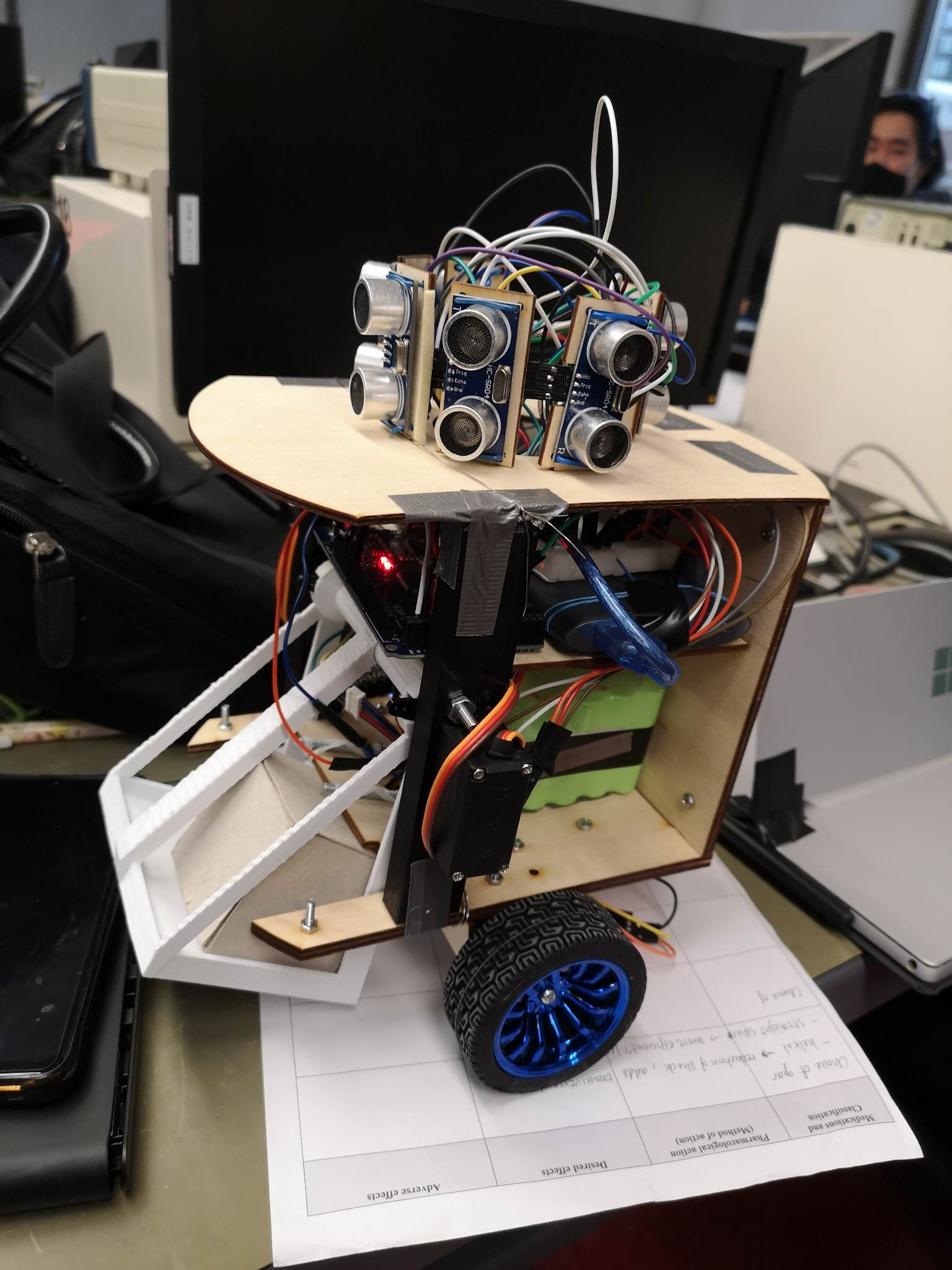
**Autonomous Rover Project**

**Final Report**

Faculty of Applied Science and Engineering

MIE444 Mechatronics Principles



Team 8

| Jimin Kwon | 1005088441 |
| --- | --- |
| Vanessa Jade Tan | 1005102933 |
| Yi Ling Soh | 1004900878 |
| Justin Zhou | 1006075140 |
| Anchorage Rainwalker | 1005771540 |

**December 16, 2022**

# **Executive Summary**

Our final rover performs obstacle avoidance, localization, and block pick-up with various degrees of success. In the following page, we will summarize the reliability and performance of the rover on each individual task.

For obstacle avoidance, the rover performs well except for very few adjustments. We use 6 ultrasonic sensors for obstacle avoidance and localization. Our robot attempts to align and center itself initially, then self-adjust while moving. The main concern is maneuvering corners due to unpredictable turns, as it sometimes turns more than the desired 90 degrees. Since the ultrasonic sensors have inaccurate readings, and despite having 2 ultrasonic sensors positioned at +/- 45 degrees, the rover isn’t able to respond in time to corner obstacles. Oftentimes, the erroneous movements are caused by erroneous sensor readings that lead to the wrong if/else blocks and power fluctuations. This means that at times, the rover will stall given the same load and same power, and perform unintended movements such as abrupt stops, or does not drive straight.

For localization, we are only able to localize correctly and reliably at a few specific spots, specifically the landmarks such as a point with no walls, and a few points with 1 wall. The inaccuracy with localization is due to erroneous IR line sensor and ultrasonic sensor readings. The ultrasonic sensors are unable to consistently detect ranges above 40 cm. We achieve localization through hard coding. We used matlab/simmer to simulate the robot’s movement, then translate matlab code to arduino to support wireless connectivity.

For block pick-up, our rover is able to locate, pick-up, and transport the block. The accuracy for block pick-up is high, as long as the block face is parallel to the robot. If placed at an angle, the rover has trouble lifting the block, and the block may drag across the floor. The rover also has trouble identifying the block at times due to erroneous IR distance sensor reading, mainly affected by lighting conditions.

The overall integration of the rover worked fine with a few manual adjustments during the trial. More improvements on the physical robot’s body are needed to make it work perfectly reliably. In the sections below, we will describe the state of the robot and its performance for each milestone in detail. The links for the robot’s performance in the maze and in the simulator are attached below.

Maze performance: [Maze performance video](https://drive.google.com/file/d/1aui9V4PIAv8P8vD5j_Y_LZpIOKRmp1hy/view?usp=share_link)

Simulator performance: [Simulator performance video](https://drive.google.com/file/d/1ax3t4HHbt800ASdH78dW5jQ2UWpiO4d0/view?usp=share_link)

Arduino code: [Milestone 3 - Arduino code](https://drive.google.com/file/d/1AeVpQDwzVeHDxUMP_yGwMPL1pGm463mg/)

Matlab Simmer: [Simmer - Google Drive](https://drive.google.com/drive/folders/1IcFWqYptG_9Nh-NpLd0CKP4cWuFcH6vL)

# 1.0 Rover Control Strategy

The algorithms are developed in accordance with the three milestones listed below.

1. Obstacle Avoidance: to drive autonomously for at least 20 ft without collision.
2. Localization: to determine where it is located within the maze.
3. Block delivery: an integration of obstacle avoidance, localization and block pickup and release at the specified location.

The following three subsections each describe the development, evaluation, and improvements of respective algorithms in the order of milestone objectives. In the last sub-section, the integration of the three algorithms is detailed.

## 1.1 Obstacle Avoidance Strategy

Initially, the rover had four ultrasonic sensors that measures the distances from the front, back, left, and right. In order to ensure the clearance for rotation, each of the four ultrasonic must read a distance larger than the rover rotation radius, *R*. The first rover design had sides measuring up to 8.2” and a total rotation radius of close to 5.8”. This allowed very little to no room for error when centering the bot for rotation, making obstacle avoidance difficult even with ideal sensor readings from simulation. The four perpendicular sensors also made it difficult, though not impossible, to estimate the distances from the walls to rover corners. As such, the rover would either drive into walls regardless of lacking clearance or would stall despite excessive clearance.



*Fig. 1.1 Modified rover sensor positions and distance definitions.*

In order to overcome the aforementioned issues, the rover is rebuilt to have a smaller circular frame. The rebuilt rover design had a maximum radius of 4.2” and had an additional two ultrasonic sensors for a total of six, as shown in Fig. 1.1. The two additional ultrasonic sensors were placed at ±45° towards the front, in order to better detect corners and walls when the rover is not aligned with the walls.



*Fig. 1.2. Obstacle avoidance strategy flowchart*

The rover’s obstacle avoidance strategy, outlined in *Fig. 1.2* above, is to initially align itself to be either parallel or perpendicular to the walls before it starts moving. Owing to the robot’s circular design, it does not need to check for clearances for rotation. Upon startup, the rover takes and compares the ±45° sensor readings to determine if either side is too close to the wall. The rover is assumed to be aligned if the two diagonal distances d2 and d6 or side distances d5 and d6 vary less than the allowable error value. In this case, the rover will rotate in place until it meets the alignment criteria. This is the most important part of the obstacle avoidance strategy, and the wall detection parameters were set by testing different variables through real-world tuning.

Once the rover has aligned itself to the walls in the maze, it can move forward as long as there are no obstacles in its path. This is determined by having a clearance of 6” (15.24cm) – that is, at least half a block – in the forward direction. By default, the left and right motors were set to the same PWM speed for forward motion. However, based on sensor readings, the PWM speed of each motor was adjusted accordingly to move away or toward the indicated nearby wall. If the diagonal sensor readings were smaller than the allowable threshold, i.e. the rover is nearing a wall on one side, that side’s motor is given a higher relative PWM signal in order to steer away from the wall. Similarly, if the diagonal sensor reading was higher than the allowable threshold, i.e. it is moving away from the wall, then its opposite motor is given a higher PWM signal in order to steer back towards the wall. This constant adjustment method allows our rover to move in a fairly straight manner, keeping itself parallel to the walls. The emulated rover path is exaggerated in *Fig.1.3* to demonstrate the adjustment process, though in reality the robot’s movement was less erratic and quite smooth when sensor readings were consistent.



*Fig. 1.3 Ideal path of rover (red) vs actual movement with readjustment (blue)*

It was quickly realized though that sensor readings can be inaccurate or have some noise, which causes disruptions or inconsistencies in the distance readings. In order to cross-check the diagonal sensor readings for straightforward movement clearances, the side sensor readings are also checked to be within the threshold. When the above adjustments are made, the side sensor readings are further assessed to counter excessive steering. For example, if the side sensor readings are within the safety threshold, the adjustment based on diagonal readings turns the rover too much. In this case, the adjusted PWM values are readjusted to reduce the steering effect. Every robot’s movement is set up to stall after every step to avoid slamming into walls due to the time lag between sensor readings and motor control.

Considering that the rover is aligned to the maze walls and can calibrate itself to go straight, it will be able to maneuver the entire maze by moving forward and making 90° turns. As with the alignment strategy, the rover does not need to check for clearances to make rotations in place. The direction of turn (clockwise or counterclockwise) is set by the forward/reverse motor directions, and turning corners was hard-coded by measuring the amount of time needed for the rover to rotate almost exactly 90° at the set speed. The real-world tuned values for motor speed and duration to achieve the basic rover movements are outlined in *Table 1.1*, with positive motor speed indicating forward rotation (digital LOW-HIGH signals) and negative speed indicating backward rotation (digital HIGH-LOW signals).

*Table 1.1: Summary of parameters set to control basic rover movement*

| **rover Motion** | **Left Motor PWM** | **Right Motor PWM** | **Duration** |
| --- | --- | --- | --- |
| Forward | 110\* | 110\* | 0.3s |
| Reverse | -110 | -110 | 0.3s |
| Rotate CW 90° | 83 | -83 | 1.1s |
| Rotate CCW 90° | -83 | 83 | 1.1s |

*\* PWM values are adjusted according to sensor readings to maintain alignment*

The obstacle avoidance algorithm was primarily based on the simulated rover on Matlab Simmer [1]. Motor control relied on sensor readings to adjust PWM to control motor speed. The sensor-reliant drive was decent in the simulation and worked well when the sensors behaved as expected. The differences between Simmer performance and real-world testing presented major challenges for the team to overcome.

One particular challenge was the level of uncertainty in the accuracy of sensor readings. Although there was a tolerance for small errors accounted for in the code, the sensor discrepancies were at times quite large. In these cases, large errors were mitigated by cross-checking sensor readings with each other for a reasonable variation. Additionally, moving in small fixed steps helped to prevent the rover from running into walls in between sensor readings. This approach seemed to work quite well during test runs in the Myhal maze, as can be observed in the videos. However, there were times when the sensors did not behave as expected and the inaccurate readings became more apparent. By this point, the team did not have enough materials and time to implement and test the rover using the DC motor encoders instead of relying fully on sensors.

The second major obstacle faced was inconsistent motor operation due to power fluctuation. It was observed that the motors would run at the intended speed for a few seconds upon startup and then slow down despite the same PWM signal. This made the rover stall unintentionally or move erratically, especially when the two motors were not synced. This problem manifested when the rover was trying to align with the wall: when it attempted to move away from a wall, the motor slowed down and it was unable to steer, causing it to run into the wall. It also caused 90° turns to be under or over-estimated. This may be mitigated by using a buck converter to regulate the voltage input to the motors so that it is not as sensitive to the battery voltage changes.

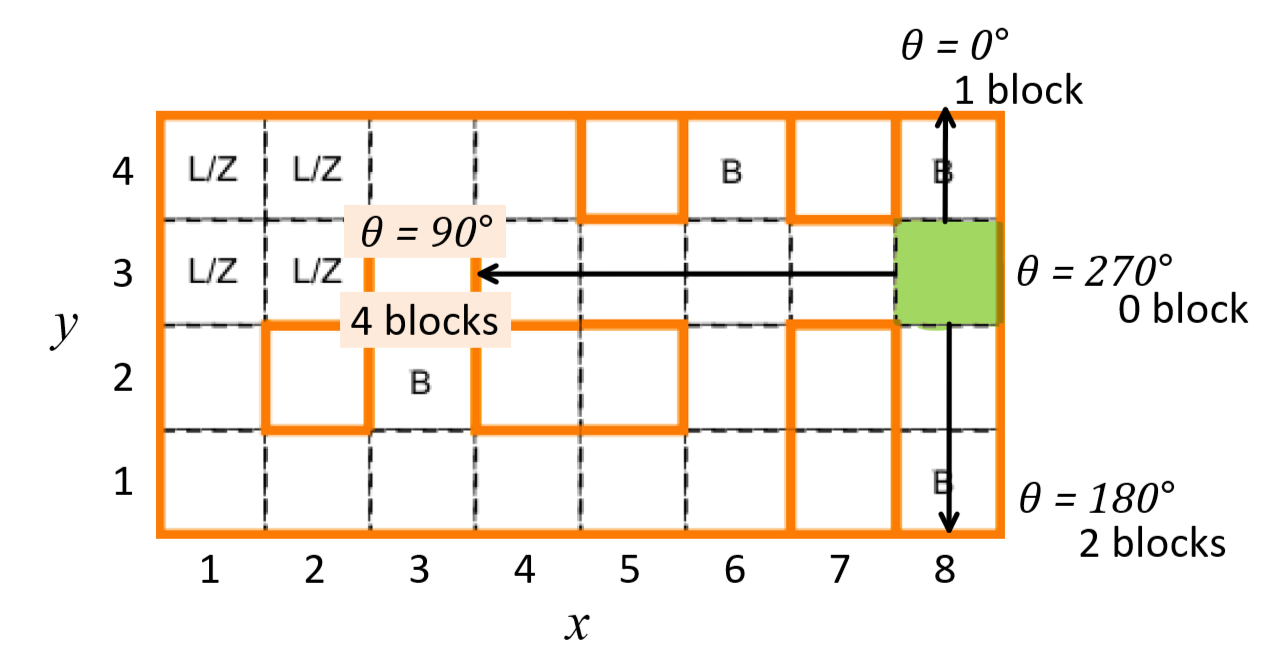
## 1.2 Localization and Navigation Strategy

For localization, the rover is placed randomly in the maze. It must determine its location and navigate to the loading zone followed by the drop-off zone. In order to achieve this, the maze map is redefined into x, and y coordinates of blocks as shown in *Fig. 1.4*. Each block is first categorized by the surrounding walls (See *Table 1.2*) [2]. This first isolates one unique location when surrounded by no walls.

*Table 1.2: Map categorization based on detected surrounding walls.*

| **Surrounded by** | **Blocks (x, y)** |
| --- | --- |
| 3 walls | (3, 2), (6, 4), (8, 1), (8, 4) |
| 2 adjacent walls | (1, 1), (1, 4), (2, 3), (4, 3), (4, 4), (6, 1) |
| 2 opposite walls | (1, 2), (2, 1), (3, 4), (4, 1), (5, 1), (5, 3), (6, 2), (7, 1), (8, 2) |
| 1 wall | (1, 3), (2, 4), (3, 1), (8, 3) |
| No walls | (6, 3) |

When the rover is surrounded by at least one wall, it needs more information to localize. Consequently, coordinates are differentiated by the number of blocks prior to detecting the wall obstacle. As an example, *Fig. 1.4* illustrates the number of blocks between coordinate (8,3) and each of the four walls around it.The counted data is organized under *Appendix A.0*, excluding coordinates that are surrounded by 2 opposite walls (highlighted row of *Table 1.2*) due to their lacking unique characteristic despite integrating distance and line detections. In order to reduce complexity and length of the localization code, if the rover is in a position with 2 opposite walls, it will move to a new position in the direction of largest distance reading to complete localization.



*Fig. 1.4 Sample of block-counting for coordinate (8, 3), with x and y coordinates labeled in the provided map [3]. Directions are denoted by θ in CCW direction.*

Localization is achieved through a hard-coded approach, leveraging known map data organized in *Appendix A.0* to avoid complex algorithms [2]. Assuming the rover is aligned to walls from the obstacle avoidance algorithm, distance readings from the four perpendicular ultrasonic sensors are used, such that the number of blocks between the rover and a wall is:

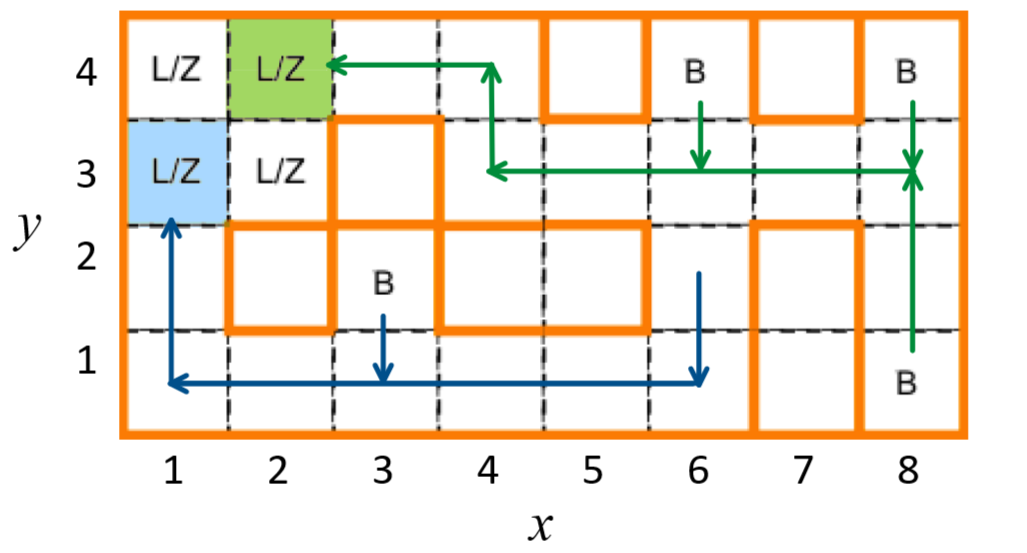
Every instance of 0 blocks is a surrounding wall based on *Table 1.2* categorization, and the count of surrounding blocks forms a matrix **[front, left, back, right]**. This feature is then used to identify rover orientation in terms of θ. For example when the rover reads a matrix of [2, 2, 2, 1], there are no surrounding walls (i.e. position (8,3)) and the right sensor is facing the direction of θ = 0° meaning the rover is oriented in θ = 90°.

Even with walls detected, coordinates with unique matrices are located easily with ultrasonic distance readings, along with the rover orientation in terms of θ. For example, when the rover reads a matrix [4, 2, 0, 1], the algorithm follows the logic below:

1. One instance of 0: 1 wall detected, leaving four possible positions.
2. Maximum is 4 blocks, unique to position (8, 3).
3. Wall is detected by the back sensor, where the wall of (8, 3) is in the direction of θ = 270°. This identifies the orientation of the rover to be θ = 90°.

However, there are coordinates that cannot be differentiated with the range data. To overcome this confusion, the two IR line followers are used to identify two pixels of the map floor pattern, which are integrated with the range data for position and orientation.

Localization is achieved when the rover knows its position and orientation in terms of (x, y, θ). The next task, to navigate to the loading zone then to a specified delivery zone, was also hard-coded []. *Fig. 1.5* illustrates the path to the loading zone (L/Z) from any location. There are two paths, each aiming towards one of the two entrances of the L/Z: green path towards green block (2, 4), blue path towards blue block (1, 3).



*Fig. 1.5 Paths to loading zone (L/Z) depicted by arrows. Green path reaches green entrance at (2, 4); blue path reaches blue entrance (1, 3).*

When provided with (x, y, θ) from localization, the navigation algorithm identifies which path the rover should take to the L/Z. This is saved as an integer variable pickup.

pickup = 1; // green path

pickup = 2; // blue path

This variable determines the goal location and is set to 0 upon reaching the L/Z.

Upon starting the navigation code, the rover updates pickup and saves a copy of the position coordinates as a matrix variable: tracking\_xy = [x, y]. Based on the predefined path, tracking\_xy is updated to new coordinates as the rover moves. Its orientation, θ, is assessed to face the direction of travel. Any rotation immediately updates θ by the amount of rotation. The rotation is followed by a forward motion where the rover follows the path of *Fig. 1.5* block by block. The navigation code is only entered after the rover moves by one full block to update the instruction towards the next block.

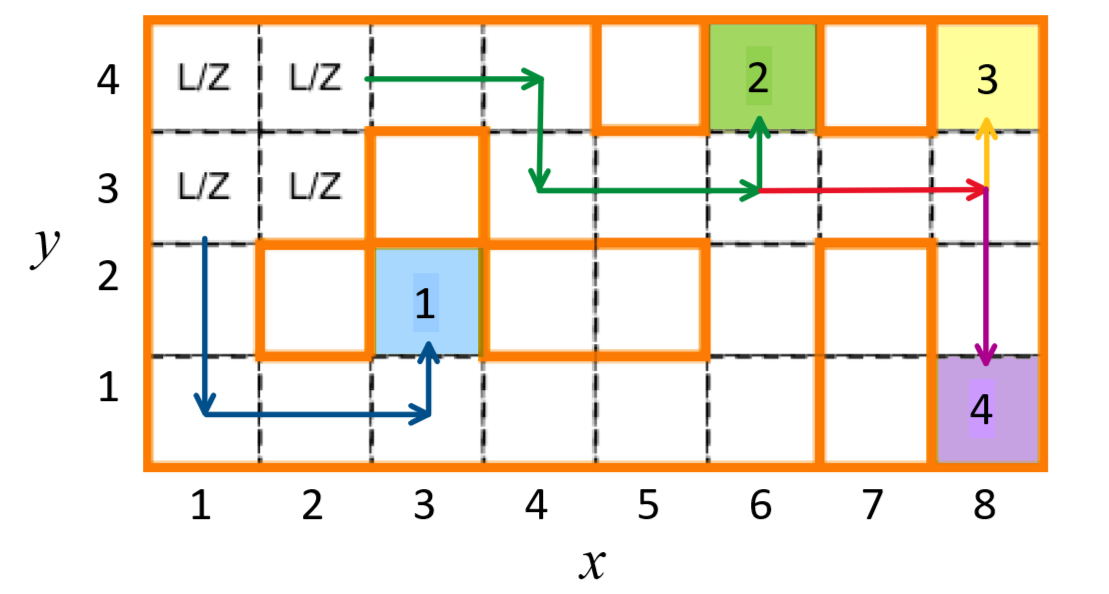
In the simulation, the step size was 1”, which requires 12 steps for the rover to move by 1 block. The counter variable is used with pickup to enter the navigation code whenever pickup != 0 and counter = 12. The position of the rover is overwritten by tracking\_xy when the counter reaches 6, meaning the rover has entered a new block. This loop of updating path instruction with possible rotation followed by 12 steps is repeated until the rover reaches the L/Z. The two terminating conditions for the L/Z navigation code are:

1. If pickup = 1, AND (x, y) = (1, 3).
2. If pickup = 2, AND (x, y) = (2, 4).

Because the position is updated when counter reaches 6, the rover will confirm reaching the loading zone when it is halfway into the L/Z. This algorithm performed decently during the simulation with one issue. In some instances, the rover would try to move forward despite hitting the wall. This is prevented by adding one more condition in entering the navigation code incorporating front ultrasonic distance measurement. If the d1 is less than 7” (half of block + one step), the rover should rotate instead of moving forward to prevent the collision with the front wall.

The algorithm for navigating to the delivery zone is built similarly with new sets of paths illustrated by *Fig. 1.6*. An integer variable dropoff corresponds to the indicated delivery zone.

Assuming the rover is realigned and centered in one of the L/Z blocks and relocalized, the rover follows the path towards the specified delivery zone, which is hard-coded. The navigation method is identical to L/Z navigation, loops of updating path instruction followed by steps until reaching the next block is repeatedly done to reach delivery zone.



*Fig. 1.6 Paths to the delivery zone (colored blocks) from L/Z, depicted by arrows. The four delivery zones are labeled to its corresponding ‘dropoff’ value.*

For dropoff = 1, rover leaves the L/Z through the (1, 3) exit:

1. If x = 2, move one block such that x = 1.
2. If y = 4, move one block such that y = 3.
3. At this point the rover should be at (1, 3) where it can follow the blue path to reach delivery zone 1.

For dropoff = 2, 3, 4, the rover leaves the L/Z through the (2, 4) exit in a similar logic. It then follows the green path until the first branching at (6, 3). If dropoff = 2, the rover will follow the remaining green path to reach delivery zone 2. Otherwise, it will follow the red path until branching at (8, 3) to reach either yellow (3) or purple (4) zones through respectively colored paths. Sharing code for overlapping paths reduces the complexity and length of the navigation code, rather than coding a path for each dropoff.

Localization and navigation algorithms proved to work with ideally-functioning sensors in MATLAB Simmer simulations [1]. However, physically implementing the algorithm was a challenge when sensors did not work as expected. The sensors had erroneous readings on occasions, where it would read much less than actual distance. This was a problem especially since the hard-coded localization is heavily dependent on distance sensor measurements. Because of this, the algorithm was modified to incorporate pixel data from line followers with distance data for non-unique matrices for localization. The distance sensor also failed to read distances larger than 3 blocks, giving readings of 0. Since the sensor will never be directly against a wall, it is assumed that a wall is 3 blocks away when the sensor reads 0 and the measurement is adjusted to be 100cm.

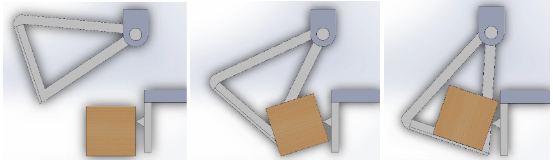
The inconsistent motor operation due to power fluctuation was also an obstacle for actualizing the navigation algorithm. Every step of the physical rover did not result in consistent travel distance. This inconsistency prevented the block-by-block motion to work as expected. Therefore instead of updating instruction and position every block, the instruction is updated whenever rotation (turn in path) is required. For example, when taking the green path towards delivery zone 2 from (1, 4, 270°), the simulated rover would first navigate to (2, 4). However, the Arduino code is modified such that the rover moves to (4, 4) where a turn is required. This is achieved by removing counter and redefining the condition of entering the navigation code. In this example, from (1, 4) to (4, 4) the rover needs to stall and turn when its front sensor detects a wall prior to collision and at least one of its side sensors detects a path. The side sensor is assumed to detect a path when d3 or d5 reads more than one block (12”). The conditional value for d1 was tuned through rounds of testing within the Myhal maze.

In cases when the rover needs to make a turn without first detecting a wall, such as entering delivery zone 1 from (3, 1) and delivery zone 2 from (6, 3), the float variable check determines the number of blocks before the next wall when the rover needs to turn. The rover stalls and rotates when **d1 < 13 + (34 \* check)**. It is multiplied by 34cm instead of actual block length of 30.48 cm due to occasions of the rover moving past the desired location. For delivery zone 2, check = 2. Additionally, check is set to 1.5 for the rover stall when halfway entering the L/Z to avoid pushing the block too deep into the L/Z prior to starting the block pick-up mechanism. It is also set to 0.5 for the rover to stall halfway entering the delivery zone to secure room for releasing the block.

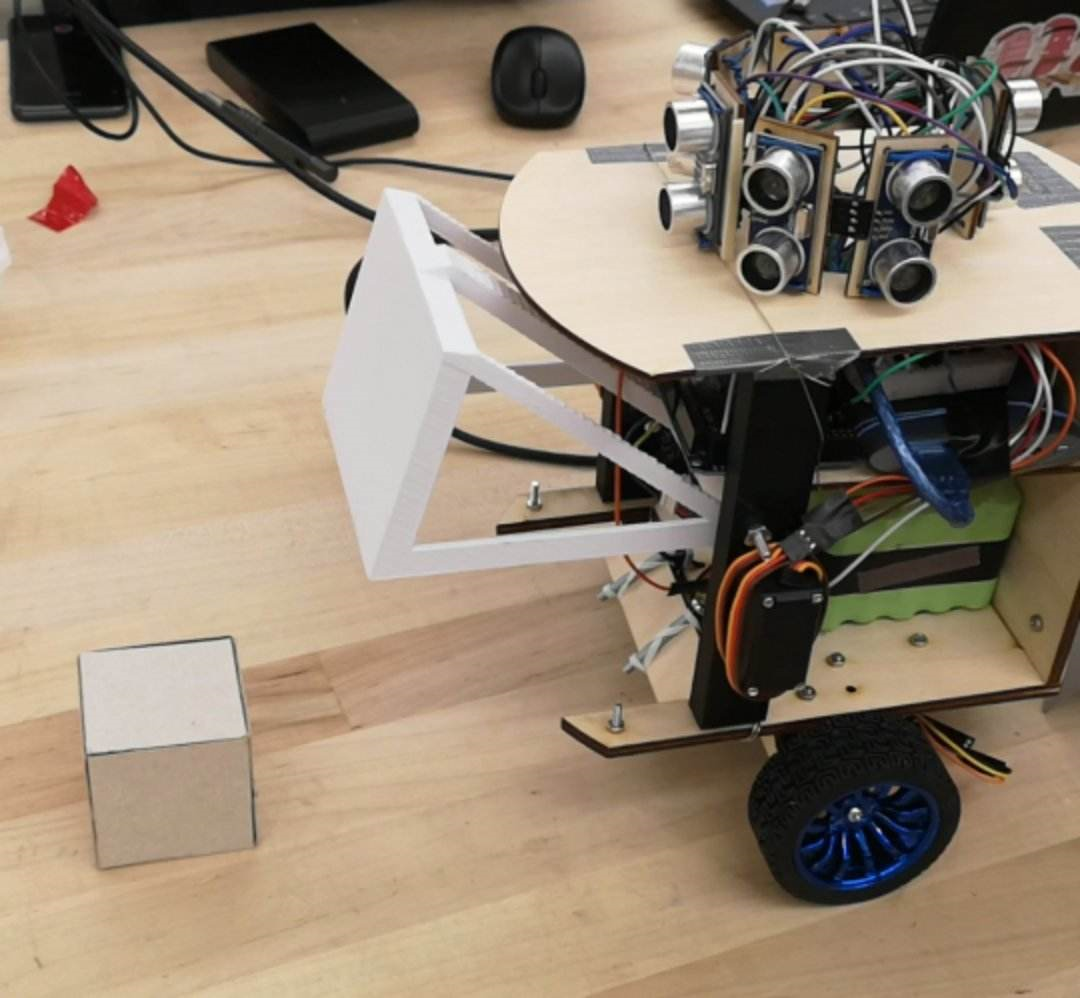
The localization, navigation to L/Z, and navigation to delivery zone are each coded as a separate function on the Arduino and called only when needed to organize the code and also reduce the length of the loop code. The conditions needed and integration of the code are detailed in Section 1.4.

## 1.3 Block Delivery Strategy

The rover uses a shovel and a plate to scoop the block. As shown in *Fig. 1.7*, the initial plate design had a triangular protrusion to tip the block onto the shovel. In theory, the design would work with proper calculations of the ideal angle and position of the tip. However, the prototyped plate was not effective and resulted in the block getting stuck without properly loading onto the shovel. Due to lack of time to redesign and 3D print the plate, a new plate was improvised using plywood, screws, and elastics (See *Fig. 1.8*).



*Fig. 1.7 Scooping mechanism to pick up the block using a plate with protrusion.*



*Fig. 1.8 Final rover scooping mechanism with improvised plate.*

When the rover successfully navigates to the L/Z, the shovel lifts up to 90° and the IR sensor mounted beneath the rover can be used to detect the block. The block detection algorithm is outlined in *Fig. 1.9* and will only be performed when the rover is in the L/Z..



*Fig. 1.9 Block detection and retrieval flowchart.*

The front ultrasonic sensor reading, d1, is taken and compared with the IR distance sensor measurement. If they are equidistant, no block is detected. The rover will rotate by 15° in the direction away from the L/Z walls. This is defined by two possible cases. If the rover enters the L/Z through (2, 4) or pickup = 1, it will rotate counterclockwise; if it enters L/Z through (1, 3) or pickup = 2, it will rotate clockwise. Rotation stops when the IR distance is less than d1, indicating that the block is in the path of the rover. It then moves forward until IR dist = 0 and the block is in front of the plate. The shovel closes to pick up the block, and block retrieval is completed. The modified plate uses elastics such that it protrudes initially to push the top of the block upon contact, tipping the block, and then retracts to prevent the block getting stuck as it lands on the shovel.

With the block collected, the rover can navigate towards the specified delivery zone from the L/Z, as described in Section 1.2. As it arrives at the drop-off location, the shovel lifts, and the elastic plate gently pushes the block to release it. This completes the block delivery process.

The block pick-up and drop-off was not simulated on Matlab, but was directly coded in Arduino and tuned through multiple iterations of real-world testing. This posed several challenges to develop a well-functioning strategy. Firstly, block detection was inaccurate due to the IR sensor sensitivity giving erroneous readings. Like the ultrasonic sensors, this can be mitigated to a certain degree by allowing for tolerances. Furthermore, the conversion of IR distance voltage readings to distance measurements was based on the experimental line of best fit in Excel, which only gave a rough estimation of the actual distance measurement assuming the reading was precise to begin with. Due to these problems, there are instances when the rover incorrectly detects a block and closes the shovel without loading the block. Since there is no check to determine that the block is physically loaded, the rover assumes the block is picked up and immediately executes the drop-off algorithm. The original rover design incorporated a limit switch on the shovel which would be activated when the block is successfully loaded, but this was not implemented in the final rover due to material and time constraints. The other issue faced was the physical capability of the rover to pick up the block. This was resolved with a makeshift plate, which was attached underneath the base and did not require any other modifications to the design. During the trials, the rover can successfully pick up, transport, and release the block with the implemented block-delivery mechanism.

## 

## 1.4 Integration

Although the project was delivered in milestones, the final execution was an integration of each part. *Table 1.3* describes each code block that is implemented in *Fig. 1.10*.

In the main loop() section, the rover starts with taking ultrasonic sensor readings, which are converted into distances and used to track rover location. These readings are used in adjust(), alignment(), center(), localization(), block\_detection(), to\_pickup(), and to\_dropoff(). As shown in Fig. *1.10*, implementation occurs in checkpoints. The rover will only execute alignment() until aligned = 1, then it only executes center() until centered = 1, and so on. Whenever the rover moves, adjust() is called in every iteration to maintain a straight path. Since the movements are hard-coded to be exact (ex: go forward 1 block, turn 90°, etc.), the given direction is expected to avoid obstacles since the map is given.

*Table 1.3: Description of each code block used in rover control.*

| **Function** | **Description** |
| --- | --- |
| get\_Distance | At every iteration, the 6 ultrasonic sensor readings are updated. The readings are converted to usable distance measurements. The variables x, y, tracking\_x, tracking\_y are used to track rover location. |
| serial\_Monitor | This block reads various user inputs, such as drop off location, showing sensor readings, stop, start, and reset. |
| adjust | The function uses ultrasonic sensor readings to adjust the speed offset of each wheel to steer the robot, maintaining a straight path. |
| alignment | From the start, align = 0. It will continue to execute alignment() until the conditions for rover alignment to the maze walls are met, which sets align = 1. |
| center | From the start, centered = 0. It will continue to execute center() until the rover is centered, which sets centered = 1. |
| localization | From the start, localized = 0. It will continue to execute localization() until the rover has determined its position (x, y, θ), which sets localized = 1. |
| block\_pickup | The path to the L/Z is determined with the algorithm detailed in Section 1.2. Then the number of steps to move or turn are relayed to the rover through dedicated movement functions. At the L/Z, the shovel servo opens, exposing the IR sensor to detect the block, as detailed in Section 1.3. When the block is located, the shovel closes. |
| block\_dropoff | The path to the drop-off zone (given through serial monitor) is calculated. The rover then traverses the pre-set path in the to\_dropoff() module. |

Untitled drawing (1).jpg, get_Distance() 
serial_Monitor() 
adjust() 
alignment() 
center() 
localization() 
block_pickup() 
block_dropoff()  
Loop diagram
Ink Drawings
Always gets executed 
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
aligned = 1
aligned = 0
centered = 0
centered = 1
localized = 0
localized = 1
to_pickup=0

to_pickup=1
to_dropoff=0


*Figure 1.10 Integration block diagram*

Integrating the code blocks is relatively easy and its performance is at the same level as when performed individually. The main problem is that when the rover begins executing each code block, it does not stop (due to time constraints). Thus when the rover is adjusted mid-drive, it will potentially trigger incorrect sensor readings. As a result, the pre-planned path is disrupted for the remainder of the run. Unfortunately, the rover does not have a pause button that pauses the executions mid-drive to avoid this problem.

The other issue with the integrated algorithm is that it is assumed that checkpoints are maintained and do not need to be reset throughout the run. For instance, when the rover has localized, it will not localize again. However, due to the limitations of the rover in getting accurate sensor readings, its movement may not be as expected. It was found that although the rover aligns and localizes correctly in the beginning, it may drift from the expected path, which becomes a problem since it will not realign or relocalize mid-drive.

Overall, the rover is able to execute all the tasks in a timely manner, with little to no intervention between each task. The accuracy of each task can be improved with modifications to the algorithm and a few physical changes as mentioned previously.

# **2.0 Final Results**

In this section, we will detail the overall performance of each milestone.

**Obstacle avoidance:** We are able to perform obstacle avoidance well due to fine tuning the pre-defined movements. In the final runs, the rover is able to complete the instructed maneuvers satisfactorily (clean 90 degree turns and driving straight overall). On top of that, we also constantly adjust the speed of the rover (controlling the PWM values) while the rover is moving, to help the rover traverse straight. Unfortunately, our alignment and centering code did not work well. Due to inaccurate and erratic sensor readings, the centering and alignment is almost always off. Thus, we have to center and align the rover prior to actual rover traversal. Otherwise, the pre-defined movements would not work as planned.

**Localization & Navigation:** The rover is able to localize without fail at very distinctive locations, namely (6, 3) and (8, 3) when aligned with walls within an acceptable level. The alignment of the rover and sensor performance greatly affected the localization where it could not localize with accuracy when starting at the orientation of 45°. Assuming correct localization, the navigation algorithm worked as expected where it stalled and turned prior to colliding with the wall to the following the hard-coded path. During actual trials, when adjustments were made, the sensor detected our hand and thought it was the wall. This made the rover rotate at the wrong location and resulted in incorrect movements.

**Pick-up and delivery of the load:** After localizing correctly, we are able to execute the pick-up/delivery well. Our algorithm is able to find the path to the loading zone. During our runs, the rover traverses the maze with little help in pick-up, save a few minor adjustments when hitting the corners. The rover then stops at the edge of the loading zone, where it lifts the shovel to 90 degrees. The block detection works fairly well, as long as the block is aligned parallel to the robot’s plate. This means that we had to shift so that the IR distance sensor is able to locate the block accurately. Since we are the ones who place the block down, there is little to no need for adjustment, as we place the block parallel to the robot. The rover has a hard time picking up the block if the block is placed at an awkward angle (not in multiples of 30s), and can get stuck in the claw. The accuracy for block drop off depends on the drop-off location. For example, the drop off location located at the upper right corner, we are able to drop-off successfully almost 100% of the time. This accuracy decreases for the other 2 drop off locations. This is due to inaccurate sensor readings and no walls to align the rover well. For those drop off locations, the rover tends to miss the entrance, or drop off the block in an incorrect spot (too early or too late), which terminates the entire program.

**Integration:** Our rover requires little to no external help to integrate all parts of the rover together. This means that our rover can execute from start to finish, with few adjustments for obstacle avoidance and alignment. The starting of the rover is crucial, since our rover does not align and center satisfactorily. Prior to moving and localization, we had to center and align the rover parallel to the wall so that it can work properly. Due to our localization only working at specific spots, there are limited spots where we can place the robot. After taking in all the considerations, the rover is able to drive itself to the loading zone, pick up the block itself (with no help), and drop off the block (though not at the right location at times) with no interruptions in between each segment.

Overall, the rover completes many parts of the milestones, albeit with mediocre accuracy. Obstacle avoidance, rover traversal, and block pickup is executed well with high accuracy. Alignment, centering, and localization was difficult to implement and tune, thus our team was unable to reach our desired accuracy for these sections.

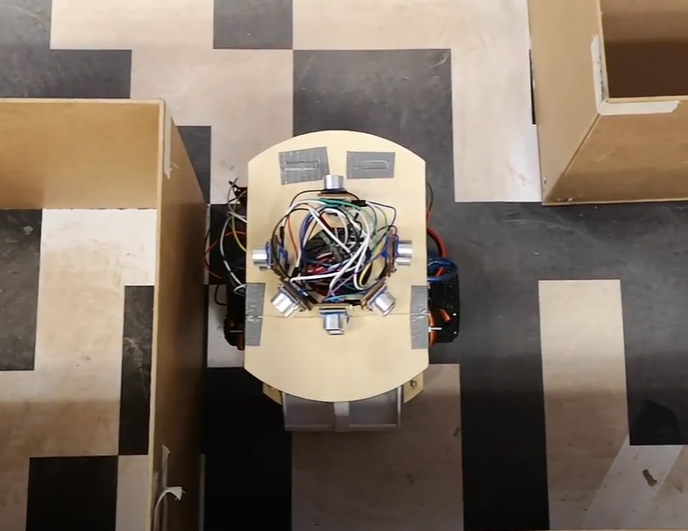
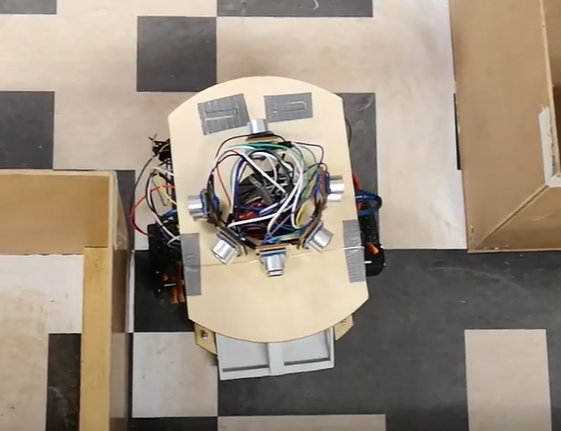
# **3.0 Discussion**

The team encountered various challenges and successes throughout the progression of the project, including two major physical redesigns of the rover. For Milestone 1, the rover was built based on the RFP, which had a square footprint and 4 ultrasonic sensors. The obstacle avoidance algorithm was primarily based on steering toward or away from a nearby wall, similar to the adjust() strategy, but it was difficult to tune accurately with only 4 sensors. In the end, due to the large size, limited sensor readings, and perhaps inexperience or unfamiliarity with the hardware and software involvements of the rover, the team was unsuccessful in the official runs for Milestone 1.

The rover redesign started immediately after Milestone 1. Its overall footprint size was reduced, changed to a circular design, and two more sensors were added to measure diagonal distances and avoid corners. The frame was made from laser cut plywood because of the shorter fabrication time compared to 3D printing. Due to time spent on rebuilding the rover, the team decided to redirect efforts to simultaneously building the physical bot and working on simulation for the Milestone 2 deliverable.

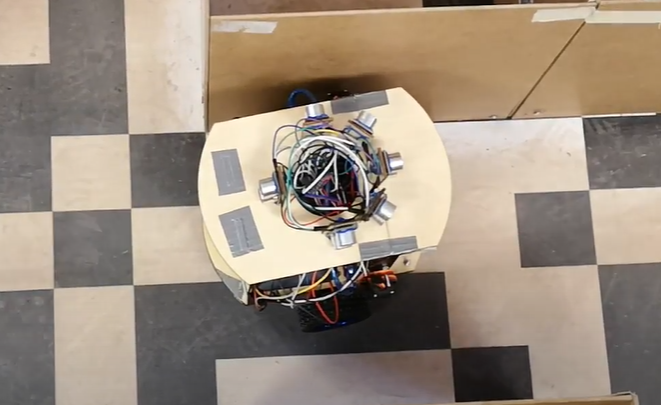
In the end, the team was more familiar with the expectations and issues working with the physical and code aspects of the robot, especially after having rebuilt it several times, and was able to complete Milestone 3 fairly well, with some persisting areas for improvement which will be discussed later.

The best feature of the rover is its ability to adjust steering based on sensor readings to maintain alignment along the walls as it moves. Because of the power fluctuation which caused the motors to behave inconsistently, even when the robot was intended to move straight, it would lean towards the right or left. Thus, it was invaluable for the rover to be able to accommodate the changes dynamically by adjusting the speed of either motor faster than the other to steer. The PWM values provided to the DC motors would be adjusted to let the rover move slightly left if it was too close to the right wall or slightly right when it was too far away from the right wall. It also helped realign the rover after 90° turns. Since turning corners was hard-coded assuming consistent behavior, in instances when the rover under- or over-estimated corners, its movement would be compensated with the steering adjustment, as shown in *Fig. 3.1*.



*Fig.3.1 (a) Rover entering the turn, not perfectly facing forward; (b) Rover adjusted after the turn, perfectly facing forward and aligned to the wall.*

Unfortunately, the worst feature of the rover was the control of DC motors. On a physical level, the mechanical structure supporting the DC motors was tilted from the installation of the motor mounts and couplers. Uneven distribution of the weight of the rover, such as when the battery is placed on the side rather than the middle, also increased wheel friction on one side. As mentioned previously, the motors had inconsistent drive when the battery was not fully charged. They would start at full speed and slow to a stall within seconds. Thus, despite efforts to adjust the motor speeds to regain alignment with walls, the rover would still end up running into walls because of insufficient power to the motors. The team tried to mitigate this using a linear voltage regulator, which slightly helped. However, a buck converter would have been preferred for efficiency and reduced power drawn from the battery, ensuring more consistent results.



*Fig. 3.2 Rover stuck at the wall despite attempts to steer right.*

From the building of the robot, it was learned that integrating mechanical, electrical, and coding knowledge in the project is not very simple. Many real-world conditions are needed to be considered in the development of the robot, which makes applying only theoretical knowledge not reliable and unsustainable. For example, while the decision to add two more ultrasonic sensors significantly improved the rover’s obstacle avoidance performance, the team was unprepared for the level of inaccuracy of sensor readings. It was expected that the ultrasonic sensors worked when facing the wall directly or at a small angle, but were inaccurate with large angles. However, even when aligned to the wall, the sensors would still read 0 at times and were unreliable with distances larger than 2 blocks.

The issues found in the components and equipment, along with real-world implications and conditions, required more testing and troubleshooting of the rover than expected. The ultrasonic sensors created many difficulties in the implementation of the distance detection, obstacle avoidance, and localization algorithms. Furthermore, the ultrasonic sensors did not perform the same in the maze as in the testing. It was able to detect obstacles about 3 blocks away in testing, but it could not differentiate walls that were two blocks away or three blocks away when actually running in the maze. Thus, the path instruction algorithm was not reliable since it was hard-coded depending on the ultrasonic sensor readings. The team also found issues with the Bluetooth module not working properly and needed to replace the module itself at least 3 times.

The application of only theoretical concepts and course knowledge was not enough, and creative solutions were needed to solve many unexpected issues. One issue in particular was finding that the Arduino Mega had a defective Vin pin. This was not detected early on since most of the testing was done while connected to the computer with USB. Unfortunately, the MIE444 teaching team did not have any extra Megas that could be provided. The Arduino was replaced with an Uno for obstacle avoidance, but the Uno did not have enough pins for all ultrasonic sensors, IR sensors, motor controls, etc. The team decided to proceed with the Mega while trying to find another way to power it. The team was unable to get a barrel jack connector from MyFab, and in the end used an external power bank to power the Mega via USB.

Besides limitations from components and hardware, the overall code of the rover also had areas for improvement, the majority of which had to do with dealing with the imperfect real-world conditions where the rover operated. Without changing the rover design, an improvement could be made by implementing regular checks to ensure the rover was still aligned, centered, and localized properly mid-drive. The current algorithm only checks once and assumes perfect behavior from the motors and sensors along the way. However, as was found, the inaccuracies lead to significant errors in the rover path especially when the rover thinks it is in a particular location or orientation which is incorrect. Noise in sensor readings could have been mitigated through code with the use of Kalman filters or functions to account for unreasonable readings.The rover also did not have a system in place to check whether the block was picked up before going to the delivery zone. In general, the rover was able to complete the tasks with the current strategy, so modifications would only enhance performance and add additional measures in cases when any physical component does not work properly.

Overall, the team will leave the project with a better understanding of the systems that comprise an autonomous rover, including the extensive mechanical, electrical, and programming design needed. For a similar project in the future, a clearer plan of the overall project can be made at the start, where the main issues should be addressed in the initial design phase of the project. The mechanical, electrical, and coding parts of the project and the integration of them should all be considered since issues in one part of the project impacts other parts and slows the overall progress down. Moreover, adaptive and creative problem-solving was needed to overcome issues that came along during testing. What the team could have done better on an organizational level was communicate any concerns at any point with each other. There was a disconnect between team members, which led to slow progress and resolution of project-related issues. In the end, the team had a small glimpse through this project of how various areas of mechatronics design come together, how iteration and critical thinking is required, and how to deal with different people’s working styles, all of which will be important moving forward in future projects and in their careers.

# References

[1] S. Colic. (2022). MIE444 SimMeR Robot Simulator. Available: https://q.utoronto.ca/courses/282545/modules/items/3883268.

[2] S. Colic. (2022). MIE444 Week5 Lecture Localization [PDF document]. Available: https://q.utoronto.ca/courses/282545/files/22584096?module\_item\_id=3883244.

[3] S. Colic. (2022). MIE444 Project Guideline [PDF document]. Available: https://q.utoronto.ca/courses/282545/files/21924060?module\_item\_id=3883254.

# Appendices

## Appendix A.0 Map Data

*Table A.0 Blocks prior to wall detection in four directions of* θ*, for each coordinate introduced in table x.x excluding the case of 2 opposite walls.*

| (x, y) | θ | | | | (x,y) | θ | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0° | 90° | 180° | 270° | 0° | 90° | 180° | 270° |
| (3, 2) | 0 | 0 | 1 | 0 | (1, 3) | 1 | 0 | 2 | 1 |
| (6, 4) | 0 | 0 | 3 | 0 | (2, 4) | 0 | 1 | 1 | 2 |
| (8, 1) | 3 | 0 | 0 | 0 | (3, 1) | 1 | 2 | 0 | 3 |
| (8, 4) | 0 | 0 | 3 | 0 | (8, 3) | 1 | 4 | 2 | 0 |
| (1, 1) | 3 | 0 | 0 | 5 | (6, 3) | 1 | 2 | 2 | 2 |
| (1, 4) | 0 | 0 | 3 | 3 |  |  |  |  |  |
| (2, 3) | 1 | 1 | 0 | 0 |  |  |  |  |  |
| (4, 3) | 1 | 0 | 0 | 4 |  |  |  |  |  |
| (4, 4) | 0 | 3 | 1 | 0 |  |  |  |  |  |
| (6, 1) | 3 | 5 | 0 | 0 |  |  |  |  |  |

## 