MIE444 - Mechatronics Principles

Final Project Report

Group 1:

Karim Mansour	-	1006152725
Max Mizrahi	-	1005706149
Jesse Chudnow	-	1004251711
Qilong Cheng	-	1003834103

December 15, 2022

1.0 Executive Summary

Our team built a robot to autonomously navigate a predefined maze, with the ultimate goal of being able to locate a wooden block and deliver it to a dropoff location.
We missed the first milestone of the project as hours before the competition, our breadboard shorted with all the electronics on it. An ongoing problem we had with the initial design was difficulty wiring, partially due to the layout of the robot but also due to the wires not fitting firmly in the breadboard. As a result, we resorted to using superglue to hold wires in place which seems to be conductive when heated. This caused a short between the 12V and ground lines, causing both motor drivers and most of the ultrasonic sensors to be destroyed. Our team had a working simulation for milestone 1 which we submitted in place of actually competing.

We redesigned the robot completely after the failure of the first robot. The new design put ease of wiring, assembly, and maintenance at the forefront. Also, due to the lack of time until the second milestone, laser cutting was used more extensively than the first design. The robot exceeded our expectations, completing the second milestone without any failures or need for preliminary trials. We did note that due to our algorithm, we may have needed to speed up the motors for the third milestone.

For the third milestone, a gripper was added to the robot. Some redesign was needed such as moving the back, left and right ultrasonic sensors to the top level to make room for the gripper. Our gripper was designed to accommodate a 2”x2”x2” wooden block, which in retrospect was a mistake. We chose this size block as it works best with an ultrasonic sensor. However, we should have used a 1”x1”x1” block and a TOF sensor instead. This would have simplified the gripping mechanism and lowered the height of the robot which was only 5 mm under the 300 mm limit. The first iteration of the gripper used a four bar parallelogram powered by a high torque servo. Due to oversized holes in the prints, the design did not work reliably. The day before the third milestone, we switched to using a mechanically simpler design in which a pinion drove a rack to push the block against a fixed gripper. This mechanism also proved finicky as it snapped right before our first trial, although we were able to glue it back together and ultimately succeed with very minimal intervention.

2.0 Detailed Rover Control Strategy

The rover control strategy uses the following terminology and coordinate system:

- A ‘tile’ is a 12x12in square in the maze, as shown in Figure 2.0-1.
- An ‘orthogonal’ orientation is orthogonal to the grid of the maze; aligned either straight up, down, left, or right relative to the maze.
- The coordinate (3, 6) is the center of the maze tile 3 down from the top, 6 right from the left. This coordinate system is shown in Figure 2.0-1.

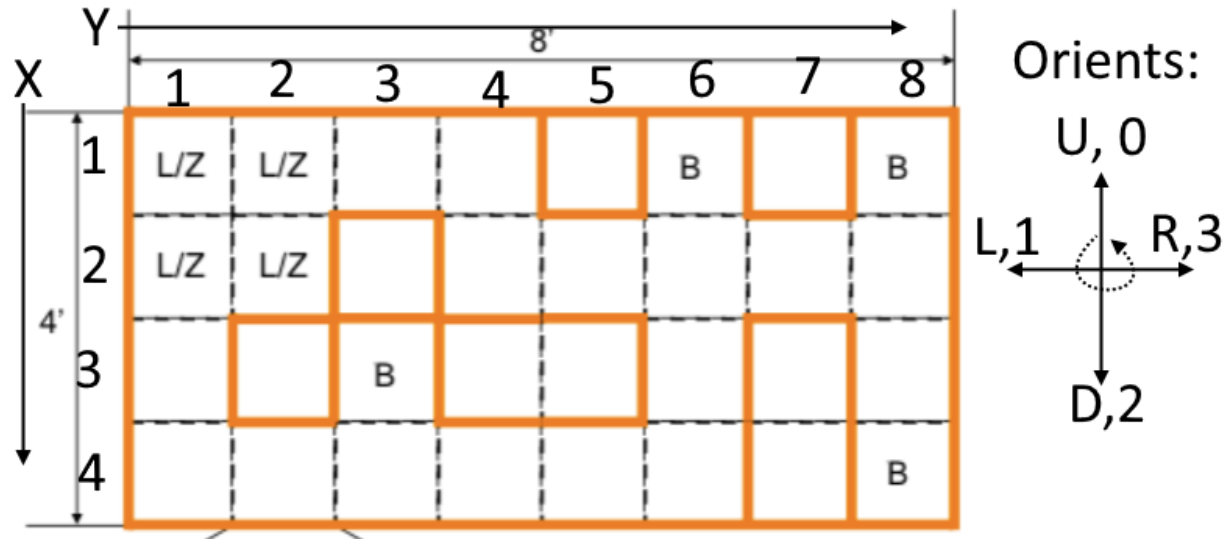

Figure 2.0-1: Maze Coordinate System

When developing the algorithms, the following design principles were used:

- Design simple systems and add complexity as needed. Don’t start with a complex system, as it may be sensitive to all components working properly and it will be difficult to debug and modify.
- Distrust the physical robot and increase trust as needed. Don’t assume the sensors are precise or even accurate, don’t assume the motors are even remotely reliable, and to the greatest extent possible only rely on sensor values when they were measured under ‘best case’ conditions.

The system that arises from these principles is very slow, very cautious, but also very safe–even if everything goes wrong at any point, it’s able to recover. However, we have a time limit of five minutes, so throughout the development process, we evaluated which physical components were most reliable, and redesigned the algorithm to ‘trust’ those components more to speed things up. Examples of this will be apparent below.

Part of why this ‘distrust until proven otherwise’ principle was used is that much of the code was written much earlier than the hardware was built; it had weeks of testing in the simulator, but none in a physical device. With dramatic hardware redesigns and testing data ongoing, it was considered safest to assume that no part of the hardware was guaranteed to actually work. The simulator was used with percentage errors at unrealistically high values–while they were later lowered somewhat, even now the physical robot behaves better than the simulated one (as stepper motors were used, which drive quite straight compared to the horizontal error setting in the simulator, and as real ultrasonics can be calibrated by adjusting for experimental measurement curves to get the error down).

2.1 Obstacle Avoidance

In order to ensure this project can be completed properly, obstacle avoidance was critical to get right, as it both helps the robot avoid crashing into walls and enables other components in the algorithm to function as intended.

The first and most important assumption made was: the robot will never drive straight. A lot of factors contribute to this, but ultimately there were only two methods of resolution the team came up with when figuring out how to solve this problem. The first method was using some type of analog controller (PID etc.) to ensure that during movement it would try to move as straight as it can, but was quickly discarded due to its complexity. The second method was conducting movement checks and using a mixture of trigonometry with discrete steps of motion to dynamically fix the robot’s position as it moved, which is the method we chose.

This algorithm works on a basis of distrusting the robot’s position and angle, mainly due to two factors: motors being desynced, and frictional losses or other contributing factors. While moving ‘straight forward’, the algorithm divides the forward movement into a series of discrete steps. With each step, it attempts to follow the wall to correct errors of distance from the wall and angle error relative to an orthogonal orientation. The movement of the robot is mapped below in Figure 2.1-1:

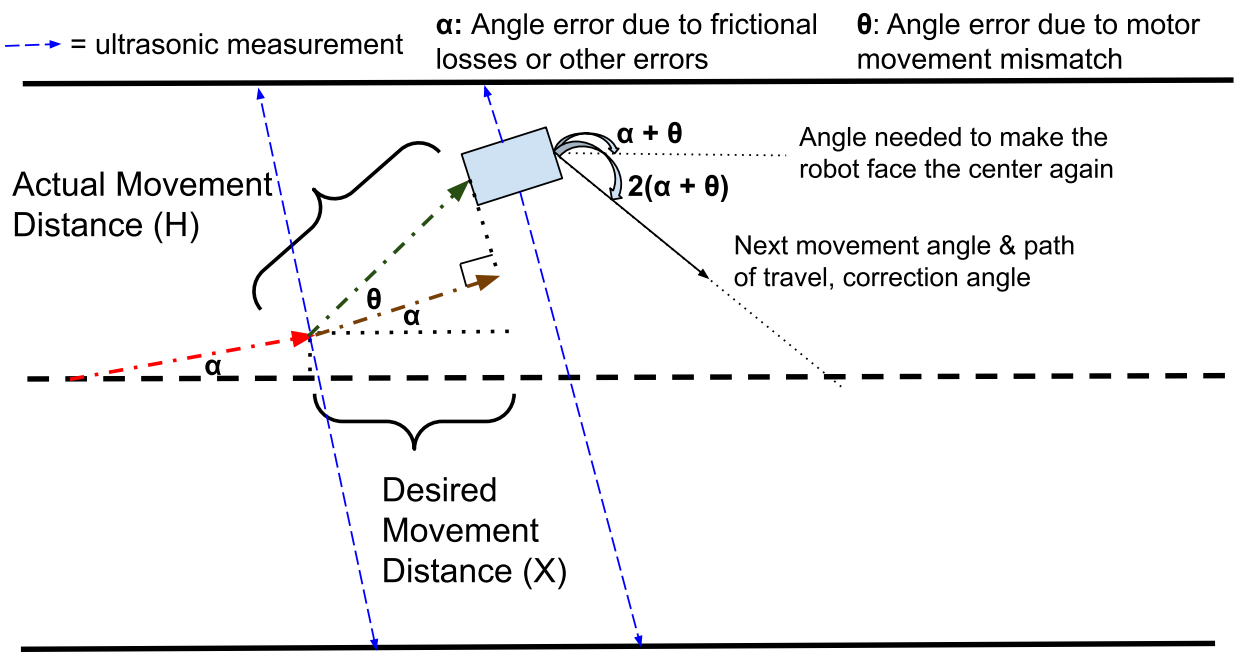

Figure 2.1-1: The figure above displays the positional situation of the robot in consideration of all the displacement and angular errors that would occur. Blue arrows are ultrasonic measurements. Theta represents a deliberate turn to correct for distance-to-wall error, Alpha represents unintentional angle error that existed at the start of the movement step.

By using simple trigonometry and solving for Theta and Alpha, distance-to-wall error and angle error could be solved and rectified. How this algorithm works is as follows: the ultrasonics will be checked on all four sides. A target ‘step size’ of distance to move forward in one step is calculated (typically 3 inches); if the front ultrasonic detects an imminent collision, or if we’re about to move into the center of the next tile (such that the center of the tile is 6 inches + (some multiple of 1ft) from the forward wall), the step size is adjusted for safety and stopping in the center of the tile.

Side values are measured and compared with each other to determine which distance should be used as the reference (the shorter ultrasonic measurement is taken as more precise), with a special case if both measurements indicate that there’s a wall on both sides (in a corridor, use both ultrasonics in tandem for additional precision). The robot determines its distance to the reference wall, compared to the ideal distance to be in the center lane of the tile; this is our measured distance-to-wall error. There is a caveat–this method assumes that angle error (Alpha) is zero; it has been found to be reliable for small Alpha, and sanity checks were placed on the calculations to prevent radical movements due to not-yet-corrected Alpha values.

The robot calculates the angle Theta it should turn to move diagonally, such that it moves ‘forward’ the desired movement value and normal to the wall a distance that corrects the distance-to-wall error. It drives that distance, then straightens up (turns -Theta).

Between each incremental movement, the robot measures side ultrasonic distances again; it compares the expected change in distance-to-wall error (should have dropped to 0 error if Alpha was 0) to the actual in distance-to-wall error to calculate Alpha, the actual angle error; it rotates to correct Alpha.

In effect, distance-to-wall error is fixed before each movement step, and angle error is fixed after each movement step. With each movement step, it continues to correct for remaining and additional errors, improving its alignment with the orthogonal orientation and the center of the lane with each movement. Using this, the robot perfectly travels without hitting any object and ensuring it’s centered at all times.

There is an edge case that breaks this, however–and in a maze, everything is made of edge cases, so it had to be fixed. In certain cases where there is a single guiding wall which switches sides during a step, or around certain corners, the wall-following could get confused, veering a couple inches off course or following around a corner. By combining wall following with positional information, it was possible to disable wall following for certain individual movement steps to correct these issues; this is covered in more detail in section 2.4 below.

2.2 Localization and Navigation

The initial basic concept for localization was based on using four ultrasonics–one in each orthogonal direction–and measuring adjacent walls. The ‘adjacent’ is key; while the ultrasonics could theoretically measure the distances to walls much farther away, distance measurements become less precise at greater distances; a ±10% error that might only be ±1in at 10in distance increases to ±4in at 40in; we didn’t want to trust the measurements under anything other than best case conditions.

There was also an IMU which could measure angle, but we didn’t want to use it due to initial concerns that they might be unreliable.

The robot ‘scans’ the adjacent walls, assigning the tile a ‘code’ based on whether the path to each adjacent tile was open or closed; it didn’t attempt to quantify how many tiles it could travel in each direction; only open or closed. The ‘codes’ for a tile are in

binary; 4 bits, with each bit representing one of the four directions; this format is shown in Table 2.2-1.

Table 2.2-1: Binary representation of maze tile

Bit 3 (dec8)	Bit 2 (dec4)	Bit 1 (dec2)	Bit 0 (dec1)
Right wall closed	Bottom wall closed	Left wall closed	Top wall closed

Using this format, we can 'hardcode' the layout of the maze; this is shown in Figure 2.2-1, with wall-codes for each tile written in based on reference orientation pointing up. Code 15 means 'all walls are closed' and was used for blocked tiles inside the maze area.

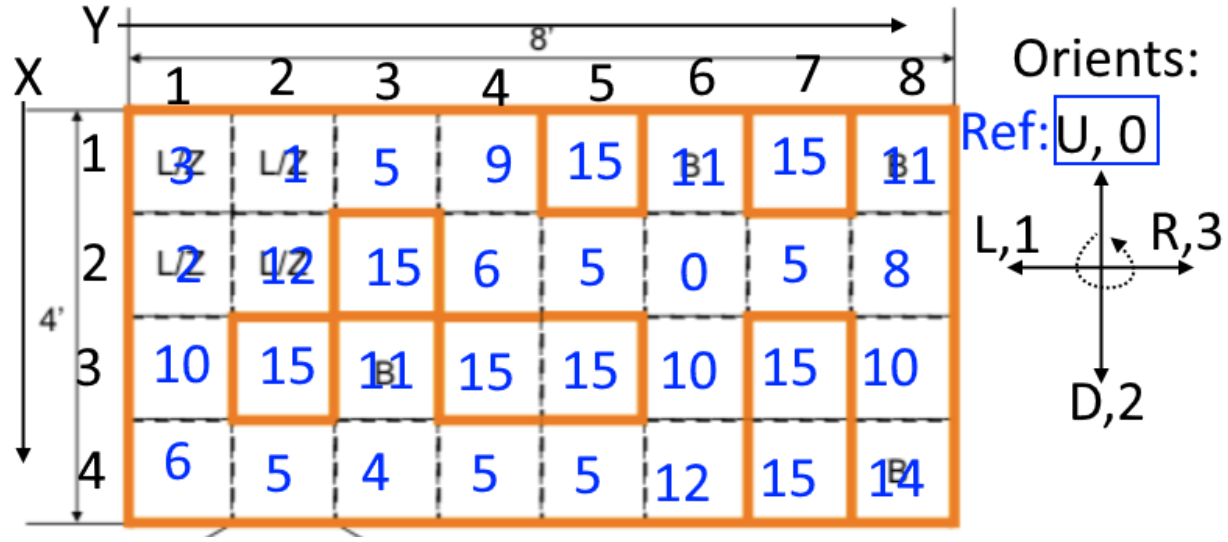

Figure 2.2-1: Binary representation of the maze, written here in decimal

The motivation for this format for a tile is that we can *rotate our frame of reference* by *rotating the binary number* (shifts each bit right or left, and moves the rightmost/leftmost bit to the leftmost/rightmost position; done in Matlab with the *bitror* or *bitrol* function). This is especially useful because, under our initial assumption that we couldn't trust the compass, when the robot starts localizing, it *doesn't know what direction it's facing*. We had developed a way for the compass to line up with an orthogonal orientation based on ultrasonic measurements, but this didn't tell the robot which orthogonal orientation this was–this method of lining up isn't detailed further in this report, as it was later eliminated from the final methodology.

If the robot were starting in the bottom-right tile, we'd expect it to measure tile-code (based on adjacent walls) of 14 if its frame of reference was oriented up. But if, for

example, its frame of reference were pointed left, it would measure 13–but rotating binary ‘13’ clockwise once (corresponding to rotating our reference frame to point up) gives 14.

The robot knows what its tile looks like without knowing orientation, and it knows what the maze looks like (for any orientation; it can rotate the maze map as needed). However, this (usually) isn’t enough to localize–there’s only 1 point on the maze where there’s a unique match for a single tile even if you don’t know orientation (point (1, 6), code 0, the 4-way intersection). The solution is to map multiple tiles.

The following is an example. For this example, for the sake of simplicity assume the robot knows its ‘global’ orientation at all times–as previously described, it actually wouldn’t and it would need to check all four possible orientations for a unique match, which would probably need to move an extra tile.

The robot starts at a random tile. It measures the tile, and it’s code 12 (open up and left). There are two different tiles with code 12 on the map in Figure 2.2-1; the robot could be in either one. The robot chooses to move up (this isn’t a random choice; there is a predefined scheme for choosing path based on avoiding retracing steps and turns, but the impact is very small even compared to a random choice). The robot moves up 1 tile (12in), using wall following and obstacle avoidance, and measures it again. Now it sees code 10. There are three places in the maze with code 10, but only 1 place with a code 10 above a code 12, position (3, 6) in Figure 2.2-1. The robot knows where it is, and localization has finished.

Under the hood, it’s rather more complicated–there’s a lot of sanity checks, the ability to map out arbitrarily large regions of the local maze, tracking of current orientation relative to starting orientation, etc, but these are mere implementation details.

Through testing, it was decided that localization was taking too long; there’s a time limit, and even moving 1 extra tile during localization eats up too much time, both to move to that tile, then to move back towards the target destination if it happened to be the wrong way to go. It was found during testing that the IMU could be made reliable to better than ±30° (this was done by disabling the magnetometer and relying on gyroscope and accelerometer, then calibrating the starting position of the gyroscope when powering up the robot before placing it in the maze). The algorithm was changed to ‘trust’ the IMU; instead of starting up, needing to align to an orthogonal direction based on a lengthy process using ultrasonics, then spend extra time localizing and then verifying which way it was facing, it could immediately align to an orthogonal (closely enough that the angle correction built into obstacle avoidance could take it from there), and know which

orientation it had, speeding up the whole process. In practice, the robot was able to localize accurately with at most only a single 1-tile movement, whereas before it had taken up to 3.

Once the robot had localized, navigation was relatively easy by relying on obstacle avoidance and knowing its position. At any point, the robot doesn't need to know its whole route–only its current position, and therefore its *next step*. The list of 'next steps' for any current position for a given end point is simple enough that it was hard coded; for an example, see Figure 2.2-2, a diagram of 'next steps' for what adjacent tile to move to to ultimately reach the bottom-right dropoff zone.

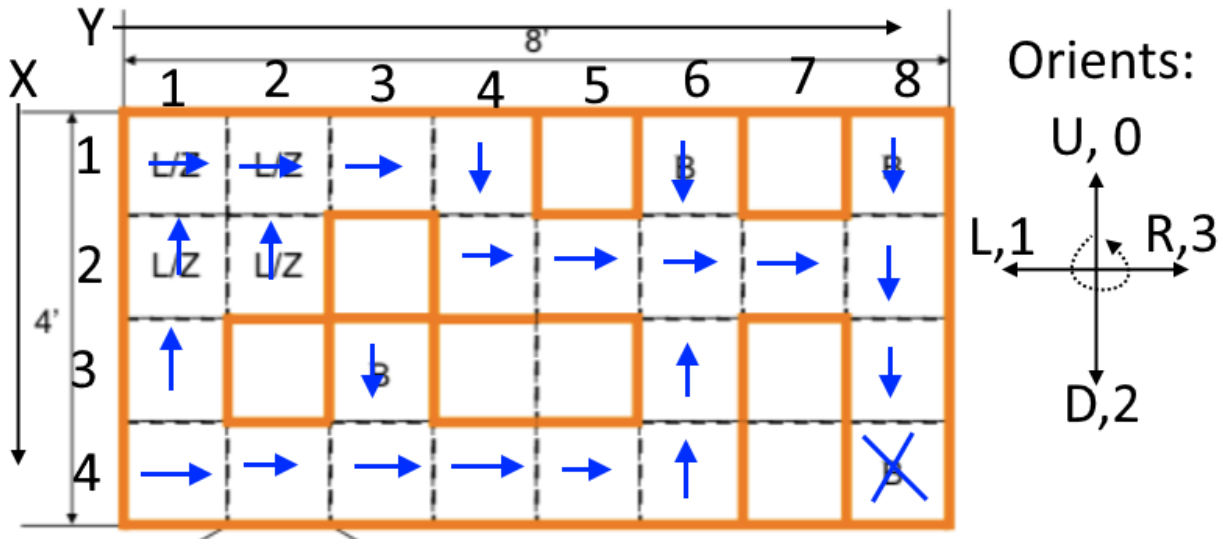

Figure 2.2-2: Route to bottom-right dropoff zone

The basic loop is as follows:

- Compare what the robot thinks the position should be (based on localization and previous movement) to the actual walls. This is done by measuring the wall code of the current tile, and comparing it to the map. If it doesn't match, restart the localization process. In practice, localization was made fast enough and movement reliable enough that this didn't cause too much delay.
- If we've relocalized, or the believed position is consistent with the wall measurements, get the next movement direction (and adjacent position) we move to on our route.
- Move 1 tile in that direction and update our believed position. Obstacle avoidance is utilized, and augmented by our knowledge of our position in the maze.
- Repeat; or, if we've reached our final destination, move on to the next stage.

2.3 Block Delivery

The process of picking up and delivering the block was expected to be straightforward, and would utilize two dedicated ultrasonics: a low forward facing ultrasonic for detecting the block, and a downward facing ultrasonic inside the gripper to determine whether the block was in the gripping area. The plan was, the robot would lower the gripper once it entered the loading zone, scan for the block in an area in front of it (by comparing the ultrasonic measurement from the lower and upper forward ultrasonics as it rotated), and if found, move straight towards it until the downward ultrasonic saw the block. It would then close and raise the gripper. If it didn't see the block in the initial sweep, it would advance into the loading zone and do several much wider sweeps; if it still couldn't find it, it would 'abandon' the block (assuming that either the block finding wasn't working or there was no block in the zone), and leave the loading zone to go to the dropoff zone.

This ran into an issue we hadn't expected; while it worked fine in a simulator, the physical robot was vulnerable to what we refer to as the 'angle effect' for ultrasonics. When an ultrasonic measures a surface at an angle, it gives reasonably accurate readings until ~30° tilt, at which point the measurements are chaotic, giving wildly different values. We had mistakenly assumed that while the measurement to the block and wall would be vulnerable to angle effect during the sweep, while both measurements would be chaotic, they would be the same (chaotic) value when both the lower and upper forward ultrasonic were measuring to the wall–they would be measuring the same surface at the same distance and same angle, so it had seemed reasonable they would read the same number. Then, if one saw the block, we simply had to check for a different reading.

It was not until too late that we discovered this assumption was incorrect–there were delays in building the physical gripper that meant we could only test the hardware very late into the process. As it turned out, even when measuring the same surface at an angle, the two forward ultrasonics gave drastically different values, invalidating this method for detecting the block; a failure to abide by the original design principle of avoiding assumptions about hardware had led to issues down the line. An alternative strategy was quickly developed; there was insufficient time to test it fully with the real robot.

As the block position was known beforehand to be in one of five places, the new algorithm implemented utilizes a mixture of brute force with smart checks. The algorithm uses a path following method, as the robot reaches the loading zone, it will lower the gripper and scan the entire region around the entrance to see if it can find the block, if it finds it then it heads towards it and grips it, if not it moves forward as the gripper is lowered and continuously scans using the two block ultrasonics (one pointing forward

and one on the gripper pointing down), once the ultrasonic detects the block the gripper closes and the robot returns to normal functionality in order to traverse to the drop off zone, centering itself and placing the block down. The path that the robot follows during the brute force pickup method is outlined below for both the bottom and left side entrance of the loading zone, shown below in Figure 2.3-1.

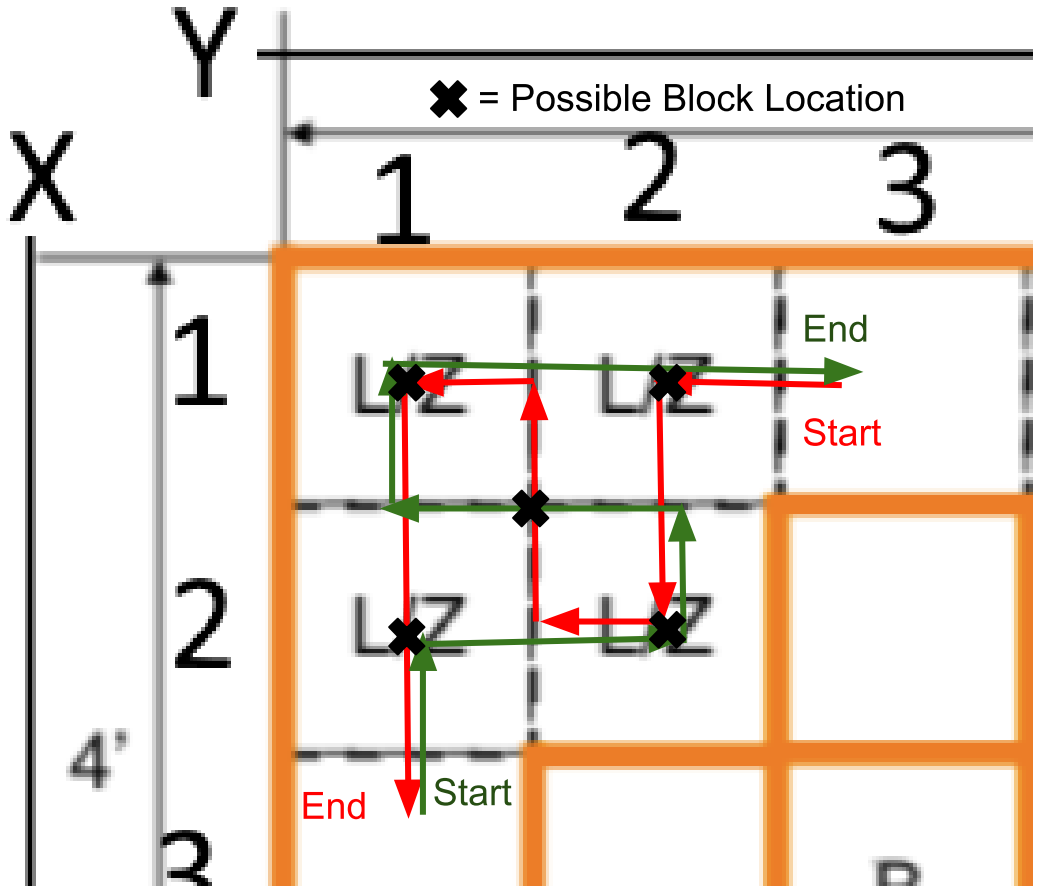

Figure 2.3-1: The path the robot follows in both entrances to try and find the block.

2.4 Integration

The code was split into two independent halves: the Matlab code which did the bulk of the control, and the Arduino code loaded onto the robot. The Arduino code was configured to act as a simple drone, acting on basic commands received over bluetooth and sending a response when finished. Commands were limited to basic things, such as ‘read left ultrasonic’, not advanced actions like ‘move forward 1 tile, following the wall’, which were handled in Matlab. The simulator was modified to support many of the same commands, allowing testing in the simulator using much of the same code.

The code is divided into several interdependent modules, mainly: localization, wall following, navigation, and block handling, in addition to small modules such as bluetooth

communication. The interdependency meant that care had to be taken in integration; for example, localization determines our position, and depends on our ability to drive straight without hitting anything, however, wall following depends on knowing our position to disable wall following behavior at key points for improved reliability.

The main code that controlled overall robot behavior was mainly responsible for tracking our stage at the maze (ex. Initial localization, going to loading zone, going to drop off), invoking the relevant module, and linking things together. The first stage was always initial localization; even in deliverable 1, where we only needed to avoid collisions, the position information was useful for wall following. Initial localization is ‘cautious’; because it doesn’t know its location for wall following while still localizing, it uses a modified motion algorithm which uses additional sensor measurements and movements to re-center along the tile with each tile-movement. This slows things down, but improves reliability.

Once we have position, the main navigation loop begins. We follow the basic loop given in section 2.2 above; there’s a hard-coded list of positions where wall following should be disabled for part of the movement (ex. the middle 3-inches of the movement between (1,4) and (2,4), where the wall switches sides). If we didn’t disable wall following, actual collision is still rare, but it leads to the robot veering off course and needing to re-localize more frequently.

Once we reach our end-position for the current stage (such as by reaching the loading zone), we activate the relevant module (such as block-handling), send signals and log information, and once that completes, we begin navigation along the next route. Whenever there is a discrepancy in position, we quickly re-localize.

The main loop also has a variety of ‘quality-of-life’ features, such as the ability to send manual commands to the robot before starting the automated process (used for testing and to set target drop off zones during setup), the ability to write to a log file on the hard drive, and the ability to communicate with either the simulator or real robot using the same code.

3. Final Results

We were not ready to test our physical robot for the first milestone due to an electrical short occurring hours before the competition. However, the MATLAB simulation performed perfectly in numerous starting positions even with unrealistically high sensor and motor error.

For the second milestone, we redesigned our robot to minimize its diameter by increasing its height. We also switched from an Arduino Uno to an Arduino Mega. This allowed us to give the ultrasonic sensors their own echo and trigger pins and minimize the signal interference in the low level code. We tested our robot in the Myhal Centre maze the day before the test day. It functioned perfectly on the first trial and only ran into one problem during further testing due to lack of motor calibration. On the test day, our robot continued to function perfectly and completed the maze on the first trial in 3 minutes. This was faster than we had expected and was a result of the ideal starting position which allowed us to localize within the starting square.

For the third milestone, we missed our first trial as our gripper snapped in multiple places minutes before our trial time. We had tested the gripper extensively so we knew that it gripped the block with sufficient strength.

We used multiple glues and epoxies to mend the gripper in preparation for our delayed trial. We were not able to test the block finding algorithm in the maze due to lack of time but it did work in the simulator. The entire integration of localization, block detection, block pickup, and block dropoff was working in the simulator. However, we knew there would be issues in the actual maze so the first trial for us was more of a practice run than a test. The robot functioned better than expected at picking up the block. However, due to an issue we had with our servos causing them to jitter significantly, the block was dropped after the robot had exited the loading zone. The robot continued to the dropoff point successfully, albeit getting partially stuck on the way. We believe the robot got stuck as we didn't have time to calibrate the motors and sensors. For our second trial, we increased the clamping force on the block to prevent accidental loosening. Additionally, we modified the algorithm to raise the gripper before turning in the loading zone as we were very close to hitting walls in the first trial. This proved to be a mistake as in the second trial, the robot lowered the gripper onto the block. Once we moved the block from under the gripper, the robot picked up the block and delivered it to the drop off zone in approximately 4 minutes without any further intervention.

4. Discussion

4.1 Mechanical Changes

The mechanical design went through the most vigorous modifications. The initial design was too compact and did not put wiring into consideration. Moreover, despite the compactness, the overall radius of the robot was still too large that it created a lot of difficulty navigating the tight maze. The structural support in the initial design was made so that each layer is dependent on the previous layer. For example, the top layer is supported by the bottom layer via the battery pack. Although the design achieved compactness, the assembly process's difficulty was not accounted for. Thus when constructing the robot, the top layer was not able to be completed due to the lack of structural support from the battery pack. Moreover, once the battery pack was fixed in place, we were not able to access the wiring anymore. The first design can be referred to in Figure 4.1:

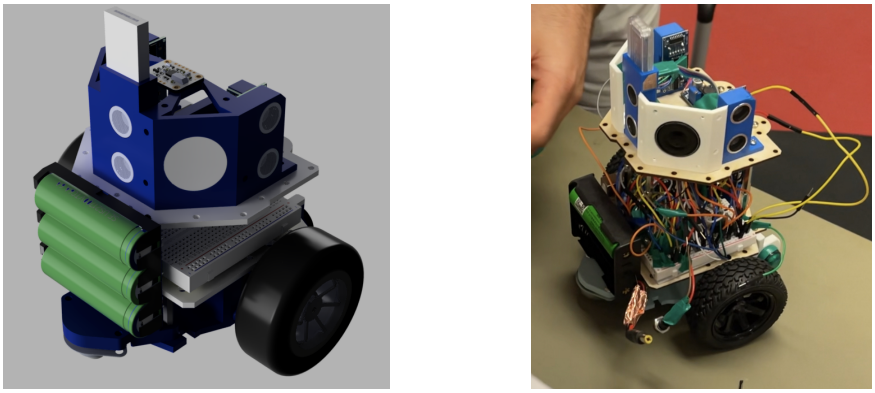

Figure 4.1. Initial robot CAD design and the final assembly

Therefore, the team had a drastic redesign after the first milestone to correct all the mistakes. The overall size of the robot became our biggest strength in the contests as we managed to reduce the size from 205mm in diameter down to 170mm. This reduced 35mm was tremendously helpful in reducing the chance of hitting obstacles. Second, the design was made more modular so that the team can construct it layer by layer. Upon assembly, each component did not interfere with one another and can be tested and disassembled individually. Third, we moved all the electrical wiring to the top layer for more space and easier modifications, as can be seen in the image below:

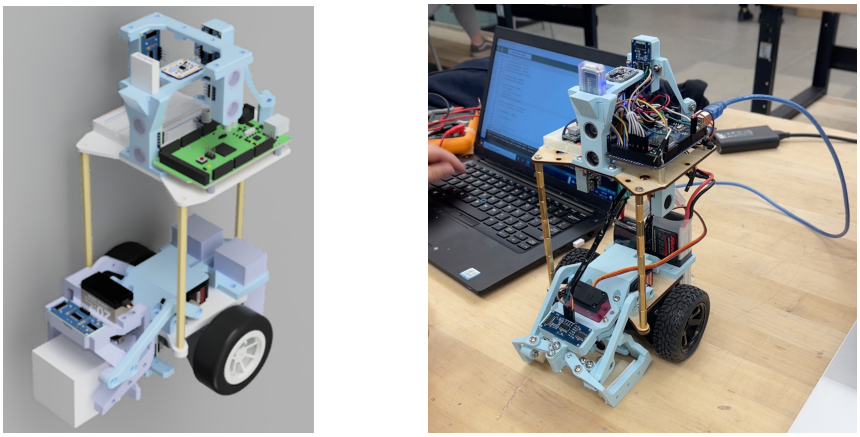

Figure 4.2. CAD design of the final robot and the robot's final assembly

The initial gripper design was made out of two four bar linkages that were powered by two high-torque servos. This design failed several times due to the low rigidity from the 3D-printed linkage parts. Additionally, the joints for the bar linkages were designed to use M3 screws and nuts instead of bearings, thus creating a lot of friction. Due to the accumulated frictions and vibration movements, the parts tended to get loose, causing the gripper to fail to pick up the block. Overall, due to the high complexity and large number of moving parts, the team abandoned the design and modified it into a rack-and-pinion mechanism as shown in the following:

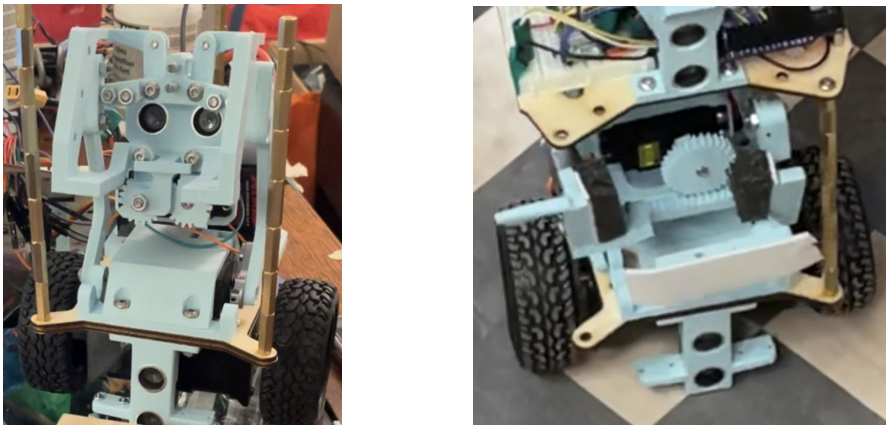

Figure 4.3 Initial 4-bar linkage gripper design versus the final rack and pinion gripper design

The updated gripper design was drastically simplified and was directly driven by the high-torque servos without reducing the gripping torque. This allowed the team to tune the servo turning angles more easily without breaking the linkages. The resulting gripper was also able to retract back into the robot body without increasing the effective robot diameter as well.

Finally, the team noticed the ultrasonic sensor on the gripper was not able to detect the block accurately. This could be resulted from the interference from the gripper or the tilted angle of the block in respect to the ultrasonic sensor. For future improvement, a time of flight sensor (VL6180) could be implemented to replace the ultrasonic sensor for distance measuring.

4.2 Electrical Changes

In the first milestone, having the idea to simplify the design as much as possible, we decided to use an Arduino Uno with all the ultrasonic sensors sharing the same echo pin. In addition, speakers and high-voltage LED lights were added for special effects outputs. The design was sound on paper, but the team quickly realized that there were not enough I/O pins on the Uno if more sensors were to be added in the later milestones. To control the speakers and the high-voltage LEDs, MOSFETs were needed to control the ON/OFF signals, which increases the number of components and the likelihood of failing on the circuit part. Moreover, each MOSFET requires an additional signal output from the microcontroller, in other words, an additional digital pin will be taken up from the Uno. Finally, when the ultrasonic sensors were sharing the same echo pin, not only from the Arduino code side the readings from each sensor would be delayed, also the signal from the echo pin would interfere with each other, unable to generate accurate distance data. The team tried to add a diode to each echo pin to prevent unstable signals to resolve the issue but without success.

Therefore, we simplified the design by replacing the uno with a mega for its multiplied I/O pins and additional memory. In this case, we were able to attach each individual ultrasonic sensor with a trig and echo pinout from the microcontroller. Speakers and high-voltage LEDs were abandoned to eliminate the need for MOSFETs. Instead, normal low-current LEDs were installed for outputting signals. In the end, the team managed to organize the wiring within one small breadboard and the circuit was reliable enough that no modification was required by the team.

4.3 Firmware Changes

One of the main changes in software is the implementation of the 9-axis IMU sensor. Although compass is built into the IMU module, the team quickly found out it was hard to calibrate the sensor each time the program was re-uploaded and the accuracy was not ideal for our use case. On the other hand, the built-in gyro was extremely accurate, but it did not give us the orientation in respect to the ground frame. Hence, the team modified the software with an additional orientation calibration for the robot – the robot was calibrated manually when powered up before being set in the maze. In this case, instead of investigating the ground truth orientation in respect to the world frame, having the angle difference is enough for us to tell the robot’s world-frame orientation.

The greatest strength from the software department was the ability to relocate the robot when the robot was placed into another location manually. This helped us greatly in the actual testing – not only saved us a vast amount of time without needing to reset the robot, it also gave the robot the ability to relocate itself anytime after the block was picked up or when the robot was disorientated in the maze by unforeseen incidents.

4.4 Learning Experiences

The biggest lesson learned for the team is to have a better project timeline in the future; getting hardware finished earlier would have allowed more testing and tuning of the software. In milestone three, we could not get the block locating software tested until the actual first trial. Within the limited time between trial one and trial two, we managed to modify the code quickly enough to allow better gripping and locating movements in the final trial. Given more testing time, the results could have been a perfect run.

From the robot design aspect, we learnt that modularity and ease of assembly is crucial to speed up the testing process. Hardware is unreliable and is required to be swapped out easily and frequently. Having a design that is easily accessible to modify the wiring and change the broken components is crucial to expedite the design process. On top of that, the team also learnt and put into practice various fabrication techniques for wiring the electronics. Despite incidents of the battery short-circuiting and the Arduino board frying, the team eventually had the working robot with the wires well organized. However, battery hazards and safety protection circuit features are to be considered and should be added in the future designs.

Overall, the team gained great skills in mechatronic design, and after comparing our design with other teams', the greatest take away is that there is no standard solution for a mobile robot design, and only by consistently iterating and improving can the design withstand the unpredictability of the real-world.

5.0 Bill of Materials:

The following is the bill of materials for components included in the final robot. All parts were purchased by the team, with the exception of the bluetooth module and the two servos, which were borrowed.

Item	Unit Price	Quantity	Cost
Stepper	$14.50	2	$28.99
Servo	$22.49	2	$44.98
Wheel	$5.40	2	$10.80
Caster	$2.68	1	$2.68
Battery	$23.99	1	$23.99
Ultrasonic	$2.40	6	$14.40
Arduino Mega	$29.95	1	$29.95
Breadboard	$7.90	1	$7.90
LED	$0.31	2	$0.63
Bluetooth - HC05	$12.99	1	$12.99
IMU - BNO055	$33.74	1	$33.74
Buck Converter	$7.00	2	$14.00
Motor Driver - A4988	$5.11	2	$10.21
Jumper Wires	$11.18	1	$11.18
			$246.43