WORCESTER POLYTECHNIC INSTITUTE
ROBOTICS ENGINEERING PROGRAM

Lab 4: Simultaneous Localization and Mapping (SLAM)

Submitted By:
April Bollinger
Pranav Jain
Michael Primavera

Date Completed: December 11, 2024
Course Instructor: Professor Aloi
Lab Section: RBE 3002-BX01

Introduction

In this lab, ROS gmapping was implemented to create a map of a maze using a *turtlebot* with a LIDAR sensor attached to it. When there were no undiscovered areas left, the robot returned to its starting position. A person then moved the robot to a random place on the field, the robot was given a target point for the to go to using RViz, and the robot localized itself using ROS's AMCL package. When the robot was localized, it generated a path to the point it was given and drove to that point.

Methodology

There were two launch files: one that ran phases 1 and 2, and a second launch file for phase 3. The first launch file started gmapping (phase 1/2). The launch files would begin the code and establish all the nodes. This process was initiated with a few simple lines of code in the terminal that would begin *roscore*, establish an SSH connection with the robot, and start the initialization and communication of the nodes by running the launch file.

Once the launch file had been run, the LIDAR on the robot would acquire a partial map of the field; centroids were determined by finding all frontier cells in the map. Frontier cells were found by looking at the four neighboring cells and checking to see if they were part of the unknown space (-1 in the map OccupancyGrid). If a cell had a neighbor in the unknown space, it was added to the list of frontier cells. A dictionary was then used to group the frontier cells by calculating the Euclidean distance between all the points. The centroid for each group was calculated by finding the mean of the position of each frontier cell and the actual centroid was set to the frontier cell closest to the calculated centroid. Once all centroids had been calculated, they were sorted by cost–dependent on distance from the robot–and the lowest-cost centroid was sent to the path planner as the first goal. The other centroids were sent to path planner as a Path message to be used for error checking in A*. When a path couldn't be made to one of the centroids, it started going through the list and trying to find a path to each sequential centroid. When A* ran out of centroids to look at, the maze had been mapped and the code then planned a path to the robot's initial (home) position.

Path_planner used A* to calculate the path from the robot's current position to the goal it had received. Our implementation of A* used 50% Dijkstra and 50% Greedy Best First Search. It also used a heuristic that checks the eight neighbors of each cell in a path and determined if they were part of the c-space. For every neighbor that was part of the c-space, the cost of that cell increased by 12. This keeps the generated path closer to the middle of the maze.

When a path had been generated, the pure_pursuit was called. The pure pursuit function was used in combination with wall avoidance to navigate the maze. Pure pursuit used a radius (look-ahead distance) and a kp value to determine how it moved. To make sure that the robot followed the path more effectively, it was limited to turning a maximum of roughly 33° away from the path; an angle greater than 33° caused the robot to spin in place. The wall avoidance used LIDAR scans on the front left, front right, and front middle to determine how the robot should

move to avoid hitting a wall. If it sensed a wall on the front right or left, it backed up and turned away from the wall. If it saw a wall directly in front of it, it was programmed to back up for 1 second. After it backed away, it calculated a new path to the next centroid or recalculated the path home.

The second launch file started AMCL (phase 3). The 2D Nav goal in Rviz was used to give the robot a goal to drive to after localization and start the localization process. A service proxy was used to call the AMCL global localization service which spread the particles evenly across the maze. This increased the speed of localization and prevented the robot from remembering a previously-localized position. After the particles had been spread across the field, the robot started localizing using the map. While localizing, the robot spins in place to make localizing easier. The robot attempts to localize until the determinant of the covariance is less than 0.0004, which we found to be a good value through testing.The matrix determinant was recreated from the 6x6 matrix given by the AMCL pose. When the robot was sure of where it was (determinant < 0.0004), it published the AMCL pose and the goal that was received from the 2D nav goal in RViz. Both the goal and the AMCL pose were received by path planner, which generated the path and called pure pursuit to handle navigation.

Results

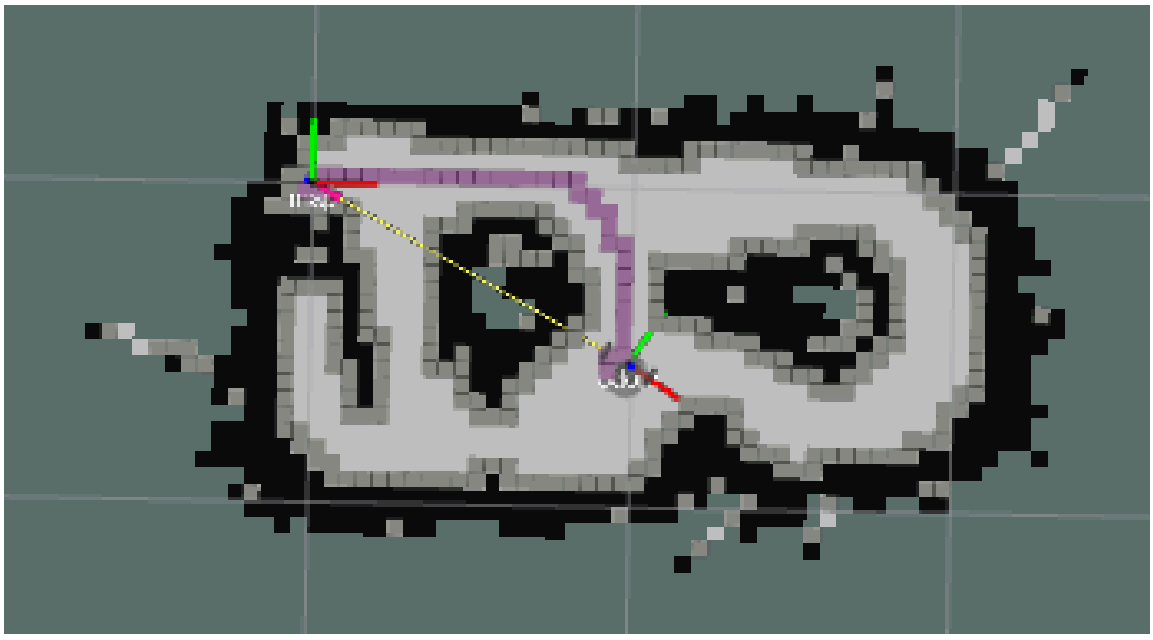

Figure 1: The path generated by A*, shown on the field mapped using the LIDAR radar.

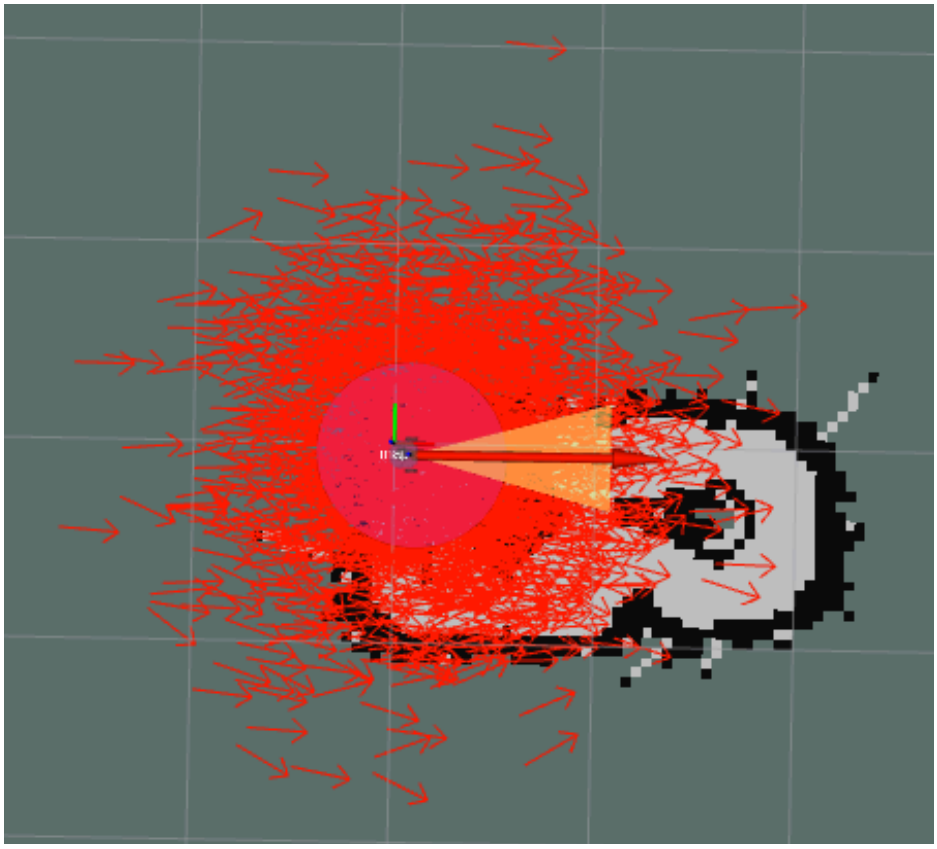

Figure 2: Initial view of AMCL particles before ROS global localization of particles and localization starts.

Our implementation of pure pursuit took a considerable amount of time to map the field. The time to map the field and go home was 463 seconds. The time to localize and go home was 137 seconds.

Table 1: The outcomes of each attempt of the demo

Attempt	Gmapping	AMCL
1	Got stuck due to wall avoidance 3 wall collisions	Process terminated prior to mapping the entire field. No AMCL attempt.
2	0 wall collisions Failed map save	Correctly localized and navigated back with 0 wall collisions. Used map from tele-op run.
3	Tele-operated to get map 0 wall collisions	

Discussion

The goals of the lab were to map the field using a *turtlebot* robot with a LIDAR radar. The robot was moved to a random position on the map where it localized itself and drove back to its home position autonomously.

A key component to this lab was creating an algorithm that would follow the path smoothly. The choice was made to create a simple algorithm that calculated the distance from the robot to the points on the path, and if the distance was within a set range, that index would be the next goal, or "look ahead" point, for the robot to drive towards. The result was a smoothed path that the robot could follow in a streamlined process, eliminating jagged turns (Figure 3).

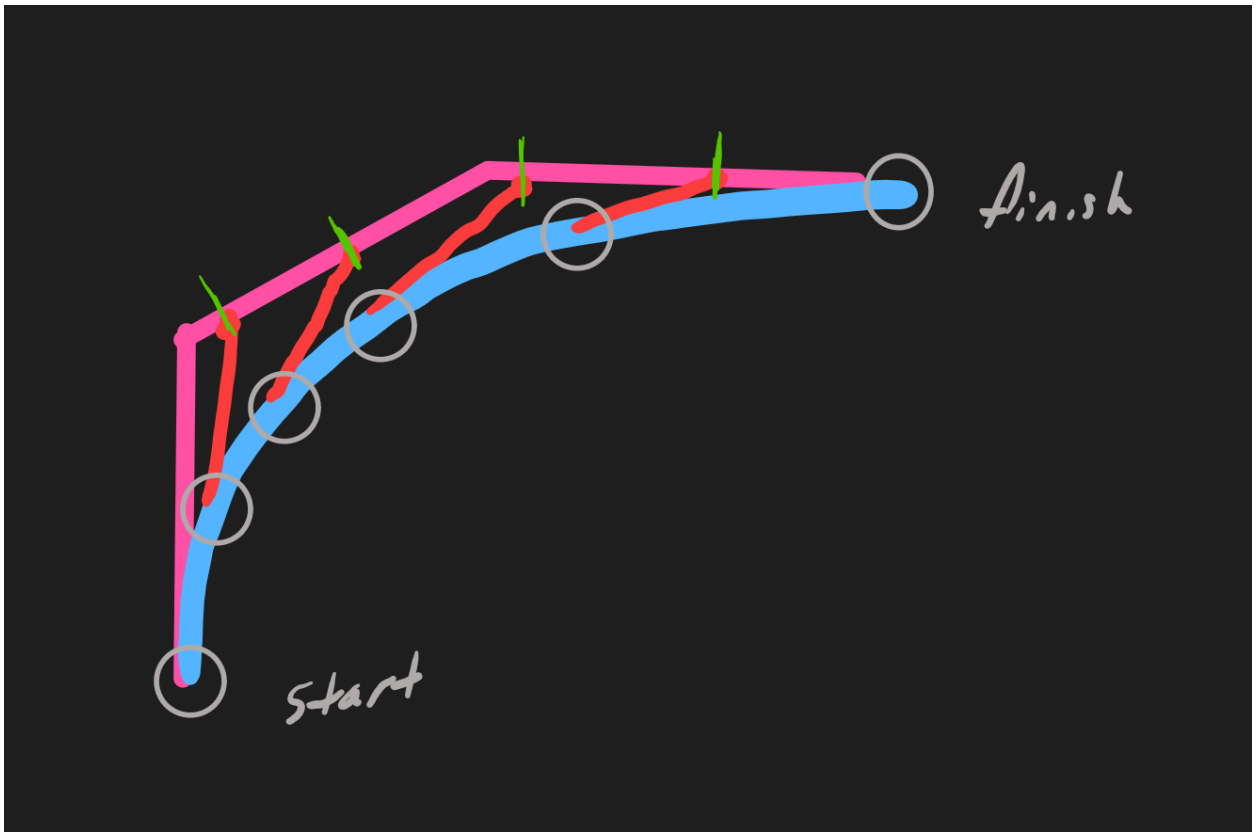

Figure 3: "Follow the Carrot" logic used to develop the pure pursuit code. The pink line represents the path generated using A*; the red lines represent the constant length that was compared to the Euclidean distance to each cell on the path, the green dashes represent the updated goal points that the robot (grey circles) will drive to, and the blue path is the path the robot follows.

A problem encountered using this algorithm, however, was that the robot would occasionally hit its tire against the poles on the field. This was due to the robot cutting the

corner, an inherent feature of pure pursuit, causing a new issue after eliminating the jagged-path problem. Our team chose to implement a wall sensing algorithm that would detect objects within a specified range of the robot, roughly within the front ⅓ of the robot's LIDAR range (Figure 4). The result was that the robot would stop abruptly before hitting a wall.

Solving the problem of hitting the wall then introduced the problem of the robot needing to be slowed down to prevent the crash, and there was also the problem of what exactly to have the robot do to not have a path that would direct it into a wall. The decision was made to have the robot recalculate a path to the current centroid after the robot stopped from sensing a wall: since the new path would be out of the c-space, the robot would turn away from the wall and be directed towards the path that was centered between walls.

At this point, the robot wouldn't hit walls but was slow to navigate the field. A possible solution to this problem was considered, but time didn't allow for further refining of the algorithm. There are numerous ways one could solve the problem, such as dynamic slowing when a wall is sensed, and preventing the robot from actually stopping until an object was detected at a threshold closer to the robot. This would enable the robot to begin following the path at a slower speed when within tight quarters, and drive faster when not sensing walls in its general vicinity.

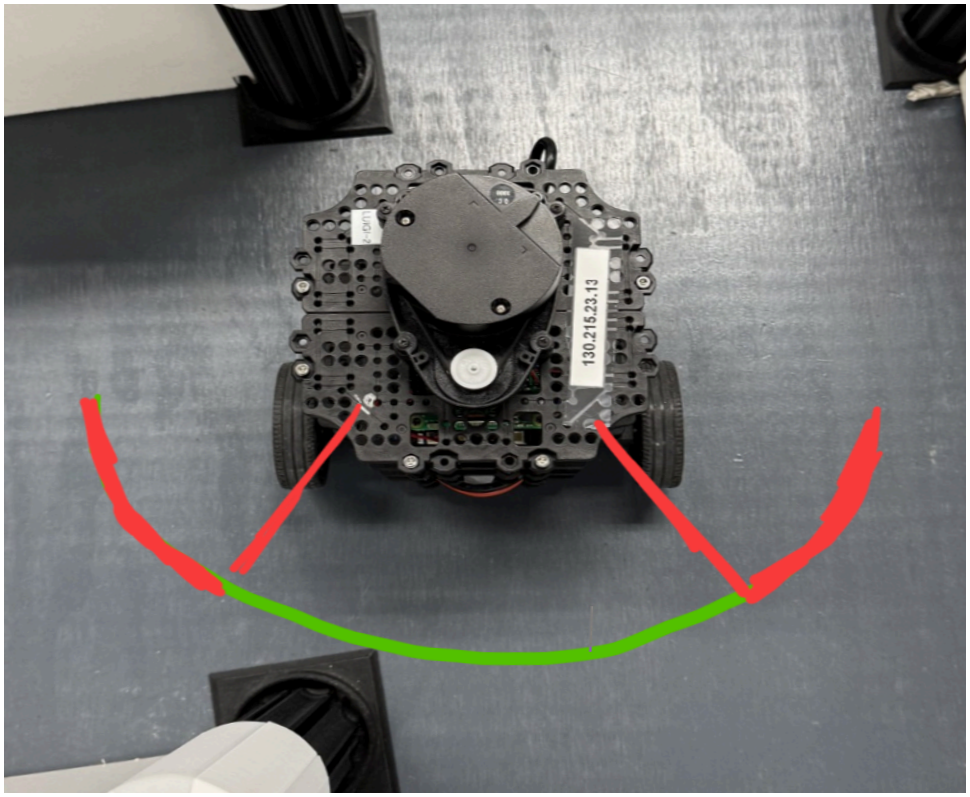

Figure 4: Wall avoidance visualization. Red indicates the front left and front right angles, and green indicates the front middle angle.

Another topic worth discussing was the process of debugging. There were many bugs in our code in the beginning phases of the lab, and these were sometimes overlooked–the code would run, but not optimally. Since the code would run, tracking down the bug, and often even being aware of its existence in the first place, was often a difficult task. The primary method used to debug in this lab was to use print statements.

One recurring bug encountered was that the robot would often stop mid-run with no indication as to why. To track down where in the code the program wasn't working properly, we would print "Here" to the screen. Since we knew where we put the print statement, if the print was never executed, we would know that part of the code was never reached. We could continue moving the print statement upstream until it was displayed, and that would often lead us to discovering where the bug was.

Another method of debugging was to simply print variables out and see if they were initiated to what we intended them to be. For example, one bug we had was that the robot would go to the first centroid, then stop without going to the remaining ones. We went through the code by printing "Here", and the logic was flowing the way intended. We then would print each of the variables out that were arguments to the functions being used, and that led us to the realization that we were sending *world* coordinates to A* rather than grid coordinates for the remaining centroids in the list we had created.

Conclusion

The primary learning takeaway from this lab were the skills gained using ROS and Python. Utilizing these packages, engineers can simulate and test the robotic systems they design. The primary benefit is that the engineer can focus their efforts on designing the system rather than developing ways to simulate and test the system. As a student, acquiring hands-on experience using the software that'll soon be used when graduating as an engineer will be an excellent preparation for entering the workforce.

Final code release

Section	**Contributors**
Group Code	Pranav, April, Michael
Report	Michael, April, Pranav