# MIE444 - Mechatronics Principles

## Final Project Report

December 16th, 2022

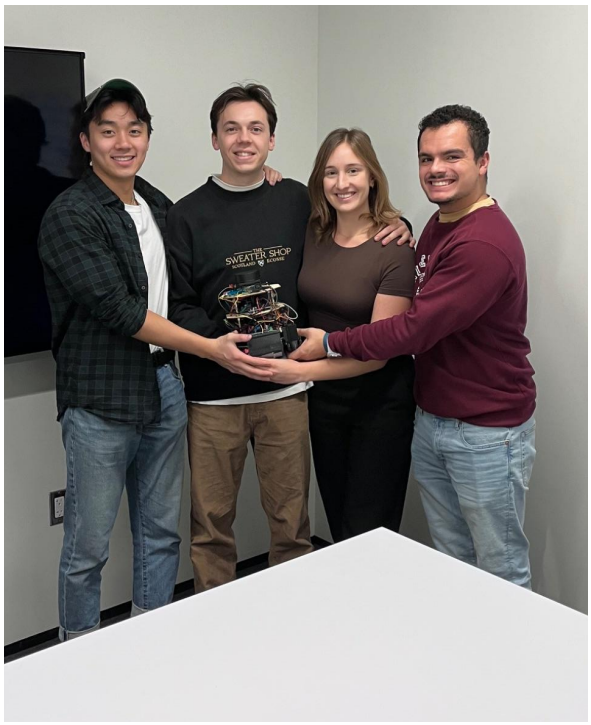
Group 11

Liam Toner (1005178754)

Chris Tong (1005661874)

Andres Cervera-Rozo (1005469217)

Nathalie Cristofaro (1004838009)




Link to milestone 3 videos:
https://drive.google.com/drive/folders/1ZH4aUpbSbr8QGiYuTMWXnF71OBrIfj4q?usp=sharing

# Table of Contents

# 1.0 Executive Summary

This report investigates the performance of group 11's robot for the Mechatronics Principles course term project. The project consisted of designing, building, and programming a robot that would complete three milestones throughout the semester.

The first milestone was obstacle avoidance, where the robot had to travel 20ft in a maze without hitting walls. The team's robot completed this challenge successfully, with the robot only needing one adjustment to continue on its course. In order to complete this achievement, the team used a Raspberry Pi, LiDAR sensor, and an Arduino Uno to control the robot systems, with the Raspberry Pi acting as the main controller. The Raspberry Pi and LiDAR communicated with a remote PC through the ROS framework to determine the direction and response of the robot. The PC would send PWM values to the Arduino Uno through the Pi to control the direction and speed of the omni-wheels.

The team was not able to complete the second milestone, which involved navigating and localizing in the maze, due to the short timeline available to debug the Hector SLAM mapping package. However, the initial plan to complete this task was to model the robot as a URDF file, use Hector SLAM to map the maze, and use AMCL to create a particle distribution probability map that would determine the final location of the robot by comparing a live map with the pre-generated reference map. A Gazebo simulation was intended to be used for debugging and as a back-up plan to show the robot localization in a simulation.

The third milestone combined the previous milestones and added the additional challenge of locating and delivering a block. The robot was able to successfully complete the third milestone by combining the original idea for milestone two with some alterations to facilitate the programming. A sweeping mechanism was added to the robot frame, which was able to efficiently acquire, contain, and deliver the block to a drop off location. The robot was also capable of localizing at the four-way intersection in the maze.

Overall, consistent with the agreed upon goals at the start of the project, the team learned a great deal about Hector SLAM, ROS architecture, and other common packages and frameworks used commonly in the mechatronics industry. The physical design of the robot was very successful, whereas programming posed a few challenges that would have been resolved with more time, or if the team had chosen a simpler robot design and concept for the project.

# 2.0 Detailed Rover Control Strategy

If the reader is not familiar with ROS, in order to comprehend the terminology in the following sections of the report, the team has recommended the following online references which provide definitions for terms such as node, topic, publish, subscribe, etc. and general information on the ROS framework used for this project.

Online resources:
[1] https://wiki.ros.org/ROS/Introduction
[2] https://ros.org/

## 2.1 Obstacle Avoidance Strategy

In order to complete milestone one, the robot needed to achieve an efficient obstacle avoidance strategy to navigate through the maze without colliding into walls. The robot accomplished milestone one using a LiDAR sensor, remote master PC, Raspberry Pi, and Arduino to collect and send data to the motor controllers which controlled three DC motors with encoders, each attached to an omni-wheel.

### 2.1.1 Component Software and Integration

The Raspberry Pi is essentially a small computer that is compatible with multiple distributions of the Linux operating system. The team's Raspberry Pi used the Ubuntu 20.04 distribution in conjunction with the Robot Operating System (ROS) framework. The ROS framework is a free open-source software that defines the various tools, libraries, and interfaces used to connect to the actuators and control the systems of a robot. The use of this framework facilitated the integration of a LiDAR sensor into the robot design, which would typically be very complicated without utilizing pre-built libraries. The Raspberry Pi comes with an integrated wifi connectivity option, which the team used to connect to an external PC. The external PC used by the team was dual-booted in order to use Linux as a second operating system. The PC's Ubuntu system also had the ROS framework installed. Once the Raspberry Pi and external PC were confirmed to be on the same wifi network, the bashrc files were configured to link the IP addresses of both devices; the PC was set as the master and the Raspberry Pi as the slave so that the Raspberry Pi could be controlled from the PC.

The LiDAR was connected to the Raspberry Pi using a standard USB cable. The Pi would take in the raw LiDAR data and convert this data into a Laser Scan message which was published to a /scan topic. This procedure would transform the data into a global variable that could be read by the PC. The PC accessed this data by subscribing to the /scan topic.

### 2.1.2 Movement Strategy

The LiDAR sensor gathered data by taking 720 distance and angle measurements during one full 360° rotation. This data was sorted into a 720 element array which was divided into a 12 section circular map, as seen in Figure 1.

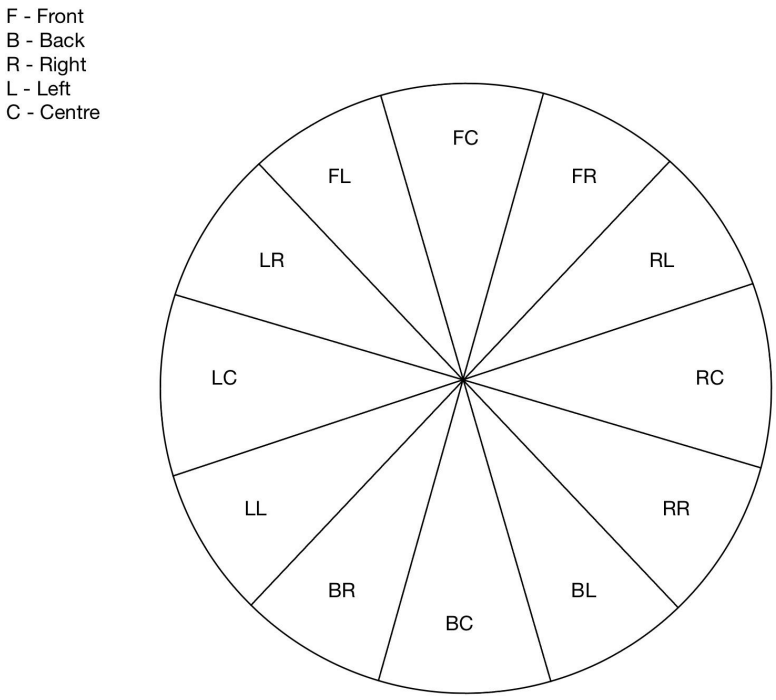



**Figure 1:** LiDAR Map Visualization.

Each slice of the map contained 60 data points that helped the robot determine if an obstacle was present in the region. The robot was designed to move in such a way where the front-center slice was always the preferred movement direction. If an obstacle was not detected by the LiDAR in this region, the Arduino would send a “move straight” command to the motor drivers and the robot would move forward. If an obstacle was present in the front-center slice, the robot would check if obstacles were present in the next two outer slices, and continue around the circle until an obstacle was not detected in one of the slices. Once a slice with no obstacle was found, the Arduino would send a “rotate” command, and the robot would align the front-center slice with the slice that contained no obstacles before moving straight. If every slice contained an obstacle, the robot would rotate and move in the direction of the furthest obstacle.

### 2.1.3 Calculating and Generating Movement Commands

The output of the obstacle avoidance algorithm used in the robot movement strategy was a command velocity that would either indicate that the robot needed to rotate or

drive straight. As there were three omni-wheels and because the wheels were multi-directional, the node that received the command velocity performed matrix multiplications in order to convert the command velocities into a pulse width modulation (PWM) magnitude and rotation direction for each wheel based on their geometric configuration. These calculations were done on the PC and transmitted to the Raspberry Pi, which was connected to the Arduino using serial communication. The Pi ran a script to ensure that the Arduino was subscribed to the relevant topic, and published data accordingly. The gathered PWM signals and rotation directions were combined into a smaller 6 item array message that was received by the Arduino. 1s and 0s in the message would indicate whether the wheels would need to turn clockwise or counter-clockwise, respectively. These direction commands were sent to the motor controller via digital write pins on the Arduino. The PWM signals in the message were sent to the motor's enable pins via the analog write pins on the Arduino, and these signals would determine the motor speed. The LiDAR data retrieval and communication with the other components occurred at a speed of 10Hz, which explains why the robot movement was smooth and efficient.

### 2.1.4 Milestone One Challenges and Solutions

For the robot to effectively avoid obstacles, some obstacle avoidance script parameters were created in the code which helped the robot determine when it should begin turning depending on its footprint size. These parameters also helped the robot drive straight, since they would regulate how close the robot should be to walls while it was driving. By adjusting these parameters, the team was able to more effectively avoid the maze walls; however, the wheels would often graze corners as it was difficult to find the correct parameter balance. In addition, the wheel PWM signals had to be individually adjusted to account for some motors which spun faster than others at the same PWM. By adjusting these signals, the robot was able to drive straight when it was not surrounded by obstacles.

## 2.2 Localization and Navigation Strategy

Milestone two required the same hardware from milestone one along with a second Arduino to control the LEDs that indicated robot localization and position in the maze. The communication between components was the same as in milestone one, however the source of the command velocity topic was different. The team was unfortunately unable to complete milestone two, however the following sections describe the original plan the team devised for this milestone. The team ended up using a different procedure for localization and navigation due to time constraints and coding complexity, which is discussed in Section 3.0.

### 2.2.1 Creating Simulation in Gazebo

In order to localize and navigate inside of the maze, it was faster to test the code inside of a simulation as opposed to relocating to a physical maze every time. Additionally, simulation was not limited by battery life, there were no hardware bugs, and the same code could be used for simulation and real life. Although most teams opted to use Matlab for their simulation, ROS integrates better with Gazebo: a 3D robotics and physics simulation software. Since Gazebo is open source, there are many readily available plugins that allow for the simulation of different sensors like LiDAR and TOF (time of flight). The role of Gazebo in this project was to test and visualize the LiDAR, the obstacle avoidance code, the localization code, and the pick up mechanism code. This section describes some basic terms and the process that was used to set up the simulation in Gazebo.

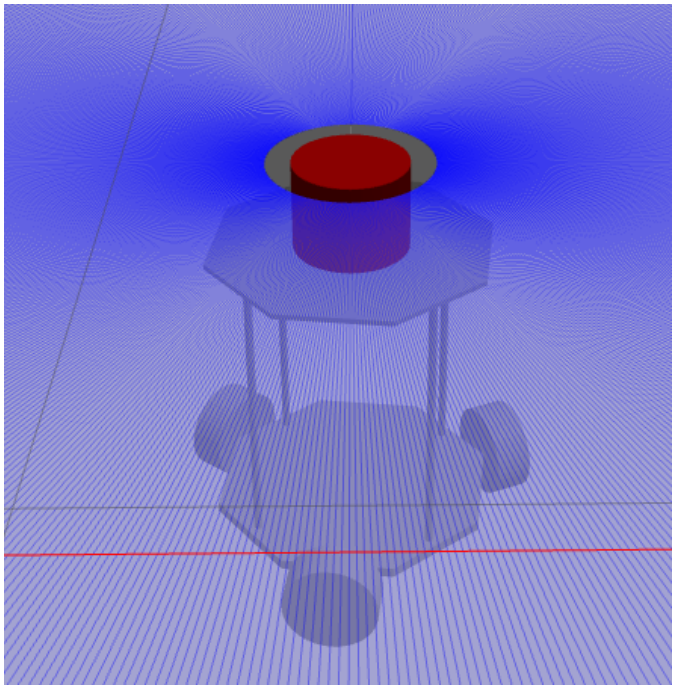


**Figure 2:** Robot URDF with Red LiDAR in Gazebo.

In order to import objects into Gazebo like the robot and maze, the objects must have a unified robot description format (URDF) file type (see Figure 2): a file format that describes the physical properties of an object like weight, center of mass, and dimensions. Mechanical engineers are familiar with using CAD to model 3D objects, however, a URDF file is generated using a coding language called extensible markup language (XML). To make the task simpler, Solidworks was used to model the robot and then a Solidworks plug-in was used to convert the CAD file (.sldprt) into a URDF file (.urdf). From there, the bulk of the code was generated by the plug-in and small tweaks

to the code to specify the linkage masses, moments of inertia, and visual representation were added. A collision box around the robot was also defined, which specified the allowable movement radius of the robot body before it would collide with obstacles in the simulation. Each linkage was also given a separate central coordinate system so that the link movement could be modeled relative to the robot in the graphical user interface (GUI) of the URDF plug-in.

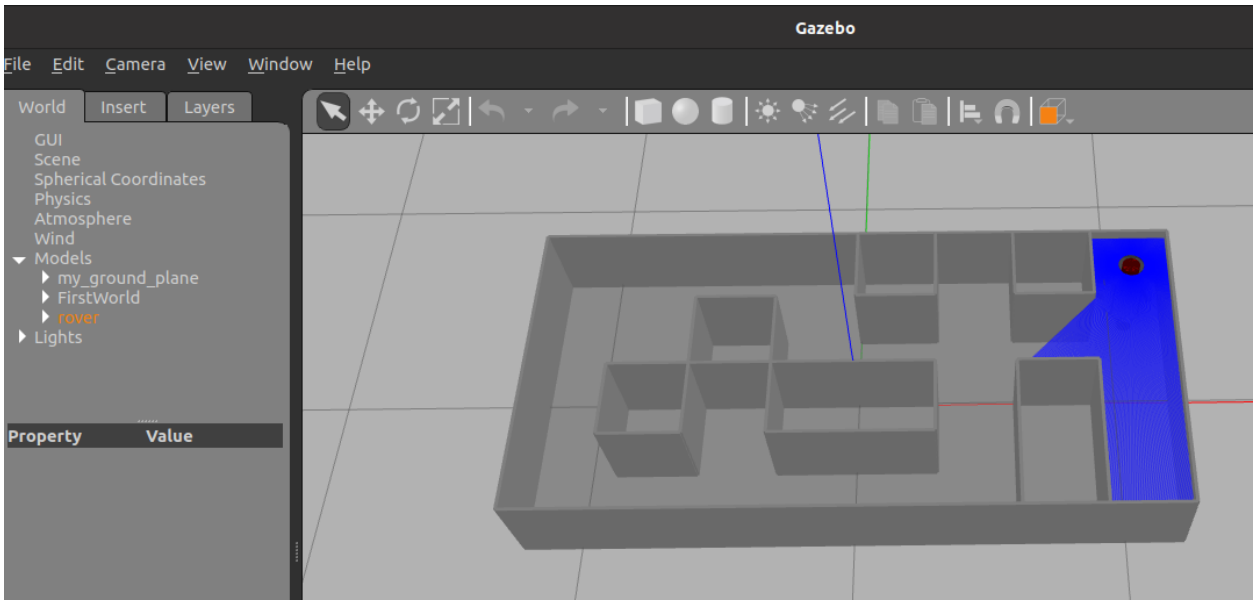



**Figure 3:** Gazebo Generated Maze (robot with LiDAR seen in top right of maze and blue highlight represents the field of view of the LiDAR).

The maze was created differently, using the editor tool in Gazebo, the team created a scaled maze with walls in the same positions as the course's maze (see Figure 3). Two plugins were then added to the URDF to better simulate the design; libgazebo_ros_laser.so to simulate LiDAR data and record it in /scan (see Figure 4) and libgazebo_ros_planar_move.so to represent the omni-wheel drive system of the robot. In order to localize the robot in the simulation, the /scan data was read by the AMCL ros package (more on this in 2.2.3). A robot's pose describes the robot's position in a space. In this case, the robot operated in a three-dimensional space and thus had six degrees of freedom: three cartesian (x, y, z) and three angles (roll, yaw, pitch). Any time the robot would need to change poses, an original reference position as well as a final position was needed. The drive system plug-in accepted command velocities from a topic and moved the robot (in x, y, z, roll, yaw, pitch velocities where z, r, y are zero) based on the pose changes in the Gazebo simulation. A launch file was then created to spawn the robot URDF inside of the world (maze) created at a predefined location.

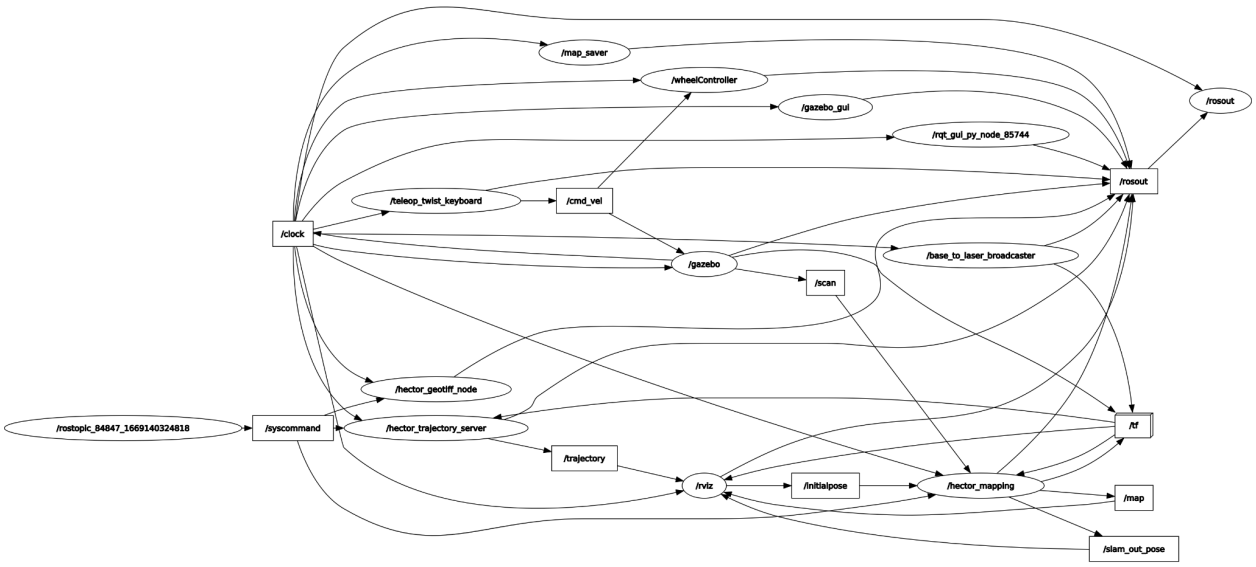



**Figure 4:** Node Network of Hector SLAM Operating with Gazebo Simulation.

### 2.2.2 Introduction to Hector SLAM

The Hector Simultaneous Localization and Mapping (SLAM) package was installed in ROS, which mapped an environment as a grid map and improved the accuracy of robot navigation. Hector SLAM used LiDAR data to create an iterative scaled reference map of the environment that the robot was in. Hector SLAM created a transform between the LiDAR laser frame of reference (FOR) and the map FOR, to create a map that was free of noise from the motor movement. The map was fully generated by manually driving the rover across the maze to map every area. This map was saved locally to memory, to be used for future localization.

### 2.2.3 AMCL ROS Package

The adaptive Monte Carlo localization approach (AMCL) was chosen as the means by which to localize the robot based on purely LiDAR data. AMCL is a probabilistic localization system that uses a particle filter to track the pose of the robot relative to the Hector SLAM reference map. The dispersed particle cloud created by the AMCL algorithm contained all the possible locations of where the robot could be in the maze at that instance. As the robot would advance through the maze, this cloud would converge to a centralized location and orientation by matching the LiDAR data to the existing map. This output pose was a position coordinate array (in x, y, z, roll, yaw, pitch coordinates) of the most probable location of the robot in the maze. This AMCL pose was refreshed up to 10 times a second with a new pose, where the probability of the robot being in this specific pose was adjusted after each update. All the relevant

package parameters such as the number of particles, LiDAR height scan, etc. could be configured for this specific application.

This pose was then overlaid onto the reference map using RViz: a GUI tool that visualized the surroundings of the robot based on multiple data streams. The active localized map was displayed in RViz in real time based on the incoming LiDAR sensor data.

### 2.2.4 Sending Navigation Goals Node

By using the map reference frame, a location on the map could be chosen as the final destination objective for the robot. In milestone two, the first objective was reaching the loading zone. The first navigation goal was the entrance to the loading zone. After having arrived and retrieved the block, the navigation goal would then change to one of the four drop off locations in the maze, and also specify the final position and orientation of the robot. The four states of operation (navigating to loading zone, loading, navigating to drop off zone, and unloading) were represented by different color LEDs that were controlled by the Raspberry Pi GPIO pins.

### 2.2.5 Move_base ROS Package

The move_base ROS package handled the movement of the robot towards the current navigation goal. The move_base package created the live cost maps of the system, which assigned a cost to each obstacle and an inflated region around them. There are two types of cost maps: global and local. The global cost map was based on the Hector SLAM reference map, and assigned a cost of 1 to the walls (should be avoided entirely). Each wall also had an inflated radius, which represented an area that could be entered but would be preferably avoided, and had a cost between 0 and 1 (could be a viable route region if there are no alternatives). The inflated radius used in the software was 40mm. The local cost map was generated with a radius of 2ft around the robot based on the live LiDAR data of the robot, which helped the robot avoid nearby obstacles. For both types of cost maps, the relevant parameters were configured as YAML (Yet Another Markup Language) files; a data serialization language used to create configuration files for any programming language.

### 2.2.6 Trajectory Planner, a Move_base Sub-Package

Move_base contained a subpackage called trajectory_planner. The input to this subpackage was a navigation goal and the output was the path that the robot would follow to arrive at the destination. The output path had curvature to avoid obstacles in the most efficient and ‘inexpensive’ way. Trajectory_planner used information about the robot footprint, cost maps, and limitations of the robot accelerating and turning speed.

The robot was instructed on how to move on the path using the command velocity topics, which were 6-element arrays (with velocities in x, y, z, roll, yaw, pitch). The Trajectory_planner adjusted to the movement of the robot and updated the desired path and the respective command velocity at a rate of 10Hz. These command velocity messages were converted to PWM signals for each wheel using the same transformation matrix as in milestone one.

## 2.3 Block Delivery Strategy

The team was able to complete milestone three, however used a different procedure than what is described in Section 2.3.2 due to time constraints and coding complexity. Section 2.3.2 describes the original plan for the programming structure, whereas the adapted plan inspired from the team's original idea can be found in Section 3.0.

### 2.3.1 Design of Block Retrieval Mechanism

The final block retrieval mechanism consisted of  4 components: the ramp, two arms, and the sweeper. A TOF sensor was used to assist in block detection, and a servo motor was used as the sweeper actuation method. The ramp was mounted on the first layer of the robot using two 3D printed brackets. The ramp contained a slot where the TOF sensor was mounted within a cutout in the ramp such that the block sliding up the ramp could not hit the sensor. The two arms held the servo and sweeper, and were mounted to the second layer underneath the motor controllers. The servo connected to a long stand-off, which slid through the sweeper and into the arm on the opposite side.

The sweeper was epoxied to the stand-off, and the stand-off was attached securely to the servo using thread-locker. The sweeper design was curved to allow for a smooth scooping of the block. At the end of the sweeper, beauty blender sponges were secured to the sweeper, and served as flexible grippers to ensure that the mechanism didn't jam and that the block didn't catch on its ascent up the ramp. The ramp was at a 30˚ angle, which was chosen after prototyping mechanisms at various angles and selecting the most effective angle for both block retrieval and expulsion. Figure 5 shows the final design for the robot with the labeled block retrieval mechanism components.

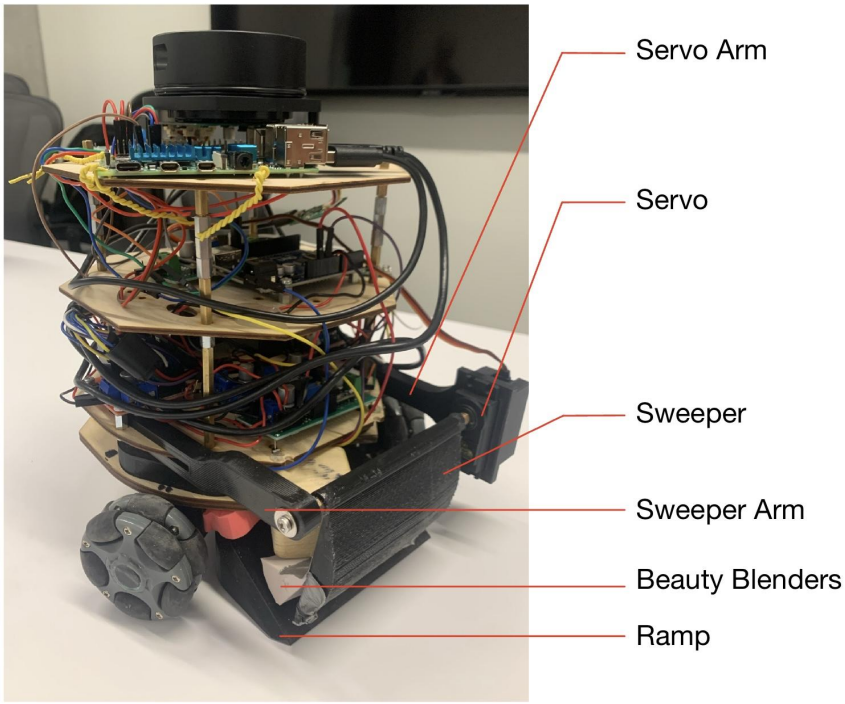



**Figure 5:** Final Robot Design.

### 2.3.2 Programming Block Pickup and Dropoff

The Arduino was connected over an Inter-Integrated Circuit (I2C) communication protocol to talk to the TOF sensor. I2C allows multiple peripheral circuits to communicate with one or more microcontrollers. There were no filters applied to the incoming TOF data, as it was already in digital form. Although it would have been ideal to output a filtered moving average of the data, this proved unfeasible due to the memory limitations of the Arduino UNO. The data from the TOF sensor was in millimeters, and it was published to a topic on the ROS network over ROS serial through the Pi. Once the robot achieved the loading zone navigation goal, the robot mode switched and was disconnected from navigation mode to stop auto-driving. The robot entered scan mode, which involved rotating and scanning the zone until a difference between the LiDAR front angle and TOF sensor was detected. This discrepancy in distance ensured that a block, and not a wall, had been detected by the robot. Once the block had been detected, the robot drove forward slowly until the TOF sensor detected that the block was around 40mm away. The servo controlling the sweeper would activate to bring in the block. The servo during this time was subscribing to a topic that was sending the PWM state the servo should operate at. After retrieving the block, a new navigation goal was set at the unloading zone. The robot followed the set path to one of the four potential unloading zones based on a parameter defined prior to the trial. When the robot arrived at the zone, the trajectory_planner outputted a

message that the robot had arrived at the destination. The servo would be actuated once again, an LED would turn on, and the block would fall out, completing the final milestone.

## 2.4 Integration Strategy

In order to integrate obstacle avoidance, localization, block localization and retrieval, the hardware and software was iterated upon for each milestone. The plan was to have the following functions for each milestone:

Milestone 1: Reading LiDAR data and obstacle avoidance.
Milestone 2: Localization and obstacle avoidance.
Milestone 3: Localization, obstacle avoidance, and object pick up/drop off.

Since the team did not complete milestone 2, the main jump from was from milestone 1 to 3 and included the addition of the block retrieval mechanism as well as the code for the block search/retrieval. In order to control the hardware of the block retrieval mechanism (servo and TOF sensor) a second Arduino was added to the robot. The addition of a second Arduino and the components that came with it had been planned for since the beginning of the project. Space was allocated on the third level of the robot for the second Arduino along with the power supply needed for the servo, and 5V power for the Arduino was ready and waiting for its arrival after the initial buildout of hardware in milestone one, facilitating a smooth integration process of new functionality.

To ensure that the different functions worked harmoniously, the code for milestone 3 was divided into four modes so the robot functioned as a “state machine”:

Mode 1: AMCL localization and navigation to loading zone.
Mode 2: Block searching and pickup.
Mode 3: Navigation to drop off zone.
Mode 4: Block drop off at drop off zone.

All of these modes were set off by different “if” statements in the code and each mode was coded as a separate function so that it was easier to debug. What enabled all of these separate functions to integrate together was the ROS structure; all the nodes were constantly listening and publishing to topics that other nodes could listen to, so as soon as the if condition was met, a function knew whether to start or stop.

When placed in the maze, the robot starts off in mode 1, it localizes and drives to the entrance of the loading zone based on the navigation goal. Once the robot reaches the entrance of the loading zone, it enters mode 2. In mode 2, the robot rotates in place to search for the block. While rotating, values from the TOF and LiDAR are collected to record when a block is detected. If no block is detected, the robot drives 4 inches into

the loading zone and rotates in place again to search for the block. Driving deeper into the loading zone ensures that the block is not obstructed from view during the first rotation. If the difference between the 0th degree of the LiDAR and TOF measurement is greater than the displacement between the sensors, the block is presumed to be straight ahead. In this state, the robot drives forwards towards the block and once the ramp makes contact with the block, the servo motor actuates to capture the block. Once the block is onboard, mode 3 is triggered and the robot drives to the drop off zone (relocalization was not necessary, since localization was continuously running in the background). When the robot reaches the drop off zone and the block is released, mode 4 is triggered and the delivery would be considered completed and successful.

# 3.0 Final Results

## 3.1 Obstacle Avoidance

In milestone one, the robot was able to avoid most obstacles present in the maze, only needing to be manually adjusted once when it got stuck on a corner. This incident occurred because the obstacle avoidance algorithm did not account for the robot's physical width when a narrow, but clear path appeared at a specific point in the map with a zig-zag wall pattern. The plan outlined in Section 2.1 proved to be very effective for this milestone. Overall, the robot excelled in obstacle avoidance.

## 3.2 Localization

The robot was unable to complete milestone 2, as the code was not ready in time for the trail. The team had issues getting Hector SLAM configured to the odometry frame of the system, as well as to the frame of the LiDAR coordinate system, and instead decided to focus their efforts on the third milestone. As a result, the team did not attend milestone two, and the final results are discussed in Section 3.3.

## 3.3 Pick-up and Delivery of the Load

Figure 6 shows the OnShape model of the block retrieval mechanism (note that the sweeper is version 3, which is not the same version used in the maze). In Figure 6, "1" represents the block, "2" represents the sweeper mechanism which was powered by a servo motor, and "3" represents the ramp where the block was stored.

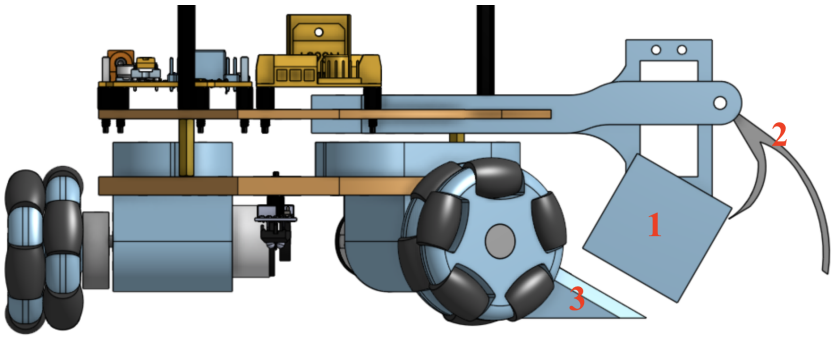



**Figure 6:** OnShape Model of Block Retrieval Mechanism.

The team was able to complete the third milestone using most of the procedure described in Section 2. However, a few alterations to the original plan were required, as the team could not get all the parts of the program to function fully for the milestone. In

order to complete the milestone, the robot was programmed to localize at the four-way intersection of the maze, where it could navigate to the final drop-off zone. This was achieved by collecting LiDAR data at the four-way intersection and creating an if statement such that if the outputs of the scan topic ever resembled that of the intersection, the robot would localize at the intersection. This was the only section of the maze where the robot could reliably localize. To begin the milestone, the robot entered a general navigation mode and constantly searched for the block by comparing the distance of the LiDAR sensor with the TOF sensor distance. If the TOF sensor distance was ever less than the LiDAR distance, this meant the block was detected. Once the block was detected, the robot would approach the block slowly and sweep it in once the TOF distance was less than 90mm. After obtaining the block, the robot navigated around the maze to find the four-way intersection. At this intersection, the robot drove to the final drop-off zone and deposited the block.

There were a few instances where the TOF sensor detected a block when in reality, it was a corner of the maze; this issue is further discussed in Section 4. Otherwise, in the first attempt, the robot was able to find, pick-up, localize, and deliver the block. In the few minutes of the attempt, the robot entered block detection mode (mode 2) prematurely due to the TOF confusing walls with the block, so the robot had to be pushed into the loading zone since this mode of operation does not navigate autonomously around the maze. In the second attempt, the robot was able to make it to the loading zone, obtain the block, and properly identify and stop at the dropoff point, but was unable to unload the block after localizing. When combining the two attempts, the robot was able to demonstrate that it could localize, pick-up, and deliver the block to a drop-off location, and thus the team successfully completed the final milestone objectives.

## 3.4 Integration

The team was able to effectively integrate the components from each milestone and add new features through extensive planning during the design proposal stage. Between milestone one and three, the block retrieval mechanism, new hardware, and new driving modes were added.

To add the block retrieval mechanism, the spacing of the components was organized during the initial OnShape modeling of the design. As much space as possible was left at the front of the robot to allow for flexibility when designing the mounting for the ramp. Mounting the sweeper was also successful, as there was space left underneath the motor controllers on the second layer. It was difficult however, to disassemble the sweeper arms from the robot, as the motor controllers would have to be removed first.

Adding additional hardware such as the sweeper servo, second Arduino, and TOF sensor was also planned for in both the design layout and when creating a wiring diagram. Wire routing cut-outs were used to route the new components neatly into the pre-existing design. The wiring diagram proved to be very helpful, as the team knew the number of required pins for each component, could plan ahead when sizing the DC-DC converters, and used the diagram as a reference when adding to the electrical system between milestones.

The software side of the robot was also continuously reiterated and repurposed, allowing for integration between milestones. The original script for converting velocities to PWM signals and the first Arduino code to control the motor PWMs were both reused for the second and third milestones. The main change for the third milestone was how the command velocities were generated in coordination with the LiDAR sensor. In addition, to conceptually plan out the programming, a ROS nodes block diagram was created in the initial design proposal to understand the tasks and progress the team needed to complete between each milestone.

Overall, the team achieved effective integration of the programming and design components between the three milestones through extensive planning and thoughtful methods of organization.

# 4.0 Discussion

## 4.1 Obstacle Avoidance and Localization

The overall obstacle avoidance strategy proved to be successful, with the exception of 1 specific edge case that required additional assistance to turn the corner. The high rate of velocity commands ensured that the robot was driving smoothly by making minor changes to its direction when driving straight. A future consideration would be to increase the angle range of the ‘forward direction’ that needs to be clear of obstacles in order to move forward. This would ensure that a narrow, but clear path would not be considered viable.

Although the overarching localization and navigation pipeline was completed in time, the actual performance of the localization proved to be unreliable. The AMCL package required an odometry frame as an input so that it could track the robot’s movement and adjust the particle cloud accordingly. Because the encoder data was not linked to the ROS network, a laser-based odometry package named laser_scan_matcher was used. Although there should not be a drift in the laser based odometry frame, the resulting localization output of AMCL jumped across the map as the robot moved, causing the cloud to require an extensive amount of data before localizing. The source of this jump was not identified. A further investigation into the accuracy of the odometry to map transform would be required to address this issue. Alternatively, this issue may have been addressed by using an encoder based odometry solution. The motivation against utilizing encoder data for odometry initially was due to the potential for drift to accumulate throughout the course.

Had the robot been able to localize in a more efficient manner, the pipeline was ready to navigate autonomously to any chosen point on the map.

## 4.2 Robot Design and Assembly

The final mechanical design for the robot proved to be very sturdy, highly effective, and well organized. There were no iterations required for the main structure, wheel mounts, battery mount, or ultrasound brackets. The robot was sturdy due to the lock nuts used to fasten all the brackets, which ensured that connections would not loosen over time. All the electrical components were mounted on standoffs to ensure proper air flow and prevent overheating, which was one of the design strengths considering all components had proper ventilation. The main structure of the robot was made of laser-cut plywood which had various cutouts that could be used to route wiring for organization. Any holes

for mounting brackets were also already added in the OnShape file so that drilling would not be required, making assembly easy and efficient.

The wheel mounts ensured that the omni-wheels were each at a 120˚ angle from each other. These wheel mounts were secured to the base layer with screws and locknuts, and the motors connected to the wheels using a connector on the shaft. For the wheel shaft mounting system, the team would recommend using either metal or wood for the shaft mounts, as the fastening strategy used to secure the motor shaft to the wheel had a nut eating away at the edges of the 3D printed material. To resolve this temporarily, the team placed small slices of sheet metal on either side of the nut so that it would not damage the mount further.

The sweeper mechanism required a few iterations for the flap; however, in the end the team used the first iteration and taped beauty blenders to the tip. The beauty blenders added some grip to the sweeping mechanism so it would effectively grip/tip-over the block when the flap came into contact with it. The flexibility of the blenders ensured that the block could be swept in while being oriented at various angles.

The sweeper design had some flaws including occasionally unscrewing from the servo when hitting the block during intake, TOF sensor placement issues, and requiring that the standoff be epoxied to the 3D printed sweeper. To resolve the unscrewing issue, the servo should be mounted on the opposite side of the robot so that block intake can only tighten the screw connection. The TOF sensor should also be mounted so that it is angled correctly to ensure that the block is detected at further distances, which can be done by angling the screw holes or ensuring the ramp is level. The standoff should be either press fit into the sweeper or a heat gun could have been used to have the 3D printed material flow around the standoff to secure it.

One major feature that could be redesigned is the footprint of the robot, as it was larger than expected which made it harder for the robot to turn. The omni wheels and sweeper mechanism were the main causes for the larger footprint. An improvement would be to integrate the sweeping mechanism into the body of the chassis so that it doesn’t protrude as much from the structure. Space could be made by mounting all the motor driver electronics on vertical walls instead of mounting them horizontally on each layer.

## 4.3 Electrical System Discussion

The final electrical design of the rover was effective and reliable, but could have used some improvement in user friendliness/ease of use. Throughout the project, the team experienced no electrical incidents, and no components were damaged as a result of improper wiring, incorrectly spec’d parts, etc.. All components received adequate

current and voltage supply, and pin allocation was kept consistent throughout the project to avoid confusion. More robust electrical connectors and/or harnesses would have made the design less precarious, however, the team made the most of the materials and budget which were available.

Part of what made the electrical system so robust was the care that was taken in ensuring that all components on the robot were being powered by a supply with adequate voltage and current capacity. For example the Raspberry Pi needed a dedicated 5.1V, 3A buck converter. Had this not been accounted for, the Pi would not have been able to operate properly and the entire robot would cease operation. Additionally, the servo motor used in the block retrieval mechanism had a stall current of 2.5A, so it too needed a dedicated power supply to prevent drawing excess current from, and frying the Arduino which controlled it. The electrical requirements for this project were not overly complex, but since the design was constantly in flux (frequently being assembled/disassembled) the team was reminded of the value of a consistent electrical layout. Getting electrical sorted out early allowed for more resources to be allocated to the more complex part of this project for the team - the software.

An improvement which could have been made would be adding standardized quick disconnects (Molex connectors or something similar) for the wires traveling between layers of the robot structure. This would have allowed for the layers of the robot to be more modular, and eliminated any worry that wires were being reconnected erroneously between the many disassembly/assembly cycles as the robot was iterated on. All things considered though, the electrical system was thoughtfully planned early on, executed well, and met all necessary requirements throughout the entire design process.

## 4.4 Gazebo Simulation

The Gazebo simulation was functional in providing a world environment to test the inputs and outputs of each subsystem rapidly without needing to boot the robot and launch the files. The simulation rover was able to respond to command velocity commands and output live LiDAR data about the maze. The goal of the simulation was to create a test setup where the various parameters of the localization and navigation packages could be rapidly configured and tuned to this specific application. Because the robot was not able to localize using the AMCL package, the full capabilities and benefits were not realized.

Had the original localization plan worked, the simulation would have identified any potential logical issues in the navigation strategy. Additionally, a TOF sensor would have

been integrated into the simulation to test the identification and retrieval of the block in the loading zone.

# 5.0 References

[1] Dattalo, A. (2018, October 8). *ROS/Introduction*. ros.org. Retrieved December 11, 2022, from https://wiki.ros.org/ROS/Introduction

[2] *ROS - Robot Operating System*. ROS. (n.d.). Retrieved December 11, 2022, from https://ros.org/