```
  }
  while (millis() - startTime <= duration2 + 4000) {

    PID();
    Serial.println("PID 2 running ");
  }
  while (millis() - startTime <= duration2 + 10000) {
      Turn180();
  }
  while (millis() - startTime <= duration3) {
   PID();
  Serial.println("PID 3 running ");
 }
  while (millis() - startTime  <= duration3 + 8000) {
  Turnminus90();
 }
 while (millis() - startTime  <= duration4) {
  PID();
  Serial.println("PID 4 running ");
 }
    stopMotors();
    Serial.println("Reached home position ");
}

void PID(){
 mpu.getMotion6(&accelX, &accelY, &accelZ, &gyroX, &gyroY, &gyroZ);
 Input = abs(gyroZ / 131.0);
 Serial.print("Current Motor Speed (dps): ");
 Serial.println(Input);
 myPID.Compute();
```

```
  Serial.print("PID Output (PWM): ");
  Serial.println(Output);
  int motorSpeed = constrain(abs(Output), 0, 85);
  analogWrite(MotorRight, motorSpeed);
  analogWrite(MotorLeft, motorSpeed);
  Serial.print("Motor Speed PWM: ");
  Serial.println(motorSpeed);
  delay(timeStep);

}

void Turn90()
{
   targetYaw = 90;
    if (mpu.dmpGetCurrentFIFOPacket(FIFOBuffer))
    {

      mpu.dmpGetQuaternion(&q, FIFOBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

      yaw = ypr[0] * 180/M_PI;

    }
    yawDifference = targetYaw - yaw;
    Serial.print("Yaw : ");
    Serial.println(yaw);
    Serial.print("Yaw difference: ");
    Serial.println(yawDifference);
```

```
if (abs(yawDifference) < 3)
{
  stopMotors();
  analogWrite(MotorLeft, 0);
  analogWrite(MotorRight, 0);
  Serial.println("Stopping Robot 90 (Target Reached)");
}
else if (yawDifference > 0)
{
  digitalWrite(MotorLeftBack, HIGH);
  digitalWrite(MotorLeftBack_1, HIGH);
  digitalWrite(MotorRightBack, LOW);
  digitalWrite(MotorRightBack_1, LOW);

analogWrite(MotorLeft, 80);
analogWrite(MotorRight, 85);
Serial.println("Rotating Right");
}
else
{
    digitalWrite(MotorLeftBack, LOW);
    digitalWrite(MotorLeftBack_1, LOW);
    digitalWrite(MotorRightBack, HIGH);
    digitalWrite(MotorRightBack_1, HIGH);

  analogWrite(MotorLeft, 80);
  analogWrite(MotorRight, 85);
  Serial.println("Rotating Left");

}
```

```
}

void Turn180()
{
   targetYaw = 270;
    if (mpu.dmpGetCurrentFIFOPacket(FIFOBuffer))
    {

      mpu.dmpGetQuaternion(&q, FIFOBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

      yaw = ypr[0] * 180/M_PI;

    }
  yawDifference = targetYaw - yaw;
  if (yawDifference > 180) {
    yawDifference -= 360;
  } else if (yawDifference < -180) {
    yawDifference += 360;
  }
    Serial.print("Target Yaw ");
    Serial.println(targetYaw);
    Serial.print("Yaw : ");
    Serial.println(yaw);
    Serial.print("Yaw difference: ");
    Serial.println(yawDifference);

  if (abs(yawDifference) <4)
  {
```

```
        stopMotors();
        analogWrite(MotorLeft, 0);
        analogWrite(MotorRight, 0);
        Serial.println("Stopping Robot 180 (Target Reached)");

 } else if (yawDifference > 0)
  {
        digitalWrite(MotorLeftBack, HIGH);
        digitalWrite(MotorLeftBack_1, HIGH);
        digitalWrite(MotorRightBack, LOW);
        digitalWrite(MotorRightBack_1, LOW);
        analogWrite(MotorLeft, 80);
        analogWrite(MotorRight, 85);
        Serial.println("Rotating Right 180");
 }   else  {
       digitalWrite(MotorLeftBack, LOW);
       digitalWrite(MotorLeftBack_1, LOW);
       digitalWrite(MotorRightBack, HIGH);
       digitalWrite(MotorRightBack_1, HIGH);

     analogWrite(MotorLeft, 80);
     analogWrite(MotorRight, 85);
     Serial.println("Rotating Left");

         }
}

void Turnminus90()
{
 targetYaw = 0;
```

```
  if (mpu.dmpGetCurrentFIFOPacket(FIFOBuffer))
    {

      mpu.dmpGetQuaternion(&q, FIFOBuffer);
      mpu.dmpGetGravity(&gravity, &q);
      mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

      yaw = ypr[0] * 180/M_PI;

    }
yawDifference = targetYaw + yaw;
if (yawDifference > 180) {
    yawDifference -= 360;
  } else if (yawDifference < -180) {
    yawDifference += 360;
  }
    Serial.print("Target Yaw ");
    Serial.println(targetYaw);
    Serial.print("Yaw : ");
    Serial.println(yaw);
    Serial.print("Yaw difference: ");
    Serial.println(yawDifference);
if (abs(yawDifference) > 177 && abs(yawDifference) < 180) {
      stopMotors();
      analogWrite(MotorLeft, 0);
      analogWrite(MotorRight, 0);
    Serial.println("Stopping Robot -90 (Target Reached)");
  } else if (yawDifference > 0) {
          digitalWrite(MotorLeftBack, HIGH);
          digitalWrite(MotorLeftBack_1, HIGH);
```

```
        digitalWrite(MotorRightBack, LOW);
        digitalWrite(MotorRightBack_1, LOW);
        analogWrite(MotorLeft, 80);
        analogWrite(MotorRight, 85);
        Serial.println("Rotating Right ");
   Serial.println("Rotating Right");
  } else {
    digitalWrite(MotorLeftBack, LOW);
       digitalWrite(MotorLeftBack_1, LOW);
       digitalWrite(MotorRightBack, HIGH);
       digitalWrite(MotorRightBack_1, HIGH);

       analogWrite(MotorLeft, 80);
       analogWrite(MotorRight, 85);
       Serial.println("Rotating Left");
  }
}
void stopMotors()
{
  analogWrite(MotorLeft, 0);
  analogWrite(MotorRight, 0);
  digitalWrite(MotorLeftBack, LOW);
  digitalWrite(MotorLeftBack_1, LOW);
  digitalWrite(MotorRightBack, LOW);
  digitalWrite(MotorRightBack_1, LOW);
  }
void forward() {
  digitalWrite(MotorLeftBack, LOW);
  digitalWrite(MotorLeftBack_1, LOW);
  digitalWrite(MotorRightBack, LOW);
```

```
  digitalWrite(MotorRightBack_1, LOW);
  analogWrite(MotorLeft, 85);
  analogWrite(MotorRight, 85);

}
```

9.2.5 Staircase Climbing

```
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050.h"

MPU6050 mpu;

int16_t ax, ay, az, gx, gy, gz;
const int MotorLeft = 9;
const int MotorRight = 5;

const float pitchClimbThreshold = 17.0;
const float pitchLevelThreshold = 17.0;
const float zAccelThreshold = 2.5;

int baseSpeed = 83;
int climbSpeed = 130;
int torqueBoostSpeed = 115;

bool isClimbing = false;
bool stepDetected = false;
bool stepCompleted = false;
bool torqueBoosted = false;

unsigned long lastStairTime = 0;
```

```
const unsigned long stairTimeout = 3000;
bool sensorsReady = false;
unsigned long sensorStartTime = 0;
float prevPitch = 0;
float prevZAccel = 0;

void setup() {
  Serial.begin(115200);
  Wire.begin();
  mpu.initialize();

  if (!mpu.testConnection()) {
    Serial.println("MPU6050 connection failed!");
    while (1);
  }

  Serial.println("MPU6050 connected.");
  Serial.println("Warming up sensors...");
  sensorStartTime = millis();

  pinMode(MotorLeft, OUTPUT);
  pinMode(MotorRight, OUTPUT);
  analogWrite(MotorLeft, baseSpeed);
  analogWrite(MotorRight, baseSpeed);
}

void loop() {
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

  float pitch = atan2(ay, az) * 180.0 / M_PI;
```

```
float zAccel = (float)az / 16384.0 * 9.81;
float zAccelChange = abs(zAccel - prevZAccel);

if (!sensorsReady && millis() - sensorStartTime > 1500) {
  sensorsReady = true;
  Serial.println("Sensors ready. Starting detection...");
  prevZAccel = zAccel;
  prevPitch = pitch;
  return;
}

if (!sensorsReady) return;

Serial.print("Pitch: ");
Serial.print(pitch);
Serial.print(" | Z-Accel Δ: ");
Serial.println(zAccelChange);

if (!isClimbing && zAccelChange > zAccelThreshold && !torqueBoosted) {
  Serial.println(" Step detected → Torque boost!");
  stepDetected = true;
  lastStairTime = millis();
  torqueBoosted = true;
  driveWithRollControl(torqueBoostSpeed);
}
if (stepDetected && pitch > pitchClimbThreshold && !isClimbing) {
  Serial.println(" Climbing detected...");
  isClimbing = true;
  stepCompleted = false;
  stepDetected = false;
```

```
    lastStairTime = millis();

    driveWithRollControl(climbSpeed);
    torqueBoosted = false;
  }
  if (isClimbing && pitch < pitchLevelThreshold && !stepCompleted) {
    Serial.println(" Step climbed successfully!");
    isClimbing = false;
    stepCompleted = true;
    lastStairTime = millis();

    driveWithRollControl(baseSpeed);
  }

  // Reset for next climb
  if (stepCompleted && pitch > pitchClimbThreshold) {
    stepCompleted = false;
  }

  if (millis() - lastStairTime > stairTimeout && !isClimbing) {
    stopMotors();
    Serial.println(" No stairs for 3s → Robot stopped.");
    while (1);
  }

  prevPitch = pitch;
  prevZAccel = zAccel;

  delay(100);
}
```

```
void driveWithRollControl(int targetSpeed) {
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
  float roll = atan2(-ax, az) * 180.0 / M_PI;

  int startSpeed = 85;
  int rampStep = 5;
  int delayBetweenSteps = 10
  if (abs(roll) < 5.0) {
    Serial.print("speed ");
    Serial.println(targetSpeed);

    for (int s = startSpeed; s <= targetSpeed; s += rampStep) {
      analogWrite(MotorLeft, s);
      analogWrite(MotorRight, s);
      Serial.print("  Speed: ");
      Serial.println(s);
      delay(delayBetweenSteps);
    }

    analogWrite(MotorLeft, targetSpeed);
    analogWrite(MotorRight, targetSpeed);
  } else {
    int correction = roll * 1.0;
    int leftTarget = constrain(targetSpeed - correction, 70, 150);
    int rightTarget = constrain(targetSpeed + correction, 70, 150);

    Serial.print("Roll: ");
    Serial.print(roll);
    Serial.print(" | L: ");
    Serial.print(leftTarget);
```

```
    Serial.print(" | R: ");
    Serial.println(rightTarget);

    for (int s = startSpeed; s <= targetSpeed; s += rampStep) {
      int leftPWM = constrain(s - correction, 70, 150);
      int rightPWM = constrain(s + correction, 70, 150);
      analogWrite(MotorLeft, leftPWM);
      analogWrite(MotorRight, rightPWM);
      delay(delayBetweenSteps);
    }

    analogWrite(MotorLeft, leftTarget);
    analogWrite(MotorRight, rightTarget);
  }
}
void stopMotors() {
  analogWrite(MotorLeft, 0);
  analogWrite(MotorRight, 0);
  Serial.println(" Motors stopped.");
}
```