IMPERIAL

ELEC50015 Electronics Design Project 2

Imperial College London

Department of Electrical & Electronic Engineering 2024

Enginbeers Team: Balance Bot

Authors:
Alexander Charlton, Quentin Duff, Pavalarahul Ganeshsankar, Hector Oga, Dylan Toussaint, Rishabh Varia

Examiner: Dr. Edward Stott

Word Count: 9981

Group GitHub Repository

August 31, 2024

Abstract

This project aims to create an autonomous maze solving robot. Given a square maze with partitions, the robot can map the maze and provide a solution to the user.

The project follows the basic requirements set out in the project brief in addition to its main aim. Namely, it is self-balancing on two wheels, monitors its power consumption and battery level, features an interactive user interface and features a constructed head-unit.

The different subsystems were first characterized through a thorough set of requirements, and assigned to team members on the basis of our relative strengths and weaknesses. Once a basic system was setup, incremental iterations slowly added new features until the requirements were met.

This report will detail the design and implementation of the different units that were integrated and how they fit together. custom voltage and current sensing circuits, along with a motor controller to power the motors and some ultrasonic sensors for spatial awareness interfaced with an esp32 running zephyr to coordinate the core of the robot. commands where sent via a two way Art protocol to a raspberry pi, in charge of higher level functions and control of the rover, including acting as a server for a client designed to allow an end user to interface with the entire system.

Each unit and the robot was first tested individually, and then tested together as the systems became more interlinked. The robot achieves accurate position, velocity and angle control and can navigate and solve a maze, traversing it algorithmically until it reaches its target position in the maze. Additionally, users may switch to manual control.

Contents

1 Introduction 6
1.1 Problem Identification . . . 6
1.2 Purpose . . . 6
1.3 Role Distribution . . . 6

2 Maze Solving 8
2.1 Maze Solving Theory & Possible Algorithms . . . 8
2.1.1 1st Algorithm: Depth-First Search . . . 8
2.1.2 2nd Algorithm: Breadth-First Search . . . 8
2.1.3 4th Algorithm: Flood Fill . . . 8
2.2 Coding Implementation . . . 9
2.2.1 Prioritising Shortest L-Shaped Path . . . 10
2.3 Backtracking . . . 10
2.3.1 Data Structures . . . 10
2.4 Physical Implementation . . . 10

3 Control 13
3.1 Physical Modelling . . . 14
3.2 Translational Equations of Motion . . . 14
3.2.1 Angle Dynamics . . . 14
3.2.2 Velocity Dynamics . . . 15
3.2.3 Position Dynamics . . . 15
3.3 Translational Motion Controller Designs: Loop Shaping . . . 16
3.3.1 Angle Controller . . . 16
3.3.2 Velocity Controller . . . 18
3.3.3 Position Controller . . . 18
3.4 Yaw Equations of Motion . . . 19
3.5 Yaw Controller Design . . . 19
3.5.1 1st Iteration: Modifying the motor controller via acceleration . . . 19
3.5.2 2nd Iteration: Modifying the stepper drivers . . . 19
3.6 Experimental Controller Design . . . 20
3.6.1 1st Iteration: Changing Motor Target Angular Velocity . . . 20
3.6.2 2nd Iteration: Changing Motor Angular Acceleration . . . 20
3.7 Controller Performance: Simulations VS Experiment . . . 20
3.7.1 Angle Controller . . . 20
3.7.2 Velocity Controller . . . 22
3.7.3 Position Controller . . . 24
3.7.4 Yaw Controller . . . 26
3.8 Conclusion . . . 27

4 Sensing and Monitoring 28
4.1 Power Monitoring . . . 28
4.1.1 Hardware Interface . . . 29
4.1.2 Software Interface . . . 31
4.1.3 Conclusion . . . 33
4.2 Sensing . . . 34
4.2.1 Sensor Choice . . . 34
4.2.2 Physical Implementation of Sensing . . . 34
4.2.3 Maze Error Handling via Sensing . . . 35
4.2.4 Inertial Measurement . . . 35

5 Software 38
5.1 Overview . . . 38

5.1.1 Overall implementation . . . 39
5.2 Firmware . . . 41
5.2.1 Robot Movement . . . 42
5.2.2 Stepper Driver . . . 42
5.2.3 Ultrasonic Sensor Implementation . . . 42
5.2.4 Non-Volatile storage . . . 42
5.2.5 Communication with the Raspberry Pi (Server) . . . 42
5.2.6 Evaluation . . . 43
5.3 Server . . . 45
5.3.1 Overview . . . 45
5.3.2 Implementation . . . 46
5.4 Client . . . 47
5.4.1 Overview . . . 47
5.4.2 Implementation . . . 47
5.4.3 Architecture overview . . . 48
5.4.4 UI . . . 48

6 Conclusion **50**

A Appendix **51**
A.1 Figures . . . 51
A.2 Bill of Materials . . . 56
A.3 FMEA Analysis . . . 57
A.4 Risk Assessment . . . 59

List of Figures

1.3.1 Dependency Diagram . . . 7
2.1.1 DFS: Intersection 1 noted . . . 8
2.1.2 DFS: Dead end taken, steps traced back . . . 8
2.1.3 DFS: Exploring other unseen branches . . . 8
2.1.4 BFS: Intersection 1a Noted . . . 9
2.1.5 BFS: Going to Intersection 1b, Noted . . . 9
2.1.6 BFS: Exploring Unexplored Path Past Intersection 1a. . . . 9
2.1.7 comparison of algorithms [1] . . . 9
2.2.1 Weight to neighbor code . . . 10
2.4.1 Maze 1 . . . 11
2.4.2 Maze 2 . . . 11
2.4.3 Maze 3 . . . 11
2.4.4 Maze 4 . . . 11
2.4.5 Maze 5 . . . 11
2.4.6 T-Junction Module . . . 11
2.4.7 Corner Piece . . . 11
2.4.8 Side Stand . . . 11
2.4.9 Modularising the Mazes . . . 11
2.4.10 Modular Assembly of Maze 1 . . . 11
3.2.1 Model Showing Forces Applied to the System . . . 16
3.2.2 Full Control Scheme . . . 16
3.3.1 Basic Block Diagram of the Angular Control . . . 18
3.7.1 Angle Test Angle Loop Time Response . . . 21
3.7.2 Angle Test Simulated Angle Loop Time Response . . . 21
3.7.3 Velocity Test Velocity Time Response . . . 22
3.7.4 Velocity Test Angle Time Response . . . 22
3.7.5 Velocity Test Velocity Simulated Results . . . 23
3.7.6 Position Test Position Time Response . . . 24
3.7.7 Position Test Velocity Time Response . . . 24
3.7.8 Position Test Angle Time Response . . . 24
3.7.9 Position Test Simulated Position Response . . . 25
3.7.10 Position Test Simulated Velocity Response . . . 25
3.7.11 Position Test Simulated Angle Response . . . 25
3.7.12 Yaw Test Yaw Step Response . . . 26
3.7.13 Yaw Test Position Response to a Yaw Step . . . 26
4.1.1 Differential Amplifier with Voltage Reference [2] . . . 29
4.1.2 Hardware Interface Schematic . . . 30
4.1.3 Hardware Interface Construction . . . 30
4.1.4 ADC Characterization (PlatformIO) . . . 31
4.1.5 ADC Characterization (ZephyrOS) . . . 31
4.1.6 Calibration Curve for ADC . . . 32
4.1.7 Battery Discharge Curve [3] . . . 32
4.2.1 Ultrasound Sensor Clip . . . 34
4.2.2 Bot Head Unit Cap . . . 34
4.2.3 Parts Mounted onto the Bot . . . 35
4.2.4 Force Diagram . . . 36
4.2.5 Filtered and Unfiltered Data . . . 36
5.1.1 Overall architecture . . . 39
5.1.2 software architecture . . . 40
5.4.1 UI Interface in maze mode . . . 48
5.4.2 UI interface in manual mode . . . 49
A.1 Tilt Natural System Response . . . 51

A.2 Tilt Open Loop System Response . . . 51
A.3 Tilt Closed Loop System Response . . . 52
A.4 Velocity Natural System Response . . . 52
A.5 Velocity Open Loop System Response . . . 52
A.6 Velocity Closed Loop System Response . . . 53
A.7 Position Natural System Response . . . 53
A.8 Position Open Loop System Response . . . 53
A.9 Position Closed Loop System Response . . . 54
A.10 HC-SR04 Ultrasound Sensor Board [4] . . . 54
A.11 Flood Fill Algorithm Iterative Solving Demonstration . . . 55

List of Tables

1.3.1 Role Distribution According to Strengths . . . 6
3.1.1 Balance Bot Variables . . . 14
4.1.1 Differential Amplifier Gain Table . . . 29
4.1.2 Discharge to Percentage Conversion . . . 33
5.2.1 Protocol Evaluation . . . 43
A.2.1 Complete Bill of Materials . . . 56
A.3.1 FMEA Process Identification and Classification . . . 57
A.3.2 FMEA Recommend Actions and Action Taken . . . 58

List of Listings

4.1.1 ADC Correction Polynomial . . . 32
4.1.2 ADC Readings, Full Interface . . . 33
A.1.1 Additional Parameters and Functions Needed to Implement Yaw . . . 51

1 Introduction

1.1 Problem Identification

The robot is a two-wheeled inverted rigid body pendulum that can move in a three dimensional plane. Tuning the controllers will be done through loop shaping and simulations, implying the robot will need to be modelled. Considerations will also be made when implementing the controllers in practice into hardware, as results may shift from the simulations. We must also find a way to communicate between the ESP32 module, the Raspberry Pi, and the client's computer, allowing for display of useful information to the user. This information includes, but is not limited to, battery percentage, power usage, PID coefficient tuning and updating, and mode of operation.

1.2 Purpose

The robot will have multiple modes of operation: the user can choose between maze and manual mode. In manual mode, the robot behaves like a regular remote controlled vehicle, guided by inputs from the user. The user will be able to modify the robot's controllers on the fly, all while giving it commands. Maze mode is an autonomous mode where the robot is expected to solve a maze it is placed in and has not previously seen. Upon startup, it will be told where it is in the maze and where it is expected to go. With no input from the user, the robot will figure out exactly how to solve the maze, all while updating its built-in map on the UI for the user to see.

1.3 Role Distribution

Roles were distributed according to each team member's respective strengths in a field, leading us to having a defined set of roles that was kept throughout the robot's conception. Cooperation and peer review was encouraged between roles.

Dylan	**Power Management Component**, Hardware Implementation and Testing, ADC, FMEA
Hector	**Control Component**, Control Model Characterization, PID Implementation and Tuning, Maze Solving Theory
Rishabh	**Control Component**, PID Testing and Tuning, CAD, Maze Designing and Building
Rahul	**PI Comms and MPU Unit**, MPU6050 Library Implementation, Maze Algorithm Implementation
Alexander	**PI Comms**, User Interface, Client, UART, ADC Firmware Implementation
Quentin	**ESP32 Firmware and Server**, Stepper Motor Driver, Client, PC-PI-ESP32 Communication Bridge, PID Implementation

Table 1.3.1: Role Distribution According to Strengths

From the role allocation and the dependency tree. In this project the roles have been assigned to optimise the strengths of each of the team's members. Considering that the group consists of 3 EEE and EIE people. The EIE people have been tasked with implementing the zephyr OS firmware and server while those in EEE have been assigned the battery management, control and CAD. This division allowed each member of the EEE and EIE partition to proof check over each others work. Such that section was solely worked on by one person, and so that at least 2 people knew the state of a given module at a time. This provided redundancy in case a group member fell ill as well as helping keeping track of the progress of the project. We also made extensive use of GitHub, which each commit having a descriptive description such that the programming team could keep track of what was being worked on and what still needed doing.

A risk assessment was performed prior to any work. This can be found in Appendix A.4.

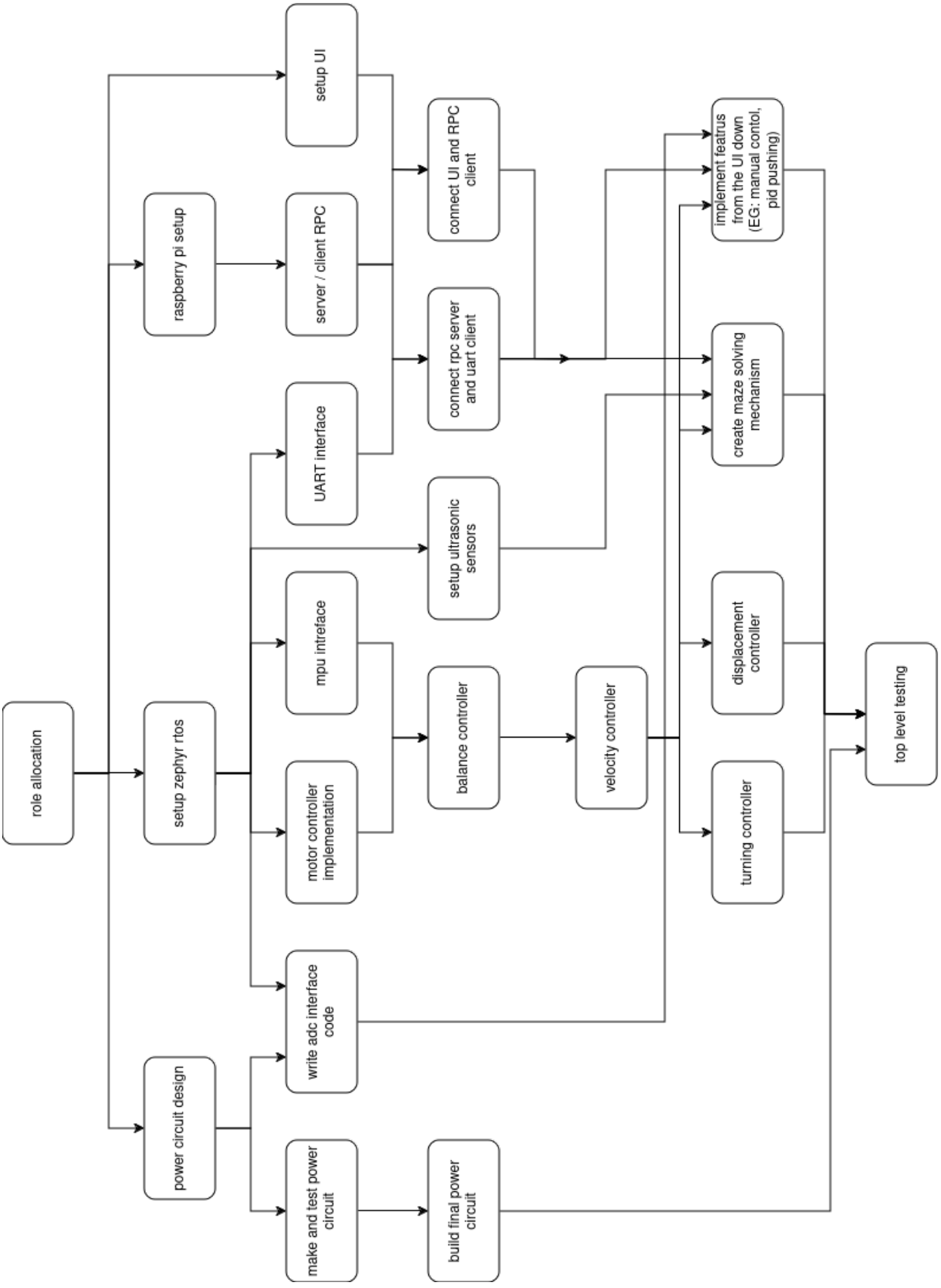


Figure 1.3.1: Dependency Diagram

2 Maze Solving

2.1 Maze Solving Theory & Possible Algorithms

Solving a maze at first glance may sound simple, but unlike us, the robot does not have any top-view of the maze, and can only rely on our choice of ultrasound sensors to map the maze. In the Micromouse Competition organized by IEEE [5], a small electronic mouse is required to solve a 16x16 maze. We wish for our robot to accomplish maze solving in similar way: with a minimal number of runs, and in the most efficient way possible. For our purposes, we have chosen a smaller 5x5 maze, small enough to build but large enough to demonstrate our robot's capabilities. There are many algorithms capable of solving a maze while traversing it, but each offers its own advantages and drawbacks [6].

If the maze has the goal on a corner, and presents no standing walls, it is possible to solve it with no complicated math and simply following a wall until the end is reached. For this reason, our maze will have standing walls and a variable goal.

2.1.1 1st Algorithm: Depth-First Search

This algorithm consists of running through the maze noting every fork in the road. Whenever the robot can no longer advance, it goes back to the last recorded fork in the road and explores the next option. If a fork led nowhere, the robot must then backtrack to the last intersection it remembers having unexplored options. This strategy is not guaranteed to find the shortest route, as the mouse only turns back when it needs to, leading to possible shortcuts missed.

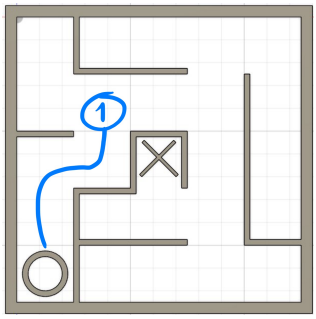


Figure 2.1.1: DFS: Intersection 1 noted

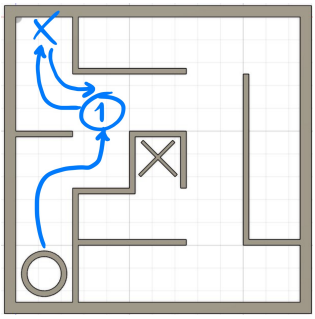


Figure 2.1.2: DFS: Dead end taken, steps traced back

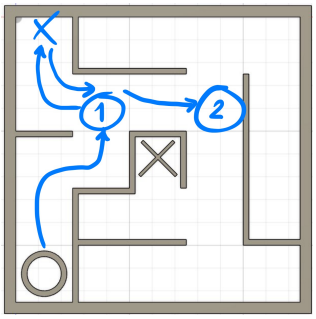


Figure 2.1.3: DFS: Exploring other unseen branches

2.1.2 2nd Algorithm: Breadth-First Search

This is the sibling algorithm to Depth First-Search. It is guaranteed to find the shortest path, but is significantly longer. This strategy makes the mouse travel to an intersection, at which, instead of exploring it, checks the path it skipped by going to this intersection. When the mouse then reaches the next intersection, it goes back to the first intersection it found to explore the path it skipped. The process repeats until the robot finds the goal, where then the robot is capable of smoothing out the actual shortest path it must take to get to the goal. Unfortunately, the extremely long execution time of this algorithm makes it undesirable, as it is rerunning paths multiple times (dozens, for larger mazes). Searching the whole maze often takes less time!

2.1.3 4th Algorithm: Flood Fill

The most popular maze-solving algorithm used in the Micromouse competition is known as Flood Fill [7]. Using this algorithm, the robot's plan is to make optimistic journeys throughout the maze; so optimistic in fact, that the robot's first map of the maze contains no walls at all. The robot simply draws the shortest path to the goal and tries to go. The robot will inevitably encounter walls and obstacles to its optimistic

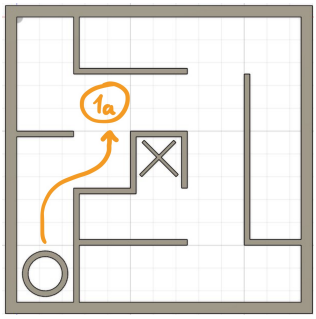


Figure 2.1.4: BFS: Intersection 1a Noted

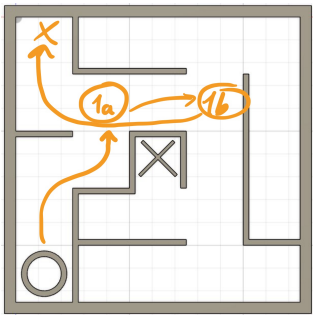


Figure 2.1.5: BFS: Going to Intersection 1b, Noted

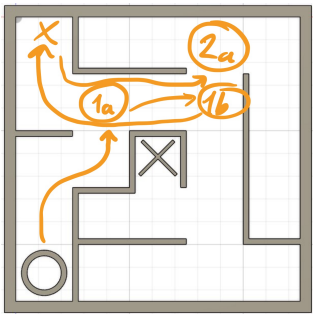


Figure 2.1.6: BFS: Exploring Unexplored Path Past Intersection 1a.

journey: it will, at that point, rethink its journey, and updating its optimistic path, accounting for the new obstacles. It is an iterative process of running and updating its stored maze map, always bee-lining for the goal. Although it is guaranteed to find a solution in a single run, it is not always the fastest.

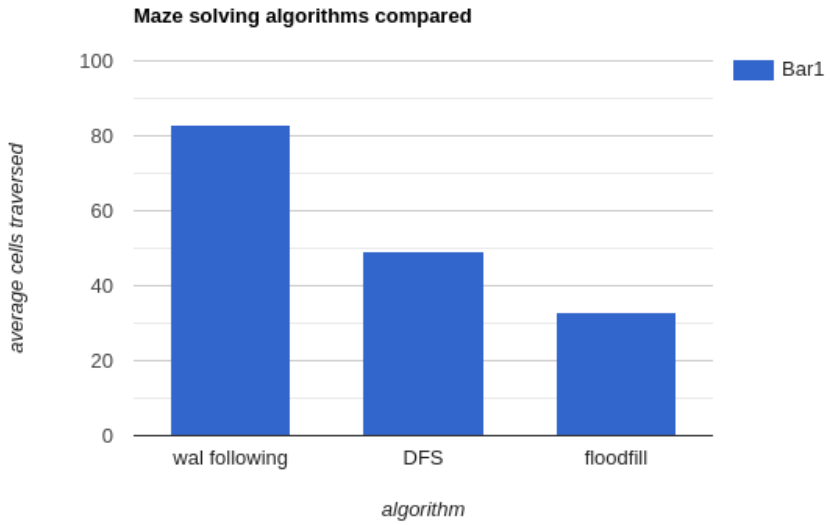


Figure 2.1.7: comparison of algorithms [1]

A variation of this algorithm will be used in order to implement backtracking. Whilst it may not be as computationally efficient as flood fill, it is poised to take advantage of information gained from multiple blocks away whilst flood fill will blindly follow a preconceived path until it reaches the blockage.

The iterative process for this algorithm has been shown in Figure A.11

2.2 Coding Implementation

In order to implement the flood fill algorithm, there are two broad cases which have to be thought about.

- Being able to follow the shortest L shaped path
- Backtracking in the event of a blockage

The A* algorithm that can have a new iteration run every can be used underneath to cover these cases. The heuristic part and weightings of the A* algorithm can used to prioritize the shortest L path and having a new iteration every time new information about the maze is discovered will allow backtracking.

2.2.1 Prioritising Shortest L-Shaped Path

The fact that the shortest L-shaped path only requires one turn makes it quicker. This means that the weights between nodes that are fed into the A* underlying algorithm need to be varied by the amount of turning required such that turning multiple times is penalised.

```
fn dn(neighbor_direction: maze::Direction, robot_facing: maze::Direction)
-> cost float{
    //Sub operator is implemented for maze::Direction and it
    gives the number of turn
    return (neighbor_direction - robot_facing) *
    TURN_COST + ONE_STEP_FORWARD;
}
```

Figure 2.2.1: Weight to neighbor code

This is achieved by using a vector map that stores the hypothetical orientation of the robot at every node and compares it to the direction of travel to the neighbor as shown in Snippet:2.2.1.

2.3 Backtracking

Having an A* iteration run once new data about the maze is discovered allows the algorithm to backtrack.

2.3.1 Data Structures

The data-structure that holds the information about the maze will have to hold the following information:

- If each block has been traversed or not.
- If walls exist between nodes.

In order to avoid duplication of data storage which would be in efficient, each edge between neighbors is only stored once. Each block holds the information regarding to 2 out of its 4 surrounding walls.

2.4 Physical Implementation

The maze is mapped as a grid on a 5×5 matrix with each square in the matrix which are all $0.25m \times 0.25m$ each. From this we can design the following mazes. Here the start position of the robot is marked with an $'O'$ and the finish position with an $'\times'$. Here are 5 possible maze designs that the robot could solve. Each side the robot has to access the centre of the maze with is different between each of these designs, thus this gives some variety on the mazes the bot solves and provides more rigorous testing.

From this the maze therefore needs to be modular, so that the inside of the maze borders can be for any amount of variable setups. The designs given are just mere examples and using the following CAD blocks can be used to achieve this modularisation. There are 3 blocks in total, one being a corner piece, a side stand and t-junction module.

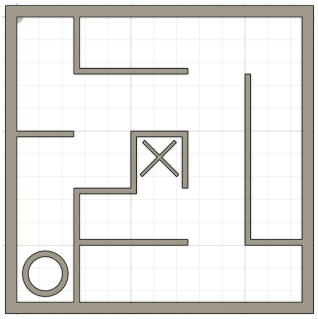


Figure 2.4.1: Maze 1

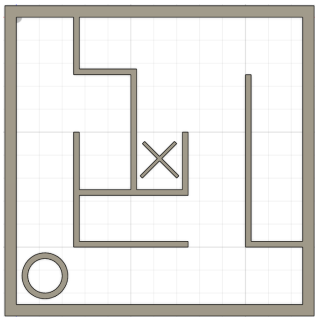


Figure 2.4.2: Maze 2

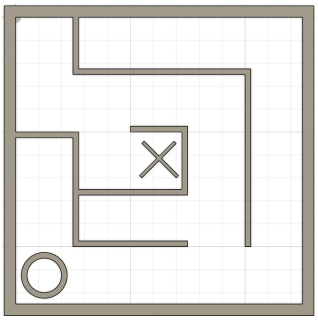


Figure 2.4.3: Maze 3

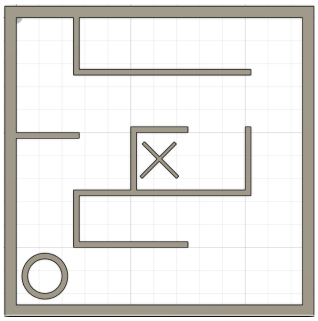


Figure 2.4.4: Maze 4

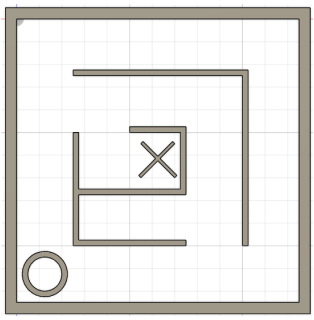


Figure 2.4.5: Maze 5

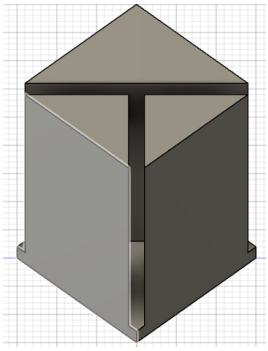


Figure 2.4.6: T-Junction Module

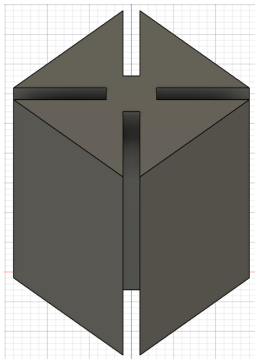


Figure 2.4.7: Corner Piece

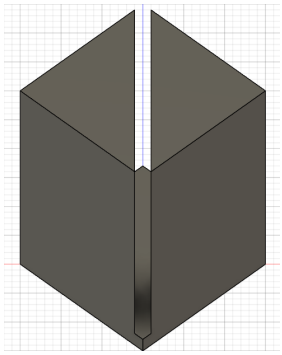


Figure 2.4.8: Side Stand

Figure 2.4.9: Modularising the Mazes

Figure 2.4.10: Modular Assembly of Maze 1

These pieces will hold together modular cardboard pieces which are 25cm in width and 25-30cm in height. The t-junction allows a cardboard piece to be attached to the outside cardboard frame. The corner piece allows the cardboard pieces to attached at 90^o to each other. Finally the side stand then finally allows for a single free standing piece to be placed. As the cardboard pieces are of variable height, the corner pieces are open ended on either end. The use of this can be seen in the figure below. These pieces are only 3cm tall and with a 20% infill 10 of these pieces can be printed in 10 hours meaning that we can have an abundance of these and have no shortage of maze configurations. Furthermore this shows that this method of maze modularisation is effective and simple and can provide a breadth of test mazes. To further improve the modularisation of this maze the quality of the cardboard used can be improved.

3 Control

The balancing of the robot is a critical part of the successful completion of our main objectives. The robot must be able to control its tilt angle, velocity and maintain a known position in order to accurately map a maze.

Additionally, the robot must also change its yaw without affecting its balance or positioning control.

The following section presents our design and implementation process for the main controller. Control is achieved by a set of cascaded PID controllers which were tuned using Matlab modelling. Testing was completed in order to ensure stability and precise positioning.

Control Unit Requirements	
Technical Requirements	
Angle Control	The robot must be able to reach and maintain a tilt angle between 0 and $\pm 5^o$, sufficiently fast for velocity and position control loops to operate effectively. The robot must be able to remain upright independently of any offset during the calibration of the gyroscope unit.
Velocity Control	The robot must be able to reach and maintain a velocity between 0 and 0.5 m/s along its longitudinal axis, fast enough for the position controller to operate effectively.
Position Control	The robot must be able to reach and maintain a position setpoint along its longitudinal axis. The position controller must have little absolute error as accurate positioning is crucial to our main objective.
Yaw Control	The robot must be able to rotate along its yaw axis while stationary. This must not affect the tilt angle of the robot.

3.1 Physical Modelling

We have chosen to model the balance bot as a two-wheeled inverted rigid body pendulum.

Variable	Unit	Description
M	[kg]	Mass of the wheels
m_{bod}	[kg]	Mass of the body
m_{batt}	[kg]	Mass of the batteries
m	[kg]	Mass of the robot ($m = m_{bod} + m_{batt}$)
g	[m.s^{-2}]	Gravitational Constant
l	[m]	Distance from center of wheel to bot's center of mass
R	[m]	Wheel radius
T_b	[s]	Body natural period of oscillation
I_b	[kg.m^{-2}]	Body moment of inertia
I_w	[kg.m^{-2}]	Wheel moment of inertia
I_z	[kg.m^{-2}]	Yaw moment of inertia
x	[m]	Horizontal displacement
θ	[rad]	Pitch angle from the vertical
ψ	[rad]	Yaw angle about the vertical axis

Table 3.1.1: Balance Bot Variables

3.2 Translational Equations of Motion

3.2.1 Angle Dynamics

To determine analytically the equations of motions, the Lagrangian method of balancing energies was used. To determine kinetic energy, we consider the rotational energy of the body and the wheels, as well as the translational energy of the body. The potential energy is comprised of only the body's as we consider the wheels to be at height 0.

$$L = T - V = T_{body,trans} + T_{body,rot} + T_{wheels,rot} - V_{body} \tag{1}$$

Now to determine each of these values individually. Considering the body's center of mass, its position in the xy-plane is $(x_{bod}, y_{bod}) = (x + l\sin(\theta), l\cos(\theta))$, its velocity is therefore $(\dot{x}_{bod}, \dot{y}_{bod}) = (\dot{x} + l\dot{\theta}\cos(\theta), -l\dot{\theta}\sin(\theta))$. Hence $v_{bod} = \sqrt{\dot{x}_{bod}^2 + \dot{y}_{bod}^2}$, and therefore we can establish $T_{body,trans} = \frac{1}{2}mv_{bod}^2 = \frac{1}{2}m((\dot{x} + l\dot{\theta}\cos(\theta))^2 + (l\dot{\theta}\sin(\theta))^2$. The rotational kinetic energy of the body depends on its moment of inertia: $T_{body,rot} = \frac{1}{2}I_b\dot{\theta}^2$. In summary:

$$\begin{cases} T_{wheels,rot} = \dfrac{1}{2}M\dot{x}^2 \\ T_{body,trans} = \dfrac{1}{2}m(\dot{x}^2 + \dot{\theta}^2 l^2 + 2l\dot{x}\dot{\theta}\cos(\theta)) \\ T_{body,rot} = \dfrac{1}{2}I_b\dot{\theta}^2 \\ V_{body} = mgl\cos(\theta) \end{cases} \tag{2}$$

Using the Euler-Lagrange equations:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial q_i}\right) - \frac{\partial L}{\partial q_i} = Q_i \quad i = \{1, 2\}, q_i = \{x, \theta\}, Q_i \text{ are the generalized forces} \tag{3}$$

Including an input torque τ to the wheels, rearranging gives the system of equations of motion:

$$\begin{cases} (M+m)\ddot{x} + ml\cos(\theta)\ddot{\theta} - ml\sin(\theta)\dot{\theta}^2 = \dfrac{\tau}{R} \\ mlcos(\theta)\ddot{x} + (ml^2 + I_b)\ddot{\theta} + mgl\sin(\theta) \end{cases} \tag{4}$$

To simplify these equations, small angle approximations can be used for $\theta \ll 1$: $\cos\theta \approx 1$ and $\sin\theta \approx \theta$. Hence the revised system of equations is:

$$\begin{cases} (M+m)\ddot{x} + ml\ddot{\theta} - ml\theta\dot{\theta}^2 = \dfrac{\tau}{R} \\ ml\ddot{x} + (ml^2 + I_b)\ddot{\theta} + mgl\theta = 0 \end{cases} \tag{5}$$

Define the state variable $\mathbf{x} = \left(x, \dot{x}, \theta, \dot{\theta}\right)^T = (x_1, x_2, x_3, x_4)^T \implies \dot{\mathbf{x}} = \left(\dot{x}, \ddot{x}, \dot{\theta}, \ddot{\theta}\right)^T = \left(x_2, \ddot{x}, x_4, \ddot{\theta}\right)^T$. Hence, linearising around the equilibrium point $\mathbf{x} = 0, \tau = 0$ gives the state space system below, using $\mathbf{y}$ as our output.

$$\begin{cases} \dot{\mathbf{x}} = A\mathbf{x} + Bu \\ \mathbf{y} = C\mathbf{x} \end{cases} \tag{6}$$

$$\text{With: } A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & k_2 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 0 \\ k_1 \\ 0 \\ -k_3 \end{pmatrix} \quad C = I_4$$

$$\text{And: } \begin{cases} k_1 = \dfrac{1}{R\left(M + \frac{I_w}{R^2} + m + \frac{m^2 l^2}{ml^2 + I_b}\right)} \\ k_2 = \dfrac{mgl}{ml^2 + I_b} \\ k_3 = \dfrac{mlk_1}{ml^2 + I_b} \end{cases} \tag{7}$$

3.2.2 Velocity Dynamics

Figure 3.2.1 shows the forces applied to the system, considering it from the center of mass. Considering a fixed position wheel for now, the weight of the bot $\mathbf{P}$ creates a torque upon the wheel axle, thus creating a rotational force normal to the bot's trajectory $\mathbf{F}_\tau$. This force has a translational horizontal component $\mathbf{F}_t$, which can be extracted from the diagram to be as per Equation (8) below.

$$F_t = P\tan\theta \tag{8}$$

Hence, using Newton's second law and small angle approximations, we can establish a relationship between ground speed of the robot and the pitch angle:

$$ma = mg\tan\theta \implies a \approx g\theta \implies v \approx \int g\theta \tag{9}$$

3.2.3 Position Dynamics

Ground position and ground velocity are simply linked by integration:

$$p = \int v dt$$

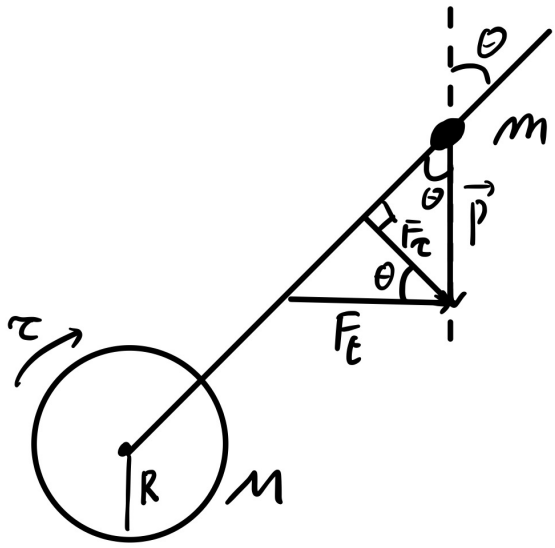


Figure 3.2.1: Model Showing Forces Applied to the System

From the dynamics, we can obtain the full block diagram for robot's movement, as shown in Figure 3.2.2, relating position to velocity, and velocity to tilt angle.

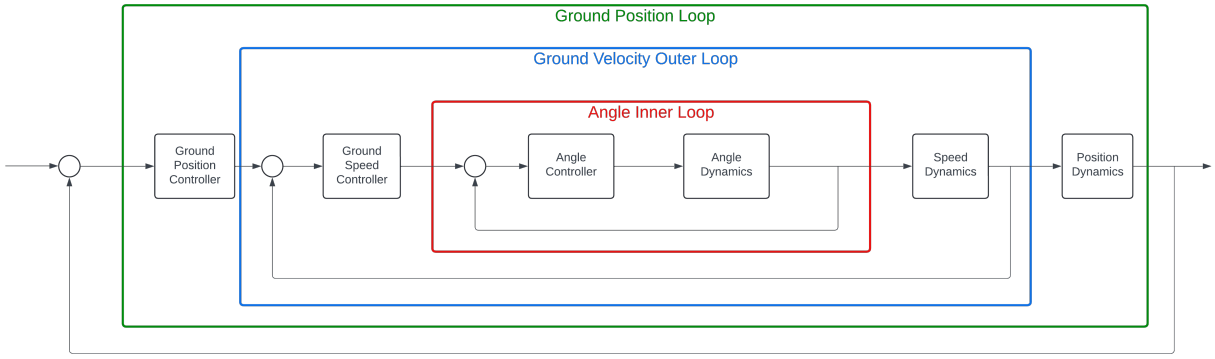


Figure 3.2.2: Full Control Scheme

3.3 Translational Motion Controller Designs: Loop Shaping

3.3.1 Angle Controller

As expected, the transfer function between input torque to the wheels and the angle from the vertical is unstable: any disturbance from the equilibrium point results in the robot toppling. It is of the form Equation (10), where one of the poles is positive.

$$G_\theta(s) = \frac{\mu}{s^2 - \omega_n^2} \qquad \{\mu, \omega_n\} \in \Re \tag{10}$$

The values of μ and ω_n were obtained experimentally. μ was calculate from k_1, k_2, and k_3 previously calculated, whereas ω_n is the natural angular frequency of the robot, modelled as an inverted pendulum. A video was taken of the robot swinging from side to side, and timestamps revealed that its natural oscillation period was $T_b = 0.836869231s$, using multiple swing test iterations, and averaging all results. This was

confirmed in the simulations as the value for ω_n^2 obtained in `MATLAB` was the same obtained using the free oscillation method. This confirms the validity of our model, and gives confidence its results will be accurate.

$$\omega_n = \frac{2\pi}{T_b} = 7.508 rad/s = \sqrt{\omega_n^2(\text{simulated})}$$

To stabilize this system, we have chosen a PID structure as they offer a practical, versatile, and cost-effective solution for many control applications, providing robust performance with a simple design and implementation, easily implemented in the C language, which is the one used for this project. Their balance of simplicity and effectiveness makes them our preferred choice. The basic structure for a PID controller is:

$$C(s) = K_p + \frac{K_i}{s} + \frac{K_d}{1+\tau_f s} \tag{11}$$

Hence, the open loop system is:

$$L_\theta(s) = C_\theta(s) G_\theta(s) = \left(K_p + \frac{K_i}{s} + \frac{K_d}{1+\tau_f s} \right) \frac{\mu}{s^2 - \omega_n^2} \tag{12}$$

However, once we factorise out $\frac{K_i}{s}$ and $\frac{1}{1+\tau_f s}$, we are left with a system that would result in an unstable closed loop system, due to the integrator.

$$L_\theta(s) = \frac{\mu K_i}{s(1+\tau_f s)} \left(1 + \frac{K_p + K_i \tau_f}{1+\tau_f s} s + \frac{K_d + K_p \tau_f}{1+\tau_f s} s^2 \right) \frac{1}{s^2 - \omega_n^2} \tag{13}$$

Therefore, to ensure stability, we must set $K_i = 0$, leaving us with a PD controller. From this, we can rewrite the loop equation as such:

$$\begin{aligned} C_\theta(s) = K_p + \frac{K_d}{1+\tau_f s} \implies L_\theta(s) &= \mu \left(K_p + \frac{K_d}{1+\tau_f s} \right) \frac{1}{s^2 - \omega_n^2} \\ \implies L_\theta(s) &= \frac{\mu K_p}{1+\tau_f s} \left(1 + \frac{K_d + K_p \tau_f}{K_p} s + \frac{K_d}{K_p} s^2 \right) \frac{1}{s^2 - \omega_n^2} \end{aligned} \tag{14}$$

It is more useful to factorize this expression, setting two zeros, a pole, and a variable static gain, all of which can be tuned. From these values, the PD coefficients can be extracted:

$$L_\theta(s) = \frac{\bar{\mu}}{1+\tau_f s} (1+\tau_1 s)(1+\tau_2 s) \frac{1}{s^2 - \omega_n^2} \tag{15}$$

$$\text{Where: } \begin{cases} \bar{\mu} = \mu K_p \\ \tau_1 + \tau_2 = \dfrac{K_d + K_p \tau_f}{K_p} \\ \tau_1 \tau_2 = \dfrac{K_d}{K_p} \end{cases} \implies \begin{cases} K_p = \dfrac{\bar{\mu}}{\mu} \\ K_d = (\tau_1 + \tau_2 - \tau_f) K_p \end{cases}$$

We wish to have the desired crossover frequency ω_c^dbetween these two zeros, so we can set a high and a low frequency zero by setting $\tau_1 = \frac{10}{\omega_c^d}$ and $\tau_2 = \frac{1}{2\omega_c^d}$. The static gain $\bar{\mu} = \frac{4\omega_c^d}{\tau_1}$ yields the best results. Using these three values, we are able to obtain the best simulation results. We set ω_c^d to 30rad/s, as this is a fast enough controller to catch the robot in case it falls.

It is more useful to work in angular acceleration applied to the wheel, as that is a parameter easy to control in the code, given we are using stepper motors with a fixed step angle. Using microstepping, otherwise known as

dividing each stepper motor angle into smaller steps, we are able to control the angular acceleration applied to the wheels with more precision. Hence, we can draw the block diagram below, where α is the angular acceleration applied to the wheels:

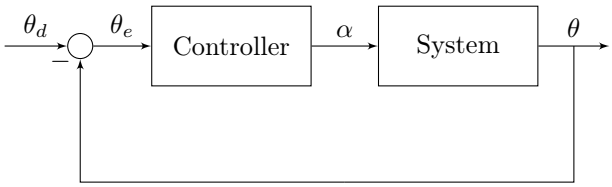


Figure 3.3.1: Basic Block Diagram of the Angular Control

To convert from torque to acceleration, it is a simple matter of scaling by the wheels' moment of inertia I_w. After this scaling, we can draw the Bode plots of the system response, open loop response, and closed loop response (Figure A.1, Figure A.2, and Figure A.3 respectively).

3.3.2 Velocity Controller

From Equation (8), we have

With some abuse of notation, we can write the transfer function relating the velocity to the pitch angle:

$$G_v(s) = \frac{g}{s} \implies L_v(s) = C_v(s) \times \text{Angle Inner Loop} \times G_v(s) \approx C_v(s)G_v(s), \forall \omega < \omega_c^d(\text{Angle}) \tag{16}$$

It is important to note that this is an approximation. In reality, the ground velocity depends both on the pitch angle and the torque applied to the wheels, according to Equation (5). As such, a more complete controller would include a feed-forward branch into the velocity dynamics, including the input torque. However, for simplicity and ease of implementation, we consider ground velocity to be independent of the input torque. Hence, we use a full PID controller, instead of the expected P-only controller the simplified dynamics would suggest.

Therefore, setting a desired crossover frequency ω_c^d to be 10 rad/s, we can extract the K_p, K_i and K_d coefficients using Equation (17) below by placing two poles and two zeros.

$$C_v(s) = \left(K_p + \frac{K_i}{s} + \frac{K_d s}{1+\tau_f s}\right) = \frac{K_i}{1+\tau_f s}\left(1 + \frac{K_p + K_i\tau_f}{K_i}s + \frac{K_d + K_p\tau_f}{K_i}s^2\right) \tag{17}$$

Without loss of generality, setting $\tau_1 = \frac{1}{2\omega_c^d}$ and $\tau_2 = \frac{10}{3\omega_c^d}$ (one high frequency and another low frequency), we can obtain the desired performance in practice.

Using the full model, we obtain the Bode plots in Figure A.4, Figure A.5, and Figure A.6 below showing the system response, the open loop response, and the closed loop response respectively.

3.3.3 Position Controller

Relating ground position to ground velocity is trivial. Position is the integral of velocity:

$$G_p(s) = \frac{1}{s} \implies L_p(s) = C_p(s) \times \text{Velocity Loop} \times G_p(s) \approx C_p(s)G_p(s), \forall \omega < \omega_c^d(\text{Velocity}) \tag{18}$$

As such, a proportional-only controller is enough. Setting $K_p = \omega_c^d$ is enough to set the desired crossover frequency. Using the full model, we obtain the Bode plots in Figure A.7, Figure A.8, and Figure A.9, showing the system, open-loop, and closed-loop responses respectively.

3.4 Yaw Equations of Motion

The wheels are driven by individual torques applied to the left and right motors, τ_L and τ_R respectively. The net torque causing yaw rotation is therefore $\tau_{yaw} = \tau_R - \tau_L$. The yaw motion can therefore be described by Equation (19). If we set $\tau_L = -\tau_R$, making the input torques equal and opposite, we have a relationship between the input torque to either wheel and yaw angle. As we are operating stepper motors, it is easier to work in terms of angular velocity ω, or angular acceleration α to a wheel, as will become apparent later. This heavily simplifies the equation, and makes the loop implementation more straightforward.

$$I_z\ddot{\psi} = \tau_{yaw} \implies \ddot{\psi} = \frac{2\tau_R}{I_z} \implies \ddot{\psi} = 2\alpha \implies \dot{\psi} = 2\omega \tag{19}$$

With some abuse of notation, we can obtain the transfer function relating yaw angle to input angular velocity or acceleration to the wheel:

$$G_{\omega,\psi}(s) = \frac{2}{s} \qquad G_{\alpha,\psi} = \frac{2}{s^2} \tag{20}$$

3.5 Yaw Controller Design

In this robot, turning in place is essential to maze solving. Indeed, in order to discretise the actions the robot can take, turning on the spot would allow us to not make additional positional considerations. Additionally, small errors will build up enormously over time, hence we need this controller to be as accurate as possible.

3.5.1 1st Iteration: Modifying the motor controller via acceleration

In theory, adding the obtained required acceleration for yaw turning via a PD controller (no integral gain due to the dynamics: see Section 3.3.1) to the ground speed obtained via the acceleration from the tilt controller, should work, by linearity. In practice however, this was not the observed result and we obtained significant oscillations and jitters from the robot.

This two methods was deemed unsuccessful due to the conflict with the tilt controller: with the addition of the Raspberry Pi and the power monitoring module to the head unit, the weight distribution and location of the center of mass significantly changed, which in turn shifted the tilt equilibrium angle to a non-zero value. As such, the dynamics imply that any change in yaw at a non-zero tilt angle would yield a change in said tilt angle. Therefore, the tilt controller was constantly trying to fight the changes installed by the yaw controller. To solve this issue, the addition of a yaw controller was rethought and instead the driver was modified

3.5.2 2nd Iteration: Modifying the stepper drivers

In practice, although the stepper motor acceleration was thought to give the same velocity, this was not the case, hence, the two motors were linked so that every acceleration change relating to translational movement. New parameters and functions were added onto `stepper_driver.c` allowing us to implement yaw, listed in Listing A.1.1. By setting the desired step turn in `motor_controller.c`, the driver determines whether or not an input velocity is needed via an error calculation, from which a K_p-only controller updates the `yaw_target_speed` directly in the driver, using a separate function to the speed update given by the tilt controller. The `stepper_run` function then updates the stepper drivers sequentially, adding on each of the updated speeds together (`yaw_target_speed` and `target_speed`). This method gave satisfactory results, as will later be explored in Section 3.7.4

3.6 Experimental Controller Design

From the equations of motion, the outer loops for velocity and displacement/position can be formed. This will be configured so that the position loop, with a given position set point, gives a velocity set point, and given the current velocity from the stepper motors, results in a velocity error which then determines the angle set point for the balance bot. This is so that the robot slightly tilts forward for movement forwards and same for backwards. This is all easily achieved as the acceleration, velocity can be directly set to the stepper motors and because the velocity and position can be directly read from the stepper motors. The frequency of these loops will run slower given which is loop is outermost.

Figure 3.2.2 shows the control dynamics between angle, acceleration, velocity and position.

In order to communicate with the motors, the `stepper_driver.c` file was written, enabling an instantiation of a motor struct. Microstepping is the chosen form of speed-setting: this means that we cannot tell the steppers to have a desired angular acceleration, and instead there is an unknown proportionality constant modifying the input angular acceleration into observed angular acceleration. For this reason, the K_p, K_i and K_d coefficients found in simulation are not entirely usable. However this does mean they're irrelevant: by keeping the same ratio between the coefficients as in the simulation, we are able to get the best performance in accordance with our model.

More will be explored in Section 5

3.6.1 1st Iteration: Changing Motor Target Angular Velocity

Changing the motor's target angular velocity is achieved by gradually changing the velocity every δt until the input target velocity is achieved.

Although this controller achieves static balance, it still effectively implements several step changes in velocity over arbitrarily small δt values. Though the oscillations produced are small, this required the K_d to very large to increase damping and K_p and K_i to be relatively smaller to decrease the amplitude of these oscillations. However this means that the robot is incredibly poor at handling disturbances, i.e. moving over a cable or a slight push. Thus this would not be suitable for the 2 outer loops for velocity and position as well as be incredibly poor at handling yaw.

3.6.2 2nd Iteration: Changing Motor Angular Acceleration

From this, a pure acceleration controller was chosen, directly changing the acceleration of the wheels of the bot to meet the angle set point. Directly changing the acceleration avoids the step changing velocity problem and decreases oscillations. This also means that for the inner loop $K_p = 1140$ can be large and $K_d = 435$ can be relatively smaller as the oscillations are tiny when static, as well as clamping acceleration, and damping is only required for dealing with large disturbances to the system, which with a larger K_p this system can handle. K_i in this system must be 0. This is because with enough error overtime, a large oscillation will be sent into the system knocking it over.

For the velocity controller, unlike the angle controller requires some K_i, This is used for dealing with changes in the environment, so that enough error is summed over time so it can overcome some obstacles. As expected, its velocity of execution is lesser.

The position controller follows the theory only requiring proportional gain.

3.7 Controller Performance: Simulations VS Experiment

3.7.1 Angle Controller

Simulations show the K_p/K_d ratio must be of 3.4483. In practice, the values that gave the best performance were:

$$K_p = 1600 \quad | \quad K_d = 480 \quad | \quad K_p/K_d = 3.33$$

To showcase the performance of this controller, a doublet response test was created. To ensure the robot wouldn't fall while testing, some support was given to the head unit, while letting the base unit move freely. Figure 3.7.1 show the angle loop's response to this doublet.

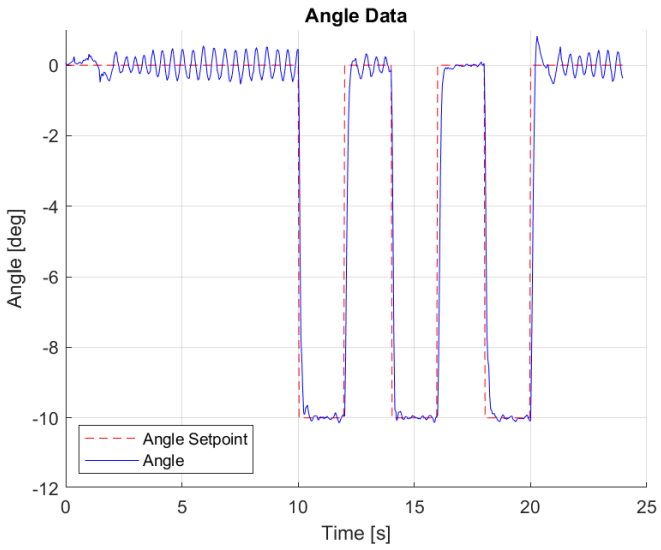


Figure 3.7.1: Angle Test Angle Loop Time Response

This shows adequate and satisfactory performance for the angle controller, supplemented by the simulation results in Figure 3.7.2

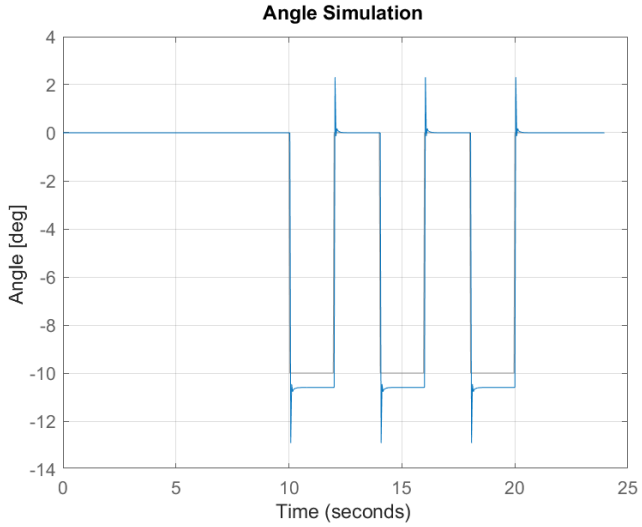


Figure 3.7.2: Angle Test Simulated Angle Loop Time Response

3.7.2 Velocity Controller

Simulations show we must have $K_i = 5K_p$ and $K_d = 0.019K_p$. In practice, the chosen values were:

$$K_p = 0.0066 \quad | \quad K_i = 0.033 \quad | \quad K_d = 0.000121 \quad | \quad K_i/K_p = 5 \quad | \quad K_p/K_d = 0.018$$

A doublet response test was also chosen to demonstrate the performance of the velocity controller, as is shown in Figure 3.7.4 and Figure 3.7.4. Although there are significant oscillations in the velocity time response, our model does not include friction. Friction is a key part in the dynamics of the robot. The arena's inherent imperfections and irregularities make it difficult for the robot to traverse space freely with no interruptions. There is also additional friction between the axle of the motors and the wheels which contributes to the observed effect. Friction contributes to the natural toppling angle of the robot. The higher the friction coefficient on the spot the robot is static at, the steeper the angle needed to make it move from its equilibrium position. Modelling an everchanging friction coefficient varying with the location the arena is a challenge and one of the shortcomings of this model. Therefore, all the observed oscillations around the setpoints are due to effects of the environment, namely friction. Nevertheless, the angle data and response is quite accurate and follows the setpoints well, despite the velocity's difficulties with the terrain.

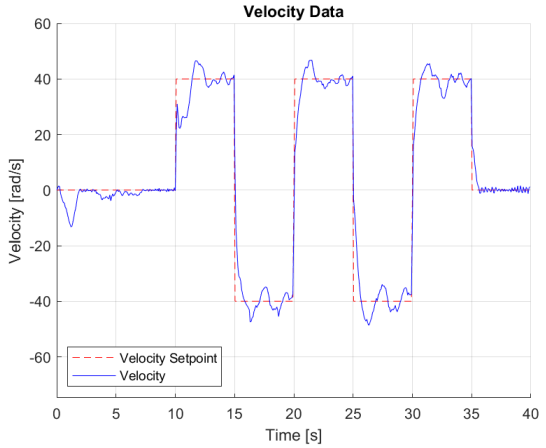


Figure 3.7.3: Velocity Test Velocity Time Response

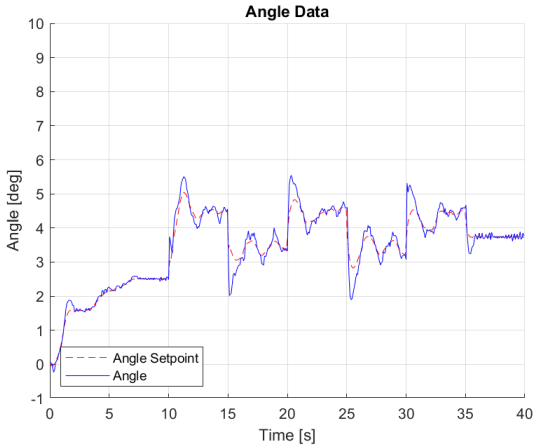


Figure 3.7.4: Velocity Test Angle Time Response

Simulation results confirm the theoretical performance of the velocity controller, as can be seen in Figure 3.7.5. Friction is the limiting factor between a good theoretical performance and our observed controller in practice.

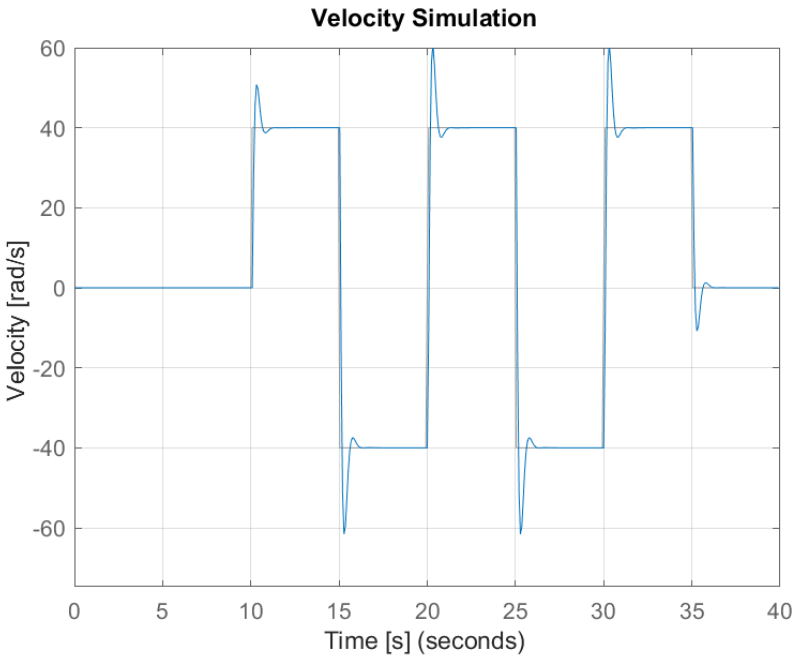


Figure 3.7.5: Velocity Test Velocity Simulated Results

3.7.3 Position Controller

Simulation results state that we only require a K_p value corresponding to the desired crossover frequency. This is also what's observed in practice, as a value of $K_p = 6$ fits the requirements and gives adequate position performance. This can be seen in Figure 3.7.6. Velocity and Angle responses are in Figure 3.7.7 and Figure 3.7.8 respectively.

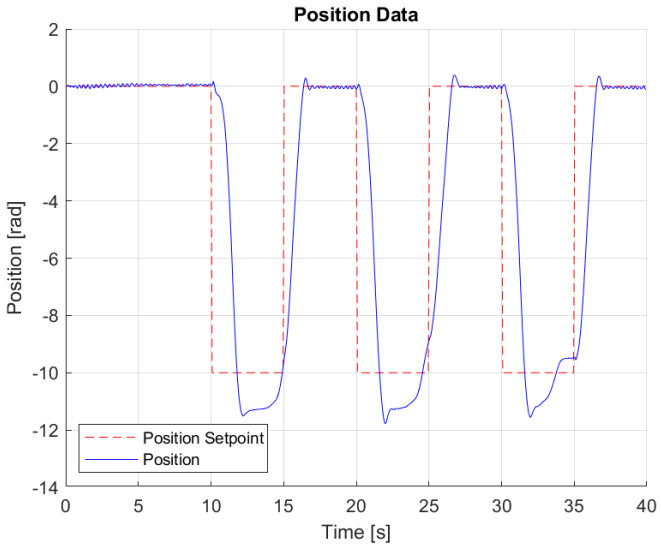


Figure 3.7.6: Position Test Position Time Response

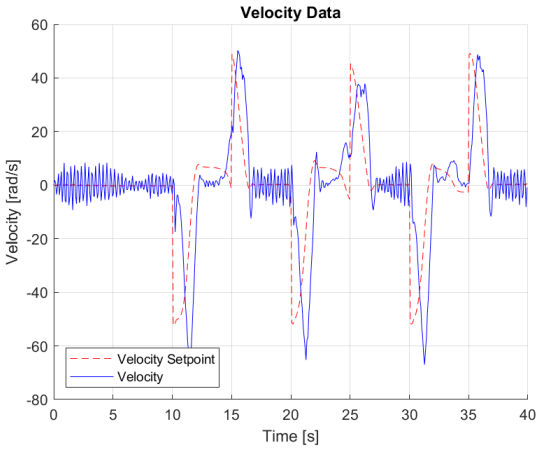


Figure 3.7.7: Position Test Velocity Time Response

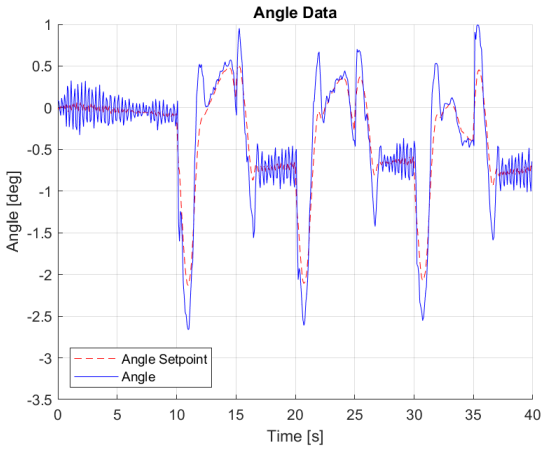


Figure 3.7.8: Position Test Angle Time Response

Like in the previous section, there is a significant error in the setpoint, whereas the equilibrium position is very stable. This is due to the inclination of the arena, where it is significantly easier for the robot to stabilize its position while going uphill, whereas it's harder for it to slow itself down when going downhill: the inherent dynamics of the robot make it harder to respond to a desired position and velocity. This is once again observed and confirmed in the initial overshoot in velocity and angle in the downhill direction, compared to the minimized errors obtained going uphill in both. Adding on to this, motor heating plays a significant factor in performance: after many trial runs, its performance can significantly degrade, leading to data that doesn't follow expectations.

In a perfectly flat surface, the simulations done in Figure 3.7.9, Figure 3.7.10, and Figure 3.7.11 (position,

velocity, and angle, respectively) show the ideal performances of the controllers on a flat arena. Hence, the designed controllers are adequate for our application.

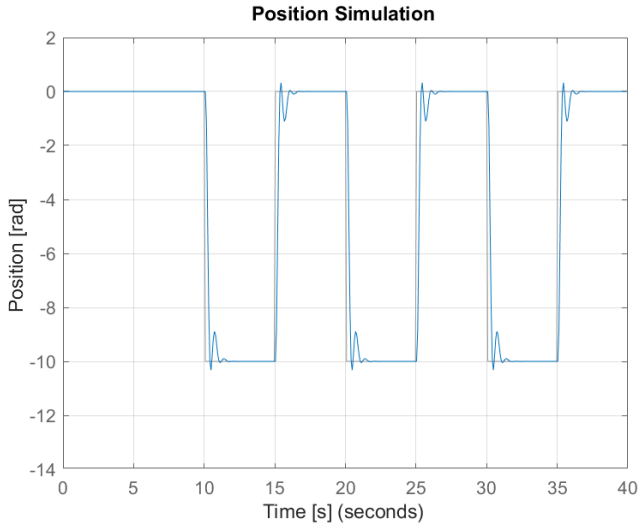


Figure 3.7.9: Position Test Simulated Position Response

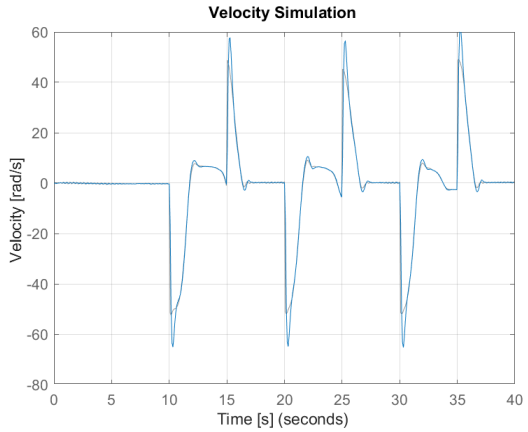


Figure 3.7.10: Position Test Simulated Velocity Response

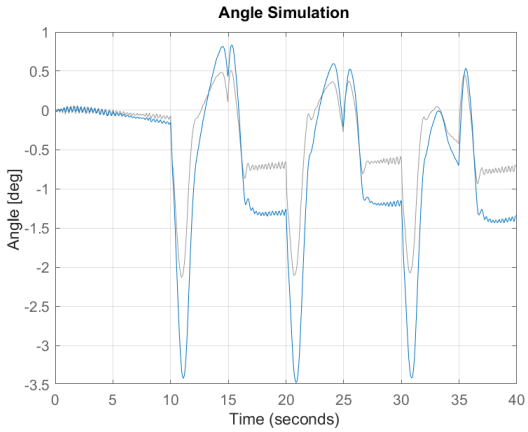


Figure 3.7.11: Position Test Simulated Angle Response

3.7.4 Yaw Controller

As was explored in Section 3.6, we have a proportional-only controller, that is only active while the robot is a state of turning. To test the capabilities of yaw, a step response was plotted as it is the closest response that will be observed in practice. The yaw and position response are plotted in Figure 3.7.12 and Figure 3.7.13 below respectively.

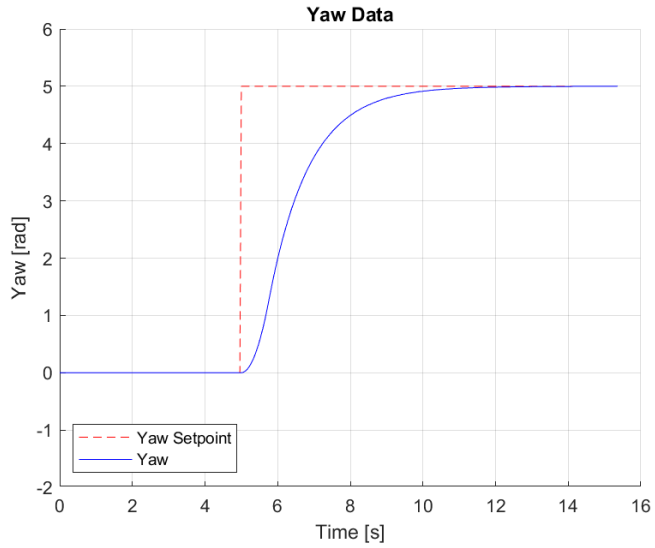


Figure 3.7.12: Yaw Test Yaw Step Response

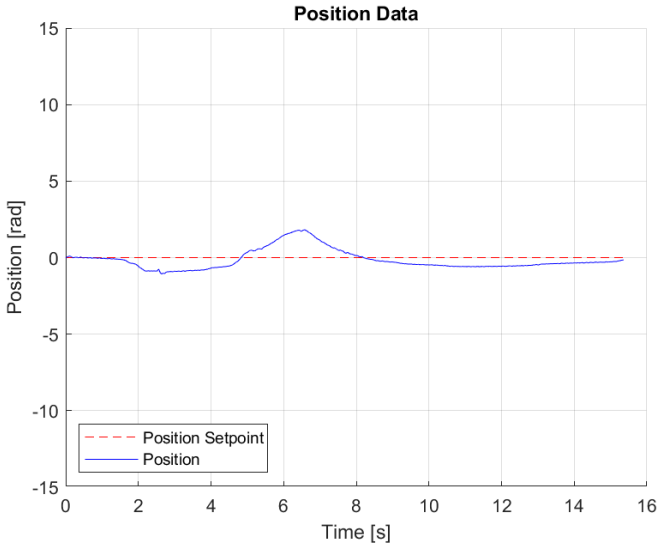


Figure 3.7.13: Yaw Test Position Response to a Yaw Step

Due to the unique implementation of this controller, no simulation was produced. Although there is some error arising in the position while the initial turning is in effect, both the yaw and the position reach equilibrium well within accepted error thresholds: position has an error of 0.3rad by the time the step response is over, and the yaw is accurate to 0.001rad after 10 seconds past the initial the step response instruction. This performance is sufficiently accurate to be included in the final design.

3.8 Conclusion

We have established a model describing the dynamics of the system, relating input angular acceleration to tilt angle, tilt angle to ground velocity, and ground velocity to position. We have also established the validity of this model through data verification, and simulated test comparisons against real world data. Overall, adequate performance was obtained after tuning using loop shaping on the three controllers. Limitations of our model include heating from the motors, the ubiquitous varying friction, and the curved nature of the arena. This friction is most notable between the wheels and the motors, and between the wheels and the ground, as the natural toppling angle varies with the location of the robot in the arena, and adds an unknown to the model. Overall, we have achieved a desirable control system, whose purpose is what it does.

For future work, it is strongly recommended to implement the code to directly output desired angular acceleration into the steppers of the motor, instead of having to work around it like we had to here. This would allow for the simulation results to be directly implemented, rather than having to find an unknown proportionality constant relating the PID coefficients and the actual observed angular acceleration.

4 Sensing and Monitoring

4.1 Power Monitoring

An important target for our robot is accurate battery management and power consumption analytics. The provided power PCB allows for direct connection to battery voltage and current sense resistors for both the battery and the motor connections.

The input should then be processed by the analog to digital converter channels on the ESP32 for processing. However, a direct connection between the J2 port on the PCB and the ADC ports of the ESP32 would cause the destruction of the board.

In this section the design of an appropriate analog interface between the power PCB and the ESP32 is presented, along with the calibration of the ADC channels on the ESP32 for accurate measurements.

Power Management Unit Requirements	
User Requirements	
Power Monitoring	The user should see the power consumed in Watts and in real time (+-30s) by both the motors and the components connected to the 5V power supply.
Battery Monitoring	The user should see the remaining battery life before shutdown of the robot as a percentage or user design element.
Technical Requirements	
Hardware Interface	A Hardware Interface needs to be constructed to interface between the J2 port pins and the microcontroller ADC. The interface needs to step down voltages from 15V and amplify voltage differential of 5mV to a voltage range between 0 and 3.3V for the ADC. This should be done accurately and with minimal noise (+- 1mV)
ADC Calibration	The microcontroller ADC needs to be suitably calibrated to read voltages within an accuracy of 20mV as stepped downed voltages are subject to significant absolute errors.
Software Interface	The power management controller needs to be suitably interfaced with the user interface. Polling of the ADC needs to be done frequently (+-10s) while not blocking critical control functions. The battery voltage needs to be transferred to a user-readable format (percentage) and the current sense need to be transferred to immediate power.

4.1.1 Hardware Interface

The interface, figure 4.1.1, is based on the Texas Instrument design [2].

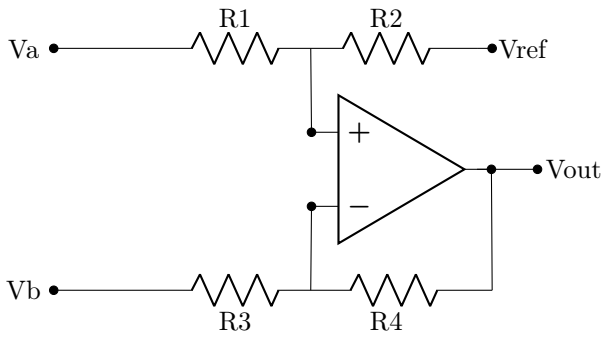


Figure 4.1.1: Differential Amplifier with Voltage Reference [2]

With R1=R3, R2=R4, the output is the following [2]:

$$V_O = (V_A - V_B) \cdot \frac{R_4}{R_3} + V_{\texttt{ref}} \tag{21}$$

The gain and reference voltage were carefully chosen to match the range of the ESP32 ADC. The gain was selected to produce an output voltage to ADC of less than 3V if possible. The reference of 1.25V was used as the ADC is inaccurate at low measured votlages.

System	Voltage Range [V]	Gain	Reference [V]	Ouput [V]
Battery Voltage	14.2 - 8	1/10	1.25	1.25-2.75
Battery Current Sense	5-4.97 (3A)	50	1.25	1.25 - 2.75
Motor Current Sense	Vbat-Vbat-0.01 (3A)	50	1.25	1.25 - 2.75

Table 4.1.1: Differential Amplifier Gain Table

Due to the high input voltage requirements, and to avoid the need to use potential dividers at the amplifier inputs, it was decided that the upper rail be connected to the battery voltage. The LMC648x was chosen for this purpose as it features a quad op-amp packaging, rail-to-rail inputs, high power supply rejection ratio and has a maximum upper voltage rail of 16V [8].

The decision to use this op-amp was motivated by the simplicity of powering the amplifier through the battery voltage, combined with having a single op-amp which reduces potential failure modes as identified in our failure modes and effects analysis. Two 1uF capacitors were used to perform averaging on the battery current sense in order to remove high-frequency noise.

The design was modelled using CAD software, figure 4.1.2, and was fully tested on a breadboard before being transferred to a strip board for completion. The decision to use a strip board stems from the increased reliability as identified in our failure modes and effects analysis, found in A.3.

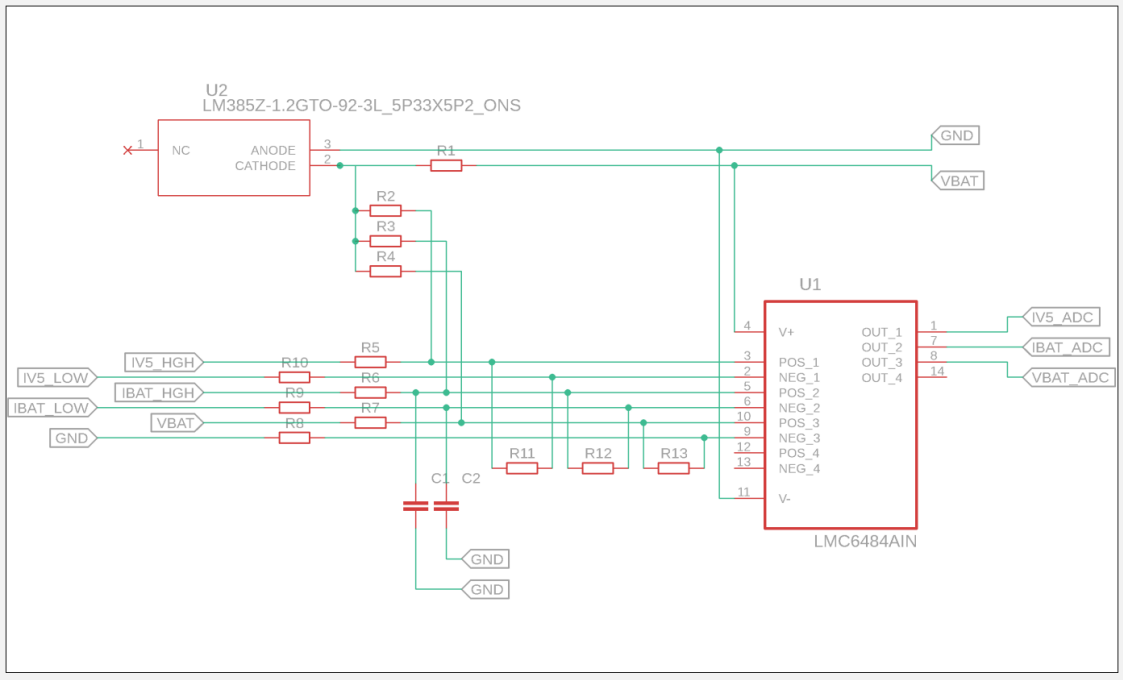


Figure 4.1.2: Hardware Interface Schematic

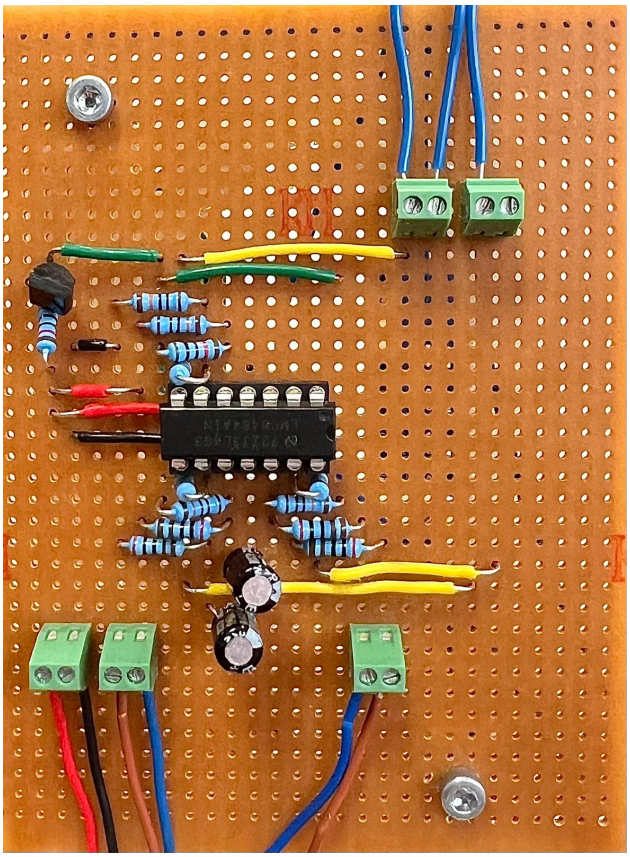
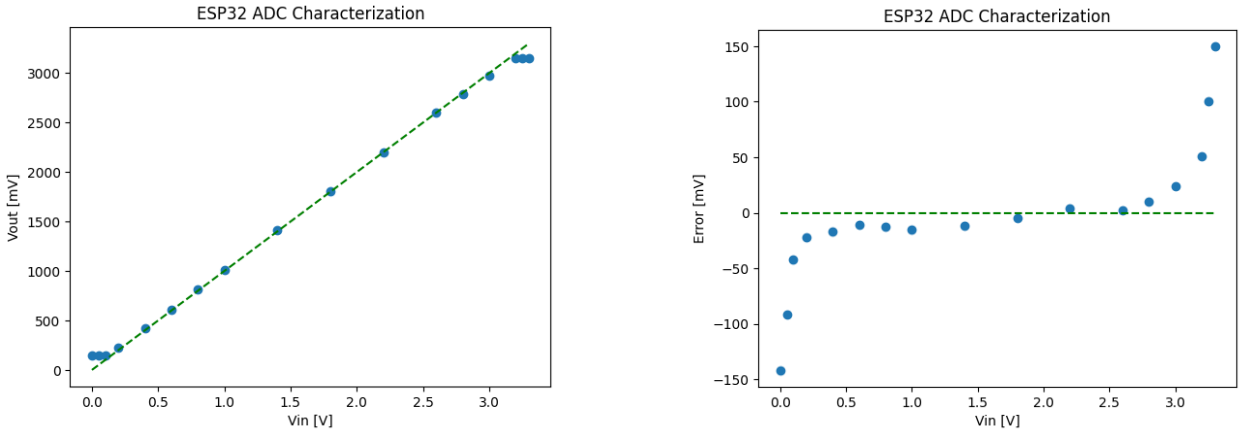


Figure 4.1.3: Hardware Interface Construction

4.1.2 Software Interface

Once a reliable interface between the power PCB and the ESP32 was established, the necessary software needed to be created to accurately sample the voltages and transcribe them to power or percentage ratings.

Firstly, the ADC was characterized to understand its operating range. The precision was set to 12 bits with 128 averaged samples and the measured voltage was recorded. Figure 4.1.4a shows the measured ADC voltage against the input voltage while figure 4.1.4b shows the absolute error in measured voltage.



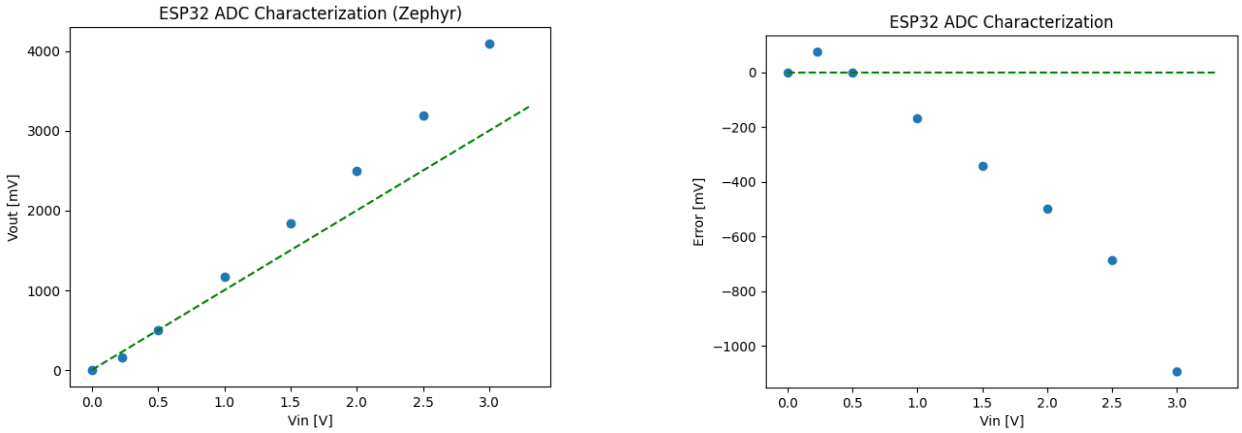
(a) Output Voltage against Input Voltage

(b) Absolute Error in Output Voltage

Figure 4.1.4: ADC Characterization (PlatformIO)

We conclude that the ADC is accurate between 0.2 and 3.0 Volts.

The use of ZephyrOS, required for the other subsystems, meant that the ADC came uncalibrated. A new characterisation was achieved to achieve calibration in INSERT FIG.



(a) Output Voltage against Input Voltage

(b) Absolute Error in Output Voltage

Figure 4.1.5: ADC Characterization (ZephyrOS)

To correct the significant ADC measurement error, a second degree polynomial was fitted to the measurement data in order to accurately remove the non-linearity. The polynomial coefficients were then used to calculate the correct ADC reading.

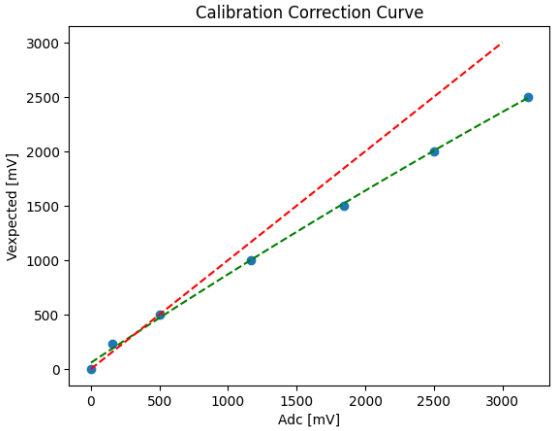


Figure 4.1.6: Calibration Curve for ADC

```
int32_t correct_adc_error(int32_t reading){
    return (-2.2517e-5 * reading * reading) + (0.83558 * reading) + 57.226;
}
```

Source Code 4.1.1: ADC Correction Polynomial

The states are measured every second through a separate thread which was declared as non-blocking so that the function of the ADC does not affect the motor controller or stepper driver.

Determining the power through the motor and 5V system is straightforward as the relationship is linear. However, determining the remaining battery percentage as a function of the battery voltage is more difficult, as battery percentage is a non-linear function of the battery voltage.

Thankfully, the discharge information provided by the battery manufacturer allows us to estimate the remaining battery percentage [3].

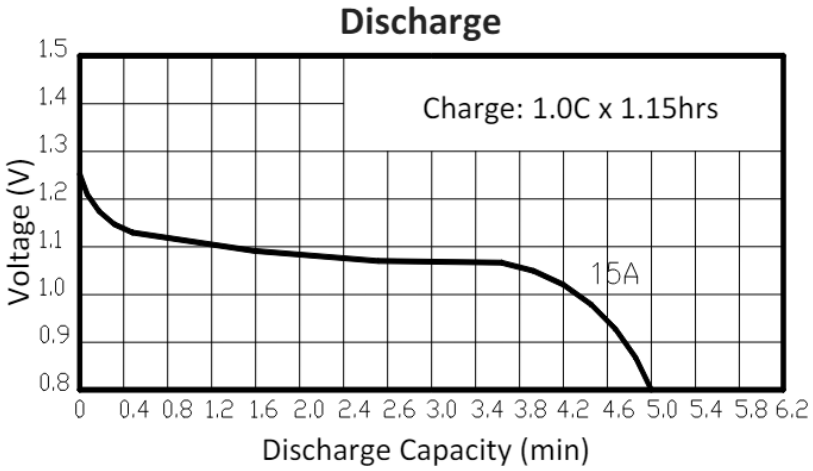


Figure 4.1.7: Battery Discharge Curve [3]

With the additional information that the batteries are mounted as a package of 12 cells in series, we can construct the following percentage table:

Voltage [V]	Discharge Capacity 15A [min]	Percentage
15	5.0	100
13.2	3.8	76
12.96	1.6 - 2.4	32 - 48
12.24	0.8	16
9.6	0	0

Table 4.1.2: Discharge to Percentage Conversion

Due to the low granularity of the battery percentage curve, it was decided to display the information to users using a bar chart that utilises the setpoints found above.

```
[00:00:46.961,000] <inf> adc: Battery Voltage: 15658 mV, Battery Current: 430 mA, V5 Current
    : 190 mA
[00:00:48.154,000] <inf> adc: Battery Voltage: 15635 mV, Battery Current: 400 mA, V5 Current
    : 190 mA
[00:00:49.348,000] <inf> adc: Battery Voltage: 15630 mV, Battery Current: 440 mA, V5 Current
    : 190 mA
```

Source Code 4.1.2: ADC Readings, Full Interface

4.1.3 Conclusion

To conclude, the power monitoring unit was found to adequately meet the requirements. Future work should focus on the introduction of precision voltage clipper circuits which would improve the reliability by adding overvoltage and overcurrent protection to the ESP32 ADC.

Additionally, more work should be done to allow for the automatic calibration of the ESP32 ADC using ZephyrOS, potentially contributing to the Zephyr Project .

4.2 Sensing

4.2.1 Sensor Choice

In order to map out the maze as the robot traverses it, ultrasound sensors are the perfect balance of cost to performance, allowing precise measurements of up to 3mm of accuracy up to 4m away [4]. For this reason, the HC-SR04 ultrasound sensor (Figure A.10) was chosen from RS components at £2.1. In comparison, other sensors like LiDAR offer much better performance, but are expensive and would make us go over the maximum budget. Infrared is also a potential contender, but would require us to make a separate board with our own design of opamps and filtering, as the most common pre-made board, the HCSENS0016, has a limited range unfit for our needs.

4.2.2 Physical Implementation of Sensing

For the maze solving 3 ultrasound sensors were required. One for the front and two for each of the sides. A back sensor is not necessary as that section of the maze has been mapped out already. These sensors are mounted to the bot using clips which snap onto the carbon fibre rods, this is done for secure mounting as well making the sensors easy to detach from the bot. However on of the specification requirements is to make a head unit for the balance bot. As the sensors are not going to be mounted on the head unit. A simple cap was designed as the head unit for the balance bot.

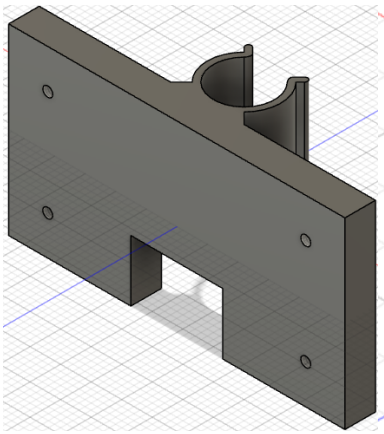


Figure 4.2.1: Ultrasound Sensor Clip

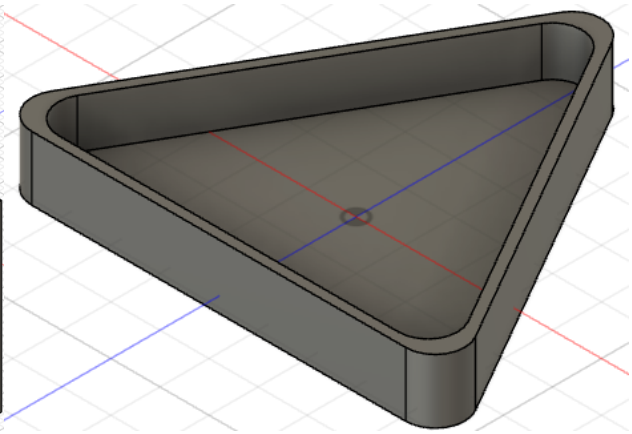


Figure 4.2.2: Bot Head Unit Cap

From this these will be mounted on the bot as such:

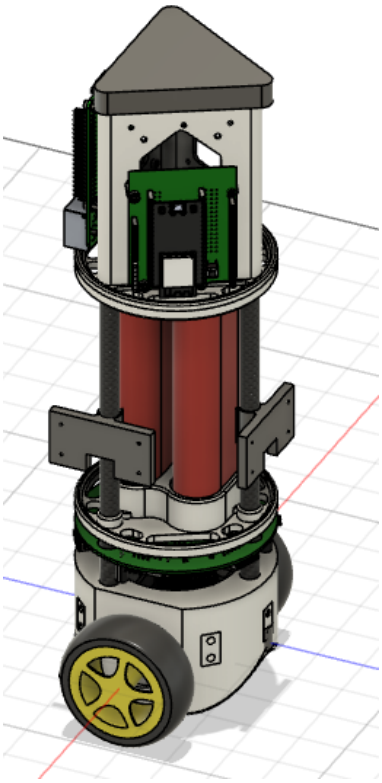


Figure 4.2.3: Parts Mounted onto the Bot

4.2.3 Maze Error Handling via Sensing

There are two main errors that can occur, one while the robot is static and the another while the robot is moving. The first is the robot not being in the centre of its square when static. This is resolved by performing the rough scheme using the front ultrasound sensor. This is done by the method of half squares. It is not important for the bot to be in the very centre of its current square, it only needs to be in the centre of the axis that the front sensor is facing. To achieve this multiples of 0.125m (half a unit square) is used. This issue will crucially arise when the robot turns. This will shift the robot off centre in the square. Thus when the robot moves one square forward the it needs to account for the robot being off centre. Thus we use front sensor distance mod 12.5 + 12.5 + wheel sensor distance as the distance the bot needs to move forward to be centred in the next square.

The second is an obstacle suddenly appearing in front of the robot both while static and while moving. This can be achieved by making a 10 entry FIFO. The error will occur, be passed through the FIFO to find the average distance read by the ultrasound sensor and given it runs fast enough will pass through.

4.2.4 Inertial Measurement

The IMU's accurate function is crucial to the completion of the Control Unit. The sampling rate must be high enough to allow for the control system to be stable. The angles that are detected shouldn't deviate from its actual attitude when the sensor is actually accelerating. This can be an issue with simpler algorithms which only treat acceleration caused by gravity as the only force applied on the sensor.

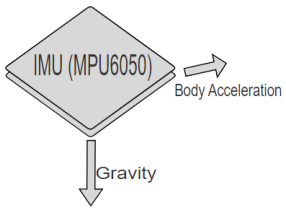


Figure 4.2.4: Force Diagram

- Accelerometer Gyroscope Sampling Method:
  The sensor which we are using (MPU6050) has an accelerometer and a gyroscope on one single die. It also contains a co-processor (DMP) which automatically filters the raw measurements and combines the accelerometer and gyroscope data to provide an attitude reading.
  This gives us the option to:
  - Sample the raw data at 1KHz
  - Sample pre-proccessed data at a maximum of 200Hz.

  In order to choose between the two, they were both tested with an accelerating body held at a constant pitch. The resulting samples are shown in the plot below:

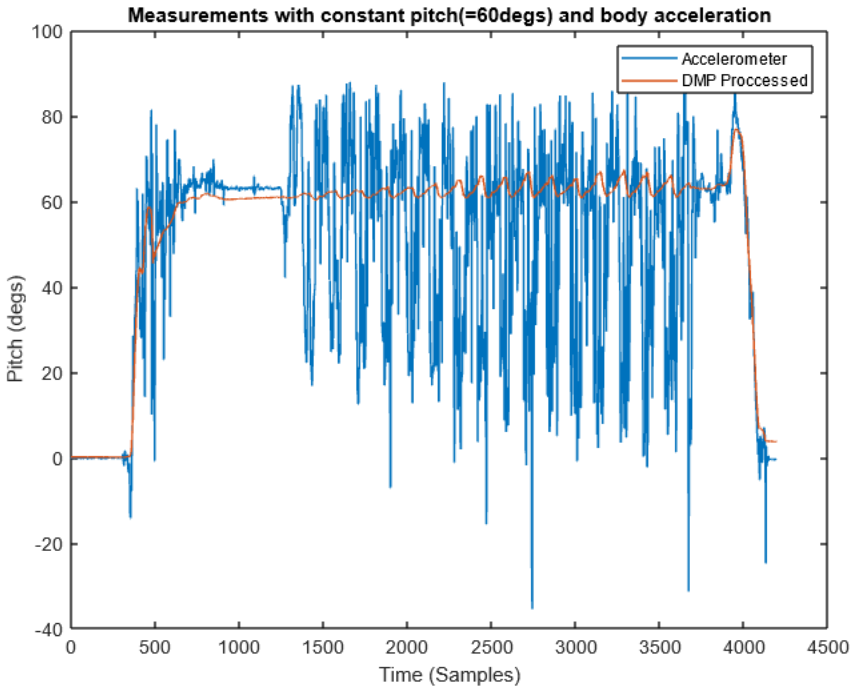


Figure 4.2.5: Filtered and Unfiltered Data

As shown in the figure: 4.2.5, the DMP-processed data is much more accurate and deviates far less from 60°. Although it would be possible to low pass the raw data, this would offset the usefulness of having small sampling times. Additionally, according to the Nyquist sampling theorem, we will be able to sample data up to 100Hz, which we will never be in the position to do. This is well within the requirements, allowing to say we can minimize effects of quantization, and maximize the approximation as a continuous-time system.

The DMP is used by flashing its code over the I2C bus and then setting its configuration parameters. The manufacturer (TDK Invensense) provides a portable set of source code which needed I2C and timing functionality implemented.

- Calibration tools:
  It is important to be able to remove consistent error. The DMP has an inbuilt factory calibration routine which works by the chip artificially introduces an electro-mechanical perturbation to the sensors and measures offset characteristics. The robot is programmed to run this utility on boot or can be programmed to store the offset values. [9]

5 Software

5.1 Overview

Software Requirements	
User Requirements	
Power Monitoring	The user should see the power consumed in Watts and in real time ($\pm 30s$) by both the motors and the components connected to the 5V power supply. This prevents accidental damage to the batteries.
Control of the robot	The end user must be able to control the main functionality of the robot, including manual driving of the robot as well as the control and monitoring of the robots maze solving capabilities. This means that the software must interface all the sensors, to co-ordinate the actuators, and provide feedback to the user interface.
User experience	The user experience must be intuitive and reliable, providing concise debugging feedback if anything goes wrong. For ease of use, control loop tuning parameters should be remotely tune-able restarting. Reliability measures include ensuring that randomly closing the user interface doesn't cause any erratic behaviour.
2-wheeled driving	The software must implement the necessary control systems to allow the robot to stably drive whilst balancing on two wheels.
Autonomy	The robot must be able to autonomously traverse a maze, mapping it out as it goes along until it arrives at a desired "end position" in the maze, effectively solving it.
Technical Requirements	
Modularity, efficiency and portability	To make the robot as flexible and cost effective as possible, it is important that interchanging individual components, or adding new components altogether is a straightforward process. The software should not be wasteful, nor platform specific (whilst not sacrificing maintainability) such that more affordable, less (or more) powerful hardware can be used down the line to optimise production costs.
Interfaces	The software should provide interfaces not only with the various sensors and motors, but externally between the various software driven modules and internally between concurrent threads and callbacks. this means making use of synchronisation primitives where necessary to prevent concurrency bugs, and extensively testing custom protocols for robustness.
Data Processing	The software should act as more than simple plumbing between modules - it must concurrently perform the necessary processing on the various data to fulfill the user requirements.

5.1.1 Overall implementation

Below is the overall architecture of our system:

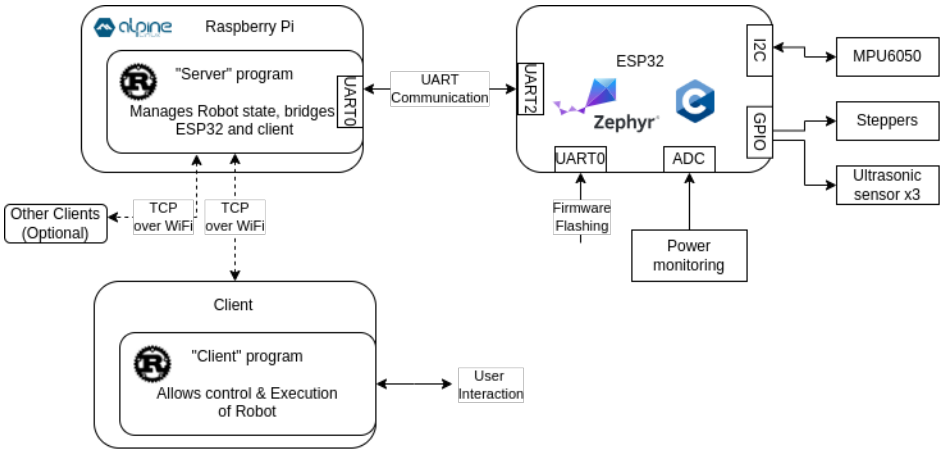


Figure 5.1.1: Overall architecture

Below is the overall software architecture, explained in greater detail later.

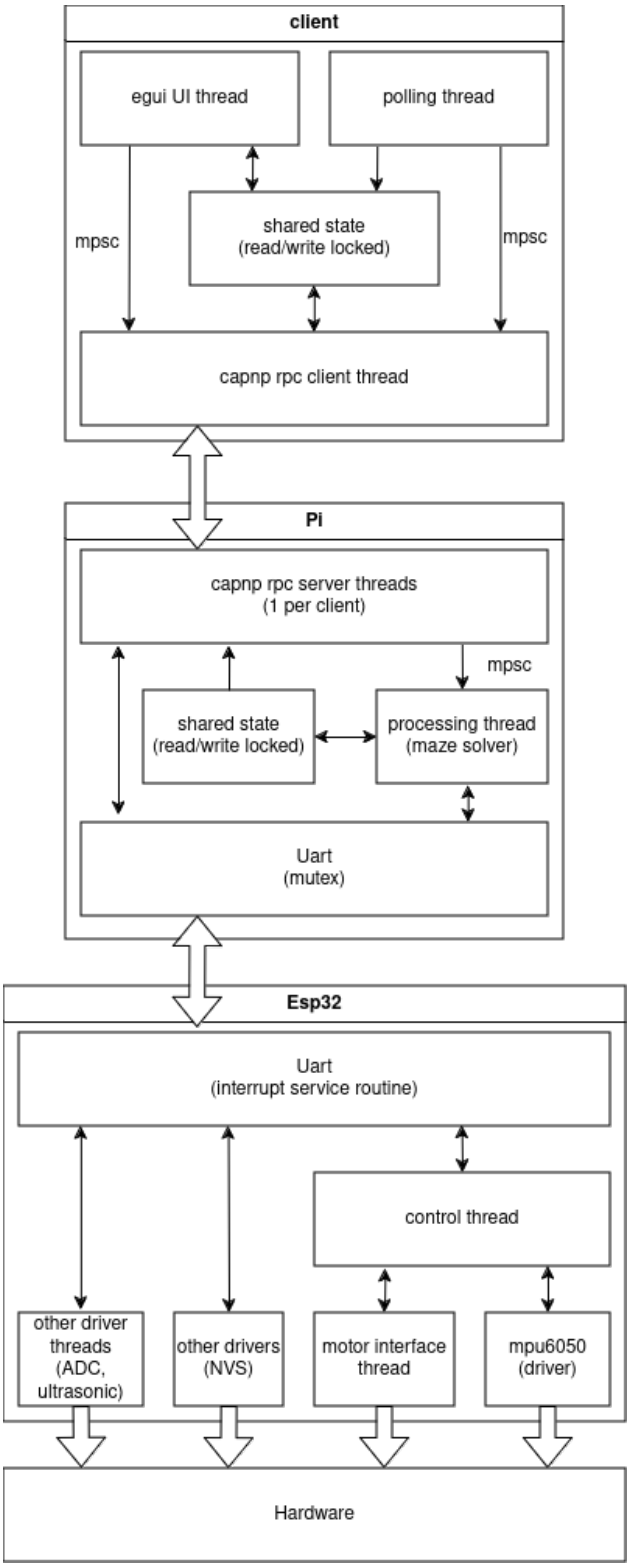


Figure 5.1.2: software architecture

5.2 Firmware

The Firmware is the code designed to run on the microcontroller, it requires reliability (the microcontroller must not crash and must do processing in real time), efficiency (using resources efficiently and using vendor features where possible), and capability (must implement all of our required functionality).

Firmware Requirements	
Technical Requirements	
Robot Balancing	
Accelerometer/Gyroscope	Data must be sampled from the sensors accurately and without blocking the operation of the robot. Tilt angle, yaw and acceleration along the robot's longitudinal access must be readily available.
Stepper Motors	Unit to be designed so that a desired acceleration is given as input and achieved by the stepper motors. The unit must also keep track of the distance travelled by the motors.
Control Algorithm	Need to implement the PID controller structure laid out in the Control Unit section. The operation must be fast enough to satisfy the control requirements with regards to sampling frequency (<5ms).
Interfacing Other Hardware	
Power Monitoring	Needs to accurately sample ADC values and convert them to power and battery percentage ratings. This must not block the control operation.
Ultrasonic Sensors	Needs to design and implement a driver for the ultrasonic sensors to accurately read the distance to an obstacle.
Interfacing Server	
Raspberry PI	Needs to receive and process commands from the Raspberry PI without blocking and be able to send data back to the PI.

We decided to use the provided ESP32 as our microcontroller due to it already being provided, and also meeting our requirements. It also has plentiful IO and is a popular, well documented and supported platform.

Instead of the provided Arduino-based starter code, we instead opted to use Zephyr RTOS, a Real-time Operating System, for the following reasons:

- It provides advanced scheduler that allows us to run multiple simultaneous threads that handle different activities, as well as providing synchronization primitives (e.g. mutexes & workqueues). This is extremely useful for such a system where multiple tasks are performed simultaneously with data transfer between them in a thread-safe way.
- Excellent support of the underlying ESP HAL, allowing full use of all required peripherals (I2C, ADCs, interrupt-driven UART, non-volatile-storage etc..), as well as timers, interrupts and even symmetric multiprocessing [10], allowing us to make fullest use of the hardware we are given.
- Zephyr projects are very portable, requiring only a device-tree overlay to support a new board/core, meaning none of the C code needs to be changed to port the firmware to another board. This system is famously used to configure the Linux kernel to support different hardware. Zephyr comes with premade device-trees for each supported platform, which must be modified with a overlay to overwrite parts of the default device-tree. `device-tree` bindings (`yaml` files) are used to compile-time check the device-tree by specifying the allowed configuration options.
- Zephyr is configurable through it's `Kconfig` system. This allows unwanted features to be left out, reducing the size of the firmware image

- Zephyr can be configured with a whole host of other useful features, such as a GDB stub, a customisable shell and and even a thread analyzer.

5.2.1 Robot Movement

To facilitate switching between manual and maze solving mode, our motor controller operates in two modes.

Both modes share a common angle and velocity controller to stabilize the robot. In order to move a desired position, an outer loop is added to create a position controller. In manual mode however, the setpoint of the velocity controller is directly controlled and the position controller is bypassed. The inner angle loop gets its error from the MPU, whilst the outer loops get their errors from the motor driver.

Turning is handled by applying a direct velocity differential to the two motors, with maze mode using a controller to turn the robot by a desired angle. The controller uses feedback from the motor driver, as integrating steps has less error than the MPU.

The design and implementation, as well as test cases to showcase the performance of these controllers is explored in Section 3.

5.2.2 Stepper Driver

The motors need to be stepped every time they are to move: a motor driver was written to step the motors at the correct intervals to rotate the wheels at the desired speed. In order to make the steps more precise, the microstepping technique was used, dividing each physical step angle on the motor into finer steps.

Due to the driver having to run constantly, it has to be put in its own thread so as to not block other parts of the firmware. The stepper driver thread provides interfaces for setting a desired common acceleration, and a differential velocity between the individual motors. In order to provide feedback to the control loops, the stepper interface also provides the robots acceleration and displacement.

5.2.3 Ultrasonic Sensor Implementation

We decided to use three ultrasonic sensors - one on the front, one on the left and another on the right. One on the rear of the robot is not necessary as it will have come from that direction (and therefore know what is there), or rotated, in which case either the left or right sensor will have already measured that direction.

The driver is implemented as a custom zephyr driver conforming to the sensor subsystem. The sensors have a trigger pin and an echo pin, and to measure the distance a pulse is sent on the trig pin, and the time is measured for a pulse on the echo pin. To implement this in code we set a callback on the echo pin, pulse the trig pin, and in the callback handler we measure the time taken.

Measuring distances from all ultrasonic sensors simultaneously is not possible, as the emitted ultrasonic waves may be received by other sensors. Therefore a global `mutex` is used in the driver, ensuring only one sensor can be measured from at once.

From our code we can then measure distances using Zephyr's sensor API. We first request a reading using `sensor_sample_fetch_chan`, and the resulting value can then be accessed using `sensor_channel_get`.

5.2.4 Non-Volatile storage

PID values are stored in the ESP32's non-volatile storage, which are loaded on startup. This means the client application can update PIDs, either just temporarily in the robots memory, or persistently by loading saving them to the NVS. To implement this we use Zephyr's Non-Volatile storage API, and use the ESP32's storage partition area in flash for storage.

5.2.5 Communication with the Raspberry Pi (Server)

The ESP32 must communicate with the Pi Server to receive commands, such as movement requests, ADC Reading requests, PID update/retrieve requests.

Instead of using UART0 (on the USB port) we decided to instead use UART2 so that we could continue to read logs/flash firmware from our devices over USB with ease. The communication is designed in a way such that the Pi must first make a request to the ESP32 which may then respond, instead of fully bidirectionally, since otherwise both devices could start transmitting at the same time since when the devices listen for the reply they cannot be sure if it is the reply to their request or an entirely new request.

We tested several packet layouts starting with encoding commands as new line terminated text. A mixture of text and bytes, however the use of raw ASCII proved to be wasteful, with each character taking up at least a whole byte. This also came with the issue of ensuring that raw data bytes didn't take on the form of a newline.

Our final improved protocol is as follows: (see `src/uart/uart.c` for full implementation).

- The first byte specifies the size n of the payload. This is done since the enclosed message can contain any valued/ordering of bytes, including newline/carriage return/null termination characters, so listening for such characters to mark the end of the message simply cannot work. Therefore the software will listen for this number of many bytes following it.
- The next byte specifies the message type (e.g. ADC read request).
- The next n bytes form the struct for the respective message type.

Protocol Evaluation		
Command Method	bytes (for PID packet)	Overheads
ASCII commands	17	ASCII to float / int Conversion & Regex
Mixed Commands	9	Regex
Polymorphic struct (our solution)	7	Typecasting has practically 0 overhead, so the decoding overhead is very small

Table 5.2.1: Protocol Evaluation

This is implemented as follows:

- When the ESP32 receives a character over UART, our callback function is called from an IRQ.
- This first character is the size n, and we wait for n more characters to be received, and place them in consecutive places in our buffer.
- When n characters have been read, this buffer is placed in our message queue[11], which is delivered to a separate thread, and the index is reset to 0. Sending this to a different thread via a message queue is very important, since it is vital we don't block in an interrupt request. The other thread can then process this message (and optionally block), and this is not a problem since this thread can be interrupted, and the message queue also allows for messages from the interrupt handler to build up to a given size.
- The first byte of this message is the message type. Message types are stored in an `enum`, allowing quick checking of message type in a switch statement. The respective handler is called.
- The handler will then cast this buffer to the respective struct for that message type. The performance is great as the input buffer can simply be cast to the struct from which values can subsequently be retrieved, with 0 memory copy operations since placement into the message queue.

5.2.6 Evaluation

From a qualitative standpoint, all our desired functionality works extremely well and reliably during all our testing, and due to us using absolutely no dynamic memory allocation it is unlikely to crash after running for long periods.

We can also use tools to determine some metrics about the performance. Zephyr's thread analyzer allows us to determine what threads use the most CPU, and how much of its allocated stack each thread has used. We compiled the firmware with it enabled, and after 2 minutes, we collected the following results.

```
Thread analyze:
:  stepper_driver       : STACK: unused 1752 usage 296 / 2048 (14 %); CPU: 7 %
                        : Total CPU cycles used: 2026917572
:  uart_handler_tid     : STACK: unused 888 usage 1160 / 2048 (56 %); CPU: 0 %
:                       : Total CPU cycles used: 2527471
:  thread_analyzer      : STACK: unused 2704 usage 1392 / 4096 (33 %); CPU: 0 %
:                       : Total CPU cycles used: 64578855
:  motor_controller_tid: STACK: unused 2784 usage 1312 / 4096 (32 %); CPU: 1 %
:                       : Total CPU cycles used: 286428340
:  adc_measure_tid      : STACK: unused 680 usage 344 / 1024 (33 %); CPU: 8 %
:                       : Total CPU cycles used: 2346960189
:  idle                 : STACK: unused 824 usage 200 / 1024 (19 %); CPU: 83 %
:                       : Total CPU cycles used: 23481415681
:  ISR0                 : STACK: unused 1872 usage 176 / 2048 (8 %)
```

The threads are as follows:

- stepper_driver: Steps the motors. Mostly sleeps but runs at a fast rate so uses a moderate fraction of the CPU cycles.
- uart_handler_tid: Handles inputs from the UART, fully interrupt driven so uses very few CPU cycles.
- motor_controller_tid: Runs the PIDS to balance the robot, mostly sleeps so uses few of the CPU cycles.
- motor_controller_tid: Runs the PIDS to balance the robot, mostly sleeps so uses few of the CPU cycles.
- adc_measure_tid: Continuously samples the ADC so uses a fair amount of CPU.

The CPU is idle for 83% of clock cycles meaning there is no issue with resource contention.

5.3 Server

5.3.1 Overview

Our system needs a server that will run on the robot, that allows communication with the client, performs the necessary logic to solve the mazes, and to send/receive data from the ESP32.

Server Requirements	
Technical Requirements	
Efficiency	The Raspberry PI has very limited resources available. The choice of operating system must be as lightweight as possible while maintaining the other requirements.
Interaction	Must be able to communicate and interact with both firmware and client. It must send instructions to the firmware and pull or push certain parameter values like control PIDs.
Reliability	Must not crash and must be bug free. While not fatal for the robot, a crash would ruin the user experience and effectively prevent a maze being solved.

We decided to use the Raspberry Pi we were given as it meets our needs. For the Operating System we opted to use Alpine Linux:

- Can be run in diskless mode where the system is run entirely from memory - this massively improves file I/O performance and reduces SD card wear. Disk modifications can be persisted if desired.
- Alpine software repo contains all the packages we will need.
- Extremely minimal, our install uses only 40Mb of RAM at idle, with only a handful of processes, which is important for the Pi which already has very limited resources. Explicitly does not include a Desktop environment.
- Excellent support for the Raspberry Pi, and supports use of peripherals such as the full hardware UART using the disable-dt overlay.

We also decided to use Rust for the following reasons:

- Extremely performing systems programming language that compiles directly to machine code, which is important due to the limited processing power of the Raspberry Pi.
- Rust's ownership system and borrow checker prevent entire categories of memory errors like dangling pointers and buffer overflows. This is important for this system due to our requirement of stability.
- Rust programs will not segfault or unexpectedly crash, and must explicitly handle all errors (e.g. I/O errors).
- Rust has excellent support for concurrency, and the ownership system and type system help catch concurrency bugs at compile time.
- Rust is well supported on the Pi (`aarch64-unknown-linux-musl`).

5.3.2 Implementation

The server is connected to any number of clients over TCP, using Capnproto remote procedure calls. This allows clients to send requests to the server, such as instructing it to start solving mazes, updating PID values, or polling the state of the maze from the robot.

It is also connected to the ESP32 using UART - The protocol has been discussed in the firmware section, and the Rust implementation mirrors it.

The following data structures are created:

- A UART communication object referring to the UART interface from which bytes can be read/written to. It is very important that this interface is only used at once (e.g. it must be forbidden for two threads to write consecutively before the first reply has been received), so we therefore place in a Mutex to enforce only one access at once. This is wrapped in an Arc, to allow sharing between threads, as described later.
- A struct containing all state - this is largely maze state (a representation of the robots knowledge of what exists in the maze). This is wrapped in an RwLock, which allows either a number of readers or at most one writer at any time, which is also nested within an Arc (a thread-safe reference-counting pointer), allowing it to be given to multiple threads.
- A `mpsc` object (a Multi-producer, single-consumer FIFO queue), which is a channel that can cross a thread boundary with both multiple senders and one receiver.

The following threads are then created:

- A main "event loop" that executes the maze solving algorithm and communicates with the ESP32 over UART. It receives the receiver side of the MPSC so can see what clients want the robot to do, and logic is implemented to change what it is doing based on current state and requests. This thread has access to the UART which is used to command the robot.
- Threads are also created for each incoming client TCP connection to handle RPC requests from clients. They are given a MPSC sender, allowing it to submit request for actions that the server can perform. Some request types need to communicate with the robot directly (e.g. setting/retrieving PID values) so it also has a copy of the UART object (hence the importance of the Mutex). The MPSC is necessary, since the RPC handler is not capable of knowing if the robot is currently moving one unit square, so therefore it is sent to the main thread, which can retrieve this request from the MPSC once it has finished moving and is able to service new requests. This is able to eliminate edge cases where clients send conflicting/nonsensical requests.

On startup, the server requests the uptime from the ESP32 over UART, to test the communication link. The server then starts the main event loop, and places the robot in manual mode. It then waits for requests coming from clients. The maze state is reset to default (empty with only outer walls),

The client periodically polls the state using the RPC connection, and receives the current maze state, along with battery ADC readings which are requested from the ESP32.

To start the maze solving, the client will send a MazeStartRequest, which is sent to the main thread. Upon receiving this request, the server will set the mode to Maze solving and runs an iteration of the maze solving algorithm which determines the next square to move to. If the rover is not facing the correct direction to move, the server will send a rotation request to the ESP32, and will poll the robot, waiting for completion. Now that it is facing the correct direction, the server will send a move request to the ESP32 which will move it one unit in the direction it is facing. It can now see if there are any queued requests in the MPSC channel (such as stopping the maze solving algorithm), and if not, will continue - in this case it will keep solving the maze step by step.

5.4 Client

5.4.1 Overview

The client is a program running on an end user's device, which interfaces with the server running on the Raspberry Pi.

Client Requirements	
User Requirements	
Displaying Information	The client must display the battery voltages, currents, the current state of the maze and position of the robot.
Mode Switching	The client must allow for mode switching between manual control where a user input is directly fed as a set point for the control algorithm, and maze mode, where the robot can autonomously solve a maze it is placed in.
Uploading Information	The client must allow for parameters to be entered by the user to be uploaded to the robot such as PID values for the control algorithms.
Appearance	The user interface must be intuitive and visually appealing. A first time user should be able to effortlessly operate the robot.

We therefore decided to use Rust for the client, as

- Rust was already in use on the server, allowing code reuse for certain parts.
- Rust supports GUI libraries - we decided to use eGUI due to it being reasonably simple, fast and supporting all needed UI features (e.g. buttons, text inputs, as well as arbitrary painting which we need for rendering the maze).

To maximise reliability, the client's internal state must only affect whats shown on the the UI, such that an internet cutout doesn't cause the robot to behave erratically. To achieve this, most of the UI elements simply trigger a remote procedure call, and populate the UI with the response.

5.4.2 Implementation

To communicate with the server, we use Remote Procedure Calls. Since requests may take a long time (100ms+, especially if I/O operations are required on the server), these requests must happen in a different thread to the thread which renders the UI, otherwise the UI would be hang until the server responded and the client could continue rendering.

To implement this we use a MPSC, which is a channel that can cross a thread boundary with a sender which is given to the renderer thread, and a receiver which is given to the RPC-sender thread. Furthermore, a state struct is created that contains all data needed to render the UI from, such as the robot battery voltages, maze state etc... . This state struct is also handed to both threads, and is nested within a RwLock, which allows either a number of readers or at most one writer at any point in time (needed for thread safety), which is also nested within an Arc (a thread-safe reference-counting pointer), allowing it to be given to both threads.

When the user interacts with the UI, e.g. presses a button that changes the robots operating mode, a message is sent from the UI thread to the RPC-sender thread. It then sends the request and waits for the reply, all whilst the UI thread continues rendering. When the server responds, the state struct is updated, which the UI thread will then use when rendering the next frame.

Alongside this, the client also periodically polls the server, for e.g. new battery voltage values or new maze state, which updates the local state struct.

5.4.3 Architecture overview

The client is split up into 3 main interconnected components. The UI, the interface with RPC server, and the glue that connects the two systems together. Rust was chosen again for our client to allow code reuse in our codebase, which much of the knowledge gained from working on the client being transferable to developing the server, and vice-versa. The client makes extensive use of Rust's memory and thread safety features, allowing us to avoid concurrency issues.

This meant that we had to use the rust implementation of the capnp rpc client, which is a relatively experimental library with limited functionality. The UI implemented with eGUI, running natively with eframe because a group member has prior experience with the library, and it is easy to use and fast to develop for.

5.4.4 UI

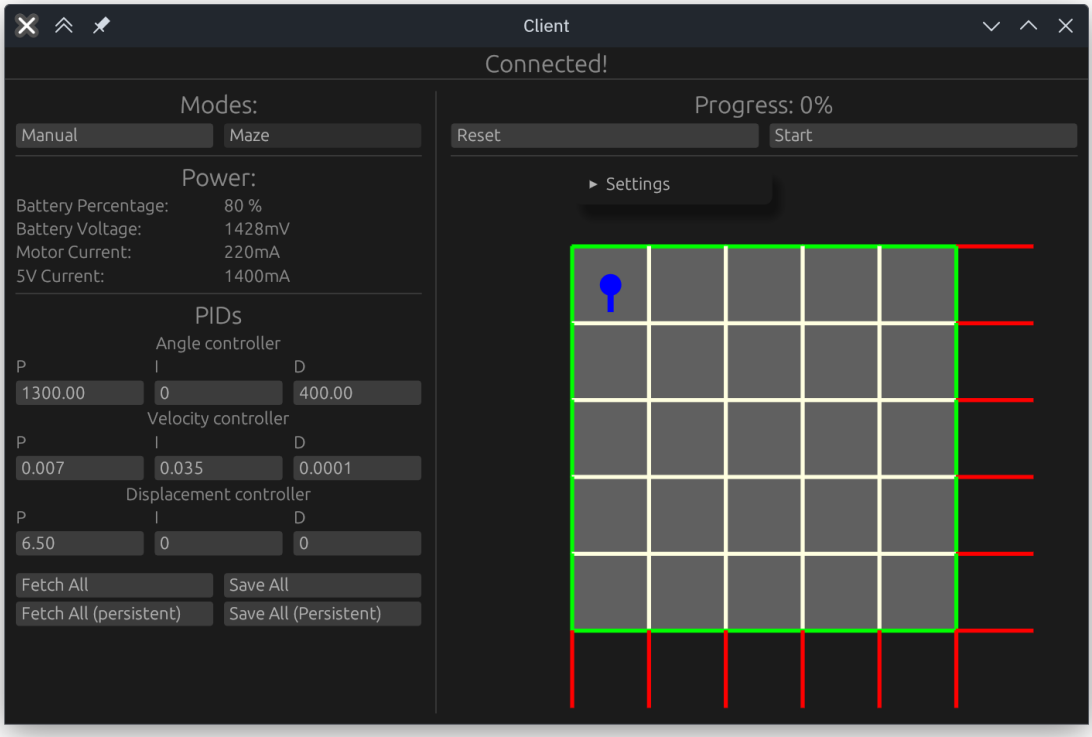


Figure 5.4.1: UI Interface in maze mode

On the left of the client application is a mode selector that selects whether the robot is in manual control mode or maze solving mode. Below is a panel that displays the robots' power readings, and there is also a panel that can set PID values remotely.

The right side changes based on the current mode - in maze mode it displays a rendering of the maze as it is known to the robot. Walls that are known to exist are displayed in green, walls that are known to not exist are displayed in red, and walls that are unknown are displayed in white. The outer walls start in a known existent state. Due to the way the maze is represented in memory (each cell only contains a left and top wall to avoid repeated walls), there needs to also exist an extra row and column of cells on the bottom and right, and some of these walls can be set to known nonexistent. The maze will update as the robot discovers the maze using the polling functionality described.

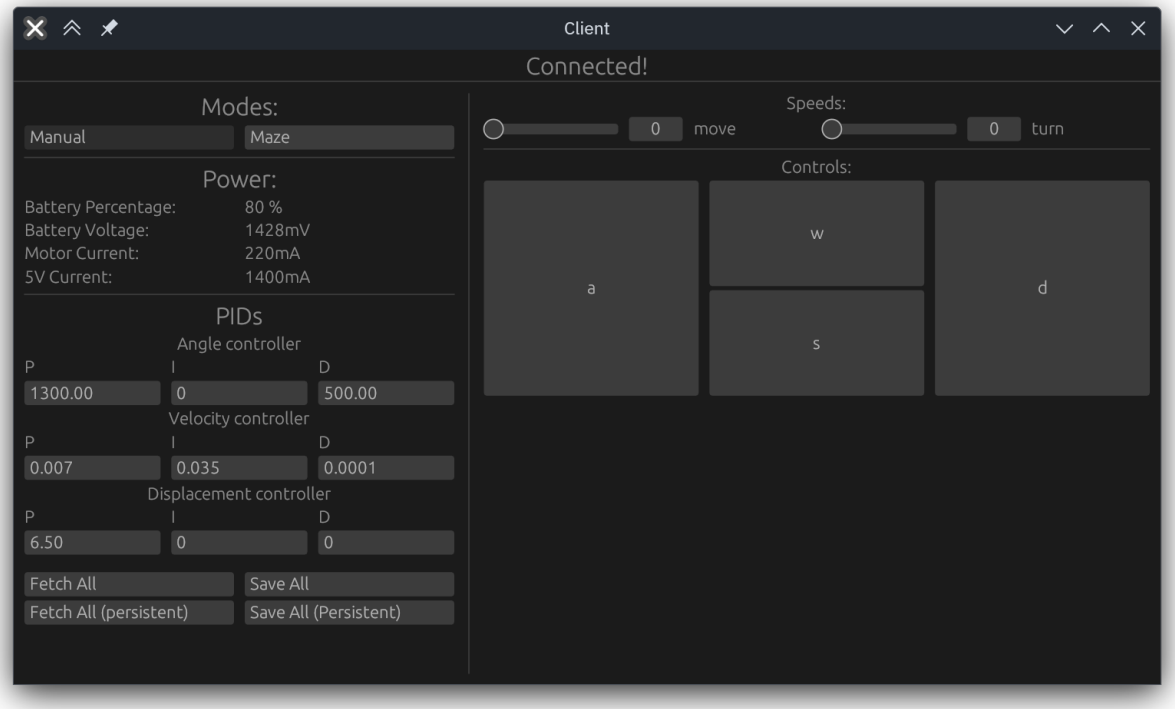


Figure 5.4.2: UI interface in manual mode

In manual mode we are able to control the robot as shown above. There are also keyboard keybinds for ease of use. These are sent over RPC to the server where it is handled.

6 Conclusion

This project has successful produced a self-balancing, two-wheeled maze solving robot with battery management. From this we have succeeded in implementing a Zephyr operating system allowing for multi threading communicating with a robust memory safe RUST server between the ESP 32 and Raspberry PI all of which has a deploy-able tech stack that can be implemented on most micro controllers. This departure from the initial platformIO code, has allowed for successful parallel implementations for the battery management module, complex motor control system involving, acceleration, speed, position and yaw and finally multiple ultrasound sensors for fast maze mapping. More importantly, it allows a reproducible tech stack on another micro controller which improves product deploy-ability. The maze on which the product is to be deployed can be modified in a modular fashion.

Time permitted, the UART serialisation packs could have been made a single byte smaller. For the battery management side, a breaker circuit may have been devised in the event of extremely high current draws. Furthermore trigonometric implementations could have been used to accurately calculate the angle turned for yaw. However these improvements are either minor and inefficient to implement time-wise.

Throughout this project, we learned a multi-disciplinary set of skills, ranging from how to develop concurrent interconnected system in Zephyr and rust, to modeling and stabilizing unknown systems. We also learned a variety of soft skills such as how to collaboratively work on a code base, and make optimal use of individual group members technical strengths.

A Appendix

A.1 Figures

```
int32_t yaw_step_timer;  // time since last step (us)
int32_t yaw_step_period;
int32_t yaw_current_speed;
int32_t yaw_target_speed;
int32_t yaw_position;
int32_t yaw_rotation_target;

[...]

void stepper_set_yaw_speed_rad(struct stepper_state *sconfig, float speed_rad);

void stepper_set_yaw_steps_rad(struct stepper_state *sconfig, float pos);
```

Source Code A.1.1: Additional Parameters and Functions Needed to Implement Yaw

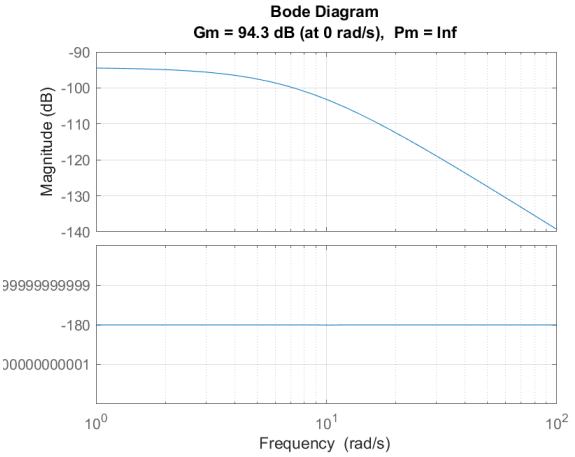


Figure A.1: Tilt Natural System Response

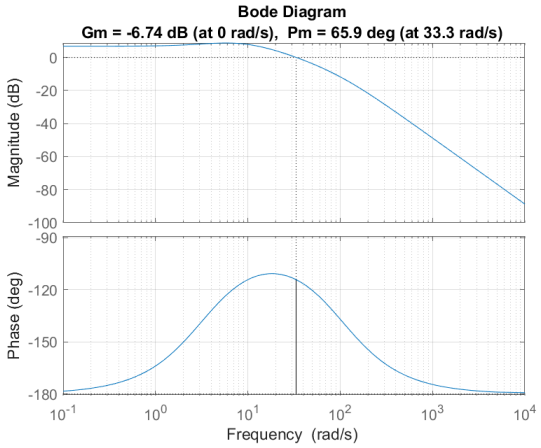


Figure A.2: Tilt Open Loop System Response

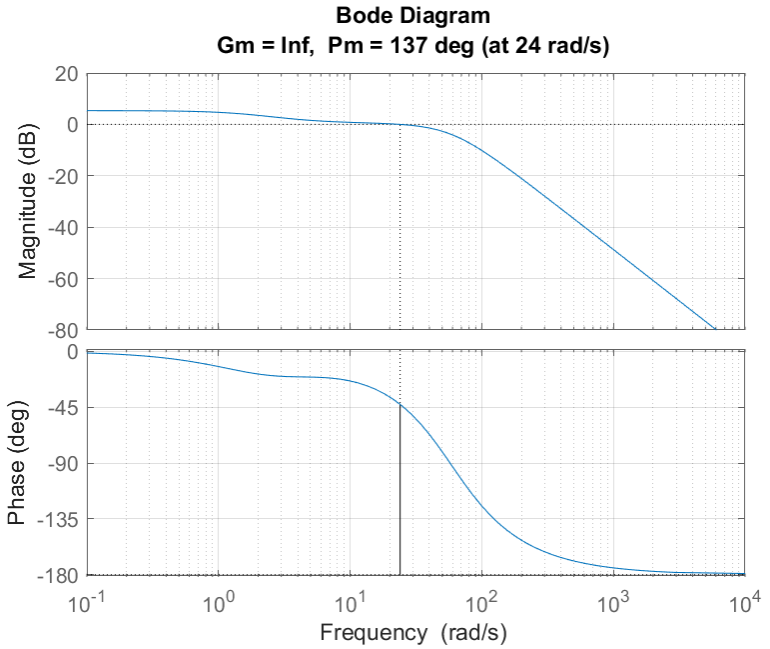


Figure A.3: Tilt Closed Loop System Response

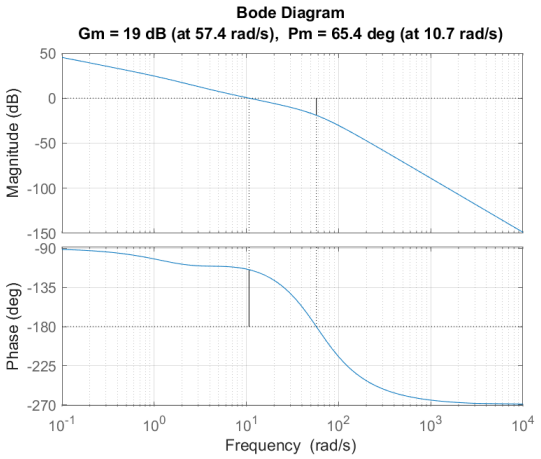


Figure A.4: Velocity Natural System Response

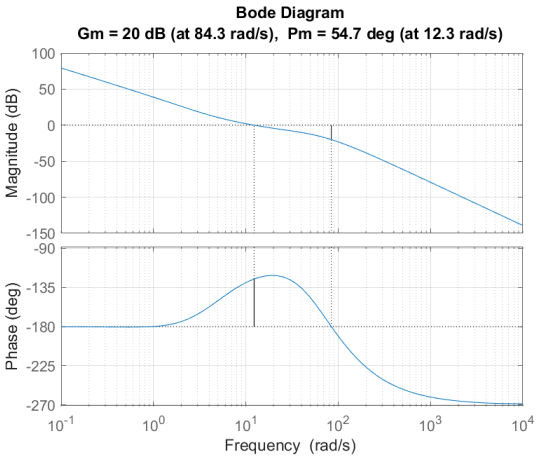


Figure A.5: Velocity Open Loop System Response

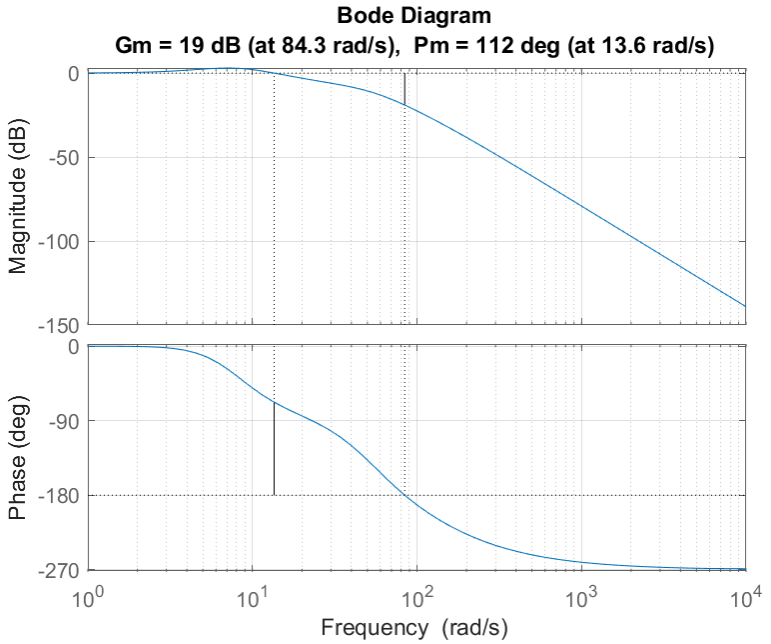


Figure A.6: Velocity Closed Loop System Response

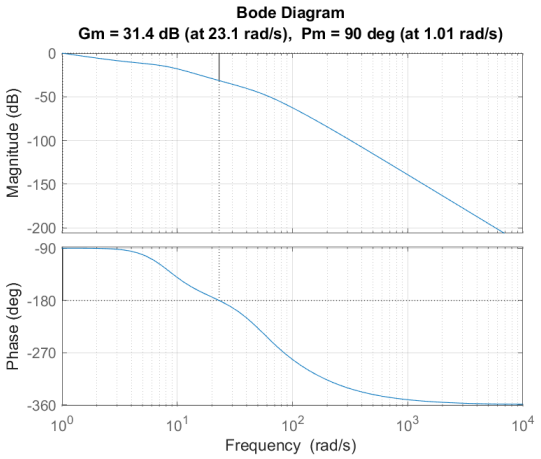


Figure A.7: Position Natural System Response

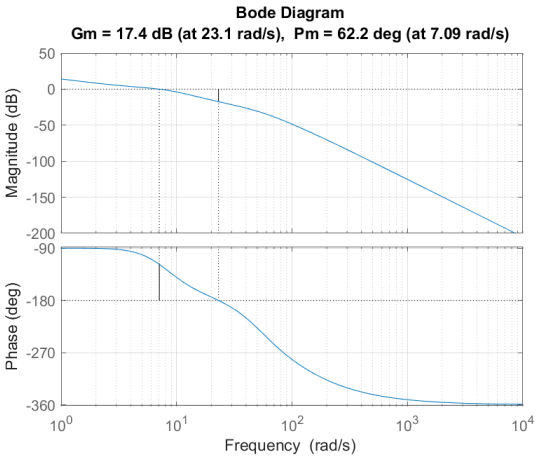


Figure A.8: Position Open Loop System Response

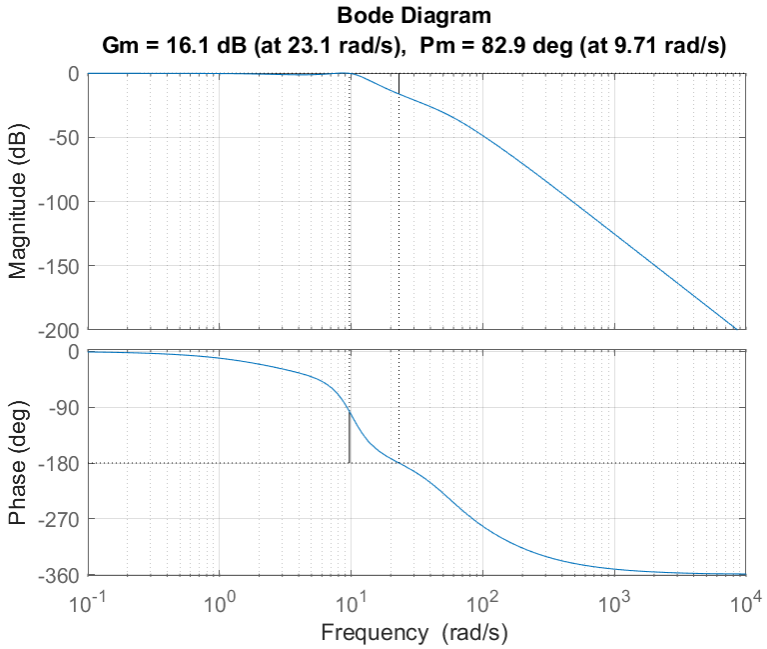


Figure A.9: Position Closed Loop System Response

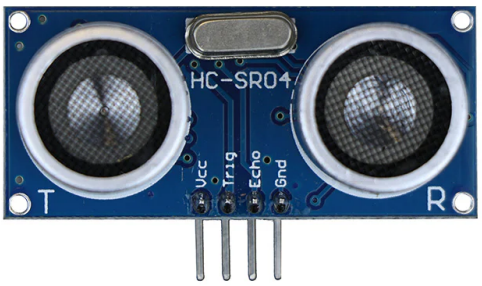


Figure A.10: HC-SR04 Ultrasound Sensor Board [4]

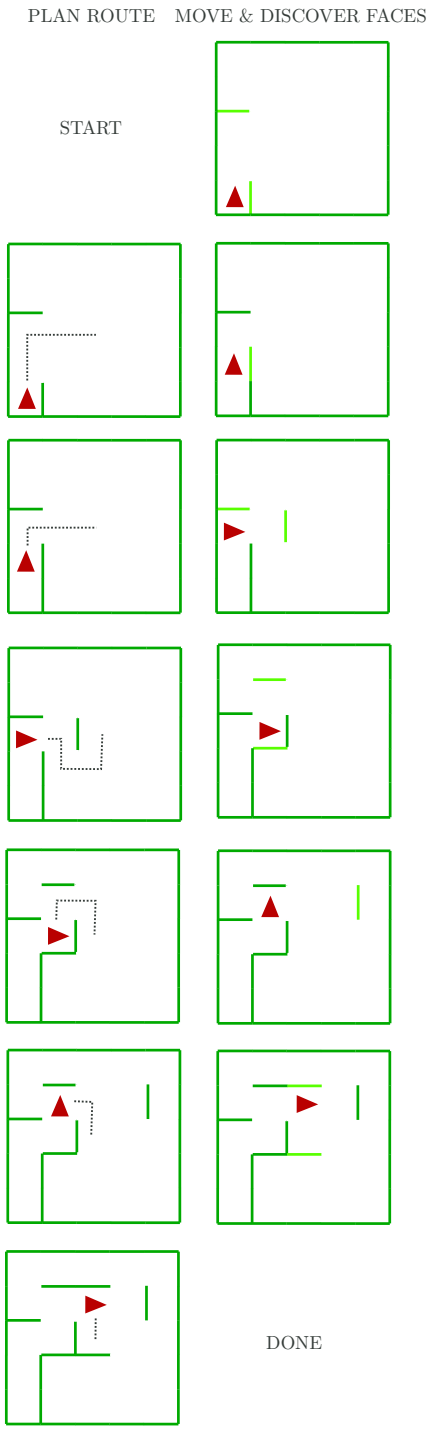


Figure A.11: Flood Fill Algorithm Iterative Solving Demonstration

A.2 Bill of Materials

Item #	Description	Quantity	Price (£)
	Power Monitoring		
1	LMC6484AIN/NOPB Op-amp	1	2.87
2	Fixed Shunt Voltage Reference 1.235V	1	0.60
3	RS PRO PCB Terminal Block	5	3.23
4	STRIPBOARD MEDIUM 95mm X 127mm	1	1.44
		Sub Total:	8.14
	Sensing, Maze Building & Modifications		
1	HC-SR04 Ultrasound Sensor	6	2.1
2	Sensor Mount (3D Printed)	3	from lab
3	T-Juntion Maze Holder (3D Printed)	15	from lab
4	Corner Piece Maze Holder (3D Printed)	15	from lab
5	Side Stand Maze Holder (3D Printed)	15	from lab
6	Robot Cap (3D Printed)	1	from lab
		Sub Total:	12.6
		Total:	£20.74

Table A.2.1: Complete Bill of Materials

A.3 FMEA Analysis

Table A.3.1: FMEA Process Identification and Classification

Project	EngiBeers Balance Robot					
Sub Assembly	Power Management Unit					
Prepared by	Dylan Toussaint					
Approved by	Hector Oga					
Page	1 of 2					
Notes	Power Monitoring SubUnit FMEA					
Process Purpose	Potential Failure Modes	Severity	Potential Causes of Failure	Occurence	Process Control	Detection	RPN
Power Monitoring, Battery Management	Robot Shutdown Due to Low Battery, Fuse Blow causing Robot Shutdown.	7	Short between Vbat and Ground.	7	Short identified through blowing of Fuse.	1	49
	Failure of hardware mount, breadboard	7	Loose connections, short connections on breadboard	3	Hard to identify	5	105
	Component Failure	5	Bent Pins, Fall Damage	4	Physical Inspection, Inspect working with probes	2	40

Table A.3.2: FMEA Recommend Actions and Action Taken

Project	EngiBeers Balance Robot					
Sub Assembly	Power Management Unit					
Prepared by	Dylan Toussaint					
Approved by	Hector Oga					
Page	2 of 2					
Notes	Power Monitoring SubUnit FMEA					
Action Recommended	Area/Person Responsible	Delivery Date	Action Taken	Severity	Occurence	Detection	RPN
Solid Isolation of Vbat and Ground. Testing of power consumption through ADC and bench power supply for validation	Dylan Toussaint	14/06	Solid Isolation of Ground and Vbat	7	2	1	14
Use a secure mounting: stripboard	Dylan Toussaint	14/06	Soldered using stripboard, use of terminal blocks	7	1	3	21
Use as few components as necessary	Dylan Toussaint	14/06	Used quad mounted op-amp instead of multiple chips	7	2	1	14

A.4 Risk Assessment

Risk	Explanation	Mitigation	I	P	S
Burn	Electrical Short circuit, from hot components or traces. Hot Soldering Iron.	Include short-circuit protections (fuse), inspect constructed circuits, never touch circuits while power is being delivered. Use soldering iron carefully, never touch hot soldering iron tip, use soldering iron holder	2	2	4
Fire Risk	Use of NiMH batteries, Use of own batteries, improper charging of batteries, short circuit	Always use the provided power pcb, never use your own batteries. Always ask a lab technician to replace depleted batteries. Provide short circuit protection of battery terminals. Report damaged batteries to lab technician.	5	2	10
Trips and Falls	Spillage of liquids Trailing Cables Inappropriate footwear Poor storage of personal items	Wear appropriate clothing, leave bags and loose items in locker. No drinks should be brought to the lab.	2	1	2
Manual Handling	Use of hand tools, lifting equipment	Wear appropriate footwear in the laboratory. Appropriately handle items or ask for assistance for heavy items. Ask for assistance if unsure on how to use a piece of equipment.	2	1	1
Electric Shock	Exposure to mains, exposure to short circuit from power supply	Always set an appropriate voltage on power supply while powered off, include short circuit protection, never work directly with the mains	4	1	4
Soldering	Exposure to fumes	Use the provided fume hoods when soldering, remain at a safe distance from soldering tip	1	1	1

Impact	Explanation
5	Catastrophic Damage - Risk to Life
4	Major Damage - Risk of Serious Injury
3	Moderate Damage - Risk of Injury
2	Minor Damage - Risk of Small Injury
1	Very Minor Damage - Low Risk

Probability	Explanation
5	Will Happen
4	Very Likely to Happen
3	Moderately Likely to Happen
2	Unlikely
1	Rare

Combined Score	Action Required
>=15	Project may not go ahead unless additional safety precautions are undertaken.
5-10	Project may go ahead, review current precautions
<5	No Action Needed

References

[1] "A Comprehensive and Comparative Study of Maze-Solving Techniques by Implementing Graph Theory," 2010. [Online]. Available: https://www.researchgate.net/publication/224202469_A_Comprehensive_and_Comparative_Study_of_Maze-Solving_Techniques_by_Implementing_Graph_Theory

[2] T. Instruments, "Difference Amplifier (Subtractor) Circuit," 2023. [Online]. Available: https://www.ti.com/lit/an/sboa274a/sboa274a.pdf?ts=1716303606938

[3] O. Batteries, "Product Specification: SubC 2000mAh 1.2V." [Online]. Available: https://www.farnell.com/datasheets/3684490.pdf

[4] K. Ltd, "HC-SR04 Datasheet." [Online]. Available: https://docs.rs-online.com/8bc5/A700000007388293.pdf

[5] M. Online, "Micromouse Competition Introduction." [Online]. Available: https://micromouseonline.com/micromouse-book/introduction/

[6] V. Derek Muller, "The Fastest Maze-Solving Competition On Earth," 2023. [Online]. Available: https://www.youtube.com/watch?v=ZMQbHMgK2rw&t=474s&ab_channel=Veritasium

[7] S. Mishra and P. Bande, "Maze solving algorithms for micro mouse," in *2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*, 2008.

[8] T. Instruments, "LMC648x CMOS Rail-to-Rail Input and Output Operational Amplifiers." [Online]. Available: https://www.ti.com/lit/ds/symlink/lmc6484.pdf?ts=1718448009999&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FLMC6484%252Fpart-details%252FLMC6484AIN%252FNOPB

[9] "Invensense MPU60XX DMP Documentation," 2012. [Online]. Available: https://invensense.tdk.com/products/motion-tracking/6-axis/mpu-6050/

[10] Zephyr Project, "Symmetric Multiprocessing in Zephyr Kernel." [Online]. Available: https://docs.zephyrproject.org/latest/kernel/services/smp/smp.html

[11] ——, "Message Queues in Zephyr Kernel." [Online]. Available: https://docs.zephyrproject.org/latest/kernel/services/data_passing/message_queues.html