Final Milestone Design Report

Micromouse Software Design

Prepared by:

Rachael Guise-Brown and Heinrich Crous

Prepared for:
EEE3099S

Department of Electrical Engineering

University of Cape Town

PLAGIARISM 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. 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.
4. The sections were authored as detailed below
    a. Rachael Guise-Brown
        i. Introduction
        ii. Optimization
        iii. Conclusion
    b. Heinrich Crous
        i. Executive Summary
        ii. Navigation

Name: Rachael Guise-Brown
Signed:

Date: 24 September 2024

Name: Heinrich Crous
Signed:

Date: 24 September 2024

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This report presents the final progress and results for the micromouse project in EEE3099S, focusing on milestones three and four. The objective of the project is to develop a micromouse capable of solving a maze autonomously. Milestone one covered the design and implementation of the power and sensing modules for EEE3088F, as well as the initial hardware setup for the standardized mouse used in EEE3099S. In milestone two, the focus was directed toward basic movement and sensor integration.

Milestone Three: Navigation

The focus of milestone three is to use the progress and information from milestone two to move, decide, and make turns. From milestone two, the micromouse must still have a calibration, an initialization sequence, and surrounding awareness, and it must be able to identify and hold a line. Our understanding for this milestone is that the micromouse must be able to successfully move through the maze; however, it does not need to be able to move through the maze optimally. This is broken down into smaller objectives:

1. The micromouse must be able to stop at intersections.
2. The micromouse must be able to sense the difference between when there is a wall and when there is no wall at its right, left, and front.
3. The micromouse must be able to turn at intersections and continue following the line after it has turned.
4. Logic is required so that the micromouse does not get stuck in a loop and explores new areas of the maze and so that the micromouse knows when it has reached the end of the maze.

Navigation Implementation

The design can be implemented using Simulink's stateflow software and MATLAB code. The main stateflow consists of various blocks like standby, calibration, and navigation. The code was split into seven sub-blocks or modules which were all implemented inside the navigation block. The micromouse continuously checks whether it has reached an intersection while it is in motion. At each intersection, the micromouse first moves forward slightly to position itself favorably to turn if possible. The micromouse uses its left sensor to check whether there is a wall present on its left. If there is no wall present, the micromouse is free to turn left. If there is a wall present, the micromouse continues moving forward. If the micromouse reaches a wall, it stops and moves backward slightly. If there is a wall on its left, it turns right. If there is no wall present on its left, it turns left. The micromouse then proceeds to move forward.

Milestone Three Achievements

The micromouse successfully achieved basic navigation. The mouse was able to stop at intersections, identify walls, and make accurate turns. The micromouse was also able to avoid getting stuck in loops and could reliably detect the end of the maze.

Milestone Four: Optimization

The focus of milestone four was to optimize the path through the maze. Optimization can include minimizing the number of errors the mouse makes and minimizing the time it takes for the mouse to complete the maze. There are many ways that this can be done. Our understanding based on the micromouse guidelines is that the micromouse gets time to explore the maze and is then timed to see how fast it can reach the end. Therefore, the optimization would be to use the information that the micromouse collected while it was exploring to determine the optimal route through the maze. On its

timed attempt, the micromouse would follow the predetermined route and follow this to reach the end in the quickest possible time. To achieve this the following was required:

1. The micromouse needs to store the information on the placements of the walls in the maze.
2. Based on the stored information, the micromouse needs to determine the shortest path through the maze.
3. The micromouse needed a simplified set of instructions to follow to reach the end from the starting position.

Optimization Implementation

A MATLAB function block is used within Stateflow to implement the algorithm that was designed to calculate the shortest path. This function uses the maze array, which is constructed during navigation by storing information about the presence or absence of walls at each intersection. The maze is represented as a 7x7 array where each cell contains a 4-bit binary value. These bits correspond to the walls in the cardinal directions—North, East, South, and West. As the micromouse navigates, its sensors detect walls and update the corresponding bits in the array. The function takes this maze array and the finish point as its two inputs. Using the algorithms, an array of instructions is returned. These instructions are a series of numbers that represent whether the micromouse should move forward, turn left, or turn right. This function is used after the micromouse has found the finish. Once the function runs, the array that is returned is set to an array (path) that will be used to move the micromouse through the maze.

Milestone Four Achievements

In milestone four, the micromouse successfully stored maze information, calculated the shortest path, and followed optimal instructions to complete the maze efficiently. The optimized route showed improvements in minimizing errors and reducing the time required to complete the maze.

Results and Recommendations

Although the micromouse was able to complete the maze during navigation, some errors affected the maze array, making optimization less effective. Improvements should focus on error handling if the shortest path cannot be found. Further recommendations include improving the calibration process and using dynamic calibration to ensure accurate line following throughout varying maze conditions.

INTRODUCTION

This report presents the final progress and results for the micromouse project in EEE3099S, focusing on milestones three and four. The objective of the project is to develop a micromouse capable of solving a maze autonomously. Milestone one covered the design and implementation of the power and sensing modules for EEE3088F, as well as the initial hardware setup for the standardized mouse used in EEE3099S. In milestone two, the focus was directed toward basic movement and sensor integration. Milestone three involved demonstrating the micromouse's ability to move, make decisions, and execute turns based on the prior development work. Milestone four concentrated on optimizing the micromouse's path through the maze, reducing errors, and minimizing the time required for completion. This report details the requirements for milestones three and four, analyzing the design decisions, implementation processes, and performance of the micromouse during the final demonstration. The conclusion reflects on the overall outcomes of this project, what has been achieved, and what could be improved.

Summary of Requirements

Milestone Three: Navigation

The focus of milestone three is to use the progress and information from milestone two to move, decide, and make turns. From milestone two, the micromouse must still have a calibration, an initialization sequence, and surrounding awareness and it must be able to identify and hold a line. Our understanding for this milestone is that the micromouse must be able to successfully move through the maze, however, it does not need to be able to move through the maze optimally. This is broken down into smaller objectives. For the micromouse to be able to successfully move through the maze, the following objectives are required:

1. The micromouse must be able to stop at intersections.

To stop at an intersection, the micromouse must either be able to know how far it has traveled or sense where the intersections are and use this information to tell it to stop once it has reached an intersection.

2. The micromouse must be able to sense the difference between when there is a wall and when there is no wall at its right, left, and front.

The micromouse must use the information from its side sensors and the sensor facing forwards to interpret if there is a wall or no wall. The side sensors can be used at the intersections to decide if the micromouse should turn at these points. The forward sensors should continually be checked to ensure if the micromouse misses a wall at an intersection that it is able to stop and make decisions once it has detected a wall.

3. The micromouse must be able to turn at intersections and continue line following after it has turned.

Once the micromouse has made a decision from the information and has stopped at an intersection, the mouse must be able to turn at the intersection so that it is able to find the line afterward and continue holding the line as it moves forwards.

4. Logic is required so that the micromouse does not get stuck in a loop and explores new areas of the maze and so that the micromouse knows when it has reached the end of the maze

The micromouse uses all of its sensors to make decisions and memory of previous decisions at intersections which allows the micromouse to not get stuck in a loop. Additionally, for the micromouse to know when it has reached the end, the micromouse must be able to determine that it has reached a 2x2 block.

Milestone Four: Optimization

The focus of milestone four was to optimize the path through the maze. Optimization can include minimizing the number of errors the mouse makes and minimizing the time it takes for the mouse to complete the maze. There are many ways that this can be done. Our understanding based on the micromouse guidelines is that the micromouse gets time to explore the maze and is then timed to see how fast it can reach the end. Therefore, the optimization would be to use the information that the micromouse collected while it was exploring to determine the optimal route through the maze. On its timed attempt, the micromouse would follow the predetermined route and follow this to reach the end in the quickest possible time. To achieve this the following was required:

1. The micromouse needs to store the information on the placements of the walls in the maze.

While the micromouse explores the maze, it needs to store the information that it collects about what walls are at each intersection. This information is already collected as it moves through the maze for it to decide when and in what direction it should turn. Each time it collects this information, it must store it so that it can be used for optimization.

2. Based on the stored information, the micromouse needs to determine the shortest path through the maze.

The information about where walls are in the maze is collected from where the micromouse has been explored. This information can be used to determine the shortest path from the starting position to the ending position that the micromouse found.

3. The micromouse needed a simplified set of instructions to follow to reach the end from the starting position.

Using the shortest path, this can be turned into a simplified set of instructions that the micromouse can follow so that at each intersection it decides if it must turn right, left, or move forward. This will move the micromouse through the maze directly from the start position to the end position.

NAVIGATION

Design

The flow diagram in Figure 1 shows the navigation process's design and is explained further in the subsequent sections. The micromouse uses a left-wall-following strategy to navigate through the maze. In this approach, the micromouse prioritizes following the left wall at every opportunity. This method simplifies decision-making by always turning left when possible, ensuring that the mouse explores the maze systematically while avoiding unnecessary loops or backtracking. The strategy helps ensure that the micromouse can successfully navigate even without complete knowledge of the maze.

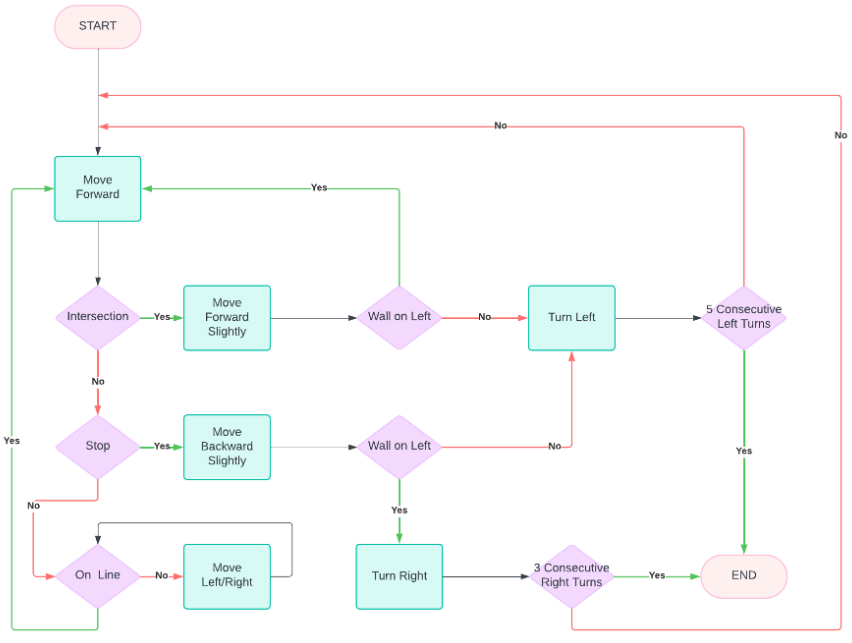


Figure 1: Flow chart explaining the navigation process

Once the calibration sequence is complete, the micromouse is ready to transition into its navigation state. How the micromouse transitions from its calibration state to its navigation state is detailed in the "Navigation: Implementation" section of the report.

During navigation, the micromouse relies on environmental prompts, primarily intersections and walls, to make decisions.

After the micromouse has been calibrated, it is ready to start navigating through the maze. Once in the navigation state, the micromouse must make decisions based on prompts from its environment. Arguably, the most important prompt is when the micromouse reaches an intersection. It is imperative that the micromouse collects data at each intersection since this data will be used to optimize maze-solving in subsequent sections. Furthermore, pausing at intersections slows down the micromouse's movement through the maze. This provides valuable time during which the micromouse can turn specific infrared LEDs ON and OFF and collect data.

Therefore, the micromouse continuously checks whether it has reached an intersection while it is in motion. At each intersection, the micromouse first moves forward slightly to position itself favorably to turn if possible. The micromouse uses its left sensor to check whether there is a wall present on its left. If there is no wall present, the micromouse is free to turn left.

The micromouse should also keep track of how many times it has turned left consecutively. If the micromouse has turned left more than five consecutive times, it has (likely) reached the end of the maze. Five consecutive left turns are extremely unlikely unless the micromouse has reached the 2x2 end of the maze.

After the micromouse has turned left, it continues moving forward and starts checking for the next intersection.

If it has not reached an intersection, the micromouse's second priority is to check whether it has reached a wall. Intersections are placed at a higher priority than walls since the micromouse should theoretically never reach a wall before reaching an intersection – unless it veers significantly off course.

If the micromouse reaches a wall, its front-facing sensors will ensure it becomes aware of the impending collision. It must then move backward slightly – once again to position itself favorably to turn left or right.

Since the micromouse employs left-wall-following, it first checks whether there is a wall on its left. If there is no wall present, the micromouse is free to turn left, verify that it has not turned left more than five consecutive times, and continue moving forward.

However, if there is, in fact, a wall present on its left, the micromouse turns right instead of left. Since turning right is not prioritized by design, the micromouse should check whether it has turned right more than three consecutive times. More than three consecutive right turns imply that the micromouse has reached the end of the maze.

If the micromouse has not reached the end of the maze, it continues moving forward and proceeds with the checks detailed thus far.

Finally, it is of the utmost importance that the micromouse does not veer off course. The line following algorithm was explained in the milestone 2 report. Simply put, if the micromouse realizes it is not perfectly on the line (by using its downward-facing sensors), it corrects course by moving left or right (depending on which sensor is not perfectly positioned above the line).

Implementation

The design detailed above can be implemented using Simulink's stateflow software and MATLAB code. The main stateflow consists of various blocks and sub-blocks like: standby, calibration, and navigation.

The code was split into 7 sub-blocks or modules which were all implemented inside the navigation block:

1. Move forward
2. Stop at an intersection
3. Perform intersection checks
4. Turn left
5. Stop
6. Turn right
7. End of maze reached

The code in each of these modules uses data captured during the calibration process.

A Brief Overview of the Calibration Process

Firstly, it should be noted that due to the physical placement of the left and right sensors, its ADC readings were weak and unreliable. Therefore, its infrared LEDs were operated in high-current mode to increase their brightness. This significantly improved the quality of the ADC readings; however, it meant the LEDs had to be turned off when not in use to conserve power and protect them from damage.

During calibration, the values read from the ADCs are captured to be used during navigation. While in the calibration state, pushbutton 2 (SW2) is used to transition between six sub-states:

1. The micromouse has veered slightly to the right
2. The micromouse has veered slightly to the left
3. The micromouse is positioned perfectly on the line
4. The micromouse is on an intersection
5. The micromouse is not surrounded by any walls
6. The micromouse is surrounded by walls

During sub-states 1-4, the left and right LEDs are OFF. During sub-states 5-6, the left and right LEDs are ON.

Now that the calibration process is well understood, the navigation process can be explained in greater detail.

Move Forward

This is the micromouse default state during navigation. Most, if not all, decisions ultimately end with the micromouse moving forward again – except when the micromouse has reached the end of the maze.

To move the micromouse forward, the left and right wheels are set at 75% of their maximum speed by updating the leftWheel and rightWheel output variables from 0 to -75 (the direction of the wheels is opposite from what one would expect; hence, the negative sign). After performing various tests, it was found that less than 75% of the maximum speed is not sufficient to overcome the inertia of the wheels.

Conversely, operating the micromouse at a relatively low speed, especially in its default state, has other advantages. It is imperative that the micromouse senses when it has reached an intersection, stops when it reaches a wall, and remains steadily on the line. To enhance the accuracy of the micromouse, it was operated at 75% of its top speed in its default state.

Stop at an Intersection

The sensors used to track the wheels were manually bent to point downward. They were henceforth used to identify an intersection.

During calibration, a value is read from the ADC associated with the intersection sensors and stored in a local variable. After performing various tests using Simulink's debug mode, it was found that the value on the ADC decreases significantly when the sensor crosses an intersection. Therefore, a condition could be set: if the value on the ADC associated with the intersection sensors drops below the threshold (which was recorded during calibration), the micromouse is almost certainly at an intersection.

Due to varying ambient lighting conditions throughout the maze, the value of the ADC may fluctuate slightly. To compensate for potential fluctuations, the infrared LEDs were wrapped with electrical tape to direct their light more precisely. Furthermore, the photodiodes were also wrapped with electrical tape to prevent interference from ambient light. Lastly, the threshold values were adjusted upwards to accommodate for any potential fluctuations and ensure all intersections are sensed.

Since there are two intersection sensors (one on the left and one on the right), the condition could either be set using an AND or an OR statement. In other words, should both or at least one of the sensors be above an intersection to detect the intersection?

Practical tests proved that the micromouse was better equipped to sense intersections when an OR statement was used. This is because the micromouse may not always be moving in a perfectly straight line (it may be in the process of correcting its course); however, since checking for intersections takes priority, the alignment of the micromouse is not considered.

The micromouse was ultimately able to sense intersections with great accuracy.

Perform Intersection Checks

At each intersection, the micromouse is brought to a halt by adjusting the leftWheel and rightWheel output variables from -75 to 0.

As discussed previously, the left and right infrared LEDs are operated in high-current mode. To conserve power, they are only activated at an intersection. Two output variables, input_left and input_right, were defined. The values of these variables specify the state (1 or 0) of the infrared LEDs which can be updated in a different stateflow. At an intersection, these variables are set to 1.

A small delay (1 sec) is implemented before the values on the ADCs are compared to the values captured during calibration. Testing showed that the value of the ADC increases when a wall is present (except when there is significant ambient light). Therefore, if the value on the ADC is greater than the threshold value, a wall must be present.

The micromouse's onboard LEDs are used to indicate which walls are present. If there is a wall on the left, LED0 is switched ON. If there is a wall in front, LED1 is switched ON. If there is a wall on the right, LED2 is switched ON. The walls surrounding the micromouse are captured in three Boolean local variables: front_wall, left_wall, and right_wall. This is an essential step needed to map the maze and optimize the micromouse's maze-solving strategy.

To compensate for ambient light, which contains significant infrared light, these infrared LEDs and photodiodes were also wrapped with electrical tape. Furthermore, the threshold values were adjusted downward. Lastly, testing showed that the value of the ADC skyrocketed to 3.3V when there was abundant ambient light. This *only* occurs when there is *no wall* present. Therefore, an upper limit was set.

If the value on the ADC exceeds this upper limit, it is assumed that this is due to interference from ambient light and that a wall is not present.

After a small delay (0.5 sec), the infrared and on-board LEDs are turned off. The micromouse is programmed to move forward slightly (for 0.5 sec) to position itself optimally for turning.

Based on the walls surrounding the micromouse, decisions can be made. As explained in the design section of this report, the micromouse will check whether there is a wall to its left. If there is a wall present, the micromouse will continue moving forward. If there is no wall present, the micromouse will turn left, and *then* move forward.

Note that the length of the delays was determined experimentally.

Turn Left

To turn left (in a perfect circle to avoid collisions with the maze walls), the micromouse's wheels must move at the same speed, but in opposite directions. The output variable leftWheel is set to 75 (moving backward) and the output variable rightWheel is set to -75 (moving forward).

As mentioned previously, the micromouse must keep track of the number of consecutive left turns it has made. Continuous left turns imply that the micromouse has reached the 2x2 end of the maze. Therefore, a local variable, left_turns, was defined. Its initial value is set to zero and it increments each time the micromouse turns left. It is reset to zero each time the micromouse turns right.

After a small delay (0.8 sec), the same line following logic that was explained in the milestone 2 report is used to check whether the micromouse has reached a line. Once it senses a line, it stops turning left and starts moving forward – both leftWheel and rightWheel are set to -75.

Note that the length of the delays was determined experimentally.

Stop

When the micromouse reaches a wall, it is brought to a halt by adjusting the leftWheel and rightWheel output variables from -75 to 0. Similarly to before, the input_left and input_right output variables are set to 1 to activate the left and right infrared LEDs. Since there is a wall in front of the micromouse, LED1 is turned ON.

If the micromouse is too close to a wall, it will not be able to turn properly and may collide with the maze walls. Therefore, the micromouse first moves back for 0.5 sec. This is done by setting leftWheel and rightWheel to 75. The micromouse is once again brought to a halt.

A small delay (0.5 sec) is implemented before the values on the ADCs are compared to the values captured during calibration. If there is a wall on the left, LED0 is switched ON and if there is a wall on the right, LED2 is switched ON. The walls surrounding the micromouse are captured in three Boolean local variables: front_wall, left_wall, and right_wall.

After a small delay (0.5 sec), the infrared and on-board LEDs are turned off. The micromouse is programmed to move forward slightly (for 0.5 sec) to position itself optimally for turning.

Based on the walls surrounding the micromouse, decisions can be made. As explained in the design section of this report, the micromouse will check whether there is a wall to its left. If there is a wall present, the micromouse will *turn right*, and *then* move forward. If there is no wall present, the micromouse will turn left, and *then* move forward.

Note that the length of the delays was determined experimentally.

Turn Right

To turn right (in a perfect circle to avoid collisions with the maze walls), the micromouse's wheels must move at the same speed, but in opposite directions. The output variable leftWheel is set to -75 (moving forward) and the output variable rightWheel is set to 75 (moving backward).

As mentioned previously, the micromouse must keep track of the number of consecutive right turns it has made. Continuous right turns imply that the micromouse has reached the 2x2 end of the maze – this is especially true since the left-wall-following algorithm prioritizes left turns. Right turns are therefore rare. Consequently, a local variable, right_turns, was defined. Its initial value is set to zero and it increments each time the micromouse turns right. It is reset to zero each time the micromouse turns left.

After a small delay (0.8 sec), the same line following logic that was explained in the milestone 2 report is used to check whether the micromouse has reached a line. Once it senses a line, it stops turning left and starts moving forward – both leftWheel and rightWheel are set to -75.

Note that the length of the delays was determined experimentally.

End of Maze Reached

Before the micromouse turns left or right, it prioritizes checking the value stored in left_turns and right_turns, respectively.

If the number of consecutive left turns exceeds five, one can reasonably assume that the micromouse has reached the end of the maze. Similarly, if the number of consecutive right turns exceeds 3, one can reasonably assume that the micromouse has reached the end of the maze.

Once the end has been reached, the micromouse is programmed to spin in a circle at full speed (leftWheel = -100 and rightWheel = 100) and flash its onboard LEDs. This continues for 5 seconds before the micromouse is brought to a halt.

Initially, the micromouse often thought it had reached the end of the maze when, in fact, it had not. Therefore, code was implemented to also count the number of intersections encountered (using a local variable called forward_count) between left and right turns. If more than one intersection had been encountered, the left or right turn was not truly consecutive and did not indicate that the end of the maze had been reached.

Results, Analysis, and Recommendations

Table 1 provides an in-depth analysis of the project outcome – specifically regarding the navigation process. It clearly states what was successful and provides recommendations to improve the design.

Table 1: Results, analysis, and recommendations to improve the navigation process

Result	Analysis	Recommendation
Line following was not always accurate and relied heavily on the calibration process	At first, the line following algorithm seemed to work well; however, it became less reliable when turns were introduced. Furthermore, as the micromouse moved through the maze, the ambient conditions changed. Therefore, the thresholds determined during calibration may not be relevant anymore. The micromouse often over-corrected. Once it had veered off course	The calibration process should be improved. Dynamic calibration would be ideal since this will continuously update the thresholds and ensure more accurate line following. Furthermore, the line following algorithm could be improved by using proportional control or a PID controller. This may prevent the micromouse from over-

	too much, it struggled to find the line again.	correcting and adjustments will be proportional to the deviation from the line. Lastly, there should be a failsafe to prevent the micromouse from veering too much off course.
The micromouse was able to move forward reliably – albeit very slowly	Speed may not have been a priority (compared to accuracy); however, operating the micromouse at a greater speed will enhance its overall performance.	The micromouse should be operated at a greater speed. The speed that provides a balance between accuracy and efficiency should be chosen carefully. The time delays should be updated with the new speed in mind.
The micromouse almost always sensed intersections	The micromouse was able to sense intersections with great accuracy; however, there is still room for improvement. The high accuracy can be attributed to the use of an OR statement as well as the use of electrical tape to enhance precision.	The intersection infrared LEDs should also be operated in high-current mode. This will provide a broader range of values that indicate that an intersection is present. As was seen with the left and right sensors, this will undoubtedly enhance the micromouse's sensing abilities.
The micromouse was not able to reliably detect walls on its left and right	Even though operating the left and right infrared LEDs in high-current mode improved the accuracy of the sensors, they were still unreliable. This is due to the physical limitations of the micromouse design. The left and right sensors are simply too far from the walls to reliably detect their presence. Attempts were made to compensate for the weak sensor readings in code; however, the increase in accuracy was marginal.	There are few things one can do about the physical design. One option is to desolder the sensors, construct a "bridge," and physically move the sensors closer to the walls. However, this comes at the risk of damaging the sensors.
The micromouse did not collide with walls	The micromouse was able to accurately sense when it had reached a wall.	Some improvements can still be made: once again, operating the forward-facing infrared LEDs in high-current mode could potentially improve the sensor readings, but should be balanced with power consumption.
Turning ON the infrared LEDs only when needed proved to be challenging and lead to unnecessary complications	The fact that the infrared LEDs had to be turned ON and OFF periodically proved tedious. It complicated the code, lead to erroneous readings, and unpredictable behavior.	A better approach is to always leave the infrared LEDs ON; however, to conserve power, they should be pulse width modulated. However, PWM cannot be implemented in MATLAB since it is too slow. Therefore, it would have to be implemented directly in C.

OPTIMIZATION

Design

As outlined under the requirements summary, the objectives for optimization are to use the information collected about the maze during navigation to complete it more efficiently. A more efficient way can mean fewer errors, or it can mean a faster time. The approach used was to meaningfully capture the information about the maze during navigation to calculate the shortest route possible from start to end. The assumptions that this was based on are:

- The start and end positions will remain the same from navigation.
- The start position is always in the left-most top block.
- The micromouse will be physically moved back to the start position once it reaches the end during navigation.
- Once back at the start position, the micromouse will be timed to determine how quickly it can reach the determined end position.

From this, the design of the optimization has three parts:

1. Firstly, a method for collecting and storing information about the maze during navigation is designed.
2. Secondly, an algorithm calculates the shortest possible path between the start position and the end position.
3. The shortest possible route is turned into a set of usable instructions that can be followed by the micromouse to move from the start to the end position.

Collecting and Storing Information

The information is stored in a 7x7 array of 4-bit binary numbers. The binary numbers are all 4-bit numbers where each bit relates to whether there is a wall or no wall around on each side of the block. It is designed so that the first bit is the north side, the second is the east side, the third is the south side, and the fourth is the west side (0bWSEN). All the binary numbers start at 0b0000 which means that the block has not been visited. Once information about the block is added, a one means that there is a wall and a zero means that there is no wall.

This array is updated at every intersection based on the location, the direction the mouse is facing and moving, and the information from the sensors when a wall has been detected. This requires that the direction be tracked and every time the micromouse turns, the direction is updated based on the turn. Additionally, the location of the micromouse is tracked so that the information on the walls is updated for the correct position in the maze. The location starts at the starting position and is updated each time the mouse moves forward and reaches a new intersection based on the direction it is facing.

Calculating the Shortest Path

Once the mouse has reached the end position, there will be two pieces of data that have been collected that can be used to determine the shortest path. This includes the end position and the placement of walls determined during navigation. An algorithm uses this information to determine the shortest path.

The algorithm uses a Breadth-First Search (BFS) approach to compute the shortest distance from each cell of a 7x7 maze to a specified endpoint. BFS is an algorithm that explores all nodes at the present depth level before moving on to nodes at the next depth level, making it ideal for finding the shortest path in an unweighted grid like this maze.

A distance matrix is initialized with -1, representing unvisited cells, except the finish point, which starts at 0. The BFS begins at the finish point, enqueuing it and marking its distance as zero. As BFS progresses, it processes each cell in the queue by checking moves (north, east, south, west) using predefined directions.

For each potential move, it checks if the new position is within the maze's bounds and whether the path is not blocked by a wall. If a move is possible and the target cell is unvisited (distance -1), the algorithm updates the target cell's distance to the current cell's distance plus one and enqueues this new cell for further exploration. This process continues level by level until all reachable cells are processed, ensuring that each cell's distance represents the shortest path to the finish point.

Optimal Path Instructions

The next step is to use the distance array to determine the path the micromouse can follow through the maze. It uses the distances to trace a route that leads to progressively lower distance values, effectively following the shortest path identified by the BFS.

The algorithm initializes the starting position and a list to store movement instructions. It defines direction vectors ([-1, 0; 0, 1; 1, 0; 0, -1]) corresponding to north, east, south, and west, as well as numeric codes for these directions (0, 1, 2, 3 respectively).

A while loop runs until the current position matches the finish point, indicating that the path is complete. At each iteration, the algorithm checks the current cell's distance and attempts to move to an adjacent cell with a distance value that is one less than the current cell's distance—signifying a step closer to the finish. It examines each of the four possible directions (north, east, south, and west) and checks if moving in that direction is within the maze's bounds and results in a cell with a lower distance value. If a valid next position is found, it determines the direction of movement based on the change in row (delta_x) and column (delta_y) and appends the corresponding direction code to the instructions list.

After recording the direction, it updates the current position to the next position and repeats the process until the finish point is reached. The resulting instructions list contains a sequence of numeric codes that represent the directions the micromouse should take to follow the optimal path from the start to the finish.

Lastly, the cardinal direction movements are converted to a set of actionable instructions the micromouse can follow, including turning and moving forward. It considers the current facing direction of the micromouse and adjusts it to align with each target direction as it navigates through the maze.

The initial state is set with the starting direction as east (current_direction = 1). The movement instructions list (move_instructions) is initialized. Three constants (forward, right, left) are defined to represent movements.

The algorithm iterates over each direction in the instructions list, where each value represents a desired target direction. For each target direction, it calculates the difference between the current direction and the target direction. This difference determines how the micromouse should turn:

- If the difference is 0, the micromouse continues forward in its current direction.
- If the difference is 1 or -3, the micromouse needs to turn right (a 90-degree turn to the right). After turning, it moves forward, and the current_direction is updated to reflect the new direction.
- If the difference is -1 or 3, the micromouse turns left (a 90-degree turn to the left). After turning, it moves forward, and the current_direction is updated accordingly.
- If the difference is 2 or -2, it means the micromouse needs to turn around (a 180-degree turn), which is achieved by making two right turns consecutively before moving forward.

After processing each instruction, the algorithm updates the current_direction to reflect the new facing direction of the micromouse. The result is a list where each element represents a command to either move forward, turn right, or turn left.

Implementation

This design is implemented in MATLAB, and it is focused in the main stateflow block of the program. The implementation of the design as well as the challenges and changes that were made in implementation are broken down into distinctive design sections. The MATLAB stateflow is shown in the Appendix.

Implementation of Collection and Storing of Information

This part of the optimization is done during navigation. Upon entry of the navigation block, the following variables are initialized:

- The 7x7 array (maze) of 4-bit numbers is initialized so that all the numbers are 0b0000. The only number that is different is the start block at [1,1] which is set to 0b1101 because the micromouse does not stop at the intersection in this block and these are the walls that are always present in the first block.
- Four 4-bit numbers, for north, east, south, and west are defined so that these can be easily used to set the walls around the micromouse at each position. Additionally, to make the logic easier in the blocks, three 4-bit numbers are defined for right, left, and front. These are the positions that the micromouse recognizes walls at. Based on the current direction, these three variables are set to one of the cardinal position 4-bit numbers. These three numbers are used to set the walls at each location for the three sides with sensors.
- The direction is set to east (direction = 1). This is changed when the micromouse turns so that north is zero, east is one, south is two and west is three. These numbers for each direction correspond to the bit number that represents that direction in the maze array.
- The direction vector is set to move the micromouse east if it moves forward (direction_vector = [1,0]). This vector changes the current location of the micromouse so that information is added to the correct 4-bit number. North is [0,1], east is [1,0] south is [0,-1] and west is [-1,0].
- The location is set to the leftmost top block (location = [1,1]). This corresponds to the current position in the 7x7 array as the micromouse moves through the maze.

During navigation, the location of walls in the maze is collected based on the following logic. The main changes occur when the micromouse is at an intersection. At each intersection:

1. The location is updated based on the direction vector. With the updated location, the row (x) and column (y) in the maze array for the current location are set to change the 4-bit number at this point.
2. The direction the micromouse is facing is checked and based on this the right, left and front bit numbers are updated.
3. Using the sensor logic, which is already implemented for navigation, the front, right, and left sensors are checked to see if there is a wall. If the sensor detects a wall, then using the 4-bit numbers for each side (front, right, and left), the 4-bit number at this location in the array is updated. An OR statement with the side's number and the location's number is used to update the wall placement. The OR statement means that each time a wall is detected, this is updated in the array, but it does not change any of the walls that are already updated at this location.

The maze array is also updated at one other point. The front sensors sometimes do not detect walls in front of it at intersections but only when it is closer. The logic is already in place from navigation to stop

when a wall is sensed in front of it. In this block, the same logic is used to get the x and y positions, but the location is not updated. The direction is also checked but only the front value is set based on the direction. The maze is updated at this position using the front value.

Lastly, it is important to update the variables that were initialized when the micromouse turns. In the turn right and turn left block, the direction vector and the direction are updated based on the current direction. In the turn-right block, the direction is increased by one unless it is three, in which case the direction is set to zero as shown in Figure 2. In the turn-left block, the direction value is decreased by one unless it is three in which case it is set to zero as shown in figure 3.

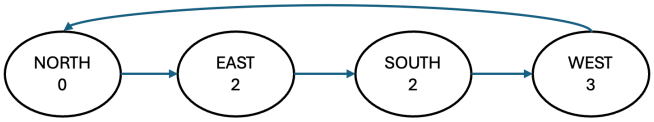


Figure 2: Clockwise rotation of directions after turning right

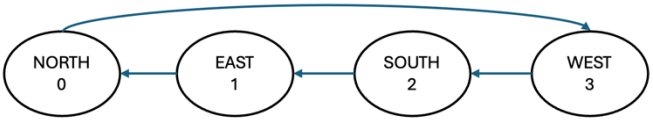


Figure 3: Anti-clockwise rotation of directions after turning left

When this is all implemented together, the micromouse is able to build a 7x7 array of the maze to represent where there are walls from the information that it collected while navigating the maze.

Shortest Path Implementation

A MATLAB function block is used within stateflow to implement the algorithm that was designed to calculate the shortest path (shortestPath). This function uses the maze array built during navigation and the endpoint as its two inputs. Using the algorithms, an array of instructions is returned. These instructions are a series of numbers that represent if the micromouse should move forward, turn left, or turn right. This includes that if it turns, it will always need to move forward directly after to reach the next intersection in the new direction.

This function is used after the micromouse has found the finish. This function is run and the array that is returned is set to an array (path) that will be used to move the micromouse through the maze. From the initial algorithm, some changes had to be made for the algorithm to work in a MATLAB function in stateflow. The most important change was that array sizes all need to be predefined and constant. The variable size was set under the data properties, and the arrays were all initialized to a constant size and filled with ‘empty’ data. This meant that the new data could not be added at the end of the array or list, but an index must be used to move through the array.

Optimal Movement Through the Maze

For the micromouse to optimally move through the maze, the instruction array (path) is sequentially moved through using an index. To restart the micromouse at the start position switch two (SW2) is used. This starts the next sequence of instructions after the shortestPath algorithm has already been run and the path array has been set. Using the path array of instructions, the current instruction is collected, and depending on its value, the micromouse either moves forward, turns right, or turns left. The sequence to check how it should move is shown in Figure 4 and explained in the below points:

- Firstly, the current instruction is checked to see if it is zero, which means that the micromouse has reached the end of the instructions and therefore, the micromouse has reached the finish. Zero means that it has reached the end because the path array is originally filled with zeros; therefore, any unchanged point in the array will be zero.
- If the instruction is one, the micromouse moves forward, if the instruction is two, the micromouse turns right and if the instruction is three then it turns left.
- After it has moved, the next instruction in the sequence is collected from the path array by using the index. The index variable is increased after each instruction is collected.

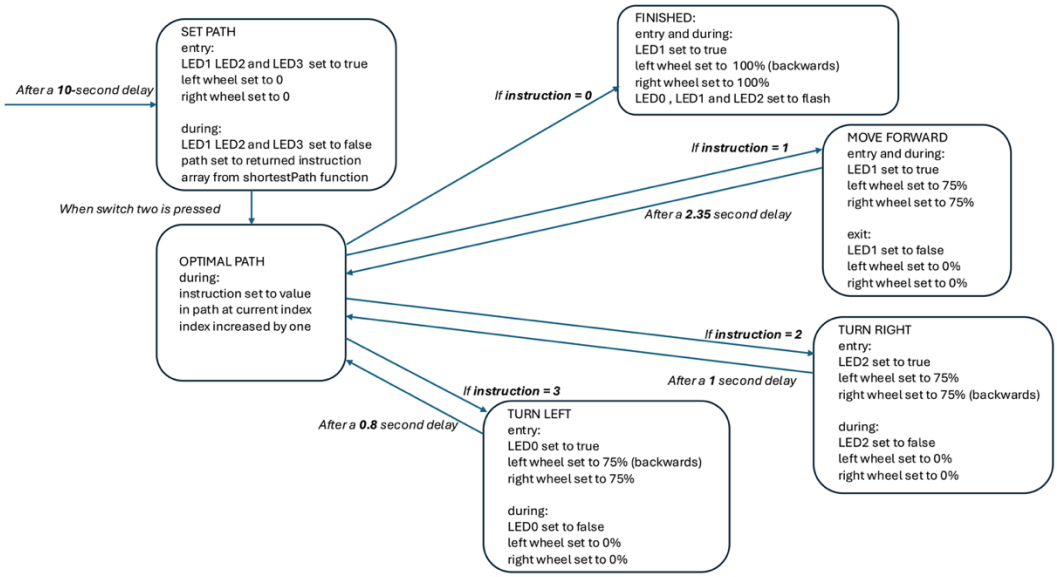


Figure 4: Path following flow chart during optimization

This will move the micromouse through the maze based on the optimal path that it found during navigation.

Results

There were many challenges that arose when testing the optimization. For the optimization to run effectively, the micromouse first had to complete the maze so that it had built a map of the maze in the maze array. If this map is incorrect, then the algorithm may not be able to find any path, or it would find the incorrect path. Although, with a few errors, the micromouse was able to complete the maze during navigation, these errors would make the maze array incorrect. Some of the errors during navigation and how they would affect the maze array are detailed in Table 2.

Table 2: The effects of errors during navigation on optimization

Error during Navigation	Effect on Optimization
The micromouse missed an intersection	After the missed intersection, the location of the micromouse will be incorrect. At intersections, the location updates based on the direction it is moving. If an intersection is missed, the location will not have been updated. Additionally, the walls at this point will not be detected because this is done at intersections. After the missed intersection, the maze array will be wrong because any further updates will be for the incorrect location.

The micromouse did not detect a wall that was there	Not detecting a wall there means this will not be updated in the maze array. If it misses a wall that is on the left, it will try to turn left, and the front sensors will detect the wall to correct the error. If a wall on the right is not detected, depending on the other surroundings this might cause the micromouse to turn this direction and the error will be corrected from the front sensors. If the micromouse does not turn this direction it might cause an error in the maze array; however, if it goes past the same point and does detect the wall, this will be updated and corrected. Therefore, this does not have a significant impact.
The micromouse detected a wall that was not there	If a wall is detected that is not there, this will cause an error in the maze array which cannot be corrected. Since an OR statement is used to update the array, once the wall is set for a position it cannot be changed if a sensor then detects that a wall is not there. During optimization, this could mean that the algorithm cannot find a possible path based on an incorrect wall blocking it.

The algorithm was tested and showed that if there is an accurate mapping of the maze it will find the shortest path and turn this into a set of usable instructions. Three examples are given in Appendix B of different mazes, the array that the micromouse would have built of these mazes using its navigation if it navigated the maze perfectly, and the set of instructions that would have been used to optimally move through the maze. This shows that the algorithm can find the correct and optimal path with accurate data. The disadvantage of this method is that it is only able to find the optimal path based on routes it has explored during navigation. This means there may be a more optimal path, but if it reaches the finish before exploring this area of the maze, it will not find this path during optimization. This is shown in the three mazes in Appendix B and explained in Table 3.

Table 3: Optimization comparison for three different mazes.

Maze 1	Although there is a more optimal route, from the navigation this route is unknown and so the route does not differ from the navigation route.
Maze 2	The route is optimized significantly from the navigation route, however, for the same reason as maze 1, the most optimal route is not found.
Maze 3	The most optimal route is found.

Improvements that can be made to improve accuracy and increase optimization are detailed in the conclusion.

CONCLUSION

Milestones three and four were the final milestones in the development of the micromouse. This project started in EEE3088F with designing a power board and sensor board for the micromouse. The projects continued in EEE3099S which used a standard micromouse and required code implementation to solve a maze. The first milestone focused on the design of the power and sensing modules for EEE3088F and understanding the hardware provided for our standardized mouse. Milestone two focused on initial functionality and using the sensor subsystem to produce useful information for the mouse to be able to move in a desired direction and change its movements based on its surroundings. From milestone two, the micromouse had a working calibration sequence, was able to stop when there were walls detected in front of it, and was able to identify and hold a line. This was the starting point for milestones three and four.

The focus for milestone three was to use the sensor information to navigate and solve the maze. The micromouse navigation process successfully achieved its objective of autonomously moving through the

maze by making decisions based on sensor data. The design involved multiple components: stopping at intersections, detecting walls, and making appropriate turns to continue navigating. These steps allowed the micromouse to explore new areas of the maze without getting stuck in loops, and it was able to detect when it reached the end of the maze.

However, the effectiveness of the navigation was influenced by the accuracy of the sensor readings and the micromouse's ability to consistently follow a line. Any inaccuracies in detecting walls or intersections could cause navigation errors, which might lead to inefficient movement or incorrect decisions. Improving the calibration process and refining the line-following algorithm could enhance the overall accuracy of the navigation phase, reducing the likelihood of errors and improving performance.

The focus for milestone four was to optimize the way the micromouse solved the maze. The micromouse optimization process achieved its goal of improving the efficiency of maze navigation by using information gathered during the navigation phase. The design involved three main components: collecting maze data, determining the shortest path using a Breadth-First Search (BFS) algorithm, and converting this path into a sequence of movement instructions for the micromouse. These steps enabled the micromouse, once returned to the starting point, to navigate the maze again, following the optimized path with reduced errors and in less time.

The BFS algorithm effectively calculated the shortest route from the start to the finish by using data stored in a 7x7 maze array, where each cell recorded the presence or absence of walls. This approach allowed the micromouse to determine the most direct path to its destination, converting the path into a series of forward movements, left turns, and right turns. The implementation demonstrated that when the maze data was accurately captured during navigation, the micromouse was able to find the correct path and follow it consistently.

However, the effectiveness of the optimization was highly dependent on the accuracy of the initial data collected during the exploration phase. If the micromouse missed intersections, detected walls incorrectly, or failed to detect existing walls, the resulting maze array could become inaccurate, leading to errors during the path calculation or making it impossible to find a valid path. While some errors could be corrected during subsequent passes through the maze, others, such as false positives for walls and more importantly missed intersections, could not be resolved, impacting the overall performance of the micromouse.

To improve the performance of the micromouse, a different strategy could have been used. After completing the design and implementing milestones three and four a different method was found for maze solving and optimization. Instead of only running optimization after navigation, the optimization can be included during navigation. Instead of using left-wall-following to complete the maze, the micromouse would be running an algorithm like the shortestPath algorithm at every intersection. This algorithm assigns a numerical distance value to each cell, representing the shortest path from that cell to a target (the end). Initially, the goal cell is set to zero, and surrounding cells incrementally receive higher values based on their distance from the goal. As the micromouse explores the maze, it updates these values based on the walls it finds at each intersection and uses them to decide its next move. It always attempts to move to the neighboring cell with the lowest value, thus gradually progressing toward the goal. When the micromouse reaches the end, it can use this algorithm to backtrack, finding the shortest path for the most optimal route.

APPENDICES

Appendix A – MATLAB Screenshots

The following screenshots are from the main stateflow in the MATLAB program which controls the logic for the micromouse.

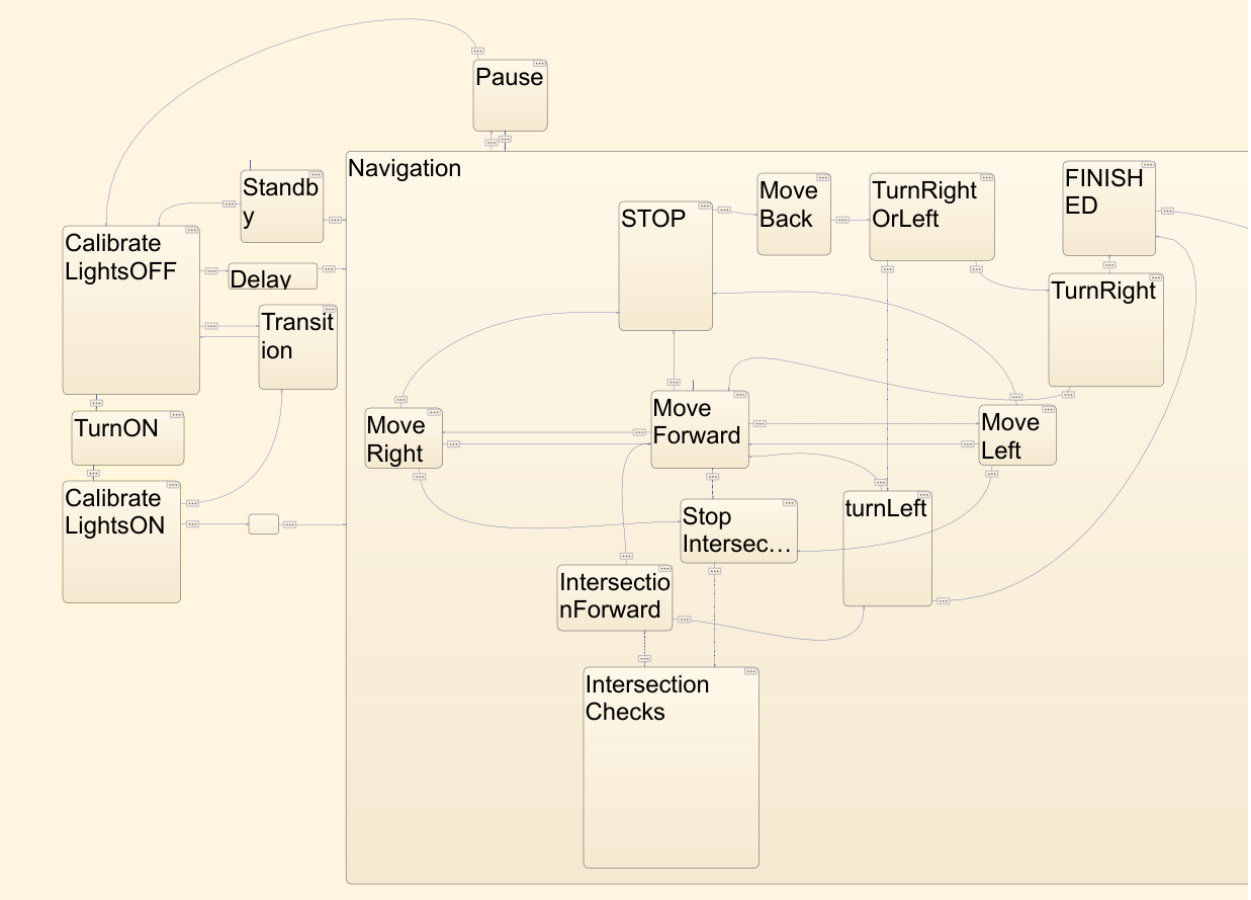


Figure 5: Overview of calibration and navigation sequence

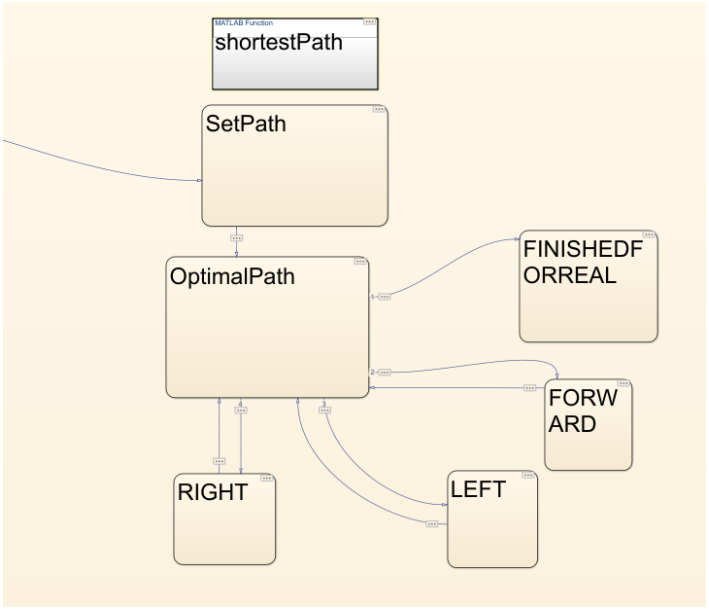


Figure 6: Overview of optimization sequence

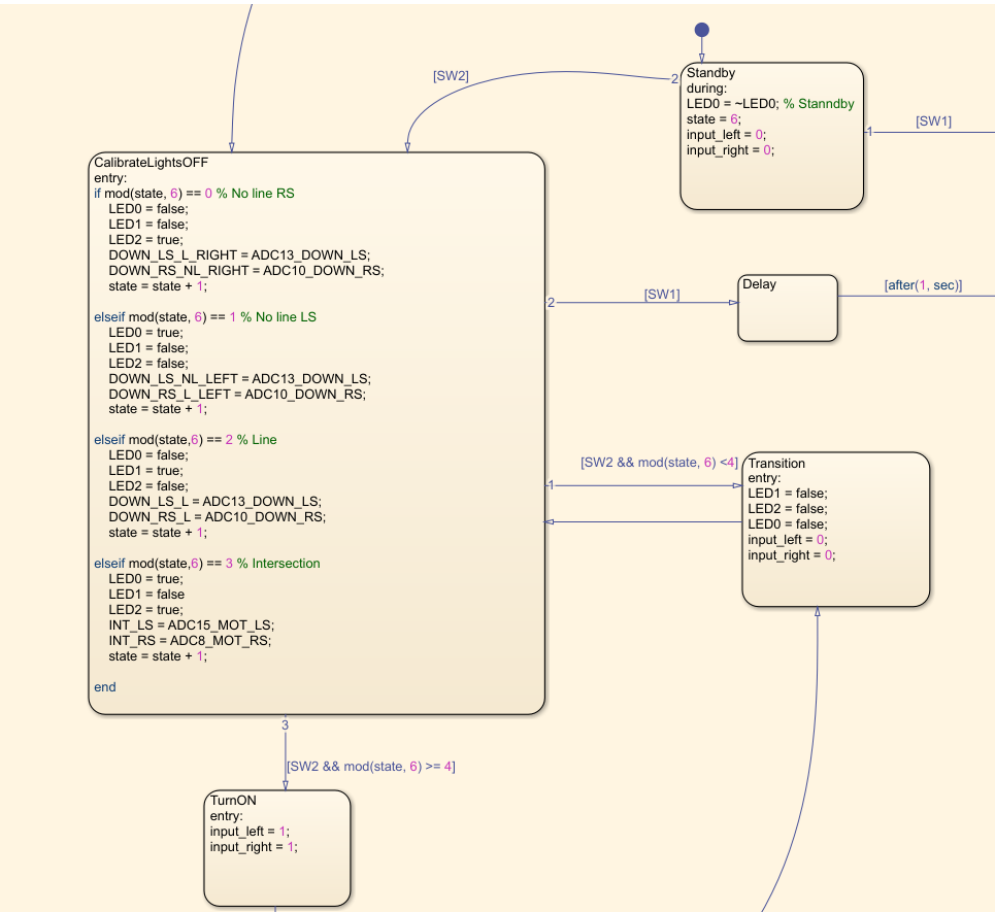


Figure 7: Calibration code in detail

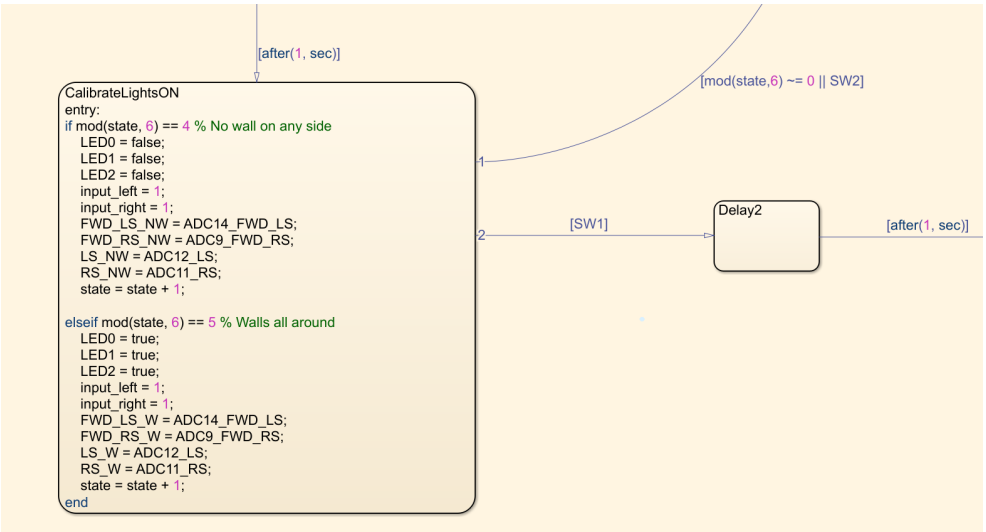


Figure 8: Second part of calibration code

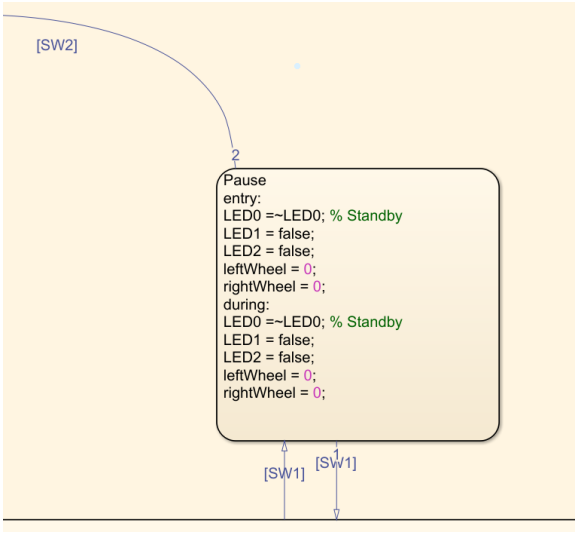


Figure 9: Pause logic between calibration and navigation

```
Navigation
entry:
LED0 = false;
LED1 = false;
LED2 = false;
input_left = 0;
input_right = 0;
right_turns = 0;
left_turns = 0;
forward_count = 0;
direction = 1;
direction_vector = [1,0];
location = [1,1];
maze = [
   0b1101, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000;
   0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000;
   0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000;
   0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000;
   0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000;
   0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000;
   0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000;
   ];
north = 0b0001;
east = 0b0010;
south = 0b0100;
west = 0b1000;
right = 0b0000;
left = 0b0000;
front = 0b0000;
start = [1,1]
finish = [1,1];
index = 0;
%rightWheel = leftWheel
%leftWheel = rightWheel
%left sensor = right sensor
%right sensor = left sensor
% maze is a 7x7 array
% 0b0000 is how it starts and means that block hasn't been explored
% 0 = wall, 1 = no wall
% 0bWSEN, N=north, E=east, S=south, W=West
% assumes initially facing east
% updates this at each intersection
% first block is different because don't set it's walls but assume walls all around except east
```

Figure 10: Entry code for the navigation block

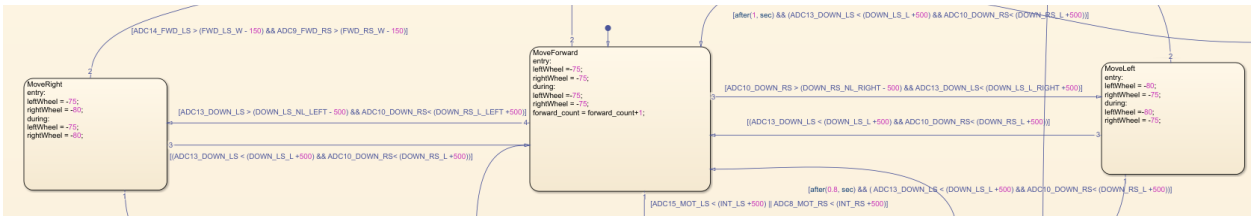


Figure 11: Line following logic

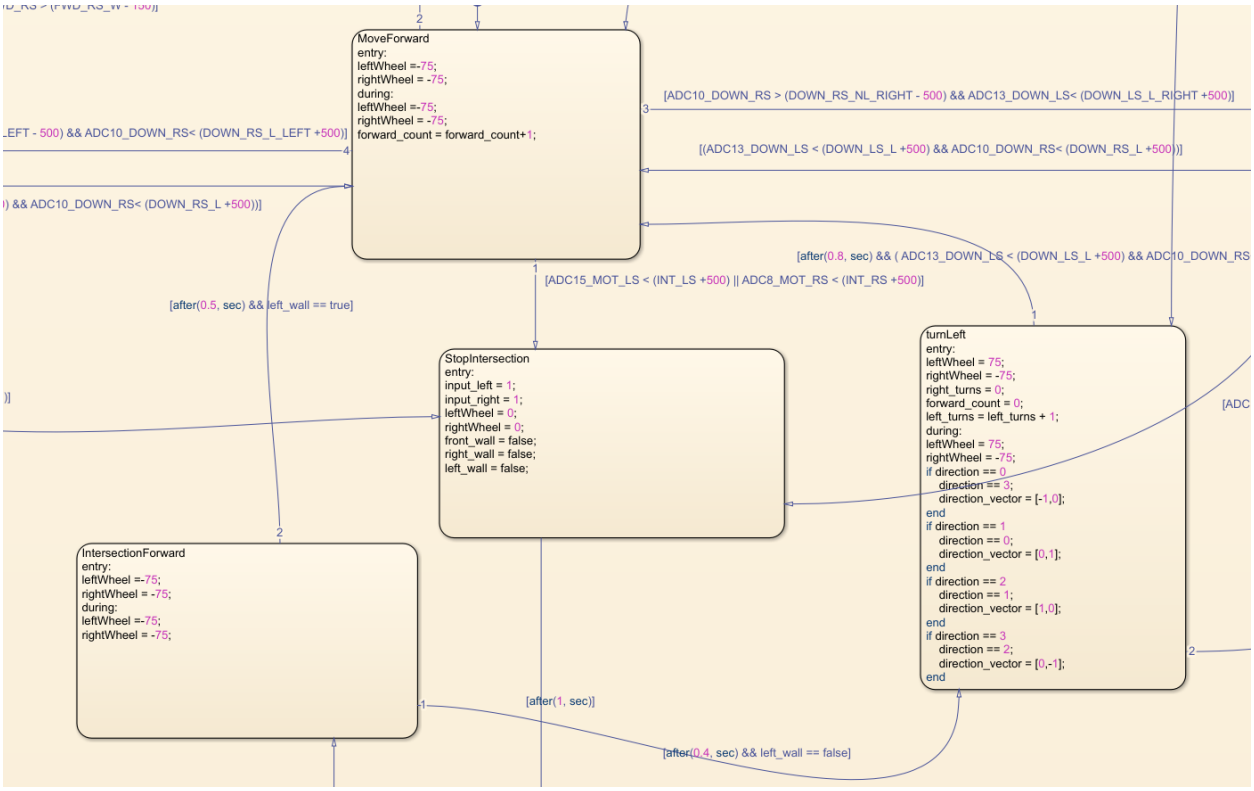


Figure 12: Finding intersection logic

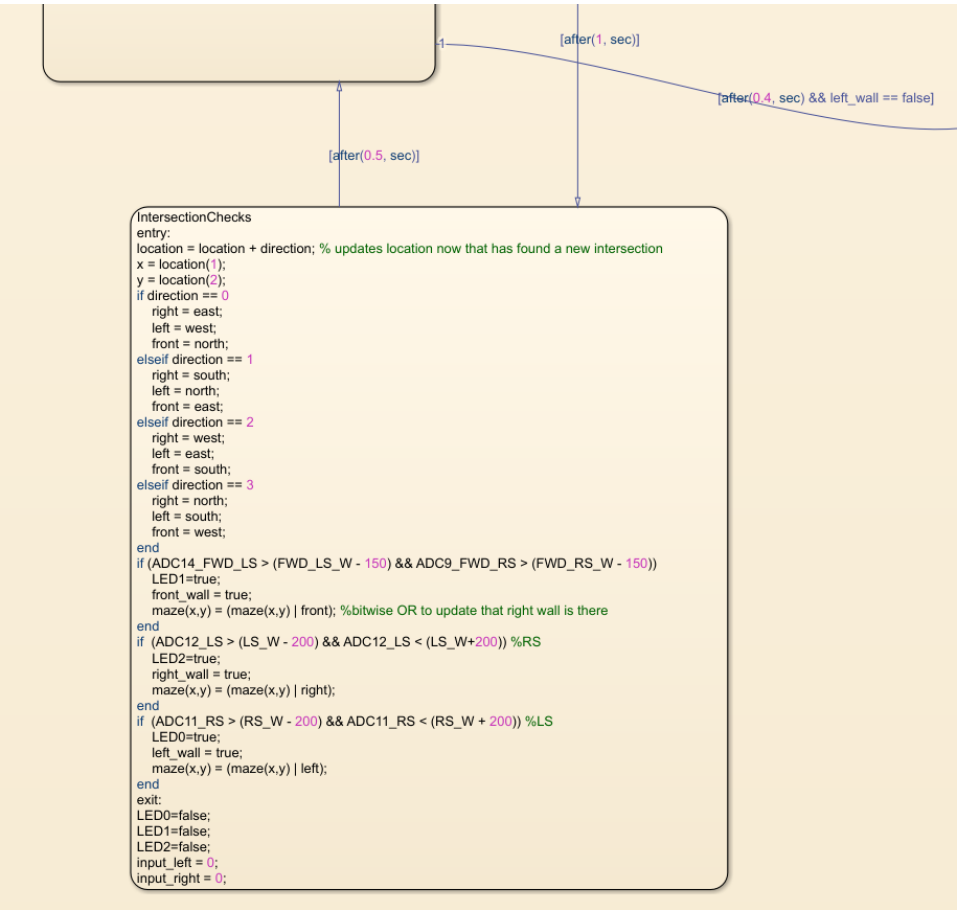


Figure 13: Code for intersection checks

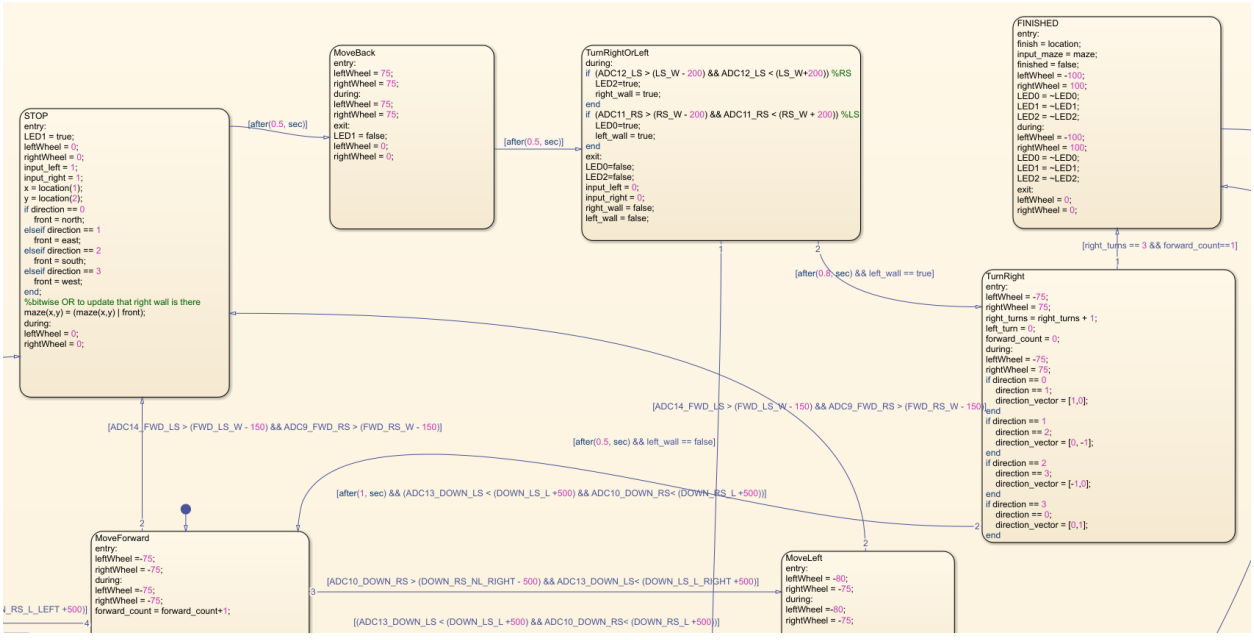


Figure 14: Code for stopping at a wall and turning right or left

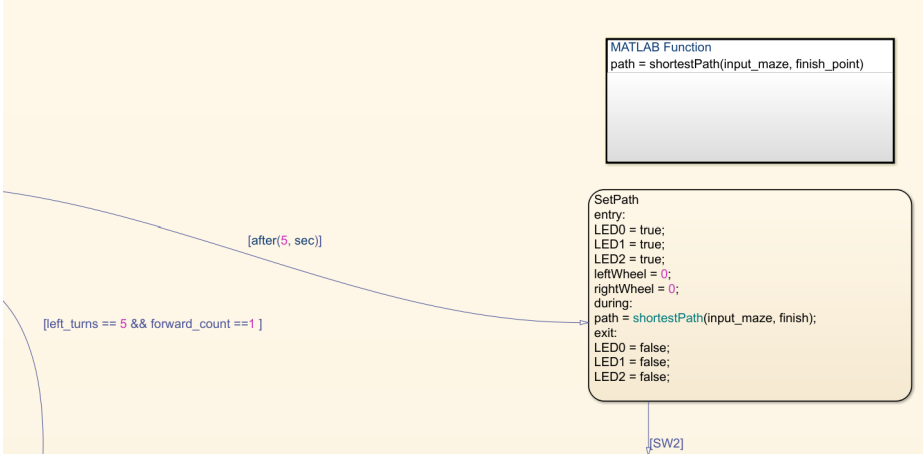


Figure 15: After finished navigation, optimization algorithm

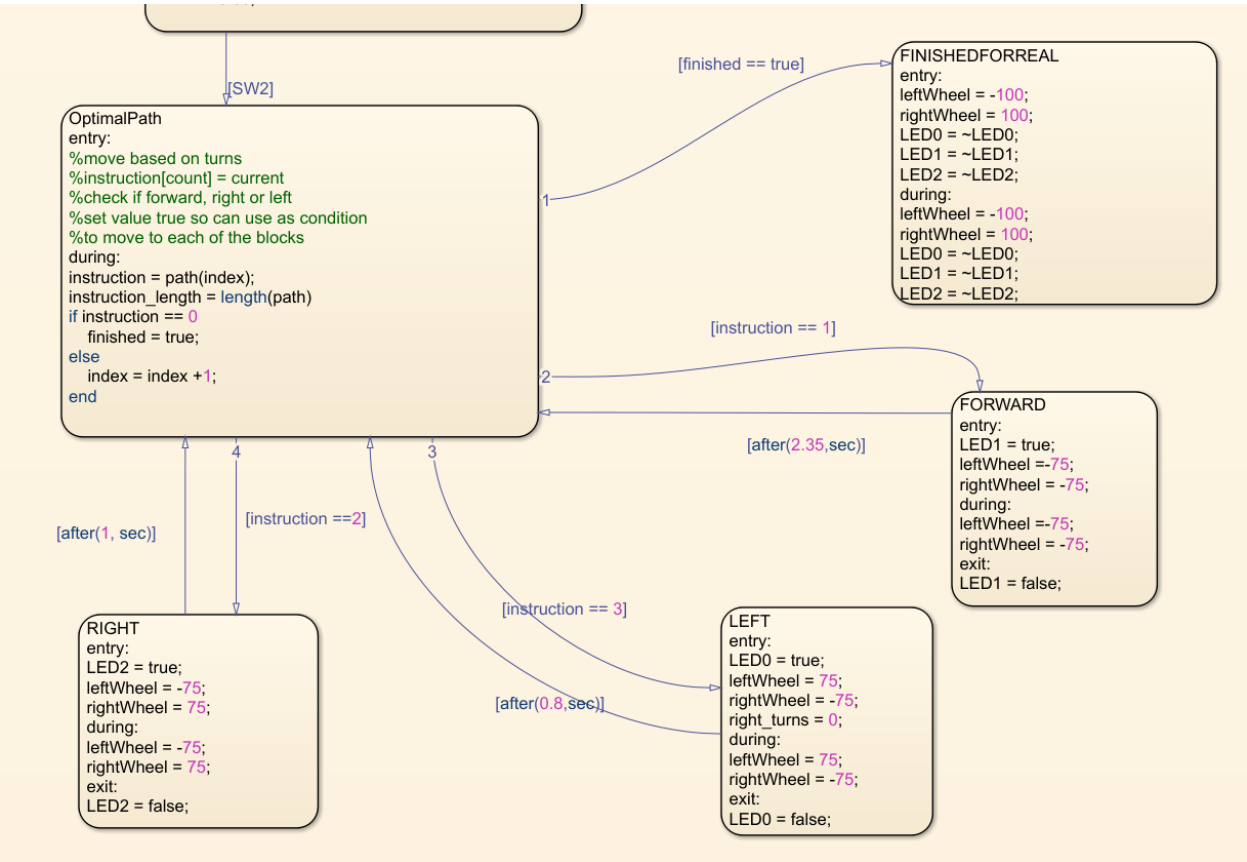


Figure 16: Code to move based on the optimal path

Appendix B – Optimization Algorithm Results

The shortestPath algorithm is:

```
function path = shortestPath(input_maze, finish_point)
   % Define the 7x7 maze array with the input maze
   maze = input_maze;
   %start = [1, 1];  % Example start point
   path = zeros(1,100);
   finish = finish_point;
% Example start and finish points
start = [1, 1];
% Initialize the distance array with -1s
distance = -ones(7, 7);

% Define directions and corresponding bits (north, east, south, west)
directions = [-1, 0, 0; 0, 1, 1; 1, 0, 2; 0, -1, 3]; % [dx, dy, bit]

% BFS queue starting from the finish position
queue = zeros(49, 2);  % Preallocate for maximum possible size (7x7)
queue(1, :) = finish;
queue_idx = 1;  % Tracks the next available spot in the queue
distance(finish(1), finish(2)) = 0;  % Set distance of finish point to 0
begin = 1;  % Initialize begin index

while begin <= queue_idx
   % Get current position
   x = queue(begin, 1);
   y = queue(begin, 2);
   current_distance = distance(x, y);
   begin = begin + 1;  % Increment begin
```

```
    % Check each possible direction (north, east, south, west)
    for i = 1:size(directions, 1)
        dx = directions(i, 1);
        dy = directions(i, 2);
        bit = directions(i, 3);
        nx = x + dx; % New x
        ny = y + dy; % New y

        % Ensure the new position is within bounds and not blocked by a wall
        if nx >= 1 && nx <= 7 && ny >= 1 && ny <= 7
            % Check if there is a wall in the direction (bit set to 1)
            if ~bitand(maze(x, y), bitshift(1, bit))  % Check if the wall bit is not set
                if distance(nx, ny) == -1  % Check if it hasn't been visited
                    distance(nx, ny) = current_distance + 1;
                    queue_idx = queue_idx + 1;  % Increment queue index
                    queue(queue_idx, :) = [nx, ny];  % Append new position to queue
                end
            end
        end
    end
end
disp(distance);

% Define direction codes and vectors
direction_codes = [0, 1, 2, 3];  % Corresponding numeric codes for directions
direction_vectors = [-1, 0; 0, 1; 1, 0; 0, -1];  % [north; east; south; west]

% Initialize the start position
current_pos = start;  % Ensure this is a [1 x 2] array
instructions = -ones(1, 100);  % Preallocate instruction list for up to 100 instructions
```

```
instruction_idx = 1;  % Keep track of the next instruction index

while true
  x = current_pos(1);
  y = current_pos(2);
  current_distance = distance(x, y);

  % Check if we reached the finish point
  if current_pos(1) == finish(1) && current_pos(2) == finish(2)
    break;  % Exit if we reached the finish
  end

  next_pos = [0,0];  % Initialize next_pos as an empty array

  % Check each direction to find the next step in the path
  for d = 1:length(direction_codes)
    nx = x + direction_vectors(d, 1);
    ny = y + direction_vectors(d, 2);

    % Check bounds and valid movement
    if nx >= 1 && nx <= 7 && ny >= 1 && ny <= 7
      if distance(nx, ny) == current_distance - 1  % Move towards lower distance
        next_pos = [nx, ny];  % Ensure next_pos is always [1 x 2]
        break;
      end
    end
  end

  % Determine movement direction based on current and next position
  delta_x = next_pos(1) - current_pos(1);
```

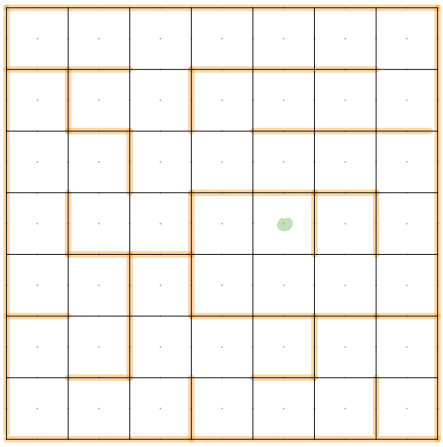
```
    delta_y = next_pos(2) - current_pos(2);

    % Update instructions based on direction codes
    if delta_x == -1  % Moving north
        instructions(instruction_idx) = direction_codes(1);  % 0
    elseif delta_x == 1  % Moving south
        instructions(instruction_idx) = direction_codes(3);  % 2
    elseif delta_y == 1  % Moving east
        instructions(instruction_idx) = direction_codes(2);  % 1
    elseif delta_y == -1  % Moving west
        instructions(instruction_idx) = direction_codes(4);  % 3
    end

    instruction_idx = instruction_idx + 1;  % Increment instruction index
    current_pos = next_pos;  % Update current position
end
disp(instructions);

     % Define initial state
    current_direction = 1;  % Starting direction (east = 1)
    move_instructions = zeros(1, 100); % Preallocate movement instructions list
    move_idx = 1; % Index to keep track of where to add the next instruction
    forward = 1;
    right = 2;
    left = 3;

    % Iterate through the direction instructions
    for i = 1:length(instructions)
        target_direction = instructions(i);
```

```
if target_direction == -1
   break;
end

% Calculate the difference between current and target directions
diff = target_direction - current_direction;

% Determine turning and movement based on the difference
if diff == 0
   move_instructions(move_idx) = forward;  % move forward
   move_idx = move_idx + 1;  % Increment the index
elseif diff == 1 || diff == -3
   move_instructions(move_idx) = right;  % turn right
   move_idx = move_idx + 1;
   move_instructions(move_idx) = forward;  % move forward after turn
   move_idx = move_idx + 1;
   current_direction = target_direction; % Update direction
elseif diff == -1 || diff == 3
   move_instructions(move_idx) = left;  % turn left
   move_idx = move_idx + 1;
   move_instructions(move_idx) = forward;  % move forward after turn
   move_idx = move_idx + 1;
   current_direction = target_direction; % Update direction
elseif diff == 2 || diff == -2
   move_instructions(move_idx) = right;  % turn right twice (equivalent to turn around)
   move_idx = move_idx + 1;
   move_instructions(move_idx) = right;  % turn right again
   move_idx = move_idx + 1;
   move_instructions(move_idx) = forward;  % move forward after turn
   move_idx = move_idx + 1;
```

```
        current_direction = target_direction; % Update direction
      end
    end


    path = move_instructions;
    end
```

Three mazes are run through this algorithm to show it can find the optimal path based on the information from navigation if there are no errors.

Maze 1: Maze from the demo



```
maze = [
    0b1101, 0b10101, 0b0001, 0b10101, 0b10101, 0b10101, 0b0011;
    0b0000, 0b0000, 0b0000, 0b1001, 0b10101, 0b10101, 0b0110;
    0b0000, 0b0000, 0b0000, 0b0100, 0b10101, 0b10101, 0b0011;
    0b0000, 0b0000, 0b0000, 0b1001, 0b0011, 0b1111, 0b1010;
    0b0000, 0b0000, 0b0000, 0b1100, 0b0100, 0b0100, 0b0010;
    0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000;
    0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b0000];
finish  = [4,5]
```

The following outputs from this are:

The distance array:

```
   19   18   17   16   15   14   13
```

-1 -1 18 9 10 11 12

-1 -1 9 8 7 6 5

-1 -1 -1 1 0 3 4

-1 -1 -1 2 1 2 3

-1 -1 -1 -1 -1 -1 4

-1 -1 -1 -1 -1 -1 -1

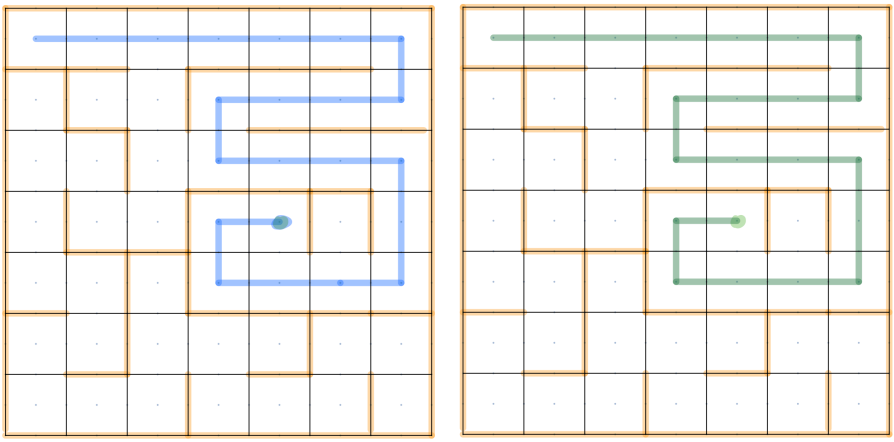
North, South, East, West Instructions:

1 1 1 1 1 1 2 3 3 3 2 1 1 1 2 2 3 3 0 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

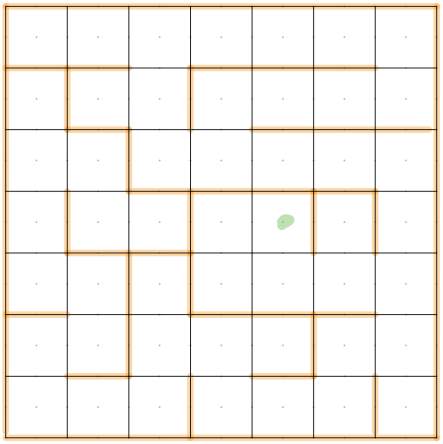
Final Movement Instructions:

1 1 1 1 1 1 2 1 2 1 1 1 3 1 3 1 1 1 2 1 1 2 1 1 2 1

Based on this, the original movement is shown in blue and the optimized path is shown in green. The shade on these paths shows if it had to go back over the same path.



Maze 2: Similar to the demo



```
maze = [
   0b1101, 0b10101, 0b0001, 0b10101, 0b10101, 0b10101, 0b0011;
   0b1011, 0b0000, 0b0000, 0b1001, 0b10101, 0b10101, 0b0110;
   0b1000, 0b0011, 0b0000, 0b0100, 0b10101, 0b10101, 0b0011;
   0b1010, 0b1100, 0b0111, 0b1001, 0b0011, 0b1111, 0b1010;
   0b1100, 0b0011, 0b1011, 0b1100, 0b0100, 0b0100, 0b0010;
   0b1001, 0b0110, 0b1000, 0b0001, 0b0111, 0b1001, 0b0010;
   0b1100, 0b0101, 0b0110, 0b1100, 0b0101, 0b0110, 0b1110;
];
finish  = [4,5]
```

<u>The following outputs from this are:</u>

The distance array:

```
  19   18   17   16   15   14   13
  20   -1   18    9   10   11   12
  19   20    9    8    7    6    5
  18   21   22    1    0    3    4
  17   16   11    2    1    2    3
  14   15   10    9   10    5    4
  13   12   11    8    7    6    5
```

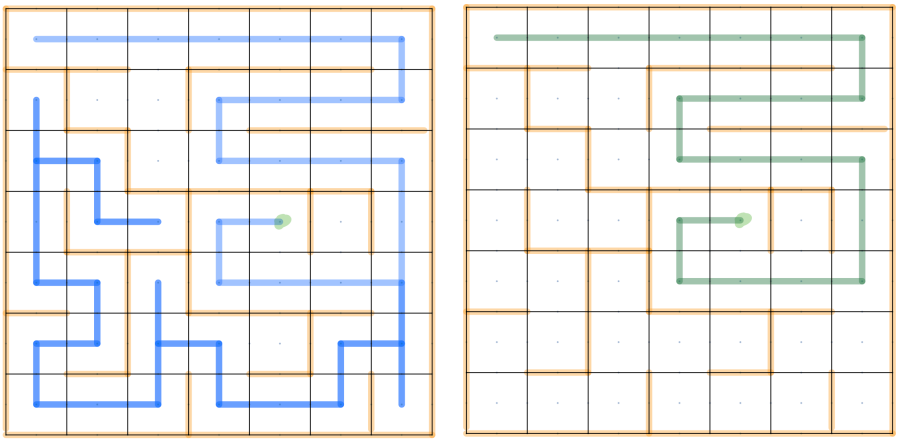
North, South, East, West Instructions:

1 1 1 1 1 1 2 3 3 3 2 1 1 1 2 2 3 3 0 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

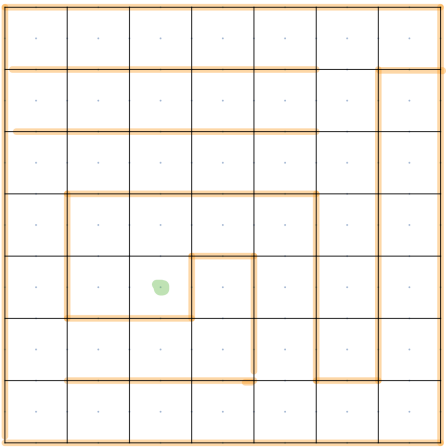
Final Movement Instructions:

1 1 1 1 1 1 2 1 2 1 1 1 3 1 3 1 1 1 2 1 1 2 1 1 2 1

Based on this, the original movement is shown in blue and the optimized path is shown in green. The shade on these paths shows if it had to go back over the same path.



Maze 3: Different maze



maze = [

0b1101, 0b0101, 0b0101, 0b0101, 0b0101, 0b0001, 0b0111;

0b0000, 0b0000, 0b0000, 0b0000, 0b0000, 0b1010, 0b0000;

0b1001, 0b0101, 0b0101, 0b0101, 0b0101, 0b0010, 0b0000;

0b1010, 0b1001, 0b0001, 0b0101, 0b0011, 0b1010, 0b0000;

0b1010, 0b1100, 0b0110, 0b1011, 0b1010, 0b1010, 0b0000;

0b1000, 0b0101, 0b0101, 0b0110, 0b1010, 0b1110, 0b0000;

0b1100, 0b0101, 0b0101, 0b0101, 0b0100, 0b0000, 0b0000];

finish = [5, 3];

<u>The following outputs from this are:</u>

The distance array:

26 25 24 23 22 21 22

-1 -1 -1 -1 -1 20 -1

14 15 16 17 18 19 -1

13 2 1 2 3 20 -1

12 1 0 15 4 21 -1

11 12 13 14 5 22 -1

10 9 8 7 6 7 -1

North, South, East, West Instructions:

1 1 1 1 1 2 2 3 3 3 3 3 2 2 2 2 1 1 1 1 0 0 0 3 3 2 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

Final Movement Instructions:

1 1 1 1 1 2 1 1 2 1 1 1 1 1 3 1 1 1 1 3 1 1 1 1 3 1 1 1 3
1 1 3 1

Based on this, the original movement is shown in blue and the optimized path is shown in green. The shade on these paths shows if it had to go back over the same path.