Final Report

Micromouse Project

Prepared by:
Emanuele Vichi, Aaron Isserow

Prepared for:
EEE3099S
Department of Electrical Engineering
University of Cape Town

October 22, 2024

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

October 22, 2024

Name Surname

Date

Contents

Chapter 1

Executive summary

1.1 Introduction to Micromouse Project

A micromouse is a small, autonomous robot designed to navigate through a maze in the shortest and most efficient manner possible. It uses various sensors, motors, and algorithms to detect walls, lines, and intersections.

The primary goal of the micromouse is to autonomously solve a maze as quickly and efficiently as possible. This project is often part of engineering and robotics competitions, where participants aim to design and program the most effective maze-solving robot.

1.1.1 Key components

- Sensors: Micromouse robots are equipped with sensors that use infrared (IR) light to detect the prersence or absence of walls, lines and intersections.
- Motors and Wheels: The micromouse moves using small wheels that are controlled by a microcontroller. It must be able to move with precision to turn accurately in tight spaces in the maze.
- Microcontroller: At the core of the micromouse is a microcontroller that processes sensor data and executes the maze-solving algorithm. The STM32 microcontroller is used in the this micromouse robot.
- Maze-Solving Algorithm: The micromouse uses an algorithm to navigate through the maze. It constantly updates its understanding of the maze so that it can solve the maze in the most optional manner possible.

1.2 Problem Statement

The micromouse needs to efficiently find a solution to solve the maze without any prior knowledge of this maze. Maze navigation can be broken up into a few challenges. A few of the key challenges include wall detection, line following, real time decision making and sensor reliability. These factors interact with one another introducing increased complexity to the system.

1.3 Specifications and Requirements

The requirements for the micromouse system are described in Table 1.1.

Table 1.1: Functional requirements of the micromouse system.

Req ID	Description
FR01	Calibration algorithm
FR02	Initialisation routine
FR03	A line identification algorithm
FR04	Wall and cross-section identification
FR05	Decision making algorithm
FR06	Optimization of the micromouse
FR07	Error minimization
FR08	Time minimization

The specifications, refined from the requirements in Table 1.1, for the micromouse system are described in Table 1.2.

Table 1.2: Specifications of the micromouse system derived from the requirements.

Spec ID	Description
SP01	The calibration should be only a few seconds and be able to be performed in high and low light conditions.
SP02	The initialisation routine should be a subset of the calibration routine and set thresholds for each of the 8 IR LEDs.
SP03	The line following algorithm should focus on forward movement and only correct itself if the micromouse is veering off track when both downwards facing LEDs are on the same side of a line.
SP04	The micromouse must be able to reliably sense front and side walls without error. It must also be able to detect intersections and stop when one is detected.
SP05	The micromouse must make informed decisions. When faced with open walls to the left and right, there must be priority to turn left.
SP06	The micromouse should be able to optimize its path through the maze by acquiring information during the first run through the maze.
SP07	The micromouse should never hit a wall, get stuck in an infinite loop, or make the wrong navigation decision.
SP08	The micromouse should always prioritise the most optimal path through the maze.

1.4 Summary of Key Achievements

This is the same table as found in the conclusion of this report. It summarises all of the achievements that the micromouse was able to meet. It relates directly to the specifications listen in Table 1.2.

Table 1.3: Summary of Achievements of the micromouse

Spec ID	Description	Achieved	Comment
SP01	Calibration	Yes	The calibration is quick and successfully works in all ambient conditions
SP02	Initialisation	Yes	Initialisation routine successfully sets the thresholds for all the sensors
SP03	Line Following	Yes	Micromouse is able to follow the line perfectly without veering off.
SP04	Wall and Cross-Section Detection	Partial	Cross-section detection not consistent if approached at an angle. Front wall detection successful. Left and right wall detection is inconsistent due to ambient conditions.
SP05	Maze Solving Algorithm	Yes	Left Wall Follower Logic is successful in solving mazes of simple complexity by following the left wall exclusively.
SP06	Path Optimization	No	Not successful as it was not implemented
SP07	Error Minimization	Partial	The micromouse is able to detect when it has made a mistake and tries to correct itself. If sensors do not detect walls properly it leads to wrong decisions made.
SP08	Time Minimization	No	No algorithm to find optimal path through the maze. Therefore time cannot be improved.

Chapter 2

Design and Implementation of Navigation

2.1 Inroduction to the navigation system

2.1.1 Project objective

The primary objective of this project is to design a micromouse that can autonomously navigate through a maze as efficiently as possible. Navigation in this context refers to the micromouse's ability to determine its position relative to its environment, adjust its movements accordingly, and ultimately reach the end of the maze in an optimal manner.

2.1.2 Navigation system overview

The navigation system integrates sensor inputs, decision-making algorithms, and motor controls. Sensors provide real-time feedback based on the micromouse's environment, delivering essential information such as line detection, wall detection, and intersection identification. Based on this data, the micromouse makes decisions on whether to move forward, turn, or stop. Key components of a well-functioning navigation system include proper sensor calibration and the execution of an effective pathfinding algorithm.

2.1.3 Importance of calibration and pathfinding

This section outlines the techniques and strategies used to design the navigation system. It begins by describing improvements to the calibration routine, which ensure that the micromouse accurately interprets sensor data regardless of environmental conditions. The discussion then details the specific navigation algorithms used to detect walls, lines, and intersections, and how the micromouse makes decisions based on this information. Ultimately, the navigation system seeks to strike a balance between accuracy and efficiency, allowing the micromouse to solve the maze in the shortest time possible (efficiency) without colliding with walls (accuracy).

2.2 Design

Creating an initial flowchart for the navigation system was a critical step in the design process, as it provided clear direction for the subsequent implementation of the MATLAB code.

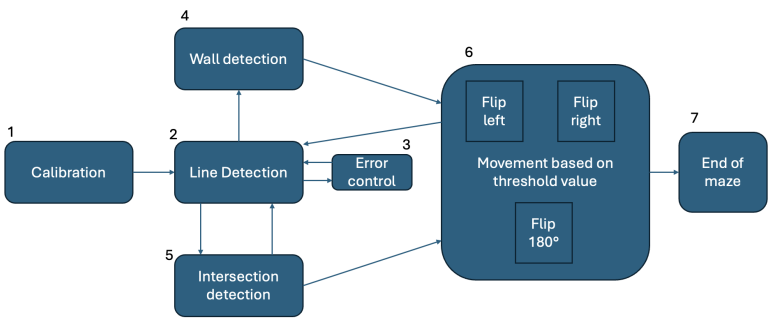

Figure 2.1: Flow-chart for micromouse system

As shown in Figure 2.1, the flowchart visually outlines the sequence of operations and decisions, helping to ensure that the entire process is clearly understood before addressing the complexity of the code. Each block in the flow-chart is elaborated

on below.

2.2.1 Calibration

The first state is the calibration state, where values for each of the eight ADC thresholds are set.

Figure 1 in the appendix shows the arrangement of Infrared (IR) emitters and detectors on the sensing system. Note that the sensing system is flipped upside down when placed on the micromouse, so all LEDs on the left side of this figure are actually on the right side, and vice versa. These thresholds are critical to the navigation system, as nearly all operations depend on them.

2.2.2 Line detection

The line detection system will involve an algorithm that constantly checks the values on minDR and minDL. When these values pass a certain threshold, the micromouse should recenter itself by turning towards the appropriate direction.

2.2.3 Error control

There should be a state in the control loop dedicated to error detection. This state is an unknown state that the micromouse enters only when it comes to a stop and is not in the forward, left turn, or right turn states. In this error state, the micromouse should rotate to a position that allows its sensors to realign, enabling it to re-enter the line-following state.

2.2.4 Wall detection

The wall detection state must check for a wall in front of the micromouse. If there is a wall, the micromouse must make a decision on what direction it should turn in depending on its environment. If there is an opening only to its left, it must turn left. If there is an opening only to its right, it must turn right. However if there is an opening to its left and right, it must always choose to turn left. This design decision will be elaborated on more in the optimization section of this report.

2.2.5 Intersection detection

Furthermore, when the micromouse encounters an intersection, it must look to see if there is an opening to its left, if so, it must turn left. However if this is not true, the code must transition back into the line detecion state and continue moving forward.

2.2.6 End of maze

There must be some sort of counter incorporated into the code that can keep track of consecutive right turns. If there are three right turns in a row, without any left turns or u-turns, it should indicate that the end of the maze has been reached and the micromouse should come to a stop.

2.3 Improved calibration algorithm

2.3.1 Calibration overview

In the current version, shown in Figures 2.2 and 2.3, the calibration process begins by turning on all the IR emitters and initialising all sensor readings to their maximum ADC values of 4095. This ensures that the system starts with the highest possible threshold before the calibration process begins.

2.3.2 Initial calibration

To initiate calibration, the micromouse is placed in the center of a box surrounded by four walls. Once SW1 is pressed, the calibration process starts. During this phase, the micromouse takes readings from the two downward-facing LEDs, which are stored in the variables minDL and minDR, respectively. The same procedure is followed for the forward-facing LEDs and side-facing LEDs, with the readings stored in minFWDL, minFWDR, minL, and minR, respectively. This calibration process takes 3 seconds.

```
Calibration
entry:
LED_MOT_L = true;
LED_MOT_R = true;
LED_FWD_L = true;
LED_FWD_R = true;
LED_R = true;
LED_L = true;
LED_DOWN_L = true;
LED_DOWN_R = true
LED0=true;
minDL=4095;
minDR=4095;
minFWDL=4095;
minFWDR=4095;
minL=4095;
minR=4095;
minML=4095;
minMR=4095;
RTurnCount=0;
end
during:
if(ADC13_DOWN_LS<minDL)
   minDL=ADC13_DOWN_LS
end
if(ADC10_DOWN_RS<minDR)
   minDR=ADC10_DOWN_RS
end
if(ADC14_FWD_LS<minFWDL)
   minFWDL=ADC14_FWD_LS
end
if(ADC9_FWD_RS<minFWDR)
   minFWDR=ADC9_FWD_RS
end
if(ADC12_LS<minL)
   minL=ADC12_LS;
end
if(ADC11_RS<minR)
   minR=ADC11_RS;
end
end
```

Figure 2.2: Calibration for wall detection and line following

2.3.3 Cross-section calibration

After this initial calibration phase, SW2 is pressed to trigger a state change into the 'CrossSection' mode. In this mode, a similar threshold-setting process occurs, but with one key difference: the micromouse is placed with its motor downward-facing LEDs directly over an intersection in the maze. This setup allows for accurate threshold detection specifically for navigating intersections.

```
CrossSection
entry:
LED0=false;
LED1=true;
end
during:
if(ADC8_MOT_RS<minMR)
   minMR=ADC8_MOT_RS;
end
if(ADC15_MOT_LS<minML)
   minML=ADC15_MOT_LS;
end
end
```

Figure 2.3: Cross section calibration

2.3.4 ADC values use as a thresholds

These calibration algorithms place the micromouse in a strong position moving forward with the design of the navigation system. The values that are held in the variables created in the calibration sequence can be estimated using the debug mode. Figure 2.4 and Figure 2.5 show the voltage and corresponding ADC values of the sensors in debug mode when the micromouse is placed in the middle of 4 walls. This process was also performed for cross section calibration to obtain threholds for the motor LEDs. The values are stored in local variables:

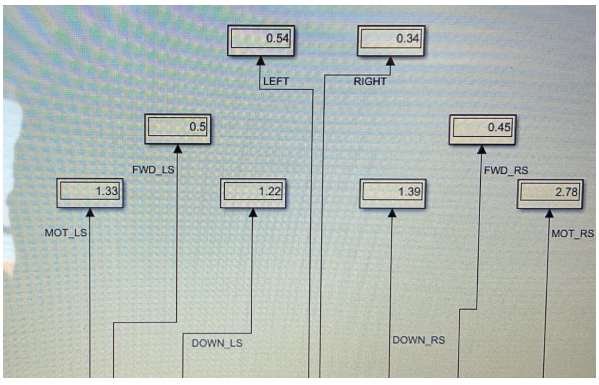

Figure 2.4: Approximate voltage values after calibration

Variable	ADC Value
minDL	1513
minDR	1724
minFWDL	620
minFWDR	558
minL	670
minR	421
minML	1950
minMR	3226

Figure 2.5: Table of Minimum Values

2.4 Line following algorithm

The line following algorithm was adapted since the last report for increased accuracy and reliability. This algorithm is shown in Figure 2 in the appendix. The corresponding ADC values for the left and right downward-facing detectors are minDL = 1513 and minDR = 1724, respectively.

Initially, all on-board LEDs on the micromouse are turned off. These LEDs aid in the design process as they turn on in various combinations depending on the state of the micromouse. The general process used in the line-following algorithm involves comparing the current ADC values from the downward-facing IR detectors with the calibrated threshold values.

It is important to note that when the micromouse is placed with its emitter and detector directly over a line, the ADC value will be low (logic 0). Conversely, when the micromouse is not on a line, the value will be high (logic 1). Initially, the micromouse is positioned on the line. In this state, the values are somewhere between the low and high ranges since the micromouse is not entriely on the line but not entirely off the line. Therefore, this state can be considered a '0.5' state.

2.4.1 Right turn

```
if (ADC13_DOWN_LS <= minDL - 300) && (ADC10_DOWN_RS >= minDR + 300) % turn right
```

A right turn occurs when the downward-facing detector on the left side falls below the calibrated value with a specified offset, and the detector on the right side rises above the calibrated value with a similar offset. The offset in this case involves subtracting 300 from the left side and adding 300 to the right side. This offset is necessary to prevent the micromouse from changing states prematurely when it briefly dips below or rises above a threshold, which would otherwise prevent it from continuing straight. By incorporating the offset, the micromouse remains on a straight path until it veers significantly off track.

When analysing the micromouse in debug mode, with the micromouse offset to the left and requiring a right turn, the values of ADC13_DOWN_LS and ADC10_DOWN_RS are 2891 and 1092, respectively. Thus, ADC13_DOWN_LS >= minDL - 300 and ADC10_DOWN_RS <= minDR + 300. This is however the opposite of what was used in the final code due to the following reason:

It became apparent that the unexpected behavior was due to the output blocks of the ADC being flipped. Specifically, the ADC13 value was actually the ADC10 value, and vice versa. This misconfiguration caused the readings to be swapped, which explained the inverted behavior during testing.

Consequently, both conditions are satisfied in the conditional statement, and the micromouse will make a right turn.

```
if (1092 <= 1513 - 300) && (2891 >= 1724 + 300) % turn right
```

2.4.2 Left turn

```
if (ADC13_DOWN_LS >= minDL + 300) && (ADC10_DOWN_RS <= minDR - 300) % turn left
```

The left turn algorithm is the exact opposite of the right turn algorithm. The ADC values obtained in debug mode, when analysing the micromouse slightly offset to the right (indicating a need to turn left), are as follows: ADC13_DOWN_LS = 682 and ADC10_DOWN_RS = 3375. Thus, when the micromouse transitions from being on the line to requiring a left turn.

The left turn occurs when the current ADC value from the downward-facing detector on the left exceeds minDL with an offset, and the current ADC value from the downward-facing detector on the right falls below minDR with an offset. The offset for both conditions is 300. Once both conditions are met in the conditional statement, the micromouse will execute a left turn. Again the ADC block have been swopped and thus ADC13 is actually ADC10 and visa versa.

```
if (3375 >= 1513 + 300) && (682 <= 1724 - 300) % turn left
```

2.4.3 Go forward

```
    elseif (ADC13_DOWN_LS <= minDL + 400) && (ADC10_DOWN_RS <= minDR + 400) % forward
```

The forward state in the line-following algorithm is the default state that occurs between the left and right turn states. It represents the neutral position between both extreme cases.

2.5 Wall detection

2.5.1 Wall detection overview

The wall detection algorithm is critical for the navigation system since it tells the micromouse when to stop and make a decision on whether to turn left, right or make a u-turn.

The important characteristic that the IR detectors sense in the case of wall detection is that the ADC values increase as the micromouse approaches a wall. The threshold values set in minFWDL and minFWDR is 620 and 558, respectively. However, at the start of the maze, there will be no walls placed in front of the micromouse and thus the ADC values of the front-facing detectors will be lower than these threshold values.

2.5.2 Transition state to detect for the presence of a wall

Initially with no walls present in fornt of the micromouse, the value stored in ADC14_FWD_LS is 198 and the value stored in ADC19_FWD_RS is 174. As the micromouse approaches a wall, this ADC value will begin to rise. Thus a conditional statement is implemented and shown in Figure 2.6:

```
[ADC9_FWD_RS>minFWDR-200&&ADC14_FWD_LS>=minFWDL-200]
%minFWDR=558 and minFWDL=620
```

Figure 2.6: Detection of a wall in front of the micromouse

Once a front facing wall is detected there will be a state transition and 'Mode2' will be entered. In this mode, the micromouse will analyse its surroundings and make a decision on whether to turn left, right or make a u-turn. This algorithm is shown in Figure 3 in the appendix. This state makes use of the left detector and right detector. Specifically, the threshold values stored in minL and minR. There are 4 conditional statements used whereby the current ADC value is compared to the threshold value.

2.5.3 Right wall detection

```
Relevant codntions include: if (ADC12_LS >= minL - 50) and if (ADC12_LS <= minL - 100)
```

It became apparent that two conditions were required to indicate the presence of a wall. The first conditional state was used to set flag 'RWall' to true. Where 'RWall' represents the presence of a wall to the right. This would be counter intuitive since we are checking ADC12_LS but it is importance to remember that all ADC blocks are switched and thus ADC12_LS actually represents the right side. The second conditional state is used to reset these flag back to false if there is no wall present.

2.5.4 Left wall detection

```
Relevant codntions include: if if (ADC11_RS >= minR - 50) and if (ADC11_RS <= minR - 100)
```

The same logic is applied for the presence of a left wall. The same counter intuitive analysis must be done whereby the value stored in ADC11_RS is used to find if there is a wall on the left. The flag in this case also needs to be reset if the value in ADC11_RS drops below the threshold with some offset.

2.5.5 FlipRight algorithm

Figure 2.7 shows the algorithm used for the initialisation for the micromouse to flip to the right. Once the micromouse senses the presence of a wall on the left hand side, it wil flip to the right. Thus the left wheel is set to a speed of 95 and the right wheel is set to a speed of -95. This code will flip the micromouse until the following condition is met in figure 2.8:

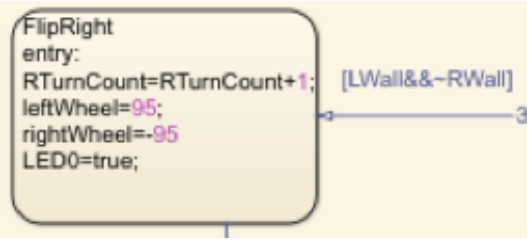

Figure 2.7: FlipRight

Figure 2.8: Condition to stop flipping right

This condition aims to compare the current ADC value of the two front sensors to their respective threshold values. As the micromouse turns to the right, the ADC value of the left detector will begin to decrease and the ADC value of the right detector will also begin to decrease. These values will continuously decrease until the designed thresholds are met and there are no walls in front of the micromouse. At this point code will transition into a temporary wait state and thereafter back into the line following algorithm.

2.5.6 FlipLeft algorithm

Initial flip left algorithm

Figure 2.9 shows the algorithm used for the initialisation for the micromouse to flip to the left.

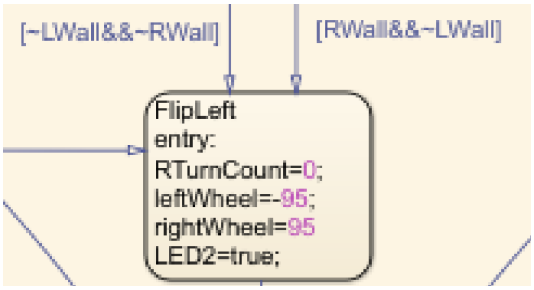

Figure 2.9: FlipLeft algorithm

Once the micromouse senses the presence of a wall on the right hand side or no wall on the left hand side, it will flip to the left. Thus the right wheel is set to a speed of 95 and the left wheel is set to a speed of -95. This code will flip the micromouse until 0.3 seconds has passed. This time was experimentally designed for and was proven to be a successful design decision since the micromouse would always be able to correct itself after flipping for 0.3 seconds. After the micromouse has flipped for 0.3 seconds, it will transition into a temporary wait state and thereafter back into the line following algorithm.

Constant left wall checker

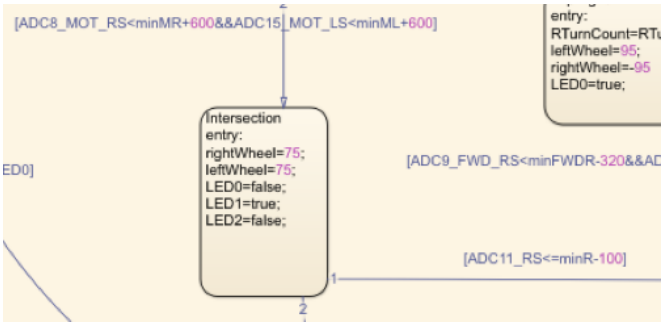

Figure 2.10: Constant left wall checker at intersections

There is one other condition that could be met for the micromouse to flip to its left as seen in Figure 2.10. This state flows from the line following state and checks the value of the motor sensors that are decting intersections. From this intermediary state, the value of ADC11_RS is checked (which actually represnts the left side of the micromouse). If the value of this ADC is less than the thrshold value, thus representing the absence of a wall. The micromouse will enter the FlipLeft state and thereafter make the necessary flip to the left hand side.

2.5.7 FullTurn algorithm

For the final case of a wall being in front, to the left and to the right, the micromouse needs to perform a 180°. This is performed by setting the left wheel to a speed of 95 and right wheel to a speed of -95 as seen in Figure 2.11. The flipping state will stop once the front facing detectors go below the threshold values, thus indicating the absence of a wall in front of the micromouse. Code snippet seen in Figure 2.12.

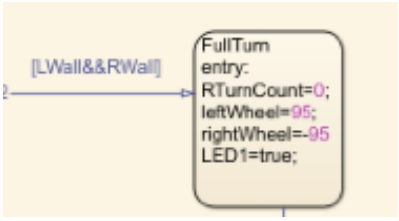

Figure 2.11: A full rotation algorithm

Figure 2.12: Condition to stop turning

2.5.8 Error detection

A simple extension to the line detection state was used for error detection as seen in Figure 2.13. The code works as follows: If the micromouse is in the line following state for more than 3 seconds, it should realize that it is stuck and therefore try to fix itself. It does so by turning to its right. Thus, the speed of the wheel on the left is set to 85 and the speed of the right-hand side wheel is set to -75. Turning to the right proved to be an optimal choice since the micromouse often overturned when turning left and under-turned when turning right. This implementation was very successful in testing.

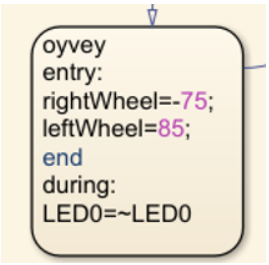

Figure 2.13: Error detection

2.6 Final Results of Navigation

Category	Summary of Performance and Issues
Calibration	The new and improved calibration routine was largely successful. The threshold values were reliable and consistent in most, but not all, categories.
Line Detection	The micromouse excelled at line following, consistently demonstrating that the algorithm was correct and robust. The introduction of the error control state was the final piece that solidified the success of the line-following algorithm.
Cross-section Detection	The motor LEDs were effective in detecting intersections. However, when approaching an intersection at a slight angle, the micromouse sometimes failed to detect it, leading to missed intersections. This issue could result in inefficiencies as the micromouse might repeat paths instead of making the correct turn initially.
Wall Detection	This was the most problematic section during implementation and testing. While the micromouse successfully detected walls in front, it often failed to detect side walls. Ambient light interfered with the side sensors, which were further from the walls than the front and downward-facing LEDs. As a result, the micromouse made incorrect 180° turns when it should have turned correctly, leading to confusion.

Table 2.1: Performance Summary of Key Micromouse Functions

Chapter 3

Design and Implementation of Optimization

3.1 Introduction to Optimization

The micromouse is now able to navigate through the maze, being able to detect its surroundings and identify open paths. The next step is for the micromouse to solve the maze and optimize its navigation path. The aim is to improve the overall performance of the micromouse by finding ways to reduce the time and resources needed to finish the maze. This will involve finding a suitable maze solving strategy and ensuring that it can do so reliably. Due to the time constraints of the project deadline, only a maze solving algorithm was implemented successfully. A working optimization was not able to be implemented and tested effectively. The procedure and potential code structure will still be analysed and explained in the Software Implementation Section below in great detail.

3.1.1 Optimization Goals and Potential Limitations

The optimization goal in the micromouse project focuses on improving the robot's ability to efficiently navigate the maze, with the main objectives being speed, resource efficiency, and reliability. The robot must solve the maze without prior knowledge and make real-time decisions based on its surroundings to find the most efficient path. Therefore, the path-finding strategy must be optimized to avoid repeating movements or backtracking through already visited sections.
This optimization faces both software and hardware limitations. The main hardware issue is sensor accuracy. The sensors are highly sensitive to ambient light changes, affecting real-time readings. While real-time calibration helps mitigate this, it remains a factor that can impact the micromouse's decision-making during navigation. Software limitations include the need for a robust algorithm that can handle unexpected situations, such as avoiding infinite loops caused by sensing errors, and adapting to complex mazes with multiple routes, which may require more advanced algorithms to find the most optimal path.

3.2 Optimization Techniques

The micromouse competition has been around for many years and various algorithms have been utilised and thus a lot of documentation is readily available to research. There are three algorithms which stand out as the most popular amongst all, mainly the simple Wall Follower Logic, Djikstra's Algorithm and Flood Fill Algorithm. These three will be briefly introduced below with a summary of what the algorithm entails.

3.2.1 Wall Follower Logic

This algorithm involves following either the left or right wall exclusively (typically left) until the center of the maze is reached. For a left wall follower, the logic is as follows: At each intersection, if the left wall is open, turn 90° left and continue. If the left wall is present, move forward. If a front wall is detected, check the right wall. If the right is open, turn 90° right, otherwise turn 180°and continue. The priority of movements is: left turn, move forward, right turn, full turn.

Advantages and Disadvantages of Wall Follower Logic

Advantages	**Disadvantages**
Simplicity: Easy implementation in terms of software and does not require complex calculations or extensive memory.	Lack of Optimal Path Finding: Often results in sub-optimal paths, taking longer to execute, especially in complex mazes.
Low Resource Usage: Efficient in terms of computational power and memory.	Limited Applicability: Can only solve mazes without isolated walls, as following one wall can lead to an infinite loop.

Quick decision making: Only requires a wall sensor check, allowing fast decisions.	No Global Awareness: Does not map the entire maze, leading to inefficient exploration based solely on local surroundings.
	No End Detection: Provides no way of determining when the end of the maze has been reached.

Table 3.1: Advantages and Disadvantages of Wall Follower Logic

3.2.2 Djikstra's Algorithm

This algorithm represents the maze as a graph, with nodes as intersections and edges as paths, each with a value indicating the distance or time to navigate. The micromouse starts by setting its distance to zero, marking all other nodes as infinitely far. It then senses the closest node, explores it, and updates distances. If a new calculated distance is shorter, the algorithm adjusts accordingly. This process continues until all nodes are explored or the end is reached. The algorithm then holds the shortest path to the maze's center, which can be traced back to map the optimal route for future runs.

Advantages and Disadvantages of Dijkstra's Algorithm

Advantages	**Disadvantages**
Guaranteed Optimal Path: The algorithm is designed to find the shortest distance between the starting point and the end.	High Time Complexity: The algorithm can have slow computing times depending on the number of nodes (intersections) and edges (paths), as it continuously updates itself and recalculates the shortest path.
Wide Range of Maze Complexities: Due to full exploration and mapping, this algorithm can handle any type of maze.	Memory-Intensive: Requires storing all nodes and edges, which can be taxing for large mazes.
	Full Exploration: The algorithm explores the entire maze to ensure the shortest path, which can lead to unnecessary exploration and increased navigation time.

Table 3.2: Advantages and Disadvantages of Dijkstra's Algorithm

3.2.3 Flood Fill Algorithm

In this algorithm, the maze is treated as a grid where each cell represents either an open space or a wall. Starting from its initial position, the flood fill algorithm explores reachable areas by marking the current cell as visited and moving to all open, unvisited adjacent cells. This continues until all open spaces are explored. As the micromouse moves, it stores information about accessible paths. This data is then used to map a route from start to finish, ensuring no path is repeated and no dead ends are taken.

Advantages and Disadvantages of the Flood Fill Algorithm

Advantages	**Disadvantages**
Wide Range of Maze Complexities: The flood fill algorithm can be easily adapted to any maze, as it fully maps out the entire maze regardless of its layout.	Memory-Intensive: Consumes large memory to keep track of all visited cells, especially in large mazes.
Efficient Path Generation: With full maze mapping, the flood fill algorithm ensures that the path to the end avoids dead ends and does not repeat any path.	Time Inefficiency: The algorithm explores every unvisited cell, which can be time-consuming in large or complex mazes.
	Lack of Optimal Path Finding: While it identifies all reachable areas, the flood fill algorithm does not guarantee the shortest path to the end.

Table 3.3: Advantages and Disadvantages of the Flood Fill Algorithm

3.2.4 Optimization Algorithm Implemented

For this micromouse project, the Left Follower algorithm will be used. This is based on the fact that the maze that will have to be solved is not complex and therefore will be solvable using a simple left wall follower logic. The left wall follower was

chosen based on the time constraints that prevented successful implementation of a more complex navigation strategy. A more advanced approach would need significantly more software implementation and extensive testing to ensure that the code is reliable. Given the project's strict deadline, the left follower offered a simple yet achievable solution which allows for effective navigation and maze solving capabilities within the time frame.

Mitigating Disadvantages

In the context of this project the concerning disadvantages of a left wall follower logic are the following three: Lack of Optimal Path Finding, No Global Awareness and No End Detection. Since the optimal path cannot always be found using a left wall follower, the only thing that can be done to optimize the path to the end is prevent the micromouse from repeating a path that it has already taken due to having encountered a dead end. This problem can be fixed by introducing some form of maze awareness, which will also in turn help with the other two disadvantages. By making the mouse store basic information about the path that it is taking when following the left wall (such as how many times a certain cell has been visited, or which walls are present) then the micromouse could be made to use this information to keep this information after an initial run through the maze and on its next run use it to neglect paths that were visited multiple times as they lead to a dead end or even detect when the ending is reached due to the unique configuration of the end block.
This can be done by having the micromouse keep a MxN array of the maze, where M is the number of rows and N is the number of columns. At every intersection, the information will be stored at the corresponding array index, starting from [0][0] and incrementing either the row or the column index based on the decision taken. Further explanation will be given in the Software Implementation section below.

3.3 Software Implementation

The code that was implemented to realise the Left Wall Follower Logic will be shown and explained below. The optimisation routine introduced in the Mitigating Disadvantages Subsection above was not developed into the final MATLAB code. The exact implementation technique and procedure will be thoroughly explained below.

3.3.1 Code Implementation of Left Wall Follower Logic

In order to have the micromouse follow the left wall exclusively, the following algorithm steps were considered:

1. Enter Forward Line Following mode until the next intersection is detected.
2. At intersection, sense the left wall.
3. If left wall present, go to 5.
4. If left wall not present, turn 90° left and return to 1.
5. Sense front wall.
6. If front wall present, go to 8.
7. If front wall not present, return to 1.
8. Sense right wall.
9. If right wall present, turn 180° and return to 1.
10. If right wall not present, turn 90° right and return to 1.

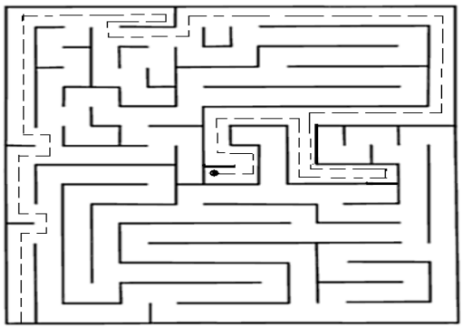

Figure 3.1: Left Wall Follower Logic example maze

This implementation will prioritise turning left as soon as the micromouse senses an opening on the left, if left is not available then the next priority is continuing forward. If forward is not available either, it will turn right. If it is a dead end, then it will do a full turn and find the next left turn available.
A visual representation of this algorithm can be seen in Figure 3.1. As seen below, the micromouse will stick to the left wall completely until the end is found.

The code implementation of this algorithm in MATLAB Stateflow can be seen below in Figure 3.2.
From the code, it can be seen that the micromouse starts in the **Forward Line Following** state. While it is moving forward and following the line, it constantly checks if the left and right MOT sensors detect a value below the threshold, indicating a cross on the floor (i.e., an intersection).
When an intersection is detected, the micromouse leaves the **Forward Line Following** state and enters the **Intersection** state. In this state, it checks the left sensor to determine if the ADC value is below the threshold. If the value is below the threshold, there is no wall present on the left, and the micromouse transitions to the **Flip Left** state, where it turns 90° to

the left, then returns to the **Forward Line Following** state. If the ADC value is above the threshold, a wall is present, and the micromouse transitions directly back to the **Forward Line Following** state.
While in the **Forward Line Following** state, the micromouse also checks the front-facing sensors. If the front sensor ADC value exceeds the threshold, a front wall is detected, and the micromouse transitions to the **Sense Front Right** state. In this state, it checks the left sensor to see if an opening on the left was missed and then checks the right sensor for a wall on the right. If there is no wall on the left, regardless of there being a wall on the right, the micromouse enters **Flip Left** state and turns 90° to the left and returns to the **Forward Line Following** state. If there is a wall on the left and no wall on the right, the micromouse enters **Flip Right** state and turns 90° to the right and returns to the **Forward Line Following** state. If there is a wall on the left and a wall on the right, the micromouse enters **Full Turn** state and turns a full 180° and then returns to **Forward Line Following** state.

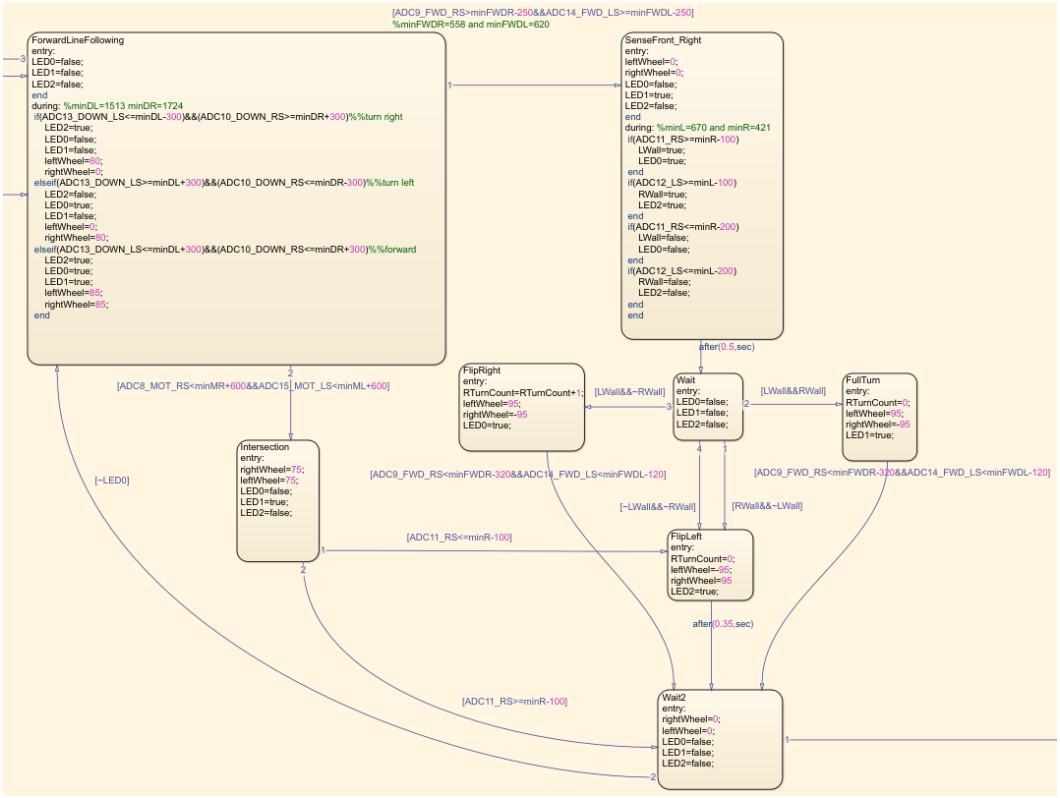

Figure 3.2: Code Implementation of Left Follower Logic in MATLAB Stateflow

3.3.2 Code Implementation of End Detection

Since a Left Wall follower does not have any global awareness, it cannot automatically detect when it has reached the end block. In order to fix this issue it is important to first note that the end block is always a 2x2 block with only one entrance as seen in the following representation, Figure 3.3. Due to the left following it means that no matter how the micromouse enters the end block it will end up turning right three times in a row, before leaving the end block. Taking this into consideration with the fact that the micromouse is most likely to turn left, if a counter is incremented every time a right turn is taken and reset every time a left turn is taken, then if the counter reaches a value of three it must mean that the micromouse has entered the final block.

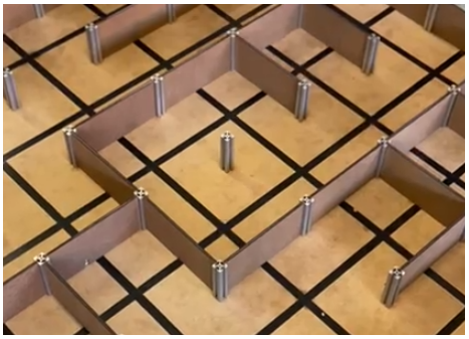

Figure 3.3: Typical 2x2 End Block for the micromouse maze with only one entrance

The code that represents this can be partially seen in Figure 3.2, as it can be seen that in the **Flip Right** state the RTurnCount is incremented by one and in the **Flip Left** state the RTurnCount is reset to a value of zero. The other part of the code that implements the stop can be seen in Figure 3.4. The condition to enter the **End Block Reached** state is that RTurnCount reaches a value of three. The condition is checked in the **Wait2** state after a turn is completed, before returning to **Forward Line Following** state. When entering **End Block Reached** state, the micromouse begins to spin and the three front LEDs are made to blink on and off to signal to the user that the end block has been reached.

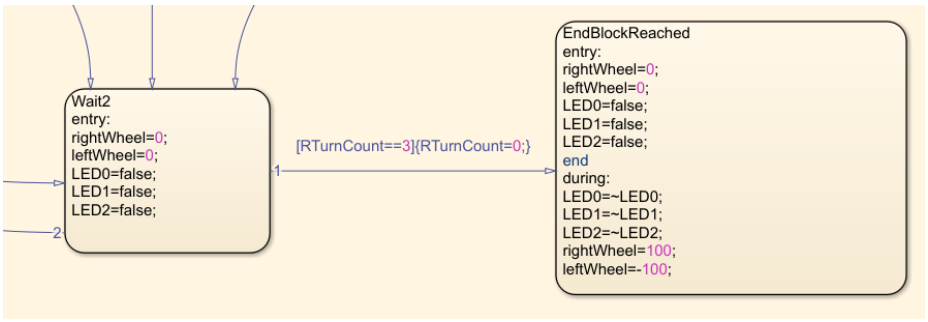

Figure 3.4: Code Implementation of End Block Detection

3.3.3 Theoretical Implementation of Optimization Techniques

In order to make the micromouse aware of its position relative to the maze, the maze is assumed be an array of 7 rows and 7 columns where the starting cell is at row 0 and column 0. If the maze array is denoted by M, then each cell in the maze can be referenced by M[R][C]. Row and Column variables can be created to hold the previous, current and next position of the micromouse in the maze, these variables are R1 and C1, R2 and C2, R3 and C3 respectively. For example, the cell M[R2][C2] corresponds to cell where the micromouse is currently located inside the maze. The variables are changed after a decision is made and the micromouse has moved ahead.

Defining Directions

Since the micromouse needs to be aware of the direction it is facing, the following relationships between the previous, current and next row and column variables can be made:

- If the difference between the current and the previous row indexes is more than 0, thus R2 - R1>0, then the micromouse is moving straight upwards (or North) through the maze.
- If R2 - R1<0, then the micromouse is moving straight downwards (or South) through the maze.
- If the difference between the current and the previous column indexes is more than 0, thus C2 - C1>0, then the micromouse is moving to the right (or East) through the maze.
- If C2 - C1<0, then the micromouse is moving to the left (or West) through the maze.

Changing the array index during navigation

In order to store meaningful information in the array, the micromouse needs to know how to update the index depending on what choice it decides to take in the maze. The micromouse has the following movement options: left turn, move forward, right turn, full turn. These movements can be done in any of the 4 directions: upwards (North), downwards (South), rightwards (East) or leftwards (West). This results in 16 different ways of updating the index numbers. This can be simplified if the fact that only forward movements are done, the only thing that changes is the direction that the micromouse is facing. Thus the options are tabulated below in Table 3.4.

Movement Type	**Upwards (North)**	**Downwards (South)**	**Rightwards (East)**	**Leftwards (West)**
Straight Movement	$R_3 = R_2 + 1$ $C_3 = C_2$	$R_3 = R_2 - R_1$ $C_3 = C_2$	$R_3 = R_2$ $C_3 = C_2 + 1$	$R_3 = R_2$ $C_3 = C_2 - 1$

Table 3.4: Movement Equations for Different Directions

Only one of these is performed depending on the direction that the micromouse is facing. The micromouse will have to store the direction it is facing and update it depending on the decision made. This can be done by first creating a direction variable and a possible directions array:

```
    direction = 'North'
    possibleDirections = ['North', 'East', 'South', 'West']
```

Then depending on the action performed ('Move Forward', 'Turn Right', 'Turn Left', 'Full Turn') the following pseudocode shows how the directions will be updated:

```
FUNCTION updateDirection(currentDirection, action)
    // Find the index of the current direction in the 'possibleDirections' array
    currentIndex = index of currentDirection in possibleDirections //Eg 0 = 'North', 1 = 'East' ...
    // Perform action based on input
    IF action == 'Move Forward'
        newDirection = currentDirection // Moving forward doesn't change the direction
    ELSE IF action == 'Turn Right'
        newIndex = (currentIndex + 1) MOD 4 // Move to the next direction clockwise
        newDirection = possibleDirections[newIndex]
    ELSE IF action == 'Turn Left'
        newIndex = (currentIndex - 1) MOD 4 // Move to the previous direction counterclockwise
        newDirection = possibleDirections[newIndex]
    ELSE IF action == 'Full Turn'
        newIndex = (currentIndex + 2) MOD 4 // Move two positions (180 degrees turn)
        newDirection = possibleDirections[newIndex]
    END IF
    RETURN newDirection // Return the updated direction
END FUNCTION
```

So depending on the output of this function, one of the index operations shown in Table 3.4 will be performed when the micromouse returns to **Forward Line Following** state.

Storing data in the maze array

Now that the indexes are being updated correctly everytime the micromouse moves one cell in the maze, the type of information stored needs to be discussed. In each value of the array meaningful information about each visited cell can be stored. The proposed system stores a 4 bit value, where each bit represents one of the following information:

- Bit 1: '1' if right wall present, '0' if no right wall.
- Bit 2: '1' if left wall present, '0' if no left wall.
- Bit 3: '1' if front wall present, '0' if no front wall.
- Bit 4: '1' if cell visited twice, '0' if cell visited once.

By storing this information the micromouse can remember what the path looked like. This will help it to potentially remove every cell that was visited twice from its next run through the maze as it must have lead to a dead end. Or by analysing the pattern of wall presences it would be able to detect the end block more efficiently.

Unfortunately this implementation was never coded and flashed onto the micromouse due to time constraints, thus it is only a theoretical approach.

3.4 Left Wall Follower Logic Limitations

The Left Wall Follower has a major disadvantage, this being the fact that it can only solve mazes of a certain style. Looking at the maze example in Figure 3.1, it can be seen that by following the left wall, the end can be reached consistently. This is because the left wall is connected to the walls around the end block. If the end block was separated from the outside perimeter the left wall following would never lead to the end and rather into an infinite loop of following the outside perimeter. This can be seen in the following maze example, Figure 3.5. The algorithm is not efficient enough to solve mazes of high complexities and the ones that have multiple paths leading to the end.

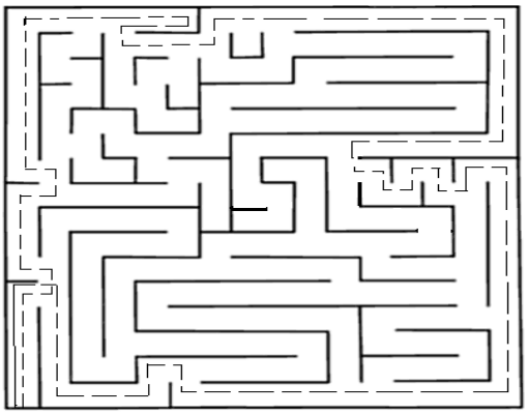

Figure 3.5: Example maze that cannot be solved with Left Wall Following

3.5 Testing and Results

The testing for the maze solving algorithm was based on the left wall following logic and the optimization testing was not conducted due to the fact that it was never successfully implemented to the micromouse itself.

3.5.1 Left Wall Follower Logic Testing

The left wall follower logic algorithm is easily testable. The only criterion for its success is the ability of the micromouse to follow the left wall completely until it detects it is inside the end block and stops. This is entirely dependent on the left and right wall sensing. Due to the inconsistency of the readings of the left and right wall sensors, the left wall following algorithm suffered. This is because if a left wall is not sensed properly due to ambient interference, it will miss the left wall, thus failing the left wall following and potentially not reach the maze or get stuck in an infinite loop.
When the sensors were not affected by ambient light, the micromouse was placed inside of a maze that is solvable by left wall following, such as the illustration in Figure 3.1, and its effectiveness was measured and evaluated. The results are tabulated in Table 3.5.
The micromouse wall following algorithm was also tested inside of a maze that is not solvable by left wall following, such as the illustration in Figure 3.5, and its effectiveness was measured and evaluated. The results are tabulated in Table 3.5.

3.5.2 Optimization Testing

The optimization was not tested inside of the maze. The code was never finalised and thus it is just a theoretical approach at how the optimization would have been implemented to the micromouse if more time was available.

3.5.3 Final Results of Maze Solving and Optimization

Category	**Summary of Performance and Issues**
Maze Solving in Solvable Left Wall Following Maze	The micromouse was able to reach the end of the maze by following the left was exclusively during testing. This proves that the algorithm of priority turning left was implemented correctly. The micromouse was not able to perform as expected during the demonstration session and wasd not able to detect and follow the left wall. This issue rose from the inaccuracies and ineffective reliability of the wall detection sensors to provide the correct readings due to the effects of surrounding ambient conditions, as discussed in the results of the Final Results of Navigation in chapter 2.
Maze Solving in Non-Solvable Left Wall Following Maze	The micromouse was not able to solve the maze due to the fact that the left wall does not connect to the end block. This results in the micromouse going into an infinite loop, never being able to reach the end.
End Block Detection	When the micromouse was able to detect the left walls correctly and successfully reach the end block, after three right turns it stopped successfully. This result was not consistent due to the inaccuracies in the wall detection of the sensors.
Optimization	The optimization was not successful due to the fact that it was not implemented.

Table 3.5: Performance Summary of Maze Solving and Optimization implementations

Chapter 4

Conclusion

The final remarks regarding the achievements and challenges that the micromouse experienced will be summarised and explained below.

4.1 Summary of Achievements

The ability of the micromouse to meet the requirements and specifications in both the navigation and the optimization sections was tabulated below in Table 4.1.

Table 4.1: Summary of Achievements of the micromouse

Spec ID	Description	Achieved	Comment
SP01	Calibration	Yes	The calibration is quick and successfully works in all ambient conditions
SP02	Initialisation	Yes	Initialisation routine successfully sets the thresholds for all the sensors
SP03	Line Following	Yes	Micromouse is able to follow the line perfectly without veering off.
SP04	Wall and Cross-Section Detection	Partial	Cross-section detection not consistent if approached at an angle. Front wall detection successful. Left and right wall detection is inconsistent due to ambient conditions.
SP05	Maze Solving Algorithm	Yes	Left Wall Follower Logic is successful in solving mazes of simple complexity by following the left wall exclusively.
SP06	Path Optimization	No	Not successful as it was not implemented
SP07	Error Minimization	Partial	The micromouse is able to detect when it has made a mistake and tries to correct itself. If sensors do not detect walls properly it leads to wrong decisions made.
SP08	Time Minimization	No	No algorithm to find optimal path through the maze. Therefore time cannot be improved.

4.2 Challenges and Limitations

The challenges faced during the micromouse project consisted of the sensitivity of the sensors when exposed to the ambient lighting affecting the ADC readings. These would affect the wall detection and potentially lead to mistakes in navigation through the maze as it did during the demonstration session. Due to the micromouse not detecting left and right openings, it continued to do full turns, only going up and down without ever solving the maze.
The other major issue faced was the time constraint. The troubleshooting and compiling of the code and dealing with the inaccuracies of the sensors lead to very little time being available to develop and implement a working optimization algorithm which is able to find the optimal path through the maze. Research was done on optimization techniques and a proposed implementation was described but it was never tested.

4.3 Improvements

A potential improvement is to develop a more robust calibration and sensor value acquisition method and code implementation. This could be in the form of flashing the IR LEDs at different intervals and capturing the difference between the readings when the IR LED is ON versus when it is OFF to determine and potentially mitigate the effect of the ambient light.
Additionally, utilizing the on-board gyroscope to get more precise 90° and 180° turns, enhancing the micromouse's ability to navigate corners with better accuracy.
Another improvement would be the development and implementation of a more complex maze solving algorithm with global mapping functionality, such as a Djikstra's Algorithm or a Flood Fill, so that more complex mazes could be solved with a more optimized path and thus achieve much faster times.

Chapter 5

Report Work Split

#	Section Title	Person that worked on it
1	Executive summary	ISSAAR001 & VCHEMA001
2	Navigation	ISSAAR001
3	Optimization	VCHEMA001
4	Conclusion	VCHEMA001

Table 5.1: Work Split for Report

.1 Appendix A: LED configuration on the micromouse sensing system

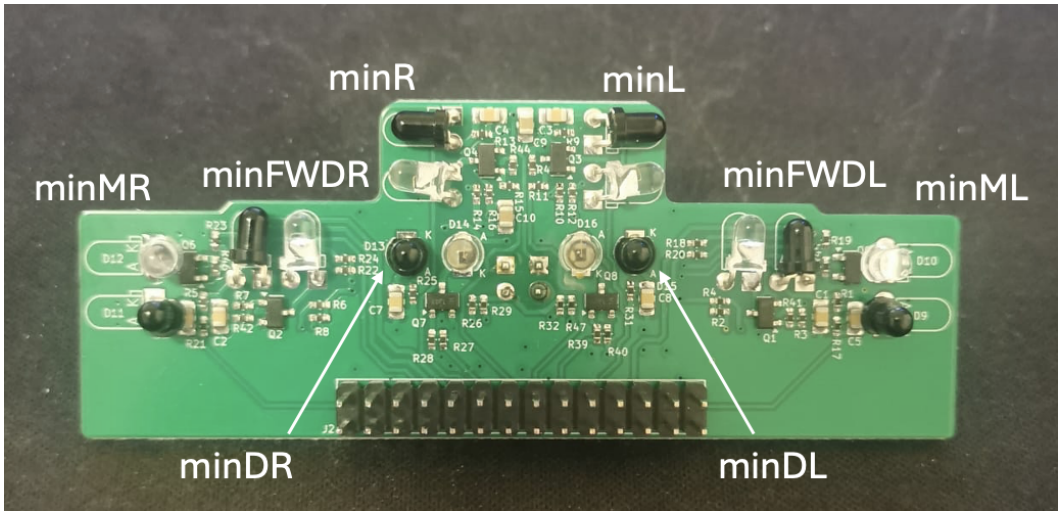

Figure 1: LED configuration on sensing system

.2 Appendix B: Line following algorithm

```
Mode1
entry:
LED0=false;
LED1=false;
LED2=false;
end
during: %minDL=1513 minDR=1724
 if(ADC13_DOWN_LS<=minDL-300)&&(ADC10_DOWN_RS>=minDR+300)%%turn right
    LED2=true;
    LED0=false;
    LED1=false;
    leftWheel=95;
    rightWheel=85;
 elseif(ADC13_DOWN_LS>=minDL+300)&&(ADC10_DOWN_RS<=minDR-300)%%turn left
    LED2=false;
    LED0=true;
    LED1=false;
    leftWheel=85;
    rightWheel=95;
 elseif(ADC13_DOWN_LS<=minDL+400)&&(ADC10_DOWN_RS<=minDR+400)%%forward
    LED2=true;
    LED0=true;
    LED1=true;
    leftWheel=85;
    rightWheel=85;
 end
```

Figure 2: Line following algorithm

.3 Appendix C: Wall detection algorithm

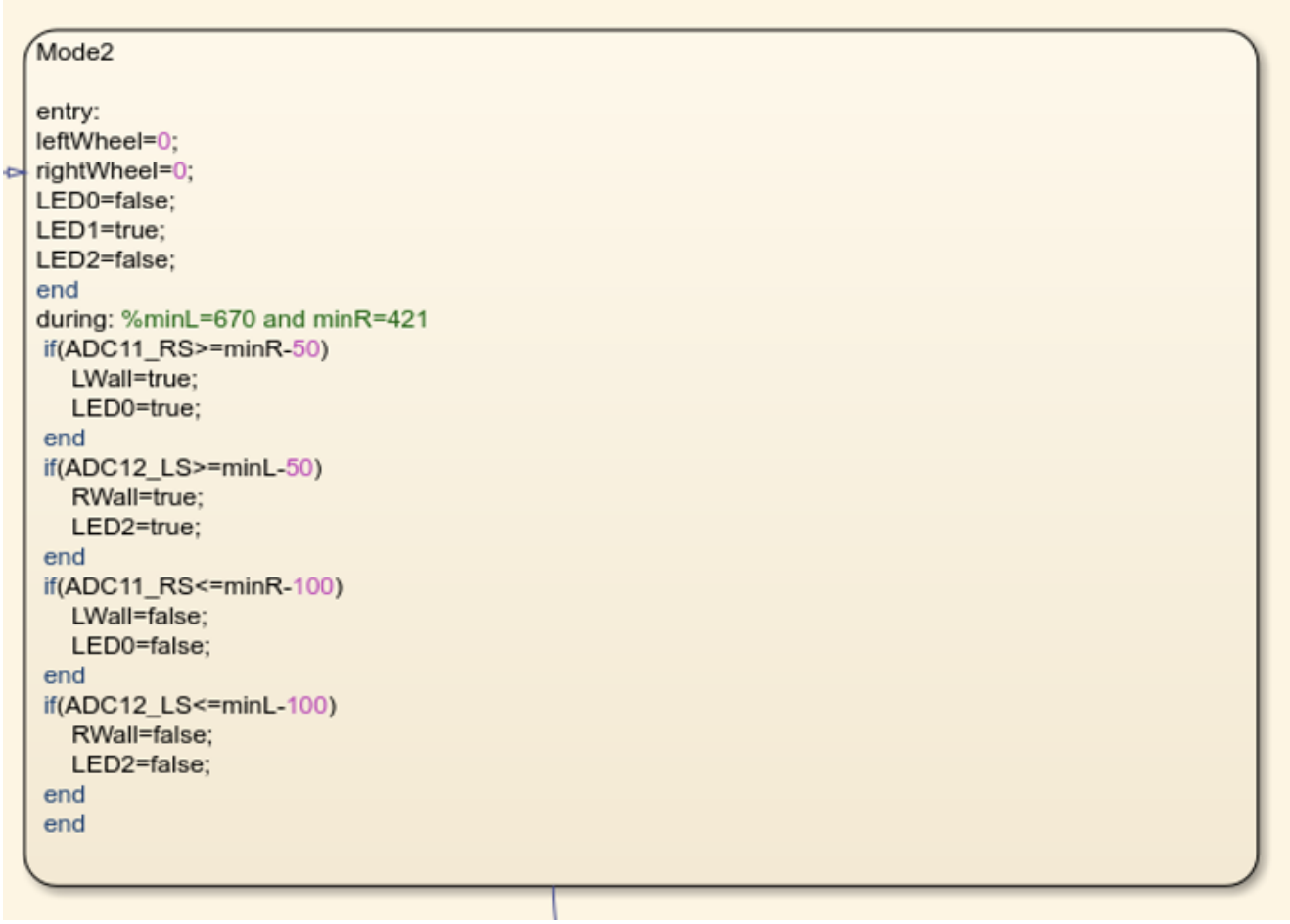

Figure 3: Wall detection algorithm