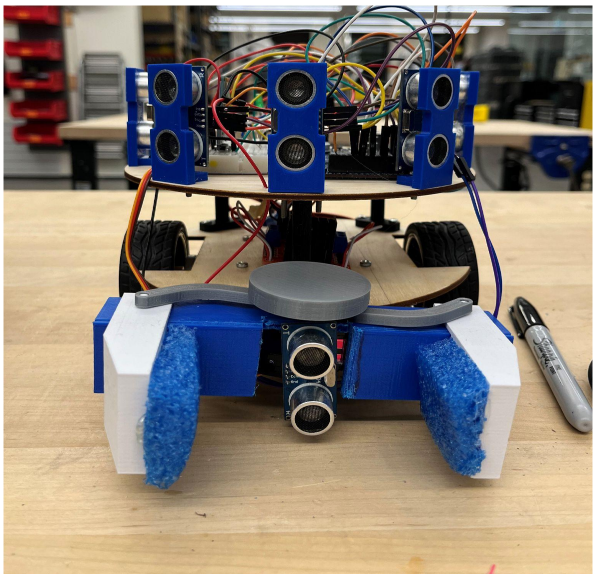
# MIE 444 – Mechatronics Principles

## Project Team 03 Final Report

## Autonomous Rover

08/12/2022



*Group Members*

*Miguel Gomez - 1005138628*

*Diego Gomez - 1005139690*

*Kevon Seechan - 1004941708*

*Zishan Karmali – 1004756365*

***Google Drive Link for Videos and Code***
https://drive.google.com/drive/folders/1OGMEgJjwYh1BuncgKAjX4zdO5-zs099Y?usp=sharing

## EXECUTIVE SUMMARY

The goal of the project was to design and program an electro-mechanical rover that can autonomously navigate within a maze, avoid obstacles, localise from a random starting point and pick/place a block. The rover was able to complete all of these tasks in a time of 2 minutes and 30 seconds.

The design of the rover is split into two 7in circular levels. The upper level contains a breadboard, an arduino mega, LEDs and 6 ultrasonic sensors. The lower level is made up of the wheels, motors and a clamping mechanism. The maze is 4ft x 8ft and contains a cross-roads, a loading zone and 4 delivery zones.

The obstacle avoidance algorithm utilised 3 forward facing sensors on the rover's upper level. These sensors were used to keep a minimum distance away from the walls such that the rover was able to navigate corners and drive relatively centred within the maze walls.

The localization algorithm took advantage of the unique cross-roads within the maze. Of the 6 ultrasonic sensors on the rover's upper level, 4 of them are placed at 'cardinal' locations. These sensors combined with the knowledge of the maze dimensions allowed the rover to detect that it was at the cross-roads. The rover simply avoided obstacles until it arrived at the cross-roads upon which it turned on its green LED and oriented itself to face the loading zone.

The block pick/place algorithm followed directly after localization. Once localised, the rover used its obstacle avoidance feature to drive to the loading zone. It detects this zone by using its 4 cardinal sensors in a similar manner to how it detects the cross-roads. Once at the loading zone, the rover searched for a block by sweeping right to left until an ultrasonic sensor embedded within its clamp mechanism detects a block. The block is then picked up and the rover reuses its obstacle avoidance to navigate the maze and search for 1 of the 4 delivery zones.

In completing the project, the team acquired a wide array of skills associated with mechatronics. These include the ability to wire circuits, write code and select appropriate components. Additionally, all group members learned to quickly troubleshoot and develop creative solutions to challenging circumstances.

# 1.0 ROVER CONTROL STRATEGY

The rover was controlled using arduino scripts loaded onto an arduino mega board. This section describes the development of those scripts, the integration between software and hardware and the testing of algorithms for obstacle avoidance, localization and block delivery.

## *1.1 OBSTACLE AVOIDANCE STRATEGY*

Obstacle avoidance forms the basis of the rover's functionality within the maze. It acts as a foundation for localization, navigation and block delivery.

### 1.1.1 Initial Obstacle Avoidance Strategy

The first iteration of the obstacle avoidance strategy was developed using the MATLAB SimMeR simulator. The size, shape and sensor placements of the rover were recreated on the simulator and 2% noise was added to help match real life sensor performance. The team then identified the 5 different scenarios that the rover could encounter in the maze, namely: driving forward, right turns, left turns, u-turns and the option of both left and right turns. Code was then written to control the rover's movement for each of these scenarios. A detailed explanation of this algorithm can be found in appendix A.

The MATLAB code was repeatedly tested and adjusted until the simulated rover avoided the maze walls (*see 'Simulated Obstacle Avoidance' video in the 'Simulations' folder*). This code was then rewritten onto an arduino script and loaded onto the rover's arduino board. When the rover was tested in the real maze, the algorithm did not work as well as expected. It repeatedly got stuck and/or crashed into walls (*see 'Initial OA' video in Obstacle Avoidance folder*). The team determined that this was due to a combination of sensor errors, rover shape and the complexity of having numerous scenarios for the rover to consider.

### 1.1.2 Final Obstacle Avoidance Strategy

On reflecting on the rovers initial performance, the main challenges associated with obstacle avoidance were found to be driving straight while keeping a safe distance from the walls and avoiding hitting the walls when turning / rotating. To account for these challenges the following steps were undertaken.

Firstly, the rover's footprint was reduced by 1in and changed from an octagonal shape to a circular shape (*see appendix B*). Secondly, different drive functions were created for the rover. These included 'drive_straight', 'drive_break', 'turn_left' and 'turn_right'. The left and right turn functions were created such that the rover rotates on the spot, this helped reduce the potential for wall collisions when navigating corners / deadends.

Additionally, all the functions are integrated with delays such that the rover can be programmed to, for example, drive straight for 10 seconds (*see appendix C*). Lastly, to reduce complexity, the obstacle avoidance algorithm was rewritten as a callable arduino function that utilises only the 3 front ultrasonic sensors on the arduino upper level. Namely, U1 (front) which points directly forward, U2 (right-front) and U6 (left-front) which also face forward but are angled between 30-35 degrees. The team placed the rover at multiple positions in the maze and at multiple different angles in order to determine a minimum distance for each of the sensors that would allow the rover to drive without hitting the walls. The final distance values were 17cm for U1 and 10cm for U2 and U6. These values were initialised at the beginning of the arduino script. See figure 1 below for clarification.

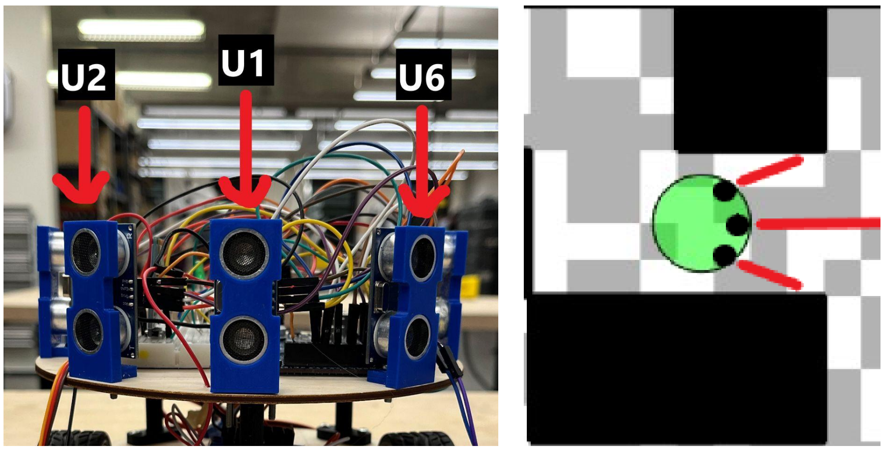


*Figure 1: Placement of U1, U2 and U6*

Figure 2 below details the pseudo code for the final obstacle avoidance algorithm.

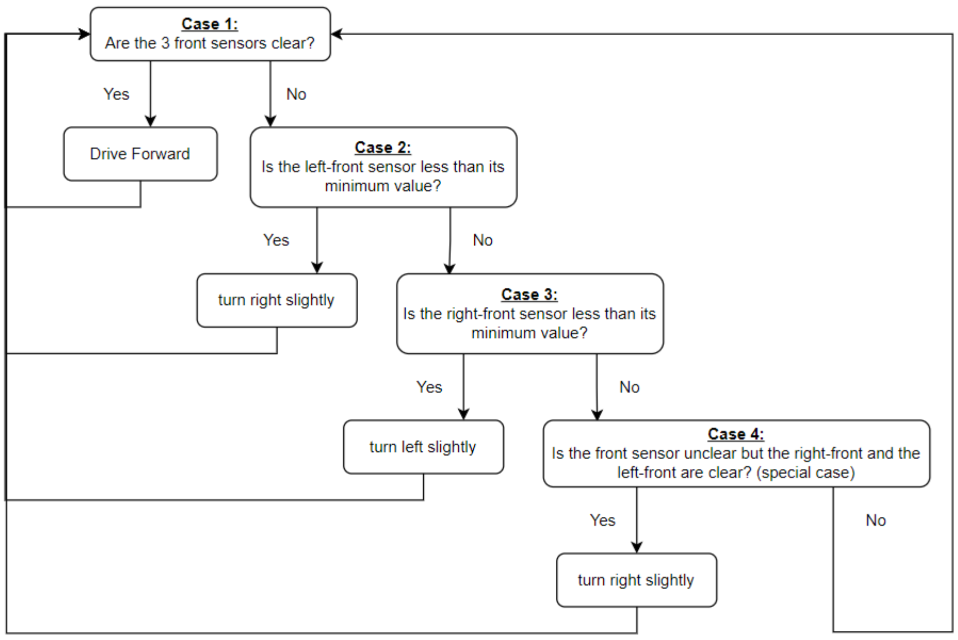

Case 1:
Are the 3 front sensors clear?
Yes
No
Drive Forward
Case 2:
Is the left-front sensor less than its minimum value?
Yes
No
turn right slightly
Case 3:
Is the right-front sensor less than its minimum value?
Yes
No
turn left slightly
Case 4:
Is the front sensor unclear but the right-front and the left-front are clear? (special case)
Yes
No
turn right slightly

*Figure 2: Pseudo Code of Final Obstacle Avoidance Strategy*

Arduino scripts can be split into 3 sections: variable initialization, code that runs once and code that is looped. The obstacle avoidance function is called within the looping code. At the beginning of the loop, the distances seen by U1, U2 and U6 are recorded. These values are then passed through all 4 cases in sequential order. If any of the cases are found to be true, the rover executes the respective code and another iteration of the loop is performed.

Case 1 represents the situation where all the values are greater than their minimum values. If this is true, the ‘drive_straight’ function is called and the rover drives forward in the maze *(see ‘Rover Driving Forward’ video in Obstacle Avoidance folder)*.

If Case 1 is false, then the values are passed into Case 2 which tests whether the rover is tilted to the left by comparing U6 (left-front) with its minimum value. If Case 2 is true, the ‘drive_break’ function is called for 300ms and the ‘turn_right’ function is called for 150ms. This acts as an adjustment to the right and is repeated as long as U6 is less than its minimum value. In other words, if the rover is tilted to the left, the code for Case 2 adjusts the rover until it is realigned with the maze walls. If Case 2 is false, then the values are passed into Case 3 which tests whether the rover is tilted to the right by comparing U2 (right-front) with its minimum value. If Case 3 is true, the ‘drive_break’ function is called for 300ms and the ‘turn_left’ function is called for 300ms. Similarly to case 2, this code is repeated until the rover is realigned with the maze walls *(see ‘Rover left and right adjustment’ video in Obstacle Avoidance folder)*.

Note that the ‘turn_left’ function of Case 3 is programmed to last twice as long as the ‘turn_right’ function of Case 2. This is done purposefully to ensure the rover does not get stuck in a loop at corners or intersections *(see ‘Rover Navigating Corner’ video in Obstacle Avoidance folder)*.

The values are passed into case 4 only when all other cases are false. Case 4 is a special case that was found via real world maze testing. It was expected that when the rover is placed directly in front of a wall, all 3 front sensors will read values lower than their minimums and as such, Cases 2 and 3 will help the rover navigate. However, it was found that occasionally, when in front of a wall, U2 and U6 recorded values greater than their minimum while U1 recorded values lower than its minimum. In this scenario, the rover crashed into the wall. To correct his error, the code is written such that if this scenario occurs, the rover stops and rotates slightly to the right. This sets the left-front sensor to be smaller than its minimum, the code then enters Case 2 and the rover avoids crashing into the wall.

To visualise the rover’s full obstacle avoidance algorithm at work, refer to any one of the videos within the *‘Full Maze Runs’ folder*. The obstacle avoidance arduino function can be seen in appendix D.

### *1.2 LOCALIZATION AND NAVIGATION STRATEGY*

Localization is the ability of the rover to know where it is in the maze. Localization is a mandatory component for helping the rover navigate the maze. The figure below details the locations of the loading zones (L/Z), the delivery zones (B), the location of the cross-roads (indicated by the 'X') and a cardinal system.

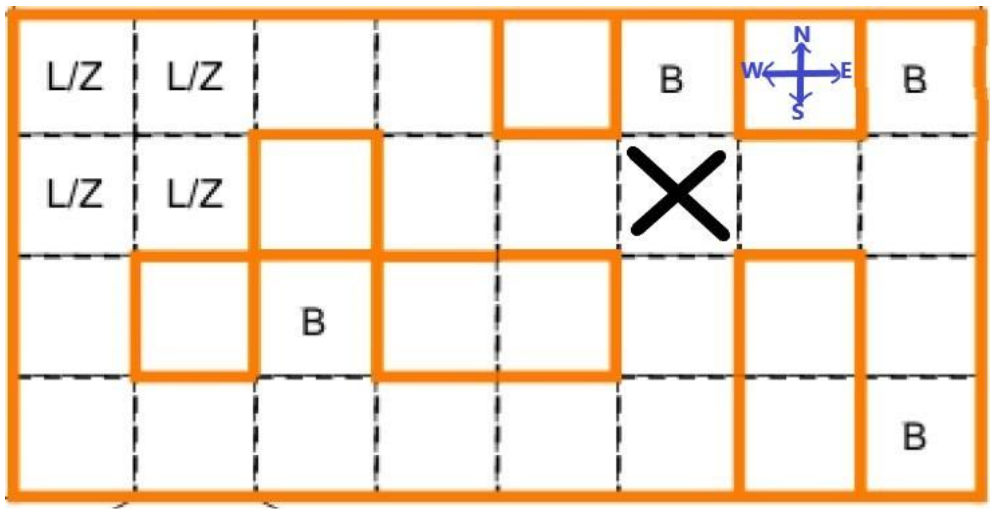


*Figure 3: Maze Diagram of Loading / Delivery Zones and Cross Roads*

#### 1.2.1 Localization

There are a number of methods that can be used to localise the rover within the maze. The team decided to employ a simple approach since the maze was known beforehand and was guaranteed to be unchanged. As seen in figure 3, the cross-roads represents a unique location in the maze and as such, the localization method was built around this location.

On the rover's upper layer, there are 4 'cardinal' ultrasonic sensors: U1 (front), U3 (right), U4 (back) and U5 (left). See figure 4 below for clarification.

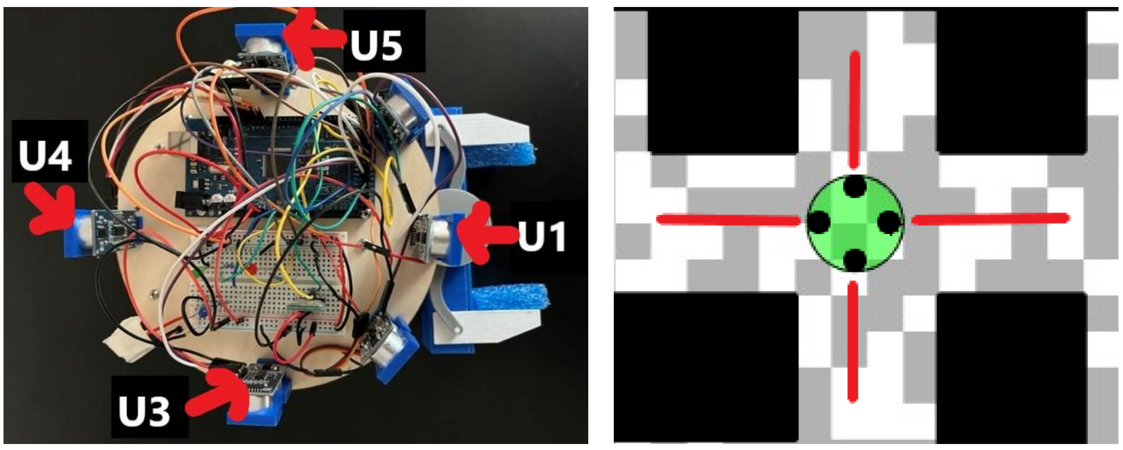


*Figure 4: Placement of 'Cardinal' Ultrasonic Sensors*

The obstacle avoidance strategy ensures that the rover is roughly centred as it drives through the maze. Thus, the team ignored any angled approaches to the cross-roads and assumed that the rover can only enter the cross-roads in 4 different orientations (seen in the diagram below where 'F' represents the front of the robot).

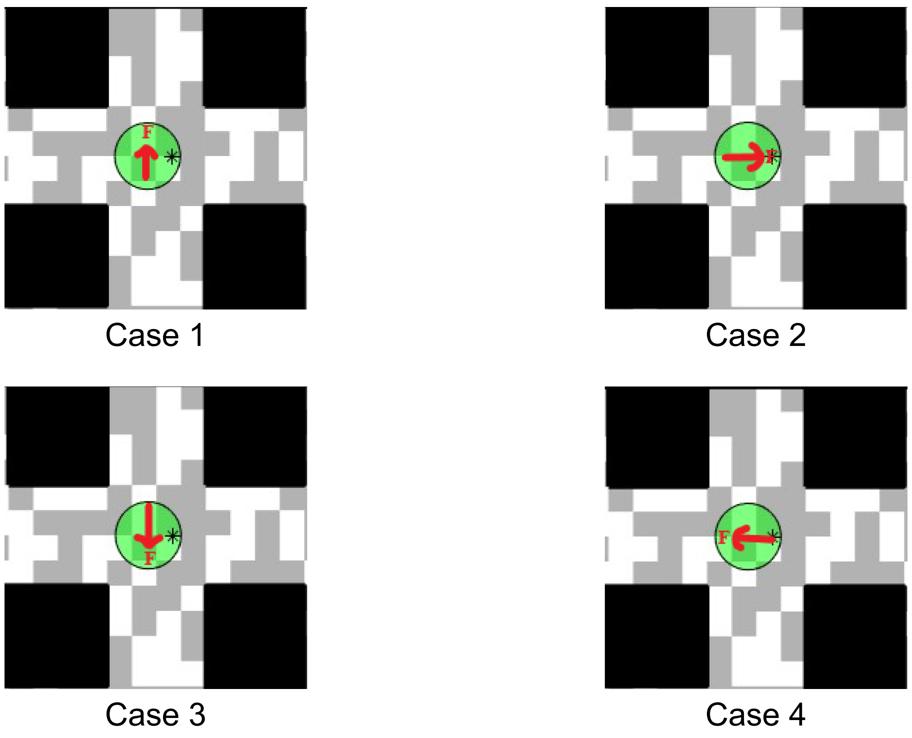


*Figure 5: The 4 ways the rover can approach the cross roads.*

The dimensions of each block of the maze was given to be 30cm x 30cm. Therefore it was possible to write code that accounted for each of the cases. For example, if the robot is in the case 1 orientation, the cardinal sensors should read approximately the following values: U1 - 30cm, U3 - 60cm, U4 - 60cm and U5 - 60cm. This is because U1 detects 1 block of clearance while U3, U4 and U5 detect 2 blocks of clearance. This logic was tested in the MATLAB simulator, the details of which can be found in appendix E. The simulator proved that the rover was able to identify it was at the cross-roads for each of the 4 cases. Therefore, the cross-roads was used as the localization point for the robot.

After sufficient simulator testing, the MATLAB code was adapted onto an arduino script and the rover was tested in the real world maze. In order to ensure the rover localises at a 100% success rate, it was placed at the cross-roads both on and off centre and in the 4 different orientations. This allowed for a range of sensor values to be obtained. For example, case 3 is true when U1, U3 and U5 are within a range of 55-75 cm (2 blocks clearance) and U4 is within a range of 30-45cm (1 block clearance). See appendix F for the different cases are their respective value ranges.

### 1.2.2 Navigation

To ensure fast rover performance and low code complexity, the rover's obstacle avoidance algorithm was reused to help it navigate to the loading zone. As explained in section 1.1, the obstacle avoidance algorithm is tied to the rover's forward movement. Therefore, once at the cross-roads, the rover simply needs to be pointed in the direction of the loading zone and, by using its avoidance function, it will navigate itself to the loading zone. To identify that the rover has arrived at the loading zone, the cardinal sensors are used once more. Consider figure 6 below.

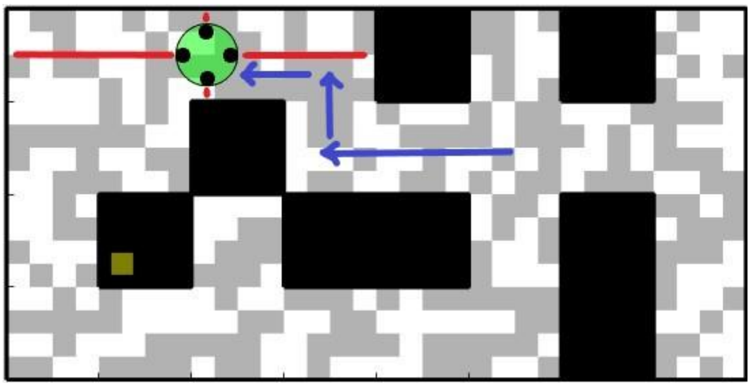


*Figure 6: Navigation to Loading Zone*

The blue arrows represent the path taken from the cross-roads. It can be seen that the block just before the loading zone represents another unique location. At this location, U1 and U4 detect approximately 2 blocks of clearance while U3 and U5 detect less than 10 cm of clearance.

### 1.2.3 Localization and Navigation Algorithm

Pseudo code for the localization and navigation algorithm can be seen below.

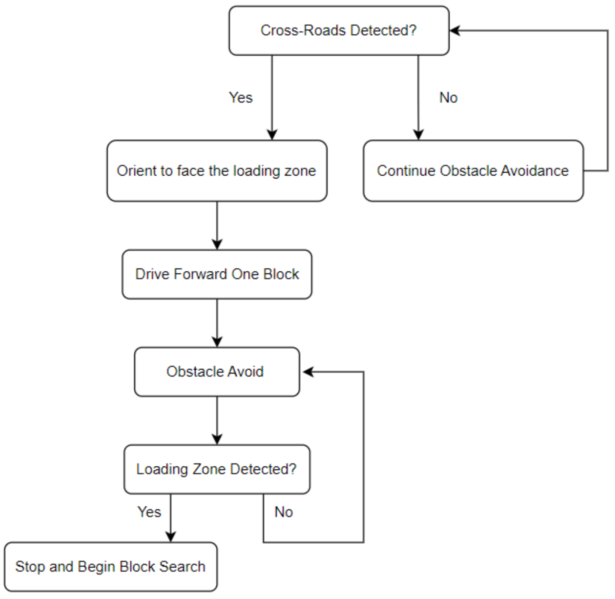


*Figure 7: Pseudo code for Localization and Navigation Algorithm*

The working of the algorithm is as follows. The rover performs obstacle avoidance as explained in section 1.1.2. However, at the beginning of the loop, the distances of all 6 sensors are now recorded instead of only U1, U2 and U6. With each iteration of the loop, ‘if statements’ describing each of the cases detailed in figure 5 are constantly being checked.

If the rover enters the cross-roads facing north (case 1), east (case 2) or south (case 3), the following code is executed. Firstly, the rover stops and its green LED is turned on for 2 seconds to indicate that the rover has recognised it is at the cross-roads and is now localised. Secondly, the rover orients itself to face the direction of the loading zone by rotating until U3 (its right sensor) detects a clearance of approximately 1 block (*to aid with visualisation, see ‘Rover Case 2 Localization’ video in the localization folder*).

If the rover enters the cross-roads facing west (case 4), the same code sequence is executed with the exception of the orientation code since the rover is already pointing towards the loading zone (*see ‘Rover Case 4 Localization’ video in the localization folder*).

Once properly oriented, the rover stops for 2 seconds, drives forward for 1 second to get out of the cross-roads and a boolean variable ‘state1’ is set to true. ‘State1’ is used to indicate that the rover is localised and pointing in the direction of the loading zone. While ‘state1’ is true, the rover employs its obstacle avoidance function until its cardinal sensors detect the loading zone. Upon arrival at the loading zone, the rover stops and turns its red LED on for 2 seconds.

To visualise the localization and navigation strategy at work, refer to the *‘Localization + Loading Zone Detection'* video in the localization folder.

### *1.3 BLOCK DELIVERY STRATEGY*

The block delivery strategy is built upon the obstacle avoidance and navigation / localisation strategies. The block pick up method used was a simple clamp mounted onto the rover, controlled by a servo motor. See figure 8 below.

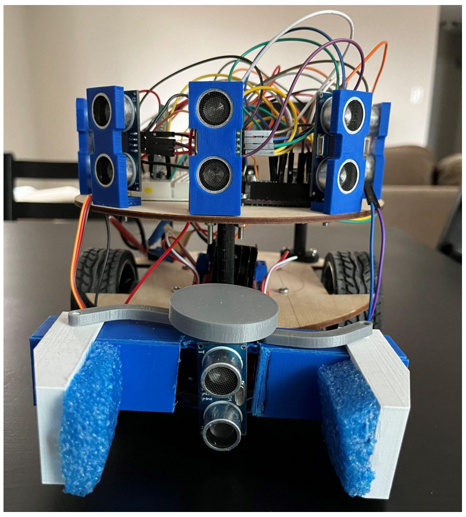


*Figure 8: Rover with clamp attached*

Note that this clamp mechanism was not the original design, but an improvised version that was developed due to a faulty servo motor. See appendix G for the original clamp design. Within the clamp, there is an ultrasonic sensor (U7) that is used exclusively for block detection.

The relative locations of the block was known beforehand, the team used this knowledge to simplify the block detection algorithm. The figure below shows the different potential block locations (represented by black squares), the drop-off zones (B) as well as the point where the rover begins its search (R). Note that to limit complexity, the team opted to program the rover to search for a delivery zone rather than programming the rover to deliver to specific zones. All the delivery zones share an identical condition, they are all dead-ends. As such, the rover uses its U1, U3 and U5 sensors to search for a drop of location.

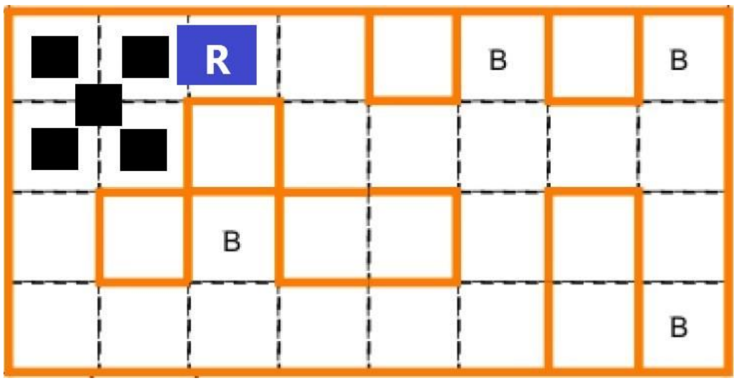


*Figure 9: Diagram of potential block locations*

Pseudo code for the block detection and delivery algorithm can be seen below.

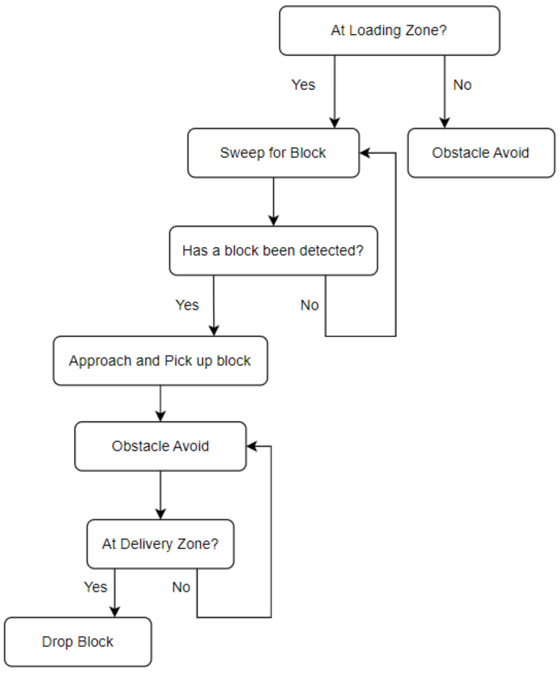


*Figure 10: Pseudo code for block detection and delivery*

As explained in section 1.2.2, the rover drives to the loading zone and stops at the location indicated by ‘R’ seen in figure 9. Once at this location, the following code is executed. Firstly, the rover drives forward slightly and then stops. Secondly, the rover checks its block detection sensor (U7). At the loading zone, U7 can either detect the maze wall roughly 2 blocks away (approximately 60 cm) or the block itself (the furthest block being less than 60cm away). Using this fact, the rover sweeps slowly to the left while U7 reads values greater than 35cm (i.e. while U7 sees the wall, the rover rotates slowly to the left).

If U7 records a value less than 35cm, this indicates that a block is present. The rover stops at this point and begins driving slowly towards the block. Once U7 reads a value less than or equal to 2cm, the block is deemed to be close enough to grab. The rover then stops, turns on its blue LED, uses the servo motor to close the clamps and a boolean variable ‘state2’ is set to true.

If ‘state2’ is true, ‘state1’ is set to false to ensure that the rover does not try to localise itself again. Next, the rover begins its search for a delivery zone. It does this by applying its obstacle avoidance function until U1, U3 and U5 detect the conditions of a delivery zone. If a delivery zone is found, the robot stops, turns on its red and blue LEDs for 2 seconds, then drops the block by opening the clamps.

To visualise the block pick up strategy at work, refer to the *‘Block Pick up and delivery’* video in the block pick and place folder.

## *1.4 INTEGRATION*

The integration of the obstacle avoidance, localization and block delivery strategies are best explained by describing the structure of the arduino code. At the very beginning of the code, all the necessary variables are initialised. These include initialising the ultrasonic sensors with the ‘New Ping’ library, establishing the minimum distances for obstacle avoidance and setting pins for the LEDs and motors.

The second aspect of the code is the ‘setup’ function where code that is only required to run once is placed. Within this function, the boolean variables (‘state1’ and ‘state2’) are set to false, the motor and LED pins are set as ‘outputs’, serial communications are established and the servo motor is configured such that the clamps are open.

The third aspect of the code are the different functions created to control the rover. In total, 6 functions were created, 4 of which are the drive functions detailed in appendix C. The other 2 functions are the obstacle avoidance function (explained in section 1.1) and the block search function (explained in section 1.3).

The final aspect of the code is the ‘loop’ function. This code is repeated infinitely in sequential order. The details of the rover’s loop function is as follows. At the beginning of the loop, the distances detected by all the sensors are recorded. Next, the obstacle avoidance function is called to help the rover navigate the maze. After, the 4 ‘if statements’ that check the different localization cases are listed. Following is an ‘if statement’ that checks whether ‘state1’ is true and the final line of code is an ‘if statement’ that checks whether ‘state2’ is true.

Figure 11 below details the pseudo code of the loop function of the rover control code. It can clearly be seen that the integration of the different strategies is very linear in nature. This was purposefully done to decrease complexity without sacrificing performance.

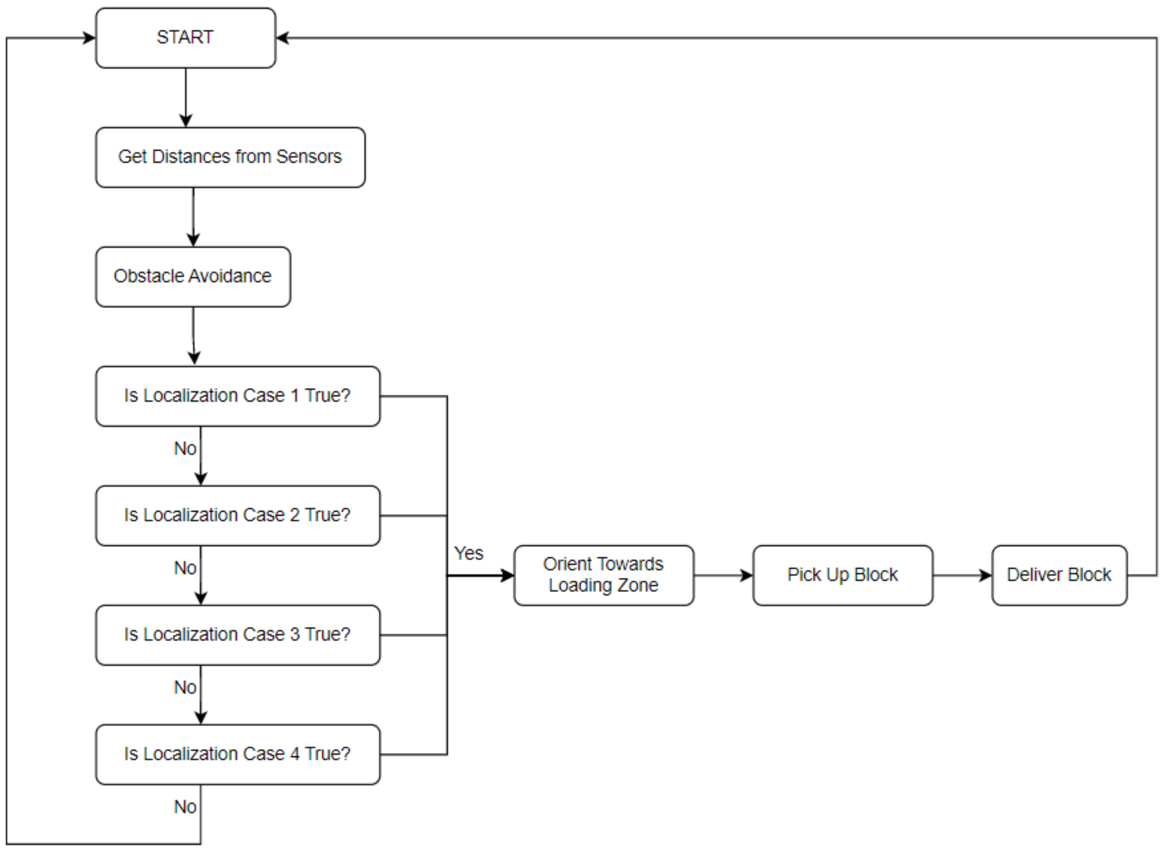

START
Get Distances from Sensors
Obstacle Avoidance
Is Localization Case 1 True?
No
Is Localization Case 2 True?
No
Is Localization Case 3 True?
No
Is Localization Case 4 True?
No
Yes
Orient Towards Loading Zone
Pick Up Block
Deliver Block

*Figure 11: Pseudo Code of Rover Control Loop*

## 2.0 FINAL RESULTS

The rover's deliverables were broken down into three milestones, each building on the previous. This section describes the rover's performance for each of the milestones.

### *2.1 OBSTACLE AVOIDANCE*

For milestone 1, the rover was required to drive autonomously for at least 20 feet in the maze without hitting any obstacle within a time period of 5 minutes. The team was given two trials with roughly an hour break in between each trial. The first trial can be seen in the *'M1 Trial 1'* video in the *'Milestone 1'* folder. The rover started well by travelling roughly 4ft without hitting any walls. However its performance deteriorated dramatically soon after as it began scraping and colliding with the walls.

The team tried to improve the rover performance during the 1 hour gap but was unsuccessful. The second trial can be seen in the *'M1 Trial 2'* video in the *'Milestone 1'* folder. In this trial, the rover simply rotated on the spot indicating that there was a clear error in the code. Overall, milestone 1 can be described as a failure.

### *2.2 LOCALIZATION*

For milestone 2, the rover was required to localise itself from a random starting point, drive to the loading zone and navigate to a delivery zone all while avoiding obstacles and within a time frame of 5 minutes. Similarly to milestone 1, two trials were given with roughly an hour break in between. The first trial and second trials can be seen in the *'M2 Trial 1'* and 'M2 Trial 2' videos in the *'Milestone 2'* folder. The rover's obstacle avoidance was slightly improved from milestone 1 however, the rover still needed multiple adjustments. Despite the collisions, the rover was able to confirm with its LEDS that it localised, arrived at the loading zone and arrived at the delivery zone. Thus milestone 2 was passed.

### *2.3 LOAD PICK AND PLACE*

For milestone 3, the rover was required to complete a combination of all the tasks within 5 minutes. These tasks involved starting in a random maze location, driving while avoiding the walls, localising itself with confirmation, driving to the loading zone with confirmation, detecting and picking up the block with confirmation and delivering the block to a delivery zone with confirmation.
The rover, with some assistance, was able to successfully complete the various tasks in a time of 2 minutes and 34 seconds. The rover performance can be seen in the *'Rover Milestone 3'* and *'Rover Myhal Run'* videos in the *'Full Maze Runs'* folder.

## *2.4 INTEGRATION*

The tasks associated with the three milestones were smoothly integrated with each other. The code was constructed such that the rover had to execute the following tasks in sequential order:

1. Localize
2. Drive to the loading zone
3. Search for the block
4. Pick up the block
5. Search for a delivery zone
6. Drop off the block

The sequential nature of the code ensured low complexity which allowed for easy troubleshooting and fast rover performance.

With respect to hardware, the major components were the 7 ultrasonic sensors, 1 of which was used exclusively for block detection. The rover also utilised LEDs to provide confirmation of localization, block pick up, etc. These components were easy to mount onto the rover and incorporate into the code.

## 3.0 DISCUSSION

### *3.1 BEST FEATURES*

At the end of milestone 3, the rover was able to complete all of the given tasks with minimal assistance. The best features of the rover were found to be the obstacle avoidance algorithm, the speed at which the rover navigated the maze and the overall simplicity of both the code and physical structure of the rover.

A detailed explanation of the obstacle avoidance algorithm can be found in section 1.1.2. The team observed that once the minimum distances for the front sensors were appropriately adjusted, the rover avoided obstacles almost flawlessly. Additionally, the algorithm allowed for the rover to drive relatively central between the walls of the maze.

The two wheels of the rover were driven by two 6V DC motors that were set at roughly 20% of their max speed. The localization algorithm and block delivery algorithms benefited from this speed since it allowed the rover travel quickly across the maze to find the cross-roads and delivery zones. All of the tasks of milestone 3 were finished in approximately 2 minutes and 30 seconds which was ½ of the max maze time of 5 minutes.

Lastly, the design used the most basic components to operate which ensured that the total design cost was below budget. The components of the final design included 7 ultrasonic sensors, 1 servo motor, 1 arduino mega board, 1 small bread board and 2 DC motors. Additionally, the simplicity of the components also meant that very advanced code would not be required. This allowed the team to minimise the time spent writing code and maximise the time spent debugging and testing the rover in the real world.

### *3.2 WORST FEATURES*

Although the overall performance of the robot was satisfactory, the team identified three weaknesses involving both the physical components and the different algorithms.

Firstly, the algorithms relied on very precise sensor readings. Due to the inaccuracy of the ultrasonic sensors, they are not foolproof. For example, the inconsistency of the block detection sensor (U7) meant that the rover would sometimes fail to detect the block. This caused the rover to get stuck in a block search loop.

Secondly, the pick up mechanism was originally based around two servo motors; one to lower the claw and another to close the claw *(see appendix G)*.

The long arm of the original design created a large moment arm, which, when paired with the heavy claw, caused the servo motor at the base of the arm to fail. Therefore, the mechanism was redesigned and the claw was stuck to the bottom of the rover. However, this meant that the block would be dragged across the maze so a team member had to manually elevate the block before the gripper closed.

Lastly, the team opted for the robot to search for a block delivery zone. While this algorithm guaranteed that block delivery occurs within 5 minutes, it did not allow for a predetermined delivery zone to be set. This can be seen as a negative feature since in real life scenarios, a user would want control over where a rover delivers its load.

## *3.3 INITIAL ROVER DESIGN VS FINAL ROVER DESIGN*

The final design differed from the initial design in two main ways: rover footprint and ultrasonic sensor placement.

The initial rover had an 8in octagonal frame with each of the wheels contributing 1in on either side. In total, the rover had a 10in footprint. The idea behind the octagonal shape was to utilise its edges to align the ultrasonic sensors accurately on the rover. However, the size and shape of the footprint became an issue since the vertices of the frame would collide against maze corners while turning. Therefore, the team decided to reduce the footprint to 7in. This was done by reducing the frame itself to 7in and making cut-outs for the wheels so that they are within the frame. Additionally, the frame was changed from an octagonal shape to a circular shape to minimise the chance of corner collisions.

The initial rover had a total of 5 ultrasonic sensors on its top layer: 1 at the front, 2 on the right, 1 on the left and 1 at the back. There was also an additional ultrasonic at the front of the lower layer for block detection. The placement of these sensors were related to the initial obstacle avoidance algorithm. However, since the initial algorithm did not work, this sensor arrangement was scrapped. The final sensor placement had a total of 6 ultrasonic sensors on its top layer and 1 ultrasonic within its clamping mechanism *(see appendix B for design changes)*.

## *3.4 LEARNING EXPERIENCES*

The team developed a range of mechanical and electrical skills over the course of this project. The construction of the rover itself ensured that each group member acquired experience with using a drill, modifying 3-D printed parts to account for design discrepancies and turning scrap parts found in Myhal into valuable components.

Additionally, the complex levels of circuitry involved in wiring components such as the motors, the arduino mega board and the battery ensured that the team obtained a sense of proper power distribution and electrical safety.

At the beginning of the project, no group members had experience with arduino code. However, by studying the relationship between code and electrical connections and completing the 3 milestones, the team is now adequately prepared for future arduino projects.

Troubleshooting was also an important skill that was developed. Every time the rover would perform unexpectedly, it was the task of the team to first determine the various issues and then come up with solutions. For example, it was found that the initial clamp design caused one of the servo motors to stall. The team worked as a group to figure out why this happened and to redesign the gripper mechanism.

## *3.5 FUTURE PROJECTS*

There are various items the team would perform differently if a similar project is to be completed in the future. Firstly, more effort would be placed on understanding the physics behind a design before creating 3-D models. This will ensure that the mistake of the clamp failure does not happen again.

Secondly, the team would use more advanced components such as a lidar and time of flight sensors to create a more foolproof design. Additionally, the code complexity will be increased to ensure that all algorithms are robust. For example, instead of using a hardcoded localization approach a more complex method such as histogram localization or particle filters will be used.

Lastly, to allow for real time tracking of the rover, the bulk of the programming will be done on MATLAB or Python and communicated wirelessly instead of uploading code directly to the arduino board.

## 4.0 APPENDICES

### *Appendix A - Simulated Obstacle Avoidance*

In order to initialise the rover, the shape and size was first altered to match the proposed rover design, where the sensor placements would match in both cases. Once the rover was initialised a flow chart to summarise the pseudo-code was created, which can be seen in figure 12.

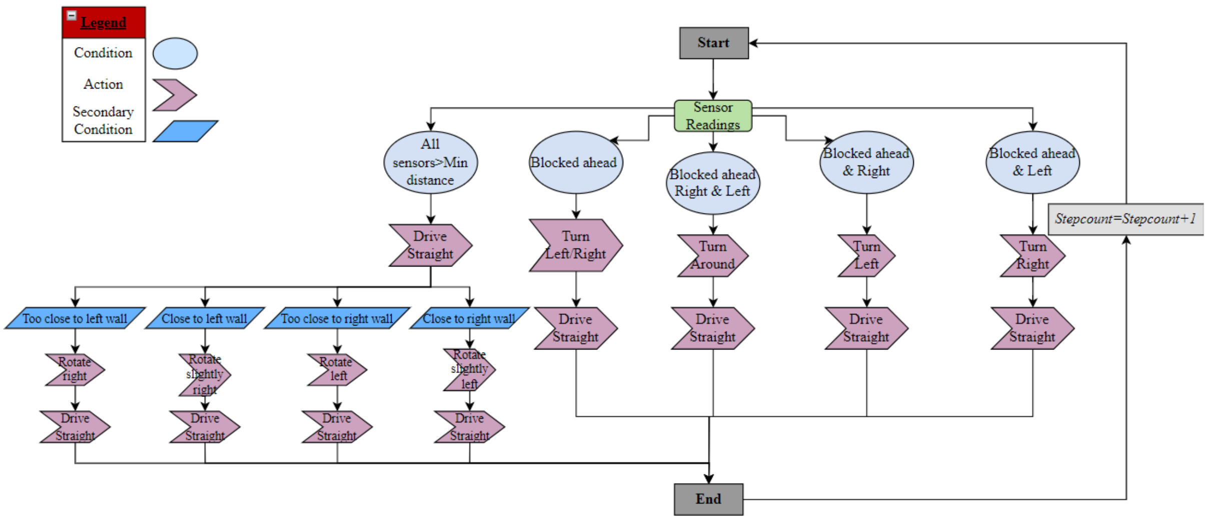


*Figure 12 - Initial Simulator Pseudo Code*

Once the pseudo-code had been converted into a runnable script, the logic conditions required numeric values in order to initiate corrective actions from the rover, in order to avoid the maze's obstacles. With the proper choice of numerical parameters, the risk of the rover colliding during operation would be mitigated. An example of the display during run time and summary of parameters are shown in figure 13 and table 01.

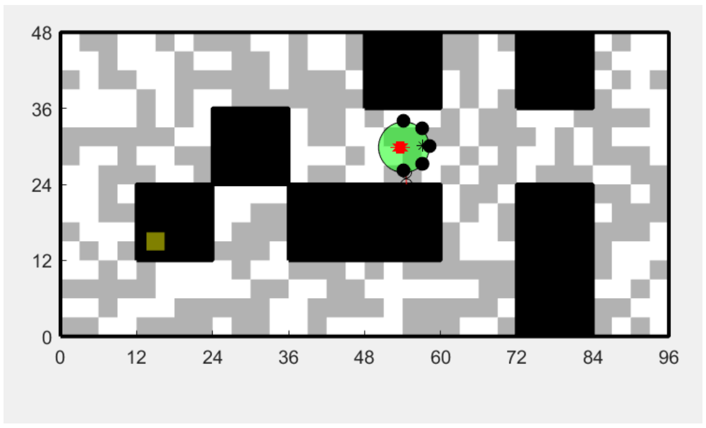


*Figure 13 - Initial Simulator Pseudo Code*

The black dots on the rover represent the ultrasonic sensors, labelling starting from the top going in a clockwise motion: U1, U2, U3, U4, U5. The current heading of the rover is 0 degrees, where anti-clockwise rotations indicate positive angle changes.

*Table 01 - Initial Simulator Pseudo Code*

| Parameter | Numerical Value |
|---|---|
| Rover Footprint (Circular) | 8 in |
| U1 Placement | (0, 4, 5, 90 °) |
| U2 Placement | (2.83, 2.83, 5, 45 °) |
| U3 Placement | (4, 0, 5, 0 °) |
| U4 Placement | (2.83, -2.83, 5, 315 °) |
| U5 Placement | (0, -4, 5, 270 °) |
| U1 Minimum Distance | 2.10 in |
| U2 Minimum Distance | 2.35 in |
| U3 Minimum Distance | 2.85 in |
| U4 Minimum Distance | 2.35 in |
| U5 Minimum Distance | 2.10 in |
| Rover Drive Forward Speed | 1.4 |
| Close Rotation | 5 ° |
| Too Close Rotation | 30 ° |
| Blocked Ahead & Left/Right | 90 ° |
| Blocked Ahead & Left & Right | 180 ° |

The parameters summarised above provided an acceptable result when judging against the goal for obstacle avoidance. During trial runs it was noticed that any collision would result in the rover being immobilised completely. The rover would attempt to correct its direction, however, given that each corrective statement involved a forward distance call no movement was permitted. In order to correct this, a negative speed call was implemented within each secondary condition such that the rover reversed, corrected its direction then proceeded forward.
Secondly, occasionally within the maze some or all of the ultrasonic sensors were within a range that had no defined operation, to rectify this an additional *elseif* statement was implored alongside three 'Error-Conditions'. These are shown in Figure 12. Additionally, the set speed of 1.4 was slightly too high than the optimal value, however, due to the time constraint of five minutes for the completion of Milestone 01 this speed was chosen.

Due to this relatively large movement in relation to the minimum wall distance parameters, during rotation operations the rover frequently overshot the ‘middle-route’ for the turn and collided with an obstacle. This issue was resolved by initiating the ultrasonic sensors to take measurements after each action. This reduced the overall speed of the rover, however, this was compensated by the speed setting of 1.4.

## *Appendix B - Change of Rover Design*

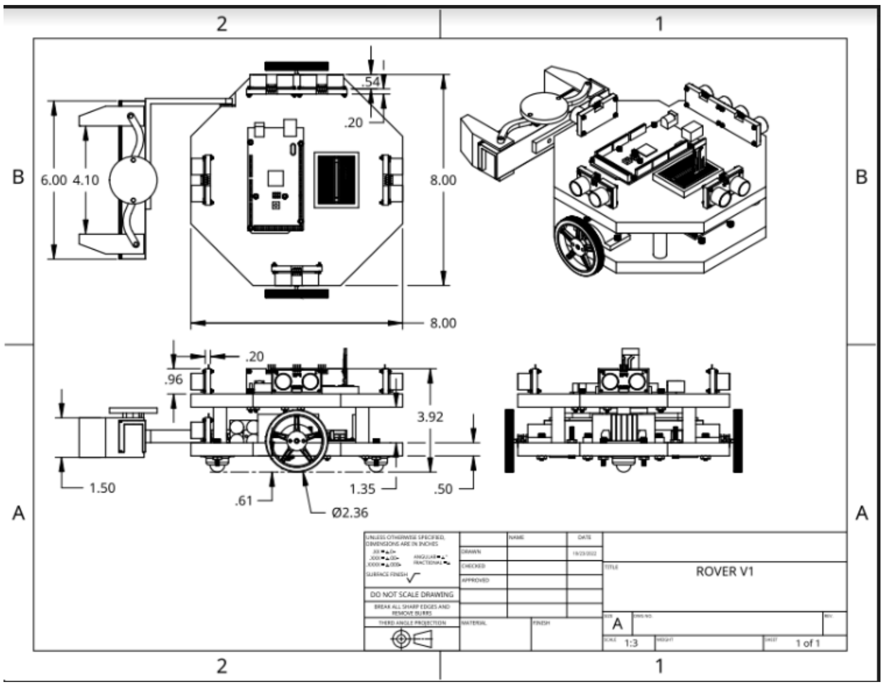


*Figure 14 - Rover Initial Design*

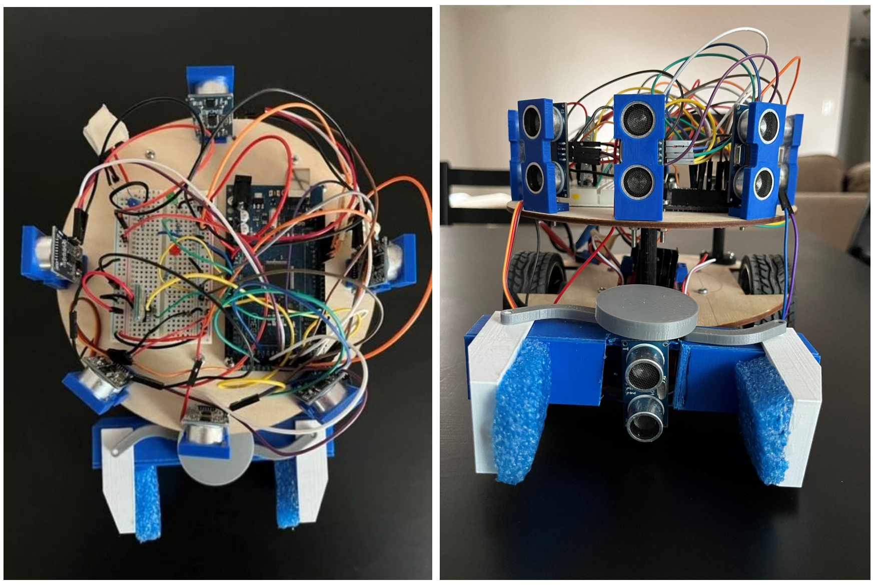
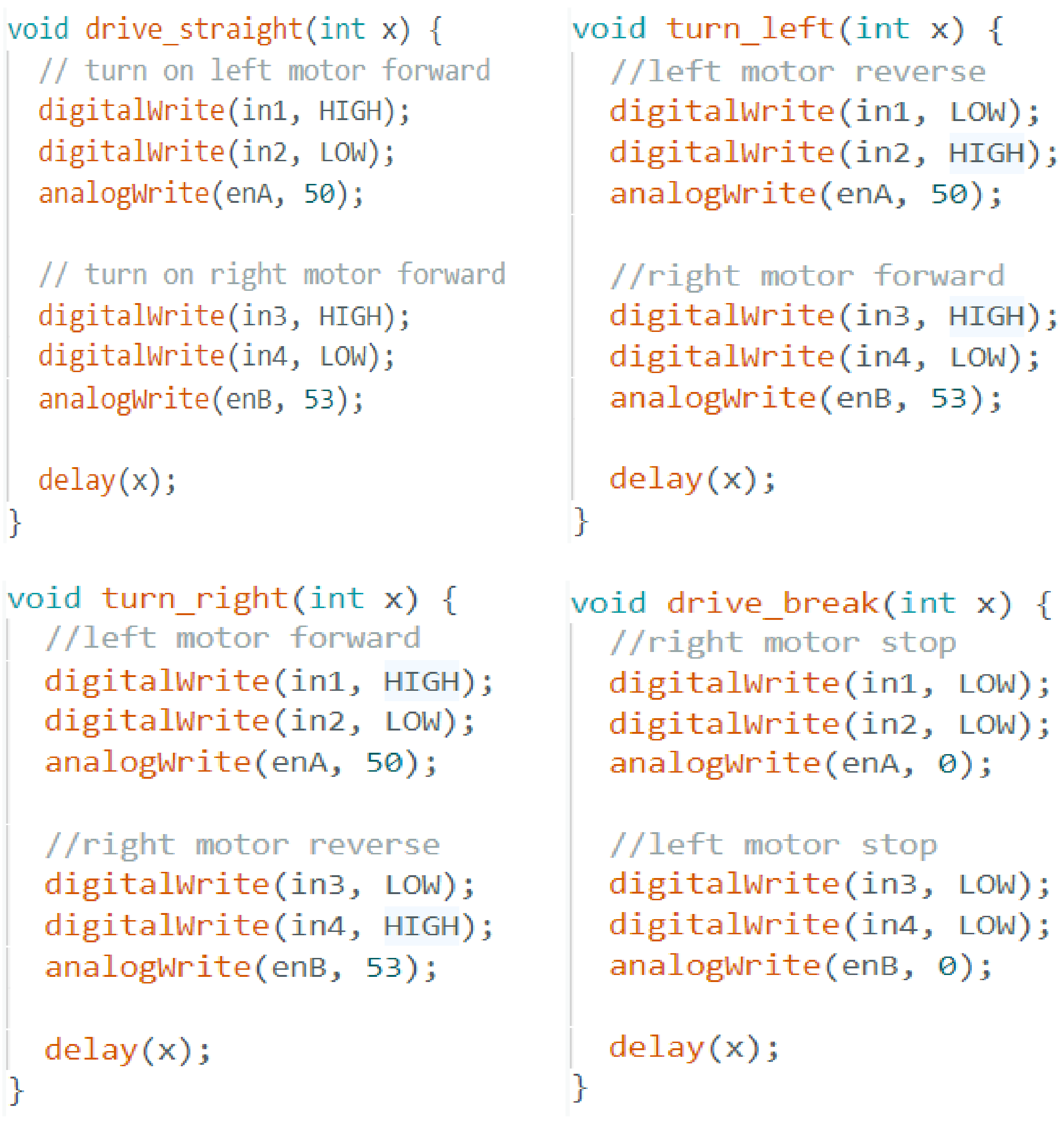


*Figure 15 - Final Rover Design*

## *Appendix C - Drive Functions*



```
void drive_straight(int x) {
  // turn on left motor forward
  digitalWrite(in1, HIGH);
  digitalWrite(in2, LOW);
  analogWrite(enA, 50);

  // turn on right motor forward
  digitalWrite(in3, HIGH);
  digitalWrite(in4, LOW);
  analogWrite(enB, 53);

  delay(x);
}
```

```
void turn_left(int x) {
  //left motor reverse
  digitalWrite(in1, LOW);
  digitalWrite(in2, HIGH);
  analogWrite(enA, 50);

  //right motor forward
  digitalWrite(in3, HIGH);
  digitalWrite(in4, LOW);
  analogWrite(enB, 53);

  delay(x);
}
```

```
void turn_right(int x) {
  //left motor forward
  digitalWrite(in1, HIGH);
  digitalWrite(in2, LOW);
  analogWrite(enA, 50);

  //right motor reverse
  digitalWrite(in3, LOW);
  digitalWrite(in4, HIGH);
  analogWrite(enB, 53);

  delay(x);
}
```

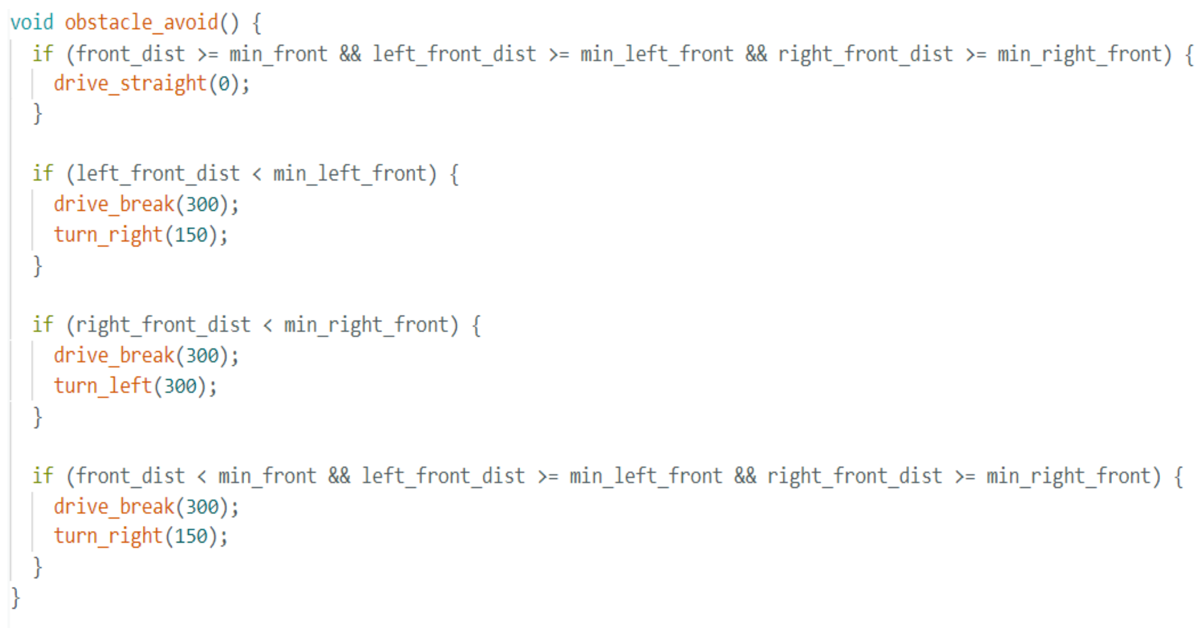
```
void drive_break(int x) {
  //right motor stop
  digitalWrite(in1, LOW);
  digitalWrite(in2, LOW);
  analogWrite(enA, 0);

  //left motor stop
  digitalWrite(in3, LOW);
  digitalWrite(in4, LOW);
  analogWrite(enB, 0);

  delay(x);
}
```

*Figure 16: The 4 Drive Functions*

## *Appendix D - Obstacle Avoidance Function*



```
void obstacle_avoid() {
  if (front_dist >= min_front && left_front_dist >= min_left_front && right_front_dist >= min_right_front) {
    drive_straight(0);
  }

  if (left_front_dist < min_left_front) {
    drive_break(300);
    turn_right(150);
  }

  if (right_front_dist < min_right_front) {
    drive_break(300);
    turn_left(300);
  }

  if (front_dist < min_front && left_front_dist >= min_left_front && right_front_dist >= min_right_front) {
    drive_break(300);
    turn_right(150);
  }
}
```

*Figure 17: Obstacle Avoidance Function*

## *Appendix E - Simulated Localization and Navigation*

The code for localization was embedded into the continuous drive function (when all sensors read greater than their respective minimum wall distances), such that during the instance the *elseif* statements are called the rover is within the centre of the lane. This is possible due to the secondary conditions, as highlighted in the flow diagram in Figure 0x.

There are a number of ways that can be used to localise the rover within the maze. Since the layout of the maze is known beforehand, the most straightforward way was to hard code a pre-guided route around the maze once the rover arrived at a unique location. In this way the rover would be able to drive towards the loading zone and search for the block to then be delivered. The localization route and point can be seen in Figure 18.

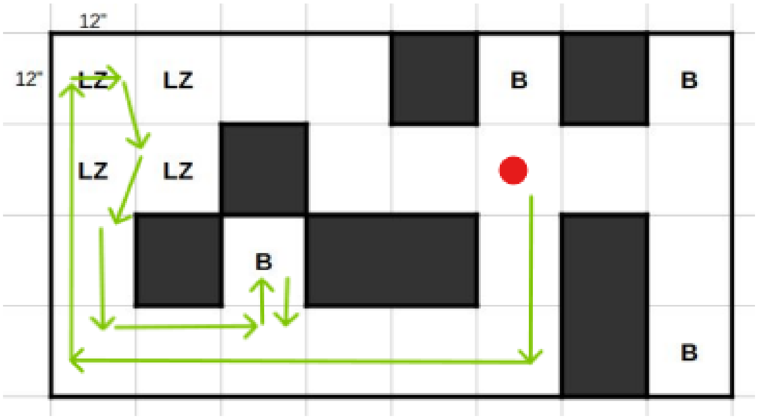


*Figure 18: Hardcoded localization route and point*

To initialise the localization protocol correctly, the rover must only run this section of the code once arriving at a unique designation point. This point is marked by a red dot as shown above, where the hard-coded route is represented by the green arrows. Once the general logic was confirmed, the flowchart shown in Figure 19 was produced to highlight the pseudo-code for this section.

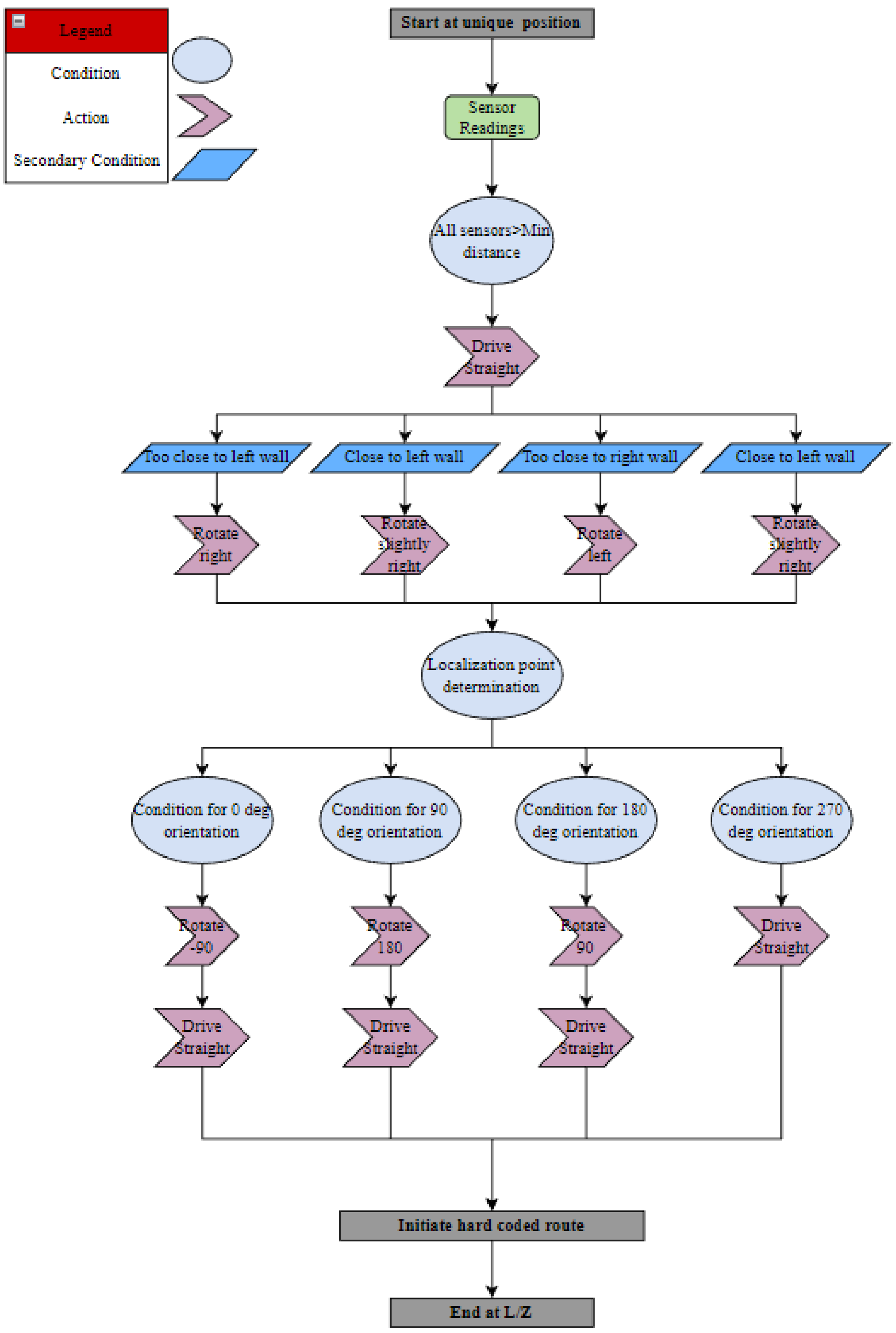


*Figure 19: Flow Chart for obstacle Avoidance Pseudo-Code*

The localization code will execute and includes the hard coded route. Once the rover arrives at the loading zone the *if* statement is terminated and the robot shall initiate the block location script.

Upon writing the localization script in accordance with the pseudo-code, the rover was unable to precisely position itself in the middle of the lane, therefore, the hard-coded method always resulted in a collision.

Instead, once the rover has orientated itself upon parsing either of the orientation conditions, the localization code ends and the obstacle avoidance code is run until the rover arrives at the loading zone.
Secondly, during trials it was seen that the rover detected the localization point as soon as it entered the ‘cross-roads’, leaving half of the robot still behind the apex of the corner. The rover assumes it is at the centre of the localization point and rotates, thus colliding with an obstacle. To counteract this, once the localization statement was met, two drive forward statements were hard coded into the statement to bring the rover roughly at the centre of the ‘cross-roads’.
The final issue was the detection of the loading zone, since the code is now running the obstacle avoidance code the rover does not automatically stop within the desired area. To solve this, another *elseif* statement was created within the continuous drive function in order to detect the **Loading Zone**, the statement is:

*elseif* (u(5)>=25) && (30<=u(6)) && (u(3)<=6)&&(u(1)<=10)

This statement, however, was also called at the second corner during the localization portion. Therefore, a counter was initialised such that the script within this statement would only run when this section had been called for the second time. This allowed for the rover to skip over the incorrect loading zone and identify and execute the appropriate code within the desired area.
The conditions for each of the orientation *elseif* statements are summarised in Table 02.
*Table 2.0: Parameters for Simulation - Localization*

| Condition | ***elseif* statement ()** |
|---|---|
| Localization point determination | (u(1)>=20) && (u(3)>=20 )&& (u(5)>=10) && (u(6)>=20) \|\| (u(1)>=10 )&& (u(3)>=20) && (u(5)>=20 )&& (u(6)>=20) |
| 0 ° orientation | u(5) > u(1) |
| 90 ° orientation | (u(3) < u(6) |
| 180 ° orientation | u(1) > u(5) |
| 270 ° orientation | (u(6) > u(3) |

The conditions set above provided an acceptable result when judging against the goal for localization (orient and arrive at loading zone).

## *Appendix F - Localization Cases and Value Ranges*

*Table 3.0: Distance Ranges for Localization Cases*

| Case # | U1 | U3 | U4 | U5 |
|---|---|---|---|---|
| Case 1 - front facing north | 30 - 45 cm | 55 - 75 cm | 55 - 75 cm | 55 - 75 cm |
| Case 2 - front facing east | 55 - 75 cm | 55 - 75 cm | 55 - 75 cm | 30 - 45 cm |
| Case 3 - front facing south | 55 - 75 cm | 55 - 75 cm | 30 - 45 cm | 55 - 75 cm |
| Case 4 - front facing west | 55 - 75 cm | 30 - 45 cm | 55 - 75 cm | 55 - 75 cm |

## *Appendix G - Original Clamp Design*

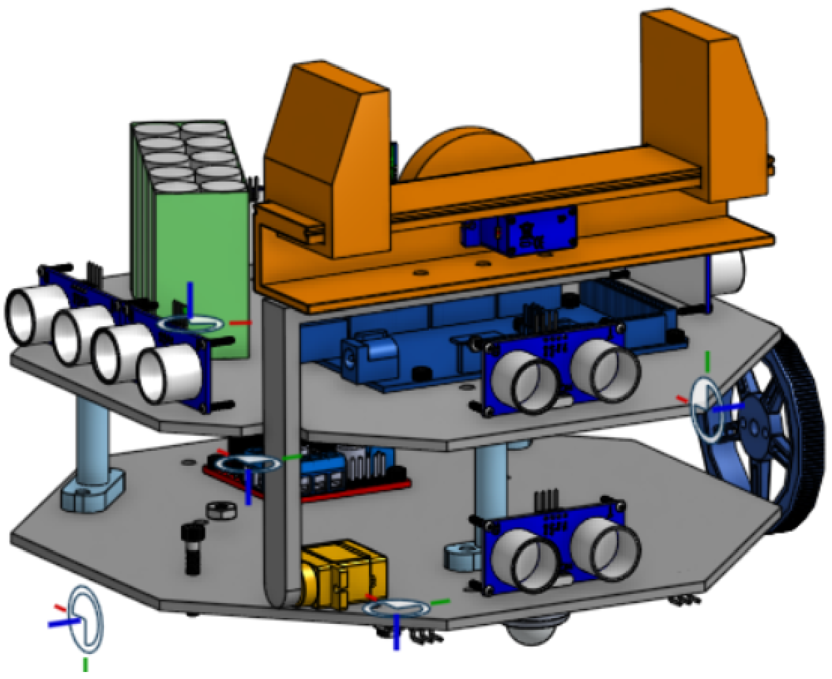


Figure 20: Original Clamp Design