RBE 3002: Final Lab Report

Submitted To:

Prof. Michalson

Unified Robotics IV - Navigation

Report Prepared By:

Brian Boxell

Omri Green

Owen Pfannenstiehl

Special Thanks:

Cooper, Eli, Yoni, and Annie

May 2, 2022

INTRODUCTION

In this lab, we had several goals. Our overarching objective was to master robotic navigation in a practical setting by programming a turtlebot to map out an unknown arena using a custom A* algorithm for path planning. This was further broken down into three goals: (1) establishing effective communication between the robot and our python scripts using ROS, (2) mapping the arena with SLAM through GMapping and localization with AMCL, and (3) identifying and selecting frontiers and using A* to plan paths to them. We built off the code in our previous labs that contained functions for movement, although we did optimize over the course of this lab. We also took advantage of pre-existing nodes for GMapping and AMCL in our implementation of SLAM and localization, respectively.

We tested the execution of our goals by running the turtlebot through a 3-phase gauntlet. In the first phase, our robot had to navigate an unknown space until all reachable areas in the arena had been detected and saved to map files. The second phase required the robot to navigate back to its starting position. In the third phase, the robot needed to navigate to a 2D nav goal passed to it through RViz, which we used to visualize the robot's path planning and run simulations.

METHODOLOGY

Implementing A*

A key component of this project was the ability to pathfind between two points on the map and avoid obstacles. There were several algorithms that we could have used to find a path between two points, but A* was best suited for our application. A* is a variation on Dijkstra's algorithm that combines the distance to adjacent nodes with a user-defined heuristic to create a cost associated with every node and then chooses which nodes to explore first based on those costs. A* was particularly suited for our use case because it allowed us to define a custom heuristic. For example, we could have in theory defined a heuristic based on the euclidean distance to the target, or prioritize a path that minimizes the number of turns. Playing with this heuristic allowed us to change the geometry of the path that the robot followed to make our path as optimized as possible while avoiding obstacles.

Extracting Waypoints

After the A* algorithm generates a path to a destination, our code runs that path through an optimization algorithm that removes intermediate waypoints. This algorithm traverses through the path and determines if each node lies on a straight line between the previous and following nodes. If that node is on a straight line, it gets removed from the path. The result of this algorithm is a smooth path that does not include intermediate stoppages. The algorithm returns the new, optimized version of the original path that is ready to be followed by the robot.

Launch Files

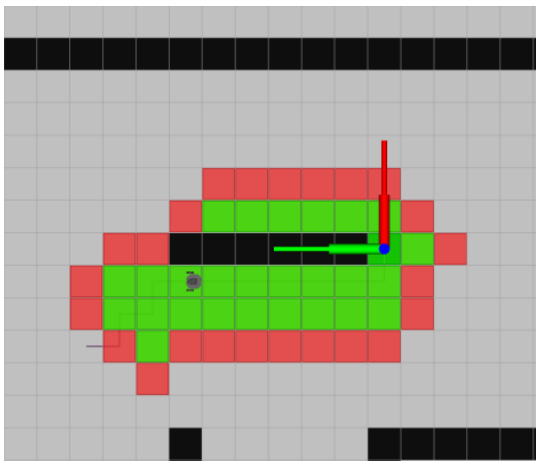
Unlike previous stages of the lab, the map is provided by the GMapping node instead of a MapService. In real life, GMapping creates a map based on the LIDAR data it gathers from the robot projecting the robot onto the physical walls. In simulation, however, these physical walls that the robot collects LIDAR data off of are provided by the Gazebo simulation. Therefore, the simulation launch file must launch Gazebo and an appropriate map file for the robot to explore in addition to RViz whereas the real life launch file only launches RViz. Both launch files also launch all of our custom nodes. We also created variations on these launch files that only launch certain nodes and expect the user to manually launch others to be used for debugging.

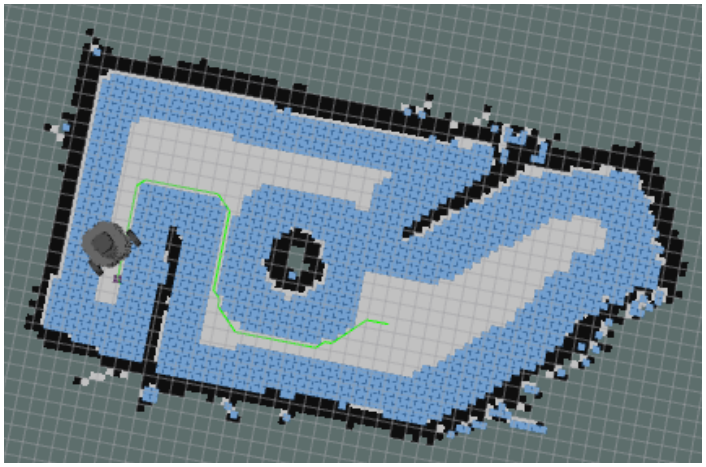
Service Calls

The node that contains the path planning (A*) algorithm is called Path_Planner.py. This node contains a variety of services that can be called by external nodes, specifically our state machine node, Pathfind.py. These services include path planning between points and frontier selection. This enables the Pathfind node to be able to calculate the path between points and manage the frontier exploration stage. Because the Pathfind node is reliant on the services in Path_Planner, it will wait for the Path_Planner node to exist before moving along in code.

RViz

In this lab, we wrote a lot of functions that dealt with calculations in physical space, such as path planning or obstacle expansion or frontier exploration, or moving our robot within a map. We used RViz to visualize all of our data in real-time by simulating the robot, plotting path plans, and displaying other forms of data. For example, we managed to use RViz to help visualize the early versions of our A* algorithm by watching it expand outward with every loop of the algorithm (Figure 1). In the later versions of the code, we use multiple GridCells and Path messages to help display C-space with blue cells and our robot's path plan with a green line leading to the desired end position (marked with a purple cell) in real time (Figure 2).



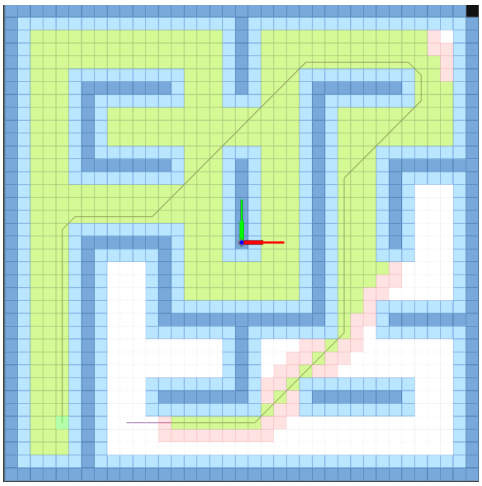


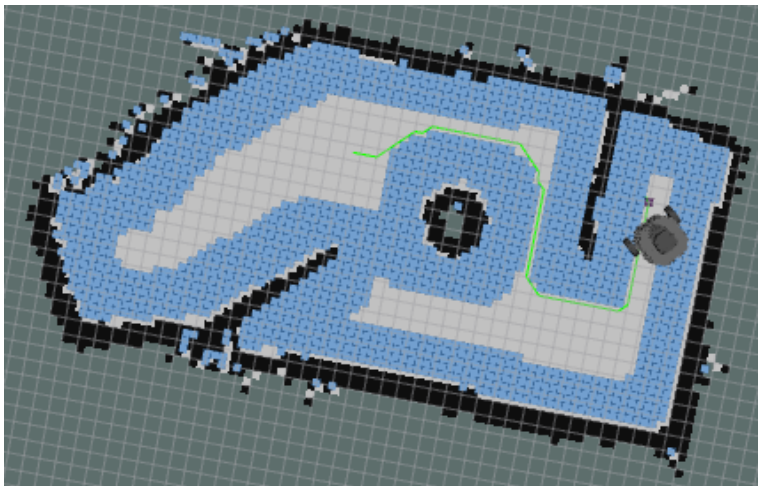
(Figures 1, 2)

C-Space Calculation

We calculate the C-space by first grabbing the map information from the map topic. We then traverse this map and look for every non-obstacle cell (a cell with an obstacle probability value between 0 and 4 on a scale of 0-100). Every time we find a non-obstacle cell, we look at neighboring cells within a range determined by the scale of our obstacle expansion and our cell size and mark them as C-space cells. C-space cells have much higher costs in our path planning algorithm than free space cells; additionally, C-space cells closer to obstacles have even higher costs than C-space cells that are further away from any obstacles. This means that C-space cells are still walkable so the robot won't become stuck, yet traveling through C-space will still be a last resort. Additionally, if the robot has to path through C-space, it will choose the path through the C-space that is furthest from all obstacles thanks to the prioritization of C-space cells with lower costs over C-space cells that are more dangerous (closer to obstacles).

While a single-cell expansion may be the simplest (Figure 3), when we reduce our cell size to improve the accuracy of our map we need to increase the C-space expansion scale to multiple layers of C-space cells (Figure 4). When our C-space map is complete, we are free to run A* without worry of colliding with map geometry.





(Figures 3, 4)

Frontier Identification and Selection

Our frontier exploration has two sides: frontier identification and frontier selection. Frontier identification begins by pulling the map from the map topic. From there, it creates an empty copy of the original map that's all zeros. The values in this new 'frontier map' represent if a cell is a frontier cell (1) or not (0). We then traverse both the original map and the frontier map simultaneously. Whenever we reach a known free space cell in the original map, we check its neighbors in a neighbors-of-8 pattern. If its neighbors contain both a known free space cell and an unknown cell, it is considered to be a frontier cell and the equivalent cell in the frontier map is marked as a frontier cell. Once we've fully traversed the maps, we then traverse through the frontier map again so we can group the frontier cells into bins by creating a list of frontier cell coordinates, beginning with the first frontier cell encountered. We then remove that cell

from the frontier map and add every neighboring frontier cell to the list while removing them from the map. We then do that to each of the cells in the list until no more neighbors are found by calling the function recursively. This adds all frontier cells in a frontier to a bin, which we then save. We then continue to traverse the map until we find a new frontier cell and repeat the process until the map is fully traversed.

Once we have identified our list of frontiers, we move on to frontier selection. First, we filter out all lists of frontier cells that don't contain a minimum number of frontier cells (the number depending on the size of our cells) in order to filter out frontiers that are just noise. From there we determine which frontier to explore by assigning each frontier a priority equal to the number of cells in the frontier squared minus the distance. We then choose the frontier we want to go to and calculate its centroid by averaging the x and y coordinates of every cell to get the x and y coordinates of the centroid of the frontier. We then plan a path through the C-space map using A* and the robot starts moving. The robot then begins the entire frontier identification and selection process again immediately and continuously (not upon reaching the centroid). This allows us to continuously approach frontiers with little risk of a collision.

Localization with AMCL

AMCL is a probabilistic localization library that enables the robot to recognize its position on a map based on a comparison between its surroundings and a known map. AMCL continuously spreads a bunch of dots across the map and compares the surroundings at each dot to what the surroundings of the robot looks like. It then calculates the error between those two and uses that error to calculate a weighting for where the algorithm should generate the next set of points. Over time, the dots will localize around where the robot is and it will continuously get a more and more accurate idea of the robot's true pose.

RESULTS

Our first final demo had mixed results (with our second pending after this report). Our robot was capable of exploring the entire map quickly and effectively, and as accurately as GMapping would allow with our filters in place. It was also able to save and export the map correctly. However, it struggled over time during Phase 2 due to sensor drift and the fact that the map was so enclosed that the robot was often forced to pass by obstacles that were inside the robot's LIDAR's minimum distance, producing erratic and erroneous values which conflicted with our localization routines. This caused the robot to collide with an obstacle and become stuck, meaning that although Phase 2 began perfectly, it was incomplete.

Despite going above and beyond with our absolute best efforts, we did not have the time and resources to complete the GMapping-less AMCL localization required for Phase 3, and thus did not get to experiment with it.

Phase 1

The goal of Phase 1 was to autonomously explore the space that the robot is in and create a complete map of everything the robot observes. This task can be broken down into a few basic parts: frontier selection and pathfinding, path following, and map updating. A simple approach to this task would be to first select the frontier the robot wants to explore, path-find to it, drive along that path while updating the map along the way, and then repeat until the map is fully explored. However, this method has a fundamental flaw in that it has to predetermine the route for the robot based on the map data available at the time. In the scenario where the robot detects an obstacle or an undesirable cluster of C-space cells in its route while halfway through executing the path following, the robot would need to be able to recognize that and recalculate accordingly. To prevent these issues, the basic steps need to run simultaneously so that the robot is continuously updating its path while it is driving. This also solves problems where LIDAR readings on areas more than ~30cm away are unreliable, as the robot recalculates long before entering these uncertain spaces.

Having identified continuous path planning to be a requirement, another issue was identified where the robot was often forced to travel closer to obstacles than the LIDAR's minimum detection distance allowed. This caused the SLAM algorithm to detect open space in the middle of a wall, updating the path to explore the perceived 'hole' in the wall. A simple solution would be to increase the c-space; however, given the size of the maze the robot was exploring, there was no way the robot could maintain this minimum distance from both walls and still explore all areas of the map. An interesting quirk of this issue we observed was that the longer the robot spent looking at a wall closer than the LIDAR's minimum distance, the worse the hole in the wall got (as the more erroneous values the GMapping received, the more convinced it was that they were right and previous values were wrong). Our solution was to make the robot drive as fast as possible within the limits of our navigation code so the SLAM algorithm was never given the opportunity to propagate the perceived holes in walls. This allowed us to complete Phase 1 without error. This approach also had the positive side effect of completing the maze navigation really quickly.

Phase 2

Phase 2 began as soon as Phase 1 ended, running continuously under the same launch file. Once every reachable valid frontier had been explored in the map, the robot exported the map to two map files in the maps folder called MyMap.yaml and MyMap.pgm. At that point, it was (correctly) assumed that the robot had accurately mapped every obstacle in the maze and it was therefore acceptable to follow a predetermined path all the way back to the home position. This was advantageous because that predetermined path was significantly more efficient than continuously updating A* on the way home. The robot utilizes all the same nodes as in Phase 1. It uses the AMCL built into the GMapping node for localization and it uses our lab_3 node for driving.

Phase 3

Since we did not manage to accomplish Phase 3 during our original demo run, we will be writing most of this section in theory.

Phase 3 begins with the robot placed in a random location on the map and, given the map that was generated in Phase 1, determining where the robot is. Once the robot has localized itself, it waits for a 2DNavGoal message from RViz and attempts to pathfind to that location.

The localization part of this would have been accomplished using AMCL, which is a library for probabilistic localization that takes in what the robot sees around it and compares it to a predetermined map to figure out what the most likely pose of the robot relative to the map is. Given that this is a probabilistic algorithm, there are several edge cases that could have caused incorrect results. For example, if the map that the robot is in is symmetric in any way, there is no good way for this algorithm to distinguish one position from the corresponding symmetric position. However, the map that our robot explored was not symmetric, and therefore would not have experienced this problem.

Once the robot has localized, it should sit and wait to receive a message from the user in the form of a 2D nav goal through RViz representing where to drive to next. something we accomplished earlier during our testing of A*. This was easily implemented (albeit unused) by simply re-enabling our earlier testing code.

DISCUSSION

We had several goals in this lab that comprised a plan to allow our robot to navigate through an unknown space. We managed to accomplish our first goal of establishing effective communication between the robot and our python scripts using ROS topics as was evidenced by the working function of our robot throughout the demo. We achieved our second goal of mapping the arena and localizing in real time, which was proven by our robot's successful procedural navigation of the arena without collisions by using the GMapping and AMCL nodes provided to us. Finally, we achieved our third goal of identifying and selecting frontiers while simultaneously using A* for path planning, which was demonstrated by our robot's safe and effective navigation throughout the arena without stopping during Phase 1.

CONCLUSION

We learned a lot of new topics in this lab. Whereas in previous labs we learned about the elemental structure of ROS and how to make the robot move, in this lab we learned about topics that are fundamental to robot navigation. We learned how to implement an A* algorithm and what kind of greedy heuristic works best for our purposes and how to tune it. We even learned how to optimize a path for our Turtlebot. We learned how to implement frontier identification and selection, and we learned what kind of selection heuristics were most effective at exploring an unknown field quickly. We learned how to expand obstacles using C-space and how to efficiently manipulate the costs of cells in a field in order to balance a robot's speed and accuracy in navigation. We even learned how to display all these processes through RViz. Finally, we learned how to make all these separate processes work in concert to achieve our goals.